TR-10 March 1981

The Portable Standard LISP Users Manual

by

The Utah Symbolic Computation Group

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

Version 3: 26 July 1982

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.

Copyright (c) 1982 M. L. Griss, B. Morrison, and B.Othmer

Work supported in part by the National Science Foundation under Grant No. MCS80-07034.

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, 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, Martin Griss, Bob Kessler, Steve Lowder, Chip Maguire, Beryl Morrison, Don Morrison, Bobbie Othmer, Bob Pendleton, and John Peterson.

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 and M. Griss; major sections have been contributed by E. Benson, W. Galway, and D. Morrison.

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, Case to make symbols more readable, and RLISP syntax for code examples. We would appreciate comments on these choices. Please feel free to comment on Chapter order also.

Report bugs, errors and mis-features by sending MAIL to PSL-BUGS@Utah-20; alternatively, send a message to Benson and Griss from within PSL by calling the Bug function, BUG(); in RLISP.

Permission is given to copy this manual for internal use with the PSL system.

TABLE OF CONTENTS

CHAPTER 1. INTRODUCTION

1.1. Opening Remarks 1.1
1.2. Scope of the Manual 1.2
  1.2.1. Typographic Conventions within the Manual 1.2
  1.2.2. The Organization of the Manual 1.3
1.3. The Development of Portable Standard LISP 1.6
1.4. Brief Overview of PSL as a Portable Modern LISP 1.7

CHAPTER 2. GETTING STARTED WITH PSL

2.1. Purpose of This Chapter 2.1
2.2. Defining Logical Devices for PSL on the DEC-20 2.1
2.3. Starting PSL 2.2
  2.3.1. DEC-20 2.2
  2.3.2. VAX-11/750 2.2
2.4. Running the PSL System 2.3
  2.4.1. Notes on Running PSL and RLISP 2.3
  2.4.2. Transcript of a Short Session with PSL 2.3
2.5. Error and Warning Messages 2.6
2.6. Compilation Versus Interpretation 2.7
2.7. Function Types 2.7
2.8. Flags and Globals 2.8
2.9. Reporting Errors and Misfeatures 2.8

CHAPTER 3. RLISP SYNTAX

3.1. Motivation for RLISP Interface to PSL 3.1
3.2. An Introduction to RLISP 3.2
  3.2.1. LISP equivalents of some RLISP constructs 3.2
3.3. An Overview of RLISP and LISP Syntax Correspondence 3.3
  3.3.1. Function Call Syntax in RLISP and LISP 3.4
  3.3.2. RLISP Infix Operators and Associated LISP Functions 3.4
  3.3.3. Differences between Parse and Read 3.6
  3.3.4. Procedure Definition 3.6
  3.3.5. Compound Statement Grouping 3.7
  3.3.6. Blocks with Local Variables 3.7
  3.3.7. The If Then Else Statement 3.8
3.4. Looping Statements 3.8

3.4.1. While Loop 3.8
3.4.2. Repeat Loop 3.8
3.4.3. For Each Loop 3.8
3.4.4. For Loop 3.9
3.4.5. Loop Examples 3.9
3.5. Flag Syntax 3.10
3.5.1. RLISP I/O Syntax 3.10

CHAPTER 4. DATA TYPES

4.1. Data Types Supported in PSL 4.1
4.1.1. Other Notational Conventions 4.4
4.1.2. Structures 4.4
4.2. Predicates Useful with Data Types 4.5
4.2.1. Functions for Testing Equality 4.5
4.2.2. Predicates for Testing the Type of an Object 4.7
4.2.3. Boolean Functions 4.8
4.3. Converting Data Types 4.9

CHAPTER 5. NUMBERS AND ARITHMETIC FUNCTIONS

5.1. Big Integers 5.1
5.2. Conversion Between Integers and Floats 5.2
5.3. Arithmetic Functions 5.2
5.4. Functions for Numeric Comparison 5.5
5.5. Bit Operations 5.6

CHAPTER 6. IDENTIFIERS

6.1. Introduction 6.1
6.2. Fields of Ids 6.1
6.3. Identifiers and the Id-Hash-Table 6.2
6.4. Property List Functions 6.3
6.4.1. Functions for Flagging Ids 6.4
6.4.2. Direct Access to the Property Cell 6.5
6.5. Value Cell Functions 6.5
6.6. Package System Functions 6.8

CHAPTER 7. LIST STRUCTURE

7.1. Introduction to Lists and Pairs 7.1
7.2. Basic Functions on Pairs 7.2
7.3. Functions for Manipulating Lists 7.4
7.3.1. Selecting List Elements 7.4

7.3.2. Membership and Length of Lists 7.6
7.3.3. Constructing, Appending, and Concatenating Lists 7.6
7.3.4. Lists as Sets 7.8
7.3.5. Deleting Elements of Lists 7.8
7.3.6. List Reversal 7.9
7.4. Comparison and Sorting Functions 7.10
7.5. Functions for Building and Searching A-Lists 7.11
7.6. Substitutions 7.13

CHAPTER 8. STRINGS AND VECTORS

8.1. Vector-Like Objects 8.1
8.2. Strings 8.1
8.3. Vectors 8.2
8.4. Word Vectors 8.4
8.5. General X-Vector Operations 8.5
8.6. Arrays 8.6
8.7. Common LISP String Functions 8.7

CHAPTER 9. FLOW OF CONTROL

9.1. Conditionals 9.1
9.1.1. The Case Statement 9.3
9.2. Sequencing Evaluation 9.4
9.3. Iteration 9.6
9.3.1. For 9.8
9.3.2. Mapping Functions 9.12
9.3.3. Do 9.14
9.4. Non-Local Exits 9.16

CHAPTER 10. FUNCTION DEFINITION AND BINDING

10.1. Function Definition in PSL 10.1
10.1.1. Notes on Code Pointers 10.1
10.1.2. Functions Useful in Function Definition 10.2
10.1.3. Short Calls on PutD for LISP Syntax Users 10.4
10.1.4. Function Definition in RLISP Syntax 10.5
10.1.5. Low Level Function Definition Primitives 10.5
10.1.6. Function Type Predicates 10.6
10.2. Variables and Bindings 10.6
10.2.1. Binding Type Declaration 10.7
10.2.2. Binding Type Predicates 10.8
10.3. User Binding Functions 10.8
10.3.1. Funargs, Closures and Environments 10.9

CHAPTER 11. THE INTERPRETER

11.1. Evaluator Functions Eval and Apply 11.1
11.2. Support Functions for Eval and Apply 11.3
11.3. Special Evaluator Functions, Quote, and Function 11.3
11.4. Support Functions for Macro Evaluation 11.4

CHAPTER 12. SYSTEM GLOBAL VARIABLES, FLAGS AND OTHER "HOOKS"

12.1. Introduction 12.1
12.2. Flags 12.1
12.3. Globals 12.3
12.4. Special Put Indicators 12.4
12.5. Special Flag Indicators 12.5
12.6. Displaying Information About Globals 12.6

CHAPTER 13. INPUT AND OUTPUT

13.1. Introduction 13.1
13.2. The Underlying Primitives for Input and Output 13.1
13.3. Opening, Closing, and Selecting Channels 13.3
13.4. Reading Functions 13.5
13.5. Scan Table Utility Functions 13.11
13.6. Procedures for Complete File Input and Output 13.11
13.7. Printing Functions 13.13
13.8. S-Expression Parser 13.17
13.8.1. Read Macros 13.18
13.9. I/O to and from Lists and Strings 13.18
13.10. Example of Simple I/O in PSL 13.20

CHAPTER 14. USER INTERFACE

14.1. Introduction 14.1
14.2. Stopping PSL and Saving a New Executable Core Image 14.1
14.3. Changing the Default Top Level Function 14.2
14.4. The General Purpose Top Loop Function 14.2
14.5. The HELP Mechanism 14.5
14.6. The Break Loop 14.6
14.7. Terminal Interaction Commands in RLISP 14.6

CHAPTER 15. ERROR HANDLING

15.1. Introduction 15.1
15.2. The Basic Error Functions 15.1
15.3. Break Loop 15.3
15.4. Interrupt Keys 15.6
15.5. Details on the Break Loop 15.6
15.6. Some Convenient Error Calls 15.7
15.7. Special Purpose Error Handlers 15.8

CHAPTER 16. DEBUGGING TOOLS

16.1. Introduction 16.1
16.1.1. Mini-Trace Facility 16.2
16.1.2. Step 16.3
16.1.3. Brief Summary of Full DEBUG Package 16.4
16.1.4. Use 16.5
16.1.5. Functions Which Depend on Redefining User Functions 16.5
16.1.6. Special Considerations for Compiled Functions 16.5
16.1.7. A Few Known Deficiencies 16.6
16.2. Tracing Function Execution 16.6
16.2.1. Saving Trace Output 16.7
16.2.2. Making Tracing More Selective 16.9
16.2.3. Turning Off Tracing 16.10
16.2.4. Automatic Tracing of Newly Defined Functions 16.11
16.3. Automatic BREAK Around Functions 16.11
16.4. A Heavy Handed Backtrace Facility 16.11
16.5. Embedded Functions 16.12
16.6. Counting Function Invocations 16.13
16.7. Stubs 16.13
16.8. Functions for Printing Useful Information 16.14
16.9. Printing Circular and Shared Structures 16.14
16.10. Library of Useful Functions 16.15
16.11. Internals and Customization 16.15
16.11.1. User Hooks 16.15
16.11.2. Functions Used for Printing/Reading 16.16
16.11.3. Flags 16.17
16.12. Example 16.18

CHAPTER 17. EDITORS

17.1. A Mini-Structure Editor 17.1
17.2. The EMODE Screen Editor 17.3
17.2.1. Windows and Buffers in Emode 17.5
17.3. Introduction to the Full Structure Editor 17.6
17.4. User Entry to Editor 17.6
17.5. Editor Command Reference 17.8

CHAPTER 18. NEW UTILITIES

18.1. Introduction 18.1
18.2. RCREF - Cross Reference Generator for PSL Files 18.1
  18.2.1. Restrictions 18.2
  18.2.2. Usage 18.3
  18.2.3. Options 18.3
18.3. Picture RLISP 18.4
  18.3.1. Running PictureRLISP on HP2648A and on TEKTRONIX 4006-1 Terminals 18.10
18.4. Tools for Defining Macros 18.11
  18.4.1. DefMacro 18.11
  18.4.2. BackQuote 18.12
  18.4.3. Sharp-Sign Macros 18.12
  18.4.4. MacroExpand 18.13
  18.4.5. DefLambda 18.13
18.5. Simulating a Stack 18.14
18.6. DefStruct 18.14
  18.6.1. Options 18.17
  18.6.2. Slot Options 18.18
  18.6.3. A Simple Example 18.18
18.7. DefConst 18.21
18.8. Find 18.21
18.9. Hashing Cons 18.22
18.10. Graph-to-Tree 18.23
18.11. Inspect Utility 18.24
18.12. Trigonometric and Other Mathematical Functions 18.25

CHAPTER 19. LOADER AND COMPILER

19.1. Introduction 19.1
19.2. The Compiler 19.1
  19.2.1. Compiling Functions into Memory 19.2
  19.2.2. Compiling Functions into FASL Files 19.2
  19.2.3. Loading FASL Files 19.3
  19.2.4. Functions to Control the Time When Something is Done 19.3
  19.2.5. Order of Functions for Compilation 19.4
  19.2.6. Fluid and Global Declarations 19.4
  19.2.7. Flags Controlling Compiler 19.5
  19.2.8. Differences between Compiled and Interpreted Code 19.6
  19.2.9. Compiler Errors 19.7
19.3. The Loader 19.8
  19.3.1. Legal LAP Format and Pseudos 19.9
  19.3.2. Examples of LAP for DEC-20, VAX and Apollo 19.9
  19.3.3. Lap Flags 19.12
19.4. Structure and Customization of the Compiler 19.13
19.5. First PASS of Compiler 19.13
  19.5.1. Tagging Information 19.14
  19.5.2. Source to Source Transformations 19.14

19.6. Second PASS - Basic Code Generation 19.14
19.6.1. The Cmacros 19.14
19.6.2. Classes of Functions 19.17
19.6.3. Open Functions 19.17
19.7. Third PASS - Optimizations 19.22
19.8. Some Structural Notes on the Compiler 19.22

CHAPTER 20. OPERATING SYSTEM INTERFACE

20.1. Introduction 20.1
20.2. System Dependent Functions 20.1
20.3. TOPS-20 Interface 20.2
20.3.1. User Level Interface 20.2
20.3.2. The Basic Fork Manipulation Functions 20.4
20.3.3. File Manipulation Functions 20.5
20.3.4. Miscellaneous Functions 20.6
20.3.5. Jsys Interface 20.6
20.3.6. Bit, Word and Address Operations for Jsys Calls 20.8
20.3.7. Examples 20.9

CHAPTER 21. SYSLISP

21.1. Introduction to the SYSLISP level of PSL 21.1
21.2. The Relationship of SYSLISP to RLISP 21.2
21.2.1. SYSLISP Declarations 21.2
21.2.2. SYSLISP Mode Analysis 21.3
21.2.3. Defining Special Functions for Mode Analysis 21.3
21.2.4. Modified FOR Loop 21.4
21.2.5. Char and IDLOC Macros 21.4
21.2.6. The Case Statement 21.5
21.2.7. Memory Access and Address Operations 21.7
21.2.8. Bit-Field Operation 21.7
21.3. Using SYSLISP 21.9
21.3.1. To Compile SYSLISP Code 21.9
21.4. SYSLISP Functions 21.10
21.4.1. W-Arrays 21.11
21.5. Remaining SYSLISP Issues 21.11
21.5.1. Stand Alone SYSLISP Programs 21.12
21.5.2. Need for Two Stacks 21.12
21.5.3. New Mode System 21.12
21.5.4. Extend CREF for SYSLISP 21.12

CHAPTER 22. IMPLEMENTATION

22.1. Overview of the Implementation 22.1
22.2. Files of Interest 22.1

22.3. Building PSL on the DEC-20 22.2
22.4. Building the LAP to Assembly Translator 22.5
22.5. The Garbage Collectors and Allocators 22.5
    22.5.1. Compacting Garbage Collector on DEC-20 22.5
    22.5.2. Two-Space Stop and Copy Collector on VAX 22.6
22.6. The HEAPs 22.6
22.7. Allocation Functions 22.8

CHAPTER 23. THE EXTENSIBLE RLISP PARSER AND THE MINI PARSER GENERATOR

23.1. Introduction 23.1
23.2. The Table Driven Parser 23.2
    23.2.1. Flow Diagram for the Parser 23.2
    23.2.2. Associating the Infix Operator with a Function 23.4
    23.2.3. Precedences 23.5
    23.2.4. Special Cases of 0 <-0 and 0 0 23.5
    23.2.5. Parenthesized Expressions 23.5
    23.2.6. Binary Operators in General 23.6
    23.2.7. Assigning Precedences to Key Words 23.7
    23.2.8. Error Handling 23.7
    23.2.9. The Parser Program for the RLISP Language 23.7
    23.2.10. Defining Operators 23.8
23.3. The MINI Translator Writing System 23.10
    23.3.1. A Brief Guide to MINI 23.10
    23.3.2. Pattern Matching Rules 23.12
    23.3.3. A Small Example 23.12
    23.3.4. Loading Mini 23.13
    23.3.5. Running Mini 23.13
    23.3.6. MINI Error messages and Error Recovery 23.13
    23.3.7. MINI Self-Definition 23.13
    23.3.8. The Construction of MINI 23.15
    23.3.9. History of MINI Development 23.16
23.4. BNF Description of RLISP Using MINI 23.17

CHAPTER 24. BIBLIOGRAPHY

CHAPTER 25. INDEX OF FUNCTIONS, GLOBALS, AND FLAGS

CHAPTER 26. INDEX OF CONCEPTS

CHAPTER 1
INTRODUCTION

1.1. Opening Remarks 1.1
1.2. Scope of the Manual 1.2
1.2.1. Typographic Conventions within the Manual 1.2
1.2.2. The Organization of the Manual 1.3
1.3. The Development of Portable Standard LISP 1.6
1.4. Brief Overview of PSL as a Portable Modern LISP 1.7

1.1. Opening Remarks

Although the programming language LISP was first formulated in 1960 [McCarthy 73], a widely accepted standard has never appeared. Various dialects of LISP have been produced (e.g. MACLISP, Inter LISP, Franz LISP, Common LISP, LISP 1.6, UCI LISP, UT LISP, LISP 360), in some cases several on the same machine! Consequently, a user often faces considerable difficulty in moving programs from one system to another. In addition, it is difficult to write and use programs which depend on the structure of the source code such as translators, editors and cross-reference programs.

In order to enhance the portability of the REDUCE Computer Algebra system [Hearn 73] and related tools (such as Meta Compilers, Editors, etc.), a reasonable dialect called Standard LISP was defined [Marti 79] and implemented on a number of machines by either mapping to an existing LISP (with slight interpreter modifications) or by implementing a totally new LISP. Machines now supporting Standard LISP include the DEC-10/DEC-20, IBM 360/370, CDC 6600/7600, CRAY-1, Univac 1108, Burroughs B1800, B6500 and B6700.

This document describes PSL (a Portable Standard LISP), a portable, efficient, "modern" LISP for a variety of machines. The interpreter is written entirely in itself, using a machine-oriented mode (i.e. a LISP-based systems implementation language) called SYSLISP [Benson 81]. Current implementations compile SYSLISP to assembly language on a DEC System 20 and on a VAX-11/750; an extended addressing DEC-20 version is planned. Implementations are under way for Motorola 68000-based personal machines. PSL is upward-compatible with Standard LISP; we have used function definitions as given in the Standard LISP Report [Marti 79] as much as possible, and have only deviated if LISP programming experience we have had since writing that Report made a change seem desirable. 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 much more specific, and users can take advantage of these details.

The goals of PSL implementations include:

a. Providing a library of LISP implementation modules 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).

b. Effectively supporting the REDUCE algebra system on a number of machines, and permitting LISP-coded algebra modules extracted from or modeled upon REDUCE to be included in applications such as CAI and CAGD.

c. Providing the same, uniform, modern LISP programming environment on all of the machines that we use (DEC-20, VAX/750, and 68000 based personal machine) of the power of Franz LISP, UCI LISP or MACLISP, with some extensions and enhancements motivated by the work on NIL, Spice LISP and the LISP Machine (now being combined into a new Common LISP [Steele 81]).

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

1.2. Scope of the Manual

While we have attempted to make this manual comprehensive, it is not intended for use as a LISP primer. Some minimal prior exposure to LISP will prove very helpful. A selection of LISP Primers is listed in the bibliography in Chapter 24; see for example [Allen 79, Charniak 80, Weissman 67, Winston 81]. The manual is intended to describe the syntax and the semantics of PSL. It is also intended to be an implementation description, and it specifies many of the details left out of the Standard LISP Report [Marti 79].

1.2.1. Typographic Conventions within the Manual

Each function is provided with a prototypical header line. Each formal parameter is given a name and followed by its allowed type. The names of classes referred to in the definition are printed like this, and parameter names are printed LIKE THIS. The name of the function looks **Like This**. If a parameter type is not commonly used, it may be a specific set enclosed in brackets {...}. For example:

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

PutD is a function with three parameters. The parameter FNAME is an id which is the name of the function being defined. TYPE is the type of function being defined and BODY is a lambda expression or a code-pointer. PutD returns the name of the function being defined. Some functions are compiled open; these have a note saying "open-compiled" next to the function type.

Functions which accept formal parameter lists of arbitrary length have the type class and parameter enclosed in square brackets indicating that zero or more occurrences of that argument are permitted. For example:

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

And is a function which accepts zero or more arguments which may be any forms.

In some cases, some code is given in the function definition. The code given is not necessarily the same as that in the source files; it is given to clarify the semantics of the function.

Some components of PSL have not yet been fully implemented. Some things which have not yet been fully implemented are documented here anyway for the sake of completeness. We flag the cases not yet implemented by the words:

[not implemented yet]

1.2.2. The Organization of the Manual

The 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. It will also aid in the skimming of topics online.

The rest of this Chapter discusses organization of the manual, the history of the development of PSL, and PSL as a modern portable LISP.

Chapter 2 is particularly useful in first using PSL. It begins with directions for starting PSL and getting help. Among other topics presented briefly are errors, a discussion of some of the consequences of PSL being both a compiled and an interpreted language, function types, and flags and globals. It would be worth while to glance at those Sections. PSL treats the parameters for various function types rather differently from a number of other dialects, and the serious user should definitely become familiar

with this information.

While most LISP implementations use only a fully parenthesized syntax, PSL gives the user the option of using an ALGOL-like (or PASCAL-like) syntax (RLISP), which many users find more pleasant. Chapter 3 describes the syntax of RLISP. Most of the examples and definitions in this manual are presented in that syntax.

[??? Should we use only Lisp-like through-out ???]

Chapter 4 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 5, 6, 7, and 8 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 (5, 6 and 7) should be included in an overview. As vectors and strings are used less extensively, Chapter 8 may be skipped on a first reading.

Chapter 9 and, to some extent, Chapter 4 describe the basic functions used to drive a computation. The reader wanting an overview of PSL should certainly read these two.

Chapter 10 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. Also described in this Section is context-switching in the form of the funarg and closures.

Chapter 11 describes functions associated with the interpreter. It includes functions having to do with evaluation (Eval and Apply.)

Chapter 12 describes the global variables which control the operation of PSL.

Chapter 13 describes the I/O facilities. Most LISP programs are written with almost no 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 14 presents information about the user interface for PSL. It includes some generally useful information on running the system.

Chapter 15 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 16 discusses some tools for debugging and statistics gathering based on the concept of embedding function definitions.

Chapter 17 describes the structure editor, which permits the user to construct and modify list structure, including the bodies of interpreted functions, and erroneous expressions within the BREAK loop.

Chapter 18 briefly describes modules of interest that have been implemented only recently in PSL, and have not been given Sections or Chapters of their own. Currently this includes the PSL cross-reference generator, and various macro tools.

The rest of the manual may be skipped on first reading.

Chapter 19 describes functions associated with the compiler. Chapter 20 describes some functions for communicating with the TOPS-20 and UNIX operating systems. Chapter 21 describes SYSLISP, a language incorporating features from both BCPL and LISP and which is used as an implementation language for PSL. Chapter 22 presents details of the portable implementation which may be of interest to sophisticated users, including a description of the garbage collector. Chapter 23 describes the extensible parser. Section 23.4 provides BNF descriptions of the input accepted by the token scanner, standard reader, and syntactic (RLISP) reader.

Chapter 24 contains the bibliography.

In Chapter 25 is an alphabetical index of all defined LISP functions and globals. Chapter 26 is an alphabetical index of concepts.

1.3. The Development of Portable Standard LISP

[??? put into appendix ???]

In this Section we give a brief history of the development of PSL and mention current and future work.

In 1966, a model for a standard LISP subset was produced [Hearn 66] as part of a general effort to make REDUCE [Hearn 73], a large LISP-based algebraic manipulation program, as portable as possible. The goal of this work was to define a uniform subset of LISP 1.5 and its variants so that programs written in this subset could run on any reasonable LISP system.

In the intervening years, two deficiencies in the approach taken in the original proposal [Hearn 66] emerged. First, in order to be as general as possible, the specific semantics and values of several key functions were left undefined. Consequently, programs built on this subset could not be written with any assumptions made about the form of the values of such functions. The second deficiency related to the proposed method of implementation of this language. The model considered two versions of LISP on any given machine, namely Standard LISP and the LISP of the host machine, which we shall refer to as Target LISP. This meant that if any definition were stored as interpretive Target LISP, it would vary from implementation to implementation; consequently, one could not write programs in Standard LISP which needed to assume any knowledge about the structure of such forms. This deficiency became apparent during recent work on the development of a portable compiler for LISP [Griss 81]. It is clearly easier to write a compiler if we deal with a single dialect (Standard LISP) than if we must change it to conform with the various Target LISPs.

As a result of this study, we produced a more aggressive definition of Standard LISP in the Standard LISP Report [Marti 79]. That paper can serve as a standard for a reasonably large subset of LISP with as precise as possible a statement about the semantics of each function.

Recent work has concentrated on producing a complete specification and portable implementation of a LISP based on Standard LISP. Experience with a Portable LISP Compiler (hereafter PLC) [Griss 81] and with an earlier experimental portable LISP implementation [Griss 79a]) has led to the current PSL implementation strategy: write most of the system in LISP, compiled with the PLC. A small non-LISP kernel is written in a portable, LISP-like systems language, SYSLISP.

The previous systems had the problem that the special implementation language (called BIL), although oriented to LISP implementations, was a distinct language from LISP, so that communication between "system" code

and "LISP" code was difficult. The pattern-driven BIL compiler was not very efficient. Consequently, the BIL work resulted in a number of experimental LISPs on a number of machines. These implementations were quite flexible, portable, and useful for LISP and REDUCE on machines that did not already have any LISP, but somewhat inefficient. We therefore developed the much more powerful, LISP-like systems language, SYSLISP, in which to recode all useful modules. SYSLISP has been targeted to high-level languages (such as FORTRAN, PASCAL, C or ADA), and also to assembly code. We believe this approach will advance our goal of producing a portability strategy which could lead to a system efficient enough for realistic experiments with computer algebra and ultimately to portable, production quality systems.

DEC-20 PSL is built by cross-compiling kernel modules written in RLISP/SYSLISP to DEC-20 assembly code (currently MIDAS), using a SYSLISP compiler running on a previous PSL system. PSL was first bootstrapped from a LISP 1.6-based Standard LISP. Additional modules are then loaded as LAP, FASL, or are recompiled with the resident compiler.

VAX PSL was boot-strapped from DEC-20 PSL, and it, too, has a resident LAP, FASL, and compiler.

1.4. Brief Overview of PSL as a Portable Modern LISP

PSL is an extended Standard LISP, written entirely in itself and SYSLISP and compiled with an extended Portable LISP Compiler (with machine-oriented extensions). It is to be used as a transportable replacement for the various Standard LISP interpreters which we have used to support the REDUCE algebra system and other LISP-based tools. The extensions make it more efficient and pleasant than Standard LISP for our programming activities and allow it to maintain a high degree of compatibility with Standard LISP, so that existing software can run without much change. By writing the entire LISP in an extended LISP, we are able to more rapidly experiment with changes to produce a range of LISP-like systems for other purposes, using LISP techniques for debugging even systems code.

PSL currently provides the following facilities on the DEC-20 and VAX-11/750:

a. A variety of data types, represented as explicitly tagged items: id, inum, fixnum, float, bignum, pair, string, code-pointer, word-Vector, vector, env-pointer, etc.

b. Compacting garbage collector on the DEC-20, stop-and-copy on the VAX.

c. Shallow Binding using a Binding Stack for old values, funarg based on Baker's method.

d. **Catch** and **Throw**, used to implement Error/ErrorSet and interpreted control structures such as **Prog**, **While**, etc.

e. A single lambda type, corresponding to the interpreted form of compiled code. **Apply**({lambda, code-pointer}, [ARGS]) passes the ARGS in a set of registers, used by the Standard LISP/SYSLISP compiler. Compiled and interpreted code is uniformly called through the function cell, which always contains the address of executable code. We have chosen the function calling mechanism to optimize the speed of calling a compiled function from within another compiled function without sacrificing the ability to trace or redefine any functions.

f. All argument processing of the various Spread/Eval/macro combinations (expr, fexpr, macro, nexpr), are attributes of an id only, not of code or lambdas. This permits uniform, efficient compilation.

g. Resident LISP/SYSLISP Compiler and Loader (LAP), permitting LISP and SYSLISP code to be intermixed; FAST loader (FAP).

h. Stream-directed I/O, with the ability to associate user functions (e.g. windows) with any stream; JSYS "native mode" I/O and a clean interface to TOPS-20 via JSYS function enables rapid addition of features on the DEC-20; Compress and Explode are done as I/O to and from an "in-core file".

i. Table driven scanner, with user definable scanner-tables (these are simply vectors, which can be easily switched); Read-Macros and Splice-Macros.

j. Simple interrupt processing (user hits BREAK computation to enter a BREAK Loop, or to return to top-level, or to print name of current function), interface for Stack Overflow, Arithmetic Overflow, etc. (Not on DEC-20)

k. TRACE and BREAK packages, invokable within continuable and non-fatal errors (a **Read-Eval-Print** loop before stack unwind); has stack backtrace, error correction, LISP-DDT, structure editor, primitives to examine some of the Stack and Bstack, examine LISP and SYSLISP variables, RETRY a computation that caused a continuable-error after correction, etc.

[??? Relate to features in MACLISP, FranzLISP, CommonLISP, InterLISP, etc ???]

[??? Detail what to avoid in writing code to run on other Standard LISPs. Maybe write and talk about a "Lint" program? ???]

CHAPTER 2
GETTING STARTED WITH PSL

2.1. Purpose of This Chapter 2.1
2.2. Defining Logical Devices for PSL on the DEC-20 2.1
2.3. Starting PSL 2.2
2.3.1. DEC-20 2.2
2.3.2. VAX-11/750 2.2
2.4. Running the PSL System 2.3
2.4.1. Notes on Running PSL and RLISP 2.3
2.4.2. Transcript of a Short Session with PSL 2.3
2.5. Error and Warning Messages 2.6
2.6. Compilation Versus Interpretation 2.7
2.7. Function Types 2.7
2.8. Flags and Globals 2.8
2.9. Reporting Errors and Misfeatures 2.8

2.1. Purpose of This Chapter

This Chapter is for beginning users of PSL on the DEC-20 and the VAX-11/750 at Utah. It also is meant to be a guide to those familiar with LISP, and particularly STANDARD LISP, who would like to use PSL as they read the manual.

It begins with descriptions of how to set up various logical device definitions required by PSL and how to run PSL. A number of miscellaneous hints and reminders are given in the remainder of the Chapter.

2.2. Defining Logical Devices for PSL on the DEC-20

It is absolutely essential that one say TAKE <PSL>LOGICAL-NAMES.CMD to the EXEC before entering PSL. This is not just an option; PSL is written to rely on these logical device definitions. For example, "PH:" is defined as the directory (or search list) on which PSL looks for help files, "PL:" is the directory (or search list) on which PSL looks for Lap and Fasl files, etc. One can provide these definitions easily by entering the TAKE command given above into one's LOGIN.CMD file.

There is no equivalent of logical device definitions on the VAX.

[??? Perhaps we can define standard Csh aliases or variables ???]

2.3. Starting PSL

2.3.1. DEC-20

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. [Messages concerning bugs or mis-features should be directed to PSL-BUGS@UTAH-20; see section 2.9.]

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. In particular, the compiler is not normally part of PSL.EXE.

RLISP.EXE is PSL with additional modules loaded, corresponding to the most common system run at Utah. It contains the compiler and an RLISP parser. If one types just RLISP to the EXEC, rather than PSL:RLISP, one ends up in an older version of RLISP, built upon a LISP 1.6 system. For more information about RLISP see Chapter 3.

It is recommended 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.

2.3.2. VAX-11/750

The executable files are /usr/local/psl and /usr/local/rlisp.
Loadable files are /usr/local/lib/psl/*.b.
Help files are on /usr/local/lib/psl/help.

/usr/local/psl is analogous to psl:psl.exe on the DEC-20, and /usr/local/rlisp is analogous to psl:rlisp.exe on the DEC-20. This version has the RLISP parser and compiler. Additional modules can be loaded from /usr/local/lib/psl using the **Load** function. <Ctrl-C> causes a call to **Error**, and may be used to stop a runaway computation. <Ctrl-Z> or the function **Quit** cause the process to be stopped, and control returned to the shell; the process may be continued. A sequence of <Ctrl-D>'s (EOF) causes the process to be terminated. This is to allow the use of I/O redirection from the shell.

Unix only allows 14 characters for file names, and case is significant. The use of ".r" instead of ".red" is recommended as the extension for RLISP files to save on meaningful characters; other extensions are as on the DEC-20.

2.4. Running the PSL System

2.4.1. Notes on Running PSL and RLISP

a. Use **Help**(); [(**Help**) in LISP] for general help or an indication of what help is available; use **Help** (a, b, c); [(**Help** a b c) in LISP] for information on topics a, b, and c. This call prints files from the PH: (i.e. <PSL.HELP>) directory. Try **Help** x; [(**Help** x) in LISP] on:

?	Exec	Mini	Step
Br	Find	MiniEditor	Strings
Break	Flags	MiniTrace	TopLoop
Bug	For	Package	Tr
Debug	Globals	PRLISP	Trace
Defstruct	GSort	PSL	UnBr
Edit	Help	RCREF	UnTr
EditF	JSYS	RLISP	Useful
Editor	Load	ShowFlags	ZFiles
Emode	Manual	Slate	ZPEdit
EWindow			

[??? Help() does not work in RLISP ???]

b. File I/O needs string-quotes (") around file names. File names may use full TOPS-20 or UNIX conventions, including directories, sub-directories, etc.

Input in RLISP mode is done using the **In** "File-Name"; command.

Use (**Dskin** "File-Name") for input from LISP mode.

For information on similar I/O functions see Chapter 13.

c. Use **Quit**; [(**Quit**) in LISP] or <Ctrl-C> on the DEC-20 (<Ctrl-Z> on the VAX) to exit. <Ctrl-C> (<Ctrl-Z> on the VAX) is useful for stopping run-away computations. On the DEC-20, typing START or CONTINUE to the @ prompt from the EXEC usually restarts in a reasonable way.

2.4.2. Transcript of a Short Session with PSL

The following is a transcript running RLISP on the DEC-20.

```
@psl:rlisp
[RLISP0]
PSL 3.0 Rlisp, 28-Jun-82
```

```
[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] Lisp 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]                             % To find out how to use the trace
[5] Help MiniTrace;             %    function use the function "Help".
The Mini-Trace Package:
-----------------------

The following 4 functions (all FEXPRs) are defined:
 (they each redefine the functions, saving an old definition)

TR ([F:id])               Cause TRACE message to be printed on entry to
                           and exit from calls to the functions F1 ... Fn.
UNTR ([F:id])             Restore original definitions

BR ([F:id])               Cause BREAK on entry and on exit from functions,
                           permitting arguments and results to be examined
                           and modified.
UNBR ([F:id])             Restore original definitions of the functions
                           F1 ... Fn.

Fluids:
-------
TrSpace!*                  Controls indentation, may need to be reset to 0
                           in "funny" cases.
!*NoTrArgs                 Set to T to suppress printing of arguments of
                           traced functions.

[See also the Full DEBUG package (do Help Debug; in RLISP, (Help
Debug) in LISP).]
NIL
[6]                     % Trace the recursive execution of "Count".
[6] Tr Count;           % Notice that this causes redefintion.
*** Function "COUNT" has been redefined
NIL
[7]                     % A call on "Count" now shows the value of
[7]                     %   "Count" and of its argument each time it
[7] Count Z;            %   is called.
 COUNT 1: Arg1:=(1 2 3),
  COUNT 2: Arg1:=(2 3),
   COUNT 3: Arg1:=(3),
    COUNT 4: Arg1:=NIL,
    COUNT 4:=0
```

```
   COUNT 3:=1
  COUNT 2:=2
 COUNT 1:=3
3
[8] Lisp Procedure Factorial X;
[8]   If X <= 1 Then 1
[8]     Else X * Factorial (X-1);
FACTORIAL
[9] Tr Factorial;
*** Function "FACTORIAL" has been redefined
NIL
[10] Factorial 4;             % Trace execution of "Factorial".
 FACTORIAL 1: Arg1:=4,
  FACTORIAL 2: Arg1:=3,
   FACTORIAL 3: Arg1:=2,
    FACTORIAL 4: Arg1:=1,     % Notice values being returned.
    FACTORIAL 4:=1
   FACTORIAL 3:=2
  FACTORIAL 2:=6
 FACTORIAL 1:=24
24
?         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.
MAIN COUNT COND PLUS PLUS2 COUNT CDR TOPLOOP APPLY BREAKEVAL
NIL
3 lisp break> Q               % To exit the Break Loop.
[13]                          % Load in a file, showing the file
[13] 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
[14]                          % This leaves RLISP and enters
[14] End;                     %   LISP mode.
Entering LISP...
```

```
PSL 3.0, 9-Jun-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...
[15] Quit;                  % To exit call "Quit".
@continue
"Continued"
[16] X;                     % Notice the prompt number.
3
[17] ^C                     % One can also quit with <Ctrl-C>.
@start                     % Alternative immediate re-entry.
[18] Quit;
@
```

2.5. Error and Warning Messages

Many functions detect and signal appropriate errors (see Chapter 15 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, give the expected type, and print 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 Chapter 14. 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.

The flag !*Usermode, described in section 12.2, can be set to suppress these warning messages.

2.6. Compilation Versus Interpretation

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

The exact semantic differences between compiled and interpreted functions are more fully discussed in Chapter 19 and in the Portable LISP Compiler paper [Griss 81].

[??? We intend to consider the modification of the LISP semantics so as to ensure that these differences are minimized. If a conflict occurs, we will restrict the interpreter, rather than extending (and slowing down) the capabilities of the compiled code. ???]

We indicate on the function definition line if it is typically compiled OPEN; this information helps in debugging code that uses these functions. These functions do not appear in backtraces and cannot be redefined, traced or broken in compiled code.

[??? Should we make open-compiled functions totally un-redefinable without special action, even for interpreted code. Consistency! E.g. flag 'COND LOSE. ???]

2.7. 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.

fexpr A NoEval, NoSpread function. There is no limit on the number of arguments. In referring to the formal parameters we mean the unevaluated argument.

nexpr An Eval, NoSpread function.

macro The macro is a function which creates a new S-expression for subsequent evaluation or compilation. There is no limit to the number of arguments a macro may have. The descriptions of the Eval and Expand functions in Chapter 11 provide precise details.

2.8. Flags and Globals

Generally, flag names begin with !* and global names end with !*,where "!" is an escape character. One can set a flag !*xxx to T by using On xxx; in RLISP [(on xxx) in LISP]; one can set it to NIL by using Off xxx; in RLISP [(off xxx) in LISP]. For example) !*ECHO, !*PVAL and !*PECHO are flags that control Input Echo, Value Echo and Parse Echo. These flags are described more fully in Chapters 13 and 14.

For more information, type "HELP FLAGS;" or "HELP GLOBALS;", or see Chapter 12.

2.9. Reporting Errors and Misfeatures

Send MAIL to PSL-BUGS@UTAH-20. This remails messages to a number of users on a mailing list, and place a message in <PSL>USER-BUGS.TXT.

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 Griss and Benson. 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.

[??? needs flags ???]

CHAPTER 3
RLISP SYNTAX

3.1. Motivation for RLISP Interface to PSL 3.1
3.2. An Introduction to RLISP 3.2
3.2.1. LISP equivalents of some RLISP constructs 3.2
3.3. An Overview of RLISP and LISP Syntax Correspondence 3.3
3.3.1. Function Call Syntax in RLISP and LISP 3.4
3.3.2. RLISP Infix Operators and Associated LISP Functions 3.4
3.3.3. Differences between Parse and Read 3.6
3.3.4. Procedure Definition 3.6
3.3.5. Compound Statement Grouping 3.7
3.3.6. Blocks with Local Variables 3.7
3.3.7. The If Then Else Statement 3.8
3.4. Looping Statements 3.8
3.4.1. While Loop 3.8
3.4.2. Repeat Loop 3.8
3.4.3. For Each Loop 3.8
3.4.4. For Loop 3.9
3.4.5. Loop Examples 3.9
3.5. Flag Syntax 3.10
3.5.1. RLISP I/O Syntax 3.10

3.1. Motivation for RLISP Interface to PSL

Most of the PSL users at Utah prefer to write LISP code using a 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, case statement, **If-Then-Else**, 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. Some PSL functions in this manual are accompanied by a definition in RLISP; such definitions are guides to the function's semantics rather than exact copies of code used. Sometimes a LISP S-expression form is also given, if it is not simply related to the RLISP form. The exact definitions of all PSL functions can be found in the sources (see Chapter 22 for a guide to the PSL sources). A complete BNF-like definition of RLISP and its translation to LISP using the MINI system is given in Section 23.4. Also discussed in Chapter 23 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 introduction to make the examples and sources readable, and to assist the user to write RLISP code.

3.2. 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 flags, described in appropriate Sections.

3.2.1. LISP equivalents of some RLISP constructs

The following gives a small set of RLISP statements and functions and their corresponding LISP forms. To see the exact LISP equivalent of RLISP code, set the flag !*PECHO to T [On PEcho; in RLISP].

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))))
```

3.3. 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 flag !*PECHO to T [On Pecho; in RLISP]. 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)))
```

In RLISP, a function is recognized as an "ftype" (one of the tokens EXPR, FEXPR, etc. or none) 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. In LISP syntax, a function is defined using one of the "Dx" functions, i.e. one of De, Df, Dm, or Dn, depending on "ftype". For example:

```
EXPR PROCEDURE NULL(X);
  EQ(X, NIL);
   ==>  (DE NULL (X) (EQ X NIL))
```

3.3.1. 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.

3.3.2. 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 23 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 to the left (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:

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, an 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.

3.3.3. 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 (SETQ X '(Y . Z)). Note that **Read** called from LISP or from RLISP uses slightly different scan tables (see Chapter 13). In order to use the function **Cons** in LISP one must use the word **Cons** in a prefix position.

3.3.4. Procedure Definition

Procedure definitions in PSL (both RLISP and LISP) are not nested as in ALGOL; all appear at the same top level as in C. The basic function for defining procedures is **PutD** (see Chapter 10). Special syntactic forms are provided in both RLISP and LISP:

```
  mode ftype PROCEDURE name(v_1,...,v_n); body;
     ==> (Dx name (v_1 ... v_N) body)
```

Examples:

```
  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.

In the general definition given above "mode" is usually optional; it can be LISP or SYMBOLIC (which mean the same thing) 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.

The smacro is a simple substitution macro.

```
SMACRO PROCEDURE ELEMENT X;     % Defines ELEMENT(x)  to substitute
 CAR CDR (X);                   % as Car Cdr x;
   ==> (DS ELEMENT (X) (CAR (CDR X)))
```

In code which calls ELEMENT after it was defined, ELEMENT(foo); behaves exactly like CAR CDR foo;.

3.3.5. 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)))
```

3.3.6. 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))
```

3.3.7. The If Then Else Statement

RLISP provides an **If** statement, which maps into the LISP **Cond** statement. See Chapter 9 for full details. For example:

```
IF e THEN s;
   ==> (COND (e s))

IF e THEN s1 ELSE s2;
   ==> (COND (e s1) (T s2))

IF e1 THEN s1
 ELSE IF e2 THEN s2
 ELSE s3;
   ==> (COND (e1 s1)
             (e2 s2)
             (T  s3))
```

3.4. Looping Statements

RLISP provides **While**, **Repeat**, **For** and **For Each** loops. These are discussed in greater detail in Chapter 9. Some examples follow:

3.4.1. While Loop

```
WHILE e DO s;          % As long as e NEQ NIL, do s
   ==>  (WHILE e s)
```

3.4.2. Repeat Loop

```
REPEAT s UNTIL e;      % repeat doing s until "e" is not NIL
   ==>  (REPEAT s e)
```

3.4.3. 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 Chapter 9.
```

3.4.4. For Loop

The For loop permits an iterative form with a compacted control variable.

```
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)

      % Other options can compute sums and products.
```

3.4.5. 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)))
```

3.5. Flag Syntax

Two declarations are offered to the user for turning on or off a variety of flags in the system. Flags are global variables that have only the values T or NIL. By convention, the flag name is XXXX, but the associated global variable is !*XXXX. The RLISP commands ON and OFF take a list of flag names as argument and turn them on and off respectively (i.e. set the corresponding !* variable to T or NIL).

Example:

```
ON ECHO, FEE, FUM;     % Sets !*ECHO, !*FEE, !*FUM to T;
   ==> (ON '(ECHO FEE FUM))

OFF INT,SYSLISP;        % Sets !*INT and !*SYSLISP to NIL
   ==> (OFF '(INT SYSLISP))
```

[??? Mention SIMPFG property ???]

See Chapter 12 for a complete set of flags and global variables.

3.5.1. RLISP I/O Syntax

RLISP provides special commands to OPEN and SELECT files for input or for output and to CLOSE files. File names must be enclosed in "....". Files with the extension ".sl" or ".lsp" are read by In in LISP mode rather than RLISP mode.

```
IN "<griss.stuff>fff.red","ggg.lsp"; % First reads fff.red
                                     % Then reads ggg.lsp
OUT "keep-it.output";                % Diverts output to "keep-it.output"
OUT "fum";                           % now to fum, keeping the other open
SHUT "fum";                          % to close fum and flush the buffer
```

File names can use the full system conventions. See Chapter 13 for more detail on I/O.

CHAPTER 4
DATA TYPES

4.1. Data Types Supported in PSL 4.1
4.1.1. Other Notational Conventions 4.4
4.1.2. Structures 4.4
4.2. Predicates Useful with Data Types 4.5
4.2.1. Functions for Testing Equality 4.5
4.2.2. Predicates for Testing the Type of an Object 4.7
4.2.3. Boolean Functions 4.8
4.3. Converting Data Types 4.9

4.1. Data Types Supported in PSL

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 DEC-20), an _info_ field (18 bits on DEC-20), and 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 13 for full details on syntax, restrictions, etc.). More explicit implementation details can be found in Chapters 21 and 22.

The basic data types supported in PSL and a brief indication of their representation are described below.

integer The _integers_ are also called "fixed" numbers. The magnitude of _integers_ is essentially unrestricted if BIG is loaded. The _notation_ 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, do LOAD 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 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 dty(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. The id space contains the pointers and structures. The notation for an id is an alphanumeric character sequence, starting with a letter. If presented with an appropriate identifier character string, the PSL reader usually finds a unique id to associate with this string. See Chapters 6 and 13 for more information on the fields, specific id syntax, escape characters, case conversion, etc.

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 13).

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 8.3. 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, sub-string and concatenation functions, defined in Chapter 8. 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 13.

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

[??? The w-vector could be used to implement machine-code blocks on some machines. ???]

Byte-Vector
: 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
: 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.

env-pointer
: A data type used to support a funarg capability. [**not implemented yet**]

Others
: An implementation may have functions which deal with specific data types other than those listed above. The use of these entities is to be avoided in portable PSL code (see Chapters 21 and 22 for more details.) These include low-level, machine-dependent SYSLISP objects, such as:

s-integer
: An untagged word, or multi-word on some machines, used to represent a machine-level integer. This is used to communicate with the SYSLISP level and with some operating system services. Inums are a subset of s-integers. s-integers will be phased out eventually.

word
: A true machine word, the size of which is implementation-dependent (36 bits on DEC-20, same size as item, fits in one register, etc.).

[??? Other low-level types ???]

4.1.1. 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. An 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 12).

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 11).

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 [not implemented yet]}. The ftype is ONLY an attribute of identifiers, and is not associated with either executable code (code-pointers) or lambda expressions.

io-channel — A small integer representing an io channel.

number — The class of {integer, float}.

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

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

None Returned — A notational convenience used to indicate control functions that do not return directly to the calling point, and hence do not return a value. (e.g. Go)

4.1.2. 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 extra parentheses and dots. The list (a . (b . (c . NIL))) in list-notation is (a b c). List-notation and dot-notation may be mixed, as in (a b . c) or (a (b . c) d), which are equivalent to (a . (b . c)) and (a . ((b . c) . (d . NIL))), respectively. (See Section 3.3.3.) Note: () is an alternate input representation of NIL.

a-list An association list; each element of the list is a pair, the **Car** part being a key associated with the value in the **Cdr** part.

form Any list which is legally acceptable to **Eval**; that is, it is syntactically and semantically accepted by the interpreter or the compiler. (See Chapter 11 for more details.)

lambda A lambda expression which 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 with the **Eval** function (see Chapter 11).

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

4.2. 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.

4.2.1. Functions for Testing Equality

[??? This stuff should probably be expanded for the folks who aren't too terribly familiar with LISP ???]

Functions for testing equality are listed below. For other functions comparing arithmetic values see Chapter 5.

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
    X:='(A B C); and  Y:=X;  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
```

LispEqual (U:any, V:any): boolean *expr*

An alternate name for **Equal** for use in SYSLISP mode.

Neq (U:any, V:any): boolean *macro*

Not **Equal**.

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

Not(U **Eq** 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 8.7.1). 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 **Car** U **Eq** V. If the first argument is not a pair, **EqCar** returns NIL.

4.2.2. 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 4.2.3. 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.

4.2.3. 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 9.1). For example, the following two constructs should give the same result:

```
IF A AND B AND C THEN D;

A AND B AND C AND 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:form): boolean open-compiled, expr

Returns T if U is NIL. This is exactly the same function as **Null**, defined in Section 4.2.2. 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.

4.3. Converting Data Types

The following functions are used in interconverting data types. They are grouped according to the type returned. Numeric types may be interconverted using functions such as **Fix** and **Float**, described in Section 5.2.

Intern (U:{id,string}): id expr

> Converts string to 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. 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. If U has more than the maximum number of characters permitted by the implementation (???), an error is signalled:
>
> ***** Too many characters to INTERN
>
> **[??? Rewrite for package system; include search path, global, local, intern, etc. See the Ids Chapter. ???]**
>
> 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.

Int2Id (I:integer): id expr

Convert integer to id pointer; this refers to the I'th id in the id space. Since 0 ... 127 correspond to ASCII characters, **Int2Id** converts an ASCII code to the corresponding single character id.

Id2Int (D:id): integer expr

Returns the id space position of id as a LISP number.

Sys2Int (U:s-integer): integer expr

Convert a word-sized integer to a LISP number; space may have to be allocated from the heap if not in inum range.

Int2Sys (I:integer): s-integer expr

Converts a tagged LISP integer to an untagged full-word s-integer.

Id2String (D:id): string expr

Get name from id space. **Id2String** returns the **Print** name of its argument as a string. This is not a copy, so destructive operations should not be performed on the result. See **CopyString** in Chapter 8.

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

String2List (S:string): inum-list expr

Creates a list of **Length** one plus **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 ???]

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.

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

String ([I:inum]): string nexpr

Creates and returns a string containing all the inums given.

Vector2String (V:vector): string expr

Pack the small integers in the vector into a string of the same **Size**, using the integers as ASCII values.

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

String2Vector (S:string): vector expr

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

Vector2List (V:vector): list expr

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

List2Vector (L:list): vector expr

Copy the elements of the list into a vector of the same **Size**.

Vector ([U:any]): vector nexpr

Creates and returns a vector containing all the Us given.

CHAPTER 5
NUMBERS AND ARITHMETIC FUNCTIONS

5.1. Big Integers 5.1
5.2. Conversion Between Integers and Floats 5.2
5.3. Arithmetic Functions 5.2
5.4. Functions for Numeric Comparison 5.5
5.5. Bit Operations 5.6

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_:

***** An attempt was made to do ARITHMETIC on "XXX", which is not a number

XXX is the name of the parameter at fault. 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.

5.1. Big Integers

LOAD Big; redefines the basic arithmetic operations, including the logical operations, to permit arbitrary precision integer operations. This is not yet implemented on the VAX.

Note that fixnums which are present before loading Big can cause problems.

Most functions seem to work, including **Fix** and **Float**.

5.2. 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 4.3.

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, an 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). ???]

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. ???]

5.3. 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] load useful;
NIL
[2] Y:='(1 5 7);
(1 5 7)
[3] Decr Car Y;
0
[4] Y;
(0 5 7)
[5] Decr (Cadr Y, 3, 4);
-2
[6] 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. The quotient part is computed by **Quotient** and the remainder by **Remainder**. 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.

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 the negative truncation of the **Quotient** of **Abs** U and **Abs** V. 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:number, V:number): number — expr

If both U and V are integers the result is the integer remainder of U divided by V. If either parameter is a float, the result is the difference between U and V*(U/V), all in float (probably 0.0). If either number is negative the remainder is negative. If both are positive or both are negative the remainder is positive. An error occurs if V is zero:

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

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.

Times2 (U:number, V:number): number expr

Returns the product of U and V.

5.4. Functions for Numeric Comparison

The following functions compare the values of their arguments. Although most return a boolean quantity, note that **Max**, **Min**, and some others are extra-boolean, i.e. they return one of their arguments. For functions testing equality (or non-equality) see Section 4.2.1.

Geq (U:any, V:any): boolean macro

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

GreaterP (U:number, V:number): extra-boolean expr

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

Leq (U:number, V:number): boolean macro

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

LessP (U:number, V:number): extra-boolean expr

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

Max ([U:number]): number   macro

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

Max2 (U:number, V:number): number   expr

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

Min ([U:number]): number   macro

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

Min2 (U:number, V:number): number   expr

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

MinusP (U:any): extra-boolean   expr

Returns U if U is a number and less than 0. If U is not a number or is a positive number, NIL is returned.

OneP (U:any): extra-boolean   expr

Returns U if U is a number and has the value 1 or 1.0. Returns NIL otherwise.

ZeroP (U:any): extra-boolean   expr

Returns U if U is a number and has the value 0 or 0.0. Returns NIL otherwise.

5.5. 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 ???]

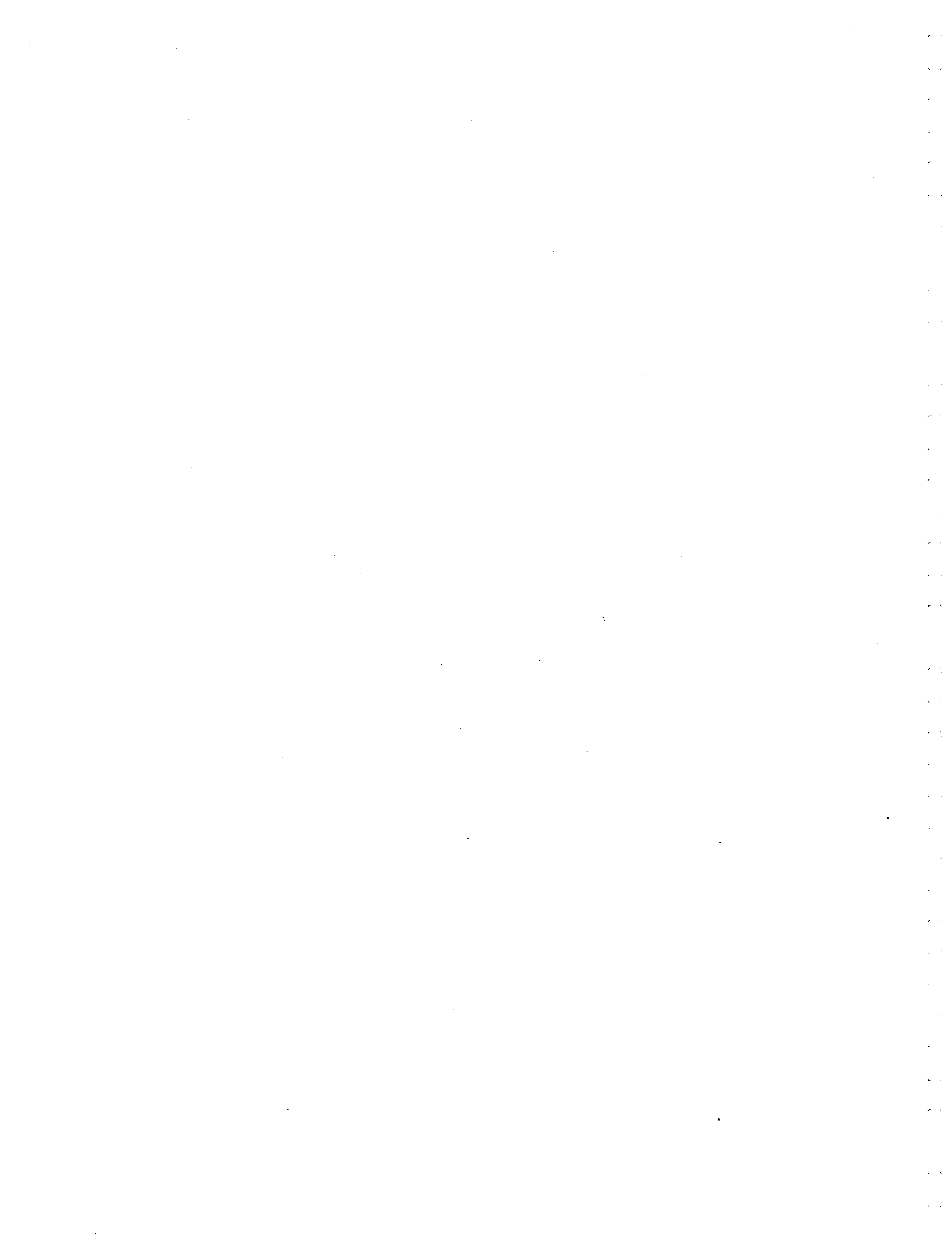
LXOr (U:integer, V:integer): integer expr

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^K. It is an arithmetic shift. Negative values are acceptable for K, and cause a right shift (in the usual manner).



CHAPTER 6
IDENTIFIERS

6.1. Introduction 6.1
6.2. Fields of Ids 6.1
6.3. Identifiers and the Id-Hash-Table 6.2
6.4. Property List Functions 6.3
6.4.1. Functions for Flagging Ids 6.4
6.4.2. Direct Access to the Property Cell 6.5
6.5. Value Cell Functions 6.5
6.6. Package System Functions 6.8

6.1. Introduction

In this Chapter ids are discussed. Ids are represented externally as a character string, typically alphanumeric. Internally they are converted into unique structures in a symbol-table with the aid of an id-hash-table. In PSL, ids are variables with values or names of functions and have property lists to store relationships between ids. PSL also provides a package system, which gives the user a tree-structured id-hash-table.

6.2. 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 5 fields. The fields (items) are:

print-name — The print name points at a string of the characters of the identifier. ids begin with a letter or any character preceded by an escape character. They may contain letters, digits, and escaped characters.

property-cell — The property cell points at a property list in the heap to store flags and (indicator . value) pairs. Access is by means of the **Flag**, **RemFlag**, **FlagP**, **Put**, **Get**, and **RemProp** functions defined in Section 6.4.

value-cell — The value used by the **Eval** function for an id is stored in this field. A binding type determines how a value is attached to an identifier and can be either LOCAL, GLOBAL, or FLUID (see Chapter 10). The **Global**, **GlobalP**, **Fluid**, **FluidP**, and **UnFluid** functions are used to access the binding type. Access to values is by means of the **ValueCell**, **Set**, and **SetQ** functions defined later 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 10.1.2.

package-cell — An id usually exists on a structure called an id-hash-table (the "oblist" in older LISPs), which is used by **Read** to find a unique id if given an identifier character string. PSL optionally permits the use of a multiple package facility (multiple id-hash-table or "multiple-oblist"). The package cell refers to the appropriate id-hash-table. Access to the id-hash-table is by means of **Intern**, **RemOb**, **Read** and other functions defined in Chapters 4, 6, and 13.

6.3. Identifiers and the Id-Hash-Table

The following functions deal with identifiers and the id-hash-table. The id-hash-table is used by **Read** and similar functions to ensure that ids that have the same input characters refer to the same id (i.e. **Eq**). This process is called Intern-ing an id or string.

Basic PSL currently provides a single id-hash-table; ids either are interned on this table or are un-interned. PSL provides all the "hooks" to permit a package system to be loaded as an option; certain functions are redefined in this process. If the package system is loaded, a tree-structured id-hash-table can be created in which each level can be thought of as a smaller id-hash-table. If a new id or string is to be interned, it is searched for in the tree according to a specified rule.

The character "\" is normally reserved in the basic Read-Table (see Chapter 13) to make up multi-part names of the form "PackageName\LocalId". If the package system is loaded, the Intern process starts searching a path in a linked structure from "PackageName", itself an id accessible in the "CurrentPackage". The print-name is still "LocalId", but the additional package field in each id records "PackageName". **Prin1** and **Prin2** are modified to access this field in loading the package system. The root of the tree is the GLOBAL package, indicated by \. If the package system is loaded, the basic id-hash-table is made into the GLOBAL package. Thus \ID is guaranteed in the root (in fact the pre-existing id-hash-table). More information is given in Section 6.6.

[??? Explain further or at least more clearly. ???]

Information on converting ids to other types can be found in Chapter 13 and Section 4.3.

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 ???]

StringGensym (): string expr

Similar to GenSym but returns a string 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): boolean expr

Test if an id is interned in the current search path.

6.4. Property List Functions

The property cell of an id usually points at 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.

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.

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.

```
EXPR PROCEDURE DEFLIST(U, IND);
  IF NULL U THEN NIL
    ELSE <<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.

MapObl (FNAME:function): Undefined expr

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

6.4.1. 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

U is a list of ids. Flag flags each id 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 after using **Flag**; that is, V becomes an element of the list **Prop** X;. 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.

FlagP (U:id, V:any): 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

Give id U flag V.

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

Remove a single flag.

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

6.4.2. 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

Access the property list of U.

SetProp (U:id, L:any): L:any expr

Store item L as the property list of U.

6.5. Value Cell Functions

The contents of the value cell are usually accessed by **Eval** (Chapter 11) or **ValueCell** (below) and changed by **Set** or **SetQ**.

Set (EXP:id, VALUE:any): any *expr*

EXP must be an identifier or a type mismatch error occurs. The effect of Set is replacement of the item bound to the identifier by VALUE. 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
```

EXP must not evaluate to T or NIL or an error occurs:

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

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

```
SETQ(X, 1);
```

is similar to

```
SET ('X, 1);
```

The value of the current binding of VARIABLE is replaced by the value of VALUE. One can use the symbol := to call this function in RLISP. SetQ now conforms to the Common LISP standard, allowing sequential assignment:

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

DeSetQ (U:any, V:any): V:any *macro*

This is a function in "USEFUL" (Load USEFUL; in RLISP). **DeSetQ** is a destructuring **SetQ**. That is, the first argument is a piece of list structure whose atoms are all ids. 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 *macro*

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

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.

On the DEC-20, 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 Chapter 21, for more direct access].

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

6.6. Package System Functions

The following fluid variables are managed by the package system, which is currently optional (LOAD PACKAGE;).

\CURRENTPACKAGE!* (Initially: Global) — global

This is the start of the search path if interning. \CurrentPackage!* is rebound in the token scanner on encountering a "\".

\PACKAGENAMES!* (Initially: (Global)) global

List of ALL package names currently created.

Our current package model uses a set of general path functions that access functions specific to each level of the id-hash-table tree to do various things: "Localxxxx(s)" and "Pathxxxx(s)" in which "xxxx" is one of the set (InternP, Intern, RemOb, MapObl). By storing different functions, each package may have a different structure and associated functions. The current implementation of a package uses a vector

[Name Father GetFn PutFn RemFn MapFn]

stored under the indicator 'Package on the PackageName id.

A simple bucket id-hash-table can also be used for experiments, or the user can build his own. As far as possible, each function checks that a legal package is given before performing the operation.

[??? Should we have a package Tag ???]

The following functions should be used.

\CreatePackage (NAME:id, FATHERPACKAGE:id): id expr

This creates a convenient size id-hash-table, generates the functions to manage it for this package, and links the new package to the FATHERPACKAGE so that path searches for ids are required.

\SetPackage (NAME:id): id expr

Selects another package such as GLOBAL\.

\PathInternP (S:{id, string}): boolean expr

Searches from CurrentPackage!* to see if S is interned.

\PathIntern (S:{id, string}): id expr

Look up or insert an id.

\PathRemob (S:{id, string}): id *expr*

Remobs, puts in NIL package.

\PathMapObl (F:function): NIL *expr*

Applies F to ALL ids in path.

\LocalInternP (S:{id, string}): boolean *expr*

Searches in CURRENTPACKAGE!*.

\LocalIntern (S:{id, string}): id *expr*

Look up or insert in CURRENTPACKAGE!* (forces ids uninterned in CURRENTPACKAGE!* into CURRENTPACKAGE!*) .

\LocalRemob (S:{id, string}): id *expr*

Remobs, puts in NIL package.

\LocalMapObl (F:function): NIL *expr*

Applies F to ALL ids in (CurrentPackage!*).

Note that if a string is used, it CANNOT include the \. Also, since most ids are "RAISED" on input, be careful.

Current intern, etc. are **\PathIntern**, etc.

Several restrictions are placed on the use of packages when compiled. Since it is a loaded module and not integrated with the basic PSL system, all ids in the compiled package are Interned in Global\ before they are defined. This requires a slightly more complex loading system for packages. Names and function ids which conflict with names in Global\ (or other packages in the path) must be forced into the id-hash-table of the desired package. The package is compiled WITHOUT the package module loaded.

In addition, if a function call must be issued for a function which has been redefined in the package the function name must be changed. When PACKAGE has been integrated with **Fasl** and PSL, it will be sufficient to prefix the function name with the package name (e.g. Global\Print). Currently, one must actually change the function name (e.g. Global!.Print).

As an example, a small package which redefines the system function **Print** is shown. The assumed file name is PrintPack.red.

```
symbolic procedure GetFieldFn(relation,field);
  slotdescslotfn
        cdr assoc(field,dsdescslotalist getdefstruct relation);

symbolic fexpr procedure Print(args);
  begin scalar fields;
    fields := Get(car args,'Fields);
    ForEach elem in eval car args do
       Global!.Print (ForEach field in fields Collect
                      apply(GetFieldFn(car args,field),'list elem));
    return car args
  end;
```

This package would be compiled as follows (immediately after entering RLISP):

```
faslout "PrintPackage";
in "PrintPack.red"$
faslend;
quit;
```

This package would be loaded as follows (immediately after entering RLISP):

```
load '(defstruct package);
CopyD('Global!.Print,Print);
<<\CreatePackage('PrintPack,'Global);
  \SetPackage('PrintPack);
  LocalIntern 'Print>>;
faslin "PrintPack.b";
```

CHAPTER 7
LIST STRUCTURE

7.1. Introduction to Lists and Pairs 7.1
7.2. Basic Functions on Pairs 7.2
7.3. Functions for Manipulating Lists 7.4
 7.3.1. Selecting List Elements 7.4
 7.3.2. Membership and Length of Lists 7.6
 7.3.3. Constructing, Appending, and Concatenating Lists 7.6
 7.3.4. Lists as Sets 7.8
 7.3.5. Deleting Elements of Lists 7.8
 7.3.6. List Reversal 7.9
7.4. Comparison and Sorting Functions 7.10
7.5. Functions for Building and Searching A-Lists 7.11
7.6. Substitutions 7.13

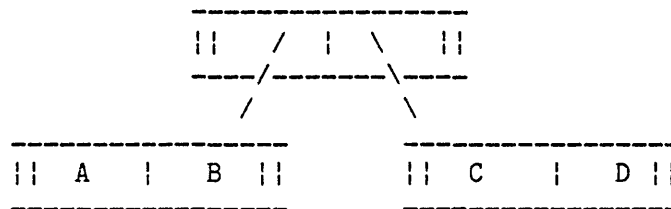
7.1. Introduction to Lists and Pairs

The pair is a fundamental PSL data type, and is one of the major attractions of LISP programming; it provides the ability to easily build and manage binary tree objects. Automatic storage management in PSL enables the user to concentrate on the programming task. Trees built of pairs are used to do sorting and to represent parse trees and complex hierarchical relationships.

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 as below in Box notation.



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); (see Section 3.3.3).

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)
```

7.2. 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)). See Section 3.3.3 for more discussion of how dot is read.

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_. **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_. **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 7.3.1), or possibly even the Defstruct package (Section 18.6).

NCons (_U_:_any_): _pair_ _open-compiled_, _expr_

Equivalent to **Cons** (_U_, NIL).

XCons (_U_:_any_, _V_:_any_): _pair_ _open-compiled_, _expr_

Equivalent to **Cons** (_V_, _U_).

Copy (_X_:_any_): _any_ _expr_

Copy all _pairs_ in an S-expression; this does not copy each element of _vectors_, etc. See **TotalCopy** in Section 8.5.

See Chapter 8 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 portion of the pair U is replaced by V. If pair 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 portion of the pair U is replaced by V. If pair 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

Replace whole pair.

```
RPLACA(RPLACD(A, CDR B), CAR B)
```

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

7.3. Functions for Manipulating Lists

The following functions are meant for the special pairs which are lists, as described in Section 7.1.

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

7.3.1. Selecting List Elements

First (L:pair): any macro

A synonym for Car L.

Second (L:pair): any *macro*

A synonym for **Cadr** L.

Third (L:pair): any *macro*

A synonym for **Caddr** L.

Fourth (L:pair): any *macro*

A synonym for **Cadddr** L.

Rest (L:pair): any *macro*

A synonym for **Cdr** L.

LastPair (L:pair): any *expr*

Last pair of a list. 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 L is atomic a type mismatch error occurs.

```
EXPR PROCEDURE LASTPAIR L;
  IF NULL REST L THEN L
   ELSE LASTPAIR REST L;
```

LastCar (L:any): any *expr*

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

Nth (L:pair, N:integer): any *expr*

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

PNth (L:list, N:integer): any *expr*

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

```
EXPR PROCEDURE PNTH(L,N);
  IF N <= 1 THEN L
   ELSE NTH(CDR L,N-1);
```

7.3.2. Membership and Length of Lists

Member (A:any, L:list): extra-boolean — expr

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

```
EXPR PROCEDURE MEMBER(A,L);
  IF NULL L THEN NIL
    ELSE IF A = FIRST L THEN L
    ELSE MEMBER(A,REST L);
```

MemQ (A:any, B:list): extra-boolean — expr

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

```
EXPR PROCEDURE MEMQ(A, L);
  IF NULL L THEN NIL
    ELSE IF A EQ FIRST L THEN L
    ELSE MEMQ(A, REST L);
```

Length (X:any): integer — expr

The top level length of the list X is returned.

```
EXPR PROCEDURE LENGTH(X);
  IF ATOM X THEN 0
   ELSE LENGTH REST X + 1;
```

7.3.3. 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.

```
EXPR PROCEDURE APPEND(U, V);
  IF NULL U THEN V
   ELSE 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 7.3 about the use of destructive functions.

```
EXPR PROCEDURE NCONC(U,V);
  IF NULL U THEN V
   ELSE << RPLACD(LASTCDR U,V);
           U >>;
```

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.

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.

7.3.4. Lists as Sets

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 Xn.

7.3.5. 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 — expr

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

```
EXPR PROCEDURE DELETE(U,V);
  IF NULL V THEN NIL
   ELSE IF FIRST V = U THEN REST V
   ELSE FIRST V . DELETE(U, REST V);
```

Del (F:function, U:any, V:list): list — expr

Generalized **Delete** function with F as the comparison function.

DeletIP (U:any, V:list): list — expr

Destructive **Delete**; modifies V using **RplacD**. Do not depend on V itself correctly referring to list.

DelQ (U:any, V:list): list *expr*

Delete U from V, using **Eq** for comparison.

DelQIP (U:any, V:list): list *expr*

Destructive version of **DelQ**; see **DeletIP**.

DelAsc (U:any, V:a-list): a-list *expr*

Remove first (U . xxx) from V.

DelAscIP (U:any, V:a-list): a-list *expr*

Destructive **DelAsc**.

DelatQ (U:any, V:a-list): a-list *expr*

Delete first (U . xxx) from V, using **Eq** to check equality with U.

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

Destructive **DelatQ**.

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**.

7.3.6. List Reversal

Reverse (U:list): list *expr*

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

```
EXPR PROCEDURE REVERSE(U);
BEGIN SCALAR W;
  WHILE U DO << W := CAR U . W;
                U := CDR U >>;
  RETURN W
END;
```

ReversIP (_U_:_list_): _list_ _expr_

Destructive Reverse.

7.4. Comparison and Sorting Functions

The module GSort (use LOAD GSORT) contains a number of general sorting functions and associated key comparison functions. The key comparison functions are given two objects to compare, and return NIL if they are not in correct order. The package defines the following **SortFns**:

NumberSortFn (_N1_:_number_, _N2_:_number_): _boolean_ _expr_

Numeric comparison _N1_ <= _N2_.

StringSortFn (_S1_:_string_, _S2_:_string_): _boolean_ _expr_

Compares elements of the strings using ASCII collating code. Case is currently observed, but a case-folding version will be added. If the characters are identical then the shorter string is first in the collating order.

IdSortFn (_D1_:_id_, _D2_:_id_): _boolean_ _expr_

Calls **StringSortFn** on the print names of the _D1_ and _D2_.

AtomSortFn (_X1_:_atom_, _X2_:_atom_): _boolean_ _expr_

Sorts _numbers_ ahead of _ids_ and _ids_ ahead of _strings_. Uses the above functions if types are same.

The general sorting functions expect a SortFn (which MUST be an _id_) of the above type.

GSortP (LST:list, SORTFN:id): boolean expr

Returns T if the list is sorted according to the SORTFN.

GSort (LST:list, SORTFN:id): list expr

Does a tree sort of LST, using SORTFN to compare elements.

GMergeSort (LST:list, SORTFN:id): list expr

Does a Merge sort of LST using SORTFN for comparison. Currently, **GSort** is often fastest, but **GMergeSort** is more stable.

[??? Its major time seems to be in splitting the original list. ???]

Example: To sort a list of ids call GSort(Dlist, 'Idsortfn) or GSort(Dlist, 'IDc2) for faster sort.

To sort a list of records (e.g. pairs), the user must define a comparison function. E.g. to sort LP, a List of dotted pairs (Number . Info), define

```
PROCEDURE NPSORTFN(P1,P2);
  NUMBERSORTFN(CAR P1, CAR P2);
```

then execute GSort(LP, 'NPSortfn);

See "PU:Gsort.Red" for the code and further ideas.

7.5. 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**.

```
EXPR PROCEDURE ASSOC(U, V);
  IF NULL V THEN NIL
    ELSE IF ATOM CAR V THEN
      ERROR(000, LIST(V,
        "is a poorly formed alist"))
    ELSE IF U = CAAR V THEN CAR V
    ELSE 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.

```
EXPR PROCEDURE SASSOC(U, V, FN);
  IF NULL V THEN FN()
    ELSE IF U = CAAR V THEN CAR V
    ELSE 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.

```
EXPR PROCEDURE PAIR(U, V);
  IF AND(U, V) THEN (CAR U . CAR V) .
        PAIR(CDR U, CDR V)
    ELSE IF OR(U, V) THEN ERROR(000,
            "Different length lists in PAIR")
    ELSE NIL;
```

7.6. Substitutions

Subst (U:any, V:any, W:any): any — expr

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

```
EXPR PROCEDURE SUBST(U, V, W);
  IF NULL W THEN NIL
   ELSE IF V = W THEN U
   ELSE IF ATOM W THEN W
   ELSE SUBST(U, V, CAR W) . SUBST(U, V, CDR W);
```

SubstIP (U:any, V:any, W:any): any — expr

Destructive **Subst**.

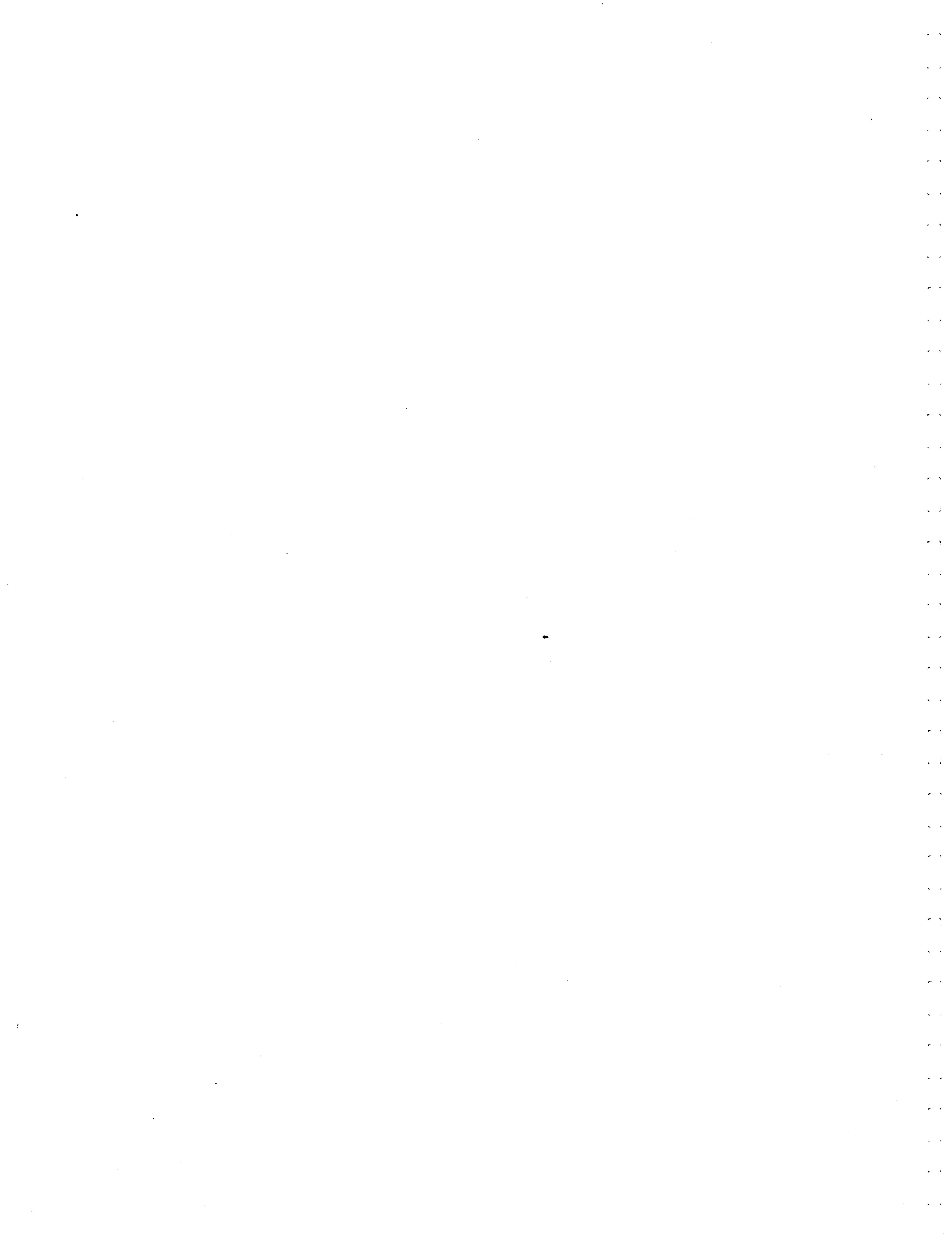
SubLis (X:a-list, Y:any): any — expr

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

```
EXPR PROCEDURE SUBLIS(X, Y);
  IF NULL X THEN Y
    ELSE BEGIN SCALAR U;
      U := ASSOC(Y, X);
      RETURN IF U THEN CDR U
             ELSE IF ATOM Y THEN Y
             ELSE SUBLIS(X, CAR Y) .
                  SUBLIS(X, CDR Y)
      END;
```

SublA (U:a-list, V:any): any — expr

Eq version of **SubLis**; replaces atoms only.



CHAPTER 8
STRINGS AND VECTORS

8.1. Vector-Like Objects 8.1
8.2. Strings 8.1
8.3. Vectors 8.2
8.4. Word Vectors 8.4
8.5. General X-Vector Operations 8.5
8.6. Arrays 8.6
8.7. Common LISP String Functions 8.7

8.1. 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. ???]

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**.

[??? Change names to GetIndex, PutIndex ???]

For functions which change an object from one data type to another, see Section 4.3.

8.2. 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.)

Make!-String (UPLIM:integer, INITVAL:integer): string expr

Construct string with UPLIM + 1 elements, each initialized to the ASCII code INITVAL.

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

An old form of **Make!-String**.

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? ???]

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 heap string, allocating space.

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

8.3. 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 13).

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. An error occurs if UPLIM is < 0 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:integer): 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

A similar effect can be obtained in RLISP by typing in V[INDEX]:=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 LOADing FAST!-VECTOR;

IGetV (): expr

Used the same way as **GetV**.

IPutV (): expr

Fast version of **PutV**.

ISizeV (): expr

Fast version of **UpbV**.

ISizeS (): expr

Fast version of **Size**.

8.4. 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 (): Word-Vector expr

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

Make!-Halfwords (): Halfword-Vector expr

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

Make!-Bytes (): 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 ???]

8.5. 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, I2:integer): x-vector expr

Extract a subrange of x-vector, starting at I1, producing a new x-vector of Size I2.

SetSub (X:x-vector, I1:integer, I2:integer, Y:x-vector): x-vector expr

Store subrange of Y of size I2 into X starting at I1.

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. For example,

```
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.

```
RLISP
[Keeping rlisp]
PSL 3.0 Rlisp, 1-Jun-82
[1] a:="0123456";
"0123456"
[2] b:="abcd";
"abcd"
[3] setsubseq(a,3,7,b);
"abcd"
[4] a;
"012abcd"
[5] b;
"abcd"
```

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

Concatenate 2 x-vectors. Currently they must be of same type.

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

TotalCopy (S:any): any expr

Unique copy of entire structure.

See also Chapter 7 for copying functions.

8.6. Arrays

Arrays do not exist in PSL as distinct data-types; rather an array macro package will be available to declare and manage multi-dimensional arrays of items, characters and words, by mapping them onto single dimension vectors.

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

8.7. Common LISP String Functions

A Common LISP compatible package of string and character functions has been implemented in PSL, obtained by LOADing STRINGS. 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 other functions with the same name.

[??? More documentation on these functions will be provided. ???]

The following functions are present for compatibility.

Standard!-CharP (C:character): boolean — expr

Non-control character.

GraphicP (C:character): boolean — expr

Printable character.

String!-CharP (C:character): boolean — expr

A character that can be an element of a string.

AlphaP (C:character): boolean — expr

An alphabetic character.

UpperCaseP (C:character): boolean — expr

An upper case letter.

LowerCaseP (C:character): boolean — expr

A lower case letter.

BothCaseP (C:character): boolean — expr

Same as **AlphaP**.

DigitP (C:character): boolean expr

A digit character (optional radix not supported).

AlphaNumericP (C:character): boolean expr

A digit or an alphabetic.

Char!= (C1:character, C2:character): boolean expr

Strict character comparison.

Char!-Equal (C1:character, C2:character): boolean expr

Similar character objects; case, font and bits are ignored.

Char!< (C1:character, C2:character): boolean expr

Strict character comparison.

Char!> (C1:character, C2:character): boolean expr

Strict character comparison.

Char!-LessP (C1:character, C2:character): boolean expr

Ignore case and bits for **Char<**.

Char!-GreaterP (C1:character, C2:character): boolean expr

Ignore case and bits for **Char>**.

Char!-Code (C:character): character expr

Character to integer conversion.

Char!-Bits (C:character): integer expr

Bits attribute of a character. Returns 0.

Char!-Font (C:character): integer expr

Font attribute of a character. Returns 0.

Code!-Char (I:integer, BITS:integer): extra-boolean expr

Integer to character conversion, optional bits, font ignored.

Character (C:character, BITS:integer, FONT:integer): extra-boolean expr

Character plus bits and font, which are ignored.

Char!-UpCase (C:character): character expr

Raise a character.

Char!-DownCase (C:character): character expr

Lower a character.

Digit!-Char (C:character): integer expr

Convert character to digit (optional radix, bits, font [not implemented yet]).

Char!-Int (C:character): integer expr

Convert character to integer.

Int!-Char (I:integer): character expr

Convert integer to character.

RplaChar (S:string, I:integer, C:character): character expr

Store a character C in a string S at position I.

The string functions follow.

String!= (S1:string, S2:string): boolean expr

Compare two strings S1 and S2 (substring options not implemented).

String!-Equal (*S1*:*string*, *S2*:*string*): *boolean* *expr*

Compare two strings *S1* and *S2*, ignoring case, bits and font.

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

Lexicographic comparison of strings.

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

Lexicographic comparison of strings.

String!<!= (*S1*:*string*, *S2*:*string*): *extra-boolean* *expr*

Lexicographic comparison of strings.

String!>!= (*S1*:*string*, *S2*:*string*): *extra-boolean* *expr*

Lexicographic comparison of strings.

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

Lexicographic comparison of strings.

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.

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

Construct a string of length I filled with character C. C is optional.

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

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

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

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

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

Remove leading characters from string.

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

Remove trailing characters from string.

String!-UpCase (S:string): string expr

Copy and raise all alphabetic characters in string.

NString!-UpCase (S:string): string expr

Destructively raise all alphabetic characters in string.

String!-DownCase (S:string): string expr

Copy and lower all alphabetic characters in string.

NString!-DownCase (S:string): string expr

Destructively raise all alphabetic characters in string.

String!-Capitalize (S:string): string *expr*

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

NString!-Capitalize (S:string): string *expr*

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

String!-to!-List (S:string): list *expr*

Unpack string characters into a list.

String!-to!-Vector (S:string): vector *expr*

Unpack string characters into a vector.

SubString (S:string, C1:character, C2:character): string *expr*

Subsequence restricted to strings. C2 is optional.

String!-Length (S:string): integer *expr*

Last index of a string, plus one.

CHAPTER 9
FLOW OF CONTROL

9.1. Conditionals . . . 9.1
9.1.1. The Case Statement . . . 9.3
9.2. Sequencing Evaluation . . . 9.4
9.3. Iteration . . . 9.6
9.3.1. For . . . 9.8
9.3.2. Mapping Functions . . . 9.13
9.3.3. Do . . . 9.15
9.4. Non-Local Exits . . . 9.17

Most of the constructs presented in this Chapter have a special syntax in RLISP. This syntax is presented along with the definitions of the underlying functions. Most of the examples are presented using RLISP syntax.

9.1. Conditionals

Cond ([U:form-list]): any   open-compiled, fexpr

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 **Else**s 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.

Taken simply as a function, without RLISP syntax, 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**. The dangling **Else** example above is:

```
(COND ((F X) (COND ((G X) (PRINT X))
                   ( T    (PRINT Y)) ) ))
```

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 9.2. 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, mostly used from LISP syntax:

If (): 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. It cannot be used in RLISP syntax.

```
(IF <TEST> <THEN-CLAUSE> <ELSE1>...<ELSEn>)
```

The <THEN-CLAUSE> is evaluated if and only if the test is non-NIL, otherwise the ELSEs are evaluated, and the last returned. There may be zero ELSEs.

Related macros for common COND forms are WHEN and UNLESS.

When (): any macro

```
(WHEN <TEST> S1 S2 ... Sn)
```

evaluates the Si and returns the value of Sn if and only if <TEST> is non-NIL. Otherwise **When** returns NIL.

Unless (): _any_ _macro_

```
(UNLESS <TEST> S1 S2 ... Sn)
   <=> (WHEN (NOT <TEST>) 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 4.2.3.

9.1.1. The 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. There is also an FEXPR, CASE, for the interpreter.

The RLISP syntax is:

```
Case-Statement ::= CASE expr OF case-list END

Case-list      ::=  Case-expr [; Case-list ]

Case-expr      ::=  Tag-expr : expr

tag-expr       ::=  DEFAULT | OTHERWISE  |
                    tag | tag, tag ... tag |
                    tag TO tag

Tag            ::=  Integer | Wconst-Integer
```

For example:

```
CASE i OF
   1:       Print("First");
   2,3:     Print("Second");
   4 to 10: Print("Third");
   Default: Print("Fourth");
END
```

The RLISP syntax parses into the following LISP form:

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

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

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

The above example becomes:

```
(CASE i ((1)              (Print "First"))
        ((2 3)            (Print "Second"))
        (((Range 4 10))   (Print "Third"))
        ( NIL             (Print "Fourth")))
```

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

9.2. 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; to use it, type LOAD USEFUL;. **Prog1** evaluates its arguments in order, like

ProgN, but returns the value of the first.

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

VARS is a list of ids which are considered FLUID if the **Prog** is interpreted and LOCAL if compiled (see the "Variables and Bindings" Section, 10.2). 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 9.4.

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)

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 open-compiled, expr

Within a **Prog**, **Return** terminates the evaluation of a **Prog** and returns 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

9.3. Iteration

While (E:form, [S:form]): NIL macro

In RLISP syntax this is **While ... Do ...** . Note that in RLISP syntax there may be only a single expression after the **Do**; however, it may be a **ProgN** delimited by <<...>>. This is the most commonly used construct for indefinite iteration in LISP. E is evaluated; if non-NIL, the elements of S are evaluated from left to right and then the process is repeated. If E evaluates to NIL the **While** returns NIL. **Exit** may be used to terminate the **While** from within the body and to return a value. **Next** may be used to terminate the current iteration.

Repeat (E:form, [S:form]): NIL *macro*

The S's are evaluated left to right, and then E is evaluated. This is repeated until the value of E is NIL, if **Repeat** returns NIL. **Next** and **Exit** may be used in the S'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 S's. The RLISP syntax for **Repeat** is **Repeat Until**. Like **While**, RLISP syntax only allows a single S, so

```
REPEAT E S1 S2
```

should be written in RLISP as

```
REPEAT << S1; S2 >> UNTIL E
```

[??? maybe do REPEAT S1 ... Sn E ???]

Next (): None Returned *open-compiled, restricted, macro*

With no argument, this terminates the current iteration of the most closely surrounding **While**, **Repeat**, or **Loop**, and causes the next to commence. See the note in Section 9.2 about the lexical restrictions on placement of this construct, which is essentially a GO.

Exit ([U:form]): None Returned *open-compiled, restricted, macro*

The U's are evaluated left to right, the most closely surrounding **While**, **Repeat**, or **Loop** is terminated, and the value of the last U is returned. With no arguments, NIL is returned. See the note in Section 9.2 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**. In RLISP use

```
WHILE E DO << S1;...;EXIT(1);...;Sn>>
```

not

```
WHILE E DO BEGIN S1;...;EXIT(1);...;Sn;END;
```

9.3.1. For

The **For** construct has been partially implemented in the resident PSL; a more satisfactory version is described below and can be loaded on the Dec-20 by loading USEFUL.

For ([S:form]): any — macro

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 an example. The following function zips together three lists into a list of three element lists.

```
(DE ZIP3 (X Y Z)
 (FOR (IN U X) (IN V Y) (IN W Z) (COLLECT (LIST U V W))))
```

The three **In** clauses specify that their first argument should take successive elements of the respective lists, and the **Collect** clause specifies that the answer should be a list built out of its argument. For example,

```
(zip3 '(1 2 3 4) '(a b c d) '(w x y z))
```

is

```
((1 a w)(2 b x)(3 c y)(4 d z))
```

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 **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 SUMLENGTHS (L)
   (FOR (IN N L LENGTH) (SUM N)))
```

For example, (SumLengths '((1 2 3 4 5)(a b c)(x y))) is 10.

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

(FROM VAR INIT FINAL STEP)
is a numeric 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 UNZIP3 (L)
  (FOR (U IN L) (WITH X Y Z)
    (COLLECT (CAR U) X)
    (COLLECT (CADR U) Y)
    (COLLECT (CADDR U) Z)
    (RETURNS (LIST X Y Z))))
```

This is essentially the opposite of **Zip3**. Given a list of three element lists, it unzips them into three lists, and returns a list of those three lists. For example, (unzip '((1 a w)(2 b x)(3 c y)(4 d z))) is ((1 2 3 4)(a b c d)(w x y z)).

(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.

(UNION EXP)
is similar, but only adds an element to the list if it is not equal to anything already there.

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

(JOIN EXP)
causes them to be appended.

(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.

(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. **For** only works in LISP syntax.

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

9.3.2. Mapping Functions

The mapping functions long familiar to LISP programmers are present in PSL. However, we believe that the **For** construct described above 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:

```
BEGIN SCALAR I;
  I := 0;
  RETURN MAPCAR(L, FUNCTION LAMBDA X; <<I:=I+1; I . X >>)
END;
```

may be expressed more transparently as

```
FOR X IN L AS I FROM 1 COLLECT I . ELEM;
```

Note that the RLISP syntax is not yet implemented.

ForEach (U:any): any macro

Although in LISP syntax one must use the form **ForEach**, in RLISP syntax one may use either **ForEach** or **For Each**. This **macro** is

used by the map functions as follows:

```
Possible forms are:
(FOREACH X IN U DO (FOO X))      --> (MAPC U (FUNCTION
                                      (LAMBDA (X) (FOO X))))
(FOREACH X IN U COLLECT (FOO X))--> (MAPCAR U ...)
(FOREACH X IN U CONC (FOO X))   --> (MAPCAN U ...)
(FOREACH X ON U DO (FOO X))     --> (MAP U ...)
(FOREACH X ON U COLLECT (FOO U))--> (MAPLIST U ...)
(FOREACH X ON U CONC (FOO X))   --> (MAPCON U ...)
```

Map (X:list, FN:function): NIL — expr

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

```
FOR EACH 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:

```
FOR EACH 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:

```
FOR EACH 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:

```
FOR EACH 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:

```
FOR EACH 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:

```
FOR EACH U ON X COLLECT FN(U);
```

9.3.3. Do

Do and **Let** are now partially implemented in a package called USEFUL. To find out more about this package type HELP USEFUL; in RLISP. To use the package type in LOAD USEFUL;.

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. It 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 _id_, 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_ _macro_

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_ _macro_

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 err out, or otherwise throw to a surrounding context).

Do-Loop!* (_A_:_list_, _B_:_list_, _C_:_list_, [_S_:_form_]): _any_ _macro_

Do-Loop!* does the variable bindings and undates 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.

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.

9.4. Non-Local Exits

[Note: Catch and Throw will probably be reimplemented shortly so that they only evaluate their second args once, not twice.]

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, None Returned} *expr*

Catch calls **Eval** on FORM. If during this evaluation **Throw**(TAG,VAL) occurs, **Catch** immediately returns VAL. If no **Throw** occurs, the value of 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 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 a **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.)

Note that **Error** and **ErrorSet** are implemented by means of **Catch** and **Throw**, using a tag of **!$ERROR!$** (see Chapter 15).

CHAPTER 10
FUNCTION DEFINITION AND BINDING

10.1. Function Definition in PSL 10.1
10.1.1. Notes on Code Pointers 10.1
10.1.2. Functions Useful in Function Definition 10.2
10.1.3. Short Calls on PutD for LISP Syntax Users 10.4
10.1.4. Function Definition in RLISP Syntax 10.5
10.1.5. Low Level Function Definition Primitives 10.5
10.1.6. Function Type Predicates 10.6
10.2. Variables and Bindings 10.6
10.2.1. Binding Type Declaration 10.7
10.2.2. Binding Type Predicates 10.8
10.3. User Binding Functions 10.8
10.3.1. Funargs, Closures and Environments 10.9

10.1. 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. The final Section presents binding functions useful in building new interpreter functions.

10.1.1. 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:nnn> on the DEC-20 and VAX). A code-pointer may not be created by **Compress**. (See Chapter 13 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.

10.1.2. 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 function 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 interpreted, the lambda expression is saved on the property list 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.

[??? Should we check arglist 0<=15 for exprs or =1 for fexprs, macros and nexprs. Should we expand macros. ???]

After using **PutD**, **GetD** returns a pair with the function's TYPE and definition. Likewise, the **GlobalP** predicate returns T if queried with the function 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 already exists, a warning message appears:

*** Function FNAME has been redefined

Note: All function types may be compiled.

The following flags are useful when defining functions.

!*REDEFMSG (Initially: T) *flag*

If !*REDEFMSG is not NIL, the message

*** Function `FOO' has been redefined

is printed whenever a function is redefined.

!*USERMODE (_Initially_: _T_) _flag_

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 14. 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_) _flag_

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 GLOBAL/function attribute of U is removed and the name may be used subsequently as a variable.

10.1.3. Short Calls on PutD for LISP Syntax Users

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 !*COMP is T.

[??? Add a macro or parsing function Args to make it possible to use these in RLISP [why bother] ???]

De (FNAME:id, PARAMS:id-list, [FN:form]): id macro

The forms FN are made into a lambda expression with the formal parameter list PARAMS, and are added to the set of defined functions with the name FNAME. Previous definitions of the function are lost. The function created is of type expr. The name of the defined function is returned.

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

The function FN with formal parameter PARAM is added to the set of defined functions with the name FNAME. Any previous definitions of the function are lost. The function created is of type fexpr. The name of the defined function is returned.

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

The function FN with formal parameter PARAM is added to the set of defined functions with the name FNAME. Any previous definitions of the function are lost. The function created is of type nexpr. The name of the defined function is returned.

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

The macro FN with the formal parameter PARAM is added to the set of defined functions with the name MNAME. Any previous definitions of the function are overwritten. The function created is of type macro. The name of the macro is returned.

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

Defines the smacro SNAME.

[??? Explain as a macro generator-- see pu:define-smacro.red. ???]

10.1.4. Function Definition in RLISP Syntax

In RLISP mode, procedures are defined by using the Procedure construct, as discussed in Chapter 3.

```
type PROCEDURE name(args);
   body;
```

10.1.5. Low Level Function Definition Primitives

The following functions are used especially by **PutD** and **GetD**, defined above in Section 10.1.2, and by **Eval** and **Apply**, defined in Chapter 11.

FUnBoundP (U:id): boolean expr

Tests whether there is a definition in the function cell of U; returns T if so, NIL if not.

FLambdaLinkP (U:id): boolean expr

Tests whether U is an interpreted function; return T if so, NIL if not.

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 in the function cell of U.

MakeFLambdaLink (U:id): NIL expr

[??? definition ???]

MakeFCode (U:id, C:code-pointer): NIL expr

[??? definition ???]

GetFCodePointer (U:id): code-pointer expr

Gets the code-pointer for U.

10.1.6. Function Type Predicates

See Section 2.7 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.

10.2. 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.

10.2.1. 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]).

10.2.2. 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.

10.3. 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)

10.3.1. Funargs, Closures and Environments

[??? Not yet connected to V3 ???]

We currently have an experimental implementation of Baker's re-rooting funarg scheme [Baker 78], in which we always re-root upon binding; this permits efficient use of a GLOBAL value cell in the compiler. We are also considering implementing a restricted FUNARG or CLOSURE mechanism.

This currently uses a module (ALTBIND) to redefine the fluid binding mechanism of PSL to be functionally equivalent to an a-list binding scheme. However, it retains the principal advantage of the usual shallow binding scheme: variable lookup is extremely cheap -- just look in a value cell. Typical LISP programs currently run about 8% slower if using ALTBIND than with the initial shallow binding mechanism. It is expected that this 8% difference will go away presently. This mechanism will also probably become a standard part of PSL, rather than an add on module.

To use ALTBIND simply do "load altbind;" ["(load altbind)" in LISP]. Existing code, both interpreted and compiled, should then commence using the new binding mechanism.

The following functions are of most interest to the user:

Closure (U:form): form macro

This is similar to **Function**, but returns a function closure including environment information, similar to **Function** in LISP 1.5 and **Function*** in LISP 1.6 and MACLISP. **Eval** and **Apply** are redefined to handle closures correctly. Currently only closures of exprs are supported.

EvalInEnvironment (F:form, ENV:env-pointer): any — expr

ApplyInEnvironment (FN:function, ARGS:form-list, ENV:env-pointer): any — expr

These are like **Eval** and **Apply**, but take an extra, last argument, and environment pointer. They perform their work in this environment instead of the current one.

The following functions should be used with care:

CaptureEnvironment (): env-pointer — expr

Save the current bindings to be restored at some later point. This is best used inside a closure. **CaptureEnvironment** returns an environment pointer. This object is normally a circular list structure, and so should not be printed. The same warning applies to closures, which contain environment pointers. It is hoped that environment pointers will be made a new LISP data type soon, and will be made to print safely, relaxing this restriction.

[??? add true envpointer ???]

RestoreEnvironment (PTR:env-pointer): Undefined — expr

Restore old bindings to what they were in the captured environment, PTR.

ClearBindings (): Undefined — expr

Restore bindings to top level, i.e strip the entire stack.

For a demonstration of closures, do (in RLISP) `in "PU:altbind-tests.red";'.

[??? Give a practical example ???]

CHAPTER 11
THE INTERPRETER

11.1. Evaluator Functions Eval and Apply 11.1
11.2. Support Functions for Eval and Apply 11.3
11.3. Special Evaluator Functions, Quote, and Function 11.3
11.4. Support Functions for Macro Evaluation 11.4

11.1. Evaluator Functions Eval and Apply

The PSL evaluator uses the function cell (SYMFNC(id#)) to access the address of some code to execute for each function, as described in Chapters 10 and 22. This is either the entry address of a compiled function, or the address of a support routine that either signals 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 JUMP 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. Note **Step** in Section 16.1.2, also.

Eval (_U_:_form_): _any_ _expr_

The value of the form _U_ is computed. The following is an approximation of the real code.

```
EXPR PROCEDURE EVAL(U);
BEGIN SCALAR FN;
   IF IDP U THEN RETURN VALUECELL U;
        % ValueCell  returns the contents of Value Cell
        %  if ID is BoundP, else signals unbound error
   IF NOT PAIRP U THEN RETURN U;
        % This is a "constant" which EVAL's to itself
   IF EQCAR(CAR U,'LAMBDA) THEN
        RETURN LAMBDAEVALAPPLY(CAR U, CDR U)
   IF CODEP CAR U THEN RETURN CODEEVALAPPLY(CAR U, CDR U);
        % Lambda-expressions and code-pointers applied to
        % EVLIS'd argument list
   IF NOT IDP CAR U THEN RETURN
     CONTINUABLEERROR(1101,"Ill-formed expression in EVAL",U);
        % permit user to correct U, and reevaluate.
   FN := GETD CAR U;
   IF NULL FN THEN RETURN
       CONTINUABLEERROR(1001,"Undefined function EVAL",U);
        % user might define missing function and retry
   IF CAR FN EQ 'EXPR THEN
     RETURN
        IF CODEP CDR FN THEN CODEEVALAPPLY(CDR FN, CDR U)
         ELSE LAMBDAEVALAPPLY(CDR FN,CDR U);
   IF CAR FN EQ 'FEXPR THEN
     RETURN IDAPPLY1(CDR U,CAR U) ;
   IF CAR FN EQ 'MACRO THEN
     RETURN EVAL IDAPPLY1(U,CAR U) ;
   IF CAR FN EQ 'NEXPR THEN
     RETURN IDAPPLY1(EVLIS CDR U, CAR U) ;
END;
```

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

Apply 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. We permit macros and fexprs to be applied; 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:

```
EXPR PROCEDURE APPLY(FN, ARGS);
BEGIN SCALAR DEFN;
   IF CODEP FN THEN RETURN CODEAPPLY(FN,ARGS);
     % Spread the ARGS into the registers and
     % transfer to the entry point of the function
   IF EQCAR(FN,'LAMBDA) THEN RETURN LAMBDAAPPLY(FN,ARGS);
     % 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", FN . ARGS);
   IF NOT IDP FN THEN RETURN
      CONTINUABLEERROR(1104,
         "Ill-formed Function in APPLY", FN . ARGS);
   IF NULL(DEFN := GETD FN) THEN RETURN
      CONTINUABLEERROR(1004,
         "Undefined function in Apply",FN . ARGS);
   RETURN APPLY(CDR DEFN, ARGS);
     % Do EXPR's, FEXPR's and MACRO's alike, as EXPR's
     % Instead, could check.
END;
```

[??? Mention CodeApply, LambdaApply, LambdaEvalApply, etc. ???]

11.2. 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. **Eval** uses more efficient functions, hand-coded in LAP, to avoid the **Conses**.

EvProgN (U:form-list): any — *expr*

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

11.3. Special Evaluator Functions, Quote, and Function

Quote (U:any): any — *fexpr*

Returns **Car**(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 10.3.

11.4. 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.

```
EXPR PROCEDURE EXPAND(L,FN);
  IF NULL CDR L THEN CAR L
   ELSE LIST(FN, CAR L, EXPAND(CDR L, FN));
```

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

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

CHAPTER 12
SYSTEM GLOBAL VARIABLES, FLAGS AND OTHER "HOOKS"

12.1. Introduction 12.1
12.2. Flags 12.1
12.3. Globals 12.3
12.4. Special Put Indicators 12.4
12.5. Special Flag Indicators 12.5
12.6. Displaying Information About Globals 12.6

12.1. 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 "flags"), and XXXXX!* for other GLOBALs.

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

12.2. Flags

This section contains a list of system flags used in PSL with references to chapters where they are defined. Strictly speaking, XXXX is a flag and !*XXXX is a corresponding global variable that assumes the T/NIL value; both are loosely referred to as flags elsewhere in the manual.

The On and Off functions are used to change the values of the variables associated with flags, and also to evaluate an s-expression (if present) stored under the SIMPFG indicator on the property list of XXXXX. The s-expression should have the form of a COND list:

```
((T (action-for-ON)) (NIL (action-for-OFF)))
```

In LISP, use "(ON xxxx yyyy ... zzzz)" and "(OFF xxxx yyyy ... zzzz)". In RLISP, the On and Off functions have the syntax:

On XXXX, YYYY, ... ,ZZZZ ;

and

Off XXXX, YYYY, ... ,ZZZZ ;

On ([U:id]): None macro

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

Off ([U:id]): None macro

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

!*BACKTRACE	Chapter 15
!*BREAK	Chapter 15
!*BTR	Chapter 16
!*BTRSAVE	Chapter 16
!*COMP	Chapter 10
!*COMPRESSING	Chapter 13
!*CREFSUMMARY	Chapter 18
!*DEFN	Chapter 19
!*ECHO	Chapter 13
!*EOLINSTRINGOK	Chapter 13
!*GC	Chapter 22
!*INSTALL	Chapter 16
!*INSTALLDESTROY	Chapter 19
!*INT	Chapter 19
!*MSGP	Chapter 15
!*MODULE	Chapter 19
!*NOFRAMEFLUID	Chapter 19
!*NOLINKE	Chapter 19
!*NOTRARGS	Chapter 16
!*ORD	Chapter 19
!*PECHO	Chapter 14
!*PGWD	Chapter 19
!*PLAP	Chapter 19
!*PVAL	Chapter 14
!*PWRDS	Chapter 19
!*R2I	Chapter 19
!*RAISE	Chapter 13
!*REDEFMSG	Chapter 10
!*SAVECOM	Chapter 19

!*SAVEDEF	Chapter 19
!*SAVENAMES	Chapter 16
!*SHOWDEST	Chapter 19
!*SYSLISP	Chapter 19
!*TIME	Chapter 14
!*TRACE	Chapter 16
!*TRACEALL	Chapter 16
!*TRCOUNT	Chapter 16
!*TRUNKNOWN	Chapter 16
!*UNSAFEBINDER	Chapter 19
!*USEREGFLUID	Chapter 19
!*USERMODE	Chapter 10

12.3. Globals

The following GLOBALs are used mostly to control options and to communicate between various routines.

\CURRENTPACKAGE!*	Chapter 6
\PACKAGENAMES!*	Chapter 6
BREAKIN!*	Chapter 15
BREAKOUT!*	Chapter 15
CURRENTSCANTABLE!*	Chapter 13
DFPRINT!*	Chapter 19
EMSG!*	Chapter 15
!$EOF!$	Chapter 13
!$EOL!$	Chapter 13
!$ERROR!$	Chapter 15
ERRORFORM!*	Chapter 15
ERRORHANDLERS!*	Chapter 15
ERROUT!*	Chapter 13
GCKNT!*	Chapter 22
HELPIN!*	Chapter 13
HELPOUT!*	Chapter 13
IN!*	Chapter 13
LISPSCANTABLE!*	Chapter 13
NOLIST!*	Chapter 18
OPTIONS!*	Chapter 19
OUT!*	Chapter 13
OUTPUTBASE!*	Chapter 13
PPFPRINTER!*	Chapter 16
PROMPTSTRING!*	Chapter 13
PROPERTYPRINTER!*	Chapter 16
PUTDHOOK!*	Chapter 16
RLISPSCANTABLE!*	Chapter 13
SPECIALCLOSEFUNCTION!*	Chapter 13
SPECIALRDSACTION!*	Chapter 13
SPECIALREADFUNCTION!*	Chapter 13
SPECIALWRITEFUNCTION!*	Chapter 13

SPECIALWRSACTION!*	Chapter 13
STDIN!*	Chapter 13
STDOUT!*	Chapter 13
STUBPRINTER!*	Chapter 16
STUBREADER!*	Chapter 16
SYSLISP!*	Chapter 19
TOKTYPE!*	Chapter 13
TOPLOOPEVAL!*	Chapter 14
TOPLOOPPRINT!*	Chapter 14
TOPLOOPREAD!*	Chapter 14
TRACEMINLEVEL!*	Chapter 16
TRACEMAXLEVEL!*	Chapter 16
TRACENTRYHOOK!*	Chapter 16
TRACEXITHOOK!*	Chapter 16
TRACEXPANDHOOK!*	Chapter 16
TREXPRINTER!*	Chapter 16
TRINSTALLHOOK!*	Chapter 16
TRPRINTER!*	Chapter 16
TRSPACE!*	Chapter 16

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**.

12.4. Special Put Indicators

Some actions search on 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.

FlagInfo A string describing the purpose of the FLAG, see **ShowFlags** 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 15.

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.

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

12.5. 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 10.

See also the functions **LoadTime** and **CompileTime** in Chapter 19.

[??? Mention Parser properties ???]

12.6. Displaying Information About Globals

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

The function **ShowFlags**(flag-list); may be used to print names, current settings and purpose of some flags. Use NIL as the flag-list to get information on ALL flags of interest; **ShowFlags** in this case does a **MapObl** (Section 6.4.2) looking for 'FlagInfo 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 13
INPUT AND OUTPUT

13.1. Introduction 13.1
13.2. The Underlying Primitives for Input and Output 13.1
13.3. Opening, Closing, and Selecting Channels 13.3
13.4. Reading Functions 13.5
13.5. Scan Table Utility Functions 13.11
13.6. Procedures for Complete File Input and Output 13.11
13.7. Printing Functions 13.13
13.8. S-Expression Parser 13.17
13.8.1. Read Macros 13.18
13.9. I/O to and from Lists and Strings 13.18
13.10. Example of Simple I/O in PSL 13.20

13.1. Introduction

Most LISP programs are written with no sophisticated I/O, so this chapter may be skimmed by those with simple I/O requirements. Section 13.10 contains an example showing the use of some I/O functions. This should help the beginning PSL user get started. Sections 13.4 and 13.5 deal extensively with customizing the scanner and reader, which is of interest only to the sophisticated user.

13.2. The Underlying Primitives for Input and Output

All input and output functions are implemented in terms of operations on "channels".[1] A channel is just a small integer 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 ASCII value of the next character of the input stream as an integer. If the channel is intended for writing only or has not been opened, an error is generated.

---

[1] The range of channel numbers is from 0 to MaxChannels; MaxChannels is a system-dependent constant (currently 31).

b. A writing function, which is called with the channel as its first argument and the ASCII value of the character to be written as an *integer* as its second argument. If the channel is intended for reading only or has not been opened, an error is generated.

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, an error is generated.

The other information associated with a channel includes the current position in the output line (used by **Posn**), the maximum line length allowed (used by **LineLength** and the printing functions), the single character input backup buffer (used by the token scanner), and other system-dependent information.

Ordinarily, the user need not be aware of the existence of this mechanism. However, because of its generality, it is possible to implement operations other than just reading from and writing to files using it. In particular, the LISP functions **Explode** and **Compress** are performed by writing to a *list* and reading from a *list*, respectively (on channels 3 and 4 respectively).

Ordinarily, user interaction with the system is done by reading from the standard input channel and writing to the standard output channel. These are 0 and 1 respectively, to which the GLOBAL variables STDIN!* and STDOUT!* are bound. These channels usually refer to the user's terminal, and cannot be closed. Other files are accessed by calling the function **Open**, which returns a channel. Most functions which perform input and output come in two forms, one which takes a channel as its first argument, and one which uses the "currently selected channel". The functions **Rds** and **Wrs** are used to change the currently selected input and output channels. The GLOBAL variables IN!* and OUT!* are bound to these channels.

GLOBAL variables containing information about channels are listed below.

IN!* (*Initially*: *0*) *global*

Contains the currently selected input channel. This is changed by the function **Rds**.

OUT!* (*Initially*: *1*) *global*

Contains the currently selected output channel. This is changed by the function **Wrs**.

STDIN!* (Initially: 0) global

The standard input channel.

STDOUT!* (Initially: 1) global

The standard output channel.

ERROUT!* (Initially: 1) global

The channel used by the **ErrorPrintF**.

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!*.

13.3. Opening, Closing, and Selecting Channels

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 ACCESSTYPE is **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.

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

One can use **FileP** to find out whether a file exists.

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.

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!*).

The following GLOBALs are used by the functions in this section.

SPECIALCLOSEFUNCTION!* (Initially: NIL) *global*

SPECIALRDSACTION!* (Initially: NIL) *global*

SPECIALREADFUNCTION!* (*Initially*: *NIL*) *global*

SPECIALWRITEFUNCTION!* (*Initially*: *NIL*) *global*

SPECIALWRSACTION!* (*Initially*: *NIL*) *global*

13.4. Reading Functions

The functions described here pertain to the scanner and reader. Globals and flags used by these functions are defined at the end of the section.

ChannelReadChar (*CHANNEL*:*io-channel*): *character* *expr*

Reads one character from *CHANNEL*. All input is defined in terms of this function. If *CHANNEL* is not open or is open for writing only, an error is generated. If there is a non-zero value in the backup buffer associated with *CHANNEL*, the buffer is set to zero and the value returned. Otherwise, the reading function associated with *CHANNEL* is called with *CHANNEL* as argument, and the value it returns is returned by **ChannelReadChar**.

***** Channel not open

***** Channel open for write only

ReadChar (): *character* *expr*

Reads one character from current input.

```
EXPR PROCEDURE READCHAR();
   CHANNELREADCHAR IN!*;
```

ReadCH (): *id* *expr*

Reads a single character *id*; it calls **ReadChar**, and then converts into an *id* using **Int2ID** or **MkID**.

ChannelUnReadChar (*CHAN*:*io-channel*, *CH*:*character*): *Undefined* *expr*

Input backup function. *CH* is deposited in the backup buffer associated with *CHAN*. This function should be called only after **ChannelReadChar** is called without any intervening input operations, since it is used by the token scanner.

UnReadChar (CH:character): Undefined expr

Backup on current input channel.

```
EXPR PROCEDURE UNREADCHAR CH;
   CHANNELUNREADCHAR(IN!*, CH);
```

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.

In the last case, the value returned is the id whose print name is the same as the delimiter.

This function uses a vector bound to the FLUID variable CURRENTSCANTABLE!* to determine how various characters should be parsed. It has entries 0 ... 128 in it, one for each member of the ASCII character set (0 ... 127); the last entry is not a number, but rather an id which can be the Diphthong Indicator or the Read Macro Indicator. The following encoding 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 end of line.
13	DIPHTHONG: indicates that the character is a delimiter which may be the starting character of a diphthong. (A dihthong 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.

If the function Read is called, CURRENTSCANTABLE!* is bound to the value of LISPSCANTABLE!* or to the value of RLISPSCANTABLE!* in RLISP. Embedded system builders who wish to use their own parsers can use this function for lexical scanning by binding an appropriate scan table to CURRENTSCANTABLE!*. Utility functions for building scan tables are described in the next Section.

The following standards for scanning atoms are used.

- _Ids_ begin with a letter or any character preceded by an escape character. They may contain letters, digits and escaped characters. 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 flag 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 ???]

  [??? Permit - in ids ???]

  Examples:

  * ThisIsALongIdentifier
  * ThisIsALongIdentifierAndDifferentFromTheOther
  * !*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 !*RAISE. Examples:

* "This is a string"
* "This is a ""string"""
* ""

- Code-pointers cannot be input directly, but can be output and constructed. Currently printed as "##octal-address".

- 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 #. Thus 63 may be entered as 8#77, or 255 as 16#ff or 16#FF. The Global 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 ???]

- 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

RAtom (): {*id*, *number*, *string*} *expr*

```
EXPR PROCEDURE RATOM();
    CHANNELREADTOKEN IN!*;
```

[??? Should we bind CurrentScanTable!* for this function too ???]

!*COMPRESSING (*Initially*: *NIL*) *flag*

If !*COMPRESSING is non-NIL, ChannelReadToken does not intern *ids*.

!*EOLINSTRINGOK (*Initially*: *NIL*) *flag*

If !*EOLINSTRINGOK is non-NIL, the warning message

*** STRING CONTINUED OVER END-OF-LINE

is suppressed.

!*RAISE (*Initially*: *T*) *flag*

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.

CURRENTSCANTABLE!* (Initially:
[17 11 11 11 11 11 11 11 11 17 17 11 17 17 11 11 11 11 11 11 11
11 11 11 11 11 11 11 11 11 11 11 17 14 15 11 11 12 11 11 11 11
13 11 11 11 20 11 0 1 2 3 4 5 6 7 8 9 13 11 13 11 13 11 11 10
10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
10 10 10 10 11 16 11 11 10 11 10 10 10 10 10 10 10 10 10 10 10
10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 11 11 11 11 11
RLISPDIPTHONG]) global

CURRENTSCANTABLE!* is bound by the function **Read**.

LISPSCANTABLE!* (Initially:
[17 10 10 10 10 10 10 10 10 17 17 10 17 17 10 10 10 10 10 10 10
10 10 10 10 10 11 10 10 10 10 10 17 14 15 10 10 12 10 11 11 11
10 19 10 18 20 10 0 1 2 3 4 5 6 7 8 9 10 10 10 10 10 10 10 10
10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
10 10 10 10 11 16 11 10 10 10 10 10 10 10 10 10 10 10 10 10 10
10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
LISPDIPHTHONG]) global

RLISPSCANTABLE!* (Initially:
[17 11 11 11 11 11 11 11 11 17 17 11 17 17 11 11 11 11 11 11 11
11 11 11 11 11 11 11 11 11 11 11 17 14 15 11 11 12 11 11 11 11
13 11 11 11 20 11 0 1 2 3 4 5 6 7 8 9 13 11 13 11 13 11 11 10
10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
10 10 10 10 11 16 11 11 10 11 10 10 10 10 10 10 10 10 10 10 10
10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 11 11 11 11 11
RLISPDIPTHONG]) global

OUTPUTBASE!* (Initially: 10) global

This global 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.

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.

13.5. Scan Table Utility Functions

The following functions are provided to manage scan tables, in the READ-TABLE-UTILS module (use via LOAD READ-TABLE-UTIL):

PrintScanTable (TABLE:vector): NIL expr

Prints the entire scantable, gives the 0 ... 127 entries with the name of the character class. Also prints the indicators used for Diphthongs and Read Macros.

[??? 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 and Read macros.

[??? Change when we use Property Lists in extra slots of the Scan-Table ???]

PutDipthong (TABLE:vector, ID1: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.

13.6. Procedures for Complete File Input and Output

The following procedures are used to read complete files, by first calling Open, and then looping until end of file. Recall that file-names are strings, and therefore need 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.

DskIn ([U:string-list]): None Returned fexpr

Enters a **Read-Eval-Print** loop on the contents of each of the files in U. **DskIn** expects LISP syntax. Use the following format: (DskIn "File" "File" ...).

In ([U:string-list]): None Returned macro

Does the same as **DskIn** but expects RLISP syntax. On the DEC-20, VAX and Apollo, the function **In** causes files with extension .LSP and .SL to be read as LISP files, not as RLISP. This is a great convenience in using both LISP and RLISP files. Because of this, it is best to use the extension .RED 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 input, either end the **In** command with a "$", as

```
In "INFILE" $
```

,

or else include the statement "**Off** ECHO;" in your file.

EvIn (L:List): None Returned expr

L must be a list of strings that are filenames. **EvIn** is the same as **In** except that it computes its arguments.

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. It ignores file extensions. Use (Load F1 F2 ...) to do a (LapIn "PL:Fi.LAP") or "PL:Fi.B" to load modules such as USEFUL, EDITOR, etc.

Out (U:string): None Returned expr

Opens file for output, redirecting standard output.

Shut (U:string): None Returned *expr*

Closes the output file.

For information about fast loading of functions with **Fasl** see Chapter 19.

13.7. Printing Functions

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.

WriteChar (CH:character): character *expr*

Write single character to current output.

```
EXPR PROCEDURE WRITECHAR CH;
   CHANNELWRITECHAR(OUT!*, CH);
```

ChannelPrin1 (CHN:integer, 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**.

- Strings are printed surrounded by double quotes (").
- Delimiters inside ids are preceded by the escape character (!).
- Floats are printed as {-}nnn.nnn{E{-}nn}.
- Integers are printed as {-}nnn, unless the value of OUTPUTBASE!* is not 10, in which case they are printed as {-}r#nnn; r is the value of OutPutBase!*.
- Pairs are printed in list notation; that is, (a . (b . c)) is printed as (a b . c), and (a . (b . (c . NIL))) is printed as (a b c).
- Vectors are printed in vector notation; a vector of three

elements a, b, and c is printed as [a b c].

Other items can be printed but cannot be parsed by **Read**. The elements of dotted pairs and lists are printed by recursive calls on ChannelPrin1. Code-pointers are printed as #<Code:nnnn>; nnnn is the octal machine address of the entry point of the code vector. Anything else is printed as #<Unknown:nnnn>; 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.

Prin1 (ITM:any): ITM:any — expr

```
EXPR PROCEDURE PRIN1 ITM;
   CHANNELPRIN1(OUT!*, ITM);
```

ErrPrin (U:any): None Returned — expr

Prin1 with special quotes to highlight U.

ChannelPrin2 (CHN: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.

Prin2 (ITM:any): ITM:any — expr

```
EXPR PROCEDURE PRIN2 ITM;
   CHANNELPRIN2(OUT!*, ITM);
```

Print (U:any): U:any — expr

Display U using **Prin1** and terminate line.

PrintF (FORMAT:string, [ARGS:any]): NIL — expr

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

- For SYSLISP arguments, 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

- 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
%e	eval the next argument for side-effect -- most useful if the thing evaled does some printing

Not using arguments:

%n	print end-of-line character

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.

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.

TerPri (): NIL expr

Terminate current OUTPUT line, and reset the POSN counter to 0.

Eject (): NIL expr

Skip to top of next output page.

Posn (): integer expr

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

LPosn (): integer expr

[not implemented yet]Returns number of lines output on this page (i.e. LPosn counter since last Eject).

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

Set maximum output line length if a positive integer, returning previous value. If NIL just return previous value. Controls the insertion of automatic Terpri's.

RPrint (U:form): NIL expr

Print in RLISP format. Autoloading.

PrettyPrint (U:form): U expr

Prettyprints U. Autoloading.

Prin2L (L:any): L expr

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

Spaces (N:integer): NIL expr

Prin2 N spaces.

Prin2T (X:any): any expr

Prin2 and TerPri.

Tab (N:integer): NIL expr

Move to position N, emitting spaces as needed. TerPri() if past column N.

13.8. S-Expression Parser

ChannelRead (CHN:io-channel): any expr

Returns the next expression from input channel CHN. Valid input forms are: vector notation, pair notation, list notation, numbers, code-pointers, strings, and identifiers with escape characters. Identifiers are interned (see the Intern function in Chapter 6), unless the FLUID variable !*COMPRESSING is non-NIL. ChannelRead returns the value of !$EOF!$ when the end of the currently selected input file is reached. ChannelRead has a

```
CATCH('!$READ!$,'(ChannelReadWithHooks Channel));
```

call in its body, so that a Read can be aborted by a Throw with tag !$READ!$. This is in fact used to exit levels of Read when an end of file (!$EOF!$) is detected.

ChannelRead uses the ChannelReadToken function, with tokens scanned according to the current scan table. In addition, Read macros are applied, if appropriate. The default reader uses the Read macro mechanism to do S-expression parsing:

(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 (Quote x).

!$EOF!$ Does a Throw(!$READ!$, !$EOF!$) to rapidly exit a Read.

'!{ In some Read tables, { acts as a SuperParen, matched by '!}. } closes all pending (, upto a matching {.

Read (): any **expr**

Parse S-expression from current input. This does a ChannelRead(IN!*) after binding the fluid variable CURRENTSCANTABLE!* to LISPSCANTABLE!*. Thus **Read** always uses the LISP scan table. The user can define similar read functions for his own use with other scan tables.

13.8.1. 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 calling **PutReadMacro**. A ReadMacro 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:

```
PROCEDURE 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.

13.9. I/O to and from Lists and Strings

Digit (U:any): boolean **expr**

Returns T if U is a digit, otherwise NIL.

Liter (U:any): boolean expr

Returns T if U is a character of the alphabet, NIL otherwise.

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 4.1.1. Identifiers are not interned on the OBLIST. 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.

FlatSize (U:any): integer expr

Character length of Prin1 S-expression.

FlatSize2 (U:any): integer expr

Prin2 version of flatsize.

BldMsg (FORMAT:string, [ARGS:any]): string expr

PrintF to string. BldMsg returns a string stating that the string could not be constructed if overflow occurs.

13.10. Example of Simple I/O in PSL

In the following example a list of S-expressions is read, one expression at a time, from a file STUFF.IN and is written to a file STUFF.OUT. The file EXAMPIO.RED contains the RLISP function TRYIO that makes this transfer. The files EXAMPIO and STUFF.IN were prepared using EMACS.

Following is the contents of STUFF.IN.

```
(r e d)
(a b c)
(1 2 3 4)
"ho ho ho"
6.78
5000
xyz
```

The following shows the execution of the function TRYIO.

```
PSL:rlisp
[Keeping rlisp]
PSL 3.0 Rlisp, 8-Jul-82
[1] In "EXAMPIO.RED";
Procedure Tryio (FIL1,FIL2);
Begin Scalar OLDIN, OLDOUT, EXP;
   OLDIN:=Rds Open(FIL1,'input);
   OLDOUT:=Wrs Open(FIL2,'output);
   While (EXP:=Read()) NEQ !$EOF!$
      Do Print EXP;
   Close Rds OLDIN;
   Close Wrs OLDOUT;
End;TRYIO

NIL
[2] Tryio ("STUFF.IN","STUFF.OUT");
NIL
```

```
[3] quit;
```

The output file STUFF.OUT contains the following.

```
(R E D)
(A B C)
(1 2 3 4)
"ho ho ho"
6.78
5000
XYZ
```

CHAPTER 14
USER INTERFACE

14.1. Introduction 14.1
14.2. Stopping PSL and Saving a New Executable Core Image 14.1
14.3. Changing the Default Top Level Function 14.2
14.4. The General Purpose Top Loop Function 14.2
14.5. The HELP Mechanism 14.5
14.6. The Break Loop 14.6
14.7. Terminal Interaction Commands in RLISP 14.6

14.1. Introduction

In this Section 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.

14.2. Stopping PSL and Saving a New Executable Core Image

The normal way to stop 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.

After either of these actions, PSL may be re-entered by typing START or CONTINUE to the EXEC on the DEC-20. After exiting, the core image may also be saved using the Tops-20 monitor command "SAVE filename". On the VAX, **Quit** causes a stop signal to be sent, so that PSL may be continued from the shell.

A better way to exit and save the core image is to call the function **SaveSystem**.

SaveSystem (MSG:string): Undefined expr

This records the welcome message (after attaching a date) in the global variable LISPBANNNER!* 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. In RLISP,

it sets USERMODE!* to T.

If RLISP has been loaded, **SaveSystem** will have been redefined to save the message in global variable DATE!*, and redefine **Main** to call **RlispMain**, which uses DATE!* in **Begin1**.

DumpLisp (): 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 name of global DUMPFILENAME!* (initially "PSL-SAVE.EXE" on the DEC-20 and "a.out" on the VAX). If desired, the user can then **Quit** and rename the file name, or change DUMPFILENAME!* before calling **SaveSystem**.

Reset (): Undefined expr

Return to top level of LISP. Equivalent to <Ctrl-C> and Start on DEC-20.

14.3. 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 **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**.

14.4. 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 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.

!*PECHO (Initially: NIL) flag

Causes parsed form read in top-loop **StandardLisp** to be printed, if T.

!*PVAL (Initially: T) flag

Causes values computed in top-loop **StandardLisp** to be printed, if T.

!*TIME (Initially: NIL) flag

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 macro

Reevaluate N'th input.

Ans (N:integer): any expr

Return N'th result.

TopLoop has been used to define the following **StandardLisp** and RLISP top loops.

StandardLisp (): NIL expr

Interpreter LISP syntax top loop, defined as:

```
LISP PROCEDURE STANDARDLISP();  %. Lisp top loop
BEGIN SCALAR CURRENTREADMACROINDICATOR!*,
        CURRENTSCANTABLE!*;
    CURRENTREADMACROINDICATOR!* :='LISPREADMACRO;
    CURRENTSCANTABLE!* := LISPSCANTABLE!*;
    TOPLOOP('READ, 'PRINT, 'EVAL, "LISP",
                "PORTABLE STANDARD LISP");
end;
```

Note that the scan tables are modified.

RLisp (): NIL expr

Alternative interpreter RLISP syntax top loop, defined as:

[??? xread described in RLISP Section ???]

```
LISP PROCEDURE RLISP();      % RLisp top loop
    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.

14.5. The HELP Mechanism

PSL provides a general purpose Help mechanism, that is called in the TopLoop by invoking Help(); sometimes a ? may be used, as for example in the break loop.

Help ([TOPICS:id]): NIL fexpr

If no arguments are given, a message describing Help itself and known topics is printed. Otherwise, each of the id 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 TopLoop fluids, and finally searches the current Id-Hash-Table for loaded modules.

HELPIN!* (Initially: NIL) global

The channel used for input by the Help mechanism.

HELPOUT!* (<u>Initially</u>: <u>NIL</u>) <u>global</u>

The channel used for output by the Help mechanism.

14.6. The Break Loop

The Break Loop is described in detail in Chapter 15. For information, look there.

14.7. Terminal Interaction Commands in RLISP

Two commands are available in RLISP for use in interactive computing. 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**.

YesP (<u>MESSAGE</u>:<u>string</u>): <u>boolean</u> <u>expr</u>

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.

CHAPTER 15
ERROR HANDLING

15.1. Introduction 15.1
15.2. The Basic Error Functions 15.1
15.3. Break Loop 15.3
15.4. Interrupt Keys 15.6
15.5. Details on the Break Loop 15.6
15.6. Some Convenient Error Calls 15.7
15.7. Special Purpose Error Handlers 15.8

15.1. 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" to a currently active error handler (called **ErrorSet**) 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, a call on **Throw** with a tag of!$ERROR!$ is done, and fluids are unbound.

[??? What about errors signalled to the Interrupt Handler ???]

15.2. The Basic Error Functions

!*BACKTRACE (_Initially_: _T_) _flag_

Set in **ErrorSet**. Controls whether an unwind backtrace is requested.

!*MSGP (_Initially_: _T_) _flag_

Set in **ErrorSet**. Controls error message printing during call on error.

EMSG!* (Initially: NIL) global

Contains the message generated by the last error call.

ErrorSet (U:any, !*MSGP:boolean, !*BACKTRACE:boolean): any expr

If an uncorrected error occurs during the evaluation of U, the value of NUMBER from the associated error call is returned as the value of **ErrorSet**. Note that **ErrorSet** is an expr, so U gets evaluated twice, once as the parameter is passed and once inside **ErrorSet**. [Actually, **ErrorSet** executes a **Catch** with tag !$ERROR!$, and so intercepts any **Throw** with this tag.] In addition, if the value of !*MSGP is non-NIL, the message from the error call is displayed upon both the standard output device and the currently selected output device unless the standard output device is not open. The message appears prefixed with 5 asterisks. The message list is displayed without top level parentheses. The message from the error call is available in the GLOBAL variable EMSG!*. 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, since the cause of the error is the same and the information to the user is the same. The error number is then used to indicate which function actually detected the error.

[??? Describe Error # ranges here, or have in a file on machine ???]

If no error occurs during the evaluation of U, the value of (**List** (**Eval** U)) is returned.

If an error has been signalled and the value of !*BACKTRACE is non-NIL, a traceback sequence is initiated on the selected output device. The traceback displays information such as unbindings of FLUID variables, argument lists and so on in an implementation--dependent format.

Error (NUMBER:integer, MESSAGE:any): None Returned expr

MESSAGE is placed in the GLOBAL variable EMSG!* and the error number becomes the value of the surrounding **ErrorSet** (if any intervening **Break** loop is exited). FLUID variables and LOCAL bindings are unbound to return to the environment of the **ErrorSet**. GLOBAL variables are not affected by the process. **Error** actually signals a non-continuable error to the **Break** loop,

and it subsequently does a throw with tag !$ERROR!$.

ContinuableError (NUMBER:integer, MESSAGE:any, FORM:form): any *expr*

MESSAGE is placed in the GLOBAL variable EMSG!* and the error number becomes the value of the surrounding **ErrorSet** if the intervening **Break** loop is "QUIT" rather than "Continued" or "Retried". FLUID variables and LOCAL bindings are unbound to return to the environment of the **ErrorSet**. GLOBAL variables are not affected by the process. **Error** actually signals a continuable error to the **Break** loop, and it subsequently does a throw with tag !$ERROR!$.

The FORM is stored in the GLOBAL variable ERRORFORM!*, for examination, editing or possible reevaluation after defining missing functions, etc. Setting up the ERRORFORM!* can get a bit tricky, often involving **MkQuote**ing 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. The ReEvalForm is something like Foo(X, Y) which becomes list('Foo, MkQuote X, MkQuote Y) to be passed to the function **ContinuableError**.

[??? Give example, e.g. Divide in pi:easy-sl.red ???]

15.3. 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 FLUIDs, or define missing functions. He can then dismiss the error call to the normal error handling mechanism (the **ErrorSet** above) or (in some situations) continue the computation. By setting the flag !*BREAK to NIL, all **Break** loops can be suppressed.

!*BREAK (Initially: T) *flag*

Controls whether the **Break** package is called before unwinding the stack on error.

The prompt "Break>" indicates that PSL has entered a **Break** loop. A message of the form "Continuation requires a value for ..." 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, I, and C. These 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 "Continuation requires a value for ..." message. The Retry command (below) uses the corrected version of ERRORFORM!*. The currently available editors are described in Chapter 17.

M Show the modified ERRORFORM!*.

R Retry. This tries to evaluate the offending expression again, 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.

C This causes the expression last printed by the **Break** loop to be returned as the value of the offending expression. 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.

Q Quit. This exits the **Break** loop by throwing to the closest surrounding **ErrorSet**.

T Trace. This prints a backtrace of functions call on the stack.

I Interpreter Trace. This prints a backtrace of only interpreted functions call on the stack.

InterpBackTrace (): Undefined expr

Prints a backtrace of ONLY interpreted functions. Called by "I" in break loop.

[??? Put this with backtrace? ???]

An attempt to continue a non-continuable error with R or C prints a message and behaves as Q.

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> foo(1)+foo(2);
***** `FOO' is an undefined function {1001}
***** Continuation requires a value for `(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}
***** Continuation requires a value for `(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> procedure fee(x);x+1;
FEE
3> foo(1)+foo(2);             % now the bad expression
***** `FOO' is an undefined function {1001}
***** Continuation requires a value for `(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}
***** Continuation requires a value for `(FOO 2)'
Break loop
1 lisp break> (de foo(x) (plus2 x 1))  % define foo
FOO
2 lisp break> r                         % and retry
5
```

15.4. Interrupt Keys

Need to "LOAD 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.

15.5. Details on the Break Loop

If the FLAG !*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 a function of no arguments, and is applied instead of **Eval**.

15.6. Some Convenient Error Calls

The following functions may be useful in user packages:

```
FatalError S;
  <<ErrorPrintF("***** Fatal error: %s", S);
    while T do Quit; >>;

 RangeError(Object, Index, Fn);
    StdError BldMsg("Index %r out of range for %p in %p",
                          Index,          Object, Fn);

 StdError Message;      %. Error without number
    Error(99, Message);

 TypeError(Offender, Fn, Typ);
   StdError BldMsg("An attempt was made to do %p on %r, which is not %w",
                                          Fn, Offender,      Typ);
 UsageTypeError(Offender, Fn, Typ, Usage);
   StdError
    BldMsg("An attempt was made to use %r as %w in %p, where %w is needed",
                                    Offender, Usage, Fn,    Typ);
 IndexError(Offender, Fn);
    UsageTypeError(Offender, Fn, "an integer", "an index");

 NonPairError(Offender, Fn);
    TypeError(Offender, Fn, "a pair");

 NonIDError(Offender, Fn);
    TypeError(Offender, Fn, "an identifier");
```

```
NonNumberError(Offender, Fn);
   TypeError(Offender, Fn, "a number");

NonIntegerError(Offender, Fn);
   TypeError(Offender, Fn, "an integer");

NonPositiveIntegerError(Offender, Fn);
   TypeError(Offender, Fn, "a non-negative integer");

NonCharacterError(Offender, Fn);
   TypeError(Offender, Fn, "a character");

NonStringError(Offender, Fn);
   TypeError(Offender, Fn, "a string");

NonVectorError(Offender, Fn);
   TypeError(Offender, Fn, "a vector");

NonSequenceError(Offend
```

CHAPTER 16
DEBUGGING TOOLS

16.1. Introduction 16.1
16.1.1. Mini-Trace Facility 16.2
16.1.2. Step 16.3
16.1.3. Brief Summary of Full DEBUG Package 16.4
16.1.4. Use 16.5
16.1.5. Functions Which Depend on Redefining User Functions 16.5
16.1.6. Special Considerations for Compiled Functions 16.5
16.1.7. A Few Known Deficiencies 16.6
16.2. Tracing Function Execution 16.6
16.2.1. Saving Trace Output 16.7
16.2.2. Making Tracing More Selective 16.9
16.2.3. Turning Off Tracing 16.10
16.2.4. Automatic Tracing of Newly Defined Functions 16.11
16.3. Automatic BREAK Around Functions 16.11
16.4. A Heavy Handed Backtrace Facility 16.11
16.5. Embedded Functions 16.12
16.6. Counting Function Invocations 16.13
16.7. Stubs 16.13
16.8. Functions for Printing Useful Information 16.14
16.9. Printing Circular and Shared Structures 16.14
16.10. Library of Useful Functions 16.15
16.11. Internals and Customization 16.15
16.11.1. User Hooks 16.15
16.11.2. Functions Used for Printing/Reading 16.16
16.11.3. Flags 16.17
16.12. Example 16.18

16.1. Introduction

PSL offers a small group of debugging functions in a mini-trace package described in Section 16.1.1; in addition, there is a separate debugging package which is the subject of the bulk of this Chapter.

The PSL debugging package contains a selection of functions that can be used to aid program development and to investigate faulty programs.[1]

---

[1] Much of this Chapter was adapted from a paper by Norman and Morrison.

It contains the following facilities.

- 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 16.2).

- A backtrace facility. This allows one to see which of a set of selected functions were active as an error occurred (see Section 16.4).

- Embedded functions make it possible to do everything that the trace package can do, and much more besides (see Section 16.5).

- Some primitive statistics gathering (see Section 16.6).

- Generation of simple stubs. If invoked, procedures defined as stubs simply print their argument and read a value to return (see Section 16.7).

- Some functions for printing useful information, such as property lists, in an intelligible format (see Section 16.8).

- **PrintX** is a function that can print circular and re-entrant lists, 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 16.9).

- A set of functions **!:Car**, ..., **!:Cddddr**, **!:RplacA**, **!:RplacD** and **!:RplacW** that behave exactly as the corresponding functions with the !: removed, except that they explicitly check that they are not used improperly on atomic arguments (see Section 16.10).

- A collection of utility functions, not specifically intended for examining or debugging code, but often useful (see Section 16.10).

16.1.1. Mini-Trace Facility

A small trace package is provided in the small core PSL and RLISP. This provides a command **Tr** for tracing LISP function calls, as does the full Debug package. This command and the associated command **UnTr** are used in the form:

```
   Tr <function name>, <function name>,..., <function name>;
 or
   Tr( <function name>, <function name>,..., <function name>);
```

from RLISP, and

```
   (Tr <function name> <function name> ... <function name>)
```

from LISP.

Tr ([FNAME:id]): Undefined fexpr

UnTr ([FNAME:id]): Undefined fexpr

Mini-Trace also provides a crude BREAKFN capability, that causes the modified functions to call **Break** before and after execution of the function body:

```
   Br <function name>, <function name>,..., <function name>;
 or
   Br( <function name>, <function name>,..., <function name>);
```

from RLISP, and

```
   (Br <function name> <function name> ... <function name>)
```

from LISP.

Br ([FNAME:id]): Undefined fexpr

UnBr ([FNAME:id]): Undefined fexpr

Do HELP(TRACE); for more information.

16.1.2. Step

Step (F:form): any expr

> **Step** is a loadable option (LOAD STEP;). Evaluates form, 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.

16.1.3. Brief Summary of Full DEBUG Package

Useful diagnostic information can often be obtained by observing the values which variables acquire during assignments in particular functions. To do this in PSL one uses the command **Trst**. There are some restrictions on the function names which appear in the arguments of this function, however. First, the functions must obviously be in the system in symbolic form; compiled functions no longer contain information on the assignment variable names. Secondly, the functions must have a compound statement at their top level.

This particular trace may be turned off by the command **UnTrst**.

Both **Trst** and **Untrst** are used in the same way as **Tr**.

!*NOTRARGS (Initially: NIL) flag

Control arg-trace. T suppresses printing of the arguments of traced functions.

16.1.4. Use

To use do LOAD Debug;. The saved PSL may have this already loaded by default.

16.1.5. Functions Which Depend on Redefining User Functions

A number of facilities in Debug depend on redefining user functions, so that they may log or print behavior if called. The Debug package tries to redefine user functions once and for all, and then keep specific information about what is required at run time in a table. This allows considerable flexibility, and is used for a number of different facilities, including trace/traceset in Section 16.2, a backtrace facility in Section 16.4, some statistics gathering in Section 16.6 and EMB functions in Section 16.5.

Some, like trace and EMB, only take effect if further action is requested on specific user functions. Others, like backtrace and statistics, are of a more global nature. Once one of these global facilities is enabled it applies to all functions which have been made "known" to Debug. To undo this, use **Restr** in Section 16.2.3.

16.1.6. Special Considerations for Compiled Functions

All functions in Debug which depend on redefining user functions must make some assumptions about the number of arguments. The Debug package is able to find the correct names for the arguments of interpreted functions. If Debug can not find out for itself how many arguments a function has, it interactively asks for assistance. In reply to the question

```
HOW MANY ARGUMENTS DOES xxxx HAVE?
```

valid replies are:

? ask for assistance

UNKNOWN give up

<number> specify the number of arguments

(name ...)
give the names of arguments.

[??? How about a push to a new command level so one can look around and then come back to answer the question ???]

If you give an incorrect answer to the question, the system may misbehave in an arbitrary manner. There can be problems if the answer UNKNOWN is given and subsequently functions get redefined or recompiled - if at all possible find out how many arguments are taken by the function that you wish to trace.

It is possible to suppress the argument number query with

```
ON TRUNKNOWN
```

This is equivalent to always answering "UNKNOWN".

[??? Add feature to enable compiled code to store argument names, etc. in LAP files ???]

16.1.7. 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 - so report trouble to a system expert.

- 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.

16.2. Tracing Function Execution

Tr ([FNAME:id]): Undefined expr

Trst ([FNAME:id]): Undefined expr

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,...;
```

If the specified functions are defined (as expr, fexpr or macro), this REDUCE statement modifies the function definition to include print statements. The following example shows the style of output produced by this sort of tracing:

The input...

```
SYMBOLIC PROCEDURE 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 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, and the use of **UnTr** automatically does an **UnTrst** if necessary in Section 16.2.3, so that it is not possible to have a function subject to **Trst** but not **Tr**.

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.

16.2.1. Saving Trace Output

NewTrBuff (N:integer): Undefined expr

TrOut ([FNAME:id]): Undefined expr

StdTrace (): Undefined expr

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 16.2.2). 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;
```

To set the number of entries retained to n use

```
NEWTRBUFF(n);
```

It is initially set to 5.

Turning off the TRACE flag

```
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.

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**.

16.2.2. Making Tracing More Selective

TraceCount (N:integer): Undefined *expr*

TrIn ([FNAME:id]): 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.

Thus 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 direct of **TraceCount** is **TrIn**. 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 sub-part of your entire program. **UnTr** undoes the effect of **TrIn** in Section 16.2.3.

TRACEMINLEVEL!* (Initially: 0) *global*

In DEBUG package.

TRACEMAXLEVEL!* (Initially: 1000) *global*

In DEBUG package.

The global variables TRACEMINLEVEL!* and TRACEMAXLEVEL!* (which 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

```
TRACEMAXLEVEL!* := 1;
```

16.2.3. Turning Off Tracing

UnTr ([FNAME:id]): Undefined expr

Restr ([FNAME:id]): Undefined expr

UnTrst ([FNAME:id]): Undefined expr

If a particular function no longer needs tracing, do

```
UNTR functionname;
```

or

```
UNTR name1,name2...;
```

This merely suppresses géneration of trace output. Other information, such as invocation counts, backtrace information, and the number of arguments is retained. Thus **UnTr** followed later by **Tr** does not have to enquire about the number of arguments.

To completely destroy information about a function use

```
RESTR name1,name2...;
```

This returns the function to it's original state.

To suppress traceset output without suppressing normal trace output use

```
UNTRST name1,name2...;
```

UnTring a **Trst**ed function also **UnTrst**'s it.

TrIn in Section 16.2.2 is undone by **UnTr** (but not by **UnTrst**).

16.2.4. Automatic Tracing of Newly Defined Functions

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.

To enable those facilities (such as **Btr** in Section 16.4 and **TrCount** in Section 16.6) which require redefinition, but without tracing, use

```
ON INSTALL;
```

Thus, a common scenario might look like

```
ON INSTALL;
IN MYFNS.RED$
OFF INSTALL;
```

which would enable the backtrace and statistics routines to work with all the functions defined in MYFNS.RED.

!*INSTALL (_Initially_: _NIL_) _flag_

In DEBUG package. Causes DEBUG to know about all functions defined with PutD.

!*TRACEALL (_Initially_: _NIL_) _flag_

In DEBUG package. Causes all functions defined with PutD to be traced.

16.3. Automatic BREAK Around Functions

[??? Install a feature BR and UNBR to wrap a break around functions. See mini-trace ???]

16.4. A Heavy Handed Backtrace Facility

Btr ([FNAME:id]): Undefined expr

ResBtr ([FNAME:id]): Undefined expr

```
BTR f1,f2,...;
```

arranges that a stack of functions entered but not left is kept - 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.

To print this information call **Btr** without any arguments

```
BTR;
```

Calling **Btr** on new functions resets the stack. This may also be done by explicitly calling **ResBtr**

```
RESBTR;
```

The disposition of information about functions which failed within an **ErrorSet** is controlled by the !*BTRSAVE. ON BTRSAVE causes them to be saved separately, and printed when the stack is printed; OFF BTRSAVE causes them to be thrown away.

OFF BTR suppresses saving of any **Btr** information. Note that any traced function has its invocations pushed and popped by the **Btr** mechanism.

16.5. 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;
```

16.6. Counting Function Invocations

If the flag 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. To make Debug aware of a function use

```
TRCNT name1,name2,...;
```

See also Section 16.2.4.

TrCnt ([FNAME:id]): Undefined expr

16.7. Stubs

Stub (FN:id): expr

FStub (FN:id): expr

The statement

```
STUB FOO(U,V);
```

defines an **expr**, **Foo**, of two arguments. If executed such a stub prints its arguments and reads a value to return. **FStub** is used to define **fexprs**. This is often useful in developing programs in a top down fashion.

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.

16.8. Functions for Printing Useful Information

PList ([X:id]): *expr*

Ppf ([FNAME:id]): *expr*

```
PLIST id1,id2,...;
```

prints the property lists of the specified ids.

```
PPF fn1,fn2,...;
```

prints the definitions and other useful information about the specified functions.

16.9. Printing Circular and Shared Structures

PrintX (A:any): NIL *expr*

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**. WARNING: This function does NOT work for circular vectors! See Section 18.10 for functions dealing with circular vectors. 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:

```
A:=NIL . NIL; % make a node
RPLACA(A,A); RPLACD(A,A); % point it at itself
```

If **PrintX** is called on the list A it discovers that the node is referred to repeatedly, and invents the label %L1 for it. The structure is then printed as

```
%L1: (%L1 . %L1)
```

%L1: sets the label, and the other instances of %L1 refer back to it. Labeled sublists can appear anywhere within the list being printed. Thus the list B := 'X . A; could be printed as

```
(X . %L1: (%L1 . %L1))
```

This use of dotted pair representation is often clumsy, and so it gets contracted to

```
(X %L1, %L1 . %L1)
```

A label set with a comma (rather than a colon) is a label for part of a list, not for the sublist.

16.10. Library of Useful Functions

Debug contains a library of utility functions which may be useful to those debugging code. The collection is as yet very small. Suggestions for further functions to be in corporated are definitely solicited.

16.11. 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.

16.11.1. User Hooks

These are all global variables whose value is 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.

16.11.2. 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.

16.11.3. Flags

These are all flags which can be set with the Reduce/Rlisp ON/OFF statements. Their initial setting is given in square brackets. Many have been described above, but are collected here for reference.

!*BTR (Initially: T) flag

enables backtracing of functions which the Debug package has been told about.

!*BTRSAVE (Initially: T) flag

causes backtrace information leading up to an error within an errorset to be saved.

!*SAVENAMES (Initially: NIL) flag

!*TRACE (Initially: T) flag

enables runtime printing of trace information for functions which have been traced.

!*TRUNKNOWN (Initially: NIL) flag

instead of querying the user for the number of arguments to a compiled EXPR, just assumes the user says "UNKNOWN".

!*TRCOUNT (Initially: T) flag

enables counting invocations of functions known to Debug. Note that ON TRCOUNT resets the count, and OFF TRCOUNT prints a simple histogram of the available counts.

16.12. Example

This contrived example demonstrates many of the available features. It is a transcript of an actual REDUCE session.

```
REDUCE 2 (Dec-1-80) ...
FOR HELP, TYPE HELP<ESCAPE>

1:

2: LOAD DEBUG;

3: SYMBOLIC PROCEDURE FOO N;
3: BEGIN SCALAR A;
3:   IF REMAINDER(N,2) NEQ 0 AND N < 0 THEN
3:     A := !:CAR N; % Should err out if N is a number
3:   IF N = 0 THEN
3:     RETURN 'BOTTOM;
3:   N := N-2;
3:   A := BAR N;
3:   N := N-2;
3:   RETURN LIST(A,BAR N,A)
3: END FOO;

FOO

4: SYMBOLIC PROCEDURE FOOBAR N;
4: << FOO N; NIL>>;

FOOBAR

5: SYMBOLIC OPERATOR FOOBAR;

NIL

6: TR FOO,FOOBAR;

(FOO FOOBAR)

7: PPF FOOBAR,FOO;


EXPR procedure FOOBAR(N) [Traced;Invoked 0 times;Flagged: OPFN]:
```

```
<<FOO N; NIL>>;

EXPR procedure FOO(N) [Traced;Invoked 0 times]:
BEGIN SCALAR A;
   IF NOT REMAINDER(N,2)=0 AND N<0 THEN A := !:CAR N;
   IF N=0 THEN RETURN 'BOTTOM;
   N := N - 2;
   A := BAR N;
   N := N - 2;
   RETURN LIST(A,BAR N,A)
 END;

FOOBAR(FOO)

8: ON COMP;

9: SYMBOLIC PROCEDURE BAR N;
9: IF REMAINDER(N,2)=0 THEN FOO(2*(N/4)) ELSE FOO(2*(N/4)-1);

*** BAR 164896 BASE 20 WORDS 63946 LEFT

BAR

10: OFF COMP;

11: 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
```

```
0

12: % Notice how in the above PRINTX printed the return values
12: % to show shared structure
12: TRST FOO;

(FOO)

13: 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

0

14: TR BAR;

*** How many arguments does BAR take ?  1

(BAR)
```

```
15: FOOBAR 8;
FOOBAR being entered
   N:   8
  FOO being entered
     N: 8
  N := 6
    BAR being entered
       A1:      6
      FOO (level 2) being entered
         N:     2
      N := 0
        BAR (level 2) being entered
           A1:  0
          FOO (level 3) being entered
             N: 0
          FOO (level 3) = BOTTOM
        BAR (level 2) = BOTTOM
      A := BOTTOM
      N := -2
        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)
  A := (BOTTOM BOTTOM BOTTOM)
  N := 4
    BAR being entered
       A1:      4
      FOO (level 2) being entered
         N:     2
      N := 0
        BAR (level 2) being entered
           A1:  0
          FOO (level 3) being entered
             N: 0
          FOO (level 3) = BOTTOM
        BAR (level 2) = BOTTOM
      A := BOTTOM
      N := -2
        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

0

16: OFF TRACE;

17: FOOBAR 8;

0

18: 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 ***

19: FOOBAR 13;

***** -1 illegal CAR

20: TR;
*** Start of saved trace information ***
    BAR being entered
       A1:      11
      FOO (level 2) being entered
         N:     3
      N := 1
        BAR (level 2) being entered
           A1:  1
          FOO (level 3) being entered
             N: -1
*** End of saved trace information ***

21: 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 ***

22: SYMBOLIC EMB PROCEDURE FOO N;
22: IF N < 0 THEN <<
22:   LPRIM "FOO would have failed";
22:   NIL >>
22: ELSE
22:   FOO N;

FOO

23: RESBTR;

24: FOOBAR 13;

*** FOO WOULD HAVE FAILED

*** FOO WOULD HAVE FAILED

*** FOO WOULD HAVE FAILED

*** FOO WOULD HAVE FAILED

0

25: TR;
*** Start of saved trace information ***
        BAR (level 2) = NIL
      FOO (level 2) = (NIL NIL NIL)
    BAR = (NIL NIL NIL)
  FOO = (%L1: (NIL NIL NIL) (NIL NIL NIL) %L1)
FOOBAR = NIL
*** End of saved trace information ***

26: BTR;
*** No traced functions were left abnormally ***

27: UNEMBED FOO;

(FOO)

28: FOOBAR 13;

***** -1 illegal CAR

29: STUB FOO N;

*** FOO REDEFINED

30: FOOBAR 13;
 Stub FOO called
```

```
N: 13
Return? :
30: BAR(N-2);
 Stub FOO called

N: 3
Return? :
30: BAR(N-2);
 Stub FOO called

N: -1
Return? :
30: 'ERROR;

0

31: 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 ***

32: OFF TRCOUNT;


FOOBAR(8)           ****************
BAR(24)             ********************************************


33: QUIT;
```

CHAPTER 17
EDITORS

17.1. A Mini-Structure Editor 17.1
17.2. The EMODE Screen Editor 17.3
17.2.1. Windows and Buffers in Emode 17.5
17.3. Introduction to the Full Structure Editor 17.6
17.4. User Entry to Editor 17.6
17.5. Editor Command Reference 17.8

17.1. A Mini-Structure Editor

PSL and RLISP provide a fairly simple structure editor, essentially a subset of the full editor below. This 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!*.

17.2. 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.

```
PSL:RLISP
[1] LOAD EMODE;
[2] EMODE();
```

<Ctrl-X Ctrl-Z>
"quits" to the EXEC (you can continue or start again).

<Ctrl-Z Ctrl-Z>
goes back into "normal" RLISP 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	

Key	Function	Description
<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

17.2.1. Windows and Buffers in Emode

- Global Variable `WINDOWNAMES' is list of (windows.info)

- CreateWindow(Wname,Bname,Coord(Left,Top),Coord(Right,Bottom))

[Left,Right:1..18, Top,Bottom:1..70]

- SelectWindow(Wname); DeselectWindow(Wname); KillWindow(Wname);

17.3. 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.

17.4. User Entry to Editor

This Section describes normal user entry to the editor (EditF, EditP and EditV) 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)
```

---

[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 INTER LISP editor.

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 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.

17.5. Editor Command Reference

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 paren is inserted before element N1 of the current expression. A right paren 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, COMS:forms): any expr

This function initiates editing of the function whose name is FN. The argument COMS is a possibly null sequence of edit commands which is executed before calling for input from the terminal.

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 paren); i.e. Left Paren Insert. The left paren is inserted before the Nth element of the current expression and the right paren 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 paren, of course) from the Nth element of the current expression; i.e. Left Paren 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 paren 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 Paren 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 Paren 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 fsubr, 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 fsubr 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 **edit-command**

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 18
NEW UTILITIES

18.1. Introduction 18.1
18.2. RCREF - Cross Reference Generator for PSL Files 18.1
18.2.1. Restrictions 18.2
18.2.2. Usage 18.3
18.2.3. Options 18.3
18.3. Picture RLISP 18.3
18.3.1. Running PictureRLISP on HP2648A and on TEKTRONIX 4006-1 Terminals 18.10
18.4. Tools for Defining Macros 18.10
18.4.1. DefMacro 18.11
18.4.2. BackQuote 18.11
18.4.3. Sharp-Sign Macros 18.12
18.4.4. MacroExpand 18.13
18.4.5. DefLambda 18.13
18.5. Simulating a Stack 18.13
18.6. DefStruct 18.14
18.6.1. Options 18.16
18.6.2. Slot Options 18.17
18.6.3. A Simple Example 18.18
18.7. DefConst 18.21
18.8. Find 18.21
18.9. Hashing Cons 18.22
18.10. Graph-to-Tree 18.23
18.11. Inspect Utility 18.24
18.12. Trigonometric and Other Mathematical Functions 18.25

18.1. Introduction

This Chapter describes an assortment of utility packages that are too new either to be included elsewhere or to have a Chapter of their own. The main purpose is to record the existence and capabilities of a number of tools. More information on existing packages can be found by looking at the current set of HELP files (DIR PH:*.* on the DEC-20).

18.2. 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:

It should also be remembered that in REDUCE (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.

18.3. 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. 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);

18.2.2. 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.

18.2.3. Options

!*CREFSUMMARY (_Initially_: _NIL_) _flag_

> If the flag CREFSUMMARY is ON (or !*CREFSUMMARY is true in LISP), 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_:
(_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_)) _global_

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.

18.2.1. 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.

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
```

18.3.1. Running PictureRLISP on HP2648A and on TEKTRONIX 4006-1 Terminals

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.

18.4. Tools for Defining Macros

The following (and other) macro utilities are in the file PU:USEFUL.SL; use LOAD(useful) to access.[1] See PH:USEFUL.HLP for more information.

18.4.1. DefMacro

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.

---

[1] Useful was written by D. Morrison.

18.4.2. 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 "`" 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 `.

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**.

18.4.3. Sharp-Sign Macros

USEFUL 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 it non-nil. For example, #\a = #\A = 65, while #\!a = #\(lower a) = 97. Char has been redefined in USEFUL to be slightly more table driven -- users can now add new "prefixes" such as META or CONTROL: just hang the appropriate function (from integers to integers) off the char-prefix-function property of the "prefix". A LARGE number of additional alias for various characters have been added, including all the "standard" ASCII names like NAK and DC1.

#. : 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.

18.4.4. 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.

18.4.5. DefLambda

DefLambda (): — macro

Yet another little (two line) macro has been added to USEFUL: **DefLambda**. This 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 expensivee 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))).

18.5. 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.

18.6. 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. It is implemented in PSL[2] as a function which builds access macros and fns for "typed" vectors, including constructor and alterant macros, a type

---

[2] Defstruct was implemented by Russ Fish.

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.

18.6.1. 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 LHS or the RHS 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.

18.6.2. 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.

18.6.3. A Simple Example

(Input lines have a "> " prompt at the beginning.)

```
> % (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


> CO := MakeComplex();    % Constructor with default inits.
[COMPLEX 0 0]

> ComplexP CO;% 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


> M0 := 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
```

18.7. 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.

18.8. Find

These functions take a string or id, map the oblist to collect a list of ids with Prefix or Suffix as given:

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

Scans current id-hash-table for an id whose prefix matches KEY.

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

Scans current id-hash-table for an id whose suffix matches KEY.

Thus

```
X:=FindPrefix '!*;      Finds all ids starting with *
```

Use the 'GSORT' package to sort the list:

```
Gsort(X,'Idsort);
```

See Section 7.4 for more information about sorting functions.

18.9. 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 very 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 a large amount of common substructure, since only one copy of the substructure is stored.

The system works by keeping a 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 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). Two arguments 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

HCons version of **Append**.

HReverse (U:list): list expr

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

18.10. Graph-to-Tree

GRAPH-TO-TREE is a loadable module. For resident functions printing circular lists see Section 16.9.

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.

18.11. Inspect Utility

INSPECT is a loadable module.

Inspect (): 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 IN 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(); % Later will get a file Name
IN ....;
IN ...;
InspectEnd;
```

Now use GSort, etc. to process the lists.

18.12. Trigonometric and Other Mathematical Functions

The MATHLIB package contains some useful mathematical functions as an interim until more efficient functions can be defined in SYSLISP, or higher precision BIGFLOAT's implemented using Sasaki's package. This BigFloat package has more functions, with user controlled precision; the FLOAT package should define equivalent functions.[3]

Ceiling (_N:number_): _integer_ **expr**

Returns the largest _integer_ smaller than its argument.

---

[3] MATHLIB consists of contributions by M. Griss, E. Benson, D. Morrison, W. Galway and D. Irish.

Round (N:number): integer   expr

Rounds to the closest integer. Kind of sloppy -- it's biased if the digit causing rounding is a five.

The trigonometric functions usually expect Radians (=PI*degrees/180), except as indicated.

DegreesToRadians (X:number): number   expr

RadiansToDegrees (X:number): number   expr

RadiansToDMS (X:number): number   expr

Converts degrees, minutes, seconds to radians. DegreesToRadians(Degs+Mins/60.0+Sex/3600.0)

DMSToRadians (DEGS:number, MINS:number, SEX:number): number   expr

Sin (X:number): number   expr

Accurate to about 6 decimal places, so long as the argument is of commensurate precision. This, of course, is NOT true for large arguments, since they come in with small precision.

ScaledSine (X:number): number   expr

This assumes its argument is scaled to between 0 and pi/2.

Cos (X:number): number   expr

Accurate to about 6 decimal places, so long as the argument is of commensurate precision. This, of course, is NOT true for large arguments, since they come in with small precision.

ScaledCosine (X:number): number   expr

Expects its argument to be between 0 and pi/2.

Tan (X:number): number   expr

Accurate to about 6 decimal places, so long as the argument is of commensurate precision. This, of course, is NOT true for large

arguments, since they come in with small precision.

Cot (X:number): number expr

Accurate to about 6 decimal places, so long as the argument is of commensurate precision. This, of course, is NOT true for large arguments, since they come in with small precision.

ScaledTangent (X:number): number expr

Expects its argument to be between 0 and pi/4.

ScaledCotangent (X:number): number expr

Expects its argument to be between 0 and pi/4.

Sec (X:number): number expr

Csc (X:number): number expr

SinD (X:number): number expr

X in Degrees, converted to radians.

CosD (X:number): number expr

X in Degrees, converted to radians.

TanD (X:number): number expr

X in Degrees, converted to radians.

CotD (X:number): number expr

X in Degrees, converted to radians.

SecD (X:number): number expr

X in Degrees, converted to radians.

CscD (X:number): number expr

X in Degrees, converted to radians.

Asin (X:number): number expr

Acos (X:number): number expr

CheckedArcCosine (X:number): number expr

Atan (X:number): number expr

Acot (X:number): number expr

CheckedArcTangent (X:number): number expr

This assumes it's argument is in the range 0 <= x <= 1.

Asec (X:number): number expr

Acsc (X:number): number expr

AsinD (X:number): number expr

X in Degrees, converted to radians.

AcosD (X:number): number expr

X in Degrees, converted to radians.

AtanD (X:number): number expr

X in Degrees, converted to radians.

AcotD (X:number): number expr

X in Degrees, converted to radians.

AsecD (X:number): number   expr

X in Degrees, converted to radians.

AcscD (X:number): number   expr

X in Degrees, converted to radians.

Sqrt (N:number): number   expr

Simple Newton-Raphson floating point square root calculator. Not warranteed against truncation errors, etc.

Exp (X:number): number   expr

Returns the exponential (i.e. e**x) of its float argument as a float. The argument is scaled to the interval -ln 2 to 0, and a Taylor series expansion used (formula 4.2.45 on page 71 of Abramowitz and Stegun, "Handbook of Mathematical Functions").

Log (X:number): number   expr

CheckedLogarithm (X:number): number   expr

Log2 (X:number): number   expr

Log10 (X:number): number   expr

Random (): number   expr

The declarations below constitute a linear, congruential random number generator (see Knuth, "The Art of Computer Programming: Volume 2: Seminumerical Algorithms", pp. 9-24). With the given constants it has a period of 392931 and potency 6. To have deterministic behavior, set RANDOM!-SEED.

Constants are:
   modulus: 392931 = $3^6 * 7^2 * 11$
   multiplier: 232 = 3 * 7 * 11 + 1
   increment: 65537 is prime

[??? Would benefit from being recoded in SYSLISP, when full word integers should be used with "automatic" modular arithmetic (see Knuth). Perhaps we should have a longer period version? ???]

The following constants have been defined:

```
put('Number2Pi,'NewNam,6.2831853);
put('NumberPi,'NewNam,3.1415927);
put('NumberPi!/2,'NewNam,1.5707963);
put('NumberPi!/4,'NewNam,0.78539816);
put('Number3Pi!/4,'NewNam,2.3561945);
put('Number!-2Pi,'Newnam,-6.2831853);
put('Number!-Pi,'NewNam,-3.1415927);
put('Number!-Pi!/2,'NewNam,-1.5707963);
put('Number!-Pi!/4,'NewNam,-0.78539816);

put('SmallestFloNum,'NewNam,(1/1.0e38));  %/ Fix for new Token
                                          % scanner
put('SqrtTolerance,'NewNam,0.0000001);
put('NaturalLog2,'NewNam,0.69314718);
put('NaturalLog10,'NewNam,2.3025851);
```

CHAPTER 19
LOADER AND COMPILER

19.1. Introduction 19.1
19.2. The Compiler 19.1
19.2.1. Compiling Functions into Memory 19.2
19.2.2. Compiling Functions into FASL Files 19.2
19.2.3. Loading FASL Files 19.3
19.2.4. Functions to Control the Time When Something is Done 19.3
19.2.5. Order of Functions for Compilation 19.4
19.2.6. Fluid and Global Declarations 19.4
19.2.7. Flags Controlling Compiler 19.5
19.2.8. Differences between Compiled and Interpreted Code 19.6
19.2.9. Compiler Errors 19.7
19.3. The Loader 19.8
19.3.1. Legal LAP Format and Pseudos 19.9
19.3.2. Examples of LAP for DEC-20, VAX and Apollo 19.9
19.3.3. Lap Flags 19.12
19.4. Structure and Customization of the Compiler 19.13
19.5. First PASS of Compiler 19.13
19.5.1. Tagging Information 19.14
19.5.2. Source to Source Transformations 19.14
19.6. Second PASS - Basic Code Generation 19.14
19.6.1. The Cmacros 19.14
19.6.2. Classes of Functions 19.17
19.6.3. Open Functions 19.17
19.7. Third PASS - Optimizations 19.22
19.8. Some Structural Notes on the Compiler 19.22

19.1. Introduction

The functions and facilities in the PSL LISP/SYSLISP compiler and supporting loaders (LAP and FASL) are described in this Chapter.

19.2. 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

---

[1] Many of the recent extensions to the PLC were implemented by John Peterson.

compilation. See the later Sections in this Chapter and Refs. [Griss 81] and [Benson 81] for more details.

19.2.1. Compiling Functions into Memory

!*COMP (Initially: NIL) flag

If the compiler is loaded (which is usually the case, otherwise execute LOAD COMPILER;), turning on the flag !*COMP (via **on comp;** in RLISP) 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. See !*PWRDS in Chapter 12.

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

19.2.2. 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 in RLISP mode:

FaslOut (FILE:string): NIL expr

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 (): NIL expr

terminates this process.

The FaslOut and FaslEnd pair also use the DFPRINT!* mechanism, turning on the flag !*DEFN, and redefining DFPRINT!* to trap the parsed input in the RLISP top-loop. Currently this is not useable from pure LISP level.

[??? Fix, by adding !*DEFN mechanism to basic top-loop. ???]

19.2.3. Loading FASL Files

Two convenient procedures are available for loading FASL files (.b files on the VAX); see Section 19.2.2 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!*. A FLAG 'LOADED is placed on the property list of FILE to prevent subsequent reloading.

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.

Imports (): expr

[??? Describe FASL format in more detail ???]

19.2.4. 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 ???]

19.2.5. 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 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. ???]

19.2.6. 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 flag to make the default Fluid, or should we make Interpreter have a LOCAL variable as default, or both ???]

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 on the Binding Stack (BSTACK), and all access is always to the VALUE cell (SYMVAL) 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 ???]

19.2.7. Flags Controlling Compiler

The compilation process is controlled by a number of flags, as well as the above declarations and the !*COMP flag, of course.

!*R2I (Initially: T) flag

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) flag

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) flag

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) flag

Indicates block compilation (a future extension of this compiler). When implemented, even more function and variable names are "compiled away".

The following flags control the printing of information during the compilation process:

!*PWRDS (Initially: NIL) flag

If T, causes the compiled size to be printed in the form

*** NAME: base NNN, length MMM

The base is in octal, the length is in current Radix.

[??? more mnemonic name ???]

!*PLAP (Initially: NIL) flag

If T, causes the printing of the portable cmacros produced by the the compiler.

Most of this information is printed by the resident LAP, and controlled by its flags, described below.

19.2.8. 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 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 flags 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.

19.2.9. 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.

19.3. The Loader

[??? update ???]

Currently, PSL on the DEC-20 provides a simple LISP assembler, LAP. This is modeled after the original LISP 1.6 LAP, although completely reimplemented to take advantage of PSL constructs, and to support the additional requirements of SYSLISP. In the process of implementing the VAX LAP and developing the LAP-to-ASM translator required to bootstrap PSL onto the next machine (Apollo MC68000), a much more table-driven form of LAP was designed to make all LAP's, LAP-to-ASM's and FASL's (fast loaders,

sometimes called FAP) easier to maintain. This is now in use on the VAX and being used to implement Apollo PSL.

[??? FASL now works ???]

Until that is complete, we will briefly describe the available functions, and give a sample of current and future LAP; this Section will be completely rewritten in the next revision. LAP is currently a full two pass assembler; on the VAX and Apollo it also includes a pass to optimize long and short jumps.

LAP (CODE:list): code-pointer   expr

CODE is a list of legal LAP forms, including:

a. Machine specific Mnemonics (using opcode-names from the assembler on the DEC-20, VAX or Apollo).

b. Compiler cmacros (which expand in a machine specific way). These can be thought of as "generic" or LISP-oriented instructions. See the next Section on the Compiler details, and list of legal cmacros.

c. LAP pseudo instructions, to declare entry points, indicate data and constants, etc.

The first pass of LAP converts mnemonics into LISP integers, doing as much of the assembly as possible, allocating labels and constants. The second (and third?) pass fills in labels and completes the assembly, depositing code into the next available locations in BPS, or creating FASL or LAP files.

[??? What is BPS (binary program space) ???]

19.3.1. Legal LAP Format and Pseudos

[??? Describe LAP format in detail ???]

19.3.2. Examples of LAP for DEC-20, VAX and Apollo

The following is a piece of VAX specific LAP, using the current NEW format. Apart from the VAX mnemonics, notice the extra tags around the register names, and the symbols to indicate addressing modes (essentially PREFIX syntax rather then INFIX @ etc.). This is from PV:APPLY-LAP.RED. Note they are almost ENTIRELY written in cmacros, to aid in re-coding for the next machine.

```
lap '((!*entry FastApply expr 0)
%. Apply with arguments loaded
% Called with arguments in the registers and functional form in t1
        (!*FIELD (reg t2) (reg t1)
                 (WConst TagStartingBit) (WConst TagBitLength))
        (!*FIELD (reg t1) (reg t1)
                 (WConst InfStartingBit) (WConst InfBitLength))
        (!*JUMPNOTEQ (Label NotAnID) (reg t2) (WConst ID))
        (!*WTIMES2 (reg t1) (WConst AddressingUnitsPerFunctionCell))
        (!*JUMP (MEMORY (reg t1) (WArray SymFnc)))
NotAnID
        (!*JUMPNOTEQ (Label NotACodePointer) (reg t2) (WConst CODE))
        (!*JUMP (MEMORY (reg t1) (WConst 0)))
NotACodePointer
        (!*JUMPNOTEQ (Label IllegalFunctionalForm) (reg t2) (WConst PAIR))
        (!*MOVE (MEMORY (reg t1) (WConst 0)) (reg t2))
                                      % CAR with pair already untagged
        (!*JUMPNOTEQ (Label IllegalFunctionalForm) (reg t2) (QUOTE LAMBDA))
        (!*MOVE (reg t1) (reg t2))    % put lambda form in t2
        (!*PUSH (QUOTE NIL))                  % align stack
        (!*JCALL FastLambdaApply)
IllegalFunctionalForm
        (!*MOVE (QUOTE "Illegal functional form in Apply") (reg 1))
        (!*MOVE (reg t1) (reg 2))
        (!*CALL List2)
        (!*JCALL StdError)
);

lap '((!*entry UndefinedFunction expr 0)
%. Error Handler for non code
%  Called by JSB
%
        (subl3 (immediate (plus2 (WArray SymFnc) 6))
               (autoincrement (reg st))
               (reg t1))
        (divl2 6 (reg t1))
        (!*MKITEM (reg t1) (WConst ID))
        (!*MOVE (reg t1) (reg 2))
        (!*MOVE (QUOTE "Undefined function %r called from compiled code")
                (reg 1))
        (!*CALL BldMsg)
        (!*JCALL StdError)
);
```

The following is a piece of Apollo specific LAP, using the current NEW format. Apart from the MC68000 mnemonics, notice the extra tags around the register names, and the symbols to indicate addressing modes (essentially PREFIX syntax rather then INFIX @ etc.). This is from P68:M68K-USEFUL-LAP.RED.

```
% Signed multiply of 32 bits numbers in A1 and A2,
% returns 64 bits in A1 and A2, low in A1 high in A2
% Clobbers D1,D2,D3,D4,D5,D6,D7, no saving
%   [Can insert MOVEM!.L D1-D7,-(SP)
%    and        MOVEM!.L (SP)+,D1-D7]
LAP '((!*entry Mult32 expr 2)  % Arguments in A1 and A2
      (move!.l (reg a1) (reg d1))
      (move!.l (reg a1) (reg d6))
      (move!.l (reg a2) (reg d2))
      (move!.l (reg a2) (reg d7))  % Need copies
 % Now do Unsigned Multiply
      (move!.l (reg d1) (reg d3))
      (move!.l (reg d1) (reg d4))
      (swap    (reg d4))
      (move!.l (reg d2) (reg d5))
      (swap    (reg d5))           % Swapped for partial products
      (mulu!.w (reg d2) (reg d1))  % partial products (pp1)
      (mulu!.w (reg d4) (reg d2))  %                   pp2
      (mulu!.w (reg d5) (reg d3))  %                   pp3
      (mulu!.w (reg d5) (reg d4))  %                   pp4
      (swap    (reg d1))           % sum1=pp#2low+pp#1hi
      (add     (reg d2) (reg d1))
      (clr!.l  (reg d5))
      (addx!.l (reg d5) (reg d4))  % propagate carry
      (add     (reg d3) (reg d1))  % sum2=sum1+pp#3low
      (addx!.l (reg d5) (reg d4))  % carry inot pp#4
      (swap    (reg d1))           % low order product
      (clr     (reg d2))
      (swap    (reg d2))
      (clr     (reg d3))
      (swap    (reg d3))
      (add!.l  (reg d3) (reg d2)) % Sum3=pp2low+pp3Hi
      (add!.l  (reg d4) (reg d2)) % Sum4=Sum3+pp4
 % Now do adjustment
      (tst!.l  (reg d7))          % Negative
      (bpl!.s  chkd6)     %  nope
      (sub!.l  (reg d6) (reg d2)) % Flip
  chkd6
      (tst!.l  (reg d6))          % Negative
      (bpl!.s  done)      %  nope
      (sub!.l  (reg d7) (reg d2)) % Flip
  done
      (movea!.l (reg d1) (reg a1)) % low part
      (movea!.l (reg d2) (reg a2)) % high part
      (rts));
```

19.3.3. Lap Flags

The following flags control the printing of information from LAP and other optional behavior of LAP:

!*PLAP (_Initially_: _NIL_) _flag_

Causes LAP forms to printed before expansion. Used mainly to see output of compiler before assembly.

!*PGWD (_Initially_: _NIL_) _flag_

Causes LAP to print the actual DEC-20 mnemonics and corresponding assembled instruction in octal, displaying OPCODE, REGISTER, INDIRECT, INDEX and ADDRESS fields.

!*PWRDS (_Initially_: _T_) _flag_

Prints a LAP message of the form

*** NAME: base NNN, length MMM

The base is in octal, the length is in current Radix.

!*SAVECOM (_Initially_: _T_) _flag_

If T, the LAP is deposited in BPS, and the returned Code-Pointer used to (re)define the procedure associated with the (!*entry name type n).

!*SAVEDEF (_Initially_: _NIL_) _flag_

If T, and if !*SAVECOM is T, saves any preexisting procedure definition under '!*SAVEDEF on the property list of the procedure name, "just in case".

LAP also uses the following indicators on property lists:

'MC Cmacros and some mnemonics have associated PASS1 expansions in terms of simpler instructions or operations. The form (mc a1 ... an) has its associated function applied to (a1 ... an).

For more details, see "P20:LAP.RED".

19.4. Structure and Customization of the Compiler

The following is a brief summary of the compiler structure and model. The purpose of this Section is to aid the user to add new compilation forms, and to understand the task of bootstrapping a new version of PSL. The original paper on the Portable LISP Compiler [Griss 81] has complete details on the original version of the compiler, and should be read in conjunction with this Section. It might be useful to also examine the paper on recent work on the compiler [Griss82].

[??? This needs a LOT of work ???]

The compiler is basically three-pass:

a. The first pass expands ordinary macros, and compiler specific cmacros. It also uses some special purpose 'PA1REFORM and 'PA1FN functions on the property lists of certain functions to produce a simpler and more explicit LISP for the next pass. Variables and constants, x, are explicitly tagged as (FLUID x), (GLOBAL x), (QUOTE x), (WCONST x), etc.

b. The second pass recursively compiles the code, using 'COMPFN's to handle special cases, and the recursive function !&COMPILE for the general case. In general, code is compiled to cause function arguments to be loaded into R1...Rn in order, a CALL to the function to be made, and the returned value to appear in R1. Temporaries and function arguments to be reused later are saved on the stack. The compiler allocates a single FRAME for the maximum stack space that might be needed, and then trims it down in the third pass. PSL requires registers R1 ... R15, though not all need be "REAL registers"; the extra are simulated as memory locations. Special cases avoid a lot of LOAD/STORES to move arguments around. The compiled code is emitted as a sequence of abstract LISP machine cmacros. The current set of cmacros is described below.

c. The third pass scans the list of cmacros for patterns, removing LOADs and STOREs, redundant JUMP's and LABEL's, compressings the stack frame, and possibly mapping temporaries stored on the stack into any of the REAL registers that would otherwise be unused. This optimized cmacro list is then passed to LAP.

19.5. First PASS of Compiler

19.5.1. Tagging Information

This affects many parts of the compiler. The basic idea is that all information is to be tagged. These tags fit in three categories: variable tags, location (register and frame) tags, and constant tags. Tags used for variables must be flagged 'VAR; tags for constants must be flagged 'CONST. Currently, the register tag is REG and the frame tag is FRAME. Frame locations are always positive integers.

These tags are used everywhere; thus, register 1 is always described by (REG 1) in both emitted cmacros and internally in the register list REGS. Pass 1 tags all variable references with a source to source transformation of the variables (suitably obscure names must be used for these tags to prevent conflicts with named functions).

The purpose behind this tagging is to make the compiler easier to work with in adding new features; new notions of registers, constants, and variables can all be accommodated through new tags. Also, the components of the cmacros are more clearly identified for pass 3.

19.5.2. Source to Source Transformations

A PA1REFORMFN has been provided to augment PA1FN's. The only difference between these functions is that the PA1REFORM function is passed code which has already been through PASS1. This was previously done by calling pass 1 within a PA1FN.

19.6. Second PASS - Basic Code Generation

19.6.1. The Cmacros

The compiler second pass compiles the input LISP into a series of abstract machine instructions, called cmacros. These are instructions for a LISP-oriented Register machine.

<u>The current DEC-20 cmacros</u>

Definitions of arguments

```
reg:   (REG n)        n = 1,2,... MAXNARGS
var:   frame ¦ (GLOBAL name) ¦ (FLUID name)
frame: (FRAME n)      n = 0,1,2, ..
const: (QUOTE value) ¦ (WCONST value)
label: (LABEL symbol)
regn:  reg ¦ NIL ¦ frame
regf:  reg ¦ frame
loc:   reg ¦ var ¦ const
anyreg: (CAR anyreg) ¦ (CDR anyreg) ¦ loc
```

Basic Cmacros for LISP and SYSLISP

```
(!*ALLOC nframe)
(!*DEALLOC nframe)
(!*ENTRY fname ftype nargs)
(!*EXIT  nframe)
(!*FREERSTR (NONLOCALVARS f1 f2 ...))
(!*JUMP label)
(!*JUMPxx label loc loc')
        where xx = ATOM, EQ, NOTEQ, NOTTYPE, PAIRP, TYPE
(!*JUMPON lower upper (label-1 ... Label-n))
(!*LINK fname ftype nargs)
(!*LINKE nframe fn type nargs)
(!*LINKF nargs reg) where reg contains the function name,
                          nargs an integer
(!*LINKEF nframe nargs reg) %/ ?
(!*LBL label)
(!*LAMBIND (REGISTERS reg1 reg2 ...) (NONLOCALVARS f1 f2 ...))
         where f1, f2, ... = (FLUID name )
          No frame location will be allocated (depends on flag)
(!*LOAD reg anyreg)
(!*PROGBIND (NONLOCALVARS f1 f2 ...))
(!*PUSH reg)
(!*RPLACA regf loc)
(!*RPLACD regf loc)
(!*STORE regn var) | (!*STORE regn reg)
```

SYSLISP oriented Cmacros

```
(!*ADDMEM loc)
(!*ADJSP ?)
(!*DECMEM loc)
(!*INCMEM loc)
(!*INTINF loc)
(!*JUMPWGEQ label loc loc')
(!*JUMPWGREATERP label loc loc')
(!*JUMPWITHIN label loc loc')
(!*JUMPWLEQ label loc loc')
(!*JUMPWLESSP label loc loc')
(!*MKITEM loc loc')
(!*MPYMEM loc loc')
(!*NEGMEM loc)
(!*SUBMEM loc loc')
(!*WAND loc loc')
(!*WDIFFERENCE loc loc')
(!*WMINUS loc)
(!*WNOT loc)
(!*WOR loc loc')
(!*WPLUS2 loc loc')
(!*WSHIFT loc loc')
(!*WTIMES2 loc loc')
(!*WXOR loc loc')
```

68000 Cmacros

Basic LISP and SYSLISP Cmacros

```
(!*ALLOC nframe)
(!*CALL fname)
(!*DEALLOC nframe)
(!*ENTRY fname ftype nargs)
(!*EXIT nframe)
(!*JCALL fname)
(!*JUMP label)
(!*JUMPEQ label loc loc')
(!*JUMPINTYPE label type)
(!*JUMPNOTEQ label loc loc')
(!*JUMPNOTINTYPE label loc type)
(!*JUMPNOTTYPE label loc type)
(!*JUMPTYPE label loc type)
(!*LAMBIND label loc loc')
(!*LBL label)
(!*LINK fname ftype nargs)
(!*LINKE fname ftype nargs nframe)
(!*MOVE loc loc')
(!*PROGBIND label loc loc')
(!*PUSH loc)
```

SYSLISP specific Cmacros

```
(!*APOLLOCALL label loc loc')
(!*ASHIFT loc loc')
(!*FIELD loc loc')
(!*FOREIGNLINK loc loc')
(!*INF loc loc')
(!*JUMPON loc loc')
(!*JUMPWGEQ loc loc')
(!*JUMPWGREATERP loc loc')
(!*JUMPWITHIN loc loc')
(!*JUMPWLEQ loc loc')
(!*JUMPWLESSP loc loc')
(!*LOC loc loc')
(!*MKITEM loc loc')
(!*PUTFIELD loc loc')
(!*PUTINF loc loc')
(!*PUTTAG loc loc')
(!*SIGNEDFIELD loc loc')
(!*TAG loc loc')
(!*WAND loc loc')
(!*WDIFFERENCE loc loc')
(!*WMINUS loc loc')
(!*WNOT loc loc')
(!*WOR loc loc')
(!*WPLUS2 loc loc')
```

```
(!*WSHIFT loc loc')
(!*WTIMES2 loc loc')
(!*WXOR loc loc')
```

19.6.2. Classes of Functions

The compiler groups functions into four basic classes:

a. ANYREG functions. No side effects and can be done in a single register. Passed directly to CMACROs. Viewed as a form of "extended addressing" mode.

b. Specially compiled or "OPEN" functions. These are functions have a special compiling function stored under a 'COMPFN indicator. While many of these functions are specially coded, many are written with the aid of supporting patterns; these are called 'OPENFN or 'OPENTST patterns. Some OPEN functions alter registers which are in use, allocate new frames or obtain unused registers. These open functions also include open compilation of tests.

c. Built-in or 'stable' functions. These functions are called in the standard fashion by the compiler, but they have properties which are useful to the compiler and are assumed to always hold. Currently, a function may be flagged as NOSIDEEFFECT and have the property DESTROYS, which contains a list of registers destroyed by the function.

d. All other functions are assumed to be totally random, destroying every register and causing side effects.

[??? Mark non-random functions of various levels elsewhere ???]

The most important of these categories is the OPEN function. It is hoped that improved OPEN functions will eliminate the need for temporary registers to be allocated by the assembler. Most OPEN functions emit cmacros especially tailored for each function.

19.6.3. Open Functions

[??? Explain how to CODE them ???]

There are 3 basic kinds of open function:

a. Test: the destination is a LABEL.
b. Value: the result is to be placed in a particular register.

c. Effect: the result is a side effect, and no destination is needed.

Note that an EFFECT open function does not have a destination. It is not really a separate class of function, just a separate usage. Example:

```
(PROGN (SETQ X 0) ... )
```

- the SETQ is for effect only - could be implemented with a "clear" instruction.

```
(FOO (SETQ X 0) ... )
```

- here the 0 is also placed in a register (the destination register).

The use of OPENTST is also derived from context: in

```
(COND ((EQ A B) ...))
```

- EQ is interpreted as a test.

```
(RETURN (EQ A B))
```

, though, must have a value. It should be noted that a pseudo source-source transformation occurs if an OPENTST is called for value:

```
(RETURN (EQ A B)) ->
  (RETURN (COND ((EQ A B) T) (T NIL)))
```

An OPENTST function always returns T/NIL if called for value. No separate handling for non test cases is needed (as opposed to the effect/value cases for normal OPEN funs in which two separate expansions can be supplied)

Also, there are 3 basic issues encountered in generating the code:

a. Bringing arguments into registers as needed.
b. Emitting the actual code.
c. Updating the final register contents.

Initially, the arguments to an open function are removed of all but ANYREG functions. Thus, these arguments fall into four classes:

a. Registers
b. Memory locations (FLUID, GLOBAL, FRAME, !*MEMORY)

c. Constants
d. ANYREG functions (viewed as extended addressing modes)

Also, along with the arguments coming in is the destination (register or label).

The first step is to replace some arguments by registers by emitting LOAD's. This step can be controlled by a function, called the adjust function, which emits LOAD's and replaces the corresponding arguments by registers. Next, cmacros are emitted. These cmacros are selected through a pattern which defines the format of the particular OPEN function call.

Note that the pattern is matching the locations of the arguments to the open function. For example, assume that FOO is OPEN, and the call

```
(FOO 'A (CDR B) C D)
```

is encountered. Assume also that B is frame 1, C is frame 2, and D was found in reg 1.

The argument list being matched is thus

```
('A (CDR (FRAME 1)) (FRAME 2) (REG 1))
```

For most purposes, this would be interpreted as (const anyreg mem reg). Of course, a pattern can use the value of a constant (you might recognize (!*WPLUS2 1 X) as an increment). Also, the actual register may be important for register args, especially if one of the args is also the destination. You would probably emit different code for

```
(REG 1) := (!*WPLUS2 (REG 2) (REG 3))
```

than

```
(REG 1) := (!*WPLUS2 (REG 1) (REG 2))
```

To avoid a profusion of properties which would be associated with an OPEN function, two properties of the function name are used to hold all information associated with OPEN compiling. These properties are OPENFN and OPENTST.

The OPENFN and OPENTST properties have the following format:

```
      (PATTERN MACRONAME PARAMETERS)
   or function name.
```

The PATTERN field contains either the pattern itself or a pattern name. A pattern name is an id having the PATTERN property. In the following material, DEST refers to the destination label in an OPENTST and to the destination register in an OPENFN. If the function is being evaluated for effect only, DEST is a temporary register which need not be used.

A pattern has the following format:

```
(ADJUST_FN
 REG_FN
 (P1 M11 M12 M13 ..)
 (P2 M21 M22 M23 ..)
 ...)
```

The Pi are patterns and Mij are cmacros or pseudo cmacros. ADJUST_FN is a register adjustment function used to place things in registers as required, and to factor out basic properties of the function from the pattern. For example, you almost never could do anything with ANYREG stuff except load it somewhere (emitting (!*WPLUS2 X (CDR (CAR Y))) directly probably won't work - you must bring (CDR (CAR Y)) into a reg before further progress can be made). The most common adjust function is NOANYREG, which replaces ANYREG stuff with registers. This eliminates the problem of having to test for ANYREG stuff in the patterns.

Some pattern elements currently supported are:

ANY	matches anything
DEST	matches the destination register or label
NOTDEST	matches any register except the destination
REG	matches any register
REGN	Any register or 'NIL or a frame location
VAR	A LOCAL, GLOBAL, or FLUID variable
MEM	A memory address, currently constants + vars (NOT REGS)
ANYREGFN	matches an ANYREG function
'literal	matches the literal
(p1 p2 ... pn)	matches a field whose components match p1 ... pn
NOVAL	matches only if STATUS > 1; must be the first component of a pattern, consumes no part of the subject.

The cmacros associated with the patterns fall into two classes: actual cmacros to be emitted and pseudo cmacros which are interpreted by the compiler. In either case, the components of the cmacros are handled in the same fashion. The cmacros contain:

Ai	replaced by the ith argument to the OPEN function (after adjustment)

Ti replaced by a temporary register
Li replaced by a temporary label
Pi replaced by corresponding parameter from OPENFN
DEST replaced by the destination register or label (depending on OPENFN or OPENTST).
FN replaced by the name of the OPEN function
MAC synonym for P1, by convention a cmacro name
'literal
(x1 x2 ...)
xi as above, forms a list

The pseudo cmacros currently supported are:

!*DESTROY (R1, R2, ...): list — cmacro

Remove any register values from R1 ... RN.

!*DO (FUNCTION ARG1 ARG2 ...): list — cmacro

Call the FUNCTION.

!*SET (REG VAL): list — cmacro

Set the value in REG to VAL.

The cmacros which are known to the compiler are

!*LOAD (): list — cmacro

!*STORE (): list — cmacro

!*JUMP (): list — cmacro

!*LBL (): list — cmacro

These cmacros have special emit functions which are called as they are emitted; otherwise the cmacro is directly attached to CODELIST.

19.7. Third PASS - Optimizations

The third pass of the compiler is responsible for doing optimizations, getting rid of extra labels and jumps, removing redundant code, adjusting the stack frame to squeeze out "holes" or even reallocating temporaries to excess registers if no "random" functions are called by this function.

This pass also does "peephole" optimizations (controlled by patterns that examine the Output CMACRO list for cmacros that can be merged). These tables can be adjusted by the user. This pass also gathers information on register usage that may be accumulated to aid block compilation or recompilation of a set of functions that are NOT redefined, and so can use information about each other (i.e. become "stable").

The 'OPTFN property is used to associate an optimization function with a particular CMACRO name. This function looks at the CMACRO arguments and some subsequent CMACROs in the code-list, to see if a transformation is possible. The OPTFN takes a single argument, the code-list in reverse order starting at the associated CMACRO. The OPTFN can also examine certain parameters. Currently !*LBL, !*MOVE and !*JUMP have 'OPTFNS. For example, !&STOPT, associated with !*MOVE, checks if previous CMACRO was !*ALLOC, and that this !*MOVE moves a register to the slot just allocated. If so, it converts the !*ALLOC and !*MOVE into a single !*PUSH. Likewise, !&LBLOPT removes duplicate labels defined at one place, aliasing one with the other, and so permitting certain JUMP optimizations to take place.

Tags in the cmacros are processed in a final pass through the code. At this time the compiler can do substitutions using functions attached to these tags. Currently, (!*FRAMESIZE) is converted to the frame size and holes are squeezed out (using the FRAME tag) by !&REFORMMACROS. Transformation functions are attached to tags (or any function) through the TRANFN property currently.

19.8. Some Structural Notes on the Compiler

[??? This Section is very ROUGH, just to give some additional information in interim ???]

External variables and properties used by the compiler:

<u>Variables and Flags</u>

!*ERFG (Initially:) flag

!*INSTALLDESTROY (Initially: NIL) flag

If true, causes the compiler to install the DESTROYS property on any function compiled which leaves one or more registers unchanged

!*INT (Initially: T) flag

!*NOFRAMEFLUID (Initially: T) flag

If true, inhibits allocation of frame locations for FLUIDS

!*SHOWDEST (Initially: NIL) flag

If true, compiler prints out which registers a function destroys unless all are destroyed

!*SYSLISP (Initially: NIL) flag

Switch compilation mode from default of LISP to SYSLISP. This affects constant tagging, and in RLISP also causes LISP functions to be replaced by SYSLISP equivalents. Also, non-locals default to WVAR's rather than FLUIDs. See Chapter 21.

!*UNSAFEBINDER (Initially: NIL) flag

for Don's BAKER problem...GC may be called in Binder, so regs cannot be preserved, and Binder called as regular function.

!*USEREGFLUID (Initially: NIL) flag

If true, LAMBIND and PROGBIND cmacros may contain registers as well as frame locations (through FIXFRM).

Globals:

LASTACTUALREG (Initially: 5) global

The number of the last real register; FIXFRM does not map stack locations into registers > LASTACTUALREG. Also, temporary registers are actual registers if possible.

MAXNARGS (Initially: 15) global

Number of registers

Properties and Flags:

CONST A tag property, indicates tags for constants (WCONST and QUOTE)

EXTVAR A tag property, indicates a variable type whose name is externally known (!$FLUID, !$GLOBAL, !$WVAR)

MEMMOD A cmacro property, indicates in place memory operations. The first argument to the cmacro is assumed to be the memory location (var or !*MEMORY)

NOSIDEEFFECT
A function property, used both in dealing with !*ORD and to determine if the result should be placed in register status

REG A tag property, indicates a register (REG)

TERMINAL A tag property, indicates terminals (leaves) whose arguments are not tagged items (!$FLUID !$GLOBAL !$WVAR REG LABEL QUOTE WCONST FRAME !*FRAMESIZE IREG)

TRANSFER A property of cmacros and functions, indicates cmacros & functions which cause unconditional transfers (!*JUMP !*EXIT !*LINKE !*LINKEF ERROR)

VAR A tag property, indicates a variable type (!$LOCAL !$FLUID !$GLOBAL !$WVAR)

Properties:

ANYREG A function property, non-NIL indicates an ANYREG function

CFNTYPE Used in compiler to relate to Recursion-to-iteration conversion.

DESTROYS A function property, contains a (tagged) list of registers destroyed by the function

DOFN A function property, contains the name of a compile time evaluation function for numeric arguments.

EMITFN A cmacro or pseudo cmacro property, contains the name of a special function for emitting (or executing) the cmacro, such as !&ATTJMP for !*JUMP.

EXITING A cmacro property, used in FIXLINKS. Contains the name of an associated exiting cmacro (!*LINK : !*LINKE, !*LINKF : !*LINKEF)

FLIPTST A function property, contains the name of the opposite of a test function. All open compiled test functions must have one. (EQ : NOTEQ, ATOM : PAIRP)

GROUPOPS A function property, used in constant folding. Attached to the three functions of a group, always a list of the three functions in the order +, -, MINUS. (!*WPLUS2, !*WDIFFERENCE, !*WMINUS : (!*WPLUS2 !*WDIFFERENCE !*WMINUS))

MATCHFN A property attached to an atom in a pattern. Contains the name of a boolean function for use in pattern matching.

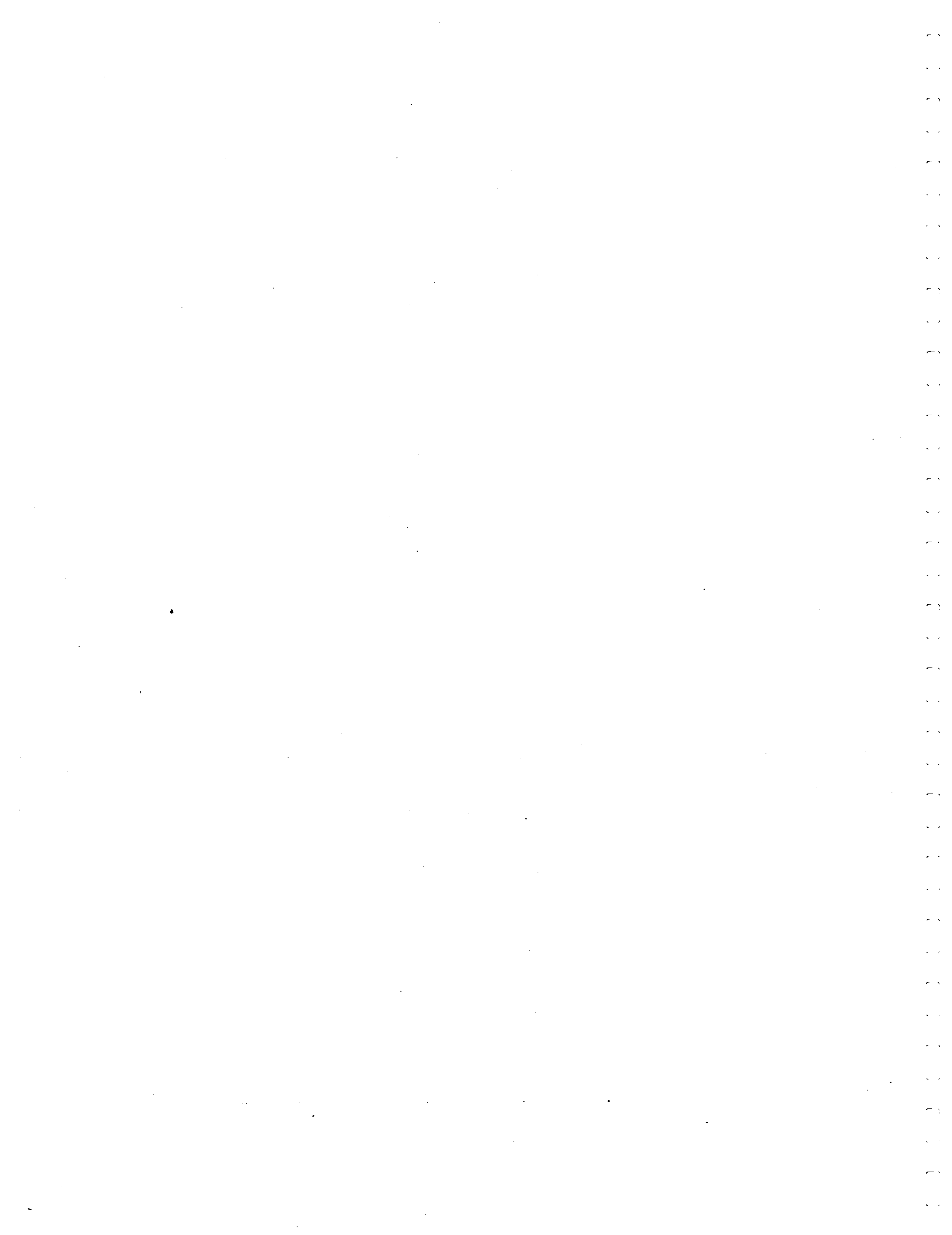
NEGJMP A cmacro property, contains the inverted test jump cmacro name. (!*JUMPEQ : !*JUMPNOTEQ, !*JUMPNOTEQ : !*JUMPEQ ...)

ONE   A function property, contains the (numeric) value of an identity associated with the function (!*WPLUS2 : 0, !*WTIMES2 : 1, ...)

PATTERN   A property associated with atoms appearing in OPENFN or OPENTST properties, contains a pattern for open coding of functions.

SUBSTFN   A property of atoms found in cmacros which are inside patterns. Contains a function name; the function value is substituted into the cmacro as emitted.

ZERO   Like ONE, designates a value which acts as a 0 in a ring over *. (!*WTIMES2 : 0 , !*LOGAND : 0)



CHAPTER 20
OPERATING SYSTEM INTERFACE

20.1. Introduction 20.1
20.2. System Dependent Functions 20.1
20.3. TOPS-20 Interface 20.2
  20.3.1. User Level Interface 20.2
  20.3.2. The Basic Fork Manipulation Functions 20.4
  20.3.3. File Manipulation Functions 20.5
  20.3.4. Miscellaneous Functions 20.6
  20.3.5. Jsys Interface 20.6
  20.3.6. Bit, Word and Address Operations for Jsys Calls 20.8
  20.3.7. Examples 20.9

20.1. Introduction

From within each PSL implementation, there will be a set of functions that permit the user to access specific operating system services. On the DEC-20 and VAX 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 will attempt to provide such calls (**EXEC** and **CMDS**) in all PSL implementations. We also will provide as clean an interface to Low-level services as possible. On the DEC-20, this is the **Jsys** function. Appropriate support functions (such as bit operations, byte-pointers, etc.) are also used by the assembler. On the VAX we will provide the **SYSCALL** capability.

20.2. System Dependent Functions

If_System (SYS-NAME:id, TRUE-CASE:any, FALSE-CASE:any): any cmacro

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"));
```

20.3. TOPS-20 Interface

20.3.1. 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.

DoCmds (L:string-list): any — expr

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 (Initially: "<cr><lf>") — global

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 — expr

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 — fexpr

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 — expr

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.

[??? Figure out how to pass a buffer to from Emacs ???]

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.

20.3.2. 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.

20.3.3. 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.

20.3.4. Miscellaneous Functions

GetUName (): string expr

Get USER name as a string

GetCDir (): string expr

Get Connected DIRECTORY

InFile ([FILS:id-list]): Unknown fexpr

Either solicit user for file name (InFile), and then open that file, else open specified file, for input.

20.3.5. Jsys Interface

The **Jsys** interface and jsys-names (as symbols of the form jsXXX) are defined in the source file PU:JSYS0.RED.

The access to the Jsys call is modeled after IDapply to avoid CONS, 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 strings 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 convertions 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.

20.3.6. Bit, Word and Address Operations for Jsys Calls

RecopyStringToNULL (*S*:*w-string*): *string* expr

S is assumed to be the address of a string, and a legal, tagged string is created, 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* expr

Swap half words of X; actually Xword(LowHalfWord X,HighHalfWord X).

LowHalfWord (*X*:*integer*): *integer* expr

Return the low-half word of the machine representation of X. Actually Land(X,8#777777).

HighHalfWord (*X*:*integer*): *integer* expr

Return the Upper half word as a small integer, of the machine word representation of X. Actually Lsh(Land(X,8#777777000000),-18).

Xword (*X*:*integer*,*Y*:*integer*): *integer* expr

Build a Word from Half-Words, actually Lor(Lsh(LowHalfWord(X),18),LowHalfWord Y).

JBits (L:list): integer expr

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 macro

A convenient access to Jbits: JBits cdr L.

20.3.7. Examples

The following range of examples illustrate the use of the above functions. More examples can be found in PU:exec0.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.

Procedure OPENOLDJfn Jfn;          %. OPEN to READ
 JSYS0(Jfn,Bits( (7 . 5),19),0,0,jsOPENF);

Lisp procedure GetFork Jfn;       %. Create Fork, READ File on Jfn
   Begin scalar FH;
      FH := JSYS1(Bits(1),0,0,0,jsCFork);
      JSYS0(Xword(FH ,Jfn),0,0,0,jsGet);
      return FH
   END;

Procedure GetOLDJfn FileName; %. test If file OLD and return Jfn
   Begin scalar Jfn;
      If NULL StringP FileName then return NIL;
      Jfn := JSYS1(Bits(2,3,17),FileName,0,0,jsGTJfn);
         % OLD!MSG!SHORT
      If Jfn<0 then return NIL;
      return Jfn
   END;

Procedure GetUNAME;        %. USER name
 Begin Scalar S;
   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
 End;

Procedure ReadTTY;
Begin Scalar S;
        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
end;
```

CHAPTER 21
SYSLISP

21.1. Introduction to the SYSLISP level of PSL 21.1
21.2. The Relationship of SYSLISP to RLISP 21.2
21.2.1. SYSLISP Declarations 21.2
21.2.2. SYSLISP Mode Analysis 21.3
21.2.3. Defining Special Functions for Mode Analysis 21.3
21.2.4. Modified FOR Loop 21.4
21.2.5. Char and IDLOC Macros 21.4
21.2.6. The Case Statement 21.5
21.2.7. Memory Access and Address Operations 21.7
21.2.8. Bit-Field Operation 21.7
21.3. Using SYSLISP 21.9
21.3.1. To Compile SYSLISP Code 21.9
21.4. SYSLISP Functions 21.10
21.4.1. W-Arrays 21.11
21.5. Remaining SYSLISP Issues 21.11
21.5.1. Stand Alone SYSLISP Programs 21.12
21.5.2. Need for Two Stacks 21.12
21.5.3. New Mode System 21.12
21.5.4. Extend CREF for SYSLISP 21.12

21.1. Introduction to the SYSLISP level of PSL

SYSLISP [Benson 81] is a BCPL-like language, couched in LISP form, providing operations on machine words, machine bytes and LISP ITEMs (tagged objects, packed into one or more words). We actually think of SYSLISP as a lower level of PSL, dealing with words, bytes, bit-fields, machine operations, and compile-time storage allocation, enabling us to write essentially all of the kernel in PSL.

The control structures and definition language are those of LISP, but the familiar **Plus2**, **Times2**, etc. are mapped to word operations **WPlus2**, **WTimes2**, etc. SYSLISP handles static allocation of SYSLISP variables and arrays and initial LISP symbols, permitting the easy definition of higher level Standard LISP functions and storage areas. SYSLISP provides convenient compile-time constants for handling strings, LISP symbols, etc. The SYSLISP compiler is based on the PORTABLE STANDARD LISP Compiler, with extensions to handle word level objects and efficient, open-coded, word-level operations. The SYSLISP mode of the compiler does efficient compile-time folding of constants and more comprehensive register allocation than in the distributed version of the PLC. Currently, SYSLISP handles bytes through the explicit packing and unpacking operations **GetByte**(word-address,byte-number) /

PutByte(word-address,byte-number,byte-value) without the notion of byte-pointer; it is planned to extend SYSLISP to a C-like language by adding the appropriate declarations and analysis of word/byte/structure operations.

SYSLISP is a collection of functions and their corresponding data types which are used to implement low level primitives in PSL, such as storage allocation, garbage collection and input and output. The basic data object in SYSLISP is the "word", a unit of storage large enough to contain a LISP item. On the PDP-10, a SYSLISP word is just a 36-bit PDP-10 word. On the VAX and most other byte addressed machines, a word is 4 bytes, or 32 bits. Conceptually, SYSLISP functions manipulate the actual bit patterns found in words, unlike normal LISP functions which manipulate higher-level objects, such as pairs, vectors, and floats or arbitrary-precision numbers. Arithmetic in SYSLISP is comparable to the corresponding operations in FORTRAN or PASCAL. In fact, SYSLISP is most closely modeled after BCPL, in that operations are essentially "typeless".

21.2. The Relationship of SYSLISP to RLISP

RLISP was extended with a CASE statement, SYSLISP declarations, smacros and macros to provide convenient infix syntax (+, *, / etc.) for calling the SYSLISP primitives. Even though SYSLISP is semantically somewhat different from LISP (RLISP), we have tried to keep the syntax as similar as possible so that SYSLISP code is "familiar" to RLISP users, and easy to use. RLISP functions can be easily converted and interfaced to functions at the SYSLISP level, gaining considerable efficiency by declaring and directly using words and bytes instead of tagged LISP objects.

21.2.1. SYSLISP Declarations

SYSLISP variables are either GLOBAL, memory locations (allocated by the compiler), or local stack locations. Locals are declared by SCALAR, as usual. Globals come in the following flavors:

```
WCONST id = wconstexp {,id = wconstexp} ;
```

Wconstexp is an expression involving constants and wconsts.

```
WVAR wvardecl {, wvardecl} ;

wvardecl ::= id | id = wconstexp

WARRAY warraydecl {, warraydecl} ;

warraydecl ::= id[wconstexp] | id[] = [ wconstexp {,wconstexp} ]
                    | id[] = string
```

WSTRING warraydecl {, warraydecl} ;

Each of these declarations can also be prefixed with the keywords:

INTERNAL or EXTERNAL.

If nothing appears, then a DEFAULT is used.

(Notice there are no metasyntactic square brackets here,
only curly brackets.)

For example, the following GLOBAL-DATA is used in PSL:

```
on SysLisp;

exported WConst MaxSymbols = 8000,
                MaxConstants = 500,
                HeapSize = 100000;

external WArray SymNam, SymVal, SymFnc, SymPrp, ConstantVector;

external WVar NextSymbol, NextConstant;

exported WConst MaxRealRegs = 5,
                MaxArgs = 15;

external WArray ArgumentBlock;

off SysLisp;

END;
```

21.2.2. SYSLISP Mode Analysis

In SYSLISP mode, the basic operators +, *, -, /, etc., are bound to word operators (WPlus2, WTimes2, WMinus, etc.), which compile OPEN as conventional machine operations on machine words. Thus most SYSLISP expressions, loops, etc. look exactly like their RLISP equivalents.

21.2.3. Defining Special Functions for Mode Analysis

To have the Mode analyzer (currently a REFORM function) replace LISP function names by SYSLISP ones, do:

```
PUT('LispName,'SYSNAME,'SysLispName);
```

The Following have been done:

```
DefList('((Plus WPlus2)
          (Plus2 WPlus2)
          (Minus WMinus)
          (Difference WDifference)
          (Times WTimes2)
          (Times2 WTimes2)
          (Quotient WQuotient)
          (Remainder WRemainder)
          (Mod WRemainder)
          (Land WAnd)
          (Lor WOr)
          (Lxor WXor)
          (Lnot WNot)
          (LShift WShift)
          (LSH WShift)), 'SysName);

DefList('((Neq WNeq)
          (Equal WEq)
          (Eqn WEq)
          (Eq WEq)
          (Greaterp WGreaterp)
          (Lessp WLessp)
          (Geq WGeq)
          (Leq WLeq)
          (Getv WGetv)
          (Indx WGetv)
          (Putv WPutv)
          (SetIndx WPutv)), 'SysName);
```

21.2.4. Modified FOR Loop

The FOR loop is modified in SYSLISP mode to use the **Wxxxx** functions to do loop incrementation and testing.

[??? Should pick up via SysReform ???]

21.2.5. Char and IDLOC Macros

In SYSLISP mode, '<id> refers to the tagged item, just as in LISP mode, **IdLoc** <id> refers to the id space offset of the <id>, and **LispVar** <id> refers to the GLOBAL value cell of a GLOBAL or FLUID variable. Note: **LispVar** can be used on the left hand side of an argument sentence. For example, to store a NIL in the value cell of id FOO, we do any one of the following.

```
SYMVAL IDLOC FOO := 'NIL;

LISPVAR FOO := MKITEM(ID,IDLOC NIL);
```

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 certain special characters, remove QUOTE marks that may be needed to pass special characters through the parser, and can accept a prefixes to compute LOWER case, <Ctrl> characters, and <Meta> characters. For example:

```
Little_a:= Char LOWER A;  % In case we think RAISE will occur
Little_a:= Char '!a;      % !a should not be raised
Meta_X := Char META X;
Weird := Char META Lower X;
Dinger := Char <Ctrl-G>;
Dinger := Char BELL;
```

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)
          (RUB 8#177)
          (RUBOUT 8#177)
          (DEL 8#177)
          (DELETE 8#177)), 'CharConst);
```

21.2.6. The Case Statement

RLISP in SYSLISP mode provides a Numeric case statement, that is implemented quite efficiently; some effort is made to examine special cases (compact vs. non compact sets of cases, short vs. long sets of cases, etc.).

[??? Note, CASE can also be used from LISP mode, provided tags are numeric. There is also an FEXPR, CASE ???]

The syntax is:

```
Case-Statement ::= CASE expr OF case-list END

Case-list      ::=  Case-expr [; Case-list ]

Case-expr      ::=  Tag-expr : expr

tag-expr       ::=  DEFAULT | OTHERWISE  |
                    tag | tag, tag ... tag |
                    tag TO tag

Tag            ::=  Integer | Wconst-Integer
```

```
% This is a piece of code from the Token Scanner,
% in file "PI:token-Scanner.red"
.....
    case ChTokenType of
    0 to 9:      % digit
    <<  TokSign := 1;
        goto InsideNumber >>;
    10:  % Start of ID
    <<  if null LispVar !*Raise then
            goto InsideID
        else
        <<  RaiseLastChar();
            goto InsideRaisedID >> >>;
    11:  % Delimiter, but not beginning of diphthong
    <<  LispVar TokType!* := '3;
        return MkID TokCh >>;
    12:  % Start of comment
        goto InsideComment;
    13:  % Diphthong start - Lisp function uses P-list of starting char
        return ScanPossibleDipthong(TokChannel, MkID TokCh);
    14:  % ID escape character
    <<  if null LispVar !*Raise then
            goto GotEscape
        else goto GotEscapeInRaisedID >>;
    15:  % string quote
    <<  BackupBuf();
        goto InsideString >>;
    16:  % Package indicator - at start of token means use global package
    <<  ResetBuf();
        ChangedPackages := 1;
        Package 'Global;
        if null LispVar !*Raise then
            goto GotPackageMustGetID
        else goto GotPackageMustGetIDRaised >>;
    17:  % Ignore - can't ever happen
        ScannerError("Internal error - consult a wizard");
    18:  % Minus sign
    <<  TokSign := -1;
        goto GotSign >>;
```

```
    19:  % Plus sign
    <<  TokSign := 1;
        goto GotSign >>;
    20:  % decimal point
    <<  ResetBuf();
        ReadInBuf();
        if ChTokenType >= 10 then
        <<  UnReadLastChar();
            return ScanPossibleDipthong(TokChannel, '!.) >>
        else
        <<  TokSign := 1;
            TokFloatFractionLength := 1;
            goto InsideFloatFraction >> >>;
    default:
        return ScannerError("Unknown token type")
    end;
 .....
```

21.2.7. Memory Access and Address Operations

The operators @ and & (corresponding to GetMem and Loc) may be used to do direct memory operations, similar to * and & in C.

@ may also be used on the LHS of an assignment. Example:

```
WARRAY FOO[10];
WVAR   FEE=&FOO[0];

...
@(fee+2) := @(fee+4) + & foo(5);
...
```

21.2.8. Bit-Field Operation

The Field and PutField operations are used for accessing fields smaller than whole words:

```
PUTFIELD(LOC, BITOFFSET, BITLENGTH, VALUE);
```

and

```
GETFIELD(LOC,BITOFFSET, BITLENGTH);
```

Special cases such as bytes, halfwords, single bits are optimized if

possible.

For example, the following definitions on the DEC-20 are used to define the fields of an item (in file p20c:data-machine.red):

```
% Divide up the 36 bit DEC-20 word:

WConst  TagStartingBit = 0,
        TagBitLength = 18,
        StrictTagStartingBit = 9,
        StrictTagBitLength = 9,
        InfStartingBit = 18,
        InfBitLength = 18,
        GCStartingBit = 0,
        GCBitLength = 9;

% Access to tag (type indicator) of Lisp item in ordinary code

syslsp macro procedure Tag U;
    list('Field, cadr U, '(wconst TagStartingBit), '(wconst TagBitLength));

syslsp macro procedure PutTag U;
    list('PutField, cadr U, '(wconst TagStartingBit),
                            '(wconst TagBitLength), caddr U);

% Access to tag of Lisp item in garbage collector,
%  if GC bits may be in use

syslsp macro procedure StrictTag U;
    list('Field, cadr U, '(wconst StrictTagStartingBit),
                         '(wconst StrictTagBitLength));

syslsp macro procedure PutStrictTag U;
    list('PutField,
                cadr U, '(wconst StrictTagStartingBit),
                        '(wconst StrictTagBitLength), caddr U);

% Access to info field of item (pointer or immediate operand)

syslsp macro procedure Inf U;
    list('Field, cadr U, '(wconst InfStartingBit), '(wconst InfBitLength));

syslsp macro procedure PutInf U;
    list('PutField, cadr U, '(wconst InfStartingBit),
                            '(wconst InfBitLength), caddr U);
```

21.3. Using SYSLISP

Restriction: SYSLISP code is currently ONLY compiled, since it is converted into machine level operations, most of which are dangerous or tricky to use in an interpreted environment.

Note: In SYSLISP mode, we currently execute some commands in the above PARSE/EVAL/PRINT mode, either to load files or select options, but most SYSLISP code is compiled to a file, rather than being immediately interpreted or compiled in-core.

21.3.1. To Compile SYSLISP Code

Use PSL:RLISP, which usually has the Compiler, with SYSLISP extensions, loaded. Alternatively, one may use <psl>syscmp.exe. This is a version of RLISP built upon <PSL>psl.exe with the SYSLISP compiler and data-machine macros loaded.

```
% Turn on SYSLISP mode:

ON SYSLISP; % This is causes the "mode-analysis" to be done
            % Converting some LISP names to SYSLISP names.

% Use SYSLSP as the procedure type.
```

Example:

```
% Small file to access BPS origin and end.
% Starts in LISP mode

Fluid '(NextBPO LastBPO);

NextBPO:=NIL;
LastBPO:=NIL;

On SYSLISP,COMP; % Switch to SYSLISP mode

syslsp procedure BPSize();
 Begin scalar N1,L1;
   If Null LispVar NextBPO then LispVar NextBPO:=GtBPS 0;
   If Null LispVar LastBPO then LispVar LastBPO:=GtWarray 0;
   N1 :=GtBPS(0);
   L1:= GtWarray(0);
   PrintF('" NextBPS=8#%o, used %d,  LastBPS=8#%o, used %d%n",
              N1,   N1-LispVar(NextBPO),     L1,LispVar(LastBPO)-L1);
   LispVar NextBPO:=N1;
   LispVar LastBPO:=L1;
 End;
```

```
BPSize();       % Call the function
```

21.4. SYSLISP Functions

[??? What about overflow in Syslisp arithmetic? ???]

WPlus2 (U:word, V:word): word — open-compiled, expr

WDifference (U:word, V:word): word — open-compiled, expr

WTimes2 (U:word, V:word): word — open-compiled, expr

WQuotient (U:word, V:word): word — open-compiled, expr

WRemainder (U:word, V:word): word — open-compiled, expr

WShift (U:word, V:word): word — open-compiled, expr

WAnd (U:word, V:word): word — open-compiled, expr

WOr (U:word, V:word): word — open-compiled, expr

WXor (U:word, V:word): word — open-compiled, expr

WNot (U:word): word — open-compiled, expr

WEQ (U:word, V:word): boolean — open-compiled, expr

WNEQ (U:word, V:word): boolean — open-compiled, expr

WGreaterP (U:word, V:word): boolean — open-compiled, expr

WLessP (_U_:_word_, _V_:_word_): _boolean_ _open-compiled_, _expr_

WGEQ (_U_:_word_, _V_:_word_): _boolean_ _open-compiled_, _expr_

WLEQ (_U_:_word_, _V_:_word_): _boolean_ _open-compiled_, _expr_

WGetV (_U_:_word_, _V_:_word_): _word_ _open-compiled_, _macro_

WPutV (_U_:_word_, _V_:_word_, _W_:_word_): _word_ _open-compiled_, _macro_

Byte (_U_:_word_, _V_:_word_): _word_ _open-compiled_, _expr_

PutByte (_U_:_word_, _V_:_word_, _W_:_word_): _word_ _open-compiled_, _expr_

21.4.1. W-Arrays

CopyWArray (_NEW_:_w-vector_, _OLD_:_w-vector_, _UPLIM_:_any_): _NEW_:_w-vector_ _expr_

Copy _UPLIM_ + 1 words.

CopyWRDSToFrom (_NEW_:_w-vector_, _OLD_:_any_): _any_ _expr_

Like **CopyWArray** in heap.

CopyWRDS (_S_:_any_): _any_ _expr_

Allocate new WRDS array in heap.

21.5. Remaining SYSLISP Issues

The system should be made less dependent on the assemblers, compilers and loaders of the particular machine it is implemented on. One way to do this is to bring up a very small kernel including a fast loader to load in the rest.

21.5.1. Stand Alone SYSLISP Programs

In principle it works, but we need to clearly define a small set of support functions. Also, need to implement EXTERNAL properly, so that a normal LINKING loader can be used. In PSL, we currently produce a single kernel module, with resident LAP (or later FAP), and it serves as dynamic linking loader for SYSLISP (ala MAIN SAIL).

21.5.2. Need for Two Stacks

We must distinguish between true LISP items and untagged SYSLISP items on the stack for the garbage collector to work properly. Two of the options for this are

1. Put a mark on the stack indicating a region containing untagged items.

2. Use a separate stack for untagged items.

Either of these involves a change in the compiler, since it currently only allocates one frame for temporaries on the stack and does not distinguish where they get put.

The garbage collector should probably be recoded more modularly and at a higher level, short of redesigning the entire storage management scheme. This in itself would probably require the existence of a separate stack which is not traced through for return addresses and SYSLISP temporaries.

21.5.3. New Mode System

A better scheme for intermixing SYSLISP and LISP within a package is needed. Mode Reduce will probably take care of this.

21.5.4. Extend CREF for SYSLISP

The usual range of LISP tools should be available, such as profiling, a break package, tracing, etc.

CHAPTER 22
IMPLEMENTATION

22.1. Overview of the Implementation . . . 22.1
22.2. Files of Interest . . . 22.1
22.3. Building PSL on the DEC-20 . . . 22.2
22.4. Building the LAP to Assembly Translator . . . 22.5
22.5. The Garbage Collectors and Allocators . . . 22.5
  22.5.1. Compacting Garbage Collector on DEC-20 . . . 22.5
  22.5.2. Two-Space Stop and Copy Collector on VAX . . . 22.6
22.6. The HEAPs . . . 22.6
22.7. Allocation Functions . . . 22.8

22.1. Overview of the Implementation

In this Chapter we give a guide to the sources, although they are still rapidly changing. With these notes in mind, and an understanding of SYSLISP and the compiler at the level of Chapters 19 and 21, it is hoped the user will be able to understand and change most of the system. Much of the current information is contained in comments in the source files, and cannot be reproduced here.

[??? This Section needs a LOT of work ???]

22.2. Files of Interest

The complete sources are divided up into a fairly large number of files, spread over a number of sub-directories of <PSL>. This is so that files representing a common machine-independent kernel are in a single directory, and additional machine specific files in others. Furthermore, we have separated the compiler and LAP files from the rest of the files, since they are looked at first when doing a new implementation, but are not actually important to understanding the working of PSL.

Some convenient logical device names are defined in <psl>logical-names.cmd. This file should have been TAKEn in your LOGIN.CMD. Current definitions are:

```
; Officially recognized logical names for PSL subdirectories on UTAH-20
define psl: <psl>                ! Executable files and miscellaneous
define ploc: <psl.local>         ! Non-distributed miscellaneous
define pi: <psl.interp>          ! Interpreter sources
```

```
define pc: <psl.comp>             ! Compiler sources
define pu: <psl.util>             ! Utility program sources
define plocu: <psl.local.util>    ! Non-distributed utility sources
define pd: <psl.doc>              ! Documentation to TYPE
define pe: <psl.emode>            ! Emode sources and build files
define plpt: <psl.lpt>            ! Printer version of Documentation
define ph: <psl.help>             ! Help files
define plap: <psl.lap>            ! LAP and B files
define ploclap: <psl.local.lap>   ! Non-distributed LAP and B files
define pred: <reduce.psl-reduce>! Temporary home of Reduce built upon PSL
define p20: <psl.20-interp>       ! Dec-20 specific interpreter sources
define p20c: <psl.20-comp>        ! Dec-20 specific compiler sources
define p20d: <psl.20-dist>        ! Dec-20 distribution files
define pv: <psl.vax-interp>       ! Vax specific interpreter sources
define pvc: <psl.vax-comp>        ! Vax specific compiler sources
define pvd: <psl.vax-dist>        ! Vax distribution files
define p68: <psl.68000-interp>    ! M68000 specific interpreter sources
define p68c: <psl.68000-comp>     ! M68000 specific compiler sources
define pcr: <psl.cray-interp>     ! Cray-1 interpreter sources
define pcrc: <psl.cray-comp>      ! Cray-1 compiler sources
define pcrd: <psl.cray-dist>      ! Cray-1 distribution files
define pl: plap:,ploclap:         ! Search list for LOAD
```

Sources mostly live on PI:. DEC-20 build files and very machine specific files live on P20:.

22.3. Building PSL on the DEC-20

[??? fix as FASL works ???]

Building proceeds in number of steps. First the kernel files are compiled to MIDAS, using a LAP-to-MIDAS translator, which follows the normal LISP/SYSLISP compilation to LAP. This phase also includes the conversion of constants (atoms names, strings, etc) into structures in the heap, and initialization code into an INIT procedure. The resulting module is assembled, linked, and saved as BARE-PSL.EXE. If executed, it reads in a batch of LAP files, previously compiled, representing those functions that should be in a minimal PSL, but in fact are not needed to implement LAP.

[??? When FAP is implemented, these LAP files will become FAP files, and the kernel will get smaller ???]

•

The BARE-PSL kernel build file is P20:PSL-KERNEL.CTL, and is reproduced here, slightly edited:

```
; This requires PL:PSL-NON-KERNEL.LAP and P20C:PSLDEF.MID
copy BARE-PSL.SYM PSL.SYM
PSL:MIDASCMP                ! previously saved with LAPtoMIDAS
in "PSL-KERNEL.RED";        % Files for kernel
quit;
MIDAS                       ! assemble kernel data
dpsl
MIDAS                       ! assemble kernel init code
spsl
MIDAS                       ! assemble kernel code
psl
load DPSL.REL, SPSL.REL, PSL.REL  ! link into one module
save BARE-PSL.EXE                 ! save executable
```

The kernel files mentioned in PSL-KERNEL.RED are:

```
MIDASOUT "PSL";
IN "BINDING.RED"$                  % binding from the interpreter
IN "FAST-BINDER.RED"$              % for binding in compiled code, in LAP
IN "SYMBOL-VALUES.RED"$            % SET, and support for Eval
IN "FUNCTION-PRIMITIVES.RED"$      % used by PutD, GetD and Eval
IN "OBLIST.RED"$                   % Intern, RemOb and GenSym
IN "CATCH-THROW.RED"$              % non-local GOTO mechanism
IN "ALLOCATORS.RED"$               % heap, symbol and code space alloc
IN "COPIERS.RED"$                  % copying functions
IN "CONS-MKVECT.RED"$              % SL constructor functions
IN "GC.RED"$                       % the garbage collector
IN "APPLY-LAP.RED"$                % low-level function linkage, in LAP
IN "EQUAL.RED"$                    % equality predicates
IN "EVAL-APPLY.RED"$               % interpreter functions
IN "PROPERTY-LIST.RED"$            % PUT and FLAG and friends
IN "FLUID-GLOBAL.RED"$             % variable declarations
IN "PUTD-GETD.RED"$                % function defining functions
IN "KNOWN-TO-COMP-SL.RED"$         % SL functions performed online in code
IN "OTHERS-SL.RED"$                % DIGIT, LITER and LENGTH
IN "CARCDR.RED"$                   % CDDDDR, etc.
IN "EASY-SL.RED"$                  % highly portable SL function defns
IN "EASY-NON-SL.RED"$              % simple, ubiquitous SL extensions
IN "COMP-SUPPORT.RED"$             % optimized CONS and LIST compilation
IN "ERROR-HANDLERS.RED"$           % low level error handlers
IN "TYPE-CONVERSIONS.RED"$         % convert from one type to another
IN "ARITH.RED"$                    % Lisp arithmetic functions
IN "IO-DATA.RED"$                  % Data structures used by IO
IN "SYSTEM-IO.RED"$                % system dependent IO functions
IN "CHAR-IO.RED"$                  % bottom level IO primitives
IN "OPEN-CLOSE.RED"$               % file primitives
IN "RDS-WRS.RED"$                  % IO channel switching functions
IN "OTHER-IO.RED"$                 % random SL IO functions
IN "READ.RED"$                     % S-expression parser
```

```
IN "TOKEN-SCANNER.RED"$          % table-driven token scanner
IN "PRINTERS.RED"$               % Printing functions
IN "WRITE-FLOAT.RED"$            % Floating point printer
IN "PRINTF.RED"$                 % formatted print routines
IN "IO-ERRORS.RED"$              % I/O error handlers
IN "IO-EXTENSIONS.RED"$          % Random non-SL IO functions
IN "VECTORS.RED"$                % GetV, PutV, UpbV
IN "STRING-OPS.RED"$             % Indx, SetIndx, Sub, SetSub, Concat
IN "EXPLODE-COMPRESS.RED"$       % Access to characters of atoms
IN "BACKTRACE.RED"$              % Stack backtrace
IN "DEC-20-EXTRAS.RED"$          % Dec-20 specific routines
IN "LAP.RED"$                    % Compiled code loader
IN "INTERESTING-SYMBOLS.RED"$    % to access important WCONSTs
IN "MAIN-START.RED"$             % first routine called
MIDASEND;
InitSymTab();
END;
```

The current non-kernel files are defined in PSL-NON-KERNEL.RED:

```
LapOut "PL:PSL-NON-KERNEL.LAP";
in "EVAL-WHEN.RED"$              % control evaluation time (load first)
in "CONT-ERROR.RED"$             % macro for ContinuableError
in "MINI-TRACE.RED"$             % simple function tracing
in "TOP-LOOP.RED"$               % generalized top loop function
in "PROG-AND-FRIENDS.RED"$       % Prog, Go and Return
in "ERROR-ERRORSET.RED"$         % most basic error handling
in "TYPE-ERRORS.RED"$            % type mismatch error calls
in "SETS.RED"$                   % Set manipulation functions
in "DSKIN.RED"$                  % Read/Eval/Print from files
in "LISP-MACROS.RED"$            % If, SetF
in "LOOP-MACROS.RED"$            % While, Repeat, ForEach
in "CHAR.RED"$                   % Character constant macro
in "LOAD.RED"$                   % Standard module LAP loader
in "PSL-MAIN.RED"$                       % SaveSystem and Version stuff
LapEnd;
```

The model on the VAX is similar.

The file GLOBAL-DATA.RED is automatically loaded by the compiler in the LAP-to-Assembly phase. It defines most important external symbols.

A symbol table file, PSL.SYM is produced, and is meant to be used to aid in independent recompilation of modules. It records assigned ID numbers, locations of WVARS, WARRAYS, and WSTRINGs, etc. It is not currently used.

The file P20C:DATA-MACHINE.RED defines important macros and constants, allocating fields within a DEC-20 word (the TAGs, etc). It is used only with compiled code, and is so associated with the P20C: (20 compiler specific code); other files on this directory include the code-generator tables and compiler customization files. More information on the compiler and its support can be found in Chapter 19.

22.4. Building the LAP to Assembly Translator

[??? Write after new table-driven LAP and LAP-to-ASM is stable ???]

22.5. The Garbage Collectors and Allocators

22.5.1. Compacting Garbage Collector on DEC-20

DEC-20 PSL uses essentially the same compacting garbage collector developed for the previous MTLISP systems: a single heap with all objects tagged in the heap in such a way that a linear scan from the low end permits objects to be identified; they are either tagged as normal objects, and are thus in a PAIR, or are tagged with a "pseudo-tag", indicating a header item for some sort of BYTE, WORD or ITEM array. Tracing of objects is done using a small stack, and relocation via a segment table and extra bits in the item. The extra bits in the item can be replaced by a bit-table, and this may become the default method.

During compaction, objects are "tamped" to the low end of the heap, permitting "genetic" ordering for algebraic operations, and rapid stack-like allocation.

Since the MTLISP systems included a number of variable sized data-types (e.g. vectors and strings), we had to reduce the working set, and ease the addition of new data-types, by using a single heap with explicitly tagged objects, and compacting garbage collector. In some versions, a bit-table was used both for marking and for compaction. To preserve locality, structures are "tamped" to one end of the heap, maintaining relative (creation time or "Genetic" [Terashima 78]) ordering. The order preservation was rather useful for an inexpensive canonical ordering required in the REDUCE algebra system (simply compare heap positions, which are "naturally" related to object creation). The single heap, with explicit tags made the addition of new data-types rather easy. The virtual memory was implemented as a low level "memory" extension, invisible to the allocator and garbage collector.

This garbage collector has been rewritten a number of times; it is fairly easy to extend, but does waste lot of space in each DEC-20 word. Among

possible alternative allocators/GC is a bit-table version, which is semantically equivalent to that described above but has the Dmov field replaced by a procedure to count ones in a segment of the bit-table. At some point, the separate heap model (tried on Z-80 and PDP-11 MTLISP's) may be implemented, but the separate page-per-type method (BIBOP:="big bag of pages") might also be tried; this permits user definition of new types.

Allocation proceeds as from a stack, permitting rapid allocation, and preserving creation time ordering. The current implementation uses a recursive mark phase with a small stack (G stack) of about 500 entries.

Relocation is accomplished with aid the of the SEGMENT table (overlays G stack), and a small field (Dmov) in each item (header) that gives additional motion of this item relative to the relocation of its segment.

22.5.2. Two-Space Stop and Copy Collector on VAX

Another alternative is a copying, 2-space GC, which is fast and good for large address space (e.g. extended addressing DEC-20 or VAX).

22.6. The HEAPs

The HEAP is used to store variable sized objects. Since one of the possible implementations is to have a separate heap for each of the data types PAIR, STR, CODE, and VECT (or for the groupings PAIR, CODE+STR, VECT), the heap is accessed in type specific fashion only. The current implementation of the allocator and garbage collector maps these type-specific operations onto a single array of item sized blocks, the first of which is a normal tagged item (CAR of a PAIR), or a pseudo-item (header of CODE, STR or VECT). The following blocks are either tagged items or packed bytes. The header item contains a "length" in items, or bytes, as appropriate. Using item sized blocks results in a slight wastage at the end of strings and code-vectors.

Reclamation:

h:=INF(x) For garbage collection, compaction and relocation. The heap is viewed as a set of ITEM sized blocks
PUTINF(x,h)
PUTTYPE(x,t)
MARK(h)
UNMARK(h) Modify the garbage collector mark
MARKED(h) Test the mark (in a bit-table, ITEM header, or ITEM itself).

Other Garbage collector primitives include:

GCPUSH(x) Push an ITEM onto GCSTACK for later trace
x:=GCPOP()
  Retrieve ITEM for tracing
x:=GCTOP()
  Examine top of GCSTACK

The Garbage collector uses a GCSTACK for saving pointers still to be traced. The compaction and relocation takes place by "tamping", without structure reorganization, so that any structure is relocated by the same or more than a neighboring structure, lower in the heap. This "monotonicity" means that the heap can be divided into "segments", and the relocation of any structure computed as the relocation of its segment, plus an additional movement within the segment. The segment table is an additional structure, while the "offset" is computed from the bits in the bit-table, or from a small field (if available) in the ITEM. This garbage collector is similar to that described in [Terashima 78].

RELOC(h):=SEGKNT(SEG(h))+DMOV(h)
  SEGKNT(SEG(h)) is the segment relocation of the segment in which h is, and DMOV is the incremental move within this segment.

i:=SEG(h) Computes the segment number

i:=DSEG(h)
  The "offset" in the segment

Note that DMOV may actually be a small field in an ITEM header, if there is space, or can be computed from the bits in a segment of the BIT-table, or may map to some other construct. The segment table may actually overlay the GCSTACK space, since these are active in different passes of the garbage collection. The garbage collector used in the MTLISP system is an extension of that attributed to S. Brown in [Harrison 73, Harrison 74]. See also [Terashima 78].

!*GC (_Initially_: _NIL_) _flag_

!*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.

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.

Reclaim (): integer expr

User call on GC; does a mark-trace and compaction of HEAP. Returns size of current Heap top. If !*GC is T, prints some statistics. Increments GCKNT!*. Reclaim(); is the user level call to the garbage collector.

!%Reclaim (): expr

!%Reclaim(); is the system level call to the garbage collector. Active data in the heap is made contiguous and all tagged pointers into the heap from active local stack frames, the binding stack and the symbol table are relocated.

22.7. Allocation Functions

GtHEAP (NWRDS:word): word expr

Return address in HEAP of a block of NWRDS item sized pieces. Generates HeapOverflow Message if can't satisfy. GtHeap NIL; returns the number of words (Lisp items) left in the heap. GtHeap 0; returns a pointer to the top of the active heap. GtHeap N; returns a pointer to N words (items).

GtStr (UPLIM:word): word expr

Address of string, 0..UPLIM bytes. (Allocate space for a string UPLIM characters.)

GtConstStr (N:string): expr

(Allocate un-collected string for print name. Same as **GtStr**, but uses BPS, not heap.)

GtWrds (UPLIM:word): word expr

Address of WRD, 0..UPLIM WORDS. (Allocate space for UPLIM untraced words.)

GtVect (UPLIM:word): word expr

Address of vector, UPLIM items. (Allocate space for a vector UPLIM items.)

GtFixN (): *s-integer* *expr*

Allocate space for a fixnum.

GtFltN (): *s-integer* *expr*

Allocate space for a *float*.

GtID (): *id* *expr*

Allocate a new *id*.

GtBps (*N*:*s-integer*): *s-integer* *expr*

Allocate *N* words for binary code.

GtWArray (*N*:*s-integer*): *s-integer* *expr*

Allocate *N* words for WVar/WArray/WString.

DelBps (): *expr*

DelWArray (): *expr*

GtBps NIL; returns the number of words left in BPS. **GtWArray** NIL returns the same quantity.

GtBps 0; returns a pointer to the bottom of BPS, that is, the current value of NextBPS. **GtWArray** 0; returns a pointer to the top of BPS, the current value of LastBPS. This is sometimes convenient for use with **DelBps** and **DelWArray**.

GtBps N; returns a pointer to N words in BPS, moving NextBPS up by that amount. **GtWArray** returns a pointer to (the bottom of) N words at the top of BPS, pushing LastBPS down by that amount. Remember that the arguments are number of WORDS to allocate, that is, 1/4 the number of bytes on the VAX or 68000.

DelBps(Lo, Hi) returns a block to BPS, if it is contiguous with the current free space. In other words, if Hi is equal to NextBPS, then NextBPS is set to Lo. Otherwise, NIL is returned and no space is added to BPS. **DelHeap**(Lo, Hi) is similar in action to **DelBps**.

DelWArray(Lo, Hi) returns a block to the top of BPS, if it is contiguous with the current free space. In other words, if Lo is equal to LastBPS, then LastBPS is set to Hi. Otherwise, NIL is returned and no space is added to BPS.

The storage management routines above are intended for either very long term or very short term use. BPS is not examined by the garbage collector at all. The routines below should be used with great care, as they deal with the heap which must be kept in a consistent state for the garbage collector. All blocks of memory allocated in the heap must have header words describing the size and type of data contained, and all pointers into the heap must have type tags consistent with the data they refer to.

CHAPTER 23
THE EXTENSIBLE RLISP PARSER AND THE MINI PARSER GENERATOR

23.1. Introduction 23.1
23.2. The Table Driven Parser 23.2
  23.2.1. Flow Diagram for the Parser 23.2
  23.2.2. Associating the Infix Operator with a Function 23.4
  23.2.3. Precedences 23.5
  23.2.4. Special Cases of 0 <-0 and 0 0 23.5
  23.2.5. Parenthesized Expressions 23.5
  23.2.6. Binary Operators in General 23.6
  23.2.7. Assigning Precedences to Key Words 23.7
  23.2.8. Error Handling 23.7
  23.2.9. The Parser Program for the RLISP Language 23.7
  23.2.10. Defining Operators 23.8
23.3. The MINI Translator Writing System 23.10
  23.3.1. A Brief Guide to MINI 23.10
  23.3.2. Pattern Matching Rules 23.12
  23.3.3. A Small Example 23.12
  23.3.4. Loading Mini 23.13
  23.3.5. Running Mini 23.13
  23.3.6. MINI Error messages and Error Recovery 23.13
  23.3.7. MINI Self-Definition 23.13
  23.3.8. The Construction of MINI 23.15
  23.3.9. History of MINI Development 23.16
23.4. BNF Description of RLISP Using MINI 23.17

23.1. 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 programer: an extensible table-driven parser that is used for the RLISP parser (described in Chapter 3), 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.

23.2. 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 Chapter 13, and performs some additional checks, described below. Each token is defined to be one of the following:

non-operator	0
right operator	0->
binary operator	<-0->

All combinations of . . .0-> 0. . . and 0 <-0->. . . are supposed to be legal, while the combinations . . .0-> <-0->. . ., . . .<-0-> <-0->. . . and 0 0. . . 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).

23.2.1. 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:

```
. . . O-> Y <-O . . .
        rp lp
```

Y is a token or an already parsed expression between two operators (as indicated). If O->'s RP is greater than <-O's LP, then O-> is the winner and Y goes to O->'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.

23.2.2. 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 "!$). This controls the RLISP echo/no-echo capability. Finally, if the renamed token is 'COMMENT then characters are **ReadCh**'d until a '!; or '!$.

23.2.3. 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).

23.2.4. 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.

23.2.5. 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.

23.2.6. 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.

23.2.7. 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.

23.2.8. 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.

23.2.9. 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 13, 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.

23.2.10. 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)>>;
```

23.3. The MINI Translator Writing System

Note that MINI is now autoloading.

23.3.1. 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.

23.3.2. 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.

23.3.3. 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>>;
```

23.3.4. Loading Mini

MINI is loaded from PH: using LOAD MINI;.

23.3.5. 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.

23.3.6. 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 ???]

23.3.7. 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
```

23.3.8. 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.

23.3.9. 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.

23.4. 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.

[??? 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
```

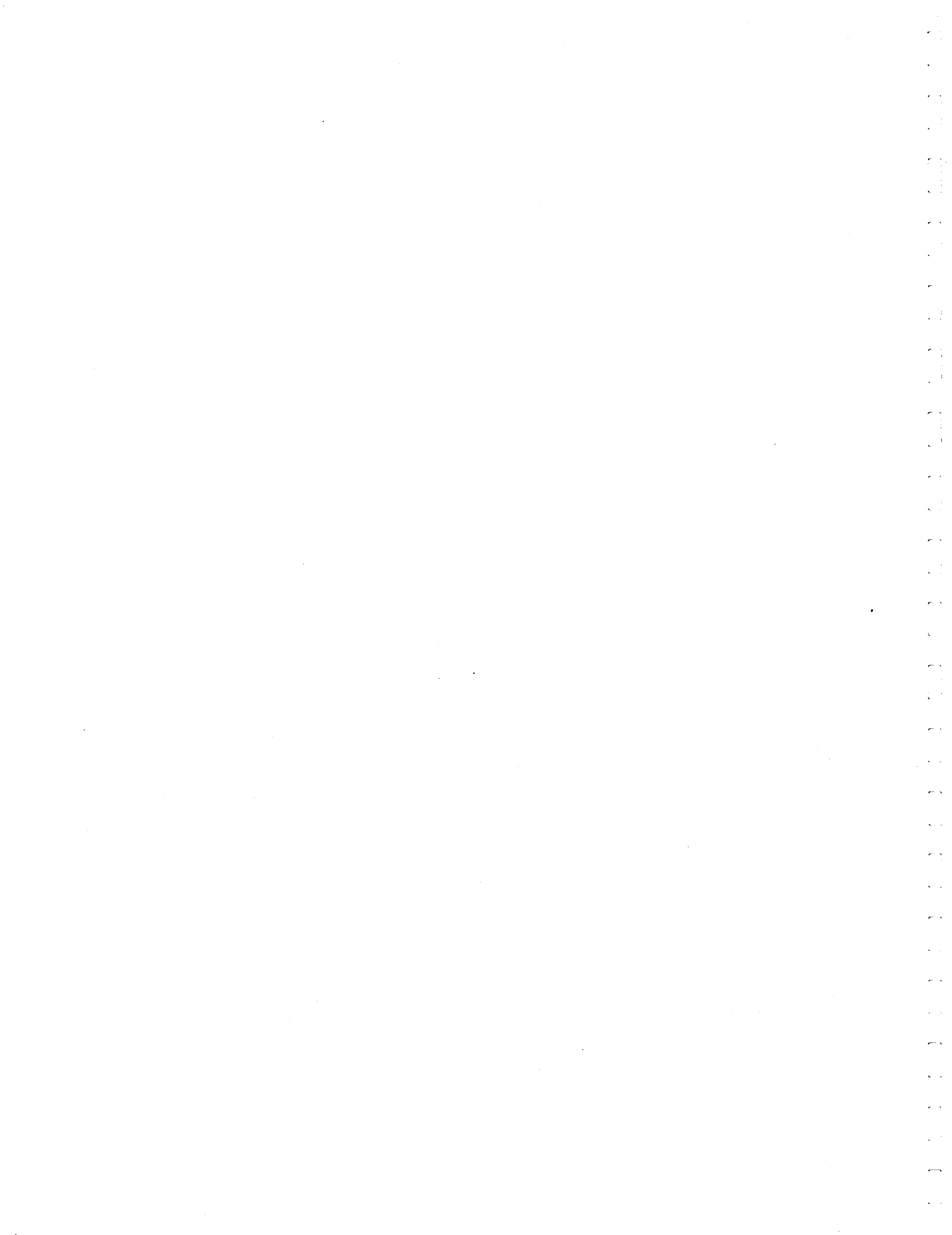
```
         'macro .(setq FTYPE!* 'macro) /
         'smacro .(setq FTYPE!* 'smacro) /
         'nmacro .(setq FTYPE!* 'nmacro) /
         ('expr / =T) .(setq FTYPE!* 'expr) ;


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) flaglists ;

flaglists: 'defn +(set '!*defn #1) ;

timop: ('* +times / '/ +quotient) ;

plusop: ('+ +plus2 / '- +difference) ;

relop: ('< +lessp / '<= +lep / '= +equal /
           '>= +gep / '> +greaterp) ;
```

FIN



CHAPTER 24
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.

[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 No. 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 25
INDEX OF FUNCTIONS, GLOBALS, AND FLAGS

The following is an alphabetical list of the PSL functions and global variables, with the page on which they are defined.

!$BREAK!$ . . . . global 15.7
!$EOF!$ . . . . global 13.17
!$ERROR!$ . . . . global 15.1, 15.2
!%Reclaim . . . . expr 22.8
!*BACKTRACE . . . . flag 15.1, 15.2
!*BREAK . . . . flag 15.3, 15.6
!*BTR . . . . flag 16.17
!*BTRSAVE . . . . flag 16.12, 16.17
!*COMP. . . . . flag 10.3, 19.2
!*COMPRESSING . . . . flag 13.6, 13.9, 13.17
!*CREFSUMMARY . . . . flag 18.3
!*DEFN. . . . . flag 19.3
!*DESTROY . . . . cmacro 19.21
!*DO. . . . . cmacro 19.21
!*EOLINSTRINGOK . . . . flag 13.9
!*ERFG. . . . . flag 19.23
!*GC. . . . . flag 22.7
!*INSTALL . . . . flag 16.11, 16.15
!*INSTALLDESTROY. . . . . flag 19.23
!*INT . . . . flag 19.23
!*JUMP. . . . . cmacro 19.21
!*LBL . . . . cmacro 19.21
!*LOAD. . . . . cmacro 19.21
!*MODULE. . . . . flag 19.6
!*MSGP. . . . . flag 15.1, 15.2
!*NOFRAMEFLUID. . . . . flag 19.23
!*NOLINKE . . . . flag 19.5
!*NOTRARGS. . . . . flag 16.5
!*ORD . . . . flag 19.6
!*PECHO . . . . flag 14.3
!*PGWD. . . . . flag 19.12
!*PLAP. . . . . flag 19.6, 19.12
!*PVAL. . . . . flag 14.3
!*PWRDS . . . . flag 19.6, 19.12
!*R2I . . . . flag 19.5
!*RAISE . . . . flag 13.7, 13.9
!*REDEFMSG. . . . . flag 10.2
!*SAVECOM . . . . flag 19.12
!*SAVEDEF . . . . flag 19.12
!*SAVENAMES . . . . flag 16.17
!*SET . . . . cmacro 19.21
!*SHOWDEST. . . . . flag 19.23
!*STORE . . . . cmacro 19.21
!*SYSLISP . . . . flag 19.23
!*TIME. . . . . flag 14.3

!*TRACE . . . . flag 16.17
!*TRACEALL. . . . . flag 16.11, 16.15
!*TRCOUNT . . . . flag 16.18
!*TRUNKNOWN . . . . flag 16.17
!*UNSAFEBINDER. . . . . flag 19.23
!*USEREGFLUID . . . . flag 19.23
!*USERMODE. . . . . flag 10.3

\CreatePackage. . . . . expr 6.9
\CURRENTPACKAGE!* . . . . global 6.8
\LocalIntern. . . . . expr 6.10
\LocalInternP . . . . expr 6.10
\LocalMapObl. . . . . expr 6.10
\LocalRemob . . . . expr 6.10
\PACKAGENAMES!* . . . . global 6.9
\PathIntern . . . . expr 6.9
\PathInternP. . . . . expr 6.9
\PathMapObl . . . . expr 6.10
\PathRemob. . . . . expr 6.10
\SetPackage . . . . expr 6.9

A . . . . edit 17.8
Abs . . . . expr 5.2
AConc . . . . expr 7.7
Acos. . . . . expr 18.28
AcosD . . . . expr 18.28
Acot. . . . . expr 18.28
AcotD . . . . expr 18.28
Acsc. . . . . expr 18.28
AcscD . . . . expr 18.29
Add1. . . . . expr 5.2
Adjoin. . . . . expr 7.7
AdjoinQ . . . . expr 7.7
AlphaNumericP . . . . expr 8.8
AlphaP. . . . . expr 8.7
And . . . . fexpr 4.9
Ans . . . . expr 14.4
AnyId . . . . expr 23.11
AnyTok. . . . . expr 23.11
Append. . . . . expr 7.6
Apply . . . . expr 11.2
ApplyInEnvironment. . . . . expr 10.10
Asec. . . . . expr 18.28
AsecD . . . . expr 18.29
Asin. . . . . expr 18.28
AsinD . . . . expr 18.28
Ass . . . . expr 7.12
Assoc . . . . expr 7.11
Atan. . . . . expr 18.28
AtanD . . . . expr 18.28
Atom. . . . . expr 4.7
AtomSortFn. . . . . expr 7.10

Atsoc . . . . expr 7.12

B . . . . edit 17.3, 17.8
BackQuote . . . . macro 18.12
BELOW . . . . edit 17.8
BF. . . . . edit 17.8
BI. . . . . edit 17.9
BIND. . . . . edit 17.9
Bits. . . . . macro 20.9
BK. . . . . edit 17.10
BldMsg. . . . . expr 13.20
BO. . . . . edit 17.10
BothCaseP . . . . expr 8.7
BothTimes . . . . expr 19.4
Br. . . . . fexpr 16.3
BREAKEVALUATOR!*. . . . . global 15.4
BREAKIN!* . . . . global 15.6, 15.7
BREAKOUT!*. . . . . global 15.6, 15.7
BREAKPRINTER!*. . . . . global 15.4
BREAKREADER!* . . . . global 15.4
Btr . . . . expr 16.12
Bug . . . . expr 2.8
Byte. . . . . expr 21.11

CaptureEnvironment. . . . . . . expr 10.10
Car . . . . expr 7.3
Case. . . . . fexpr 9.4
Catch . . . . expr 9.16
Cdr . . . . expr 7.3
Ceiling . . . . expr 18.25
CHANGE. . . . . edit 17.10
ChannelPrin1. . . . . expr 13.13
ChannelPrin2. . . . . expr 13.14
ChannelRead . . . . expr 13.17
ChannelReadChar . . . . expr 13.5
ChannelReadToken. . . . . expr 13.6
ChannelUnReadChar . . . . expr 13.5
ChannelWriteChar. . . . . expr 13.13
Char!-Bits. . . . . expr 8.8
Char!-Code. . . . . expr 8.8
Char!-DownCase. . . . . expr 8.9
Char!-Equal . . . . expr 8.8
Char!-Font. . . . . expr 8.9
Char!-GreaterP. . . . . expr 8.8
Char!-Int . . . . expr 8.9
Char!-LessP . . . . expr 8.8
Char!-UpCase. . . . . expr 8.9
Char!<. . . . . expr 8.8
Char!=. . . . . expr 8.8
Char!>. . . . . expr 8.8
Char. . . . . macro 21.5
Character . . . . expr 8.9

CheckedArcCosine. . . . . expr 18.28
CheckedArcTangent . . . . expr 18.28
CheckedLogarithm. . . . . expr 18.29
ClearBindings . . . . expr 10.10
Close . . . . expr 13.4
Closure . . . . macro 10.9
Cmds. . . . . fexpr 20.2
Code!-Char. . . . . expr 8.9
CodeP . . . . expr 4.7
CommentOutCode. . . . . macro 19.3
Compile . . . . expr 19.2
CompileTime . . . . expr 19.3
Compress. . . . . expr 13.19
COMS. . . . . edit 17.10
COMSQ . . . . edit 17.10
Concat. . . . . expr 8.6
ConcatS . . . . expr 20.2
Cond. . . . . fexpr 9.1
Cons. . . . . expr 7.2
Const . . . . macro 18.21
ConstantP . . . . expr 4.7
ContError . . . . macro 15.3
ContinuableError. . . . . expr 15.3
Copy. . . . . expr 7.3
CopyD . . . . expr 10.3
CopyScanTable . . . . expr 13.11
CopyString. . . . . expr 8.2
CopyStringToFrom. . . . . expr 8.2
CopyVector. . . . . expr 8.4
CopyVectorToFrom. . . . . expr 8.3
CopyWArray. . . . . expr 21.11
CopyWRDS. . . . . expr 21.11
CopyWRDSToFrom. . . . . expr 21.11
Cos . . . . expr 18.26
CosD. . . . . expr 18.27
Cot . . . . expr 18.27
CotD. . . . . expr 18.27
CPrint. . . . . expr 18.24
CRLF. . . . . global 20.2
Csc . . . . expr 18.27
CscD. . . . . expr 18.28
CURRENTSCANTABLE!*. . . . . . . global 13.7, 13.10, 13.11, 13.18

De. . . . . macro 10.4
Decr. . . . . macro 5.3
DefConst. . . . . macro 18.21
DefLambda . . . . macro 18.13
DefList . . . . expr 6.4
DefMacro. . . . . macro 18.11
Defstruct . . . . fexpr 18.15
DefstructP. . . . . expr 18.15
DefstructType . . . . expr 18.15

DegreesToRadians. . . . . expr 18.26
Del . . . . expr 7.8
DelAsc. . . . . expr 7.9
DelAscIP. . . . . expr 7.9
DelatQ. . . . . expr 7.9
DelatQIP. . . . . expr 7.9
DelBps. . . . . expr 22.9
DELETE. . . . . edit 17.10
Delete. . . . . expr 7.8
DeletIP . . . . expr 7.8
DelQ. . . . . expr 7.9
DelQIP. . . . . expr 7.9
DelWArray . . . . expr 22.9
DeSetQ. . . . . macro 6.6
Df. . . . . macro 10.4
DFPRINT!* . . . . global 19.3
Difference. . . . . expr 5.3
Digit!-Char . . . . expr 8.9
Digit . . . . expr 13.18
DigitP. . . . . expr 8.8
Divide. . . . . expr 5.3
Dm. . . . . macro 10.4
DMSToRadians. . . . . expr 18.26
Dn. . . . . macro 10.4
Do!*. . . . . macro 9.15
Do-Loop!* . . . . macro 9.15
Do-Loop . . . . macro 9.15
Do. . . . . macro 9.14
DoCmds. . . . . expr 20.2
Ds. . . . . macro 10.5
DskIn . . . . fexpr 13.12
DumpLisp. . . . . expr 14.2

E . . . . edit 17.11
EditF . . . . expr 17.11
EditFns . . . . fexpr 17.11
EditP . . . . fexpr 17.11
EditV . . . . fexpr 17.11
Eject . . . . expr 13.16
Emacs . . . . expr 20.3
EMBED . . . . edit 17.11
EMSG!*. . . . . global 15.2
Eq. . . . . expr 4.5
EqCar . . . . expr 4.7
EqN . . . . expr 4.6
EqStr . . . . expr 4.7
Equal . . . . expr 4.6
Error . . . . expr 15.2
ERRORFORM!* . . . . global 15.3, 15.4
ERRORHANDLERS!* . . . . global 15.8
ErrorPrintF . . . . expr 13.15
ErrorSet. . . . . expr 15.2

ERROUT!*. . . . . global 13.3, 13.15
ErrPrin . . . . expr 13.14
Eval. . . . . expr 11.1
EvalInEnvironment . . . . . . expr 10.10
EvIn. . . . . expr 13.12
EvLis . . . . expr 11.3
EvProgN . . . . expr 11.3
Exec. . . . . expr 20.3
Exit. . . . . macro 9.7
Exp . . . . expr 18.29
Expand. . . . . expr 11.4
Explode2. . . . . expr 13.19
Explode . . . . expr 13.19
ExprP . . . . expr 10.6
Expt. . . . . expr 5.3
Extended-Get. . . . . expr 18.23
Extended-Put. . . . . expr 18.23
EXTRACT . . . . edit 17.12

F=. . . . . edit 17.14
F . . . . edit 17.2, 17.12
FaslEnd . . . . expr 19.2
FaslIn. . . . . expr 19.3
FaslOut . . . . expr 19.2
FCodeP. . . . . expr 10.5
FExprP. . . . . expr 10.6
FileP . . . . expr 13.4, 20.5
FindPrefix. . . . . expr 18.22
FindSuffix. . . . . expr 18.22
First . . . . macro 7.4
Fix . . . . expr 5.2
FixP. . . . . expr 4.7
Flag1 . . . . expr 6.5
Flag. . . . . expr 6.4
FlagP . . . . expr 6.5
FLambdaLinkP. . . . . expr 10.5
FlatSize2 . . . . expr 13.20
FlatSize. . . . . expr 13.19
Float . . . . expr 5.2
FloatP. . . . . expr 4.7
Fluid . . . . expr 10.7, 19.5
FluidP. . . . . expr 10.8
For!* . . . . macro 9.12
For . . . . macro 9.8
ForEach . . . . macro 9.12
Fourth. . . . . macro 7.5
FS. . . . . edit 17.13
FStub . . . . expr 16.13
FUnBoundP . . . . expr 10.5
Function. . . . . fexpr 11.4

GCKNT!* . . . . global 22.7

GenSym. . . . . expr 6.3
Geq . . . . macro 5.5
Get . . . . expr 6.3
GetCDir . . . . expr 20.6
GetD. . . . . expr 10.3
GetFCodePointer . . . . expr 10.6
GetFork . . . . expr 20.4
GetNewJfn . . . . expr 20.5
GetOldJfn . . . . expr 20.5
GetRescan . . . . expr 20.5
GetUName. . . . . expr 20.6
GetV. . . . . expr 8.2
Global. . . . . expr 10.7, 19.5
GlobalP . . . . expr 10.8
GMergeSort. . . . . expr 7.11
Go. . . . . fexpr 9.5
Graph-to-Tree . . . . expr 18.24
GraphicP. . . . . expr 8.7
GreaterP. . . . . expr 5.5
GSort . . . . expr 7.11
GSortP. . . . . expr 7.11
GtBps . . . . expr 22.9
GtConstStr. . . . . expr 22.8
GtFixN. . . . . expr 22.9
GtFltN. . . . . expr 22.9
GtHEAP. . . . . expr 22.8
GtID. . . . . expr 22.9
GtJfn . . . . expr 20.6
GtStr . . . . expr 22.8
GtVect. . . . . expr 22.8
GtWArray. . . . . expr 22.9
GtWrds. . . . . expr 22.8

HAppend . . . . expr 18.23
HCons . . . . macro 18.23
HCopy . . . . macro 18.23
HELP. . . . . edit 17.3, 17.14
Help. . . . . fexpr 14.5
HelpDir . . . . expr 20.3
HelpIn!*. . . . . global 14.5
HelpOut!* . . . . global 14.6
HighHalfWord. . . . . expr 20.8
Hist. . . . . nexpr 14.4
HList . . . . nexpr 18.23
HReverse. . . . . expr 18.23

I . . . . edit 17.14
Id2Int. . . . . expr 4.10
Id2String . . . . expr 4.10
Id. . . . . expr 23.10
IdP . . . . expr 4.7
IdSortFn. . . . . expr 7.10

IF. . . . . edit 17.14
If. . . . . macro 9.2
If_System . . . . cmacro 20.1
IGetV . . . . expr 8.4
Implode . . . . expr 13.19
Imports . . . . expr 19.3
IN!*. . . . . global 13.2, 13.4, 13.18
In. . . . . macro 13.12
Incr. . . . . macro 5.3
Indx. . . . . expr 8.5
InFile. . . . . fexpr 20.6
Inp . . . . expr 14.4
INSERT. . . . . edit 17.14
Inspect . . . . expr 18.24
Int!-Char . . . . expr 8.9
Int2Id. . . . . expr 4.10
Int2Str . . . . expr 20.8
Int2Sys . . . . expr 4.10
Intern. . . . . expr 4.9
InternP . . . . expr 6.3
InterpBackTrace . . . . expr 15.5
InterSection. . . . . expr 7.8
InterSectionQ . . . . expr 7.8
IPutV . . . . expr 8.4
ISizeS. . . . . expr 8.4
ISizeV. . . . . expr 8.4

JBits . . . . expr 20.9
JConv . . . . expr 20.7
Jsys0 . . . . expr 20.7
Jsys1 . . . . expr 20.7
Jsys2 . . . . expr 20.7
Jsys3 . . . . expr 20.7
Jsys4 . . . . expr 20.7

KillFork. . . . . expr 20.4

LAnd. . . . . expr 5.7
LAP . . . . expr 19.9
LapIn . . . . expr 13.12
LASTACTUALREG . . . . global 19.23
LastCar . . . . expr 7.5
LastPair. . . . . expr 7.5
LBind1. . . . . expr 10.9
LC. . . . . edit 17.14
LCL . . . . edit 17.15
LConc . . . . expr 7.7
Length. . . . . expr 7.6
Leq . . . . macro 5.5
LessP . . . . expr 5.5
Let!* . . . . macro 9.16
Let . . . . macro 9.15

LI. . . . . edit 17.15
LineLength. . . . . expr 13.16
LispEqual . . . . expr 4.6
LISPSCANTABLE!* . . . . global 13.7, 13.10, 13.18
List2Set. . . . . expr 7.9
List2SetQ . . . . expr 7.9
List2String . . . . expr 4.10
List2Vector . . . . expr 4.11
List. . . . . fexpr 7.6
Liter . . . . expr 13.19
LNot. . . . . expr 5.7
LO. . . . . edit 17.15
Load. . . . . macro 19.3
LoadTime. . . . . expr 19.4
Log10 . . . . expr 18.29
Log2. . . . . expr 18.29
Log . . . . expr 18.29
LOr . . . . expr 5.7
LowerCaseP. . . . . expr 8.7
LowHalfWord . . . . expr 20.8
LP. . . . . edit 17.16
LPosn . . . . expr 13.16
LPQ . . . . edit 17.16
LShift. . . . . expr 5.7
LXOr. . . . . expr 5.7

M . . . . edit 17.16
MacroExpand . . . . macro 18.13
MacroP. . . . . expr 10.6
Main. . . . . expr 14.2
Make!-Bytes . . . . expr 8.4
Make!-Halfwords . . . . expr 8.4
Make!-String. . . . . expr 8.2, 8.11
Make!-Vector. . . . . expr 8.3
Make!-Words . . . . expr 8.4
MakeFCode . . . . expr 10.6
MakeFLambdaLink . . . . expr 10.5
MAKEFN. . . . . edit 17.17
MakeFUnBound. . . . . expr 10.5
MakeUnBound . . . . expr 6.8
Map . . . . expr 9.13
MapC. . . . . expr 9.13
MapCan. . . . . expr 9.13
MapCar. . . . . expr 9.13
MapCon. . . . . expr 9.13
MapList . . . . expr 9.14
MapObl. . . . . expr 6.4
MARK. . . . . edit 17.17
Max2. . . . . expr 5.6
Max . . . . macro 5.6
MAXLEVEL. . . . . global 17.13
MAXNARGS. . . . . global 19.24

MBD . . . . edit 17.18
Member. . . . . expr 7.6
MemQ. . . . . expr 7.6
Min2. . . . . expr 5.6
Min . . . . macro 5.6
Minus . . . . expr 5.4
MinusP. . . . . expr 5.6
MkQuote . . . . expr 11.4
MkString. . . . . expr 8.2
MkVect. . . . . expr 8.3
MM. . . . . expr 20.4
MOVE. . . . . edit 17.18

N . . . . edit 17.19
NameFromJfn . . . . expr 20.6
NConc . . . . expr 7.7
NCons . . . . expr 7.3
Ne. . . . . expr 4.6
Neq . . . . macro 4.6
NewId . . . . expr 4.9
NewTrBuff . . . . expr 16.7
NEX . . . . edit 17.19
NExprP. . . . . expr 10.6
Next. . . . . macro 9.7
NIL . . . . global 12.4
NOLIST!*. . . . . global 18.3
Not . . . . expr 4.8
NString!-Capitalize . . . . expr 8.12
NString!-DownCase . . . . expr 8.11
NString!-UpCase . . . . expr 8.11
NTH . . . . edit 17.19
Nth . . . . expr 7.5
Null. . . . . expr 4.8
Num . . . . expr 23.11
NumberP . . . . expr 4.8
NumberSortFn. . . . . expr 7.10
NX. . . . . edit 17.20

Off . . . . macro 12.2
OK. . . . . edit 17.3, 17.20
On. . . . . macro 12.2
OneP. . . . . expr 5.6
Open. . . . . expr 13.3
OpenFork. . . . . expr 20.4
OpenNewJfn. . . . . expr 20.5
OpenOldJfn. . . . . expr 20.5
OPTIONS!* . . . . global 19.3
Or. . . . . fexpr 4.9
ORF . . . . edit 17.20
ORR . . . . edit 17.21
OUT!* . . . . global 13.2, 13.4
Out . . . . expr 13.12

OUTPUTBASE!*. . . . . global 13.8, 13.10

P . . . . edit 17.1, 17.21
Pair. . . . . expr 7.12
PairP . . . . expr 4.8
PBind1. . . . . expr 10.9
PL. . . . . edit 17.1
PLEVEL. . . . . global 17.1
PList . . . . expr 16.14
Plus2 . . . . expr 5.4
Plus. . . . . macro 5.4
PNth. . . . . expr 7.5
Pop . . . . macro 18.14
Posn. . . . . expr 13.16
PP. . . . . edit 17.22
Ppf . . . . expr 16.14
PPFPRINTER!*. . . . . global 16.16
PrettyPrint . . . . expr 13.16
Prin1 . . . . expr 13.14
Prin2 . . . . expr 13.14
Prin2L. . . . . expr 13.16
Prin2T. . . . . expr 13.17
Print . . . . expr 13.14
PrintF. . . . . expr 13.14
PrintScanTable. . . . . expr 13.11
PrintX. . . . . expr 16.14
Prog1 . . . . macro 9.4
Prog2 . . . . expr 9.4
Prog. . . . . fexpr 9.5
ProgN . . . . fexpr 9.4
PROMPTSTRING!*. . . . . global 13.3
Prop. . . . . expr 6.5
PROPERTYPRINTER!* . . . . global 16.16
PSetF . . . . macro 6.8
PSetQ . . . . macro 6.6
Push. . . . . macro 18.14
Put . . . . expr 6.3
PutByte . . . . expr 21.11
PutD. . . . . expr 10.2
PUTDHOOK!*. . . . . global 16.15
PutDipthong . . . . expr 13.11
PutReadMacro. . . . . expr 13.11
PutRescan . . . . expr 20.5
PutV. . . . . expr 8.3

Quit. . . . . expr 14.1
Quote . . . . fexpr 11.3
Quotient. . . . . expr 5.4

R . . . . edit 17.2, 17.23
RadiansToDegrees. . . . . expr 18.26
RadiansToDMS. . . . . expr 18.26

Random. . . . . expr 18.29
RAtom . . . . expr 13.9
Rds . . . . expr 13.4
Read. . . . . expr 13.18
ReadCH. . . . . expr 13.5
ReadChar. . . . . expr 13.5
Recip . . . . expr 5.4
Reclaim . . . . expr 22.8
RecopyStringToNULL. . . . . expr 20.8
ReDo. . . . . macro 14.4
RelJfn. . . . . expr 20.5
Remainder . . . . expr 5.4
RemD. . . . . expr 10.4
RemFlag1. . . . . expr 6.5
RemFlag . . . . expr 6.5
RemOb . . . . expr 6.3
RemProp . . . . expr 6.4
RemPropL. . . . . expr 6.4
REPACK. . . . . edit 17.23
Repeat. . . . . macro 9.7
ResBtr. . . . . expr 16.12
Reset . . . . expr 14.2, 20.4
Rest. . . . . macro 7.5
RestoreEnvironment. . . . . expr 10.10
Restr . . . . expr 16.10
Return. . . . . expr 9.6
Reverse . . . . expr 7.9
ReversIP. . . . . expr 7.10
RI. . . . . edit 17.23
RLisp . . . . expr 14.5
RLISPSCANTABLE!*. . . . . global 13.7, 13.10
RO. . . . . edit 17.24
Round . . . . expr 18.26
RplacA. . . . . expr 7.4
RplacD. . . . . expr 7.4
RplaChar. . . . . expr 8.9
RplacW. . . . . expr 7.4
RPrint. . . . . expr 13.16
Run . . . . expr 20.3
RunFork . . . . expr 20.4

S . . . . edit 17.24
SAssoc. . . . . expr 7.12
SAVE. . . . . edit 17.24
SaveSystem. . . . . expr 14.1
ScaledCosine. . . . . expr 18.26
ScaledCotangent . . . . expr 18.27
ScaledSine. . . . . expr 18.26
ScaledTangent . . . . expr 18.27
Sec . . . . expr 18.27
SecD. . . . . expr 18.27
SECOND. . . . . edit 17.24

Second. . . . . macro 7.5
Set . . . . expr 6.6
SetF. . . . . macro 6.7
SetIndx . . . . expr 8.5
SetProp . . . . expr 6.5
SetQ. . . . . fexpr 6.6
SetSub. . . . . expr 8.5
SetSubSeq . . . . expr 8.6
Shut. . . . . expr 13.13
Sin . . . . expr 18.26
SinD. . . . . expr 18.27
Size. . . . . expr 8.5
Spaces. . . . . expr 13.16
SPECIALCLOSEFUNCTION!*. . . . . global 13.3, 13.4
SPECIALRDSACTION!*. . . . . . . global 13.4
SPECIALREADFUNCTION!* . . . . . global 13.3, 13.5
SPECIALWRITEFUNCTION!*. . . . . global 13.3, 13.5
SPECIALWRSACTION!*. . . . . . . global 13.4, 13.5
Sqrt. . . . . expr 18.29
Standard!-CharP . . . . expr 8.7
StandardLisp. . . . . expr 14.4
StartFork . . . . expr 20.4
STDIN!* . . . . global 13.2, 13.3, 13.4
STDOUT!*. . . . . global 13.2, 13.3, 13.4
StdTrace. . . . . expr 16.8
Step. . . . . expr 16.3
STOP. . . . . edit 17.24
Str2Int . . . . expr 20.8
Str . . . . expr 23.11
String!-Capitalize. . . . . expr 8.12
String!-CharP . . . . expr 8.7
String!-DownCase. . . . . expr 8.11
String!-Equal . . . . expr 8.10
String!-GreaterP. . . . . expr 8.10
String!-Left!-Trim. . . . . expr 8.11
String!-Length. . . . . expr 8.12
String!-LessP . . . . expr 8.10
String!-Not!-Equal. . . . . expr 8.11
String!-Not!-GreaterP . . . . . . expr 8.10
String!-Not!-LessP. . . . . expr 8.10
String!-Repeat. . . . . expr 8.11
String!-Right!-Trim . . . . expr 8.11
String!-to!-List. . . . . expr 8.12
String!-to!-Vector. . . . . expr 8.12
String!-Trim. . . . . expr 8.11
String!-UpCase. . . . . expr 8.11
String!<!=. . . . . expr 8.10
String!<!>. . . . . expr 8.10
String!<. . . . . expr 8.10
String!=. . . . . expr 8.9
String!>!=. . . . . expr 8.10
String!>. . . . . expr 8.10

String2List . . . . expr 4.10
String2Vector . . . . expr 4.11
String. . . . . nexpr 4.11, 8.2
StringGensym. . . . . expr 6.3
StringP . . . . expr 4.8
StringSortFn. . . . . expr 7.10
Stub. . . . . expr 16.13
STUBPRINTER!* . . . . global 16.16
STUBREADER!*. . . . . global 16.16
Sub1. . . . . expr 5.5
Sub . . . . expr 8.5
SublA . . . . expr 7.13
SubLis. . . . . expr 7.13
SubSeq. . . . . expr 8.5
Subst . . . . expr 7.13
SubstIP . . . . expr 7.13
SubString . . . . expr 8.12
SubTypeP. . . . . expr 18.15
SW. . . . . edit 17.25
Swap. . . . . expr 20.8
Sys2Int . . . . expr 4.10
Sys . . . . expr 20.3

T . . . . edit 17.2
T . . . . global 12.4
Tab . . . . expr 13.17
Take. . . . . expr 20.3
Tan . . . . expr 18.26
TanD. . . . . expr 18.27
TConc . . . . expr 7.7
TerPri. . . . . expr 13.16
TEST. . . . . edit 17.25
THIRD . . . . edit 17.25
Third . . . . macro 7.5
THROUGH . . . . edit 17.25
Throw . . . . expr 9.16
Times2. . . . . expr 5.5
Times . . . . macro 5.5
TO. . . . . edit 17.25
TOKTYPE!* . . . . global 13.6, 13.10, 13.18
TopLoop . . . . expr 14.3
TopLoopEval!* . . . . global 14.3, 15.7
TopLoopPrint!*. . . . . global 14.3, 15.7
TopLoopRead!* . . . . global 14.3, 15.7
TotalCopy . . . . expr 8.6
Tr. . . . . expr 16.6
Tr. . . . . fexpr 16.3
TraceCount. . . . . expr 16.9
TRACEMAXLEVEL!* . . . . global 16.9
TRACEMINLEVEL!* . . . . global 16.9
TRACENTRYHOOK!* . . . . global 16.15
TRACEXITHOOK!*. . . . . global 16.16

TRACEXPANDHOOK!* . . . . global 16.16
TrCnt . . . . expr 16.13
TREXPRINTER!* . . . . global 16.17
TrIn. . . . . expr 16.9
TRINSTALLHOOK!* . . . . global 16.16
TrOut . . . . expr 16.7
TRPRINTER!* . . . . global 16.17
TRSPACE!* . . . . global 16.17
Trst. . . . . expr 16.6
TTY:. . . . . edit 17.26
Type. . . . . expr 20.3

UnBindN . . . . expr 10.8
UNBLOCK . . . . edit 17.26
UnBoundP. . . . . expr 10.8
UnBr. . . . . fexpr 16.3
UNDO. . . . . edit 17.26
UnFluid . . . . expr 10.8
Union . . . . expr 7.8
UnionQ. . . . . expr 7.8
Unless. . . . . macro 9.3
UnQuote . . . . fexpr 18.12
UnQuoteL. . . . . fexpr 18.12
UnReadChar. . . . . expr 13.6
UnTr. . . . . expr 16.10
UnTr. . . . . fexpr 16.3
UnTrst. . . . . expr 16.10
UP. . . . . edit 17.2, 17.27
UpbV. . . . . expr 8.3
UPFINDFLG . . . . global 17.13
UpperCaseP. . . . . expr 8.7

ValueCell . . . . expr 6.8
VDir. . . . . expr 20.2
Vector2List . . . . expr 4.11
Vector2String . . . . expr 4.11
Vector. . . . . nexpr 4.11, 8.3
VectorP . . . . expr 4.8

WaitFork. . . . . expr 20.4
WAnd. . . . . expr 21.10
WDifference . . . . expr 21.10
WEQ . . . . expr 21.10
WGEQ. . . . . expr 21.11
WGetV . . . . macro 21.11
WGreaterP . . . . expr 21.10
When. . . . . macro 9.2
While . . . . macro 9.6
WINDOWNAMES . . . . global 17.5
WLEQ. . . . . expr 21.11
WLessP. . . . . expr 21.11
WNEQ. . . . . expr 21.10

WNot. . . . . expr 21.10
WOr . . . . expr 21.10
WPlus2. . . . . expr 21.10
WPutV . . . . macro 21.11
WQuotient . . . . expr 21.10
WRemainder. . . . . expr 21.10
WriteChar . . . . expr 13.13
Wrs . . . . expr 13.4
WShift. . . . . expr 21.10
WTimes2 . . . . expr 21.10
WXor. . . . . expr 21.10

XCons . . . . expr 7.3
XJsys0. . . . . expr 20.6
XJsys1. . . . . expr 20.7
XJsys2. . . . . expr 20.7
XJsys3. . . . . expr 20.7
XJsys4. . . . . expr 20.7
XTR . . . . edit 17.27
Xword . . . . expr 20.8

YesP. . . . . expr 14.6

ZeroP . . . . expr 5.6

CHAPTER 26
INDEX OF CONCEPTS

The following is an alphabetical list of concepts, with the page on which they are discussed.

<< >> 3.4

A-Lists 4.4, 7.11
Absolute Value. 5.2
Abstract Machines 19.14
Access to Value Cell. 19.4
Addition. 5.2
Addressing Modes. 19.9
Allocation Functions. 22.8
Allocation. 19.22
Always. 9.8
And 4.8, 9.8
Any 4.4
ANYREG Functions. 19.17
Apollo LAP. 19.9
Appending Lists 7.6
Arguments in Traces 16.5
Arguments 2.7, 10.6
Arithmetic. 5.2
Array 8.6
As. 9.12
ASCII 13.1, 13.5, 13.13
Assigning Precedence. 23.7
Assignment. 6.5
Atom. 4.4, 4.7
Automatic Break 16.11
Automatic Tracing 16.11

Back Quote. 18.12
Back Trace Functions. 16.5
Back Trace. 16.11
Backup Buffer 13.5
Big Integers. 5.1
BigNum. 4.1
Binary Infix Operators. . . . 23.2
Binary Operators. 23.6
Binary Trees. 7.1
Binding Type. 10.7, 10.8
Binding 6.5, 10.6, 10.9
Bit Field Operation 21.7
Bit Operations. 5.6
BNF 23.10, 23.17
Boolean Functions 4.8
Boolean 4.4, 4.7, 5.5
Box Diagrams. 7.1

Break Commands. 15.3
Break Loop. 12.3, 14.6, 15.1, 15.3, 15.6
Buffers in EMODE. 17.5
Bugs. 2.2, 2.8
Building A-Lists. 7.11
Building LAP. 22.5
Building PSL. 22.2
Built-In Functions. 19.17
Byte-Vector 4.1, 8.4

Car Manipulation. 7.2
Case Statement. 9.3, 21.5
Catch 15.1, 15.6
Cdr Manipulation. 7.2
CGOL. 23.2
Channels. 13.1, 13.13
Char and IDLOC Macros 21.4
Character 3.6, 4.4
Circular Structures 16.14, 18.23
Classes of Data Types 4.4
Classes of Functions. 19.17
Closing Functions 13.1
Closure 10.9
Cmacros 19.14
Code Generation 19.14
Code-Pointer. 4.1, 4.7, 10.1, 10.5, 13.5
Collect 9.8
Comments. 23.4
Common LISP 1.1
Compacting G. C.. . . . . . . 22.5
Comparison. 7.10
Compilation 2.7, 10.6, 19.6
Compiled Functions. 10.5, 16.5
Compiled Tracing. 16.5
Compiled vs. Interpreted. . . 19.6
Compiler Second Pass. 19.14
Compiler Third Pass 19.22
Compiler. 19.1
Compiling Functions 19.2
Compiling SYSLISP Code. . . . 21.9
Compiling to FASL Files . . . 19.2
Compiling to Memory 19.2
Composites of Car and Cdr . . 7.3
Compound Statements 3.7
Conc. 9.8
Concatenating Lists 7.6
Cond. 9.4
Conditional Statements. . . . 3.8
Conditionals. 9.1
Constant. 4.4, 4.7
Construction Function 23.2
Construction of MINI. 23.15

Continuing After Errors . . . 15.1
Control Time of Execution . . 19.3
Converting Data Types 4.9, 5.2
Copying Functions 10.2
Copying Strings 8.1
Copying Vectors 8.2
Copying X-Vectors 8.5
Copying 7.2
Count 9.8
Counting Function Calls . . . 16.13
CREF. 18.1
Cross Reference Generator . . 18.1
Customizing Debug 16.15

Data Type Conversion. 4.9, 5.2
Data Types. 4.1, 13.5, 13.13, 13.17
Debug and Redefinition. . . . 16.5
Debug Deficiencies. 16.6
Debug Example 16.18
Debug Printing Functions. . . 16.16
Debug Reading Functions . . . 16.16
Debugging Function Library. . 16.15
Debugging Tools 16.1
Dec-20 LAP. 19.9
DEC-20 PSL. 22.2, 22.5
Decimal Output. 13.13
Declaration 10.6, 10.7
Default Top Level 14.2
DefConst. 18.21
Deficiencies in Debug 16.6
DefMacro. 18.11
Delimiters. 13.5, 13.13
Details of the Compiler . . . 19.13
Digits. 13.5
Diphthong 13.11
Division. 5.2
Do. 9.8
Dot Notation. 3.6, 7.1

Each. 9.12
Edit Commands 17.1, 17.8
Editing in the Break Loop . . 15.3, 17.1
Editing with EMODE. 17.3
Editor. 17.1
EMB Functions 16.5
Embedded Functions. 16.12
EMODE 17.3
End of File 12.3
Env-Pointer 4.1
Environment 10.9
Equality. 4.5
Error Calls 15.7

Error Functions 15.1
Error Handling in MINI. . . . 23.13
Error Handling. 15.1, 23.7
Error Messages. 2.6, 12.3, 13.13
Error Number. 15.1
Error Recovery in MINI. . . . 23.13
Errors. 2.6, 2.8, 10.8
Escaped Characters. 23.7
Eval Type Functions 2.7
Evaluation. 11.1
Example of MINI 23.12
Examples. 2.3, 3.2, 3.3, 7.11, 15.3, 16.18, 18.18, 19.9, 21.9, 23.6, 23.8
Exclamation Point in RLISP. . 23.7
Executable. 14.1
Exit. 9.1, 9.16
Explicit Sequence Control . . 9.4
Exponent. 4.1, 5.2
Expr. 2.7, 10.6
Extend CREF for SYSLISP . . . 21.12
Extensible Parser 23.1
External Form 23.4
Extra-Boolean 4.4

Fexpr 2.7, 10.6
Field 4.1
File Input. 13.11
File Names. 13.3, 13.11
File Output 13.11
Files about MINI. 23.15
Finally 9.8
Find. 18.21
FixNum. 4.1
Flag indicators 12.5
Flagging Ids. 6.4
Flags Controlling Compiler. . 19.5
Flags 2.8, 3.10, 6.3, 6.4, 12.1, 12.6
Float 4.1, 4.7, 5.1, 13.5
Fluid Binding 10.6, 10.9
Fluid Declarations. 19.4
For 9.8
Form Oriented Editor. 17.6
Form. 4.4
Format. 13.5, 13.13, 13.18
Formatted Printing. 13.13
Franz LISP. 1.1
From. 9.8
FType 4.4
Full Structure Editor 17.6
Funarg. 10.9
Function Calls. 23.4
Function Cell 6.1, 11.1

Function Definition 3.3, 10.1
Function Execution Tracing. . 16.6
Function Order. 19.4
Function Redefinition 2.6, 16.5
Function types. 2.7, 10.6
Function. 4.4

Garbage Collector 22.5
GC. 22.5
Generator 23.17
Global Binding. 10.6
Global Declarations 19.4
Global Variables. 3.10
Globals 2.8, 6.8, 12.3, 12.6
Go. 9.1
Graph-to-Tree 18.23

Halfword-Vector 4.1, 8.4
Handlers. 13.3
Hashing Cons. 18.22
Heap. 4.1, 22.6
Help. 2.3, 2.8, 12.6, 14.5
Hexadecimal Output. 13.13
History Mechanism 2.3, 14.2
History of MINI 23.16
Hook. 6.2

I/O Buffer. 13.5
I/O 13.18
Id Space. 4.1
Id-Hash-Table 6.2, 6.8, 14.5
Id. 4.1, 4.7, 4.9, 6.1, 13.5
Identifier. 4.1, 4.7, 4.9, 6.1, 13.5
If Then Construct 9.1
If Then Statements. 3.8
Implementation. 22.1
In. 9.8
Index 8.1
Indicator 6.3
Infix Operators 3.4, 23.4
Initially 9.8
Input in Files. 13.11
Input 3.10, 13.1, 23.2
Integer 4.1, 4.7, 4.9, 5.1, 13.5
Inter LISP. 1.1, 17.6
Intern. 4.9, 6.2, 6.8
InternalForm. 23.4
Internals in Debug. 16.15
Interpretation. 2.7, 19.6
Interpreted Functions 10.5, 10.8
Interpreted Tracing 16.5
Interpreter 11.1

Interrupt Keys. 15.6
Inum. 4.1, 4.9
Item. 4.1
Iteration 9.6

Join. 9.8

Key Words 23.7

Lambda. 4.4, 10.6, 10.8, 11.3
Lap Flags 19.12
LAP Format. 19.9
LAP-to-ASM for Apollo 19.8
LAP 22.5
Length. 7.6
Letter as Token Type. 13.5
LISP 1.6. . . . . 1.1
LISP 360. . . . . 1.1
LISP Machine. 1.2
LISP Surface Language 23.2
LISP vs. RLISP. 3.3
LISP-RLISP Relationship . . . 3.3
List Concatenation. 7.6
List Element Deletion 7.8
List Element Selection. . . . 7.4
List IO 13.18
List Length 7.6
List Manipulation 7.4
List Membership Functions . . 7.6
List Notation Reader. 23.12
List Notation 7.1
List Reversal 7.9
List Substitutions. 7.13
List. 4.4, 4.9, 6.3, 7.1
Loader. 19.8
Loading FASL Files. 19.3
Local Binding 10.6
Local Variables 3.7
Logical And 5.6
Logical Devices for PSL . . . 2.1, 22.1
Logical Exclusive Or. 5.6
Logical Input Record. 12.3
Logical Not 5.6
Logical Or. 5.6
Looping Constructs. 9.6
Loops 3.8, 3.9

Machine Instructions. 19.14
MACLISP 1.1
Macro Defining Tools. 18.11
Macro Expand. 18.13
Macro 2.8, 10.6, 11.4

Mapping Functions 9.12
Mathematical Functions. . . . 18.25
MaxChannels 13.1
Maximize. 9.8
Memory Access Operations. . . 21.7
Memory Address Operations . . 21.7
Messages. 2.6
Meta Compiler 23.1
MINI Development. 23.16
MINI Error Handling 23.13
MINI Error Recovery 23.13
MINI Example. 23.12
MINI Operators. 23.10
MINI Self-Definition. 23.13
Mini Trace. 16.2
MINI. 23.10
Minimize. 9.8
Minus as Token Type 13.5
Mode Analysis Functions . . . 21.3
Modified FOR Loop 21.4
Multiplication. 5.2

N-ary Expressions 23.6
N-ary Functions 3.4
Need for Two Stacks 21.12
Never 9.8
New Mode System 21.12
Nexpr 2.8, 10.6
Next. 9.1
NIL 1.2, 4.7, 4.8
NoEval Type Functions 2.7
Non-Local Exit. 9.16
None Returned 4.4
NoSpread Type Functions . . . 2.7
Not 4.8, 9.8
Notation. 4.1
Number. 4.4, 4.7, 4.9, 5.1, 13.5
Numeric Comparison. 5.5, 7.10

Oblist. 13.17
Octal Output. 13.13
Off 12.1
On. 9.8, 12.1
Open Coding 19.6
OPEN Functions. 19.17
Operator Definition 23.8
Operator Precedence 3.4
Operators 23.2
Optimizations 19.22
Or. 4.8, 9.8
Order of Functions. 19.4
Output Base 13.13

Output. 3.10, 13.1
OutPutBase!*. 13.13
Overflow. 13.18

Package Cell. 6.1
Package 6.2, 6.8
Pair Construction 7.2
Pair Manipulation 7.2
Pair. 4.1, 4.4, 4.7, 7.1
Pairs 7.1
Parameters. 2.7, 10.6
Parentheses 23.5
Parse 3.6
Parser Flow Diagram 23.2
Parser Generator. 23.1
Parser. 13.17, 23.1
Parsing Precedence. 23.2
PASS1 of Compiler 19.13
Pattern Matcher 23.12
Pattern Matching in MINI. . . 23.12
Picture RLISP 18.4
Plus as Token Type. 13.5
Precedence Table. 23.2
Precedence. 3.4, 23.5
Predicates. 4.5, 5.5, 7.6, 10.5, 10.6, 10.8
Print Name. 6.1, 23.7
Printing Circular Lists . . . 16.14, 18.23
Printing Circular Vectors . . 18.23
Printing Functions. 16.14
Printing Property Lists . . . 16.14
Printing Registers. 13.13
Printing. 13.13
PRLISP. 18.4
Procedures. 3.6
Product 9.8
Productions 23.10
Prog. 9.4, 10.6, 10.8
ProgN 9.4
Prompt. 12.3
Properties. 6.3
Property Cell Access. 6.5
Property Cell 6.1, 6.3
Property List 6.1, 6.3, 12.4, 23.4
Pseudos 19.9
PSL Files 22.1
PSL History 1.6
PSL Sample Session. 2.3
Put Indicators. 12.4

Quote Mark in RLISP 23.7
Quote Mark. 23.4

Radix for I/O 13.5
Random Functions. 19.17
RCREF 18.1
Read Macros 13.11, 13.18
Read. 3.6, 23.2
Reading Functions 13.1, 13.5
Reclaim 12.3
Recognizer. 23.17
Reduce. 3.1, 16.18
Register and Tracing. 16.6
Registers 13.13
Remainder 5.2
Remaining SYSLISP Issues. . . 21.11
Removing Functions. 10.2
Return. 9.1
Returns 9.8
Right Precedence. 23.2
RLISP Commands. 14.6
RLISP Diphthong 13.17
RLISP Input 3.10
RLISP Output. 3.10
RLISP Parser. 23.7
RLISP Syntax. 3.2
RLISP to LISP Translation . . 23.17
RLISP to LISP Using MINI. . . 23.17
RLISP vs. LISP. 3.3
RLISP vs. SYSLISP 21.2
RLISP-LISP Relationship . . . 3.3
RLISP-SYSLISP Relationship. . 21.2
RLISP 3.1
Routines. 12.3
Running MINI. 23.13

S-Expression. 4.4, 13.17
S-Integer 4.1, 4.9
Saving Executable PSL 14.1
Saving Trace Output 16.7
Scalar. 3.4, 3.7, 3.9
Scan Table. 13.5, 13.11, 14.2, 23.4
Scope of Variables. 10.6
Screen Editor 17.3
Searching A-Lists 7.11
Selective Trace 16.9
Sequence of Evaluation. . . . 9.4
Set Functions 7.8
Sharp-Sign Macros 18.12
Side Effects. 19.17
Skip to Top of Page 13.13
Sorting Functions 7.10
Special Error Handlers. . . . 15.8
Special I/O Functions 13.3
Spice LISP. 1.2

Spread Type Functions 2.7
Stable Functions. 19.17
Stack 18.14
Stand Alone SYSLISP 21.12
Starting MINI 23.13
Starting PSL. . . . . 2.1, 2.2, .i
Starting the Debug Package. . 16.5
Starting the Editor 17.6
Statistics Functions. 16.5
Stop and Copy on VAX. 22.6
Stopping PSL. 14.1
String Comparison 7.10
String IO 13.18
String Operations 8.1
String Quotes 13.5
String. 4.1, 4.7, 4.9, 13.5
Structural Notes: Compiler. . 19.22
Structure Definition. 18.14
Structure 4.4
Stubs 16.13
Substitutions 7.13
Substring Matching in Ids . . 18.21
Substring Matching. 18.21
Subtraction 5.2
Sum 9.8
SYSLISP Arguments 13.13
SYSLISP Declarations. 21.2
SYSLISP Functions 21.10
SYSLISP Level of PSL. 21.1
SYSLISP Mode Analysis 21.3
SYSLISP Programs. 21.12
SYSLISP vs. RLISP 21.2
SYSLISP-RLISP Relationship. . 21.2
System Dependent Functions. . 20.1

Table Driven Parser 23.2
Tag Field 4.1
Tagging Information 19.14
Template and Replacement. . . 23.12
Terminal Interaction. 14.6
Throw 15.1, 15.8
Time Control Functions. . . . 19.3
Tokens. 23.2
Top Level Function. 14.2
Top Loop Mechanism. 15.6
Top Loop. 14.2
Trace Output. 16.7
Trace 16.5
Tracing Compiled Functions. . 16.5
Tracing Functions 2.3, 16.2
Tracing Macros. 16.6
Tracing New Functions 16.11

Trees 23.10
Trigonometric Functions . . . 18.25
Truth 4.8
Turning Off Trace 16.10
Type Checking Functions . . . 4.7
Type Conversion 4.9, 5.2
Type Declarations 4.1
Type Field. 4.1
Type Mismatch 13.18

UCI LISP. 1.1, 17.6
Unary Functions 3.4, 5.2
Unary Prefix Operators. . . . 23.2
Undefined 4.4
Union 9.8
Unless. 9.8
Until 9.8
Untraceable Functions 16.6
Useful Functions. 16.15
User Function Redefinition. . 16.5
User Hooks in Debug 16.15
User Interface. 14.1
Using SYSLISP 21.9
UT LISP 1.1
Utilities 18.1

Value Cell. 6.1, 6.5, 10.6
Variable Binding. 6.5, 10.6
VAX LAP 19.8, 19.9
VAX PSL 22.6
Vector Index. 8.1
Vector Operations 8.2
Vector. 4.1, 4.7, 4.9

Warning Messages. 2.6
When. 9.8
While 9.8
Windows in EMODE. 17.5
With. 9.8
Word Operations 8.4
Word-Vector 4.1, 8.4
Word. 4.1
Writing Functions 13.1

X-Vector Operations 8.5
X-Vector. 4.4, 8.1