



INTERLISP REFERENCE MANUAL

BY WARREN TEITELMAN

contributions by:

**J. W. GOODWIN
A. K. HARTLEY
D. C. LEWIS
J. J. VITTAL
M. D. YONKE**

BOLT BERANEK & NEWMAN

**D. G. BOBROW
R. M. KAPLAN
L. M. MASINTER
B. A. SHEIL**

XEROX PALO ALTO RESEARCH CENTER

XEROX
PALO ALTO RESEARCH CENTER
3333 COYOTE HILL ROAD / PALO ALTO / CALIFORNIA 94304

ACKNOWLEDGMENTS AND BACKGROUND

Interlisp has evolved from a succession of LISP systems that began with a LISP designed and implemented for the DEC PDP-1 by D. G. Bobrow and D. L. Murphy¹ at Bolt, Beranek and Newman in 1966, and documented by D. G. Bobrow. An upwards compatible version of this LISP was implemented for the SDS 940 in 1967, by Bobrow and Murphy. This system contained the seeds for many of the capabilities and features of the current system: a compatible compiler and interpreter,² uniform error handling, an on-line LISP oriented editor,³ sophisticated debugging facilities,⁴ etc. 940 LISP was also the first LISP system to demonstrate the feasibility of using software paging techniques and a large virtual memory in conjunction with a list-processing system [Bob2]. DWIM, the Do-What-I-Mean error correction facility, was introduced into the system in 1968 by W. Teitelman [Tei2], who was also responsible for documentation for the 940 LISP system.

In 1970, an upwards compatible version of 940 LISP called BBN LISP⁵ was designed for the PDP-10 by D. G. Bobrow, D. L. Murphy, A. K. Hartley, and W. Teitelman, and implemented by Hartley with assistance from Murphy. A. K. Hartley was also responsible for modifying the 940 LISP compiler to generate code for the PDP-10. BBN-LISP ran under TENEX, a sophisticated time sharing system for the PDP-10 designed and implemented by D. G. Bobrow, J. D. Burchfiel, D. L. Murphy, T. R. Strollo, and R. S. Tomlinson.[Bob1] With hardware paging and 256K of virtual memory provided by TENEX, it became practical to provide extensive and sophisticated interactive user support facilities, such as the programmer's assistant [Tei4], CLISP [Tei5], and a more sophisticated DWIM, all of which were designed and developed by W. Teitelman. In 1971, the block compiler was designed and implemented by D. G. Bobrow. In 1974, the ability to swap, i.e., overlay, compiled code was added, significantly increasing the virtual address space. In 1975, implementation of the spaghetti stack capability based on the Bobrow-Wegbreit model described in [Bob3] was completed by A.K. Hartley. The BBN-LISP Manual [Tei3] was written by W.

¹ D. G. Bobrow is currently at Xerox Palo Alto Research Center (PARC), D. L. Murphy is with Digital Equipment Corp.

² The preliminary version of the compiler was written by L. P. Deutsch, now at Xerox PARC. This was considerably modified and extended by D. L. Murphy before producing the final working version.

³ The original idea of a LISP oriented structure editor belongs to L. P. Deutsch. The editor in its current form was written by W. Teitelman, now of Xerox PARC.

⁴ Designed and implemented by W. Teitelman.

⁵ The design, construction and documentation for BBN LISP was sponsored by the Information Processing Techniques Section of the Advanced Research Project Agency, as was all of the subsequent work on the system that was performed at BBN. Since March 1972, the contributions made to the development of the system by W. Teitelman, including the preparation of this manual, were sponsored by Xerox Palo Alto Research Center.

Teitelman, with contributions from A. K. Hartley and from J. W. Goodwin, who also wrote TRANSOR and the special arithmetic functions, as well as a number of other utility functions. The name of the system was changed from BBN-LISP to Interlisp in 1973, when the maintenance and development of the system evolved into a joint effort between Bolt Beranek and Newman, and Xerox Palo Alto Research Center. The Interlisp reference manual was written by W. Teitelman, with contributions from (in alphabetic order) D. G. Bobrow, J. W. Goodwin, A. K. Hartley, P. C. Jackson, D. C. Lewis, and L. M. Masinter. The cover was designed by Alice R. Fikes.

Interlisp-10 is currently the LISP system used at Bolt Beranek and Newman, Xerox Palo Alto Research Center, Stanford Research Institute Artificial Intelligence Center, Information Sciences Institute, the SUMEX facility at Stanford University, Yale University, Caltech, Rutgers, the University of Linkoping in Sweden and others. Interlisp-10 now runs under both the Tenex and TOPS-20 operating system, and is available through DECUS; the total Interlisp-10 user community now comprises over three hundred users. To aid other Interlisp implementations and to encourage compatibility between implementations, specifications of Interlisp can be found in "The Interlisp Virtual Machine Specification" [Moo].

Interlisp is a continuously evolving system, both in response to complaints, suggestions, and requests of the many users scattered throughout the ARPA network, as well as the long range goals of the individuals primarily responsible for the system, which are currently:

Person

W. Teitelman
Xerox Palo Alto
Research Center
3333 Coyote Hill Road
Palo Alto, Calif. 94304

A. K. Hartley
Bolt Beranek & Newman
50 Moulton St.
Cambridge, Mass. 02138

D. C. Lewis
Bolt Beranek & Newman
50 Moulton St.
Cambridge, Mass. 02138

L. M. Masinter
Xerox Palo Alto
Research Center
3333 Coyote Hill Road
Palo Alto, Calif. 94304

R. M. Kaplan
Xerox Palo Alto
Research Center
3333 Coyote Hill Road
Palo Alto, Calif. 94304

Responsible for

User Facilities: i.e., pretty-print, editor,
break and trace, advising,
DWIM, CLISP, programmer's assistant, etc.

Interlisp-10 interpreter, garbage collector,
all SUBR's (hand-code machine language functions),
compiler, spaghetti stack.

Interlisp-10 input-output, readtables,
terminal tables, user data types, overlays, TENEX
functions.

pattern match compiler, record package,
MASTERSCOPE.

LISPUSERS packages

The preparation of this manual has involved the efforts of several persons at Xerox PARC, whom I specifically want to mention, and to express my appreciation for their support through this arduous, and at times seemingly endless task. Thank you Suzan (Jerome), Janet (Farness), Peter (Deutsch), Bob (Walker), and Larry (Tesler). I couldn't have done it without you.

Warren Teitelman
Palo Alto
December, 1973

Special thanks go to R. L. Walker, L. M. Masinter, and L. P. Deutsch for assistance in the preparation of this first revision.

W.T.
October, 1974

Special thanks go to Carol Van Jepmond for assistance in the preparation of this second revision.

W.T.
December, 1975

Special thanks go to Sara Dake who did the bulk of the editing, to L. M. Masinter and R. M. Kaplan for their valuable suggestions and help in proofreading, to R. E. Fikes, who proofread and checked the entire index, and also to J. Maleson and D. C. Swinehart, for assisting with the software for generating a publishable quality copy of the manuscript.

W.T.
October, 1978



TABLE OF CONTENTS

SECTION 1: INTRODUCTION

SECTION 2: USING INTERLISP

Using the Interlisp Manual	1
Using the Interlisp System	3

SECTION 3: DATA TYPES, STORAGE ALLOCATION, GARBAGE COLLECTION, AND OVERLAYS

Data Types	1
Literal Atoms	1
Pnames	2
Value Cells	2
Numerical Atoms	3
Lists	4
Arrays	5
Strings	6
User Defined Data-Types	7
Storage Allocation and Garbage Collection	9
Shared Interlisp-10	11
The Interlisp-10 Swapper	12
Overlays	12
Efficiency	13
Specifications	14

SECTION 4: FUNCTION TYPES AND IMPLICIT PROGN

Exprs	1
Compiled Functions	2
Function Type	2
Progn	3
Implicit Progn.	3

SECTION 5: PRIMITIVE FUNCTIONS AND PREDICATES

Primitive Functions	1
Changing and Restoring System State	7
Predicates and Logical Connectives	9

TABLE OF CONTENTS (cont.)

SECTION 6: LIST MANIPULATION AND CONCATENATION

SECTION 7: PROPERTY LISTS AND HASH LINKS

Property Lists	1
Property List Functions	1
Hash Links	3
Hash Link Functions	4
Hash Overflow	5

SECTION 8: FUNCTION DEFINITION AND EVALUATION

SECTION 9: THE INTERLISP EDITOR

Introduction	1
Commands for the New User	7
Attention Changing Commands	10
Local Attention-Changing Commands	10
Commands That Search	14
Search Algorithm	16
Search Commands	17
Location Specification	19
Commands That Save and Restore the Edit Chain	22
Commands That Modify Structure	24
Implementation of Structure Modification Commands	25
The A, B, and : Commands	26
Form Oriented Editing and the Role of UP	28
Extract and Embed	29
The MOVE Command	31
Commands That Move Parentheses	33
TO and THRU	35
The R Command	38
Commands That Print	40
Commands That Evaluate	41
Commands That Test	43
Macros	45
Miscellaneous Commands	47
Commands for Leaving the Editor	47
Nested Calls to Editor	49
Manipulating the Characters of an Atom or String	50
Manipulating Predicates and Conditional Expressions	51
History Commands in the Editor	52
Miscellaneous	52
Undo	54
Editdefault	56
Editor Functions	58

TABLE OF CONTENTS (cont.)

SECTION 10: ATOM, STRING, ARRAY, AND STORAGE MANIPULATION

Pnames and Atom Manipulation	1
String Functions	4
Searching Strings	6
Example Problem	6
String Storage	7
Array Functions	8
Storage Functions	10

SECTION 11: FUNCTIONS WITH FUNCTIONAL ARGUMENTS

Funarg	3
Example	4

SECTION 12: VARIABLE BINDINGS, PUSHDOWN LIST FUNCTIONS, AND THE SPAGHETTI STACK

The Pushdown List and the Interpreter	2
Blip Functions	4
The Spaghetti Stack	4
Stack Functions	6
Releasing and Reusing Stack Pointers	12
Coroutines and Generators	13
Possibilities Lists	16

SECTION 13: NUMBERS AND ARITHMETIC FUNCTIONS

Integer Arithmetic	1
Small Integers	1
Integer Functions	2
Floating Point Arithmetic	5
Mixed Arithmetic	6
Special Functions	7
Reusing Boxed Numbers in Interlisp-10 - Setn	8
Caveats Concerning Use of Setn	9
Box and Unbox in Interlisp-10	10

SECTION 14: INPUT/OUTPUT FUNCTIONS

Files	1
Manipulating File Names	6
Addressable Files	7
Closing and Reopening Files - The Whenclose Package	9
Input Functions	11
Output Functions	17
Printlevel	18

TABLE OF CONTENTS (cont.)

Printing Numbers - The Printnum Package	20
User Defined Printing	22
Dumping Unusual Data Structures	22
Readtables and Terminal Tables	23
Readable Functions	24
Syntax Classes	24
Format Characters	25
Breaks, Separators, and Readtables	26
Read-macro Characters	26
Terminal Tables	29
Terminal Table Functions	29
Terminal Syntax Classes	30
Terminal Control Functions	30
Line-Buffering and Control	32
Turning-Off Line-Buffering	33
Miscellaneous Input/Output Control Functions	34
Sysin and Sysout	36
Symbolic File Input	38
File Maps	41
Readfile and Writefile	42
Prettyprint	43
Comment Feature	44
Comment Pointers	45
Converting Comments to Lower Case	45
Special Prettyprint Controls	48
Font Package	49
File Package	54
Typed Definitions	55
File Package Commands	56
File Package Types	61
Marking Changes	62
Noticing Files	63
Updating Files	63
File Package Functions	64
Functions for Manipulating Typed Definitions	69
Defining New File Package Types	72
Defining New File Package Commands	73
Functions for Manipulating File Command Lists	75
Remaking a Symbolic File	77
Prettydef Functions	78

SECTION 15: DEBUGGING - THE BREAK PACKAGE

Debugging Facilities	1
Break1	3
Break Commands	5
Brkcoms	13
Brkfile	13
Breakmacros	13
Breakresetforms	14

TABLE OF CONTENTS (cont.)

Break Functions	14
Breakin	16

SECTION 16: ERROR HANDLING

Unbound Atoms and Undefined Functions	1
Terminal Initiated Breaks	2
Control-H	2
Control-B	2
Control-E	2
Other Types of Errors	3
Breakcheck - When to Break	3
Error Types	4
Error Handling by Error Type	9
Error Functions	10
Interrupt Characters	12

SECTION 17: AUTOMATIC ERROR CORRECTION - THE DWIM FACILITY

Introduction	1
Interaction with Dwim	3
Spelling Correction Protocol	4
Parentheses Errors Protocol	5
U.D.F. T Errors Protocol	5
Spelling Correction	7
Synonyms	7
Spelling Lists	8
Generators for Spelling Correction	9
Error Correction	10
Unbound Atoms	10
Undefined Car or Form	12
Undefined Function in Apply	13
Dwimuserforms	14
Spelling Corrector Algorithm	16
Dwim Functions and Variables	17
Askuser	22
Startup Protocol	22
Operation	23
Format of Keylist	24
Completing a Key	25
Options	26
Special Keys	28

TABLE OF CONTENTS (cont.)

SECTION 18: THE COMPILER AND ASSEMBLER

The Compiler	1
Compiler Questions	2
Compiling Nlambda	3
Global Variables Vs Shallow Binding	4
Compiler Functions	5
Declare:	6
Recompile	7
Open Functions	9
Compiler Macros	10
Constant	12
Compiletypest	12
Macrotran	13
Function and Functional Arguments	13
Block Compiling	14
Retfns	14
Blkapplyfns	14
Blklibrary	15
Linked Function Calls	15
Relinking	17
Localvars and Specvars	18
The Block Compiler	19
Blockcompile	19
Block Declarations	20
Bcompl	22
Bcompile	22
Compiler Structure	23
Assemble	24
Assemble Statements	24
Corevals	27
Lap	27
Lap Statements	27
Using Assemble	31
Miscellaneous	32
Compiler Printout and Error Messages	33

SECTION 19: ADVISING

Implementation of Advising	2
Advisc Functions	3

TABLE OF CONTENTS (cont.)

SECTION 20: MASTERSCOPE AND HELPSYS

Masterscope	1
Command Language	3
Relations	4
Sets	6
Conjunctions	9
Commands	9
Paths	14
Affecting Masterscope Analysis	16
Data-base Updating	18
Masterscope Entries	18
Error Messages	20
Noticing Changes that Require Recompiling	20
Implementation Notes	21
Helpsys	22

SECTION 21: MISCELLANEOUS

Measuring Functions	1
Breakdown	4
More Accurate Measurement	6
Interfork Communication in Interlisp-10	6
Subsys	7
Miscellaneous Operating System Functions	8
Manipulating Tenex File Directories from Interlisp-10	10
JFN Functions in Interlisp-10	11
Pmap Package	12
Typecript files	15
Display Terminals	15
Gainspace	16

SECTION 22: THE PROGRAMMER'S ASSISTANT

Introduction	1
Overview	4
Event Specification	10
History Commands	12
Implementation of Redo, Use, and Fix	15
History Commands Applied to History Commands	16
History Commands That Fail	17
More History Commands	17
Miscellaneous Features and Commands	23
Undoing	31
Testmode	32
Undoing out of order	33
SaveSet	33
Undonlsetq and Resctundo	34
Format and Use of the History List	34

TABLE OF CONTENTS (cont.)

Lispx and Readline	35
Functions	36
The Editor and the Programmer's Assistant	45
Statistics	46
Greeting and User Profiles	46

SECTION 23: CLISP-CONVERSATIONAL LISP

Introduction	1
Cisp Syntax	6
Infix Operators	6
Prefix Operators	8
Constructing Lists - the < and > Operators	10
If, Then, Else	10
Iterative Statements	11
Miscellaneous	17
Errors in Iterative Statements	18
Defining New Iterative Statement Operators	19
English Phrases	21
Cisp Translations	22
Declarations	24
Table of Declarations	24
Local Declarations	25
Record Package	25
Record Declarations	27
Create	32
Type?	33
Data-paths	34
Changing Record Declarations	35
Other Functions	35
Changetran	36
Change Words Recognized by Changetran	37
Cispify	38
Dwimify	40
Compiling Cisp	42
The Printout Package	43
Overview and Examples	43
The Printout Form	44
Horizontal Spacing Commands	45
Vertical Spacing Commands	46
Special Formatting Controls	46
Printing Specifications	47
Escaping to Lisp	49
User-Defined Commands	49
Special Printing Functions	49
Cisp Operation	50
Cisp Interaction with User	52
Cisp Internal Conventions	53
Cisp Functions and Variables	56

TABLE OF CONTENTS (cont.)

SECTION 24: LISPUSERS PACKAGES

Pattern Match Compiler	1
Pattern Elements.	2
Element Patterns.	2
Segment Patterns.	4
Assignments	5
Place-Markers	6
Replacements	6
Reconstruction	7
Edita	8
Overview.	9
Input Protocol	10
Edita Commands and Variables	11
Editing Arrays	14
Printing Reentrant and Circular List Structures	15
Circlprint	15
Printl	19
Transor	21
Introduction	21
Using Transor	22
Translating	22
The Translation Notes	23
Errors and Messages.	24
Transorset	24
Transorset Commands	25
The Remark Feature.	26
Controlling the Sweep	28
Indexing and Cross Referencing Files	30
Multifileindex	30
Singlefileindex	31
Databsefns	31
Lambdatran	32
Permstatus	34
Whereis	34
Cjsys	35
Scratchlist.	36
Nobox.	37
Cons Cells	37
Number Boxes	37
Cautions	39
Dateformat	40
Exec	41
Exec Lispxmacros	41
Exec Functions	43
Passwords	44
Telnet.	44
FTP	44
NET	46
Hash--A Hash-coded Dictionary Facility	47
Unstructured Pages and Symbol Tables	51

TABLE OF CONTENTS (cont.)

The Printing Region	53
The Decl Package.	53
Introduction	53
Using Declarations in Programs	54
Dlambdas	55
Dprogs	57
Declarations in Iterative Statements	58
Declaring a Variable for a Restricted Lexical Scope	58
Declaring the Values of Expressions	59
Assertions	59
Using Type Expressions as Predicates	59
Enforcement.	59
Decltypes	60
Predefined Types	61
Type Expressions	61
Named Types	63
Manipulating Named Types	65
Relations Between Types.	65
The Declaration Database	66
Declarations and Masterscope	67

APPENDIX 1: CONTROL CHARACTERS

MASTER INDEX

INDEX OF FUNCTIONS

INDEX OF VARIABLES AND PARAMETERS

SECTION 1

INTRODUCTION

This document is a reference manual for Interlisp, a LISP system that is currently implemented on (or implementations are in progress for) at least five different machines. This manual is a reference manual for all Interlisp implementations, although it does contain some material that is relevant only to Interlisp-10, the implementation of Interlisp for the DEC PDP-10, models KA and KI, using the BBN TENEX time sharing system [Bob1] and the KL-10 using the TOPS-20 operating system.¹ Where this is the case, such material is clearly marked.

Interlisp has been designed to be a good on-line *interactive* system (from which it derives its name). Some of the features provided include elaborate debugging facilities with tracing and conditional breakpoints (Section 15), and a sophisticated LISP oriented editor within the system (Section 9). Utilization of a uniform error processing through user accessible routines (Section 16) has allowed the implementation of DWIM, a Do-What-I-Mean facility, which automatically corrects many types of errors without losing the context of computation (Section 17). The CLISP facility (Section 23) extends the LISP syntax by enabling ALGOL-like infix operators such as +, -, *, /, =, ←, AND, OR, etc., as well as IF-THEN-ELSE statements and FOR-WHILE-DO statements. CLISP expressions are automatically converted to equivalent Interlisp forms when they are first encountered. CLISP also includes a sophisticated pattern match compiler, as well as a record package that facilitates "data-less" programming.

Interlisp has also been designed to be a *flexible* system. Advising (Section 19) enables users to selectively modify or short-circuit any system function. Even such "built-in" aspects of the system as interrupt characters, garbage collection allocation and messages, output radix, action on various error conditions, line-buffering protocol, etc., all can be modified through system functions provided for that purpose. Readtables and terminal tables (Section 14) allow the user complete control over input, including the ability to define read macro characters, specify echo modes, even redefine the action of formatting characters such as parentheses. The user can also define new datatypes (Section 23) in addition to the lists, strings, arrays, and hash association tables (hash links) already provided.

A novel and useful facility of the Interlisp system is the programmer's assistant (Section 22), which monitors and records all user inputs. The user can instruct the programmer's assistant to repeat a

¹ Interlisp-10 is designed to provide the user access to the large virtual memory allowed by TENEX and TOPS-20, with relatively small penalty in speed (using special paging techniques described in [Bob2]). Interlisp-10 also provides for essentially unlimited quantity of compiled code via the overlay facility described in Section 3. Interlisp-10 was the first implementation of Interlisp, and is still the most widely used.

particular operation or sequence of operations, with possible modifications, or to UNDO the effects of specified operations. The programmer's assistant also includes a (limited) error analysis capability. The goal of the programmer's assistant, DWIM, CLISP, etc. is to provide a programming environment which will "cooperate" with the user in the development of his programs, and free him to concentrate more fully on the conceptual difficulties and creative aspects of the problem he is trying to solve.

Masterscope is an interactive program for analyzing user programs and determining what functions are called and by whom, how and where variables are bound, set, or referenced, which functions use particular record declarations, etc. It is extremely useful for building large systems.

To aid in converting to Interlisp programs written in other LISP dialects, e.g., LISP 1.5, Stanford LISP, we have implemented TRANSOR, a subsystem which accepts transformations (or can operate from previously defined transformations), and applies these transformations to source programs written in another LISP dialect, producing object programs which will run on Interlisp (Appendix 1). In addition, TRANSOR alerts the programmer to problem areas that (may) need further attention. TRANSOR was used extensively in converting from 940 LISP to BBN-LISP on the PDP-10. A set of transformations is available for converting from Stanford LISP and LISP 1.5 to Interlisp.

A complete format directed list processing system FLIP [Teil], is available for use within Interlisp. The TXDT package [Moo2] provides a powerful set of text editing primitives implemented in Interlisp from which the user can easily construct and experiment with a variety of editors. HELPSYS, an on-line information retrieval package that uses this manual as a data-base is also available to provide the user with immediate access to documentation.

Although we have tried to be as clear and complete as possible, this document is not designed to be an introduction to LISP. Therefore, some parts may only be clear to people who have had some experience with other LISP systems. A good introduction to LISP has been written by Clark Weissman [Weil]. Although not completely accurate with respect to Interlisp, the differences are small enough to be mastered by use of this manual and on-line interaction. Another useful introduction is given by Berkeley [Ber1] in the collection of Berkeley and Bobrow [Ber2].

* * *

First revision, October, 1974.

The first revision of the Interlisp Reference Manual corresponds to changes or additions to the Interlisp system during the first ten months of 1974. Approximately 200 (out of 700) pages have been changed to some extent in this revision. A significant number of these (about 60 pages) occur in Section 14 (input/output). About 30 pages of chapter 23 (CLISP) have been changed, and the rest of the changes are scattered throughout the manual. Changed material in the text is flagged in the outside margin by the appearance of either a "+" (for addition of completely new material), "-" (for deletion of original material), or "*" (indicating changes to existing material that more or less preserve its original structure.) Thus the reader who is already familiar with the Interlisp manual can quickly determine what has been changed. Note: very few of these changes are not "upwards compatible" with the original manual, i.e., almost all of them represent extensions or additions. Nevertheless, the reader is encouraged to skim through the manual noting changes which may affect him.

* * *

Second revision, November, 1975.

The second revision of the Interlisp Reference Manual corresponds to changes or additions to Interlisp from October, 1974 to November, 1975; the most important change is the introduction of the spaghetti stack capability, Section 12. As before, material in the text is flagged in the outside margin by the appearance of either a "+", "**", or "-".

* * *

Third revision, October, 1978.

The third revision of the Interlisp Reference Manual corresponds to changes or additions to Interlisp from November, 1975 to October, 1978. As before, new material is flagged with a "+" in the margin, changed material a "**". The most important change is the switch from a deep binding scheme to shallow binding (Section 12), with a significant improvement in performance of interpreted and non-block compiled programs.

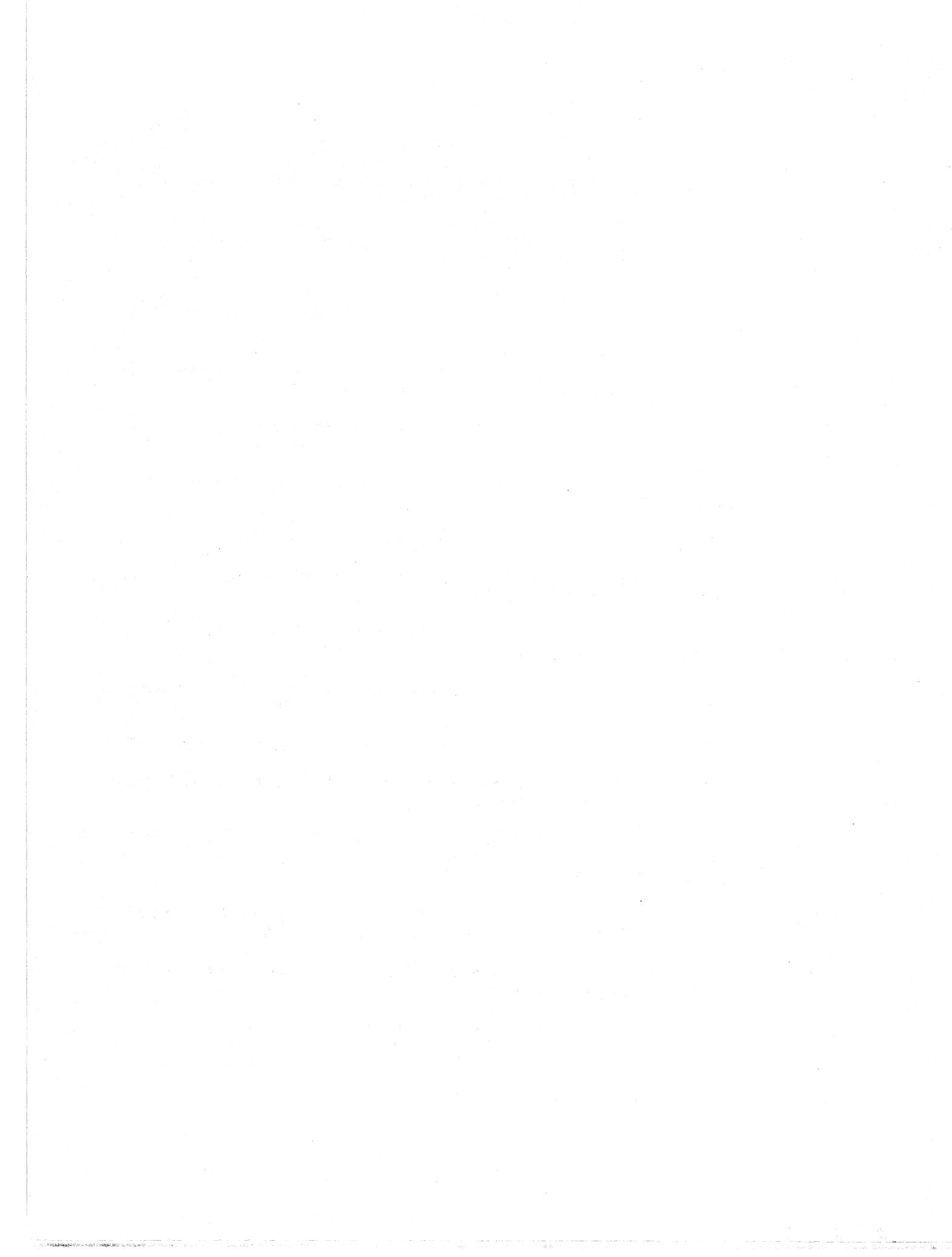
In addition, this new manual includes a number of packages and tools developed and then proven in actual use over the last several years by the user communities at PARC and BBN. Some of these are now included as part of the standard system, and are documented in the appropriate places in the manual, such as the font package for producing listings with multiple fonts (Section 14); the editdate package for time stamping changes to functions plus many new and powerful edit commands (Section 9); an improved and generalized file package (Section 14); the printout package which permits the user to specify in a single succinct expression, large variety of fancy printing controls, e.g. various flavors of prints, prettyprints, special number formatting, paragraph printing, centering, right justifying (Section 23); the Pmap facility which allows paged access to files in

Interlisp-10 (Section 21), etc. Others are not a part of the system (in the interest of space) but are available as loadable files. These are documented in Section 24, Lispusers Packages, and include: a hash package used for maintaining large symbolic databases on files, which permits information to be associated with an atom or string and quickly returned; the Decl package which extends Interlisp to allow declaration of the types of variables and expressions appearing in functions; the Exec package which provides a number of TENEX/TOPS-20 capabilities to Interlisp programs directly; the Ftp package which makes it possible to deal with files at other hosts on the Arpa network as if they were files on the user's local machine; etc.

The last three years have seen a rapid growth in the Interlisp community, especially since the availability of Interlisp on the TOPS-20 operating system. As of this writing, Interlisp-10 is in use at 13 computer centers on 23 different computers, 11 of them KL-10's running TOPS-20.

BIBLIOGRAPHY

- [Ber1] Berkeley, E.C., "LISP, A Simple Introduction" in Berkeley, E.C. and Bobrow, D.G.
[Ber2].
- [Ber2] Berkeley, E.C., and Bobrow, D.G. (Eds.), *The Programming Language LISP, its Operation and Applications*, MIT Press, 1966.
- [Bob1] Bobrow, D. G., Burchfiel, J. D., Murphy, D. L., and Tomlinson, R. S. "TENEX, a Paged Time Sharing System for the PDP-10", *Communications of the ACM*, March, 1972.
- [Bob2] Bobrow, D.G., and Murphy, D.L. "The Structure of a LISP System Using Two Level Storage", *Communications of the ACM*, Vol. 10, 3, March 1967.
- [Bob3] Bobrow, D.G., and Wegbreit, B. "A Model and Stack Implementation for Multiple Environments", *Communications of the ACM*, Vol. 16, 10, October 1973.
- [Boy] Boyer, Robert J., and Moore, J Strother. "A Fast String Searching Algorithm," to be published in CACM.
- [McC1] McCarthy, J. et al. *LISP 1.5 Programmer's Manual*, MIT Press, 1966.
- [Moo] Moore, J Strother. *The Interlisp Virtual Machine Specification* (in preparation), Xerox Palo Alto Research Center Report, 1975.
- [Mur1] Murphy, D.L. "Storage Organization and Management in TENEX", *Proceedings of Fall Joint Computer Conference*, December 1972.
- [Smi1] Smith, D. "MLISP" Artificial Intelligence Memo No. 135 Stanford University, October 1970.
- [Tei1] Teitelman, W. *FLIP, A Format Directed List Processor in LISP*, BBN Report, 1967.
- [Tei2] Teitelman, W. "Toward a Programming Laboratory" in Walker, D. (ed.) *International Joint Conference on Artificial Intelligence*, May 1969.
- [Tei3] Teitelman, W., Bobrow, D.G., Hartley, A.K. Murphy, D.L. *BBN-LISP TENEX Reference Manual*, Bolt Beranek and Newman, July 1971, first revision February 1972, second revision August 1972.
- [Tei4] Teitelman, W. "Automated Programming - The Programmer's Assistant", *Proceedings of the Fall Joint Computer Conference*, December 1972.
- [Tei5] Teitelman, W. "CLISP - Conversational LISP", *Third International Joint Conference on Artificial Intelligence*, August 1973.
- [Wei1] Weissman, C. *LISP 1.5 Primer*, Dickenson Press (1967).



SECTION 2

USING INTERLISP

2.1 USING THE INTERLISP MANUAL - FORMAT, NOTATION, AND CONVENTIONS

The Interlisp manual is divided into separate, more or less independent sections. Each section is paginated independently, to facilitate issuing updates of sections. Each section contains an index to key words, functions, and variables contained in that section. In addition, there is a composite index for the entire manual, plus several appendices and a table of contents.

Interlisp is currently implemented on (or implementations are in progress for) at least four different computers. This manual purports to be a reference manual for all implementations of Interlisp, both present and future. However, since the largest user community is still that of Interlisp-10, the implementation for the DEC PDP-10 series,¹ the manual does contain some implementation dependent material. Where this is the case, the text refers to Interlisp-10, and is indicated as such.

Throughout the manual, terminology and conventions will be offset from the text and typed in italics, frequently at the beginning of a section. For example, one such notational convention is:

The names of functions and variables are written in lower case and underlined when they appear in the text. Meta-LISP notation is used for describing forms.

Examples: member[x;y] is equivalent to (MEMBER X Y), member[car[x];FOO] is equivalent to (MEMBER (CAR X) (QUOTE FOO)). Note that in meta-LISP notation lower case variables are evaluated, upper case quoted.

. notation is used to distinguish between cons and list.

e.g., if x=(A B C), (FOO x) is (FOO (A B C)), whereas (FOO . x) is (FOO A B C). In other words, x is cadr of (FOO x) but cdr of (FOO . x). Similarly, y is caddr of (FOO x y), but cddr of (FOO x . y). Note that this convention is in fact followed by the read program, i.e., (FOO . (A B C)) and (FOO A B C) read in as equal structures.

¹ includes the KA and KI versions of the PDP-10 running the TENEX operating system, and the KL-10 or DEC 2020 running the TOPS-20 operating system.

Other important conventions are:

TRUE in Interlisp means not NIL.

The purpose of this is to allow a single function to be used both for the computation of some quantity, and as a test for a condition. For example, the value of `member[x;y]` is either `NIL`, or the tail of `y` beginning with `x`. Similarly, the value of `or` is the value of its first TRUE, i.e., non-`NIL`, expression, and the value of `and` is either `NIL`, or the value of its last expression.

Although most lists terminate in `NIL`, the occasional list that ends in an atom, e.g., `(A B . C)` or worse, a number or string, could cause bizarre effects. Accordingly, we have made the following implementation decision:

All functions that iterate through a list, e.g., `member`, `length`, `mapc`, etc. terminate by an `nlistp` check, rather than the conventional null-check, as a safety precaution against encountering data types which might cause infinite `cdr` loops, e.g., strings, numbers, arrays.

Thus,

```
member[x;(A B . C)] = member[x;(A B)]
reverse[(A B . C)] = reverse[(A B)]
append[(A B . C);y] = append[(A B);y]
```

For users with an application requiring extreme efficiency,² we have provided fast versions of `memb`, `last`, `nth`, `assoc`, and `length` which compile open and terminate on `NIL` checks, and therefore may cause infinite `cdr` loops if given poorly formed arguments. However, to help detect these situations, `fmemb`, `flast`, `fnth`, `fassoc`, and `flength` all generate errors when interpreted if their argument ends in a non-list other than `NIL`, e.g., BAD ARGUMENT - FLAST.

Most functions that set system parameters, e.g., `printlevel`, `linelength`, `radix`, etc., return as their value the old setting. If given `NIL` as an argument, they return the current value without changing it.

All SUBRS, i.e., hand coded functions, such as `read`, `print`, `eval`, `cons`, etc., have "argument names" selected from `U`, `V`, `W`, `X`, `Y`, `Z`, as described under `arglist`, Section 8. However, for tutorial purposes, more suggestive names are used in the descriptions of these functions in the text.

Most functions whose names end in `p` are predicates, e.g., `numberp`, `tailp`, `exprp`; most functions whose names end in `q` are nlambda's, i.e., do not require quoting their arguments, e.g., `setq`, `defineq`, `nlsetq`.

"`x` is equal to `y`" means `equal[x;y]` is true, as opposed to "`x` is `eq` to `y`" meaning `eq[x;y]` is true, i.e., `x` and `y` are the same identical LISP pointer.

² A `NIL` check can be executed in only one instruction, an `nlistp` on Interlisp-10 requires about 8, although both generate only one word of code.

When new literal atoms are created (by the read program, pack, or mkatom), they are provided with a function definition cell initialized to NIL (Section 8), and a property list initialized to NIL (Section 7). The function definition cell is accessed by the functions getd and putd described in Section 8, and the property list by the functions getproplist and setproplist, (Section 7).

car of NIL and cdr of NIL are always NIL, and the system will resist attempts to change them.

Neither NIL nor T can be used as bound variables, i.e. used as the names of arguments in a lambda expression, or as a prog variable.

The term list refers to any structure created by one or more conses, i.e., it does not have to end in NIL. For example, (A . B) is a list. The function listp, Section 5, is used to test for lists. Note that not being a list does not necessarily imply an atom, e.g., strings and arrays are not lists, nor are they atoms. See Section 10.

Many system functions have extra optional arguments for internal use that are not described in the writeups. For example, readline is described as a function of one argument, but arglist[READLINE] is (RDTBL LINE LISPXFLG). These arguments are indicated in the manual by the appearance of "-" in the arglist, e.g. readline[rdtbl;-;]. In such cases, the user should just ignore the extra arguments.

Interlisp departs from LISP 1.5 and other LISP dialects in that car of a form is never evaluated. In other words, if car of a form is not an atom with a function definition, and not a function object, i.e., a list car of which is LAMBDA, NLAMBDA, or FUNARG, an error is generated. apply or apply* (Section 8) must be used if the name of a function is to be computed as, for example, when functional arguments are applied.

2.2 USING THE INTERLISP SYSTEM - AN OVERVIEW

Call Interlisp by typing LISP followed by a carriage return. Interlisp will type an identifying message, the date, and a greeting, followed by a "+". This prompt character indicates that the user is "talking to" the top level Interlisp executive, called evalqt, (for historical reasons), just as "@" indicates the user is talking to the operating system. evalqt calls lispx which accepts inputs in either eval or apply format: if just one expression is typed on a line, it is evaluated; if two expressions are typed, the first is apply-ed to the second. eval and apply are described in Section 8. In both cases, the value is typed, followed by + indicating Interlisp is ready for another input.

Interlisp is normally exited via the function logout, i.e., the user types LOGOUT(). However, typing control-C at any point in the computation returns control immediately to the operating system. The user can then continue his program with no ill effects with the CONTINUE command, even if he interrupted it during a garbage collection.³ Typing control-D at any point during a computation will return control to evalqt. If typed during a garbage collection, the garbage collection will first be completed, and then control will be returned to Interlisp's top level; otherwise, control returns immediately.

³ Typing START will attempt to re-enter Interlisp at top level evalqt. However, there are many situations where this might leave the system in an inconsistent state. Thus control-C START is definitely not advisable unless no other recourse is available (e.g. user accidentally turns all interrupts off).

When typing to the Interlisp read program, typing a control-Q⁴ will cause Interlisp to print "# #" and clear the line buffer, i.e., erase the entire line up to the last carriage return. Typing control-A⁵ erases the last character typed in, echoing a \ and the erased character. Control-A will not back up beyond the last carriage return. Typing control-W within a call to read or readline will erase the last *expression* typed, echoing a \\|. Control-W will back up to previous lines. Control-O can be used to *immediately* clear the output buffer, and < del > to *immediately* clear the input buffer.⁶ In addition, typing control-U⁷ will cause the Interlisp editor (Section 9) to be called on the expression being read, when the read is completed. Appendix 1 contains a list of all control characters, and a reference to that part of the manual where they are described. Section 16 describes how the system's interrupt characters can be disabled or redefined, as well as how the user can define his own interrupt characters.

Since the Interlisp read program is normally line-buffered to make possible the action of control-Q,⁸ the user must type a carriage return before any characters are delivered to the function requesting input, e.g.,

```
←E T2      9
T
```

However, the read program *automatically* supplies (and prints) this carriage return when a matching right parenthesis is typed, making it unnecessary for the user to do so, e.g.,

```
←CONS(A B)
(A . B)
```

The Interlisp read program treats square brackets as "super-parentheses": a right square bracket automatically supplies enough right parentheses to match back to the last left square bracket (in the expression being read), or if none has appeared, to match the first left parentheses, e.g.,

```
(A (B (C])=(A (B (C))),
(A [B (C (D] E)=(A (B (C (D))) E).
```

% is the universal escape character for read. Thus to input an atom containing a syntactic delimiter, precede it by %, e.g., AB%. (C or %%%. See Section 14 for more details.

↑V (control-V) can be used to type a control character that would otherwise interrupt the input process, e.g., control-D, control-C, etc. If the character following ↑V is A, B, ... or Z, the corresponding control character is input, e.g., ↑VA↑VB↑VC is the atom control-Acontrol-Bcontrol-C.

- + ⁴ control-U for Interlisp-10 on TOPS-20.
- + ⁵ < del > for Interlisp-10 on TOPS-20.
- ⁶ The action of control-Q takes place when it is *read*. If the user has "typed ahead" several inputs, control-Q will only affect at most the last line of input. < del > (control-Z on TOPS-20), however, is an interrupt character that will clear the entire input buffer as soon as it is *typed*, i.e., even during a garbage collection.
- + ⁷ control-N for Interlisp-10 on TOPS-20.
- ⁸ Except following control[T], see Section 14.
- ⁹ '2' is used throughout the manual to denote carriage-return.

\uparrow V followed by any other character has no effect, i.e., FOO \uparrow V1 and FOO1 are identical. For more details, see Appendix 1.

Typing ' immediately in front of any expression has the effect of quoting the expression, i.e., (A 'B C) is read as (A (QUOTE B) C). See Section 14.

At any point during input, the user can type ?= followed by carriage-return and be given the argument names and corresponding values (if any) of the expression (form) being typed. Typing just ? carriage-return will cause the user to receive information from the Interlisp manual about the expression or command being typed. In the following example underlined characters were typed by the user.

←(SETQ FOO (ELT 3 ?=)
A = 3
N =

(ELT 3 ?=

elt[a;n]

Value is nth element of the array a.
elt generates an error, ARG NOT ARRAY,
if a is not the beginning of an array.

(ELT 3 ↑W\3 A 3)

? is also a programmer assistant command for use following an error. In many cases, the programmer's assistant can analyze the cause of the error (see example below). If not, the programmer's assistant simply presents information about the error from the manual.

EXAMPLE

Most of the "basics" of on-line use of Interlisp, e.g., defining functions, error handling, editing, saving your work, etc., are illustrated in the following brief console session. Underlined characters were typed by the user.

1. The user calls Interlisp from the operating system, in this case Tenex, Interlisp prints a date, and a greeting. The prompt character ← indicates the user is at the top level of Interlisp.
2. The user defines a function, fact, for computing factorial of n. In Interlisp, functions are defined via DEFINE or DEFINEQ, (Section 8). Functions may independently evaluate arguments, or not evaluate them, and spread their arguments, or not spread them (Section 4). The function fact shown here is an example of an everyday run-of-the-mill function of one argument, which is evaluated. The function definition uses CLISP (Conversational LISP, Section 23). The CLISP will automatically be converted to LISP when the function is run.
3. The user "looks" at the function definition. Function definitions in Interlisp are stored in a special cell called the function definition cell, which is associated with the name of the function (Section 8). This cell is accessible via the two functions, getd and putd, (define and

defineq use putd). Note that the user typed an input consisting of a single expression, i.e., (GETD (QUOTE FACT)), which was therefore interpreted as a form for eval. The user could also have typed GETD(FACT).

4. The user runs his function. Two errors occur and corrections are offered by DWIM (Section 17). In each case, the user indicates his approval, DWIM makes the correction, i.e., actually changes the definition of fact, and then continues the computation.
5. An error occurs that DWIM cannot handle, and the system goes into a break. At this point, the user can type in expressions to be eval-ed or apply-ed exactly as at the top level. The prompt character ":" indicates that the user is in a break, i.e., that the context of his computation is available. In other words, the system is actually "within" or "below" the call to itimes in which the error occurred.
6. The user types in the break command, BT, which calls for a backtrace to be printed. In Interlisp, interpreted and compiled code (see Section 18 for discussion of the compiler) are completely compatible, and in both cases, the name of the function that was called, as well as the names and values of its arguments are stored on the stack. The stack can be searched and/or modified in various ways (see Section 12).

Break commands are discussed in Section 15, which also explains how the user can "break" a particular function, i.e., specify that the system go into a "break" whenever a certain function or functions are called. At that point the user can examine the state of the computation. This facility is very useful for debugging.

7. The user asks for the value of the variable n, i.e., the most recent value, or binding. The interpreter will search the stack for the most recent binding, and failing to find one, will obtain the top level value from the atom's value cell, which is car of the atom (Section 3). If there are no bindings, and the value cell contains the atom NOBIND, an unbound atom error is generated (Section 16).
8. The user types ?, a command to the programmer's assistant (Section 22). The p.a. looks at the error and the context and using its information about ITIMES, "explains" the error.
9. The user realizes his error, and calls the editor to fix it. (Note that the system is *still* in the break.) The editor is described at length and in detail in Section 9. It is an extremely useful facility of Interlisp. Section 9 begins with a simple introduction designed for the new user.
10. The user instructs the editor to replace all NIL's (in this case there is only one) by 1. The editor physically changes the expression it is operating on so when the user exits from the editor, his function, *as it is now being interpreted*, has been changed.

@LISP2 1

INTERLISP-10 14-JUL-78 ...

Good Evening.

←DEFINEQ((FACT (LAMBDDA (N) (IF N=0 THEN NIL ELSE N*(FACTT N-1]) 2
 (FACT)
 ←(GETD 'FACT) 3
 (LAMBDDA (N) (IF N=0 THEN NIL ELSE N*(FACTT N-1)))
 ←FACT(3) 4
 LAMBDDA {below FACT} -> LAMBDA ? Yes
 FACTT {in FACT} -> FACT ? Yes

NON-NUMERIC ARG 5
 NIL
 IN ITIMES

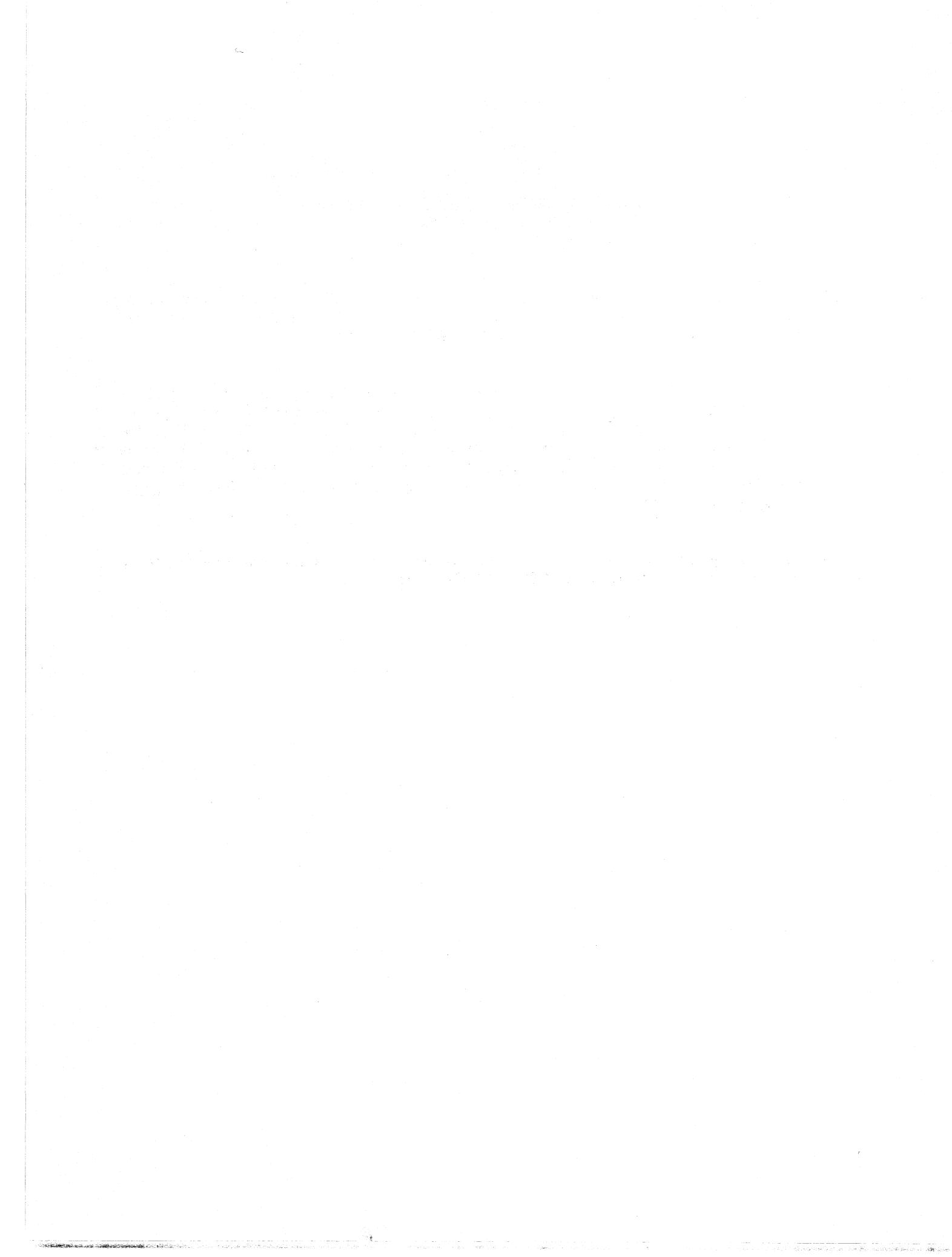
(broken)
 :BT2
 ITIMES 6
 COND
 FACT
 COND
 FACT
 COND
 FACT
 COND
 FACT
 TOP

:N2 7
 1
 :?2 8
 because ITIMES requires that each of its arguments be a number
 but in (ITIMES N (FACT (SUB1 N))) {IN FACT},
 the value of (FACT (SUB1 N)) is NIL when N=1 9
 :EDITF(FACT)
 EDIT
 *(R NIL 1) 10
 *OK2 11
 FACT
 :RETURN 12
 'BREAK' = 1
 6
 ←PP FACT2 13

(FACT
 [LAMBDA (N)
 (COND
 ((ZEROP 0)
 1)
 (T (ITIMES N (FACT (SUB1 N)))) 14
 FACT
 ←MAKEFILES()
 ****NOTE: The following are not contained on any file:
 the functions: FACT
 want to say where they go? Yes
 (functions)
 FACT File Name: FACT2
 new file? Yes +
 +
 +
 +
 +
 +
 +

- + FACT ...
- + (<TEITELMAN>FACT.;1)
- + ←LOGOUT()

11. The user exits from the editor and returns to the break.
12. The user specifies the value to be used by itimes in place of NIL by using the break command RETURN. This causes the computation to continue, and 6 is ultimately returned as the value of the original input, fact(3).
13. The user prettyprints (Section 14) fact, i.e., asks it be printed with appropriate indentations to indicate structure. Prettyprint also provides a comment facility. Note that both the changes made to fact by the editor and those made by DWIM are in evidence.
14. The user calls makefiles (Section 14) to write out any changes he has made. The file package knows that the function fact has been changed, but doesn't know on which file it belongs, and asks the user as shown in the example. After this interaction, the symbolic file <TEIITELMAN>FACT.;1 is written out. This file can be loaded into Interlisp at a later date via the function load (Section 14), will cause fact to be defined as it currently is. There is also a facility in Interlisp for saving and restoring an entire core image via the functions sysout and sysin (Section 14). *
15. The user logs out, returning control to Tenex. However, he can still continue his session by re-entering Interlisp via the Tenex CONTINUE command.



Index for Section 2

Page
Numbers

	Page Numbers
APPLY[FN;ARGS] SUBR	2.3
apply format	2.3
APPLY*[FN;ARG1;ARG2;...;ARGn] SUBR*	2.3
ARGLIST[FN]	2.2
backtrace	2.6
BAD ARGUMENT - FASSOC (error message)	2.2
BAD ARGUMENT - FLAST (error message)	2.2
BAD ARGUMENT - FLENGTH (error message)	2.2
BAD ARGUMENT - FMEMB (error message)	2.2
BAD ARGUMENT - FNTH (error message)	2.2
BT (break command)	2.6
car of NIL	2.3
carriage-return	2.4
clearing input buffer	2.4
clearing output buffer	2.4
CONTINUE (TENEX command)	2.3,9
CONTROL[FLG;RDTBL] SUBR	2.4
control characters	2.3-4
control-A	2.4
control-C	2.3
control-D	2.3
control-N (TOPS-20)	2.4
control-O	2.4
control-Q	2.4
control-U	2.4
control-U (TOPS-20)	2.4
control-V	2.4
control-W	2.4
control-Z (TOPS-20)	2.4
debugging	2.6
DEFINE[X;TYPE-IN]	2.5
DEFINEQ[X1;X2;...;Xn] NL*	2.5-6
dot notation	2.1
DWIM	2.6
EQ[X;Y] SUBR	2.2
eq	2.2
EQUAL[X;Y] SUBR	2.2
equal	2.2
escape character	2.4
EVAL[X] SUBR	2.3,6
eval format	2.3
EVALQT[LISPID]	2.3
FASSOC[KEY;ALST]	2.2
FLAST[X]	2.2
FLENGTH[X]	2.2
FMEMB[X;Y]	2.2
FNTH[X:N]	2.2
function definition cell	2.3,5
functional arguments	2.3
GETD[X] SUBR	2.3,5
GETPROPLIST[ATM]	2.3
GETTOPVAL[ATM] SUBR	2.3
interrupt characters	2.4
LINELENGTH[N] SUBR	2.2
line-buffering	2.4
LISTP[X] SUBR	2.3

Page
Numbers

lists	2.3
LOAD[FILE;LDFLG;PRINTFLG]	2.9
LOGOUT[] SUBR	2.3
meta-LISP notation	2.1
NIL	2.2
NLISTP[X]	2.2
NOBIND	2.6
null-check	2.2
predicates	2.2
PRETTYPRINT[FNS;PRETTYDEFLG;FNSLST]	2.9
PRINTLEVEL[CARN;CDRN] SUBR	2.2
prompt character	2.3,5-6
property list	2.3
pushdown list	2.6
PUTD[FN;DEF] SUBR	2.3,5
RADIX[N] SUBR	2.2
RETURN (break command)	2.9
SETPROPLIST[ATM;LST]	2.3
SETTOPVAL[ATM;VAL] SUBR	2.3
square brackets (use in input)	2.4
subrs	2.2
SYSIN[FILE] SUBR	2.9
SYSOUT[FILE] SUBR	2.9
TENEX	2.1,3,5,9
true	2.2
user interrupt characters	2.4
U.B.A. (error message)	2.6
variable bindings	2.6
## (printed by system)	2.4
% (escape character)	2.4
' (as a read-macro)	2.5
- (in argument list, in manual)	2.3
. notation	2.1
: (printed by system)	2.6
	2.4
 (TOPS-20)	2.4
? (prog. asst. command)	2.5
?= (edit command)	2.5
\ (printed by system)	2.4
\\" (printed by system)	2.4
] (use in input)	2.4
← (printed by system)	2.3,5

SECTION 3

DATA TYPES, STORAGE ALLOCATION GARBAGE COLLECTION, AND OVERLAYS

Interlisp operates in an 18-bit address space.¹ This address space is divided into 512 word pages with a limit of 512 pages, or 262,144 words, but only that portion of address space currently in use actually exists on any storage medium. Interlisp itself and all data storage are contained within this address space. A pointer to a data element such as a number, atom, etc., is simply the address of the data element in this 18-bit address space.

3.1 DATA TYPES

The data-types of Interlisp are lists, atoms, pnames, value cells, arrays, large and small integers, floating point numbers, stack pointers, string characters and string pointers. There is also a way to define new data-types described below in section 3.2. Compiled code, read tables, terminal tables, and hash arrays are currently included with arrays.

In the descriptions of the various data-types given below, for each data type, first the input syntax and output format are described, that is, what input sequence will cause the Interlisp read program to construct an element of that type, and how the Interlisp print program will print such an element. Next, those functions that construct elements of that data-type are given. Note that some data-types cannot be input, they can only be constructed, e.g. arrays. Finally, the format in which an element of that data-type is stored in memory is described.

3.1.1 LITERAL ATOMS

A literal atom is input as any string of non-delimiting characters that cannot be interpreted as a number. The syntactic characters that delimit atoms called separator or break characters (Section 14) and normally are space, end-of-line,² line-feed, % () "] and [. However, these characters may be included in atoms by preceding them with the escape character %.

Literal atoms are printed by print and prin2 as a sequence of characters with %'s inserted before all

¹ Interlisp is currently implemented on (or implementations are in progress for) at least four different machines. This section treats subjects that are for the most part somewhat implementation dependent. Where this is the case, the discussion refers to Interlisp-10, the implementation for the DEC PDP-10, on which Interlisp was first implemented.

² An end-of-line character is transmitted by TENEX when it sees a carriage-return.

delimiting characters (so that the atom will read back in properly). Literal atoms are printed by prinl as a sequence of characters without these extra %'s. For example, the atom consisting of the five characters A, B, C, (, and D will be printed as ABC%(D by print and ABC(D by prinl. The extra %'s are an artifact of the print program; they are not stored in the atom's pname.

Literal atoms can be constructed by pack, mkatom, and gensym (which uses mkatom).

Literal atoms are unique. In other words, if two literal atoms have the same pname, i.e. print the same, they will *always* be the same identical atom, that is, they will always have the same address in memory, or equivalently, they will always be eq.³ Thus if pack or mkatom is given a list of characters corresponding to a literal atom that already exists, they return a pointer to that atom, and do *not* make a new atom. Similarly, if the read program is given as input of a sequence of characters for which an atom already exists, it returns a pointer to that atom.

A literal atom is a datum consisting of the following components: a property list, initially NIL, (accessed by getproplist and setproplist, Section 7), a value, (accessed by getatomval and setatomval, Section 5), a function definition, initially NIL, (accessed by getd and putd, Section 8), and a pname, (not directly accessible).

3.1.2 PNAMES

The pnames of atoms⁴ comprise another data-type with storage assigned as it is needed. This data-type only occurs as a component of an atom or a string. It does not appear, for example, as an element of a list.

Pnames have no input syntax or output format as they cannot be directly referenced by user programs.

A pname is a sequence of 7 bit characters packed 5 to a word, beginning at a word boundary. The first character of a pname contains its length; thus the maximum length of a pname is 126 characters.

+ 3.1.3 VALUE CELLS

+ The value cells of atoms comprise another data-type with storage assigned as it is needed. Value
+ cells occur only as part of an atom. If an atom is ever referenced as a variable, either by setq, or
+ by use as the argument of a function, or bound as a prog variable, a value cell is created and
+ assigned to the atom. The value cell will then always contain the current value of the atom. Note
+ that from the standpoint of a function that references the value of an atom, an atom having no

³ Note that this is *not* true for strings, large integers, floating point numbers, and lists, i.e. they all can print the same without being eq.

⁴ All Interlisp pointers have pnames, since we define a pname simply to be how that pointer is printed. However, only literal atoms and strings have their pnames explicitly stored. Thus, the use of the term pname in a discussion of data-types or storage allocation means pnames of atoms or strings, and refers to a sequence of characters stored in a certain part of Interlisp's memory.

value cell is indistinguishable from an atom whose value cell contains the atom NOBIND.⁵

Value cells have no input syntax or output format as they cannot be directly referenced by user programs.

3.1.4 NUMERICAL ATOMS

Numerical atoms, or simply numbers, do not have property lists, value cells, functions definition cells, or explicit pnames. There are currently two types of numbers in Interlisp: integers, and floating point numbers.

INTEGERS

The input syntax for an integer is an optional sign (+ or -) followed by a sequence of digits, followed by an optional Q.⁶ If the Q is present, the digits are interpreted in octal, otherwise in decimal, e.g. 77Q and 63 both correspond to the same integers, and in fact are indistinguishable internally since no record is kept of how integers were created.

The setting of radix (Section 14), determines how integers are printed: signed or unsigned, octal or decimal.

Integers are created by pack and mkatom when given a sequence of characters observing the above syntax, e.g. (PACK (LIST 1 2 (QUOTE Q))) = 10. Integers are also created as a result of arithmetic operations, as described in Section 13.

An integer is stored in one 36 bit word; thus its magnitude must be less than 2^{135} .⁷ To avoid having to store (and hence garbage collect) the values of small integers, a few pages of address space, overlapping the Interlisp-10 machine language code, are reserved for their representation. The small number pointer *itself*, minus a constant, is the value of the number. Currently the range of "small" integers is -1536 thru +1535. The predicate smallp is used to test whether an integer is "small".

While small integers have a unique representation, large integers do not. In other words, two large integers may have the same value, but not the same address in memory, and therefore not be eq. For this reason the function icqp (or equal) should be used to test equality of large integers.

⁵ The latter case can occur for example if the user types SET(FOO exp), then undoes (Section 22) the set operation, thereby restoring the atom to its original "value".

⁶ and terminated by a delimiting character. Note that some data-types are self-delimiting, e.g. lists.

⁷ If the sequence of digits used to create the integer is too large, the high order portion is discarded. (The handling of overflow as a result of arithmetic operations is discussed in Section 13.)

FLOATING POINT NUMBERS

A floating point number is input as a signed integer, followed by a decimal point, followed by another sequence of digits called the fraction, followed by an exponent (represented by E followed by a signed integer).⁸ Both signs are optional, and either the fraction following the decimal point, or the integer preceding the decimal point may be omitted. One or the other of the decimal point or exponent may also be omitted, but at least one of them must be present to distinguish a floating point number from an integer. For example, the following will be recognized as floating point numbers:

5.	5.00	5.01	.3	5E2	5.1E2
		5E-3	-5.2E+6		

Floating point numbers are printed using the facilities provided by TENEX. Interlisp-10 calls the floating point number to string conversion routines⁹ using the format control specified by the function flfmt (Section 14).¹⁰ flfmt is initialized to T, or free format. For example, the above floating point numbers would be printed free format as:

5.0	5.0	5.01	.3	500.0	510.0
		.005	-5.2E6		

Floating point numbers are also created by pack and mkatom, and as a result of arithmetic operations as described in Section 13.

A floating point number is stored in one 36 bit word in standard PDP-10 format. The range is $\pm 2.94\text{E-}39$ thru $\pm 1.69\text{E}38$ (or 2^{128} thru 2^{127}).

3.1.5 LISTS

The input syntax for a list is a sequence (at least one)¹¹ of Interlisp data elements, e.g. literal atoms numbers, other lists, etc. enclosed in parentheses or brackets. A bracket can be used to terminate several lists, e.g. (A (B (C]], as described in Section 2.

If there are two or more elements in a list, the final element can be preceded by a . (delimited on both sides), indicating that cdr of the final node in the list is to be the element immediately following the ., e.g. (A . B) or (A B C . D), otherwise cdr of the last node in a list will be NIL.¹² Note that the input sequence (A B C . NIL) is thus equivalent to (A B C), and that (A B . (C D)) is thus equivalent to (A B C D). Note however that (A B . C D) will create a list containing the five literal atoms A B . C and D.

⁸ and terminated by a delimiter.

⁹ Additional information concerning these conversions may be obtained from the TENEX JSYS Manual.

+ ¹⁰ The printnum package (Section 14) permits greater controls on the printed appearance of floating point numbers, allowing such things as left-justification, suppression of trailing decimals, etc.

¹¹ () is read as the atom NIL.

¹² Note that in Interlisp terminology, a list does *not* have to end in NIL, it is simply a structure composed of one or more conses.

Lists are constructed by the primitive functions cons and list.

Lists are printed by printing a left parenthesis, and then printing the first element of the list,¹³ then printing a space, then printing the second element, etc. until the final node is reached. Lists are considered to terminate when cdr of some node is not a list. If cdr of this terminal node is NIL (the usual case), car of the terminal node is printed followed by a right parenthesis. If cdr of the terminal node is *not* NIL, car of the terminal node is printed, followed by a space, a period, another space, cdr of the terminal node, and then the right parenthesis. Note that a list input as (A B C . NIL) will print as (A B C), and a list input as (A B . (C D)) will print as (A B C D). Note also that printlevel affects the printing of lists to teletype, and that carriage returns may be inserted where dictated by linelength, as described in Section 14.

A list is stored as a chain of list *nodes*. A list node is stored in one 36 bit word, the right half containing car of the list (a pointer to the first element of the list), and the left half containing cdr of the list (a pointer to the next node of the list).

3.1.6 ARRAYS

An array in Interlisp is a one dimensional block of contiguous storage of arbitrary length. Arrays do not have input syntax; they can only be created by the function array. Arrays are printed by both print, prin2, and prinl, as # followed by the address of the array pointer (in octal). Array elements can be referenced by the functions elt and eltd, and set by the functions seta and setd, as described in Section 10.

Arrays are partitioned into four sections: a header, a section containing unboxed numbers, a section containing Interlisp pointers, and a section containing relocation information. The last three sections can each be of arbitrary length (including 0); the header is two words long and contains the length of the other sections as indicated in the diagram below. The unboxed number region of an array is used to store 36 bit quantities that are not Interlisp pointers, and therefore not to be chased from during garbage collections, e.g. machine instructions. The relocation information is used when the array contains the definition of a compiled function, and specifies which locations in the *unboxed* region of the array must be changed if the array is moved during a garbage collection.

The format of an array in Interlisp-10 is as follows:

¹³ The individual elements of a list are printed using prin2 if the list is being printed by print or prin2, and by prinl if the list is being printed by prinl.

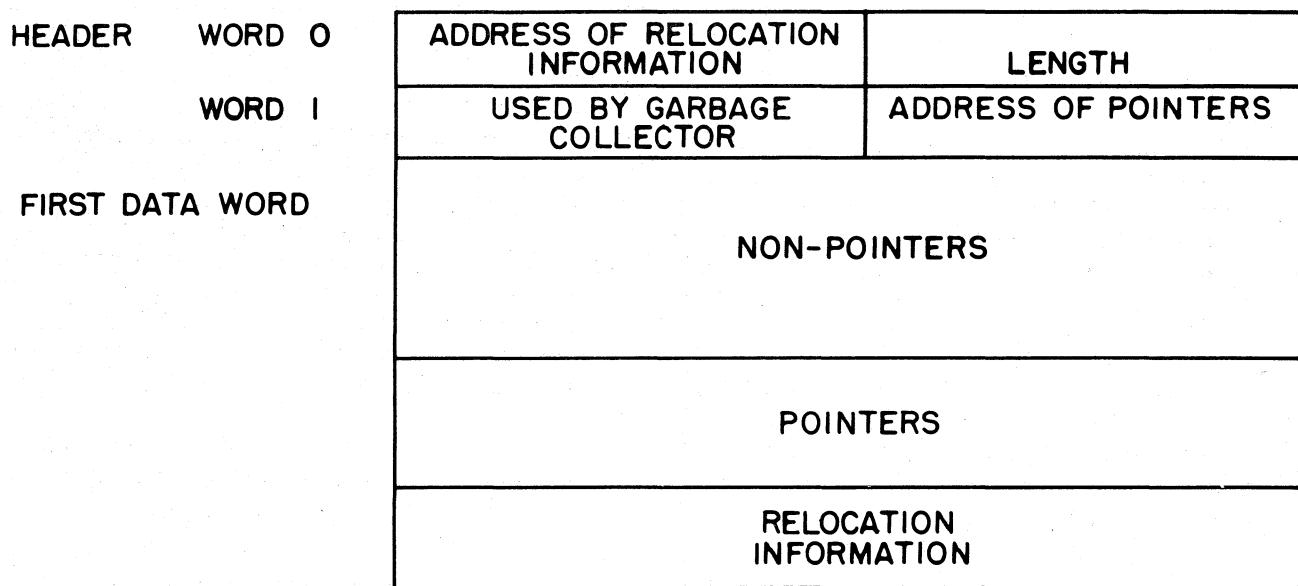


FIGURE 3-1

The header contains:

- word 0 right - length of entire block = ARRSIZE+2.
- left - address of relocation information relative to word 0 of block (> 0 if relocation information exists, negative if array is a hash array, 0 if ordinary array).
- word 1 right - address of pointers relative to word 0 of block.
- left - used by garbage collector.

3.1.7 STRINGS

The input syntax for a string is a ", followed by a sequence of any characters except " and %, terminated by a ". " and % may be included in a string by preceding them with the escape character %.

Strings are printed by print and prin2 with initial and final "'s, and %'s inserted where necessary for it to read back in properly. Strings are printed by prinl without the delimiting "'s and extra %'s.

Strings are created by mkstring, substring, and concat.

Internally a string is stored in two parts; a string pointer and the sequence of characters. The Interlisp pointer to a string is the address of the string pointer. The string pointer, in turn, contains the character position at which the string characters begin, and the number of characters.

String pointers and string characters are two separate data-types,¹⁴ and several string pointers may reference the same characters. This method of storing strings permits the creation of a substring by creating a new string pointer, thus avoiding copying of the characters. For more details, see Section 10.

String characters are 7 bit bytes packed 5 to a word. The format of a string pointer is:



FIGURE 3-2

The maximum length of a string is 32K ($K = 1024$) characters.

3.2 USER DEFINED DATA TYPES¹⁵

In addition to the several built-in data-types such as atoms, lists, arrays, etc., Interlisp provides a way of defining completely *new* classes of objects, with a fixed number of *fields* determined by the definition of the data-type. Facilities are provided for declaring the number and *type* of the fields for a given class, creating objects of a given class, accessing and replacing the contents of each of the fields of such an object, as well as interrogating such objects.

In order to define a new class of objects, the user must supply a name for the new data-type and specifications for each of its fields. Each field may contain either a pointer (i.e., any arbitrary Interlisp datum), an integer, a floating point number, or an n-bit integer.

The above operations are accomplished via the function declaredatatype.

declaredatatype[typename; fieldspecs; -]

typename is a literal atom, which specifies the name of the data-type. fieldspecs is a list of *field specifications*. Each field specification may be one of the following:

POINTER	field may contain any Interlisp datum
FIXP	field contains an integer

¹⁴ String characters are not directly accessible by user programs.

¹⁵ The most convenient way to define new data-types is via DATATYPE declarations in the RECORD package (as described in Section 23).

	FLOATP (BITS n)	field contains a floating point number field contains a non-negative integer less than 2^n .
	<u>declareddatatype</u>	returns a list of <i>field descriptors</i> , one for each element of <i>fieldspecs</i> . The field descriptor contains information about where within the datum the field is actually stored. If <i>typename</i> is already declared a datatype, it is re-declared. If <i>fieldspecs</i> is NIL, <i>typename</i> is "undeclared".
fetchfield[descriptor;datum]	Returns the contents of the field described by <i>descriptor</i> from <i>datum</i> . <i>descriptor</i> must be a "field descriptor" as returned by <u>declareddatatype</u> . If <i>datum</i> is not an instance of the datatype of which <i>descriptor</i> is a descriptor, causes error DATUM OF INCORRECT TYPE. ¹⁶	
replacefield[descriptor;datum;newvalue]	Store <i>newvalue</i> into the field of <i>datum</i> described by <i>descriptor</i> . <i>descriptor</i> must be a field descriptor, as returned by <u>declareddatatype</u> . If <i>datum</i> is not an instance of the datatype of which <i>descriptor</i> is a descriptor, causes error DATUM OF INCORRECT TYPE. Value is <i>newvalue</i> .	
+ ncreate[typename;from]	creates and returns a new instance of datatype <i>typename</i> .	
+ + +	If <i>from</i> is also a datum of datatype <i>typename</i> , the fields of the new object are initialized to the values of the corresponding fields in <i>from</i> .	
+ +	If <i>typename</i> is not the type name of a previously declared user ¹⁷ data type, generates an error, ILLEGAL DATA TYPE.	
getfieldspecs[typename]	Returns a list which is equal to the <i>fieldspecs</i> argument given to <u>declareddatatype</u> for <i>typename</i> ; if <i>typename</i> is not a currently declared data-type, returns NIL.	
getdescriptors[typename]	Returns a list of field descriptors, equal to the value of <u>declareddatatype</u> for <i>typename</i> .	

¹⁶ In Interlisp-10, if *descriptor* is quoted, fetchfield compiles open. This capability is used by the record package.

+ ¹⁷ i.e., ncreate will not work for built in datatypes, such as ARRAYP, STRINGP, etc.

`userdatatypes[]` Returns list of names of currently declared user datatypes.

In Interlisp-10, user datatypes are allocated starting with type number 31. The minimum amount of space that can be assigned to any one data-type is one page, thus effectively limiting the number of possible data types. The fields are rearranged internally so that pointer fields come first; the remaining numeric fields, if any, are packed so as to optimize the space used.

Note that the user can define how user datatypes are to be printed via defprint (Section 14), and how they are to be evaluated by the interpreter via defeval (Section 8), and how they are to be compiled by the compiler via compiletypelst (Section 18).

+
+
+

3.3 STORAGE ALLOCATION AND GARBAGE COLLECTION

In the following discussion, we will speak of a quantity of memory being assigned to a particular data-type, meaning that the space is reserved for storage of elements of that type. *Allocation* will refer to the process used to obtain from the already assigned storage a particular location for storing one data element.

A small amount of storage is assigned to each data-type when Interlisp-10 is started; additional storage is assigned only during a garbage collection.

The page is the smallest unit of memory that may be assigned for use by a particular data-type. For each page of memory there is a one word entry in a type table. The entry contains the data-type residing on the page as well as other information about the page. The type of a pointer is determined by examining the appropriate entry in the type table.

Storage is allocated as is needed by the functions which create new data elements, such as cons, pack, mkstring. For example, when a large integer is created by iplus, the integer is stored in the next available location in the space assigned to integers. If there is no available location, a garbage collection is initiated, which may result in more storage being assigned.

The storage allocation and garbage collection methods differ for the various data-types. The major distinction is between the types with elements of fixed length and the types with elements of arbitrary length. List *nodes*, atoms, large integers, floating point numbers, and string pointers are fixed length; all occupy 1 word except atoms which use 3 words. Arrays, pnames, and strings (string characters) are variable length.

Elements of fixed length types are stored so that they do not overlap page boundaries. Thus the pages assigned to a fixed length type need not be adjacent. If more space is needed, any empty page will be used. The method of *allocating* storage for these types employs a free-list of available locations; that is, each available location contains a pointer to the next available location. A new element is stored at the first location on the free-list, and the free-list pointer is updated.¹⁸

Elements of variable length data-types are allowed to overlap page boundaries. Consequently all pages assigned to a particular variable length type must be contiguous. Space for a new element is allocated following the last space used in the assigned block of contiguous storage.

¹⁸ The allocation routine for list nodes is more complicated. Each page containing list nodes has a separate free list. First a page is chosen (see CONS for details), then the free list for that page is used. Lists are the only data-type which operate this way.

When Interlisp-10 is first called, a few pages of memory are assigned to each data-type. When the allocation routine for a type determines that no more space is available in the assigned storage for that type, a garbage collection is initiated. The garbage collector determines what data is currently in use and reclaims that which is no longer in use. A garbage collection may also be initiated by the user with the function reclaim (Section 10).

Data in use (also called active data) is any data that can be "reached" from the currently running program (i.e., variable bindings and functions in execution) or from atoms. To find the active data the garbage collector "chases" all pointers, beginning with the contents of the push-down lists and the components (i.e., car, cdr, and function definition cell) of all atoms with at least one non-trivial component.

When a previously unmarked datum is encountered, it is marked, and all pointers contained in it are chased. Most data-types are marked using bit tables; that is tables containing one bit for each datum. Arrays, however, are marked using a half-word in the array header.

When the mark and chase process is completed, unmarked (and therefore unused) space is reclaimed. Elements of fixed length types that are no longer active are reclaimed by adding their locations to the free-list for that type. This free list allocation method permits reclaiming space without moving any data, thereby avoiding the time consuming process of updating all pointers to moved data. To reclaim unused space in a block of storage assigned to a variable length type, the active elements are compacted toward the beginning of the storage block, and then a scan of all active data that can contain pointers to the moved data is performed to update the pointers.¹⁹

Whenever a garbage collection of any type is initiated,²⁰ unused space for all fixed length types is reclaimed since the additional cost is slight. However, space for a variable length type is reclaimed only when that type initiated the garbage collection.

If the amount of storage reclaimed for the type that initiated the garbage collection is less than the minimum free storage requirement for that type, the garbage collector will assign enough additional storage to satisfy the minimum free storage requirement. The minimum free storage requirement for each data type may be set with the function minfs (Section 10). The garbage collector assigns additional storage to fixed length types by finding empty pages, and adding the appropriate size elements from each page to the free list. Assigning additional storage to a variable length type involves finding empty pages and moving data so that the empty pages are at the end of the block of storage assigned to that type.

In addition to increasing the storage assigned to the type initiating a garbage collection, the garbage collector will attempt to minimize garbage collections by assigning more storage to other fixed length types according to the following algorithm.²¹ If the amount of active data of a type has increased since the last garbage collection by more than 1/4 of the minfs value for that type,

+ 19 If Interlisp-10 types the message ARRAYS FOULED during a garbage collection, it means that an array header has been clobbered and no longer makes sense. This can be due to hardware malfunction, or an as yet undiscovered bug in Interlisp. The best thing to do under these circumstances is to give up and start over with a fresh system or sysout.

20 The "type of a garbage collection" or the "type that initiated a garbage collection" means either the type that ran out of space and called the garbage collector, or the argument to reclaim.

21 We may experiment with different algorithms.

storage is increased (if necessary), to attain the minfs value. If active data has increased by less than 1/4 of the minfs value, available storage is increased to 1/2 minfs. If there has been no increase, no more storage is added. For example, if the minfs setting is 2000 words, the number of active words has increased by 700, and after all unused words have been collected there are 1000 words available, 1024 additional words (two pages) will be assigned to bring the total to 2024 words available. If the number of active words had increased by only 300, and there were 500 words available, 512 additional words would be assigned.

3.4 SHARED INTERLISP-10

The Interlisp-10 system initially obtained by the user is shared; that is, all active users of Interlisp-10 are actually using the same pages of memory. As a user adds to the system, private pages are added to his memory. Similarly, if the user changes anything in the original shared Interlisp-10, for example, by advising a system function, a private copy of the changed page is created.

In addition to the swapping time saved by having several users accessing the same memory, the sharing mechanism permits a large saving in garbage collection time, since we do not have to garbage collect any data in the shared system, and thus do not need to chase from any pointers on shared pages during garbage collections.

This reduction in garbage collection time is possible because the shared system usually is not modified very much by the user. If the shared system is changed extensively, the savings in time will vanish, because once a page that was initially shared is made private, every pointer on it must be assumed active, because it may be pointed *to* by something in the shared system. Since every pointer on an initially shared but now private page can also point to *private* data, they must always be chased.

A user may create his own shared system with the function makesys. If several people are using the same system, making the system be shared will result in a savings in swapping time. Similarly, if a system is large and seldom modified, making it be shared will result in a reduction of garbage collection time, and may therefore be worthwhile even if the system is only being used by one user.

makesys[file]

creates a saved file in which all pages in this system, including private user pages, are made read execute, i.e. shared. This system can then be run via the TENEX command RUN, or GET and START.

For example, new Interlisp-10 systems are brought up by loading the appropriate compiled files and then performing makesys[LISP.SAV].²²

herald[string]

makes string be the 'herald' for the system, i.e. the message printed when the system is first started. Primarily for use in conjunction

²² makesys is also advised (see section 19) to set the variable makesysdate to (DATE), i.e. the time and date the system was made.

with makesys.²³

3.5 THE INTERLISP-10 SWAPPER²⁴

Interlisp-10 provides a very large auxiliary address space exclusively for swappable arrays (primarily compiled function definitions). In addition to the 256K of resident address space, this "shadow space" can currently accomodate an additonal 256K words, can easily be expanded to 3.5 million words, and with some further modifications, could be expanded to 128 million words. Thus, the overlay system provides essentially unlimited space for compiled code.²⁵

Shadow space and the swapper are intended to be more or less transparent to the user. However, this section is included in the manual to give programmers a reasonable feeling for what overlays are like, without getting unnecessarily technical, as well as to document some new functions and system controls which may be of interest for authors of exceptionally large systems.

3.4.1 OVERLAYS

The shadow space is a very large auxiliary address space used exclusively for an Interlisp data-type called a swappable array. The regular address space is called the "resident" space to distinguish it from shadow space. Any kind of resident array - compiled code, pointer data, binary data, or a hash array - can be copied into shadow space ("made swappable"), from which it is referred to by a one-word resident entity called a handle. The resident space occupied by the original array can then be garbage collected normally (assuming there are no remaining pointers to it, and it has not been made shared by a makesys). Similarly, a swappable array can be made resident again at any time, but of course this requires (re)allocating the necessary resident space.

The main purpose and intent of the swapping system is to permit utilization of swappable arrays directly and interchangeably with resident arrays, thereby saving resident space which is then available for other data-types, such as lists, atoms, strings, etc.

This is accomplished as follows: A section of the resident address space is permanently reserved for

²³ makesys is advised to set the variable heraldstring to the concatenation of "Interlisp-10", the month and day of the makesys, and "..." and to call herald on this string. Alternatively, makesys can be given as a second argument a string to be used instead of "Interlisp-10", e.g. makesys[STREK.SAV;STAR-TREK] would cause the message STAR-TREK followed by the date and "..." to be printed when STREK.SAV was run.

²⁴ The Interlisp-10 swapper was designed by E. L. Wegbreit (PARC) and J. W. Goodwin (BBN), and implemented by J. W. Goodwin.

²⁵ Since compiled code arrays point to atoms for function names, and strings for error messages, not to mention the fact that programs usually have data base, which are typically lists rather than arrays, there is still a very real and finite limit to the total size of programs that Interlisp-10 can accomodate. However, since much of the system and user compiled code can be made swappable, there is that much more resident space available for these other data-types.

a swapping buffer.²⁶ When a particular swappable array is requested, it is brought (swapped) in by mapping or overlays the pages of shadow space in which it lies onto a section of the swapping buffer. This process is the swapping or overlaying from which the system takes its name. The array is now (directly) accessible. However, further requests for swapping could cause the array to be overlaid with something else, so in effect it is liable to go away at any time. Thus all system code that relates to arrays must recognize handles as a special kind of array, fetch them into the buffer (if not already there), when necessary check that they have not disappeared, fetch them back in if they have, and even be prepared for the second fetch to bring the swappable array in at a different place than did the first.

The major emphasis in the design of the overlay system has been placed on running compiled code, because this accounts for the overwhelming majority of arrays in typical systems, and for as much as 60% of the overall data and code. The system supports the running of compiled code directly from the swapping buffer, and the function calling mechanism knows when a swappable definition is being called, finds it in the buffer if it is already there, and brings it in otherwise. Thus, from the user's point of view, there is no need to distinguish between swappable and resident compiled definitions, and in fact ccodep will be true for either.

3.4.2 EFFICIENCY

Once of the most important design goals for the overlay system was that swappable code should not execute any extra instructions compared to resident code, once it had been swapped in. Thus, the instructions of a swappable piece of code are identical (except for two instructions at the entry point) to those of the resident code from which it was copied,²⁷ and similarly when a swappable function calls another function (of any kind) it uses the exact same calling sequence as any other code. Thus, all costs associated with running of swappable code are paid at the point of entry (both calling and returning).²⁸

The cost of the swapping itself, i.e. the fetch of a new piece of swapped code into the buffer, is even harder to measure meaningfully, since two successive fetches of the same function are not the same, due to the fact that the instance created by the first fetch is almost certain to be resident when the second is done, if no swapping is done in between. Similarly, two successive PMAP's (the Tenex operation to fetch one page) are not the same from one moment to another, even if the virtual state of both forks is exactly the same - a difficult constraint to meet in itself.²⁹ Thus, all that can be reported is that empirical measurements and observations have shown no consistent slowdown in performance of systems containing swappable functions viz a viz resident functions.

²⁶ Initially 64 512 word pages, but can be changed via the function setsbsize described below.

²⁷ The relocatable instructions are indexed by a base register, to make them run equally well at any location in the buffer. The net slowdown due to this extra level of indirection is too small to measure accurately in the overall running of a program. On analytical grounds, one would expect it to be around 2%.

²⁸ If the function in question does nothing, e.g. a compiled (LAMBDA NIL NIL), it costs approximately twice as much to enter its definition if it is swappable as compared to resident. However, very small functions are normally not made swappable (see mkswapp, page 3.14), because they don't save much space, and are (typically) entered frequently. Larger programs don't exhibit a measurable slow down since they amortize the entry cost over longer runs.

²⁹ The cost of fetching is probably not in the mapping operation itself but in the first reference to the page, which has a high probability of faulting. This raises the problem of measuring page fault activity, another morass of uncertainty.

3.4.3 SPECIFICATIONS

Associated with the overlay system is a datatype called a swparray, (type name SWPARRAYP), which occupies one word of resident space, plus however much of shadow space needed for the body of the array. arglist, fntyp, nargs, gctd, putd, argtype, arraysize, changename, calls, break, advise, and edita all work equally well with swappable as resident programs. ccodep is true for all compiled functions/definitions.

swparrayp[x]

Analogous to arrayp. Returns x if x is a swappable array and, NIL otherwise.

+ **scodep[x]**

analogous to ccodep. Returns T if x is or has a swapped compiled definition.

mkswap[x]

If x is a resident array, returns a swappable array which is a copy of x. If x is a literal atom and ccodep[x] is true, its definition is copied into a swappable array, and it is (undoably) redefined with the latter. The value of mkswap is x.

mkunswap[x]

the inverse of mkswap. x is either a swappable array, or an atom with swapped definition on its CODE property.

mkswapp[fname;cdef]

All compiled definitions begin life as resident arrays, whether they are created by load, or by compiling to core. Before they are stored away into their atom's function cell, mkswapp is applied to the atom and the array. If the value of mkswapp is T, the definition is made swappable; otherwise, it is left resident. By reddefining mkswapp or advising it, the user can completely control the swappability of all future definitions as they are created. The initial definition of mkswapp will make a function swappable if (1) noswapflg is NIL, and (2) the name of the function is not on noswapfns, and (3) the size of its definition is greater than mkswapsize words, initially 128.

setsbsize[n]

Sets the size of the swapping buffer to n, a number of pages. Returns the previous value. setsbsize[] returns the current size without changing it.³⁰

30

Currently, the system lacks error recovery routines for situations such as a call to a swappable function which is too big for the swapping buffer, or when the size is zero. Therefore, setsbsize should be used with care.

Index for Section 3

	Page Numbers
ARRAY[N;P;V] SUBR	3.5
array header	3.5
array pointer	3.5
ARRAYP[X] SUBR	3.14
arrays	3.5,1,9-10
ARRAYS FOULED (error message)	3.10
atoms	3.1,9
BITS (as a field specification)	3.8
carriage-return	3.1
CCODEP[FN] SUBR	3.13-14
CODE (property name)	3.14
compacting	3.10
CONCAT[X1;X2;...;Xn] SUBR*	3.6
CONS[X;Y] SUBR	3.5,9
data-types	3.1,2-7,9
DATUM OF INCORRECT TYPE (error message)	3.8
DECLAREDATATYPE[TYPENAME;FIELDSPECS;FLG]	3.7
E (in a floating point number)	3.4
ELT[A;N] SUBR	3.5
ELTD[A;N] SUBR	3.5
end-of-line	3.1
EQP[X;Y] SUBR	3.3
escape character	3.1
FETCHFIELD[DESCRIPTOR;DATUM]	3.8
FIXP (as a field specification)	3.7
floating point numbers	3.4,1,3,9
FLOATP (as a field specification)	3.8
FLTfmt[FORMATBITS]	3.4
free-list	3.9-10
garbage collection	3.9-11
GENSYM[CHAR]	3.2
GETDESCRIPTIONS[TYPENAME]	3.8
GETFIELDSPECS[TYPENAME]	3.8
handle	3.12
hash arrays	3.1
HERALD[STRING] SUBR	3.11
HERALDSTRING (system variable/parameter)	3.12
ILLEGAL DATA TYPE (error message)	3.8
integers	3.3
large integers	3.3,1,9
LINELENGTH[N] SUBR	3.5
line-feed	3.1
LIST[X1;X2;...;Xn] SUBR*	3.5
list nodes	3.6,9
lists	3.4,1
literal atoms	3.1,2
MAKESYS[FILE] SUBR	3.11
MAKESYSDATE (system variable/parameter)	3.11
MINFS[N;TYPE]	3.10-11
MKATOM[X] SUBR	3.2-4
MKSTRING[X] SUBR	3.6,9
MKSWAP[X]	3.14
MKSWAPP[FNAME;CDEF]	3.14
MKSWAPSIZE (Overlay variable/parameter)	3.14
MKUNSWAP[X]	3.14
NCREATE[TYPENAME;FROM]	3.8

Page
Numbers

NOBIND	3.3
NOSWAPFNS (Overlay variable/parameter)	3.14
octal	3.3,5
overlays	3.12-14
PACK[X] SUBR	3.2-4,9
page	3.9
pnames	3.2,1,9
pointer	3.1
POINTER (as a field specification)	3.7
PRINT[X;FILE;RDTBL] SUBR	3.1,5-6
PRINTLEVEL[CARN;CDRN] SUBR	3.5
PRIN1[X;FILE] SUBR	3.2,5-6
PRIN2[X;FILE;RDTBL] SUBR	3.1,5-6
private pages	3.11
Q (following a number)	3.3
RADIX[N] SUBR	3.3
RECLAIM[TYPE]	3.10
relocation information (in arrays)	3.5
REPLACEFIELD[DESCRIPTOR;DATUM;NEWVALUE]	3.8
RUN (TENEX command)	3.11
SCODEP[FN] SUBR	3.14
SETA[A;N;V]	3.5
SETD[A;N;V]	3.5
SETSBSIZE[N] SUBR	3.14
shared pages	3.11
shared system	3.11
sharing	3.11
small integers	3.3,1
SMALLP[N]	3.3
space	3.1
storage allocation	3.9
string characters	3.7,1,6,9
string pointers	3.7,1,9
strings	3.6
SUBSTRING[X;N;M;OLDPTR] SUBR	3.6
swappable array	3.12
swapping buffer	3.13
SWPARRAY[N;P;V] SUBR	3.14
SWPARRAYP[X] SUBR	3.14
TENEX	3.1,4,11
unboxed numbers (in arrays)	3.5
USERDATATYPES[]	3.9
value cell	3.2
[.....	3.1
"	3.6,1
# (followed by a number)	3.5
% (escape character)	3.1,6
(.....	3.1
()	3.4
)	3.1
.	3.4
: (in a floating point number)	3.4
] (use in input)	3.1

SECTION 4

FUNCTION TYPES AND IMPLICIT PROGN

In Interlisp, each function may independently have:

- a. its arguments evaluated or not evaluated;
- b. a fixed number of arguments or an indefinite number of arguments;
- c. be defined by an Interlisp expression, by built-in machine code, or by compiled machine code.

Hence there are twelve function types ($2 \times 2 \times 3$).

4.1 EXPRS

Functions defined by Interlisp expressions are called exprs. Exprs must begin with either **LAMBDA** or **NLAMBDA**,¹ indicating whether the arguments to the function are to be evaluated or not evaluated, respectively. Following the **LAMBDA** or **NLAMBDA** in the expr is the "argument list", which is either

- (1) a list of literal atoms or **NIL** (fixed number of arguments); or
- (2) any literal atom other than **NIL**, (indefinite number of arguments).²

Case (1) corresponds to a function with a *fixed* number of arguments. Each atom in the list is the *name* of an argument for the function defined by this expression. When the function is called, its arguments will be evaluated or not evaluated, as dictated by whether the definition begins with **LAMBDA** or **NLAMBDA**, and then paired with these argument names.³ This process is called

¹ Where unambiguous, the term expr is used to refer to either the function, or its definition.

² Anything else will cause an ARG NOT LITATOM error, e.g. (**LAMBDA "FOO" --**), or (**LAMBDA (X Z --**). In addition, if **NIL** or **T** is used as an argument name, the error ATTEMPT TO BIND NIL OR T is generated. +

³ Note that the function itself can evaluate selected arguments by calling eval. In fact, since the function type can specify only that all arguments are to be evaluated or none are to be evaluated, if it is desirable to write a function which only evaluates *some* of its arguments, e.g. setq, the function is defined as an **nlambda**, i.e. no arguments are evaluated in the process of calling the function, and then included in the definition itself are the appropriate calls to eval. In this case, the user should also put on the property list of the function under the property **INFO** the value **EVAL** to inform the various system packages such as DWIM, CLISP, PRINTSTRUCTURE, etc., that this function in fact *does* evaluate its arguments, even though it is an **nlambda**.

"spreading" the arguments, and the function is called a spread-LAMBDA or a spread-NLAMBDA, or simply a spread function.

Case (2) corresponds to a function with an *indefinite* number of arguments. Such a function is called a nospread function. If its definition begins with NLAMBDA, the atom which constitutes its argument list is bound to the list of arguments to the function (unevaluated). For example, if FOO is defined by (NLAMBDA X --), when (FOO THIS IS A TEST) is evaluated, X will be bound to (THIS IS A TEST).

If a nospread function begins with a LAMBDA, indicating its arguments are to be evaluated, each of its n arguments are evaluated and their values stored on the pushdown list. The atom following the LAMBDA is then bound to the *number* of arguments which have been evaluated. For example, if FOO is defined by (LAMBDA X --) when (FOO A B C) is evaluated, A, B, and C are evaluated and X is bound to 3. A built-in function, arg[atm;m], is available for computing the value of the mth argument for the lambda-atom variable atm. arg is described in section 8.

4.2 COMPILED FUNCTIONS

Functions defined by expressions can be compiled by the Interlisp compiler, as described in section 18, "The Compiler and Assembler". In Interlisp-10, functions may also be written directly in machine code using the ASSEMBLE directive of the compiler. Functions created by the compiler, whether from S-expressions or ASSEMBLE directives, are referred to as compiled functions. In Interlisp-10, compiled functions may be resident or swappable, as described in section 3.

4.3 FUNCTION TYPE

The function fntyp returns the function type of its argument. The value of fntyp is one of the following 12 types:

EXPR	CEXPR	SUBR
FEXPR	CFEXPR	FSUBR
EXPR*	CEXPR*	SUBR*
FEXPR*	CFEXPR*	FSUBR*

The types in the first column are all defined by expressions. The types in the second column are compiled versions of the types in the first column, as indicated by the prefix C. In the third column are the parallel types for built-in subroutines. Functions of types in the first two rows have a fixed number of arguments, i.e., are spread functions. Functions in the third and fourth rows have an indefinite number of arguments, as indicated by the suffix *. The prefix F indicates no evaluation of arguments. Thus, for example, a CFEXPR* is a compiled form of a nospread-NLAMBDA.

A standard feature of the Interlisp system is that no error occurs if a spread function is called with too many or too few arguments. If a function is called with too many arguments, the extra arguments are evaluated but ignored. If a function is called with too few arguments, the unsupplied ones will be delivered as NIL. In fact, the function itself cannot distinguish between being given NIL as an argument, and not being given that argument, e.g., (FOO) and (FOO NIL) are exactly the same for spread functions.

4.4 PROGN

progn is a function of an arbitrary number of arguments. progn evaluates the arguments in order and returns the value of the last, i.e., it is an extension of the function prog2 of LISP 1.5. Both cond and lambda/nlambda expressions have been generalized to permit "implicit progns" as described below.

4.5 IMPLICIT PROGN

The conditional expression has been generalized so that each clause may contain n forms ($n \geq 1$) which are interpreted as follows:

```
(COND
  (P1 E11 E12 E13)
  (P2 E21 E22) [1]
  (P3)
  (P4 E41))
```

will be taken as equivalent to (in LISP 1.5):

```
(COND
  (P1 (PROGN E11 E12 E13))
  (P2 (PROGN E21 E22)) [2]
  (P3 P3)
  (P4 E41)
  (T NIL))
```

Note however that P3 is evaluated only once in [1], while it is evaluated a second time if the expression is written as in [2]. Thus a clause in a cond with only a predicate and no following expression causes the value of the predicate itself, if non-NIL, to be returned. Note also that NIL is returned if all the predicates have value NIL, i.e., the cond "falls off the end". No error is generated.

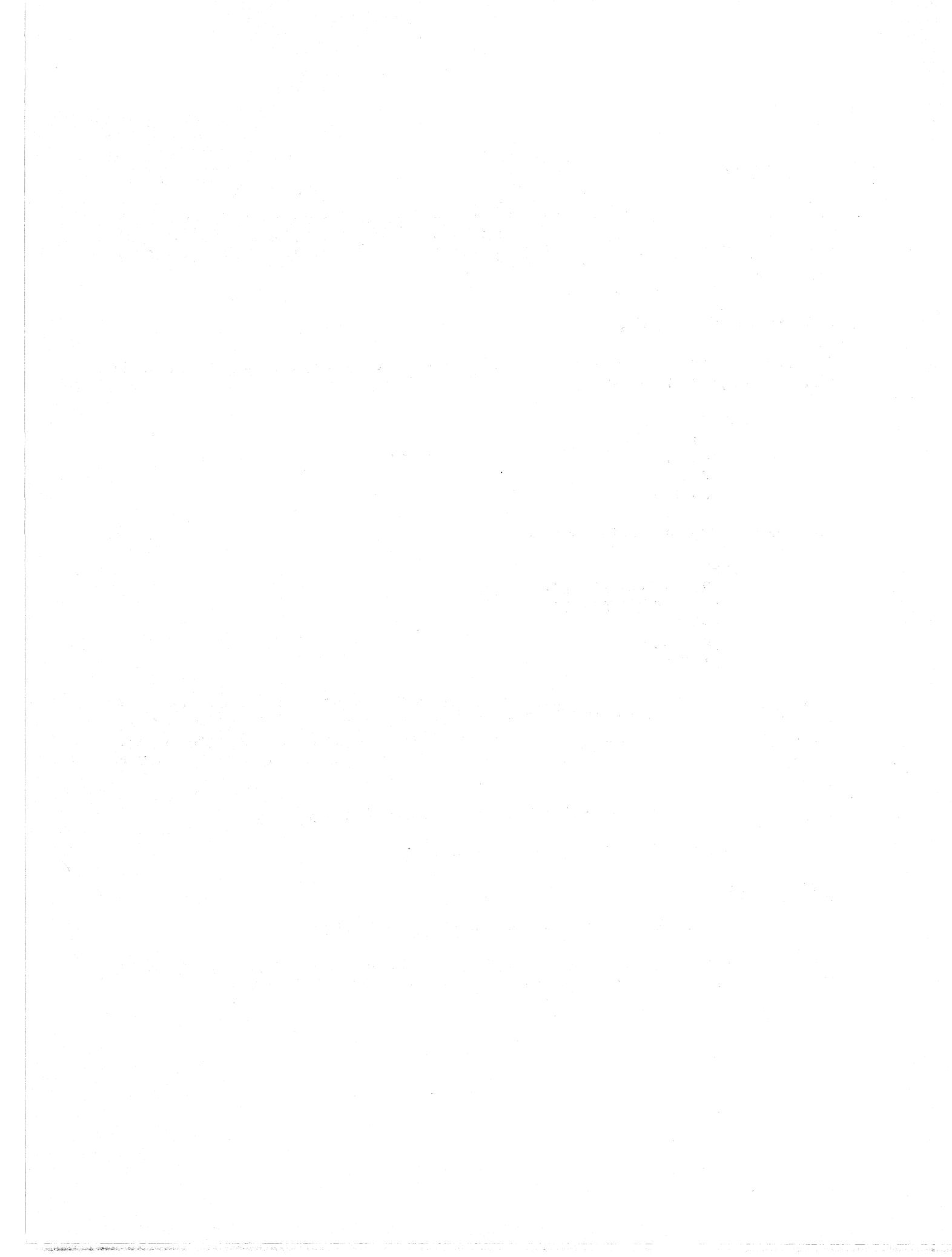
LAMBDA and NLAMBDA expressions also allow implicit progn's; thus for example:

```
(LAMBDA (V1 V2) (F1 V1) (F2 V2) NIL)
```

is interpreted as:

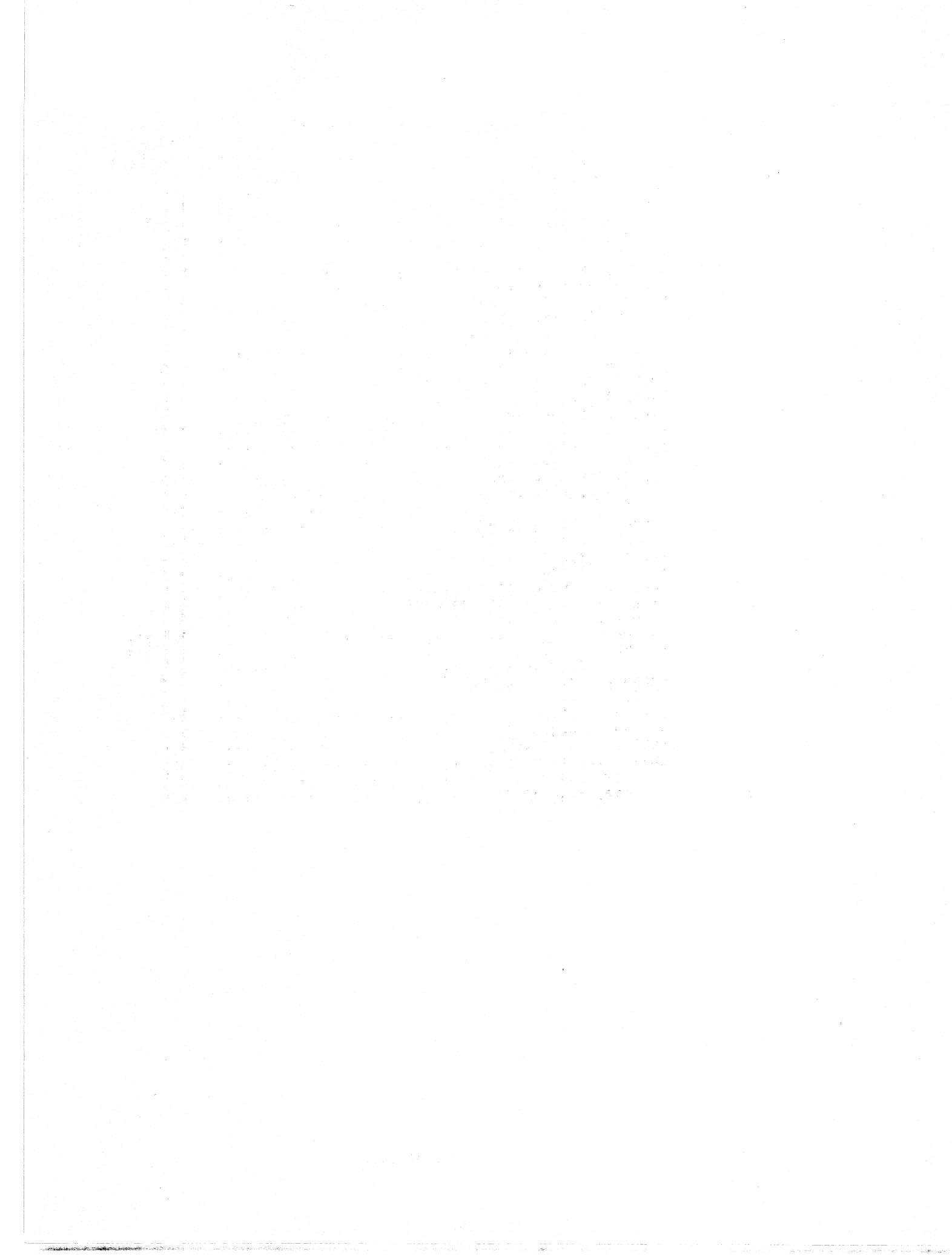
```
(LAMBDA (V1 V2) (PROGN (F1 V1) (F2 V2) NIL))
```

The value of the last expression following LAMBDA (or NLAMBDA) is returned as the value of the entire expression. In this example, the function would always return NIL.



Index for Section 4

	Page Numbers
ARG[VAR;M] FSUBR	4.2
ARG NOT LITATOM (error message)	4.1
argument evaluation	4.1-2
argument list	4.1
ASSEMBLE	4.2
ATTEMPT TO BIND NIL OR T (error message)	4.1
CEXPR (function type)	4.2
CEXPR* (function type)	4.2
CFEXPR (function type)	4.2
CFEXPR* (function type)	4.2
compiled functions	4.2
compiler	4.2
COND[C1;C2;...;Cn] FSUBR*	4.3
EVAL[X] SUBR	4.1
EXPR (function type)	4.2
exprs	4.1
EXPR* (function type)	4.2
FEXPR (function type)	4.2
FEXPR* (function type)	4.2
fixed number of arguments	4.1
FNTYP[FN]	4.2
FSUBR (function type)	4.2
FSUBR* (function type)	4.2
function types	4.1-2
implicit progn	4.3
incorrect number of arguments	4.2
indefinite number of arguments	4.2
INFO (property name)	4.1
LAMBDA	4.1,2-3
NLAMBDA	4.1,2-3
nospread functions	4.2
PROGN[X1;X2;...;Xn] FSUBR*	4.3
pushdown list	4.2
spread functions	4.2
spreading arguments	4.2
SUBR (function type)	4.2
SUBR* (function type)	4.2
too few arguments	4.2
too many arguments	4.2



SECTION 5

PRIMITIVE FUNCTIONS AND PREDICATES

5.1 PRIMITIVE FUNCTIONS

car[x]

car gives the first element of a list x, or the left element of a dotted pair x. car of NIL is always NIL. For all other nonlists, e.g., atoms, strings, arrays, and numbers, the value is undefined (and in some implementations may generate an error).

cdr[x]

cdr gives the rest of a list (all but the first element). This is also the right member of a dotted pair. cdr of NIL is always NIL. The value of cdr is undefined for other nonlists.

caar[x] = car[car[x]]
cadr[x] = car[cdr[x]]
cdddr[x] =
cdr[cdr[cdr[cdr[x]]]]

All 30 combinations of nested cars and cdrs up to 4 deep are included in the system. All are compiled open by the compiler.

cons[x;y]

cons constructs a dotted pair of x and y. If y is a list, x becomes the first element of that list. To minimize drum accesses the following algorithm is used in Interlisp-10, for finding a page on which to put the constructed Interlisp word.

cons[x;y] is placed

- 1) on the page with y if y is a list and there is room; otherwise
- 2) on the page with x if x is a list and there is room; otherwise
- 3) on the same page as the last cons if there is room; otherwise
- 4) on any page *with a specified minimum* of storage, presently 16 LISP words.

conscount[-]

value is the number of conses since this Interlisp was started up.

rplacd[x;y]

Places the pointer y in cdr of the cell pointed to by x. Thus it

physically changes the internal list structure of x, as opposed to cons which creates a new list element. The only way to get a circular list is by using rplacd to place a pointer to the beginning of a list in a spot at the end of the list.

The value of rplacd is x. An attempt to rplacd NIL will cause an error, ATTEMPT TO RPLAC NIL, (except for rplacd[NIL;NIL]). An attempt to rplacd any other non-list will cause an error ARG NOT LIST.

rplaca[x;y]

similar to rplacd, but replaces the address pointer of x, i.e., car, with y. The value of rplaca is x. An attempt to rplaca NIL will cause an error, ATTEMPT TO RPLAC NIL, (except for rplaca[NIL;NIL]). An attempt to rplaca any other non-list will cause an error, ARG NOT LIST.

Convention: Naming a function by prefixing an existing function name with f usually indicates that the new function is a fast version of the old, i.e., one which has the same definition but compiles open and runs without any "safety" error checks.

frplacd[x;y]

Has the same definition as rplacd but compiles open as one instruction. Note that no checks are made on x, so that a compiled frplacd can clobber NIL, producing strange and wondrous effects.

frplaca[x;y]

Similar to frplacd.

+ rplnode[x;a;d]

performs rplaca[x;a], rplacd[x;d], and returns x.

+ rplnode2[x;y]

performs rplaca[x;car[y]], rplacd[x;cdr[y]] and returns x.

+ frplnode[x;a;d]

fast version of rplnode that compiles open with no error checks.

+ frplnode2[x;y]

fast version of rplnode2 that compiles open with no error checks.

quote[x]

This is a function that prevents its arguments from being evaluated. Its value is x itself, e.g., (QUOTE FOO) is FOO.¹

¹ Since giving quote more than one argument, e.g., (QUOTE EXPR (CONS X Y)), is almost always a parentheses error, and one that would otherwise go undetected, quote itself generates an error in this case, PARENTHESIS ERROR.

kwote[x]

Value is an expression which when evaluated yields x. If x is NIL or a number, this is x itself. Otherwise, (LIST (QUOTE QUOTE) x). For example, if x=A, and y=B, then
(KWOTE (CONS x y)) = (QUOTE (A . B)).

cond[c₁;c₂;...;c_k]

The conditional function of Interlisp, cond, takes an indefinite number of arguments c₁, c₂, ... c_k, called clauses. Each clause c_i is a list (e_{1i} ... e_{ni}) of n > 1 items, where the first element is the predicate, and the rest of the elements the consequents. The operation of cond can be paraphrased as
IF e₁₁ THEN e₂₁ ... e_{n1} ELSEIF e₁₂ THEN e₂₂ ... e_{n2} ELSEIF e₁₃ ...

The clauses are considered in sequence as follows: the first expression e_{1i} of the clause c_i is evaluated and its value is classified as false (equal to NIL) or true (not equal to NIL). If the value of e_{1i} is true, the expressions e_{2i} ... e_{ni} that follow in clause c_i are evaluated in sequence, and the value of the conditional is the value of e_{ni}, the last expression in the clause. In particular, if n=1, i.e., if there is only one expression in the clause c_i, the value of the conditional is the value of e_{1i}. (which is evaluated only once).

If e_{1i} is false, then the remainder of clause c_i is ignored, and the next clause c_{i+1} is considered. If no e_{1i} is true for any clause, the value of the conditional expression is NIL.

selectq[x;clause₁;clause₂;...;clause_n;default]

selects a form or sequence of forms based on the value of its first argument x. Each clause_i is a list of the form (s_i e_{1i} e_{2i} ... e_{ki}) where s_i is the selection key. The operation of selectq can be paraphrased as:

**IF x=s₁ THEN e₁₁ ... e_{k1}
ELSEIF x=s₂ THEN ... ELSE default.**

If s_i is an atom, the value of x is tested to see if it is eq to s_i (not evaluated). If so, the expressions e_{1i} ... e_{ki} are evaluated in sequence, and the value of the selectq is the value of the last expression evaluated, i.e., e_{ki}.

If s_i is a list, the value of x is compared with each element (not evaluated) of s_i, and if x is eq to any one of them, then e_{1i} to e_{ki} are evaluated in turn as above.

If clause_i is not selected in one of the two ways described, clause_{i+1} is tested, etc., until all the clause's have been tested. If none is selected, the value of the selectq is the value of default. default must be present.

An example of the form of a selectq is:

```
[SELECTQ (CAR X)
  (Q (PRINT FOO)
    (FIE X))
  ((A E I O U)
    (VOWEL X))
  (COND
    ((NULL X)
      NIL)
    (T (QUOTE STOP))]
```

which has two cases, Q and (A E I O U) and a default condition which is a cond.

selectq compiles open, and is therefore very fast; however, it will not work if the value of x is a list, a large integer, or floating point number, since selectq uses eq for all comparisons.

prog1[x₁; x₂; ...; x_n]

evaluates its arguments in order, that is, first x₁, then x₂, etc, and returns the value of its first argument x₁, e.g., (PROG1 X (SETQ X Y)) sets x to y, and returns x's original value.

progn[x₁; x₂; ...; x_n]

progn evaluates each of its arguments in order, and returns the value of its last argument as its value. progn is used to specify more than one computation where the syntax allows only one, e.g., (SELECTQ ... (PROGN ...)) allows evaluation of several expressions as the default condition for a selectq.

prog[varslst; e₁; e₂; ...; e_n]

This function allows the user to write an ALGOL-like program containing Interlisp expressions (forms) to be executed. The first argument, varslst, is a list of local variables (must be NIL if no variables are used). Each atom in varslst is treated as the name of a local variable and bound to NIL. varslst can also contain lists of the form (atom form). In this case, atom is the name of the variable and is bound to the value of form. The evaluation takes place before any of the bindings are performed, e.g., (PROG ((X Y) (Y X)) ...) will bind x to the value of y and y to the (original) value of x. An attempt to use anything other than a literal atom as a prog variable will cause an error, ARG NOT LITATOM. An attempt to use NIL or T as a prog variable will cause an error, ATTEMPT TO BIND NIL OR T.

+
+
+
+

The rest of the prog is a sequence of non-atomic statements (forms) and atomic symbols, i.e. litatoms, used as labels for go. The forms are evaluated sequentially; the labels serve only as markers. The two special functions go and return alter this flow of control as described below. The value of the prog is usually specified by the function return. If no return is executed, i.e., if the prog "falls off the end," the value of the prog is NIL.

go[x] *go* is the function used to cause a transfer in a prog. (GO L) will cause the program to continue at the label L. A go can be used at any level in a prog. If the label is not found, go will search higher progs *within the same function*, e.g., (PROG -- A -- (PROG -- (GO A))). If the label is not found in the function in which the prog appears, an error is generated, UNDEFINED OR ILLEGAL GO.

return[x] A return is the normal exit for a prog. Its argument is evaluated and is the value of the prog in which it appears.

If a go or return is executed in an interpreted function which is not a prog, the go or return will be executed in the last interpreted prog entered if any, otherwise cause an error.

go or return inside of a compiled function that is not a prog is not allowed, and will cause an error at compile time.

As a corollary, go or return in a functional argument, e.g., to sort, will not work compiled. Also, since nlsetq's and ersetq's compile as *separate* functions, a go or return *cannot* be used inside of a compiled nlsetq or ersetq if the corresponding prog is outside, i.e., above, the nlsetq or ersetq.

set[x;y] This function sets x to y. Its value is y. If x is not a literal atom, causes an error, ARG NOT LITATOM. If x is NIL, causes an error, ATTEMPT TO SET NIL. Note that set is a normal lambda-spread function, i.e., its arguments are evaluated before it is called. Thus, if the value of x is c, and the value of y is b, then set[x;y] would result in c having value b, and b being returned as the value of set.

setq[x;y] An nlambda version of set: the first argument is not evaluated, the second is.² Thus if the value of X is C and the value of Y is B, (SETQ X Y) would result in X (not C) being set to B, and B being returned. If x is not a literal atom, an error is generated, ARG NOT LITATOM. If x is NIL, the error ATTEMPT TO SET NIL is generated.

setq[x;y] Like setq except that neither argument is evaluated, e.g., (SETQQ X (A B C)) sets x to (A B C).

gettopval[atm] returns top level value of atm from its value cell (even if NOBIND), regardless of any intervening bindings. Interpreted, generates an error, ARG NOT LITATOM, if atm is not a literal atom.

² Since setq is an nlambda, *neither* argument is evaluated during the calling process. However, setq itself calls eval on its second argument. Note that as a result, typing (SETQ var form) and SETQ(var form) to lispX is equivalent: in both cases var is not evaluated, and form is.

settopval[atm;val]

Sets top level value of atm, regardless of any intervening bindings, i.e., stores val in value cell of atm. Value is val. Interpreted, generates an error ATTEMPT TO SET NIL, if atm=NIL, or ARG NOT LITATOM, if atm is not a literal atom.

- + Note: gettopval and settopval are rarely used in a system which employs shallow binding such as Interlisp-10, as it is more economical to simply rebind variables. In a deep bound system, gettopval and settopval are very efficient since they do not have to search the stack, but can simply access the value cell directly. In a shallow bound system, the opposite is true: gettopval and settopval have to search the stack for rebinding and hence can be quite expensive.
- + In order to provide a convenient way for both deep and shallow bound systems of referring to what are conceptually global variables, the following functions are provided:
- + **getatomval[atm]** in shallow system, such as Interlisp-10, same as evalv[atm], or simply atm. In deep bound system, same as gettopval[atm].
- + **setatomval[atm;val]** in shallow bound system, same as set, in deep bound system, same as settopval.
- + In other words, getatomval and setatomval always access a variable's value cell,³ regardless of whether the system is deep or shallow bound.

rpaq[x;y]

An nlambda function like setq, except always works on top level value of x, i.e., on the value cell.

rpaqq[x;y]

An nlambda function like setqq for top level values.

rpaq and rpaqq are used by the file package (Section 14). Both rpaq and rpaqq generate errors if x is not a literal atom. Both are affected by the value of dfnflg (Section 8). If dfnflg=ALLPROP (and the value of x is other than NOBIND), instead of setting x, the corresponding value is stored on the property list of x under the property VALUE. Both are undoable.

- + **addtovar[var;x₁;x₂;...;x_n]** nlambda, nospread function. Each x_i that is not a member of the value of var is added to it, i.e. after addtovar completes, the value of var will be union[list[x₁;x₂;...;x_n];var]. addtovar is used by prettydef for implementing the ADDVARS command. It performs some file package (see Section 14) related operations, i.e. "notices" that var has been changed. Value is var (not the value of var).

³ As described in Section 3, an atom which has not previously been used as a variable does not have a value cell in Interlisp-10. However, from the standpoint of getatomval and evalv the atom looks like it has a value cell which contains the atom NOBIND.

CHANGING AND RESTORING SYSTEM STATE

In Interlisp, a computation can be interrupted/aborted at any point due to an error, or more forcefully, because a control-D was typed, causing return to the top level. This situation creates problems for programs that need to perform a computation with the system in a "different state", e.g., different radix, input file, readable, etc. but want to "protect" the calling environment, i.e., be able to restore the state when the computation has completed. While errors can be "caught" by errorsets, control-D cannot.⁴ Thus the system may be left in its changed state as a result of the computation being aborted. The following functions address this problem:⁵

resetlst[resetx]

nlambda, nospread. resetx is a list of forms. resetlst sets up an errorset so that any reset operations performed by resetsave (see below) are restored when the evaluation of resetx has been completed (or an error occurs, or a control-D is typed). If no error occurs, the value of resetlst is the value of the last form on resetx, otherwise resetlst generates an error (after performing the necessary restorations). resetlst compiles open.

resetsave[resetx]

nlambda, nospread function for use under a resetlst.⁶ If car of resetx is atomic, resets the top level value of car of resetx to the value of cadr of resetx,⁷ e.g., (RESETSAVE LISPXHISTORY EDITHISTORY) resets the value of lispxhistory to be edithistory and provides for the original value of lispxhistory to be restored when the resetlst completes operation, (or an error occurs, or a control-D is typed).

If car of resetx is not atomic, it is a form that is evaluated. If cdr of resetsave is NIL, e.g., (RESETSAVE (RADIX 8)), the form must return as its value its "former state", so that the effect of evaluating the form can be reversed, and the system state can be restored, by applying car of the form to the value of the form,⁸ e.g., (RESETSAVE (RADIX 8)) performs (RADIX 8), and provides for radix to be reset to its original value when the resetlst completes by applying radix to the value returned by (RADIX 8).

⁴ i.e., not conveniently. The program could of course redefine control-D as a userinterrupt, check for it, reenable it, and call reset or something similar.

⁵ Note that these functions do not (cannot) handle the situation where their environment is exited via anything other than a normal return, an error, or a reset. E.g. a releval, reffrom, resume, etc., will never be seen.

⁶ resetsave can be called when *not* under a resetlst. In this case, the restoration will be performed at the next RESET, i.e., control-D or call to reset. In other words, there is an "implicit" resetlst at the top level in evalqt.

⁷ This use is somewhat anachronistic in Interlisp-10 in that in a shallow bound system, it is sufficient to simply rebind the variable. Furthermore, if there are any rebindings, the resetsave will *not* affect the most recent binding but will change only the top level value, and therefore probably not have the intended effect.

⁸ except if car of the form is setq, the setq is transparent for the purposes of resetsave, i.e. the user could also have written (RESETSAVE (SETQ X (RADIX 8))), and restoration would be performed by applying radix, not setq, to the previous value of radix.

For functions which do not return their "previous setting", the restoring expression can be specified as the value of the second argument to resetsave, which in this case is evaluated *before* the first argument, e.g.,

[RESETSAVE(SETBRK --)(LIST(QQUOTE SETBRK))(GETBRK]⁹
will restore the break characters by applying setbrk to the value returned by (GETBRK), which was computed before the (SETBRK --) expression was evaluated.

(RESETSAVE NIL form) is permissible. It simply specifies that the value of form be treated as a restoration expression, e.g.,
(RESETSAVE NIL (LIST (QUOTE CLOSEF) FILE)) will cause file to be closed when the resetlst that the resetsave is under completes (or an error occurs or a control-D is typed).

resetsave compiles open. Its value is not a "useful" quantity.

resetvar[var;newvalue;form]

Nlambda function. Simplified form of resetlst and resetsave for resetting and restoring global variables.¹⁰ Equivalent to (RESETLST (RESETSAVE var newvalue) form), e.g.,
(RESETVAR LISPXHISTORY EDITHISTORY (FOO)) resets lispxhistory to the value of edithistory while evaluating (FOO). resetvar compiles open. If no error occurs, its value is the value of form.

- + resetvars[varslist;e₁;e₂;...;e_n]
- + nlambda function, similar to prog, except the variables in varslist are global variables. In a shallow bound system (Interlisp-10) resetvars and prog are identical.¹¹ In a deep bound system, each variable is "rebound" using resetsave.
- + resetvars, like getatomval and scatomval, is provided to permit compatibility (i.e. transportability) between a shallow bound and deep bound system with respect to conceptually global variables.

resetform[resetform;form₁;form₂;...;form_n]

Nlambda function. Simplified form of resetlst and resetsave for resetting a system state when the corresponding function returns as its value the "previous setting." Equivalent to (RESETLST (RESETSAVE resetform) form₁ form₂ ... form_n), e.g., (RESETFORM (RADIX 8) (FOO)). resetform compiles open. If no error occurs, its value is the value returned by form_n.

⁹ Note that the restoration expression is still "evaluated" by *applying* its car to its cdr.

+ ¹⁰ Unnecessarily expensive in a shallow bound system as the variable can simply be rebound.

+ ¹¹ except that the compiler insures that variables bound in a resetvars are declared as SPECVARS (see Section 18).

For some applications, the restoration operation must be different depending on whether the computation completed successfully or was aborted by an error or control-D. To facilitate this, while the restoration operation is being performed, the value of resetstate will be bound to NIL, ERROR, or RESET, depending on whether the exit was normal, due to an error, or reset (i.e., control-D, or in Interlisp-10, control-C followed by reenter). For example,

```
(RESETLST (RESETSAVE (INFILE X) (LIST '[LAMBDA (FL)
(AND (EQ RESETSTATE 'RESET) (CLOSEF FL) (DELFILE FL] X))
. forms)
```

will cause X to be closed and deleted only if a control-D was typed during the execution of forms.

For convenience in specifying complicated restoring expressions, the variable oldvalue is bound at the time the restoring operation is performed to the value of the saving expression. For example,

```
(RESETLST (RESETSAVE (INPUT FL) '(AND RESETSTATE (INPUT OLDVALUE)))12
. forms)
```

will restore the primary input file if an error or control-D occurs.

In addition, the function resetundo, in conjunction with resetlst and resetsave, provides a way of specifying that the system be restored to its prior state by *undoing* the side effects of the computations performed under the resetlst. Undoing and resetundo are described in Section 22.

5.2 PREDICATES AND LOGICAL CONNECTIVES

atom[x] is T if x is an atom; NIL otherwise.

litatom[x] is T if x is a literal atom, i.e., an atom and not a number, NIL otherwise.

numberp[x] is x if x is a number, NIL otherwise.

Convention: Functions that end in p are usually predicates, i.e., they test for some condition.

stringp[x] is x if x is a string, NIL otherwise.¹³

¹² As mentioned earlier, restoring is performed by applying car of the restoring expression to cdr. This particular example works because and applied to (RESETSTATE (INPUT OLDVALUE)) is the same as evaling (AND RESETSTATE (INPUT OLDVALUE)). progn also has this property. Note that without using oldvalue, the user would have to write

```
(RESETLST
  (SETQ TEM (INPUT FL))
  (RESETSAVE NIL (LIST '(LAMBDA (FL) (AND RESETSTATE (INPUT FL)))
    TEM))
  forms)
```

¹³ For other string functions, see Section 10.

arrayp[x]	is <u>x</u> if <u>x</u> is an array, NIL otherwise.
listp[x]	is <u>x</u> if <u>x</u> is a list-structure, i.e., one created by one or more <u>conses</u> ; NIL otherwise.
<i>Note that arrays and strings are not atoms, but are also not lists, i.e., both <u>atom</u> and <u>listp</u> will return NIL when given an array or a string.</i>	
nlistp[x]	not[listp[x]]
eq[x;y]	The value of <u>eq</u> is T, if <u>x</u> and <u>y</u> are pointers to the same structure in memory, and NIL otherwise. <u>eq</u> is compiled open by the compiler. Its value is not guaranteed T for equal numbers which are not small integers. See <u>eqp</u> .
neq[x;y]	The value of <u>neq</u> is T, if <u>x</u> is not <u>eq</u> to <u>y</u> , and NIL otherwise.
null[x]	eq[x;NIL]
not[x]	same as <u>null</u> , that is eq[x;NIL].
nill[]	value is NIL. <u>nill</u> is useful for those cases where one wants to supply a functional argument which will always return NIL.
+ negate[x]	returns the negation of <u>x</u> . For example: negate[(MEMBER X Y)]=(NOT (MEMBER X Y)), negate[(EQ X Y)]=(NEQ X Y), negate[(AND X (NLISTP X))]=(OR (NULL X) (LISTP X)), etc.
eqp[x;y]	The value of <u>eqp</u> is T if <u>x</u> and <u>y</u> are <u>eq</u> , i.e., pointers to the same structure in memory, or if <u>x</u> and <u>y</u> are numbers and are equal in value. ¹⁴ Its value is NIL otherwise.
equal[x;y]	The value of <u>equal</u> is T (1) if <u>x</u> and <u>y</u> are <u>eq</u> , i.e., pointers to the same structure in memory; or (2) <u>eqp</u> , i.e., numbers with equal value; or (3) <u>strequal</u> , i.e., strings containing the same sequence of characters; or (4) lists and <u>car</u> of <u>x</u> is <u>equal</u> to <u>car</u> of <u>y</u> , and <u>cdr</u> of <u>x</u>

14 For more discussion of eqp and other number functions, see Section 13.

is equal to cdr of y.¹⁵ The value of equal is NIL otherwise. Note that x and y do *not* have to be eq.

equalall[x;y] like equal, except always descends to atomic levels, i.e. will compare the contents of arrays, hash arrays, user data types, etc. + +

eqlength[x;n] equivalent to equal[length[x];n], but more efficient, i.e. stops as soon as it knows that x is longer than n. Note that eqlength is safe to use on circular lists, since it is "bounded" by n. + + +

equaln[x;y;depth] like equal, for use with (possibly) circular structures. Whenever depth of car recursion plus depth of cdr recursion exceeds depth, equaln does not search further along that chain. If equaln determines that the expressions x and y are equal, i.e. recursion never exceeds depth, value of equaln is T. If they are determined to be unequal, value is NIL. Otherwise, i.e. recursion exceeds depth at some point, value is ?, e.g. equaln[(((A)) B);(((Z)) B);2]=?. For depth=3, the value would be NIL. * * * * * * *

and[x₁;x₂;...;x_n] Takes an indefinite number of arguments (including 0). If all of its arguments have non-null value, its value is the value of its last argument, otherwise NIL. e.g., and[x;member[x;y]] will have as its value either NIL or a tail of y. and[]=T. Evaluation stops at the first argument whose value is NIL.

or[x₁;x₂;...;x_n] Takes an indefinite number of arguments (including 0). Its value is that of the first argument whose value is not NIL, otherwise NIL if all arguments have value NIL. e.g., or[x;numberp[y]] has its value x, y, or NIL. or[]=NIL. Evaluation stops at the first argument whose value is not NIL.

every[everyx;everyfn1;everyfn2] Is T if the result of applying everyfn1 to each element in everyx is true, otherwise NIL. e.g., every[(X Y Z);ATOM]=T.

every operates by computing everyfn1[car[everyx]].¹⁶ If this yields NIL, every immediately returns NIL. Otherwise, every computes everyfn2[everyx], or cdr[everyx] if everyfn2=NIL, and uses this as the "new" everyx, and the process continues, e.g., every[x;ATOM;CDDR] is true if every *other* element of x is atomic.

every compiles open.

¹⁵ A loose description of equal might be to say that x and y are equal if they print out the same way.

¹⁶ Actually, everyfn1[car[everyx];everyx] is computed, so for example everyfn1 can look at the next element on everyx if necessary.

some[somex;somefn1;somefn2] value is the tail of somex beginning with the first element that satisfies somefn1, i.e., for which somefn1 applied to that element is true. Value is **NIL** if no such element exists. e.g., `some[x; (LAMBDA (Z) (EQUAL Z Y))]` is equivalent to `member[y;x]`. some operates analogously to every. At each stage, `somefn1[car[somex];somex]` is computed, and if this is not **NIL**, somex is returned as the value of some. Otherwise, `somefn2[somex]` is computed, or `cdr[somex]` if somefn2=**NIL**, and used for the next somex.

some compiles open.

notany[somex;somefn1,somefn2]

same as `not[some[somex;somefn1,somefn2]]`

notevery[everyx;everyfn1;everyfn2]

`not[every[everyx;everyfn1;everyfn2]]`

memb[x;y]

Determines if x is a member of the list y, i.e., if there is an element of y eq to x. If so, its value is the tail of the list y starting with that element. If not, its value is **NIL**.

fmemb[x;y]

Fast version of memb that compiles open as a five instruction loop, terminating on a NULL check. Interpreted, fmemb gives an error, BAD ARGUMENT - FMEMB, if y ends in a non-list other than **NIL**.

member[x;y]

Identical to memb except that it uses equal instead of eq to check membership of x in y.

+ **eqmemb[x;y]**

true if either x is eq to y, or else y is a list and x is an fmemb of y. Compiles open.

The reason for the existence of both memb and member is that eq compiles as one instruction but equal requires a function call, and is therefore considerably more expensive. Wherever possible, the user should write (and use) functions that use eq instead of equal.

tailp[x;y]

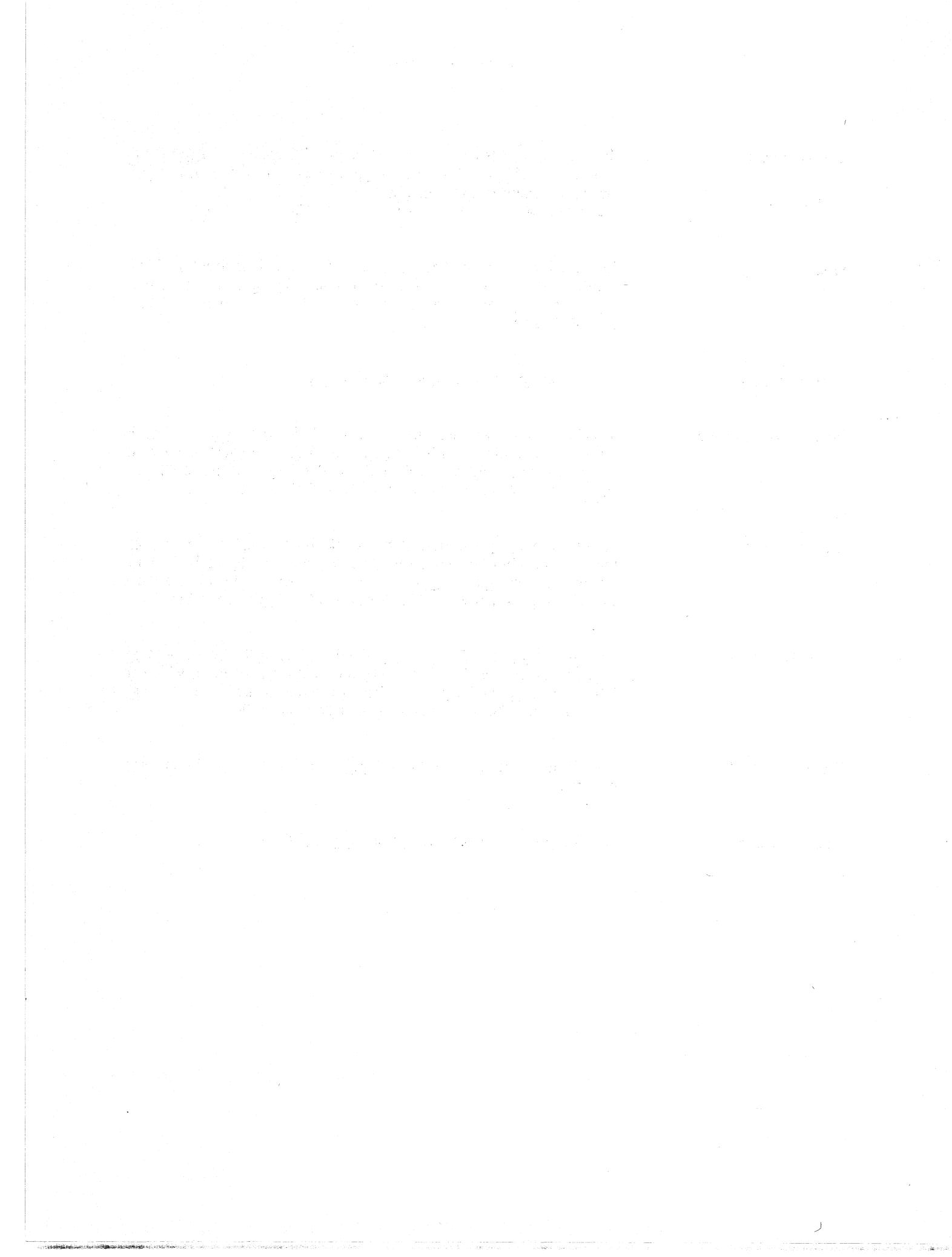
Is x, if x is a tail of y, i.e., x is eq to some number of cdrs ≥ 0 of y,¹⁷ **NIL** otherwise.

17 If x is eq to some number of cdrs > 0 of y, we say x is a proper tail.

assoc[key;alst]	<u>alst</u> is a list of lists (usually dotted pairs). The value of <u>assoc</u> is the first sublist of <u>alst</u> whose <u>car</u> is <u>eq</u> to <u>key</u> . If such a list is not found, the value is <u>NIL</u> . Example: assoc[B;((A . 1) (B . 2) (C . 3))] = (B . 2).
fassoc[key;alst]	Fast version of <u>assoc</u> that compiles open as a 6 instruction loop, terminating on a <u>NULL</u> check. Interpreted, <u>fassoc</u> gives an error if <u>alst</u> ends in a non-list other than <u>NIL</u> , BAD ARGUMENT - FASSOC.
sassoc[key;alst]	Same as <u>assoc</u> but uses <u>equal</u> instead of <u>eq</u> .
putassoc[key;val;alst]	Searches <u>alst</u> for an element <u>car</u> of which is <u>eq</u> to <u>key</u> . If one is found, <u>cdr</u> is replaced (using <u>rplacd</u>) with <u>val</u> . If no such element is found, <u>cons[key;val]</u> is added at the end of <u>alst</u> . Value is <u>val</u> . If <u>alst</u> is not a list, generates an error, ARG NOT LIST. *
listget[lst;prop]	Similar to <u>getprop</u> (Section 7) but works on lists using property list format. Searches <u>lst</u> two elements at a time, i.e., by <u>cddr</u> , looking for an element <u>eq</u> to <u>prop</u> . If one is found, returns the next element of <u>lst</u> , otherwise <u>NIL</u> . Returns <u>NIL</u> if <u>lst</u> is not a list.
listput[lst;prop;val]	Similar to <u>putprop</u> . Searches <u>lst</u> by <u>cddr</u> looking for an element <u>eq</u> to <u>prop</u> . If <u>prop</u> is found, replaces the next element of <u>lst</u> with <u>val</u> . Otherwise, <u>prop</u> and <u>val</u> are added to the end of <u>lst</u> . ¹⁸ Value is <u>val</u> . Generates an ARG NOT LIST error if <u>lst</u> is not a list.
listget1[lst;prop]	Like <u>listget</u> , but searches <u>lst</u> one <u>cdr</u> at a time, i.e., looks at each element. ¹⁹
listput1[lst;prop;val]	Like <u>listput</u> , except searches <u>lst</u> one <u>cdr</u> at a time.

¹⁸ If lst is a list with an odd number of elements, or ends in a non-list other than NIL, prop and val are added at its beginning.

¹⁹ listget1 used to be called get.



Index for Section 5

	Page Numbers
ADDTOVAR[VAR;X1;X2;...;Xn] NL*	5.6
ALLPROP (as value of DFNFLG)	5.6
AND[X1;X2;...;Xn] FSUBR*	5.11
ARG NOT LIST (error message)	5.2,13
ARG NOT LITATOM (error message)	5.4-6
ARRAYP[X] SUBR	5.10
arrays	5.10
ASSOC[KEY;ALST]	5.13
ATOM[X] SUBR	5.9
ATTEMPT TO BIND NIL OR T (error message)	5.4
ATTEMPT TO RPLAC NIL (error message)	5.2
ATTEMPT TO SET NIL (error message)	5.5
BAD ARGUMENT - FASSOC (error message)	5.13
BAD ARGUMENT - FMEMB (error message)	5.12
CAR[X] SUBR	5.1
CDR[X] SUBR	5.1
COND[C1;C2;...;Cn] FSUBR*	5.3
cond clause	5.3
CONS[X;Y] SUBR	5.1
cons algorithm	5.1
CONSCOUNT[N] SUBR	5.1
DFNFLG (system variable/parameter)	5.6
dotted pair	5.1
EQ[X;Y] SUBR	5.10
EQMemb[X;Y]	5.12
EQP[X;Y] SUBR	5.10
EQUAL[X;Y] SUBR	5.10
EQUALALL[X;Y]	5.11
EQUALN[X;Y;DEPTH]	5.11
ERSETQ[ERSETX] NL	5.5
EVALQT[LISPXID]	5.7
EVERY[EVERYX;EVERYFN1;EVERYFN2]	5.11
false	5.3
FASSOC[KEY;ALST]	5.13
FMEMB[X;Y]	5.12
FRPLACA[X;Y] SUBR	5.2
FRPLACD[X;Y] SUBR	5.2
FRPLNODE[X;A;D]	5.2
FRPLNODE2[X;Y]	5.2
GETATOMVAL[ATM] SUBR	5.6
GETTOPVAL[ATM] SUBR	5.5
GO[X] FSUBR*	5.5
ILLEGAL RETURN (error message)	5.5
KWOTE[X]	5.3
large integers	5.10
LISTGET[LST;PROP]	5.13
LISTGET1[LST;PROP]	5.13
LISTP[X] SUBR	5.10
LISTPUT[LST;PROP;VAL]	5.13
LISTPUT1[LST;PROP;VAL]	5.13
lists	5.10
LITATOM[X] SUBR	5.9
literal atoms	5.9
local variables	5.4
MEMB[X;Y]	5.12
MEMBER[X;Y]	5.12

Page
Numbers

NEGATE[X]	5.10
NEQ[X;Y]	5.10
NILL[]	5.10
NLISTP[X]	5.10
NLSETQ[NLSETX] NL	5.5
NOBIND	5.6
NOT[X] SUBR	5.10
NOTANY[SOMELEX;SOMEFN1;SOMEFN2]	5.12
NOTEVERY[EVERYX;EVERYFN1;EVERYFN2]	5.12
NULL[X] SUBR	5.10
NUMBERP[X] SUBR	5.9
numbers	5.9
OLDVALUE (system variable/parameter)	5.9
OR[X1;X2;...;Xn] FSUBR*	5.11
PARENTHESIS ERROR (error message)	5.2
predicates	5.9
PROG[VARLST;E1;E2;...;En] FSUBR*	5.4
PROG Label1	5.4
PROGN[X1;X2;...;Xn] FSUBR*	5.4
PROG1[X1;X2;...;Xn] FSUBR*	5.4
proper tail	5.12
PUTASSOC[KEY;VAL;ALST]	5.13
QUOTE[X] NL*	5.2
RESET	5.7
RESETFORM[RESETFORM;FORM1;FORM2;...;FORMn] NL*	5.8
RESETLST[RESETX] NL*	5.7
RESETSAVE[RESETX] NL*	5.7
RESETUNDO[X;STOPFLG]	5.9
RESETVAR[VAR;NEWVALUE;FORM] NL	5.8
RESETVARS[VARSLST;E1;E2;...;En] FSUBR*	5.8
RETURN[X] SUBR	5.5
RPAQ[X;Y] NL	5.6
RPAQQ[X;Y] NL	5.6
RPLACA[X;Y]	5.2
RPLACD[X;Y]	5.1
RPLNODE[X;A;D]	5.2
RPLNODE2[X;Y]	5.2
SASSOC[KEY;ALST]	5.13
SELECTQ[X;CLAUSE1;CLAUSE2;...;CLAUSEn;DEFAULT] NL*	5.3,4
SET[X;Y] SUBR	5.5
SETATOMVAL[ATM;VAL] SUBR	5.6
SETO[X;Y] FSUBR*	5.5
SETQQ[X;Y] NL	5.5
SETTOPVAL[ATM;VAL] SUBR	5.6
shallow binding	5.6
small integers	5.10
SOME[SOMELEX;SOMEFN1;SOMEFN2]	5.12
SPECVARS (compiler variable/parameter)	5.8
STRINGP[X] SUBR	5.9
strings	5.9
tail of a list	5.12
TAILP[X;Y]	5.12
top level value	5.6
true	5.3
UNDEFINED OR ILLEGAL GO (error message)	5.5
value cell	5.6
VALUE (property name)	5.6

SECTION 6

LIST MANIPULATION AND CONCATENATION

`list[x1;x2;...;xn]`

lambda-nospread function. Its value is a list of the values of its arguments.

`append[x1;x2;...;xn]`

Copies the top level of the list x_1 and appends this to a copy of top level list x_2 appended to ... appended to x_n , e.g.,
 $\text{append}[(A\ B)\ (C\ D\ E)\ (F\ G)] = (A\ B\ C\ D\ E\ F\ G)$.
Note that only the first $n-1$ lists are copied. However $n=1$ is treated specially; i.e., $\text{append}[x]$ can be used to copy the top level of a single list.¹

The following examples illustrate the treatment of non-lists.

$\text{append}[(A\ B\ C)\ ;D] = (A\ B\ C\ .\ D)$
 $\text{append}[A\ ;(B\ C\ D)] = (B\ C\ D)$
 $\text{append}[(A\ B\ C\ .\ D)\ ;(E\ F\ G)] = (A\ B\ C\ E\ F\ G)$
 $\text{append}[(A\ B\ C\ .\ D)] = (A\ B\ C\ .\ D)$.

`nconc[x1;x2;...;xn]`

Returns same value as append but actually modifies the list structure of $x_1 \dots x_{n-1}$.

Note that nconc cannot change NIL to a list. In other words, if the value of foo is NIL, then the value of (NCONC FOO (QUOTE (A B C))) is (A B C), but foo will *not* have been changed. The "problem" is that nconc simply has a collection of pointers to work with, and does not know where they originally came from, i.e., does not know that this NIL is the value of foo, and while it is possible to alter list structure using rplaca, there is no way to *change* a non-list to a list.

`nconc1[lst;x]`

Performs nconc[lst; list[x]].

¹ To copy a list to all levels, use copy.

tconc[ptr;x]

tconc is useful for building a list by adding elements one at a time at the end, i.e., its role is similar to that of nconcl. However, unlike nconcl, tconc does not have to search to the end of the list each time it is called. It does this by keeping a pointer to the end of the list being assembled, and updating this pointer after each call. The savings can be considerable for long lists. The cost is the extra word required for storing both the list being assembled, and the end of the list. ptr is that word: car[ptr] is the list being assembled, cdr[ptr] is last [car[ptr]]. The value of tconc is ptr, with the appropriate modifications to car and cdr. Example:

```
←(RPTQ 5 (SETQ FOO (TCONC FOO RPTN)))
((5 4 3 2 1) 1).
```

tconc can be initialized in two ways. If ptr is NIL, tconc will make up a ptr. In this case, the program must set some variable to the value of the first call to tconc. After that, it is unnecessary to reset ptr since tconc physically changes it. Thus:

```
←(SET FOO (TCONC NIL 1))
((1) 1)
←(RPTQ 4 (TCONC FOO RPTN))
((1 4 3 2 1) 1).
```

If ptr is initially (NIL), the value of tconc is the same as for ptr=NIL, but tconc changes ptr, e.g.,

```
←(SETQ FOO (CONS))
(NIL)
←(RPTQ 5 (TCONC FOO RPTN))
((5 4 3 2 1) 1).
```

The latter method allows the program to initialize, and then call tconc without having to perform setq on its value.

lconc[ptr;x]

Where tconc is used to add *elements* at the end of a list, lconc is used for building a list by adding *lists* at the end, i.e., it is similar to nconc instead of nconcl, e.g.,

```
←(SETQ FOO (CONS))
(NIL)
←(LCONC FOO (LIST 1 2))
((1 2) 2)
←(LCONC FOO (LIST 3 4 5))
((1 2 3 4 5) 5)
←(LCONC FOO NIL)
((1 2 3 4 5) 5)
```

Note that

```
←(TCONC FOO NIL)
((1 2 3 4 5 NIL) NIL)
←(TCONC FOO (LIST 3 4 5))
((1 2 3 4 5 NIL (3 4 5)) (3 4 5))
```

lconc uses the same pointer conventions as tconc for eliminating searching to the end of the list, so that the same pointer can be given to tconc and lconc interchangeably.

The functions docollect and endcollect also permit building up lists from left-to-right a la tconc, but without the overhead of the extra cons cell. The list being maintained is kept as a circular list. docollect adds items; endcollect replaces the tail with its second argument, and returns the full list.

docollect[item;lst] "adds" item to end of lst.

endcollect[lst;tail] "adds" tail onto end of lst and returns lst.

For example, mapcar is essentially:

```
(LAMBDA (L FN)
  [PROG (VAL)
    LP (COND ((NLISTP L) (RETURN (ENDCOLLECT VAL)))
              (SETQ VAL (DOCOLLECT (APPLY* FN (CAR L)) VAL))
              (SETQ L (CDR L))
              (GO LP))]
```

attach[x;l] Value is equal to cons[x;l], but attaches x to the front of l by doing an rplaca and rplacd, i.e., the value of attach is eq to l, which it physically changes. attach[x;NIL] is the same as cons[x;NIL]. Otherwise, if l is not a list, an error is generated, ARG NOT LIST.

remove[x;l] Removes all occurrences of x from list l, giving a copy of l with all elements equal to x removed.

Convention: Naming a function by prefixing an existing function with d frequently indicates the new function is a destructive version of the old one, i.e., it does not make any new structure but cannibalizes its argument(s).

dremove[x;l] Similar to remove, but uses eq instead of equal, and actually modifies the list l when removing x, and thus does not use any additional storage. More efficient than remove.

Note that dremove cannot change a list to NIL. For example, if the value of foo is (A), then (DREMOVE (QUOTE A) FOO) will return NIL, and not perform any conses, but the value of foo will still be (A) because there is no way to change a list to a non-list. See discussion following description of nconc on page 6.2.

mklist[x] if x is a list or NIL, value is x. Otherwise, value is list[x].

copy[x]	Makes a copy of the list <u>x</u> . The value of <u>copy</u> is the copied list. All levels of <u>x</u> are copied, ² down to non-lists, so that if <u>x</u> contains arrays and strings, the copy of <u>x</u> will contain the same arrays and strings, not copies. <u>Copy</u> is recursive in the <u>car</u> direction only, so that very long lists can be copied.
copyall[x]	Like <u>copy</u> except copies down to atoms, i.e., arrays, hash-arrays, strings, user data types, etc., are all copied. ³
+ hcopyall[x]	Like <u>copyall</u> except will work even if data structure contains circular pointers.
+ reverse[l]	Reverses (and copies) the top level of a list, e.g., reverse[(A B (C D))] = ((C D) B A). If <u>l</u> is not a list, value is <u>l</u> .
dreverse[l]	Value is same as that of <u>reverse</u> , but <u>dreverse</u> destroys the original list <u>l</u> and thus does not use any additional storage. More efficient than <u>reverse</u> .
subst[new;old;expr]	Value is the result of substituting the S-expression <u>new</u> for all occurrences of the S-expression <u>old</u> in the S-expression <u>expr</u> . Substitution occurs whenever <u>old</u> is <u>equal</u> to <u>car</u> of some subexpression of <u>expr</u> , or when <u>old</u> is both atomic and not NIL and <u>eq</u> to <u>cdr</u> of some subexpression of <u>expr</u> . For example:
	subst[A ; B ; (C B (X . B))] = (C A (X . A)) subst[A ; (B C) ; ((B C) D B C)] = (A D B C), not (A D . A).
	The value of <u>subst</u> is a copy of <u>expr</u> with the appropriate changes. Furthermore, if <u>new</u> is a list, it is copied at each substitution.
dsubst[new;old;expr]	Similar to <u>subst</u> , but does not copy <u>expr</u> , but changes the list structure <u>expr</u> itself. Like <u>subst</u> , <u>dsubst</u> substitutes with a copy of <u>new</u> . More efficient than <u>subst</u> .
lsubst[new;old;expr]	Like <u>subst</u> except <u>new</u> is substituted as a segment, e.g., lsubst[(A B) ; Y ; (X Y Z)] is (X A B Z). Note that if <u>new</u> is NIL, produces a copy of <u>expr</u> with all <u>old</u> 's deleted.

² To copy just the *top* level of x, do append[x].

³ Note: equalall (Section 5) is to copyall what equal is to copy, i.e. it will descend into arrays, hash arrays, etc. and compare their elements.

`esubst[new;old;expr;errorflg;charflg]`

Similar to `dsubst`, but first checks to see if `old` actually appears in `expr`. Implements the editors R and RC command. More complete discussion can be found in Section 9.

`sublis[alst;expr;flg]`

`alst` is a list of pairs: $((u_1 . v_1) (u_2 . v_2) \dots (u_n . v_n))$ with each u_i atomic.

The value of `sublis[alst;expr;flg]` is the result of substituting each `y` for the corresponding `u` in `expr`,⁴ e.g.,

`sublis[((A . X) (C . Y));(A B C D)] = (X B Y D).`

New structure is created only if needed, or if `flg=T`, e.g., if `flg=NIL` and there are no substitutions, value is `eq` to `expr`.

`dsublis[alst;expr;flg]`

Similar to `sublis`, but does not copy `expr`, but changes the list structure `expr` itself.

`subpair[old;new;expr;flg]`

Similar to `sublis`, except that elements of `new` are substituted for corresponding atoms of `old` in `expr`, e.g.,

`subpair[(A C);(X Y);(A B C D)] = (X B Y D)`

As with `sublis`, new structure is created only if needed, or if `flg=T`, e.g., if `flg=NIL` and there are no substitutions, the value is `eq` to `expr`.

If `old` ends in an atom other than `NIL`, the rest of the elements on `new` are substituted for that atom. For example, if `old=(A B . C)` and `new=(U V X Y Z)`, `U` is substituted for `A`, `V` for `B`, and `(X Y Z)` for `C`. Similarly, if `old` itself is an atom (other than `NIL`), the entire list `new` is substituted for it.

Note that `subst`, `dsubst`, `lsubst`, and `esubst` all substitute copies of the appropriate expression, whereas `subpair` and `sublis` substitute the identical structure (unless `flg=T`).

`last[x]`

Value is a pointer to the last node in the list `x`, e.g., if `x=(A B C)` then `last[x] = (C)`. If `x=(A B . C)` `last[x] = (B . C)`. Value is `NIL` if `x` is not a list.

`flast[x]`

Fast version of `last` that compiles open as a 5 instruction loop, terminating on a null-check. Interpreted, generates an error, `BAD ARGUMENT - FLAST`, if `x` ends in other than `NIL`.

`nleft[l;n;tail]`

`Tail` is a tail of `l` or `NIL`. The value of `nleft` is the tail of `l` that

⁴ To remember the order on `alst`, think of it as `old` to `new`, i.e., $u_i \rightarrow v_i$.

contains n more elements than tail,⁵ e.g., if x=(A B C D E), nleft[x;2]=(D E), nleft[x;1;cddr[x]]=(B C D E). Thus nleft can be used to work backwards through a list. Value is NIL if l does not contain n more elements than tail.

lastn[l;n]

Value is cons[x;y], where y is the last n elements of l, and x is the initial segment, e.g.,

lastn[(A B C D E);2]=((A B C) D E)
lastn[(A B);2]=(NIL A B).

Value is NIL, if l is not a list containing at least n elements.

nth[x;n]

Value is the tail of x beginning with the nth element, e.g., if n=2, value is cdr[x], if n=3, cddr[x], etc. If n=1, value is x, if n=0, for consistency, value is cons[NIL;x]. If x has fewer than n elements, value is NIL, e.g., nth[(A B);3]=NIL, as is nth[(A . B);3]. Note that nth[(A . B);2]=B.

fnth[x;n]

Fast version of nth that compiles open as a 3 instruction loop, terminating on a null-check. Interpreted, generates an error, BAD ARGUMENT - FNTH, if x ends in other than NIL.

length[x]

Value is the length of the list x where length is defined as the number of cdrs required to reach a non-list, e.g.,

length[(A B C)] = 3
length[(A B C . D)] = 3
length[A] = 0

flength[x]

Fast version of length that compiles open as a 4 instruction loop, terminating on a null-check. Interpreted, generates an error, BAD ARGUMENT - FLENGTH, if x ends in other than NIL.

+ eqlength[x;n]

+

+

+

equivalent to equal[length[x];n], but more efficient, i.e. eqlength stops as soon as it knows that x is longer than n. Note that eqlength is also safe to use on (possibly) circular lists, since it is "bounded" by n.

count[x]

Value is the number of list words in the structure x. Thus, count is like a length that goes to all levels. Count of a non-list is 0.

⁵ If tail is not NIL, but not a tail of l, the result is the same as if tail were NIL, i.e., nleft operates by scanning l looking for tail, not by computing the lengths of l and tail.

countdown[x;n]

Counts number of list words in x, decrementing n for each one. Stops when it finishes counting x, or when n reaches 0. Value is the current n. In other words, the value of countdown is the larger of 0 and idifference[n;count[x]], but countdown is more efficient, and can be used on circular structures since it is "bounded" by n. Note that countdown is to count what equaln (Section 5) is to equal.

ldiff[x;y;z]

y must be a tail of x, i.e., eq to the result of applying some number of cdrs to x. ldiff[x;y] gives a list of all elements in x up to y, i.e., the list difference of x and y. Thus ldiff[x;memeber[FOO;x]] gives all elements in x up to the first FOO.

Note that the value of ldiff is always new list structure unless y=NIL, in which case the value is x itself.

If z is not NIL, the value of ldiff is effectively nconc[z;ldiff[x;y]], i.e., the list difference is added at the end of z.

If y is not a tail of x, generates an error, **LDIFF: NOT A TAIL**. ldiff terminates on a null-check.

ldifference[a;b]

list difference. Value is list of those elements in a that are not members of b.

intersection[x;y]

Value is a list whose elements are members of both lists x and y. Note that intersection[x;x] gives a list of all members of x without any duplications.

union[x;y]

Value is a (new) list consisting of all elements included on either of the two original lists. It is more efficient to make x be the shorter list.⁶

sort[data;comparefn]⁷

data is a list of items to be sorted using comparefn, a predicate function of two arguments which can compare any two items on data and return T if the first one belongs before the second. If comparefn is NIL, alphorder is used; thus sort[data] will alphabetize a list. If comparefn is T, car's of items that are lists are given to alphorder, otherwise the items themselves; thus

⁶ The value of union is y with all elements of x not in y consed on the front of it. Therefore, if an element appears twice in y, it will appear twice in union[x;y]. Also, since union[(A);(A A)] = (A A), while union[(A A);(A)] = (A), union is non-commutative.

⁷ sort was written by L. P. Deutsch.

`sort[a-list;T]` will alphabetize an assoc list by the car of each item.
`sort[x;I LESSP]` will sort a list of integers.

The value of sort is the sorted list. The sort is destructive and uses no extra storage. The value returned is eq to data but elements have been switched around. Interrupting with control D, E, or B may cause loss of data, but control H may be used at any time, and sort will break at a clean state from which ↑ or control characters are safe. The algorithm used by sort is such that the maximum number of compares is $n^* \log_2 n$, where n is length[data].

Note: if comparefn[a;b] = comparefn[b;a], then the ordering of a and b may or may not be preserved.

For example, if (FOO . FIE) appears before (FOO . FUM) in x, `sort[x;T]` may or may not reverse the order of these two elements. Of course, the user can always specify a more precise comparefn.

`merge[a;b;comparefn]`

a and b are lists which have previously been sorted using sort and comparefn. Value is a destructive merging of the two lists. It does not matter which list is longer. After merging both a and b are equal to the merged list. (In fact, cdr[a] is eq to cdr[b]). merge may be aborted after control-H.

`alphorder[a;b]`

A predicate function of two arguments, for alphabetizing. Returns T if its arguments are in order, i.e., if b does not belong before a. Numbers come before literal atoms, and are ordered by magnitude (using greaterp). Literal atoms and strings are ordered by comparing the (ASCII) character codes in their pnames. Thus `alphorder[23;123]` is T, whereas `alphorder[A23;A123]` is NIL, because the character code for the digit 2 is greater than the code for 1.

Atoms and strings are ordered before all other data types. If neither a nor b are atoms or strings, the value of alphorder is T, i.e., in order.

Note: alphorder does no unpacks, chcons, conses or nthchars. It is several times faster for alphabetizing than anything that can be written using these other functions.

`mergeinsert[new;lst;oneflg]`

lst is NIL or a list of partially sorted items. mergeinsert tries to find the "best" place to (destructively) insert new, e.g., `mergeinsert[FIE2; (FOO FOO1 FIE FUM)]`=
(FOO FOO1 FIE FIE2 FUM). Value is lst. mergeinsert is undoable.

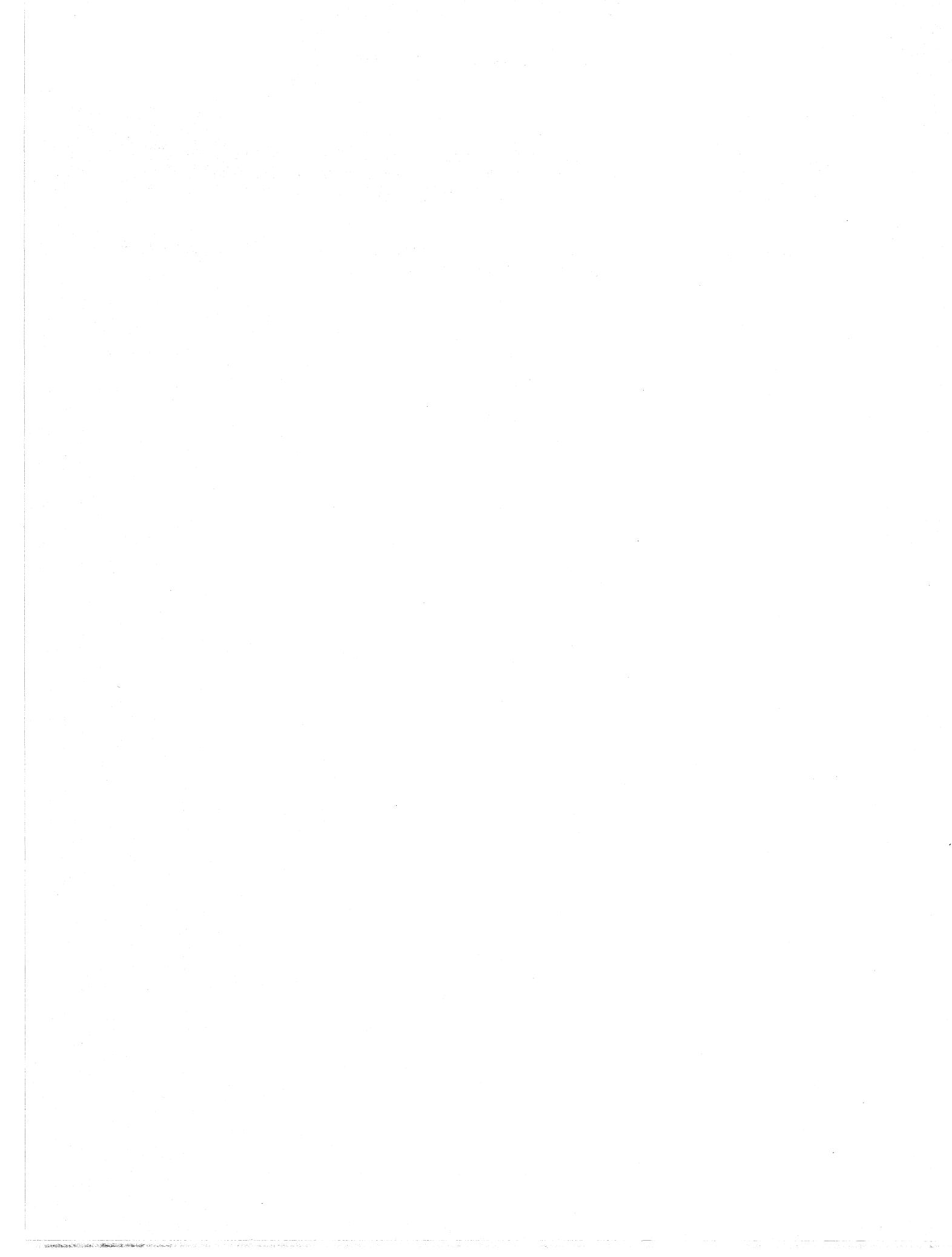
+
+

If oneflg=T and new is already a member of lst, mergeinsert does nothing and returns lst.

mergeinsert is used by addtofile (Section 14) to insert the name of a new function into a list of functions. The algorithm is essentially to look for the item with the longest common leading sequence of characters with respect to new, and then merge new in starting at that point.

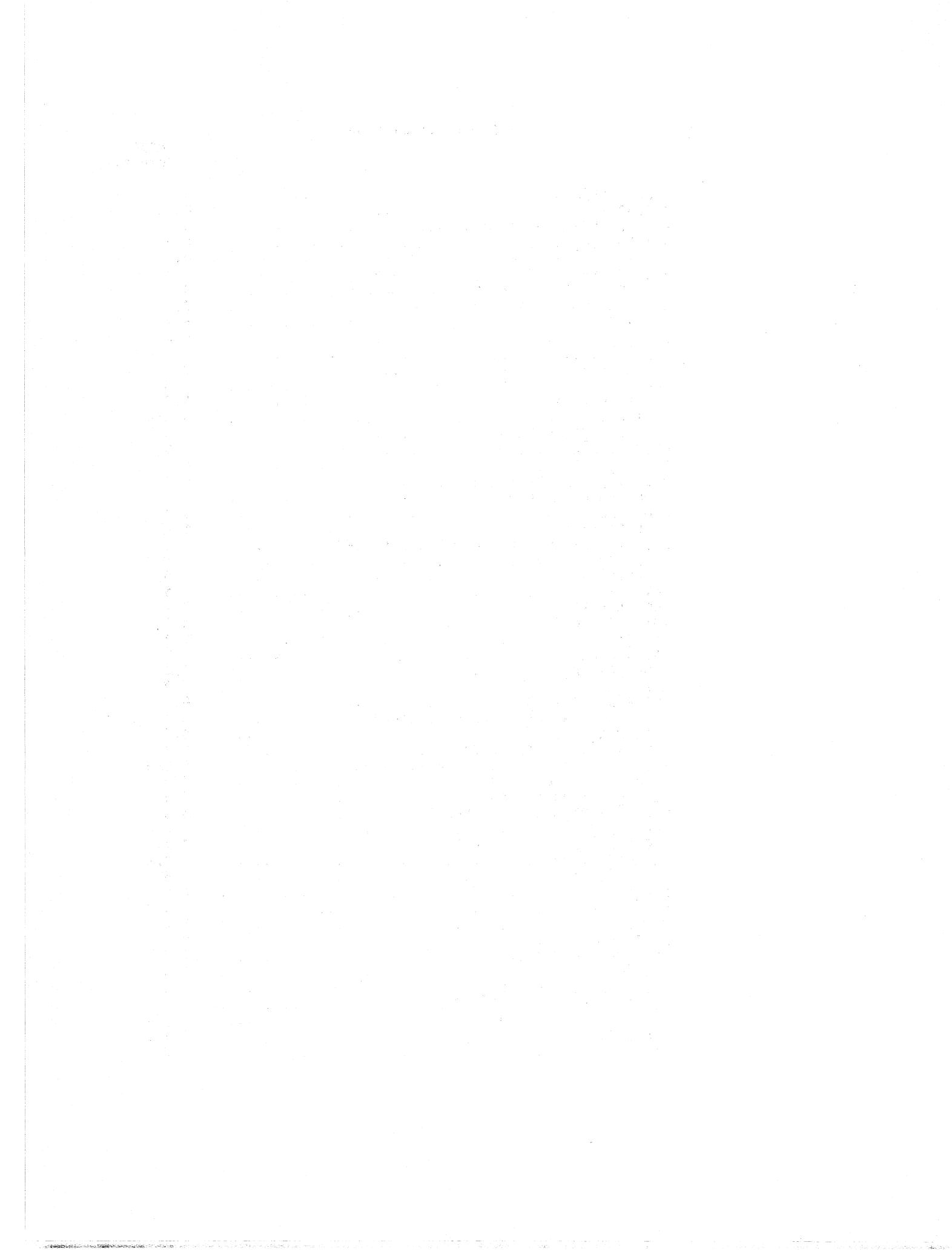
comparelists[x;y]

compares x and y and prints their differences, i.e., comparelists is +
essentially a SRCCOM for list structures. +



Index for Section 6

	Page Numbers
ALPHORDER[A;B]	6.8
APPEND[X1;X2;...;Xn] *	6.1
ARG NOT LIST (error message)	6.3
ATTACH[X;L]	6.3
ATTEMPT TO RPLAC NIL (error message)	6.3
BAD ARGUMENT - FLAST (error message)	6.5
BAD ARGUMENT - FLENGTH (error message)	6.6
BAD ARGUMENT - FNTH (error message)	6.6
COMPARELISTS[X;Y]	6.9
COPY[X]	6.4
copy	6.4,1,5
COPYALL[X]	6.4
COUNT[X]	6.6
COUNTDOWN[X;N]	6.7
destructive functions	6.3-4
DOCOLLECT[ITEM;LST]	6.3
DREMOVE[X;L]	6.3
DREVERSE[L]	6.4
DSUBLIS[ALST;EXPR;FLG]	6.5
DSUBST[NEW;OLD;EXPR]	6.4,5
ENDCOLLECT[LST;TAIL]	6.3
EQLLENGTH[X;N]	6.6
ESUBST[NEW;OLD;EXPR;ERRORFLG;CHARFLG]	6.5
FLAST[X]	6.5
FLENGTH[X]	6.6
FNTH[X;N]	6.6
HCOPYALL[X]	6.4
INTERSECTION[X;Y]	6.7
LAST[X]	6.5
LASTN[L;N]	6.6
LCONC[PTR;X]	6.2,3
LDIFF[X;Y;Z]	6.7
LDIFFERENCE[A;B]	6.7
LDIFF: NOT A TAIL (error message)	6.7
LENGTH[X]	6.6
LIST[X1;X2;...;Xn] SUBR*	6.1
list manipulation and concatenation	6.1-9
LSUBST[NEW;OLD;EXPR]	6.4,5
MERGE[A;B;COMPAREFN]	6.8
MERGEINSERT[NEW;LST;ONEFLG]	6.8
MKLIST[X]	6.3
NCONC[X1;X2;...;Xn] SUBR*	6.1,2
NCONC1[LST;X]	6.1,2
NLEFT[L;N;TAIL]	6.5
NTH[X;N]	6.6
null-check	6.5-7
REMOVE[X;L]	6.3
REVERSE[L]	6.4
SORT[DATA;COMPAREFN]	6.7
SRCCOM	6.9
SUBLIS[ALST;EXPR;FLG]	6.5
SUBPAIR[OLD;NEW;EXPR;FLG]	6.5
SUBST[NEW;OLD;EXPR]	6.4,5
TCONC[PTR;X]	6.2,3
UNION[X;Y]	6.7



SECTION 7

PROPERTY LISTS AND HASH LINKS

7.1 PROPERTY LISTS

Property lists are entities associated with literal atoms. Property lists are conventionally lists of the form (property value property value ... property value) although the user can store anything he wishes in the property list of a literal atom. However, the functions which manipulate property lists observe this convention by cycling down the property lists two cdrs at a time. Most of these functions also generate an error, ARG NOT LITATOM, if given an argument which is not a literal atom, i.e., they cannot be used directly on lists.

The term "property name" or "property" is used for the property indicators appearing in the odd positions, and the term "property value" or "value of a property" or simply "value" for the values appearing in the even positions. Sometimes the phrase "to store on the property --" is used, meaning to place the indicated information on the property list under the property name --.

Properties are usually atoms, although no checks are made to eliminate use of non-atoms in an odd position. However, the property list searching functions all use eq.

PROPERTY LIST FUNCTIONS

getproplist[atm]	if <u>atm</u> is a literal atom, returns property list of <u>atm</u> . Otherwise, generates ARG NOT LITATOM error. In Interlisp-10, <u>getproplist</u> compiles open without any error checks.
setproplist[atm;lst]	if <u>atm</u> is a non-NIL literal atom, sets property list of <u>atm</u> to be <u>lst</u> , and returns <u>lst</u> as its value. If <u>atm</u> is NIL, generates an ATTEMPT TO RPLAC NIL (unless <u>lst</u> is also NIL). If <u>atm</u> is not a literal atom, generates an ARG NOT LITATOM error.
getprop[atm;prop] ¹	gets the property value for <u>prop</u> from the property list of <u>atm</u> . The value of <u>getprop</u> is NIL if <u>atm</u> is not a literal atom, or <u>prop</u> is not found.

¹ getprop used to be called gtp.

- * Note: the value of getprop may also be NIL, if there is an occurrence of prop but the corresponding property value is NIL. The only way to distinguish these two cases is to perform memb[prop;getproplist[atm]].

Note: Since getprop searches a list two items at a time, the same object can be used as both a property name and a property value, e.g., if the property list of atm is (PROP1 A PROP2 B A C), then getprop[atm;A] = C.

putprop[atm;prop;val]

puts on the property list of atm,² the property prop with value val. val replaces any previous value for the property prop on this property list. Generates an error, ARG NOT LITATOM, if atm is not a literal atom. Value is val.

+ putprops[atm;prop₁;val₁;...;prop_n;val_n]

lambda nospread version of putprop. For i=1..n, puts prop_i, value val_i, on property list of atm. Performs some file package (see Section 14) related operations, i.e. "notices" that the corresponding properties have been changed. Used by for implementing various file package commands.

+ saveput[atm;prop;val]

+ same as put, but marks the corresponding property value as having been changed (for the file package, Section 14).

addprop[atm;prop;new;flg]

adds the value new to the list which is the value of property prop on property list of atm. If flg is T, new is consed onto the front of value of prop, otherwise it is nconced on the end (nconc). If atm does not have a property prop, the effect is the same as putprop[atm;prop;list[new]]. for example, if addprop[FOO;PROP;FIE] is followed by addprop[FOO;PROP;FUM], getprop[FOO;PROP] will be (FIE FUM). The value of addprop is the (new) property value. If atm is not a literal atom, generates an error, ARG NOT LITATOM.

remprop[atm;prop]

removes all occurrences of the property prop (and its value) from the property list of atm. Value is prop if any were found, otherwise NIL. If atm is not a literal atom, generates an error, ARG NOT LITATOM.

+ remproplist[atm;props]

removes all occurrences of all properties on props and their

² listput, listputl, listget, and listgetl are functions similar to putprop and getprop that work directly on lists. They are described in Section 5.

corresponding property values from property list of atm. Value is NIL. +

changeprop[x;prop1;prop2] Changes *name* of property prop1 to prop2 on property list of x, (but does not affect the value of the property). Value is x, unless prop1 is not found, in which case, the value is NIL. If x is not a literal atom, generates an error, ARG NOT LITATOM.

propnames[atm] value is a list of the property names on the property list of atm. +

getlis[x;props] searches the property list of x, and returns the property list as of the first property on props that it finds e.g., if the property list of x is (PROP1 A PROP3 B A C),
 $\text{getlis}[x;(\text{PROP2 PROP3})]=(\text{PROP3 B A C})$
Value is NIL if no element on props is found. x can also be a list itself, in which case it is searched as above.

deflist[l;prop] is used to put values under the same property name on the property lists of several atoms. l is a list of two-element lists. The first element of each is a literal atom, and the second element is the property value for the property prop. The value of deflist is NIL.

Note: Many atoms in the system already have property lists, with properties used by the compiler, the break package, DWIM, etc. Be careful not to clobber such system properties. The value of sysprops gives the complete list of the property names used by the system.

7.2 HASH LINKS

The description of the hash link facility in Interlisp is included in the chapter on property lists because of the similarities in the ways the two features are used. A property list provides a way of associating information with a particular atom. A hash link is an association between any Interlisp pointer (atoms, numbers, arrays, strings, lists, et al) called the hash-item, and any other Interlisp pointer called the hash-value. Hash links are implemented by computing an address, called the hash-address, in a specified array, called the hash-array, and storing the hash-value and the hash-item into the cell with that address. The contents of that cell, i.e. the hash-value and hash-item, is then called the hash-link.³

Since the hash-array is obviously much smaller than the total number of possible hash-items,⁴ the

³ The term hash link (unhyphenated) refers to the process of associating information this way, or the "association" as an abstract concept.

⁴ which is the total number of Interlisp pointers, i.e. in Interlisp-10, 256K.

hash-address computed from item may already contain a hash-link. If this link is from item,⁵ the new hash-value simply replaces the old hash-value. Otherwise, another hash-address (in the same hash-array) must be computed, etc, until an empty cell is found,⁶ or a cell containing a hash-link from item.

When a hash link for item is being retrieved, the hash-address is computed using the same algorithm as that employed for making the hash link. If the corresponding cell is empty, there is no hash link for item. If it contains a hash-link from item, the hash-value is returned. Otherwise, another hash-address must be computed, and so forth.

Note that more than one hash link can be associated with a given hash-item by using more than one hash-array.

HASH LINK FUNCTIONS

In the description of the functions below, the argument array has one of three forms: [1] **NIL**, in which case the hash-array provided by the system, syshasharray, is used;⁷ [2] a hash-array created by the function harray; or [3] a list car of which is a hash-array. The latter form is used for specifying what is to be done on overflow, as described below.

harray[n] creates a hash-array of size at least n,⁸ equivalent to clrhash[array[n]].

+ **harryp[x]** value is x, if x is a hash array.

+ **harraysize[harray]** returns size of harray.

* **clrhash[harray]** clears all hash links of harray. Value is harray.

puthash[item;val;harray] puts into harray a hash-link from item to val. Replaces previous link from same item, if any. If val=NIL any old link is removed, (hence a hash-value of **NIL** is not allowed). Value is val.

gethash[item;harray] finds hash-link from item in harray, and returns the hash-value. Value is **NIL**, if no link exists. gethash compiles open.

⁵ eq is used for comparing item with the hash-item in the cell.

⁶ When the hash array becomes 7/8 full, it is considered to be full, and the array is either enlarged, or an error is generated, as described below in the discussion of overflow.

⁷ syshasharray is not used by the system, it is provided solely for the user's benefit. It is initially 512 words large, and is automatically enlarged by 50% whenever it is "full". See page 7.5.

+ ⁸ In Interlisp-10, the size of the hash array may be increased so that it is relatively prime to possible probe intervals.

rehash[oldar;newar]

hashes all items and values in oldar into newar. The two arrays do not have to be (and usually aren't) the same size. Value is newar.

maphash[array;maphfn]

maphfn is a function of two arguments. For each hash-link in array, maphfn will be applied to the hash-value and hash-item, e.g. maphash[a;(LAMBDA (X Y) (AND (LISTP Y) (PRINT X)))] will print the hash-value for all hash-links from lists. The value of maphash is array.

dmphash[arrayname₁;arrayname₂;...;arrayname_n]

Nlambda-nospread that prints on the primary output file loadable forms which will restore what is in the hash-array specified by arrayname_i, e.g. (E (DMPHASH SYHASHARRAY)) as a file package command will dump the system hash-array.

Note: all eq identities except atoms and small integers are lost by dumping and loading because read will create new structure for each item. Thus if two lists contain an eq substructure, when they are dumped and loaded back in, the corresponding substructures while equal are no longer eq.⁹

HASH OVERFLOW

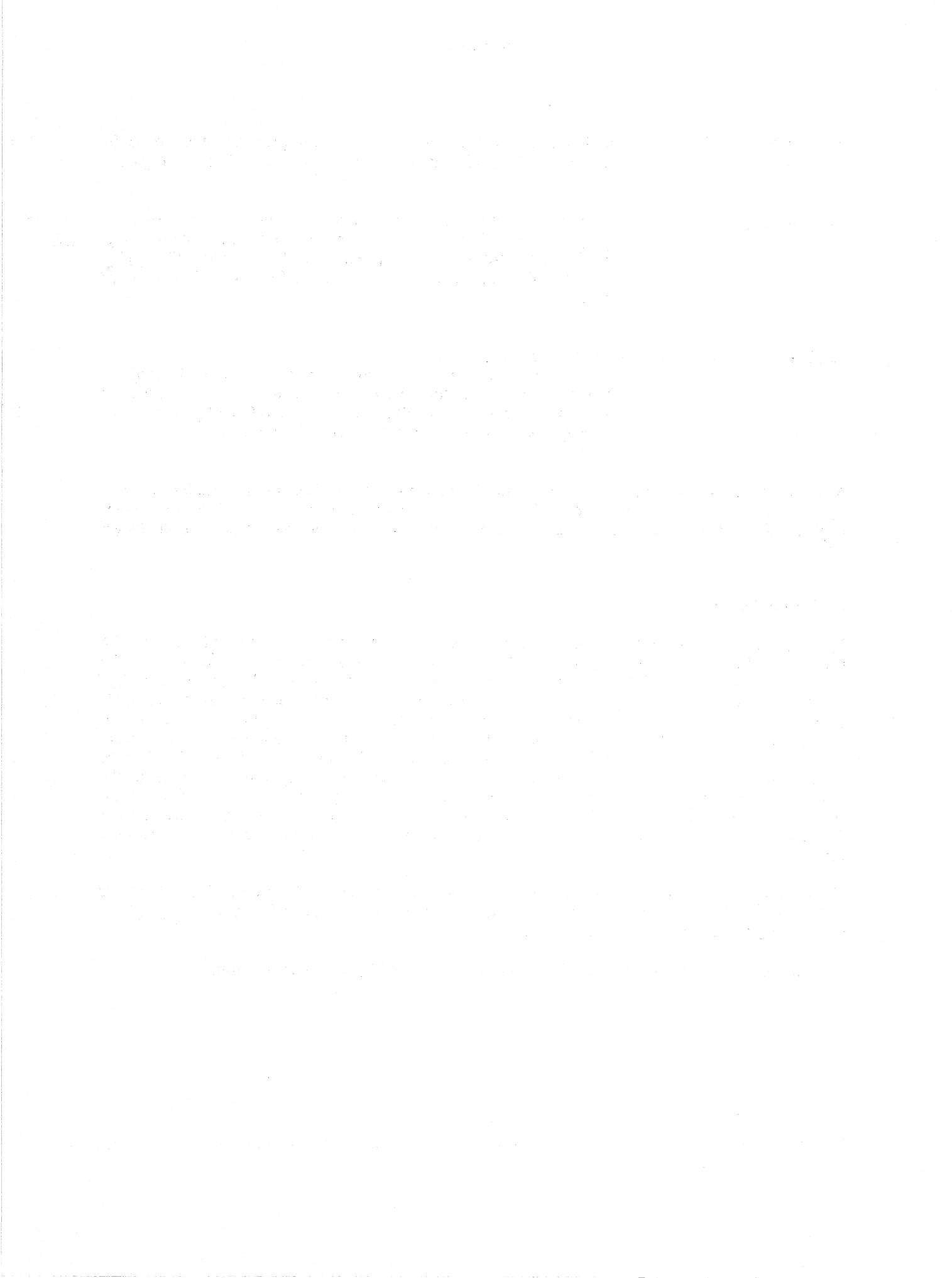
By using an array argument of a special form, the user can provide for automatic enlargement of a hash-array when it overflows, i.e., is full and an attempt is made to store a hash link into it. The array argument is either of the form [1] (hash-array . n), n a positive integer; [2] (hash-array . f), f a floating point number; [3] (hash-array); or [4] (hash-array . fn), fn a function name or a lambda expression. In the first case, a new hash-array is created with n more cells than the current hash-array. In the second case, the new hash array will be f times the size of the current hash-array. The third case, (hash-array), is equivalent to (hash-array . 1.5). In the fourth case, (hash-array . fn), fn is called with (hash-array . fn) as its argument. If fn returns a number, the number will be the size of the new hash array. Otherwise, the new size defaults to 1.5 times the size of the old hash array, e.g. fn could be used to print a message, or perform some monitor function. In each case, the new hash-array is replaced into the dotted pair, and the computation continues.

If a hash-array overflows, and the array argument used was not one of these three forms, the error HASH TABLE FULL is generated, which will either cause a break or unwind to the last errorset, as per treatment of errors described in Section 16.

The system hash array, syshasharray, is automatically enlarged by 1.5 when it is full.

⁹

The HORRIBLEVARS prettydef command (Section 14) provides a way of dumping hash tables such that these identities are preserved.



Index for Section 7

	Page Numbers
ADDPROP[ATM;PROP;NEW;FLG]	7.2
ARG NOT LITATOM (error message)	7.1-3
ATTEMPT TO RPLAC NIL (error message)	7.1
CHANGEPROP[X;PROP1;PROP2]	7.3
CLRHASH[HARRAY] SUBR	7.4
DEFLIST[L;PROP]	7.3
DMPHASH[ARRAYNAME1;...;ARRAYNAME _n] NL*	7.5
ERRORSET[FORM;FLG] SUBR	7.5
GETHASH[ITEM;HARRAY] SUBR	7.4
GETLIS[X;PROPS]	7.3
GETPROP[ATM;PROP]	7.1
GETPROPLIST[ATM]	7.1
HARRAY[N] SUBR	7.4
HARRAYP[X]	7.4
HARRAYSIZE[HARRAY]	7.4
hash link functions	7.4-5
hash links	7.3-5
hash overflow	7.5
HASH TABLE FULL (error message)	7.5
hash-address	7.3
hash-array	7.3-5
hash-item	7.3-5
hash-link	7.3-5
hash-value	7.3-5
MAPHASH[ARRAY;MAPFN]	7.5
property list	7.1,2-3
property name	7.1,3
property value	7.1,3
PROPNAMES[ATM]	7.3
PUTHASH[ITEM;VAL;HARRAY] SUBR	7.4
PUTPROP[ATM;PROP;VAL]	7.2
PUTPROPS[ATM;PROP1;VAL1;...;PROPN;VALn] NL*	7.2
REHASH[OLDAR;NEWAR] SUBR	7.5
REMPROP[ATM;PROP]	7.2
REMPROPLIST[ATM;PROPS]	7.2
SAVEPUT[ATM;PROP;VAL]	7.2
SETPROPLIST[ATM;LST]	7.1
SYSHASHARRAY (system variable/parameter)	7.4-5
SYSPROPS (system variable/parameter)	7.3
value of a property	7.1

SECTION 8

FUNCTION DEFINITION AND EVALUATION

GENERAL COMMENTS

A function definition in Interlisp is stored in a special cell called the function definition cell, which is associated with each literal atom. This cell is directly accessible via the two functions putd, which puts a definition in the cell, and getd which gets the definition from the cell. In addition, the function fntyp returns the function type, i.e., EXPR, EXPR* ... FSUBR* as described in Section 4. Exprp, ccodep, and subrp are true if the function is an expr, compiled function, or subr respectively; argtype returns 0, 1, 2, or 3, depending on whether the function is a spread or nospread (i.e., its fntyp ends in *), or evaluate or no-evaluate (i.e., its fntyp begins with F or CF); arglist returns the list of arguments; and nargs returns the number of arguments. fntyp, exprp, ccodep, subrp, argtype, arglist, and nargs can be given either a literal atom, in which case they obtain the function definition from the atom's definition cell, or a function definition itself.

SUBRS

Because subrs¹ are called in a special way, their definitions are stored differently than those of compiled or interpreted functions. getd of a subr returns a dotted pair, car of which is an encoding of the argtype and number of arguments of the subr, and cdr of which is the address of the first instruction. Note that each getd of a subr performs a cons. Similarly, putd of a definition of the form (number . address), where number and address are in the appropriate ranges, stores the definition as a subr.

VALIDITY OF DEFINITIONS IN INTERLISP-10

Although the function definition cell is intended for function definitions, putd and getd do not make thorough checks on the validity of definitions that "look like" exprs, compiled code, or subrs. Thus if putd is given an array pointer, it treats it as compiled code, and simply stores the array pointer in the definition cell. getd will then return the array pointer. Similarly, a call to that function will simply transfer to what would normally be the entry point for the function, and produce random results if the array were not compiled function.

¹ Basic functions, handcoded in machine language, e.g., cons, car, cond. The terms subr includes spread/nospread, eval/noeval functions, i.e., the four fntyp's SUBR, FSUBR, SUBR*, and FSUBR*.

Similarly, if putd is given a dotted pair of the form (number . address) where number and address fall in the subr range, putd assumes it is a subr and stores it away as described earlier. getd would then return a dotted pair equal (but not eq) to the expression originally given putd. Similarly, a call to this function would transfer to the corresponding address.

Finally, if putd is given any other list, it simply stores it away. A call to this function would then go through the interpreter as described in the appendix.

Note that putd does not actually check to see if the s-expression is valid definition, i.e., begins with LAMBDA or NLAMBDA. Similarly, exprp is true if a definition is a list and not of the form (number . address), number = 0, 1, 2, or 3 and address a subr address; subrp is true if it is of this form. arglist and nargs work correspondingly.

Only fntyp and argtype check function definitions further than that described above: both argtype and fntyp return NIL when exprp is true but car of the definition is not LAMBDA or NLAMBDA.² In other words, if the user uses putd to put (A B C) in a function definition cell, getd will return this value, the editor and prettyprint will both treat it as a definition, exprp will return T, ccodep and subrp NIL, arglist B, and nargs 1.

getd[x] gets the function definition of x. Value is the definition.³ Value is NIL if x is not a literal atom, or has no definition.

fgetd[x] fast version of getd that compiles open. Interpreted, generates an error, BAD ARGUMENT - FGETD, if x is not a literal atom. Egetd is intended primarily to check whether a function *has* a definition, rather than to obtain the definition. Therefore, for subrs, fgetd returns just the address of the function definition, not the dotted pair returned by getd, page 8.1, thereby saving the cons.

putd[fn;def] puts the definition def into fn's function cell. Value is def. Generates an error, ARG NOT LITATOM, if fn is not a literal atom. Generates an error, ILLEGAL ARG, if def is a string, number, or literal atom other than NIL.

putdq[fn;def] nlambda version of putd; both arguments are considered quoted. Value is fn.

+ putdq?[fn;def] nlambda version of putd. If fn is not defined, same as putd[fn;def]. Otherwise, does nothing and returns NIL.

² These functions have different values on LAMBDA and NLAMBDA and hence must check. The compiler and interpreter also take different actions for LAMBDA and NLAMBDA, and therefore generate errors if the definition is neither.

³ Note that in Interlisp-10, getd of a subr performs a cons, as described on page 8.1.

movd[from;to;copyflg]	Moves the definition of <u>from</u> to <u>to</u> , i.e., redefines <u>to</u> . If <u>copyflg=T</u> , a <u>copy</u> of the definition of <u>from</u> is used. <u>copyflg=T</u> is only meaningful for exprs, although <u>movd</u> works for compiled functions and subrs as well. The value of <u>movd</u> is <u>to</u> .
movd?[from;to;copyflg]	if <u>to</u> is not defined, same as movd[from;to;copyflg]. Otherwise, + does nothing and returns NIL.

Note: fntyp, subrp, ccodep, scodep, exprp, argtype, nargs, and arglist all can be given either the name of a function, or a definition.

fntyp[fn]	Value is NIL if <u>fn</u> is not a function definition or the name of a defined function. Otherwise <u>fntyp</u> returns one of the following as defined in the section on function types:
	EXPR CEXPR SUBR
	FEXPR CFEXPR FSUBR
	EXPR* CEXPR* SUBR*
	FEXPR* CFEXPR* FSUBR*
	The prefix <u>F</u> indicates unevaluated arguments, the prefix <u>C</u> indicates compiled code, and the suffix <u>*</u> indicates an indefinite number of arguments.
	<u>fntyp</u> returns FUNARG if <u>fn</u> is a funarg expression. See Section 11.
subrp[fn]	is true if and only if fntyp[fn] is either SUBR, FSUBR, SUBR*, or FSUBR*, i.e., the third column of fntyp's.
ccodep[fn]	is true if and only if fntyp[fn] is either CEXPR, CFEXPR, CEXPR*, or CFEXPR*, i.e., second column of fntyp's.
scodep[fn]	is true if <u>fn</u> has or is a swapped compiled definition (see Section 3). +
exprp[fn]	is true if fntyp[fn] is either EXPR, FEXPR, EXPR*, or FEXPR*, i.e., first column of fntyp's. However, exprp[fn] is also true if <u>fn</u> is (has) a list definition that is not a SUBR, but does not begin with either LAMBDA or NLAMBDA. In other words, <u>exprp</u> is not quite as selective as <u>fntyp</u> .
argtype[fn]	<u>fn</u> is the name of a function or its definition. The value of <u>argtype</u> is the argtype of <u>fn</u> , i.e., 0, 1, 2, or 3, or NIL if <u>fn</u> is not a function. The interpretation of the argtype is:

0 eval/spread function (EXPR, CEXPR, SUBR)

- 1 no-eval/spread functions (FEXPR, CFEPR, FSUBR)
- 2 eval/nospread functions (EXPR*, CEXPR*, SUBR*)
- 3 no-eval/nospread functions (FEXPR*, CFEPR*, FSUBR*)

i.e., argtype corresponds to the *rows* of fntyps.

nargs[fn]

value is the number of arguments of fn, or NIL if fn is not a function.⁴ nargs uses exprp, not fntyp, so that nargs[(A (B C) D)]=2. If fn is a nospread function, the value of nargs is 1.

arglist[fn]

value is the "argument list" for fn. Note that the "argument list" is an atom for nospread functions. Since NIL is a possible value for arglist, an error is generated, ARGs NOT AVAILABLE, if fn is not a function.⁵

If fn is a SUBR or FSUBR in Interlisp-10, the value of arglist is (U), (U V), (U V W), etc. depending on the number of arguments, if a SUBR* or FSUBR*, the value is U. This is merely a "feature" of arglist, subrs do not actually store the names of their argument(s) on the stack.

smartarglist[fn;explainflg;tail]

If explainflg=T and fn is a nospread function, e.g., list, selectq, etc., smartarglist uses helpsys to interrogate the Interlisp manual to obtain more descriptive argument names, e.g., smartarglist[SELECTQ;T]=(X Y1 Y2 ... YN Z). If fn is a nospread function, and explainflg=NIL, then smartarglist returns arglist[fn].

If fn is a spread SUBR, regardless of the value of explainflg, smartarglist also consults the manual, e.g., smartarglist[READ]=(FILE RDTBL FLG), smartarglist[STKPOS]=(FN N POS).

For all other cases, and when helpsys is undefined or unsuccessful in finding the arguments, smartarglist simply returns arglist[fn].

smartarglist first calls fncheck (Section 17) on fn. fncheck will

4 i.e., if exprp, ccodep, and subrp are all NIL.

5 If fn is a compiled function, the argument list is constructed, i.e., each call to arglist requires making a new list. For interpreted functions, whose definitions are lists of the form (LAMBDA --) or (NLAMBDA --), the argument list is simply cadr of getd. If fn has a list definition, and car of the definition is not LAMBDA or NLAMBDA, arglist will check to see if car of the definition is a member of lambdaclist (see Section 17). If it is, arglist presumes this is a function object the user is defining via an appropriate dwluserform (Section 17), and simply returns cadr of the def as its argument list. Otherwise arglist generates an error as described above.

attempt spelling correction if fn is not the name of a function.⁶ If unsuccessful, an error will be generated, fn NOT A FUNCTION.

smartarglist is used by break (Section 15) and advise (Section 19) with explainflg=NIL for constructing equivalent EXPR definitions, and by the ?= lispmacro (Section 22), with explainflg=T. In order to avoid repeated calls to helpsys, and also to provide the user with an override, smartarglist stores the arguments returned from helpsys on the property list of fn under the property ARGNAMES and checks for this property before calling helpsys.⁷

define[x;-]

The argument of define is a list. Each element of the list is itself a list either of the form (name definition) or (name arguments ...). In the second case, following "arguments" is the body of the definition. As an example, consider the following two equivalent expressions for defining the function null.

- 1) (NULL (LAMBDA (X) (EQ X NIL)))
- 2) (NULL (X) (EQ X NIL))

define will generate an error, INCORRECT DEFINING FORM, on encountering an atom where a defining list is expected. If dfnflg=NIL, an attempt to redefine a function fn will cause define to print the message (fn REDEFINED) and to save the old definition of fn using savedef before redefining it.⁸ If dfnflg=T, the function is simply redefined. If dfnflg=PROP or ALLPROP, the new definition is stored on the property list under the property EXPR. (ALLPROP affects the operation of rpaqq and rpaq, Section 5). dfnflg is initially NIL.

dfnflg is reset by load to enable various ways of handling the defining of functions and setting of variables when loading a file. For most applications, the user will not reset dfnflg directly himself.

Note: define will operate correctly if the function is already defined and broken, advised, or broken-in.

defineq[x₁;x_i;...;x_n]

nlambda nospread version of define, i.e., takes an indefinite number of arguments which are not evaluated. Each x_i must be a list, of the form described in define. defineq calls define, so dfnflg affects its operation the same as define.⁹

⁶ tail is used for the call to fixspell.

⁷ For spread functions, the argument list itself is stored. For nospread, the form is (NIL arglist1 . arglist2) where arglist1 is the value of smartarglist when explainflg=T, and arglist2 the value when explainflg=NIL, e.g., getp[SELECTQ;ARGNAMES]=(NIL (X CLAUSE1 CLAUSE2 ... CLAUSEN DEFAULT) . SELCQ).

⁸ except if the old and new definitions are the same, i.e. equal, the effect is simply a no-op.

⁹ For expressions involving type-in only, if the time stamp facility is enabled (Section 9), both define and defineq will stamp the definition with the user's initials and date.

savedef[fn]¹⁰

Saves the definition of fn on its property list under property EXPR, CODE, or SUBR depending on its fntyp. Value is the property name used. If gctd[fn] is non-NIL, but fntyp[fn] is NIL, saves on property name LIST. This situation can arise when a function is redefined which was originally defined with LAMBDA misspelled or omitted.

If fn is a list, savedef operates on each function in the list, and its value is a list of the individual values.

unsavedef[fn;prop]¹¹

Restores the definition of fn from its property list under property prop (see savedef above). Value is prop. If nothing saved under prop, and fn is defined, returns (prop NOT FOUND), otherwise generates an error, NOT A FUNCTION.

If prop is not given, i.e., NIL, unsavedef looks under EXPR, CODE, and SUBR, in that order. The value of unsavedef is the property name, or if nothing is found and fn is a function, the value is (NOTHING FOUND); otherwise generates an error, NOT A FUNCTION.

If dfnflg=NIL, the current definition of fn, if any, is saved using savedef. Thus one can use unsavedef to switch back and forth between two definitions of the same function, keeping one on its property list and the other in the function definition cell.

If fn is a list, unsavedef operates on each function of the list, and its value is a list of the individual values.

eval[x]¹²

eval evaluates the expression x and returns this value i.e., eval provides a way of calling the interpreter. Note that eval is itself a lambda type function, so its argument is first evaluated, e.g.,

+ 10 Note: both savedef and unsavedef are redefined in more general terms in Section 14 to operate on typed definitions of which a function definition is but one example. Thus, their actual argument lists in Interlisp are different than given here: savedef is a function of three arguments, name, type, and definition, and unsavedef a function of two arguments, name and type. However, when their extra arguments are defaulted to NIL, they operate as described above.

11 See previous footnote to savedef.

12 In Interlisp-10, eval is a subr so that the "name" x does not actually appear on the stack.

```

←SET(FOO (ADD1 3))
(ADD1 3)
←(EVAL FOO)
4
←EVAL(FOO) or (EVAL (QUOTE FOO))
(ADD1 3)

```

boundp[var]

returns T if var is bound in the current context (even if bound to NOBIND), or if var has a top level value other than NOBIND. In other words, eval[x], where x is an atom, will cause an error, i.e. a call to faulteval (Section 16), if and only if the value of boundp[x] is NIL.

+
+
+
+
+
+

defeval[type;fn]

specifies how a datum of a particular type is to be evaluated.¹³ Intended primarily for user defined data types, but works for all data types except lists, literal atoms, and numbers. type is a type name or type number. fn is a function object, i.e. name of a function or a lambda expression. Whenever the interpreter encounters a datum of the indicated type, fn is applied to the datum and its value returned as the result of the evaluation. The value of defeval is the previous evaling fn. If fn=NIL, defeval returns current evaling fn without changing it. If fn=T, sets evaling function back to system default (which for all data types except lists is to return the datum itself).

+
+
+
+
+
+
+
+
+
+

apply[fn;args]

apply applies the function fn to the arguments args. The individual elements of args are not evaluated by apply, fn is simply called with args as its argument list.¹⁴ Thus for the purposes of apply, nlambda's and lambda's are treated the same. However like eval, apply is a lambda function so its arguments are evaluated before it is called e.g.,

+

¹³ compiletypecl (Section 18) permits the user to specify how a datum of a particular type is to be compiled.

+

¹⁴ Note that fn may still explicitly evaluate one or more of its arguments itself, as in the case of setq. Thus, (APPLY (QUOTE SETQ) (QUOTE (FOO (ADD1 3)))) will set FOO to 4, whereas (APPLY (QUOTE SET) (QUOTE (FOO (ADD1 3)))) will set FOO to the expression (ADD1 3).

```

←SET(F001 3)
3
←SET(F002 4)
4
←(APPLY (QUOTE IPLUS) (LIST F001 F002]
7

```

Here, foo1 and foo2 were evaluated when the second argument to apply was evaluated. Compare with:

```

←SET(F001 (ADD1 2))
(ADD1 2)
←SET(F002 (SUB1 5))
(SUB1 5)
←(APPLY (QUOTE IPLUS) (LIST F001 F002]

```

NON-NUMERIC ARG
 (ADD1 2)

apply*[fn;arg₁;arg₂;...;arg_n]

equivalent to apply[fn;list[arg₁;arg₂;...;arg_n]] For example, if fn is the name of a functional argument to be applied to x and y, one can write (APPLY* FN X Y), which is equivalent to (APPLY FN (LIST X Y)). Note that (FN X Y) specifies a call to the function FN itself, and will cause an error if FN is not defined. (See Section 16.) FN will *not* be evaluated.

evala[x;a]

Simulates a-list evaluation as in LISP 1.5. x is a form, a is a list of dotted pairs of variable name and value. a is "spread" on the stack, and then x is evaluated, i.e., any variables appearing free in x, that also appears as car of an element of a will be given the value in the cdr of that element.

rpt[rptn;rptf]

Evaluates the expression rptf rptn times. At any point, rptn is the number of evaluations yet to take place. Returns the value of the last evaluation. If rptn ≤ 0, rptf is not evaluated, and the value of rpt is NIL.

Note: rpt is a lambda function, so both its arguments are evaluated before rpt is called. For most applications, the user will probably want to use rptq.

rptq[n;form₁;form₂;...;form_n] nlambda, nospread version of rpt: n is evaluated, form_i are not, e.g., (RPTQ 10 (READ)) will perform ten calls to read. rptq compiles open.

+ **frptq[n;form₁;form₂;...;form_n]** fast version of rptq that compiles open using an assemble macro. Does not bind n.

arg[var;m]

Used to access the individual arguments of a lambda nospread function. arg is an nlambda function used like set. var is the *name* of the atomic argument list to a lambda-nospread function, and is not evaluated; m is the number of the desired argument, and is evaluated. For example, consider the following definition of iplus in terms of plus.

```
[LAMBDA X
  (PROG ((M 0)
          (N 0))
        LP (COND
            ((EQ N X)
             (RETURN M)))
        (SETQ N (ADD1 N))
        [SETQ M (PLUS M (ARG X N)))
        (GO LP)]
```

The value of arg is undefined for m less than or equal to 0 or greater than the *value* of var.¹⁵ Lower numbered arguments appear earlier in the form, e.g., for (IPLUS A B C),

arg[X; 1] = the value of A,
arg[X; 2] = the value of B, and
arg[X; 3] = the value of C.

Note that the lambda variable should *never* be reset. However, individual arguments can be reset using setarg described below.

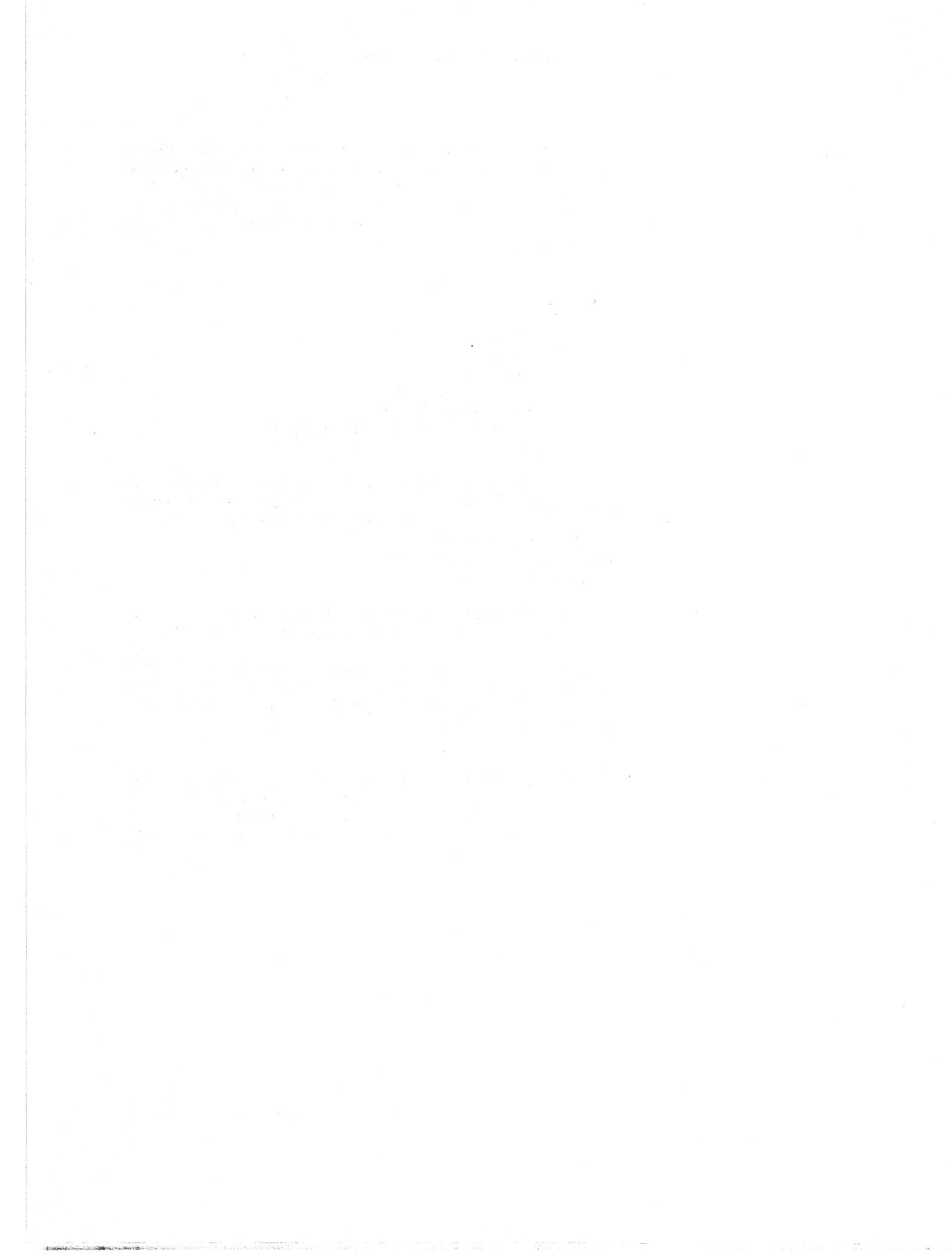
arg compiles open if var is bound locally in the function in which the call to arg appears. Otherwise, arg compiles closed and evaluates its first argument on each call. Note that this means that the value of var must be a SPECVAR.

setarg[var;m;x]

sets to x the mth argument for the lambda nospread function whose argument list is var. var is considered quoted, m and x are evaluated; e.g., in the previous example, (SETARG X (ADD1 N)(MINUS M)) would be an example of the correct form for setarg.

15

For lambda nospread functions, the lambda variable is bound to the number of arguments actually given to the function. See Section 4.



Index for Section 8

	Page Numbers
ADVISED (property name)	8.5
ALLPROP (as value of DFNFLG)	8.5
APPLY[FN;ARGS] SUBR	8.7
APPLY*[FN;ARG1;ARG2;...;ARGn] SUBR*	8.8
ARG[VAR;M] FSUBR	8.9
ARG NOT LITATOM (error message)	8.2
ARGLIST[FN]	8.4,1-3
ARGNAMES (property name)	8.5
ARGS NOT AVAILABLE (error message)	8.4
ARGTYPE[FN] SUBR	8.3,1-3
argument list	8.1
a-list	8.8
BAD ARGUMENT - FGETD (error message)	8.2
BOUNDP[VAR]	8.7
BROKEN (property name)	8.5
BROKEN-IN (property name)	8.5
CCODEP[FN] SUBR	8.3,1-3
CEXPR (function type)	8.3
CEXPR* (function type)	8.3-4
CFEXPR (function type)	8.3-4
CFEXPR* (function type)	8.3-4
CODE (property name)	8.6
COMPILETYPELST (compiler variable/parameter)	8.7
DEFEVAL[TYPE;FN]	8.7
DEFINE[X;TYPE-IN]	8.5
DEFINEQ[X1;X2;...;Xn] NL*	8.5
DFNFLG (system variable/parameter)	8.5,6
EVAL[X] SUBR	8.6
EVALA[X;A] SUBR	8.8
EXPR (function type)	8.3
EXPR (property name)	8.5,6
EXPRP[FN] SUBR	8.3,1-2,4
EXPR* (function type)	8.3-4
FEXPR (function type)	8.3-4
FEXPR* (function type)	8.3-4
FGETD[X]	8.2
FNCHECK[FN;NOERRORFLG;SPELLFLG;PROPFLOG;TAIL]	8.4
FNTYP[FN]	8.3,1-2,4,6
FRPTQ[N;FORM1;FORM2;...;FORMn] NL*	8.8
FSUBR (function type)	8.3-4
FSUBR* (function type)	8.3-4
FUNARG (function type)	8.3
function definition and evaluation	8.1-9
function definition cell	8.1
functional arguments	8.8
GETD[X] SUBR	8.2,1,6
ILLEGAL ARG (error message)	8.2
INCORRECT DEFINING FORM (error message)	8.5
interpreter	8.6
LAMBDA	8.2-3,6
LAMBDA SPLST (dwim variable/parameter)	8.4
LIST (property name)	8.6
MOVD[FROM;TO;COPYFLG]	8.3
MOVD?[FROM;TO;COPYFLG]	8.3
NARGS[FN]	8.4,1-3
NLAMBDA	8.2-3

Page
Numbers

NOBIND	8.7
nospread functions	8.1
NOT A FUNCTION (error message)	8.5-6
(NOT FOUND) (value of unsavedef)	8.6
(NOTHING FOUND)	8.6
PROP (as value of DFNFLG)	8.5
PUTD[FN;DEF] SUBR	8.2,1-2
PUTDQ[FN;DEF] NL	8.2
PUTDQ?[FN;DEF] NL	8.2
(REDEFINED) (printed by system)	8.5
RPT[RPTN;RPTF]	8.8
RPTQ[N;FORM1;FORM2;...;FORMn] NL*	8.8
SAVEDEF[NAME;TYPE;DEFINITION]	8.6
SCODEP[FN] SUBR	8.3,3
SETARG[VAR;M;X] FSUBR	8.9
SMARTARGLIST[FN;EXPLAINFLG;TAIL]	8.4
spread functions	8.1
SUBR. (function type)	8.3-4
SUBR (property name)	8.6
SUBRP[FN] SUBR	8.3,1-3
subrs	8.1
SUBR* (function type)	8.3-4
time stamp package	8.5
U (value of ARGLIST)	8.4
unbound atom	8.7
UNSAVEDEF[NAME;TYPE;DEF]	8.6

SECTION 9

THE INTERLISP EDITOR¹

The Interlisp editor allows rapid, convenient modification of list structures. Most often it is used to edit function definitions, (often while the function itself is running) via the function editf, e.g., EDITF(FOO). However, the editor can also be used to edit the value of a variable, via edity, to edit a property list, via editp, or to edit an arbitrary expression, via edite. It is an important feature which allows good on-line interaction in the Interlisp system.

This chapter begins with a lengthy introduction intended for the new user. The reference portion begins on page 9.10.

9.1 INTRODUCTION

Let us introduce some of the basic editor commands, and give a flavor for the editor's language structure by guiding the reader through a hypothetical editing session. Suppose we are editing the following incorrect definition of append:

```
[LAMBDA (X)
  Y
  (COND
    ((NUL X)
     Z)
    (T (CONS (CAR)
              (APPEND (CDR X Y]
```

We call the editor via the function editf:

```
<EDITF(APPEND)
EDIT
*
```

The editor responds by typing EDIT followed by *, which is the editor's prompt character, i.e., it signifies that the editor is ready to accept commands.²

¹ The editor was written by and is the responsibility of W. Teitelman.

² In other words, all lines beginning with * were typed by the user, the rest by the editor.

At any given moment, the editor's attention is centered on some substructure of the expression being edited. This substructure is called the *current expression*, and it is what the user sees when he gives the editor the command P, for print. Initially, the current expression is the top level one, i.e., the entire expression being edited. Thus:

```
*P
(LAMBDA (X) Y (COND & &))
*
```

Note that the editor prints the current expression as though printlevel were set to (2 . 20), i.e., sublists of sublists are printed as &, tails of long lists printed as -- . The command ? will print the current expression as though printlevel were 1000.

```
*?
(LAMBDA (X) Y (COND ((NUL X) Z) (T (CONS (CAR) (APPEND (CDR X Y))))))
*
```

and the command PP will prettyprint the current expression.

A positive integer is interpreted by the editor as a command to descend into the correspondingly numbered element of the current expression. Thus:

```
*2
*p
(X)
*
```

A negative integer has a similar effect, but counting begins from the end of the current expression and proceeds backward, i.e., -1 refers to the last element in the current expression, -2 the next to the last, etc. For either positive integer or negative integer, if there is no such element, an error occurs,³ the editor types the faulty command followed by a ?, and then another *. *The current expression is never changed when a command causes an error.* Thus:

```
*P
(X)
*2

2 ?
*1
*p
X
*
```

A phrase of the form "the current expression is changed" or 'the current expression becomes' refers to a shift in the editor's attention, not to a modification of the structure being edited.

³ 'Editor errors' are not of the flavor described in Section 16, i.e., they never cause breaks or even go through the error machinery but are direct calls to `error!` indicating that a command is in some way faulty. What happens next depends on the context in which the command was being executed. For example, there are conditional commands which branch on errors. In most situations, though, an error will cause the editor to type the faulty command followed by a ? and wait for more input. Note that typing control-E while a command is being executed aborts the command exactly as though it had caused an error.

When the user changes the current expression by descending into it, the old current expression is not lost. Instead, the editor actually operates by maintaining a *chain* of expressions leading to the current one. The current expression is simply the last link in the chain. Descending adds the indicated subexpression onto the end of the chain, thereby making it be the current expression. The command 0 is used to ascend the chain; it removes the last link of the chain, thereby making the *previous* link be the current expression. Thus:

```
*P
X
*0 P
(X)
*0 -1 P
(COND (& Z) (T &))
*
```

Note the use of several commands on a single line in the previous output. The editor operates in a line buffered mode, the same as evalqt. Thus no command is actually seen by the editor, or executed, until the line is terminated, either by a carriage return, or a matching right parenthesis. The user can thus use control-A and control-Q for line-editing edit commands, the same as he does for inputs to evalqt.

In our editing session, we will make the following corrections to append: delete Y from where it appears, add Y to the end of the argument list,⁴ change NUL to NULL, change Z to Y, add Z after CAR, and insert a right parenthesis following CDR X.

First we will delete Y. By now we have forgotten where we are in the function definition, but we want to be at the "top" so we use the command ↑, which ascends through the entire chain of expressions to the top level expression, which then becomes the current expression, i.e., ↑ removes all links except the first one.

```
*↑ P
(LAMBDA (X) Y (COND & &))
*
```

Note that if we are already at the top, ↑ has no effect, i.e., it is a no-op. However, 0 would generate an error. In other words, ↑ means "go to the top," while 0 means "ascend one link."

⁴

These two operations could be thought of as one operation, i.e., MOVE Y from its current position to a new position, and in fact there is a MOVE command in the editor. However, for the purposes of this introduction, we will confine ourselves to the simpler edit commands.

The basic structure modification commands in the editor are:

- (n) $n \geq 1$ deletes the corresponding element from the current expression.
- (n e₁ ... e_m) $n, m \geq 1$ replaces the nth element in the current expression with e₁ ... e_m.
- (-n e₁ ... e_m) $n, m \geq 1$ inserts e₁ ... e_m before the nth element in the current expression.

Thus:

```
*P
(LAMBDA (X) Y (COND & &))
*(3)
*(2 (X Y))
*P
(LAMBDA (X Y) (COND & &))
*
```

All structure modification done by the editor is destructive, i.e., the editor uses rplaca and rplacd to physically change the structure it was given.

Note that all three of the above commands perform their operation with respect to the nth element from the front of the current expression; the sign of n is used to specify whether the operation is replacement or insertion. Thus, there is no way to specify deletion or replacement of the nth element from the end of the current expression, or insertion before the nth element from the end without counting out that element's position from the front of the list. Similarly, because we cannot specify insertion after a particular element, we cannot attach something at the end of the current expression using the above commands. Instead, we use the command N (for nconc). Thus we could have performed the above changes instead by:

```
*P
(LAMBDA (X) Y (COND & &))
*(3)
*2 (N Y)
*P
(X Y)
*↑ P
*(LAMBDA (X Y) (COND & &))
*
```

Now we are ready to change NUL to NULL. Rather than specify the sequence of descent commands necessary to reach NUL, and then replace it with NULL, e.g., 3 2 1 (1 NUL), we will use F, the find command, to find NUL:

```
*P
(LAMBDA (X Y) (COND & &))
*F NUL
*P
(NUL X)
*(1 NULL)
*O P
((NULL X) Z)
*
```

Note that F is special in that it corresponds to *two* inputs. In other words, F says to the editor, "treat your *next* command as an expression to be searched for." The search is carried out in printout order in the current expression. If the target expression is not found there, F automatically ascends and searches those portions of the higher expressions that would appear after (in a printout) the current expression. If the search is successful, the new current expression will be the structure where the expression was found,⁵ and the chain will be the same as one resulting from the appropriate sequence of ascent and descent commands. If the search is not successful, an error occurs, and neither the current expression nor the chain is changed:⁶

```
*P
((NULL X) Z)
*F COND P

COND ?
*P
*((NULL X) Z)
*
```

Here the search failed to find a cond following the current expression, although of course a cond does appear earlier in the structure. This last example illustrates another facet of the error recovery mechanism: to avoid further confusion when an error occurs, all commands on the line *beyond* the one which caused the error (and all commands that may have been typed ahead while the editor was computing) are forgotten.⁷

We could also have used the R command (for replace) to change NUL to NULL. A command of the form (R e₁ e₂) will replace all occurrences of e₁ in the current expression by e₂. There must be at least one such occurrence or the R command will generate an error. Let us use the R command to change all Z's (even though there is only one) in append to Y:

⁵ If the search is for an atom, e.g., F NUL, the current expression will be the structure containing the atom.

⁶ F is never a no-op, i.e., if successful, the current expression after the search will never be the same as the current expression before the search. Thus F expr repeated without intervening commands that change the edit chain can be used to find successive instances of expr.

⁷ i.e., the input buffer is cleared (and saved) (see clearbuf, Section 14). It can be restored, and the type-ahead recovered via the command \$BUFS (esc BUFS), described in Section 22.

```

*↑ (R Z Y)
*F Z

Z ?
*PP
[LAMBDA (X Y)
(COND
((NULL X)
Y)
(T (CONS (CAR)
(APPEND (CDR X Y]
*

```

The next task is to change (CAR) to (CAR X). We could do this by (R (CAR) (CAR X)), or by:

```

*F CAR
*(N X)
*P
(CAR X)
*
```

The expression we now want to change is the next expression after the current expression, i.e., we are currently looking at (CAR X) in (CONS (CAR X) (APPEND (CDR X Y))). We could get to the append expression by typing 0 and then 3 or -1, or we can use the command NX, which does both operations:

```

*P
(CAR X)
*NX P
(APPEND (CDR X Y))
*
```

Finally, to change (APPEND (CDR X Y)) to (APPEND (CDR X) Y), we could perform (2 (CDR X) Y), or (2 (CDR X)) and (N Y), or 2 and (3), deleting the Y, and then 0 (N Y). However, if Y were a complex expression, we would not want to have to retype it. Instead, we could use a command which effectively inserts and/or removes left and right parentheses. There are six of these commands: BI, BO, LI, LO, RI, and RO, for both in, both out, left in, left out, right in, and right out. Of course, we will always have the same number of left parentheses as right parentheses, because the parentheses are just a notational guide to structure that is provided by our print program.⁸ Thus, left in, left out, right in, and right out actually do not insert or remove just one parenthesis, but this is very suggestive of what actually happens.

In this case, we would like a right parenthesis to appear following X in (CDR X Y). Therefore, we use the command (RI 2 2), which means insert a right parentheses after the second element in the second element (of the current expression):

⁸ Herein lies one of the principal advantages of a LISP oriented editor over a text editor: unbalanced parentheses errors are not possible.

```
*P
(APPEND (CDR X Y))
*(RI 2 2)
*P
(APPEND (CDR X) Y)
*
```

We have now finished our editing, and can exit from the editor, to test append, or we could test it while still inside of the editor, by using the E command:

```
*E APPEND((A B) (C.D E))
(A B C D E)
*
```

The E command causes the next input to be given to evalqt. If there is another input following it, as in the above example, the first will be applied (apply) to the second. Otherwise, the input is evaluated (eval).

We prettyprint append, and leave the editor.

```
*PP
[LAMBDA (X Y)
(COND
 ((NULL X)
 Y)
 (T (CONS (CAR X)
 (APPEND (CDR X) Y)))
*OK
APPEND
←
```

9.2 COMMANDS FOR THE NEW USER

As mentioned earlier, the Interlisp manual is intended primarily as a reference manual, and the remainder of this chapter is organized and presented accordingly. While the commands introduced in the previous scenario constitute a complete set, i.e., the user could perform any and all editing operations using just those commands, there are many situations in which knowing the right command(s) can save the user considerable effort. We include here as part of the introduction a list of those commands which are not only frequently applicable but also easy to use. They are not presented in any particular order, and are all discussed in detail in the reference portion of the chapter.

UNDO

undoes the last modification to the structure being edited, e.g., if the user deletes the wrong element, UNDO will restore it. The availability of UNDO should give the user confidence to experiment with any and all editing commands, no matter how complex, because he can always reverse the effect of the command.

BK

like NX, except makes the expression immediately *before* the current expression become current.

BF backwards find. Like F, except searches backwards, i.e., in inverse print order.

\ Restores the current expression to the expression before the last "big jump", e.g., a find command, an ↑, or another \. For example, if the user types F COND, and then F CAR, \ would take him back to the COND. Another \ would take him back to the CAR.

\P like \ except it restores the edit chain to its state as of the last print, either by P, ?, or PP. If the edit chain has not been changed since the last print, \P restores it to its state as of the printing before that one, i.e., two chains are always saved.

Thus if the user types P followed by 3 2 1 P, \P will take him back to the first P, i.e., would be equivalent to 0 0 0. Another \P would then take him back to the second P. Thus the user can use \P to flip back and forth between two current expressions.

&-- The search expression given to the F or BF command need not be a literal S-expression. Instead, it can be a pattern. The symbol & can be used anywhere within this pattern to match with any single *element* of a list, and -- can be used to match with any *segment* of a list. Thus, in the incorrect definition of append used earlier, F (NUL &) could have been used to find (NUL X), and F (CDR --) or F (CDR & &), but not F (CDR &), to find (CDR X Y).

Note that & and -- can be nested arbitrarily deeply in the pattern. For example, if there are many places where the variable X is set, F SETQ may not find the desired expression, nor may F (SETQ X &). It may be necessary to use F (SETQ X (LIST --)). However, the usual technique in such a case is to pick out a unique atom which occurs prior to the desired expression, and perform two F commands. This "homing in" process seems to be more convenient than ultra-precise specification of the pattern.

\$ (< esc >) \$ is equivalent to -- at the character level, e.g., VER\$ will match with VERYLONGATOM, as will \$ATOM, \$LONG\$, (but not \$LONG) and \$V\$N\$M\$. \$ can be nested inside of a pattern, e.g., F (SETQ VER\$ (CONS --)).

If the search is successful, the editor will print = followed by the atom which matched with the \$-atom, e.g.,

*F (SETQ VER\$ &
=VERYLONGATOM
*

Frequently the user will want to replace the entire current expression, or insert something before it. In order to do this using a command of the form (n e₁ ... e_m) or (-n e₁ ... e_m), the user must be above the current expression. In other words, he would have to perform a 0 followed by a

command with the appropriate number. However, if he has reached the current expression via an F command, he may not know what that number is. In this case, the user would like a command whose effect would be to modify the edit chain so that the current expression became the first element in a new, higher current expression. Then he could perform the desired operation via (1 $e_1 \dots e_m$) or (-1 $e_1 \dots e_m$). UP is provided for this purpose.

UP

after UP operates, the old current expression is the first element of the new current expression. Note that if the current expression happens to be the first element in the next higher expression, then UP is exactly the same as 0. Otherwise, UP modifies the edit chain so that the new current expression is a tail⁹ of the next higher expression:

```
*F APPEND P
(APPEND (CDR X) Y)
*UP P
... (APPEND & Y))
*0 P
(CONS (CAR X) (APPEND & Y))
*
```

The ... is used by the editor to indicate that the current expression is a *tail* of the next higher expression as opposed to being an element (i.e., a member) of the next higher expression. Note: if the current expression is *already* a tail, UP has no effect.

(B $e_1 \dots e_m$)

inserts $e_1 \dots e_m$ before the current expression, i.e., does an UP and then a -1.

(A $e_1 \dots e_m$)

inserts $e_1 \dots e_m$ after the current expression, i.e., does an UP and then either a (-2 $e_1 \dots e_m$) or an (N $e_1 \dots e_m$), if the current expression is the last one in the next higher expression.

(: $e_1 \dots e_m$)

replaces current expression by $e_1 \dots e_m$, i.e., does an UP and then a (1 $e_1 \dots e_m$).

DELETE

deletes current expression; equivalent to (:).

Earlier, we introduced the RI command in the append example. The rest of the commands in this family: BI, BO, LI, LO, and RO, perform similar functions and are useful in certain situations. In addition, the commands MBD and XTR can be used to combine the effects of several commands of the BI-BO family. MBD is used to embed the current expression in a larger expression. For example, if the current expression is (PRINT bigexpression), and the user wants to replace it by

⁹ Throughout this chapter "tail" means "proper tail" (see Section 5).

(COND (FLG (PRINT bigexpression))), he could accomplish this by (LI 1), (-1 FLG), (LI 1), and (-1 COND), or by a single MBD command, page 9.30.

XTR is used to extract an expression from the current expression. For example, extracting the PRINT expression from the above COND could be accomplished by (1), (LO 1), (1), and (LO 1) or by a single XTR command. The new user is encouraged to include XTR and MBD in his repertoire as soon as he is familiar with the more basic commands.

This ends the introductory material.

9.3 ATTENTION CHANGING COMMANDS

Commands to the editor fall into three classes: commands that change the current expression (i.e., change the edit chain) thereby "shifting the editor's attention," commands that modify the structure being edited, and miscellaneous commands, e.g., exiting from the editor, printing, evaluating expressions, etc.

Within the context of commands that shift the editor's attention, we can distinguish among (1) those commands whose operation depends only on the *structure* of the edit chain, e.g., 0, UP, NX; (2) those which depend on the *contents* of the structure, i.e., commands that search; and (3) those commands which simply restore the edit chain to some previous state, e.g., \, \P. (1) and (2) can also be thought of as local, small steps versus open ended, big jumps. Commands of type (1) are discussed on page 9.10-14, type (2) on page 9.14-22, and type (3) on page 9.22-23.

9.3.1 LOCAL ATTENTION-CHANGING COMMANDS

UP

- (1) If a P command would cause the editor to type ... before typing the current expression, i.e., the current expression is a tail of the next higher expression, UP has no effect; otherwise
- (2) UP modifies the edit chain so that the old current expression (i.e., the one at the time UP was called) is the first element in the new current expression.¹⁰

¹⁰ If the current expression is the first element in the next higher expression UP simply does a 0. Otherwise UP adds the corresponding tail to the edit chain.

Examples: The current expression in each case is (COND ((NULL X) (RETURN Y))).

1. *1 P
COND
*UP P
(COND (& &))
2. *-1 P
((NULL X) (RETURN Y))
*UP P
... ((NULL X) (RETURN Y))
*UP P
... ((NULL X) (RETURN Y)))
3. *F NULL P
(NULL X)
*UP P
((NULL X) (RETURN Y))
*UP P
... ((NULL X) (RETURN Y)))

The execution of UP is straightforward, except in those cases where the current expression appears more than once in the next higher expression. For example, if the current expression is (A NIL B NIL C NIL) and the user performs 4 followed by UP, the current expression should then be ... NIL C NIL). UP can determine which tail is the correct one because the commands that descend save the last tail on an internal editor variable, lastail. Thus after the 4 command is executed, lastail is (NIL C NIL). When UP is called, it first determines if the current expression is a tail of the next higher expression. If it is, UP is finished. Otherwise, UP computes memb[current-expression;next-higher-expression] to obtain a tail beginning with the current expression.¹¹ If there are no other instances of the current expression in the next higher expression, this tail is the correct one. Otherwise UP uses lastail to select the correct tail.¹²

n (n ≥ 1) adds the nth element of the current expression to the front of the edit chain, thereby making it be the new current expression. Sets lastail for use by UP. Generates an error if the current expression is not a list that contains at least n elements.

-n (n ≥ 1) adds the nth element from the end of the current expression to the front of the edit chain, thereby making it be the new current expression. Sets lastail for use by UP. Generates an error if the current expression is not a list that contains at least n elements.

¹¹ The current expression should *always* be either a tail or an element of the next higher expression. If it is neither, for example the user has directly (and incorrectly) manipulated the edit chain, UP generates an error.

¹² Occasionally the user can get the edit chain into a state where lastail cannot resolve the ambiguity, for example if there were two non-atomic structures in the same expression that were eq, and the user descended more than one level into one of them and then tried to come back out using UP. In this case, UP prints LOCATION UNCERTAIN and generates an error. Of course, we could have solved this problem completely in our implementation by saving at each descent both elements and tails. However, this would be a costly solution to a situation that arises infrequently, and when it does, has no detrimental effects. The lastail solution is cheap and resolves 99% of the ambiguities.

- 0** Sets edit chain to cdr of edit chain, thereby making the next higher expression be the new current expression. Generates an error if there is no higher expression, i.e., cdr of edit chain is **NIL**.

Note that 0 usually corresponds to going back to the next higher left parenthesis, but not always. For example, if the current expression is **(A B C D E F B)**, and the user performs:

```
*3 UP P
... C D E F G)
*3 UP P
... E F G)
*0 P
... C D E F G)
```

If the intention is to go back to the next higher left parenthesis, regardless of any intervening tails, the command !0 can be used.¹³

- !0** does repeated 0's until it reaches a point where the current expression is *not* a tail of the next higher expression, i.e., always goes back to the next higher left parenthesis.

- ↑** sets edit chain to last of edit chain, thereby making the top level expression be the current expression. Never generates an error.

- NX** effectively does an **UP** followed by a **2**,¹⁴ thereby making the current expression be the next expression. Generates an error if the current expression is the last one in a list. (However, **INX** described below will handle this case.)

- BK** makes the current expression be the previous expression in the next higher expression. Generates an error if the current expression is the first expression in a list.

For example, if the current expression is **(COND ((NULL X) (RETURN Y)))**:

```
*F RETURN P
(RETURN Y)
*BK P
(NULL X)
```

- (NX n) n ≥ 1** equivalent to n **NX** commands, except if an error occurs, the edit chain is not changed.

¹³ !0 is pronounced bang-zero.

¹⁴ Both **NX** and **BK** operate by performing a !0 followed by an appropriate number, i.e., there won't be an extra tail above the new current expression, as there would be if **NX** operated by performing an **UP** followed by a **2**.

(BK n) n \geq 1

equivalent to n BK commands, except if an error occurs, the edit chain is not changed.

Note: (NX -n) is equivalent to (BK n), and vice versa.

!NX

makes current expression be the next expression at a higher level, i.e., goes through any number of right parentheses to get to the next expression.

For example:

```
*PP
(PROG ((L L)
        (UF L))
      LP (COND
            ((NULL (SETQ L (CDR L)))
             (ERROR!))
            ([NULL (CDR (FMEMB (CAR L)
                                (CADR L)
                                (GO LP)))
              (EDITCOM (QUOTE NX))
              (SETQ UNFIND UF)
              (RETURN L))
             *F CDR P
             (CDR L)
             *NX
             NX ?
             *!NX P
             (ERROR!)
             *!NX P
             ((NULL &) (GO LP))
             *!NX P
             (EDITCOM (QUOTE NX))
             *
```

!NX operates by doing 0's until it reaches a stage where the current expression is *not* the last expression in the next higher expression, and then does a NX. Thus !NX always goes through at least one unmatched right parenthesis, and the new current expression is always on a different level, i.e., !NX and NX always produce different results. For example using the previous current expression:

```
*F CAR P
(CAR L)
*!NX P
(GO LP)
*\P P
(CAR L)
*NX P
(CADR L)
*
```

(NTH n) n ≠ 0

equivalent to n followed by UP, i.e., causes the list starting with the nth element of the current expression (or nth from the end if n < 0) to become the current expression.¹⁵ Causes an error if current expression does not have at least n elements.

A generalized form of NTH using location specifications is described on page 9.21.

- + <line-feed> moves to the "next" expression and prints it, i.e. performs a NX if possible, otherwise performs a !NX. (The latter case is indicated by first printing ">".)
- + <control-x>¹⁶ moves to "previous" thing and then prints it, i.e. performs a BK if possible, otherwise a !0 followed by a BK.
- + <control-z>¹⁷ moves to last expression and prints it, i.e. does -1 followed by P.
- + Both <line-feed>, <control-x>, and <control-z> are *immediate* read macros; as soon as they are read, they abort the current printout. They thus provide a convenient way of moving around in the editor.¹⁸

9.3.2 COMMANDS THAT SEARCH

All of the editor commands that search use the same pattern matching routine.¹⁹ We will therefore begin our discussion of searching by describing the pattern match mechanism. A pattern pat matches with x if:

1. pat is eq to x.
2. pat is &.
3. pat is a number and eqp to x.
4. pat is a string and strequal[pat;x] is true.
5. If car[pat] is the atom *ANY*, cdr[pat] is a list of patterns and pat matches x if and only if one of the patterns on cdr[pat] matches x.

¹⁵ (NTH 1) is a no-op, as is (NTH -n) where n is the length of the current expression.

+ ¹⁶ <control-A> in Interlisp on TOPS-20.

+ ¹⁷ <control-L> in Interlisp on TOPS-20.

+ ¹⁸ In order to facilitate using different control characters for those macros, the function settermchars is provided. It is described in Section 14.

¹⁹ This routine is available to the user directly, and is described on page 9.63.

- 6a. If pat is a literal atom or string containing one or more <esc>s,²⁰ each \$ can match an indefinite number (including 0) of contiguous characters in a literal atom or string, e.g., VER\$ matches both VERYLONGATOM and "VERYLONGSTRING" as do \$LONG\$ (but not \$LONG), and \$V\$L\$T\$.
- 6b. If pat is a literal atom or string ending in *two* <esc>s, pat matches with the first atom or string that is "close" to pat, in the sense used by the spelling corrector (Section 17). E.g. CONSS\$\$ matches with CONS, CNONC\$\$ with NCONC or NCONC1.
 The pattern matching routine always types a message of the form =x to inform the user of the object matched by a pattern of type 6a or 6b,²¹ e.g. =VERYLONGATOM.
7. If car[pat] is the atom --, pat matches x if
- a. cdr[pat]=NIL, i.e., pat=(--), e.g.
 (A --) matches (A) (A B C) and (A . B)
 In other words, -- can match any tail of a list.
 - b. cdr[pat] matches with some tail of x,
 e.g., (A -- (&)) will match with (A B C (D)),
 but not (A B C D), or (A B C (D) E). However,
 note that (A -- (&) --) will match with
 (A B C (D) E).
 In other words, -- can match any interior segment of a list.
8. If car[pat] is the atom ==, pat matches x if and only if cdr[pat] is eq to x.²²
9. If cadr[pat] is the atom .., pat matches x if car[pat] matches car[x] and cddr[pat] is contained in x, as described on page 9.21. +
10. Otherwise if x is a list, pat matches x if car[pat] matches car[x], and cdr[pat] matches cdr[x].

When the editor is searching, the pattern matching routine is called to match with *elements* in the structure, unless the pattern begins with ..., in which case cdr of the pattern is matched against proper tails in the structure. Thus if the current expression is (A B C (B C)),

```
*F (B --)
*P (B C)
*O F (... B --)
*P
... B C (B C))
```

Matching is also attempted with atomic tails (except for NIL). Thus

²⁰ except that the atom \$ (<esc>) matches only with itself.

²¹ unless editquietflg=T.

²² Pattern 8 is for use by programs that call the editor as a subroutine, since any non-atomic expression in a command typed in by the user obviously cannot be eq to already existing structure.

```
*P
(A (B . C))
*F C
*P
.... . C)
```

Although the current expression is the atom C after the final command, it is printed as C) to alert the user to the fact that C is a *tail*, not an element. Note that the pattern C will match with either instance of C in (A C (B . C)), whereas (... . C) will match only the second C. The pattern NIL will only match with NIL as an element, i.e., it will not match in (A B), even though cddr of (A B) is NIL. However, (... . NIL) (or equivalently (...) may be used to specify a NIL *tail*, e.g., (... . NIL) will match with cdr of the third subexpression of ((A . B) (C . D) (E)).

SEARCH ALGORITHM

Searching begins with the current expression and proceeds in print order. Searching usually means find the next instance of this pattern, and consequently a match is not attempted that would leave the edit chain unchanged.²³ At each step, the pattern is matched against the next element in the expression currently being searched, unless the pattern begins with ... in which case it is matched against the next tail of the expression.

If the match is not successful, the search operation is recursive first in the car direction, and then in the cdr direction, i.e., if the element under examination is a list, the search descends into that list before attempting to match with other elements (or tails) at the same level.²⁴

However, at no point is the total recursive depth of the search (sum of number of cars and cdrs descended into) allowed to exceed the value of the variable maxlevel. At that point, the search of that element or tail is abandoned, exactly as though the element or tail had been completely searched without finding a match, and the search continues with the element or tail for which the recursive depth is below maxlevel. This feature is designed to enable the user to search circular list structures (by setting maxlevel small), as well as protecting him from accidentally encountering a circular list structure in the course of normal editing. maxlevel is initially set to 300.²⁵

If a successful match is not found in the current expression, the search automatically ascends to the next higher expression,²⁶ and continues searching there on the next expression after the expression it just finished searching. If there is none, it ascends again, etc. This process continues until the entire edit chain has been searched, at which point the search fails, and an error is generated. If the search fails (or, what is equivalent, is aborted by control-E), the edit chain is not changed (nor are any conses performed).

²³ However, there is a version of the find command which can succeed and leave the current expression unchanged (see page 9.17).

²⁴ There is also a version of the find command (see page 9.18) which only attempts matches at the top level of the current expression, i.e., does not descend into elements, or ascend to higher expressions.

²⁵ maxlevel can also be set to NIL, which is equivalent to infinity.

²⁶ See footnote 21.

If the search is successful, i.e., an expression is found that the pattern matches, the edit chain is set to the value it would have had had the user reached that expression via a sequence of integer commands.

If the expression that matched was a list, it will be the final link in the edit chain, i.e., the new current expression. If the expression that matched is not a list, e.g., is an atom, the current expression will be the tail beginning with that atom,²⁷ i.e., that atom will be the first element in the new current expression. In other words, the search effectively does an UP.²⁸

SEARCH COMMANDS

All of the commands below set lastail for use by UP, set unfind for use by \ (page 9.23), and do not change the edit chain or perform any conses if they are unsuccessful or aborted.

F pattern

i.e., two commands: the F informs the editor that the *next* command is to be interpreted as a pattern. This is the most common and useful form of the find command. If successful, the edit chain always changes, i.e., F pattern means find the next instance of pattern.

If memb[pattern;current-expression] is true, F does not proceed with a full recursive search. If the value of the memb is NIL, F invokes the search algorithm described earlier.

Thus if the current expression is (PROG NIL LP (COND (-- (GO LP1))) ... LP1 ...), F LP1 will find the prog label, not the LP1 inside of the GO expression, even though the latter appears first (in print order) in the current expression. Note that 1 (making the atom PROG be the current expression), followed by F LP1 would find the first LP1.

(F pattern N)

same as F pattern, i.e., finds the next instance of pattern, except the memb check of F pattern is not performed.

(F pattern T)

Similar to F pattern, except may succeed without changing edit chain, and does not perform the memb check.

Thus if the current expression is (COND ..), F COND will look for the next COND, but (F COND T) will "stay here".

(F pattern n) n \geq 1

Finds the nth place that pattern matches. Equivalent to (F pattern T) followed by (F pattern N) repeated n-1 times. Each time pattern successfully matches, n is decremented by 1, and the search continues, until n reaches 0. Note that the pattern does not have to

²⁷ Unless the atom is a tail, e.g., B in (A . B). In this case, the current expression will be B, but will print as B.

²⁸ Unless upfindflg = NIL (initially set to T). For discussion, see page 9.28-29.

match with n identical expressions; it just has to match n times. Thus if the current expression is (FOO1 FOO2 FOO3), (F FOO\$ 3) will find FOO3.

If the pattern does not match successfully n times, an error is generated and the edit chain is unchanged (even if the pattern matched $n-1$ times).

(F pattern) or
(F pattern NIL)

only matches with elements at the top level of the current expression, i.e., the search will not descend into the current expression, nor will it go outside of the current expression. May succeed without changing edit chain.

For example, if the current expression is (PROG NIL (SETQ X (COND & &)) (COND &) ...), F COND will find the COND inside the SETQ, whereas (F (COND --)) will find the top level COND, i.e., the second one.

(FS pattern₁ ... pattern_n)

equivalent to F pattern₁ followed by F pattern₂ ... followed by F pattern_n, so that if F pattern_m fails, edit chain is left at place pattern_{m-1} matched.

(F= expression x)

equivalent to (F (= . expression) x), i.e., searches for a structure eq to expression, see page 9.15.

(ORF pattern₁ ... pattern_n)

equivalent to (F (*ANY* pattern₁ ... pattern_n) N), i.e., searches for an expression that is matched by either pattern₁, pattern₂, ... or pattern_n. See page 9.14.

BF pattern

backwards find. Searches in reverse print order, beginning with expression immediately before the current expression (unless the current expression is the top level expression, in which case BF searches the entire expression, in reverse order).

BF uses the same pattern match routine as F, and maxlevel and upfindflg have the same effect, but the searching begins at the end of each list, and descends into each element before attempting to match that element. If unsuccessful, the search continues with the next previous element, etc., until the front of the list is reached, at which point BF ascends and backs up, etc.

For example, if the current expression is

(PROG NIL (SETQ X (SETQ Y (LIST Z))) (COND ((SETQ W --) --)) --), F LIST followed by BF SETQ will leave the current expression as (SETQ Y (LIST Z)), as will F COND followed by BF SETQ.

(BF pattern T)

search always includes current expression, i.e., starts at the end of current expression and works backward, then ascends and backs up, etc.

Thus in the previous example, where F COND followed by BF SETQ found (SETQ Y (LIST Z)), F COND followed by (BF SETQ T) would find the (SETQ W --) expression.

(BF pattern)	same as BF pattern.	
(BF pattern NIL)		
(GO label)	makes the current expression be the first thing after the <u>prog</u> label + label, i.e. goes where an executed <u>go</u> would go.	+

LOCATION SPECIFICATION

Many of the more sophisticated commands described later in this chapter use a more general method of specifying position called a location specification. A location specification is a list of edit commands that are executed in the normal fashion with two exceptions. First, all commands not recognized by the editor are interpreted as though they had been preceded by F.²⁹ For example, the location specification (COND 2 3) specifies the 3rd element in the first clause of the next COND.³⁰

Secondly, if an error occurs while evaluating one of the commands in the location specification, and the edit chain had been changed, i.e., was not the same as it was at the beginning of that execution of the location specification, the location operation will continue. In other words, the location operation keeps going unless it reaches a state where it detects that it is "looping", at which point it gives up. Thus, if (COND 2 3) is being located, and the first clause of the next COND contained only two elements, the execution of the command 3 would cause an error. The search would then continue by looking for the next COND. However, if a point were reached where there were no further CONDs, then the first command, COND, would cause the error; the edit chain would not have been changed, and so the entire location operation would fail, and cause an error.

The IF command in conjunction with the # # function provide a way of using arbitrary predicates applied to elements in the current expression. IF and # # will be described in detail later in the chapter, along with examples illustrating their use in location specifications.

Throughout this chapter, the meta-symbol @ is used to denote a location specification. Thus @ is a list of commands interpreted as described above. @ can also be atomic, in which case it is interpreted as list[@].

(LC . @)	provides a way of explicitly invoking the location operation, e.g., (LC COND 2 3) will perform the the search described above.
(LCL . @)	Same as LC except the search is confined to the current expression, i.e., the edit chain is rebound during the search so that it looks as though the editor were called on just the current expression. For

²⁹ Normally such commands would cause errors.

³⁰ Note that the user could always write F COND followed by 2 and 3 for (COND 2 3) if he were not sure whether or not COND was the name of an atomic command.

example, to find a COND containing a RETURN, one might use the location specification (COND (LCL RETURN) \) where the \ would reverse the effects of the LCL command, and make the final current expression be the COND.

(2ND . @) Same as (LC . @) followed by another (LC . @) except that if the first succeeds and second fails, no change is made to the edit chain.

(3RD . @) Similar to 2ND.

(\leftarrow pattern) ascends the edit chain looking for a link which matches pattern. In other words, it keeps doing 0's until it gets to a specified point. If pattern is atomic, it is matched with the first element of each link, otherwise with the entire link.³¹

For example:

```
*PP
[PROG NIL
  (COND
    [(NULL (SETQ L (CDR L)))
     (COND
       [(FLG (RETURN L)]
       ([NULL (CDR (FMEMB (CAR L)
                           (CADR L)))]))]
      *F CADR
      *( $\leftarrow$  COND)
      *P
      (COND (& &) (& &))
      *
```

Note that this command differs from BF in that it does not search *inside* of each link, it simply ascends. Thus in the above example, F CADR followed by BF COND would find (COND (FLG (RETURN L))), not the higher COND.

If no match is found, an error is generated, and the edit chain is unchanged.

(BELOW com x) ascends the edit chain looking for a link specified by com, and stops x³² links below that,³³ i.e. BELOW keeps doing 0's until it gets to a specified point, and then backs off x 0's.

³¹ If pattern is of the form (IF expression), expression is evaluated at each link, and if its value is NIL, or the evaluation causes an error, the ascent continues.

³² x is evaluated, e.g., (BELOW com (IPLUS X Y)).

³³ Only links that are elements are counted, not tails.

(BELOW com) same as (BELOW com 1).

For example, (BELOW COND) will cause the cond clause containing the current expression to become the new current expression. Thus if the current expression is as shown above, F CADR followed by (BELOW COND) will make the new expression be ([NULL (CDR (FMEMB (CAR L) (CADR L] (GO LP))), and is therefore equivalent to 0 0 0 0.

The BELOW command is useful for locating a substructure by specifying something it contains. For example, suppose the user is editing a list of lists, and wants to find a sublist that contains a FOO (at any depth). He simply executes F FOO (BELOW \).

(NEX x) same as (BELOW x) followed by NX.

For example, if the user is deep inside of a SELECTQ clause, he can advance to the next clause with (NEX SELECTQ).

NEX same as (NEX \leftarrow).

The atomic form of NEX is useful if the user will be performing repeated executions of (NEX x). By simply MARKing (see page 9.22) the chain corresponding to x, he can use NEX to step through the sublists.

(NTH x) generalized NTH command. Effectively performs (LCL . x), followed by (BELOW \), followed by UP.

In other words, NTH locates x, using a search restricted to the current expression, and then backs up to the current level, where the new current expression is the tail whose first element contains, however deeply, the expression that was the terminus of the location operation. For example:

```
*P
(PROG (& &) LP (COND & &) (EDITCOM &) (SETQ UNFIND UF) (RETURN L))
*(NTH UF)
*P
... (SETQ UNFIND UF) (RETURN L))
*
```

If the search is unsuccessful, NTH generates an error and the edit chain is not changed.

Note that (NTH n) is just a special case of (NTH x), and in fact, no special check is made for x a number; both commands are executed identically.

(pattern .. @)³⁴ e.g., (COND .. RETURN). Finds a cond that contains a return, at

³⁴ An infix command, "..." is not a meta-symbol, it is the name of the command. @ is caddr of the command. Note that (pattern .. @) can also be used directly as an edit pattern as described on page 9.15, e.g. F (pattern .. @).

any depth. Equivalent to (but more efficient than) (F pattern N), (LCL . @) followed by (\leftarrow pattern).

For example, if the current expression is

(PROG NIL [COND ((NULL L) (COND (FLG (RETURN L] --)), then (COND .. RETURN) will make (COND (FLG (RETURN L))) be the current expression. Note that it is the innermost COND that is found, because this is the first COND encountered when ascending from the RETURN. In other words, (pattern .. @) is not *always* equivalent to (F pattern N), followed by (LCL . @) followed by \.

Note that @ is a location specification, not just a pattern. Thus (RETURN .. COND 2 3) can be used to find the RETURN which contains a COND whose first clause contains (at least) three elements. Note also that since @ permits any edit command, the user can write commands of the form (COND .. (RETURN .. COND)), which will locate the first COND that contains a RETURN that contains a COND.

9.3.3 COMMANDS THAT SAVE AND RESTORE THE EDIT CHAIN

Several facilities are available for saving the current edit chain and later retrieving it: MARK, which marks the current chain for future reference, \leftarrow ³⁵ which returns to the last mark without destroying it, and $\leftarrow\leftarrow$, which returns to the last mark and also erases it.

MARK

adds the current edit chain to the front of the list marklst.

\leftarrow

makes the new edit chain be (CAR MARKLST). Generates an error if marklst is NIL, i.e., no MARKs have been performed, or all have been erased.

$\leftarrow\leftarrow$

similar to \leftarrow but also erases the MARK, i.e., performs (SETQ MARKLST (CDR MARKLST)).

Note that if the user has two chains marked, and wishes to return to the first chain, he must perform $\leftarrow\leftarrow$, which removes the second mark, and then \leftarrow . However, the second mark is then no longer accessible. If the user wants to be able to return to either of two (or more) chains, he can use the following generalized MARK:

(MARK atom)

sets atom to the current edit chain,

(\ atom)

makes the current edit chain become the value of atom.

If the user did not prepare in advance for returning to a particular edit chain, he may still be able to return to that chain with a single command by using \ or \P.

³⁵ An atomic command; do not confuse \leftarrow with the list command (\leftarrow pattern).

\

makes the edit chain be the value of unfind. Generates an error if unfind=NIL.

unfind is set to the current edit chain by each command that makes a "big jump", i.e., a command that usually performs more than a single ascent or descent, namely \uparrow , \downarrow , \leftrightarrow , INX , all commands that involve a search, e.g., F, LC, ..., BELOW, et al and \ and \P themselves.³⁶

For example, if the user types F COND, and then F CAR, \ would take him back to the COND. Another \ would take him back to the CAR, etc.

\P

restores the edit chain to its state as of the last print operation, i.e., P, ?, or PP. If the edit chain has not changed since the last printing, \P restores it to its state as of the printing before that one, i.e., two chains are always saved.

For example, if the user types P followed by 3 2 1 P, \P will return to the first P, i.e., would be equivalent to 0 0 0.³⁷ Another \P would then take him back to the second P, i.e., the user could use \P to flip back and forth between the two edit chains.

(S var . @)

Sets var (using setq) to the current expression after performing (LC . @). Edit chain is not changed.

Thus (S FOO) will set foo to the current expression, (S FOO -1 1) will set foo to the first element in the last element of the current expression.

This ends the section on "Attention Changing Commands."

³⁶ Except that unfind is not reset when the current edit chain is the top level expression, since this could always be returned to via the \uparrow command.

³⁷ Note that if the user had typed P followed by F COND, he could use either \ or \P to return to the P, i.e., the action of \ and \P are independent.

9.4 COMMANDS THAT MODIFY STRUCTURE

The basic structure modification commands in the editor are:

- (n) $n \geq 1$ deletes the corresponding element from the current expression.
- (n e₁ ... e_m) $n, m \geq 1$ replaces the n th element in the current expression with e₁ ... e_m.
- (-n e₁ ... e_m) $n, m \geq 1$ inserts e₁ ... e_m before the n th element in the current expression.
- (N e₁ ... e_m) $m \geq 1$ attaches e₁ ... e_m at the end of the current expression.

As mentioned earlier:

all structure modification done by the editor is destructive, i.e., the editor uses rplaca and rplacd to physically change the structure it was given.

However, all structure modification is undoable, see UNDO page 9.54.

All of the above commands generate errors if the current expression is not a list, or in the case of the first three commands, if the list contains fewer than n elements. In addition, the command (1), i.e., delete the first element, will cause an error if there is only one element, since deleting the first element must be done by replacing it with the second element, and then deleting the second element. Or, to look at it another way, deleting the first element when there is only one element would require changing a list to an atom (i.e., to NIL) which cannot be done.³⁸

If the value of changesarray is a hash array, the editor will mark all structures that are changed by doing puthash[structure;fn;changesarray], where fn is the name of the function. The algorithm used for marking is as follows:

- (1) If the expression is inside of another expression already marked as being changed, do nothing.
- (2) If the change is an insertion of or replacement with a list, mark the list as changed.
- (3) If the change is an insertion of or replacement with an atom, or a deletion, mark the parent as changed.

³⁸ However, the command DELETE will work even if there is only one element in the current expression, since it will ascend to a point where it *can* do the deletion.

changesarray is primarily for use by prettyprint (Section 14). When the value of changechar is not NIL, prettyprint, when printing to a file or display terminal, prints changechar in the right margin while printing an expression marked as having been changed. changechar is initially |.

9.4.1 IMPLEMENTATION OF STRUCTURE MODIFICATION COMMANDS

Note: Since all commands that insert, replace, delete or attach structure use the same low level editor functions, the remarks made here are valid for all structure changing commands.

For all replacement, insertion, and attaching at the end of a list, unless the command was typed in directly to the editor,³⁹ copies of the corresponding structure are used, because of the possibility that the exact same command, (i.e., same list structure) might be used again. Thus if a program constructs the command (1 (A B C)) e.g., via (LIST 1 FOO), and gives this command to the editor, the (A B C) used for the replacement will not be eq to foo.⁴⁰

The rest of this section is included for applications wherein the editor is used to modify a data structure, and pointers into that data structure are stored elsewhere. In these cases, the actual mechanics of structure modification must be known in order to predict the effect that various commands may have on these outside pointers. For example, if the value of foo is cdr of the current expression, what will the commands (2), (3), (2 X Y Z), (-2 X Y Z), etc. do to foo?

Deletion of the first element in the current expression is performed by replacing it with the second element and deleting the second element by patching around it. Deletion of any other element is done by patching around it, i.e., the previous tail is altered. Thus if foo is eq to the current expression which is (A B C D), and fie is cdr of foo, after executing the command (1), foo will be (B C D) (which is equal but not eq to fie). However, under the same initial conditions, after executing (2) fie will be unchanged, i.e., fie will still be (B C D) even though the current expression and foo are now (A C D).⁴¹

Both replacement and insertion are accomplished by smashing both car and cdr of the corresponding tail. Thus, if foo were eq to the current expression, (A B C D), after (1 X Y Z), foo would be (X Y Z B C D). Similarly, if foo were eq to the current expression, (A B C D), then after (-1 X Y Z), foo would be (X Y Z A B C D).

The N command is accomplished by smashing the last cdr of the current expression a la nconc. Thus if foo were eq to any tail of the current expression, after executing an N command, the corresponding expressions would also appear at the end of foo.

³⁹ Some editor commands take as arguments a list of edit commands, e.g., (LP F FOO (1 (CAR FOO))). In this case, the command (1 (CAR FOO)) is not considered to have been "typed in" even though the LP command itself may have been typed in. Similarly, commands originating from macros, or commands given to the editor as arguments to editf, editv, et al, e.g., EDITF(FOO F COND (N --)) are not considered typed in.

⁴⁰ The user can circumvent this by using the I command, which computes the structure to be used. In the above example, the form of the command would be (I 1 FOO), which would replace the first element with the value of foo itself. See page 9.42.

⁴¹ A general solution of the problem just isn't possible, as it would require being able to make two lists eq to each other that were originally different. Thus if fie is cdr of the current expression, and fum is cddr of the current expression, performing (2) would have to make fie be eq to fum if all subsequent operations were to update both fie and fum correctly. Think about it.

In summary, the only situation in which an edit operation will *not* change an external pointer occurs when the external pointer is to a *proper tail* of the data structure, i.e., to cdr of some node in the structure, and the operation is deletion. If all external pointers are to *elements* of the structure, i.e., to car of some node, or if only insertions, replacements, or attachments are performed, the edit operation will *always* have the same effect on an external pointer as it does on the current expression.

9.4.2 THE A, B, AND : COMMANDS

In the (n) , $(n \ e_1 \dots \ e_m)$, and $(-n \ e_1 \dots \ e_m)$ commands, the sign of the integer is used to indicate the operation. As a result, there is no direct way to express insertion after a particular element, (hence the necessity for a separate N command). Similarly, the user cannot specify deletion or replacement of the nth element from the end of a list without first converting n to the corresponding positive integer. Accordingly, we have:

(B $e_1 \dots \ e_m$) inserts $e_1 \dots \ e_m$ before the current expression. Equivalent to **UP** followed by $(-1 \ e_1 \dots \ e_m)$.

For example, to insert FOO before the last element in the current expression, perform -1 and then **(B FOO)**.

(A $e_1 \dots \ e_m$) inserts $e_1 \dots \ e_m$ after the current expression. Equivalent to **UP** followed by $(-2 \ e_1 \dots \ e_m)$ or $(N \ e_1 \dots \ e_m)$ whichever is appropriate.

(: $e_1 \dots \ e_m$) replaces the current expression by $e_1 \dots \ e_m$. Equivalent to **UP** followed by $(1 \ e_1 \dots \ e_m)$.

DELETE or **(:)** deletes the current expression.

DELETE first tries to delete the current expression by performing an **UP** and then a **(1)**. This works in most cases. However, if after performing **UP**, the new current expression contains only one element, the command **(1)** will not work. Therefore, **DELETE** starts over and performs a **BK**, followed by **UP**, followed by **(2)**. For example, if the current expression is **(COND ((MEMB X Y)) (T Y))**, and the user performs -1, and then **DELETE**, the **BK-UP-(2)** method is used, and the new current expression will be ... **((MEMB X Y))**

However, if the next higher expression contains only one element, **BK** will not work. So in this case, **DELETE** performs **UP**, followed by **(: NIL)**, i.e., it *replaces* the higher expression by **NIL**. For example, if the current expression is **(COND ((MEMB X Y)) (T Y))** and the user performs **F MEMB** and then **DELETE**, the new current expression will be ... **NIL (T Y)** and the original expression would now be **(COND NIL (T Y))**. The rationale behind this is that deleting **(MEMB X Y)** from **((MEMB X Y))** changes a list of one element to a list of no elements, i.e., **()** or **NIL**.

If the current expression is a tail, then **B**, **A**, **:**, and **DELETE** all work exactly the same as though the current expression were the first element in that tail. Thus if the current expression were

... (PRINT Y) (PRINT Z)), (B (PRINT X)) would insert (PRINT X) before (PRINT Y), leaving the current expression ... (PRINT X) (PRINT Y) (PRINT Z)).

The following forms of the A, B, and : commands incorporate a location specification:

(INSERT e₁ ... e_m BEFORE . @)⁴² Similar to (LC .@)⁴³ followed by (B e₁ ... e_m).

```
*P
(PROG (& & X) **COMMENT** (SELECTQ ATM & NIL) (OR & &) (PRIN1 & T)
(PRIN1 & T) (SETQ X &
*(INSERT LABEL BEFORE PRIN1)
*P
(PROG (& & X) **COMMENT** (SELECTQ ATM & NIL) (OR & &) LABEL
(PRIN1 & T) (
*
```

44

Current edit chain is not changed, but unfind is set to the edit chain after the B was performed, i.e., \ will make the edit chain be that chain where the insertion was performed.

(INSERT e₁ ... e_m AFTER . @)

Similar to INSERT BEFORE except uses A instead of B.

(INSERT e₁ ... e_m FOR . @)

similar to INSERT BEFORE except uses : for B.

(REPLACE @ WITH e₁ ... e_m)⁴⁵

Here @⁴⁶ is the *segment* of the command between REPLACE and WITH. Same as (INSERT e₁ ... e_m FOR . @).

Example: (REPLACE COND -1 WITH (T (RETURN L)))

(CHANGE @ TO e₁ ... e_m) Same as REPLACE WITH.

⁴² i.e., @ is cdr[member[BEFORE;command]]

⁴³ except that if @ causes an error, the location process does *not* continue as described on page 9.19. For example if @=(COND 3) and the next COND does not have a 3rd element, the search stops and the INSERT fails. Note that the user can always write (LC COND 3) if he intends the search to continue.

⁴⁴ Sudden termination of output followed by a blank line return indicates printing was aborted by control-E.

⁴⁵ BY can be used for WITH.

⁴⁶ See footnote on page 9.27.

(DELETE . @) does a (LC . @)⁴⁷ followed by DELETE. Current edit chain is not changed,⁴⁸ but unfind is set to the edit chain after the DELETE was performed.

Example: (DELETE -1), (DELETE COND 3)

Note: if @ is NIL (i.e., empty), the corresponding operation is performed here (on the current edit chain).

For example, (REPLACE WITH (CAR X)) is equivalent to (: (CAR X)). For added readability, HERE is also permitted, e.g., (INSERT (PRINT X) BEFORE HERE) will insert (PRINT X). before the current expression (but not change the edit chain).

Note: @ does not have to specify a location within the current expression, i.e., it is perfectly legal to ascend to INSERT, REPLACE, or DELETE

For example, (INSERT (RETURN) AFTER ↑ PROG -1) will go to the top, find the first PROG, and insert a (RETURN) at its end, and not change the current edit chain.

The A, B, and : commands, commands, (and consequently INSERT, REPLACE, and CHANGE), all make special checks in e_1 thru e_m for expressions of the form (# # . coms). In this case, the expression used for inserting or replacing is a *copy* of the current expression after executing coms, a list of edit commands.⁴⁹ For example, (INSERT (## F COND -1 -1) AFTER 3)⁵⁰ will make a copy of the last form in the last clause of the next cond, and insert it after the third element of the current expression.

9.4.3 FORM ORIENTED EDITING AND THE ROLE OF UP

The UP that is performed before A, B, and : commands⁵¹ makes these operations form-oriented. For example, if the user types F SETQ, and then DELETE, or simply (DELETE SETQ), he will delete the entire SETQ expression, whereas (DELETE X) if X is a variable, deletes just the variable X. In both cases, the operation is performed on the corresponding *form*, and in both cases is probably what the user intended. Similarly, if the user types (INSERT (RETURN Y) BEFORE

47 See footnote on page 9.27.

48 Unless the current expression is no longer a part of the expression being edited, e.g., if the current expression is ... C) and the user performs (DELETE 1), the tail, (C), will have been cut off. Similarly, if the current expression is (CDR Y) and the user performs (REPLACE WITH (CAR X)).

49 The execution of coms does not change the current edit chain.

50 Not (INSERT F COND -1 (## -1) AFTER 3), which inserts four elements after the third element, namely F, COND, -1, and a copy of the last element in the current expression.

51 and therefore in INSERT, CHANGE, REPLACE, and DELETE commands after the location portion of the operation has been performed.

SETQ), he means before the SETQ expression, not before the atom SETQ.⁵² A consequent of this procedure is that a pattern of the form (SETQ Y --) can be viewed as simply an elaboration and further refinement of the pattern SETQ. Thus (INSERT (RETURN Y) BEFORE SETQ) and (INSERT (RETURN Y) BEFORE (SETQ Y --)) perform the same operation⁵³ and, in fact, this is one of the motivations behind making the current expression after F SETQ, and F (SETQ Y --) be the same.

Occasionally, however, a user may have a data structure in which no special significance or meaning is attached to the position of an atom in a list, as Interlisp attaches to atoms that appear as car of a list, versus those appearing elsewhere in a list. In general, the user may not even *know* whether a particular atom is at the head of a list or not. Thus, when he writes (INSERT expression BEFORE FOO), he means before the atom FOO, whether or not it is car of a list. By setting the variable upfindflg to NIL,⁵⁴ the user can suppress the implicit UP that follows searches for atoms, and thus achieve the desired effect. With upfindflg=NIL, following F FOO, for example, the current expression will be the atom FOO. In this case, the A, B, and : operations will operate with respect to the atom FOO. If the user intends the operation to refer to the list which FOO heads, he simply uses instead the pattern (FOO --).

9.4.4 EXTRACT AND EMBED

Extraction involves replacing the current expression with one of its subexpressions (from any depth).

(XTR . @) replaces the original current expression with the expression that is current after performing (LCL . @).⁵⁵

For example, if the current expression is (COND ((NULL X) (PRINT Y))), (XTR PRINT), or (XTR 2 2) will replace the cond by the print.

If the current expression after (LCL . @) is a *tail* of a higher expression, its first element is used.

For example, if the current expression is (COND ((NULL X) Y) (T Z)), then (XTR Y) will replace the cond with Y, even though the current expression after performing (LCL Y) is ... Y).

If the extracted expression is a list, then after XTR has finished, the current expression will be that list.

Thus, in the first example, the current expression after the XTR would be (PRINT Y).

⁵² There is some ambiguity in (INSERT expr AFTER functionname), as the user might mean make expr be the function's first argument. Similarly, the user cannot write (REPLACE SETQ WITH SETQQ) meaning change the name of the function. The user must in these cases write (INSERT expr AFTER functionname 1), and (REPLACE SETQ 1 WITH SETQQ).

⁵³ assuming the next SETQ is of the form (SETQ Y --).

⁵⁴ Initially, and usually, set to T.

⁵⁵ See footnote on page 9.27.

If the extracted expression is not a list, the new current expression will be a tail whose first element is that non-list.

Thus, in the second example, the current expression after the XTR would be ... Y followed by whatever followed the COND.

If the current expression *initially* is a tail, extraction works exactly the same as though the current expression were the first element in that tail. Thus if the current expression is ... (COND ((NULL X) (PRINT Y))) (RETURN Z)), then (XTR PRINT) will replace the cond by the print, leaving (PRINT Y) as the current expression.

The extract command can also incorporate a location specification:

(EXTRACT @₁ FROM . @₂)⁵⁶ Performs (LC . @₂)⁵⁷ and then (XTR . @₁). Current edit chain is not changed, but unfind is set to the edit chain after the XTR was performed.

Example: If the current expression is (PRINT (COND ((NULL X) Y) (T Z))) then following (EXTRACT Y FROM COND), the current expression will be (PRINT Y). (EXTRACT 2 -1 FROM COND), (EXTRACT Y FROM 2), (EXTRACT 2 -1 FROM 2) will all produce the same result.

While extracting replaces the current expression by a subexpression, embedding replaces the current expression with one containing *it* as a subexpression.

(MBD e₁ ... e_m)

MBD substitutes⁵⁸ the current expression for all instances of the atom * in e₁ ... e_m, and replaces the current expression with the result of that substitution.

Examples: If the current expression is (PRINT Y),

(MBD (COND ((NULL X) *) ((NULL (CAR Y)) * (GO LP)))) would replace (PRINT Y) with (COND ((NULL X) (PRINT Y)) ((NULL (CAR Y)) (PRINT Y) (GO LP))).

If the current expression is (RETURN X), (MBD (PRINT Y) (AND FLG *)) would replace it with the *two* expressions (PRINT Y) and (AND FLG (RETURN X)) i.e., if the (RETURN X) appeared in the cond clause (T (RETURN X)), after the MBD, the clause would be (T (PRINT Y) (AND FLG (RETURN X))).

If * does not appear in e₁ ... e_m, the MBD is interpreted as (MBD (e₁ ... e_m *)).

Examples: If the current expression is (PRINT Y), then (MBD SETQ X) will replace it with (SETQ X (PRINT Y)). If the current expression is (PRINT Y), (MBD RETURN) will replace it with (RETURN (PRINT Y)).

⁵⁶ @₁ is the *segment* between EXTRACT and FROM.

⁵⁷ See footnote on page 9.27.

⁵⁸ as with subst, a fresh copy is used for each substitution.

MBD leaves the edit chain so that the larger expression is the new current expression.

If the current expression *initially* is a tail, embedding works exactly the same as though the current expression were the first element in that tail. Thus if the current expression were ... (PRINT Y) (PRINT Z)), (MBD SETQ X) would replace (PRINT Y) with (SETQ X (PRINT Y)).

The embed command can also incorporate a location specification:

(EMBED @ IN . x)⁵⁹ does (LC . @)⁶⁰ and then (MBD . x). Edit chain is not changed, but unfind is set to the edit chain after the MBD was performed.

Example: (EMBED PRINT IN SETQ X), (EMBED 3 2 IN RETURN),
(EMBED COND 3 1 IN (OR * (NULL X))).

WITH can be used for IN, and SURROUND can be used for EMBED, e.g., (SURROUND NUMBERP WITH (AND * (MINUSP X))).

9.4.5 THE MOVE COMMAND

The MOVE command allows the user to specify (1) the expression to be moved, (2) the place it is to be moved to, and (3) the operation to be performed there, e.g., insert it before, insert it after, replace, etc.

(MOVE @₁ TO com . @₂)⁶¹

where com is BEFORE, AFTER, or the name of a list command, e.g., :, N, etc. performs (LC . @₁)⁶² and obtains the current expression there (or its first element, if it is a tail), which we will call expr; MOVE then goes back to the original edit chain, performs (LC . @₂) followed by (com expr)⁶³ then goes back to @₁ and deletes expr. Edit chain is not changed. Unfind is set to edit chain after (com expr) was performed.

For example, if the current expression is (A B C D), (MOVE 2 TO AFTER 4) will make the new current expression be (A C D B). Note that 4 was executed as of the original edit chain, and that the second element had not yet been removed.⁶⁴

⁵⁹ @ is the segment between EMBED and IN.

⁶⁰ See footnote on page 9.27.

⁶¹ @₁ is the segment between MOVE and TO.

⁶² see footnote on page 9.27.

⁶³ Setting an internal flag so expr is not copied.

⁶⁴ If @₂ specifies a location *inside of the expression to be moved*, a message is printed and an error is generated, e.g., (MOVE 2 TO AFTER X), where X is contained inside of the second element.

As the following examples taken from actual editing will show, the MOVE command is an extremely versatile and powerful feature of the editor.

```
*?
(PROG ((L L)) (EDLOC (CDDR C)) (RETURN (CAR L)))
*(MOVE 3 TO : CAR)
*?
(PROG ((L L)) (RETURN (EDLOC (CDDR C))))
*

*P
... (SELECTQ OBJPR & &) (RETURN &) LP2 (COND & &))
*(MOVE 2 TO N 1)
*P
... (SELECTQ OBJPR & & &) LP2 (COND & &))
*

*P
(OR (EQ X LASTAIL) (NOT &) (AND & & &))
*(MOVE 4 TO AFTER (BELOW COND))
*P
(OR (EQ X LASTAIL) (NOT &))
*\ P
... (& &) (AND & & &) (T & &))
*

*P
((NULL X) **COMMENT** (COND & &))
*(-3 (GO NXT])
*(MOVE 4 TO N (< PROG))
*P
((NULL X) **COMMENT** (GO NXT))
*\ P
(PROG (&) **COMMENT** (COND & & &) (COND & & &) (COND & &))
*(INSERT NXT BEFORE -1)
*P
(PROG (&) **COMMENT** (COND & & &) (COND & & &) NXT (COND & &))
```

Note that in the last example, the user could have added the prog label NXT and moved the cond in one operation by performing (MOVE 4 TO N (< PROG) (N NXT)). Similarly, in the next example, in the course of specifying @₂, the location where the expression was to be moved to, the user also performs a structure modification, via (N (T)), thus creating the structure that will receive the expression being moved.

```
*P
((CDR &) **COMMENT** (SETQ CL &) (EDITSMASH CL & &))
*MOVE 4 TO N 0 (N (T)) -1]
*P
((CDR &) **COMMENT** (SETQ CL &))
*\ P
*(T (EDITSMASH CL & &))
*
```

If @₂ is NIL, or (HERE), the current position specifies where the operation is to take place. In this case, unfind is set to where the expression that was moved was originally located, i.e., @₁. For example:

```

*P
(TENEX)
*(MOVE ↑ F APPLY TO N HERE)
*P
(TENEX (APPLY & &))
*

*P
(PROG (& & & ATM IND VAL) (OR & &)    **COMMENT**    (OR & &)
(PRIN1 & T) (
PRIN1 & T) (SETQ IND
65
*(MOVE * TO BEFORE HERE)
*P
(PROG (& & & ATM IND VAL) (OR & &) (OR & &) (PRIN1 &

*P
(T (PRIN1 C-EXP T))
*(MOVE ↑ BF PRIN1 TO N HERE)
*P
(T (PRIN1 C-EXP T) (PRIN1 & T))
*

```

Finally, if @_1 is NIL, the MOVE command allows the user to specify where the *current expression* is to be moved to. In this case, the edit chain is changed, and is the chain where the current expression was moved to; unfind is set to where it was.

```

*P
(SELECTQ OBJPR (&) (PROGN & &))
*(MOVE TO BEFORE LOOP)
*P
... (SELECTQ OBJPR & &) LOOP (FRPLACA DFPRP &) (FRPLACD DFPRP
&) (SELECTQ
*
```

9.4.6 COMMANDS THAT MOVE PARENTHESES

The commands presented in this section permit modification of the list structure itself, as opposed to modifying components thereof. Their effect can be described as inserting or removing a single left or right parenthesis, or pair of left and right parentheses. Of course, there will always be the same number of left parentheses as right parentheses in any list structure, since the parentheses are just a notational guide to the structure provided by print. Thus, no command can insert or remove just one parenthesis, but this is suggestive of what actually happens.

In all six commands, n and m are used to specify an element of a list, usually of the current expression. In practice, n and m are usually positive or negative integers with the obvious interpretation. However, all six commands use the generalized NTH command, page 9.21, to find their element(s), so that nth element means the first element of the tail found by performing (NTH n). In other words, if the current expression is (LIST (CAR X) (SETQ Y (CONS W Z))), then (BI 2 CONS), (BI X -1), and (BI X Z) all specify the exact same operation.

⁶⁵ Sudden termination of output followed by a blank line indicates printing was aborted by control-E.

All six commands generate an error if the element is not found, i.e., the NTH fails. All are undoable.

(BI n m) both in, inserts a left parentheses before the nth element and after the mth element in the current expression. Generates an error if the mth element is not contained in the nth tail, i.e., the mth element must be "to the right" of the nth element.

Example: If the current expression is (A B (C D E) F G), then (BI 2 4) will modify it to be (A (B (C D E) F) G).

(BI n) same as (BI n n).

Example: If the current expression is (A B (C D E) F G), then (BI -2) will modify it to be (A B (C D E) (F) G).

(BO n) both out. Removes both parentheses from the nth element. Generates an error if nth element is not a list.

Example: If the current expression is (A B (C D E) F G), then (BO D) will modify it to be (A B C D E F G).

(LI n) left in, inserts a left parenthesis before the nth element (and a matching right parenthesis at the end of the current expression), i.e. equivalent to (BI n -1).

Example: if the current expression is (A B (C D E) F G), then (LI 2) will modify it to be (A (B (C D E) F G)).

(LO n) left out, removes a left parenthesis from the nth element. *All elements following the nth element are deleted.* Generates an error if nth element is not a list.

Example: If the current expression is (A B (C D E) F G), then (LO 3) will modify it to be (A B C D E).

(RI n m) right in, inserts a right parenthesis after the mth element of the nth element. The rest of the nth element is brought up to the level of the current expression.

Example: If the current expression is (A (B C D E) F G), (RI 2 2) will modify it to be (A (B C) D E F G). Another way of thinking about RI is to read it as "move the right parenthesis at the end of the nth element in to after its mth element."

(RO n) right out, removes the right parenthesis from the nth element, moving it to the end of the current expression. All elements

following the nth element are moved inside of the nth element. Generates an error if nth element is not a list.

Example: If the current expression is (A B (C D E) F G), (R0 3) will modify it to be (A B (C D E F G)). Another way of thinking about R0 is to read it as "move the right parenthesis at the end of the nth element *out* to the end of the current expression."

9.4.7 TO AND THRU

EXTRACT, EMBED, DELETE, REPLACE, and MOVE can be made to operate on several contiguous elements, i.e., a segment of a list, by using in their respective location specifications the TO or THRU command.

(@1 THRU @2)

does a (LC . @1), followed by an UP, and then a (BI 1 @2), thereby grouping the segment into a single element, and finally does a 1, making the final current expression be that element.

For example, if the current expression is (A (B (C D) (E) (F G H) I) J K), following (C THRU G), the current expression will be ((C D) (E) (F G H)).

(@1 TO @2)

Same as THRU except last element not included, i.e., after the BI, an (RI 1 -2) is performed.

If both @1 and @2 are numbers, and @2 is greater than @1, then @2 counts from the beginning of the current expression, the same as @1. In other words, if the current expression is (A B C D E F G), (3 THRU 5) means (C THRU E) not (C THRU G). In this case, the corresponding BI command is (BI 1 @2-@1+1).

THRU and TO are not very useful commands by themselves; they are intended to be used in conjunction with EXTRACT, EMBED, DELETE, REPLACE, and MOVE. After THRU and TO have operated, they set an internal editor flag informing the above commands that the element they are operating on is actually a segment, and that the extra pair of parentheses should be removed when the operation is complete. Thus:

```

*P
(PROG (& & ATM IND VAL WORD) (PRIN1 & T) (PRIN1 & T) (SETQ IND &)
(SETQ VAL &) **COMMENT** (SETQQ

*(MOVE (3 THRU 4) TO BEFORE 7)
*P
(PROG (& & ATM IND VAL WORD) (SETQ IND &) (SETQ VAL &) (PRIN1 & T)
(PRIN1 & T) **COMMENT**)

*
*P
(* FAIL RETURN FROM EDITOR. USER SHOULD NOTE THE VALUES OF SOURCEXPR
AND CURRENTFORM. CURRENTFORM IS THE LAST FORM IN SOURCEXPR WHICH
WILL HAVE BEEN TRANSLATED, AND IT CAUSED THE ERROR.)
*(DELETE (USER THRU CURR$))
=CURRENTFORM.
*P
(* FAIL RETURN FROM EDITOR. CURRENTFORM IS

*
*P
... LP (SELECTO & & & & NIL) (SETQ Y &) OUT (SETQ FLG &) (RETURN Y))
*(MOVE (1 TO OUT) TO N HERE]
*P
... OUT (SETQ FLG &) (RETURN Y) LP (SELECTQ & & & & NIL) (SETQ Y &))
*

*PP
[PROG (RF TEMP1 TEMP2)
(COND
  ((NOT (MEMB REMARG LISTING))
    (SETQ TEMP1 (ASSOC REMARG NAMEDREMARKS)) **COMMENT**
    (SETQ TEMP2 (CADR TEMP1))
    (GO SKIP))
  (T **COMMENT**
    (SETQ TEMP1 REMARG)))
(NCONC1 LISTING REMARG)
(COND
  ((NOT (SETQ TEMP2 (SASSOC
    *(EXTRACT (SETQ THRU CADR) FROM COND)
*P
(PROG (RF TEMP1 TEMP2) (SETQ TEMP1 &) **COMMENT** (SETQ TEMP2 &)
(NCONC1 LISTING REMARG) (COND & &

*)

```

TO and THRU can also be used directly with XTR.⁶⁶ Thus in the previous example, if the current expression had been the COND, e.g., the user had first performed F COND, he could have used (XTR (SETQ THRU CADR)) to perform the extraction.

(@₁ TO), (@₁ THRU)

both same as (@₁ THRU -1), i.e., from @₁ through the end of the list.

Examples:

```
*P  
(VALUE (RPLACA DEPRP &) (RPLACD &) (RPLACA VARSWORD &) (RETURN))  
*(MOVE (2 TO) TO N (< PROG))  
*(N (GO VAR))  
*P  
(VALUE (GO VAR))  
  
*P  
(T **COMMENT** (COND &) **COMMENT** (EDITSMASH CL & &) (COND &))  
*(-3 (GO REPLACE))  
*(MOVE (COND TO) TO N ↑ PROG (N REPLACE))  
*P  
(T **COMMENT** (GO REPLACE))  
*\ P  
(PROG (&) **COMMENT** (COND & & &) (COND & & &) DELETE (COND & &) DELETE (COND & &) (COND &))  
REPLACE (COND &) **COMMENT** (EDITSMASH CL & &) (COND &))  
*
```

⁶⁶ Because XTR involves a location specification while A, B, :, and MBD do not.

```

*PP
[LAMBDA (CLAUSALA X)
(PROG (A D)
      (SETQ A CLAUSALA)
      LP (COND
           ((NULL A)
            (RETURN)))
           (SERCH X A)
           (RUMARK (CDR A))
           (NOTICECL (CAR A))
           (SETQ A (CDR A))
           (GO LP])
*(EXTRACT (SERCH THRU NOT$) FROM PROG)
=NOTICECL
*P
(LAMBDA (CLAUSALA X) (SERCH X A) (RUMARK &) (NOTICECL &))
*(EMBED (SERCH TO) IN (MAP CLAUSALA (FUNCTION (LAMBDA (A) *]
*PP
[LAMBDA (CLAUSALA X)
(MAP CLAUSALA (FUNCTION (LAMBDA (A)
      (SERCH X A)
      (RUMARK (CDR A))
      (NOTICECL (CAR A)
*

```

9.4.8 THE R COMMAND

(R x y) replaces all instances of x by y in the current expression, e.g., (R CAADR CADAR). Generates an error if there is not at least one instance.

The R command operates in conjunction with the search mechanism of the editor. The search proceeds as described on page 9.16-17, and x can employ any of the patterns on page 9.14-16. Each time x matches an element of the structure, the element is replaced by (a copy of) y; each time x matches a tail of the structure, the tail is replaced by (a copy of) y.

For example, if the current expression is (A (B C) (B . C)),
(R C D) will change it to (A (B D) (B . D)),
(R .(... . C) D) to (A (B C) (B . D)),
(R C (D E)) to (A (B (D E)) (B D E)), and
(R (... . NIL) D) to (A (B C . D) (B . C) . D).

If x is an atom or string containing <esc>s, <esc>s appearing in y stand for the characters matched by the corresponding <esc> in x. For example, (R FOO\$ FIE\$) means for all atoms or strings that begin with FOO, replace the characters "FOO" by "FIE".⁶⁷ Applied to the list (FOO FOO2 XFO01), (R FOO\$ FIE\$) would produce (FIE FIE2 XFO01), and

⁶⁷ If x matches a string, it will be replaced by a string. Note that it does not matter whether x or y themselves are strings, i.e. (R \$D\$ \$A\$), (R "\$D\$" "\$A\$"), (R \$D\$ "\$A\$"), and (R "\$D\$" "\$A\$") are equivalent. Note also that x will never match with a number, i.e., (R \$1 \$2) will not change 11 to 12.

(R \$FOO\$ \$FIE\$) would produce (FIE FIE2 XFIE1). Similarly, (R \$D\$ \$A\$) will change (LIST (CADR X) (CADDR Y)) to (LIST (CAAR X) (CAADDR)).⁶⁸

The user will be informed of all such <esc> replacements by a message of the form $x \rightarrow y$, e.g., CADR->CAAR.

Note that the \$ feature can be used to delete or add characters, as well as replace them. For example, (R \$1 \$) will delete the terminating 1's from all literal atoms and strings. Similarly, if an <esc> in x does not have a mate in y , the characters matched by the \$ are effectively deleted. For example, (R \$/\$ \$) will change AND/OR to AND.⁶⁹ y can also be a list containing <esc>s, e.g., (R \$1 (CAR \$)) will change FOO1 to (CAR FOO), FIE1 to (CAR FIE).

If x does not contain <esc>s, \$ appearing in y refers to the *entire* expression matched by x , e.g., (R LONGATOM '\$) changes LONGATOM to 'LONGATOM, (R (SETQ X &) (PRINT \$)) changes every (SETQ X &) to (PRINT (SETQ X &)).⁷⁰

Since (R \$x\$ \$y\$) is a frequently used operation for replacing characters, the following command is provided:

(RC x y) equivalent to (R \$x\$ \$y\$)

R and RC change all instances of x to y . The commands R1 and RC1 are available for changing just one, (i.e., the first) instance of x to y .

(R1 x y) find the first instance of x and replace it by y .

(RC1 x y) (R1 \$x\$ \$y\$).

In addition, while R and RC only operate within the current expression, R1 and RC1 will continue searching, a la the F command, until they find an instance of x , even if the search carries them beyond the current expression.

(SW n m) switches the nth and mth elements of the current expression.

For example, if the current expression is

(LIST (CONS (CAR X) (CAR Y)) (CONS (CDR X) (CDR Y))),

⁶⁸ Note that CADDR was *not* changed to CAAAR, i.e., (R \$D\$ \$A\$) does not mean replace every D with A, but replace the first D in every atom or string by A. If the user wanted to replace every D by A, he could perform (LP (R \$D\$ \$A\$)).

⁶⁹ However, there is no similar operation for changing AND/OR to OR, since the first \$ in y always corresponds to the first \$ in x , the second \$ in y to the second in x , etc.

⁷⁰ If x is a pattern containing an <esc> pattern somewhere *within* it, the characters matched by the <esc>s are not available, and for the purposes of replacement, the effect is the same as though x did not contain any <esc>s. For example, if the user types (R (CAR F\$) (PRINT \$)), the second \$ will refer to the entire expression matched by (CAR F\$).

(SW 2 3) will modify it to be

(LIST (CONS (CDR X) (CDR Y)) (CONS (CAR X) (CAR Y))). The relative order of n and m is not important, i.e., (SW 3 2) and (SW 2 3) are equivalent.

SW uses the generalized NTH command to find the nth and mh elements, a la the BI-BO commands.

Thus in the previous example, (SW CAR CDR) would produce the same result.

- + (SWAP @1 @2) like SW except switches the expressions specified by @1 and @2, not the corresponding elements of the current expression, i.e. @1 and @2 can be at different levels in current expression, or one or both be outside of current expression.
- + Thus, using the previous example, (SWAP CAR CDR) would result in
- + (LIST (CONS (CAR X) (CDR Y)) (CONS (CDR X) (CAR Y))).

9.5 COMMANDS THAT PRINT

PP prettyprints the current expression.

P prints the current expression as though printlevel were set to 2.

(P m) prints mh element of current expression as though printlevel were set to 2.

(P 0) same as P

(P m n) prints mh element of current expression as though printlevel were set to n.

(P 0 n) prints current expression as though printlevel were set to n.

? same as (P 0 100)

Both (P m) and (P m n) use the generalized NTH command to obtain the corresponding element, so that m does not have to be a number, e.g., (P COND 3) will work. PP causes all comments to be printed as **COMMENT** (see Section 14). P and ? print as **COMMENT** only those comments that are (top level) elements of the current expression.⁷¹

⁷¹ Lower expressions are not really seen by the editor; the printing command simply sets printlevel and calls print.

PP*prettyprints current expression, *including* comments.**PP*** is equivalent to **PP** except that it first resets ****comment**flg** to **NIL** (see Section 14).**PPV**

prettyprints current expression as a variable, i.e., no special treatment for LAMBDA, COND, SETQ, etc., or for CLISP.

PPT

prettyprints current expression, printing CLISP translations, if any.

?=prints the argument names and corresponding values for current expression. Analogous to **?=** break command (Section 15).For example, if the current expression is **(STRPOS "A0????" X N (QUOTE ?) T)**, **?=** prints

```
X = "A0????"
Y = X
START = N
SKIP = (QUOTE ?)
ANCHOR = T
TAIL = 72
```

All printing functions print to the terminal, regardless of the primary output file. All use the readable **T**. No printing function ever changes the edit chain. All record the current edit chain for use by **\P**, page 9.23. All can be aborted with control-E.

9.6 COMMANDS THAT EVALUATE

E

*only when typed in,*⁷³ causes the editor to call **lisp_x** giving it the next input as argument.⁷⁴

Example:

```
*E BREAK(FIE FUM)
(FIE FUM)
*E (FOO)

(FIE BROKEN)
```

⁷²

The command **MAKE** described on page 9.54 is an imperative form of **?=**. It allows the user to specify a change to the element of the current expression that corresponds to a particular argument name.

+

⁷³

e.g. **(INSERT D BEFORE E)** will treat **E** as a pattern, and search for **E**.

⁷⁴

lisp_x is used by **evalqt** and **break** for processing terminal inputs. If nothing else is typed on the same line, **lisp_x** evaluates its argument. Otherwise, **lisp_x** applies it to the next input. In both cases, **lisp_x** prints the result. See above example, and Sections 2 and 22.

(E x) evaluates x, i.e., performs eval[x], and prints the result on the terminal.

(E x T) same as (E x) but does not print.

The (E x) and (E x T) commands are mainly intended for use by macros and subroutine calls to the editor; the user would probably type in a form for evaluation using the more convenient format of the (atomic) E command.

(I c x₁ ... x_n) same as (C y₁ ... y_n) where y₁=eval[x₁].

Example: (I 3 (GETD (QUOTE FOO))) will replace the 3rd element of the current expression with the definition of foo.⁷⁵ (I N FOO (CAR FIE)) will attach the value of foo and car of the value of fie to the end of the current expression. (I F= FOO T) will search for an expression eq to the value of foo.

If c is not an atom, c is evaluated also.

Example: (I (COND ((NULL FLG) (QUOTE -1)) (T 1)) FOO), if flg is NIL, inserts the value of foo before the first element of the current expression, otherwise replaces the first element by the value of foo.

- + EVAL does an eval of the current expression.
- + Note that EVAL, <line-feed>, and the GO command together effectively allow the user to "single-step" a program through its symbolic definition.
- + GETVAL replaces the current expression by the result of evaluating it.

#[com₁;com₂; ... ;com_n] is an NLAMBDA, NOSPREAD function (not a command). Its value is what the current expression would be after executing the edit commands com₁ ... com_n starting from the present edit chain. Generates an error if any of com₁ thru com_n cause errors. The current edit chain is never changed.⁷⁶

⁷⁵ The I command sets an internal flag to indicate to the structure modification commands *not* to copy expression(s) when inserting, replacing, or attaching.

⁷⁶ Recall that A, B, :, INSERT, REPLACE, and CHANGE make special checks for # # forms in the expressions used for inserting or replacing, and use a copy of # # form instead (see page 9.28). Thus, (INSERT (## 3 2) AFTER 1) is equivalent to (I INSERT (COPY (## 3 2)) (QUOTE AFTER) 1).

Example: (I R (QUOTE X) (## (CONS .. Z))) replaces all X's in the current expression by the first cons containing a Z.

The I command is not very convenient for computing an *entire* edit command for execution, since it computes the command name and its arguments separately. Also, the I command cannot be used to compute an atomic command. The following two commands provide more general ways of computing commands.

(COMS $x_1 \dots x_n$) Each x_i is evaluated and its value is executed as a command.

For example, (COMS (COND (X (LIST 1 X)))) will replace the first element of the current expression with the value of x if non-NIL, otherwise do nothing.⁷⁷

(COMSQ com₁ ... com_n) executes com₁ ... com_n.

COMSQ is mainly useful in conjunction with the COMS command. For example, suppose the user wishes to compute an entire list of commands for evaluation, as opposed to computing each command one at a time as does the COMS command. He would then write (COMS (CONS (QUOTE COMSQ) x)) where x computed the list of commands, e.g., (COMS (CONS (QUOTE COMSQ) (GETP FOO (QUOTE COMMANDS)))).

9.7 COMMANDS THAT TEST

(IF x) generates an error *unless* the value of eval[x] is true, i.e., if eval[x] causes an error or eval[x]=NIL, IF will cause an error.

For some editor commands, the occurrence of an error has a well defined meaning, i.e., they use errors to branch on, as cond uses NIL and non-NIL. For example, an error condition in a location specification may simply mean "not this one, try the next." Thus the location specification (IPLUS (E (OR (NUMBERP (## 3)) (ERROR!)) T)) specifies the first IPLUS whose second argument is a number. The IF command, by equating NIL to error, provides a more natural way of accomplishing the same result. Thus, an equivalent location specification is (IPLUS (IF (NUMBERP (## 3)))).

The IF command can also be used to select between two alternate lists of commands for execution.

(IF x coms₁ coms₂) If eval[x] is true, execute coms₁; if eval[x] causes an error or is equal to NIL, execute coms₂.⁷⁸

For example, the command (IF (READP T) NIL (P)) will print the current expression provided the input buffer is empty.

⁷⁷ because NIL as a command is a no-op, see page 9.52.

⁷⁸ Thus IF is equivalent to (COMS (CONS (QUOTE COMSQ) (COND ((CAR (NLSETQ (EVAL X))) COMS1) (T COMS2)))).

(IF x coms₁) if eval[x] is true, execute coms₁; otherwise generate an error.

(LP . coms) repeatedly executes coms, a list of commands, until an error occurs.

For example, (LP F PRINT (N T)) will attach a T at the end of every print expression. (LP F PRINT (IF (## 3) NIL ((N T)))) will attach a T at the end of each print expression which does not already have a second argument.⁷⁹

When an error occurs, LP prints n OCCURRENCES, where n is the number of times coms was successfully executed. The edit chain is left as of the last complete successful execution of coms.

(LPQ . coms) same as LP but does not print the message n OCCURRENCES.

In order to prevent non-terminating loops, both LP and LPQ terminate when the number of iterations reaches maxloop, initially set to 30.⁸⁰ Since the edit chain is left as of the last successful completion of the loop, the user can simply continue the LP command with REDO (Section 22).

(SHOW . x) x is a list of patterns. SHOW does a LPQ printing all instances of the indicated expression(s), e.g. (SHOW FOO (SETQ FIE &)) will print all FOO's and all (SETQ FIE &)'s. Generates an error if there aren't any instances of the expression(s).

(EXAM . x) like SHOW except calls the editor recursively (via the TTY: command described on page 9.49) on each instance of the indicated expression(s) so that the user can examine and/or change them.

(ORR coms₁ ... coms_n) ORR begins by executing coms₁, a list of commands. If no error occurs, ORR is finished. Otherwise, ORR restores the edit chain to its original value, and continues by executing coms₂, etc. If none of the command lists execute without errors, i.e., the ORR "drops off the end", ORR generates an error. Otherwise, the edit chain is left as of the completion of the first command list which executes without an error.⁸¹

⁷⁹ i.e., the form (## 3) will cause an error if the edit command 3 causes an error, thereby selecting ((N T)) as the list of commands to be executed. The IF could also be written as (IF (CDDR (##)) NIL ((N T))).

⁸⁰ maxloop can also be set to NIL, which is equivalent to setting it to infinity.

⁸¹ NIL as a command list is perfectly legal, and will always execute successfully. Thus, making the last "argument" to ORR be NIL will insure that the ORR never causes an error. Any other atom is treated as (atom), i.e., the above example could be written as (ORR NX INX NIL).

For example, (ORR (NX) (!NX) NIL) will perform a NX, if possible, otherwise a !NX, if possible, otherwise do nothing. Similarly, DELETE could be written as (ORR (UP (1)) (BK UP (2)) (UP (: NIL))).

9.8 MACROS

Many of the more sophisticated branching commands in the editor, such as ORR, IF, etc., are most often used in conjunction with edit macros. The macro feature permits the user to define new commands and thereby expand the editor's repertoire, or redefine existing commands.⁸²

*

Macros are defined by using the M command.

(M c . coms) For c an atom, M defines c as an atomic command.⁸³ Executing c is then the same as executing the list of commands coms.

For example, (M BP BK UP P) will define BP as an atomic command which does three things, a BK, and UP, and a P. Macros can use commands defined by macros as well as built in commands in their definitions. For example, suppose Z is defined by (M Z -1 (IF (READP T) NIL (P))), i.e., Z does a -1, and then if nothing has been typed, a P. Now we can define ZZ by (M ZZ -1 Z), and ZZZ by (M ZZZ -1 -1 Z) or (M ZZZ -1 ZZ).

Macros can also define list commands, i.e., commands that take arguments.

(M (c) (arg₁ ... arg_n) . coms) c an atom. M defines c as a list command. Executing (c e₁ ... e_n) is then performed by substituting e₁ for arg₁, ... e_n for arg_n throughout coms, and then executing coms.

For example, we could define a more general BP by (M (BP) (N) (BK N) UP P). Thus, (BP 3) would perform (BK 3), followed by an UP, followed by a P.

A list command can be defined via a macro so as to take a fixed or indefinite number of "arguments", as with spread vs. nospread functions. The form given above specified a macro with a fixed number of arguments, as indicated by its argument list. If the "argument list" is *atomic*, the command takes an indefinite number of arguments.⁸⁴

(M (c) arg . coms) c, arg both atoms, defines c as a list command. Executing (c e₁ ... e_n) is performed by substituting (e₁ ... e_n), i.e., cdr of the command, for arg throughout coms, and then executing coms.

For example, the command 2ND, page 9.20, can be defined as a macro by (M (2ND) X (ORR ((LC . X) (LC . X)))).

⁸² To refer to the original definition of a built-in command when redefining it via a macro, use the ORIGINAL + command described below.

+

⁸³ If a macro is redefined, its new definition replaces its old.

⁸⁴ Note parallelism to EXPR's and EXPR*'s.

Note that for all editor commands, "built in" commands as well as commands defined by macros, atomic definitions and list definitions are *completely* independent. In other words, the existence of an atomic definition for c in *no* way affects the treatment of c when it appears as car of a list command, and the existence of a list definition for c in *no* way affects the treatment of c when it appears as an atom. In particular, c can be used as the name of either an atomic command, or a list command, or both. In the latter case, two entirely different definitions can be used.

Note also that once c is defined as an atomic command via a macro definition, it will *not* be searched for when used in a location specification, unless it is preceded by an F. Thus (INSERT -- BEFORE BP) would not search for BP, but instead perform a BK, and UP, and a P, and then do the insertion. The corresponding also holds true for list commands.

Occasionally, the user will want to employ the S command in a macro to save some temporary result. For example, the SW command could be defined as:

```
(M (SW) (N M) (NTH N) (S FOO 1) MARK 0 (NTH M) (S FIE 1) 85
      (I 1 FOO) ↔ (I 1 FIE))
```

Since this version of SW sets foo and fie, using SW may have undesirable side effects, especially when the editor was called from deep in a computation, we would have to be careful to make up unique names for dummy variables used in edit macros, which is bothersome. Furthermore, it would be impossible to define a command that called itself recursively while setting free variables. The BIND command solves both problems.

(BIND . coms) binds three dummy variables #1, #2, #3, (initialized to NIL), and then executes the edit commands coms. Note that these bindings are only in effect while the commands are being executed, and that BIND can be used recursively; it will rebind #1, #2, and #3 each time it is invoked.⁸⁶

Thus we could now write SW safely as:

```
(M (SW (N M) (BIND (NTH N) (S #1 1) MARK 0 (NTH M) (S #2 1)
      (I 1 #1) ↔ (I 1 #2)))).
```

User macros are stored on a list usermacros. The file package command USERMACROS (Section 14), is available for dumping all or selected user macros.

+ **(ORIGINAL . coms)** executes coms without regard to macro definitions. Useful for redefining a built in command in terms of itself., i.e. effectively allows user to "advise" edit commands.

85 A more elegant definition would be:

(M (SW) (N M) (NTH N) MARK 0 (NTH M) (S FIE 1) (I 1 (## ← 1)) ↔ (I 1 FIE)),
but this would still use one free variable.

86 BIND is implemented by (PROG (#1 #2 #3) (EDITCOMS (CDR COM))) where com corresponds to the BIND command, and editcoms is an internal editor function which executes a list of commands.

9.9 MISCELLANEOUS COMMANDS

9.9.1 COMMANDS FOR LEAVING THE EDITOR

OK exits from the editor

STOP exits from the editor with an error. Mainly for use in conjunction with TTY: commands that the user wants to abort.

Since all of the commands in the editor are errorset protected, the user must exit from the editor via a command.⁸⁷ STOP provides a way of distinguishing between a successful and unsuccessful (from the user's standpoint) editing session. For example, if the user is executing (MOVE 3 TO AFTER COND TTY:), and he exits from the lower editor with an OK, the MOVE command will then complete its operation. If the user wants to abort the MOVE command, he must make the TTY: command generate an error. He does this by exiting from the lower editor with a STOP command. In this case, the higher editor's edit chain will not be changed by the TTY: command.

SAVE exits from the editor and saves the "state of the edit" on the property list of the function or variable being edited under the property EDIT-SAVE. If the editor is called again on the same structure, the editing is effectively "continued," i.e., the edit chain, mark list, value of unfind and undolist are restored.

For example:

```
*P  
(NULL X)  
*F COND P  
(COND (& &) (T &))  
*SAVE  
FOO  
. . .  
←EDITF(FOO)  

```

SAVE is necessary only if the user is editing many different expressions; an exit from the editor via

⁸⁷ Or by typing a control-D. STOP is preferred even if the user is editing at the evalqt level, as it will perform the necessary "wrapup" to insure that the changes made while editing will be undoable (see Section 22).

OK always saves the state of the edit of that call to the editor.⁸⁸ Whenever the editor is entered, it checks to see if it is editing the same expression as the last one edited. In this case, it restores the mark list, the undolist, and sets unfind to be the edit chain as of the previous exit from the editor. For example:

```

←EDITF(FOO)
EDIT
*P
(LAMBDA (X) (PROG & & LP & & & &))

.
.
.
*P
(COND & &)
*OK
FOO
.
.
.
any number of lisp inputs
except for calls to the editor

←EDITF(FOO)
EDIT
*P
(LAMBDA (X) (PROG & & LP & & & &))
*\ P
(COND & &)
*
```

Furthermore, as a result of the history feature (section 22), if the editor is called on the same expression within a certain number of lisp inputs,⁸⁹ the state of the edit of that expression is restored, regardless of how many other expressions may have been edited in the meantime.

⁸⁸ on the property list of the atom EDIT, under the property name LASTVALUE. OK also removes EDIT-SAVE from the property list of the function or variable being edited.

⁸⁹ Namely, the size of the history list, initially 30, but it can be increased by the user.

For example:

```

←EDITF(FOO)
EDIT
*
.
.
.
*P
(COND (& &) (& &) (&) (T &))
*OK
FOO
← .           less than 30 lisp inputs, including editing
.
.
←EDITF(FOO)
EDIT
*\ P
(COND (& &) (& &) (&) (T &))
*
```

Thus the user can always continue editing, including undoing changes from a previous editing session, if

- (1) No other expressions have been edited since that session;⁹⁰ or
- (2) That session was "sufficiently" recent; or
- (3) It was ended with a SAVE command.

9.9.2 NESTED CALLS TO EDITOR

TTY: calls the editor recursively. The user can then type in commands, and have them executed. The TTY: command is completed when the user exits from the lower editor. (see OK and STOP below).

The TTY: command is extremely useful. It enables the user to set up a complex operation, and perform interactive attention-changing commands part way through it. For example the command (MOVE 3 TO AFTER COND 3 P TTY:) allows the user to interact, in effect, *within* the MOVE command. Thus he can verify for himself that the correct location has been found, or complete the specification "by hand." In effect, TTY: says "I'll tell you what you should do when you get there."

The TTY: command operates by printing TTY: and then calling the editor. The initial edit chain in the lower editor is the one that existed in the higher editor at the time the TTY: command was entered. Until the user exits from the lower editor, any attention changing commands he executes

⁹⁰ Since saving takes place at *exit* time, intervening calls that were aborted via control-D or exited via STOP will not affect the editor's memory of this last session.

only affect the lower editor's edit chain.⁹¹ When the TTY: command finishes, the lower editor's edit chain becomes the edit chain of the higher editor.

+ EF calls editf on car of current expression.

+ EV, EP calls editv, editp on car of current expression.

9.9.3 MANIPULATING THE CHARACTERS OF AN ATOM OR STRING

RAISE is an edit macro defined as UP followed by (I 1 (U-CASE (## 1))), i.e., it raises to upper-case the current expression, or if a tail, the first element of the current expression.

LOWER Similar to RAISE, except uses l-case.

CAP First does a RAISE; and then lowers all but the first character, i.e., the first character is left capitalized.

Note: RAISE, LOWER, and CAP are all no-ops if the corresponding atom or string is already in that state.

(RAISE x) equivalent to (I R (L-CASE x) x), i.e., changes every lower-case x to upper-case in the current expression.

(LOWER x) similar to RAISE, except performs (I R x (L-CASE x)).

Note in both (RAISE x) and (LOWER x), x is typed in in upper case.

⁹¹ Of course, if the user performs any structure modification commands while under a TTY: command, these will modify the structure in both editors, since it is the same structure.

REPACK Permits the "editing" of an atom or string.

For example:

```
*P
... "THIS IS A LOGN STRING")
*REPACK
*EDIT
P
(T H I S % I S % A % L O G N % S T R I N G)
*(S W G N)
*OK
"THIS IS A LONG STRING" 92
*
```

REPACK operates by calling the editor recursively on unpack of the current expression, or if it is a list, on unpack of its first element. If the lower editor is exited successfully, i.e., via OK as opposed to STOP, the list of atoms is made into a single atom or string, which replaces the atom or string being 'repacked.' The new atom or string is always printed.

(REPACK @) does (LC . @) followed by REPACK, e.g. (REPACK THIS\$).

9.9.4 MANIPULATING PREDICATES AND CONDITIONAL EXPRESSIONS

JOINC is used to join two neighboring COND's together, e.g. (COND clause₁ clause₂) followed by (COND clause₃ clause₄) becomes (COND clause₁ clause₂ clause₃ clause₄). JOINC does an (F COND T) first so that you don't have to be at the first COND.

(SPLITC x) splits one COND into two. x specifies the last clause in the first COND,
e.g. (SPLITC 3) splits
(COND clause₁ clause₂ clause₃ clause₄) into (COND clause₁ clause₂)
(COND clause₃ clause₄). Uses generalized NTH command, so that x does not have to be a number, e.g., the user can say (SPLITC RETURN), meaning split after the clause containing RETURN. SPLITC also does an (F COND T) first.

NEGATE negates current expression, i.e. performs (MBD NOT), except is smart about simplifying. For example, if current expression is: (OR (NULL X) (LISTP X)), NEGATE would change it to (AND X (NLISTP X)). NEGATE is implemented via the function ncgate. + + + + +

SWAPC takes a conditional expression of the form (COND (A B)(T C)) +

92 Note that this could also have been accomplished by (R \$GN\$ \$NG\$) or simply (RC GN NG).

+ and rearranges it to an equivalent (COND ((NOT A) C)(T B)),
+ or (COND (A B) (C D)) to
+ (COND ((NOT A) (COND (C D)))) (T B)).

+ SWAPC is smart about negations (uses NEGATE) and simplifying
+ conds. It always produces an equivalent expression. It is useful for
+ those cases where one wants to insert extra clauses or tests.

9.9.5 HISTORY COMMANDS IN THE EDITOR

As described in Section 22, all of the user's inputs to the editor are stored on edithistory, the editor's history list, and all of the programmer's assistant commands for manipulating the history list, e.g. REDO, USE, FIX, NAME, etc., are available for use on events on edithistory. In addition, the following four history commands are recognized specially by the editor. They always operate on the last, i.e. most recent, event.

DO com allows the user to supply the command name when it was omitted.⁹³

For example, suppose the user wants to perform (-2 (SETQ X (LIST Y Z))) but instead types just (SETQ X (LIST Y Z)). The editor will type SETQ ?, whereupon the user can type DO -2. The effect is the same as though the user had typed FIX, followed by (LI 1), (-1 -2), and OK, i.e., the command (-2 (SETQ X (LIST Y Z))) is executed. DO also works if the command is a line command.

!F same as DO F.

In the case of !F, the previous command is always treated as though it were a line command, e.g., if the user types (SETQ X &) and then !F, the effect is the same as though he had typed F (SETQ X &), not (F (SETQ X &)).

!E same as DO E.

!N same as DO N.

9.9.6 MISCELLANEOUS

NIL unless preceded by F or BF, is always a no-op. Thus extra right parentheses or square brackets at the ends of commands are ignored.

CL Clispifies current expression. See Section 23.

⁹³ USE is useful when a command name is *incorrect*.

DW

Dwimifies current expression. See Section 17 and 23.

GET*

If the current expression is a comment pointer (see Section 14), reads in the full text of the comment, and replaces the current expression by it.

(* . x)

x is the text of a comment. * ascends the edit chain looking for a "safe" place to insert the comment, e.g., in a cond clause, after a prog statement, etc., and inserts (* . x) *after* that point, if possible, otherwise before. For example, if the current expression is (FACT (SUB1 N)) in

```
[COND ((ZEROP N) 1)
      (T (ITIMES N (FACT (SUB1 N)))
```

(* CALL FACT RECURSIVELY) would insert
 (* CALL FACT RECURSIVELY) *before* the itimes expression.⁹⁴

* does not change the edit chain, but unfind is set to where the comment was actually inserted.

GETD

essentially "expands" the current expression in line: (1) if (car of) the current expression is the name of a macro, expands the macro in line; (2) if a clisp word, translates the current expression and replaces it with the translation; (3) if car is the name of a function,⁹⁵ substitutes the argument expressions for the corresponding argument names in the body of the definition and replaces the current expression with the result; (4) if car of the current expression is an open lambda, substitutes the arguments for the corresponding argument names in the body of the lambda, and then removes the lambda and argument list.

(MAKEFN (fn . actualargs) arglist n1 n2)

the inverse of GETD: makes the current expression into a function. fn is the function name, arglist its arguments. The argument names are substituted for the corresponding argument values in actualargs, and the result becomes the body of the function definition for fn. The current expression is then replaced with (fn . actualargs).⁺

For example, if the current expression is (COND ((CAR X)(PRINT Y T))(T (HELP))), then (MAKEFN (FOO (CAR X) Y) (A B)), will define FOO as

⁹⁴ If inserted after the itimes, the comment would then be (incorrectly) returned as the value of the cond. However, if the cond was itself a prog statement, and hence its value was not being used, the comment could be (and would be) inserted after the itimes expression.

⁹⁵ for which the editor can obtain a symbolic definition, either in-core or from a file.

+ (LAMBDA (A B) (COND (A (PRINT B T)) (T (HELP)))) and then replace the current expression with (FOO (CAR X) Y).

+ If n1 and n2 are supplied, (N1 THRU N2) is used rather than the current expression; if just n1 is supplied, (N1 THRU -1) is used.

+ If arglist is omitted, MAKEFN will make up some arguments, using elements of actualargs, if they are literal atoms, otherwise arguments selected from (X Y Z A B C ...), avoiding duplicate argument names.

+ (MAKE argname exp) makes the value of argname be exp in the call which is the current expression, i.e. a ?= command following a MAKE will always print argname=exp. For example:

```
*P
(JSYS)
*?=
JSYS[N;AC1,AC2,AC3,RESULTAC]
*(MAKE N 10)
*(MAKE RESULTAC 3)
*P
(JSYS 10 NIL NIL NIL 3)
```

+ Q quotes current expression, i.e. MBD QUOTE.

+ D deletes current expression, then prints new current expression, i.e. (: I P.

9.10 UNDO

Each command that causes structure modification automatically adds an entry to the front of undolst that contains the information required to restore all pointers that were changed by that command.

UNDO

undoes the last, i.e., most recent, structure modification command that has not yet been undone,⁹⁶ and prints the name of that command, e.g., MBD UNDONE. The edit chain is then *exactly* what

⁹⁶ Since UNDO and !UNDO cause structure modification, they also add an entry to undolst. However, UNDO and !UNDO entries are skipped by UNDO, e.g., if the user performs an INSERT, and then an MBD, the first UNDO will undo the MBD, and the second will undo the INSERT. However, the user can also specify precisely which commands he wants undone by identifying the corresponding entry on the history list as described in Section 22. In this case, he can undo an UNDO command, e.g., by typing UNDO UNDO, or undo a !UNDO command, or undo a command other than that most recently performed.

it was before the "undone" command had been performed.⁹⁷ If there are no commands to undo, UNDO types **NOTHING SAVED**.

!UNDO undoes all modifications performed during this editing session, i.e. this call to the editor. As each command is undone, its name is printed a la UNDO. If there is nothing to be undone, !UNDO prints **NOTHING SAVED**.

Whenever the user *continues* an editing session as described on page 9.47-49, the undo information of the previous session is protected by inserting a special blip, called an undo-block, on the front of undolst. This undo-block will terminate the operation of a !UNDO, thereby confining its effect to the current session, and will similarly prevent an UNDO command from operating on commands executed in the previous session.

Thus, if the user enters the editor continuing a session, and immediately executes an UNDO or !UNDO, the editor will type **BLOCKED** instead of **NOTHING SAVED**. Similarly, if the user executes several commands and then undoes them all, another UNDO or !UNDO will also cause **BLOCKED** to be typed.

UNBLOCK removes an undo-block. If executed at a non-blocked state, i.e., if UNDO or !UNDO *could* operate, types **NOT BLOCKED**.

TEST adds an undo-block at the front of undolst.

Note that TEST together with !UNDO provide a "tentative" mode for editing, i.e., the user can perform a number of changes, and then undo all of them with a single !UNDO command.

UNDO \diamond \diamond is an event specification (Section 22). Undoes the indicated event on the history list. In this case, the event does not have to be in the current editing session, even if the previous session has not been unblocked as described above. However, the user does have to be editing the same expression as was being edited in the indicated event.⁹⁸

⁹⁷ Undoing an event containing an I, E, or S command will also undo the side effects of the evaluation(s), e.g., undoing (I 3 (/NCONC FOO FIE)) will not only restore the 3rd element but also restore FOO. Similarly, undoing an S command will undo the set. See discussion of UNDO in Section 22. (Note that if the I command was typed directly to the editor, /NCONC would automatically be substituted for NCONC as described in Section 22.)

⁹⁸ If the expressions differ, the editor types the warning message "**different expression**", and does not undo the event. The editor enforces this to avoid the user accidentally undoing a random command by giving the wrong event specification.

9.11 EDITDEFAULT

Whenever a command is not recognized, i.e., is not "built in" or defined as a macro, the editor calls an internal function, editdefault, to determine what action to take.⁹⁹ If a location specification is being executed, an internal flag informs editdefault to treat the command as though it had been preceded by an F.

If the command is a list, an attempt is made to perform spelling correction on car of the command¹⁰⁰ using editcomsl, a list of all list edit commands.¹⁰¹ If spelling correction is successful,¹⁰² the correct command name is replaced into the command, and the editor continues by executing the command. In other words, if the user types (LP F PRINT (MBBD AND (NULL FLG))), only one spelling correction will be necessary to change MBBB to MBD. If spelling correction is not successful, an error is generated.

If the command is atomic, the procedure followed is a little more elaborate.

- 1) If the command is one of the list commands, i.e., a member of editcomsl, and there is additional input on the same terminal line, treat the entire line as a single list command.¹⁰³ Thus, the user may omit parentheses for any list command typed in at the top level (provided the command is not also an atomic command, e.g. NX, BK). For example,

```
*P
(COND (& &) (T &))
*XTR 3 2]
*MOVE TO AFTER LP
*
```

If the command is on the list editcomsl but no additional input is on the terminal line, an error is generated, e.g.

```
*P
(COND (& &) (T &))
*MOVE

MOVE ?
*
```

⁹⁹ Since editdefault is part of the edit block, the user cannot advise or redefine it as a means of augmenting or extending the editor. However, the user can accomplish this via edituserfn. If the value of the variable edituserfn is T, editdefault calls the function edituserfn giving it the command as an argument. If edituserfn returns a non-NIL value, its value is interpreted as a single command and executed. Otherwise, the error correction procedure described below is performed.

¹⁰⁰ unless dwinflg=NIL. See Section 17 for discussion of spelling correction.

¹⁰¹ When a macro is defined via the M command, the command name is added to editcomsa or editcomsl, depending on whether it is an atomic or list command. The prettydef command USERMACROS (Section 14), is aware of this, and provides for restoring editcomsa and editcomsl.

¹⁰² Throughout this discussion, if the command was not typed in directly, the user will be asked to approve the spelling correction. See Section 17.

¹⁰³ The line is read using readline (Section 14). Thus the line can be terminated by a square bracket, or by a carriage return not preceded by a space.

If the command is on editcomsl, and *not* typed in directly, e.g., it appears as one of the commands in a LP command, the procedure is similar, with the rest of the command stream at that level being treated as "the terminal line", e.g.

(LP F (COND (T &)) XTR 2 2).¹⁰⁴

- 2) If the command was typed in and the first character in the command is an 8, treat the 8 as a mistyped left parenthesis, and and the rest of the line as the arguments to the command, e.g.,

```
*P
(COND (& &) (T &))
*8-2 (Y (RETURN Z)))
=(-2
*P
(COND (Y &) (& &) (T &))
```

- 3) If the command was typed in, is the name of a function, and is followed by NIL or a list car of which is not an edit command, assume the user forgot to type E and means to apply the function to its arguments, type =E and the function name, and perform the indicated computation, e.g.

```
*BREAK(FOO)
=E BREAK
(FOO)
*
```

- 4) If the last character in the command is P, and the first n-1 characters comprise a number, assume that the user intended two commands, e.g.,

```
*P
(COND (& &) (T &))
*0P
=0 P
(SETQ X (COND & &))
```

- 5) Attempt spelling correction using editcomsa, and if successful,¹⁰⁵ execute the corrected command.
- 6) Otherwise, if there is additional input on the same line, or command stream, spelling correct using editcomsl as a spelling list, e.g.,

¹⁰⁴ Note that if the command is being executed in location context, editdefault does not get this far, e.g., (MOVE TO AFTER COND XTR 3) will search for XTR, *not execute it*. However, (MOVE TO AFTER COND (XTR 3)) will work.

¹⁰⁵ See footnote on page 9.57.

```
*MBBD SETQ X
=MBD
*
```

- 7) Otherwise, generate an error.

9.12 EDITOR FUNCTIONS

`editf[name;com1;com2;...;comn]`

nlambda, nospread function for editing a function. name is the name of the function, com₁, com₂, ..., com_n (optional) edit commands.

The value of editf is name.

- (1) In the most common case, name is an expr, and editf simply performs `putd[name;edite[getd[name];list[com1;com2;...;comn];name;FNS]]`. However, if name is an expr by virtue of its being broken or advised, and
 - (1a) the original definition is also an expr, then the broken/advised definition is given to edite to be edited (since any changes there will also affect the original definition because all changes are destructive). However, a warning message is printed to alert the user that he must first position himself correctly before he can begin typing commands such as (-3 --), (N --), etc.
 - (1b) the original definition is not an expr, and there is no EXPR property, and the file package "knows" which file name is contained in (see discussion of editloadfn? below), then the expr definition of name is loaded onto its property list as described in (3) below, and proceeds to (1c), otherwise a warning message is printed, and the edit proceeds, e.g., the user may have called the editor to examine the advice on a subr.
 - (1c) the original definition is not an expr, and there is an EXPR property, then the function is unbroken/unadvised (latter only with user's approval, since the user may really want to edit the advice) and proceed as in (2).
- (2) If name is not an expr, but has an EXPR property, editf prints PROP, and performs `edite[gctprop[name;EXPR];list[com1;com2;...;comn];name;PROP]]`. In this case, if the edit completes and no changes have been made, edite prints NOT CHANGED, SO NOT UNSAVED. If changes were made, but the value of dnameflg is PROP, edite prints CHANGED, BUT NOT UNSAVED. Otherwise if changes were made, edite prints UNSAVED and does an unsaveddef.
- (3) if name is neither an expr nor has an EXPR property, and the file package (see Section 14) "knows" which file name is contained in (see discussion of editloadfn? below), the expr definition of name is automatically loaded (using loadnames) onto its property list, and

proceed to (2) above.¹⁰⁶ In addition, if name is a member of a block (see Section 18), the user will be asked whether he wishes the rest of the functions in the block to be loaded at the same time.¹⁰⁷

- (4) If name is neither an expr nor has an EXPR property, but it does have a definition, editf generates an name NOT EDITABLE error.
- (5) If name is neither defined, nor has an EXPR property, but its top level value is a list, editf assumes the user meant to call edity, prints =EDITV, calls edity and returns. Similarly, if name has a non-NIL property list, editf prints =EDITP, calls editp and returns.
- (6) If name is neither a function, nor has an EXPR property, nor a top level value that is a list, nor a non-NIL property list, editf attempts spelling correction using the spelling list userwords,¹⁰⁸ and if successful, goes back to (1).

Otherwise, editf generates an name NOT EDITABLE error.

In all cases, if a function was edited, and changes were made, the function is time-stamped (by edite), which consists of inserting a comment of the form (* xxx date), where xxx are the user's initials.¹⁰⁹ If the function was already time-stamped, then only the date is changed.¹¹⁰

editfns[name;com₁;com₂;...;com_n]
lambda, nospread function, used to perform the same editing operations on several functions. name is evaluated to obtain a list

¹⁰⁶ Because of the existence of the file map (see Section 14), this operation is extremely fast, essentially requiring only the time to perform the READ to obtain the actual definition.

¹⁰⁷ The editor's behaviour in case (3) is controlled by the value of editloadfnsflg, which is a dotted pair of two flags, the first of which (i.e., car of editloadfnsflg) controls the loading of the function, and the second the loading of the block. A value of NIL for either flag means "load but ask first," a value of T means "don't ask, just do it" and anything else means "don't ask, don't do it." The initial value of editloadfnsflg is (T), meaning load the function without asking, ask about loading the block.

¹⁰⁸ Unless dwinflg=NIL. Spelling correction is performed using the function misspelled?. If name=NIL, misspelled? returns the last "word" referenced, e.g., by defineq, editf, prettyprint etc. Thus if the user defines foo and then types editf[], the editor will assume he meant foo, type =FOO, and then type EDIT. See Section 17.

¹⁰⁹ xxx is the value of the variable initials. After greeting, or following a sysin, the function setinitials is called. setinitials searches initialslist, a list of elements of the form (username . initials) or (username firstname initials). If the user's name is found, initials is set accordingly. If the user's name is not found on initialslist, initials is set to the value of defaultinitials, initially edited:. Thus, the default is to always time stamp. To suppress time stamping, the user must either include an entry of the form (username) on initialslist, or set defaultinitials to NIL before greeting, i.e. in his user profile, or else, after greeting, explicitly set initials to NIL.

¹¹⁰ The following three functions may be of use for specialized applications with respect to time-stamping: fixeddate[expr] which, given a lambda expression, inserts or smashes a time-stamp comment; editdate? [comment] which returns T if comment is a time stamp; and editdate[olddate;initls] which returns a new time-stamp comment. If olddate is a time-stamp comment, it will be reused.

of functions.¹¹¹ com₁, com₂, ..., com_n are (optional) edit commands. editfns maps down the list of functions, prints the name of each function, and calls the editor (via editf) on that function.¹¹²

For example, EDITFNS(FOOFNS (R FIE FUM)) will change every FIE to FUM in each of the functions on foofns.

The call to the editor is errorset protected, so that if the editing of one function causes an error, editfns will proceed to the next function.¹¹³

Thus in the above example, if one of the functions did not contain a FIE, the R command would cause an error, but editing would continue with the next function.

The value of editfns is NIL.

- + The function editcallers provides a way of rapidly searching a file or entire set of files, even files not loaded into Interlisp or "noticed" by the file package, for the appearance of one or more key words (atoms) anywhere in the file.¹¹⁴
- + editcallers[atoms;files;coms] uses ffilepos to search the file(s) files for occurrences of the atom(s) atoms. It then calls edite on each of those objects,¹¹⁵ performing the edit commands coms. If coms=NIL, then (EXAM . ATOMS) is used. Both atoms and files may be single atoms. If files is NIL, filelist is used. Elements on atoms may contain < esc > s.
- + editcallers prints the name of each file as it searches it, and when it finds an occurrence of one of atoms, it prints out either the name of the containing function or, if the atom occurred outside a function definition, it prints out the byte position that the atom was found.
- + editcallers will read in and use the filemap of the file. In the case that the editor is actually called, editcallers will loadfrom the file if the file has not previously been noticed.

¹¹¹ If name is atomic, and its value is not a list, and it is the name of a file, filefnslst[name] will be used as the list of functions to be edited.

¹¹² i.e., the definition of editfns might be:

```
[MAPC (EVAL (CAR X)) (FUNCTION (LAMBDA (Y)
                                         (APPLY (QUOTE EDITF)
                                         (CONS (PRINT Y T) (CDR X)
```

¹¹³ In particular, if an error occurred while editing a function via its EXPR property, the function would not be unsaved. In other words, in the above example, only those functions which contained a FIE, i.e., only those actually changed, would be unsaved.

+ ¹¹⁴ editcallers was written by L. M. Masinter.

+ ¹¹⁵ editcallers uses getdef (Section 14) to obtain the "definition" for each object. When edite returns, if a change was made, putdef is called to store the changed object.

`findcallers[atoms;files]` like `editcallers`, except does not call the editor, but instead simply +
returns the list of files that contain one of `atoms`. +

`editv[name;com1;com2;...;comn]` nlambda, nospread function, similar to `editf`, for editing values.

If `name` is a list, it is evaluated and its value given to `edite`, e.g., EDITV((CDR (ASSOC (QUOTE FOO) DICTIONARY))). In this case, the value of `editv` is T.

However, for most applications, `name` is a variable name, i.e., atomic, as in EDITV(FOO). If the value of this variable is NOBIND, `editv` checks to see if it is the name of a function, and if so, assumes the user meant to call `editf`, prints =EDITF, calls `editf` and returns. Otherwise, `editv` attempts spelling correction using the list `userwords`.¹¹⁶ Then `editv` will call `edite` on the value of `car[editvx]` (or the corrected spelling thereof), and `type=VARS`. Thus, if the value of `foo` is NIL, and the user performs (EDITV FOO), no spelling correction will occur, since `foo` is the name of a variable in the user's system, i.e., it has a value. However, `edite` will generate an error, since `foo`'s value is not a list, and hence not editable. If the user performs (EDITV FOO0), where the value of `foo0` is NOBIND, and `foo` is on the user's spelling list, the spelling corrector will correct FOO0 to FOO. Then `edite` will be called on the value of `foo`. Note that this may still result in an error if the value of `foo` is not a list.

The value of `editv` is the name of the variable whose value was edited.

`editp[name;com1;com2;...;comn]` nlambda, nospread function, similar to `editf` for editing property lists. If the property list of `name` is NIL, `editp` attempts spelling correction using `userwords`. Then `editp` calls `edite` on the property list of `name`, (or the corrected spelling thereof), with `type=PROPLST`. When (if) `edite` returns, `editp` calls `setproplist` on `name` with the value returned.

The value of `editp` is the atom whose property list was edited.

`edite[expr;coms;atm;type;ifchangedfn]` edits the expression, `expr`, by calling `editl` on `list[expr]` and returning the last element of the value returned by `editl`. Generates *
an error if `expr` is not a list. *

`atm` and `type` are for use in conjunction with the file package. If supplied, `atm` is the `name` of the object that `expr` is associated with, and `type` describes the association. (i.e., `type` corresponds to the `type` argument of `newfile`.) For example, if `expr` is the definition of `foo`, `atm=FOO` and `type=FNS`. When `edite` is called from `editp`, `expr` is the property list of `atm`, and `type=PROPLST`, etc.. +

`edite` calls `editl` to do the editing (described below). Upon return, if +

¹¹⁶ Unless `dwimflg=NIL`. Misspelled? is also called if `car[editvx]` is NIL, so that EDITV() will edit lastword.

+ both atm and type are non-NIL, addspell is called to add atm to the appropriate spelling list. Then, if expr was changed,¹¹⁷ and the value of ifchangedfn is not NIL, the value of ifchangedfn is applied to the arguments atm, expr, type, and a flag which is T for normal edits from editor, NIL for calls that were aborted via control-D or STOP. Otherwise, if expr was changed, and the value of ifchangedfn is NIL, and type is not NIL, newfile? (see Section 14) is called on, atm and type. edite uses resetsave to insure that ifchangedfn/newfile? are called if any change was made even if editing is subsequently aborted via control-D. (In this case, the fourth argument to ifchangedfn will be NIL.)

editl[1;coms;atm;mess;edithanges]

editl¹¹⁸ is the editor. Its first argument is the edit chain, and its value is an edit chain, namely the value of l at the time editl is exited.¹¹⁹

coms is an optional list of commands. For interactive editing, coms is NIL. In this case, editl types EDIT and then waits for input from terminal.¹²⁰ All input is done with editrdtbl as a readable. Exit occurs only via an OK, STOP, or SAVE command.

If coms is not NIL, no message is typed, and each member of coms is treated as a command and executed. If an error occurs in the execution of one of the commands, no error message is printed, the rest of the commands are ignored, and editl exits with an error, i.e., the effect is the same as though a STOP command had been executed. If all commands execute successfully, editl returns the current value of l.

atm is optional. On calls from editf, it is the name of the function being edited; on calls from edyt, the name of the variable, and calls from editp, the atom whose property list is being edited. The property list of atm is used by the SAVE command for saving the state of the edit. Thus SAVE will not save anything if atm=NIL, i.e., when editing arbitrary expressions via edite or editl directly.

edithanges is used for communicating with edite.

+

¹¹⁷ For type=FNS or type=PROP, i.e. calls from editf, edite performs some additional operations as described earlier under editf.

¹¹⁸ edit-all, not edit-one.

¹¹⁹ l is a specvar, and so can be examined or set by edit commands. For example, ↑ is equivalent to (E (SETQ L (LAST L)) T). However, the user should only manipulate or examine l directly as a last resort, and then with caution.

¹²⁰ If mess is not NIL, editl types it instead of EDIT. For example, the TTY: command is essentially (SETQ L (EDITL L NIL NIL (QUOTE TTY:))).

`editl0[l;coms;mess;-]`

like editl except does not rebind or initialize the editor's various state variables, such as lastail, unfind, undolist, marklist, etc. Should only be called when already under a call to editl.

`edit4e[pat;x;-]`

is the pattern match routine. Its value is T, if pat matches x. See page 9.14-16 for definition of "match".

Note: before each search operation in the editor begins, the entire pattern is scanned for atoms or strings containing <esc>s. These are replaced by patterns of the form (CONS (QUOTE \$) (UNPACK atom/string)) for 6a, and (CONS (QUOTE \$\$) (CONS (NCHARS atom/string) (UNPACK atom/string))), for 6b.¹²¹ Thus from the standpoint of edit4e, pattern type 6a is indicated by car[pat] being the atom \$ (\$ is <esc>) and pattern type 6b by car[pat] being the atom \$\$ (<esc><esc>).

Therefore, if the user wishes to call edit4e directly, he must first convert any patterns which contain atoms or strings ending in <esc>s to the form recognized by edit4e. This is done with the function editfpat.

`editfpat[pat;-]`

makes a copy of pat with all patterns of type 6 converted to the form expected by edit4e.

`editfindp[x;pat;flg]`

allows a program to use the edit find command as a pure predicate from outside the editor. x is an expression, pat a pattern. The value of editfindp is T if the command F pat would succeed, NIL otherwise. editfindp calls editfpat to convert pat to the form expected by edit4e, unless flg=T. Thus, if the program is applying editfindp to several different expressions using the same pattern, it will be more efficient to call editfpat once, and then call editfindp with the converted pattern and flg=T.

`esubst[old;new;expr;errorflg;charflg]`

equivalent to performing (R y x)¹²² with z as the current expression, i.e., the order of arguments is the same as for subst. Note that y and/or x can employ <esc>s. The value of esubst is the modified z. Generates an error if y not found in z. If errorflg=T, also prints an error message of the form y ?.

esubst is always undoable.

¹²¹ In latter case, atom/string corresponds to the atom or string up to but not including the final two-<esc>s. In both cases, dunpack is used wherever possible.

¹²² unless charflg=T and no <esc>s are specified in new or old, in which case it is equivalent to (RC y x). See page 9.39. In other words, if charflg=T, and no <esc>s appear, esubst will supply them.

- + `editloadfn? [fn;str;askflg;files]` fn is the name of a function. Value is name of file fn is contained in, or NIL.
- + editloadfn? performs whereis[fn;FNS;files] to obtain the name of the file(s) containing fn, if any.¹²³ It then checks the FILEDATES property for each file to see if the version that was originally loaded still exists.¹²⁴ If the file that was *originally* loaded no longer exists, but there is a different version of the file on that directory, editloadfn? prints "****can't find" followed by the file name, and then uses the version that it could find. Similarly, if the original version *is* found, but a newer version is also found, editloadfn? prints "****Note: <filename> is not the newest version" and then uses the newest version.
- + Having decided which file the function is on, if askflg=NIL, editloadfn? prints the value of str followed by the name of the file, and returns the name of the file. If askflg=T, editloadfn? calls askuser giving list[fn;str;filename] as mess, the message to be printed. If askuser returns Y, editloadfn? returns the filename. If str=NIL, "loading from" is used.
- + editloadfn? is used by the editor, loadfn (when file name is not supplied), by prettyprint, and by dwim.

changename[fn;from;to]

replaces all occurrences of from by to in the definition of fn. If fn is an expr, changename performs nlsetq[esubst[to;from;getd[fn]]]. If fn is *compiled*, changename searches the literals of fn (and all of its compiler generated subfunctions), replacing each occurrence of from with to.¹²⁵

The value of changename is fn if at least one instance of from was found, otherwise NIL.

changename is used by break and advise for changing calls to fn1 to calls to fn1-IN-fn2.

editracefn[com]

is available to help the user debug complex edit macros, or subroutine calls to the editor. If editracefn is set to T, the function editracefn is called whenever a command that was not typed in by the user is about to be executed, giving it that command as its argument. However, the TRACE and BREAK options described below are probably sufficient for most applications.

- + 123 If there is more than one file, editloadfn? asks the user to indicate which file.
- + 124 In the case that files=T and the whereis package (Section 24) has been loaded, file(s) may be found that have not been loaded or otherwise noticed, and thus will not have FILEDATES property. In this case, editloadfn? does not do any version checks, but simply uses the latest version.
- 125 Will succeed even if from is called from fn via a linked call. In this case, the call will also be relinked to call_to instead.

If editracefn is set to TRACE, the name of the command and the current expression are printed. If editracefn=BREAK, the same information is printed, and the editor goes into a break. The user can then examine the state of the editor.

edittracefn is initially NIL.

Index for Section 9

	Page Numbers
(A e1 ... em) (edit command)	9.26,9
AFTER (in INSERT command) (in editor)	9.27
AFTER (in MOVE command) (in editor)	9.31
(B e1 ... em) (edit command)	9.26,9
BEFORE (in MOVE command) (in editor)	9.31
(BELOW com x) (edit command)	9.20
(BELOW com) (edit command)	9.21
(BF pattern T) (edit command)	9.18
BF (edit command)	9.18,8
(BI n m) (edit command)	9.34,6
(BI n) (edit command)	9.34
(BIND . coms) (edit command)	9.46
(BK n) (n a number, edit command)	9.13
BK (edit command)	9.12,7,13
BLOCKED (printed by editor)	9.55
(BO n) (edit command)	9.34,6
BY (in REPLACE command) (in editor)	9.27
CAN'T - AT TOP (printed by editor)	9.12,3
CAP (edit command)	9.50
(CHANGE @ TO ...) (edit command)	9.27
CHANGECHAR (prettyprint variable/parameter)	9.25
CHANGENAME[FN:FROM;TO]	9.64
CHANGESARRAY (system variable/parameter)	9.24
CL (edit command)	9.52
commands that move parentheses (in editor)	9.33-35
comment pointers	9.53
(COMS x1 ... xn) (edit command)	9.43
(COMSQ . coms) (edit command)	9.43
continuing an edit session	9.48-49
control-A (TOPS-20) (edit command)	9.14
control-D	9.47
control-E	9.2
control-L (TOPS-20) (edit command)	9.14
control-X (edit command)	9.14
control-Z (edit command)	9.14
current expression (in editor)	9.2,3,6,8-14,16-23
D (edit command)	9.54
DEFAULTINITIALS (editor variable/parameter)	9.59
DELETE (edit command)	9.26,9,24
(DELETE . @) (edit command)	9.28
DESTINATION IS INSIDE EXPRESSION BEING MOVED (printed by editor)	9.31
DIFFERENT EXPRESSION (printed by editor)	9.55
DO (edit command)	9.52
DW (edit command)	9.53
DWIMFLG (dwim variable/parameter)	9.56,59,61
(E x T) (edit command)	9.42
(E x) (edit command)	9.42
E (edit command)	9.41,7,42
edit chain	9.3,5,8-14,16-23
edit commands that search	9.14-22
edit commands that test	9.43
edit macros	9.45-46
EDIT (printed by editor)	9.62
EDITCALLERS[ATOMS;FILES;COMS]	9.60
EDITCOMSA (editor variable/parameter)	9.56,57

	Page Numbers
EDITCOMSL (editor variable/parameter)	9.56,57
EDITDATE[OLDDATE;INITLS]	9.59
EDITDATE?[COMMENT]	9.59
EDITDEFAULT	9.56,57-58
EDITE[EXPR;COMS;ATM;TYPE;IFCHANGEDFN]	9.61,1,61
EDITF[NAME;COM1;COM2;...;COMn] NL*	9.58,1,59,61
EDITFINDP[X;PAT;FLG]	9.63
EDITFNS[NAME;COM1;COM2;...;COMn] NL*	9.59,60
EDITFPAT[PAT;FLG]	9.63
editing compiled code	9.64
EDITL[L;COMS;ATM;MESS;EDITCHANGES]	9.62
EDITLOADFNSFLG (editor variable/parameter)	9.59
EDITLOADFNS?[FN;STR;ASKFLG;FILES]	9.64
EDITL0[L;COMS;MESS;EDITLFLG]	9.63
EDITP[NAME;COM1;COM2;...;COMn] NL*	9.61,1
EDITQUIETFLG (editor variable/parameter)	9.15
EDITTRACEFN	9.64,65
EDITUSERFN[COM]	9.56
EDITV[NAME;COM1;COM2;...;COMn] NL*	9.61,1
EDIT-SAVE (property name)	9.47-48
EDIT4E[PAT;X;CHANGEFLG]	9.63
EF (edit command)	9.50
(EMBED @ IN ...) (edit command)	9.31
EP (edit command)	9.50
errors (in editor)	9.2
ESUBST[NEW;OLD;EXPR;ERRORFLG;CHARFLG]	9.63
EV (edit command)	9.50
EVAL (edit command)	9.42
(EXAM . x) (edit command)	9.44
EXPR (property name)	9.58-60
(EXTRACT @1 from . @2) (edit command)	9.30
(F pattern N) (edit command)	9.17
(F pattern n) (n a number, edit command)	9.17
(F pattern T) (edit command)	9.17
(F pattern) (edit command)	9.18
F pattern (edit command)	9.17
F (edit command)	9.17,4-5
FINDCALLERS[ATOMS;FILES]	9.61
FIXEDITDATE[EXPR]	9.59
FOR (in INSERT command) (in editor)	9.27
FROM (in EXTRACT command) (in editor)	9.30
(FS . patterns) (edit command)	9.18
(F= expression fig) (edit command)	9.18
generalized NTH command (in editor)	9.21,33,40
GETD (edit command)	9.53
GETVAL (edit command)	9.42
GET* (edit command)	9.53
(GO label) (edit command)	9.19
HERE (in edit command)	9.28
history list	9.48,54
(I c x1 ... xn) (edit command)	9.42
(IF x coms1 coms2) (edit command)	9.43
(IF x coms1) (edit command)	9.44
(IF x) (edit command)	9.43
implementation of structure modification commands (in editor)	9.25-26
IN (in EMBED command) (in editor)	9.31

Page
Numbers

INITIALS (editor variable/parameter)	9.59
INITIALSLST (editor variable/parameter)	9.59
(INSERT ... AFTER . @) (edit command)	9.27
(INSERT ... FOR . @) (edit command)	9.27
JOINC (edit command)	9.51
LASTAIL (editor variable/parameter)	9.11, 17, 63
LASTVALUE (property name)	9.48
LASTWORD (dwim variable/parameter)	9.61
(LC . @) (edit command)	9.19
(LCL . @) (edit command)	9.19
(LI n) (edit command)	9.34, 6
line-feed (edit command)	9.14
LISPX[LISPXX;LISPXID;LISPXXMACROS;LISPXXUSERFN; LISPXFLG]	9.41, 48
(LO n) (edit command)	9.34, 6
location specification (in editor)	9.19, 43
LOCATION UNCERTAIN (printed by editor)	9.11
(LOWER x) (edit command)	9.50
LOWER (edit command)	9.50
(LP . coms) (edit command)	9.44
(LPQ . coms) (edit command)	9.44
L-CASE[X;FLG]	9.50
M (edit command)	9.45
macros (in editor)	9.45-46
(MAKE argname exp) (edit command)	9.54
(MAKEFN (fn . actualargs) arglist n1 n2) (edit command)	9.53
(MARK atom) (edit command)	9.22
MARK (edit command)	9.22
MARKLST (editor variable/parameter)	9.22, 63
MAXLEVEL (editor variable/parameter)	9.16, 18
MAXLOOP EXCEEDED (printed by editor)	9.44
MAXLOOP (editor variable/parameter)	9.44
(MBD e1 ... em) (edit command)	9.30
(MOVE @1 TO com . @2) (edit command)	9.31-33
(N e1 ... em) (edit command)	9.24
(n e1 ... em) (n a number, edit command)	9.24, 4
(n) (n a number, edit command)	9.24, 4
n (n a number, edit command)	9.11, 2
NEGATE (edit command)	9.51
(NEX x) (edit command)	9.21
NEX (edit command)	9.21
NIL (edit command)	9.52, 43
NOBIND	9.61
NOT BLOCKED (printed by editor)	9.55
NOT EDITABLE (error message)	9.59, 61
NOTHING SAVED (printed by editor)	9.55
(NTH n) (n a number, edit command)	9.14
(NTH x) (edit command)	9.21
(NX n) (n a number, edit command)	9.12
NX (edit command)	9.12, 6
OCCURRENCES (printed by editor)	9.44
OK (edit command)	9.47, 51, 62
(ORF . patterns) (edit command)	9.18
(ORIGINAL . coms) (edit command)	9.45-46
(ORR coms1 ... comsn) (edit command)	9.44
(P m n) (edit command)	9.40

Page
Numbers

(P m) (edit command)	9.40
P (edit command)	9.40,2
pattern match (in editor)	9.14,15-16,63
(pattern .. @) (edit command)	9.21
PP (edit command)	9.40,2
PPT (edit command)	9.41
PPV (edit command)	9.41
PP* (edit command)	9.41
prompt character	9.1
PROP (printed by editor)	9.58
Q (edit command)	9.54
R (edit command)	9.38,5
(RAISE X) (edit command)	9.50
RAISE (edit command)	9.50
(RC x y) (edit command)	9.39
(RC1 x y) (edit command)	9.39
READLINE[RDTBL;LINE;LISPXFLG]	9.56
REPACK (edit command)	9.51
(REPACK @) (edit command)	9.51
(REPLACE @ WITH ...) (edit command)	9.27
(RI n m) (edit command)	9.34,6
(RO n) (edit command)	9.34,6
(R1 x y) (edit command)	9.39
(S var . @) (edit command)	9.23
SAVE (edit command)	9.47,49,62
search algorithm (in editor)	9.16-17
SETTERMCHARS[NEXTCHAR;BKCHAR;LASTCHAR; UNQUOTECHAR;2CHAR;PPCHAR]	9.14
(SHOW . x) (edit command)	9.44
single-stepping a program	9.42
spelling correction	9.56-57,59,61
spelling lists	9.56-57,59
(SPLITC x) (edit command)	9.51
STOP (edit command)	9.47,51,62
structure modification commands (in editor)	9.24-40
(SURROUND @ IN ...) (edit command)	9.31
(SW n m) (edit command)	9.39-40
(SWAP @1 @2) (edit command)	9.40
SWAPC (edit command)	9.51
terminal	9.41
TEST (edit command)	9.55
THRU (edit command)	9.35-38
time stamp package	9.59
TO (edit command)	9.35-38
TTY: (edit command)	9.49,44,47
TTY: (printed by editor)	9.49
UNBLOCK (edit command)	9.55
UNDO (edit command)	9.54,7
undoing (in editor)	9.54,7,24,55
UNDOLST (editor variable/parameter)	9.54,47,55,63
UNDONE (printed by editor)	9.54
UNFIND (editor variable/parameter)	9.23,17,27-28,30-33, 47-48,53,63
UP (edit command)	9.10,9,11,17,28
UPFINDFLG (editor variable/parameter)	9.29,17-18
USERMACROS (editor variable/parameter)	9.46
USERMACROS (file package command)	9.46,56

Page
Numbers

USERWORDS (dwim variable/parameter)	9.59,61
U-CASE[X]	9.50
whereis package	9.64
WITH (in REPLACE command) (in editor)	9.27
WITH (in SURROUND command) (in editor)	9.31
(XTR . @) (edit command)	9.29
(change character)	9.25
!E (edit command)	9.52
!F (edit command)	9.52
!N (edit command)	9.52
!NX (edit command)	9.13
!UNDO (edit command)	9.55
!O (edit command)	9.12
##[COMS] NL*	9.42,19
## (in INSERT, REPLACE, and CHANGE commands)	9.28
\$ (<esc>) (in edit pattern)	9.8,15
\$ (<esc>) (in R command) (in editor)	9.38
\$ (<esc>, in R command) (in editor)	9.39
\$BUFS (<esc>BUFS) (prog. asst. command)	9.5
\$\$ (two <esc>s) (in edit pattern)	9.15
& (in edit pattern)	9.8,14
& (printed by editor)	9.2
* (in MBD command) (in editor)	9.30
* (printed by editor)	9.1
(* . x) (edit command)	9.53
ANY (in edit pattern)	9.14
COMMENT (printed by editor)	9.40
COMMENTFLG (prettyprint variable/parameter)	9.41
(-n e1 ... em) (n a number, edit command)	9.24,4
-n (n a number, edit command)	9.11,2
-- (in edit pattern)	9.8,15
-- (printed by editor)	9.2
-> (printed by editor)	9.39
.. (edit command)	9.21
.. (in edit pattern)	9.15
... (in edit pattern)	9.15-16
... (printed by editor)	9.9-10
0 (edit command)	9.12,3
(2ND . @) (edit command)	9.20
(3RD . @) (edit command)	9.20
8 (instead of left parenthesis)	9.57
(: e1 ... em) (edit command)	9.26,9
= (printed by editor)	9.8
=E (printed by editor)	9.57
=EDITF (printed by editor)	9.61
=EDITP (printed by editor)	9.59
=EDITV (printed by editor)	9.59
== (in edit pattern)	9.15
? (edit command)	9.40,2
? (printed by editor)	9.2
?= (edit command)	9.41
@ (location specification) (in editor)	9.19
(@1 THRU) (edit command)	9.37
(@1 THRU @2) (edit command)	9.35
(@1 TO) (edit command)	9.37
(@1 TO @2) (edit command)	9.35
(\ atom) (edit command)	9.22

**Page
Numbers**

\ (edit command)	9.23,8,27
\P (edit command)	9.23,8,41
↑ (edit command)	9.12,3
(← pattern) (edit command)	9.20
← (edit command)	9.22
↔ (edit command)	9.22

SECTION 10

ATOM, STRING, ARRAY, AND STORAGE MANIPULATION

10.1 P NAMES AND ATOM MANIPULATION

The term "print name" (of an atom) in LISP 1.5 referred to the characters that were output whenever the atom was printed. Since these characters were stored on the atom's property list under the property **PNAME**, pname was used interchangeably with "print name". In Interlisp, all pointers have pnames, although only literal atoms and strings have their pname explicitly stored.

The pname of a pointer are those characters that are output when the pointer is printed using prin1.¹

e.g., the pname of the atom ABC%(D²) consists of the five characters ABC(D. The pname of the list (A B C) consists of the seven characters (A B C) (two of the characters are spaces).

Sometimes we will have occasion to refer to the prin2-pname.

The prin2-pname are those characters output when the corresponding pointer is printed using prin2.

Thus the prin2-pname of the atom ABC%(D is the six characters ABC%(D.³

1 except that for the purposes of the functions described in this chapter, i.e., unpack, nchars, etc. the prin1-pname of an integer is defined as though radix=10. Note that integers will still be *printed* by prin1 using the current radix, as described in Section 14. However, we want pack[unpack[X9]] to always be X9 (and not sometimes X11) regardless of the setting of the radix. The user can force the prin1-pname of an integer to use current radix by setting the variable prxflg to T (initially NIL), e.g. with radix 8 and prxflg=NIL, nchars[9]=1, but with prxflg=T, nchars[9]=2, (since 9 would now be printed by prin1 as 11). Note that with radix[8] and either setting of prxflg, nchars[9 T], meaning use prin2-pname, would return 3, since 9 would print as 11Q.

+ + + +

2 % is the escape character. See Sections 2 and 14.

3 Note that the prin2-pname also depends on what readable is being used (see Section 14), since this determines where %'s will be inserted. Note also that the prin2-pname of an integer depends on the setting of radix.

pack[x]

If x is a list of atoms, the value of pack is a single atom whose pname is the concatenation of the pnames of the atoms in x, e.g., pack[(A BC DEF G)]=ABCDEFG. If the pname of the value of pack[x] is the same as that of a number, pack[x] will be that number, e.g., pack[(1 3.4)]=13.4, pack[(1 E -2)]=.01.

Although x is usually a list of atoms, it can be a list of arbitrary Interlisp pointers. The value of pack is still a single atom whose pname is the same as the concatenation of the pnames of all the pointers in x, e.g., pack[((A B)"CD")) = %(A% B%)CD.

In other words, mapc[x;prinl] and prinl[pack[x]] always produce the same output.⁴ In fact, pack actually operates by calling prinl to convert the pointers to a stream of characters (without printing) and then makes an atom out of the result. If x is not a list or NIL, pack generates an error, ILLEGAL ARG.

Note: In Interlisp-10, atoms are restricted to < 127 characters. Attempting to create a larger atom either via pack or by typing one in (or reading from a file) will cause an error, ATOM TOO LONG.

+ pack*[x]
+

lambda, nospread version of pack, e.g. (PACK*'< DIR '>) is equivalent to (PACK (LIST '< DIR '>)) but avoids the conses.

unpack[x;flg;rdtbl]

The value of unpack is the pname of x as a list of characters (atoms),⁵ e.g.,

$$\text{unpack}[ABC] = (A B C)$$

$$\text{unpack}["ABC(D)"] = (A B C %(D)$$

In other words prinl[x] and mapc[unpack[x];prinl] produce the same output.

If flg=T, the prin2-pname of x is used, (and computed with respect to rdtbl) e.g., unpack["ABC(D);T"]=(%" A B C %(D %").

Note: unpack[x] performs n conses, where n is the number of characters in the pname of x.

⁴ * Except for integers when radix is other than 10, e.g., mapc[(X 9);PRIN1] produces X11 when radix is 8, but pack[(X 110)]=X9. (See footnote 1.) Also, mapc[(0 5 1);PRIN1] and PRIN1[pack[(0 5 1)]] produce different results because pack[(0 5 1)] is 51.

⁵ * There are no special "character-atoms" in Interlisp, i.e., an atom consisting of a single character is the same as any other atom.

dunpack[x;scratchlist;flg;rdtbl] a destructive version of unpack that does not perform any conses but instead uses scratchlist to make a list equal to unpack[x;flg]. If the p-name is too long to fit in scratchlist, dunpack calls unpack and returns unpack[x;flg]. Gives an error if scratchlist is not a list.

nchars[x;flg;rdtbl] number of characters in pname of x.⁶ If flg=T, the prin2-pname is used. E.g. nchars["ABC"]=3, nchars["ABC";T]=5.

nthchar[x;n;flg;rdtbl] Value is nth character of pname of x. Equivalent to car[nth[unpack[x;flg];n]] but faster and does no conses. n can be negative, in which case counts from end of pname, e.g., -1 refers to the last character, -2 next to last, etc. If n is greater than the number of characters in the pname, or less than minus that number, or 0, the value of nthchar is NIL.

packc[x] like pack except x is a list of character codes,⁷ e.g., packc[(70 79 79)]=FOO.

chcon[x;flg;rdtbl] like unpack, except returns the pname of x as a list of character codes, e.g., chcon[FOO]=(70 79 79). If flg=T, the prin2-pname is used.

chconl[x] returns character code of first character of pname of x, e.g., chconl[FOO]=70. Thus chcon[x] could be written as mapcar[unpack[x];chconl].

dchcon[x;scratchlist;flg;rdtbl] similar to dunpack

character[n] n is a character code. Value is the atom having the corresponding single character as its pname,⁸ e.g., character[70]=F. Thus, unpack[x] could be written as mapcar[chcon[x];character].

fcharacter[n] fast version of character that compiles open.

gensym[char] Generates a new atom of the form xnnnn, where x=char (or A if

⁶ Both nthchar and nchars work much faster on objects that actually have an internal representation of their pname, i.e., literal atoms and strings, than they do on numbers and lists, as they do not have to simulate printing.

⁷ Interlisp-10 uses ASCII code.

⁸ See footnote 2.

char is NIL) in which each of the n's is a digit. Thus, the first one generated is A0001, the second A0002, etc. gensym provides a way of generating new atoms for various uses within the system. The value of gennum, initially 10000, determines the next gensym, e.g., if gennum is set to 10023, gensym[] = A0024.

The term gensym is used to indicate an atom that was produced by the function gensym. Atoms generated by gensym are the same as any other literal atoms: they have property lists, and can be given function definitions. Note that the atoms are not guaranteed to be new.

For example, if the user has previously created A0012, either by typing it in, or via pack or gensym itself, when gennum gets to 10011, the next value returned by gensym will be the A0012 already in existence.

mapatoms[fn]

Applies fn to every literal atom in the system, e.g., mapatoms[(LAMBDA(X) (AND (SUBRP X) (PRINT X)))] will print every subr. Value of mapatoms is NIL.

l-case[x;flg]

value is lower case version of x. If flg is T, the first letter is capitalized, e.g., l-case[FOO;T] = Foo, l-case[FOO] = foo. If x is a string, the value of l-case is also a string, e.g., l-case["FILE NOT FOUND";T] = "File not found". If x is a list, result is a new list in which l-case is computed for each corresponding element and non-NIL tail of the original list.

u-case[x]

Similar to l-case

+ **u-casep[x]**
+
+

Value is T, if x contains no lower case letters, e.g. u-casep[\$FOO]=T, u-casep[foo]=NIL. u-casep is handcoded and very efficient.

10.2 STRING FUNCTIONS

stringp[x]

Is x if x a string, NIL otherwise. Note: if x is a string, nlistp[x] is T, but atom[x] is NIL.

strequal[x;y]

Is x if x and y are both strings and equal, i.e., print the same, otherwise NIL. Equal uses strcqual. Note that strings may be equal without being eq.

mkstring[x]

Value is string corresponding to prin1-pname of x.

rstring[-;-]

Reads a string - see Section 14.

substring[x;n;m;oldptr]	<p>Value is the substring of <u>x</u> consisting of the <u>n</u>th through <u>m</u>th characters of <u>x</u>. If <u>m</u> is NIL, the substring is the <u>n</u>th character of <u>x</u> thru the end of <u>x</u>. <u>n</u> and <u>m</u> can be negative numbers, as with <u>nthchar</u>. Returns NIL if the substring is not well defined, e.g., <u>n</u> or <u>m</u> > nchars[x] or < minus[nchars[x]] or <u>n</u> corresponds to a character in <u>x</u> to the right of the character indicated by <u>m</u>.</p> <p>If <u>x</u> is not a string, equivalent to substring[mkstring[x];n;m], except <u>substring</u> does not have to actually make the string if <u>x</u> is a literal atom.⁹ For example, substring[(A B C);4;6] = "B C". If <u>oldptr</u> is a string pointer, it is reused, otherwise <u>substring</u> makes a new string pointer.</p>
gnc[x]	<p>get next character of string <u>x</u>. Returns the next character of the string, (as an atom), and removes the character from the string. Returns NIL if <u>x</u> is the null string. If <u>x</u> isn't a string, a string is made. Used for sequential access to characters of a string.</p> <p>Note that if <u>x</u> is a substring of <u>y</u>, gnc[x] does not remove the character from <u>y</u>, i.e., <u>gnc</u> doesn't physically change the string of characters, just the pointer and the byte count.¹⁰</p>
glc[x]	<p>gets last character of string <u>x</u>. Above remarks about <u>gnc</u> also supply to <u>glc</u>.</p>
concat[x ₁ ;x ₂ ;...;x _n]	<p><u>lambda</u> nospread function. Concatenates (copies of) any number of strings. The arguments are transformed to strings if they aren't strings. Value is the new string, e.g., concat["ABC";DEF;"GHI"] = "ABCDEFGH". The value of concat[] is the null string, "".</p>
rplstring[x;n;y]	<p>Replace characters of string <u>x</u> beginning at character <u>n</u> with string <u>y</u>. <u>n</u> may be positive or negative. <u>x</u> and <u>y</u> are converted to strings if they aren't already. Characters are smashed into (converted) <u>x</u>. Returns new <u>x</u>. Error if there is not enough room in <u>x</u> for <u>y</u>, i.e., the new string would be longer than the original.¹¹ Note that if <u>x</u> is a substring of <u>z</u>, <u>z</u> will also be modified by the action of <u>rplstring</u>.</p>
mkatom[x]	<p>Creates an atom whose pname is the same as that of the string <u>x</u> or</p>

⁹ See string storage section that follows.

¹⁰ See string storage section that follows.

¹¹ If y was not a string, x will already have been partially modified since rplstring does not know whether y will "fit" without actually attempting the transfer.

if x isn't a string, the same as that of `mkstring[x]`, e.g., `mkatom[(A B C)]` is the atom `%(A% B% C%)`. In Interlisp-10, if the atom would have > 126 characters, causes an error, ATOM TOO LONG.

- + `subatom[x;n;m]` equivalent to `mkatom[substring[x;n;m]]`, but does not make a string pointer.
- +

SEARCHING STRINGS

strpos is a function for searching one string looking for another. Roughly it corresponds to member, except that it returns a character position number instead of a tail. This number can then be given to substring or utilized in other calls to strpos.

`strpos[pat;string;start;skip;anchor;tail]`

pat and string are both strings (or else they are converted automatically). Searches string beginning at character number start, (or else 1 if start is NIL) and looks for a sequence of characters equal to pat. If a match is found, the corresponding character position is returned, otherwise NIL, e.g.,

`strpos["ABC", "XYZABCDEF"] = 4`

`strpos["ABC", "XYZABCDEF"; 5] = NIL`

`strpos["ABC", "XYZABCDEFABC"; 5] = 10`

skip can be used to specify a character in pat that matches any character in string, e.g.,

`strpos["A&C&"; "XYZABCDEF"; NIL; &] = 4`

If anchor is T, strpos compares pat with the characters beginning at position start, or 1. If that comparison fails, strpos returns NIL without searching any further down string. Thus it can be used to compare one string with some *portion* of another string, e.g.,

`strpos["ABC"; "XYZABCDEF"; NIL; NIL; T] = NIL`

`strpos["ABC"; "XYZABCDEF"; 4; NIL; T] = 4`

Finally, if tail is T, the value returned by strpos if successful is not the starting position of the sequence of characters corresponding to pat, but the position of the first character after that, i.e., starting point plus `nchars[pat]` e.g.,
`strpos["ABC"; "XYZABCDEFABC"; NIL; NIL; NIL; T] = 7`. Note that `strpos["A"; "A"; NIL; NIL; NIL; T] = 2`, even though "A" has only one character.

EXAMPLE PROBLEM

Given the strings x, y, and z, write a function foo that will make a string corresponding to that portion of x between y and z, e.g., `foo["NOW IS THE TIME FOR ALL GOOD MEN"; "IS"; "FOR"]` is " THE TIME ".

Solution:

```
(FOO
  [LAMBDA (X Y Z)
    (AND (SETQ Y (STRPOS Y X NIL NIL NIL T))
         (SETQ Z (STRPOS Z X Y)))
    (SUBSTRING X Y (SUB1 Z))]
```

strpos[a;str;start;neg]

str is a string (or else it is converted automatically to a string), a is a list of characters or character codes.¹² strpos searches str beginning at character number start (or else 1 if start=NIL) for one of the characters in a. If one is found, strpos returns as its value the corresponding character position, otherwise NIL. E.g., strpos[(A B C); "XYZBCD"]=4. If neg=T, strpos searches for a character *not* on a, e.g., strpos[(A B C); "ABCDEF"; NIL; T]=4.

If a is an array, it is treated as a bit table (see discussion of makebittable below)

If a is not a bit table (array), strpos first converts it to a bit table using makebittable described below. If strpos is to be called frequently with the same list of characters, a considerable savings can be achieved by converting the list to a bit table *once*, and then passing the bit table to strpos as its first argument.

makebittable[l;neg;a]

makes a bit table suitable for use by strpos. l is a list of characters or character codes, neg is the same as described for strpos. If a is not a suitable array, makebittable will create an array and return that as its value. Otherwise it uses (and changes) a.

Note: if neg=T, strpos must call makebittable whether a is a list *or* an array. To obtain bit table efficiency with neg=T, makebittable should be called with neg=T, to construct the "inverted" table, and the resulting table (array) should be given to strpos with neg=NIL.

STRING STORAGE

A string is stored in 2 parts: the characters of the string, and a pointer to the characters. The pointer, or "string pointer", indicates the byte at which the string begins and the length of the string. It occupies one word of storage. In Interlisp-10, the characters of the string are stored five characters to a word in a portion of the address space devoted exclusively to storing characters.

Since the internal pname of literal atoms also consists of a pointer to the beginning of a string of characters and a byte count, conversion between literal atoms and strings does not require any additional storage for the characters of the pname, although one cell is required for the string

¹² If any element of a is a number, it is assumed to be a character code. Otherwise, it is converted to a character code via chcon1. Therefore, it is more efficient to call strpos with a a list of character *codes*.

pointer.¹³

When the conversion is done internally, e.g., as in substring, strpos, or strposl, no additional storage is required for using literal atoms instead of strings.

The use of storage by the basic string functions is given below:

<code>mkstring[x]</code>	<code>x</code> string <code>x</code> literal atom other	no space new pointer new characters and pointer
<code>substring[x;n;m]</code>	<code>x</code> string <code>x</code> literal atom other	new pointer ¹⁴ new pointer new characters and pointer
<code>gnc[x]</code> and <code>glc[x]</code>	<code>x</code> string other	no space, pointer is modified like <u>mkstring</u> , but doesn't make much sense
<code>concat[x₁;x₂;...x_n]</code>	args any type	new characters for whole new string, one new pointer
<code>rplstring[x;n;y]</code>	<code>x</code> string <code>x</code> other <code>y</code> any type	no new space unless characters are in <u>pname</u> space (as result of <u>mkstring[atom]</u>) in which case <u>x</u> is quietly copied to string space new pointer and characters type of <u>y</u> doesn't matter

10.3 ARRAY FUNCTIONS

Space for arrays and compiled code are both allocated out of a common array space. Arrays of pointers and unboxed numbers may be manipulated by the following functions:

`array[n;p;v]`

This function allocates a block of $n+2$ words, of which the first two are header information. In Interlisp-10, the next $p \leq n$ are cells which will contain unboxed numbers, and are initialized to unboxed 0. The last $n-p > 0$ cells will contain pointers initialized with v, i.e., both car and cdr are available for storing information, and each initially contain v. If p is NIL, 0 is used (i.e., an array containing all Interlisp pointers). The value of array is the array, also called an array pointer. If sufficient space is not available for the array, a garbage collection of array space is initiated. If this is unsuccessful in obtaining sufficient space, an error is generated, **ARRAYS FULL**.

¹³ Except when the string is to be smashed by rplstring. In this case, its characters must be copied to avoid smashing the pname of the atom. rplstring automatically performs this operation.

¹⁴ except when substring is given a string pointer to reuse as its fourth argument. Note that substring[string;1;-1] copies just the string *pointer* without copying the characters.

Array-pointers print as #n, where n is the octal representation of the pointer. Note that #n will be read as a literal atom, and not an array pointer.

arraysize[a]	Value is the size of array <u>a</u> . Generates an error, ARG NOT ARRAY, if <u>a</u> is not an array.	
arraytyp[a]	Returns a value corresponding to second argument to <u>array</u> . In Interlisp-10, this is the number of unboxed array words of array <u>a</u> .	+ +
arrayp[x]	Value is <u>x</u> if <u>x</u> is an array pointer otherwise NIL. No check is made to ensure that <u>x</u> actually addresses the <i>beginning</i> of an array.	
arraybeg[a]	if <u>a</u> is a pointer into the middle of an array, returns the pointer to its beginning. Otherwise returns NIL.	+ +
harray[n]	Creates hash array of size <u>n</u> (see Section 7).	+ +
harraysize[a]	Value is the size of hash array <u>a</u> .	+ +
harrayp[x]	Value is <u>x</u> if <u>x</u> is a hash array (see Section 7).	+ +
swparrayp[x]	Value is <u>x</u> if <u>x</u> is a swappable array, NIL otherwise.	
elt[a;n]	Value is <u>n</u> th element of the array <u>a</u> . ¹⁵ <u>elt</u> generates an error, ARG NOT ARRAY, if <u>a</u> is not the beginning of an array. ¹⁶ If <u>n</u> corresponds to the <i>unboxed</i> region of <u>a</u> , the value of <u>elt</u> is the full 36 bit word, as a boxed integer. If <u>n</u> corresponds to the <i>pointer</i> region of <u>a</u> , the value of <u>elt</u> is the <u>car</u> half of the corresponding element.	
seta[a;n;y]	sets the <u>n</u> th element of the array <u>a</u> . Generates an error, ARG NOT ARRAY, if <u>a</u> is not the beginning of an array. If <u>n</u> corresponds to the <i>unboxed</i> region of <u>a</u> , <u>y</u> must be a number, and is unboxed and stored as a full 36 bit word into the <u>n</u> th element of <u>a</u> . If <u>n</u> corresponds to the <i>pointer</i> region of <u>a</u> , <u>y</u> replaces the <u>car</u> half of the <u>n</u> th element. The value of <u>seta</u> is <u>y</u> .	

¹⁵ elt[a;1] is the first element of the array (actually corresponds to the 3rd cell because of the 2 word header).

¹⁶ arrayp is true for pointers into the middle of arrays, but elt and seta must be given a pointer to the beginning of an array, i.e., a value of array.

Note that seta and elt are always inverse operations.

cltd[a;n] same as clt for unboxed region of a, but returns cdr half of nth element, if n corresponds to the pointer region of a.

setd[a;n;v] same as seta for unboxed region of a, but sets cdr half of nth element, if n corresponds to the pointer region of a. The value of setd is y.

In other words, eltd and setd are always inverse operations.

- + copyarray[ar] Creates a new array of same size and type as ar, i.e. same distribution of pointers and unboxed numbers, and with the same contents as ar. Value is new array. Generates an ARG NOT ARRAY error, if ar is not an array.

10.4 STORAGE FUNCTIONS

- * The table below contains the built in Interlisp data types and their corresponding type number and type name. User data types are assigned type numbers beginning with 31.

<u>* type number</u>	<u>type name</u>	<u>description</u>
* 1	ARRAYP	arrays, compiled code
* 2	STACK	machine code
* 4	SWPARRAYP	swapped array handles
* 5	STACKP	stack pointers
* 6	GC.BTAB	bit tables
* 7	ATOM.HASH	atom hash table ¹⁷
* 8	LISTP	list words
* 9	VCELLP	value cells
* 12	ATOM	atoms
* 16	FLOATP	floating point numbers
* 18	FIXP	large integers
* 20	SMALLP	small integers
* 24	STRINGP	string pointers
* 28	ATOM.CHARS	pname storage
* 30	STRING.CHARS	string storage

- + typename[datum] Value is type name for the data type of datum. For user data types (Section 3), the type name is specified by the call to declaredatatype. For built in data types, typename is one of the atoms, LISTP, FLOATP, FIXP, STRINGP, LITATOM,

+ ¹⁷ The atom hash table automatically expands by a specified number of pages each time it fills up. The number of pages is set via the function minhash. The initial setting is minhash[2] (room for 1024 new atoms).

	STACKP, ARRAYP, ¹⁸ SWPARRAYP, SMALLP.	+
typenamep[datum;typename]	True if the typename of <u>datum</u> is equal to <u>typename</u> . Compiles open.	+
typenamefromnumber[n]	value is type name for type number <u>n</u> , or NIL if <u>n</u> is not a valid type number, e.g. typenamefromnumber[30]=STRING.CHARS.	+
typenumberfromname[name]	value is corresponding type number for <u>name</u> , or NIL if <u>name</u> is not a type name, e.g. typenumberfromname[STRING.CHARS]=30. ¹⁹	+
ntyp[datum]	Value is type number for the data type of <u>datum</u> , e.g., ntyp[(A . B)] is 8, the type number for lists.	*
typep[datum;n]	value is T, if the type number of <u>datum</u> is equal to <u>n</u> .	*
reclaim[type]	Initiates a garbage collection of type <u>type</u> , where <u>type</u> is either a type name or type number. Value of <u>reclaim</u> is number of words available (for that type) after the collection.	*
<i>Garbage collections, whether invoked directly by the user or indirectly by need for storage, do not confine their activity solely to the data type for which they were called, but automatically collect some or all of the other types (see Section 3).</i>		*
gchgag[message]	affects messages printed by garbage collector. If <u>message</u> =T, whenever a garbage collection is begun, "collecting" is printed, followed by the type description of the type that initiated the collection. ²⁰ When the garbage collection is complete, two numbers are printed: the number of words collected for that type, and the total number of words available for that type, i.e., allocated but not necessarily currently in use. Note that other types may also have been collected, and had more storage assigned, as explained in Section 3.	*

¹⁸ typename distinguishes the five logical data types, READTABLEP, TERMTABLEP, CCODEP, ARRAYP and POINTER.INTO.ARRAY.SPACE, though they all have the same type number and share the same data space.

¹⁹ typenumberfromname will accept READTABLEP, TERMTABLEP, CCODEP, and ARRAYP, and return the same value for each, which for Interlisp-10 is 1. Note however that typenamefromnumber[1]=ARRAYP.

²⁰ Note that this type description can be set via the function settypedescription described below.

Example:

```
←RECLAIM(18)

collecting large numbers
511, 3071 free cells
3071
←RECLAIM(LITATOM)

collecting atoms
1020, 1020 free cells
1020
```

If message=NIL, no garbage collection message is printed, either on entering or leaving the garbage collector.

If message is a list, car of message is printed (using prinl) when the garbage collection is begun, and cdr is printed (using prinl) when the collection is finished. If message is a literal atom or string, message is printed when the garbage collection is begun, and nothing is printed when the collection finishes.

If message is a number, the message is the same as for gchgag[T], except if the total number of free pages left after the collection is less than message, the number of free pages is printed, e.g.,

```
←GCGAG(100)
T
←RECLAIM()

collecting lists
10369, 10369 free cells, 87 pages left.
```

The initial setting for gchgag is 40.

The value of gchgag is its previous setting.

+ gcmess[message#;string]

+

+

+

+

+

gchgag is implemented in terms of the primitive gcmess which can be used to further refine garbage collection messages for specialized applications. The garbage collection message is actually composed of seven separate messages:

1	2					
collecting large numbers						
3	4	5	6	7		
511, 3071 free cells, 87 pages left						

+

+

+

message#1 is the "collecting" string. If NIL, then neither it, nor the type dependent field (which is scittypcdescription described below) is printed.

+

+

message#2 is the carriage-return after the type-dependent field. Thus to simply print a string at the beginning of a garbage collection, perform gcmess[1] and gcmess[2;string].

message#3 is the "," which comes after the number of cells actually collected. If NIL, then neither it nor that number are printed.	++ ++ ++
message#4 is the "free cells" which comes after the number of cells that are now allocated. If NIL, neither it nor that number are printed.	++ ++ ++
message#5 is the number of pages left below which the system prints message#6.	++ ++
message#6 is the "pages left" message. If NIL, neither it nor the number of pages left are printed.	++ ++
message#7 is the terminating carriage return.	+
 gettypedescription[type]	 +
returns the type description string for <u>type</u> , a type name or type number.	++
 settypedescription[type;string]	 +
sets the type description string for <u>type</u> to be <u>string</u> . The type description is used in garbage collection messages and by <u>storage</u> (described below).	++ ++
 minfs[n;type]	 +
Sets the minimum amount of free storage which will be maintained by the garbage collector for data types of type number or type name <u>type</u> . If, after any garbage collection for that type, fewer than <u>n</u> free words are present, sufficient storage will be added (in 512 word chunks) to raise the level to <u>n</u> .	++
If <u>typ=NIL</u> , LISTP is used, i.e., the <u>minfs</u> refers to list words.	++
If <u>n=NIL</u> , <u>minfs</u> returns the current <u>minfs</u> setting for the corresponding type.	++
A <u>minfs</u> setting can also be changed dynamically, even during a garbage collection, by typing control-S ²¹ followed by a number, followed by a period. ²² If the control-S was typed during a garbage collection, the number is the new <u>minfs</u> setting for the type being collected, otherwise for type 8, i.e., list words.	++
 <i>Note: A garbage collection of a "related" type may also cause more storage to be assigned to that type. See discussion of garbage collector algorithm, Section 3.</i>	 +

21 control-X for Interlisp-10 on TOPS-20.

22 When the control-S is typed, Interlisp immediately clears and saves the input buffer, rings the bell, and waits for input, which is terminated by any non-number. The input buffer is then restored, and the program continues. If the input was terminated by other than a period, it is ignored.

`storage[flg;gcflg]` Prints amount of storage used by and assigned to the user, e.g.,

<code>←STORAGE()</code>		<code>USED</code>	<code>ASSIGNED</code>
<code>TYPE</code>			
<code>ARRAYP</code>	<code>arrays</code>	12754	12800
<code>STACK</code>	<code>stack, swap buffer</code>	14336	14336
<code>SWPARRAY</code>	<code>swap array handles</code>	29	512
<code>STACKP</code>	<code>stack pointers</code>	0	512
<code>GC.BTAB</code>	<code>gc bittable</code>	2048	2048
<code>LISTP</code>	<code>lists</code>	5016	11776
<code>VCELLP</code>	<code>value cells</code>	194	1024
<code>LITATOM</code>	<code>atoms</code>	2028	3072
<code>FLOATP</code>	<code>float numbers</code>	1	512
<code>FIXP</code>	<code>large numbers</code>	70	2048
<code>STRINGP</code>	<code>string pointers</code>	173	512
<code>ATOM.CHARS</code>	<code>atoms have characters</code>	1594	2048
<code>STRING.CHARS</code>	<code>string characters</code>	421	1024
<code>SUM</code>	<code>(137 pages left)</code>	38664	52224

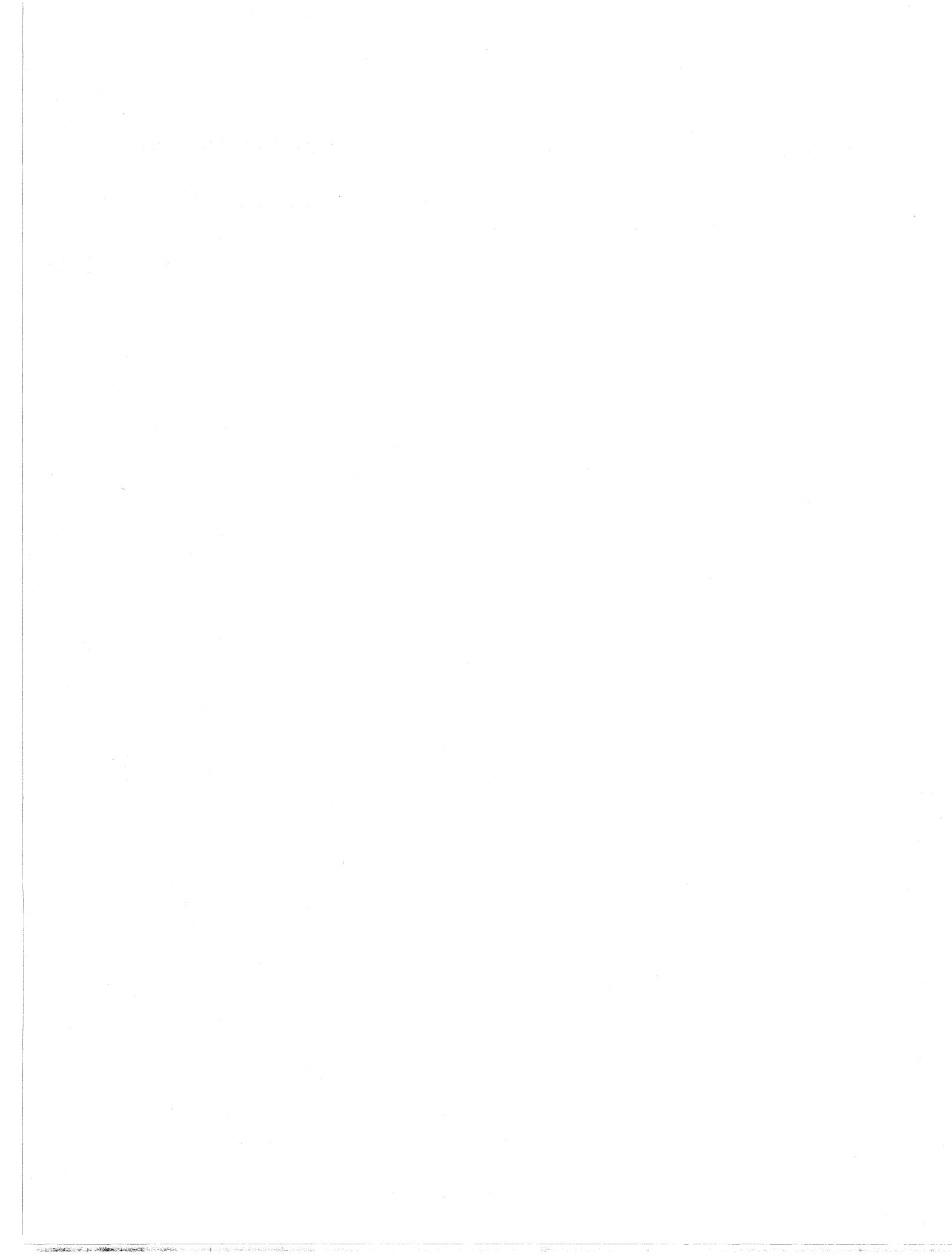
+ Note that the storage used by a particular type is only accurate immediately following a garbage collection of a related type. If `gcflg=T`, `storage` will perform the necessary garbage collections before printing its results. If `flg=T`, includes storage used by and assigned to the system. Value is `NIL`.

- + `gctrp[n]` garbage collection trap. Causes a (simulated) control-H interrupt when the number of free list words remaining equals `n`, i.e., when a garbage collection would occur in `n` more conses. The message `GCTR` is printed, the function `interrupt` (Section 16) is called, and a break occurs. Note that by advising (Section 19) `interrupt` the user can program the handling of a `gctrp` instead of going into a break.²³
- + Value of `gctrp` is its last setting.
- + `gctrp[-1]` will "disable" a previous `gctrp` since there are never -1 free list words. `gctrp` is initialized this way.
- + `gctrp[]` returns number of list words left, i.e., number of conses until next type LISTP garbage collection, see Section 21.
- + `conscount[n]` `conscount[]` returns number of conses since Interlisp started up. If `n` is not `NIL`, resets `conscount` to `n`.

²³ For `gctrp` interrupts, `interrupt` is called with `intype` (its third argument) equal to 3. If the user does not want to go into a break, the advice should still allow `interrupt` to be entered, but first set `intype` to -1. This will cause `interrupt` to "quietly" go away by calling the function that was interrupted. The advice should *not* exit `interrupt` via `return`, as in this case the function that was about to be called when the interrupt occurred would not be called.

Storage Functions

closer[a;x]	Stores <u>x</u> into memory location <u>a</u> . Both <u>x</u> and <u>a</u> must be numbers.	+
openr[a]	Value is the number in memory location <u>a</u> , i.e., boxed.	+



Index for Section 10

	Page Numbers
ARG NOT ARRAY (error message)	10.9-10
ARRAY[N;P;V] SUBR	10.8
array functions	10.8-10
array header	10.8
ARRAYBEG[A] SUBR	10.9
ARRAYP[X] SUBR	10.9
ARRAYS FULL (error message)	10.8
ARRAYSIZE[A]	10.9
ARRAYTYP[ARRAY]	10.9
atom hash table	10.10
ATOM TOO LONG (error message)	10.2,6
A000n (gensym)	10.4
bell (printed by system)	10.13
CHARACTER[N] SUBR	10.3
character atoms	10.2
character codes	10.3
CHCON[X;FLG;RDTBL] SUBR	10.3
CHCON1[X] SUBR	10.3
CLOSER[A;X] SUBR	10.15
collecting (printed by system)	10.11
compiled code	10.8
CONCAT[X1;X2;...;Xn] SUBR*	10.5,8
CONSCOUNT[N] SUBR	10.14
control-H	10.14
control-S	10.13
control-X (TOPS-20)	10.13
COPYARRAY[AR]	10.10
DCHCON[X;SCRATCHLIST;FLG;RDTBL]	10.3
DUNPACK[X;SCRATCHLIST;FLG;RDTBL]	10.3
ELT[A;N] SUBR	10.9
ELTD[A;N] SUBR	10.10
FCHARACTER[N] SUBR	10.3
garbage collection	10.8,11,13-14
garbage collection message	10.11-12
GCGAG[MESSAGE]	10.11
GCMESS[MESSAGE#;STRING] SUBR	10.12
GCTRIP[N] SUBR	10.14
GCTRIP (printed by system)	10.14
GENNUM (system variable/parameter)	10.4
GENSYM[CHAR]	10.3,4
GETTYPEDESCRIPTION[TYPE]	10.13
GLC[X] SUBR	10.5,8
GNC[X] SUBR	10.5,8
HARRAY[N] SUBR	10.9
HARRAYP[X]	10.9
HARRAYSIZE[HARRAY]	10.9
ILLEGAL ARG (error message)	10.2
input buffer	10.13
INTERRUPT[INTFN;INTARGS;INTYPE]	10.14
literal atoms	10.7
lower case	10.4
L-CASE[X;FLG]	10.4
MAKEBITTABLE[L;NEG;A]	10.7
MAPATOMS[FN] SUBR	10.4
MINFS[N;TYPE]	10.13
MKATOM[X] SUBR	10.5

Page
Numbers

MKSTRING[X] SUBR	10.4,8
NCHARS[X;FLG;RDTBL] SUBR	10.3
NTHCHAR[X;N;FLG;RDTBL] SUBR	10.3
NTYP[DATUM] SUBR	10.11
null string	10.5
OPENR[A] SUBR	10.15
PACK[X] SUBR	10.2
PACKC[X] SUBR	10.3
PACK*[X] SUBR*	10.2
pnames	10.1-3,7
print name	10.1
prin2-pnames	10.1-3
PRXFLG (system variable/parameter)	10.1
radix	10.1-2
RECLAIM[TYPE]	10.11
RPLSTRING[X;N;Y] SUBR	10.5,8
searching strings	10.6-7
SETA[A;N;V]	10.9
SETD[A;N;V]	10.10
SETTYPEDESCRIPTION[TYPE;STRING]	10.13
STORAGE[FLG;GCFLG]	10.14
STREQUAL[X;Y] SUBR	10.4
string characters	10.7
string functions	10.4-7
string pointers	10.5,7-8
string storage	10.7-8
STRINGP[X] SUBR	10.4
STRPOS[PAT;STRING;START;SKIP;ANCHOR;TAIL]	10.6
STRPOSL[A;STR;START;NEG]	10.7
SUBATOM[X;N;M]	10.6
SUBSTRING[X;N;M;OLDPTR] SUBR	10.5,8
SWPARRAYP[X] SUBR	10.9
type description	10.13
type names	10.10-11
type numbers	10.10-11
TYPENAME[DATUM] SUBR	10.10
TYPENAMEFROMNUMBER[N]	10.11
TYPENAMEP[DATUM;TYPENAME]	10.11
TYPENUMBERFROMNAME[NAME]	10.11
TYPEP[DATUM;N]	10.11
unboxed numbers (in arrays)	10.8
UNPACK[X;FLG;RDTBL] SUBR	10.2
U-CASE[X]	10.4
U-CASEP[X]	10.4
# (followed by a number)	10.9

SECTION 11

FUNCTIONS WITH FUNCTIONAL ARGUMENTS

As in all LISP 1.5 systems, arguments can be passed which can then be used as functions. However, since car of a form is *never* evaluated, apply or apply* must be used to call the function specified by the value of the functional argument.

Functions which use functional arguments should use variables with obscure names to avoid possible conflict with variables that are used by the functional argument. For example, all system functions standardly use variable names consisting of the function name concatenated with x or fn, e.g., mapx. Note that by specifying the free variables used in a functional argument as the second argument to function, thereby using the Interlisp FUNARG feature, the user can be sure of no clash.

function[fn;env]

is an nlambda function. If env=NIL, the value of function is identical to quote, for example,

(MAPC LST (FUNCTION PRINT)) will cause mapc to be called with two arguments, the value of lst and PRINT. Similarly, [MAPCAR LST (FUNCTION (LAMBDA(Z) (LIST (CAR Z))) will cause mapcar to be called with the value of lst and (LAMBDA (Z) (LIST (CAR Z))). When compiled, function will cause code to be compiled for fn; quote will not. Thus,

(MAPCAR LST (QUOTE (LAMBDA --))) will cause mapcar to be called with the value of lst and the expression (LAMBDA --). The functional argument will therefore still be interpreted. The corresponding expression using function will cause a dummy function to be created with definition (LAMBDA --), and then compiled. mapcar would then be called with the value of lst and the name of the dummy function. See Section 18.

If env is not NIL, it can be a list of variables that are (presumably) used freely by fn. In this case, the value of function is an expression of the form (FUNARG fn pos), where pos is a stack pointer to a frame that contains the variable bindings for those variables on env. env can also be a stack pointer itself, in which case the value of function is (FUNARG fn env). Finally, env can be an atom, in which case it is evaluated, and the value interpreted as described above. Funarg is described on page 11.3-5.

map[mapx;mapfn1;mapfn2] If mapfn2 is NIL, map applies the function mapfn1 to successive tails of the list mapx. That is, first it computes mapfn1[mapx], and then mapfn1[cdr[mapx]], etc., until mapx is exhausted.¹ If mapfn2 is provided, mapfn2[mapx] is used instead of cdr[mapx] for the next call for mapfn1, e.g., if mapfn2 were cddr, alternate elements of the list would be skipped.

The value of map is NIL. map compiles open.

mapc[mapx;mapfn1;mapfn2] Identical to map, except that mapfn1[car[mapx]] is computed at each iteration instead of mapfn1[mapx], i.e., mapc works on elements, map on tails. The value of mapc is NIL. mapc compiles open.

maplist[mapx;mapfn1;mapfn2] successively computes the same values that map would compute, and returns a list consisting of those values. maplist compiles open.

mapcar[mapx;mapfn1;mapfn2] computes the same values that mapc would compute, and returns a list consisting of those values, e.g., mapcar[x;FNTYP] is a list of fntyps for each element on x. mapcar compiles open.

mapcon[mapx;mapfn1;mapfn2] computes the same values as map and maplist but nconcs these values to form a list which it returns. mapcon compiles open.

mapconc[mapx;mapfn1;mapfn2]

Computes the same values as mapc and mapcar, but nconcs the values to form a list which it returns. mapconc compiles open.

Note that mapcar creates a new list which is a mapping of the old list in that each element of the new list is the result of applying a function to the corresponding element on the original list. mapconc is used when there are a *variable* number of elements (including none) to be inserted at each iteration, e.g. mapconc[x;(LAMBDA (Y) (AND Y (LIST Y))) will make a list consisting of x with all NILs removed, mapconc[x;(LAMBDA (Y) (AND (LISTP Y) Y))] will make a linear list consisting of all the lists on x, e.g., if applied to ((A B) C (D E F) (G) H I) will yield (A B D E F G).²

subset[mapx;mapfn1;mapfn2] applies mapfn1 to elements of mapx and returns a list of those elements for which this application is non-NIL, e.g.,

¹ i.e., becomes a non-list.

² Note that since mapconc uses nconc to string the corresponding lists together, in this example, the original list will be clobbered, i.e., it would now be ((A B D E F G) C (D E F G) (G) H I). If this is an undesirable side effect, the functional argument to mapconc should return instead a top level copy, e.g., in this case, use (AND (LISTP Y) (APPEND Y))).

`subset[(A B C D) ; NUMBERP] = (C D).`

mapfn2 plays the same role as with map, mapc, et al. subset compiles open.

`map2c[mapx;mapy;mapfn1;mapfn2]`

Identical to mapc except mapfn1 is a function of two arguments, and mapfn1[car[mapx];car[mapy]] is computed at each iteration.³ Terminates when either mapx or mapy are exhausted.

`map2car[mapx;mapy;mapfn1;mapfn2]`

Identical to mapcar except mapfn1 is a function of two arguments and mapfn1[car[mapx];car[mapy]] is used to assemble the new list. Terminates when either mapx or mapy is exhausted.

Note: CLISP (Section 23) provides a more general and complete facility for expressing iterative statements, e.g., (FOR X IN Y COLLECT (CADR X) WHEN (NUMBERP (CAR X)) UNTIL (NULL X)).

`mapprint[lst;file;left;right;sep;pfn;lispxpathflg]`

is a general printing function. It cycles through lst applying pfn (or prinl if pfn not given) to each element of lst. Between each application, mapprint performs prinl of sep, or " " if sep=NIL. If left is given, it is printed (using prinl) initially; if right is given it is printed (using prinl) at the end.

For example, mapprint[x;NIL;%(:%)] is equivalent to prinl for lists. To print a list with commas between each element and a final ":" one could use mapprint[x;T;NIL;%,:%].

If lispxpathflg=T, lispxpathflg is used for prinl (see Section 22).

`mapdl,searchpd1`

See Section 12.

`mapatoms`

See Section 5.

`every, some, notevery, notany` See Section 5.

FUNARG

function is a function of two arguments, fn, a function, and env is either NIL, a list of variables

³ mapfn2 is still a function of one argument, and is applied twice on each iteration; mapfn2[mapx] gives the new mapx, mapfn2[mapy] the new mapy. cdr is used if mapfn2 is not supplied, i.e., is NIL.

used freely by fn, a stack pointer, or an atom. If env is a list of variables, the value of function is an expression of the form (FUNARG fn pos), where pos is a stack pointer to a frame that contains the bindings of the variables on env at the time the call to function was evaluated. If env is a stack pointer, the value of function is (FUNARG fn env).⁴

funarg is not a function itself. Like LAMBDA and NLAMBDA, it has meaning and is specially recognized by Interlisp only in the context of applying a function to arguments. In other words, the expression (FUNARG fn pos) is used exactly like a function.⁵ When a funarg is applied or is car of a form being eval'ed, the apply or eval takes place in the access environment specified by env (see Section 12).

For example, suppose a program wished to compute (FOO X (FUNCTION FIE)), and fie used y and z as free variables. If foo rebound y and z, fie would obtain the rebound values when it was applied from inside of foo. By evaluating instead (FOO X (FUNCTION FIE (Y Z))), foo would be called with (FUNARG FIE pos) as its second argument, where pos contained the bindings of y and z (at the time foo was called). Thus when fie was applied from inside of foo, it would "see" the original values of y and z.

However, funarg is more than just a way of circumventing the clashing of variables. For example, a funarg expression can be returned as the value of a computation, and then used "higher up". Furthermore, if the function in a funarg expression *sets* any of the variables contained in the frame, only the frame would be changed. For example, suppose foo is defined as

```
(LAMBDA (LST FN) (PROG (Y Z) (SETQ Y &) (SETQ Z &)
... (MAPC LIST FN) ...))
```

and (FOO X (FUNCTION FIE (Y Z))) is evaluated. If one application of fie (by the mapc in foo) changes y and z, then the next application of fie will obtain the changed values of y and z resulting from the previous application of fie, since both applications of fie come from the exact same funarg object, and hence use the exact same frame. The bindings of y and z bound inside of foo, and the bindings of y and z *above foo* would not be affected. In other words, the variable bindings contained in pos are a part of the function object, i.e., the funarg carries its environment with it.

Thus by creating a funarg expression with function, a program can create a function object which has updateable binding(s) associated with the object which last *between* calls to it, but are only accessible through that instance of the function. For example, using the funarg device, a program could maintain two different instances of the same random number generator in different states, and run them independently.

EXAMPLE

If foo is defined as (LAMBDA (X) (COND ((ZEROP A) X) (T (MINUS X)))) and fie as (LAMBDA NIL (PROG (A) (SETQ A 2) (RETURN (FUNCTION FOO)))). then if we perform (SETQ A 0), (SETQ FUM (FIE)), the value of fum is FOO, and the value of (APPLY* FUM 3) is 3, because the value of A at the time foo is called is 0.

⁴ If env is NIL, the value of function is simply fn, i.e., not a funarg expression. If env is an atom, it is evaluated and its value interpreted as described above.

⁵ LAMBDA, NLAMBDA, and FUNARG expressions are sometimes called "function objects" to distinguish them from functions, i.e., literal atoms which have function definitions.

However if fie were defined instead as
(LAMBDA NIL (PROG (A) (SETQ A 2) (RETURN (FUNCTION FOO (A))))), the value of
fum would be (FUNARG FOO pos) and so the value of (APPLY* FUM 3) would be -3, because
the value of A seen by foo is the value A had when the funarg was created inside of fie, i.e., 2.

Index for Section 11

	Page Numbers
APPLY[FN;ARGS] SUBR	11.1
APPLY*[FN;ARG1;ARG2;...;ARGn] SUBR*	11.1
CLISP	11.3
FUNARG	11.3-4,1,5
FUNCTION[FN;ENV] NL	11.1,3-4
function objects	11.4
functional arguments	11.1
MAP[MAPX;MAPFN1;MAPFN2]	11.2
MAPC[MAPX;MAPFN1;MAPFN2]	11.2
MAPCAR[MAPX;MAPFN1;MAPFN2]	11.2
MAPCON[MAPX;MAPFN1;MAPFN2]	11.2
MAPCONC[MAPX;MAPFN1;MAPFN2]	11.2
MAPLIST[MAPX;MAPFN1;MAPFN2]	11.2
MAPPRINT[LST;FILE;LEFT;RIGHT;SEP;PFN;LSPXPRNTFLG]..	11.3
MAP2C[MAPX;MAPY;MAPFN1;MAPFN2]	11.3
MAP2CAR[MAPX;MAPY;MAPFN1;MAPFN2]	11.3
SUBSET[MAPX;MAPFN1;MAPFN2]	11.2
variable bindings	11.3-4



SECTION 12

VARIABLE BINDINGS, PUSH DOWN LIST FUNCTIONS, AND THE SPAGHETTI STACK

A number of schemes have been used in different implementations of LISP for storing the values of variables. These include:

1. Storing values on an association list paired with the variable names.
2. Storing values on the property list of the atom which is the name of the variable.
3. Storing values in a special value cell associated with the atom name, putting old values on a pushdown list, and restoring these values when exiting from a function.
4. Storing values on a pushdown list.

In Interlisp, we currently use the third scheme, so called "shallow binding". When a function is entered, the value of each variable bound by the function (function argument) is stored in a value cell associated with that variable name. The value that was in the value cell is stored in a block of storage called the basic frame for this function call. In addition, on exit from the function each variable must be individually unbound; that is, the old value saved in the basic frame must be restored to the value cell. Thus there is a higher cost for binding and unbinding a variable than in the fourth scheme, "deep binding". However, to find the current value of any variable, it is only necessary to access the variable's value cell, thus making variable reference considerably cheaper under shallow binding than under deep binding, especially for free variables. Our measurements have indicated that typically more time is spent in variable reference under the deep binding scheme than is lost in binding and unbinding under shallow binding. However, the shallow binding scheme used does require an additional overhead in switching contexts when doing "spaghetti stack" operations; this is described in more detail on page 12.6.

The basic frames are allocated on a stack or pushdown list; for most user purposes, these frames should be thought of as containing the variable names associated with the function call, and the *current* values for that frame. The descriptions of the stack functions in Sections 12.3 and 12.4 are presented from this viewpoint. Both interpreted and compiled functions store both the names and values of variables so that interpreted and compiled functions are compatible and can be freely intermixed, i.e., free variables can be used with no *special* declarations necessary.¹ The names are

¹ However, it is possible to *suppress* storing of names in compiled functions, either for efficiency or to avoid a clash, via a LOCALVAR declaration (see Section 18).

also very useful in debugging, for they make possible a complete symbolic backtrace in case of error.

In addition to the binding information, additional information is associated with each function call: control information indicating the calling function, access information indicating the path to search the basic frames, and temporary results are also stored on the stack in a block called the frame extension. The interpreter also stores information about partially evaluated expressions as described below.

12.1 THE PUSH-DOWN LIST AND THE INTERPRETER

In addition to the names and values of arguments for functions, information regarding partially-evaluated expressions is kept on the push-down list. For example, consider the following definition of the function fact (intentionally faulty):

```
(FACT
  [LAMBDA (N)
    (COND
      ((ZEROP N)
        L)
      (T (ITIMES N (FACT (SUB1 N))))
```

In evaluating the form (FACT 1), as soon as fact is entered, the interpreter begins evaluating the implicit progn following the LAMBDA (see Section 4). The first function entered in this process is cond. cond begins to process its list of clauses. After calling zerop and getting a NIL value, cond proceeds to the next clause and evaluates T. Since T is true, the evaluation of the implicit progn that is the consequent of the T clause is begun (see Section 4). This requires calling the function itimes. However before itimes can be called, its arguments must be evaluated. The first argument is evaluated by retrieving the current binding of n from its value cell; the second involves a recursive call to fact, and another implicit progn, etc.

Note that at each stage of this process, some portion of an expression has been evaluated, and another is awaiting evaluation. The output below illustrates this by showing the state of the push-down list at the point in the computation of (FACT 1) when the unbound atom L is reached.

```

←FACT(1)
u.b.a. L {in FACT} in ((ZEROP N) L)
(L BROKEN)
:BTVI

*TAIL* (L)

*ARG1 (((ZEROP N) L) (T (ITIMES N (FACT (SUB1 N))))))
COND

*FORM* (COND ((ZEROP N) L) (T (ITIMES N (FACT (SUB1 N))))))
*TAIL* ((COND ((ZEROP N) L) (T (ITIMES N (FACT (SUB1 N)))))))

N 0
FACT

*FORM* (FACT (SUB1 N))
*FN* ITIMES
*TAIL* ((FACT (SUB1 N)))
*ARGVAL* 1
*FORM* (ITIMES N (FACT (SUB1 N)))
*TAIL* ((ITIMES N (FACT (SUB1 N)))))

*ARG1 (((ZEROP N) L) (T (ITIMES N (FACT (SUB1 N))))))
COND

*FORM* (COND ((ZEROP N) L) (T (ITIMES N (FACT (SUB1 N))))))
*TAIL* ((COND ((ZEROP N) L) (T (ITIMES N (FACT (SUB1 N)))))))

N 1
FACT

**TOP**

```

Internal calls to eval, e.g., from cond and the interpreter, are marked on the push-down list by a special mark or blip which the backtrace prints as *FORM*.² The genealogy of *FORM*'s is thus a history of the computation. Other temporary information stored on the stack by the interpreter includes the tail of a partially evaluated implicit progn (e.g., a cond clause or lambda expression) and the tail of a partially evaluated form (i.e., those arguments not yet evaluated), both indicated on the backtrace by *TAIL*, the values of arguments that have already been evaluated, indicated by *ARGVAL*, and the names of functions waiting to be called, indicated by *FN*. *ARG1, ... *ARGn are used by the backtrace to indicate the (unnamed) arguments to subs.

Note that a function is not actually entered and does not appear on the stack, until its arguments have been evaluated.³ Also note that the *ARG1, *FORM*, *TAIL*, etc. "bindings" comprise

² Note that *FORM*, *TAIL*, *ARGVAL*, etc., do not actually appear on the backtrace, i.e., evaluating *FORM* or calling stkscan to search for it will not work. However, the functions blipval, setblipval, and blipscan described below are available for accessing these internal blips.

³ except for functions which do not have their arguments evaluated (although they themselves may call eval, e.g., cond).

the actual working storage. In other words, in the above example, if a (lower) function changed the value of the *ARG1 binding, the cond would continue interpreting the new binding as a list of cond clauses. Similarly, if the *ARGVAL* binding were changed, the new value would be given to itimes as its first argument after its second argument had been evaluated, and itimes was actually called.

BLIP FUNCTIONS

The temporaries of the interpreter, or blips, can be accessed by the following three functions, which currently know about four different types of blips:

- *FN* the name of a function about to be called
- *ARGVAL* an argument for a function about to be called
- *FORM* a form in the process of evaluation
- *TAIL* the tail of a cond clause, implicit progn, prog, etc.

blipval[<u>bliptyp</u> ; <u>ipos</u> ; <u>flg</u>]	Returns the value of the specified blip of type <u>bliptyp</u> . If <u>flg</u> is a number, finds the <u>n</u> th blip of the desired type, searching the control chain beginning at the frame specified by the stack descriptor <u>ipos</u> . If <u>flg</u> is NIL, 1 is used. If <u>flg</u> is T, returns the number of blips of the specified type at <u>ipos</u> .
setblipval[<u>bliptyp</u> ; <u>ipos</u> ; <u>n</u> ; <u>val</u>]	Sets the value of the specified blip of type <u>bliptyp</u> . Searches for the <u>n</u> th blip of the desired type, beginning with the frame specified by the stack descriptor <u>ipos</u> , and following the control chain.
blipscan[<u>bliptyp</u> ; <u>ipos</u>]	Returns a stack pointer to the frame in which a blip of type <u>bliptyp</u> is located. Search begins at the frame specified by the stack descriptor <u>ipos</u> and follows the control chain.

12.2 THE SPAGHETTI STACK

The Bobrow/Wegbreit paper, "A Model and Stack Implementation for Multiple Environments" [Bob3], describes an access and control mechanism more general than the simple pushdown stack. The access and control mechanism used by Interlisp is a slightly modified version of the one proposed by Bobrow and Wegbreit. This mechanism is called the "spaghetti stack."

The spaghetti system presents the access and control stack as a data structure composed of "frames." The functions described below operate on this structure. These primitives allow user functions to manipulate the stack in a machine independent way. Backtracking, coroutines, and more sophisticated control schemes can be easily implemented with these primitives.

OVERVIEW OF SPAGHETTI STACK

The evaluation of a function requires the allocation of storage to hold the values of its local variables during the computation. In addition to variable bindings, an activation of a function

requires a return link (indicating where control is to go after the completion of the computation) and room for temporaries needed during the computation. In the spaghetti system, one "stack" is used for storing all this information, but it is best to view this stack as a tree of linked objects called frame extensions (or simply frames).

A frame extension is a variable sized block of storage containing a frame name, a pointer to some variable bindings (the blink), and two pointers to other frame extensions (the alink and clink). In addition to these components, a frame extension contains other information (such as temporaries and reference counts) that does not interest us here.

The block of storage holding the variable bindings is called a basic frame. A basic frame is essentially an array of pairs, each of which contains a variable name and its value. The reason frame extensions point to basic frames (rather than just having them "built in") is so that two frame extensions can share a common basic frame. This allows two processes to communicate via shared variable bindings.

The chain of frame extensions which can be reached via the successive alinks from a given frame is called the access chain of the frame. The first frame in the access chain is the starting frame. The chain through successive clinks is called the control chain.

A frame extension completely specifies the variable bindings and control information necessary for the evaluation of a function. Whenever a function (or in fact, any form which generally binds local variables) is evaluated, it is associated with some frame extension.

In the beginning there is precisely one frame extension in existence. This is the frame in which the top-level call to the interpreter is being run. This frame is called the "top-level" frame.

Since precisely one function is being executed at any instant, exactly one frame is distinguished as having the "control bubble" in it. This frame is called the active frame. Initially, the top-level frame is the active frame. If the computation in the active frame invokes another function, a new basic frame and frame extension are built. The frame name of this basic frame will be the name of the function being called. The b-, a-, and clinks of the new frame all depend on precisely how the function is invoked. The new function is then run in this new frame by passing control to that frame, i.e., it is made the active frame.

Once the active computation has been completed, control normally returns to the frame pointed to by the clink of the active frame. That is, the frame in the clink becomes the active frame.

In most cases, the storage associated with the basic frame and frame extension just abandoned can be reclaimed. However, it is possible to obtain a pointer to a frame extension and to "hold on" to this frame even after it has been exited. This pointer can be used later to run another computation in that environment, or even "continue" the exited computation.

A separate data type, called a stack pointer, is used for this purpose. A stack pointer is just a cell that literally points to a frame extension. Stack pointers print as #adr/framename, e.g., #117753/COND. Stack pointers are returned by many of the stack manipulating functions described below. Except for certain abbreviations (such as "the frame with such-and-such a name"), stack pointers are the only way the user can reference a frame extension. As long as the user has a stack pointer which references a frame extension, that frame extension (and all those that can be reached from it) may not (will not) be garbage collected.

Note that two stack pointers referencing the same frame extension are *not* necessarily eq, i.e., (EQ (STKPOS 'FOO) (STKPOS 'FOO))=NIL. However, cqp can be used to test if two different stack pointers reference the same frame extension.

- + It is possible to evaluate a form with respect to an access chain other than the current one by using a stack pointer to refer to the head of the access chain desired. When evaluating the form, since all references to variables under the shallow binding scheme go through the variable's value cell, the values in the value cells must be adjusted to reflect the values appropriate to the desired access chain. This is done by changing all the bindings on the current access chain (all the name-value pairs) so that they contain the value current at the time of the call. Then along the new access path, all bindings are made to contain the previous value of the variable, and the current value is placed in the value cell. For that part of the access path which is shared by the old and new chain, no work has to be done. The context switching time, i.e. the overhead in switching from the current, active, access chain to another one, is directly proportional to the size of the two branches that are not shared between the access contexts. This cost should be remembered in using + coroutines and generators, as described below.

12.3 STACK FUNCTIONS

In the descriptions of the stack functions below, when we refer to an argument as a stack descriptor, we mean that it is either a stack pointer or one of the following abbreviations:

1. NIL means the active frame; that is, the frame of the stack function itself.
2. T means the top-level frame.
3. Any other literal atom is equivalent to (STKPOS ATOM -1).
4. A number is equivalent to (STKNTH number).

In the stack functions described below, the following errors can occur.

ILLEGAL STACK ARG Occurs when a stack descriptor is expected and the supplied argument is either not a legal stack descriptor (i.e., not a stack pointer, litatom, or number), or is a litatom or number for which there is no corresponding stack frame (e.g., (STKNTH -1 (QUOTE FOO)) where there is no frame named FOO in the active control chain or (STKNTH -10 (QUOTE EVALQT))).

STACK POINTER HAS BEEN RELEASED Occurs whenever a released stack pointer is supplied as a stack descriptor argument for any purpose other than as a stack pointer to re-use.

FUNCTIONS

stkpos[framename;n;ipos;opos] Search for the nth frame with name framename. The search begins with (and includes) the frame specified by the stack descriptor ipos (initial position). The search proceeds along the control chain from ipos if n is negative, or along the access chain if n is positive. If n is NIL, -1 is used. Returns a stack pointer to the frame if such a frame exists, otherwise returns NIL. If opos is supplied and is a

stack pointer, it is reused. If opos is not a stack pointer it is ignored. (Note that (STKPOS (QUOTE STKPOS)) causes an error, ILLEGAL STACK ARG; it is not permissible to create a stack pointer to the active frame.)

stknth[n;ipos;opos]

Returns a stack pointer to the nth frame back from the frame specified by the stack descriptor ipos. If n is negative, the control chain from ipos is followed. If n is positive the access chain is followed. If n equals 0, returns a stack pointer to ipos, i.e., this provides a way to copy a stack pointer. Returns NIL if there are fewer than n frames in the appropriate chain. If opos is supplied and is a stack pointer, it is reused. If opos is not a stack pointer it is ignored. (Note that (STKNTH 0) causes an error, ILLEGAL STACK ARG; it is not possible to create a stack pointer to the active frame.)

stcname[pos]

Returns the frame name of the frame specified by the stack descriptor pos.

setstcname[pos;name]

changes the frame name of the frame specified by pos to be name.
Value is name. +

stknthname[n;ipos]

Returns the frame name of the nth frame back from ipos. Equivalent to (STCKNAME (STKNTH n ipos)) but avoids creation of a stack pointer.

In summary, stckpos converts function names to stack pointers, stknth converts numbers to stack pointers, stcname converts stack pointers to function names, and stknthname converts numbers to function names.

dummyframep[pos]

T if the user never wrote write a call to the function at pos, e.g. in Interlisp-10, dummyframep is T for *PROG*LAM, *ENV*, and FOOBLOCK frames (see block compiler, Section 18).⁴ +

realframep[pos;interpflg]

is pos, if pos is a "real" frame, i.e. if pos is not a dummy frame, and, either pos is a frame that does not disappear when compiled, or interpflg=T, otherwise NIL. For example, if stcname[pos]=COND, realframep[pos T] is T but realframep[pos]=NIL. +

4

dummyframep is used by the break package for matching against baktracest.

- + **realstknth[n;pos;interpflg;oldpos]**
- + skips back n (or -n) frames for which **realframep[pos;interpflg]=T**.
- + Thus **realframep** and **realstknth** can be used to write functions which manipulate the stack and work on either interpreted or compiled code.

The following functions are used for accessing and changing bindings. Some of functions take an argument, n, which specifies a particular binding in the basic frame. If n is a literal atom, it is assumed to be the name of a variable bound in the basic frame. If n is a number, it is assumed to reference the nth binding in the basic frame. The first binding is 1. If the basic frame contains no binding with the given name or if the number is too large or too small, the error **ILLEGAL ARG** results.

stkscan[var;ipos;opos]	Searches beginning at <u>ipos</u> for a frame in which a variable named <u>var</u> is bound. The search follows the access chain. Returns a stack pointer to the frame if found, otherwise returns NIL . If <u>opos</u> is a stack pointer it is reused, otherwise it is ignored.
framescan[atom;pos]	Returns the relative position of the binding of <u>atom</u> in the basic frame of <u>pos</u> .
stkarg[n;pos]	Returns the value of the binding specified by <u>n</u> in the basic frame of the frame specified by the stack descriptor <u>pos</u> . <u>n</u> can be a literal atom or number.
stkargname[n;pos]	Returns the name of the binding specified by <u>n</u> , in the basic frame of the frame specified by the stack descriptor <u>pos</u> . <u>n</u> can be a literal atom or number.
setstkarg[n;pos;value]	Sets the value of the binding specified by <u>n</u> in the basic frame of the frame specified by the stack descriptor <u>pos</u> . <u>n</u> can be a literal atom or a number. Returns <u>value</u> .
setstkargname[n;pos;name]	Sets the <u>name</u> of the binding specified by <u>n</u> in the basic frame of the frame specified by the stack descriptor <u>pos</u> . <u>n</u> can be a literal atom or a number. Returns <u>name</u> .
stknargs[pos]	Returns the number of arguments bound in the basic frame of the frame specified by the stack descriptor <u>pos</u> .

As an example of the use of **stknargs** and **stkargname**:

variables[pos] returns list of variables bound at pos.

can be defined by:

```
(VARIABLES
  [LAMBDA (POS)
    (PROG (N L)
      (SETQ N (STKNARGS POS))
      LP (COND
        ((ZEROP N)
         (RETURN L)))
      (SETQ L (CONS (STKARGNAME N POS)
                     L)))
      (SETQ N (SUB1 N))
      (GO LP]))
```

The dual of variables is also available:

stkargs[pos;-]

Returns list of values of variables bound at pos.

The following functions are used to evaluate an expression in a different environment, and/or to alter the flow of control.

enveval[form;apos;cpos;aflg;cflg]

evaluates form in the environment specified by apos and cpos. That is, a new active frame is created with the frame specified by the stack descriptor apos as its alink, and the frame specified by the stack descriptor cpos as its clink. Then form is evaluated. If aflg is not NIL, and apos is a stack pointer, then apos will be released. Similarly, if cflg is not NIL, and cpos is a stack pointer, then cpos will be released.

envapply[fn;args;apos;cpos;aflg;cflg]

applies fn to args in the environment specified by apos and cpos. aflg and cflg have the same interpretation as with enveval.

stkeval[pos;form;flg;-]

Evaluates form in the access environment of the frame specified by the stack descriptor pos. If flg is not NIL and pos is a stack pointer, releases pos. The definition of stkeval is (ENVEVAL FORM POS NIL FLG).

stkapply[pos;fn;args;flg;-]

Similar to stkeval but applies fn to args.

reteval[pos;form;flg;-]

Evaluates form in the access environment of the frame specified by the stack descriptor pos, and then returns from pos with that value. If flg is not NIL and pos is a stack pointer, then pos is released. The definition of reteval is equivalent to

(ENVEVAL FORM POS (STKNTH -1 POS) FLG T), except that reteval does not create a stack pointer.

retapply[pos;fn;args;flg;-]	Similar to <u>reteval</u> except applies <u>fn</u> to <u>args</u> .
retfrom[pos;val;flg]	Return from the frame specified by the stack descriptor <u>pos</u> , with the value <u>val</u> . If <u>flg</u> is not NIL, and <u>pos</u> is a stack pointer, then <u>pos</u> is released. An attempt to <u>retfrom</u> the top level (e.g., (RETFROM T)) causes an error, ILLEGAL STACK ARG. <u>Retfrom</u> can be written in terms of <u>enveval</u> as follows:

```
(RETFROM
  (LAMBDA (POS VAL FLG)
    (ENVEVAL (LIST (QUOTE QUOTE) VAL)
      NIL
      (COND
        ((STKNTH -1 POS (COND (FLG POS))))
        (T (ERRORX (LIST 19 POS)))
        NIL T)))
```

retto[pos;val;flg]	like <u>retfrom</u> , except returns <i>to</i> frame specified by <u>pos</u> .
--------------------	--

evalv[x;pos]	Evaluates <u>x</u> , where <u>x</u> is assumed to be a litatom, in the access environment specified by the stack descriptor <u>pos</u> . If <u>x</u> is unbound, <u>evalv</u> returns NOBIND and does not generate an error. While <u>evalv</u> could be defined as (ENVEVAL X POS) it is in fact a <u>subr</u> which is somewhat faster. <u>evalv</u> compiles open when <u>pos</u> =NIL.
--------------	--

function[fn;env]	If <u>env</u> is NIL, <u>function</u> is equivalent to <u>quote</u> when interpreted and is also a signal to the compiler that <u>fn</u> should be compiled. If <u>env</u> is a stack pointer, then the value of <u>function</u> is the expression (FUNARG <u>fn</u> <u>env</u>). When a <u>funarg</u> expression is <u>apply</u> 'd or is <u>car</u> of a form being <u>eval</u> 'd, the <u>apply</u> or <u>eval</u> takes place in the access environment specified by <u>env</u> . For example, if FOO is a <u>funarg</u> expression, then (APPLY FOO FIE) is equivalent to (ENVAPPLY (CADR FOO) FIE (CADDR FOO)). <u>Env</u> can also be a list of variable names. In this case, a new frame is created with the values of the specified variables in the basic frame. The variables are evaluated in the active access environment (the environment of <u>function</u>). The <u>alink</u> of the new frame is the active access environment, and <u>clink</u> is the top level. The value of <u>function</u> is (FUNARG <u>fn</u> <u>pos</u>), where <u>pos</u> is a stack pointer to the new frame. ⁵
------------------	---

The following functions and variables are used to manipulate stack pointers.

⁵ Note that the effect of funarg in the spaghetti system is somewhat different from what it was previously in non-spaghetti system. Now when the funarg is apply'd or eval'd we see in the access environment first the variables given in the list, and then the access environment at the time the funarg was created. Formerly we saw the variables in the list (the "own" variables) and then the access environment at the time the funarg was used.

stackp[x]	Returns <u>x</u> if <u>x</u> is a stack pointer, otherwise returns NIL.
relstk[pos]	Release the stack pointer <u>pos</u> . If <u>pos</u> is not a stack pointer, does nothing. The value is <u>pos</u> .
relstkp[x]	returns <u>x</u> if <u>x</u> is a released stack pointer, NIL otherwise. +
clearstk[flg]	If <u>flg</u> is NIL, releases all active stack pointers, and returns NIL. If <u>flg</u> is T, returns a list of all the active (unreleased) stack pointers.
clearstklst	Is a (global) variable used by top-level <u>evalqt</u> . Every time <u>evalqt</u> is re-entered (e.g., following errors, or control-D), <u>clearstklst</u> is checked. If its value is T, all active stack pointers are released using <u>clearstk</u> . If its value is a list, then all stack pointers on that list are released. If its value is NIL, nothing is released. <u>clearstklst</u> is initially T.
noclearstklst	is a global variable used by top-level <u>evalqt</u> . If <u>clearstklst</u> is T (see above) all active stack pointers <i>except</i> those on <u>noclearstklst</u> are released. <u>noclearstklst</u> is initially NIL.
Thus if one wishes to use multiple environments that survive through control-D, either <u>clearstklst</u> should be set to T, or else those stack pointers to be retained should be explicitly added to <u>noclearstklst</u> .	
copystk[pos1;pos2]	Copies the stack, including basic frames, from the frame specified by the stack descriptor <u>pos1</u> to the frame specified by the stack descriptor <u>pos2</u> . That is, copies the frame extensions and basic frames in the access chain from <u>pos2</u> to <u>pos1</u> (inclusive). <u>Pos1</u> must be in the access chain of <u>pos2</u> , i.e., "above" <u>pos2</u> . Returns the new <u>pos2</u> . This provides a way to save an entire environment including variable bindings.
backtrace[ipos;epos;flags;file;printfn]	Performs a backtrace beginning at the frame specified by the stack descriptor <u>ipos</u> , and ending with the frame specified by the stack descriptor <u>epos</u> . <u>flags</u> is a number in which the options of the backtrace are encoded. If a bit is set, the corresponding information is included in the backtrace.
	<ul style="list-style-type: none"> bit 0 - print arguments of non-subrs bit 1 - print temporaries of the interpreter bit 2 - print subr arguments and localvars bit 3 - omit printing of UNTRACE: and function names bit 4 - follow access chain instead of control chain. bit 5 - print temporaries, i.e. the blips.

For example: if flags=7, everything is printed; if flags=21Q, follows the access chain, prints arguments.

+ file is the file that the backtrace is printed to. file must be open.
+ printfn is used when printing the values of variables, temporaries,
+ blips, etc. printfn=NIL defaults to PRINT.

mapdl[mapdlfn;mapdlpos] starts at mapdlpos and applies mapdlfn, a function of two arguments, to the function name at each frame, and the frame (stack pointer) itself, until the top of the stack is reached. Value is NIL.

For example, mapdl[(LAMBDA (X) (AND (EXPRP X) (PRINT X)))] will print all exprs on the push-down list.

mapdl[(LAMBDA (X POS) (COND ((IGREATERP (STKNARGS POS) 2) (PRINT X))))]
will print all functions of more than two arguments.

searchpd1[srchfn;srchpos] similar to mapdl, except searches the pushdown list starting at position srchpos until it finds a frame for which srchfn, a function of two arguments, applied to the name of the function and the frame itself is not NIL. Value is (name . frame) if such a frame is found, otherwise NIL.

12.4 RELEASING AND REUSING STACK POINTERS

The creation of a single stack pointer can result in the retention of a large amount of stack space. Furthermore, this space will not be freed until the next garbage collection, *even if the stack pointer is no longer being used*, unless the stack pointer is explicitly released or reused. If there is sufficient amount of stack space tied up in this fashion, a STACK OVERFLOW condition can occur, even in the simplest of computations. For this reason, the user should consider releasing a stack pointer when the environment referenced by the stack pointer is no longer needed.

The effects of releasing a stack pointer are:

1. The link between the stack pointer and the stack is broken by setting the contents of the stack pointer to the "released mark" (currently unboxed 0). A released stack pointer prints as #adr/#0.
2. If this stack pointer was the last remaining reference to a frame extension; that is, if no other stack pointer references the frame extension and the extension is not contained in the active control or access chain, then the extension may be reclaimed, and is reclaimed immediately. The process repeats for the access and control chains of the reclaimed extension so that all stack space that was reachable only from the released stack pointer is reclaimed.

A stack pointer may be released using the function relstk, but there are some cases for which relstk is not sufficient. For example, if a function contains a call to retfrom in which a stack pointer was used to specify where to return to, it would not be possible to simultaneously release the stack pointer. (A relstk appearing in the function following the call to retfrom would not be executed!)

To permit release of a stack pointer in this situation, the stack functions that relinquish control have optional flag arguments to denote whether or not a stack pointer is to be released. Note that in this case releasing the stack pointer will *not* cause the stack space to be reclaimed immediately because the frame referenced by the stack pointer will have become part of the active environment.

REUSING STACK POINTERS

Another way of avoiding creating new stack pointers is to *reuse* stack pointers that are no longer needed. The stack functions that create stack pointers (`stkpos`, `stknth`, and `stkscan`) have an optional argument which is a stack pointer to reuse. When a stack pointer is reused, two things happen. First the stack pointer is released (see above). Then the pointer to the new frame extension is deposited in the stack pointer. The old stack pointer (with its new contents) is the value of the function. Note that the reused stack pointer will be released even if the function does not find the specified frame.

Note that even if stack pointers are explicitly being released, *creation* of many stack pointers can cause a garbage collection of stack pointer space. Thus, if the user's application requires creating many stack pointers, he definitely should take advantage of reusing stack pointers.

12.5 COROUTINES AND GENERATORS⁶

This section describes an application of the spaghetti stack facility to provide mechanisms for creating and using simple generators (with and without CLISP, Section 23), generalized coroutines, and Conniver style possibility lists.

A *generator* is like a subroutine except that it retains information about previous times it has been called. Some of this state may be data (for example, the seed in a random number generator), and some may be in program state (as in a recursive generator which finds all the atoms in a list structure). For example, if `listgen` is defined using defineq as:

```
(LISTGEN (L)
  (IF L THEN (PRODUCE L:1) (LISTGEN L::1)))
```

we can use the function `generator` (described below) to create a generator that uses `listgen` to produce the elements of a list one at a time, e.g.,

```
GR←(GENERATOR (LISTGEN '(A B C))
```

creates a generator, which can be called by

```
(GENERATE GR)
```

to produce as values on successive calls, A, B, C. When `generate` (not `generator`) is called the first time, it simply starts evaluating `(LISTGEN '(A B C))`. `produce` gets called from `listgen`, and pops back up to `generate` with the indicated value after saving the state. When `generate` gets called

6

Designed and implemented by D.G. Bobrow, who also did the documentation. Early versions of the Conniver possibilites-list package were written by Henry Thompson. Daryle Lewis found and corrected a number of bugs, and wrote the compiler macros that go with the package.

again, it continues from where the last produce left off. This process continues until finally listgen completes and returns a value (it doesn't matter what it is). generate then returns gr itself as its value, so that the program that called generate can tell that it is finished, i.e., there are no more values to be generated.

generator[form# #;comvar# #]

is an nlambda function that creates a generator which uses form# # to compute values. The value of generator is a *generator handle* which is represented by a dotted pair of stack pointers.

comvar# # is optional. If its value (eval of) is a generator handle, the list structure and stack pointers will be reused. Otherwise, a new generator handle will be constructed.

generator compiles open.

produce[val]

is used from within (below) a generator to return val as the value of the corresponding call to generate.

generate[handle;val]

restarts the generator represented by handle. val will be returned as the value of the produce which last suspended the operation of the generator. When the generator runs out of values, generate returns handle itself.

EXAMPLES

The following function will go down recursively through a list structure and produce the atoms in the list structure one at a time.

```
[LEAVESG (L)
  (if (ATOM L)
      then (PRODUCE L)
      else (LEAVESG L:1)
        (if L::1
            then (LEAVESG L::1)])
```

The following function prints each of these atoms as it appears. It illustrates how a loop can be set up to use a generator.

```
(PLEAVESG1 (L)
  (PROG (X LHANDLE)
    (LHANDLE←(GENERATOR (LEAVESG L)))
    LP  (X←(GENERATE LHANDLE))
    (if X=LHANDLE
        then (RETURN NIL))
    (PRINT X)
    (GO LP)))
```

Note that the loop terminates when the value of the generator is eq to the dotted pair which is the value produced by the call to generator. A CLISP iterative operator, OUTOF, is provided which

makes it much easier to write the loop in PLEAVESG1. OUTOF (or outof) can precede a form which is to be used as a generator. On each iteration, the iteration variable will be set to successive values returned by the generator; the loop will be terminated automatically when the generator runs out. Thus we can write

```
(PLEAVESG2 (L)
  (for X outof (LEAVESG L) do (PRINT x))
```

as equivalent to the above program PLEAVESG1.

Here is another example:

```
(for X outof (MAPATOMS (FUNCTION PRODUCE))
  as I from 1 to N do (PRINT X))
```

will print the first n atoms.

COROUTINES

This package provides facilities for the creation and use of fully general coroutine structures. It uses a stack pointer to preserve the state of a coroutine, and allows arbitrary switching between n different coroutines, rather than just a call to a generator and return. This package is slightly more efficient than the generator package described above, and allows more flexibility on specification of what to do when a coroutine terminates.

coroutine[callptr##;coroutptr##;coroutform##;endform##]

This nlambda is used to create a coroutine and initialize the linkage. callptr## and coroutptr## are the names of two variables, which will be set to appropriate stack pointers. If the values of callptr## or coroutptr## are already stack pointers, the stack pointers will be reused. coroutform## is the form which is evaluated to start the coroutine; endform## is a form to be evaluated if coroutform## actually returns when it runs out of values.

coroutine compiles open.

resume[fromptr;toptr;val]

is used to transfer control from one coroutine to another. fromptr should be the stack pointer for the current coroutine, which will be smashed to preserve the current state. toptr should be the stack pointer which has preserved the state of the coroutine to be transferred to, and val is the value that is to be returned to the latter coroutine as the value of the resume which suspended the operation of that coroutine.

EXAMPLES

The following is the way one might write the LEAVES program using the coroutine package:

```
(LEAVESC (L COROUTPTR CALLPTR)
```

```
(if (ATOM L) then (RESUME COROUTPTR CALLPTR L)
  else (LEAVESC L:1 COROUTPTR CALLPTR)
    (if L::1 then (LEAVESC L::1 COROUTPTR CALLPTR))))
```

A function PLEAVESC which uses LEAVESC can be defined as follows:

```
(PLEAVESC (L)
  (bind PLHANDLE LHANDLE first (ROUTINE PLHANDLE LHANDLE
    (LEAVESC L LHANDLE PLHANDLE)
    (RETFROM 'PLEAVESC))
  do (PRINT (RESUME PLHANDLE LHANDLE))))
```

By RESUMING leavesc repeatedly, this function will print all the leaves of list L and then return out of pleavesc via the retfrom. The retfrom is necessary to break out of the non-terminating do-loop. This was done to illustrate the additional flexibility allowed through the use of endform # #.

We use two coroutines working on two trees in the example eqleaves, defined below. eqleaves tests to see whether two trees have the same leaf set in the same order, e.g., EQLEAVES((A B C)(A B (C))) is true.

```
(EQLEAVES (L1 L2)
  (bind LHANDLE1 LHANDLE2 PE EL1 EL2
    first (ROUTINE PE LHANDLE1 (LEAVESC L1 LHANDLE1 PE) 'NO-MORE)
    (ROUTINE PE LHANDLE2 (LEAVESC L2 LHANDLE2 PE) 'NO-MORE)
    do (EL1←(RESUME PE LHANDLE1))
    (EL2←(RESUME PE LHANDLE2))
    (if EL1~≡EL2
      then (RETURN NIL))
    repeatuntil EL1='NO-MORE finally (RETURN T)))
```

POSSIBILITIES LISTS⁷

A possibilities list is the interface between a generator and a consumer. The possibilities list is initialized by a call to possibilities, and elements are obtained from it by using trynext. By using the spaghetti stack to maintain separate environments, this package allows a regime in which a generator can put a few items in a possibilities list, suspend itself until they have been consumed, and be subsequently aroused and generate some more.

possibilities[form # #]

This nlambda is used for the initial creation of a possibilities list. form # # will be evaluated to create the list. It should use the functions note and au-revoir described below to generate possibilities. Normally, one would set some variable to the possibilities list which is returned, so it can be used later, e.g.:

```
(SETQ PLIST (POSSIBILITIES (GENERFN V1 V2))).
```

possibilities compiles open.

7

These functions are based on the CONNIVER system possibilities list package.

note[val;lstflg]

is used within a generator to put items on the possibilities list being generated. If lstflg is equal to NIL, val is treated as a single item. If lstflg is non-NIL, then the list val is conceded on the end of the possibilities list. Note that it is perfectly reasonable to create a possibilities list using a second generator, and note that list as possibilities for the current generator with lstflg equal to T. The lower generator will be resumed at the appropriate point.

au-revoir[val# #]

puts val# # on the possibilities list if it is given,⁸ and then suspends the generator and returns to the consumer in such a fashion that control will return to the generator at the au-revoir if the consumer exhausts the possibilities list.

adieu[val# #]

like au-revoir except releases the generator instead of suspending it.

trynext[plst# #;endform# #;val# #]

This lambda allows a consumer to use a possibilities list. It removes the first item from the possibilities list named by plst# # (i.e. plst# # must be an atom whose value is a possibilities list), and returns that item, provided it is not a generator handle. If a generator handle is encountered, the generator is reawakened. When it returns a possibilities list, this list is added to the front of the current list. When a call to trynext causes a generator to be awakened, val# # is returned as the value of the au-revoir which put that generator to sleep. If plst# # is empty, it evaluates endform# # in the caller's environment.

trynext compiles open.

cleanposlst[plst]

This function is provided to release any stack pointers which may be left in the plst which was not used to exhaustion.

EXAMPLE

fib is a generator for fibonacci numbers. It starts out by noteing its two arguments, then suspends itself. Thereafter, on being re-awakened, it will note two more terms in the series and suspends again. printfib uses fib to print the first N fibonacci numbers.

8

NIL is not put on the possibilities list unless it is explicitly given as an argument to au-revoir, i.e., au-revoir[] and au-revoir[NIL] are *not* the same. au-revoir and adieu are lambda spreads to enable them to distinguish these two cases.

```
[FIB (F1 F2)
  (do (NOTE F1)
       (NOTE F2)
       (F1←F1+F2)
       (F2←F1+F2)
       (AU-REVOIR)]9

[PRINTFIB (N)
  (PROG ((FL (POSSIBILITIES (FIB 0 1))))
         (RPTQ N (PRINT (TRYNEXT FL)))
         (CLEANPOSLST FL) )]
```

Note that fib itself will never terminate.

⁹ Note that this au-revoir just suspends the generator and adds nothing to the possibilities list except the generator.

Index for Section 12

	Page Numbers
access chain	12.5
active frame	12.5
ADIEU[VAL##] *	12.17
alink	12.5
association list	12.1
AU-REVOIR[VAL##] *	12.17
BACKTRACE[IPOS;EPOS;FLAGS;FILE;PRINTFN] SUBR	12.11
backtrace	12.2-3
backtracking	12.4
basic frame	12.5,1.8
bindings in a basic frame	12.8
blink	12.5
blip functions	12.4
blips	12.4
BLIPSCAN[BLIPTYP;IPOS] SUBR	12.4
BLIPVAL[BLIPTYP;IPOS;FLG] SUBR	12.4
CLEANPOSLST[PLST]	12.17
CLEARSTK[FLG] SUBR	12.11
CLEARSTKLST (system variable/parameter)	12.11
clink	12.5
context switching	12.6
control chain	12.5
COPYSTK[POS1;POS2] SUBR	12.11
COROUTINE[CALLPTR##;COROUTPTR##;COROUTFORM##; ENDFORM##] NL	12.15
coroutines	12.16,4,13
debugging	12.2
DUMMYFRAMEP[POS]	12.7
ENVAPPLY[FN;ARGS;APOS;CPOS;AFLG;CFLG] SUBR	12.9
ENVEVAL[FORM;APOS;CPOS;AFLG;CFLG] SUBR	12.9
EVALV[X;POS] SUBR	12.10
frame extension	12.5
frame name	12.5
frames	12.5
FRAMESCAN[ATOM;POS] SUBR	12.8
FUNCTION[FN;ENV] NL	12.10
GENERATE[HANDLE;VAL]	12.14
GENERATOR[FORM##;COMVAR##] NL	12.14
generator handle	12.14
generators	12.13
ILLEGAL ARG (error message)	12.8
ILLEGAL STACK ARG (error message)	12.6
implicit progn	12.2-3
MAPDL[MAPDLFN;MAPDLPOS]	12.12
NOBIND	12.10
NOCLEARSTKLST (system variable/parameter)	12.11
NOTE[VAL;LSTFLG]	12.17
OUTOF (clisp iterative statement operator)	12.15
POSSIBILITIES[FORM##] NL	12.16
possibilities lists	12.16
PRODUCE[VAL]	12.14
pushdown list	12.2,1,3-4
REALFRAMEP[POS;INTERPFLG]	12.7
REALSTKNTH[N;POS;INTERPFLG;OLDPOS]	12.8
releasing stack pointers	12.12
RELSTK[POS] SUBR	12.11,12

Page
Numbers

RELSTKP[X]	12.11
RESUME[FROMPTR;TOPTR;VAL] SUBR	12.15
RETAPPLY[POS;FN;ARGS;FLG;INTERNALFLG]	12.10
RETEVAL[POS;FORM;FLG;INTERNALFLG]	12.9
RETFROM[POS;VAL;FLG] SUBR	12.10
RETTO[POS;VAL;FLG] SUBR	12.10
reusing stack pointers	12.13
SEARCHPDL[SRCHFN;SRCHPOS]	12.12
SETBLIPVAL[BLIPTYP;IPOS;N;VAL] SUBR	12.4
SETSTKARG[N;POS;VALUE] SUBR	12.8
SETSTKARGNAME[N;POS;NAME] SUBR	12.8
SETSTKNAME[POS;NAME] SUBR	12.7
shallow binding	12.1
spaghetti stacks	12.4-13
stack descriptor	12.6
stack functions	12.6-12
STACK OVERFLOW (error message)	12.12
stack pointer	12.5
STACK POINTER HAS BEEN RELEASED (error message) ..	12.6
STACKP[X] SUBR	12.11
STKAPPLY[POS;FN;ARGS;FLG;INTERNALFLG]	12.9
STKARG[N;POS] SUBR	12.8
STKARGNAME[N;POS] SUBR	12.8
STKARGS[POS;NARGS]	12.9
STKEVAL[POS;FORM;FLG;INTERNALFLG]	12.9
STKNAME[POS] SUBR	12.7
STKNARGS[POS] SUBR	12.8
STKNTH[N;IPOS;OPOS] SUBR	12.7
STKNTHNAME[N;POS] SUBR	12.7
STKPOS[FRAMENAME:N;IPOS;OPOS] SUBR	12.6
STKSCAN[VAR;IPOS;OPOS] SUBR	12.8
TRYNEXT[PLST##;ENDFORM##;VAL##] NL	12.17
value cell	12.1
variable bindings	12.1,2-4
VARIABLES[POS]	12.8
#0	12.12
ARGVAL (as a blip on the stack)	12.4
*ARG1 (as a blip on the stack)	12.3
FN (as a blip on the stack)	12.4
FORM (as a blip on the stack)	12.4
TAIL (as a blip on the stack)	12.4

SECTION 13

NUMBERS AND ARITHMETIC FUNCTIONS

There are three different types of numbers in Interlisp: small integers, large integers, and floating point numbers.¹ Since a large integer or floating point number can be (in value) any full word quantity (and vice versa), it is necessary to distinguish between those full word quantities that represent large integers or floating point numbers, and other Interlisp pointers. We do this by "boxing" the number, which is sort of like a special "cons": when a large integer or floating point number is created (via an arithmetic operation or by read), Interlisp gets a new word from "number storage" and puts the large integer or floating point number into that word. Interlisp then passes around the pointer to that word, i.e., the "boxed number", rather than the actual quantity itself. Then when a numeric function needs the actual numeric quantity, it performs the extra level of addressing to obtain the "value" of the number. This latter process is called "unboxing". Note that unboxing does not use any storage, but that each boxing operation uses one new word of number storage. Thus, if a computation creates many large integers or floating point numbers, i.e., does lots of boxes, it may cause a garbage collection of large integer space, or of floating point number space.²

13.1 INTEGER ARITHMETIC

SMALL INTEGERS

Small integers are those integers for which smallp is true. In Interlisp-10, these are integers whose absolute value is less than 1536. Small integers are boxed by offsetting them by a constant so that they overlay an area of Interlisp's address space that does not correspond to any Interlisp data type. Thus boxing small numbers does not use any storage, and furthermore, each small number has a unique representation, so that eq may be used to check equality. Note that eq should not be used for large integers or floating point numbers, e.g., in Interlisp-10, eq[2000;add1[1999]] is NIL! eqp, icqp, or equal must be used instead.

-
- ¹ Floating point numbers are created by the read program when a . or an E appears in a number, e.g., 1000 is an integer, 1000. a floating point number, as are 1E3 and 1.E3. Note that 1000D, 1000F, and 1E3D are perfectly legal literal atoms.
 - ² Different implementations of Interlisp-10 may use different boxing strategies. Thus, while lots of arithmetic operations *may* lead to garbage collections, this is not necessarily always the case.

INTEGER FUNCTIONS

All of the functions described below work on integers. Unless specified otherwise, if given a floating point number, they first convert the number to an integer by truncating the fractional bits, e.g., `iplus[2.3;3.8]=5`; if given a non-numeric argument, they generate an error, **NON-NUMERIC ARG.**

It is important to use the *integer* arithmetic functions, whenever possible, in place of the more general arithmetic functions which allow mixed floating point and integer arithmetic, e.g., `iplus` vs `plus`, `igreaterp` vs `greaterp`, because the integer functions compile open, and therefore run faster than the general arithmetic functions, and because the compiler is "smart" about eliminating

- + unnecessary boxing and unboxing. In other words, if the value of an integer arithmetic function is being used in arithmetic context, i.e. as an argument to another arithmetic function, no boxing or unboxing will be performed. Thus, the expression `(IPLUS (IQUOTIENT (ITIMES N 100) M) (ITIMES X Y))` will compile to perform only one box, the outer one, and the expression `(IGREATERP (IPLUS X Y) (IDIFFERENCE A B))` will compile to do no boxing at all. The compiler also treats a conditional expression all of whose values are integer functions the same as an arithmetic function, e.g. `(IPLUS X (COND (FOO (ITIMES A B)) (T (ITIMES C D))))` will produce only one box, the outer one.

Note that the PDP-10 is a 36 bit machine, so that in Interlisp-10 all integers are between -2^{35} and $2^{35}-1$.³ Adding two integers which produce a result outside this range causes overflow, e.g., $2^{34} + 2^{34}$.

The procedure on overflow is to return the largest possible integer, i.e., in Interlisp-10 $2^{35} - 1$.⁴

$$\text{iplus}[x_1; x_2; \dots; x_n] \quad x_1 + x_2 + \dots + x_n. \text{iplus[]} = 0.$$

$$\text{iminus}[x] \quad -x$$

$$\text{idifference}[x;y] \quad x - y$$

$$\text{addl}[x] \quad x + 1$$

$$\text{subl}[x] \quad x - 1$$

$$\text{itimes}[x_1; x_2; \dots; x_n] \quad \text{the product of } x_1, x_2, \dots, x_n. \text{itimes[]} = 1.$$

³ Approximately 34 billion

⁴ If the overflow occurs by trying to create a negative number of too large a magnitude, $-2^{35}+1$ is used instead of $2^{35}-1$.

iquotient[x;y]	x/y truncated, e.g., iquotient[3;2]=1, iquotient[-3,2]=-1	
iremainder[x;y]	the remainder when <u>x</u> is divided by <u>y</u> , e.g., iremainder [3;2]=1	
igreaterp[x;y]	T, if <u>x</u> > <u>y</u> ; NIL otherwise.	
ilessp[x;y]	T, if <u>x</u> < <u>y</u> ; NIL otherwise.	
igeq[x;y]	T, if <u>x</u> \geq <u>y</u> ; NIL otherwise.	+
ileq[x;y]	T, if <u>x</u> \leq <u>y</u> ; NIL otherwise.	+
imin[x ₁ ;x ₂ ;...;x _n]	value is minimum of <u>x₁</u> , <u>x₂</u> , ..., <u>x_n</u> . imin[] returns the smallest possible integer, the value of <u>min.integer</u> . ⁵ Does not compile open.	+
imax[x ₁ ;x ₂ ;...;x _n]	value is maximum of <u>x₁</u> , <u>x₂</u> , ..., <u>x_n</u> . imax[] returns the largest possible integer, the value of <u>max.integer</u> . Does not compile open.	+
ieqp[n;m]	T, if <u>n</u> and <u>m</u> are <u>eq</u> , or equal integers, NIL otherwise. Note that <u>eq</u> may be used if <u>n</u> and <u>m</u> are known to be <i>small</i> integers. <u>ieqp</u> converts <u>n</u> and <u>m</u> to integers, e.g., ieqp[2000;2000.3]=T, causes NON-NUMERIC ARG error if either <u>n</u> or <u>m</u> are not numbers. <u>ieqp</u> compiles open.	
zerop[x]	defined as eq[x;0].	
<i>Note that <u>zerop</u> should not be used for floating point numbers because it uses <u>eq</u>. Use <u>eql[x;0]</u> instead.</i>		
minusp[x]	T if <u>x</u> is negative; NIL otherwise. Does not convert <u>x</u> to an integer, but simply checks sign bit.	
eql[n;m]	T, if <u>n</u> and <u>m</u> are <u>eq</u> , or equal numbers, NIL otherwise. ⁶ Note that <u>eq</u> may be used if <u>n</u> and <u>m</u> are known to be <i>small</i> integers. <u>eql</u>	

⁵ In other words, if x is an integer, then x \geq min.integer.

⁶ eql is also T if n and m are both stack descriptors that refer to the same frame extension (see Section 12).

	does not convert <u>n</u> and <u>m</u> to integers, e.g., <code>eqp[2000;2000.3]=NIL</code> , but it can be used to compare an integer and a floating point number, e.g., <code>eqp[2000;2000.0]=T</code> . <code>eqp</code> does <i>not</i> generate an error if <u>n</u> or <u>m</u> are not numbers.
<code>smallp[n]</code>	<u>n</u> , if <u>n</u> is a small integer, else NIL. <code>smallp</code> does <i>not</i> generate an error if <u>n</u> is not a number.
<code>fix[x]</code>	<u>x</u> , if <u>x</u> is an integer, else NIL. Does <i>not</i> generate an error if <u>x</u> is not a number.
<code>fix[x]</code>	Converts <u>x</u> to an integer by truncating fractional bits, e.g., <code>fix[2.3] = 2</code> , <code>fix[-1.7] = -1</code> . If <u>x</u> is already an integer, <code>fix[x]=x</code> and doesn't use any storage. ⁷
<code>logand[x₁;x₂;...;x_n]</code>	lambda no-spread, value is logical <u>and</u> of all its arguments, as an integer, e.g., <code>logand[7;5;6]=4</code> .
<code>logor[x₁;x₂;...;x_n]</code>	lambda no-spread, value is the logical <u>or</u> of all its arguments, as an integer, e.g., <code>logor[1;3;9]=11</code> .
<code>logxor[x₁;x₂;...;x_n]</code>	lambda no-spread, value is the logical exclusive <u>or</u> of its arguments, as an integer, e.g., <code>logxor[11;5] = 14</code> , <code>logxor[11;5;9] = logxor[14;9] = 7</code> .
<code>lsh[n;m]</code>	(arithmetic) <u>left shift</u> , value is <u>n</u> *2 ^{<u>m</u>} , i.e., <u>n</u> is shifted left <u>m</u> places. <u>n</u> can be positive or negative. If <u>m</u> is negative, <u>n</u> is shifted <u>right</u> - <u>m</u> places.
<code>rsh[n;m]</code>	(arithmetic) <u>right shift</u> , value is <u>n</u> *2 ^{-<u>m</u>} , i.e., <u>n</u> is shifted right <u>m</u> places. <u>n</u> can be positive or negative. If <u>m</u> is negative, <u>n</u> is <u>left</u> - <u>m</u> places.
<code>llsh[n;m]</code>	logical <u>left shift</u> . On PDP-10, <code>llsh</code> is equivalent to <code>lsh</code> .
<code>lrsh[n;m]</code>	logical <u>right shift</u> .

7Since `FIX` is also a `lispx` command (Section 22), typing `FIX` directly to `lispx` will not cause the function `fix` to be called.

The difference between a logical and arithmetic right shift lies in the treatment of the sign bit for negative numbers. For arithmetic right shifting of negative numbers, the sign bit is propagated, i.e., the value is a negative number. For logical right shift, zeroes are propagated. Note that shifting (arithmetic) a negative number "all the way" to the right yields -1, not 0.

gcd[x;y] value is the greatest common divisor of \underline{x} and \underline{y} , e.g., gcd[72;64]=8.

13.2 FLOATING POINT ARITHMETIC

All of the functions described below work on floating point numbers. Unless specified otherwise, if given an integer, they first convert the number to a floating point number, e.g., fplus[1;2.3] = fplus[1.0;2.3] = 3.3; if given a non-numeric argument, they generate an error, NON-NUMERIC ARG.

The largest floating point number (in Interlisp-10) is 1.7014118E38, the smallest positive (non-zero) floating point number is 1.4693679E-39. The procedure on overflow is the same as for integer arithmetic. For underflow, i.e., trying to create a number of too small a magnitude, the value will be 0.

fplus[x₁;x₂;...;x_n] $x_1 + x_2 + \dots + x_n$

fminus[x] $-x$

fdifference[x;y] $x - y$

ftimes[x₁;x₂;...;x_n] $x_1 * x_2 * \dots * x_n$

fquotient[x;y] x/y

fremainder[x;y] the remainder when \underline{x} is divided by \underline{y} , i.e. $* fdifference[x;ftimes[y;fix[fquotient[x;y]]]]$, e.g. fremainder[7.5;2.3]=.6. *

minusp[x] T, if \underline{x} is negative; NIL otherwise. Works for both integers and floating point numbers.

eqp[x;y] T, if \underline{x} and \underline{y} are eq, or equal numbers. See discussion page 13.3.

fgreaterp[x;y] T, if $\underline{x} > \underline{y}$, NIL otherwise.

flessp[x;y] T, if $\underline{x} < \underline{y}$, NIL otherwise.

- + **fmin[x₁;x₂;...;x_n]** value is minimum of x_1, x_2, \dots, x_n . fmin[] returns the smallest possible floating point number, the value of min.float.
- + **fmax[x₁;x₂;...;x_n]** value is maximum of x_1, x_2, \dots, x_n . fmax[] returns the largest possible floating point number, the value of max.float.
- floatp[x]** is x if x is a floating point number; NIL otherwise. Does *not* give an error if x is not a number.

Note that if numberp[x] is true, then either fixp[x] or floatp[x] is true.

- | | |
|-----------------|--|
| float[x] | Converts x to a floating point number, e.g., float[0] = 0.0. |
|-----------------|--|

13.3 MIXED ARITHMETIC

The functions in this section are "contagious floating point arithmetic" functions, i.e., if any of the arguments are floating point numbers, they act exactly like floating point functions, and float all arguments, and return a floating point number as their value. Otherwise, they act like the integer functions. If given a non-numeric argument, they generate an error, NON-NUMERIC ARG.

- plus[x₁;x₂;...;x_n]** $x_1 + x_2 + \dots + x_n$
- minus[x]** - x
- difference[x;y]** $x - y$
- times[x₁;x₂;...;x_n]** $x_1 * x_2 * \dots * x_n$
- quotient[x;y]** if x and y are both integers, value is iquotient[x;y], otherwise fquotient[x;y].
- remainder[x;y]** if x and y are both integers, value is iremainder[x;y], otherwise fremainder[x;y].
- greaterp[x;y]** T, if $x > y$, NIL otherwise.
- lessp[x;y]** T if $x < y$, NIL otherwise.
- + **gcq[x;y]** T, if $x \geq y$, NIL otherwise.

leq[x;y]	T, if $\underline{x} \leq \underline{y}$, NIL otherwise.	+
min[x ₁ ;x ₂ ;...;x _n]	value is minimum of $\underline{x}_1, \underline{x}_2, \dots, \underline{x}_n$. min[] returns the value of <u>min.integer</u> .	+
max[x ₁ ;x ₂ ;...;x _n]	value is maximum of $\underline{x}_1, \underline{x}_2, \dots, \underline{x}_n$. max[] returns the value of <u>max.integer</u> .	+
abs[x]	\underline{x} if $x > 0$, otherwise $-x$. abs uses <u>greaterp</u> and <u>minus</u> , (not <u>igreaterp</u> and <u>iminus</u>).	+

13.4 SPECIAL FUNCTIONS⁸

They utilize a power series expansion and their values are (supposed to be) 27 bits accurate, e.g., sin[30]=.5 exactly.

expt[m;n]	value is m^n . If <u>m</u> is an integer and <u>n</u> is a positive integer, value is an integer, e.g. expt[3;4]=81, otherwise the value is a floating point number. If <u>m</u> is negative and <u>n</u> fractional, an error is generated, ILLEGAL EXPONENTIATION. If <u>n</u> is floating and either too large or too small, an error is generated, VALUE OUT OF RANGE EXPT.	+
sqrt[n]	value is a square root of <u>n</u> as a floating point number. <u>n</u> may be fixed or floating point. Generates an error if <u>n</u> is negative. sqrt[n] is about twice as fast as expt[n;.5]	
log[x]	value is natural logarithm of <u>x</u> as a floating point number. <u>x</u> can be integer or floating point.	
antilog[x]	value is floating point number whose logarithm is <u>x</u> . <u>x</u> can be integer or floating point, e.g., antilog[1] = e = 2.71828...	
sin[x;radiansflg]	<u>x</u> in degrees unless <u>radiansflg</u> =T. Value is sine of <u>x</u> as a floating point number.	
cos[x;radiansflg]	Similar to <u>sin</u> .	

⁸ In Interlisp-10, these functions were implemented by J. W. Goodwin by "borrowing" the corresponding routines from the FORTRAN library, and hand coding them in Interlisp-10 via ASSEMBLE.

<code>tan[x;radiansflg]</code>	Similar to <u>sin</u> .
<code>arcsin[x;radiansflg]</code>	x is a number between -1 and 1 (or an error is generated). The value of <u>arcsin</u> is a floating point number, and is in degrees unless <u>radiansflg=T</u> . In other words, if $\text{arcsin}[x;\text{radiansflg}]=z$ then $\text{sin}[z;\text{radiansflg}]=x$. The range of the value of <u>arcsin</u> is -90 to +90 for degrees, $-\pi/2$ to $\pi/2$ for radians.
<code>arccos[x;radiansflg]</code>	Similar to <u>arcsin</u> . Range is 0 to 180, 0 to π .
<code>arctan[x;radiansflg]</code>	Similar to <u>arcsin</u> . Range is 0 to 180, 0 to π .
+ <code>arctan2[x;y;radiansflg]</code>	computes $\text{arctan}[[\text{fquotient}[x;y];\text{radiansflg}],$ and returns a corresponding value in the range -180 to 180 (or -pi to pi), i.e. the result is in the proper quadrant as determined by the signs of <u>x</u> and <u>y</u> .
+ <code>rand[lower;upper]</code>	Value is a pseudo-random number between <u>lower</u> and <u>upper</u> inclusive, i.e., <u>rand</u> can be used to generate a sequence of random numbers. If both limits are integers, the value of <u>rand</u> is an integer, otherwise it is a floating point number. The algorithm is completely deterministic, i.e., given the same initial state, <u>rand</u> produces the same sequence of values. The internal state of <u>rand</u> is initialized using the function <u>randset</u> described below.
+ <code>randset[x]</code>	Value is internal state of <u>rand</u> after <u>randset</u> has finished operating. If <u>x=NIL</u> , value is current state. If <u>x=T</u> , <u>randstate</u> is initialized using the clocks. Otherwise, <u>x</u> is interpreted as a previous internal state, i.e., a value of <u>randset</u> , and is used to reset <u>randstate</u> . For example,
	1. (SETQ OLDSTATE (RANDSET)) 2. Use <u>rand</u> to generate some random numbers. 3. (RANDSET OLDSTATE) 4. <u>rand</u> will generate same sequence as in 2.

13.5 REUSING BOXED NUMBERS IN INTERLISP-10 - SETN

rplaca and rplacd provide a way of cannibalizing list structure for reuse in order to avoid making new structure and causing garbage collections.⁹ This section describes an analogous function in

+ ⁹ The nobox package (Section 24) provides a more aesthetic way of reusing cons cells as well as number boxes. However, it is still the case that techniques involving reusing static storage should be used with extreme caution, and be reserved for those cases where the normal method of storage allocation and garbage collection is not workable or practical. The decl package (Section 24) takes a different approach to the same problem by avoiding creating number boxes in the first place via type declarations in the body of the function definition.

Interlisp-10 for reusing large integers and floating point numbers, setn. setn is used like setq, i.e., its first argument is considered as quoted, its second is evaluated. If the current value of the variable being set is a large integer or floating point number, the new value is deposited into that word in number storage, i.e., no new storage is used.¹⁰ If the current value is *not* a large integer or floating point number, e.g., it can be NIL, setn operates exactly like setq, i.e., the large integer or floating point number is boxed, and the variable is set. This eliminates initialization of the variable.

setn will work interpretively, i.e., reuse a word in number storage, but will not yield any savings of storage because the boxing of the second argument will still take place, when it is evaluated. The elimination of a box is achieved only when the call to setn is compiled, since setn compiles open, and does not perform the box if the old value of the variable can be reused.

CAVEATS CONCERNING USE OF SETN

There are three situations to watch out for when using setn. The first occurs when the same variable is being used for floating point numbers and large integers. If the current value of the variable is a floating point number, and it is reset to a large integer, via setn, the large integer is simply deposited into a word in floating point number storage, and hence will be interpreted as a floating point number. Thus,

```
←(SETQ FOO 2.3)
2.3
←(SETN FOO 10000)
2.189529E-43
```

Similarly, if the current value is a large integer, and the new value is a floating point number, equally strange results occur.

The second situation occurs when a setn variable is reset from a large integer to a small integer. In this case, the small integer is simply deposited into large integer storage. It will then print correctly, and function arithmetically correctly, but it is *not* a small integer, and hence will not be eq to another integer of the same value, e.g.,

```
←(SETQ FOO 10000)
10000
←(SETN FOO 1)
1
←(IPLUS FOO 5)
6
←(EQ FOO 1)
NIL
←(SMALLP FOO)
NIL
```

In particular, note that zerop will return NIL even if the variable is equal to 0. Thus a program which begins with foo set to a large integer and counts it down by (SETN FOO (SUB1 FOO)) must terminate with (EQP FOO 0), not (ZEROP FOO).

¹⁰ The second argument to setn must always be a number or a NON-NUMERIC ARG error is generated.

Finally, the third situation to watch out for occurs when you want to save the current value of a setn variable for later use. For example, if foo is being used by setn, and the user wants to save its current value on fie, (SETQ FOO FIE) is not sufficient, since the next setn on foo will also change fie, because it changes the word in number storage pointed to by foo, and hence pointed to by fie. The number must be copied, e.g., (SETQ FIE (IPLUS FOO)), which sets fie to a new word in number storage.

setn[var;x]

nlambda function like setq. var is quoted, x is evaluated, and its value must be a number. var will be set to this number. If the current value of var is a large integer or floating point number, that word in number storage is cannibalized. The value of setn is the (new) value of var.

13.6 BOX AND UNBOX IN INTERLISP-10

Some applications may require that a user program explicitly perform the boxing and unboxing operations that are usually implicit (and invisible) to most programs. The functions that perform these operations are loc and vag respectively. For example, if a user program executes a TENEX JSYS using the ASSEMBLE directive, the value of the ASSEMBLE expression will have to be boxed to be used arithmetically, e.g., (IPLUS X (LOC (ASSEMBLE --))). It must be emphasized that

Arbitrary unboxed numbers should not be passed around as ordinary values because they can cause trouble for the garbage collector.

For example, suppose the value of x were 150000, and you created (VAG X), and this just *happened* to be an address on the free storage list. The next garbage collection could be disastrous. For this reason, the function vag must be used with extreme caution when its argument's range is not known.

loc is the inverse of vag. It takes an address, i.e., a 36 bit quantity, and treats it as a number and boxes it. For example, loc of an atom, e.g., (LOC (QUOTE FOO)), treats the atom as a 36 bit quantity, and makes a number out of it. If the address of the atom FOO were 125000, (LOC (QUOTE FOO)) would be 125000, i.e., the location of FOO. It is for this reason that the box operation is called loc, which is short for location.¹¹

Note that FOO does not print as #364110 (125000 in octal) because the print routine recognizes that it is an atom, and therefore prints it in a special way, i.e., by printing the individual characters that comprise it. Thus (VAG 125000) would print as FOO, and would in fact be FOO.

loc[x]

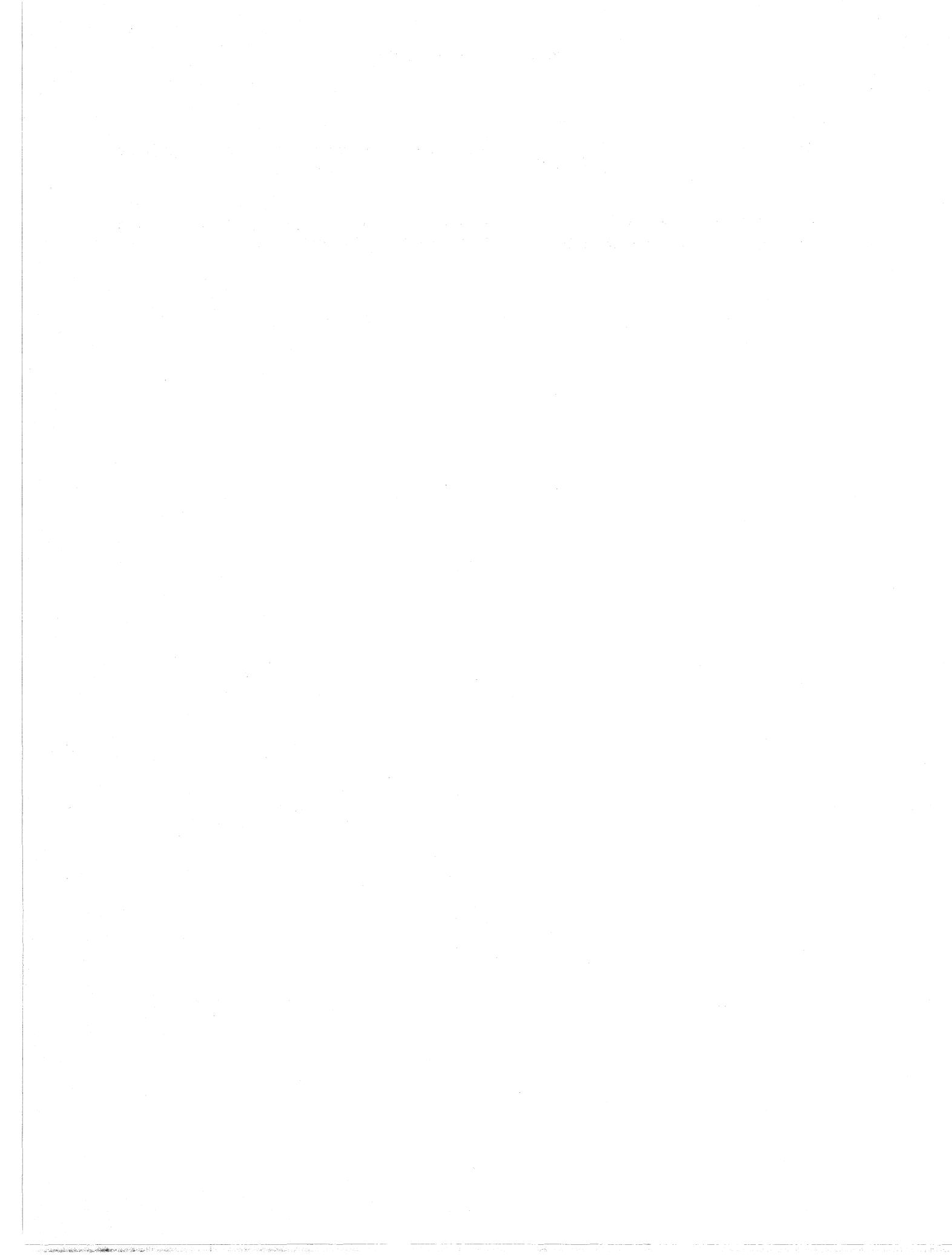
Makes a number out of x, i.e., returns the location of x.

11 vag is an abbreviation of value get.

vag[x]

The inverse of loc. x must be a number; the value of vag is the unbox of x.

The compiler eliminates extra vag's and loc's for example (IPLUS X (LOC (ASSEMBLE --))) will not box the value of the ASSEMBLE, and then unbox it for the addition.



Index for Section 13

	Page Numbers
ABS[X]	13.7
ADD1[X]	13.2
ANTILOG[X]	13.7
ARCCOS[X;RADIANSF LG]	13.8
ARCCOS: ARG NOT IN RANGE (error message)	13.8
ARCSIN[X;RADIANSF LG]	13.8
ARCSIN: ARG NOT IN RANGE (error message)	13.8
ARCTAN[X;RADIANSF LG]	13.8
ARCTAN2[X;Y;RADIANSF LG]	13.8
arithmetic functions	13.2-8
ASSEMBLE	13.10
box	13.10
boxed numbers	13.1
boxing	13.1,2,8-10
COS[X;RADIANSF LG]	13.7
DIFFERENCE[X;Y]	13.6
EQP[X;Y] SUBR	13.3,5,1,4
EQUAL[X;Y] SUBR	13.1
EXPT[M;N]	13.7
FDIFFERENCE[X;Y]	13.5
FGREATERP[X;Y] SUBR	13.5
FIX[X]	13.4
FIXP[X]	13.4
FLESSP[X;Y]	13.5
FLOAT[X]	13.6
floating point arithmetic	13.5-6
floating point numbers	13.1,3,9
FLOATP[X] SUBR	13.6
FMAX[X1;X2;...;Xn]	13.6
FMIN[X1;X2;...;Xn]	13.6
FMINUS[X]	13.5
FPLUS[X1;X2;...;Xn] SUBR*	13.5
FQUOTIENT[X;Y] SUBR	13.5
FREMAINDER[X;Y] SUBR	13.5
FTIMES[X1;X2;...;Xn] SUBR*	13.5
GCD[X;Y]	13.5
GEQ[X;Y]	13.6
GREATERP[X;Y] SUBR	13.6
IDIFFERENCE[X;Y]	13.2
IEQP[N;M] SUBR	13.3
IGEQ[X;Y]	13.3
IGREATERP[X;Y] SUBR	13.3
ILEQ[X;Y]	13.3
ILESSP[X;Y]	13.3
ILLEGAL EXPONENTIATION (error message)	13.7
IMAX[X1;X2;...;Xn] *	13.3
IMIN[X1;X2;...;Xn] *	13.3
IMINUS[X]	13.2
integer arithmetic	13.2-5
IPLUS[X1;X2;...;Xn] SUBR*	13.2
IQUOTIENT[X;Y] SUBR	13.3
IREMAINDER[X;Y] SUBR	13.3
ITIMES[X1;X2;...;Xn] SUBR*	13.2
large integers	13.1,9
LEQ[X;Y]	13.7
LESSP[X;Y]	13.6

Page
Numbers

LLSH[N;M] SUBR	13.4
LOC[X] SUBR	13.10, 10
LOG[X]	13.7
LOGAND[X1;X2;...;Xn] SUBR*	13.4
LOGOR[X1;X2;...;Xn] SUBR*	13.4
LOGXOR[X1;X2;...;Xn] SUBR*	13.4
LRSH[N;M]	13.4
LSH[N;M] SUBR	13.4
MAX[X1;X2;...;Xn] *	13.7
MAX.FLOAT (system variable/parameter)	13.6
MAX.INTEGER (system variable/parameter)	13.3
MIN[X1;X2;...;Xn] *	13.7
MINUS[X] SUBR	13.6
MINUSP[X] SUBR	13.3, 5
MIN.FLOAT (system variable/parameter)	13.6
MIN.INTEGER (system variable/parameter)	13.3
mixed arithmetic	13.6-7
NON-NUMERIC ARG (error message)	13.2, 5-6
numbers	13.1, 2-11
overflow	13.2, 5
PLUS[X1;X2;...;Xn] SUBR*	13.6
QUOTIENT[X;Y] SUBR	13.6
RAND[LOWER;UPPER]	13.8
random numbers	13.8
RANDSET[X]	13.8
REMAINDER[X;Y] SUBR	13.6
RSH[N;M]	13.4
SETN[VAR;X] FSUBR	13.10, 8-9
SIN[X;RADIANSLG]	13.7
small integers	13.1
SMALLP[N]	13.4, 1
SORT[N]	13.7
SQRT OF NEGATIVE VALUE (error message)	13.7
SUB1[X]	13.2
TAN[X;RADIANSLG]	13.8
TENEX	13.10
TIMES[X1;X2;...;Xn] SUBR*	13.6
unboxed numbers	13.10
unboxing	13.1, 2, 10
VAG[X] SUBR	13.11, 10
VALUE OUT OF RANGE EXPT (error message)	13.7
ZEROP[X]	13.3

SECTION 14

INPUT/OUTPUT FUNCTIONS

14.1 FILES

All input/output functions in Interlisp can specify their source/destination file with an optional extra argument which is the name of the file. This file must be opened as specified below. If the extra argument is not given (has value NIL), the file specified as "primary" for input (output) is used. Normally these are both T, for terminal input and output. However, the primary input/output file may be changed by

<code>input[file]¹</code>	Sets <u>file</u> as the primary input file. Its value is the name of the old primary input file.
	<code>input[]</code> returns current primary input file, which is not changed.
<code>output[file]</code>	Same as <u>input</u> except operates on primary output file.

Any file which is made primary must have been previously opened for input/output, except for the file T, which is always open.

<code>infile[file]</code>	Opens <u>file</u> for input, and sets it as the primary input file. ² The value of <u>infile</u> is the previous primary input file. If <u>file</u> is already open, same as <code>input[file]</code> . Generates a FILE WON'T OPEN error if <u>file</u> won't open, e.g., <u>file</u> is already open for <i>output</i> .
---------------------------	---

¹ The argument name file is used for tutorial purposes only. Subrs do not have argument "names", per se, as described in Section 8.0 Both input and infile can also be given a string as their argument, meaning that subsequent input operations referring to the primary input file will obtain characters from that string. +

² To open file without changing the primary input file, perform `openfile[file;INPUT]`.

outfile[file] Opens file for output, and sets it as the primary output file.³ The value of outfile is the previous primary output file. If file is already open, same as output[file]. Generates a FILE WON'T OPEN error if file won't open, e.g., if file is already open for input.

In Interlisp-10, for all input/output functions, file follows the TENEX⁴ conventions for file names, i.e., file can be prefixed by a directory name enclosed in angle brackets, can contain <esc>s or control-F's, and can include suffixes and/or version numbers. When a file is opened for input and no version number is given, the highest version number is used. Similarly, when a file is opened for output and no version number is given, a new file is created with a version number one higher than the highest one currently in use with that file name.

In Interlisp-10, regardless of the file name given to the Interlisp function that opened the file, Interlisp maintains only full file names⁵ in its internal table of open files and any function whose value is a file name always returns a full file name, e.g., openp[FOO]=<TEITELMAN>FOO. ;3. Whenever a file argument is given to an i/o function, Interlisp first checks to see if the file is in its internal table. If not, Interlisp executes the appropriate JSYS to "recognize" the file. If the operating system does not successfully recognize the file, a FILE NOT FOUND error is generated.⁶ If it does recognize the file, it returns to Interlisp the full file name. Then, Interlisp can continue with the indicated operation. If the file is being opened, Interlisp opens the file and stores its (full) name in the file table. If it is being closed, or written to or read from, Interlisp checks its internal table to make sure the file is open, and then executes the corresponding operation.

Note that each time a full file name is *not* used, Interlisp-10 must call the operating system to recognize the name. Thus if repeated operations are to be performed, it is considerably more efficient to obtain the full file name once, e.g., via infilep or outfilep. Also, note that recognition is performed on the user's entire directory. Thus, even if only one file is open, say FOO. ;1, F\$ (F altmode) will not be recognized if the user's directory also contains the file FIE. ;1. Similarly, it is possible for a file name that was previously recognized to become ambiguous. For example, a program performs infile[FOO], opening FOO. ;1, and reads several expressions from FOO. Then the user types control-C, creates a FOO. ;2 and reenters his program. Now a call to read giving it FOO as its file argument will generate a FILE NOT OPEN error, because FOO will be recognized as FOO. ;2.

infilep[file] Returns full file name of file if file is recognized as specifying the name of a file that can be opened for input, NIL otherwise. Recognition is in input context, i.e., in Interlisp-10, if no version number is given, the highest version number is returned.

³ To open file without changing the primary output file, perform openfile[file;OUTPUT].

+ ⁴ As mentioned in chapter 2, Interlisp-10 also runs under the TOPS-20 operating system. In this case, file follows the TOPS-20 conventions.

⁵ i.e., directory name, extension, and version.

⁶ except for infilep, outfilep and openp, which in this case return NIL.

+ ⁷ As described in Section 16, before the actual error occurs, it is intercepted via an entry on errortypelst, which causes spellfile (Section 17) to be called. spellfile will search alternate directories and possibly attempt spelling correction on the file name. Only if spellfile is unsuccessful will the error actually occur.

infilep and outfilep do not open any files, or change the primary files; they are pure predicates.

outfilep[file] Similar to infilep, except recognition is in output context, i.e., in Interlisp-10, if no version number is given, a version number one higher than the highest version number is returned.

A more general version of infilep and outfilep is provided by the function fullname:

fullname[x;recog] If x is recognized in the recognition mode specified by recog as an abbreviation for some file, value is the file's full name, otherwise NIL. recog can be OLD, NEW, OLDEST, or OLD/NEW.⁸ recog=NIL defaults to OLD. For all other values, generates an error ILLEGAL ARG. If x is not a literal atom, generates an error, ARG NOT LITATOM.

For example, infilep[file] could be written as fullname[file;OLD] and outfilep[file] as fullname[file;NEW].

A more general way of opening files is provided by the function openfile:

openfile[file;access;recog;bytesize;machine.dependent.parameters] Opens file with access rights as specified by access, one of INPUT, OUTPUT, BOTH, or APPEND. file is first recognized (in the fullname sense) using recognition mode recog. If recog=NIL, it defaults according to the value of access: for access=INPUT, recog=OLD is used; for access=OUTPUT, recog=NEW is used; for the other values of access, recog=OLD/NEW is used. If file cannot be recognized generates a FILE NOT FOUND error.⁹ Does not affect primary input or output files.

For example, infile[file] is equivalent to input[openfile[file;INPUT;OLD]]; outfile[file] is equivalent to output[openfile[file;OUTPUT;NEW]].

bytesize, if supplied, is the byte size in which to open the file (if bytesize=NIL, 7 is used). machine.dependent.parameters is a list of additional opening parameters. Interlisp-10 currently recognizes the following values:

WAIT wait if file is busy.
DON'T.CHANGE.DATE don't change the access dates.

⁸ OLD/NEW means if recognition fails using OLD, then use NEW (useful only for writing a file).

⁹ Various other errors may be generated in Interlisp-10 if the file cannot be opened, e.g. FILE WONT OPEN if the file is already opened by someone else, FILE SYSTEM RESOURCES EXCEEDED if there is no more room in the file system. See Section 16 for further discussion.

- + **THAWED** open file in "thawed" mode.
- + **iofile[file]** `openfile[file;BOTH;OLD]`, i.e. opens file for both input and output.
- + **openp[file;access]**
 - If access=NIL, value is file (full name) if file is open either for input or for output. Otherwise value is NIL.
 - If access is INPUT, OUTPUT or BOTH, value is file if open that access mode, otherwise NIL.

Note: the value of openp is NIL if file is not recognized, i.e., openp does not generate an error.

openp[] returns a list of all files open for input or output, excluding T and the current typescript (dribble) file, if any.
- + The function getfileinfo and setfileinfo allow the user to conveniently access such attributes of files as dates, protection and bytesize.¹⁰
- + **getfileinfo[file;attrib]** returns the current setting of the attrib attribute of file.¹¹ getfileinfo currently recognizes the following values for attrib:
- + **WRITEDATE, READDATE, CREATIONDATE** the date (and time) as a string that file was respectively last written, last read, and originally created.
- + **IWRITEDATE, IREADDATE, ICREATIONDATE** the respective date in integer form, e.g. `getfileinfo[file;IWRITEDATE]=idate[getfileinfo[file;WRITEDATE]]`.
- + **BYTESIZE** the byte size of the file.¹²
- + **LENGTH** the byte position of the end-of-file.¹³
- + **ACCESS** the current access mode of file (e.g. INPUT, OUTPUT, BOTH, APPEND) or NIL if file is not open.

¹⁰ getfileinfo and setfileinfo were written by R. M. Kaplan and J. J. Vittal.

¹¹ In Interlisp-10, file may also be a JFN as returned by gtfn.

¹² Note that in Interlisp-10, it is possible that the byte size for the "opening" of a file might differ from the "permanent" bytesize, i.e. a 7-bit text file can be opened in 36-bit mode. To obtain the "open" bytesize, use attribute OPENBYTE SIZE.

¹³ Like getcsptr[file], but file does not have to be open.

The following attributes are available in Interlisp-10:

SIZE	the size of <u>file</u> in pages.	+
PROTECTION	the "protection code" <u>file</u> as an integer.	+
DELETED	T, if <u>file</u> is the name of a deleted file, NIL otherwise.	+

Additional attributes which are available on TOPS-20 systems are:

INVISIBLE	T, if <u>file</u> has the invisible attribute, NIL otherwise.	+
ARCHIVED	T, if <u>file</u> has been archived, NIL otherwise.	+
OFF-LINE	T, if the contents of <u>file</u> are off-line (i.e. <u>file</u> has been archived and its contents flushed), NIL otherwise.	+

setfileinfo[file;attrib;value]	Sets the attribute <u>attrib</u> of <u>file</u> to be <u>value</u> . <u>setfileinfo</u> returns T if it is able to change the attribute <u>attrib</u> , and NIL if unsuccessful (some attributes cannot be changed, e.g. it doesn't make sense to change the SIZE of a file without writing something on it).	+
		+
		+
		+

closeff[file]	Closes <u>file</u> . ¹⁴ Generates an error, FILE NOT OPEN, if <u>file</u> not open. If <u>file</u> is NIL, it attempts to close the primary input file if other than terminal. Failing that, it attempts to close the primary output file if other than terminal. Failing both, it returns NIL. If it closes any file, it returns the name of that file. If it closes either of the primary files, it resets that primary file to terminal.	+
----------------------	--	---

closef?[file]	closes <u>file</u> if it is open, otherwise is a no-op. Value is <u>file</u> .	+
----------------------	--	---

closeall[-]	Closes all open files, except T and the current typescript file, if any. ¹⁵ Value is a list of the files closed.	*
--------------------	---	---

dclfile[file]	deletes <u>file</u> if possible. Value is <u>file</u> if deleted, else NIL.	+
----------------------	---	---

renamefile[old;new]	renames <u>old</u> to be <u>new</u> . Value is <u>new</u> if successful, else NIL.	+
----------------------------	--	---

¹⁴ The whenclose package described on page 14.9 permits the user to "advise" closef to perform various operations when a file is closed.

¹⁵ Note: the whenclose package (page 14.9) allows certain files to be "protected" from closeall. closeall[T] will override this protection.

+ MANIPULATING FILE NAMES

- + Different operating systems have different conventions for naming files. However, it is desireable for Interlisp to be as implementation independent as possible. Therefore, all operations that need to reference parts of a filename, or construct new file names from existing ones, use the functions filenamefield, packfilename, and unpackfilename, described below. The implementation of these functions obviously is dependent on the operating system they will run under, but as far as the programs that use them are concerned, they permit expressing operations that are implementation independent.¹⁶
- + Every file name is composed of a collection of *fields* which have different semantic interpretations.
- + A *field.name* is a literal atom which is the name of a file-name field. Interlisp assumes that **NAME** and **EXTENSION** are valid field names; the implementor is free to allow other fields. In Interlisp-10, allowable field names are: **DEVICE**, **DIRECTORY**, **NAME**, **EXTENSION**, **VERSION**, **PROTECTION**, **ACCOUNT**, and **TEMPORARY**.
- + **unpackfilename[filename]** returns a list of alternating fieldnames and field contents.

+ Examples for Interlisp-10 on Tenex:

+ filename	unpackfilename[filename]
+ <LISP>MAC.COM;3	(DIRECTORY LISP NAME MAC EXTENSION COM)
+ DSK:MYFILE	(DEVICE DSK: NAME MYFILE) ¹⁷
+ WORK.;T	(NAME WORK EXTENSION NIL TEMPORARY T)

- + **filenamefield[filename;fieldname]** returns the contents of the field fieldname of filename, i.e. equivalent to **listget[unpackfilename[filename];fieldname]**. For example, **filenamefield[<MASINTER>TEST DIRECTORY]** will return **MASINTER**.
- + **packfilename[fieldname₁;fieldcontents₁;...fieldname_n;fieldcontents_n]** lambda, nospread. Takes a list of alternating fieldnames and field contents (atoms or strings), and returns the corresponding file name. For example, **packfilename[DIRECTORY LISP NAME NET]** will return **<LISP>NET**.
- + If the first argument to packfilename is a list, packfilename is called on that argument. Thus packfilename and unpackfilename operate as inverses.

+ ¹⁶ In particular, the Interlisp-10 implementation recognizes file names in both Tenex and TOPS-20 format, and builds new names as appropriate.

+ ¹⁷ In Interlisp-10, **unpackfilename[DSK:FOO]** returns **(DEVICE DSK: NAME FOO)**, i.e. the : is left in. This is so **(DEVICE NIL:)** may be distinguished from **(DEVICE NIL)**.

Comments:

(1) If same field name is given twice, the *first* occurrence is used.

(2) packfilename recognizes the atom BODY as a fieldname meaning effectively splice into the argument list at that point all of the field names and contents specified in the operand to BODY.

For example, to take a file name and change a particular field, e.g. DIRECTORY, perform (PACKFILENAME 'DIRECTORY newdirectory BODY file). Alternatively, to provide a default for a field, e.g. the extension, perform (PACKFILENAME 'BODY file 'EXTENSION default). This will use default as the extension unless one is already specified in file.¹⁸

ADDRESSABLE FILES

For most applications, files are read starting at their beginning and proceeding sequentially, i.e., the next character read is the one immediately following the last character read. Similarly, files are written sequentially. A program need not be aware of the fact that there is a file pointer associated with each file that points to the location where the next character is to be read from or written to, and that this file pointer is automatically advanced after each input or output operation. This section describes a function which can be used to *reposition* the file pointer on those files that can be randomly accessed, thereby allowing a program to treat a file as a large block of auxiliary storage.¹⁹ For example, one application might involve writing an expression at the *beginning* of the file, and then reading an expression from a specified point in its *middle*.²⁰

A file used in this fashion is much like an array in that it has a certain number of addressable locations that characters can be put into or taken from. However, unlike arrays, files can be enlarged. For example, if the file pointer is positioned at the end of a file and anything is written, the file "grows." It is also possible to position the file pointer *beyond* the end of file and then to write.²¹ In this case, the file is enlarged, and a "hole" is created, which can later be written into. Note that this enlargement only takes place at the *end* of a file; it is not possible to make more room in the middle of a file. In other words, if expression A begins at position 1000, and expression B at 1100, and the program attempts to overwrite A with expression C, which is 200 characters long, part of B will be clobbered.

getfileptr[file]

returns current value of file pointer for file, i.e., the byte address at which the next input/output operation will commence. getfileptr compiles open.

¹⁸ Note that a null field is a field that *has* been specified, e.g. if file=FOO;1 in the above example, the default extension will be used, but if file=FOO.;1, it will not, because a null extension has been specified.

¹⁹ The pmap facility (Section 21) provides for paged access to binary files in Interlisp-10.

²⁰ This particular example requires the file be open for *both* input and output. This can be achieved via the function iosfile. However, random file input or output can be performed on files that have been opened in the usual way by infile or outfile.

²¹ If the program attempts to *read* beyond the end of file, an END OF FILE error occurs.

`setfileptr[file;adr]` sets file pointer for file to adr.²² Value is adr. adr=-1 corresponds to the end of file.²³

`geteofptr[file]` Value is byte address of end of file, i.e., the number of bytes in the file. Equivalent to performing `setfileptr[file;-1]` and returning `getfileptr[file]` except does not change the current file pointer.

+ `randaccessp[file]` value is file if file is randomly accessible, NIL otherwise, e.g. T, LPT:, NIL:, etc. are examples of files *not* randomly accessible. file must be open or an error is generated, FILE NOT OPEN.

`filepos[pattern;filename;filestart;fileend;skip;tail;casearray]`

Searches file for pattern a la strpos (Section 10). Search begins at filestart (or if filestart=NIL, the current position of the file pointer), and goes to fileend (or if fileend=NIL, to the end of file). Value is *address* of start of match, or NIL if not found. skip can be used to specify a character which matches any character in the file. If tail is T, and the search is successful, the value is the address of the first character *after* the sequence of characters corresponding to pattern, instead of the starting address of the sequence. In either case, the file is left so that the next i/o operation begins at the address returned as the value of filepos.

+ casearray, if supplied, should be an array of 128 integers (as created by `array[128;128]`). Each character in the file is mapped "thru" casearray in the sense that character code i is turned into elt[casearray;i+1] before matching. casearray=NIL means no transformation will be performed.

+ For example, to search without regard to upper and lower case differences, casearray would be an array where elt[cascarray;i]=j, except for lower case characters whose corresponding elements would be the upper case characters. To search for a delimited atom, one could use "atom" as the pattern, and specify a casearray in which all of the break and separator characters mapped into a space.

²² The address of a character (byte) is the number of characters (bytes) that precede it in the file, i.e., 0 is the address of the beginning of the file. However, the user should be careful about computing the space needed for an expression, since end-of-line in Interlisp-10 is represented as two characters in a file, but nchars only counts it as one.

²³ Note: in Interlisp-10, if a file is opened for output only, either by outfile, or `openfile[file;OUTPUT]`, TENEX assumes that one intends to write a new or different file, even if a version number was specified and the corresponding file already exists. Thus, `setfileptr[file;-1]` will set the file pointer to 0. If a file is opened for both reading and writing, either by iofile or `openfile[file;BOTH]`, TENEX assumes that there might be material on the file that the user intends to read. Thus, the initial file pointer is the beginning of the file, but `setfileptr[file;-1]` will set it to the end of the file. Note that one can also open a file for appending by `openfile[file;APPEND]`. In this case, the file pointer right after opening is set to the end of the existing file. Thus, a write will automatically add material at the end of the file, and an setfileptr is unnecessary.

seprcase[clispflg]

returns a casearray suitable for use by filepos (or ffilepos described below) in which all of the break/separators of filerdtbl are mapped into the same character. If clispflg is non-NIL, then all clisp characters will be mapped into this character as well. Useful for finding a delimited atom in a file. For example, if pattern is "FOO", and seprcase[T] is used for casearray, then filepos will find "(FOO←".

For applications calling for extensive file searches, the function ffilepos will generally be 10 to 50 times faster than filepos.

ffilepos[pattern;filename;filestart;fileend;skip;tail;casearray]

Like filepos, except much faster in most applications. ffilepos is an implementation of the Boyer-Moore fast string searching algorithm [Boy]. This algorithm preprocesses the string being searched for and then scans through the file in steps usually equal to the length of the string. Thus, ffilepos speeds up roughly in proportion to the length of the string, e.g., a string of length 10 will be found twice as fast as a string of length 5 in the same position.

Because of certain fixed overheads, it is generally better to use filepos for searches through fewer than 100 characters of the file. For longer searches, ffilepos will generally be marginally faster for short strings (length 3 or less) and significantly faster for longer strings.

copybytes[srcfil;dstfil;start;end] copies bytes, i.e., characters, from srcfil to dstfil starting from position start and up to but not including position end. Both srcfil and dstfil must be open. Value is T. If end=NIL, start is interpreted as the number of bytes to copy (starting at the current position). If start is also NIL, bytes are copied until end-of-file is reached.

CLOSING AND REOPENING FILES - THE WHENCLOSE PACKAGE²⁴

This package permits the user to associate with open files certain operations that govern how and when the file will be closed, and how the file's status will be restored when a sysout is started up. The user can specify that certain functions will be executed before closef closes the file and/or after closef closes the file. The user can make a particular file be invisible to closeall, so that it will remain open across user invocations of closeall. Finally, the user can associate a status-saving function with a file which will be called before sysout and which can specify what to do when a sysout is restarted.

The basic user entry to these capabilities is the function whenclose.

²⁴ The whenclose package was written by R. M. Kaplan and L. M. Masinter.

+ whenclose[file;prop₁;val₁;...;prop_n;val_n]

+ lambda no-spread. The first argument of whenclose must specify
+ the name of an open file other than T (NIL defaults to the non-T
+ primary input file or non-T primary output file). The remaining
+ arguments specify properties to be associated with the full name of
+ file. whenclose returns the full name of file as its value. The
+ property names are drawn from the following set:

+ BEFORE

+ val is a function that closef will apply to the full name of file just
+ before it is closed. This might be used, for example, to copy
+ information about the file from an in-core data structure to the file
+ just before it is closed.

+ AFTER

+ val is a function that closef will apply to the full name of file just
+ after it is closed. This capability permits in-core data structures that
+ know about the file to be cleaned up when the file is closed.²⁵

+ For example, the pmap package (Section 21) keeps track of the
+ pages from an open file that have been page-mapped into in-core
+ buffers. Those buffers must be cleared when a file is closed, or the
+ file's JFN will not get released. Thus, when a file is first "mapped"
+ in, whenclose[file;AFTER;PMAPAFTERCLOSE] is performed, where
+ pmapafterclose is a function which clears the buffer of any pages
+ into which its argument is mapped.

+ STATUS

+ This property provides a way of restoring the status of files when a
+ sysout is resumed. val is a function that will be applied to the full
+ name of file just before a sysout. val is expected to return a list,
+ car of which is a function which will be apply'd to the cdr when
+ the sysout is started up and which will restore the status of file. If
+ the value of the apply is NIL, it is assumed the file could not be
+ successfully restored, a warning message is printed, and then any
+ AFTER functions associated with the file are executed.²⁶

+ CLOSEALL

+ val is either YES or NO and determines whether file will be closed
+ by closeall (YES) or whether closeall will ignore it (NO).²⁷

+ EOF

+ val is a function that will be applied to the full name of file when
+ an end-of-file error occurs.²⁸ The function can examine the context
+ of the error, and can decide whether to close the file, retfrom some

+ 25 BEFORE and AFTER also differ in their behaviour with respect to sysout: The AFTER function will also be
+ executed when a sysout is started up, unless a STATUS function is also associated with the file and it causes the file
+ to be successfully restored.

+ 26 permstatus (Section 24) produces an expression for re-opening a file after sysout and restoring as many of its
+ attributes as possible.

+ 27 closeall uses closef, so that any AFTER functions will be executed if the file is in fact closed.

+ 28 and the errortypest entry for that error, if any, returns NIL.

function, or perform some other computation. If the function supplied returns normally (i.e. does not retfrom some function), the normal error machinery will be invoked (but file will not be automatically closed if the EOF function did not close it).

Note that multiple AFTER and BEFORE functions may be associated with a file; they are executed in sequence with the most recently associated function executed first. However, a second STATUS specification will supercede an earlier one. The CLOSEALL and EOF values will also override earlier values, so only the last value specified will have an effect. Files are initialized with CLOSEALL - YES, EOF - CLOSEF.

14.2 INPUT FUNCTIONS

Most of the functions described below have an (optional) argument file which specifies the name of the file on which the operation is to take place, and an (optional) argument rdtbl, which specifies the readable to be used for input. If file is NIL, the primary input file will be used. If the file argument is a string, input will be taken from that string (and the string pointer reset accordingly). If rdtbl is NIL, the primary readable will be used. readtables are described on page 14.23.

Note: in all Interlisp-10 symbolic files, end-of-line is indicated by the characters carriage-return and line-feed in that order. Accordingly, on input from files, Interlisp-10 will skip all line-feeds which immediately follow carriage-returns. On input from terminal, Interlisp will echo a line-feed whenever a carriage-return is input.

For all input functions except readc and peekc, when reading from the terminal, control-A erases the last character typed in, echoing a \ and the erased character. Control-A will not backup beyond the last carriage-return. Typing control-Q causes Interlisp to print ## and clear the input buffer, i.e., erase the entire line back to the last carriage-return.²⁹ When reading from a file, and an end of file is encountered, all input functions close the file and generate an error, END OF FILE.³⁰

read[file;rdtbl;flg]

Reads one S-expression from file. Atoms are delimited by the break and separator characters as defined in rdtbl. To input an atom which contains a break or separator character, precede the character by the escape character %, e.g., AB%(C, is the atom AB(C, %% is the atom %, %↑A (i.e., %control-A) is the atom ↑A. For input from the terminal, an atom containing an interrupt character can be input by typing instead the corresponding alphabetic character preceded by control-V, e.g., ↑VC for control-C.

Strings are delimited by double quotes. To input a string containing a double quote or a %, precede it by %, e.g., "AB%"C" is the string AB"C. Note that % can always be typed even if next character is not "special", e.g., %ABC% is read as ABC.

²⁹ Note that the CHARDELETE and LINEDELETE characters can be redefined or disabled via setsyntax, see page 14.25.

³⁰ unless the whenclose package has been used to alter this behavior (see page 14.9).

If an atom is interpretable as a number, read will create a number, e.g., 1E3 reads as a floating point number, 1D3 as a literal atom, 1.0 as a number, 1,0 as a literal atom, etc. Note that an integer can be input in octal by terminating it with a Q, e.g., 17Q and 15 read in as the same integer. The setting of radix, page 14.34, determines how integers are printed, i.e., with or without Q's.

When reading from the terminal, all input is line-buffered to enable the action of control-Q.³¹ Thus no characters are actually seen by the program until a carriage-return is typed.³² However, for reading by read, when a matching right parenthesis is encountered, the effect is the same as though a carriage-return were typed, i.e., the characters are transmitted.³³ To indicate this, Interlisp also prints a carriage-return line-feed on the terminal.

flg=T suppresses the carriage-return normally typed by read following a matching right parenthesis. (However, the characters are still given to read - i.e., the user does not have to type the carriage-return himself.)

When reading a list, typing control-W erases the last expression read, echoing a \\ and the erased expression, e.g., (NOW IS THE TIME^W \\ TIME) returns (NOW IS THE). Control-W can be used repeatedly, and can also back up and erase expressions on previous lines.³⁴

ratom[file;rdtbl]

Reads in one atom from file. Separation of atoms is defined by rdtbl. % is also an escape character for ratom, and the remarks concerning control-A, control-Q, control-V, and line-buffering also apply.

If the characters comprising the atom would normally be interpreted as a number by read, that number is also returned by ratom. Note however that ratom takes no special action for "whether or not it is a break character, i.e., ratom never makes a string.

rstring[file;rdtbl]

Reads in one string from file, terminated by next break or separator

³¹ Unless control[T] has been performed (page 14.33).

³² Actually, the line buffering is terminated by the character with terminal syntax class EOL (see page 14.30), which in most cases will be carriage-return.

³³ The line buffer is also transmitted to read whenever an IMMEDIATE read-macro character is typed (see page 14.27).

³⁴ However, since control-W is implemented as an IMMEDIATE read-macro character, (see page 14.27), once it is typed, characters typed before it cannot be deleted by control-A or control-Q, since they will already have passed through the line buffer.

character.³⁵ Control-A, control-Q, control-V, and % have the same effect as with ratom.

Note that the break or separator character that terminates a call to ratom or rstring is not read by that call, but remains in the buffer to become the first character seen by the next reading function that is called.

ratoms[a;file;rdtbl] Calls ratom repeatedly until the atom a is read. Returns a list of atoms read, not including a.

setsepr[lst;flg;rdtbl] Set separator characters for rdtbl. Value is NIL.

setbrk[lst;flg;rdtbl] Set break characters for rdtbl. Value is NIL.

For both setsepr and setbrk, lst is a list of character codes,³⁶ flg determines the action of setsepr/setbrk as follows:

- NIL clear out indicated readable and reset break/separator characters to be those in lst.
- 0 clear out only those characters in lst - i.e., this provides an unsetsepr and unsetbrk.
- 1 add characters in lst.

Characters specified by setbrk will delimit atoms, and be returned as separate atoms themselves by ratom. Characters specified by setsepr will serve only to delimit atoms, and are otherwise ignored. For example, if \$ was a break character and * a separator character, the input stream ABC**DEF\$GH*\$ would be read by 6 calls to ratom returning respectively ABC, DEF, \$, GH, \$, \$.

The elements of lst may also be characters, e.g., setbrk[(.,)] has the same effect in Interlisp-10 as setbrk[(46 44)]. Note however that the "characters" 1,2,...,9 will be interpreted as character *codes* because they are numbers.

Note: (,), [,], and " are normally break characters, i.e., will be returned as separate atoms when read by ratom. If any of these break characters are disabled by an appropriate setbrk (or by making it be a separator character), its special action for read will *not* be restored by simply making it be a break character again with setbrk.³⁷ For more details, see discussion in section on readtables, page 14.23-26.

³⁵ Note that if the next character is a break or separator, rstring will return the null string, "". This is a common source of program bugs. + +

³⁶ If lst=T, the break/separator characters are reset to be those in the system's readable for terminals, regardless of value of flg, i.e., setbrk[T] is equivalent to setbrk[getbrk[T]]. If rdtbl is T, then the characters are reset to those in the original system table.

³⁷ However, making these characters be break characters when they already are will have no effect.

Note that the action of % is not affected by setsepr or setbrk. To defeat the action of % use escape[], as described below.

<u>getsepr[rdtbl]</u>	Value is a list of separator character codes.
<u>getbrk[rdtbl]</u>	Value is a list of break character codes.
<u>escape[flg]</u>	If <u>flg=NIL</u> , makes % act like every other character for <i>input</i> . Normal setting is <u>escape[T]</u> . The value of <u>escape</u> is the previous setting.
<u>ratest[x]</u>	If <u>x = T</u> , <u>ratest</u> returns T if a separator was encountered immediately prior to the last atom read by <u>ratom</u> , NIL otherwise. If <u>x = NIL</u> , <u>ratest</u> returns T if last atom read by <u>ratom</u> or <u>read</u> was a break character, NIL otherwise. If <u>x = 1</u> , <u>ratest</u> returns T if last atom read (by <u>read</u> or <u>ratom</u>) contained a % (as an escape character, e.g., %[or %A%B%C), NIL otherwise.
<u>readc[file]</u>	Reads the next character, including %, ", etc, i.e., is not affected by break, separator, or escape character. Value is the character. Action of <u>readc</u> is subject to line-buffering, i.e., <u>readc</u> will not return a value until the line has been terminated even if a character has been typed. Thus, control-A, control-Q, and control-V will have their usual effect. If <u>control[T]</u> has been executed (page 14.33), defeating line-buffering, <u>readc</u> will return a value as soon as a character is typed. In addition, if control-A, control-Q, or control-V are typed, <u>readc</u> will return them as values.
<u>peekc[file;rdtbl]</u>	Value is the next character, but does not actually read it, i.e., remove it from the buffer. If <u>rdtbl=NIL</u> , <u>peekc</u> is not subject to line-buffering, ³⁸ i.e., it returns a value as soon as a character has been typed. Otherwise, <u>peekc</u> waits until the line has been terminated before returning its value. ³⁹ This means that control-A, control-Q, and control-V will be able to perform their usual editing functions.

+ 38 If reading from the terminal, the character is echoed as soon as peekc reads it, even though it is then "put back" into the system buffer, where a subsequent < del > (or < control-Z > on TOPS-20) before the character is read can clear it.

39 rdtbl is used to determine the escape character, parentheses characters, etc., while the line is being buffered.

`lastc[file]` Value is last character read from file.

Note: read, ratom, ratoms, peekc, readc all wait for input if there is none. The only way to test whether or not there is input is to use readp.

`readp[file;flg]` Value is T if there is anything in the input buffer of file, NIL otherwise.⁴⁰ Note that because of line-buffering, readp may return T, indicating there is input in the buffer, but read may still have to wait.

`waitForInput[file]` waits until input is available from file or from the terminal, i.e. from T. waitForInput is functionally equivalent to `(until (OR (READP T) (READP FILE)) do NIL)`, except that it does not use up machine cycles while waiting.⁴¹ Value is the device for which input is now available, i.e. file or T.

file can also be an integer, in which case waitForInput will wait until there is input available from the terminal, or until file milliseconds have elapsed. Value is T if input is now available, NIL in the case that waitForInput timed out.

`readline[rdtbl;-;-]` reads a line from the terminal, returning it as a list. If readp[T] is NIL, readline returns NIL. Otherwise it reads expressions, using read,⁴² until it encounters either:

- (1) a carriage-return (typed by the user) that is not preceded by any spaces, e.g.,
A B C
and readline returns (A B C)⁴³
- (2) a list terminating in a "]", in which case the list is included in the value of readline, e.g., A B (C D] and readline returns (A B (C D)).
- (3) an unmatched right parentheses or right square bracket, which is not included in the value of readline, e.g.,

⁴⁰ Frequently, the input buffer will contain a single EOL character left over from a previous input. For most applications, this situation wants to be treated as though the buffer were empty, and so readp returns NIL. However, if flg=T, readp will also return T in this case, i.e., will return T if there is *any* character in the input buffer.

⁴¹ waitForInput operates by dismissing, checking for available input, and then, if there is none, dismissing again, each time for an increasingly larger interval. The initial interval is the dismissinit milliseconds (initially 500), and the interval grows by 1/16 for each dismissal, up to a maximum of dismissmax milliseconds (initially 10,000).

⁴² Actually, readline performs `(APPLY* LISPXREADFN T)`, as described in Section 22. lispxreadfn is initially READ.

⁴³ Note that carriage-return, i.e., the EOL character, can be redefined with setsyntax, page 14.25. readline actually checks for the EOL character, whatever that may be. The same is true for right parenthesis and right bracket.

A B C]
and readline returns (A B C).

In the case that one or more spaces precede a carriage-return, or a list is terminated with a ")", readline will type "..." and continue reading on the next line,⁴⁴ e.g.,

A B C_2
... (D E F)
... (X Y Z]

and readline returns (A B C (D E F) (X Y Z)).

skread[file;rereadstring]⁴⁵

is a skip read function. It moves the file pointer for file ahead as if one call to read had been performed, without paying the storage and compute cost to really read in the structure. rereadstring is for the case where the user has already performed some readc's and ratom's before deciding to skip this expression.⁴⁶ In this case, rereadstring should be the material already read (as a string), and skread operates as though it had seen that material first, thus getting its paren-count, double-quote count, etc. set up properly.

The value of skread is %) if the first thing encountered was a closing paren; %] if the read terminated on an unbalanced %], i.e., one which also would have closed any extant open left parens; otherwise the value of skread is NIL.

⁴⁴ If the user then types another carriage-return, the line will terminate, e.g.,

A B C_2
... 2

and readline returns (A B C).

⁴⁵ skread was written by J. W. Goodwin. It always uses filerdtbl for its readable.

* ⁴⁶ skread may have difficulties if unusual read-macros have been added to filerdtbl. skread will not recognize read-macro characters in rereadstring, nor SPLICE or INFIX read macros. This is only a problem if the read-macros are defined to parse subsequent input in the file which does not follow the normal parenthesis and string-quote conventions in filerdtbl.

14.3 OUTPUT FUNCTIONS

Most of the functions described below have an (optional) argument file which specifies the name of the file on which the operation is to take place. If file is NIL, the primary output file will be used. Some of the functions have an (optional) argument rdtbl, which specifies the readtable to be used for output. If rdtbl is NIL, the primary readtable will be used.

Note: in all Interlisp-10 symbolic files, end-of-line is indicated by the characters carriage-return and line-feed in that order. Unless otherwise stated, carriage-return appearing in the description of an output function means carriage-return and line-feed.

prin1[x;file] prints x on file.

prin2[x;file;rdtbl] prints x on file with %'s and "'s inserted where required for it to read back in properly by read, using rdtbl.

Both prin1 and prin2 print lists as well as atoms and strings; prin1 is usually used only for explicitly printing formatting characters, e.g., (PRIN1 (QUOTE %[)) might be used to print a left square bracket (the % would not be printed by prin1). prin2 is used for printing S-expressions which can then be read back into Interlisp with read i.e., break and separator characters in atoms will be preceded by %'s, e.g., the atom "(" is printed as %()% by prin2. If radix=8, prin2 prints a Q after integers but prin1 does not (but both print the integer in octal).⁴⁷

print[x;file;rdtbl] Prints the S-expression x using prin2; followed by a carriage-return line-feed. Its value is x.

For all printing functions, pointers other than lists, strings, atoms, or numbers, are printed as #N,⁴⁸ where N is the octal representation of the address of the pointer (regardless of radix). Note that this will not read back in correctly, i.e., it will read in as the atom "#N".

spaces[n;file] Prints n spaces; its value is NIL.

terpri[file] Prints a carriage-return; its value is NIL.⁴⁹

tab[pos;minspaces;file] performs appropriate number of spaces to move to position pos.

⁴⁷ prin3 and prin4 are the same as prin1 and prin2 respectively, except that they do not increment the position counter nor perform any linelength checks. They are useful primarily for printing control characters. +

⁴⁸ unless desprint (page 14.22) has been used to specify a printing function for this particular data type. +

⁴⁹ Note: a sequence of print, prin2, spaces, and terpri expressions can often be more conveniently coded with a single printout statement (Section 23). +

- * `minspaces` indicates how many spaces must be printed (if `NIL`, 1 is used). Thus, if position + `minspaces` is greater than `pos`, `tab` does a `terpri` and then `spaces[pos]`. Note: if `minspaces` is `T`, and position is greater than `pos`, `tab` does nothing.
- + `showprin2[;x;file;rdtbl]` like `prin2` except if `sysprettyflg=T`, prettyprints `x` instead. Value is `x`.
- + `showprint[;x;file;rdtbl]` like `print` except if `sysprettyflg=T`, prettyprints `x` instead, followed by a carriage-return line-feed. Value is `x`.
- + `showprint` and `showprin2` are used by the programmer's assistant (Secton 22) for printing the values of expressions and for printing the history list, by various commands of the break package (Section 15), e.g. `?=` and `BT` commands, and various other system packages. The idea is that by simply setting or binding `sysprettyflg` to `T` (initially `NIL`), the user instructs the system when interacting with the user to prettyprint s-expressions (see page 14.43) instead of printing them.
- + `printbells[]` used by DWIM (Section 17) to print a sequence of bells to alert the user to stop typing. Can be advised or redefined for special applications, e.g. to flash the screen on a display terminal.
- + `dobe[]` dismiss until `output buffer` is `empty`, i.e. until all of the characters that have been printed by Interlisp functions have actually been printed on the user's terminal. For example, it is important to perform a `dobe` after printing an error message before clearing the input buffers to make sure that the user has actually seen the error message.

PRINTLEVEL

The print functions `print`, `prinl`, and `prin2` are all affected by a level parameter set by:

- * `printlevel[carn;cdrn]` Sets `car` print level to `carn`, `cdr` printlevel to `cdrn`. Value is dotted pair of old settings. Initial value is `(1000 . -1)`.
- + In order that `printlevel` can be used with `resetform` or `resetsave`, `printlevel[x]` where `x` is a dotted pair is equivalent to `printlevel[car[x];cdr[x]]`.
- + `printlevel[n;NIL]` changes `car` printlevel without affecting `cdr` printlevel. `printlevel[NIL;n]` changes `cdr` printlevel with affecting `car` printlevel. `printlevel[]` gives current setting without changing either.

The car printlevel controls the number of unpaired left parentheses which will be printed. Below that level, all lists will be printed as &. For example, if x = (A (B C (D (E F) G) H) K). Then if carn = 2, print[x] would print (A (B C & H) K), and if carn = 3, (A (B C (D & G) H) K), and if carn = 0, just &.

If car printlevel is *negative*, the action is similar except that a carriage-return is inserted between all occurrences of right parenthesis immediately followed by a left parenthesis.

The cdr printlevel affects the number of elements of a list that will be printed before the printing is terminated with --. For example, if cdrn=2, (A B C D E) will print as (A B --). The decision to terminate printing of a tail of a list is also affected by the depth of printing in the car direction: whenever the sum of the number of cdrs at the current level and number of cars, i.e. unmatched left parentheses, is greater than the cdr printlevel, printing stops. This gives a "triangular" effect in that less is printed the farther one goes in either car or cdr direction. For example, if cdr printlevel=2, then (A (B C (D (E F) G) H) K L) will print as (A (B --) --) and if cdr printlevel=3, as (A (B C --) K --).

If cdr printlevel is negative, then the action of cdr printlevel is disabled, i.e. same as though cdr printlevel were infinite.

The printlevel setting can be changed dynamically, even while Interlisp is printing, by typing control-P followed by a number, i.e., a string of digits, followed by a period or exclamation point.⁵⁰ The car printlevel will immediately be set to this number.⁵¹ If the print routine is currently deeper than the new level, all unfinished lists above that level will be terminated by "--". Thus, if a circular or long list of atoms, is being printed out, typing control-P0. will cause the list to be terminated.

If the string of digits following a control-P is terminated by a comma, another number may be typed terminated by a period or exclamation point. The car printlevel will then be set to the first number, the cdr printlevel to the second number.

In either case, if a period is used to terminate the printlevel setting, the printlevel will be returned to its previous setting after the current printout has finished. If an exclamation point is used, the change is permanent and the printlevel is not restored (until it is changed again).

Note: normally printlevel only affects terminal output. Output to all other files acts as though level is infinite. However, if plvlfleflg is T (initially NIL), then printlevel will affect output to files.

⁵⁰ As soon as control-P is typed, Interlisp clears and saves the input buffer, clears the output buffer, rings the bell indicating it has seen the control-P, and then waits for input which is terminated by any non-number. The input buffer is then restored and the program continues. If the input was terminated by other than a period or an exclamation point, it is ignored and printing will continue, except that characters cleared from the output buffer will have been lost.

⁵¹ Another way of "turning off" output is to type control-O, which simply clears the output buffer, thereby effectively skipping the next (up to) 64 characters.

+ PRINTING NUMBERS⁵²

+ Interlisp defines standard print-names for fixed and floating point numbers. These print-names are used whenever numbers are printed by the ordinary printing functions `prin1`, `prin2`, etc. The print-names of numbers normally can be affected in two ways: the print-name for integers is sensitive to the current setting of `radix`, and the print-name for floating numbers is sensitive to the current setting of `fltfmt`. The facilities described in this section permit greater controls on the printed appearance of numbers, allowing such things as left-justification, suppression of trailing decimals, etc.

+ The basic function for printing numbers under format control is `printnum`:

+ `printnum[format;number;file]` prints number on file according to the format `format`. `format` is a list structure of one of the following two forms:⁵³

+ `(FIX width radix pad leftflush)`

+ `(FLOAT width decpart exppart pad round)`

+ For a FIX format, `number` will be rounded to the nearest integer, and then will be printed in a field `width` characters long with `radix=radix` (or 10 if `radix=NIL`). If `pad` and `leftflush` are both NIL, the number will be right-justified in the field, and the padding characters to the left of the leading digit will be spaces. If `pad` is T, the character "0" will be used for padding. If `leftflush` is T, then the number will be left-justified in the field, with trailing spaces to fill out `width` characters.

+ The following examples illustrate the effects of the FIX format options (the vertical bars indicate the field width):

<u>format</u>	<u>number</u>	<u>result</u>
+ <code>(FIX 2)</code>	3	3
+ <code>(FIX 2 NIL T)</code>	7	07
+ <code>(FIX 12 -8 T)</code>	14	000000000016
+ <code>(FIX 5 NIL NIL T)</code>	2	2

+ For a FLOAT format, `number` will be printed as a decimal number

+ 52 The `printnum` package was designed and implemented by R. M. Kaplan. Its utility is considerably enhanced when used in conjunction with the `printout` package (Section 23), which implements a compact language for specifying complicated sequences of elementary printing operations, and makes fancy output formats easy to design and simple to program.

+ 53 `format` can also be a machine dependent format-code as returned by `numformatcode`. See discussion on page 14.22.

in a field width characters wide. There will be decpart digits to the right of the decimal point (decpart=NIL is equivalent to 0). If exppart is not 0 (or NIL), the number will be printed in exponent notation, with the exponent occupying exppart characters in the field. exppart must allow for the character E and a sign that will be printed before the exponent digits. As with FIX format, padding on the left will be with spaces, unless pad is T. If round is given, it indicates the digit position at which rounding is to take place, counting from the leading digit of the number.

- FLOAT format examples:

<u>format</u>	<u>number</u>	<u>result</u>
(FLOAT 7 2)	27.689	27.69
(FLOAT 7 2 NIL T)	27.689	0027.69
(FLOAT 11 2 2)	27.689	2768.90E-02
(FLOAT 7 2 NIL NIL 1)	27.689	30.00
(FLOAT 7 2 NIL NIL 2)	27.689	28.00

If number is not a number and not NIL, a NON-NUMERIC ARG error will be generated. If number is NIL, the effect depends on the setting of the variable nilnumprintflg. If nilnumprintflg is NIL, then the error will occur. If it is non-NIL, then the value of nilnumprintflg will be printed right-justified in the field described by format. This option facilitates the printing of numbers in aggregates with missing values coded as NIL.

The full print-name number will be printed if the value will not fit in the field specified. Then a tab is executed so that the line position of the file after printnum is always the position prior to printing plus the indicated width.

Formatted printing of numbers usually receives assistance from the operating system, provided that the format is specified in some sort of special code. printnum works by converting the machine-independent format specifications described above into machine-dependent codes the exact form of which may vary from implementation to implementation. This conversion process takes place on each call to printnum. For efficiency purposes, if the user is going to be performing a particular call to printnum frequently, he may wish to separate the conversion from the actual printing, performing the conversion process just once and saving the result. The function numformatcode is available for this purpose: numformatcode takes a format, performs the conversion and returns a machine dependent format-code, which can be given to printnum in place of a list structure format as described above. In this case, printnum will not have to perform the conversion, but can simply use the machine-dependent format code directly.

- + `numformatcode[format;-]` converts the FIX or FLOAT format to a machine-dependent format-code.⁵⁴
- + Note: in Interlisp-10, the function `fltfmt` (page 14.35) will accept FLOAT formats as described above, in addition to the bit specification required for the Tenex FLOUT JSYS.

+ USER DEFINED PRINTING

- + `defprint[type;fn]` In Interlisp-10, `type` is either a type name⁵⁵ or type number. Whenever a printing function, `print`, `prin1`, `prin2`, etc. encounters an object of the indicated type, `fn` is called with the item to be printed as its argument. If it returns NIL, the datum is printed as the system defaults, i.e. for user data types, it will be printed as #nnnnnn. If `fn` wishes to specify how the datum should be printed, it should return a list of the form (item1 . item2). `item1` will be printed using `prin1` (unless it is NIL), and then `item2` printed using `prin2` with no spaces between the two items. (Typically, `item1` will be a read macro character.)
- + Note that `defprint` also affects internal calls to print from `pack`, `concat`, etc., i.e. any operation that involves obtaining a print name (see Section 10).

DUMPING UNUSUAL DATA STRUCTURES

Hprint⁵⁶ is a package for printing and reading back in more general data structures that cannot normally be dumped and loaded easily, e.g., (possibly re-entrant or circular) structures containing user datatypes, arrays, hash tables, as well as list structures.⁵⁷ Hprint will correctly print and read back in any structure containing any or all of the above, chasing all pointers down to the level of literal atoms, numbers or strings.

Hprint operates by simulating the Interlisp `print` routine for normal list structures. When it encounters a user datatype (see Section 23), or an array or hash array, it prints the data contained therein, surrounded by special characters defined as read-macro characters (see page 14.26). While chasing the pointers of a structure, it also keeps a hash table of those items it encounters, and if any item is encountered a second time, another read-macro character is inserted before the

+ 54 In Interlisp-10, `numformatcode` has a second argument, `smashcode`. If `smashcode` is recognized as a format-code data-structure, then the new format-code will be smashed into that structure instead of allocating new storage. `numformatcode[]` will simply return an uninitialized datum that can be later smashed.

+ 55 The user can specify different action for type names `ARRAYP`, `HARRAYP`, `TERMTABLEP`, `READTABLEP`, and `CCODEP`, even though they all have the same type `number`.

56 for Horrible PRINT. The hprint package was written by L. M. Masinter.

57 Hprint currently cannot handle compiled code arrays, stack positions, or arbitrary unboxed numbers.

first occurrence,⁵⁸ and all subsequent occurrences are printed as a back reference using an appropriate macro character. Thus the inverse function, hread merely calls the Interlisp read routine with the appropriate readable.

<u>hprint</u> [<u>x</u> ;file;uncircular]	prints <u>x</u> on <u>file</u> . ⁵⁹ If <u>uncircular</u> =T, <u>hprint</u> does no checking for any circularities in <u>x</u> (but is still useful for dumping arbitrary structures of arrays, hash arrays, lists, user data types, etc., that do not contain circularities), which results in a large speed and internal-storage advantage.
<u>hread</u> [file]	reads an <u>hprint</u> -ed expression from <u>file</u> .
<u>hcopyall</u> [<u>x</u>]	copies data structure <u>x</u> . <u>x</u> may contain circular pointers as well as arbitrary structures.

Note: HORRIBLEVARS and UGLYVARS (page 14.59) are two file package commands for dumping and reloading circular and re-entrant data structures. They provide a convenient interface to hprint and hread.

14.4 READTABLES AND TERMINAL TABLES⁶⁰

The Interlisp input and (to a certain extent) output routines are table driven by readtables and terminal tables. Read tables are objects that specify the syntactic properties of characters for input (and some output) routines. Since the input routines are concerned with parsing incoming character sequences into objects, the readtable in use at the time determines which sequences are recognized as literal atoms, strings, list structures, etc. Terminal tables are objects which supply the input/output routines with information specifically pertaining to the file T, and are described later.

A readtable is a datum⁶¹ that contains information about the syntax class of each character, e.g., break character, separator character, escape character, list or string delimiter, etc. The system packages use three readtables: T for input/output from terminals, (the value of) filerdtbl for input/output from files, and (the value of) editrdtbl, for input from terminals while in the editor. These three tables are initially equal but not eq. Using the functions described below, the user may change, reset, or copy these tables. He can also create his own readtables, and either explicitly pass them to input/output functions as arguments, or install them as the primary readtable, via setreadtable, and then not specify a rdtbl argument, i.e., use NIL.

⁵⁸ by resetting the file pointer via setfileptr.

⁵⁹ Note: hprint is intended primarily for output to disk files, since the algorithm depends on being able to reset the file pointer. If file is not a disk file (and uncircular = NIL), a temporary file, HPRINT.SCRATCH, is opened, x is hprinted on it, and then that file is copied to the final output file and the temporary file is deleted.

⁶⁰ Readtables and terminal tables were designed and implemented by D. C. Lewis.

⁶¹ In Interlisp-10, readtables are represented by 129 word arrays. However, typename distinguishes readtables from other arrays, and when applied to a readtable returns READTABLEP.

In the discussion below, most functions that accept readable arguments will also accept NIL as indicating the primary readable, or T as indicating the system's readable for terminals. Where indicated, some will also accept ORIG (*not* the value of ORIG) as indicating the original system readable.

READTABLE FUNCTIONS

readablep[rdtbl]	Value is <u>rdtbl</u> , if <u>rdtbl</u> is a real readable, otherwise NIL.
getreadtable[rdtbl]	If <u>rdtbl</u> =NIL, value is primary read table. If <u>rdtbl</u> =T, value is system's readable for terminals. If <u>rdtbl</u> is a real readable, value is <u>rdtbl</u> . Otherwise, generates an ILLEGAL READTABLE error.
setreadtable[rdtbl;flg]	resets primary readable to be <u>rdtbl</u> . ⁶² Generates ILLEGAL READTABLE error if <u>rdtbl</u> is not NIL, T, or a real readable. Value is previous setting of primary readable, i.e., <u>setreadtable</u> is suitable for use with <u>resetform</u> (Section 5).
copyreadtable[rdtbl]	value is a copy of <u>rdtbl</u> . <u>rdtbl</u> can be a real readable, NIL, T, or ORIG, in which case value is a copy of <i>original</i> system readable, otherwise generates an ILLEGAL READTABLE error. Note that <u>copyreadtable</u> is the only function that <i>creates</i> a readable.
resetreadtable[rdtbl;from]	copies (smashes) <u>from</u> into <u>rdtbl</u> . <u>from</u> and <u>rdtbl</u> can be NIL, T, or a real readable. In addition, <u>from</u> can be ORIG, meaning use system's original readable.

SYNTAX CLASSES

A syntax class is a group of characters which behave the same with respect to a particular input/output operation. There are nine basic syntax classes: LEFTPAREN, RIGHTPAREN, LEFTBRACKET, RIGHTBRACKET, STRINGDELIM, ESCAPE, BREAKCHAR, SEPRCHAR, and OTHER, each associated with a primitive syntactic property. In addition, there is an unlimited assortment of user-defined syntax classes (jointly referred to as read-macros but individually constituting unique syntax classes). For example, the characters which indicate the beginning of a list structure form a basic syntax class, LEFTPAREN. Characters which indicate the beginning of a string belong to the class STRINGDELIM. Characters that are not otherwise special belong to the class OTHER.

Note that a "syntax class" is an abstraction: there is no object referencing a collection of characters and called a syntax class. Instead, a readtable provides the *association* between a character and its syntax class, and the input/output routines enforce the abstraction by using read tables to drive the

62 If flg=T, setreadtable resets the system readable for terminals. Note that the user can reset the other system readtables with setq, e.g., (SETQ FILERDtbl (GETREADTABLE)).

parsing.⁶³

The functions below are used to obtain and (re)set the syntax class of a character. ch can either be a character code, or a character, i.e., if ch is a number, it is interpreted as a character code. For example, in Interlisp-10, 1 indicates control-A, and 49 indicates the *character* 1.

getsyntax[ch;table]

Value is syntax class of ch with respect to table. table can be NIL, T, ORIG, or a real readable or terminal table. ch is either a character code, a character, or a syntax class. In the last case, the value of getsyntax is a list of the character codes in that class, e.g., getsyntax[BREAK]=getbrk[].

setsyntax[ch;class;table]

sets syntax class of ch, a character code, or a character. table can be either NIL, T, or a real readable or terminal table. class is a syntax class, or in the case of read-macro characters (page 14.26), an expression of the form (type ... options ... fn). The value of setsyntax is the previous class of ch.

setsyntax will also accept class=NIL, T, ORIG, or a real readable or terminal table, as being equivalent to getsyntax[ch;class], i.e., means give ch the syntax class it has in the table indicated by class, e.g., setsyntax[%(;ORIG]. class can also be a character code or character, which is equivalent to getsyntax[class;table], i.e., means give ch the syntax class of the character indicated by class, e.g., setsyntax[{;%}].

syntaxp[charcode;class;table]

table is NIL, T, or a real readable or terminal table. Value is T if charcode is a member of syntax class class, e.g., syntaxp[40;LEFTPAREN]=T. class can also be a read macro type (described below), e.g. MACRO, SPLICE, INFIX, or a read-macro option, e.g. FIRST, IMMEDIATE, etc.

syntaxp compiles open when class is a quoted expression. Note that syntaxp will *not* accept a character as an argument, only a character *code*.

FORMAT CHARACTERS

A format character is a character which is recognized as special by read. There are six format characters in Interlisp namely [,], (,), ", and %. The six corresponding syntax classes are: LEFTBRACKET, RIGHTBRACKET, LEFTPAREN, RIGHTPAREN, STRINGDELIM, and ESCAPE. (Note that the class ESCAPE refers to the *input* escape character.) Making a character be a format character does *not* disable the character currently filling that function, i.e., it is perfectly acceptable to have both { and [function as left brackets. To disable a format character, assign it syntax class OTHER, e.g., setsyntax[%";OTHER].

⁶³ Special thanks to J. Strother Moore for this lucid explanation taken from [MOO].

BREAKS, SEPARATORS, AND READTABLES

The syntax classes BREAKCHAR and SEPRCHAR correspond to those characters that are treated as break or separator characters by ratom but have no other special function. For convenience, the syntax class BREAK is provided to refer to *all* break characters, i.e. it is the union of LEFTPAREN, RIGHTPAREN, LEFTBRACKET, RIGHTBRACKET, STRINGDELIM, and BREAKCHAR.⁶⁴ Thus, getsyntax[BREAK;rdtbl] and getbrk[rdtbl] are equivalent.

Note that getsyntax will never return BREAK or SEPR as a value although setsyntax and syntaxp will accept them as arguments. Instead, getsyntax will return one of the six disjoint basic syntax classes that comprise BREAK. In most cases, BREAK can be used interchangeably with BREAKCHAR. However, note that setsyntax[%(;BREAK] is a nop (since %() is already a break character), but that setsyntax[%(;BREAKCHAR] means make %() be *just a break character*, and therefore disables the LEFTPAREN function of %. It is equivalent to setsyntax[%(;OTHER] followed by setsyntax[%(;BREAK]. If the user does disable one of the format characters, e.g., by performing setsyntax[%(;OTHER], it is not sufficient for restoring the formatting function simply to make the character again into a break character, i.e., setsyntax[%(;BREAK] would *not* restore %() as LEFTPAREN.

READ MACRO CHARACTERS

The user can define various characters as read-macro characters by specifying as a class an expression of the form (type fn), where type is MACRO, SPLICE, or INFIX, and fn is the name of a function, or a lambda expression. Whenever read encounters a read-macro character, it calls the associated function, giving it as arguments the input file and readtable being used for that call to read. The interpretation of the value returned depends on the type of read-macro:

- | | |
|------------|--|
| (1) MACRO | The result is inserted into the input as if that expression had been read, instead of the read-macro character. For example, ' could be defined by: |
| | <code>[MACRO (LAMBDA(FL RDTBL) (KWOTE (READ FL RDTBL))].⁶⁵</code> |
| (2) SPLICE | The result (which should be a list or NIL) is <u>nconc</u> 'ed into the input list, e.g., if ! is defined by (SPLICE (LAMBDA NIL (APPEND FOO))), and the value of <u>foo</u> is (A B C), when the user inputs (X ! Y), the result will be (X A B C Y). |
| (3) INFIX | The associated function is called with the list of what has been read (current level list only), in <u>tconc</u> format, as its third argument. ⁶⁶ |

+ ⁶⁴ For purely symmetrical reasons, the syntax class SEPR corresponds to all separator characters. However, since the only separator characters are those that also appear in SEPRCHAR, SEPR and SEPRCHAR are equivalent.

⁶⁵ ' is standardly defined as a read-macro character in T and editrdtbl. The actual definition checks to see if the next character is a separator, right paren, or right bracket, and if so returns ' itself, e.g., (A ' B) is read as (A ' B).

⁶⁶ If an INFIX read-macro character is encountered *not* in a list, the third argument to its associated function will be NIL. If the function returns NIL, the read-macro character is essentially ignored and reading continues. Otherwise, if the function returns a tconc list of one element, that element is the value of the read. If it returns a tconc list of more than one element, the list is the value of the read.

The function's value is taken as a new tconc list which replaces the old one. For example, + could be defined by:

```
(INFIX (LAMBDA (FL RDTBL Z)
                 (RPLACA (CDR Z)
                           (LIST (QUOTE IPLUS)
                                 (CADR Z)
                                 (READ FL RDTBL)))
                 Z))
```

The specification for a read-macro character can be augmented to specify various options by giving setsyntax a list of the form (type ... options ... fn), e.g., (MACRO FIRST IMMEDIATE fn), with the following interpretations:

ALWAYS the read-macro character is a break character, i.e. a member of BREAKCHAR, and is always effective (except when preceded by the escape character).

FIRST the read-macro character is *not* a break character,⁶⁷ and is interpreted as a read-macro character *only* when it is the first character seen after a break or separator character, e.g., ' is a FIRST read-macro character, so that A'B is read as A'B.⁶⁸

ALONE the read-macro character is *not* a break character, and is interpreted as a read-macro character only when the character would have been read as a separate atom if it were not a read-macro character, i.e., when its immediate neighbors are both break or separator characters, e.g., * is an ALONE read-macro character in order to implement the comment pointer feature (see page 14.45).

Note: ALWAYS is the default option for ALWAYS, FIRST, and ALONE.

ESCQUOTE, ESC when printed with prin2, the read-macro character will be preceded by the escape character.

NOESCQUOTE, NOESC the read-macro character will be printed without an escape, e.g., ' is a NOESCQUOTE character.

ESCQUOTE is the default option for ESCQUOTE and NOESCQUOTE.

⁶⁷ Making a FIRST or ALONE read-macro character be a break character disables the read-macro interpretation, i.e., converts it to syntax class BREAKCHAR. Making an ALWAYS read-macro character be a break character is a no-op.

⁶⁸ <control-Y> is also standardly defined as a FIRST read macro which returns the result of *evaluating* the next expression read as though it had been typed. + +

IMMEDIATE, IMMED

the read-macro character is immediately activated, i.e., the character is essentially seen by the line buffer, as an EOL, and the rest of the line is passed to the input function.⁶⁹ ⁷⁰ **IMMEDIATE** read-macro characters enable the user to specify a character that will take effect immediately.⁷¹ For example, control-W is defined as an **IMMEDIATE** read-macro character.

NONIMMEDIATE, NONIMMED character is a normal character with respect to the line buffering, and so will not be activated until a carriage-return or matching right parenthesis or bracket is seen.

NONIMMEDIATE is the default option for **IMMEDIATE** and **NONIMMEDIATE**.

Note that read-macro characters can be "nested". For example, if = is defined by (MACRO (LAMBDA (FL RDTBL) (EVAL (READ FL RDTBL)))) . and ! by (SPLICE (LAMBDA (FL RDTBL) (READ FL RDTBL))), then if the value of foo is (A B C), and (X =FOO Y) is input, (X (A B C) Y) will be returned. If (X !=FOO Y) is input, (X A B C Y) will be returned.

Note that if a read-macro's function calls read, and the read returns NIL, the function cannot distinguish the case where a RIGHTPAREN or RIGHTBRACKET followed the read-macro character, e.g., (A B '), from the case where the atom NIL (or "()") actually appeared. Thus the first case is disallowed, i.e., reading a single RIGHTPAREN or RIGHTBRACKET via a read inside of a read-macro function. If this occurs, the paren/bracket will be put back into the input buffer, and a READ-MACRO CONTEXT ERROR will be generated.⁷²

readmacros[flg]

If flg=NIL, turns off action of readmacros. If flg=T, turns them on. Value is previous setting.

inreadmacrop[]

value is NIL if currently *not* under a read-macro function, otherwise the number of unmatched left parentheses or brackets.

setreadmacroflg[flg]

resets the "read-macro" flag, i.e., the internal system flag that informs read that it is under a read macro function, and causes it to

⁶⁹ Note that as a result, characters typed before an **IMMEDIATE** read-macro character cannot be erased by control-A or control-Q once the **IMMEDIATE** character has been typed, since they have already passed through the linebuffer.

⁷⁰ Making a read-macro character be both ALONE and IMMEDIATE is a contradiction, since ALONE requires that the next character be input in order to see if it is a break or separator character. Thus, ALONE read-macros are always **NONIMMEDIATE**, regardless of whether or not **IMMEDIATE** is specified.

⁷¹ i.e., as soon as it is read, not as soon as it is typed. Characters that cause action as soon as they are typed are interrupt characters (see Section 16).

⁷² If a call to read from within a readmacro encounters an unmatched RIGHTBRACKET *within* a list, the bracket is also put back into the buffer to be read (again) at the higher level. Thus, inputting an expression such as (A B '(C D] will work correctly.

generate a READ-MACRO CONTEXT ERROR, if an unmatched) or] is encountered. Value is previous value of the flag.

setreadmacroflg is useful when debugging read-macro functions to avoid spurious READ-MACRO CONTEXT error messages when typing into breaks, e.g., the user can simply put (SETREADMACROFLG) on breakresetforms (see Section 15).

TERMINAL TABLES

A readable contains input/output information that is *media-independent*. For example, the action of parentheses is the same regardless of the device from which the input is being performed. A terminal table is a datum⁷³ that contains those syntax classes of characters that pertain to *terminal* input/output operations only, e.g., DELETECHAR (control-A), DELETELINE (control-Q), etc. In addition, terminal tables contain such information as how line-buffering is to be performed, how control characters are to be echoed/printed, whether lower case input is to be converted to upper case, etc.

Using the functions below, the user may change, reset, or copy terminal tables. He can also create his own terminal tables and install them as the primary terminal table via settermtable. However, unlike readtables, terminal tables cannot be passed as arguments to input/output functions.

TERMINAL TABLE FUNCTIONS

termtablep[ttbl] value is ttbl, if ttbl is a real terminal table, NIL otherwise.

gettermtable[ttbl] If ttbl=NIL, value is primary (i.e., current) terminal table. If ttbl is a real terminal table, value is ttbl. Otherwise, generates an ILLEGAL TERMINAL TABLE error.

settermtable[ttbl] resets primary terminal table to be ttbl. Value is previous ttbl. Generates an ILLEGAL TERMINAL TABLE error if ttbl is not a real terminal table.

copytermtable[ttbl] value is a copy of ttbl. ttbl can be a real terminal table, NIL, or ORIG, in which case value is a copy of the *original* system terminal table. Note that copytermtable is the only function that *creates* a terminal table.

resettermtable[ttbl;from] smashes from into ttbl. from and ttbl can be NIL or a real terminal table. In addition, from can be ORIG, meaning use system's original terminal table.

⁷³ In Interlisp-10, terminal tables are represented (currently) by 16 word arrays.

getsyntax, setsyntax, and syntaxp all work on terminal tables as well as readtables. When given NIL as a table argument, getsyntax and syntaxp use the primary readtable or primary terminal table depending on which table contains the indicated class argument, e.g., setsyntax[ch;BREAK] will refer to the primary readtable, setsyntax[ch;CHARDELETE] will refer to the primary terminal table. In the absence of such information, all three functions default to the primary readtable, e.g., setsyntax[ch1;ch2] refers to the primary read table. If given incompatible class and table arguments, all three functions generate errors, e.g., setsyntax[ch;BREAK;tbl], where tbl is a terminal table, generates an ILLEGAL READTABLE error, getsyntax[CHARDELETE;rdtbl] an ILLEGAL TERMINAL TABLE error.

TERMINAL SYNTAX CLASSES

There are currently six terminal syntax classes: CHARDELETE (or DELETECHAR), LINEDELETE (or DELETILINE), RETYPE, CTRLV (or CNTRLV), and EOL.⁷⁴ These classes correspond (initially) to the characters control-A, control-Q,⁷⁵ control-R, control-V, and carriage-return/line-feed. All other characters belong to terminal syntax class NONE. The classes CHARDELETE, LINEDELETE, RETYPE, CTRLV, and EOL can contain at most one character. When a new character is assigned one of these syntax classes by setsyntax, the previous character is disabled, i.e., reassigned the syntax class NONE, and the value of setsyntax will be the code for the previous character of that class, if any, otherwise NIL.

TERMINAL CONTROL FUNCTIONS

echocontrol[char;mode;tbl] Used to indicate how control characters are to be echoed or printed. char is a character or character code. If mode=IGNORE, char is never printed. If mode=REAL, char itself will be printed. If mode=SIMULATE, output will be simulated. If mode=UPARROW, char will be printed as ↑ followed by the corresponding alphabetic character. The value of echocontrol is the previous output mode for char. If mode=NIL, the value is the current output mode without changing it.

Note that echoing information can be independently specified for control characters only. (However, the function echomode described below can be used to disable *all* echoing.) Therefore, if char is an alphabetic character (or code), it refers to the corresponding control character, e.g., echocontrol[Z;UPARROW] makes control-Z echo as ↑Z. All other values of char generate ILLEGAL ARG errors.

echomode[flg;tbl] If flg=T, turns echoing for terminal table tbl on. If flg=NIL, turns echoing off. Value is previous setting.

⁷⁴ On input from a terminal, the EOL character signals to the line buffering routine to pass the input back to the calling function. It also is used to terminate inputs to readline, page 14.15. In general, whenever the phrase carriage-return linefeed is used, what is meant is the character with terminal syntax class EOL.

+ ⁷⁵ for Interlisp-10 on TOPS-20, the chardelete character is , and the line delete character is control-U. All remarks below referring to control-A or control-Q should be read as referring to and control-U in the TOPS-20 context.

getechomode[ttbl]

value is current echo mode for ttbl. +

deletecontrol[type; message; ttbl] used for specifying the output protocol when a CHARDELETE or LINEDELETE is typed according to the following interpretations of type:

LINEDELETE	<u>message</u> is the message printed when LINEDELETE character is typed. Initially "# # ?".
1STCHDEL	<u>message</u> is the message printed the first time CHARDELETE is typed. Initially "\\".
NTHCHDEL	<u>message</u> is the message printed on subsequent CHARDELETE's (without intervening characters). Initially "".
POSTCHDEL	<u>message</u> is the message printed when input is resumed following a sequence of one or more CHARDELETE's. Initially "\\". ⁷⁶
EMPTYCHDEL	<u>message</u> is the message printed when a CHARDELETE is typed and there are no characters in the buffer. Initially "# # ?".
ECHO	the characters deleted by CHARDELETE are echoed.
NOECHO	the characters deleted by CHARDELETE are not echoed.

For LINEDELETE, 1STCHDEL, NTHCHDEL, POSTCHDEL, and EMPTYCHDEL, the message to be printed must be less than 5 characters. The value of deletecontrol will be the previous message as a string. If message=NIL, the value will be the previous message without changing it. For ECHO and NOECHO, the value of deletecontrol is the previous echo mode, i.e., ECHO or NOECHO. message is ignored.

Note: If the user's terminal is a scope terminal, deletecontrol and echocontrol can be used to make it really delete the last character by performing the following: echocontrol[8;REAL], (8 is code for control-H, which is backspace) deletecontrol[NOECHO], (eliminates echoing of deleted characters) deletecontrol[1STCHDEL;"↑H ↑H"], and deletecontrol[NTHCHDEL;"↑H ↑H"].

getdeletecontrol[type; ttbl]value is current deletecontrol mode for type in ttbl. +

⁷⁶ This setting of 1STCHDEL, NTHCHDEL, and POSTCHDEL makes it easy to determine exactly what has been deleted, namely all of the characters *between* the \'.s.

- raise[flg;tbl] If flg=T, input is echoed as typed, but lowercase letters are converted to upper case. If flg=NIL, all characters are passed as typed. Value is previous setting.⁷⁷
- + getraisc[tbl] value is current raise mode for tbl.

LINE-BUFFERING AND CONTROL

In Interlisp's normal state, characters typed on the terminal (this section does not apply in any way to input from a file) are transferred to a line buffer. Characters are transmitted from the line buffer to whatever input function initiated the request (i.e., read, ratom, rstring, or readc)⁷⁸ when a carriage-return⁷⁹ is typed.⁸⁰ Until this time, the user can delete characters one at a time from the input buffer by typing control-A. The characters are echoed preceded by a \. Or, the user can delete the entire line buffer back to the last carriage-return by typing control-Q, in which case Interlisp echoes # #.⁸¹ (If no characters are in the buffer and either control-A or control-Q is typed, Interlisp echoes # #.)⁸²

Note that this line editing is *not* performed by read or ratom, but by Interlisp, i.e., it does not matter (nor is it necessarily known) which function will ultimately process the characters, only that they are still in the Interlisp input buffer. Note also that it is the function that is *currently* requesting input that determines whether parentheses counting is observed, e.g., if the user executes (PROGN (RATOM) (READ)) and types in A (B C D) he will have to type in the carriage-return following the right parenthesis before any action is taken, whereas if he types (PROGN (READ) (READ)) he would not. However, once a carriage-return has been typed, the entire line is "available" even if not all of it is processed by the function initiating the request for input, i.e., if any characters are "left over", they will be returned immediately on the next request for input. For example, (PROGN (RATOM) (READC)) followed by A B carriage-return will perform both operations.

⁷⁷ In Interlisp-10, both raise[] and raise[T] execute TENEX JSYS calls corresponding to the TENEX command NORAISE. Conversion of lowercase characters to uppercase *before* echoing is also available via raise[0], which executes the JSYS calls corresponding to the TENEX command RAISE. The conversion is then performed at the TENEX level, i.e., before Interlisp-10 even sees the characters. The initial setting of raise in Interlisp-10 is determined by the terminal mode at the time the user first starts up the system. Following a sysin, the raise mode is restored to whatever it was prior to the corresponding sysout.

⁷⁸ peekc is an exception; it returns the character immediately when its second argument is NIL.

⁷⁹ i.e., the character with terminal syntax class EOL.

⁸⁰ As mentioned earlier, for calls from read, the characters are also transmitted whenever the parentheses count reaches 0. In this case, if the third argument to read is NIL, Interlisp also outputs a carriage-return line-feed. The characters are also transmitted whenever an IMMEDIATE read-macro character is typed.

⁸¹ Typing < del > (control-Z for Interlisp-10 on TOPS-20) clears the entire input buffer at the time it is *typed*, whereas the action of control-A and control-Q occur at the time they are read. < del > can thus be used to clear type-ahead.

⁸² As described earlier, the CHARDELETE, LINEDELETE, and EOL characters can all be redefined. Therefore, references to control-A, control-Q, or carriage-return in the discussion actually refer to the current CHARDELETE, LINEDELETE, or EOL characters, whatever they may be.

TURNING-OFF LINE-BUFFERING

The function control is available to defeat this line-buffering. When operating with a terminal table in which control[T] has been performed, characters are returned to the calling function without line-buffering as described below. The function that initiates the request for input determines how the line is treated:

1. read

if the expression being typed is a list, the effect is the same as though control were NIL, i.e., line-buffering until carriage-return or matching parentheses. If the expression being typed is not a list, it is returned as soon as a break or separator character is encountered,⁸³ e.g., (READ) followed by ABC space will immediately return ABC. Control-A and control-Q editing are available on those characters still in the buffer. Thus, if a program is performing several reads under control[T], and the user types NOW IS THE TIME followed by control-Q, he will delete only TIME since the rest of the line has already been transmitted to read and processed.

2. ratom

characters are returned as soon as a break or separator character is encountered. Before then, control-A and control-Q may be used as with read, e.g., (RATOM) followed by ABCcontrol-Aspace will return AB. (RATOM) followed by (control-A will return (and type # # indicating that control-A was attempted with nothing in the buffer, since the (is a break character and would therefore already have been read.

3. readc/peekc

the character is returned immediately; no line editing is possible. In particular, (READC) followed by control-A will read the control-A, (READC) followed by % will read the %.

control[flg;tbl]

flg=T eliminates Interlisp's normal line-buffering for the terminal table tbl.

flg=NIL restores line-buffering (normal).

The value of control is its previous setting.

getcontrol[tbl]

value is current control mode for tbl. +

⁸³ An exception to the above occurs when the break or separator character is a (, ", or [, since returning at this point would leave the line buffer in a "funny" state. Thus if control is T and (READ) is followed by 'ABC(', the ABC will not be read until a carriage-return or matching parentheses is encountered. In this case the user could control-Q the entire line, since all of the characters are still in the buffer.

14.5 MISCELLANEOUS INPUT/OUTPUT CONTROL FUNCTIONS

clearbuf[file;flg] Clears the input buffer for file. If file is T and flg is T, contents of Interlisp's line buffer and the system buffer are saved (and can be obtained via linbuf and sysbuf described below).

When either control-D, control-E, control-H, control-P, or control-S is typed, Interlisp automatically does a clearbuf[T;T]. (For control-P and control-S, Interlisp restores the buffer after the interaction. See Appendix 1.)

linbuf[flg] if flg=T, value is Interlisp's line buffer (as a string) that was saved at last clearbuf[T;T]. If flg=NIL, clears this internal buffer.

sysbuf[flg] same as linbuf for system buffer.

If both the system buffer and Interlisp's line buffer are empty, the internal buffers associated with linbuf and sysbuf are not changed by a clearbuf[T;T].

bklinbuf[x] x is a string. bklinbuf sets Interlisp's line buffer to x. If greater than 160 characters, first 160 taken. Value is x.

bksysbuf[x] x is a string. bksysbuf sets system buffer to x. The effect is the same as though the user typed x. Value is x.

bklinbuf, bksysbuf, linbuf, and sysbuf provide a way of "undoing" a clearbuf. Thus if the user wants to "peek" at various characters in the buffer, he could perform clearbuf[T;T], examine the + buffers via linbuf and sysbuf, and then put them back. The function resetbufs provides a + convenient way of simply clearing the input buffer, performing an interaction with the user, and + then restoring the input buffer.

+ **resetbufs[form₁;form₂;...;form_n]** nlambda, nospread function. Clears any typeahead, evaluates form₁, form₂, ..., form_n, then restores the typeahead. Value is value of form_n. Useful for handling unexpected (from the user's standpoint) interactions, e.g. disambiguating an input when the user may already have begun to typeahead the next input. Compiles open.

radix[n] Resets output radix⁸⁴ to |n| with sign indicator the sign of n. For example, in Interlisp-10, -9 will print as shown with the following radices:

84 Currently, there is no input radix.

<u>radix</u>	<u>printing</u>
10	-9
-10	68719476727
	i.e., $(2^{136}-9)$
8	-11Q
-8	777777777777Q

Value of radix is its last setting. radix[] gives current setting without changing it. Initial setting is 10.

fltfmt[formatbits]

In Interlisp-10, sets floating format control to formatbits (See TENEX JSYS manual for interpretation of formatbits). fltfmt[T] specifies free format (see Section 3).⁸⁵ Value of fltfmt is last setting. fltfmt[] returns current setting without changing it. Initial setting is T.

Note: the printnum package, page 14.20, permit greater controls on the printed appearance of numbers, allowing such things as left-justification, suppression of trailing decimals, etc.

linelength[n]

Sets the length of the print line for all files. Value is the former setting of the line length. Whenever printing an atom would go beyond the length of the line, a carriage-return is automatically inserted first. linelength[] returns current setting. Initial setting is 72.

setlinelength[n]

if n is NIL, interrogates operating system for the line length of the terminal device, and sets variable ttylinelength to this value. If n is not NIL, instructs operating system to set terminal line length to n, and then sets ttylinelength to n. Then, in either case, setlinelength performs (and returns as its value) linelength[ttylinelength].

Both aftersysoutforms and resetforms (Section 22) contain a (SETLINELENGTH) so that when the user first runs a sysout, or types control-D, the system obtains the latest information about the terminal. The various system functions that print to the terminal, e.g. pp, the editor, etc., all perform linelength[ttylinelength] (rather than linelength[72]). Thus if the user has a wider terminal the "right" thing automatically happens

⁸⁵ formatbits can also be a FLOAT format type, as described on page 14.20.

- + `settermchars[nextchar;bkchar;lastchar;unquotechar;-;-]`
+ used to set up the immediate read macros used by the editor (see
+ Section 9), as well as the <control-Y> read macro described
+ earlier. nextchar corresponds to the <line-feed> edit command,
+ bkchar to <control-X>, lastchar to <control-Z>, and unquotechar
+ to <control-Y>. For each non-NIL argument, settermchars makes
+ the corresponding control character have the indicated function.
- + Normally, settermchars will complain if the character is already an interrupt character. However, if
+ settermchars is given a list as one of its arguments, it *will* use car even if the character is an
+ interrupt. In this case, if cadr of the list is non-NIL, settermchars will reassign the interrupt
+ function to cadr. For example, if <control-X> is an interrupt, settermchars[(X W)] will make
+ <control-W> have the effect <control-X> had, and make <control-X> be the nextchar
+ operator.
- + As part of the greeting operation, settermchars is applied to the value of editcharacters, which is
+ initially (J X Z Y).⁸⁶ The user can reset editcharacters in his profile, or else explicitly call
+ settermchars to specify some other characters to be used for the indicated function.

`position[file;n]`

Gives the column number the next character will be read from or printed to, e.g., after a carriage-return, position=0. If n is non-NIL, *resets position* to n.

Note that `position[file]` is *not* the same as `getfileptr[file]` which gives the position in the *file*, not on the *line*.

14.6 SYSIN AND SYSOUT

`sysout[file]`

Saves the user's private memory on file. Also saves the stacks, so that if a program performs a sysout, the subsequent sysin will continue from that point, e.g.,
`[PROGN (SYSOUT (QUOTE FOO)) (PRINT (QUOTE HELLO))]`
 will cause HELLO to be printed after (SYSIN (QUOTE FOO)). The value of sysout is file (full name). A value of NIL indicates the sysout was unsuccessful, i.e., either disk or computer error, or user's directory was full.

Sysout does *not* save the state of any open files.⁸⁷

Whenever the Interlisp system is reassembled and/or reloaded, old sysout files are *not* compatible with the new system.

+ 86 <control-J> is line-feed. The initial value of editcharacters for Interlisp on TOPS-20 is (J A L Y).

+ 87 However, the whenclose package, page 14.9, can be used to associate with open files certain operations to be performed when a sysout is started up, including reopening the file and repositioning the file pointer.

sysout is advised to evaluate the expressions on beforesysoutforms when executing the sysout, and to evaluate the expressions on aftersysoutforms when coming back from a sysin, i.e., when the value being returned by sysout is a list. beforesysoutforms includes expressions to:

- (1) set the variable sysoutdate to (DATE), i.e. the time and date that the sysout was performed; +
- (2) if file is non-NIL, to set the variable sysoutfile to (the body of) file. If file is NIL, the value of sysoutfile is used instead. Thus, after an initial sysout, the user can simply perform sysout[] to save the current state on the next higher version of a file with the same name as the previous sysout; +
- (3) if an extension and a version number are not specified, to use the value of sysout.ext as the extension (initially .SAV for Tenex sites and .EXE for TOPS-20 sites.) +
- (4) to perform any necessary operations associated with open files specified by calls to whenclose (page 14.9). +

Aftersysoutforms includes expressions to:

- (1) reset the terminal linelength appropriately (see discussion of setlinelength on previous page); +
- (2) reset the terminal control characters if the systemtype has changed, i.e. from TENEX to TOPS-20 or vice versa. +
- (3a) if the value of sysoutgag is NIL (its initial value), and the sysout was made by the same user that is performing the sysin, to greet the user by printing the value of heraldstring followed by a greeting message. If the sysout was made by a different user, to warn the user that the user profiles may be different (see Section 22 on Greeting and User Profiles), e.g.

*****ATTENTION USER LEWIS:
THIS SYSOUT IS INITIALIZED FOR USER HARTLEY.
TO REINITIALIZE, TYPE GREET().

- (3b) if the value of sysoutgag is a list, to evaluate the list in lieu of printing a message. This permits the user to print his own message.
- (3c) for all other non-NIL values of sysoutgag, no message is printed.
- (4) call setinitials to reset the initials used for time-stamping (Section 9). +
- (5) perform any necessary operations associated with previously opened files specified by calls to whenclose (page 14.9). +

sysin[file] restores the state of Interlisp from a sysout file.⁸⁸ Value is +

⁸⁸ In Interlisp-10, file is a *runnable* file, i.e., it is not necessary to start up an Interlisp and call sysin in order to restore the state of the user's program. Instead, the user can treat the sysout file the same as a SAV file, i.e., use the operating system RUN command, or simply type the file name to the operating system, and the effect will be exactly the same as having performed a sysin.

- + cons[file;makesys], where makesys is the name of the parent system.⁸⁹ If sysin returns NIL, there was a problem in reading the file. If file is not found, generates a FILE NOT FOUND error.

Since sysin continues immediately where sysout left off, the only way for a program to determine whether it is just coming back from a sysin or from a sysout is to test the value of sysout.

For example, (COND ((LISTP (SYSOUT (QUOTE FOO))) (PRINT (QUOTE HELLO)))) will cause HELLO to be printed following the sysin, but not when the sysout was performed.

- + sysoutp[file] predicate for determining whether file is a sysout file. Returns the name of the parent makesys (Section 3) if the file is a sysout file, NIL otherwise.⁹⁰

+ 14.7 SYMBOLIC FILE INPUT

- + load[file;ldflg;printflg] Reads successive S-expressions from file (with filerdtbl as readable) and evaluates each as it is read, until it reads either NIL, or the single atom STOP. Value is file (full name).

- + If printflg=T, load prints the value of each S-expression; otherwise it does not. ldflg affects the operation of define, defined, rpaq, and rpaqq. While load is operating, dfnflg (Section 8) is rebound to ldflg. Thus, if ldflg=NIL, and a function is redefined, a message is printed and the old definition saved. If ldflg=T, the old definition is simply overwritten. If ldflg=PROP, the function definitions are stored on the .property lists under the property EXPR. If ldflg=ALLPROP, not only function definitions but also variables set by rpaqq and rpaq are stored on property lists.⁹¹

- + Another option is available for users who are loading systems for others to use, and wish to conserve space by suppressing during the load operation various user features designed to aid in development and debugging. If ldflg=SYSLOAD, load will (1) rebind dfnflg to T, (2) rebind lispchist to NIL thereby making the load not be undoable and saving the cost of saving undo information (section 22), (3) rebind addspclflg to NIL to suppress adding things to spelling lists, (4) rebind filepkgflg to NIL to prevent the FILE property from being saved and the file being "noticed" by the file package, (5) rebind buildmapflg to NIL to prevent a file map from being constructed, (6) when the load has completed, set to NOBIND

⁸⁹ sysout only saves that portion of the user's environment which is private (see Section 3.4 on Shared Interlisp).

⁹⁰ In Interlisp-10, file can also be a JFN.

⁹¹ except when the variable has value NOBIND, in which case it is set to the indicated value regardless of dfnflg.

the fileCOMS and filevars⁹² and (7) add the file name to sysfiles rather than filest.

Note: all functions that have ldflg as an argument, i.e. load, loadfns, loadvars, etc., perform spelling correction using loadoptions as a spelling list when ldflg is not a member of loadoptions. loadoptions is initially (NIL T PROP ALLPROP SYSLOAD).

load? [file;ldflg;printflg] like load except does not load file if it has already been loaded,⁹³ in which case its value is NIL.

loadfns[fns;file;ldflg;vars]⁹⁴ permits selective loading of function definitions. fns is a list of function names, a single function name, or T, meaning all functions.⁹⁵ file can be either a compiled or symbolic file, i.e., any file that can be loaded by load. The interpretation of ldflg is the same as for load.

vars specifies which non-DEFINEQ expression are to be loaded (i.e., evaluated): T means all, NIL means none, VARS is same as (RPAQQ RPAQ), FNS/VARS is same as (fileCOMS fileBLOCKS), and any other atom is the same as list[atom].

When vars is a list, each atom on vars is compared with both car and cadr of non-DEFINEQ expressions, e.g., either RPAQQ or FOOCOMS can be used to indicate (RPAQQ FOOCOMS --) should be loaded. For more complicated specification, each list on vars is treated as an edit pattern and matched with the entire non-DEFINEQ expression. In other words, a non-DEFINEQ expression will be loaded if either its car or cadr is eq to some member of vars, or it matches (using edit4e, Section 9) some list on vars, e.g., (FOOCOMS DECLARE: (DEFLIST & (QUOTE MACRO))) would cause (RPAQQ FOOCOMS --), all DECLARE:'s, and all DEFLIST's which set up MACRO's to be read and evaluated.

The value of loadfns is a list of (the names of) the functions that were found, plus a list of those functions not found (if any) headed by the atom NOT-FOUND: e.g., (FOO FIE (NOT-FOUND: FUM)). If vars is non-NIL, the value will also include those expressions that

⁹² i.e. any variable appearing in a file package command of the form (filecom * variable) (see page 14.60), e.g. in (FNS * FOOFNS). FOOFNS is set to NOBIND. Note that if the user wants the value of such a variable to be retained, even when the file is loaded with ldflg=SYSLOAD, then he should replace the variable with an equivalent, *non-atomic* expression, e.g. (FNS * (PROGN FOOFNS)).

⁹³ The test is whether the root name of file has a FILEDATES property.

⁹⁴ loadfns was originally written by J. W. Goodwin, and subsequently modified by W. Teitelman.

⁹⁵ If a compiled definition is loaded, so are all compiler generated subfunctions.

were loaded, plus a list of those members of vars for which no corresponding expressions were found (if any), again headed by the atom NOT-FOUND:.

If file=NIL, loadfns will use whereis (page 14.67) to determine where the first function in fns resides, and load from that file. Note that the file must previously have been "noticed". (For more discussion, see page 14.63).⁹⁶

loadvars[vars; file; ldflg] same as loadfns[NIL, file, ldflg, vars].

loadfrom[file; fns; ldflg] same as loadfns[fns; file; ldflg; T].

As mentioned in Section 9, once the file package knows about the contents of a file, the user can edit functions contained in the file without explicitly loading them. Similarly, those functions which have not been modified do not have to be loaded in order to write out an updated version of the file. Files are normally noticed, i.e., their contents become known to the file package (page 14.54), when either the symbolic or compiled versions of the file are loaded. If the file is *not* going to be loaded, the preferred way to notice it is with loadfrom. For example, if the user wants to update the file FOO by editing the function FOO1 contained in it, he need only perform loadfrom[FOO], editf[FOO1], and makefile[FOO]. Note that the user can also load some functions at the same time by giving loadfrom a second argument, e.g., loadfrom[FOO; FOO1], but its raison d'etre is to inform the file package about the existence and contents of a particular file.

loadblock[fn; file; ldflg] calls loadfns on those functions contained in the block declaration containing fn.⁹⁷

loadefs[fns; file] like loadfns except returns a list of functions and their symbolic definitions, plus a list of those functions not found, if any, headed by the atom NOT-FOUND:, e.g.,
loadfns[(FOO FIE FUM); FOO] = ((FOO (LAMBDA ...))
(FIE (LAMBDA ...)) (NOT-FOUND: FUM)).

+ loadcomp[file; ldflg] performs all operations on file associated with compilation, i.e. evaluates all expressions under a DECLARE: EVAL@COMPILE, and "notices" the function and variable names by adding them to nofixfnslst and nofixvarslst (see Section 23).

+ ⁹⁶ If whereis[fn] returns NIL, and the whereis package (Section 24) has been loaded, loadfns will perform whereis[fn; FNS; T], which will use the whereis data base to find the file containing fn.

+ ⁹⁷ loadblock is designed primarily for use with symbolic files, i.e., to load the exprs for a given block. It will not load a function which already has an in-core expr definition, and it will not load the block name, unless it is also one of the block functions.

Thus, if building a system composed of many files, and compilation information is scattered throughout these files, to compile one file, all that is required is to loadcomp the others.

loadcomp? [file;ldflg] like loadcomp, except does not load if file has already been loaded, in which case its value is NIL.

FILE MAPS

A file map is a data structure which contains a symbolic 'map' of the contents of a file. Currently, this consists of the begin and end address⁹⁸ for each defineq expression in the file, the begin and end address for each function definition within the defineq, and the begin and end address for each compiled function.⁹⁹

makefile, prettydef, loadfns, recompile, and numerous other system functions depend heavily on the file map for efficient operation. For example, the file map enables loadfns to load selected function definitions simply by setting the file pointer to the corresponding address using setfileptr, and then performing a single read. Similarly, the file map is heavily used by the "remake" option of makefile (page 14.77): those function definitions that have been changed since the previous version are prettyprinted; the rest are simply copied from the old file to the new one, resulting in a considerable speedup.

Whenever a file is read by load or loadfns, a file map is automatically built¹⁰⁰ and stored on the property list of the root name¹⁰¹ of the file, under the property FILEMAP. Whenever a file is written by makefile, a file map for the new file is also built and stored on the FILEMAP property.¹⁰² In addition, the file map is written on the file itself.¹⁰³ Thus, in most cases, load and loadfns do not have to build the file map at all, since a file map will usually appear in the corresponding file.¹⁰⁴

The procedure followed whenever a system package that uses file maps accesses a file is embodied in the function getfilemap. getfilemap first checks the FILEMAP property to see if a file map for

98 byte address, see getfileptr, page 14.8.

99 The internal representation of the file map is not documented since it may change when the map is extended to include information about other than just function definitions.

100 unless buildmapflg = NIL. buildmapflg is initially T.

101 the file name with directory and version number stripped off.

102 Building the map in this case essentially comes for free, since it requires only reading the current file pointer before and after each definition is written or copied. However, building the map does require that prettyprint know that it is printing a DEFINEQ expression. For this reason, the user should never print a DEFINEQ expression onto a file himself, but should instead always use the FNS command, page 14.56.

103 For cosmetic reasons, the file map is written as the last expression in the file. However, the *address* of the file map in the file is (over)written into the FILECREATED expression that appears at the beginning of the file so that the file map can be rapidly accessed without having to scan the entire file.

104 unless the file was written with buildmapflg = NIL, or was written outside of Interlisp.

this file was previously obtained or built.¹⁰⁵ If there is none, getfilemap next checks the first expression on the file to see if it is a FILECREATED expression that also contains the address of a FILEMAP.¹⁰⁶ If neither are successful getfilemap returns NIL,¹⁰⁷ and a file map will be built.¹⁰⁸

READFILE AND WRITEFILE

For those applications where the user simply wants to simply read all of the s-expressions on a file, and not evaluate them, the function readfile is available:

readfile[file] Reads successive S-expressions from file using read (with filerdtbl as readable) until the single atom STOP is read, or an end of file encountered. Value is a list of these S-expressions.

writefile[x;file] Inverse of readfile. Writes a date expression onto file, followed by successive S-expressions from x, using filerdtbl as a readable. If x is atomic, its value is used. If file is not open, it is opened. If file is a list, car[file] is used and the file is left opened. Otherwise, when x is finished, a STOP is printed on file and it is closed. Value is file.

endfile[file] Prints STOP on file and closes it.

¹⁰⁵ The full name of the file is also stored on the FILEMAP property along with its map.

¹⁰⁶ currently, file maps for *compiled* files are not written onto the files themselves. However, load and loadfns will *build* maps for a compiled file when it is loaded, and store it on the property FILEMAP. Similary, loadfns will obtain and use the file map for a compiled file, when available.

¹⁰⁷ getfilemap also returns NIL, if usemapflg=NIL, initially T. usemapflg is available primarily to enable the user to recover in those cases where the file and its map for some reason do not agree. For example, if the user edits a symbolic file that contains a map using a text editor such as TECO, inserting or deleting just one character will throw that map off. The functions which use file maps contain various integrity checks to enable them to detect that something is wrong, and to generate the error FILEMAP DOES NOT AGREE WITH CONTENTS OF file-name. In such cases, the user can set usemapflg to NIL, causing the map contained in the file to be ignored, and then reexecute the operation. A new map will then be built (unless buildmapflg is also NIL).

¹⁰⁸ While building the map will not help *this* operation, it *will* help in future references to this file. For example, if the user performs loadfrom[FOO] where FOO does not contain a file map, the loadfrom will be (slightly) slower than if FOO did contain a file map, but subsequent calls to loadfns for this version of FOO will be able to use the map that was built as the result of the loadfrom, since it will be stored on FOO's FILEMAP property.

14.8 PRETTYPRINT

`prettyprint[lst; -]109 110`

lst is a list of functions (if atomic, its value is used). The definitions of the functions are printed in a pretty format on the primary output file using the primary readable. For example,

```
(FACTORIAL
  [LAMBDA (N)
    (COND
      ((ZEROP N)
        1)
      (T (ITIMES N (FACTORIAL (SUB1 N)) 111
```

A more complete example is shown on page 14.47.

Note: prettyprint will operate correctly on functions that are broken, broken-in, advised, or have been compiled with their definitions saved on their property lists: it prints the original, pristine definition, but does not change the current state of the function. If prettyprint is given an atom which is not the name of a function, but has a value, it will prettyprint the value. If the function is not defined but is known to be on one of the files noticed by the file package, prettyprint will load in the definition (using loadfns) and print it.¹¹² Otherwise, prettyprint will perform spelling correction. If all fails, prettyprint returns (atom NOT PRINTABLE).

`pp[x]`

nlambda, nospread function that performs `output[T]`, `setreadable[T]`, `linelength[ttylinelength]`, and then calls prettyprint: `PP FOO` is equivalent to `PRETTYPRINT((FOO))`; `PP(FOO FIE)` or `(PP FOO FIE)` is equivalent to `PRETTYPRINT((FOO FIE))`. Primary output file and primary readable are restored after printing.

As described earlier, when prettyprint, and hence pp, is called with the name of a function that is not defined, but whose definition is on a file known to the file package, the definition is automatically read in and then prettyprinted. However, if the user does not intend on editing or running the definition, i.e. he simply wants to *see* the definition, the function pf described below can be used to simply copy the corresponding bytes from the file to the terminal. This results in a savings in both space and time, since it is not necessary to allocate storage to actually read in the definition, and since the function is already in prettyprint format on the file, it is not necessary to re-prettyprint it.

¹⁰⁹ The prettyprint package was written by W. Teitelman.

¹¹⁰ prettyprint has a second argument that is `T` when called from prettydef. In this case, whenever prettyprint starts a new function, it prints (on the terminal) the name of that function if more than 30 seconds (real time) have elapsed since the last time it printed the name of a function.

¹¹¹ In order to save space on files, tabs are used instead of spaces for the initial spaces on each line, assuming that each tab corresponds to 8 spaces. This results in a reduction of file size by about 30%. Tabs will not be used if prettytabflg is set to `NIL` (initially `T`).

¹¹² except when called from prettydef.

- + `pf[fn;fromfiles;tofile]` nlambda, nospread function. Copies the definition of `fn` found on each of the files in `fromfiles` to `tofile`. If `tofile=NIL`, defaults to T. If `fromfiles=NIL`, defaults to `whereis[fn;NIL;T]`. Note that the typical usage of `pf` is to simply type `PF fn`.
- + When printing to the terminal, `pf` performs several transformations on the characters in the file that comprise the definition for `fn`: (1) font information (page 14.49) is stripped out; (2) `changechar` (page 14.49) is also not printed; (3) since functions typically tend to be printed to a file with a larger linelength than when printing to a terminal, the left margin on each line, i.e. the number of leading spaces, is cut in half; and (4) comments are printed as described below.

COMMENT FEATURE

A facility for annotating Interlisp functions is provided in `prettyprint`. Any S-expression beginning with * is interpreted as a comment¹¹³ and printed in the right margin. Example:

```
(FACTORIAL
  [LAMBDA (N)
    (COND
      ((ZEROP N)
        1)
      (T
        (* COMPUTES N!)
        (* 0!=1)
        (* RECURSIVE DEFINITION:
          N!=N*N-1!))
      (ITIMES N (FACTORIAL (SUB1 N))))
```

- These comments actually form a part of the function definition. Accordingly, * is defined as an NLAMBDA NOSPREAD function that returns its argument, i.e., it is equivalent to `quote`. When running an interpreted function, * is entered the same as any other Interlisp function. Therefore, comments should only be placed where they will not harm the computation, i.e., where a quoted expression could be placed. For example, writing `(ITIMES N (FACTORIAL (SUB1 N)) (* RECURSIVE DEFINITION))` in the above function + would cause an error when ITIMES attempted to multiply N, N-1!, and RECURSIVE. The + compiler (Section 18) will also detect places where a comment has been used for value, and print + an appropriate error message.

For compilation purposes, * is defined as a macro which compiles into no instructions. Thus, if the user compiles a function with comments, and load the compiled definition into another system, the extra atom and list structures storage required by the comments will be eliminated. This is the way the comment feature is intended to be used. For more options, see end of this section.

Note: comments are designed mainly for documenting *listings*. Thus when prettyprinting to the terminal, comments are suppressed and printed as the string **COMMENT**.¹¹⁴

¹¹³ Actually, any expression `car` of which is equal to the value of the variable `commentflg` is treated as a comment. `commentflg` is initially *, but the user can set it to some other value, e.g. ;, and use this to indicate comments. For more details, see page 14.48.

¹¹⁴ The value of `**comment**flg` determines the action. If `**comment**flg` is NIL, the comment is printed. Otherwise, the value of `**comment**flg` is printed. `**comment**flg` is initially set to " **COMMENT** ". The function `pp*` is provided to prettyprint functions, including their comments, to the terminal. `pp*` operates exactly like `pp` except it first sets `**comment**flg` to NIL. The function `pf*` plays an analogous role for `pf`.

COMMENT POINTERS

For a well-commented collection of programs, the list structure, atom, and pname storage required to represent the comments in core can be significant. If the comments already appear on a file and are not needed for editing, a significant savings in storage can be achieved by simply leaving the text of the comment on the file when the file is loaded, and instead retaining in core only a *pointer* to the comment. This feature has been implemented by defining * as a read-macro in FILERTBL which, instead of reading in the entire text of the comment, constructs an expression containing [1] the name of the file in which the text of the comment is contained, [2] the address of the first byte of the comment, and [3] the number of bytes.¹¹⁵ For output purposes, * is defined as a prettyprintmacro (page 14.49) that prints the comments represented by such pointers by simply copying the corresponding bytes from one file to another, or to the terminal.¹¹⁶ Normal comments are processed the same as before, and can be intermixed freely with comment pointers.

The comment pointer feature is enabled by setting normalcommentsflg to NIL. normalcommentsflg is initially T. Note that normalcommentsflg can be changed as often as desired, i.e., some files can be loaded normally, and others using comment pointers.

For convenience of editing selected comments, an edit macro, get*, is included which loads in the text of the corresponding comment.¹¹⁷ get* is defined in terms of getcomment:

getcomment[x;-;-] If x is a comment pointer, value is the comment, which it reads from the file. Otherwise, value is x. +

CONVERTING COMMENTS TO LOWER CASE

This section is for users operating on terminals without lower case, e.g. model 33 teletypes, who nevertheless would like their comments to be converted to lower case for more readable line-printer listings. Users with lower-case terminals can skip to the File Package sections (as they can type comments directly in lower case).

%% If the second atom in a comment is %%, the text of the comment is converted to lower case so that it looks like English instead of LISP (see next page). +

The output on the next page illustrates the result of a lower casing operation. Before this function was prettyprinted, all comments consisted of upper case atoms, e.g., the first comment was (* %% INTERPRETS A SINGLE COMMAND). Note that comments are converted *only* when they are actually written to a file by prettyprint.

¹¹⁵ Plus a flag to indicate whether the comment appeared at the right hand margin or centered on the page. +

¹¹⁶ If the user changes the value of commentflg but still wishes to use the comment pointer feature, he should make sure to give the new commentflg the same read-macro definition in filertbl as * has, e.g. if he resets commentflg to be ;, he should perform (SETSYNTAX ';' '* FILERTBL). +

¹¹⁷ pp* prints the comment *without* reading it by simply copying the corresponding bytes to the terminal. +

The algorithm for conversion to lower case is the following: If the first character in an atom is \uparrow , do not change the atom (but remove the \uparrow). If the first character is %, convert the atom to lower case.¹¹⁸ If the atom¹¹⁹ is an Interlisp word,¹²⁰ do not change it. Otherwise, convert the atom to lower case. Conversion only affects the upper case alphabet, i.e., atoms already converted to lower case are not changed if the comment is converted again. When converting, the first character in the comment and the first character following each period are left capitalized. After conversion, the comment is physically modified to be the lower case text minus the %% flag, so that conversion is thus only performed once (unless the user edits the comment inserting additional upper case text and another %% flag).

lcaselst

Words on lcaselst will always be converted to lower case. lcaselst is initialized to contain words which are Interlisp functions but also appear frequently in comments as English words. e.g., AND, EVERY, GET, GO, LAST, LENGTH, LIST, etc. Thus, in the example on the previous page, not was written as \uparrow NOT, and GO as \uparrow GO in order that they might be left in upper case.

ucaselst

words on ucaselst (that do not appear on lcaselst) will be left in upper case. ucaselst is initialized to NIL.

abbrevlst

abbrevlst is used to distinguish between abbreviations and words that end in periods. Normally, words that end in periods and occur more than halfway to the right margin cause carriage-returns. Furthermore, during conversion to lowercase, words ending in periods, except for those on abbrevlst, cause the first character in the *next* word to be capitalized. abbrevlst is initialized to the upper and lower case forms of ETC. I.E. and E.G.

¹¹⁸ User must type %% as % is the escape character.

¹¹⁹ minus any trailing punctuation marks.

¹²⁰ i.e., is a bound or free variable for the function containing the comment, or has a top level value, or is a defined function, or has a non-NIL property list.

```

(BREAKCOM
 [LAMBDA (BRKCOM BRKFLG) (* Interprets a
single command.)]

(PROG (BRKZ)
 TOP (SELECTQ
 BRKCOM
 [↑ (RETEVAL (QUOTE BREAK1)
 (QUOTE (ERROR))]

(GO (* Evaluate BRKEXP
unless already evaluated,
print value, and exit.))

(BREAKCOM1 BRKEXP BRKCOM NIL BRKVALUE)
(BREAKEXIT))

(OK (* Evaluate BRKEXP,
unless already evaluated,
do NOT print value,
and exit.))

(BREAKCOM1 BRKEXP BRKCOM BRKVALUE BRKVALUE)
(BREAKEXIT T))

(↑WGO (* Same as GO except
never saves evaluation
on history.))

(BREAKCOM1 BRKEXP BRKCOM T BRKVALUE)
(BREAKEXIT))

(RETURN

(* User will type in expression to be evaluated and
returned as value of BREAK. Otherwise same as GO.)


(BREAKCOM1 [SETQ BRKZ (COND
(BRKCOMS (CAR BRKCOMS))
(T (LISPXREAD T]
(QUOTE RETURN)
NIL NIL (LIST (QUOTE RETURN)
BRKZ))

(BREAKEXIT))

(EVAL (* Evaluate BRKEXP but
do not exit from BREAK.))

(BREAKCOM1 BRKEXP BRKCOM)
(COND
(BRKFLG (BREAK2)
(PRIN1 BRKFN T)
(PRIN1 (QUOTE " EVALUATED
"))

(SETQ IVALUE (CAR BRKVALUE)) (* For user's benefit.))

)

```

SPECIAL PRETTYPRINT CONTROLS

#rpars controls the number of right parentheses necessary for square bracketing to occur. If #rpars=NIL, no brackets are used. #rpars is initialized to 4.

linelength[n] determines the position of the right margin for prettyprint.¹²¹

firstcol is the starting column for comments. Initial setting is 48. Comments run between firstcol and linelength. If a word in a comment ends with a "." and is not on the list abbrevlst, and the position is greater than halfway between firstcol and linelength, the next word in the comment begins on a new line. Also, if a list is encountered in a comment, and the position is greater than halfway, the list begins on a new line.

prettylcom If a comment is bigger (using count) than prettylcom in size, it is printed starting at column 10, instead of firstcol.¹²² prettylcom is initialized to 14 (arrived at empirically).

#carefulcolumns in the interests of efficiency, prettyprint approximates the number of characters in each atom, rather than calling nchars, when computing how much will fit on a line. This procedure works satisfactorily in most cases. However, users with unusually long atoms in their programs, e.g., such as produced by clispify, may occasionally encounter some glitches in the output produced by prettyprint. The value of #carefulcolumns tells prettyprint how many columns (counting from the right hand margin) in which to actually compute nchars instead of approximating. Setting #carefulcolumns to 20 or 30 will eliminate the above glitches, although it will slow down prettyprint slightly. #carefulcolumns is initially 0.

widepaper[flg] widepaper[T] sets filelinelength to 120, firstcol to 80, and prettylcom to 28. These are useful settings for prettyprinting files to be listed on wide paper. widepaper[] restores these parameters to their initial values. The value of widepaper is its previous setting.

commentflg If car of an expression is eq to commentflg, the expression is treated as a comment. commentflg is initialized to *.

¹²¹ Note that makefile, page 14.64, resets linelength to the value of filelinelength, before calling prettydef. filelinelength is initially 72.

¹²² Comments are also printed starting at column 10, if their second element is also a *, i.e., comments of the form (* * --).

prettyflg	If <u>prettyflg</u> is NIL, <u>printdef</u> uses <u>prin2</u> instead of prettyprinting. This is useful for producing a fast symbolic dump (see FAST option of <u>makefile</u> , page 14.64). Note that the file loads the same as if it were prettyprinted. <u>prettyflg</u> is initially set to T.
clispifyprettyflg	used to inform <u>prettyprint</u> to call <u>clispify</u> on selected function definitions before printing them. See Section 23.
prettyprintmacros	is an assoc-list that enables the user to format selected expressions himself. <u>car</u> of each expression being <u>prettyprinted</u> is looked up on <u>prettyprintmacros</u> , and if found, <u>cdr</u> of the corresponding entry is applied to the expression. If the result of this application is NIL, <u>prettyprint</u> will ignore the expression. This gives the user the option of printing the expression himself in whatever format he pleases. If the result is non-NIL, it is <u>prettyprinted</u> in the normal fashion. This gives the user the option of computing some other expression to be <u>prettyprinted</u> in its place. <u>prettyprintmacros</u> is initially NIL.
prettyprintpemacros	is a list of elements of the form (typename . fn). For types other than lists and atoms, the type name of each datum to be <u>prettyprinted</u> is looked up on <u>prettyprintpemacros</u> , and if found, the corresponding function is applied to the datum about to be printed, instead of simply printing it with <u>prin2</u> . <u>prettyprintpemacros</u> is initially NIL.
prettyequivlst	is an assoc-list that enables user to tell <u>prettyprint</u> to treat a <u>car</u> -of-form the same as some other <u>car</u> -of-form, e.g. if (QLAMBDA . LAMBDA) appears on <u>prettyequivlst</u> , then QLAMBDA expressions will be <u>prettyprinted</u> the same as LAMBDA's. <u>prettyequivlst</u> is initially NIL.
changechar	if non-NIL, and <u>prettyprint</u> is printing to a file or display terminal, <u>prettyprint</u> prints <u>changechar</u> in the right hand margin while printing those expressions marked by the editor as having been changed (see Section 9). <u>changechar</u> is initially .
(* E x)	A comment of this form causes x to be evaluated at prettyprint time, e.g., (* E (RADIX 8)) as a comment in a function containing octal numbers can be used to change the radix to produce more readable printout. The comment is also printed.

FONT PACKAGE

Prettyprint contains a facility for printing elements of various classes, e.g. user functions, system functions, clisp words, comments, etc., in different fonts to emphasize (or deemphasize) their

+ importance, and in general to provide for more pleasing printout when printing to a file.¹²³ Of course, in order to be useful, this facility requires that the user has access to a printer which supports multiple fonts, such as an XGP.

+ Prettyprint signals font changes by inserting a user-defined escape sequence, e.g. $\text{\uparrow F\uparrow C}$ meaning change to font 3, $\text{\uparrow F\uparrow A}$ change back to font 1, etc. It is convenient if these sequences can consist of control characters, because by making these characters be separator characters in filerdtbl, a file with font changes in it can also be loaded back in. Otherwise, the user would have to dump two files, one for listing, and one for loading.

+ Currently, the user can specify fonts for each of the following eight classes:¹²⁴

+ **lambdafont** the font for printing the name of the function being prettyprinted, before the actual definition (usually a large font).

+ **clispfont** if clispflg is on, the font for printing any clisp words, i.e. atoms with property **CLISPWORD**.

+ **commentfont** the font for everything inside of a comment.

+ **userfont** the font for the name of any function in the file, or any member of the list fontfns.

+ **systemfont** the font for any other (defined) function.

+ **changefont** the font for anything in an expression marked by the editor as having been changed.

+ **prettycomfont** the font used in printing the operand of a file package command.

+ **defaultfont** the font for everything else, or any of the above classes for which a font is not specified.

+ 123 None of this section pertains to prettyprinting to the terminal.

+ 124 each different, or the same for several classes. Note: the output primitives print, prin1, etc., currently do not know about variable width fonts, so the user may have to experiment to find a compatible (pleasing) set of fonts. Note also that the user does not set lambdafont, clispfont, et al, but indicates what font to be used by including an appropriate entry in the fontprofile, page 14.51. fontset, described below, will then set lambdafont, clispfont, et al, to a data structure that contains the necessary information for performing the font change.

For example, the function on page 14.47 is shown on page 14.53 with lambdafont corresponding to HELVETICA12 BOLD, commentfont to TIMESROMAN6 ITALIC, userfont to HELVETICA10 BOLD, and defaultfont to GACHA10.

The operation of the font package is affected by a large number of parameters, e.g. filelinelength, listfilestr, etc. plus the various fontnames themselves. To facilitate switching back and forth between various configurations, the font package allows the user to set the various parameters to their desired values, and then use the function fontname to package up and save this configuration. Subsequently, the user invokes this configuration by performing fontset[name].

fontescapechar the character or string used to signal the start of a font escape sequence.

fontprofile list of elements of the form (fontclass NIL font#),¹²⁵ where fontclass is one of the eight font classes and font is the font number for that class.¹²⁶ For each fontclass, the escape sequence consisting of fontescapechar followed by the character *code* for the font number, i.e. for font number 1, $\uparrow A$, for font number 2, $\uparrow B$, etc.

If font is NIL for any fontclass, the defaultfont is used. Note that the defaultfont must be specified or an error is generated.

fontchangeflg if T, enables fonts, if NIL, disables fonts, i.e. no font changes are performed when prettyprinting.

listfilestr passed to the operating system by listfiles. Can be used to specify subcommands to the LIST command, e.g. to establish correspondance between font number and font name.

commentlinelength since comments are usually printed in a smaller font, commentlinelength is provided to offset the fact that Interlisp does not know about font widths. Its value is a dotted pair of numbers. When fontchangeflg=T, car of commentlinelength is the linelength used to print short comments, i.e. those printed in the right margin, and cdr is the linelength used when printing full width comments.

Note that the user may also want to reset filelinelength, prettylcom and firstcol (all described earlier) as a part of various font configurations.

¹²⁵ The NIL is a place marker. fontname replaces (rplaca) cadr when the font configuration is defined.

¹²⁶ it is assumed that the user has some way of communicating to the printing device the correspondence between font numbers and fonts.

+ fontdefsvars
+
+

the list of variables to be packaged by a fontname. initially fontchangeflg, filelinelength, commentlinelength, firstcol, prettylcom, listfilestr, and fontprofile.

+ fontname[name]
+

performs some processing on fontprofile, and then collects names and values of variables on fontdefsvars, and saves them on fontdefs.

+ fontset[name]
+

restores font configuration for name. Generates an error if name not previously defined.

+ changefont[fontclass]
+
+

e.g. (CHANGEFONT LAMBDAFONT), (*not* 'LAMBDAFONT), prints the font escape sequence to change to fontclass. For use in prettyprintmacros.

+ fontdefs
+
+
+

the dictionary of font configurations. fontdefs is a list of elements of form (name . parameter-pairs). To save a configuration on a file after performing a fontname to define it, the user could either save the entire value of fontdefs, or simply use an ALISTS file package command to dump out just the one configuration.

(BREAKCOM
 [LAMBDA (BRKCOM BRKFLG) (* Interprets a single command.)
 (PROG (BRKZ)
 TOP (SELECTQ
 BRKCOM
 [↑ (RETEVAL (QUOTE BREAK1)
 (QUOTE (ERROR)])
 (GO (* Evaluate BRKEXP unless already evaluated, print value, and exit.)
 (BREAKCOM1 BRKEXP BRKCOM NIL BRKVALUE)
 (BREAKEXIT))
 (OK (* Evaluate BRKEXP, unless already evaluated, do NOT print value, and exit.)
 (BREAKCOM1 BRKEXP BRKCOM BRKVALUE BRKVALUE)
 (BREAKEXIT T))
 (↑WGO (* Same as GO except never saves evaluation on history.)
 (BREAKCOM1 BRKEXP BRKCOM T BRKVALUE)
 (BREAKEXIT))
 (RETURN
 (* User will type in expression to be evaluated and returned as value of BREAK. Otherwise same as GO.)

(BREAKCOM1 [SETQ BRKZ (COND
 (BRKCOMS (CAR BRKCOMS))
 (T (LISPXREAD T])
 (QUOTE RETURN)
 NIL NIL (LIST (QUOTE RETURN)
 BRKZ))
 (BREAKEXIT))
 (EVAL (* Evaluate BRKEXP but do not exit from BREAK.)
 (BREAKCOM1 BRKEXP BRKCOM)
 (COND
 (BRKFLG (BREAK2)
 (PRIN1 BRKFN T)
 (PRIN1 (QUOTE " EVALUATED
 "))
 T)))
 (SETQ !VALUE (CAR BRKVALUE)) (* For user's benefit.)
)

14.9 FILE PACKAGE¹²⁷ ¹²⁸

This section describes the file package: a set of functions, conventions, and interfaces with other system packages for facilitating the bookkeeping involved with working in a large system consisting of many symbolic files and their compiled counterparts. Essentially, the file package removes from the user the burden of keeping track of where things are and what things have changed. For example, the file package keeps track of which file contains a particular datum, e.g. a function definition, record declaration, etc., and, in many cases, will automatically retrieve the datum when necessary if it is not already in the user's working environment. The file package also keeps track of which files have been in some way modified and need to be dumped, which files have been dumped, but still need to be listed and/or recompiled.

The file package requires that for each file, the value of the atom `fileCOMS`, be a list of file package commands which describe how to write out the file, e.g. for the file `FOO`, the command list would be the value of the variable `FOOCOMS`. Thus, if the file `FOO` is to contain function definitions for the functions `A` and `B` and values for the variables `C` and `D`, its command list would be `((FNS A B) (VARS C D))`. `FNS` and `VARS` are the names of the file package commands for the function and variable file package types, and the remainder of the command specifies the elements whose "definitions" of the respective type are to be written on the file. With `FOOCOMS` set as above, `makefile[FOO]129` will actually place on the file `FOO` expressions which, when evaluated as `FOO` is loaded, will restore the various definitions. Such a command list can be constructed by simply setting and editing the `fileCOMS` variable. However, the file package contains facilities which make constructing and updating command lists easier, and in some cases automatic. These are described in detail below.

All the system functions that perform global file operations,¹³⁰ e.g., `load`, `loadfn`, `prettydef`, `tcompl`, `recompile`, et al, as well as those functions that define or change data, e.g., `editf`, `editv`, `editp`, DWIM corrections to user functions, typed-in¹³¹ assignment of variables or property values etc, interact with the file package. Some of these interactions are quite complex. For example, the same function may appear in several different files, or the symbolic or compiled files may reside in other directories, or were originally made under a different name, etc. Therefore, this section will not attempt to document *how* the file package works in each and every situation, but instead make the deliberately vague statement that it does the "right" thing with respect to keeping track of what has been changed, and what file operations need to be performed in accordance with those changes.

+ 127 The file package was written by W. Teitelman, and extended by L. M. Masinter and R. M. Kaplan. The notion of a typed definition, and the designing and implementing of type-independent ways of manipulating the name-definition-type-file associations, belong to L. M. Masinter

128 The file package can be disabled by setting `filepkflg` to `NIL`.

129 All file operations in the file package are based on the root name of the file, i.e., the filename with version number and/or directory field removed, but extension, if any, included. In other words, the user could also have said `makefile[<KAPLAN>FOO. ; 27]`.

130 as opposed to "local" file operations such as those performed by `print`, `read`, `setfileptr`, etc.

131 i.e. if the user types in (`PUTPROP 'FOO prop expression`), this is noticed by the file package. If a program executes the above, it is not.

TYPED DEFINITIONS

In addition to the definitions of functions and values of variables, source files in Interlisp can contain a variety of other information, e.g. property lists, record declarations, definition of edit macros, hash arrays, etc. In order to treat uniformly from the standpoint of file operations such a diverse assortment of data, we introduce the concept of a *typed definition*, of which a function definition is just one example. We say that a definition associates with a name (usually a literal atom), a datum, the definition, of a given type (called the file package type). Since the same name may have several definitions, (e.g. a given atom may have both a function definition and a variable-value definition), it is important to think of a typed definition as a relation between three elements: the name, the definition, and the type.

The file package includes mechanisms for creating, destroying, and editing typed definitions, and for moving definitions from one file to another, or from one name to another. For example, the primitive functions for creating and destroying function and variable definitions are putd and setq, and they are edited with editf and edy respectively. There is also a need to move definitions from the user's current core image to a symbolic file, or vice versa. This adds a fourth element to the notion of a definition, namely the file where the typed definition of a particular name resides.

Whenever the user defines, edits, or otherwise changes a datum of a particular file package type, the corresponding system functions inform the file package that such new items have been defined or old definitions modified. The file package maintains a "database" of this information, so that it can tell the user (via the function files?) what files need to be rewritten, listed, or recompiled, and ask him where new definitions should be stored in the case that they do not appear on the fileCOMS of any file (by calling addtofiles?). The function cleanup will execute all the operations necessary to make the user's permanent files consistent with the definitions in his current core-image.

In addition, the file package provides:

- (1) A uniform way of augmenting a file's command list to include new items of a given file package type, the function addtofile. The function addtofile will either construct a new command appropriate for the given file package type, or add the new name to an already existing command for dumping items of that type, taking into account special information about the uses and formats of the different commands.
- (2) A uniform way of deleting items from a file's command list, the function delfromfile.
- (3) A simple way of determining whether an item of a given type resides on a particular file or whether any items of that type reside on the file, the function infilecoms?.
- (4) A way of determining what files contain definitions of a given type for a particular name, the function whereis.
- (5) A way of comparing alternative definitions of the same name, or definitions of different names, the functions compare and comparedefs.
- (6) Type-independent ways of manipulating the name-definition-type-file associations, via the functions getdef, putdef, copydef, editdef, etc.

To provide these facilities, the file package must have considerable information about each of the various file package commands and file package types. For the file package types and commands described below, this information has already been provided, and the user can simply invoke the

- + appropriate defining and editing functions, and then specify the appropriate file package commands, or allow the file package to construct the commands automatically. The user may also define new file package types by specifying how the system should operate with respect to definitions of that type via the function filepkgtype. Similarly, the user may define new file package commands by specifying the relationship between that command and the various file package types via the function filepkgcom. The methods for defining new file package types and commands will be discussed later (page 14.72). First, the built-in file package types and their associated commands will be described.

FILE PACKAGE COMMANDS

The basic mechanism for creating symbolic files is the function makefile, described in detail on page 14.64. makefile takes as its first argument a file name. It extracts the namefield of the file, packs COMS onto the end of it, and treats the value of the resulting atom as the file command list, i.e. alist of file package commands. E.g. makefile[FOO. ; 27] will use the value of FOO_{COMS} as file command list.

File package commands can be used to save on the output file definitions of functions, values of variables, property lists of atoms, arrays, advised functions, edit macros, record declarations, etc. The interpretation of each file package command is as follows:

1. (FNS fn₁ ... fn_m), a defineq is written with the definitions of fn₁ ... fn_m.¹³²
2. (VARS var₁ ... var_n), for each var_i, an expression will be written which will set its top level value when the file is loaded. If var_i is atomic, var_i will be set to the top-level value it had at the time the file was written, i.e., (RPAQQ var_i top-level-value) is written.¹³³ ¹³⁴ If var_i is non-atomic, it is interpreted as (var form). e.g., (FOO (APPEND FILE FUM)) or (FOO (QUOTE (FOO1 FOO2 FOO3))). In this case, the expression (RPAQ var form) is written.
3. (ADDVARS (var₁ . lst₁) ... (var_n . lst_n)), for each (var_i . lst_i), writes an addtovar expression such that each element of lst_i that is not a member of the value of var_i at the time the file is loaded is added to var_i, i.e., the new value of var_i will be the union of its old value and lst_i. e.g. (ADDVARS (DIRECTORIES LISP LISPUSERS)) will add LISP and LISPUSERS to the value of directories. var_i can initially be NOBIND, in which case it is first set to NIL, i.e. (ADDVARS (var)) can be used to initialize var to NIL if var has not previously been set.
- + 4. (ALISTS (alistname₁ atom₁ atom₂ ...) ... (alistname_n atom₃ atom₄ ...)), alistname_i is the name

¹³² The user should never print a DEFINEQ expression directly onto a file himself, but should instead always use the FNS command for dumping functions. For more details, see footnote on page 14.41.

¹³³ rpaqq and rpaq are like setqq and setq, except that they also perform some special operations with respect to the file package.

¹³⁴ The HORRIBLEVARS file package command described below provides a way of saving and reloading variables whose values contain re-entrant or circular list structure, user data types, arrays, or hash arrays.

of an alist, i.e. a variable whose value is an association list, e.g. editmacros, baktracelist, etc. +
 For each alistname_i, writes out expressions which will restore the corresponding entries, e.g. +
 (ALISTS (BREAKMACROS BT BTV)) will dump the definition for the BT and BTV +
 commands.¹³⁵ +

5. (PROP propname atom₁ ... atom_n) an appropriate putprops will be written which will restore the value of propname for each atom_i when the file is loaded.¹³⁶ If propname is a list, expressions will be written for each property on that list. If propname=ALL, the values of all user properties (on the property list of each atom_i) are saved.¹³⁷
6. (IFPROP propname atom₁ ... atom_n), same as PROP command, except that only saves the properties that actually appear on the property list of the corresponding atom. For example, if FOO1 has property PROP1 and PROP2, FOO2 has PROP3, and FOO3 has property PROP1 and PROP3, (IFPROP (PROP1 PROP2 PROP3) FOO1 FOO2 FOO3) will save only those 5 property values.
7. (PROPS (atom₁ propname₁) ... (atom_n propname_n)), similar to PROP command. An appropriate putprops will be written which will restore the value of propname_i for each atom_i when the file is loaded.¹³⁸
8. (P . expressions), each S-expression following P will be printed on the output file, and consequently evaluated when the file is loaded.
9. (E . forms), each form following E will be evaluated at *output* time, i.e., when makefile reaches this command.
10. (COMS com₁ ... com_n), each of the commands com₁ ... com_n will be interpreted as a file package command.
11. (* . text), used for inserting a comment in a file. First a form-feed is printed, then the comment.

¹³⁵ Some alists are handled specially, e.g. usermacros, lispymacros, etc. all of which have their own file package commands.

¹³⁶ If atom_i does not have the property propname (as opposed to having the property with NIL value), a warning message "NO propname PROPERTY FOR atom_i" is printed. The command IFPROP can be used if it is not known whether or not an atom will have the corresponding property.

¹³⁷ sysprops is a list of properties used by system functions. Only properties *not* on that list are dumped when the ALL option is used.

¹³⁸ As with the PROP command, if atom_i does not have the property propname (as opposed to having the property with NIL value), a warning message "NO propname PROPERTY FOR atom_i" is printed.

12. (ADVISE $fn_1 \dots fn_m$), for each fn_n , appropriate expressions will be written which will reinstate the function to its advised state when the file is loaded. See Section 19.
13. (ADVICE $fn_1 \dots fn_m$), for each fn_i , will write a putprops which will put the advice back on the property list of the function. The user can then use readadvice to reactivate the advice.
14. (USERMACROS $atom_1 \dots atom_p$), each $atom_i$ is the name of a user edit macro. Writes expressions for adding the edit macro definitions of $atom_i$ to usermacros, and adding the names of the commands to the appropriate spelling lists.¹³⁹
- + 15. (FILEPKGCOMS $atom_1 \dots atom_p$), each $atom_i$ is either the name of a user-defined file package command (see page 14.73) or a user-defined file package type (or both). Writes expressions which will restore each command/type.¹⁴⁰
- + 16. (LISPXMACROS $atom_1 \dots atom_n$), Each $atom_i$ is a lispxmacro or lispxhistorymacro (Section 22). Writes expressions which will save and restore the definition for each macro, as well as making the necessary additions to lispxcoms
17. (RECORDS $rec_1 \dots rec_n$), where $rec_1 \dots rec_n$ are the names of records (Section 23), writes expressions which will redeclare the records when the file is loaded.¹⁴¹
18. (I.S.OPRS $opr_1 \dots opr_n$), where $opr_1 \dots opr_n$ are the names of user-defined i.s.oprs (Section 23), writes expressions which will redefine the i.s.oprs when the file is loaded.
- + 19. (TEMPLATES $atom_1 \dots atom_n$), where $atom_1 \dots atom_n$ have Masterscope templates (Section 20), writes expressions which will restore the templates when the file is loaded.
20. (BLOCKS $block_1 \dots block_n$), for each $block_i$, a declare expression will be written which the block compile functions interpret as block declarations. See Section 18.
21. (DECLARE: . filepkgcoms/flags), Normally expressions written onto a symbolic file are (1) evaluated when loaded; (2) copied to the compiled file when the symbolic file is compiled

* 139 If $atom_i$ is not a user macro, a warning message "no EDIT MACRO for $atom_i$ " is printed.

+ 140 If $atom_i$ is not a file package command or type, a warning message "no FILE PACKAGE COMMAND for $atom_i$ " is printed.

+ 141 The file package command INITRECORDS can be used to write expressions on a file that will, when loaded, perform whatever initialization/allocation is necessary for the indicated records, but not to write out, and hence cause to be read back in, the record declarations themselves. This facility is useful for building systems on top of Interlisp, in which the implementor may want to eliminate the record declarations from a production version of the system, but the allocation for these records must still be done.

(see Section 18); and (3) not evaluated at compile time. **DECLARE:** allows the user to override these defaults. The output of those file package commands appearing within the **DECLARE:** command is embedded in a **DECLARE:** expression, along with any tags that are specified, e.g., (**DECLARE:** EVAL@COMPILE DONTCOPY (FNS --) (PROP --)) would produce (**DECLARE:** EVAL@COMPILE DONTCOPY (DEFINEQ --) (PUTPROPS --)). **DECLARE:** is *defined* as an nlambda nospread function. When declare: is called, it processes its arguments by evaluating or not evaluating each list depending on the setting of an internal state variable. The tags EVAL@LOAD, or DOEVAL@LOAD, and DONT EVAL@LOAD can be used to reset this state variable. The initial setting is to evaluate. The tag EVAL@LOADWHEN can be used to provide conditional evaluation. The value of the expression immediately following the tag determines whether or not to evaluate subsequent expressions when loading, i.e. ... EVAL@LOADWHEN T ... is equivalent to ... EVAL@LOAD ...¹⁴²

- 22. (**MACROS** atom₁ ... atom_n) writes out the **MACRO** properties for each atom_i, embedded in a **DECLARE:** EVAL@COMPILE. Equivalent to (**DECLARE:** EVAL@COMPILE (PROP MACRO atom₁ ... atom_n). See Section 18.
- 23. (**SPECVARS** . vars) (**LOCALVARS** . vars) (**GLOBALVARS** . vars), outputs corresponding compiler declaration embedded in a **DECLARE:** DOEVAL@COMPILE DONTCOPY. See Section 18.
- 24. (**UGLYVARS** var₁ ... var_n), like **VARS**, except that the value of each var_i may contain structures for which read is not an inverse of print, e.g. arrays, readtables, user data types, etc. Uses hprint package (page 14.59).
- 25. (**HORRIBLEVARS** var₁ ... var_n), like **UGLYVARS** except structures may also contain circular pointers. Uses hprint package (page 14.59).¹⁴³ The values of var₁ ... var_n are printed in the same operation, so that they may contain pointers to common substructures.
- 26. (**ARRAY** var₁ ... var_n), each var_i following **ARRAY** should have an array as its value. An appropriate expression will be written which will set the variable to an array of exactly the same size, type, and contents upon loading.¹⁴⁴

¹⁴² As indicated in Section 18, **DECLARE:** expressions are specially processed by the compiler. In this case, the relevant tags are COPY, DOCOPY, COPYWHEN, DONTCOPY, EVAL@COMPILE, DOEVAL@COMPILE, EVAL@COMPILEWHEN, DONT EVAL@COMPILE, FIRST, and NOTFIRST. The value of declaretagslst is a list of all the tags used in **DECLARE:** expressions. If a tag not on this list appears in a **DECLARE:** file package command, performs spelling correction using declaretagslst as a spelling list.

¹⁴³ UGLYVARS does not do any checking for circularities, which results in a large speed and internal-storage advantage over HORRIBLEVARS. Thus, if it is known that the data structures do *not* contain circular pointers, UGLYVARS should be used instead of HORRIBLEVARS.

¹⁴⁴ assuming that the elements of the array are objects for which read is an inverse of print. Otherwise, UGLYVARS or HORRIBLEVARS should be used.

- + 27. (ORIGINAL com₁ ... com_n), each of the commands will be interpreted as a file package command without regard to any file package macros (see below). Useful for redefining a built-in file package command in terms of itself.

In each of the commands described above, if the atom * follows the command type,¹⁴⁵ the form following the *, i.e., caddr of the command, is evaluated and its value used in executing the command, e.g., (FNS * (APPEND FNS1 FNS2)). When this form is atomic, i.e. a variable, e.g. (FNS * FOOFNS), we say that the variable is a *filevar*. Note that (COMS * form) provides a way of computing what should be done by makefile.

- * New file package commands can be defined or built in commands redefined via the function filepkgcom (page 14.73). New file package types can be defined or built in types redefined via the function filepkgtypes (page 14.72). Both file package commands and file package types can be dumped via the FILEPKGCOMS command. If a file package function is given a command or type that is not defined, it attempts spelling correction¹⁴⁶ using filepkgsplst as a spelling list. If successful, the corrected version of the list of file package commands is written (again) on the output file.¹⁴⁷ If unsuccessful, generates an error, BAD FILE PACKAGE COMMAND.

Example:

```
←SET(FOOFNS (FOO1 FOO2 FOO3))
←SET(FOOCOMS((FNS * FOOFNS) (VARS FIE)
(PROP MACRO FOO1 FOO2) (P (MOVD (QUOTE FOO1) (QUOTE FIE1]
←MAKEFILE(FOO)
```

would create a file FOO containing:

1. (FILECREATED "time and date the file was made" . "other information")
2. (PRETTYCOMPRINT FOOCOMS)
3. (RPAQQ FOOCOMS ((FNS * FOOFNS) ...))
4. (RPAQQ FOOFNS (FOO1 FOO3 FOO3))
5. (DEFINEQ "definitions of FOO1, FOO2, and FOO3")
6. (RPAQQ FIE "value of FIE")
7. (PUTPROPS FOO1 MACRO propvalue)
8. (PUTPROPS FOO2 MACRO propvalue)

¹⁴⁵ Except for the PROP and IFPROP commands, in which case the * follows the property name, e.g., (PROP MACRO * FOOMACROS).

¹⁴⁶ unless dwmflg or nospellflg=NIL. See Section 17.

¹⁴⁷ since at this point, the uncorrected list of file package commands would already have been printed on the output file. When the file is loaded, this will result in fileCOMS being reset, and may cause a message to be printed, e.g., (FOOCOMS RESET). The value of FOOCOMS would then be the corrected version.

9. (MOVD (QUOTE FOO1) (QUOTE FILE1))

10. STOP

This completes the description of file package commands.

FILE PACKAGE TYPES

A file package command is an instruction to makefile to perform an explicit, well-defined operation, usually printing some expression(s). A file package *type* is an abstract notion of a class of objects which share the property that every object of the same file package type is stored, retrieved, edited, copied etc., by the file package in the same way. Usually there is a one to one correspondence between file package types and file package commands, i.e. for each file package type, there is a file package command which is used for writing objects of that type to a file, and each file package command is used to write objects of a particular type. However, in some cases, the same file package type can be dumped by several different file package commands. For example, the file package commands PROP, IFPROP, and PROPS all dump out the file package type PROPS.¹⁴⁸ It is also permissible for the same file package command to dump several different file package types. For example, the user can define a file package command which dumps both a function definition and its macro. Conversely, some file package commands do not dump any file package types at all, e.g. the E command.

Thus, for each file package command, the file package must be able to determine what typed definitions the command will cause to be printed so that the file package can determine on what file (if any) an object of a given type is contained. Similarly, for each file package type, the file package must be able to construct a command that will print out an object of that type. In other words, the file package must be able to map file package commands into file package types, and vice versa. Information can be provided to the file package about a particular file package command via the function filepkgcom, page 14.73, and information about a particular file package type via the function filepkgtypes, page 14.72.¹⁴⁹

The file package currently implements the following file package types:¹⁵⁰ FNS (functions), VARS

¹⁴⁸ This means if the user changes an object of file package type PROPS, e.g. via editp or a typed in call to putprop or via an explicit call to markaschanged (page 14.67), this object can be written out with any of the above three commands. Thus, when the file package attempts to determine whether this typed object is contained on a particular file, it must look at instances of all three commands PROP, IFPROP, and PROPS, to see if the corresponding atom and property are specified.

¹⁴⁹ In the absence of other information, the default is simply that a file package command of the form (FOO name) prints out the definition of name as a type FOO, and, conversely, if name is an object of type FOO, then name can be written out by a command of the form (FOO name).

¹⁵⁰ The value of filepkgtypes is a list of all file package types, including any that may have been defined by the user.

- + (variables), ALISTS (alist entries),¹⁵¹ FILEVARS (filevars, page 14.60), PROPS (property name-value pairs),¹⁵² EXPRESSIONS (expressions),¹⁵³ MACROS (compiler macros, Section 18), USERMACROS (user edit macros, Section 9), LISPXMACROS (lispxmacros and lispxhistorymacros, Section 22), ADVICE (advice, Section 19), FILEPKGCOMS (file package commands/types), RECORDS (records, Section 23), FIELDS (fields of records, Section 23), I.S.OPRS (iterative statement operators, Section 23), TEMPLATES (Masterscope templates, Section 20).

MARKING CHANGES

- * Operations in the file package can be broken down roughly into three categories: (1) marking changes, (2) noticing files, and (3) updating files. The various system functions which create or modify objects of the corresponding type call markaschanged (page 14.67) to mark the corresponding object as changed.¹⁵⁴ For example, when a function is defined via define or defined, or modified either explicitly via editf, or implicitly, via a DWIM correction, the function is marked as being a changed object of type FNS. Similarly, whenever a new record is declared, or an existing record redeclared or edited, it is marked as being a changed object of type RECORDS, and so on for all of the other file package types.
- + Note that in some cases the marking procedure can be subtle, e.g. if the user edits a property list using editp, only those properties whose values are actually changed (or added) are marked. As mentioned earlier, some properties implement other file package types, e.g. EXPR, ADVICE, MACRO, I.S.OPR, etc. For example, if the user changes the value of the property I.S.OPR, he is really changing an object of type I.S.OPR, and the effect is the same as though he had redefined the i.s.opr via a direct call to the function i.s.opr. If a property whose value has been changed or added does not correspond to a specific file package type, then it is marked as a changed object of type PROPS whose *name* is (variablename propname).¹⁵⁵

¹⁵¹ A variable is declared to have a value which is an association list, i.e. a list of dotted pairs accessed via assoc and putassoc, by putting on its property list the property VARTYPE with value ALIST. In this case, each dotted pair on the list is an object of type ALISTS. When the value of such a variable is changed, only those entries in the a-list that are actually changed or added are marked as changed (objects of type ALISTS). Objects of type ALISTS are dumped via the ALISTS or ADDVARS file package commands. Note that some a-lists "implement" other file package types, e.g. the value of usermacros implements the file package type USERMACROS, the value of lispxmacros and lispxhistorymacros implements the file package type LISPXMACROS. This is indicated by having the value of the property VARTYPE be a list of the form (ALIST filepkgtypes), e.g. getprop[LISPXHISTORYMACROS;VARTYPE] = (ALIST LISPXMACROS).

¹⁵² Note that some properties implement another file package type, e.g. the property MACRO implements the file package type MACROS, the property ADVICE implements ADVICE, etc. This is indicated by the appearance of the property PROPTYPE on the property list of the property name, i.e. getprop[MACRO;PROPTYPE]=MACROS. When such a property is changed or added, an object of the corresponding file package type is marked. If getprop[propertyname;PROPTYPE]=IGNORE, the change is ignored (i.e. IGNORE cannot be the name of a file package type). FILE, FILEMAP, FILEDATES, etc. are all handled this way. Otherwise, when a property is changed or added, an object of type PROPS, with "name" (atom propertyname) is marked as being changed.

¹⁵³ Objects of type expressions are written out via the P file package command, and marked as being changed via the REMEMBER programmer assistant command, Section 22.

¹⁵⁴ The user can also call markaschanged directly to mark objects of a particular file package type as changed.

¹⁵⁵ except if the property name has a property PROPTYPE with value IGNORE as described earlier.

Similarly, if the user changes a variable which implements the file package type ALISTS (as indicated by the appearance of the property VARTYPE with value ALIST on the variable's property list), only those entries that are actually changed are marked as being changed objects of type ALISTS, and the "name" of the object will be (variablename key) where key is car of the entry on the alist that is being marked. If the variable corresponds to a specific file package type other than ALISTS, e.g. USERMACROS, LISPXMACROS, etc., then an object of that type is marked. In this case, the name of the object will be car of the corresponding entry on the a-list, e.g. if the user edits lispxmacros and changes a definition for the lispmacro PL, then the object PL of type LISPXMACROS is marked as being changed.

The section on Defining New File Package Commands/Types, page 14.73, tells how the user can change or extend the marking algorithm for particular file package commands or types.

NOTICING FILES

Files are "noticed" by load and loadfns (or loadfrom, loadvars, etc.) or by makefile. Noticing a file consists of adding its root name to the list filelst, and adding the property FILE, value ((fileCOMS . type)), to the property list of its root name,¹⁵⁶ ¹⁵⁷ where type indicates *how* the file was loaded, e.g., completely loaded, only partially loaded as with loadfns, loaded as a compiled file, etc. For example, if the user performs load[<TEITELMAN>FOO.LSP; 2], FOO.LSP is added to filelst, and ((FOOCOMS . T)) is put on the property list of FOO.LSP.

The property FILE is used to determine whether or not the corresponding file has been modified since the last time it was loaded or dumped as described below. In addition, the property FILECHANGES contains the union of the names of all changed items, undifferentiated by type, since the file was loaded (i.e., there may have been several sequences of editing and rewriting the file), and the property FILEDATES a list of version numbers and the corresponding file dates. The use and maintenance of these properties is explained below.

UPDATING FILES

Periodically, the function updatefiles is called to find which file(s) contain the elements that have been changed.¹⁵⁸ updatefiles operates by scanning filelst and interrogating the file package

- 156 The computation of the root name is actually based on the name of the file as indicated in the FILECREATED expression appearing at the front of the file, since this name corresponds to the name the file was originally made under. Similarly, the file package can detect that the file being noticed is a compiled file (regardless of its name), by the appearance of more than one FILECREATED expressions. In this case, each of the files mentioned in the FILECREATED expressions are noticed. For example, if the user performs BCOMPL((FOO FIE)), and subsequently loads FOO.COM, both FOO and FIE will be noticed.
- 157 The variable loadedfilelst contains a list of the actual names of the files as loaded by load or loadfns. For example, if the user performs LOAD[<NEWLISP>EDITA.COM; 3], EDITA will be added to filelst, but <NEWLISP>EDITA.COM; 3 is added to loadedfilelst. loadedfilelst is not used by the file package, it is maintained for the user's benefit.
- 158 updatefiles is called by files?, cleanup, and makefiles, i.e., any procedure that requires the FILE property to be up to date. (The user can also invoke updatefiles directly.) This procedure is followed rather than update the FILE property after each change because scanning filelst and examining each file package command can be a time-consuming process, and is not so noticeable when performed in conjunction with a large operation like loading or writing a file.

commands for each file. When (if) any files are found that contain the corresponding typed definition, the name of the element is added to the value of the property FILE for the corresponding file. Thus, after updatefiles has completed operating, the files that need to be dumped are simply those files on filelst for which cdr of their FILE property is non-NIL. For example, if the user loads the file FOO containing definitions for FOO1, FOO2, and FOO3, edits FOO2, and then calls updatefiles, getprop[FOO;FILE] will be ((FOOCOMS . T) FOO2). If any objects marked as changed have not been transferred to the FILE property for some file, e.g., the user defines a new function but forgets (or declines) to add it to the file package commands for the corresponding file, then both files? and cleanup will print warning messages, and then call addtofiles to permit the user to specify on which files these items belong.

Whenever a file is written using makefile, the elements that have been changed, i.e., cdr of the FILE property, are moved to the property FILECHANGES, and cdr of the FILE property is reset (rplacd) to NIL.¹⁵⁹ In addition, the file is added to the list notlistedfiles and notcompiledfiles. Whenever the user lists a file using listfiles, it is removed from notlistedfiles. Similarly, whenever a file is compiled by tcompl, recompile, bcompl, or bocompile, the file is removed from notcompiledfiles. Thus at each point, the state of all files can be determined. This information is available to the user via the function files?. Similarly, the user can see whether and how each particular file has been modified (by examining the appropriate property values), dump all files that have been modified, list all files that have been dumped but not listed, recompile all files that have been dumped but not recompiled, or any combination of any or all of the above by using one of the function described below.

FILE PACKAGE FUNCTIONS

makefile[file;options;reprintfns;sourcefile]

notices file if not previously noticed. Performs linelength[filelinelength], and calls prettydef giving it NIL, file, fileCOMS, reprintfns, sourcefile, and the list of changes as its arguments,¹⁶⁰ restores original linelength, and then adds file to notlistedfiles¹⁶¹ and notcompiledfiles.¹⁶² options is a list of options or a single option interpreted as follows:

FAST perform prettydef with prettyflg=NIL

¹⁵⁹ If the file was not on filelst, e.g., the user defined some functions and initialized the corresponding fileCOMS without loading a file, then the file will be "noticed" when it is written by makefile, i.e. it will be added to filelst, and given appropriate FILE, FILEDATES and FILECHANGES properties.

¹⁶⁰ fileCOMS are constructed from the name field only, e.g., makefile[FOO.TEM] will work. The list of changes is simply cdr of the FILE property, as described earlier, i.e., those items that have been changed since the last makefile. makefile merges those changes (using union) with those handled in previous calls to makefile, and stores the result on the property FILECHANGES. This list of changes is included in the FILECREATED expression printed at the beginning of the file by printdate, along with the date and version number of the file that was originally noticed, and the date and version number of the current file, i.e., this one. (these two version numbers and dates are also kept on the property FILEDATE for various integrity checks in connection with remaking a file as described below.)

+ ¹⁶¹ except if the file has on its property list the property FILETYPE with value DON'TLIST, or a list containing DON'TLIST.

* ¹⁶² Files that do not contain any function definitions, or those that have on their property list the property FILETYPE with value DON'TCOMPILE or a list containing DON'TCOMPILE, are not added to notcompiledfiles, nor are they compiled even when options specifies C or RC.

RC	call <u>recompile</u> after <u>prettydef</u> , or <u>bcompile</u> , if there are any block declarations specified in fileCOMS.
C	calls <u>tcompl</u> after <u>prettydef</u> , or <u>bcompl</u> , if there are any block declarations specified in fileCOMS.
CLISPIFY	perform <u>prettydef</u> with <u>clispifyprettyflg</u> =T, causing <u>clispify</u> (see Section 23) to be called on each function defined as an expr before it is prettyprinted. ¹⁶³
NOCLISP	performs <u>prettydef</u> with <u>prettytranflg</u> =T, causing the CLISP translations to be printed, if any, in place of the corresponding CLISP expressions, e.g., iterative statements, record expressions, <u>printout</u> forms, etc..
LIST	calls <u>listfiles</u> on <u>file</u> .
REMAKE	'remakes' <u>file</u> , i.e. copies the prettyprinted definitions of those functions that have not changed from an earlier version of the symbolic file, and only prettyprints those functions that have changed. See discussion, page 14.77.
NEW	does <i>not</i> remake <u>file</u> . ¹⁶⁴

Any other option is spelling corrected using the list makefileoptions.
If spelling correction fails, makefile generates an error.

+

If a remake is *not* being performed,¹⁶⁵ makefile checks the state of file to make sure that the entire symbolic file was actually loaded. If file was loaded as a compiled file, makefile prints the message "CAN'T DUMP: ONLY THE COMPILED FILE HAS BEEN LOADED." Similarly, if only some of the symbolics were load via loadfns or loadfrom, makefile prints "CAN'T DUMP: ONLY SOME OF ITS SYMBOLICS HAVE BEEN LOADED." In both cases, makefile will then ask the user if it should dump anyway, and if the user declines, makefile does not call prettydef, but simply returns (file NOT DUMPED) as its value.

If F, ST, STF, or S is the *next* item on options following C or RC, given to the compiler as the answer to the compiler's question LISTING?, e.g., makefile[FOO;(C F LIST)] will dump FOO, then tcompl or bcompl it specifying that functions are not to be redefined, and finally list the file.

The user can indicate that file must be block compiled together with other files as a unit by

¹⁶³ Alternatively, if file has the property FILETYPE with value CLISP or a list containing CLISP, prettydef is called with clispifyprettyflg reset to CHANGES, which will cause clispify to be called on all functions marked as having been changed. For more details, see discussion of clispifyprettyflg in Section 23. Note that if file has property FILETYPE with value CLISP, the compiler will know to dwimify its functions before compiling them, as described in Sections 18 and 23.

*

¹⁶⁴ If makefileremakeflg is T (its initial setting), the default for all calls to makefile is to remake. The NEW option is provided in order to override this default.

¹⁶⁵ i.e., makefileremakeflg is NIL, or the option NEW was specified.

putting a list of those files on the property list of each file under the property FILEGROUP. For example, EDIT and WEDIT are one such group, DWIM, WTFIX, CLISP, and DWIMIFY another. If file has a FILEGROUP property, the compiler will not be called until all files on this property have been dumped that need to be.

- + makefile operates by rebinding prettyflg, prettytranflg, clispifyprettyflg, and then under a resetlst,
- + evaluating each expression on makefileforms (under errorset protection), and then calling prettydef.
- + The FAST, CLISPIFY, NOCLISP are all implemented via expressions on makefileforms, e.g.
- + (AND (MEMB 'FAST OPTIONS) (SETQ PRETTYFLG NIL)). The user can add expressions to
- + makefileforms to implement his own options.

makefiles[options;files]

For each file on files that has been changed, performs makefile[file;options]. If files=NIL, filelst is used, e.g., makefiles[LIST] will make and list all files that have been changed. In this case, if any typed definitions for any items have been defined or changed and they are *not* contained in one of the files on filelst, makefiles calls addtofiles? to allow the user to specify where these go. The value of makefiles is a list of all files that are made.

listfiles[files]

nlambda, nospread function. Uses the function tenex (Section 21) to tell the operating system to list each file in files (if NIL, notlistedfiles is used).¹⁶⁶

Each file listed is removed from notlistedfiles if the listing is completed, e.g., if the user control-C's to stop the listing and QUITs. For each file not found, listfiles prints the message "file-name NOT FOUND" and proceeds to the next file on files.

compilefiles[files]

nlambda, nospread function. Executes the RC option of makefile for each member of files. (If files=NIL, notcompiledfiles is used.)¹⁶⁷

cleanup[files]

nlambda, nospread. Dumps, lists, and recompiles (or brecompiles) any and all files on files requiring the corresponding operation. If files = NIL, filelst is used. Value is NIL.¹⁶⁸

¹⁶⁶ listfiles calls the function listfiles1 on each file to be listed. listfiles1 calls tenex with concat[LIST\$;filename;listfilestr], where listfilestr is initially "2". The user can reset listfilestr to specify subcommands for the list command, or advise or redefine listfiles1 for more specialized applications.

¹⁶⁷ If car of files is a list, it is interpreted as the options argument to makefiles. This feature can be used to supply an answer to the compiler's LISTING? question, e.g., compilefiles[(STF)] will compile each file on notcompiledfiles so that the functions are redefined without the exprs being saved.

¹⁶⁸ The user can affect the operation of cleanup by resetting the variable cleanuptions, initially (LIST RC). For example, if cleanuptions is (RC F), no listing will be performed, and no functions will be redefined as the result of compiling. Alternatively, if car of files is a list, it will be interpreted as the list of options regardless of the value of cleanuptions.

files?[]	Prints on terminal the names of those files that have been modified but not dumped, dumped but not listed, dumped but not compiled, plus the names of those functions and other prettytypes (if any) that are not contained in any file. If there are any, <u>files?</u> then calls <u>addtofiles?</u> to allow the user to specify where these go.	*
whereis[name;type;files]	<u>type</u> is a file package type. <u>whereis</u> sweeps through all the files on <u>files</u> and returns a list of all files containing <u>name</u> as a <u>type</u> . <u>whereis</u> knows about and expands all file package commands and <u>filepkgmacros</u> . <u>type</u> =NIL defaults to FNS. If <u>files</u> is not a list, the value of <u>filelst</u> is used. ¹⁶⁹	+
markaschanged[name;type;newflg]	marks <u>name</u> of type <u>type</u> as being changed. <u>newflg</u> is T for calls to <u>markaschanged</u> corresponding to the <i>creation</i> of <u>name</u> , e.g. from <u>define</u> , as opposed to calls corresponding to a change to <u>name</u> , e.g. from the editor. Value of <u>markaschanged</u> is <u>name</u> . <u>markaschanged</u> is undoable.	+
unmarkaschanged[name;type]	unmarks (undoably) <u>name</u> of type <u>type</u> as being changed. ¹⁷⁰ Value is <u>name</u> if <u>name</u> was marked as changed and is now unmarked, NIL otherwise.	+
REMEMBER &	programmer assistant command (Section 22). Marks the events indicated by & as changed objects of type EXPRESSIONS.	+
filepkgchanges[type;lst]	lambda nospread. If <u>lst</u> is not specified (as opposed to being NIL), value is a list of those objects of type <u>type</u> that have been marked as changed but not yet associated with their corresponding files. If <u>lst</u> is specified, <u>filepkgchanges</u> sets the corresponding list. <u>filepkgchanges[]</u> returns a list of <i>all</i> objects marked as changed as a list of elements of the form (typename . changedobjects).	+
addtofiles?[-]	called from <u>makefiles</u> , <u>cleanup</u> , and <u>files?</u> when, after <u>updatefiles</u> has finished operating, some objects remain that have been marked as changed for which no file containing them could be found. <u>addtofiles?</u> asks the user if he wants to say where the various changed items that do not belong to any file should be placed. If	+

169 If the whereis package (Section 24) has been loaded, whereis is redefined so that files=T means use the whereis data base, i.e. whereis will find where x is even if the file has not been loaded or noticed. files=NIL always means use filelst.

170 Provided that updatefiles has not been called since name was marked, thereby moving name to the property list of the file(s) that contain it. See discussion page 14.63.

+ user answers N(o), returns NIL without taking any action. If the
+ user answers], this is taken to be an answer to each question that
+ would be asked, and interpreted as described in (4) below, i.e. all
+ changed items are marked as dummy items by adding them to
+ NILCOMS. Otherwise, addtofiles? maps through all the changedlist's,
+ prints each element, and accepts one of the following responses:

- (1) a file name or name of a list, e.g., FOO or FOOFNS.¹⁷¹ Adds the item to the corresponding file or list, using addtofile.
- (2) line-feed - means same as the user's previous response
- (3) space or carriage return - take no action
- (4)] - item is marked as a dummy item by adding it to NILCOMS, i.e., tell file package not to worry about what to do with this item.
- (5) [- the "definition" of the items in question are prettyprinted to the terminal, and then the user is asked again about their disposition.
- (6) (- addtofiles? prompts with "LISTNAME: (", the user types in the name of a list, i.e. a variable whose value is a list, terminated by a).¹⁷² item will then only be added to (under) a command in which the named list appears as a filevar. If none are found, a message is printed, and the user is asked again. For example, the user defines a new function FOO3, and when asked where it goes, types (FOOFNS). If the command (FNS * FOOFNS) is found, FOO3 will be added to the value of FOOFNS. If instead the user types (FOOCOMS), and the command (COMS * FOOCOMS) is found, then FOO3 will be added to a command for dumping functions that is contained in foocoms.
- (6) @ - addtofiles? prompts with "NEAR: (", the user types in the name of an object, and item is then inserted in a command for dumping objects (of its type) that contains the indicated name. item is inserted immediately after the indicated name.

+ 171 If the item is not the name of a file on filelst, the user will be asked whether it is a new file. If he says no, then
+ addtofiles? will check whether the item is the name of a list, i.e. whether its value is a list. If not, the user will be
+ asked whether it is a new list.

172 If the named list is not also the name of a file, the user can simply type it in without parenthesis as described under (1) above.

FUNCTIONS FOR MANIPULATING TYPED DEFINITIONS

Note: all functions described below adhere to the following conventions:

(1) if a list argument is called for, e.g. files meaning a list of files, and an atom is given, the function will operate as though list of that atom was given;

(2) type is a file package type. The singular form of the name of a type is also recognized, e.g. type=VAR is equivalent to type=VARS. type=NIL is equivalent to type=FNS;

(3) files=NIL is equivalent to files=filelst;

(4) source can be one of:

NIL means the definition currently in effect if there is one, the saved definition, or the definition from a file determined by whereis.¹⁷³

0 means the definition currently in effect.

T means the "saved" definition, as stored by savedef.

FILE means the definition contained on the (first) file determined by whereis.

a file name means the definition contained on the indicated file.

All functions which make destructive changes are undoable.

Note: the operation of most of the functions described below can be changed or extended by modifying the appropriate properties for the corresponding file package type using the function filepkgtpe, described on page 14.72.

getdef[name;type;source;options]

Returns the s-expression definition of name, of type type, from source.¹⁷⁴ type is a file package type, e.g. FNS, VARS, RECORDS. For example, for type=FNS, a lambda expression is returned,¹⁷⁵ for type=VARS, the value of name is returned, etc. getdef also recognizes type=FIELDS, in which case the list of record declarations which contain name is returned, and type=FILES, in

¹⁷³ called with files=T, so that if the whereis package (Section 24) is loaded, this will use the whereis data base to find the file containing the definition.

¹⁷⁴ copying the definition unless options is or contains NOCOPY.

¹⁷⁵ The definition will be dwimified if it is deemed likely to contain CLISP unless options is or contains NODWIM.

- + which case the command list for the file is returned.¹⁷⁶ For all other types, getdef returns the s-expression which would be prettyprinted when dumping name as type.
- + source is as described above. When source is a file name, getdef actually obtains the definition from the file.
- + getdef causes an error if an appropriate definition cannot be found, unless options is or contains NOERROR.
- + putdef[name;type;definition] defines name of type type with definition. For type=FNS, does a define; for type=VARS, does a saveset; for type=FILES, establishes the command list, and notices name, etc.
- + copydef[old;new;type;source] defines new to have a copy of the definition of old, i.e. essentially performs putdef[new;subst[new;old;getdef[old;type;source]];type]. For type=FILES, copydef not only establishes the commands and notices new, as describe above, but also calls makefile to actually dump the file new. E.g. COPYDEF(PDQ RXT FILE) sets up RXTCOMS to be a copy of PDQCOMS, changes things like (VARS * PDQVARS) to be (VARS * RXTVARS) in RXTCOMS, and performs a makefile on RXT such that the appropriate definitions get copied from PDQ.
- + deldef[name;type] Removes the definition of name as a type that is currently in effect.
- + showdef[name;type;file] prettyprints the definition of name as a type to file, i.e. shows the user how name would be written to a file. used by addtofiles? (page 14.67).
- + editdef[name;type;source;coms] edits the definition of name with type type, i.e. essentially performs putdef[name;type;editc[getdef[;name;type;source];coms]].
- + savedef[name;type;definition] Makes definition (or if definition=NIL, the definition of name as a type that is currently in effect) be the "saved" definition for name as a type. If type=FNS (or type=NIL), this consists of storing definition on name's property list under property EXPR, CODE, or SUBR. For type=VARS, definition is stored as the value of the VALUE property. For other types, definition is stored in an internal data structure, from where it can be retrieved by getdef or unsavedef.

+ 176 i.e. getatomval[filecoms[name]].

unsavedef[name;type]	Makes the "saved" definition of <u>name</u> as a <u>type</u> be the definition currently in effect. As described in Section 8, when <u>type</u> =NIL, <u>unsavedef</u> will unsave the EXPR property if any, else CODE or SUBR. <u>unsavedef</u> also recognizes <u>type</u> =EXPR, CODE, or SUBR, meaning to unsave the corresponding definition only.	+ + + +
loaddef[name;type;source]	equivalent to putdef[name;type;getdef[name;type;source]]. <u>loaddef</u> is essentially a generalization of <u>loadfns</u> , e.g. it enables loading a single record declaration from a file. Note that loaddef[fn] will give <u>fn</u> an expr definition, either obtained from its property list or a file, unless it already has one.	+ + + + +
changecallers[old;new;types;files;method]	Finds all of the places where <u>old</u> is used as any of the types in <u>types</u> and changes those places to use <u>new</u> . For example, changecallers[NLSETQ; ERSETQ] will change all calls to <u>nlsetq</u> to be calls to <u>ersetq</u> . Also changes occurrences of <u>old</u> to <u>new</u> inside the coms of any file, inside record declarations, properties, etc.	+ + + + + +
changecallers	attempts to determine if <u>old</u> might be used as more than one type; e.g. if it is both a function and a record field. If so, rather than performing the transformation of <u>old</u> -> <u>new</u> automatically, the user is allowed to edit all of the places where <u>old</u> occurs.	+ + + + + +
rename[old;new;types;files;method]	Currently there are two different methods for determining which functions are to be examined. If <u>method</u> =EDITCALLERS, <u>editcallers</u> (Section 9) is used to search <u>files</u> . If <u>method</u> =MASTERSCOPE, then the Masterscope data base is used instead. ¹⁷⁷ <u>method</u> =NIL defaults to the value of <u>defaultrenamemethod</u> (initially EDITCALLERS).	+ + + + + +
compare[name1;name2;type;source1;source2]	First performs copydef[old;new;type] for all <u>type</u> inside <u>types</u> . It then calls <u>changecallers</u> to change all occurrences of <u>old</u> to <u>new</u> , and then "deletes" <u>old</u> with <u>deldef</u> . For example, if the user has a function <u>fool</u> which he now wishes to call <u>fie</u> , he simply performs rename[F001; FIE], and <u>fie</u> will be given <u>fool</u> 's definition, and all places that <u>fool</u> are called will be changed to call <u>fie</u> instead.	+ + + + + +

¹⁷⁷ The latter method is more efficient if a Masterscope data base already exists, otherwise the functions on files will have to be analyzed.

+ **comparedefs**[name;type;sources]
+ Calls comparlists on all pairs of definitions of name as a type
+ obtained from the various sources.

+ **hasdef**[name;type] returns T if name is the name of something of type type.

+ **typesof**[name;-;-] Returns a list of the types for which name has a definition.

+ DEFINING NEW FILE PACKAGE TYPES

+ The functions in the previous discussion provide type-independent ways of manipulating typed
+ definitions, e.g. **getdef**[name;type] will obtain the definition of name regardless of which particular
+ file package type is specified by type. This section describes how these operations are specified for
+ new file package types.¹⁷⁸

+ **filepkgtype**[type;prop₁;val₁;...;prop_n;val_n]
+ lambda, nospread function for defining new file package types, or
+ changing attributes of existing file package types. prop is one of:

+ **GETDEF** val is a function of two arguments, name and type,¹⁷⁹ which returns
+ the current definition of name as a type type.¹⁸⁰ Used by getdef.

+ **PUTDEF** val is a function of three arguments, name, type, and definition
+ which stores definition. Used by putdef.

+ **DELDEF** val is a function of two arguments, name, and type, which removes
+ the definition of name as a type that is currently in effect. Used
+ by deldef.

+ **NEWCOM** val is a function of four arguments, name, type, listname, and file.
+ Specifies how to make a new (instance of a) file package command
+ to dump name, an object of type type. Used by addtofile and
+ showdef. The function should return the new command. listname is
+ non-NIL, means the user specified listname as the filevar in his
+ interaction with addtofiles?. If no NEWCOM is specified, the default
+ is to use defaultmakew newcom. defaultmakew newcom will construct
+ and return a command of the form (type * filevar), where filevar is
+ listname if listname is non-NIL, otherwise filevar is the value of

+ ¹⁷⁸ or respecified for existing file package types.

+ ¹⁷⁹ The type argument is provided here and for the other properties so that the user may have the same function for
+ more than one type.

+ ¹⁸⁰ If there is no GETDEF property, a file package command for dumping name is created (by makew newcom). This
+ command is then used to write the definition of name as a type type onto the file FILEPKG.SCRATCH, and this
+ S-expression is then read back in and returned as the current definition.

	filecoms[file;type]. In either case, <u>filevar</u> will be set to list[name]. <u>defaultmakenewcom</u> can be advised or redefined by the user.	+
WHENCHANGED	<u>val</u> is a list of functions to be applied to <u>name</u> , <u>type</u> , and <u>newflg</u> (see discussion of <u>markaschanged</u> , page 14.67) when <u>name</u> , an instance of type <u>type</u> , is changed or defined. Used for various applications, e.g. when an object of type I.S.OPRS changes, it is necessary to clear the corresponding translations from <u>clisparray</u> . ¹⁸¹	+
WHENFILED	<u>val</u> is a list of functions to be applied to <u>name</u> , <u>type</u> , and <u>file</u> when <u>name</u> , an instance of type <u>type</u> , is added to <u>file</u> .	+
DESCRIPTION	<u>val</u> is a string which describes what instances of this type are, e.g. for type RECORDS, description is "record declarations".	+
DEFINING NEW FILE PACKAGE COMMANDS		+
This section describes how the user can specify the various attributes of a file package command for a new command. ¹⁸²		+
filepkgcom[commandname;prop ₁ ;val ₁ ;...;prop _n ;val _n]	lambda, nospread function for defining new file package commands, or changing attributes of existing file package commands. <u>prop</u> is one of:	+
MACRO	defines how to dump the file package command <u>commandname</u> . Used by <u>makefile</u> . <u>val</u> is a pair (args . coms). The "arguments" to <u>commandname</u> are substituted for <u>args</u> throughout <u>coms</u> , and the result treated as a list of file package commands. ¹⁸³ For example, following filepkgcom[FOO;MACRO;((X Y) . coms)], then the file package command (FOO A B) will cause A to be substituted for X and B for Y throughout <u>coms</u> , and then <u>coms</u> treated as a list of commands ¹⁸⁴	+
ADD	Specifies how (if possible) to add an instance of an object of a particular type to a given file package command. Used by <u>addtofile</u> . <u>val</u> is <u>fn</u> , a function of three arguments, <u>com</u> , a file package command <u>car</u> of which is <u>eq</u> to <u>commandname</u> , <u>name</u> , a typed	+

-
- ¹⁸¹ Note: the WHENCHANGED function is called before the object is marked as changed, so that it can, in fact, decide that the object is *not* to be marked as changed, and retfrom[MARKASCHANGED].
- ¹⁸² or respecify for an existing command.
- ¹⁸³ filevars are evaluated *before* substitution, i.e. If the atom * follows name in the command, caddr of the command is evaluated substituting in coms.
- ¹⁸⁴ The substitution is carried out by subpair (Section 6), so that the "argument list" for the macro can also be atomic. For example, if (X . coms) was used instead of ((X Y) . coms), then the command (FOO A B) would cause (A B) to be substituted for X throughout coms.

+ object, and type, its type.¹⁸⁵ fn should return T if it (undoably)
+ adds name to com, NIL if not. If no ADD property is specified, then
+ the default is (1) if car[com]=type and cdr[com]=*, and
+ caddr[com] is a filevar (i.e. a literal atom), add name to the value of
+ the filevar, or (2) if car[com]=type and cdr[com] is not *, add
+ name to cdr[com].

+ **DELETE** Specifies how (if possible) to delete an instance of an object of a
+ particular type from a given file package command. Used by
+ delfromfile. val is fn, a function of three arguments, com, name,
+ and type, same as for ADD. fn should return T if it (undoably)
+ deletes name from com,¹⁸⁶ NIL if not. If no DELETE property is
+ specified, then the default is (1) if car[com]=type and cdr[com]=*,
+ and caddr[com] is a filevar (i.e. a literal atom), and name is
+ contained in the value of the filevar, then remove name from the
+ filevar, or (2) if car[com]=type and cdr[com] is not *, and name is
+ contained in cdr[com], then remove name from cdr[com].

+ **CONTAIN** Specifies whether an instance of an object of a given type is
+ contained in a given file package command. Used by whereis and
+ infilecoms?. val is a function of three arguments, com, a file
+ package command car of which is eq to commandname, name, and
+ type. The interpretation of name is as follows: if name is NIL, fn
+ should return a list of elements of type type contained in com. If
+ name is T, fn should return T if there are *any* elements of type type
+ in com. If name is an atom other than T or NIL, return T if name
+ of type type is contained in com. Finally, if name is a list, return a
+ list of those elements of type type contained in com that are also
+ contained in name.¹⁸⁷ If the CONTAIN property is not provided, the
+ command is simply expanded according to its MACRO definition,
+ and each command on the resulting command list is then

185 Actually, the function is given a fourth argument, near, which if non-NIL, means the function should try to add the item after near. See (6) in discussion of addtofiles?, page 14.67.

186 If the function returns the value of ALL, it means that the command is now "empty", and can be deleted entirely from the command list.

187 Note that it is sufficient for the CONTAIN function to simply return the list of items of type type in command com, i.e. it can in fact ignore the name argument. The name argument is supplied mainly for those situations where producing the entire list of items involves significantly more computation or creates more storage than simply determining whether a particular item (or any item) of type type is contained in the command.

interrogated.¹⁸⁸ ¹⁸⁹

FUNCTIONS FOR MANIPULATING FILE COMMAND LISTS

infilecoms?[name;coms;type;-] coms is a list of file package commands, or a variable whose value is a list of file package commands. type is a file package type. If name=NIL, infilecoms? returns a list of all elements of type type.¹⁹⁰ If name=T, infilecoms? returns T if there are *any* elements of type type.¹⁹¹ Otherwise, infilecoms? returns T if name is "contained" in coms.¹⁹²

addtofile[name;file;type] adds name of type type to the file package commands for file. Uses addtocoms and makenewcom. Value is file. addtofile is undoable.

delfromfiles[name;files;type] deletes all instances of name of type type from the file package commands for each of the files on files. If files is a non-NIL atom, list[files] is used. files=NIL defaults to filelst, i.e. delfromfiles[fn] will delete the function fn from all files that it appears on the user's current working environment.¹⁹³ Value is a list of files from which name was actually removed. Uses delfromcoms. delfromfiles is undoable.

addtocoms[coms;name;type;-;-] adds name as a type to coms, a list of file package commands or a

¹⁸⁸ Note that if commandname is a file package command that is used frequently, its expansion by the various parts of the system that need to interrogate files can result in a large number of conses and garbage collections. By informing the file package as to what this command actually does and does not produce via the CONTAIN property, this expansion is avoided. For example, suppose the user has a file package command called GRAMMARS which dumps various property lists but no functions. Thus, the file package could ignore this command when seeking information about FNS.

¹⁸⁹ If a CONTAIN property is specified and the corresponding function application returns NIL and car[com]=type, then the operation indicated by name is performed (1) on the value of caddr[com], if cadr[com]=*, otherwise (2) on cdr[com]. In other words, by specifying a CONTAIN property that returns NIL, e.g. the function nill, the user specifies that a file package command of name FOO produces objects of file package type FOO and only objects of type FOO.

¹⁹⁰ e.g. filefnlist and bcompl and brecompile use this option.

¹⁹¹ makefile uses this option to determine whether the file contains any FNS, and therefore should be compiled, and if so, whether it contains any BLOCKS, to determine whether to call bcompl/brecompile or tcompl/recompile.

¹⁹² whereis uses infilecoms? in this way.

¹⁹³ Deleting a function will also remove the function from any BLOCKS declarations in the fileCOMS.

- + variable whose value is a list of file package commands.¹⁹⁴ Value is NIL if addtocom was unable to find a command appropriate for adding name to.¹⁹⁵ addtocom is undoable.
- + delfromcoms[coms;name;type] deletes name as a type from coms. Value is NIL if delfromcoms was unable to modify coms to delete name.¹⁹⁶ delfromcoms is undoable.
- + makenewcom[name;type] value is a file package command for dumping name of type type. Uses the procedure described in the discussion of NEWCOM, page 14.72.
- + moveitem[name;type;fromfile;tofile] Moves the definition of name as a type from fromfile to tofile by modifying the file commands in the appropriate way (with delfromfile and addtofile).
- * filecomslist[file;type;-] returns a list of items of type type in file, e.g. filecomslist[FOO;MACROS], filecomslist[FOO;RECORDS], etc.¹⁹⁷ filecomslist knows about expanding user defined file package commands.
- + filefnslst[file] same as filecomslist[file;FNS].
- + filecoms[file;type] same as pack*[file;or[type;COMS]]. e.g. filecoms[FOO] is the atom FOOCOMS, not the value of FOOCOMS.
- + smashfilecoms[file] maps down filecomslist[file;FILEVARS] and sets to NOBIND all filevars, i.e. any variable used in a command of the form (command * variable). Also sets filecoms[file] to NOBIND. Value is file.

¹⁹⁴ Note that coms does *not* have to correspond to a file command list, i.e. the value of fileCOMS for some file. For example, coms can be the list of commands generated as a result of expanding a user defined file package command. Note also that the exact algorithm for adding commands depends the particular command itself. See discussion of the ADD property, in description of filepkgcom, page 14.73.

¹⁹⁵ Note: addtocom will not attempt to add an item to any command which is inside of a DECLARE: unless the user specified a specific name via the LISTNAME or NEAR option of addtofiles?

¹⁹⁶ See previous footnote.

¹⁹⁷ type can also be the name of a file package *command*, e.g. filecomslist[file;BLOCKS] will return the list of all BLOCKS declaration in file.

REMAKING A SYMBOLIC FILE

Most of the time that a symbolic file is written using makefile, only some, usually a few, of the functions that it contains have been changed since the last time the file was written. A considerable savings in time is afforded by copying the prettprinted definitions of those functions that have not changed from an earlier version of the symbolic file, and prettyprinting only those functions that have been changed.¹⁹⁸ makefile will operate in this way, and remake the symbolic file, when makefileremakeflg = T (its initial value).¹⁹⁹ When makefile is remaking a symbolic file, the user can either explicitly indicate those functions which are to be prettyprinted via the third argument to makefile, reprintfns, and indicate the file to be used for copying the rest of the function definitions from via the fourth argument to makefile, sourcefile. The more typical usage however is to default both of these arguments to NIL. In this case, reprintfns will be set to those functions that have been changed since the last version of the file was written. For sourcefile, makefile obtains the full name of the most recent version of the file (that it knows about) from the FILEDATES property, and checks to make sure that the file still exists, and has the same file date as that stored on the FILEDATES property. If it does, makefile uses that file as sourcefile.²⁰⁰ In the case where the most recent version of the file cannot be found, makefile will attempt to remake using the *original* version of the file, i.e., the one first loaded, and specifying as reprintfns the union of all changes that have been made since the file was first loaded, which it obtains from the FILECHANGES property. If both of these fail, makefile prints the message "CAN'T FIND EITHER THE PREVIOUS VERSION OR THE ORIGINAL VERSION OF file, SO IT WILL HAVE TO BE WRITTEN ANEW", and does not remake the file, i.e. will prettyprint all of the functions.

When a remake is specified, makefile also checks the state of the file (cdar of the FILE property) to see how the file was originally loaded (page 14.63). If the file was originally loaded as a compiled file, makefile will automatically call loadvars to obtain those DECLARE: expressions that are contained on the symbolic file, but not the compiled file, and hence have not been loaded. If the file was loaded by loadfns (but not loadfrom), then loadvars will automatically be called to obtain any non-DEFINEQ expressions.²⁰¹

¹⁹⁸ Remaking a symbolic file does *not* depend on the earlier version having a file map, although it is considerably faster if one does exist. In the case of a remake where no file map is available, makefile scans the file looking for the corresponding definition whenever it is about to copy the definition to the new file. The scan utilizes skread (page 14.16), and makefile does not begin scanning from the beginning of the file each time, but instead "walks through" the original file as it is writing the new file. Since the functions are for the most part in the same order, makefile never has to scan very far. However, makefile also builds a map of the functions it has skipped over so that if the order of functions is reversed in the new file, makefile is able to back up and pick up a function previously skipped. The net result is still a significant savings over (re)prettyprinting the entire file, although not as great a savings as occurs when a map is available.

¹⁹⁹ The user can override this default for particular files by specifying the NEW option in the call to makefile (page 14.64). Or, the user can set makefileremakeflg to NIL and explicitly indicate those files which he wants to be remade via the REMAKE option.

²⁰⁰ This procedure permits the user to load or loadfrom a file in a different directory, and still be able to makefile-remake.

²⁰¹ If the file has never been loaded or dumped, e.g., the user simply set up the fileCOMS himself, then makefile will never attempt to remake the file, regardless of the setting of makefileremakeflg, or whether the REMAKE option was specified.

PRETTYDEF FUNCTIONS

prettydef[prttyfns; prtmyfile; prttycoms; reprintfns; sourcefile; changes]

Writes a symbolic file in prettyprint format for loading. prettydef uses filerdb1 as its readable. The value of prettydef is the name of the symbolic file that was created. prettydef operates under a resetlst (see Section 5). If an error occurs, or a control-D is typed, all files that prettydef has opened will be closed, the (partially complete) file being written will be deleted, and any undoable operations executed will be undone.²⁰²

prttyfns is an optional list of function names.²⁰³

prtmyfile is the name of the file on which the output is to be written. If prtmyfile=NIL, the primary output file is used. If prtmyfile is atomic the file is opened if not already open, and it becomes the primary output file. prtmyfile is closed at end of prettydef, and the primary output file is restored. Finally, if prtmyfile is a list, car of prtmyfile is assumed to be the file name, and is opened if not already open. In this case, the file is left open at end of prettydef.

prttycoms is a list of file package commands interpreted as described previously. If prttycoms is atomic, its top level value is used and an rpaqq is written which will set that atom to the list of commands when the file is subsequently loaded. A prettycomprint expression (see below) will also be written which informs the user of the named atom or list of commands when the file is subsequently loaded.²⁰⁴

reprintfns and sourcefile are for use in conjunction with remaking a file (see page 14.77). reprintfns can be a list of functions to be prettyprinted, or EXPRS, meaning prettyprint all functions with EXPR definitions, or ALL meaning prettyprint all functions either defined as exprs, or with EXPR properties.²⁰⁵ sourcefile is the name of the file from which to copy the definitions for those functions that are *not* going to be prettyprinted, i.e., those not specified by reprintfns. sourcefile=T means use most recent version (i.e.,

202 Since prettydef operates under a resetlst, any resetsaves executed in the file package commands will also be protected, i.e., restored. For example, if one of the file package commands executes a (RESETSAVE (LINELENGTH 100)), the linelength will automatically be restored.

* 203 prttyfns is an anachronism from when prettydef did not take as its third argument a list of file package commands. It is equivalent to including (FNS * prttyfns) in the file package commands, and is rarely used now.

204 In addition, if any of the functions in the file are nlambdas, prettydef will automatically print a DECLARE: expression suitable for informing the compiler about these functions, in case the user recompiles the file without having first loaded the nlambda functions. For more discussion, see Section 18.

205 Note that doing a remake with reprintfns=NIL makes sense if there have been changes in the file, but not to any of the functions, e.g., changes to vars or property lists.

highest number) of prttyfile, the second argument to prettydef. If sourcefile cannot be found, prettydef prints the message "file NOT FOUND, SO IT WILL BE WRITTEN ANEW", and proceeds as it does when printfns and sourcefile are both NIL.

printfns[x;-]

x is a list of functions. printfns prints defineq and prettyprints the functions to primary output file using primary readtable. Used by prettydef, i.e., command (FNS * FOO) is equivalent to command (E (PRINTFNS FOO)).

printdate[file;changes]

prints the filecreated expression at beginning of prettydef files. changes is for use by the file package.

filecreated[x]

Nlambda function. Prints a message (using lispxprint) followed by the time and date the file was made, which is car[x]. The message is the value of prettyheader, initially "FILE CREATED". If prettyheader=NIL, nothing is printed. cdr[x] contains information about the file, e.g., full name, address of file map, list of changed items, etc. filecreated also stores the time and date the file was made on the property list of the file under the property FILEDATES and performs other initialization for the file package.

prettyheader

value is message printed by filecreated. prettyheader is initially "FILE CREATED". If prettyheader=NIL, neither filecreated nor prettycomprint will print anything. Thus, setting prettyheader to NIL will result in "silent loads". For example, prettyheader is reset to NIL during greeting (Section 22).

filedate[file]

returns the file date of file, i.e. the date contained in the FILECREATED expression. + +

prettycomprint[x]

prints x using lispxprint, unless value of prettyheader=NIL.

printdef[expr;left;def;tailflg;-]

prints the expression expr in a pretty format on the primary output file using the primary readtable. left is the left hand margin (linelength determines the right hand margin.)²⁰⁶

def=T means expr is a function definition, or a piece of one, i.e., prettyprint is essentially printdef[getd[fn];NIL;T]. If def=NIL, no special action will be taken for LAMBDA's, PROG's, COND's, comments, CLISP, etc. def is NIL when prettydef calls prettyprint

²⁰⁶ printdef initially performs (TAB LEFT T), which means to space to position left, unless already beyond this position, in which case do nothing. + +

to print variables and property lists, and when printdef is called from the editor via the command PPV.

tailflg=T means expr is a tail of a list, and is printed without parentheses.

comment1[l;-]

prints the comment 1. comment1 is a separate function to permit the user to intercept the printing of comments, perform some operation, e.g., reset the line length, print the comment himself, and then restore the line length. For example, this could be accomplished by adding (* LAMBDA (X) (RESETFORM (LINELENGTH 100) (COMMENT1 X))) to prettyprintmacros (page 14.49).²⁰⁷

²⁰⁷

* comment1 is an entry to the prettyprint block. However, it is called internally by prettyprint so that advising or redefining it will not affect the action of prettyprint. comment1 should *not* be called when not under a printdef.

Index for Section 14

	Page Numbers
ABBREVLST (prettyprint variable/parameter)	14.46,48
addressable files	14.7-9
ADDTOCOMS[COMS;NAME;TYPE]	14.75
ADDTOFILE[NAME;TYPE;FILE;NEAR]	14.75
ADDTOFILES?[NOASKSTR]	14.67
ADDVARS (file package command)	14.56
ADVICE (file package command)	14.58
ADVISE (file package command)	14.58
AFTERSYSOUTFORMS (system variable/parameter)	14.37,35
ALISTS (file package command)	14.56
ALL (use in file package PROP command)	14.57
ALLPROP (as value of DFNFLG)	14.38
ALONE (type of read-macro)	14.27
ALWAYS (type of read-macro)	14.27
ARG NOT LITATOM (error message)	14.3
ARRAY (file package command)	14.59
BAD FILE PACKAGE COMMAND (error message)	14.60
BCOMPL[FILES;CFILE;NOBLOCKSFLG]	14.65
BEFORESYSOUTFORMS (system variable/parameter)	14.37
bell (printed by system)	14.19
BKLINBUF[X] SUBR	14.34
BKSYSBUF[X] SUBR	14.34
block declarations	14.58
BLOCKS (file package command)	14.58
Boyer-Moore fast string searching algorithm	14.9
break characters	14.26,13-15,33
BREAKRESETFORMS (break variable/parameter)	14.29
BRECOMPILE[FILES;CFILE;FNS;NOBLOCKSFLG]	14.65-66
BUILDMAPFLG (system variable/parameter)	14.41
C (makefile option)	14.65
carriage-return	14.11-12,15-17,30
CHANGECALLERS[OLD;NEW;TYPES;FILES;METHOD]	14.71
CHANGECHAR (prettyprint variable/parameter)	14.49
CHANGEFONT[FONTCLASS]	14.52,50
CHARDELETE (syntax class)	14.30-31
CLEANUP[FILES] NL*	14.66
CLEANUOPTIONS (file package variable/parameter)..	14.66
CLEARBUF[FILE;FLG] SUBR	14.34
clearing input buffer	14.19,32
clearing output buffer	14.19
CLISPFONT	14.50
CLISPIFY[X;L]	14.65
CLISPIFY (makefile option)	14.65
CLISPIFYPRETTYFLG (prettyprint variable/parameter)	14.49,65
CLOSEALL[]	14.5,10
CLOSEF[FILE]	14.5
CLOSEF?[FILE]	14.5
closing and reopening files	14.9-11
CNTRLV (syntax class)	14.30
comment pointers	14.45
COMMENTFLG (prettyprint variable/parameter)	14.48,44-45
COMMENTFONT	14.50
COMMENTLINELENGTH (prettyprint variable/parameter)	14.51
comments (in listings)	14.44-46
COMMENT1[L;INBLOCKFLG]	14.80
COMPARE[NAME1;NAME2;TYPE;SOURCE1;SOURCE2]	14.71

Page
Numbers

COMPAREDEFS[NAME;TYPE;SOURCES]	14.72
COMPILEFILES[FILES] NL*	14.66
COMS (file package command)	14.57
CONTROL[FLG;RDTBL] SUBR	14.33,12,14,32
control character echoing	14.30
control-A	14.32,11-14,29-31,33
control-D	14.34-35
control-E	14.34
control-F (in file name)	14.2
control-H	14.34
control-O	14.19
control-P	14.19,34
control-Q	14.32,11-14,29-31,33
control-R	14.30
control-S	14.34
control-U (TOPS-20)	14.30
control-V	14.11,13-14,30
control-W	14.12
control-X (edit command)	14.36
control-Y (as a read-macro)	14.27,36
control-Z (edit command)	14.36
control-Z (TOPS-20)	14.14,32
COPY (DECLARE: option)	14.59
COPYBYTES[SRCFIL;DSTFIL;START;END]	14.9
COPYDEF[OLD;NEW;TYPE;SOURCE;OPTIONS]	14.70
COPYREADTABLE[RDTBL] SUBR	14.24
COPYTERMTABLE[TTBL] SUBR	14.29
COPYWHEN (DECLARE: option)	14.59
CTRLV (syntax class)	14.30
DECLARE	14.58
DECLARETAGSLST (prettyprint variable/parameter)	14.59
DECLARE:[X] NL*	14.59
DECLARE: (file package command)	14.58,59
DEFAULTFONT	14.50
DEFAULTMAKENEWCOM[NAME;TYPE;LISTNAME;FILE]	14.72
defining file package commands	14.73-75
defining file package types	14.72-73
DEFPRT[TYPE;FN]	14.22
DELDEF[NAME;TYPE]	14.70
DELETECHAR (syntax class)	14.30,29
DELETECONTROL[TYPE;MESSAGE;TTBL]	14.31
DELETELINE (syntax class)	14.30,29
DELFILE[FILE]	14.5
DELFROMCOMS[COMS;NAME;TYPE]	14.76
DELFROMFILES[NAME;TYPE;FILES]	14.75
DFNFLG (system variable/parameter)	14.38
DOBE[]	14.18
DOCOPY (DECLARE: option)	14.59
DOEVAL@COMPILE (DECLARE: option)	14.59
DOEVAL@LOAD (DECLARE: option)	14.59
DONTCOPY (DECLARE: option)	14.59
DONTEVAL@COMPILE (DECLARE: option)	14.59
DONTEVAL@LOAD (DECLARE: option)	14.59
dumping circular lists	14.22
dumping unusual data structures	14.22
E (file package command)	14.57
E (in a floating point number)	14.12

Page
Numbers

E (use in comments)	14.49
ECHOCONTROL[CHAR;MODE;TTBL]	14.30
echoing	14.30
ECHOMODE[FLG;TTBL] SUBR	14.30
EDITCHARACTERS (system variable/parameter)	14.36
EDITDEF[NAME;TYPE;SOURCE;EDITCOMS]	14.70
EDITRDTBL (system variable/parameter)	14.23
END OF FILE (error message)	14.7,11
ENDFILE[FILE]	14.42
end-of-line	14.11,8,17
EOL (syntax class)	14.30
ERRORTYPELIST (system variable/parameter)	14.2
ESC (type of read-macro)	14.27
ESCAPE[FLG] SUBR	14.14,14
escape character	14.11
ESCAPE (syntax class)	14.25
ESCQUOTE (type of read-macro)	14.27
EVAL@COMPILE (DECLARE: option)	14.59
EVAL@COMPILEWHEN (DECLARE: option)	14.59
EVAL@LOAD (DECLARE: option)	14.59
EVAL@LOADWHEN (DECLARE: option)	14.59
EXPR (property name)	14.38
fast symbolic dump	14.49
FAST (makefile option)	14.64
FC	14.57
FFILEPOS[PATTERN;FILENAME;FILESTART;FILEEND;SKIP; TAIL;CASEARRAY]	14.9
file attributes	14.4-5
file command list	14.56
file maps	14.41-42
file names	14.2,3,6-7
FILE NOT COMPATIBLE (error message)	14.36
FILE NOT FOUND (error message)	14.2-3,38
FILE NOT OPEN (error message)	14.2,5
file package	14.54-81
file package commands	14.56-61,73-75
file package functions	14.64-69,75-76
file package types	14.55-56,61-62,72-73
file pointer	14.7-8
FILE SYSTEM RESOURCES EXCEEDED (error message)	14.3
FILE WONT OPEN (error message)	14.3
FILE WON'T OPEN (error message)	14.1-2
FILE (property name)	14.63-64
FILECHANGES (property name)	14.63,64,77
FILECOMS[FILE;X]	14.76
fileCOMS (in file package)	14.39,54,63-65,76
FILECOMSLST[FILE;TYPE;FLG]	14.76
FILECREATED[X] NL*	14.79,60
FILEDATE[FILE;CFLG]	14.79
FILEDATES (property name)	14.63,64,77,79
FILEFNSLST[FILE]	14.76
FILEGROUP (property name)	14.66
FILELINELENGTH (file package variable/parameter)	14.64,48
FILELST (file package variable/parameter)	14.63,66
FILEMAP DOES NOT AGREE WITH CONTENTS OF file-name (error message)	14.42
FILEMAP (property name)	14.41

Page
Numbers

FILENAMEFIELD[FILENAME;FIELDNAME]	14.6
FILEPKGCHANGES[N] *	14.67
FILEPKGCOM[COMNAME;PROP1;VAL1;...;PROFn;VALn] * ..	14.73
FILEPKGCOMS (file package command)	14.58, 60
FILEPKGCOMSPLST (prettyprint variable/parameter)..	14.60
FILEPKGFLG (file package variable/parameter) ..	14.54
FILEPKGTYPE[TYPE;PROP1;VAL1;...;PROFn;VALn] * ..	14.72
FILEPKGTYPES (file package variable/parameter) ..	14.61
FILEPKG.SCRATCH (file package variable/parameter).	14.72
FILEPOS[PATTERN;FILENAME;FILESTART;FILEEND;SKIP; TAIL;CASEARRAY]	14.8
FILERDTBL (system variable/parameter)	14.16, 38, 42, 78
files	14.1-11
FILES?[]	14.67, 64
FILETYPE (property name)	14.64
filevar (in file package)	14.60, 39, 76
FIRST (type of read-macro)	14.27
FIRSTCOL (prettyprint variable/parameter)	14.48
FIX format (in printnum package)	14.20
FLOAT format (in printnum package)	14.20
floating point numbers	14.12
FLTFMT[FORMATBITS]	14.35
FNS (file package command)	14.56
font package	14.49-52
FONTCHANGEFLG (prettyprint variable/parameter) ..	14.51
FONTDEFS (prettyprint variable/parameter)	14.52
FONTDEFSVARS (prettyprint variable/parameter) ...	14.52
FONTEXCAPECHAR (prettyprint variable/parameter) ..	14.51
FONTNAME[NAME]	14.52
FONTPROFILE (prettyprint variable/parameter)	14.51
FONTSET[NAME]	14.52
format characters	14.25
full file name	14.2-3
FULLNAME[X;RECOG]	14.3
GETBRK[RDTBL] SUBR	14.14
GETCOMMENT[X;DESTFL;DEF]	14.45
GETCONTROL[TTBL]	14.33
GETDEF[NAME;TYPE;SOURCE;OPTIONS]	14.69
GETDELETECONTROL[TYPE;TTBL]	14.31
GETECHOMODE[TTBL]	14.31
GETEOFPTR[FILE] SUBR	14.8
GETFILEINFO[FILE;ATTRIB]	14.4
GETFILEMAP[FILE;FL]	14.41
GETFILEPTR[FILE] SUBR	14.7
GETRAISE[TTBL]	14.32
GETREADTABLE[RDTBL] SUBR	14.24
GETSEPR[RDTBL] SUBR	14.14
GETSYNTAX[CH;TABLE]	14.25
GETTERMTABLE[TTBL] SUBR	14.29
GET* (edit command)	14.45
GLOBALVARS (file package command)	14.59
HASDEF[NAME;TYPE;SPELLFLG]	14.72
HCOPYALL[X]	14.23
HERALDSTRING (system variable/parameter)	14.37
HORRIBLEVARS (file package command)	14.59, 23
HPRINT[EXPR;FILE;UNCIRCULAR]	14.23
HREAD[FILE]	14.23

Page
Numbers

IFPROP (file package command)	14.57,60
ILLEGAL ARG (error message)	14.3,30
ILLEGAL READTABLE (error message)	14.24,30
ILLEGAL TERMINAL TABLE (error message)	14.29-30
IMMED (type of read-macro)	14.28
IMMEDIATE (type of read-macro)	14.28
INFILE[FILE] SUBR	14.1,7
INFILECOMS?[NAME;TYPE;COMS;ONFILETYPE]	14.75
INFILEP[FILE] SUBR	14.2,2
INFIX (type of read-macro)	14.26
INITRECORDS (file package command)	14.58
INPUT[FILE] SUBR	14.1
input buffer	14.32,15,19,34
input functions	14.11-16
inputting numbers	14.12
input/output	14.1-81
INREADMACROP[] SUBR	14.28
IOFILE[FILE] SUBR	14.4,7-8
I.S.OPRS (file package command)	14.58
JSYS	14.35
LAMBDAFONT	14.50
LASTC[FILE] SUBR	14.15
LCASELST (prettyprint variable/parameter)	14.46
LEFTBRACKET (syntax class)	14.25
LEFTPAREN (syntax class)	14.25
LINBUF[FLG] SUBR	14.34
line buffer	14.32,34
LINEDELETE (syntax class)	14.30-31
LINELENGTH[N] SUBR	14.35,48
line-buffering	14.32,12,14-15,33
line-feed	14.11,17
line-feed (edit command)	14.36
LISPXCOMS (prog. asst. variable/parameter)	14.58
LISPXMACROS (file package command)	14.58
LISPXREADFN (prog. asst. variable/parameter)	14.15
LIST (makefile option)	14.65
LISTFILES[FILES] NL*	14.66,64-65
LISTFILESTR (file package variable/parameter)	14.51,66
LISTFILES1[FILE]	14.66
literal atoms	14.12
LOAD[FILE;LDFLG;PRINTFLG]	14.38
LOADBLOCK[FN;FILE;LDFLG]	14.40
LOADCOMP[FILE;LDFLG]	14.40
LOADCOMP?[FILE;LDFLG]	14.41
LOADDEF[NAME;TYPE;SOURCE]	14.71
LOADEDFILELST (file package variable/parameter)	14.63
LOADEFS[FNS;FILE]	14.40
LOADFNS[FNS;FILE;LDFLG;VARS]	14.39
LOADFROM[FILE;FNS;LDFLG]	14.40
LOADOPTIONS (system variable/parameter)	14.39
LOADVARS[VARS;FILE;LDFLG]	14.40
LOAD?[FILE;LDFLG;PRINTFLG]	14.39
LOCALVARS (file package command)	14.59
lower case comments	14.45-46
lower case input	14.32
MACRO (type of read-macro)	14.26
MACROS (file package command)	14.59

Page
Numbers

MAKEFILE[FILE;OPTIONS;REPRINTFNS;SOURCEFILE]	14.64-66
MAKEFILEFORM (file package variable/parameter) ..	14.66
MAKEFILEOPTIONS (file package variable/parameter).	14.65
MAKEFILEREMAKEFLG (file package variable/parameter)	14.77,65
MAKEFILES[OPTIONS;FILES]	14.66
MAKENEWCOM[NAME;TYPE;LISTNAME;FILE]	14.76
manipulating file names	14.6-7
margins (for prettyprint)	14.79
MARKASCHANGED[NAME;TYPE;NEWFLG]	14.67
MOVEITEM[TOFILE;NAME;TYPE;FROMFILE]	14.76
NCHARS[X;FLG;RDTBL] SUBR	14.8
NEW (makefile option)	14.65
NILCOMS (file package variable/parameter)	14.68
NO FILE PACKAGE COMMAND FOR (error message)	14.58
NO PROPERTY FOR (error message)	14.57
NO USERMACRO FOR (error message)	14.58
NOBIND	14.38
NOCLISP (makefile option)	14.65
NOESC (type of read-macro)	14.27
NOESCQUOTE (type of read-macro)	14.27
NOFIXFNSLST (dwim variable/parameter)	14.40
NOFIXVARSLST (dwim variable/parameter)	14.40
NONE (syntax class)	14.30
NONIMMED (type of read-macro)	14.28
NONIMMEDIATE (type of read-macro)	14.28
NORAISe (TENEX command)	14.32
NORMALCOMMENTSFLG (system variable/parameter) ...	14.45
NOT DUMPED (error message)	14.65
NOT FOUND (error message)	14.66
NOT FOUND, SO IT WILL BE WRITTEN ANEW (error message)	14.79
(NOT PRINTABLE)	14.43
NOTCOMPILEDFILES (file package variable/parameter)	14.64,66
noticing files	14.63
NOTLISTEDFILES (file package variable/parameter)..	14.64,66
NOT-FOUND:	14.39
numbers	14.12
NUMFORMATCODE[FORMAT;SMASHCODE]	14.22
octal	14.12,17
OPENFILE[FILE;ACCESS;RECOG;BYTESIZE; MACHINE.DEPENDENT.PARAMETERS]	14.3
opening files	14.1-4
OPENP[FILE;ACCESS] SUBR	14.4,2
ORIG (used as a readable)	14.24
ORIGINAL (file package command)	14.60
OUTFILE[FILE] SUBR	14.2,7-8
OUTFILEP[FILE] SUBR	14.3,2
OUTPUT[FILE] SUBR	14.1
output buffer	14.19
output functions	14.17-19
P (file package command)	14.57
PACKFILENAME[FIELDNAME1;FIELDCONTENTS1;...; FIELDNAME _n ;FIELDCONTENTS _n] *	14.6
parentheses counting (by READ)	14.12,32
PEEKC[FILE;RDTBL] SUBR	14.14,33
PF[FN;FROMFILES;TOFILE] NL*	14.44

Page
Numbers

PF*[FN;FROMFILES;TOFILE] NL*	14.44
PLVLFILEFLG (system variable/parameter)	14.19
POSITION[FILE;N] SUBR	14.36
PP[X] NL*	14.43
PPV (edit command)	14.80
PP*[X] NL*	14.44
PRETTYCOMFONT	14.50
PRETTYCOMPRINT[X] NL	14.79
PRETTYDEF[PRTTYFNS;PRTTYFILE;PRTTYCOMS; REPRINTFNS;SOURCEFILE;CHANGES]	14.78,64
PRETTYEQUIVLST (prettyprint variable/parameter)	14.49
PRETTYFLG (prettyprint variable/parameter)	14.49,64
PRETTYHEADER (prettyprint variable/parameter)	14.79
PRETTYLCOM (prettyprint variable/parameter)	14.48
PRETTYPRINT[FNS;PRETTYDEFLG;FNSLST]	14.43
prettyprinting by system functions	14.18
PRETTYPRINTMACROS (prettyprint variable/parameter)	14.49
PRETTYPRINTTYPEMACROS (prettyprint variable/parameter)	14.49
PRETTYTABFLG (prettyprint variable/parameter)	14.43
PRETTYTRANFLG (clisp variable/parameter)	14.65
primary input file	14.1,5,11
primary output file	14.1,5,17
primary readtable	14.23,11,17,30
primary terminal table	14.29-30
PRINT[X;FILE;RDTBL] SUBR	14.17,18
PRINTBELLS[]	14.18
PRINTDATE[FILE;CHANGES]	14.79
PRINTDEF[EXPR;LEFT;DEF;TAILFLG;FNSLST]	14.79,49,80
PRINTFNS[X;PRETTYDEFLG]	14.79
printing numbers	14.20-22
printlevel	14.18-19
PRINTLEVEL[CARN;CDRN] SUBR	14.18
PRINTNUM[FORMAT;NUMBER;FILE]	14.20
printnum package	14.20-22
PRIN1[X;FILE] SUBR	14.17,18
PRIN2[X;FILE;RDTBL] SUBR	14.17,18
PRIN3[X;FILE] SUBR	14.17
PRIN4[X;FILE;RDTBL] SUBR	14.17
PROP (file package command)	14.57,60
PROPS (file package command)	14.57
PUTDEF[NAME;TYPE;DEFINITION]	14.70
Q (following a number)	14.12,17,34
RADIX[N] SUBR	14.34,12,17
RAISE[FLG;TTBL] SUBR	14.32
RAISE (TENEX command)	14.32
RANDACCESSP[FILE]	14.8
RATEST[X] SUBR	14.14
RATOM[FILE;RDTBL] SUBR	14.12,13,33
RATOMS[A;FILE;RDTBL]	14.13
RC (makefile option)	14.65
READ[FILE;RDTBL;FLG] SUBR	14.11,12,33
READC[FILE] SUBR	14.14,33
READFILE[FILE]	14.42
reading from strings	14.11
READLINE[RDTBL;LINE;LISPXFLG]	14.15,16
READMACROS[FLG] SUBR	14.28

Page
Numbers

READP[FILE;FLG] SUBR	14.15
READTABLEP[RDTBL] SUBR	14.24
readtables	14.23,11,17,24-29
READVISE[X] NL*	14.58
read-macro characters	14.25-29
READ-MACRO CONTEXT ERROR (error message)	14.28
read-macro options	14.27
RECOMPILE[PFILE;CFILE;FNS]	14.65-66
RECORDS (file package command)	14.58
(REDEFINED) (printed by system)	14.38
REMAKE (makefile option)	14.65
remaking a file	14.77
REMEMBER (prog. asst. command)	14.67
RENAME[OLD;NEW;TYPES;FILES;METHOD]	14.71
RENAMEFILE[OLD;NEW]	14.5
RESETBUFS[FORM1;FORM2;...;FORMn]	14.34
RESETFORMS (system variable/parameter)	14.35
RESETREADTABLE[RDTBL;FROM] SUBR	14.24
RESETTERMTABLE[TTBL;FROM] SUBR	14.29
RETYPE (syntax class)	14.30
RIGHTBRACKET (syntax class)	14.25
RIGHTPAREN (syntax class)	14.25
root name of the file	14.54
RPAQ[X;Y] NL	14.38
RPAQQ[X;Y] NL	14.38,78
RSTRING[FILE;RDTBL] SUBR	14.12,13
SAVEDEF[NAME;TYPE;DEFINITION]	14.70
searching files	14.8-9
separator characters	14.26,13-15,33
SEPRCASE[CLISPFLG]	14.9
SETBRK[LST;FLG;RDTBL] SUBR	14.13
SETFILEINFO[FILE;ATTRIB;VALUE]	14.5
SETFILEPTR[FILE;ADR] SUBR	14.8,36
SETINITIALS[]	14.37
SETLINELENGTH[N]	14.35
SETREADMACROFLG[FLG] SUBR	14.28
SETREADTABLE[RDTBL;FLG] SUBR	14.24
SETSEPR[LST;FLG;RDTBL] SUBR	14.13
SETSYNTAX[CH;CLASS;TABLE]	14.25
SETTERMCHARS[NEXTCHAR;BKCHAR;LASTCHAR; UNQUOTECHAR;2CHAR;PPCHAR]	14.36
SETTERMTABLE[TTBL] SUBR	14.29
SHOWDEF[NAME;TYPE;FILE]	14.70
SHOWPRINT[X;FILE;RDTBL]	14.18
SHOWPRIN2[X;FILE;RDTBL]	14.18
SKREAD[FILE;REREADSTRING]	14.16
SMASHFILECOMS[FILE]	14.76
SPACES[N;FILE] SUBR	14.17
SPECVARS (file package command)	14.59
SPELLFILE[FILE;NOPRINTFLG;NSFLG]	14.2
spelling correction	14.59-60
spelling lists	14.59-60
SPLICE (type of read-macro)	14.26
square brackets (inserted by prettyprint)	14.48
STOP (at the end of a file)	14.38,42
STRINGDELIM (syntax class)	14.25
strings	14.11

	Page Numbers
STRPOS[PAT;STRING;START;SKIP;ANCHOR;TAIL]	14.8
symbolic file input	14.38-42
syntax classes	14.24-30
SYNTAXP[CHARCODE;CLASS;TABLE]	14.25
SYSBUF[FLG] SUBR	14.34
SYSIN[FILE] SUBR	14.37
SYSLOAD (load option)	14.38, 39
SYSOUT[FILE] SUBR	14.36, 38
SYSOUTDATE (system variable/parameter)	14.37
SYSOUTFILE (system variable/parameter)	14.37
SYSOUTGAG (system variable/parameter)	14.37
SYSOUTP[FILE]	14.38
SYSOUT.EXT (system variable/parameter)	14.37
SYSPRETTYFLG (system variable/parameter)	14.18, 18
SYSPROPS (system variable/parameter)	14.57
SYSTEMFONT	14.50
TAB[POS;MINSPACES;FILE]	14.17
TCOMPL[FILES]	14.65
TEMPLATES (file package command)	14.58
TENEX	14.8
terminal	14.1, 5, 11-12, 15, 32, 44
terminal syntax classes	14.30
terminal tables	14.29-33
TERMTABLEP[TTBL] SUBR	14.29
TERPRI[FILE] SUBR	14.17
TTYLINELENGTH (system variable/parameter)	14.35
typed definitions	14.55-56
TYPESOF[NAME;POSSIBLETYPES;IMPOSSIBLETYPES]	14.72
UCASELST (prettyprint variable/parameter)	14.46
UGLYVARS (file package command)	14.59, 23
UNMARKASCHANGED[NAME;TYPE]	14.67
UNPACKFILENAME[FILENAME]	14.6
UNSAVEDDEF[NAME;TYPE;DEF]	14.71
UPDATEFILES[PRLST;FLST]	14.63
updating files	14.63
USEMAPFLG (system variable/parameter)	14.42
user defined printing	14.22
USERFONT	14.50
USERMACROS (editor variable/parameter)	14.58
USERMACROS (file package command)	14.58
VARS[FN;USEDATABASE]	14.39
VARS (file package command)	14.56
VARTYPE (property name)	14.62
version numbers	14.2
WAITFORINPUT[FILE]	14.15
WHENCLOSE[FILENAME;PROP1;VAL1;...;PROPN;VALn] *	14.10
whenclose package	14.9-11
WHEREIS[NAME;TYPE;FILES]	14.67
whereis package	14.40, 67, 69
WIDEPAPER[FLG]	14.48
WRITEFILE[X;FILE]	14.42
[,] (inserted by prettyprint)	14.48
(change character)	14.49
"	14.11-12, 14
# (followed by a number)	14.17
#CAREFULCOLUMNS (prettyprint variable/parameter)..	14.48
#RPARS (prettyprint variable/parameter)	14.48

**Page
Numbers**

## (printed by system)	14.32,11,31,33
\$ (<esc>)	14.2
% (escape character)	14.11,12-14,17,33
% (use in comments)	14.46
%% (use in comments)	14.45-46
& (printed by system)	14.19
' (as a read-macro)	14.26
* (as a prettyprint macro)	14.45
* (as a read-macro)	14.45
* (use in comments)	14.44,48
* (use in file package command)	14.60
COMMENT (printed by system)	14.44
COMMENTFLG (prettyprint variable/parameter)	14.44
-- (printed as part of a list structure)	14.19
... (printed following a carriage-return)	14.16
	14.14,32
 (TOPS-20)	14.30
\ (printed by system)	14.11,31-32
\\" (printed by system)	14.12
] (use in input)	14.15
† (use in comments)	14.46

SECTION 15

DEBUGGING - THE BREAK PACKAGE¹

15.1 DEBUGGING FACILITIES

Debugging a collection of LISP functions involves isolating problems within particular functions and/or determining when and where incorrect data are being generated and transmitted. In the Interlisp system, there are three facilities which allow the user to (temporarily) modify selected function definitions so that he can follow the flow of control in his programs, and obtain this debugging information. These three facilities together are called the break package. All three redefine functions in terms of a system function, break1 described below.

Break modifies the definition of its argument, a function fn,² so that if a break condition (defined by the user) is satisfied, the process is halted temporarily on a call to fn. The user can then interrogate the state of the machine, perform any computation, and continue or return from the call.

Trace modifies a definition of a function fn so that whenever fn is called, its arguments (or some other values specified by the user) are printed. When the value of fn is computed it is printed also. (trace is a special case of break).

Breakin allows the user to insert a breakpoint *inside* an expression defining a function. When the breakpoint is reached and if a break condition (defined by the user) is satisfied, a temporary halt occurs and the user can again investigate the state of the computation.

The following two examples illustrate these facilities. In the first example, the user traces the function factorial. trace redefines factorial so that it calls break1 in such a way that it prints some information, in this case the arguments and value of factorial, and then goes on with the computation. When an error occurs on the fifth recursion, break1 reverts to interactive mode, and a full break occurs. The situation is then the same as though the user had originally performed **BREAK(FACTORIAL)** instead of **TRACE(FACTORIAL)**, and the user can evaluate various Interlisp forms and direct the course of the computation. In this case, the user examines the variable n, and instructs break1 to return 1 as the value of this cell to factorial. The rest of the tracing proceeds without incident. The user would then presumably edit factorial to change L to 1.

¹ The break package was written by W. Teitelman.

² break and trace can also be used on clisp operators (Section 23) which appear as car of form, e.g. **FETCH**, **REPLACE**, **IF**, **FOR**, **DO**, etc., even though these are not implemented as functions. For conditional breaking, the user can refer to the entire expression via the variable exp, e.g. **BREAK ((FOR (MEMB 'UNTIL EXP)))**.

In the second example, the user has constructed a non-recursive definition of factorial. He uses breakin to insert a call to break1 just after the PROG label LOOP. This break is to occur only on the last two iterations, i.e., when n is less than 2. When the break occurs, the user looks at the value of n, mistakenly typing NN. However, the break is maintained and no damage is done. After examining n and m the user allows the computation to continue by typing OK. A second break occurs after the next iteration, this time with N=0. When this break is released, the function factorial returns its value of 120.

```

←PP FACTORIAL
(FACTORIAL
 [LAMBDA (N)
 (COND
 ((ZEROP N
 L)
 (T (ITIMES N (FACTORIAL (SUB1 N)))
 FACTORIAL
 ←TRACE(FACTORIAL)
 (FACTORIAL)
 ←FACTORIAL(4)

FACTORIAL:
N = 4

FACTORIAL:
N = 3

FACTORIAL:
N = 2

FACTORIAL:
N = 1

FACTORIAL:
N = 0

U.B.A.
L
(FACTORIAL BROKEN)
:N
0
:RETURN 1
    FACTORIAL = 1
    FACTORIAL = 1
    FACTORIAL = 2
    FACTORIAL = 6
    FACTORIAL = 24
24
←

```

```

←PP FACTORIAL

(FACTORIAL
 [LAMBDA (N)
 (PROG ((M 1))
 LOOP(COND
 ((ZEROP N)
 (RETURN M)))
 (SETQ M (ITIMES M N))
 (SETQ N (SUB1 N))
 (GO LOOP))

FACTORIAL
←BREAKIN(FACTORIAL (AFTER LOOP) (ILESSP N 2]
SEARCHING...
FACTORIAL
←FACTORIAL(5)

((FACTORIAL) BROKEN)
:NN
U.B.A.
NN
(FACTORIAL BROKEN AFTER LOOP)
:N
1
:M
120
:OK
(FACTORIAL)

((FACTORIAL) BROKEN)
:N
0
:OK
(FACTORIAL)
120
←

```

15.2 BREAK1

The basic function of the break package is break1. Whenever Interlisp types a message of the form (- BROKEN) followed by ":" the user is then "talking to" break1, and we say he is "in a break." break1 allows the user to interrogate the state of the world and affect the course of the computation. It uses the prompt character ":" to indicate it is ready to accept input(s), in the same way as evalqt uses "<". The user can type in one of the commands specifically recognized by break1 described below, and the command will be processed, and then another : will be printed, to indicate that break1 is ready for more input.³ Or, the user may type in an expression for evaluation, and the value will be printed. The user can even type in commands to the programmer's assistant (Section 22), e.g. to redo or undo previously executed events, including break commands.⁴

³ unless of course the command caused the break to be exited.

⁴ In fact, all inputs not recognized by break1 are simply passed on to the programmer's assistant.

Since break1 puts all of the power of Interlisp at the user's command, he can do anything he can do at evalqf. For example, he can insert new breaks on subordinate functions simply by typing:

(BREAK fn1 fn2 ...)

or he can remove old breaks and traces if too much information is being supplied:

(UNBREAK fn3 fn4 ...)

He can edit functions, including the one currently broken:

EDITF(fn)

For example, the user might evaluate an expression, see that the value was incorrect, call the editor, change the function, and evaluate the expression again, all without leaving the break.

Similarly, the user can prettyprint functions, define new functions or redefine old ones, load a file, compile functions, time a computation, etc. In short, anything that he can do at the top level can be done while inside of the break. In addition the user can examine the pushdown list, via the functions described in Section 12, and even force a return back to some higher function via the function retfrom or reteval.

It is important to emphasize that once a break occurs, the user is in complete control of the flow of the computation, and the computation will not proceed without specific instruction from him. If the user types in an expression whose evaluation causes an error, the break is maintained. Similarly if the user aborts a computation⁵ initiated from within the break, the break is maintained. Only if the user gives one of the commands that exits from the break, or evaluates a form which does a retfrom or reteval back out of break1, will the computation continue.⁶

Note that break1 is just another Interlisp function, not a special system feature like the interpreter or the garbage collector. It has arguments which are explained later, and returns a value, the same as cons or cond or prog or any other function.⁷ The value returned by break1 is called "the value of the break." The user can specify this value explicitly by using the RETURN command described below. But in most cases, the value of a break is given implicitly, via a GO or OK command, and is the result of evaluating "the break expression," brkexp, which is one of the arguments to break1.

The break expression is an expression equivalent to the computation that would have taken place had no break occurred. For example, if the user breaks on the function FOO, the break expression is the body of the definition of FOO. When the user types OK or GO, the body of FOO is evaluated, and its value returned as the value of the break, i.e., to whatever function called FOO. The effect is the same as though no break had occurred. In other words, one can think of break1

⁵ By typing control-E, see Section 16.

⁶ Except that break1 does not "turn off" control-D, i.e., a control-D will force an immediate return back to the top level.

+ ⁷ Furthermore, since break1 itself calls functions, when one of these is broken, an infinite loop would occur. break1 detects this situation, and prints Break within a break on and the name of the function, and then simply calls the function without going into a break.

as a fancy eval, which permits interaction before and after evaluation. The break expression then corresponds to the argument to eval.

BREAK COMMANDS

GO	Releases the break and allows the computation to proceed. ⁸ <u>break1</u> evaluates <u>brkexp</u> , its first argument, prints the value of the break. <u>brkexp</u> is set up by the function that created the call to <u>break1</u> . For <u>break</u> or <u>trace</u> , <u>brkexp</u> is equivalent to the body of the definition of the broken function. For <u>breakin</u> , using BEFORE or AFTER, <u>brkexp</u> is NIL. For <u>breakin</u> AROUND, <u>brkexp</u> is the indicated expression. See <u>breakin</u> , page 15.16.
OK	Same as GO except the value of <u>brkexp</u> is not printed.
EVAL	Same as GO or OK except that the break is maintained after the evaluation. The user can then interrogate the value of the break which is bound on the variable <u>!value</u> , and continue with the break. Typing GO or OK following EVAL will not cause reevaluation but another EVAL will. EVAL is a useful command when the user is not sure whether or not the break will produce the correct value and wishes to be able to do something about it if it is wrong.
RETURN form or RETURN fn[args]	The value of the indicated computation is returned as the value of the break. For example, one might use the EVAL command and follow this with RETURN (REVERSE !VALUE).
↑	Calls <u>error!</u> and aborts the break. i.e., makes it "go away" without returning a value. This is a useful way to unwind to a higher level break. All other errors, including those encountered while executing the GO, OK, EVAL, and RETURN commands, maintain the break.
!EVAL	function is first unbroken, then the break expression is evaluated, and then the function is rebroken. Very useful for dealing with recursive functions.
!OK	Function is first unbroken, evaluated, rebroken, and then exited, i.e., !OK is equivalent to !EVAL followed by OK.

⁸ In order to facilitate debugging of programs that perform input operations, the carriage return that is typed to complete the GO, OK, EVAL, etc. commands is discarded, i.e., read by break1, so that it will not be part of the input stream.

!GO

Function is first unbroken, evaluated, rebroken, and exited with value typed, i.e., !EVAL followed by GO.

UB

unbreaks brkfn, e.g.,

```
(FOO BROKEN)
:UB
FOO
:
```

and FOO is now unbroken

@

resets the variable lastpos, which establishes a context for the commands ?=, ARGS, BT, BTV, BTV*, and EDIT, and IN? described below. lastpos is the position of a function call on the push-down stack. It is initialized to the function just before the call to break1, i.e., stknth[-1;BREAK1]⁹

@ treats the rest of the teletype line as its argument(s). It first resets lastpos to stknth[-1;BREAK1] and then for each atom on the line, @ searches backward, for a call to that atom. The following atoms are treated specially:

@ do not reset lastpos to stknth[-1;BREAK1] but leave it as it was, and continue searching from that point.

numbers if negative, move lastpos back that number of calls, if positive, forward, i.e., reset lastpos to stknth[n;lastpos]

/ the next atom is a number and can be used to specify more than one call e.g.,

@ FOO / 3 is equivalent to

@ FOO FOO FOO

= resets lastpos to the *value* of the next expression, e.g., if the value of FOO is a stack pointer, @ = FOO FIE will search for FIE in the environment specified by (the value of) FOO.

+ ⁹ When control passes from break1, e.g. as a result of an EVAL, OK, GO, REVERT, ↑ command, or via a RETFROM or RETEVAL typed in by the user, (RELSTK LASTPOS) is executed to release this stack pointer.

Example:

If the push-down stack looks like

BREAK1	(13)
FOO	(12)
SETQ	(11)
COND	(10)
PROG	(9)
FIE	(8)
COND	(7)
FIE	(6)
COND	(5)
FIE	(4)
COND	(3)
PROG	(2)
FUM	(1)

then @ FIE COND will set lastpos to the position corresponding to (7); @@ COND will then set lastpos to (5); and @ FIE / 3 -1 to (3).

If @ cannot successfully complete a search, it searches the stack again from that point looking for a call to a function whose name is close to that of fn, in the sense of the spelling corrector (Section 17), where fn is the name of the function for which it was searching. If the search is still unsuccessful, @ types (fn NOT FOUND), and then aborts.

When @ finishes, it types the name of the function at lastpos, i.e., stkname[lastpos]

@ can be used on brkcoms. In this case, the next command on brkcoms is treated the same as the rest of the teletype line.

?=

This is a multi-purpose command.¹⁰ Its most common use is to interrogate the value(s) of the arguments of the broken function, e.g., if FOO has three arguments (X Y Z), then typing ?= to a break on FOO, will produce:

```
:?=  
X =      value of X  
Y =      value of Y  
Z =      value of Z  
:
```

¹⁰ In fact, ?= is a universal mnemonic for displaying argument names and their corresponding values. In addition to being a break command, ?= is an edit macro which prints the argument names and values for the current expression (see Section 9), and a read-macro (actually ? is the read-macro character) which does the same for the current level list being read (see Sections 2 and 22).

?= operates on the rest of the teletype line as its arguments. If the line is empty, as in the above case, it operates on all of the arguments of the broken function. If the user types **?= X (CAR Y)**, he will see the value¹¹ of **X**, and the value of **(CAR Y)**. The difference between using **?=** and typing **X** and **(CAR Y)** directly to break1 is that **?=** evaluates its inputs as of lastpos, i.e., it uses stkeval. This provides a way of examining variables or performing computations *as of a particular point on the stack*. For example, **@ FOO / 2** followed by **?= X** will allow the user to examine the value of **X** in the previous call to **FOO**, etc.

?= also recognizes numbers as referring to the correspondingly numbered argument, i.e., it uses stkarg in this case. Thus

```
:@ FIE
FIE
:= 2
```

will print the name and value of the second argument of **FIE**.

?= can also be used on brkcoms, in which case the next command on brkcoms is treated as the rest of the teletype line. For example, if brkcoms is **(EVAL ?= (X Y) GO)**, brkexp will be evaluated, the values of **X** and **Y** printed, and then the function exited with its value being printed.

+ **PB**¹²

(print bindings) like **?=** except ascends the stack starting from lastpos, and, for each frame in which the given variable is bound, prints the frame name and value of the variable (with printlevel reset to **(2 . 3)**), e.g.

```
:PB FOO
@ FN1/*PROG*LAM: 3
@ FN2: 10
@ TOP: NOBIND
```

+ **BT**¹³

Prints a backtrace of *function names only* starting at lastpos. (See discussion of **@** above) The several nested calls in system packages such as break, edit, and the top level executive appear as the single entries ****BREAK****, ****EDITOR****, and ****TOP**** respectively.

+ ¹¹ The value of each variable is printed with the function showprint (Section 14), so that if sysprettyflg=T, the value will be prettyprinted.

+ ¹² **PB** is also a programmer's assistant command (Section 22), so that it can also be typed in when talking to the programmer's assistant, but not in a break. **PB** is implemented via the function printbindings.

+ ¹³ The value of breakdelimiter is what is printed to delimit the output of **?=** and backtrace commands. Initially "Q" but can be reset, e.g. to ",", for more linear output.

BT	Prints a backtrace of function names <i>with</i> variables ¹⁴ beginning at <u>lastpos</u> .	
BT+	Same as BT except also prints localvars and arguments to subrs. (see Section 12).	+ +
BT*	Same as BT except prints arguments to subrs, localvars, and temporaries of the interpreter, i.e. eval blips (see Section 12).	+ +
BT!	Same as BT except prints <i>everything</i> on stack (see Section 12).	

BT, BTV, BTV+, BTV*, and BTV! all permit an optional functional argument which is a predicate that chooses functions to be *skipped* on the backtrace, e.g., BT SUBRP will skip all SUBRs, BTV (LAMBDA (X) (NOT (MEMB X FOOFNS))) will skip all but those functions on FOOFNS. If used as a brkcom the functional argument is no longer optional, i.e., the next brkcom must either be the functional argument, or NIL if no functional argument is to be applied.

For BT, BTV, BTV+, BTV*, and BTV!, if control-P is used to change a printlevel during the backtrace, the printlevel will be restored after the backtrace is completed.

ARGS	Prints the names of the variables bound at <u>lastpos</u> , i.e., variables[lastpos] (Section 12). For most cases, these are the arguments to the function entered at that position, i.e., arglist[stkname[lastpos]].
------	---

The following two commands are for use only with unbound atoms or undefined function breaks (see Section 16).

= form, = fn[args] *only* for the break following an unbound atom error. Sets the atom to the value of the form, or function and arguments, exits from the break returning that value, and continues the computation, e.g.,

```
U.B.A.  
(FOO BROKEN)  
:= (COPY FILE)
```

sets FOO and goes on.

¹⁴ The value of each variable is printed with the function showprint (Section 14), so that if sysprettyflg=T, the value will be prettyprinted.

-> expr

for use either with unbound atom error, or undefined function error. Replaces the expression containing the error with expr¹⁵ (not the value of expr) e.g.,

```
U.D.F.  
(FOO1 BROKEN)  
:-> FOO
```

changes the FOO1 to FOO and continues the computation.

expr need not be atomic, e.g.,

```
U.B.A.  
(FOO BROKEN)  
:-> (QUOTE FOO)
```

For U.D.F. breaks, the user can specify a function *and* initial arguments, e.g.,

```
U.D.F.  
(MEMBERX BROKEN)  
:-> MEMBER X
```

Note that in the case of a U.D.F. error occurring immediately following a call to apply, e.g., (APPLY X Y) where the value of x is FOO and FOO is undefined, or a U.B.A. error immediately following a call to eval, e.g., (EVAL X), where the value of x is FOO and FOO is unbound, there is no expression containing the offending atom. In this case, -> cannot operate, so ? is printed and no action taken.

EDIT

designed for use in conjunction with breaks caused by errors. Facilitates editing the expression causing the break:

```
NON-NUMERIC ARG  
NIL  
(IPLUS BROKEN)  
:EDIT  
IN FOO...  
(IPLUS X Z)  
EDIT  
*(3 Y)  
*OK  
FOO  
:
```

and user can continue by typing OK, EVAL, etc.

¹⁵ -> does not change just brkexp; it changes the function or expression containing the erroneous form. In other words, the user does not have to perform any additional editing.

This command is very simple conceptually, but complicated in its implementation by all of the exceptional cases involving interactions with compiled functions, breaks on user functions, error breaks, breaks within breaks, et al. Therefore, we shall give the following simplified explanation which will account for 90% of the situations arising in actual usage. For those others, EDIT will print an appropriate failure message and return to the break.

EDIT begins by searching up the stack beginning at lastpos (set by @ command, initially position of the break) looking for a form, i.e., an internal call to eval. Then EDIT continues from that point looking for a call to an interpreted function, or to eval. It then calls the editor on either the EXPR or the argument to eval in such a way as to look for an expression eq to the form that it first found. It then prints the form, and permits interactive editing to begin. Note that the user can then type successive 0's to the editor to see the chain of superforms for this computation.

If the user exits from the edit with an OK, the break expression is reset, if possible, so that the user can continue with the computation by simply typing OK.¹⁶ However, in some situations, the break expression cannot be reset. For example, if a compiled function FOO incorrectly called putd and caused the error ARG NOT ATOM followed by a break on putd, EDIT might be able to find the form headed by FOO, and also find *that* form in some higher interpreted function. But after the user corrected the problem in the FOO-form, if any, he would still not have in any way informed EDIT what to do about the immediate problem, i.e., the incorrect call to putd. However, if FOO were *interpreted* EDIT would find the putd form itself, so that when the user corrected that form, EDIT could use the new corrected form to reset the break expression. The two cases are shown below:

<pre> ARG NOT ATOM (FUM) (PUTD BROKEN) :EDIT IN FIE... (FOO X) EDIT *(2 (CAR X)) *OK NOTE: BRKEXP NOT CHANGED FIE := U = (FUM) :(SETQ U (CAR U)) FUM :OK PUTD </pre>	<pre> ARG NOT ATOM (PUTD BROKEN) :EDIT IN FOO... (PUTD X) EDIT *(2 (CAR X)) *OK FOO :OK PUTD </pre>
--	---

IN? similar to EDIT, but just prints parent form, and superform, but does not call editor, e.g.,

```

ATTEMPT TO RPLAC NIL
T
(RPLACD BROKEN)
:IN?

```

¹⁶ Evaluating the new brkexp will involve reevaluating the form that causes the break, e.g., if (PUTD (QUOTE (FOO)) big-computation) were handled by EDIT, big-computation would be reevaluated.

FOO: (RPLACD X Z)

Although EDIT and IN? were designed for error breaks, they can also be useful for user breaks. For example, if upon reaching a break on his function FOO, the user determines that there is a problem in the *call* to FOO, he can edit the calling form and reset the break expression with one operation by using EDIT. The following two protocol's with and without the use of EDIT, illustrate this:

```
(FOO BROKEN)
:=
X = (A B C)
Y = D
:B

FOO
SETQ
COND      find which function
PROG      FOO is called from
FIE       (aborted with tE)
:EDITF(FIE)
EDIT
*F FOO P
(FOO V U)   edit it
*(SW 2 3)
*OK
FIE
:(SETQ Y X)  reset X and Y
(A B C)
:(SETQQ X D)
D
:=
X = D
Y = (A B C)  check them
:OK
FOO
```

```
(FOO BROKEN)
:=
X = (A B C)
Y = D
:EDIT
IN FIE...
(FOO V U)
EDIT
*(SW 2 3)
*OK
FIE      17
:OK
FOO
```

REVERT

goes back to position lastpos on stack and reenters the function called at that point with the arguments found on the stack.¹⁸ If the function is not already broken, REVERT first breaks it, and then unbreaks it after it is reentered.

REVERT is useful for restarting a computation in the situation where a bug is discovered at some point *below* where the problem actually occurred. REVERT essentially says "go back there and start over in a break."¹⁹

¹⁷ x and y have not been changed, but brkexp has. See previous footnote.

¹⁸ REVERT can also be given the position using the conventions described for @ on page 15.6, e.g., REVERT FOO -1 is equivalent to @ FOO -1 followed by REVERT.

¹⁹ REVERT will work correctly if the names or arguments to the function, or even its function type, have been changed.

ORIGINAL

for use in conjunction with breakmacros (described below). Form is (ORIGINAL . COMS). coms are executed without regard for breakmacros. Useful for redefining a break command in terms of itself.

+
+
+
+

BRKCOMS

The fourth argument to break1 is brkcoms, a list of break commands that break1 interprets and executes as though they were teletype input. One can think of brkcoms as another input file which always has priority over the teletype. Whenever brkcoms=NIL, break1 reads its next command from the teletype. Whenever brkcoms is not NIL, break1 takes as its next command car[brkcoms] and sets brkcoms to cdr[brkcoms].²⁰ For example, suppose the user wished to see the value of the variable x after a function was evaluated. He would set up a break with brkcoms=(EVAL (PRINT X) OK), which would have the desired effect. The function trace uses brkcoms: it sets up a break with two commands; the first one prints the arguments of the function, or whatever the user specifies, and the second is the command GO, which causes the function to be evaluated and its value printed.

If brkcoms is not NIL, the value of a break command is not printed. If you desire to see a value, you must print it yourself, as in the above example with the command (PRINT X).

Note: whenever an error occurs, brkcoms is set to NIL, and a full interactive break occurs.

BRKFILE

The break package has a facility for redirecting ouput to a file. The variable brkfile should be set to the name of the file, and the file must be opened. All output resulting from brkcoms will be output to brkfile, e.g., output due to TRACE. Output due to user typein is not affected, and will always go to the terminal. brkfile is initially T.

BREAKMACROS

Whenever an atomic command is given break1 it first searches the list breakmacros for the command. The form of breakmacros is (... (macro command₁ command₂ ... command_n) ...). If the command is defined as a macro, break1 simply appends its definition, which is a sequence of break commands, to the front of brkcoms, and goes on. If the command is not contained in breakmacros, break1 then checks to see if it is one of the built in commands, and finally, treats it as a function or variable as before.²¹

*
*
*
*
*
*

²⁰ Normally, when a user breaks or traces a function, the value of brkcoms for the corresponding call to break1 will be defaulted to NIL. However, it is possible to specify a list of break commands, as described in the discussion of break and break1 below.

²¹ If the command is not the name of a defined function, bound variable, or lisp command, break1 will attempt spelling correction using breakcomslst as a spelling list. If spelling correction is unsuccessful, break1 will go ahead and call lisp anyway, since the atom may also be a misspelled history command.

Example: the command ARG\$ could be defined by including on breakmacros:
 (ARG\$ (PRINT (VARIABLES LASTPOS T)))

BREAKRESETFORMS

If the user is developing programs that change the way a user and Interlisp normally interact, e.g., change or disable the interrupt or line-editing characters, turn off echoing, etc., debugging them by breaking or tracing may be difficult, because Interlisp might be in a "funny" state at the time of the break. breakresetforms is designed to solve this problem. The user puts on breakresetforms expressions suitable for use in conjunction with resetform or resetsave (section 5).²² When a break occurs, break1 evaluates each expression on breakresetforms before any interaction with the terminal, and saves the values. When the break expression is evaluated via an EVAL, OK, or GO, break1 first restores the state of the system with respect to the various expressions on breakresetforms. When (if) control returns to break1, the expressions on breakresetforms are again evaluated, and their values saved.²³ When the break is exited via an OK, GO, RETURN, or ↑ command, or via a RETFROM or RETEVAL typed in by the user,²⁴ break1 again restores1 state. Thus the net effect is to make the break invisible with respect to the user's programs, but nevertheless allow the user to interact in the break in the normal fashion.²⁵

15.3 BREAK FUNCTIONS

break1[brkexp;brkwhen;brkfn;brkcoms;brktype]

is an lambda. brkwhen determines whether a break is to occur. If its value is NIL, brkexp is evaluated and returned as the value of break1. Otherwise a break occurs and an identifying message is printed using brkfn. Commands are then taken from brkcoms or the teletype and interpreted. The commands, GO, !GO, OK, !OK, RETURN and ↑, are the only ways to leave break1. The command EVAL causes brkexp to be evaluated, and saves the value on the variable !value. Other commands can be defined for break1 via breakmacros. brktype is NIL for user breaks, INTERRUPT for control-H breaks, and ERRORX for error breaks.

For error breaks, the input buffer is cleared and saved. (For control-H breaks, the input buffer was cleared at the time the control-H was typed, see Section 16.) In both cases, if the break returns a value, i.e., is not aborted via ↑ or control-D, the input buffer will be restored (see Section 14).

* 22 e.g., (ECHOMODE T), (CONTROL), etc., i.e., the value of each form is its "previous state," so that the effect of evaluating the form can be undone by applying car of the form to the value. setq expressions can also be included on breakresetforms for saving and restoring system parameters, e.g. (SETQ LISPXHISTORY NIL), (SETQ DWIMFLG NIL), etc. These are handled specially by break1 in that the current value of the variable is saved before the setq is executed, and upon restoration, the variable is set back to this value.

+ 23 The state of the system is also restored if the user types control-D.

+ 24 All user type-in is scanned in order to make the operations undoable as described in section 22. At this point, RETFROMs and RETEVALs are also noticed. However, if the user types in an expression which calls a function that then does a RETFROM, this RETFROM will not be noticed, and the effects of breakresetforms will not be reversed.

+ 25 As mentioned earlier, break1 detects "Break within a break" situations, and avoids infinite loops. If the loop occurs because of an error, break1 simply rebinds breakresetforms to NIL, and calls help. This situation most frequently occurs when there is a bug in a function called by breakresetforms.

`break0[fn;when;coms;:-]`

sets up a break on the function fn²⁶ by redefining fn as a call to break1 with brkexp an equivalent definition of fn, and when, fn, and coms, as brkwhen, brkfn, brkcoms. Puts property BROKEN on property list of fn with value a gensym defined with the original definition. Puts property BRKINFO on property list of fn with value (BREAK0 when coms) (For use in conjunction with rebreak). Adds fn to the front of the list brokenfns. Value is fn.

If fn is not defined, returns (fn NOT DEFINED). +

If fn is non-atomic and of the form (fn1 IN fn2), break0 first calls a function which changes the name of fn1 wherever it appears inside of fn2 to that of a new function, fn1-IN-fn2, which it initially defines as fn1. Then break0 proceeds to break on fn1-IN-fn2 exactly as described above. This procedure is useful for breaking on a function that is called from many places, but where one is only interested in the call from a specific function, e.g., (RPLACA IN FOO), (PRINT IN FIE), etc. It is similar to breakin described below, but can be performed even when FN2 is compiled or blockcompiled, whereas breakin only works on interpreted functions.

If fn1 is not found in fn2, break0 returns the value (fn1 NOT FOUND IN fn2).

If fn1 is found in fn2, in addition to breaking fn1-IN-fn2 and adding fn1-IN-fn2 to the list brokenfns, break0 adds fn1 to the property value for the property NAMESCHANGED on the property list of fn2 and adds the property ALIAS with the value (fn2 . fn1) to the property list of fn1-IN-fn2. This will enable unbreak to recognize what changes have been made and restore the function fn2 to its original state.

If fn is nonatomic and not of the above form, break0 is called for each member of fn using the same values for when, coms, and file specified in this call to break0. This distributivity permits the user to specify complicated break conditions on several functions without excessive retyping, e.g.,

```
break0[(FOO1 ((PRINT PRIN1) IN (FOO2 FOO3)));
        (NEQ X T);(EVAL ?= (Y Z) OK)]
```

will break on FOO1, PRINT-IN-FOO2, PRINT-IN-FOO3, PRIN1-IN-FOO2 and PRIN1-IN-FOO3.

If fn is non-atomic, the value of break0 is a list of the individual values.

26

As mentioned earlier, it is also possible to break on clisp operators that appear as car of a form, e.g IF, FETCH, REPLACE, FOR, DO, etc. +

break[x]

is a nospread nlambda. For each atomic argument, it performs `break0[atom;T]`. For each list, it performs `apply[BREAK0;list]`. For example, `break[F001 (F002 (GREATERP N 5) (EVAL))]` is equivalent to `break0[F001,T]` and `break0[F002; (GREATERP N 5); (EVAL)]`

trace[x]

is a nospread nlambda. For each atomic argument, it performs `break0[atom;T; (TRACE ?= NIL GO)]`²⁷ For each list argument, car is the function to be traced, and cdr the forms the user wishes to see, i.e., trace performs:

```
break0[car[list];T;list[TRACE;? = ; cdr[list],GO]]
```

For example, `TRACE(F001 (F002 Y))` will cause both F001 and F002 to be traced. All the arguments of F001 will be printed; only the value of Y will be printed for F002. In the special case that the user wants to see *only* the value, he can perform `TRACE((fn))`. This sets up a break with commands `(TRACE ? = (NIL) GO)`.

Note: the user can always call break0 himself to obtain combination of options of break1 not directly available with break and trace. These two functions merely provide convenient ways of calling break0, and will serve for most uses.

BREAKIN

Breakin enables the user to insert a break, i.e., a call to break1, at a specified location in an interpreted function. For example, if foo calls fie, inserting a break in foo before the call to fie is similar to breaking fie. However, breakin can be used to insert breaks before or after prog labels, particular SETQ expressions, or even the evaluation of a variable. This is because breakin operates by calling the editor and actually inserting a call to break1 at a specified point *inside* of the function.

The user specifies where the break is to be inserted by a sequence of editor commands. These commands are preceded by BEFORE, AFTER, or AROUND, which breakin uses to determine what to do once the editor has found the specified point, i.e., put the call to break1 BEFORE that point, AFTER that point, or AROUND that point. For example, (BEFORE COND) will insert a break before the first occurrence of cond, (AFTER COND 2 1) will insert a break after the predicate in the first cond clause, (AFTER BF (SETQ X &)) after the *last* place X is set. Note that (BEFORE TTY:) or (AFTER TTY:) permit the user to type in commands to the editor, locate the correct point, and verify it for himself using the P command if he desires, and exit from the editor with OK.²⁸ breakin then inserts the break BEFORE, AFTER, or AROUND that point.

²⁷ The flag TRACE is checked for in break1 and causes the message "function :" to be printed instead of (function BROKEN).

²⁸ A STOP command typed to TTY: produces the same effect as an unsuccessful edit command in the original specification, e.g., (BEFORE CONDD). In both cases, the editor aborts, and breakin types (NOT FOUND).

For breakin BEFORE or AFTER, the break expression is NIL, since the value of the break is irrelevant. For breakin AROUND, the break expression will be the indicated form. In this case, the user can use the EVAL command to evaluate that form, and examine its value, before allowing the computation to proceed. For example, if the user inserted a break after a cond predicate, e.g., (AFTER (EQUAL X Y)), he would be powerless to alter the flow of computation if the predicate were not true, since the break would not be reached. However, by breaking (AROUND (EQUAL X Y)), he can evaluate the break expression, i.e., (EQUAL X Y), look at its value, and return something else if he wished.

The message typed for a breakin break, is ((fn) BROKEN), where fn is the name of the function inside of which the break was inserted. Any error, or typing control-E, will cause the full identifying message to be printed, e.g., (FOO BROKEN AFTER COND 2 1).

A special check is made to avoid inserting a break inside of an expression headed by any member of the list nobreaks, initialized to (GO QUOTE *), since this break would never be activated. For example, if (GO L) appears before the label L, breakin (AFTER L) will not insert the break inside of the GO expression, but skip this occurrence of L and go on to the next L, in this case the label L. Similarly, for BEFORE or AFTER breaks, breakin checks to make sure that the break is being inserted at a "safe" place. For example, if the user requests a break (AFTER X) in (PROG -- (SETQ X &) --), the break will actually be inserted AFTER (SETQ X &), and a message printed to this effect, e.g., BREAK INSERTED AFTER (SETQ X &).

breakin[fn;where;when;coms] breakin is an nlambda. when and coms are similar to when and coms for break0, except that if when is NIL, T is used. where specifies where in the definition of fn the call to break1 is to be inserted. (See earlier discussion).

If fn is a compiled function, breakin returns (fn UNBREAKABLE) as its value.

If fn is interpreted, breakin types SEARCHING... while it calls the editor. If the location specified by where is not found, breakin types (NOT FOUND) and exits. If it is found, breakin adds the property BROKEN-IN with value T, and the property BRKINFO with value (where when coms) to the property list of fn, and adds fn to the front of the list brokenfns.

Multiple break points, can be inserted with a single call to breakin by using a list of the form ((BEFORE ...) ... (AROUND ...)) for where. It is also possible to call break or trace on a function which has been modified by breakin, and conversely to breakin a function which has been redefined by a call to break or trace.

unbreak[x]

unbreak is a nospread nlambda. It takes an indefinite number of functions modified by break, trace, or breakin and restores them to their original state by calling unbreak0. Value is list of values of unbreak0.

unbreak[] will unbreak all functions on brokenfns, in reverse order. It first sets brkinfolst to NIL.

unbreak[T] unbreaks just the first function on brokenfns, i.e., the most recently broken function.

unbreak0[fn;-]

restores fn to its original state. If fn was not broken, value is (NOT BROKEN) and no changes are made. If fn was modified by breakin, unbreakin is called to edit it back to its original state. If fn was created from (fn1 IN fn2), i.e., if it has a property ALIAS, the function in which fn appears is restored to its original state. All dummy functions that were created by the break are eliminated. Adds property value of BRKINFO to (front of) brkinfolst.

Note: unbreak0[(fn1 IN fn2)] is allowed: unbreak0 will operate on fn1-IN-fn2 instead.

unbreakin[fn]

performs the appropriate editing operations to eliminate all changes made by breakin. fn may be either the name or definition of a function. Value is fn. Unbreakin is automatically called by unbreak if fn has property BROKEN-IN with value T on its property list.

rebreak[x]

is an nlambda, nospread function for rebreaking functions that were previously broken without having to respecify the break information. For each function on x, rebreak searches brkinfolst for break(s) and performs the corresponding operation. Value is a list of values corresponding to calls to break0 or breakin. If no information is found for a particular function, value is (fn - NO BREAK INFORMATION SAVED).

rebreak[] rebreaks everything on brkinfolst, i.e., rebreak[] is the inverse of unbreak[].

rebreak[T] rebreaks just the first break on brkinfolst, i.e., the function most recently unbroken.

+ breakread[type]
+
+
+
+
+
+
+
+

useful for writing breakmacros. If brkcoms is non-NIL (i.e. when the command in which the call to breakread appears is part of a list of break commands, rather than having been typed in) returns the next break command from the list, which it also sets to its cdr. Otherwise, if type=LINE, returns the rest of the commands on the line as a list. Otherwise, returns the next command on the line. For example, the BT command is defined as (BAKTRACE LASTPOS NIL (BREAKREAD) 0 T). Thus, if the user types BT, the third argument to baktrace will be NIL. If the user types BT SUBRP, the third argument will be SUBRP.

changename[fn;from;to]

changes all occurrences of from to to in fn. fn may be compiled or blockcompiled. Value is fn if from was found, otherwise NIL. Does not perform any modifications of property lists. Note that

from and to do not have to be functions, e.g., they can be names of variables, or any other literals.

virginfn[fn;flg]

is the function that knows how to restore functions to their original state regardless of any amount of breaks, breakins, advising, compiling and saving exprs, etc. It is used by prettyprint, define, and the compiler. If flg=NIL, as for prettyprint, it does not modify the definition of fn in the process of producing a "clean" version of the definition, i.e., it works on a copy. If flg=T as for the compiler and define, it physically restores the function to its original state, and prints the changes it is making, e.g., FOO UNBROKEN, FOO UNADVISED, FOO NAMES RESTORED, etc. Value is the virgin function definition.

baktrace[ipos;epos;skipfn;flags] prints backtrace from ipos to epos. flags specifies the options of the backtrace, e.g., do/don't print arguments, do/don't print temporaries of the interpreter, etc., and is the same as for backtrace (Section 12)²⁹. baktrace collapses the sequence of several function calls corresponding to a call to a system package into a single "function" using baktracelst as described below, e.g., any call to the editor is printed as **EDITOR**, a break is printed as **BREAK**, etc. If skipfn is not NIL and skipfn[stkname[pos]] is T, pos is skipped (including all variables). baktrace is used by the BT, BTV, BTV+, BTV*, and BTV! commands, with flags=0, 1, 5, 7, and 47Q respectively.

baktracelst

used for telling baktrace (i.e. the BT, BTV, etc. commands) to abbreviate various sequences of function calls on the stack by a single key, e.g. **BREAK**, **EDITOR**, etc.

The operation of baktrace and format of baktracelst is described so that the user can add his own entries to baktracelst. Each entry on baktracelst is a list of the form (framename key . pattern) or (framename (key₁ . pattern₁) ... (key_n . pattern_n)), where a pattern is a list of elements that are either atoms, which match a single frame, or lists, which are interpreted as a list of alternative patterns, e.g. (PROGN **BREAK** EVAL ((ERRORSET BREAK1A BREAK1)) (BREAK1)))

baktrace operates by scanning up the stack and, at each point, comparing the current frame name, with the frame names on baktracelst, i.e. it does an assoc. If the frame name does appear, baktrace attempts to match the stack as of that point with (one of) the patterns. If the match is successful, baktrace prints the corresponding key, and continues with where the match left off. If the frame name does not appear, or the match fails, baktrace simply prints the frame name³⁰ and continues with the next higher frame.

²⁹ The call to backtrace specifies that the values of any variables, blips, temporaries, etc. be printed with the function showprint (Section 14), so that if sysprettyflg=T, the values will be prettyprinted.

³⁰ unless skipfn applied to the frame name is non-NIL as described in discussion of baktrace, in which case nothing is printed.

- + Matching is performed by comparing atoms in the pattern with the current frame name, and
- + matching lists as patterns, i.e. sequences of function calls, always working up the stack. For example, either of the sequence of function calls... BREAK1 BREAK1A ERRORSET EVAL PROGN... or... BREAK1 EVAL PROGN... would match with the sample entry given above, causing **BREAK** to be printed.
- + Special features:
 - + 1. The atom & can be used to match any frame.
 - + 2. The pattern "-" can be used to match nothing. - is useful for specifying an optional match, e.g. the example above could also have been written as (PROGN **BREAK** EVAL ((ERRORSET BREAK1A) -) BREAK1).
 - + 3. It is not necessary to provide in the pattern for matching dummy frames, i.e. frames for which dummyframep (see Section 12) is true, e.g. in Interlisp-10, *PROG*LAM, *ENV*, NOLINKDEF1, etc. When working on a match, the matcher automatically skips over these frames when they do not match.
 - + 4. If a match succeeds and the key is NIL, nothing is printed. For example, (*PROG*LAM NIL EVALA *ENV)³¹

+ ³¹ This sequence will occur following an error which then causes a break if some of the function's arguments are localvars.

Index for Section 15

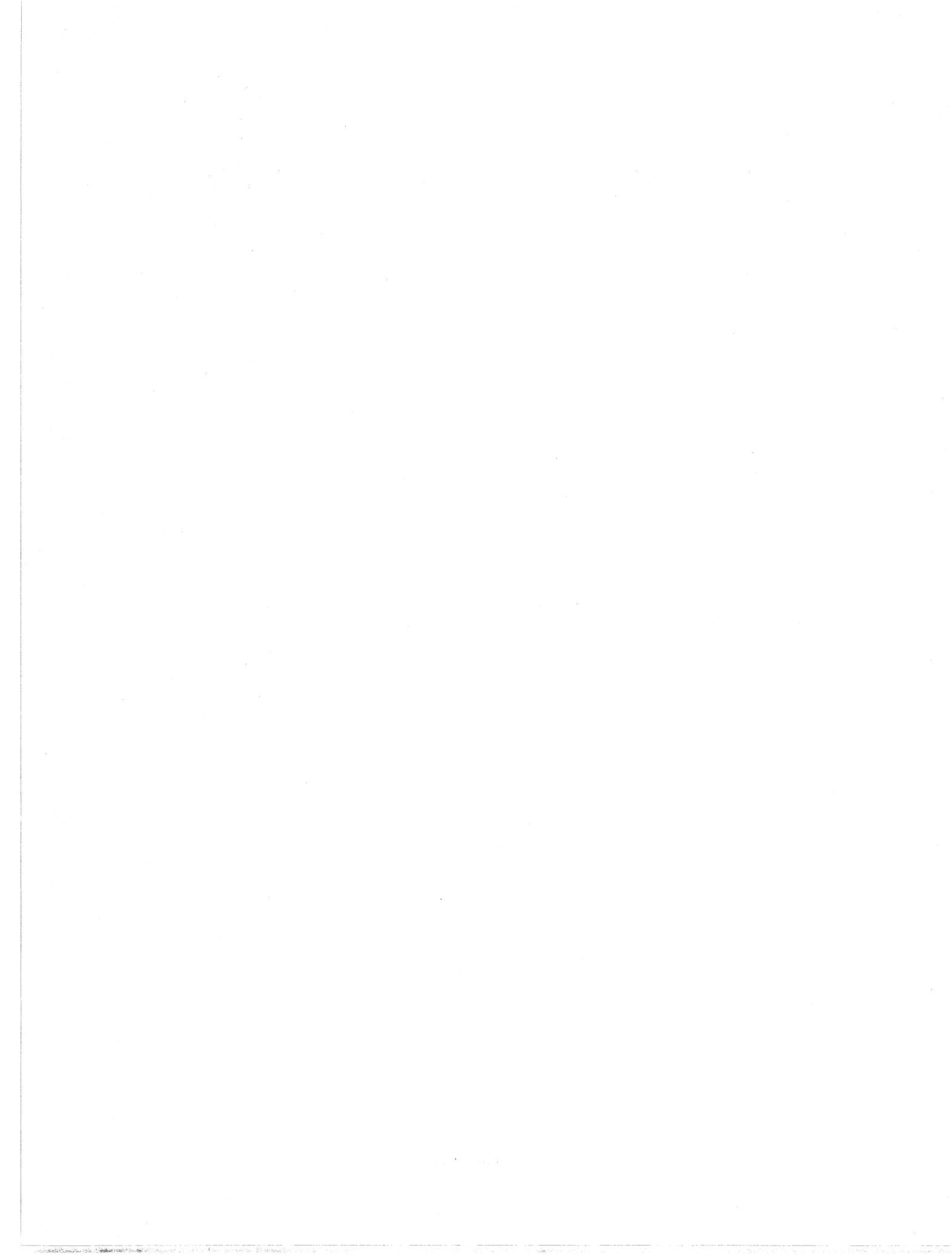
	Page Numbers
AFTER (as argument to breakin)	15.16,5,17
ALIAS (property name)	15.15,18
ARGLIST[FN]	15.9
ARGS (break command)	15.9,6
AROUND (as argument to breakin)	15.16,5,17
backtrace	15.8,9,19
BAKTRACE[IPOS;EPOS;SKIPFN;FLAGS;FILE]	15.19
BAKTRACELST (break variable/parameter)	15.19
BEFORE (as argument to breakin)	15.16,5,17
BREAK[X] NL*	15.16,1,5,17
break commands	15.5-13
break expression	15.4,11
BREAK INSERTED AFTER (typed by breakin)	15.17
break package	15.1-21
BREAK WITHIN A BREAK (printed by system)	15.4,14
BREAKCOMSLST (break variable/parameter)	15.13
BREAKDELIMITER (break variable/parameter)	15.8
BREAKIN[FN;WHERE;WHEN;BRKCOMS] NL	15.16-17,1-2,5,15,18
breaking clisp expressions	15.1
BREAKMACROS (break variable/parameter)	15.13-14
BREAKREAD[TYPE]	15.18
BREAKRESETFORMS (break variable/parameter)	15.14
BREAKO[FN;WHEN;COMS;BRKFN;TAIL]	15.15,16-18
BREAK1[BRKEXP;BRKWHEN;BRKFN;BRKCOMS;BRKTYPE] NL	15.3,14,1-2,4-13,15-17
BRKCOMS (break variable/parameter)	15.13,7-8,14-15
BRKEXP (break variable/parameter)	15.4,5,8,10-12,14-15
BRKFILE (break variable/parameter)	15.13
BRKFN (break variable/parameter)	15.14,6,15
BRKINFO (property name)	15.15,17-18
BRKINFOLST (break variable/parameter)	15.17-18
BRKTYPET (break variable/parameter)	15.14
BRKWHEN (break variable/parameter)	15.14-15
BROKEN (printed by system)	15.3
BROKEN (property name)	15.15
BROKENFNS (break variable/parameter)	15.15,17
BROKEN-IN (property name)	15.17,18
BT (break command)	15.8,6
BTV (break command)	15.9,6
BTV! (break command)	15.9
BTV* (break command)	15.9,6
BTV+ (break command)	15.9
CHANGENAME[FN;FROM;TO]	15.18
control-D	15.4,14
control-E	15.4,17
control-H	15.14
control-P	15.9
debugging	15.1
EDIT (break command)	15.10,6,11-12
editing compiled code	15.18
ERROR![] SUBR	15.5
EVAL (break command)	15.5,13-14,17
EVALQT[LISPID]	15.4
(fn1 IN fn2)	15.15,18
(fn1 NOT FOUND IN fn2)	15.15
fn1-IN-fn2	15.15,18
GENSYM[CHAR]	15.15

Page
Numbers

GO (break command)	15.5,4,6,13-14
input buffer	15.14
IN? (break command)	15.11,6,12
LASTPOS (break variable/parameter)	15.6,7-9,11
NAMES RESTORED (printed by system)	15.19
NAMESCHANGED (property name)	15.15
(NO BREAK INFORMATION SAVED)	15.18
NOBREAKS (break variable/parameter)	15.17
(NOT BROKEN)	15.18
(NOT FOUND) (typed by break)	15.7
(NOT FOUND) (typed by breakin)	15.16-17
NOTE: BRKEXP NOT CHANGED. (typed by break)	15.11
OK (break command)	15.5,4,11,13-14
ORIGINAL (break command)	15.13
PB (break command)	15.8
PRINTBINDINGS[AT;POS]	15.8
prompt character	15.3
REBREAK[X] NL*	15.18,15
RETEVAL[POS;FORM;FLG;INTERNALFLG]	15.4
RETFROM[POS;VAL;FLG] SUBR	15.4
RETURN (break command)	15.5,4,14
REVERT (break command)	15.12
SEARCHING... (typed by breakin)	15.17
SHOWPRINT[X;FILE;RDTBL]	15.8-9,19
spelling correction	15.13
spelling lists	15.13
STKARG[N;POS] SUBR	15.8
STKEVAL[POS;FORM;FLG;INTERNALFLG]	15.8
STOP (edit command)	15.16
SYSPRETTYFLG (system variable/parameter)	15.8-9,19
TRACE[X] NL*	15.16,1,5,13,17
TTY: (edit command)	15.16
UB (break command)	15.6
UNADVISED (printed by system)	15.19
UNBREAK[X] NL*	15.17,15,18
(UNBREAKABLE)	15.17
UNBREAKIN[FN]	15.18,18
UNBREAKO[FN;TAIL]	15.18
UNBROKEN (printed by system)	15.19
U.B.A. breaks	15.9
U.D.F. breaks	15.10
value of a break	15.4
VARIABLES[POS]	15.9
VIRGINFN[FN;FLG]	15.19
!EVAL (break command)	15.5
!GO (break command)	15.6,14
!OK (break command)	15.5,14
!VALUE (break variable/parameter)	15.5,14
BREAK (in backtrace)	15.8
EDITOR (in backtrace)	15.8
TOP (in backtrace)	15.8
-> (break command)	15.10
: (printed by system)	15.3
= (break command)	15.9
= (use with @ break command)	15.6
?= (break command)	15.6-8
@ (break command)	15.6,7,11

**Page
Numbers**

- ↑ (break command) 15.5,14
← (printed by system) 15.3



SECTION 16

ERROR HANDLING

16.1 UNBOUND ATOMS AND UNDEFINED FUNCTIONS

Whenever the interpreter encounters an atomic form with no binding on the push-down list, and whose value cell contains the atom NOBIND,¹ the interpreter calls the function faulteval. Similarly, faulteval is called when a list is encountered, car of which is not the name of a function or a function object.² The value returned by faulteval is used by the interpreter exactly as though it were the value of the form.

faulteval is defined to print either U.B.A., for unbound atom, or U.D.F., for undefined function, and then to call break1 giving it the offending form as brkexp.³ Once inside the break, the user can set the atom, define the function, return a specified value for the form using the RETURN command, etc., or abort the break using the ↑ command. If the break is exited with a value, the computation will proceed exactly as though no error had occurred.⁴

The decision over whether or not to induce a break depends on the depth of computation, and the amount of time invested in the computation. The actual algorithm is described in detail below in

1 As described in Section 3, in Interlisp-10, atoms are initialized (when they are created by the read program) *without* value cells. However, from the standpoint of the interpreter, an atom without a value cell is treated the same as an atom with a value cell containing NOBIND. The function boundp (Section 3) is available for testing whether or not an atom is bound in the sense described above, i.e. boundp[var]=T means that the variable either has a binding, or else has a value in its value cell other than NOBIND.

+
+
+
+
+
+

2 In Interlisp-10, because of the widespread use of CLISP (Section 23), before calling faulteval, if clisparray is non-NIL, the interpreter performs gethash[expression;CLISPARRAY], and if the result is non-NIL, treats it as the new expression to be evaluated, and goes on. This avoids going through the faulteval mechanism, and is done purely in the interests of efficiency.

+
+
+
+
+
+

3 If DWIM is enabled (and a break is going to occur), faulteval also prints the offending form (in the case of a U.B.A., the parent form) and the name of the function which contains the form. For example, if FOO contains (CONS X FIE) and FIE is unbound, faulteval prints: U.B.A. FIE [in FOO] in (CONS X FIE). Note that if DWIM is not enabled, the user can obtain this information after he is inside the break via the IN? command.

+

4 A similar procedure is followed whenever apply or apply* are called with an undefined function, i.e., one whose fntyp is NIL. In this case, faultapply is called giving it the function as its first argument and the list of arguments to the function as its second argument. The value returned by faultapply is used as the value of apply or apply*. faultapply is defined to print U.D.F. and then call break1 giving it (APPLY (QUOTE fn) QUOTE args)) as brkexp. Once inside the break, the user can define the function, return a specified value, etc. If the break is exited with a value, the computation will proceed exactly as though no error had occurred. faultapply is also called for undefined function calls from compiled code.

the section on breakcheck. Suffice it to say that the parameters affecting this decision have been adjusted empirically so that trivial type-in errors do not cause breaks, but deep errors do.

16.2 TERMINAL INITIATED BREAKS

CONTROL-H

Section 15 on the break package described how the user could cause a break when a specified function was entered. The user can also indicate his desire to go into a break at any time while a program is running by typing control-H.⁵ At the next point a function is about to be entered,⁶ the function interrupt is called instead. interrupt types INTERRUPTED BEFORE followed by the function name, constructs an appropriate break expression, and then calls break1. The user can then examine the state of the computation, and continue by typing OK, GO or EVAL, and/or retfrom back to some previous point, exactly as with a user break. Control-H breaks are thus always "safe". Note that control-H breaks are not affected by the depth or time of the computation. However, they *only* occur when a function is called, since it is only at this time that the system is in a "clean" enough state to allow the user to interact. Thus, if a compiled program is looping without calling any functions, or is in a I/O wait, control-H will not affect it. Control-B, however, will.

CONTROL-B

Control-B is a stronger interruption than control-H. It effectively generates an immediate error. This error is treated like any other error except that it always causes a break, regardless of the depth or time of the computation.⁷ Thus if the function FOO is looping internally, typing control-B will cause the computation to be stopped, the stack unwound to the point at which FOO was called, and then cause a break. Note that the internal variables of FOO are not available in this break, and similarly, FOO may have already produced some changes in the environment before the control-B

- * was typed. In other words, it may not be possible to simply continue the computation, depending on the nature of the function interrupted and when it was interrupted. Therefore whenever possible, it is better to use control-H instead of control-B.

CONTROL-E

If the user wishes to *abort* a computation, without causing a break, he should type control-E. Control-E does not go through the normal error machinery of scanning the stack, calling breakcheck, printing a message, etc. as described below, but simply types a carriage-return and unwinds.

⁵ As soon as control-H is typed, Interlisp clears and saves the input buffer, and then rings the bell, indicating that it is now safe to type ahead to the upcoming break. If the break returns a value, i.e., is not aborted via ↑ or control-D, the contents of the input buffer before the control-H was typed will be restored.

⁶ Control-H will *not* interrupt at linked function calls (see Section 18).

⁷ However, setting helpflag to NIL will suppress the break. See discussion of breakcheck below.

16.3 OTHER TYPES OF ERRORS

In addition to U.B.A. and U.D.F. errors, there are currently 28 other error types in Interlisp, e.g., P-STACK OVERFLOW, NON-NUMERIC ARG, FILE NOT OPEN, etc. A complete list is given later in this section. When an error occurs, the decision about whether or not to break is handled by breakcheck and is the same as with U.B.A. and U.D.F. errors. If a break is to occur, the exact action that follows depends on the type of error. For example, if a break is to occur following evaluation of (RPLACA NIL (ADD1 5)) (which causes an ATTEMPT TO RPLAC NIL error), the message printed will be (RPLACA BROKEN), brkexp will be (RPLACA U V W), U will be bound to NIL, V to 6, and W to NIL, and the stack will look like the user had broken on rplaca himself. Following a NON-NUMERIC ARG error, the system will type IN followed by the name of the most recently entered function, and then (BROKEN). The system will then effectively be in a break *inside* of this function. brkexp will be a call to ERROR so that if the user types OK or EVAL or GO, a ? will be printed and the break maintained. However, if the break is exited with a value via the RETURN command,⁸ the computation will proceed exactly as though no error had occurred.

16.4 BREAKCHECK - WHEN TO BREAK

The decision as to whether or not to induce a break when an error occurs is handled by the function breakcheck.⁹ The user can suppress all error breaks by setting the variable helpflag to NIL (initially set to T). If helpflag=T, the decision is affected by two factors: the length of time spent in the computation, and the depth of the computation at the time of the error.¹⁰ If the time is greater than helptime or the depth is greater than helpdepth, breakcheck returns T, meaning a break will occur. Finally, if helpflag=BREAK!, a break will always occur following an error.

Since a function is not actually entered until its arguments are evaluated,¹¹ the depth of a computation is defined to be the sum of the number of function calls plus the number of internal calls to eval. Thus if the user types in the expression [MAPC FOO (FUNCTION (LAMBDA (X) (COND ((NOT (MEMB X FIE)) (PRINT X)] for evaluation, and FIE is not bound, at the point of the U.B.A. FIE error, two functions, mapc and cond, have been entered, and there are three internal calls to eval corresponding to the evaluation of the forms (COND ((NOT (MEMB X FIE)) (PRINT X))), (NOT (MEMB X FIE)), and (MEMB X FIE)).¹² The depth is thus 5.

breakcheck begins by searching back up the parameter stack looking for an errorset.¹³ At the same

⁸ Presumably the value will be a number, or the error will occur again.

⁹ breakcheck is available to the user for advising or redefining. It is a function of two arguments, errorpos, the stack position at which the error occurred, and erxn, the error number.

¹⁰ Except that control-B errors always break.

¹¹ Unless of course the function does not have its arguments evaluated, i.e., is an FEXPR, FEXPR*, CFEXPR, CFEXPR*, FSUBR or FSUBR*.

¹² For complete discussion of the stack and the interpreter, see Section 12.

¹³ Errorsets are simply markers on the stack indicating how far back unwinding is to take place when an error occurs, i.e., they segment the stack into sections such as that if an error occurs in any section, control returns to the point at which the last errorset was entered, from which NIL is returned as the value of the errorset. See page 16.11.

time, it counts the number of internal calls to eval. As soon as (if) the number of calls to eval exceeds helpdepth, breakcheck can stop searching for errorset and return T, since the position of the errorset is only needed when a break is *not* going to occur. Otherwise, breakcheck continues searching until either an errorset is found¹⁴ or the top of the stack is reached. Breakcheck then completes the depth check by counting the number of function calls between the error and the last errorset, or the top of the stack. If the number of function calls plus the number of calls to eval (already counted) is greater than or equal to helpdepth, initially set to 9,¹⁵ breakcheck returns T. Otherwise, it records the position of the last errorset, and the value of errorset's second argument, which is used in deciding whether to print the error message, and returns NIL. Note that if a break is going to occur, the error message is printed regardless of the value of errorset's second argument.

breakcheck next measures the length of time spent in the computation by subtracting the value of the variable helpclock from the value of (CLOCK 2).¹⁶ If the difference is greater than helptime milliseconds, initially set to 1000, then a break will occur, i.e., breakcheck returns T, otherwise NIL. The variable helpclock is rebound to the current value of (CLOCK 2) for each computation typed in to lisp or to a break.

The time criterion for breaking can be suppressed by *setting* helptime to NIL (or a very big number), or by *binding* helpclock to NIL. Note that *setting* helpclock to NIL will not have any effect because helpclock is rebound by lisp and by break.

- * If breakcheck is NIL, i.e., a break is *not* going to occur, then if an errorset was found, NIL is returned (via retfrom) as the value of the errorset, after first printing the error message if the errorset's second argument was non-NIL.¹⁷ If there was no errorset, the message is printed, and control returns to evalqt. This procedure is followed for all types of errors.

Note that for all error breaks for which a break occurs, break1 will clear and save the input buffer. If the break returns a value, i.e., is not aborted via ↑ or control-D, the input buffer will be restored as described in Section 15.

16.5 ERROR TYPES

There are currently forty-plus error types in the Interlisp system.¹⁸ They are listed below by error number. The error is set internally by the code that detects the error before it calls the error handling functions. It is also the value returned by errorn if called subsequent to that type of error, and is used by errormess for printing the error message.

¹⁴ If the second argument to the errorset is INTERNAL, the errorset is ignored and searching continues. See discussion of errorset, page 16.11.

¹⁵ Arrived at empirically, takes into account the overhead due to lisp or break.

¹⁶ Whose value is number of milliseconds of compute time. See Section 21.

+ ¹⁷ If the value of nlsetqgag is NIL (initially T), the message will always be printed, regardless of the value of errorset's second argument.

¹⁸ Some of these errors are implementation dependent, i.e., appear in Interlisp-10 but may not appear in other Interlisp systems.

Most error types will print the offending expression following the message, e.g., NON-NUMERIC ARG NIL is very common. Error type 18 (control-B) always causes a break (unless helpflag is NIL). All other errors cause breaks if breakcheck returns T.

- | | | |
|--|--|---|
| 0 JSYS ERROR | In Interlisp-10 occurs following a trap in a JSYS (see Section 21) ¹⁹ . | + |
| 1 | no longer used. | |
| 2 STACK OVERFLOW | occurs when computation is too deep, either with respect to number of function calls, or number of variable bindings. ²⁰ Usually because of a non-terminating recursive computation, i.e., a bug. | |
| 3 ILLEGAL RETURN | call to <u>return</u> when not inside of an interpreted <u>prog</u> . | |
| 4 ARG NOT LIST | e.g., <u>rplaca</u> called on a non-list. | |
| 5 | no longer used. | |
| 6 ATTEMPT TO SET NIL via <u>set</u> or <u>setq</u> | | |
| 7 ATTEMPT TO RPLAC NIL | attempt either to <u>rplaca</u> or to <u>rplacd</u> NIL with something other than NIL. | |
| 8 UNDEFINED OR ILLEGAL GO | <u>go</u> when not inside of a <u>prog</u> , or <u>go</u> to nonexistent label. | |

¹⁹ As described in Section 21, TRAP AT LOCATION is printed, followed by the JSYS diagnostic, and control returns to the operating system executive. The user can then safely CONTINUE, and the Interlisp error, JSYS ERROR is then generated. A TRAP AT LOCATION can also occur if an illegal instruction is executed. In this case, the operating system also prints ILLEGAL INSTRUCTION. This can happen for example if the user is programming directly in ASSEMBLE code, or if his system somehow got smashed. In the latter case, it is quite possible that random programs or data structures might have already been smashed. Unless he is sure he knows what the problem is, the user is best advised to abandon this system as soon as possible. (If the user does elect to CONTINUE, Interlisp will (try to) generate a JSYS ERROR and unwind. In some cases, however, the system may be so badly smashed that the error message won't even print.) Note that in some cases, e.g. illegal instruction trap while in the garbage collector, Interlisp will print out CAN'T CONTINUE, because traps under those conditions are fatal. The user *may* be able to reenter his system via the START command, and, if lucky, dump some data or functions before the system totally collapses.

²⁰ In Interlisp-10, the garbage collector uses the same stack as the rest of the system, so that if a garbage collection occurs when deep in a computation, the stack can overflow (particularly if there is a lot of list structure that is deep in the car direction). If this does happen, the garbage collector will flush the stack used by the computation in order that the garbage collection can complete. Afterwards, the error message STACK OVERFLOW IN GC - COMPUTATION LOST is printed, followed by a reset[], i.e., return to top level.

- 9 FILE WON'T OPEN From infile or outfile, Section 14.
- 10 NON-NUMERIC ARG a numeric function e.g., iplus, itimes, igreaterp, expected a number.
- 11 ATOM TOO LONG In Interlisp-10, \geq 126 characters.
- 12 ATOM HASH TABLE FULL no room for any more (new) atoms.²¹
- 13 FILE NOT OPEN from an I/O function, e.g., read, print, closef.
- 14 ARG NOT LITATOM e.g., setq, put, gettopval, etc., given a non-atomic arg.
- * 15 TOO MANY FILES OPEN
* \geq 30 excluding terminal.
- 16 END OF FILE from an input function, e.g., read, readc, ratom. Note: the file will then be closed.²²
- 17 ERROR call to error.
- 18 BREAK control-B was typed.
- 19 ILLEGAL STACK ARG a stack function expected a stack position and was given something else. This might occur if the arguments to a stack function are reversed. Also occurs if user specified a stack position with a function name, and that function was not found on the stack. See Section 12.
- 20 FAULT IN EVAL artifact of bootstrap. Never occurs after faulteval has been defined as described earlier.

21 In Interlisp-10, the atom hash table will automatically expand by a specified number of pages each time it fills up until an upper limit of 32K atoms is reached.

* 22 The entries on errortypest (described below) are processed before the file is closed, so that the user can intercept and process this error via an entry on errortypest, thereby preventing the file from being closed. It is also possible to use an errortypest entry to return a character as the value of the call to errorx, and the program will continue, e.g. returning "]" may be used to complete a read operation.

- 21 ARRAYS FULL system will first initiate a garbage collection of array space, and if no array space is reclaimed, will then generate this error. *
- 22 FILE SYSTEM RESOURCES EXCEEDED (Interlisp-10) includes no more disk space, disk quota exceeded, directory full, too many jfbs, job full. *
*
*
- 23 FILE NOT FOUND file name does not correspond to a file in the corresponding directory.²³ Can also occur if file name is ambiguous.
- 24 BAD SYSOUT FILE date does not agree with date of makesys, or file is not a sysout file at all (see section 14). +
+
- 25 UNUSUAL CDR ARG LIST a form ends in a non-list other than NIL, e.g., (CONS T . 3).
- 26 HASH TABLE FULL see hash link functions, Section 7.
- 27 ILLEGAL ARG Catch-all error. Currently used by putd, evala, arg, funarg, allocate, rplstring, etc.
- 28 ARG NOT ARRAY elt or seta given an argument that is not a pointer to the beginning of an array.
- 29 ILLEGAL OR IMPOSSIBLE BLOCK (Interlisp-10) from getblk or relblk. See Section 21.
- 30 STACK PTR HAS BEEN RELEASED a released stack pointer was supplied as a stack descriptor for a purpose other than as a stack pointer to be re-used (see Section 12). +
+
- 31 STORAGE FULL following a garbage collection, if a sufficient amount of words has not been collected, and there is no un-allocated space left in the system, this error is generated. *
*
*
-
- 23 Interlisp is initialized with an entry on errortype1st (described below) to call spellfile (Section 17) for error 23. spellfile will search alternate directories or perform spelling correction on the connected directory. If spellfile fails, then the user will see this error. +
+

32 ATTEMPT TO USE ITEM OF INCORRECT TYPE

Before a field of a user-data type is changed, the type of the item is first checked to be sure that it is of the expected type. If not, this error is generated. See Section 23.

33 ILLEGAL DATA TYPE NUMBER

The argument is not a valid user-data type number. See Section 23.

34 DATA TYPES FULL All available user-data types have been allocated. See Section 23.

+ 35 ATTEMPT TO BIND NIL OR T
+ in a prog or lambda expression.

36 TOO MANY USER INTERRUPT CHARACTERS

Attempt to enable a user interrupt character when all 9 user channels are currently enabled. See page 16.12.

37 READ-MACRO CONTEXT ERROR

Occurs when a read is executed from within a read-macro function and the next token is a) or a]. See Section 14.

38 ILLEGAL READTABLE The argument was expected to be a valid readable. See Section 14.

39 ILLEGAL TERMINAL TABLE

The argument was expected to be a valid terminal table. See Section 14.

40 SWAPBLOCK TOO BIG FOR BUFFER

(Interlisp-10) An attempt was made to swap in a function/array which is too large for the swapping buffer. See setsbsize, Section 3.

+ 41 PROTECTION VIOLATION

+ (Interlisp-10) attempt to open a file that user does not have access to. Also reference to unassigned device.

+ 42 BAD FILE NAME

+ illegal character in file specification, illegal syntax, e.g. in Interlisp-10, two ';'s etc.

43 USER BREAK

Error corresponding to "hard" user-interrupt character. See page 16.12.

In addition, many system functions, e.g., define, arglist, advise, log, expt, etc, also generate errors with appropriate messages by calling error (see page 16.10) which causes an error of type 17.

ERROR HANDLING BY ERROR TYPE

Occasionally the user may want to treat certain error types differently from others, e.g., always break, never break, or perhaps take some corrective action. This can be accomplished via errortypest. errortypest is a list of elements of the form (n expression), where n is one of the 28 error numbers. After breakcheck has been completed, but before any other action is taken, errortypest is searched for an element with the same error number as that causing the error. If one is found, and the evaluation of the corresponding expression produces a non-NIL value, the value is substituted for the offender, and the function causing the error is reentered.

For this application, the following three variables may be useful:

errormess car is the error number, cadr the "offender" e.g., (10 NIL) corresponds to NON-NUMERIC ARG NIL error.

errorpos position of the function in which the error occurred, e.g., stkname[errorpos] might be IPLUS, RPLACA, INFILE, etc.

breakchk value of breakcheck, i.e., T means a break will occur, NIL means one will not.

For example, putting

```
[10 (AND (NULL (CADR ERRORMESS))
          (SELECTQ (STKNAME ERRORPOS)
                    ((IPLUS ADD1 SUB1) 0)
                    (ITIMES 1))
          (PROGN (SETQ BREAKCHK T) NIL)]
```

on errortypest would specify that whenever a NON-NUMERIC ARG - NIL error occurred, and the function in question was IPLUS, ADD1, or SUB1, 0 should be used for the NIL. If the function was ITIMES, 1 should be used. Otherwise, always break. Note that the latter case is achieved not by the value returned, but by the effect of the evaluation, i.e., setting BREAKCHK to T. Similarly, (16 (SETQ BREAKCHK NIL)) would prevent END OF FILE errors from ever breaking.

printmsg if T, means print error message, if NIL, don't print error message, i.e., corresponds to second argument to errorset. The user can force or suppress the printing of error message for various errortypes by including on errortypest an expression which explicitly sets printmsg.

Note: If the error is going to be handled by a retfrom, retto, or a reteval in the errortypest entry, it probably is a good idea to first release the stack pointer errorpos, e.g. by performing (RELSTK ERRORPOS). + + +

- + The initial value of errortypest is ((23 (SPELLFILE (CADR ERRORMESS)))), which causes
- + spellfile (Section 17) to be called in case of a FILE NOT FOUND error. If spellfile is successful,
- + the operation will be reexecuted with the new (corrected) file name.

16.6 ERROR FUNCTIONS

errorx[erxm]

is the entry to the error routines. If erxm=NIL, errorn[] is used to determine the error-message. Otherwise, seterrorn[car[erxm];cdr[erxm]] is performed, "setting" the error type and argument. Thus following either errorx[(10 T)] or (PLUS T), errorn[] is (10 T). errorx calls breakcheck, and either induces a break or prints the message and unwinds to the last errorset. Note that errorx can be called by any program to intentionally induce an error of any type. However, for most applications, the function error will be more useful.

error[mess1;mess2;nobreak]

The message that is (will be) printed is mess1 (using prinl), followed by a space if mess1 is an atom, otherwise a carriage return. Then mess2 is printed, using prinl if mess2 is a string, otherwise print, e.g., error["NON-NUMERIC ARG";T] will print
NON-NUMERIC ARG

T

and error[FOO;"NOT A FUNCTION"] will print FOO NOT A FUNCTION. (If both mess1 and mess2 are NIL, the message is simply ERROR.)

If nobreak=T, error prints its message and then calls error!²⁴ Otherwise it calls errorx[(17 (mess1 . mess2))], i.e., generates an error of type 17, in which case the decision as to whether or not to break, and whether or not to print a message, is handled as per any other error.

help[mess1;mess2]

prints mess1 and mess2 a la error, and then calls break!. If both mess1 and mess2 are NIL, HELP! is used for the message. help is a convenient way to program a default condition, or to terminate some portion of a program which the computation is theoretically never supposed to reach.

- + shouldnt[]
- +

for those situations when a program detects a condition that should never occur. Calls help with the message "Shouldn't happen!".

error![]

²⁵Programmable control-E, i.e., immediately returns from last errorset or resets.

* ²⁴ unless the value of helpflag is BREAK!, in which case a break will always occur, as described earlier.

25 Pronounced "error-bang".

reset[]	programmable control-D, i.e., immediately returns to the top level.
errorn[]	returns information about the last error in the form (<u>n</u> <u>x</u>) where <u>n</u> is the error type number and <u>x</u> is the expression which was (would have been) printed out after the error message. Thus following (PLUS T), errorn[] is (10 T).
seterrorn[num;mess]	sets <u>errorn</u> : after seterrorn[num;mess] has been performed, errorn[] will return (num mess).
errormess[u]	prints message corresponding to an <u>errorn</u> that yielded <u>u</u> . For example, errormess[(10 T)] would print NON-NUMERIC ARG T
errorstring[n]	returns as a new string the message corresponding to an error of type <u>n</u> , e.g., errorstring[10] = "NON-NUMERIC ARG".
errorset[form;flg;-] ²⁶	performs eval[form]. Note that <u>errorset</u> is a lambda-type of function, and that its arguments are evaluated <i>before</i> it is entered, i.e., errorset[x] means eval is called with the value of <u>x</u> . In most cases, <u>ersetq</u> and <u>nlsetq</u> (described below) are more useful. If no error occurs in the evaluation of <u>form</u> , the value of <u>errorset</u> is a list containing one element, the value of eval[form]. If an error did occur, the value of <u>errorset</u> is NIL. The argument <u>flg</u> controls the printing of error messages if an error occurs. If <u>flg</u> =T, the error message is printed; if <u>flg</u> =NIL it is not, unless the value of <u>nlsetqgag</u> is NIL, (initially T). ²⁷ Note that if a break occurs due to an error below an <u>errorset</u> , the message is printed regardless of the value of <u>flg</u> . If <u>flg</u> =INTERNAL, the errorset is ignored for the purpose of deciding whether or not to break or print a message. However, the errorset is in effect for the purpose of flow of control, i.e., if an error occurs, this <u>errorset</u> returns NIL.
	If <u>flg</u> =NOBREAK, no break will occur, even if the time criterion for breaking is met. Note that <u>flg</u> =NOBREAK will <i>not</i> prevent a break from occurring if the error occurs more than <u>helpdepth</u> function calls below the errorset, since <u>breakcheck</u> will stop searching before

²⁶ errorset is a subr, so the names "form" and "flg" don't actually appear on the stack nor will they affect the evaluation.

²⁷ In other words, nlsetqgag provides a global override on errorsets, effectively changing all nlsetqs to ersetqs.

it reaches the errorset, as explained on page 16.3. To guarantee that no break occurs, the user would also either have to reset helpdepth or helpflag.

ersetq[ersetx] nlambda, performs errorset[ersetx;T], i.e., (ERSETQ (FOO)) is equivalent to (ERRORSET (QUOTE (FOO)) T).

nlsetq[nlsetq] nlambda, performs errorset[nlsetq;NIL].

16.7 INTERRUPT CHARACTERS

This section describes how the user can disable and/or redefine Interlisp interrupt characters, as well as defining his own interrupt characters. Interlisp is initialized with 8 interrupt channels which we shall call: HELP, PRINTLEVEL, STORAGE, RUBOUT, ERROR, RESET, OUTPUTBUFFER, and BREAK. To these are assigned respectively, control-H, control-P, control-S, delete/rubout, control-E, control-D, control-O, and control-B. Each of these channels independently can be disabled, or have a new interrupt²⁸ character assigned to it via the function interruptchar described below. In addition, the user can enable up to 9 new interrupt channels, and associate with each channel an interrupt character and an expression to be evaluated when that character is typed. User interrupts can be either "hard" or "soft". A "hard" interrupt is like control-E or control-D: it takes place as soon as it is typed.²⁹ A soft interrupt is like control-H; it does not occur until the next function call.³⁰ Soft interrupts can always be safely continued from. Hard interrupts rip the system out of the function currently being executed and unwind back to the last function call, i.e. part of the computation that was interrupted is lost and cannot be continued.

interruptchar[char;typ/form;hardflg]

char is either a character or a terminal interrupt code.³¹

If typ/form=NIL, char is disabled. If typ/form=T, the current state of char is returned without changing it.³²

²⁸ TENEX requires that interrupt characters be one of control-A, B,...,Z, space, esc(alt-mode), rubout(delete), or break.

²⁹ Hard interrupts are implemented by generating an error of type 43, and retrieving the corresponding form from the list userinterrupts once inside of errorx. Soft interrupts are implemented by calling interrupt with an appropriate third argument, and then obtaining the corresponding form from userinterrupts. In either case, if a character is enabled as a user interrupt, but for some reason it is not found on userinterrupts, an **UNDEFINED USER INTERRUPT** error will be generated.

+ ³⁰ As soon as a soft interrupt character is typed, Interlisp clears and saves the input buffers, and then rings the bell. After the interrupt form is evaluated, the input buffers are restored.

³¹ The terminal interrupt code for break is 0, for esc is 27, for rubout/delete is 28, and for space is 29. The terminal interrupt codes for the control characters can be obtained with chconl.

³² The current state is an expression which can be given back to interruptchar to restore that state. This option is used in connection with undoing and resetform.

If typ/form is a literal atom and the name of one of the 8 Interlisp interrupt channels given above: HELP, PRINTLEVEL,...,BREAK. interruptchar assigns char to that channel, (reenabling the channel if previously disabled). Otherwise, char is enabled as an interrupt character that when typed causes typ/form to be *immediately* set to T. If char was previously defined as an interrupt character, that interpretation is disabled.

If typ/form is a list, char is enabled as a user interrupt character, and typ/form is the form that is evaluated when char is typed. The interrupt will be hard if hardflg=T, otherwise soft. Any previous interpretations of char are disabled.

All calls to interruptchar are undoable. In addition, the value of interruptchar is an expression which when given back to interruptchar will restore things as they were before the call to interruptchar. Thus, interruptchar can be used in conjunction with resetform or resetlst (see Section 5).

Note: interruptchar[T] will restore all Interlisp channels to their original state, and disable all user interrupts.

interruptable[flg] if flg=NIL, turns interrupt off. If flg=T, turns interrupt on. +
 Value is previous setting. interruptable compiles open. +

Note: Any interrupt character typed while interrupts are off is treated the same as any other character, i.e. placed in the input buffer, and will not cause an interrupt when interrupts are turned back on. +
 +
 +
 +

interruptablep[] value is T, if interrupts are enabled, NIL if disabled. +

Index for Section 16

	Page Numbers
APPLY[FN;ARGS] SUBR	16.1
APPLY*[FN;ARG1;ARG2;...;ARGn] SUBR*	16.1
ARG NOT ARRAY (error message)	16.7
ARG NOT LIST (error message)	16.5
ARG NOT LITATOM (error message)	16.6
ARRAYS FULL (error message)	16.7
ATOM HASH TABLE FULL (error message)	16.6
ATOM TOO LONG (error message)	16.6
ATTEMPT TO BIND NIL OR T (error message)	16.8
ATTEMPT TO RPLAC NIL (error message)	16.5
ATTEMPT TO SET NIL (error message)	16.5
ATTEMPT TO USE ITEM OF INCORRECT TYPE (error message)	16.8
BAD FILE NAME (error message)	16.8
BAD SYSOUT FILE (error message)	16.7
bell (printed by system)	16.2
BREAK (error message)	16.6
BREAKCHECK[ERRORPOS;ERXN]	16.3,2,4-5,9-10
BREAK1[BRKEXP;BRKWHEN;BRKFN;BRKCOMS;BRKTYPE] NL	16.1-2,4,10
BRKEXP (break variable/parameter)	16.1,3
(BROKEN) (printed by system)	16.3
control-B	16.2,3,5-6
control-D	16.2,4,11
control-E	16.2,10
control-H	16.2
DATA TYPES FULL (error message)	16.8
DWIM	16.1
ELT[A;N] SUBR	16.7
END OF FILE (error message)	16.6
ERROR[MESS1;MESS2;NOBREAK]	16.10,4,6,9
error handling	16.1-12
error number	16.4
error types	16.4-9
ERROR (error message)	16.6
ERRORMESS[U]	16.11,4
ERRORN[] SUBR	16.11,4
ERRORSET[FORM;FLG] SUBR	16.11,3-4,10
ERRORSTRING[N] SUBR	16.11
ERRORTYPELST (system variable/parameter)	16.9
ERRORX[ERXM]	16.10
ERRORI[] SUBR	16.10,10
ERSETQ[ERSETX] NL	16.12,11
EVAL[X] SUBR	16.11
EVAL (break command)	16.2-3
FAULT IN EVAL (error message)	16.6
FAULTAPPLY[FAULTFN;FAULTARGS]	16.1
FAULTEVAL[FAULTX] NL*	16.1,6
FILE NOT FOUND (error message)	16.7
FILE NOT OPEN (error message)	16.6
FILE SYSTEM RESOURCES EXCEEDED (error message) ..	16.7
FILE WON'T OPEN (error message)	16.6
function objects	16.1
GETBLK[N] SUBR	16.7
GO (break command)	16.2-3
HASH TABLE FULL (error message)	16.7
HELP[MESS1;MESS2]	16.10

Page
Numbers

HELPCLK (system variable/parameter)	16.4
HELPDEPTH (system variable/parameter)	16.3-4
HELPFLAG (system variable/parameter)	16.3,2,5
HELPTIME (system variable/parameter)	16.3-4
HELP! (printed by system)	16.10
ILLEGAL ARG (error message)	16.7
ILLEGAL DATA TYPE NUMBER (error message)	16.8
ILLEGAL INSTRUCTION (error message)	16.5
ILLEGAL OR IMPOSSIBLE BLOCK (error message)	16.7
ILLEGAL READTABLE (error message)	16.8
ILLEGAL RETURN (error message)	16.5
ILLEGAL STACK ARG (error message)	16.6
ILLEGAL TERMINAL TABLE (error message)	16.8
IN (printed by system)	16.3
input buffer	16.2,4
interpreter	16.1
INTERRUPT[INTFN;INTARGS;INTTYPE]	16.2
interrupt characters	16.12
INTERRUPTABLE[FLG] SUBR	16.13
INTERRUPTABLEP[] SUBR	16.13
INTERRUPTCHAR[CHAR;TYP/FORM;HARDFLG]	16.12
INTERRUPTED BEFORE (printed by system)	16.2
IN? (break command)	16.1
JSYS ERROR (error message)	16.5
NLSETQ[NLSETX] NL	16.12,11
NLSETQGAG (system variable/parameter)	16.4,11
NOBIND	16.1
NON-NUMERIC ARG (error message)	16.6,3
OK (break command)	16.2-3
PRINTMSG (system variable/parameter)	16.9
PROTECTION VIOLATION (error message)	16.8
READ-MACRO CONTEXT ERROR (error message)	16.8
RELBLK[ADDRESS;N] SUBR	16.7
RESET[] SUBR	16.11
RETFROM[POS;VAL;FLG] SUBR	16.4
RETURN (break command)	16.1,3
SETA[A;N;V]	16.7
SETERRORN[NUM;MESS] SUBR	16.11
SETSBSIZE[N] SUBR	16.8
SHOULDNT[]	16.10
SHOULDN'T HAPPEN (error message)	16.10
SPELLFILE[FILE;NOPRINTFLG;NSFLG]	16.7,10
STACK OVERFLOW IN GC - COMPUTATION LOST (error message)	16.5
STACK OVERFLOW (error message)	16.5
STACK PTR HAS BEEN RELEASED (error message)	16.7
STORAGE FULL (error message)	16.7
SWAPBLOCK TOO BIG FOR BUFFER (error message)	16.8
terminal initiated breaks	16.2
TOO MANY FILES OPEN (error message)	16.6
TOO MANY USER INTERRUPT CHARACTERS (error message)	16.8
TRAP AT LOCATION (error message)	16.5
unbound atom	16.1
undefined function	16.1
UNDEFINED OR ILLEGAL CO (error message)	16.5
UNDEFINED USER INTERRUPT (error message)	16.12
UNUSUAL CDR ARG LIST (error message)	16.7

**Page
Numbers**

USER BREAK (error message)	16.8
user interrupt characters	16.12
U.B.A. (error message)	16.1,3
U.D.F. (error message)	16.1,3
value cell	16.1
value of a break	16.1
? (printed by system)	16.3
↑ (break command)	16.1-2,4

SECTION 17

AUTOMATIC ERROR CORRECTION - THE DWIM FACILITY¹

17.1 INTRODUCTION

A surprisingly large percentage of the errors made by Interlisp users are of the type that could be corrected by another LISP programmer without any information about the purpose of the program or expression in question, e.g., misspellings, certain kinds of parentheses errors, etc. To correct these types of errors we have implemented in Interlisp a DWIM facility, short for Do-What-I-Mean. DWIM is called automatically whenever an error² occurs in the evaluation of an Interlisp expression. DWIM then proceeds to try to correct the mistake using the current context of computation plus information about what the user had previously been doing, (and what mistakes he had been making) as guides to the remedy of the error. If DWIM is able to make the correction, the computation continues as though no error had occurred. Otherwise, the procedure is the same as though DWIM had not intervened: a break occurs, or an unwind to the last errorset, as described in Section 16. The following protocol illustrates the operation of DWIM.

EXAMPLE

The user defines a function fact of one argument, n. The value of fact[n] is to be n factorial.

```
←DEFINEQ((FACT (LAMBDA (N) (COND
  ((ZEROP N9 1) ((T (ITIMS N (FACCT 8SUB1 N]
    (FACT)
  ←
```

Note that the definition of fact contains several mistakes: itimes and fact have been misspelled; the 9 in N9 was intended to be a right parenthesis, but the shift key was not depressed; similarly, the 8 in 8SUB1 was intended to be a left parenthesis; and finally, there is an extra left parenthesis in front of the T that begins the final clause in the conditional.

¹ DWIM was designed and implemented by W. Teitelman. It is discussed in [Tei2].

² Currently, DWIM only operates on unbound atoms and undefined function errors.

```

←PRETTYPRNT((FACT)
=PRETTYPRINT [1]
=FACT [2]
← [3]

(FACT
[LAMBDA (N)
(COND
((ZEROP N9 1)
((T (ITIMS N (FACT 8SUB1 N))
(FACT)
←

```

After defining fact, the user wishes to look at its definition using PRETTYPRINT, which he unfortunately misspells.[1] Since there is no function PRETTYPRINT in the system, a U.D.F. error occurs, and DWIM is called. DWIM invokes its spelling corrector, which searches a list of functions frequently used (by *this* user) for the best possible match. Finding one that is extremely close, DWIM proceeds on the assumption that PRETTYPRNT meant PRETTYPRINT, notifies the user of this, [2] and calls prettyprint.

At this point, PRETTYPRINT would normally print (FACT NOT PRINTABLE) and exit, since fact has no definition. Note that this is *not* an Interlisp error condition, so that DWIM would not be called as described above. However, it is obviously not what the user *meant*.

This sort of mistake is corrected by having prettyprint itself explicitly invoke the spelling corrector portion of DWIM whenever given a function with no expr definition. Thus with the aid of DWIM, prettyprint is able to determine that the user wants to see the definition of the function fact,[3] and proceeds accordingly.

```

←FACT(3) [4]
N9 [IN FACT] -> N ) ? YES
[IN FACT] (COND -- ((T --))) ->
(COND -- (T --))
ITIMS [IN FACT] -> ITIMES [5]
FACT [IN FACT] -> FACT
8SUB1 [IN FACT] -> ( SUB1 ? YES
6
←PP FACT [6]

(FACT
[LAMBDA (N)
(COND
((ZEROP N)
1)
(T (ITIMS N (FACT (SUB1 N)))
FACT
←

```

The user now calls his function fact.[4] During its execution, five errors occur, and DWIM is called five times.[5] At each point, the error is corrected, a message printed describing the action taken, and the computation allowed to continue as if no error had occurred. Following the last correction, 6 is printed, the value of fact(3). Finally, the user prettyprints the new, now correct, definition of fact.[6]

In this particular example, the user was shown operating in TRUSTING mode, which gives DWIM

carte blanche for most corrections. The user can also operate in CAUTIOUS mode, in which case DWIM will inform him of intended corrections before they are made, and allow the user to approve or disapprove of them. For most corrections, if the user does not respond in a specified interval of time, DWIM automatically proceeds with the correction, so that the user need intervene only when he does not approve. Sample output is given below. Note that the user responded to the first, second, and fifth questions; DWIM responded for him on the third and fourth.

```

←FACT(3)
N9 [IN FACT] -> N ) ? YES [1]
U.D.F. T [IN FACT] FIX? YES [2]
[IN FACT] (COND -- ((T --))) ->
  (COND -- (T --))
ITIMS [IN FACT] -> ITIMES ? ...YES [3]
FACCT [IN FACT] -> FACT ? ...YES [4]
8SUB1 [IN FACT] -> ( SUB1 ? NO [5]
U.B.A.
(8SUB1 BROKEN)
:

```

We have put a great deal of effort into making DWIM "smart", and experience with perhaps fifty different users indicates we have been very successful; DWIM seldom fails to correct an error the user feels it should have, and almost never mistakenly corrects an error. However, it is important to note that even when DWIM *is* wrong, no harm is done:³ since an error had occurred, the user would have had to intervene anyway if DWIM took no action. Thus, if DWIM mistakenly corrects an error, the user simply interrupts or aborts the computation, UNDOes the DWIM change using UNDO described in Section 22, and makes the correction he would have had to make without DWIM. It is this benign quality of DWIM that makes it a valuable part of Interlisp.

17.2 INTERACTION WITH DWIM

DWIM is enabled by performing either DWIM[C], for CAUTIOUS mode, or DWIM[T] for TRUSTING mode.⁴ In addition to setting dwimflg to T and redefining faulteval and faultapply as described on page 17.10, DWIM[C] sets approvelflg to T, while DWIM[T] sets approvelflg to NIL. The setting of approvelflg determines whether or not the user wishes to be asked for approval before a correction that will modify the definition of one of his functions. In CAUTIOUS mode, i.e., approvelflg=T, DWIM will ask for approval; in TRUSTING mode, DWIM will not. For corrections to expressions typed in by the user for immediate execution,⁵ DWIM always acts as

³ Except perhaps if DWIM's correction mistakenly caused a destructive computation to be initiated, and information was lost before the user could interrupt. We have not yet had such an incident occur.

⁴ Interlisp arrives with DWIM enabled in CAUTIOUS mode. DWIM can be disabled by executing DWIM[] or by setting dwimflg to NIL. See page 17.20.

⁵ Typed into lispx. lispx is used by evalqt and break, as well as for processing the editor's E command. Functions that call the spelling corrector directly, such as editdefault (Section 9), specify whether or not the correction is to be handled as type-in. For example, in the case of editdefault, commands typed directly to the editor are treated as type-in, so that corrections to them will never require approval. Commands given as an argument to the editor, or resulting from macro expansions, or from IF, LP, ORR commands etc. are not treated as type-in, and thus approval will be requested if approvelflg=T.

though approveflg were NIL, i.e., no approval necessary.⁶ In either case, DWIM always informs the user of its action as described below.

SPELLING CORRECTION PROTOCOL

The protocol used by DWIM for spelling corrections is as follows: If the correction occurs in type-in, print = followed by the correct spelling, followed by a carriage-return, and then continue, e.g.,

```
user types: ←(SETQ FOO (NCOCN FIE FUM))
DWIM types:      =NCONC
```

If the correction does not occur in type-in, print the incorrect spelling, followed by [IN function-name], ->, and then the correct spelling, e.g., ITIMS [IN FACT] -> ITIMES as shown on page 17.2.⁷ Then, if approveflg=NIL, print a carriage return, make the correction and continue. Otherwise, print a few spaces and a ? and then wait for approval.⁸ The user then has six options. He can:

1. Type Y; DWIM types es, and proceeds with the correction.
2. Type N; DWIM types o, and does not make the correction.
3. Type ↑; DWIM does not make the correction, and furthermore guarantees that the error will not cause a break. See footnote on page 17.10.
4. Type control-E; for error correction, this has the same effect as typing N.
5. Do nothing; in which case DWIM will wait a specified interval,¹⁰ and if the user has not responded, DWIM will type ... followed by the default answer.¹¹
6. Type space or carriage-return; in which case DWIM will wait indefinitely. This option is intended for those cases where the user wants to think about his answer, and wants to insure that DWIM does not get "impatient" and answer for him.

⁶ For certain types of corrections, e.g., run-on spelling corrections, 8-9 errors, etc., DWIM always asks for approval, regardless of the setting of approveflg.

⁷ The appearance of -> is to call attention to the fact that the user's function will be or has been changed.

⁸ DWIM uses askuser for its interactions with the user (page 17.22). Whenever an interaction is about to take place and the user has typed ahead, askuser types several bells⁹ to warn the user to stop typing, then clears and saves the input buffers, restoring them after the interaction is complete. Thus if the user has typed ahead before a DWIM interaction, DWIM will not confuse his type ahead with the answer to its question, nor will his typeahead be lost.

⁹ The bells are printed by the function printbells, which can be advised or redefined for specialized applications, e.g. to flash the screen for a display terminal.

¹⁰ Equal to dwimwait seconds. DWIM operates by dismissing for 250 milliseconds, then checking to see if anything has been typed. If not, it dismisses again, etc. until dwimwait seconds have elapsed. Thus, there will be a delay of at most 1/4 second before DWIM responds to the user's answer.

* ¹¹ The default on spelling corrections is determined by the value of the variable fixspelldefault, whose top level value is initially Y.

The procedure for spelling correction on other than Interlisp errors is analogous. If the correction is being handled as type-in, DWIM prints = followed by the correct spelling, and returns it to the function that called DWIM, e.g., =FACT as shown on page 17.2. Otherwise, DWIM prints the incorrect spelling, followed by the correct spelling. Then if approveflg=NIL, DWIM prints a carriage-return and returns the correct spelling. Otherwise, DWIM prints a few spaces and a ? and then waits for approval. The user can then respond with Y, N, control-E, space, carriage return, or do nothing as described.

Note that since the spelling corrector itself is not errorset protected, typing N and typing control-E may have different effects when the spelling corrector is called directly.¹² The former simply instructs the spelling corrector to return NIL, and lets the calling function decide what to do next; the latter causes an error which unwinds to the last errorset, however far back that may be.

PARENTHESES ERRORS PROTOCOL

As illustrated earlier on page 17.2, DWIM will correct errors consisting of typing 8 for left parenthesis and 9 for right parenthesis.¹³ In these cases, the interaction with the user is similar to that for spelling correction. If the error occurs in type-in, DWIM types = followed by the correction, e.g.,

```
user types: ←(SETQ FOO 8CONS FIE FUM]
DWIM types:      = ( CONS
lispx types: (A B C D)
```

Otherwise, DWIM prints the offending atom, [IN function-name], ->, the proposed correction, a few spaces and a ?, and then waits for approval, e.g., N9 [IN FACT] -> N) ? as shown on page 17.2. The user then has the same six options as for spelling correction.¹⁴ If the user types Y, DWIM then operates exactly the same as when approveflg=NIL, i.e., makes the correction and prints its message.

U.D.F. T ERRORS PROTOCOL

DWIM corrects certain types of parentheses errors involving a T clause in a conditional, namely errors of the form:

1. (COND --) (T --), i.e., the T clause appears outside and immediately following the COND;
2. (COND -- (-- & (T --))), i.e., the T clause appears inside a previous clause; and

¹² The DWIM error correction routines are errorset protection.

¹³ Actually, DWIM uses the value of the variables lparkey and rparkey to determine the corresponding lower case character for left and right parentheses. lparkey and rparkey are initially 8 and 9 respectively, but they can be reset for other keyboard layouts, e.g., on some terminals left parenthesis is over 9, and right parenthesis is over 0.

¹⁴ except the waiting time is 3*dwimwait seconds.

3. $(COND -- ((T --)))$, i.e., the T clause has an extra pair of parentheses around it.¹⁵

If the error occurs in type-in, DWIM simply types T FIXED and makes the correction. Otherwise if approveflg=NIL, DWIM makes the correction, and prints a message consisting of [IN function-name], followed by one of the above incorrect forms of COND, followed by ->, then on the next line the corresponding correct form of the COND, e.g.,

```
[IN FACT] (COND -- ((T --))) ->
          (COND -- (T --))
```

as shown on page 17.2.

If approveflg=T, DWIM prints U.D.F. T, followed by [IN function-name], several spaces, and then FIX? and waits for approval. The user then has the same options as for spelling corrections and parenthesis errors. If the user types Y or defaults, DWIM then proceeds exactly the same as when approveflg=NIL, i.e., makes the correction and prints its message, as shown on page 17.3.

Having made the correction, DWIM must then decide how to proceed with the computation. In case 1, $(COND --) (T --)$, DWIM cannot know whether the last clause of the COND before the T clause succeeded or not, i.e., if the T clause had been inside of the COND, would it have been entered? Therefore DWIM asks the user 'CONTINUE WITH T CLAUSE' (with a default of YES). If the user types N, DWIM continues with the form after the COND, i.e., the form that originally followed the T clause.

In case 2, $(COND -- (-- & (T --)))$, DWIM has a different problem. After moving the T clause to its proper place, DWIM must return as the value of the COND, the value of the expression corresponding to &. Since this value is no longer around, DWIM asks the user, 'OK TO REEVALUATE' and then prints the expression corresponding to &. If the user types Y, or defaults, DWIM continues by reevaluating &, otherwise DWIM aborts, and a U.D.F. T error will then occur (even though the COND has in fact been fixed).¹⁶

In case 3, $(COND -- ((T --)))$, there is no problem with continuation, so no further interaction is necessary.

¹⁵ For U.D.F. T errors that are not one of these three types, DWIM takes no corrective action at all, i.e., the error will occur.

¹⁶ If DWIM can determine for itself that the form can safely be reevaluated, it does not consult the user before reevaluating. DWIM can do this if the form is atomic, or car of the form is a member of the list okreevalst, and each of the arguments can safely be reevaluated, e.g., (SETQ X (CONS (IPLUS Y Z) W)) is safe to reevaluate because SETQ, CONS, and IPLUS are all on okreevalst.

17.3 SPELLING CORRECTION

The spelling corrector is given as arguments a misspelled word (word means literal atom), a spelling list (a list of words), and a number: xword, splst, and rel respectively. Its task is to find that word on splst which is closest to xword, in the sense described below. This word is called a *respelling* of xword. rel specifies the minimum "closeness" between xword and a respelling. If the spelling corrector cannot find a word on splst closer to xword than rel, or if it finds two or more words equally close, its value is NIL, otherwise its value is the respelling.¹⁷

The exact algorithm for computing the spelling metric is described later on page 17.16, but briefly "closeness" is inversely proportional to the number of disagreements between the two words, and directly proportional to the length of the longer word, e.g., PRTTYPRNT is "closer" to PRETTYPRINT than CS is to CONS even though both pairs of words have the same number of disagreements. The spelling corrector operates by proceeding down splst, and computing the closeness between each word and xword, and keeping a list of those that are closest. Certain differences between words are not counted as disagreements, for example a single transposition, e.g., CONS to CNOS, or a doubled letter, e.g., CONS to CONSS, etc. In the event that the spelling corrector finds a word on splst with *no* disagreements, it will stop searching and return this word as the respelling. Otherwise, the spelling corrector continues through the entire spelling list. Then if it has found one and only one "closest" word, it returns this word as the respelling. For example, if xword is VONS, the spelling corrector will probably return CONS as the respelling. However, if xword is CONZ, the spelling corrector will not be able to return a respelling, since CONZ is equally close to both CONS and COND. If the spelling corrector finds an acceptable respelling, it interacts with the user as described earlier.

In the special case that the misspelled word contains one or more <esc>s, the spelling corrector searches for those words on splst that match xword, where an <esc> (alt-mode on some terminals) can match any number of characters (including 0), e.g., FOO\$ matches FOO1 and FOO, but not NEWFOO. \$FOO\$ matches all three. Both completion and correction may be involved, e.g. RPETTY\$ will match PRETTYPRINT, with one mistake. The entire spelling list is always searched, and if more than one respelling is found, the spelling corrector prints AMBIGUOUS, and returns NIL. For example, CON\$ would be ambiguous if both CONS and COND were on the spelling list. If the spelling corrector finds one and only one respelling, it interacts with the user as described earlier.

For both spelling correction and spelling completion, regardless of whether or not the user approves of the spelling corrector's choice, the respelling is moved to the front of splst.¹⁸ Since many respellings are of the type with no disagreements, this procedure has the effect of considerably reducing the time required to correct the spelling of frequently misspelled words.

SYNOMYS

Spelling lists also provide a way of defining synonyms for a particular context. If a dotted pair appears on a spelling list (instead of just an atom), car is interpreted as the correct spelling of the misspelled word, and cdr as the antecedent for that word. If car is *identical* with the misspelled

¹⁷ The spelling corrector can also be given an optional functional argument, fn, to be used for selecting out a subset of splst, i.e., only those members of splst that satisfy fn will be considered as possible respellings.

¹⁸ fixspell has an optional argument, dontmovetopflg, which can be used to suppress moving the respelling.

word, the antecedent is returned without any interaction or approval being necessary. If the misspelled word *corrects* to car of the dotted pair, the usual interaction and approval will take place, and then the antecedent, i.e., cdr of the dotted pair, is returned. For example, the user could make IFLG synonymous with CLISPIFTRANFLG by adding (IFLG . CLISPIFTRANFLG) to spellings3, the spelling list for unbound atoms. Similarly, the user could make OTHERWISE mean the same as ELSEIF by adding (OTHERWISE . ELSEIF) to clispifwordsplst, or make L be synonymous with LAMBDA by adding (L . LAMBDA) to lambdasplst. Note that L could also be used as a variable without confusion, since the association of L with LAMBDA occurs only in the appropriate context.

SPELLING LISTS

Any list of atoms can be used as a spelling list, e.g., brokenfns, filelst, etc. Various system packages have their own spellings lists, e.g., lispcoms, prettycomsplst, clispforwordsplst, editcoms, etc. These are documented under their corresponding sections, and are also indexed under "spelling lists." In addition to these spelling lists, the system maintains, i.e., automatically adds to, and occasionally prunes, four lists used solely for spelling correction: spellings1, spellings2, spellings3, and userwords.¹⁹

Spellings1 is a list of functions used for spelling correction when an input is typed in apply format, and the function is undefined, e.g., EDTIF(FOO). Spellings1 is initialized to contain defineq, break, makefile, editif, tcompl, load, etc. Whenever lispx is given an input in apply format, i.e., a function and arguments, the name of the function is added to spellings1.²⁰ For example, typing CALLS(EDITF) will cause CALLS to be added to spellings1. Thus if the user typed CALLS(EDITF) and later typed CALLLS(EDITV), since spellings1 would then contain CALLS, DWIM would be successful in correcting CALLLS to CALLS.²¹

Spellings2 is a list of functions used for spelling correction for all other undefined functions. It is initialized to contain functions such as add1, append, cond, cons, go, list, nconc, print, prog, return, setq, etc. Whenever lispx is given a non-atomic form, the name of the function is added to spellings2. For example, typing (RETFROM (STKPOS (QUOTE FOO) 2)) to a break would add retfrom to spellings2. Function names are also added to spellings2 by define, defineq, load (when loading compiled code), unsavedef, editif, and prettyprint.

Spellings3 is a list of words used for spelling correction on all unbound atoms. Spellings3 is initialized to editmacros, breakmacros, brokenfns, and advisedfns. Whenever lispx is given an atom to evaluate, the name of the atom is added to spellings3.²² Atoms are also added to spellings3 whenever they are edited by edity, and whenever they are set via rpaq or rpaqq. For example, when a file is loaded, all of the variables set in the file are added to spellings3. Atoms are also added to spellings3 when they are set by a lispx input, e.g., typing (SETQ FOO (REVERSE (SETQ FIE --))) will add both FOO and FIE to spellings3.

* 19 All of the remarks on maintaining spelling lists apply *only* when addspellflg=T, its initial value.

20 Only if the function has a definition.

21 If CALLLS(EDITV) were typed before CALLS had been "seen" and added to spellings1, the correction would not succeed. However, the alternative to using spelling lists is to look at all the atoms in the system, a procedure that would make spelling correction intolerably slow.

22 Only if the atom has a value other than NOBIND.

Userwords is a list containing both functions and variables that the user has *referred* to, e.g., by breaking or editing. Userwords is used for spelling correction by arglist, unsavedef, prettyprint, break, editf, advise, etc. Userwords is initially NIL. Function names are added to it by define, defineq, load, (when loading compiled code, or loading exprs to property lists) unsavedef, editf, editv, editp, prettyprint, etc. Variable names are added to userwords at the same time as they are added to spellings3. In addition, the variable lastword is always set to the last word added to userwords, i.e., the last function or variable referred to by the user, and the respelling of NIL is defined to be the value of lastword. Thus, if the user has just defined a function, he can then edit it by simply typing EDITF(), or prettyprint it by typing PP().

Each of the above four spelling lists are divided into two sections separated by a special marker. The first section contains the "permanent" words; the second section contains the temporary words. New words are added to the corresponding spelling list at the front of its temporary section.²³ (If the word is already in the temporary section; it is moved to the front of that section; if the word is in the permanent section, no action is taken.) If the length of the temporary section then exceeds a specified number, the last (oldest) word in the temporary section is forgotten, i.e., deleted. This procedure prevents the spelling lists from becoming cluttered with unimportant words that are no longer being used, and thereby slowing down spelling correction time. Since the spelling corrector moves each word selected as a respelling to the front of its spelling list,²⁴ the word is thereby moved into the permanent section. Thus once a word is misspelled and corrected, it is considered important and will never be forgotten.

The maximum length of the temporary section for spellings1, spellings2, spellings3 and userwords is given by the value of #spellings1, #spellings2, #spellings3, and #userwords, initialized to 30, 30, 30, and 60 respectively. Using these values, the average length of time to search a spelling list for one word is about 4 milliseconds.²⁵

GENERATORS FOR SPELLING CORRECTION

For some applications, it is more convenient to *generate* candidates for a respelling one by one, rather than construct a complete list of all possible candidates, e.g., spelling correction involving a large directory of files, or a natural language data base. For these purposes, splst can be an array (of any size). The first element of this array is the generator function, which is called with the array itself as its argument. Thus the function can use the remainder of the array to store "state" information, e.g., the last position on a file, a pointer into a data structure, etc. The value returned by the function is the next candidate for respelling. If NIL is returned, the spelling "list" is considered to be exhausted, and the closest match is returned. If a candidate is found with no disagreements, it is returned immediately without waiting for the "list" to exhaust.

splst can also be a generator, i.e. the value of the function generator (Section 12). The generator splst will be started up whenever the spelling corrector needs the next candidate, and it should return candidates via the function produce. For example,

²³ Except that functions added to spellings1 or spellings2 by lispx are always added to the end of the permanent section.

²⁴ Unless dontmovetopflg, one of the arguments to fixspell, is T.

²⁵ If the word is at the front of the spelling list, the time required is only 1 millisecond. If the word is not on the spelling list, i.e., if the entire list must be searched, the time is proportional to the length of the list; to search a spelling list of length 60 takes about 7 milliseconds.

- + [GENERATOR (MAPATOMS (FUNCTION (LAMBDA (X) (AND (FNTYP X) (PRODUCE X)
- + could be used as a "spelling list" which effectively contains all functions in the system.

17.4 ERROR CORRECTION

As described in Section 16, whenever the interpreter encounters an atomic form with no binding, or a non-atomic form car of which is not a function or function object, it calls the function faulteval. Similarly, when apply is given an undefined function, it calls faultapply. When DWIM is enabled, faulteval and faultapply are redefined to first call dwimblock, a part of the DWIM package. If the user aborts by typing control-E, or if he indicates disapproval of DWIM's intended correction by answering N as described on page 17.4, or if DWIM cannot decide how to fix the error, dwimblock returns NIL.²⁶ In this case, faulteval and faultapply proceed exactly as described in Section 16, by printing a U.B.A. or U.D.F. message, and going into a break if the requirements of breakcheck are met, otherwise unwinding to the last errorset.

If DWIM can (and is allowed to) correct the error, dwimblock exits by performing reteval of the corrected form, as of the position of the call to faulteval or faultapply. Thus in the example at the beginning of the chapter, when DWIM determined that ITIMS was ITIMES misspelled, DWIM called reteval with (ITIMES N (FACCT 8SUB1 N)). Since the interpreter uses the value returned by faulteval exactly as though it were the value of the erroneous form, the computation will thus proceed exactly as though no error had occurred.

In addition to continuing the computation, DWIM also repairs the cause of the error whenever possible.²⁷ Thus in the above example, DWIM also changed (with rplaca) the expression (ITIMS N (FACCT 8SUB1 N)) that caused the error.

Error correction in DWIM is divided into three categories: unbound atoms, undefined cars of form, and undefined function in apply. Assuming that the user approves if he is asked, the action taken by DWIM for the various types of errors in each of these categories is summarized below. The protocol of DWIM's interaction with the user has been described earlier.

UNBOUND ATOMS

1. If the first character of the unbound atom is ', DWIM assumes that the user (intentionally) typed 'atom for (QUOTE atom) and makes the appropriate change. No message is typed, and no approval requested.

If the unbound atom is just ' itself, DWIM assumes the user wants the *next* expression quoted, e.g., (CONS X '(A B C)) will be changed to (CONS X (QUOTE (A B C))). Again no message will be printed or approval asked. (If no expression follows the ', DWIM

²⁶ If the user answers with ↑, (see page 17.4) dwimblock is exited by performing reteval[FAULTEVAL;(ERROR!)], i.e., an error is generated at the position of the call to faulteval.

²⁷ If the user's program had *computed* the form and called eval, e.g., performed (EVAL (LIST X Y)) and the value of x was a misspelled function; it would not be possible to repair the cause of the error, although DWIM could correct the misspelling each time it occurred.

gives up.)²⁸

2. If CLISP (Section 23) is enabled, and the atom is part of a CLISP construct, the CLISP transformation is performed and the result returned, e.g., N-1 is transformed to (SUB1 N), (... FOO \leftarrow 3 ...) is transformed into (... (SETQ FOO 3 ...)).
3. If the atom contains an 8,²⁹ DWIM assumes the 8 was intended to be a left parenthesis, and calls the editor to make appropriate repairs on the expression containing the atom. DWIM assumes that the user did not notice the mistake, i.e., that the entire expression was affected by the missing left parenthesis. For example, if the user types (SETQ X (LIST (CONS 8CAR Y) (CDR Z)) Y), the expression will be changed to (SETQ X (LIST (CONS (CAR Y) (CDR Z)) Y)).

The 8 does not have to be the first character of the atom, e.g., DWIM will handle (CONS X8CAR Y) correctly.

4. If the atom contains a 9³⁰ DWIM assumes the 9 was intended to be a right parenthesis and operates as in number 3.
5. If the atom begins with a 7, the 7 is treated as a ', e.g., 7FOO becomes 'FOO, and then (QUOTE FOO).
6. If the atom is an edit command (a member of editcomsa), and the error occurred in type-in, the effect is the same as though the user typed EDITF(), followed by the atom, i.e., DWIM assumes the user wants to be in the editor editing the last thing he referred to. Thus, if the user defines the function foo and then types P, he will see =FOO, followed by EDIT, followed by the printout associated with the execution of the P command, followed by *, at which point he can continue editing foo.
7. The expressions on dwimuserforms are evaluated in the order that they appear, and if any returns a non-NIL value, this value is treated as the form to be used to continue the computation, and is evaluated and its value returned by DWIM. dwimuserforms is discussed further below.
8. If the unbound atom occurs in a function, DWIM attempts spelling correction using as a spelling list the list of lambda and prog variables of the function.

²⁸ Since ' is normally defined as a read-macro character which converts 'FOO to (QUOTE FOO) on input, DWIM will not see the ' in the case of expressions that are typed-in. +

²⁹ actually the value of lparkey, initially 8. See footnote on page 17.5.

³⁰ actually the value of rparkey, initially 9. See footnote on page 17.5.

9. If the unbound atom occurred in a type-in to a break, DWIM attempts spelling correction using the lambda and prog variables of the broken function.
10. Otherwise, DWIM attempts spelling correction using spellings3.

If all fail, DWIM gives up.

UNDEFINED CAR OF FORM

1. If car of the form is T, DWIM assumes a misplaced T clause and operates as described on page 17.5.
2. If car of the form is F/L, DWIM changes the F/L to FUNCTION(LAMBDA, e.g., (F/L (Y) (PRINT (CAR Y))) is changed to (FUNCTION (LAMBDA (Y) (PRINT (CAR Y))). No message is printed and no approval requested. If the user omits the variable list, DWIM supplies (X), e.g., (F/L (PRINT (CAR X))) becomes (FUNCTION (LAMBDA (X) (PRINT (CAR X))). DWIM determines that the user has supplied the variable list when more than one expression follows F/L, car of the first expression is not the name of a function, and every element in the first expression is atomic. For example, DWIM will supply (X) when correcting (F/L (PRINT (CDR X)) (PRINT (CAR X))).
- * 3. If car of the form is IF, or one of the CLISP iterative statement operators, e.g., FOR, WHILE, DO et al, or FETCH, REPLACE, MATCH, etc., the indicated transformation is performed, and the result returned as the corrected form.³¹
4. If car of the form has a function definition, DWIM attempts spelling correction on car of the definition using as spelling list the value of lambdaasplst, initially (LAMBDA NLAMBDA).³²
- * 5. If car of the form has an EXPR or CODE property, DWIM prints car of the form, followed by 'UNSAVED', performs an unsavedef, and continues. No approval is requested.
6. If car of the form has a property FILEDEF,³³ the definition is to be found on a file. If the

+ 31 This transformation is keyed by putting on the property list of car of the form, e.g. IF, WHILE, FETCH, under the property CLISWORD, a list car of which is the atom IFWORD, FORWARD, RECORDWORD, etc. Any other value for car of this property is simply applied to the faulty form, and the value returned used as the corrected form. This provides a way of defining new transformations keyed by car of the form without having to use the more general dwimuserforms mechanism.

32 The user may wish to add to lambdaasplst if he elects to define new "function types" via an appropriate dwimuserforms entry. For example, the QLAMBDA of SRI's QLISP are handled in this way.

33 except when dwimifying.

value of the property is atomic, the entire file is to be loaded. If a list, car is the name of the file and cdr the relevant functions, and loadfns will be used. For both cases, ldflg will be SYSLOAD (see Section 14). DWIM uses findfile (described later), so that the file can be on any of the directories on directories, initially (NIL NEWLISP LISP LISPUERS). If the file is found, DWIM types "SHALL I LOAD" followed by the file name or list of functions. If the user approves, DWIM loads the function(s) or file, and continues the computation.

- 7. If CLISP is enabled, and car of the form is part of a CLISP construct, the indicated transformation is performed, e.g., (N \leftarrow N-1) becomes (SETQ N (SUB1 N)).
- 8. If car of the form contains an 8, DWIM assumes a left parenthesis was intended e.g., (CONS8CAR X).
- 9. If car of the form contains a 9, DWIM assumes a right parenthesis was intended.
- 10. If car of the form is a list, DWIM attempts spelling correction on caar of the form using lambdasplst as spelling list. If successful, DWIM returns the corrected expression itself.
- 11. If car of the form is a small number, and the error occurred in type-in, DWIM assumes the form is really an edit command and operates as described in case 6 of unbound atoms.
- 12. If car of the form is an edit command (a member of editcoms1), DWIM operates as in 11.
- 13. The expressions on dwinduserforms are evaluated in the order they appear, and if any returns a non-NIL value, this value is treated as the corrected form, i.e. it is evaluated and DWIM returns its value.
- 14. Otherwise, DWIM attempts spelling correction using spellings2 as the spelling list. When dwimifying, DWIM also attempts spelling correction on function names not defined but previously encountered, using nofixfnlst as a spelling list.

If all fail, DWIM gives up.

UNDEFINED FUNCTION IN APPLY

- 1. If the function has a definition, DWIM attempts spelling correction on car of the definition using lambdasplst as spelling list.
- 2. If the function has an EXPR or CODE property, DWIM prints its name followed by 'UNSAVED', performs an unsaveddef and continues. No approval is requested.

3. If the function has a property FILEDEF, DWIM proceeds as in case 6 of undefined car of form.
4. If the error resulted from type-in, and CLISP is enabled, and the function name contains a CLISP operator, DWIM performs the indicated transformation, e.g., the user types FOO<(APPEND FIE FUM).
5. If the function name contains an 8, DWIM assumes a left parenthesis was intended, e.g., EDIT8FOO].
6. If the "function" is a list, DWIM attempts spelling correction on car of the list using lambdasplist as spelling list.
7. If the function is a number and the error occurred in type-in, DWIM assumes the function is an edit command, and operates as described in case 6 of unbound atoms, e.g., the user types (on one line) 3 -1 P.
8. If the function is the name of an edit command (on either editcomsa or editcomsl), DWIM operates as in 7, e.g., user types F COND.
- * 9. The expressions on dwimuserforms are evaluated in the order they appear, and if any returns a non-NIL value, this value is treated as the function used to continue the computation, i.e., it will be applied to its arguments.
10. Otherwise DWIM attempts spelling correction using spellings1 as the spelling list.
11. Otherwise DWIM attempts spelling correction using spellings2 as the spelling list.

If all fail, DWIM gives up.

17.5 DWIMUSERFORMS

- * Dwimuserforms provides a convenient way of adding to the transformations that DWIM performs, e.g., the user might want to change atoms of the form \$X to (QA4LOOKUP X). DWIM will evaluate each expression on dwimuserforms in the order they appear before attempting spelling correction, but after performing its other transformations, e.g., F/L, 8, 9, CLISP, etc. If any expression returns a non-NIL value, this value is treated as a form to be evaluated, and is evaluated and the resulting value returned as the value of faulteval, or, in the case of an undefined function in apply, this value is treated as a function to be applied to faultargs, and the resulting value is returned as the value of faultapply. Otherwise, if all return NIL, DWIM proceeds as when dwimuserforms=NIL, and attempts spelling correction. Note that in the event that an expression on dwimuserforms is to handle the correction, it is also responsible for any modifications to the

original expression,³⁴ i.e., DWIM simply takes its value and returns it.

In order for an expression on dwimuserforms to be able to be effective, it needs to know various things about the context of the error. Therefore, several of DWIM's internal variables have been made SPECVARS (see Section 18) and are therefore "visible" to dwimuserforms. Below are a list of those variables that may be useful.

<u>faultx</u>	for unbound atoms and undefined car of form, <u>faultx</u> is the atom or form. For undefined functions in <u>apply</u> , <u>faultx</u> is the name of the function.
<u>faultargs</u>	for undefined functions in <u>apply</u> , <u>faultargs</u> is the list of arguments. <u>faultargs</u> may be modified or reset by expressions on <u>dwimuserforms</u> .
<u>faultapplyflg</u>	is T for undefined functions in <u>apply</u> . (Since <u>faultargs</u> may be NIL, <u>faultapplyflg</u> is necessary to distinguish between unbound atoms and undefined function in <u>apply</u> , since <u>faultx</u> is atomic in both cases). The value of <u>faultapplyflg</u> after an expression on <u>dwimuserforms</u> returns a non-NIL value determines how the latter value is to be treated. Thus, following an undefined function in <u>apply</u> , i.e. when <u>faultapplyflg</u> is T, an expression on <u>dwimuserforms</u> can construct and return an expression to be treated as a form to be evaluated, rather than a function to be applied, by first setting <u>faultapplyflg</u> to NIL.
<u>tail</u>	for unbound errors, <u>tail</u> is the tail <u>car</u> of which is the unbound atom. Thus <u>dwimuserfn</u> can replace the atom by another expression by performing (/RPLACA TAIL expr)
<u>parent</u>	for unbound atom errors, <u>parent</u> is the form in which the unbound atom appears, i.e., <u>tail</u> is a tail of <u>parent</u> .
<u>type-in?</u>	true if error occurred in type-in.
<u>faultfn</u>	name of function in which error occurred. (<u>faultfn</u> is TYPE-IN when the error occurred in type-in, and EVAL or APPLY when the error occurred under an explicit call to EVAL or APPLY).
<u>dwimifyflg</u>	true if error was encountered during dwimifying as opposed to during running the program.
<u>expr</u>	definition of <u>faultfn</u> , or argument to <u>eval</u> , i.e., the superform in which the error occurs.

The initial value of dwimuserforms is ((MACROTRAN) (DWIMLOADFNS?)). macrotran is a package for running interpreted programs containing assemble statements or calls to "functions" +

³⁴ i.e. dwimuserforms should make the transformation permanent, either by associating it with faultx via clisptran, or by physically smashing faultx. +

- + defined only by MACRO properties. It is described in Section 18. dwinloadfn? is a package for
- + automatically loading functions from files. If dwinloadfn\$flg is T (its initial value), and car of the
- + form is the name of a function, and the function is contained on a file that has been noticed by the
- + file package, the function is loaded, and the computation continues.

17.6 SPELLING CORRECTOR ALGORITHM

The basic philosophy of DWIM spelling correction is to count the number of disagreements between two words, and use this number divided by the length of the longer of the two words as a measure of their relative disagreement. One minus this number is then the relative agreement or closeness. For example, CONS and CONX differ only in their last character. Such substitution errors count as one disagreement, so that the two words are in 75% agreement. Most calls to the spelling corrector specify a relative agreement of 70.³⁵ ³⁶ so that a single substitution error is permitted in words of four characters or longer. However, spelling correction on shorter words is possible since certain types of differences such as single transpositions are not counted as disagreements. For example, AND and NAD have a relative agreement of 100.

The central function of the spelling corrector is chooz. chooz takes as arguments: a word, a spelling list, a minimum relative agreement, and an optional functional argument, xword, splst, rel, and fn respectively.³⁷

chooz proceeds down splst examining each word. Words not satisfying fn, or those obviously too long or too short to be sufficiently close to xword are immediately rejected. For example, if rel=70, and xword is 5 characters long, words longer than 7 characters will be rejected.³⁸

If tword, the current word on splst, is not rejected, chooz computes the number of disagreements between it and xword by calling a subfunction, skor.

skor operates by scanning both words from left to right one character at a time.³⁹ Characters are considered to agree if they are the same characters; or appear on the same teletype key (i.e., a shift mistake), for example, * agrees with :, 1 with !,⁴⁰ etc.; or if the character in xword is a lower case

³⁵ Integers between 0 and 100 are used instead of numbers between 0 and 1 in order to avoid floating point arithmetic.

+ ³⁶ Calls to the spelling corrector from DWIM use the value of fixspellrel, which is initially 70. Note that by setting fixspellrel to 100, only spelling corrections with "zero" mistakes, will be considered, e.g. transpositions, double characters, etc. as described below.

³⁷ fn=NIL is equivalent to fn=(LAMBDA NIL T).

³⁸ Special treatment is necessary for words shorter than xword, since doubled letters are not counted as disagreements. For example, CONNSSS and CONS have a relative agreement of 100. (Certain teletype diseases actually produce this sort of stuttering.) chooz handles this by counting the number of doubled characters in xword before it begins scanning splst, and taking this into account when deciding whether to reject shorter words.

³⁹ skor actually operates on the list of character codes for each word. This list is computed by chooz before calling skor using dchcon, so that no storage is used by the entire spelling correction process.

⁴⁰ For users on model 33 teletypes, as indicated by the value of model33flg being T, @ and P appear on the same key, as do L and /, N and L, and O and ^, and DWIM will proceed accordingly. The initial value for model33flg is NIL. Certain other terminals, e.g., Anderson Jacobs terminal, have keyboard layouts similar to the model 33, i.e., N on same key as ^, etc. In this case, the user might also want to set model33flg to T.

version of the character in xword. Characters that agree are discarded, and the skoring continues on the rest of the characters in xword and tword.

If the first character in xword and tword do *not* agree, skor checks to see if either character is the same as one previously encountered, and not accounted-for at that time. (In other words, transpositions are not handled by lookahead, but by *lookback*.) A displacement of two or fewer positions is counted as a tranposition; a displacement by more than two positions is counted as a disagreement. In either case, both characters are now considered as accounted for and are discarded, and skoring continues.

If the first character in xword and tword do not agree, and neither agree with previously unaccounted-for characters, and tword has more characters remaining than xword, skor removes and saves the first character of tword, and continues by comparing the rest of tword with xword as described above. If tword has the same or fewer characters remaining than xword, the procedure is the same except that the character is removed from xword.⁴¹ In this case, a special check is first made to see if that character is equal to the *previous* character in xword, or to the *next* character in xword, i.e., a double character typo, and if so, the character is considered accounted-for, and not counted as a disagreement.⁴²

When skor has finished processing both xword and tword in this fashion, the value of skor is the number of unaccounted-for characters, plus the number of disagreements, plus the number of tranpositions, with two qualifications: (1) if both xword and tword have a character unaccounted-for in the same position, the two characters are counted only once, i.e., substitution errors count as only one disagreement, not two; and (2) if there are no unaccounted-for characters and no disagreements, transpositions are not counted. This permits spelling correction on very short words, such as edit commands, e.g., XRT->XTR.⁴³

17.7 DWIM FUNCTIONS AND VARIABLES

dwim[x]

If x=NIL, disables DWIM; value is NIL. If x=C, enables DWIM in cautious mode; value is CAUTIOUS. If x=T, enables DWIM in trusting mode; value is TRUSTING. For all other values of x, generates an error.

dwimify[x;quietflg]

x is a form or the name of a function. dwimify performs all corrections and transformations that would occur if x were actually run, and prints the result unless quietflg=T. dwimify is undoable. *

⁴¹ Whenever more than two characters in either xword or tword are unaccounted for, skoring is aborted, i.e., xword and tword are considered to disagree.

⁴² In this case, the "length" of xword is also decremented. Otherwise making xword sufficiently long by adding double characters would make it be arbitrarily close to tword, e.g., XXXXX would correct to PP.

⁴³ Transpositions are also not counted when fasttypeflg=T, for example, IPULX and IPLUS will be in 80% agreement with fasttypeflg=T, only 60% with fasttypeflg=NIL. The rationale behind this is that transpositions are much more common for fast typists, and should not be counted as disagreements, whereas more deliberate typists are not as likely to combine tranpositions and other mistakes in a single word, and therefore can use more conservative metric. fasttypeflg is initially NIL.

DWedit macro. dwimifies current expression.**addspell[x;splst;n]**Adds x to one of the four spelling lists as follows:⁴⁴If splst=NIL, adds x to userwords and to spellings2. Used by defineq.If splst=0, adds x to userwords. Used by load when loading exprs to property lists.If splst=1, adds x to spellings1 (at end of permanent section). Used by lispX.If splst=2, adds x to spellings2 (at end of permanent section). Used by lispX.If splst=3, adds x to userwords and spellings3.splst can also be a spelling list, in which case n is the (optional) length of the temporary section.addspell sets lastword to x when splst=NIL, 0 or 3.If x is not a literal atom, addspell takes no action.

- + Note that the various systems calls to addspell, e.g. from define, editf, load, etc., can all be suppressed by setting or binding addspellflg to NIL.

misspelled?[xword;rel;splst;flg;tail;fn]If xword=NIL or esc, misspelled? prints = followed by the value of lastword, and returns this as the respelling, without asking for approval. Otherwise, misspelled? checks to see if xword is really misspelled, i.e., if fn applied to xword is true, or xword is already contained on splst. In this case, misspelled? simply returns xword. Otherwise misspelled? computes and returns fixspell[xword;rel;splst;flg;tail;fn].**fixspell[xword;rel;splst;flg;tail;fn;tieflg;dontmovetopflg;:-:-]**The value of fixspell is either the respelling of xword or NIL.⁴⁵ fixspell performs all of the interactions described earlier, including requesting user approval if necessary.If xword=NIL or \$ (<esc>), the respelling is the value of lastword, and no approval is requested.

- + If xword contains lowercase characters, and the corresponding

⁴⁴ If x is already on the spelling list, and in its temporary section, addspell moves x to the front of that section. See page 17.9 for complete description of algorithm for maintaining spelling lists.

⁴⁵ If for some reason xword itself is on splst, then fixspell aborts and calls error!. If there is a possibility that xword is spelled correctly, misspelled? should be used instead of fixspell.

uppercase word is correct, i.e. on splist or satisfies fn, the uppercase word is returned and no interaction is performed.

If rel=NIL, defaults to value of fixspellrel (initially 70).

If flg=NIL, the correction is handled in type-in mode, i.e., approval is never requested, and xword is not typed. If flg=T, xword is typed (before the =) and approval is requested if approvelflg=T. If flg=NO-MESSAGE, the correction is returned with no further processing.⁴⁶

If tail is not NIL, and the correction is successful, car of tail is replaced by the respelling (using /rplaca). In addition, fixspell will correct misspellings caused by running two words together.⁴⁷ In this case, car of tail is replaced by the two words, and the value of fixspell is the first one. For example, if fixspell is called to correct the edit command (MOVE TO AFTERCOND 3 2) with tail=(AFTERCOND 3 2), tail would be changed to (AFTER COND 2 3), and fixspell would return AFTER (subject to user approval where necessary).⁴⁸

If tieflg=NIL and a tie occurs, i.e., more than one word on splist is found with the same degree of "closeness", fixspell returns NIL, i.e., no correction. If tieflg=PICKONE and a tie occurs, the first word is taken as the correct spelling. If tieflg=LIST, the value of fixspell is a list of the respellings (even if there is only one), and fixspell will not perform any interaction with the user, nor modify tail, the idea being that the calling program will handle those tasks. Similarly, if tieflg=EVERYTHING, a list of all candidates whose degree of closeness is above rel will be returned, regardless of whether some are better than others. No interaction will be performed.

If dontmovetopflg=T and a correction occurs, it will *not* be moved to the front of the spelling list.

The time required for a call to fixspell with a spelling list of length 60 when the entire list must be searched is .5 seconds. If fixspell determines that the first word on the spelling list is the respelling and does not need to search any further, the time required is .02 seconds. In other words, the time required is proportional to the number of words with which xword is compared, with the time

⁴⁶ In this case, a run-on correction will be returned as a dotted pair of the two parts of the word, and a synonym correction as a list of the form (word1 word2), where word1 is (the corrected version of) xword, and word2 is the synonym. Note that the effect of the function chooz (documented in previous manuals but no longer available) can be obtained by calling fixspell with flg=NO-MESSAGE.

⁴⁷ In this case, user approval is always requested. In addition, if the first word contains either fewer than 3 characters, or fewer characters than the second word, the default will be N. 'Run-on' spelling corrections can be suppressed by setting the variable runonflg to NIL (initially T).

⁴⁸ If tail=T, fixspell will also perform run-on corrections, returning a dotted pair of the two words in the event the correction is of this type.

for one comparison, i.e., one call to skor takes roughly .01 seconds (varies slightly with the number of characters in the words being compared).

dwimflg if NIL, suppresses all dwim operations.

+ **fixspellrel** default for rel on calls to fixspell when rel=NIL (most system calls to fixspell specify rel=NIL). Initial value is 70.

+ **fixspelldefault** if approval is requested for a spelling correction, and user does not respond, defaults to value of fixspelldefault, initially Y. fixspelldefault is rebound to N when dwimifying.

+ **dwinwait** number of seconds before dwin assumes the user is not going to respond to a question and uses the default response.

+ **addspellflg** if NIL, suppresses calls to addspell. Initially T.

nospellflg if T, suppresses *all* spelling correction. If some other non-NIL value, suppresses spelling correction in programs but not type-in. nospellflg is initially NIL. It is rebound to T when compiling from a file.

runonflg if NIL, suppresses run-on spelling corrections. Initialy T.

+ **dwinuserforms** allows user to specify his own corrections or transformations. Initial value is ((MACROTRAN) (DWIMLOADFNS?)). See page 17.14.

+ **dwinloadfnfslg** if T, tells DWIM when it encounters a call to an undefined function contained on a file that has been noticed by the file package, to simply load the function. dwinloadfnfslg is initially T. See page 17.16.

fncheck[fn;noerrorflg;spellflg;propflg;tail]

The task of fncheck is to check whether fn is the name of a function and if not, to correct its spelling.⁴⁹ If fn is the name of a function or spelling correction is successful, fncheck adds the (corrected) name of the function to userwords using addspell, and returns it as its value.

⁴⁹ Since fncheck is called by many low level functions such as arglist, unsavedef, etc., spelling correction only takes place when dwinflg=T, so that these functions can operate in a small Interlisp system which does not contain DWIM.

noerrorflg informs fncheck whether or not the calling function wants to handle the unsuccessful case: if noerrorflg is T, fncheck simply returns NIL, otherwise it prints fn NOT A FUNCTION and generates a non-breaking error.

If fn does not have a definition, but does have an EXPR property, then spelling correction is not attempted. Instead, if propflg=T, fn is considered to be the name of a function, and is returned. If propflg=NIL, fn is *not* considered to be the name of a function, and NIL is returned or an error generated, depending on the value of noerrorflg.

fncheck calls misspelled? to perform spelling correction, so that if fn=NIL, the value of lastword will be returned. spellflg corresponds to misspelled?'s fourth argument, flg. If spellflg=T, approval will be asked if DWIM was enabled in CAUTIOUS mode, i.e., if approveflg=T. tail corresponds to the fifth argument to misspelled?.

fncheck is currently used by arglist, unsavedef, prettyprint, break0, breakin, advise, calls, and edita. For example, break0 calls fncheck with noerrorflg=T since if fncheck cannot produce a function, break0 wants to define a dummy one. calls however calls fncheck with noerrorflg=NIL, since it cannot operate without a function.

Many other system functions call misspelled? or fixspell directly. For example, break1 calls fixspell on unrecognized atomic inputs before attempting to evaluate them, using as a spelling list a list of all break commands. Similarly, lispX calls fixspell on atomic inputs using a list of all lispX commands. When unbreak is given the name of a function that is not broken, it calls fixspell with two different spelling lists, first with brokenfns, and if that fails, with userwords. makefile calls misspelled? using filclst as a spelling list. Finally, load, bcompl, bcompile, tcompl, and recompile all call misspelled? if their input file(s) won't open.

spellfile[file;noprintflg;nsflg] if file does not have a directory field, spellfile looks on the directories given by the value of directories, initially (NIL LISP). (NIL corresponds to login directory.) This correction will not require user approval, (but spellfile will indicate the correction in the usual way, by printing = followed by the new file name). Otherwise, spellfile attempts spelling correction against the files in the directory. In this case, user approval will be requested (except if nprintflg=T, see below). Value is corrected file, if any, otherwise NIL.

If noprintflg=T, spellfile does not do any printing, nor ask for approval.

If nsflg=T (or nospellflg=T), no spelling correction is attempted, though searching through directories will still be performed.

findfile[file;nsflg] If file is not the name of a file, calls spellfile specifying no interaction or printing, i.e. findfile is defined as (OR (INFILEP FILE) (SPELLFILE FILE T NSFLG)).

17.8 ASKUSER

DWIM, the compiler, the editor, and many other system packages all use askuser, an extremely general user interaction package, for their interactions with the user at the terminal. askuser takes as its principal argument a keylst which is used to drive the interaction. keylst specifies what the user can type at any given point, how askuser should respond to the various inputs, what value should be returned by askuser, and is also used to present the user at any given point with a list of the possible responses. askuser also takes other arguments which permit specifying a wait time, a default value, a message to be printed on entry, a flag indicating whether or not typeahead is to be permitted, a flag indicating whether the transaction is to be stored on the history list (Section 22), a default set of options, and an (optional) input file/string.

STARTUP PROTOCOL

Interlisp permits and encourages the user to typeahead; in actual practice, the user frequently does this. This presents a problem for askuser: when askuser is entered and there has been typeahead, was the input intended for askuser, or was the interaction unanticipated, and the user simply typing ahead to some other program, e.g. the programmer's assistant? Even where there was no typeahead, i.e., the user starts typing *after* the call to askuser, the question remains of whether the user had time to see the message from askuser and react to it, or simply began typing ahead at an inauspicious moment. Thus, what is needed is an interlock mechanism which warns the user to stop typing, gives him a chance to respond to the warning, and then allows him to begin typing to askuser.

Therefore, when askuser is first entered, and the interaction is to take place with a terminal, and typeahead to askuser is not permitted, the following protocol is observed:

- (1) If there is typeahead, askuser clears and saves the input buffers⁵⁰ and rings the bell to warn the user to stop typing.
- (2) If mess, the message to be printed on entry, is not NIL (the typical case), askuser then prints mess if it is a string, otherwise car of mess, if mess is a list.
- (3) After printing mess or car of mess, askuser waits until the output has actually been printed on the terminal to make sure that the user has actually had a chance to see the output. This also gives the user a chance to react. askuser then checks to see if anything additional has been typed in the intervening period since it first warned the user in (1). If something has been typed, askuser clears it out and again rings the bell. This latter material, i.e., that typed between the entry to askuser and this point, is discarded and will not be restored since it is not certain whether the user simply reacted quickly to the first warning (bell) and this input is intended for askuser, or whether the user was in the process of typing ahead when the call to askuser occurred, and did not stop typing at the first warning, and therefore this input is a continuation of input intended for another program.

Anything typed after (3) is considered to be intended for askuser, i.e., once the user sees mess or car of mess, he is free to respond. For example, UNDO (Section 22) calls askuser when the

50

The buffers will be restored when askuser completes operation and returns.

number of undosaves are exceeded for an event with mess=(LIST #UNDOSAVES "undosaves, continue saving"). Thus, the user can type a response as soon as the value of #undosaves is typed.

- (4) askuser then types the rest of mess, if any, and
- (5) then goes into a wait loop until something is typed. If wait, the wait time, is not NIL, and nothing is typed in wait seconds, askuser will type "..." and treat the elements of default, the default value, as a list of characters, and begin processing them exactly as though they had been typed. If the user does type anything within wait seconds, he can then wait as long as he likes, i.e., once something has been typed, askuser will not use the default value specified in default.⁵¹

If the calling program knows that the user is expecting an interaction with askuser, e.g. another interaction preceded this one, it can specify in the call to askuser that typeahead is permitted. In this case, askuser simply notes whether there is any typeahead,⁵² then prints mess and goes into a wait loop as described above.

Finally, if the interaction is not with the terminal, i.e., the optional input file/string is specified, askuser simply prints mess and begins reading from the file/string.

OPERATION

All input operations are executed with a terminal table⁵³ in which (1) control[T] has been executed, so that askuser can interact with the user after each character is typed; and (2) echemode[NIL] has been executed, so that askuser can decide *after* it reads a character whether or not the character should be echoed, and with what, e.g. unacceptable inputs are never echoed.

As each character is typed, it is matched against keylst, and appropriate echoing and/or prompting is performed. If the user types an unacceptable character, askuser simply rings the bell and allows him to try again.

At any point, the user can type ?⁵⁴ and receive a list of acceptable responses at that point (generated from keylst), or type a control-a, control-q, control-x, or del,⁵⁵ which causes askuser to reinitialize, and start over.

When an acceptable sequence is completed, askuser returns the indicated value.

⁵¹ If the user wants to consider his response for more than wait seconds, and does not want askuser to default, he can type a carriage return or a space, which are ignored if they are not specified as acceptable inputs by keylst (see below) and they are the first thing typed.

⁵² In this case, if the typeahead turns out to contain unacceptable input, askuser will assume that the typeahead was not intended for askuser, and will restore the typeahead when it completes operation and returns.

⁵³ The name of this terminal table is askusertbl.

⁵⁴ Unless ? itself is an acceptable input, i.e., it matches one of the keys on keylst.

⁵⁵ Unless these characters are acceptable inputs, and assuming del is not an interrupt character so that it is, in fact, seen by askuser.

FORMAT OF KEVLST

keylst is a list of elements of the form (key promptstring . options), where key is an atom or a string (equivalent), promptstring is an atom or a string, and options a list of options in property list format. The following options are recognized and explained below: KEVLST, CONFIRMFLG,

- + PROMPTCONFIRMFLG, NOCASEFLG, RETURN, EXPLAINSTRING, NOECHOFLG, KEYSTRING, PROMPTON, COMPLETEON, AUTOCOMPLETEFLG. If an option is specified in options, the value of the option is the next element. Otherwise, if the option is specified in optionslst (the seventh argument to askuser), its value is the next element on optionslst. Thus, optionslst can be used to provide default options for an entire keylst, rather than having to include the option at each level. If an option does not appear on either options or optionslst, its value is NIL.

For convenience, an entry on keylst of the form (key . atom/string) can be used as an abbreviation for (key atom/string CONFIRMFLG T), and an entry of just the form key, i.e., a non-list, as an abbreviation for (key NIL CONFIRMFLG T).

As each character is read, it is matched against the currently active keys. A character matches a key

- + if it is the same character as that in the corresponding position in the key, or, if the character is an alphabetic character, if the characters are the same without regard for upper/lower case differences,
- + i.e. 'A' matches 'a' and vice versa.⁵⁶ In other words, if two characters have already been input and matched, the third character is matched with each active key by comparing it with the third character of that key. If the character matches with one or more of the keys, the entries on keylst corresponding to the remaining keys are discarded. If the character does not match with any of the keys, the character is not echoed, and a bell is rung instead.

When a key is complete, promptstring is printed (NIL is equivalent to "", the empty string, i.e., nothing will be printed). Then, if the value of the CONFIRMFLG option is T, askuser waits for confirmation of the key by a ⁵⁷ or space. Otherwise, the key does not require confirmation.

Then, if the value of the KEVLST option is not NIL, its value becomes the new keylst, and the process recurses. Otherwise, the key is a "leaf," i.e., it terminates a particular path through the original, top-level keylst, and askuser returns the result of packing all the keys that have been matched and completed along the way (unless the RETURN option is used to specify some other value, as described below).

For example, the following keylst is the default keylst, i.e., is used when askuser is called with keylst=NIL: ((Y "es") (N "o"))

This keylst specifies that if (as soon as) the user types Y (or y), askuser echoes with Y, prompts with "es", and returns Y as its value. Similarly, if the user types N, askuser echoes the N, prompts with "o", and returns N. If the user types ?, askuser prints:

Yes

No

to indicate his possible responses. All other inputs are unacceptable, and askuser will ring the bell and not echo or print anything.

+ ⁵⁶ Unless the NOCASEFLG option (page 17.27) is T.

⁵⁷ ² is used throughout the discussion to denote carriage return.

Here is a more complicated example, the keylst used for the compiler questions (section 18.2):

```
((ST "ore and redefine" KEYLST ("") (F . "orget exprs"))
(S . "ame as last time")
(F . "File only")
(T . "o terminal")
1
2
(Y . "es")
(N . "o"))
```

When askuser is called with this keylst, and the user types an S, two keys are matched: ST and S. The user can then type a T, which matches only the ST key, or confirm the S key by typing a ? or space. If the user confirms the S key, askuser prompts with "ame as last time", and returns S as its value. (Note that the confirming character is not included in the value.) If the user types a T, askuser prompts with "ore and redefine", and makes ("") (F . "orget exprs")) be the new keylst, and waits for more input. The user can then type an F, or confirm the "" (which essentially starts out with all of its characters matched). If he confirms the "", askuser returns ST as its value the result of packing ST and "". If he types F, askuser prompts with "orget exprs", and waits for confirmation again. If the user then confirms, askuser returns STF, the result of packing ST and F.

As mentioned earlier, at any point the user can type a ? and be prompted with the possible responses. For example, if the user types S and then ? askuser will type:

```
STore and redefine Forget exprs
STore and redefine
Same as last time
```

COMPLETING A KEY

The decision about when a key is complete is more complicated than simply whether or not all of its characters have been matched. In the example above, all of the characters in the S key are matched as soon as the S has been typed, but until the next character is typed, askuser does not know whether the S completes the S key, or is simply the first character in the ST key. Therefore, a key is considered to be complete when:

- (1) All of its characters have been matched and it is the only key left, i.e., there are no other keys for which this key is a substring; or
- (2) All of its characters have been matched and a confirming character is typed; or
- (3) All of its characters have been matched, and the value of the CONFIRMF LG option is NIL, and the value of the KEYLST option is not NIL, and the next character matches one of the keys on the value of the KEYLST option; or
- (4) There is only one key left and a confirming character is typed.

Note that if the value of CONFIRMF LG is T, the key still has to be confirmed, regardless of whether or not it is complete. For example, if the first entry in the above example were instead

(ST "ore and redefine" CONFIRMF LG T KEYLST ("") (F . "orget exprs")), and the user wanted to specify the ST path, he would have to type ST, then confirm before typing F, even though the ST completed the ST key by the rule in case (1). However, he would be prompted with "ore and redefine" as soon as he typed the T, and completed the ST key.

Case (2) says that confirmation can be used to complete a key in the case where it is a substring of another key, even where the value of CONFIRMF LG is NIL. In this case, the confirming character doubles as both an indicator that the key is complete, and also to confirm it, if necessary. This situation corresponds to typing S α in the above example.

Case (3) says that if there were another entry whose key was STX in the above example, so that after the user typed ST, two keys, ST and STX, were still active, then typing F would complete the ST key, because F matches the (F . "orget exprs") entry on the value of the KEYLST option of the ST entry. In this case, "ore and redefine" would be printed *before* the F was echoed.

Finally, case (4) says that the user can use confirmation to specify completion when only one key is left, even when all of its characters have not been matched. For example, if the first key in the above example were STORE, the user could type ST and then confirm, and ORE would be echoed, followed by whatever prompting was specified. In this case, the confirming character also confirms the key if necessary, so that no further action is required, even when the value of CONFIRMF LG is T.

- * Case (4) permits the user not to have to type every character in a key when the key is the only one left. Even when there are several active keys, the user can type type \$ (the ESC key, or on some terminals, the key labelled ALT) to specify the next $n > 0$ common characters among the currently active keys. The effect is exactly the same as though these characters had been typed. If there are no common characters in the active keys at that point, i.e. $n=0$, the \$ is treated as an incorrect input, and the bell is rung. For example, if keylst is (CLISPFLG CLISPIFYPACKFLG CLISPITRANFLG), and the user types C followed by \$, askuser will supply the L, I, S, and P. The user can then type F followed by \$ or space to complete and confirm CLISPFLG, as per case (4), or type I, followed by \$, and askuser will supply the F, etc. Note that the characters supplied do not have to correspond to a terminal segment of any of the keys. Note also that the \$ does not confirm the key, although it may complete it in the case that there is only one key active.

If the user types a confirming character when several keys are left, the next $n > 0$ common characters are still supplied, the same as with \$. However, askuser assumes the intent was to complete a key, i.e., case (4) is being invoked. Therefore, after supplying the next n characters, the bell is rung to indicate that the operation was not completed. In other words, typing a confirming character has the same effect as typing an \$ in that the next n common characters are supplied. Then, if there is only one key left, the key is complete (case 4) and confirmation is not required. If the key is not the only key left, the bell is rung.

OPTIONS

KEYLST

When a key is complete, if the value of the KEYLST option is not NIL, this value becomes the new keylst and the process recurses. Otherwise, the key terminates a path through the original, top-level keylst, and askuser returns the indicated value.

CONFIRMF LG

If T, the key must be confirmed with either a α or a space. If the value of CONFIRMF LG is a *list*, the confirming character may be any member of the list.

PROMPTCONFIRMF LG	If T, whenever confirmation is required, the user is prompted with the string " [confirm] ".	+
NOCASEFLG	If T, says do <i>not</i> perform case independent matching on alphabetic characters. If NIL, do perform case independent matching, i.e. 'A' matches with 'a' and vice versa.	+
RETURN	If non-NIL, eval of the value of the RETURN option is returned as the value of <u>askuser</u> . Note that different RETURN options can be specified for different keys. The variable <u>answer</u> is bound in <u>askuser</u> to the list of keys that have been matched. In other words, RETURN (PACK ANSWER) would be equivalent to what <u>askuser</u> normally does.	+
EXPLAINSTRING	If the value of the EXPLAINSTRING option is non-NIL, its value is printed when the user types a ?, rather than key + promptstring. EXPLAINSTRING enables more elaborate explanations in response to a ? than what the user sees when he is prompted as a result of simply completing keys. See example below.	+
NOECHOFLG	If non-NIL, characters that are matched (or automatically supplied as a result of typing \$ or confirming) are not echoed, nor is the confirming character, if any. The value of NOECHOFLG is automatically NIL when <u>askuser</u> is reading from a file or string. The decision about whether or not to echo a character that matches several keys is determined by the value of the NOECHOFLG option for the first key.	+

Example: one of the entries on the keylst used by addtofiles (section 14) is:

```
[ ] "Nowhere?" NOECHOFLG T EXPLAINSTRING "] - nowhere, item is marked as a dummy?"
```

When the user types], askuser just prints "Nowhere?", i.e., the] is not echoed. If the user types ?, the explanation corresponding to this entry will be:

```
] - nowhere, item is marked as a dummy
```

KEYSTRING	If non-NIL, characters that are matched are echoed as though the value of KEYSTRING were used in place of the key. KEYSTRING is also used for computing the value returned. The main reason for this feature is to enable echoing in lowercase.
PROMPTON	If non-NIL, <u>promptstring</u> is printed <i>only</i> when the key is confirmed with a member of the value of PROMPTON. See example below.
COMPLETEON	When a confirming character is typed, the <u>n</u> characters that are automatically supplied, as specified in case (4), are echoed <i>only</i> when the key is confirmed with a member of the value of PROMPTON.

The PROMPTON and COMPLETEON options enable the user to construct a keylst which will cause

askuser to emulate the action of the TENEX exec. The protocol followed by the TENEX exec is that the user can type as many characters as he likes in specifying a command. The command can be completed with a λ or space, in which case no further output is forthcoming, or with a \$, in which case the rest of the characters in the command are echoed, followed by some prompting information. The following keylst would handle the TENEX COPY and CONNECT commands:

```
((COPY " (FILE LIST) " PROMPTON ($) COMPLETEON ($) CONFIRMF LG ($))
(CONNECT " (TO DIRECTORY) " PROMPTON ($) COMPLETEON ($) CONFIRMF LG ($)))
```

AUTOCOMPLETEFLG

If the value of the AUTOCOMPLETEFLG option is not NIL, askuser will automatically supply unambiguous characters whenever it can, i.e., askuser acts as though \$ were typed after each character (except that it does not ring the bell if there are no unambiguous characters).

- + MACROCHARS

value is a list of dotted pairs of form (character . form). When character is typed, and it does not match any of the current keys, form is evaluated and nothing else happens, i.e. the matching process stays where it is. For example, ? could have been implemented using this option. Essentially MACROCHARS provides a read macro facility while inside of askuser (since askuser does readc's, read macros defined via the readtable are never invoked).

- + EXPLAINDELIMITER

value is what is printed to delimit explanation in response to ?. Initially " λ " but can be reset, e.g. to ", ", for more linear output.

SPECIAL KEYS

& can be used as a key to match with any single character, provided the character does not match with some other key at that level. For the purposes of echoing and returning a value, the effect is the same as though the character that were matched actually appeared as the key.

\$ (esc) can be used as a key to match with the result of a single call to read. For example, if the first entry in the TENEX keylst above were:

```
(COPY " (FILE LIST) " PROMPTON ($) COMPLETEON ($) CONFIRMF LG ($)
KEYLST (( $ NIL RETURN ANSWER)))
```

then if the user typed COP{space}FOO λ , (COPY FOO) would be returned as the value of askuser. One advantage of using \$, rather than having the calling program perform the read, is that the call to read from inside askuser is errorset protected, so that the user can back out of this path and reinitialize askuser, e.g. to change from a COPY command to a CONNECT command, simply by typing control-E.

- + \$\$ can be used as a key to match with the result of a single call to readline.
- + A list can be used as a key, in which case the list/form is evaluated and its value "matches" the key. This feature is provided primarily as an escape hatch for including arbitrary input operations as part of an askuser sequence. For example, the effect of \$\$ could be achieved simply by using

(READLINE T) as a key.⁵⁸

"" can be used as a key. Since it has no characters, all of its characters are automatically matched. "" essentially functions as a place marker. For example, one of the entries on the keylst used by addtofiles is:

```
("" "File/list: " EXPLAINSTRING  
"a file name or name of a function list" KEYLST ($))
```

Thus, if the user types a character that does not match any of the other keys on the keylst, then the character completes the "" key, by virtue of case (4), since the character *will* match with the \$ in the inner keylst. askuser then prints "File/list: " before echoing the character, then calls read. The character will be read as part of the read. The value returned by askuser will be the value of the read.

askuser[wait;default;mess;keylst;typeahead;lispexprntflg;optionslst;file]

wait is either NIL or a number (of seconds). default is a single character or a sequence (list) of characters to be used as the default inputs for the case when wait is not NIL and more than wait seconds elapse without any input. In this case, the character(s) from default are processed exactly as though they had been typed, except that askuser first types "...".

mess is the initial message to be printed by askuser, if any, and can be a string, or a list. In the latter case, each element of the list is printed, separated by spaces, and terminated with a " ? ". keylst and optionslst were described earlier. typeahead is T if the user is permitted to typeahead a response to askuser. NIL means any typeahead should be cleared and saved. lispexprntflg determines whether or not the interaction is to be recorded on the history list. file can be either NIL (in which case it is set to T), the name of a file, or a string.⁵⁹ All input operations take place from file until an unacceptable input is encountered, i.e., one that does not conform to the protocol defined by keylst. At that point, file is set to T, default is set to NIL, the input buffer is cleared, and a bell is rung. Unacceptable inputs are not echoed.

The value of askuser is the result of packing all the keys that were matched, unless the RETURN option is specified (page 17.27).

makekeylst[lst;defaultkey;lcaseflg]

lst is a list of atoms or strings. makekeylst returns an askuser keylst which will permit the user to specify one of the elements on lst by

⁵⁸ For \$. \$\$, or a list, if the last character read by the input operation is a separator, the character is treated as a confirming character for the key. However, if the last character is a break character, it will be matched against the next key.

⁵⁹ If file is a string, and all of its characters are read before askuser finishes, file will be reset to T, and the interaction will continue with askuser reading from the terminal.

+ either typing enough characters to make the choice unambiguous,
+ or else typing a number between 1 and n, where n is the length of
+ lst.

+ For example, if askuser is called with keylst=makekeylst[(CONNECT SUPPORT COMPILE)], then
+ the user can type C-O-N, S, C-O-M, 1, 2, or 3 to indicate one of the three choices.

+ If lcaseflg=T, then echoing of upper case elements will be in lower
+ case (but the value returned will still be one of the elements of lst).
+ If defaultkey is non-NIL, it will be the last key on the keylst.
+ Otherwise, a key which permits the user to indicate "No - none of
+ the above" choices, in which case the value returned by askuser will
+ be NIL.

Index for Section 17

	Page Numbers
ADDSPELL[X;SPLST;N]	17.18,20
ADDSPELLFLG (dwim variable/parameter)	17.20,8,18
AMBIGUOUS (printed by dwim)	17.7
approval (of DWIM corrections)	17.3-6,19
APPROVEFLG (dwim variable/parameter)	17.3-6,19,21
ASKUSER[WAIT;DEFAULT;MESS;KEYLST;TYPEAHEAD; LISPXRNTFLG;OPTIONSLST;FILE]	17.22,29,23-28,30
ASKUSERTTBL (dwim variable/parameter)	17.23
AUTOCOMPLETEFLG (askuser option)	17.28
belts (printed by dwim before an interaction) ..	17.4
BREAKCHECK[ERRORPOS;ERXN]	17.10
BREAK1[BRKEXP;BRKWHEN;BRKFN;BRKCOMS;BRKTYPE] NL	17.21
BROKENFNS (break variable/parameter)	17.21
CAUTIOUS (DWIM mode)	17.3,17,21
CHOOZ[XWORD;REL;SPLST;TAIL;FN;TIEFLG;NDBLS;CLST]..	17.16
CLISP	17.11-14
CLISPWORD (property name)	17.12
COMPLETEON (askuser option)	17.27
CONFIRMFLG (askuser option)	17.26
CONTINUE WITH T CLAUSE (printed by dwim)	17.6
control-E	17.4-5,10
DIRECTORIES (system variable/parameter)	17.21
DONTMOVETOPFLG (dwim variable/parameter)	17.7,9
DW (edit command)	17.18
DWIM	17.1-30
DWIM[X]	17.17,3
DWIM interaction with user	17.3
DWIM variables	17.15
DWIMFLG (dwim variable/parameter)	17.20,3
DWIMIFY[X;QUIETFLG;L]	17.17
DWIMLOADFNSFLG (dwim variable/parameter)	17.20,16
DWIMLOADFNS?[]	17.16
DWIMUSERFORMS (dwim variable/parameter)	17.20,11,13-15
DWIMWAIT (dwim variable/parameter)	17.20,4-5
EDITCOMSA (editor variable/parameter)	17.11,14
EDITCOMSL (editor variable/parameter)	17.13-14
EDITDEFAULT	17.3
error correction	17.1-30
ERRORSET[FORM;FLG] SUBR	17.10
EXPLAINDELIMITER (askuser option)	17.28
EXPLAINSTRING (askuser option)	17.27
EXPR (property name)	17.12-13
FASTYPEFLG (dwim variable/parameter)	17.17
FAULTAPPLY[FAULTFN;FAULTARGS]	17.3,10,14
FAULTEVAL[FAULTX] NL*	17.3,10,14
FILEDEF (property name)	17.12,14
FILELST (file package variable/parameter)	17.21
FINDFILE[FILE;NSFLG]	17.21
FIXSPELL[XWORD;REL;SPLST;FLG;TAIL;FN;TIEFLG; DONTMOVETOPFLG;CLST;APPROVALFLG]	17.18,19,21
FIXSPELLDEFAULT (dwim variable/parameter)	17.20,4
FIXSPELLREL (dwim variable/parameter)	17.20,19
FNCHECK[FN;NOERRORFLG;SPELLFLG;PROPFLG;TAIL]	17.20,21
format of keylst (for askuser)	17.24
F/L (as a dwim construct)	17.12
generators for spelling correction	17.9

Page
Numbers

keyboard layouts	17.5, 16
KEYLST (askuser option)	17.26
KEYSTRING (askuser option)	17.27
LAMBDAPLST (dwim variable/parameter)	17.12, 13-14
LASTWORD (dwim variable/parameter)	17.9, 18, 21
LISPX	17.3, 8-9, 21
LPARKEY (dwim variable/parameter)	17.5
MACROCHARS (askuser option)	17.28
macrotran package	17.15
MAKEFILE[FILE;OPTIONS;REPRINTFNS;SOURCEFILE]	17.21
MAKEKEYLST[LST;DEFAULTKEY;LCASEFLG]	17.29
MISSPELLED?[XWORD;REL;SPLST;FLG;TAIL;FN]	17.18, 21
MODEL33FLG (dwim variable/parameter)	17.16
NOCASEFLG (askuser option)	17.27
NOECHOFLG (askuser option)	17.27
NOSPELLFLG (dwim variable/parameter)	17.20
OK TO REEVALUATE (printed by dwim)	17.6
OKREVALST (dwim variable/parameter)	17.6
PRINTBELLS[]	17.4
PROMPTCONFIRMFLG (askuser option)	17.27
PROMPTON (askuser option)	17.27
RETEVAL[POS;FORM;FLG;INTERNALFLG]	17.10
RETURN (askuser option)	17.27
RPARKEY (dwim variable/parameter)	17.5
RUNONFLG (dwim variable/parameter)	17.19-20
run-on spelling corrections	17.19, 4
SHALL I LOAD (system variable/parameter)	17.13
SKOR[XWORD;TWORD;NCX;NCT;FLG]	17.16-17
SPELLFILE[FILE;NOPRINTFLG;NSFLG]	17.21
spelling completion	17.7
spelling correction	17.7, 8-10
spelling correction on file names	17.21
spelling correction protocol	17.4-5
spelling corrector	17.7, 2, 8-10, 16-17
spelling lists	17.8, 9, 12-14
SPELLINGS1 (dwim variable/parameter)	17.8, 9, 14, 18
SPELLINGS2 (dwim variable/parameter)	17.8, 9, 13-14, 18
SPELLINGS3 (dwim variable/parameter)	17.8, 9, 12, 18
synonyms	17.7
SYSLOAD (load option)	17.13
T FIXED (printed by dwim)	17.6
TRUSTING (DWIM mode)	17.3, 2, 17
unbound atom	17.10-14
undefined function	17.10-14
UNDO (prog. asst. command)	17.3
UNSAVED (printed by dwim)	17.12-13
UNSAVEDDEF[NAME;TYPE;DEF]	17.12-13
USERWORDS (dwim variable/parameter)	17.9, 18, 20-21
U.B.A. (error message)	17.10
U.D.F. T FIX? (printed by dwim)	17.5
U.D.F. T (printed by dwim)	17.6
U.D.F. (error message)	17.2, 10
"" (use in askuser)	17.29
#SPELLINGS1 (dwim variable/parameter)	17.9
#SPELLINGS2 (dwim variable/parameter)	17.9
#SPELLINGS3 (dwim variable/parameter)	17.9
#USERWORDS (dwim variable/parameter)	17.9

Page
Numbers

\$ (<esc>) (in spelling correction)	17.7,18
\$ (<esc>, use in askuser)	17.28
& (use in askuser)	17.28
'	17.10
-> (printed by dwim)	17.4,2-3,5
... (printed by dwim)	17.3-4
7 (instead of ')	17.11
8 (instead of left parenthesis)	17.5,1,11,13-14
9 (instead of right parenthesis)	17.5,1,11,13
<esc> (in spelling correction)	17.7,18
= (printed by dwim)	17.4-5
? (printed by dwim)	17.4-5

SECTION 18

THE COMPILER AND ASSEMBLER¹

18.1 THE COMPILER

The compiler is available in the standard Interlisp system. It may be used to compile individual functions as requested or all function definitions in a standard format LOAD file. The resulting code may be stored as it is compiled, so as to be available for immediate use, or it may be written onto a file for subsequent loading.

The most common way to use the compiler is to compile from a symbolic (prettydef) file, producing a corresponding file which contains a set of functions in compiled form which can be quickly loaded. An alternate way of using the compiler is to compile from functions already defined in the user's Interlisp system. In this case, the user has the option of specifying whether the code is to be saved on a file for subsequent loading, or the functions redefined, or both. In either case, the compiler will ask the user certain questions concerning the compilation. The first question is:

LISTING?

The answer to this question controls the generation of a listing and is explained in full below. However, for most applications, the user will want to answer this question with either ST or F, which will also specify an answer to the rest of the questions which would otherwise be asked. ST means the user wants the compiler to STore the new definitions; F means the user is only interested in compiling to a File, and no storing of definitions is performed. In both cases, the compiler will then ask the user one more question:

OUTPUT FILE?

to which the user can answer:

N or NIL	no output file.
Y or YES	user is then asked the name of the file.
File name	file is opened if not already opened, and compiled code is written on the file.

¹ The Interlisp-10 compiler itself, i.e., the part that actually generates code, was written and documented by, and is the responsibility of A.K. Hartley. The user interfaces, i.e., tcompl, recompile, bcompl, and brecompile, were written by W. Teitelman.

Example:

```

←COMPILE((FACT FACT1 FACT2))
LISTING? ST
OUTPUT FILE? FACT.COM
(FACT COMPILED)

.
.
.
(FACT REDEFINED)2
.
.
.
(FACT2 REDEFINED)
(FACT FACT1 FACT2)
←

```

This process caused the functions FACT, FACT1, and FACT2 to be compiled, redefined, and the compiled definitions also written on the file FACT.COM for subsequent loading.

18.2 COMPILER QUESTIONS

The compiler uses the free variables lapflg, strf, svflg, lcfil and lstfil which determines various modes of operation. These variables are set by the answers to the "compset" questions. When any of the top level compiling functions are called, the function compset is called which asks a number of questions. Those that can be answered "yes" or "no" can be answered with YES, Y, or T for YES; and NO, N, or NIL for NO. The questions are:

1. LISTING?

The answer to this question controls the generation of a listing. Possible answers are:

- 1 Prints output of pass 1, the LAP macro code.³
- 2 Prints output of pass 2, the machine code.
- YES Prints output of both passes.
- NO Prints no listings.

The variable lapflg is set to the answer. If the answer is affirmative, compset will type FILE: to allow the user to indicate where the output is to be written. The variable lstfil is set to the answer.

There are three other possible answers to LISTING? - each of which specifies a complete mode for compiling. They are:

- S Same as last setting.
- F Compile to File (no definition of functions).
- ST STore new definitions.
- STF STore new definitions, Forget exprs.

² compiler printout and error messages are explained in Section 18.18, page 18.33-36.

³ The LAP and machine code are usually not of interest but can be helpful in debugging macros.

Implicit in these three are the answers to the questions on disposition of compiled code and expr's, so questions 2 and 3 would not be asked if 1 were answered with S, F, ST, or STF.

2. REDEFINE?

YES Causes each function to be redefined as it is compiled. The compiled code is stored and the function definition changed. The variable strf is set to T.

NO Causes function definitions to remain unchanged. The variable strf is set to NIL.

The answer ST or STF for the first question implies YES for this question, F implies NO, and S makes no change.

3. SAVE EXPRS?

If answered YES, svflg is set to T, and the exprs are saved on the property list of the function name. Otherwise they are discarded. The answer ST for the first question implies YES for this question, F or STF implies NO, and S makes no change.

4. OUTPUT FILE?

If the compiled definitions are to be written for later loading, you should provide the name of a file on which you wish to save the code that is generated. If you answer T or TTY:, the output will be typed on the teletype (not particularly useful). If you answer N, NO, or NIL, output will not be done. If you answer Y or YES, you will be asked the name of the file. If the file named is already open, it will continue to be used. The free variable lcfil is set to the name of the file.

Compiler output and error messages are further documented in Section 18.18.

18.3 COMPIILING NLAMBDA'S

When compiling the call to a function, the compiler must prepare the arguments to the function in one of three ways:

1. Evaluated (SUBR, SUBR*, EXPR, EXPR*, CEXPR, CEXPR*)
2. Unevaluated, spread (FSUBR, FEXPR, CFEXPR)
3. Unevaluated, not spread (FSUBR*, FEXPR*, CFEXPR*)

In attempting to determine which of these three is appropriate, the compiler will first look for a definition among the functions in the file that is being compiled. If the function is not contained there, the compiler will look for other information which can be supplied by the user by including nlambda nospread functions on the list nlama (for nlambda atoms), and including nlambda spread functions on the list nlaml (for nlambda list), and including lambda functions on the list lams.⁴ If the function is not contained in the file,⁵ or on the list nlama, nlaml, or lams, the compiler will

⁴ Including functions on lams is only necessary to override in-core nlambda definitions, since in the absence of other information, the compiler assumes the function is a lambda.

⁵ The function can be defined anywhere in any of the files given as arguments to bcompl, tcompl, bcompile or recompile.

look for a current definition. If the function is defined, its function type is assumed to be the desired type. If it is not defined, the compiler assumes that the function is of type 0 or 1, i.e., its arguments are to be evaluated.⁶⁷ In other words, if there are type 2 or 3 functions called from the functions being compiled, and they are only defined in a separate file, they must be included on nlama or nlaml, or the compiler will incorrectly assume that their arguments are to be evaluated, and compile the calling function correspondingly. Note that this is only necessary if the compiler does not "know" about the function. If the function is defined at compile time, or is handled via a macro, or is contained in the same group of files as the functions that call it, the compiler will automatically handle calls to that function correctly.

18.4 GLOBAL VARIABLES VS SHALLOW BINDING

Variables that appear on the list globalvars or have the property GLOBALVAR, with value T, are called global variables. Such variables are always accessed through their top level value when they are used freely in a compiled function. In other words, a reference to the value of this variable is equivalent to (GETTOPVAL (QUOTE variable)), regardless of whether or not it is bound in the current access chain. Similarly, (SETQ variable value) will compile as (SETTOPVAL (QUOTE variable) value); i.e., it sets the top-level value.

- + Note: Interlisp-10 employs a shallow binding scheme as described in Section 12. There is no distinction between global variables and other types of variables: all variable references are to the variable's value cell. Thus, the cost of *accessing* a variable is small and independent of the depth of computation, whereas in a deep bound system, it can be expensive to search the stack for the most recent binding of a variable, hence the need for a mechanism like global variables. Note however that in a shallow bound system, the cost of rebinding a variable is somewhat higher than in a deep bound system.⁸ We employ shallow binding in Interlisp-10 because measurements indicated more time would be spent in searching for a variable binding using the deep scheme than is lost in rebinding using a shallow scheme. For the purposes of compilation, global variables are treated the same as SPECVARS, i.e. their names are always visible on the stack when they are rebound.

All system parameters, unless otherwise specified, are global variables. Thus, *rebinding* these variables in a deep bound system will not affect the behavior of the system: instead, the variables must be *reset* to their new values, and if they are to be restored to their original values, reset again. For example, the user might write ...(SETQ globalvar new-value) form (SETQ globalvar old-value). Note that in this case, if an error occurred during the evaluation of form, or a control-D was typed,

⁶ Before making this assumption, if the value of compileuserfn is not NIL, the compiler calls (the value of) compileuserfn giving it as arguments cdr of the form and the form itself, i.e., the compiler does (APPLY* COMPILEUSERFN (CDR form) form). If a non-NIL value is returned, it is compiled instead of form. If NIL is returned, the compiler compiles the original expression as a call to a lambda-spread that is not yet defined. CLISP (Section 23) uses compileuserfn to tell the compiler how to compile iterative statements, IF-THEN-ELSE statements, and pattern match constructs. Note that compileuserfn is only called when the compiler encounters a *list car* of which is not the name of a defined function. The user can instruct the compiler about how to compile other data types via completypeclst, page 18.12.

⁷ The names of functions so treated are added to the list alams (for assumed lambdas). alams is not used by the compiler; it is maintained for the user's benefit, i.e., so that the user can check to see whether any incorrect assumptions were made.

+ ⁸ except when the variable is a LOCALVAR.

the global variable would not be restored to its original value. The function resetvar (described in Section 5) provides a convenient way of resetting global variables in such a way that their values are restored even if an error occurred or control-D is typed.

18.5 COMPILER FUNCTIONS

Note: when a function is compiled from its in core definition, i.e., via compile, recompile, or brecompile, as opposed to tcompl or bcompl (which uses the definitions on a file), and the function has been modified by break, trace, breakin, or advise, it is first restored to its original state, and a message printed out, e.g., FOO UNBROKEN. If the function is not defined as an expr, its property list is searched for the property EXPR (see savedef, Section 8). If there is a property EXPR, its value is used for the compilation. If there is no EXPR and the compilation is being performed by recompile or brecompile, the definition of the function is obtained from the file (using loadfns). Otherwise, the compiler prints (fn NOT COMPILEABLE), and goes on to the next function.

compile[x;flg]

x is a list of functions (if atomic, list[x] is used). compile first asks the standard compiler questions, and then compiles each function on x, using its in-core definition. Value is x.

If compiled definitions are being written to a file, the file is closed unless flg=T.

compilel[fn;def;-]

compiles def, redefining fn if strf=T.⁹ compilel is used by compile, tcompl, and recompile. If dwimifycomplflg is T, or def contains a CLISP declaration, def is dwimified before compiling. See Section 23.

tcompl[files]

tcompl is used to "compile files", i.e., given a symbolic load file (e.g., one created by makefile), it produces a "compiled file" that contains the same S-expressions as the original symbolic file, except that (1) a special FILECREATED expression appears at the front of the file which contains information used by the file package, and which causes the message COMPILED ON¹⁰ followed by the date, to be printed when the file is loaded; (2) every defineq in the symbolic file is replaced by the corresponding compiled definitions in the compiled file;¹¹ and (3) expressions of the form (DECLARE: -- DONTCOPY --) that appear in the symbolic file are not copied to the compiled file. This "compiled" file can be loaded into any Interlisp system with load.

⁹ strf is one of the variables set by compset, described earlier.

¹⁰ The actual string printed is the value of compileheader, initially "COMPILED ON". The user can reset compileheader, for example to distinguish between files compiled by different systems.

¹¹ The compiled definitions appear at the front of the compiled file, i.e., before the other expressions in the symbolic file, regardless of where they appear in the symbolic file. The only exceptions are expressions that follow a FIRST tag inside of a DECLARE. See discussion of DECLARE below.

files is a list of symbolic files to be compiled (if atomic, list[files] is used). tcompl asks the standard compiler questions, except for OUTPUT FILE: Instead, the output from the compilation of each symbolic file is written on a file of the same name suffixed with COM,¹² e.g., tcompl[(SYM1 SYM2)] produces two files, SYM1.COM and SYM2.COM.¹³

tcompl processes the files one at a time, reading in the entire file. For each FILECREATED expression, the list of functions that were marked as changed by the file package (see Section 14) is noted,¹⁴ and the FILECREATED expression is written onto the output file. If the rootname of file has property FILETYPE with value CLISP, or value a list containing CLISP, tcompl rebinds dwimifycompflg to T while compiling the functions on file. For each DEFINEQ expression, tcompl adds any NLAMBDA's in the DEFINEQ to nlama or lam1,¹⁵ and adds LAMBDA's to the list lams,¹⁶ so that calls to these functions will be compiled correctly. Expressions beginning with DECLARE: are processed specially as described below. All other expressions are collected to be subsequently written onto the output file. After processing the file in this fashion, tcompl compiles each function,¹⁷ and writes the compiled definition onto the output file. tcompl then writes onto the output file the other expressions found in the symbolic file.

The value of tcompl is a list of the names of the output files. All files are properly terminated and closed. If the compilation of any file is aborted via an error or control-D, all files are properly closed, and the (partially complete) compiled file is deleted.

DECLARE:

For the purposes of compilation, DECLARE: (see Section 14) has two principal applications: (1) to specify forms that are to be evaluated at compile time, presumably to affect the compilation, e.g.,

12 The actual suffix used is the value of the variable compile.ext, which is initially COM. The user can reset compile.ext or rename the compiled file after it has been written, without adversely affecting any of the system packages.

13 The file name is constructed from the name field only, e.g., tcompl[<BOBROW>FOO.TEM;3] produces FOO.COM on the connected directory. The version number will be the standard default.

14 for use by recompile and brecompile which use the same low level functions as tcompl and bcompl.

15 described earlier, page 18.3.

16 nlama, lam1, and lams are rebound to their top level values (using resetvar) by tcompl, recompile, bcompl, brecompile, compile, and blockcompile, so that any additions to these lists while inside of these functions will not propagate outside.

17 except for those functions which appear on the list dontcompilefns, initially NIL. For example, this option might be used for functions that compile open, since their definitions would be superfluous when operating with the compiled file. Note that dontcompilefns can be set via block declarations page 18.20.

to set up macros; and/or (2) to indicate which expressions appearing in the symbolic file are *not* to be copied to the output file. (Normally, expressions are *not* evaluated and *are* copied.) Each expression in cdr of a DECLARE: form is either evaluated/not-evaluated and copied/not-copied depending on the settings of two internal state variables, initially set for copy and not-evaluate. These state variables can be reset for the remainder of the expressions in the DECLARE: by means of the tags DOEVAL@COMPILE (or EVAL@COMPILE), DONTEVAL@COMPILE, DOCOPY (or COPY), and DONTCOPY, e.g., (DECLARE: DOEVAL@COMPILE DONTCOPY (PROP MACRO --)) could be used to set up macros at compile time. The tags EVAL@COMPILEWHEN and COPYWHEN can be used to provide conditional setting of the internal state variables. The expression following the tag is evaluated and determines the setting, e.g. ... EVAL@COMPILEWHEN (EQ (SYSTEMTYPE) 'TOPS20) ... For expressions that are to be copied to the compiled file, the tag FIRST can be used to specify that the subsequent expressions in the DECLARE: are to appear at the front of the compiled file, before anything else except the FILECREATED expressions. The tag NOTFIRST reverses the effect of FIRST. For example, (DECLARE: COPY FIRST (P (PRINT mess1 T)) NOTFIRST (P (PRINT mess2 T))) will cause (PRINT mess1 T) to appear first in the compiled file, followed by any functions, then (PRINT mess2 T). + + + + + + + + + + + + + + + + + + +

Note that the function loadcomp (Section 14) provides a convenient way of obtaining information necessary for compiling one file that is contained in DECLARE: expressions that reside on another file or files. + + + + + + + + + + + + +

RECOMPILE

The purpose of recompile is to allow the user to update a compiled file without recompiling every function in the file. Recompile does this by using the results of a previous compilation. It produces a compiled file similar to one that would have been produced by tcompl, but at a considerable savings in time by compiling selected functions and copying from an earlier tcompl or recompile file the compiled definitions for the remainder of the functions in the file.

recompile[pfile;cfile;fns]

pfile is the name of the pretty file to be compiled, cfile is the name of the compiled file containing compiled definitions that may be copied. fns indicates which functions in pfile are to be recompiled, e.g., have been changed or defined for the first time since cfile was made. Note that pfile, not fns, drives recompile.

recompile asks the standard compiler questions, except for OUTPUT FILE:. As with tcompl, the output automatically goes to pfile.COM.¹⁸ ¹⁹ recompile process pfile the same as does tcompl except that DEFINEQ expressions are not actually read into core. Instead, recompile uses the filemap (see Section 14)²⁰ to obtain a list of the functions contained in pfile, and simply skips over the

¹⁸ or pfile.ext, where ext is the value of compile.ext.

¹⁹ In general, all constructions of the form pfile.COM, pfileCOMS, pfileBLOCKS, etc., are performed using the name field only. For example, if pfile=<BOBROW>FOO.TEM;3, pfile.COM means FOO.COM, pfileCOMS means FOOCOMS, etc.

²⁰ A map is built if the symbolic file does not already contain one, e.g., it was written in an earlier system, or with buildmapflg=NIL.

DEFIN EQ's.²¹

After this initial scan of pfile, recompile then processes the functions defined in the file. For each function in pfile, recompile determines whether or not the function is to be (re)compiled. A function is to be recompiled²² if (1) fns is a list and the function is a member of that list; or (2) fns=T or EXPRS and the function is an expr; or (3) fns=CHANGES and the function is marked as having been changed in the FILECREATED expression; or (4) fns=ALL.²³ If a function is not to be recompiled, recompile obtains its compiled definition from cfile, and copies it (and all generated subfunctions) to the output file, pfile.COM.²⁴ Finally, after processing all functions, recompile writes out all other expressions that were collected in the prescan of pfile.

If cfile=NIL, pfile.COM is used for copying *from*.²⁵ If both fns and cfile are NIL, fns is set to the value of recompiledefault, which is initially EXPRS.²⁶

The value of recompile is the new compiled file, pfile.COM. If recompile is aborted due to an error or control-D, the new (partially complete) compiled file will be closed and deleted.

recompile is designed to allow the user to conveniently and *efficiently* update a compiled file, even when the corresponding symbolic file has not been (completely) loaded. For example, the user can perform a loadfrom (Section 14) to "notice" a symbolic file,²⁷ and then simply edit the functions he wanted to change,²⁸ call makefile,²⁹ and then perform recompile[pfile].³⁰

21 The filemap enables recompile to skip over the DEFIN EQ's in the file by simply resetting the file pointer, so that in most cases the scan of the symbolic file is very fast (the only processing required is the reading of the non-DEFIN EQ's and the processing of the DECLARE: expressions as described earlier).

22 Functions that are members of dontcompilefns are simply ignored.

23 In this latter case, cfile is superfluous, and in fact does not have to exist. This option is useful, for example, to compile a symbolic file that has never been compiled before, but which has already been loaded (since using tcomp would require reading the file in a second time).

24 If the function does not appear on cfile, recompile simply recompiles it.

25 In other words, if cfile, the file used for obtaining compiled definitions to be copied, is NIL, pfile.COM is used, i.e., same name as output file but a different version number (one less) than the output file.

26 This is the most common usage. Typically, the functions the user has changed will have been unsavedefed by the editor, and therefore will be exprs. Thus the user can perform his edits, dump the file, and then simply recompile[file] to update the compiled file.

27 The loadfrom would be unnecessary if the compiled file had been previously loaded, since this would also result in the file having been 'noticed'.

28 As described in Section 9, the editor would automatically load those functions not already loaded.

29 As described in Section 14, makefile would copy the unchanged functions from the symbolic file.

18.6 OPEN FUNCTIONS

When a function is called from a compiled function, a system routine is invoked that sets up the parameter and control push lists as necessary for variable bindings and return information. As a result, function calls can take up to 350 microseconds per call. If the amount of time spent *inside* the function is small, this function calling time will be a significant percentage of the total time required to use the function. Therefore, many "small" functions, e.g., car, cdr, eq, not, cons are always compiled "open", i.e., they do not result in a function call. Other larger functions such as prog, selectq, mapc, etc. are compiled open because they are frequently used. It is useful to know exactly which functions are compiled open in order to determine where a program is spending its time. Therefore below is a list of those functions which when compiled do not result in function calls. Note that the next section tells how the user can make other functions compile open via MACRO definitions.³¹

The following functions compile open³² in Interlisp-10:

AC, ADD1, AND, APPLY*, ARG,³³ ARRAYP, ASSEMBLE, ATOM, BLKAPPLY, BLKAPPLY*, BOUNDP, CAR, CDR, CAAR, ... CDDAR, CDDDDR, CLOSER, COND, CONS, CONSTANT, COROUTINE, DOCOLLECT, ENDCOLLECT, EQ, EQMEMB, ERSETQ, EVALV,³⁴ EVERY, EVQ, FASSOC, FCHARACTER, FDIFFERENCE, FGETD, FGREATERP, FIX, FIXP, FLAST, FLENGTH, FLOAT, FLOATP, FMEMB, FMINUS, FNTH, FPLUS, FQUOTIENT, FRPLACA, FRPLACD, FRPLNODE, FRPLNODE2, FRPTQ, FTIMES, FUNCTION, GENERATOR, GEQ,³⁵ GETATOMVAL, GETFILEPTR, GETHASH, GETPROPLIST, GO, IDIFFERENCE, IEQP, IGEQ, IGREATERP, ILEQ, ILESSP, IMINUS, INTERRUPTABLE, IPLUS, IQUOTIENT, IREMAINDER, ITIMES, JSYS,³⁶ KWOTE, LEQ,³⁷ LISPWATCH, LIST, LISTP, LITATOM, LLSH, LOC, LOGAND, LOGOR, LOGXOR, LRSH, LSH, MAP, MAPC, MAPCAR, MAPCON, MAPCONC, MAPLIST, MINUSP, MKLIST, NCONC1, NEQ, NLISTP, NLSETQ, NOT, NOTEVERY, NOTANY, NTYP, NULL, NUMBERP, OPENR, OR, POSSIBILITIES, PROG, PROG1, PROGN, RESETFORM, RESETLST, RESETSAVE, RESETVAR, RESETVARS, RETURN, RPTQ, RSH, SELECTQ, SET, SETARG, SETATOMVAL, SETN, SETPROPLIST, SETQ, SETQQ, SMALLP, SOME,

³⁰ Since prettydef automatically outputs a suitable DECLARE: expression to indicate which functions in the file (if any) are defined as NLAMBDA's, calls to these functions will be handled correctly, even though the NLAMBDA functions themselves may never be loaded, or even looked at, by recompile.

³¹ The user can also affect the compiled code via compileuserfn, described in footnote on page 18.4, and compiletypefn, described on page 18.12.

³² Some of these compile in-line via macro expansions and some compile so as to use a PUSHJ to jump into system code. The important point from the user's standpoint is that all of them do not require a function call. +

³³ when the arg variable is bound locally. +

³⁴ when given only one argument. +

³⁵ i.e., compiles open as a call to lessp. However, there is still this function call associated with the geq expression. +

³⁶ When the jsys number is itself a small integer, and resultac is either a small number or NIL, e.g. (JSYS 51Q) but not (JSYS N). +

³⁷ See footnote to GEQ above.

STACKP, STRINGP, SUB1, SUBSET, SYNTAXP,³⁸ SYSTEMTYPE, TRYNEXT, TYPENAMEP, TYPEP, UNDONLSETQ, VAG, ZEROP.

18.7 COMPILER MACROS

The Interlisp compiler includes a macro capability by which the user can affect the compiled code. Macros are defined by placing the macro definition on the property list of the corresponding function under the property MACRO.³⁹ When the compiler begins compiling a form, it retrieves a macro definition for car of the form, if any, and uses it to direct the compilation.⁴⁰ The three different types of macro definitions are given below.

(1) Open macros - (LAMBDA ...) or (NLAMBDA ...)

A function can be made to compile open by giving it a macro definition of the form (LAMBDA ...) or (NLAMBDA ...), e.g.,

(LAMBDA (X) (COND ((GREATERP X 0) X) (T (MINUS X)))) for abs. The effect is as if the macro definition were written in place of the function wherever it appears in a function being compiled, i.e., it compiles as an open LAMBDA or NLAMBDA expression. This saves the time necessary to call the function at the price of more compiled code generated.

(2) Computed macros - (atom expression)

A macro definition beginning with an atom other than LAMBDA, NLAMBDA, or NIL, allows computation of the Interlisp expression that is to be compiled in place of the form. The atom which starts the macro definition is bound to cdr of the form being compiled. The expression following the atom is then evaluated, and the result of this evaluation is compiled in place of the form.⁴¹ For example, list could be compiled this way by giving it the macro definition:

-
- + 38 when class is quoted, e.g. (SYNTAXP X (QUOTE BREAK)).
 - + 39 An expression of the form (DECLARE (DECLIST ... (QUOTE MACRO))) can be used *within* a function to define a MACRO. DECLARE is defined the same as QUOTE and thus can be placed so as to have no effect on the running of the function.
 - + 40 The compiler has built into it how to compile certain basic functions such as car, prog, etc., so that these will not be affected by macro definitions. These functions are listed above. However, some of them are themselves implemented via macros, so that the user could change the way they compile.
 - + 41 In Interlisp-10, if the result of the evaluation is the atom INSTRUCTIONS, no code will be generated by the compiler. It is then assumed the evaluation was done for effect and the necessary code, if any, has been added. This is a way of giving direct instructions to the compiler if you understand it. If the result of the evaluation is the atom IGNOREMACRO, the macro is ignored and the compilation of the expression proceeds as if there were no MACRO property. The difference between IGNOREMACRO and INSTRUCTIONS is that if the atom in question is normally treated specially by the compiler, e.g. car, cdr, cond, and, etc., and also has a macro, for those cases which the macro expansion returns IGNOREMACRO, the atom will still be treated specially.

```
[X (LIST (QUOTE CONS)
          (CAR X)
          (AND (CDR X)
                (CONS (QUOTE LIST)
                      (CDR X)))]
```

This would cause (LIST X Y Z) to compile as (CONS X (CONS Y (CONS Z NIL))). Note the recursion in the macro expansion.⁴² Ersetq, nlsetq, map, mapc, mapcar, mapconc, and some, are compiled via macro definitions of this type.

(3) Substitution macro - (NIL expression) or (list expression)

Each argument in the form being compiled is substituted for the corresponding atom in car of the macro definition, and the result of the substitution is compiled instead of the form, i.e., the compiler performs

(SUBPAIR (CAR macrodef) (CDR form) (CADR macrodef)), and compiles the result. For example, the macro definition of add1 is ((X) (IPLUS X 1)). Thus, (ADD1 (CAR Y)) is compiled as (IPLUS (CAR Y) 1). The functions add1, sub1, neq, nlistp, zerop, flength, fmemb, fassoc, flast, and fnth are all compiled open using substitution macros. Note that abs could be compiled open as shown earlier or via a substitution macro. A substitution macro, however, would cause (ABS (FOO X)) to compile as (COND ((GREATERP (FOO X) 0) (FOO X)) (T (MINUS (FOO X)))) and consequently (FOO X) would be evaluated three times.

The following function is quite useful for debugging macro definitions:

`expandmacro[form;quietflg]` takes a form whose car has a macro definition and expands the form as it would be compiled. The result is prettyprinted on the terminal, unless quietflg=T, in which case the result is simply returned.

CONSTANT

The function constant enables the user to define certain expressions as descriptions of their "constant" values. For example, if a user program needed a scratch list of length 30, the user could specify (CONSTANT (to 30 collect NIL)) instead of (QUOTE (NIL NIL ...)). The former is more concise and displays the important parameter much more directly than the latter. constant can also be used to denote values that cannot be quoted directly: (CONSTANT (PACK NIL)), (CONSTANT (ARRAY 10)). It is also useful to parameterize quantities that are constant at run time but may differ at compile time, e.g. (CONSTANT bitsperword) in a program is exactly equivalent to 36, if the variable bitsperword is bound to 36 when the constant expression is evaluated at compile time.

When interpreted, the expression occurring as the argument to constant is evaluated each time it is

⁴² list is actually compiled more efficiently.

- + encountered. If the constant form is compiled, however, the expression will be evaluated only once:
- + If the value of the expression has a readable print-name, then it will be evaluated at compile-time, and the value will be saved as a literal in the compiled function's definition, as if (QUOTE value-of-expression) had appeared instead of (CONSTANT expression).
- + If the value does not have a readable printname (e.g. the PACK and ARRAY examples), then the expression itself will be saved with the function, and it will be evaluated when the function is first executed. The value will then be stored in the function's literals, and will be retrieved on future references.
- + Whereas the function constant attempts to evaluate the expression as soon as possible (compile-time, load-time, or first-run-time), the function deferredconstant will always defer the evaluation until first running. This is useful when the storage for the constant is excessive so that it shouldn't be allocated until (unless) the function is actually invoked.

COMPILETYPELST

- + Most of the compiler's mechanism deals with how to handle forms (lists) and variables (literal atoms). The user can affect the compiler's behaviour with respect to lists and literal atoms in a number of ways, e.g. compiler macros, declarations, compileuserfn, etc. compiletypelst allows the user to tell the compiler what to do when it encounters a data type *other* than a list or an atom. It is the facility in the compiler that corresponds to defeval (Section 8) for the interpreter.
- + compiletypelst is a list of elements of the form (type-name . function). Whenever the compiler encounters a datum that is not a list and not an atom (or a number) in a context where the datum is being evaluated, the type-name of the datum is looked up on compiletypelst, i.e. assoc[typename[datum];compiletypelst] is performed. If an entry appears car of which is equal to the typename, cdr of that entry is applied to the datum. If the value returned by this application is *not eq* to the datum, then that value is compiled instead. If the value *is eq* to the datum, or if there is no entry on compiletypelst for this typename, the compiler simply compiles the datum as (QUOTE datum).

MACROTRAN

- + MACROTRAN is a package that enables the user to run programs interpretively which contain calls to functions which are only defined in terms of a compiler macro. MACROTRAN is implemented via an entry on dwimuserforms (Section 17).⁴³ When the interpreter encounters a form car of which is an undefined function, macrotran is called. If car of the form has a MACRO property, the macro is expanded, and the result of this expansion is evaluated in place of the original form.⁴⁴ If car of the

⁴³ and thus will not work if DWIM is not enabled.

⁴⁴ clisptran (Section 23) is used to save the result of this expansion so that the expansion only has to be done once. On subsequent occasions, the translation (expansion) is retrieved from clisparray the same as for other clisp constructs; macrotran never even has to be invoked.

form is ASSEMBLE (see page 18.24), or the macro contains an ASSEMBLE directive,⁴⁵ instead of the macro being expanded, a dummy function is created with the form as its definition, and the dummy function is then compiled.⁴⁶ ⁴⁷ A form consisting of a call to this dummy function with no arguments is then evaluated in place of the original form.⁴⁸

+
+
+
+

18.8 FUNCTION AND FUNCTIONAL ARGUMENTS

Expressions that begin with FUNCTION are compiled as separate functions⁴⁹ named by attaching a gensym to the end of the name of the function in which they appear, e.g., FOOA0003. This gensym function will be called at run time. Thus if FOO is defined as
 (LAMBDA (X) ... (FOO1 X (FUNCTION ...)) ...) and compiled, then when FOO is run, FOO1 will be called with two arguments, X, and FOOA000n,⁵⁰ and then FOO1 will call FOOA000n each time it must use its functional argument.

Note that a considerable savings in time could be achieved by making FOO1 compile open via a computed macro, e.g.

(Z (LIST (SUBST (CADADR Z) (QUOTE FN) def) (CAR Z)))

where def is the definition of FOO1 as a function of just its first argument and FN is the name used for its functional argument in its definition. The expression compiled contains what was previously the functional argument to FOO1, as an open LAMBDA expression. Thus you save not only the function call to FOO1, but also each of the function calls to its functional argument. For example, if FOO1 operates on a list of length ten, eleven function calls will be saved. Of course, this savings in time costs space, and the user must decide which is more important.

18.9 BLOCK COMPILING

Block compiling provides a way of compiling several functions into a single block. Function calls between the component functions of the block are very fast. Thus, compiling a block consisting of just a single recursive function may be yield great savings if the function calls itself many times, e.g., equal, copy, and count are block compiled in Interlisp-10.

The output of a block compilation is a single, usually large, function. Calls from within the block to functions outside of the block look like regular function calls, except that they are usually linked (described below). A block can be entered via several different functions, called entries. These

⁴⁵ or contains calls to functions which affect the compiler directly, e.g. cexp, storein, etc.

+

⁴⁶ There are some situations for which this procedure is not amenable, e.g. a go inside the form which is being compiled will cause the compiler to give an UNDEFINED TAG error message because it is not compiling the entire function, just a part of it.

+

⁴⁷ Note that macrotran will work on macros that do not contain ASSEMBLE directives even if the compiler is not loaded. ASSEMBLE directives, however, require the compiler.

+

⁴⁸ and clisptran used to save the translation as described above.

+

⁴⁹ except when they are compiled open, as is the case with most of the mapping functions.

⁵⁰ or an appropriate funarg expression, see Section 11.

must be specified when the block is compiled.⁵¹ For example, the error block has three entries, errorx, interrupt, and fault1. Similarly, the compiler block has nine entries.

RETFNS

Another savings in block compilation arises from omitting most of the information on the stack about internal calls between functions in the block. However, if a function's name must be visible on the stack, e.g., if the function is to be returned from retfrom, retto, reteval, etc., it must be included on the list retfns.

BLKAPPLYFNS

Normally, a call to apply from inside a block would be the same as a call to any other function outside of the block. If the first argument to apply turned out to be one of the entries to the block, the block would have to be reentered. blkapplyfns enables a program to compute the name of a function in the block to be called next, without the overhead of leaving the block and reentering it. This is done by including on the list blkapplyfns those functions which will be called in this fashion, and by using blkapply in place of apply, and blkapply* in place of apply*. For example, the calls to the functions handling RI, RO, LI, LO, BI, and BO in the editor are handled this way. If blkapply or blkapply* is given a function not on blkapplyfns, the effect is the same as a call to apply or apply* and no error is generated. Note however, that blkapplyfns must be set at *compile* time, not run time, and furthermore, that all functions on blkapplyfns must be in the block, or an error is generated (at compile time), NOT ON BLKFNS.

BLKLIBRARY

Compiling a function open via a macro provides a way of eliminating a function call. For block compiling, the same effect can be achieved by including the function in the block. A further advantage is that the code for this function will appear only once in the block, whereas when a function is compiled open, its code appears at each place where it is called.

The block library feature provides a convenient way of including functions in a block. It is just a convenience since the user can always achieve the same effect by specifying the function(s) in question as one of the block functions, provided it has an expr definition at compile time. The block library feature simply eliminates the burden of supplying this definition.

To use the block library feature, place the names of the functions of interest on the list blklibrary, and their EXPR definition on the property list of the function under the property BLKLIBRARYDEF. When the block compiler compiles a form, it first checks to see if the function being called is one of the block functions. If not, and the function is on blklibrary, its definition is obtained from the property value of BLKLIBRARYDEF, and it is automatically included as part of the block. The functions assoc, equal, getprop, last, length, lispwatch, memb, member, nconc1, nleft, nth, /rplnode, and tailp already have BLKLIBRARYDEF properties.

⁵¹ Actually the block is entered the same as every other function, i.e., at the top. However, the entry functions call the main block with their name as one of its arguments, and the block dispatches on the name, and jumps to the portion of the block corresponding to that entry point. The effect is thus the same as though there were several different entry points.

18.10 LINKED FUNCTION CALLS

Conventional (non-linked) function calls from a compiled function go through the function definition cell, i.e., the definition of the called function is obtained from its function definition cell at call time. Thus, when the user breaks, advises, or otherwise modifies the definition of the function FOO, every function that subsequently calls it instead calls the modified function. For calls from the system functions, this is clearly *not* a feature. For example, the user may wish to break on basic functions such as `print`, `eval`, `rplaca`, etc., which are used by the break package. In other words, we would like to guarantee that the system packages will survive through user modification (or destruction) of basic functions (unless the user specifically requests that the system packages also be modified). This protection is achieved by linked function calls.

For linked function calls, the definition of the called function is obtained at *link* time, i.e., when the calling function is defined, and stored in the literal table of the calling function. At *call* time, this definition is retrieved from where it was stored in the literal table, *not* from the function definition cell of the called function as it is for non-linked calls. These two different types of calls are illustrated in Figure 18-1.

Note that while function calls from block compiled functions are *usually* linked (i.e. the default for blocks is to link)⁵², and those from standardly compiled functions are *usually* non-linked, linking function calls and blockcompiling are independent features of the Interlisp compiler, i.e., linked function calls are possible, and frequently employed, from standardly compiled functions.

⁵²

In Interlisp-10, linked function calls are actually a little slower and take more space than non-linked calls, so that the user might want to include (`NOLINKFNS . T`) in block declarations to override the default.

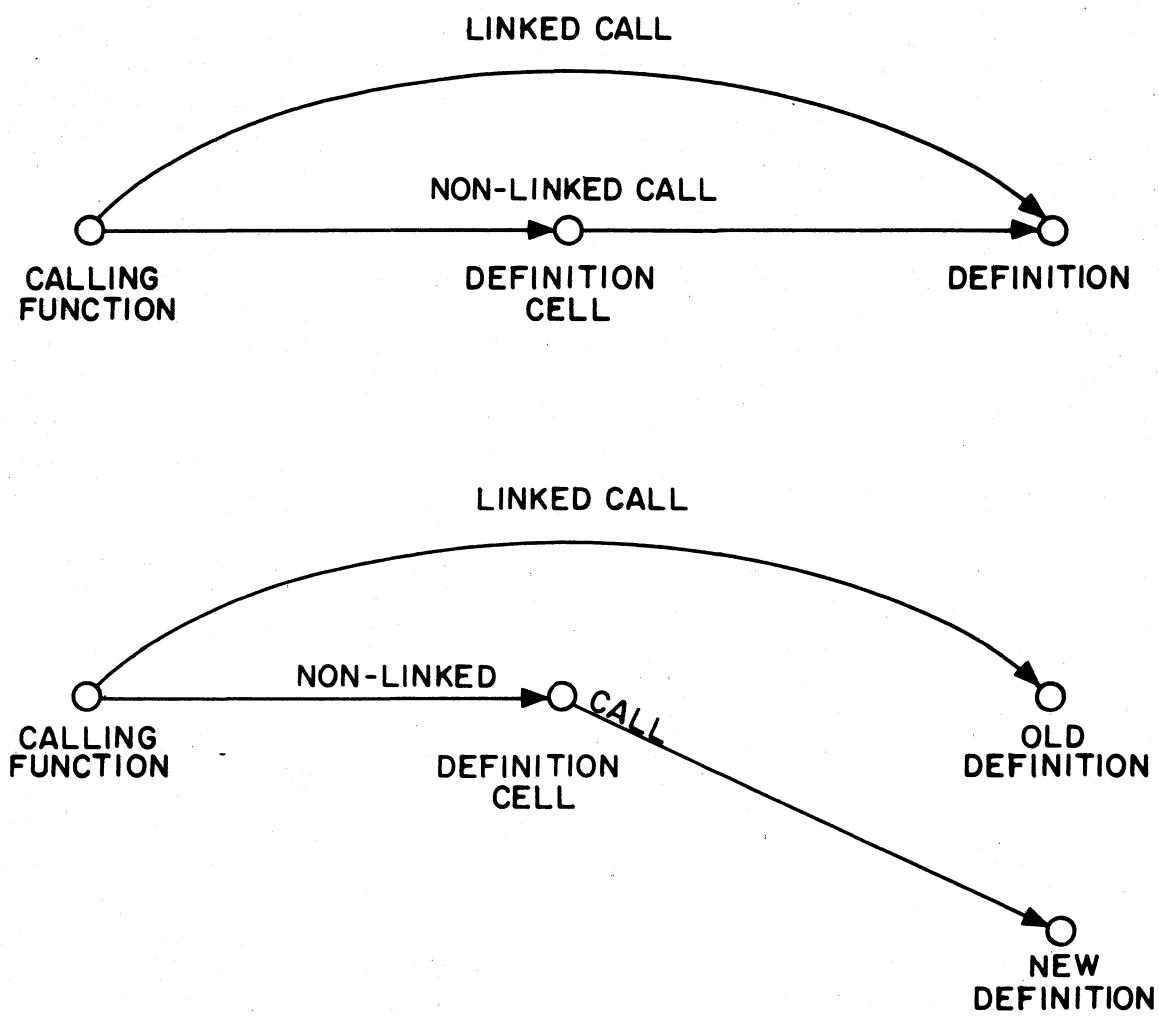


FIGURE 18-1

Note that normal function calls require only the called function's name in the literals of the compiled code, whereas a linked function call uses two literals and hence produces slightly larger compiled functions.

The compiler's decision as to whether to link a particular function call is determined by the variables linkfns and nolinkfns as follows:

- (1) If the function appears on nolinkfns, the call is not linked;
- (2) If block compiling and the function is one of the block functions, the call is internal as described earlier;
- (3) If the function appears on linkfns, the call is linked;
- (4) If nolinkfns=T, the call is not linked;
- (5) If block compiling, the call is linked;
- (6) If linkfns=T, the call is linked;
- (7) Otherwise the call is not linked.

Note that (1) takes precedence over (2), i.e., if a function appears on nolinkfns, the call to it is *not* linked, even if it is one of the functions in the block, i.e., the call will go outside of the block.

Nolinkfns is initialized to various system functions such as errorset, break1, etc. Linkfns is initialized to NIL. Thus if the user does not specify otherwise, all calls from a block compiled function (except for those to functions on nolinkfns) will be linked; all calls from standardly compiled functions will not be linked. However, when compiling system functions such as help, error, arglist, fntyp, break1, et al, linkfns is set to T so that even though these functions are not block compiled, all of their calls will be linked.

If a function is not defined at link time, i.e., when an attempt is made to link to it, it is linked instead to the function nolinkdef. When the function is later defined, the link can be completed by relinking the calling function using relink described below. Otherwise, if a function is run which attempts a linked call that was not completed, nolinkdef is called. If the function is now defined, i.e., it was defined at some point after the attempt was made to link to it, nolinkdef will quietly perform the link and continue the call. Otherwise, it will call faultapply and proceed as described in Section 16.

calls, break on fn1-IN-fn2 and advise fn1-IN-fn2 all work correctly for linked function calls, e.g., break[(FOO IN FIE)], where FOO is called from FIE via a linked function call. Note that control-H will *not* interrupt at linked function calls.

RELINKING

The function relink is available for relinking a compiled function, i.e., updating all of its linked calls so that they use the definition extant at the time of the relink operation.

relink[fn]

fn is either WORLD, the name of a function, a list of functions, or an atom whose value is a list of functions. relink performs the corresponding relinking operations. relink[WORLD] is possible because the compiled code reader maintains on linkedfns a list of all user functions containing any linked calls. syslinkedfns is a list of all system functions that have any linked calls. relink[WORLD] performs both relink[linkedfns] and relink[syslinkedfns].

The value of relink is fn.

It is important to stress that linking takes place when a function is *defined*. Thus, if FOO calls FIE via a linked call, and a bug is found in FIE, changing FIE is not sufficient; FOO must be relinked. Similarly, if FOO1, FOO2, and FOO3 are defined (in that order) in a file, and each call the others via linked calls, when a new version of the file is loaded, FOO1 will be linked to the *old* FOO2 and FOO3, since those definitions will be extant at the time it is read and defined. Similarly, FOO2 will link to the new FOO1 and *old* FOO3. Only FOO3 will link to the new FOO1 and FOO2. The user would have to perform relink[FOOFNS] following the load.

18.11 LOCALVARS AND SPECVARS

In normal compiled and interpreted code, all variable bindings are accessible by lower level functions because the variable's name is associated with its value. We call such variables *special* variables, or specvars. As mentioned earlier, the block compiler normally does *not* associate names with variable values. Such unnamed variables are not accessible from outside the function which binds them and are therefore *local* to that function. We call such unnamed variables local variables, or localvars.

The time economies of local variables can be achieved without block compiling by use of declarations. Using local variables will increase the speed of compiled code; the price is the work of writing the necessary specvar declarations for those variables which need to be accessed from outside the block.

localvars and specvars are variables that affect compilation. During regular compilation, specvars is normally T, and localvars is NIL or a list. This configuration causes all variables bound in the functions being compiled to be treated as special *except* those that appear on localvars. During block compilation, localvars is normally T and specvars is NIL or a list. All variables are then treated as local *except* those that appear on specvars.

Declarations to set localvars and specvars to other values, and therefore affect how variables are treated, may be used at several levels in the compilation process with varying scope. (1) They may be included in the COMS of a file scope of the declaration is then the entire file, e.g.,

(LOCALVARS . T), (SPECVARS X Y).⁵³

(2) The declarations may be included in block declarations; the scope is then the block, e.g.,

(BLOCKS ((FOOBLOCK FOO FIE (SPECVARS . T) (LOCALVARS X))).

(3) The declarations may also appear in individual functions, or in prog's or lambda's within a function. In this case the scope of the declaration is the function or the prog or lambda in which it appears. localvars and specvars declarations must appear immediately after the variable list in the function, prog, or lambda, but intervening comments are permitted. For example:

```
(DEFIN EQ ((FOO
  (LAMBDA (X Y)
    (DECLARE (LOCALVARS Y))
    (PROG (X Y Z)
      (DECLARE (LOCALVARS X)))
```

+ ⁵³ LOCALVARS and SPECVARS are also the names of file package commands, which is why this works. They output the indicated expression, first embedding it inside of a DECLARE: DOEVAL@COMPILE DONTCOPY.

...]

If the above function is compiled (non-block), the outer X will be special, the X bound in the prog will be local, and both bindings of Y will be local.

Declarations for localvars and specvars can be used in two ways: either to cause variables to be treated the same whether the function(s) are block compiled or compiled normally, or to affect one compilation mode while not affecting the default in the other mode. For example:

```
(LAMBDA (X Y)
  (DECLARE (SPECVARS . T))
  (PROG (Z) ... ]
```

will cause X, Y, and Z to be specvars for both block and normal compilation while

```
(LAMBDA (X Y)
  (DECLARE (SPECVARS X))
  ... ]
```

will make X a specvar when block compiling, but when regular compiling the declaration will have no effect, because the default value of specvars would be T, and therefore *both* X and Y will be specvars by default.

Although localvars and specvars declarations have the same form as other components of block declarations such as (LINKFNS . T), their operation is somewhat different because the two variables are not independent. (SPECVARS . T) will cause specvars to be set to T, and localvars to be set to NIL. (SPECVARS V1 V2 ...) will have *no* effect if the value of specvars is T, but if it is a list (or NIL), specvars will be set to the union of its prior value and (V1 V2 ...). The operation of localvars is analogous. Thus, to affect both modes of compilation one of the two (localvars or specvars) must be declared T before specifying a list for the other.

18.12 THE BLOCK COMPILER

There are three user level functions for blockcompiling, blockcompile, bcompl, and brecompile, corresponding to compile, tcompl, and recompile. All of them ultimately call the same low level functions in the compiler, i.e., there is no 'blockcompiler' per se. Instead, when blockcompiling, a flag is set to enable special treatment for specvars, rctfns, blkapplyfns, and for determining whether or not to link a function call. Note that all of the previous remarks on macros, globalvars, compiler messages, etc., all apply equally for block compiling. Using block declarations described below, the user can intermix in a single file functions compiled normally, functions compiled normally with linked calls, and block compiled functions.

BLOCKCOMPILE

blockcompile[blkname;blkfns;entries;flg]

blkfns is a list of the functions comprising the block, blkname is the name of the block, entries a list of entries to the block, e.g.,

←BLOCKCOMPILE(SUBPRBLOCK (SUBPAIR SUBLIS SUBPR) (SUBPAIR SUBLIS))

Each of the entries must also be on blkfns or an error is generated,

NOT ON BLKFNS.⁵⁴

If entries is NIL, list[blkname] is used, e.g.,
 $\leftarrow \text{BLOCKCOMPILE}(\text{COUNT } (\text{COUNT COUNT1}))$

If blkfns is NIL, list[blkname] is used, e.g.,
 $\leftarrow \text{BLOCKCOMPILE}(\text{EQUAL})$

blockcompile asks the standard compiler questions and then begins compiling. As with compile, if the compiled code is being written to a file, the file is closed unless flg=T. The value of blockcompile is a list of the entries, or if entries=NIL, the value is blkname.

The output of a call to blockcompile is one function definition for blkname, plus definitions for each of the functions on entries if any. These entry functions are very short functions which immediately call blkname.

BLOCK DECLARATIONS

Since block compiling a file frequently involves giving the compiler a lot of information about the nature and structure of the compilation, e.g., block functions, entries, specvars, linking, et al, we have implemented a special prettydef command to facilitate this communication. The user includes in the fileCOMS a command of the form (BLOCKS block₁ ... block₂ ... block_n) where each block_i is a block declaration. bcompl and brecompile described below are sensitive to these declarations and take the appropriate action.⁵⁵

The form of a block declaration is:

(blkname blkfn₁ ... blkfn_m (var₁ . value) ... (var_n . value))

blkfn₁ ... blkfn_m are the functions in the block and correspond to blkfns in the call to blockcompile. The (var . value) expressions indicate the settings for variables affecting the compilation.

As an example, the value of editblocks is shown below. It consists of three block declarations, editblock, editfindblock, and edit4e.

⁵⁴ If only one entry is specified, the block name can also be one of the blkfns, e.g., BLOCKCOMPILE(FOO (FOO FIE FUM) (FOO)). However, if more than one entry is specified, an error will be generated, CAN'T BE BOTH AN ENTRY AND THE BLOCK NAME.

+ ⁵⁵ Note: Masterscope (Section 20) includes a facility for checking the block declarations of a file or files for various anomalous conditions, e.g. functions in block declarations which aren't on the file(s), functions in ENTRIES not in the block, variables that may not need to be SPECVARS because they are not used freely below the places they are bound, etc.

```
[RPAQQ EDITBLOCKS
  ((EDITBLOCK EDITL0 EDITL1 UNDOEDITL EDITCOM EDITCOMA EDITCOML
    EDITMAC EDITCOMS EDITJUNDO UNDOEDITCOM
    UNDOEDITCOM1 EDITSMASH EDITNCONC EDIT1F EDIT2F
    EDITNTH BPNT BPNT0 BPNT1 RI RO LI LO BI BO
    EDITDEFAULT ## EDUP EDIT* EDOR EDRPT EDLOC EDLOCL
    EDIT: EDITMBD EDITXTR EDITELT EDITCONT EDITSW
    EDITMV EDITTO EDITBELOW EDITTRAN TAILP EDITSAVE
    EDITH (ENTRIES EDITL0 ## UNDOEDITL)
    (SPECVARS L COM LCFLG #1 #2 #3 LISPXBUFS
      **COMMENT**FLG PRETTYFLG UNDOLST
      UNDOLST1)
    (RETFNS EDITL0)
    (GLOBALVARS EDITCOMSA EDITCOMSL EDITOPS
      HISTORYCOMS EDITTRACEFN)
    (BLKAPPLYFNS RI RO LI LO BI BO EDIT: EDITMBD
      EDITMV EDITXTR)
    (BLKLIBRARY LENGTH NTH LAST)
    (NOLINKFNS EDITTRACEFN))
  (EDITFINDBLOCK EDIT4E EDIT4E1 EDITQF EDIT4F EDITFPAT
    EDITFPAT1 EDIT4F1 EDIT4F2 EDIT4F3 EDITSMASH
    EDITFINDP EDITBF EDITBF1 ESUBST
    (ENTRIES EDITQF EDIT4F EDITFPAT EDITFINDP
      EDITBF ESUBST))
  (EDIT4EBLOCK EDIT4E EDIT4E1 (ENTRIES EDIT4E EDIT4E1)]
```

Whenever bcompl or brecompile encounter a block declaration⁵⁶ they rebind retfns, specvars, localfreevars, globalvars, blklibrary, nolinkfns, linkfns, and dontcompilefns to their top level value, bind blkapplyfns and entries to NIL, and bind blkname to the first element of the declaration. They then scan the rest of the declaration, gathering up all atoms, and setting car of each nonatomic element to cdr of the expression if atomic, e.g., (LINKFNS . T), or else to union of cdr of the expressions with the current (rebound) value,⁵⁷ e.g., (GLOBALVARS EDITCOMSA EDITCOMSL). When the declaration is exhausted, the block compiler is called and given blkname, the list of block functions, and entries.⁵⁸

Note that since all compiler variables are rebound for each block declaration, the declaration only has to set those variables it wants *changed*. Furthermore, setting a variable in one declaration has no effect on the variable's value for another declaration.

After finishing all blocks, bcompl and brecompile treat any functions in the file that did not appear in a block declaration in the same way as do tcompl and recompile. If the user wishes a function compiled separately as well as in a block, or if he wishes to compile some functions (not blockcompile), with some compiler variables changed, he can use a special pseudo-block declaration

⁵⁶ The BLOCKS command outputs a DECLARE: expression, which is noticed by bcompl and brecompile.

⁵⁷ Expressions of the form (var * form) will cause form to be evaluated and the resulting list used as described above, e.g., (GLOBALVARS * MYGLOBALVARS).

⁵⁸ If a function appears in a block declaration, but is not defined in one of the files, then if it has an in-core definition, this definition is used and a message printed NOT ON FILE, COMPILING IN CORE DEFINITION. Otherwise, the message NOT COMPILEABLE, is printed and the block declaration processed as though the function were not on it, i.e. calls to the function will be compiled as external function calls.

of the form (NIL fn₁ ... fn_m (var₁ . value) ... (var_n . value)) which means compile fn₁ ... fn_m after first setting var₁ ... var_n as described above. For example,
(NIL CGETD FNTYP ARGLIST NARGS NCONC1 GENSYM (LINKFNS . T))
appearing as a "block declaration" will cause the six indicated functions to be compiled while linkfns=T so that all of their calls will be linked (except for those functions on nolinkfns).

BCOMPL

bcompl[files;cfile]

files is a list of symbolic files. (If atomic, list[files] is used.) bcompl differs from tcompl in that it compiles all of the files at once, instead of one at a time, in order to permit one block to contain functions in several files.⁵⁹ Output is to cfile if given, otherwise to a file whose name is car[files] suffixed with COM,⁶⁰ e.g., bcompl[(EDIT WEDIT)] produces one file, EDIT.COM.

bcompl asks the standard compiler questions, except for OUTPUT FILE:, then processes each file exactly the same as does tcompl (see page 18.6).⁶¹ ⁶² Bcompl next processes the block declarations as described above. Finally, it compiles those functions not mentioned in one of the block declarations, and then writes out all other expressions.

The value of bcompl is the output file (the new compiled file). If the compilation is aborted due to an error or control-D, all files are closed and the (partially complete) output file is deleted.

Note that it is permissible to tcompl files set up for bcompl; the block declarations will simply have no effect. Similarly, you can bcompl a file that does not contain any block declarations and the result will be the same as having tcompled it.

BRECOMPILE

Brecompile plays the same role for bcompl that recompile plays for tcompl: its purpose is to allow the user to update a compiled file without requiring an entire bcompl.

⁵⁹ Thus if you have several files to be bcompled separately, you must make several calls to bcompl.

⁶⁰ or value of compile.ext, as explained earlier.

⁶¹ In fact, tcompl is defined in terms of bcompl. The only difference is that tcompl calls bcompl with an extra argument specifying that all block declarations are to be ignored.

⁶² If any of the files have property FILETYPE with value CLISP, or a list containing CLISP, then dwimifycomplfg is rebound to T for all of the files.

`brecompile[files;cfile;fns;-]`

files is a list of symbolic files (if atomic, list[files] is used). cfile is the compiled file corresponding to bcompl[files] or a previous brecompile, i.e., it contains compiled definitions that may be copied. The interpretation of fns is the same as with recompile.⁶³

brecompile asks the standard compiler questions except for OUTPUT FILE: As with bcompl, output automatically goes to file.COM, where file is the first file in files.

brecompile processes each file the same as does recompile as described on page 18.7, then processes each block declaration. If any of the functions in the block are to be recompiled, the entire block must be (is) recompiled. Otherwise, the block is copied from cfile as with recompile. For pseudo-block declarations of the form (NIL fn1 ...), all variable assignments are made, but only those functions so indicated by fns are recompiled.

After completing the block declarations, brecompile processes all functions that do not appear in a block declaration, recompiling those dictated by fns, and copying the compiled definitions of the remaining from cfile.

Finally, brecompile writes onto the output file the "other expressions" collected in the initial scan of files.

The value of brecompile is the output file (the new compiled file). If the compilation is aborted due to an error or control-D, all files are closed and the (partially complete) output file is deleted.

If cfile=NIL, file.COM is used.⁶⁴ In addition, if fns and cfile are both NIL, fns is set to the value of recompiledefault, initially EXPRS..

18.13 COMPILER STRUCTURE

The compiler has two principal passes. The first compiles its input into a macro assembly language called LAP.⁶⁵ The second pass expands the LAP code, producing (numerical) machine language instructions. The output of the second pass is written on a file and/or stored in binary program space.

Input to the compiler is usually a standard Interlisp S-expression function definition. However, in

⁶³ In fact, recompile is defined in terms of brecompile. The only difference is that recompile calls brecompile with an extra argument specifying that all block declarations are to be ignored.

⁶⁴ See footnote on page 18.7.

⁶⁵ The exact form of the macro assembly language is extremely implementation dependent, as well as being influenced by the architecture and instruction set for the machine that will run the compiled program. The remainder of Section 18 discusses LAP for the Interlisp-10.

Interlisp-10, machine language coding can be included within a function by the use of one or more assemble forms as described below. In other words, assemble allows the user to write portions of a function in LAP. Note that assemble is only a compiler directive; it has no independent definition. Therefore, functions which use assemble must normally be compiled in order to run.⁶⁶

18.14 ASSEMBLE

Note: assemble is provided for situations where its use is unavoidable. *However, its use is definitely not encouraged.* The disadvantages are several. assemble code is unavoidably dependent on the PDP-10, TENEX, and implementation details of Interlisp-10. Thus, assemble code is not transportable to Interlisp on another machine or operating system, and implementation changes to Interlisp-10 can (and frequently do) require changes to existing assemble code.

The format of assemble is similar to that of PROG: (ASSEMBLE V S₁ S₂ . . . S_N). V is a list of variables to be bound during the first pass of the compilation, *not* during the running of the object code. The assemble statements S₁ . . . S_N are compiled sequentially, each resulting in one or more instructions of object code. When run, the value of the assemble "form" is the contents of AC1 at the end of the execution of the assemble instructions. Note that assemble may appear anywhere in an Interlisp-10 function. For example, one may write:

```
(IGREATERP (IQUOTIENT (LOC (ASSEMBLE NIL
                               (MOVEI 1 , -5)
                               (JSYS 13)))
                           1000)
        4)
```

to test if job runtime exceeds 4 seconds.⁶⁷

ASSEMBLE STATEMENTS

If an assemble statement is an atom, it is treated as a label identifying the location of the next statement that will be assembled.⁶⁸ Such labels defined in an assemble form are like prog labels in that they may be referenced from the current and lower level nested progs or assembles.

If an assemble statement is not an atom, car of the statement must be an atom and one of the following: (1) a number; (2) a LAP op-def (i.e., has a property value OPD); (3) an assembler macro (i.e., has a property value AMAC); or (4) one of the special assemble instructions given below, e.g., C, CQ, etc. Anything else will cause the error message OPCODE? - ASSEMBLE.

The types of assemble statements are described here in the order of priority used in the assemble

+ ⁶⁶ The MACROTRAN package (page 18.12) does permit the user to run programs interpretively which contain assemble directives. Each assemble directive is compiled as a separate function. There is some loss in efficiency over compiling the entire function as a unit, and not all assemble expressions are tractable to this procedure.

⁶⁷ This example is to illustrate use of assemble, and is *not* a recommendation to use the above code. The function jsys (Section 21) is the appropriate method.

⁶⁸ A label can be the last thing in an assemble form, in which case it labels the location of the first instruction *after* the assemble form.

processor; that is, if an atom has both properties OPD and AMAC, the OPD will be used. Similarly a special assemble instruction may be redefined via an AMAC. The following descriptions are of the first pass processing of assemble statements. The second pass processing is described in the section on LAP, page 18.27.

(1) numbers

If car of an assemble statement is a number, the statement is not processed in the first pass. (See page 18.27.)

(2) LAP op-defs

The property OPD is used for two different types of op-defs: PDP-10 machine instructions, and LAP macros. If the OPD definition (i.e., the property value) is a number, the op-def is a machine instruction. When a machine instruction, e.g., HRRZ, appears as car of an assemble statement, the statement is not processed during the first pass but is passed to LAP. The forms and processing of machine instructions by LAP are described on page 18.28.

If the OPD definition is not a number, then the op-def is a LAP macro. When a LAP macro is encountered in an assemble statement, its arguments are evaluated and processing of the statement with evaluated arguments is left for the second pass and LAP. For example, LDV is a LAP macro, and (LDV (QUOTE X) SP) in assemble code results in (LDV X N) in the LAP code, where N is the value of SP.

The form and processing of LAP macros are described on page 18.29.

(3) assemble macros

If car of an assemble statement has a property AMAC, the statement is an assemble macro call. There are two types of assemble macros: lambda and substitution. If car of the macro definition is the atom LAMBDA, the definition will be *applied* to the arguments of the call and the resulting list of statements will be assembled. For example, repeat could be a LAMBDA macro with two arguments, n and m, which expands into n occurrences of m, e.g., (REPEAT 3 (CAR1)) expands to ((CAR1) (CAR1) (CAR1)). The definition (i.e., value of property AMAC) for repeat is:

```
(LAMBDA (N M)
  (PROG (YY)
    A (COND
      ((ILESSP N 1)
       (RETURN (CAR YY)))
      (T (SETQ YY (TCONC YY M))
         (SETQ N (SUB1 N))
         (GO A))))
```

If car of the macro definition is not the atom LAMBDA, it must be a list of dummy symbols. The arguments of the macro call will be substituted for corresponding appearances of the dummy

symbols in cdr of the definition, and the resulting list of statements will be assembled.⁶⁹ For example, abs could be a substitution macro which takes one argument, a number, and expands into instructions to place the absolute value of the number in AC1.

The definition of ABS is:

```
((X)
(CQ (VAG X))
(CAIGE 1 , 0))
(MOVN 1 , 1))
```

(4) special assemble statements

(CQ s₁ s₂ ...) CQ (compile quote) takes any number of arguments which are assumed to be regular S-expressions and are compiled in the normal way. E.g.

```
(CQ (COND ((NULL Y) (SETQ Y 1)))
(SETQ X (IPLUS Y Z)))
```

Note: to avoid confusion and minimize dependence on the current implementation, it is best to have as much of a function as possible compiled in the normal way, e.g., to load the value of x to AC1, (CQ X) is preferred to (LDV (QUOTE X) SP).

(C s₁ s₂ ...) C (compile) takes any number of arguments which are first evaluated, then compiled in the usual way. Both C and CQ permit the inclusion of regular compilation within an assemble form.

(E e₁ e₂ ...) E (evaluate) takes any number of arguments which are evaluated in sequence. For example, (PSTEP) calls a function which increments the compiler variable SP.

(SETQ var) Compiles code to set the variable var to the contents of AC1.

(VAR (op ac , varname)) permits writing a machine instruction with the value of a variable as the operand. Generates the appropriate address and index fields to reference the value of varname. varname may be a locally bound variable, free variable, globalvar, etc. Note that VAR may generate more than one instruction.

(* ...) * is used to indicate a comment; the statement is ignored.

⁶⁹ Note that assemble macros produce a list of statements to be assembled, whereas compiler macros produce a single expression. An assemble macro which *computes* a list of statements begins with LAMBDA and may be either spread or no-spread. The analogous compiler macro begins with an atom, (i.e., is always no-spread) and the LAMBDA is understood.

COREVALS

There are several locations in the basic machine code of Interlisp-10 which may be referenced from compiled code. The current value of each location is stored on the property list under the property COREVAL.⁷⁰ Since these locations may change in different reassemblies of Interlisp-10, they are written symbolically on compiled code files, i.e., the name of the corresponding COREVAL is written, not its value. Some of the COREVALS used frequently in assemble are:

KT	contains (pointer to) atom T
KNIL	contains (pointer to) atom NIL
MKN	routine to box an integer
MKFN	routine to box floating number
IUNBOX	routine to unbox an integer
FUNBOX	routine to unbox floating number

The index registers used for the push-down stack pointers are also included as COREVALS. These are not expected to change, and are not stored symbolically on compiled code files; however, they should be referenced symbolically in assemble code. They are:

PP	parameter stack
CP	control stack
VP	basic frame pointer

18.15 LAP

LAP (for LISP assembly Processor) expands the output of the first pass of compilation to produce numerical machine instructions.

LAP STATEMENTS

If a LAP statement is an atom, it is treated as a label identifying the location of the next statement to be processed. If a LAP statement is not an atom, car of it must be an atom and one of the following: (1) a number; (2) a machine instruction; or (3) a LAP macro.

(1) numbers

If car of a LAP statement is a number, a location containing the number is produced in the object code.

70 The value of corevals is a list of all atoms with COREVAL properties.

e.g., (ADD 1 , A (1)) 71
 .
 .
 .
 A (1)
 (4)
 (9)

Statements of this type are processed like machine instructions, with the initial number serving as a 36-bit op-code.

(2) Machine Instructions

If car of a LAP statement has a numeric value for the property OPD,⁷² the statement is a machine instruction. The general form of a machine instruction is:

(opcode ac , @ address (index))

Opcode is any PDP-10 instruction mnemonic or Interlisp UUO.⁷³

Ac, the accumulator field, is optional. However, if present, it *must* be followed by a comma. Ac is either a number or an atom with a COREVAL property. The low order 4 bits of the number or COREVAL are OR'd to the AC field of the instruction.

@ may be used anywhere in the instruction to specify indirect addressing (bit 13 set in the instruction) e.g., (HRRZ 1 , @ 1 (VP)).

Address is the address field which may be any of the following:

= constant

Reference to an unboxed constant. A location containing the unboxed constant will be created in a region at the end of the function, and the address of the location containing the constant is placed in the address field of the current instruction. The constant may be a number e.g., (CAME 1 , = 3596); an atom with a property COREVAL (in which case the constant is the value of the property, at LOAD time); any other atom which is treated as a label (the constant is then the address of the labeled location) e.g., (MOVE 1 , = TABLE) is equivalent to (MOVEI 1 , TABLE); or an expression whose value is a number.

' pointer

The address is a reference to a Interlisp pointer, e.g., a list, number, string, etc. A location containing the pointer is assembled at the

⁷¹ Note that if a function is intended to be swappable (Section 3), it may not contain any relocatable, indexed instructions.

⁷² The value is an 18 bit quantity (rather than 9), since some UUO's also use the AC field of the instruction.

⁷³ The TENEX JSYS's are not defined, that is, one must write (JSYS 107) instead of (KFORK).

end of the function, and the current instruction will have the address of this location,
 e.g., (HRRZ 1 , ' "IS NOT DEFINED")
 (HRRZ 1 , ' (NOT FOUND))

* Specifies the current location in the compiled function; e.g., (JRST * 2) has the same effect as (SKIPA).

literal atom If the atom has a property COREVAL, it is a reference to a system location, e.g., (SKIPA 1 , KNIL), and the address used is the value of the coreval. Otherwise the atom is a label referencing a location in the LAP code, e.g., (JRST A).

number The number is the address; e.g.,
 (MOVSI 1 , 400000Q)
 (HLRZ 2 , 1 (1))

list The form is evaluated, and its value is the address.

Anything else in the address field causes an error message, e.g., (SKIPA 1 , KNILL) - LAPERROR. A number may follow the address field and will be added to it, e.g., (JRST A 2).

Index is denoted by a *list* following the address field, i.e., the address field *must* be present if an index field is to be used. The index (car of the list) must be either a number, or an atom with a property COREVAL, e.g., (HRRZ 1 , 0 (1)).

(3) LAP macros

If car of a LAP statement is the name of a LAP macro, i.e., has the property OPD, the statement is a macro call. The arguments of the call follow the macro name: e.g., (LQ2 FIE 3).

LAP macro calls comprise most of the output of the first pass of the compiler, and may also be used in assemble. The definitions of these macros are stored on the property list under the property OPD, and like assembler macros, may be either lambda or substitution macros. In the first case, the macro definition is applied to the arguments of the call;⁷⁴ in the second case, the arguments of the call are substituted for occurrences of the dummy symbols in the definition. In both cases, the resulting list of statements is again processed, with macro expansion continuing till the level of machine instructions is reached.

Some examples of LAP macros are shown in Figure 18-2.

⁷⁴ The arguments were already evaluated in the first pass, see page 18.25.

```

(DEFLIST(QUOTE(
  (LQ ((X)
        (HRRZ 1 , ' X)))
  (LQ2 ((X AC)
        (HRRZ AC , ' X)))
  (LDV ((A SP)
        (HRRZ 1 , (VREF A SP))))
  (STV ((A SP)
        (HRRM 1 , (VREF A SP))))
  (LDV2 ((A SP AC)
        (HRRZ AC , (VREF A SP))))
  (LDF ((A SP)
        (HRRZ 1 , (FREF A SP))))
  (STF ((A SP)
        (HRRM 1 , (FREF A SP))))
  (LDF2 ((A SP)
        (HRRZ 2 , (FREF A SP))))
  (CAR1 (NIL
        (HRRZ 1 , 0 (1))))
  (CDR1 (NIL
        (HLRZ 1 , 0 (1))))
  (CAR2 ((AC)
        (HRRZ AC , 0 (AC))))
  (CLL ((NAM N)
        (CCALL N , ' NAM)))
  (LCLL ((NAM N)
        (LNCALL N , (MKLCL NAM))))
  (RET (NIL
        (POPJ CP ,)
  (PUSHP (NIL (PUSH PP , 1)))
  (PUSHQ ((X)
        (PUSH PP , ' X)))
))(QUOTE OPD))
(* LOAD QUOTE TO AC1)
(* LOAD QUOTE TO AC)
(* LOAD LOCAL VARIABLE TO AC1)
(* SET LOCAL VARIABLE FROM AC1)
(* LOAD LOCAL VARIABLE TO AC)
(* LOAD FREE VARIABLE TO AC1)
(* SET FREE VARIABLE FROM AC1)
(* LOAD FREE VARIABLE TO AC)
(* CAR OF AC1 TO AC1)
(* CDR OF AC1 TO AC1)
(* CAR OF AC TO AC)
(* CALL FN WITH N ARGS GIVEN)
(* LINKED CALL WITH N ARGS)
(* RETURN FROM FN)
(* PUSH QUOTE)

```

Figure 18-2

Examples of LAP Macros

18.16 USING ASSEMBLE

In order to use assemble, it is helpful to know the following things about how compiled code is run. All variable bindings and temporary pointers are stored on the parameter pushdown stack (addressed by index register PP). Control information is stored on the control pushdown stack (addressed by index register CP). A function call proceeds as follows:

1. The calling function pushes the argument values on the parameter stack.
2. The calling function invokes a routine that adjusts the number of arguments if too few or too many were supplied, and binds the arguments. Binding usually implies the creation of a basic frame.⁷⁵
3. Then the called function is run.

The arguments in the basic frame are referenced relative to index register VP, e.g., 1(VP) addresses the first argument. However, it is better to reference variables in less implementation dependent ways, such as (CQ ...) or (VAR (. . .)). The compiler will then generate the correct code whether the variable is bound locally, is a free reference, is a globalvar, etc.

The parameter stack may be used for temporary storage of pointers. Both halves of a word on the parameter stack may be pointers. On the control stack the right half of a word must be a pointer, the left a non-pointer. Anything else can cause the garbage collector to fail.

For temporary storage of unboxed numbers, the following assemble macros are provided:

(PUSHN addr)	"Pushes" the number referenced by <u>addr</u> . <u>addr</u> may be any legal <u>assemble</u> code address field, for example: (PUSHN 1), (PUSHN = 0), (PUSHN @ 2)
(POPN addr)	"Pops" the most recent number to <u>addr</u> .
(NREF (op ac , n))	References a previously pushed number. <u>op</u> is the opcode, <u>ac</u> is the accumulator, <u>n</u> is the relative position of the desired number on the pseudo number stack. That is, <u>n</u> = 0 refers to the most recent number, <u>n</u> = -1 to the next most recent, etc. For example: (NREF (MOVN 1, -1))
(PUSHNN n ₁ n ₂ ...)	"Pushes" a sequence of numbers specified by <u>n_i</u> where <u>n_i</u> is a list of any legal address field. For example: (PUSHNN (1) (2) (= 0)) pushes the contents of AC1, the contents of AC2, and the constant 0.
(POPNN n)	"Pops" the <u>n</u> most recent numbers, discarding the values.

⁷⁵ Whether a basic frame is created for a prog or open lambda depends on whether any of the variables are specvars. See page 18.18.

Use of these macros is subject to the following restrictions:

1. PUSHN's and POPN's must be seen by the compiler in the same order and number in which they are executed. The compiler does not analyze the code; it assumes when it encounters a PUSHN in the sequential processing of the code that the PUSHN will in fact be executed.
2. Every number that is pushed must be popped.
3. In nested assemble statements, if a prog or open lambda occurs between the inner and outer level assemble, numbers pushed in the outer assemble may not be referenced from the inner assemble.

18.17 MISCELLANEOUS

The value of a function is always returned in AC1. Therefore, the pseudo-function, ac, is available for obtaining the current contents of AC1. For example (CQ (FOO (AC))) compiles a call to FOO with the current contents of AC1 as argument, and is equivalent to:

```
(PUSHP)
(E (PSTEP))
(PLL (QUOTE FOO) 1)
(E (PSTEPN -1))
```

In using ac, be sure that it appears as the first argument to be evaluated in the expression. For example: (CQ (IPLUS (LOC (AC)) 2))

There are several ways to reference the values of variables in assemble code. For example:

to put value of X in AC1:	(CQ X)
to put value of X in AC3:	(LDV2 (QUOTE X) SP 3)
to set X to contents of AC1:	(SETQ X)
to set X to contents of AC2:	(VAR (HRRM 2 , X))
to box and unbox a number:	
(CQ (LOC (AC))) (FASTCALL MKFN) (CQ (VAG X)) (FASTCALL FUNBOX)	box contents of AC1 floating box contents of AC1 unboxed value of X to AC1 floating unbox of AC1

To call a function directly, the arguments must be pushed on the parameter stack, and SP must be updated, and then the function called: e.g.,

(CQ (CAR X)) (PUSHP) (E (PSTEP)) (PUSHQ 3.14) (E (PSTEP))	(* stack first argument)
	(* stack second argument)

(CLL (QUOTE FUM) 2)
(E (PSTEPN -2))

(* call FUM with 2 arguments)
(* adjust stack count)

and is equivalent to:

(CQ (FUM (CAR X) 3.14))

18.18 COMPILER PRINTOUT AND ERROR MESSAGES

For each function compiled, whether from tcompl, recompile, or compile, the compiler prints:

(fn (arg₁ ... arg_n) (free₁ ... free_n))

The message is printed at the beginning of the second pass of the compilation of fn. (arg₁ ... arg_n) is the list of arguments to fn, and (free₁ ... free_n) the list of free variables referenced or set in fn.⁷⁶ The appearance of non-variables, e.g., function names, words from a comment, etc. in (free₁ ... free_n) is a good indication of parenthesis errors.

If the compilation of fn causes the generation of one or more gensym functions (see page 18.13), a compiler message will be printed for these functions before the message for fn, e.g.,

(FOOA0027 NIL (X))
(FOO (X) NIL)

When compiling a block, the compiler first prints (blkname blkfn1 blkfn2 ...). Then the normal message is printed for the entire block.⁷⁷ Then a message is printed for each *entry* to the block.

In addition to the above output, both recompile and bocompile print the name of each function that is being copied from the old compiled file to the new compiled file. The normal compiler message is printed for each function that is actually compiled.

Compiler output and error messages go to the file coutfile, initially T. coutfile can also be set to the name of a file opened for output, in which case all compiler output will go to coutfile, i.e. the compiler will compile "silently." However, any error messages will be printed to both coutfile as well as T.

+
+
+
+

COMPILER ERROR MESSAGES

Messages describing errors in the function being compiled are also printed on the teletype. These messages are always preceded by *****. Unless otherwise indicated below, the compilation will continue.

(fn NOT ON FILE, COMPILED IN CORE DEFINITION)
from calls to bocompl and bocompile.

+
+

⁷⁶ Does not include global variables, see page 18.4.

⁷⁷ The names of the arguments to the block are generated by suffixing "#" and a number to the block name, e.g., (FOOBLOCK (FOOBLOCK #0 FOOBLOCK #1) free-variables).

(fn NOT COMPILEABLE)

An expr definition for fn could not be found. In this case, no code is produced for fn, and the compiler proceeds to the next function to be compiled, if any.

(fn NOT FOUND)

Occurs when recompile or brecompile try to copy the compiled definition of fn from cfile, and cannot find it. In this case, no code is copied and the compiler proceeds to the next function to be compiled, if any.

(fn NOT ON BLKFNS)

fn was specified as an entry to a block, or else was on blkapplyfns, but did not appear on the blkfns. In this case, no code is produced for the entire block and the compiler proceeds to the next function to be compiled, if any.

(fn CAN'T BE BOTH AN ENTRY AND THE BLOCK NAME)

In this case, no code is produced for the entire block and the compiler proceeds to the next function to be compiled, if any.

(blkname - USED BLKAPPLY WHEN NOT APPLICABLE)

blkapply is used in the block blkname, but there are no blkapplyfns or entries declared for the block.

+ (var SHOULD BE A SPECVAR - USED FREELY BY fn)

In Interlisp-10, while compiling a block, the compiler has already generated code to bind var as a LOCALVAR, but now discovers that fn uses var freely. var should be declared a SPECVAR and the block recompiled.

+ ((* --) COMMENT USED FOR VALUE)

a comment appears in a context where its value is being used, e.g. (LIST X (* --) Y). The compiled function will run, but the value at the point where the comment was used is "undefined."

+ ((form) - NON-ATOMIC CAR OF FORM)

If user intended to treat the value of form as a function, he should use apply*. form is compiled as if apply* had been used. See Section 8.

+ ((SETQ var expr --) -BAD SETQ)

setq of more than two arguments.

(fn - USED AS ARG TO NUMBER FN?)

The value of a predicate, such as GREATERP or EQ, is used as an argument to a function that expects numbers, such as IPLUS.

(fn - NO LONGER INTERPRETED AS FUNCTIONAL ARGUMENT)

The compiler has assumed fn is the name of a function. If the user intended to treat the value of fn as a function, he must use apply*. See Section 8.⁷⁸

(fn - ILLEGAL RETURN)

⁷⁸ This message is printed when fn is not defined, and is also a local variable of the function being compiled. Note that earlier versions of the Interlisp-10 compiler did treat fn as a functional argument, and compiled code to evaluate it.

return encountered when not in prog.

(tg - ILLEGAL GO)

go encountered when not in a prog.

(tg - MULTIPLY DEFINED TAG)

tg is a PROG label that is defined more than once in a single PROG. The second definition is ignored.

(tg - UNDEFINED TAG)

tg is a PROG label that is referenced but not defined in a PROG.

(var - NOT A BINDABLE VARIABLE)

var is NIL, T, or else not a literal atom.

+
+

(var val -- BAD PROG BINDING)

occurs when there is a prog binding of the form (var val₁ ... val_n).

+
+

(tg - MULTIPLY DEFINED TAG, ASSEMBLE)

tg is a label that is defined more than once in an assemble form.

(tg - UNDEFINED TAG, ASSEMBLE)

tg is a label that is referenced but not defined in an ASSEMBLE form.

(tg - MULTIPLY DEFINED TAG, LAP)

tg is a label that was encountered twice during the second pass of the compilation. If this error occurs with no indication of a multiply defined tag during pass one, the tag is in a LAP macro.

(tg - UNDEFINED TAG, LAP)

tg is a label that is referenced during the second pass of compilation and is not defined. LAP treats tg as though it were a coreval, and continues the compilation.

(op - OPCODE? - ASSEMBLE)

op appears as car of an assemble statement, and is illegal. See page 18.24-26 for legal assemble statements.

(NO BINARY CODE GENERATED OR LOADED FOR fn)

a previous error condition was sufficiently serious that binary code for fn cannot be loaded without causing an error.

+
+
+

Index for Section 18

	Page Numbers
AC (in a lap statement)	18.28
AC (in an assemble statement)	18.32
AC1	18.32,24,26
ALAMS (compiler variable/parameter)	18.4
AMAC (property name)	18.24-25
APPLY[FN;ARGS] SUBR	18.14
APPLY*[FN;ARG1;ARG2;...;ARGn] SUBR*	18.14
ASSEMBLE	18.24-27,31-33
ASSEMBLE macros	18.25
ASSEMBLE statements	18.24-26
BAD PROG BINDING (compiler error message)	18.35
BAD SETQ (compiler error message)	18.34
BCOMPL[FILES;CFILE;NOBLOCKSFLG]	18.22,19-21,23
BLKAPPLY[FN;ARGS] SUBR	18.14
BLKAPPLYFNS (compiler variable/parameter)	18.14,19,21
BLKAPPLY*[FN;ARG1;ARG2;...;ARGn] SUBR*	18.14
BLKLIBRARY (compiler variable/parameter)	18.14,21
BLKLIBRARYDEF (property name)	18.14
block compiler	18.19-23
block compiling	18.13-23
block declarations	18.20,21-22
block library	18.14
BLOCKCOMPILE[BLKNAME;BLKFNS;ENTRIES;FLG]	18.19,20
BLOCKS (file package command)	18.20-21
BRECOMPILE[FILES;CFILE;FNS;NOBLOCKSFLG]	18.23,19-22
BUILDMAPFLG (system variable/parameter)	18.7
C (in an assemble statement)	18.26
CAN'T BE BOTH AN ENTRY AND THE BLOCK NAME (compiler error message)	18.34,20
CLISP and compiler	18.4-5
COM (as suffix to file name)	18.6,22
COMMENT USED FOR VALUE (compiler error message) ..	18.34
COMPILE[X;FLG]	18.5
compiled file	18.5,7
COMPILED ON	18.5
COMPILEHEADER (compiler variable/parameter)	18.5
compiler	18.1,2-36
compiler error messages	18.33-36
compiler functions	18.5,6-8,19-20,22-23
compiler macros	18.10-12
compiler printout	18.33
compiler questions	18.2-3
compiler structure	18.23
COMPILETYPELIST (compiler variable/parameter)	18.4,12
COMPILEUSERFN (compiler variable/parameter)	18.4,9
COMPILE.EXT (compiler variable/parameter)	18.6
COMPILE1[FN;DEF;COREFLG]	18.5
compiling by datatype	18.12
compiling CLISP	18.4-5
compiling files	18.5,7,22
compiling FUNCTION	18.13
compiling NLAMBDA _s	18.3-4
COMPSET[FILE;FLG;FILES]	18.2
computed macros	18.10
CONSTANT[X]	18.11
constants in compiled code	18.11

Page
Numbers

control-D	18.4-5
COPYWHEN (DECLARE: option)	18.7
COREVAL (property name)	18.27,28-29
COREVALS	18.27
COREVALS (system variable/parameter)	18.27
COUTFILE (compiler variable/parameter)	18.33
CQ (in an assemble statement)	18.26
DECLARE	18.10,21
DECLARE: (in compiler)	18.6,6
DEFERREDCONSTANT[X]	18.12
DOEVAL@COMPILE (DECLARE: option)	18.7
DONTCOMPILEFNS (compiler variable/parameter)	18.6,8,21
DONTCOPY (DECLARE: option)	18.7
DWIMIFYCOMPFLG (compiler variable/parameter)	18.5,6,22
E (in an assemble statement)	18.26
ENTRIES (compiler variable/parameter)	18.21
entries (to a block)	18.13,19-20
errors in compiler	18.33-36
EVAL@COMPILE (DECLARE: option)	18.7
EVAL@COMPILEWHEN (DECLARE: option)	18.7
EXPANDMACRO[FORM;QUIETFLG]	18.11
EXPR (property name)	18.5,14
F (response to compiler question)	18.1-3
FAULTAPPLY[FAULTFN;FAULTARGS]	18.17
FILECREATED[X] NL*	18.5
FILETYPE (property name)	18.6
FILE: (compiler question)	18.2
FIRST (DECLARE: option)	18.7
FUNARG	18.13
FUNCTION[FN;ENV] NL	18.13
function definition cell	18.15
functional arguments	18.13
GENSYM[CHAR]	18.13
global variables	18.5
GLOBALVAR (property name)	18.4
GLOBALVARS (compiler variable/parameter)	18.4,21
IGNOREMACRO (in compiler)	18.10
ILLEGAL GO (compiler error message)	18.35
ILLEGAL RETURN (compiler error message)	18.34
INSTRUCTIONS (in compiler)	18.10
LAMS (compiler variable/parameter)	18.3,6
LAP	18.27,2,23,28-29
LAP macros	18.29,25
LAP op-defs	18.24-25
LAP statements	18.27-29
LAPFLG (compiler variable/parameter)	18.2
LCFIL (compiler variable/parameter)	18.2-3
linked function calls	18.15-18
LINKEDFNS (system variable/parameter)	18.17
LINKFNS (compiler variable/parameter)	18.17,21-22
LISTING? (compiler question)	18.1-2
LOAD[FILE;LDFLG;PRINTFLOG]	18.5
LOADFROM[FILE;FNS;LDFLG]	18.8
LOCALFREEVARS (compiler variable/parameter)	18.21
LOCALVARS (compiler variable/parameter)	18.18
LSTFIL (compiler variable/parameter)	18.2
machine instructions	18.27,28-29

	Page Numbers
MACRO (property name)	18.10,9,12
macros (in compiler)	18.10-12
macrotran package	18.12,24
MAKEFILE[FILE;OPTIONS;REPRINTFNS;SOURCEFILE]	18.8
MULTIPLY DEFINED TAG (compiler error message)	18.35
MULTIPLY DEFINED TAG, ASSEMBLE (compiler error message)	18.35
MULTIPLY DEFINED TAG, LAP (compiler error message)	18.35
NIL (use in block declarations)	18.22
NLAMA (compiler variable/parameter)	18.3
NLAML (compiler variable/parameter)	18.3
NO BINARY CODE GENERATED OR LOADED (compiler error message)	18.35
NO LONGER INTERPRETED AS FUNCTIONAL ARGUMENT (compiler error message)	18.34
NOLINKDEF	18.17
NOLINKFNS (compiler variable/parameter)	18.17,21-22
NON-ATOMIC CAR OF FORM (compiler error message) ..	18.34
NOT A BINDABLE VARIABLE (compiler error message) ..	18.35
NOT COMPILEABLE (compiler error message)	18.34,5,21
NOT FOUND (compiler error message)	18.34
NOT IN FILE - USING DEFINITION IN CORE (compiler error message)	18.33
NOT ON BLKFNS (compiler error message)	18.34,14,20
NOT ON FILE, COMPILING IN CORE DEFINITION (compiler error message)	18.21
NOTFIRST (DECLARE: option)	18.7
OPCODE (in a lap statement)	18.28
OPCODE? - ASSEMBLE (compiler error message)	18.35,24
OPD (property name)	18.25,24,28-29
open functions	18.9-10
open macros	18.10
OUTPUT FILE? (compiler question)	18.1,3
PSTEP[]	18.26
PSTEPN[N]	18.32
RECOMPILE[PFILE;CFILE;FNS]	18.7,5,8,21
RECOMPILEDEFAULT (compiler variable/parameter) ..	18.8,23
REDEFINE? (compiler question)	18.3
RELINK[FN;UNLINKFLG]	18.17
relinking	18.17-18
RESETVAR[VAR;NEWVALUE;FORM] NL	18.5
RETFNS (compiler variable/parameter)	18.14,19,21
S (response to compiler question)	18.2-3
SAVE EXPRS? (compiler question)	18.3
second pass (of the compiler)	18.23
SETQ (in an assemble statement)	18.26
shallow binding	18.4
SHOULD BE A SPECVAR (compiler error message)	18.34
SP (in an assemble statement)	18.26
SPECVARS (compiler variable/parameter)	18.18,19,21
ST (response to compiler question)	18.1-3
STF (response to compiler question)	18.2-3
STRF (compiler variable/parameter)	18.2,3,5
substitution macros	18.11
SVFLG (compiler variable/parameter)	18.2-3
SYSLINKDFNS (system variable/parameter)	18.17
TCOMPL[FILES]	18.5,6-7,21-22

Page
Numbers

UNBROKEN (typed by compiler)	18.5
UNDEFINED TAG (compiler error message)	18.35
UNDEFINED TAG, ASSEMBLE (compiler error message) ..	18.35
UNDEFINED TAG, LAP (compiler error message)	18.35
USED AS ARG TO NUMBER FN? (compiler error message)	18.34
USED BLKAPPLY WHEN NOT APPLICABLE (compiler error message)	18.34
WORLD (as argument to RELINK)	18.17
' (in a lap statement)	18.28
* (in a lap statement)	18.29
* (in an assemble statement)	18.26
***** (in compiler error messages)	18.33
= (in a lap statement)	18.28
@ (in a lap statement)	18.28

SECTION 19

ADVISING¹

The operation of advising gives the user a way of modifying a function without necessarily knowing how the function works or even what it does. Advising consists of modifying the *interface* between functions as opposed to modifying the function definition itself, as in editing. break, trace, and breakdown, are examples of the use of this technique: they each modify user functions by placing relevant computations *between* the function and the rest of the programming environment.

The principal advantage of advising, aside from its convenience, is that it allows the user to treat functions, his or someone else's, as "black boxes," and to modify them without concern for their contents or details of operations. For example, the user could modify sysout to set sysdate to the time and date of creation by `advise[SYSOUT; (SETQ SYSDATE (DATE))]`

As with break, advising works equally well on compiled and interpreted functions. Similarly, it is possible to effect a modification which only operates when a function is called from some other specified function, i.e., to modify the interface between two particular functions, instead of the interface between one function and the rest of the world. This latter feature is especially useful for changing the *internal* workings of a system function.

For example, suppose the user wanted time (Section 21) to print the results of his measurements to the file FOO instead of the teletype. He could accomplish this by `ADVISE(((PRIN1 PRINT SPACES) IN TIME) BEFORE (SETQQ U FOO))`

Note that advising prin1, print, or spaces directly would have affected all calls to these very frequently used function, whereas advising `((PRIN1 PRINT SPACES) IN TIME)` affects just those calls to prin1, print, and spaces from time.

Advice can also be specified to operate after a function has been evaluated. The value of the body of the original function can be obtained from the variable !value, as with break1. For example, suppose the user wanted to perform some computation following each sysin, e.g., check whether his files were up to date. He could then:

```
ADVISE(SYSOUT AFTER (COND ((LISTP !VALUE) --)))2
```

¹ Advising was developed and implemented by W. Teitelman.

² After the sysin, the system will be as it was when the sysout was performed, hence the advice must be to sysout, not sysin. See Section 14 for complete discussion of sysout/sysin.

19.1 IMPLEMENTATION OF ADVISING

The structure of a function after it has been modified several times by advise is given in the following diagram:

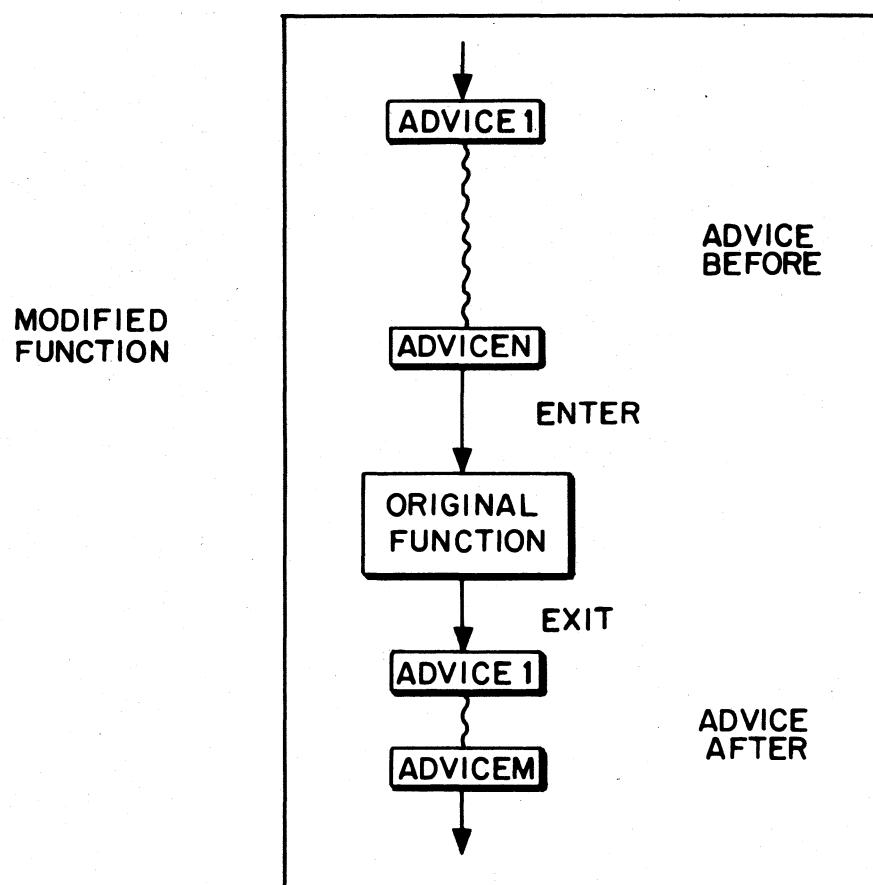


FIGURE 19-1

The corresponding Interlisp definition is:

```
(LAMBDA arguments (PROG (!VALUE)
  (SETQ !VALUE (PROG NIL
    advice1
    .
    .
    :       advice
    .
    before
    .
    advice1
    (RETURN body)))
  advice1
  .
  .
  :       advice
  .
  after
  .
  advice1
  (RETURN !VALUE)))
```

where body is equivalent to the original definition.^{3 4}

Note that the structure of a function modified by advise allows a piece of advice to bypass the original definition by using the function RETURN. For example, if (COND ((ATOM X) (RETURN Y))) were one of the pieces of advice BEFORE a function, and this function was entered with x atomic, y would be returned as the value of the inner PROG, !value would be set to y, and control passed to the advice, if any, to be executed AFTER the function. If this same piece of advice appeared AFTER the function, y would be returned as the value of the entire advised function.

The advice (COND ((ATOM X) (SETQ !VALUE Y))) AFTER the function would have a similar effect, but the rest of the advice AFTER the function would still be executed.

19.2 ADVISE FUNCTIONS

ADVISE

Advise is a function of four arguments: fn, when, where, and what. fn is the function to be modified by advising, what is the modification, or piece of advice. when is either BEFORE, AFTER, or AROUND, and indicates whether the advice is to operate BEFORE, AFTER, or AROUND the body of the function definition. where specifies exactly where in the list of advice the new advice is to be placed, e.g., FIRST, or (BEFORE PRINT) meaning before the advice containing print, or (AFTER 3) meaning after the third piece of advice, or even (: TTY:). If where is specified, advise first checks to see if it is one of LAST, BOTTOM, END, FIRST, or TOP, and operates accordingly. Otherwise, it constructs an appropriate edit command and calls the editor to insert the advice at the corresponding location.

Both when and where are optional arguments, in the sense that they can be omitted in the call to advise. In other words, advise can be thought of as a function of two arguments [fn;what], or a

³ Actually, advise uses its own versions of PROG, SETQ, and RETURN, (called ADV-PROG, ADV-SETQ, and ADV-RETURN) in order to enable advising these functions.

⁴ If fn was originally an EXPR, body is the body of the definition, otherwise a form using a gensym which is defined with the original definition.

function of three arguments: [fn;when;what], or a function of four arguments: [fn;when;where;what]. Note that the advice is always the *last* argument. If when=NIL, BEFORE is used. If where=NIL, LAST is used.

advise[fn;when;where;what] fn is the function to be advised, when=BEFORE, AFTER, or AROUND, where specifies where in the advice list the advice is to be inserted, and what is the piece of advice.

If fn is of the form (fn1 IN fn2), fn1 is changed to fn1-IN-fn2 throughout fn2, as with break, and then fn1-IN-fn2 is used in place of fn.⁵

If fn is broken, it is unbroken before advising.

If fn is not defined, an error is generated, NOT A FUNCTION.

If fn is being advised for the first time, i.e., if getp[name, ADVISED]=NIL, a gensym is generated and stored on the property list of fn under the property ADVISED, and the gensym is defined with the original definition of fn. An appropriate S-expression definition is then created for fn.⁶ Finally, fn is added to the (front of) advisedfns.⁷

If fn has been advised before, it is moved to the front of advisedfns.

If when=BEFORE or AFTER, the advice is inserted in fn's definition either BEFORE or AFTER the original body of the function. Within that context, its position is determined by where. If where=LAST, BOTTOM, END, or NIL, the advice is added following all other advice, if any. If where=FIRST or TOP, the advice is inserted as the first piece of advice. Otherwise, where is treated as a command for the editor, a la breakin, e.g., (BEFORE 3), (AFTER PRINT).

If when=AROUND, the body is substituted for * in the advice, and the result becomes the new body, e.g., advise[FOO;AROUND;(RESETFORM (OUTPUT T) *)]. Note that if several pieces of AROUND advice are specified, earlier ones will be embedded inside later ones. The value of where is ignored.

Finally list[when;where;what] is added (by addprop) to the value of

⁵ If fn1 and/or fn2 are lists, they are distributed as shown in the example on page 19.1.

⁶ Using private versions of PROG, SETQ, and RETURN, so that these functions can also be advised.

⁷ So that unadvise[T] always unadvises the last function advised. See page 19.5.

property ADVICE on the property list fn.⁸ Note that this property value is a list of the advice in order of calls to advise, not necessarily in order of appearance of the advice in the definition of fn.

The value of advise is fn.

If fn is non-atomic, every function in fn is advised with the same values (but copies) for when, where, and what. In this case, the value of advise is a list of individual functions.

Note: advised functions can be broken. (However if a function is broken at the time it is advised, it is first unbroken.) Similarly, advised functions can be edited, including their advice. unadvise will still restore the function to its unadvised state, but any changes to the body of the definition will survive. Since the advice stored on the property list is the same structure as the advice inserted in the function, editing of advice can be performed on either the function's definition or its property list.

unadvise[x]

is a no-spread NLAMBDA a la unbreak. It takes an indefinite number of functions and restores them to their original unadvised state, including removing the properties added by advise.⁹ unadvise saves on the list advinfofolst enough information to allow restoring a function to its advised state using readvise. advinfofolst and readvise thus correspond to brkinfolst and rebreak.

unadvise[] unadvises all functions on advisedfns.¹⁰ It first sets advinfofolst to NIL.

unadvise[T] unadvises the first function of advisedfns, i.e., the most recently advised function.

readvise[x]

is a no-spread NLAMBDA a la rebreak for restoring a function to its advised state without having to specify all the advise information. For each function on x, readvise retrieves the advise information either from the property READVICE for that function, or from advinfofolst, and performs the corresponding advise operation(s). In addition it stores this information on the property READVICE if not already there. If no information is found for a particular function, value is (fn - NO ADVICE SAVED).

readvise[]

readvises everything on advinfofolst.

⁸ So that a record of all the changes is available for subsequent use in readvising, see page 19.5.

⁹ Except if a function also contains the property READVICE (see readvise below), unadvise moves the current value of the property ADVICE to READVICE.

¹⁰ In reverse order, so that the most recently advised function is unadvised last.

readadvise[T] readvises just the first function on advinfolst, i.e., the function most recently unadvised.

A difference between advise, unadvise, and readadvise versus break, unbreak, and rebreak, is that if a function is not rebroken between successive unbreak[]'s, its break information is forgotten. However, once readvised, a function's advice is permanently saved on its property list (under READADVICE); subsequent calls to unadvise will not remove it. In fact, calls to unadvise update the property READADVICE with the current value of the property ADVICE, so that the sequence readadvise, advise, unadvise causes the augmented advice to become permanent. Note that the sequence readadvise, advise, readadvise removes the "intermediate advice" by restoring the function to its earlier state.

advisedump[x;flg]

Used by prettydef when given a command of the form (ADVISE --) or (ADVICE --). flg=T corresponds to (ADVISE --), i.e., advisedump writes both a deflist and a readadvise. flg= NIL corresponds to (ADVICE --), i.e., only the deflist is written. In either case, advisedump copies the advise information to the property READADVICE, thereby making it "permanent" as described above.

Index for Section 19

Page
Numbers

advice	19.1,3
ADVICE (file package command)	19.6
ADVICE (property name)	19.5-6
ADVINFOLST (system variable/parameter)	19.5
ADVISE[FN;WHEN;WHERE;WHAT]	19.4,3,5
ADVISE (file package command)	19.6
ADVISED (property name)	19.4
ADVISED FNS (system variable/parameter)	19.4-5
ADVISEDUMP[X;FLG]	19.6
advising	19.1-6
ADV-PROG	19.3-4
ADV-RETURN	19.3-4
ADV-SETQ	19.3-4
AFTER (as argument to advise)	19.3-4,1
AROUND (as argument to advise)	19.3,4
BEFORE (as argument to advise)	19.3,4
BOTTOM (as argument to advise)	19.3-4
FIRST (as argument to advise)	19.3-4
(fn1 IN fn2)	19.4
fn1-IN-fn2	19.4
GENSYM[CHAR]	19.3-4
LAST (as argument to advise)	19.3-4
NOT A FUNCTION (error message)	19.4
PRETTYDEF	19.6
READVICE (property name)	19.5-6
READVISE[X] NL*	19.5,6
TOP (as argument to advise)	19.3-4
UNADVISE[X] NL*	19.5,4,6
UNBROKEN (typed by advise)	19.4
!VALUE (use with advising)	19.1,3

SECTION 20

MASTERSCOPE AND HELPSYS

20.1 MASTERSCOPE¹

Masterscope is an interactive program for analyzing and cross referencing user programs. It contains facilities for analyzing user functions to determine what other functions are called, how and where variables are bound, set, or referenced, as well as which functions use particular record declarations. Masterscope is able to analyze definitions directly from a file as well as in-core definitions.

Masterscope maintains a database of the results of the analyses it performs. Via a simple command language, the user may interrogate the database, call the editor on those expressions in functions that were analyzed which use variables or functions in a particular way, or display the tree structure of function calls among any set of functions.

Masterscope is interfaced with the editor and file package so that when a function is edited or a new definition loaded in, Masterscope knows that it must re-analyze that function.

The following sample session illustrates some of these facilities. User input is underlined.

¹ Masterscope was designed and implemented by L. M. Masinter.

```

←MASTERSCOPE] [1]
Masterscope 1-JAN-76 ... type HELP<cr> for command summary.
←. ANALYZE FUNCTIONS ON RECORDS [2]
.....  

NIL  

←. WHO CALLS RECFIELDLOOK? [3]
(RECFIELDLOOK ACCESSDEF ACCESSDEF2 EDITREC)
←. EDIT WHERE ANY CALL RECFIELDLOOK? [4]
RECFIELDLOOK :
(RECFIELDLOOK (CDR Y) FIELD)
tty:  

*OK?  

ACCESSDEF :
(RECFIELDLOOK DECLST FIELD VAR1)
*OK?  

(RECFIELDLOOK USERRECLST FIELD)
*N VAR1?
*OK?  

ACCESSDEF2 :
(RECFIELDLOOK (RECORD.SUBDECS TRAN) FIELD)
tty:  

(RECFIELDLOOK (RECORD.SUBDECS TRAN) FIELD)
*N (CAR TAIL)
*OK?  

EDITREC :
(RECFIELDLOOK USERRECLST (CAR EDITRECX))
*OK?  

NIL  

←. WHO CALLS ERROR? [5]
..  

(EDITREC)  

←. SHOW PATHS TO RECFIELDLOOK FROM ACCESSDEF? [6]
(inverted tree)
1. RECFIELDLOOK RECFIELDLOOK
2. ACCESSDEF
3. ACCESSDEF2 ACCESSDEF2
4. ACCESSDEF
5. RECORDCHAIN ACCESSDEF
NIL  

←. WHO CALLS WHO IN /FNS? [7]
RECORDSTATEMENT -- /RPLNODE
RECORDECL1 -- /NCONC, /RPLACD, /RPLNODE
RECREDECLARE1 -- /PUTHASH
UNCLISPTRAN -- /PUTHASH, /RPLNODE2
RECORDWORD -- /RPLACA
RECORD1 -- /RPLACA, /SETTOPVAL
EDITREC -- /SETTOPVAL
←. OK? [8]

```

Figure 20-1

Sample Masterscope Session

- [1] The user calls masterscope directly. Masterscope prints a greeting and prompts with ". ". Within the top-level executive of Masterscope, the user may issue Masterscope commands, programmer's assistant commands, (e.g., REDO, FIX), or run programs.²
- [2] The user then directs that the functions on file RECORD should be analyzed. Masterscope always prints a . when it (re)analyzes a function, just to let the user know what is happening.
- [3] The user asks which functions call recfieldlook. Masterscope responds with the list.
- [4] The user asks to edit the expressions where the function recfieldlook is called. Masterscope calls editf on the functions it had analyzed that call recfieldlook, directing the editor to the appropriate expressions. The user then edits some of those expressions.
- [5] Next the user asks which functions call error. Since some of the functions in the database have been changed, Masterscope re-analyzes the changed definitions (and prints out .'s for each function it analyzes).³ Masterscope responds that editrec is the only analyzed function that calls error.
- [6] The user asks to see a map of the ways in which recfieldlook is called from accessdef. A tree structure of the calls is displayed.
- [7] The user then asks to see which functions call which functions in the list /fns. Masterscope responds with a structured printout of these relations.
- [8] Finally, the user exits from the Masterscope executive with an OK.

20.1.1 COMMAND LANGUAGE

The user communicates with Masterscope via an English-like command language. Through these commands, the user can direct that functions should be analyzed, interrogate Masterscope's database, and perform other operations. The commands deal with sets of functions, variables, etc., and relations between them (e.g., call, bind). Sets correspond to English nouns, relations to verbs.

A set of atoms can be specified in a variety of ways, either *explicitly*, e.g., FUNCTIONS ON FIE specifies the atoms in filefnslst[FIE], or *implicitly*, e.g., NOT CALLING Y, where the meaning must be determined in the context of the rest of the command. Such sets of atoms are the basic building blocks which the command language deals with.

Masterscope also deals with relations *between* sets. For example, the relation CALL relates functions

- 2 Note that the user may also enter Masterscope commands directly from the top-level of Interlisp (or from within a break or in the editor, etc.) merely by preceding the command with a period and a space. The atom . is defined as a nlambda nospread function which interprets its argument as a Masterscope command, executes the command and returns.
- 3 The feedback when Masterscope analyzes a function is controlled by the flag msprintflg: if msprintflg is the atom "...", Masterscope will print out a period. (If an error in the function is detected, a ? is printed instead.) If msprintflg is a number, Masterscope will print the name of the function it is analyzing every nth function. If msprintflg is NIL, Masterscope won't print anything. Initial setting is "...". Note that the function name is printed when Masterscope starts analyzing, and the comma is printed when it finishes.

* + * + * +

and other functions; the relations BIND and USE FREELY relate functions and variables. The relations are what are stored in the Masterscope database when functions are analyzed. In addition, Masterscope "knows" about file package conventions; CONTAIN relates files and various types of objects (functions, variables).

Sets and relations are used (along with a few additional words) to form sentence-like *commands*. For example, the command WHO ON 'FOO USE 'X FREELY will print out the list of functions contained in the file FOO which use the variable X freely. The command EDIT WHERE ANY CALL 'ERROR will call editf on those functions which have previously been analyzed that directly call error, pointing at each successive expression where the call to error actually occurs.

RELATIONS

A relation is specified by one of the keywords below. Some of these "verbs" accept modifiers. For example, USE, SET, SMASH and REFERENCE all may be modified by FREELY. The modifier may occur anywhere within the command.⁴ Verbs can occur in the present tense (e.g., USE, CALLS, BINDS, USES) or as present or past participles (e.g., CALLING, BOUND, TESTED). The relations (with their modifiers) recognized by Masterscope are:

CALL	Function F1 calls F2 if the definition of F1 contains a form (F2 --), (APPLY (QUOTE F2) --), (FUNCTION F2), etc.
CALL SOMEHOW	One function calls another SOMEHOW if there is some path from the first to the other. That is, if F1 calls F2, and F2 calls F3, then F1 CALLS F3 SOMEHOW. ⁵
USE	If unmodified, the relation USE denotes variable usage in any way; it is the union of the relations SET, SMASH, TEST, and REFERENCE.
SET	A function SETS a variable if the function contains a form (SETQ var --), (SETQQ var --), etc.
SMASH	A function SMASHES a variable if the function calls a destructive list operation (<u>rplaca</u> , <u>rplacd</u> , <u>dremove</u> , <u>sort</u> , etc.) on the value of that variable. Masterscope will also find instances where the operation is performed on a "part" of the value of the variable; for example, if a function contains a form (RPLACA (NTH X 3) T) it will be noted as SMASHING X. ⁶

⁴ If there is more than one verb, any modifier *between* two verbs is assumed to modify the first one. For example, in USING ANY FREELY OR SETTING X, the FREELY modifies USING but not SETTING-- the entire phrase is interpreted as the set of all functions which either use any variable freely or set the variable X, whether or not X is bound in the place where it is set.

⁵ This information is not stored directly in the database; instead, Masterscope stores only information about direct function calls, and (re)computes the CALL SOMEHOW relation as necessary.

⁶ Note, however, that if the function contains a sequence (SETQ Y X) (RPLACA Y T) then Y is noted as being smashed, but not X.

TEST A variable is TESTED by a function if its value is only distinguished between NIL and non-NIL. For example, the form (COND ((AND X --) --)) tests the value of X.

REFERENCE This relation includes all variable usage except for SET.

The verbs USE, SET, SMASH, TEST and REFERENCE may be modified by the words FREELY or LOCALLY. A variable is used FREELY if it is not bound in the function at the place of its use; alternatively, it is used LOCALLY if the use occurs within a PROG or LAMBDA that binds the variable.

Masterscope also distinguishes between CALL DIRECTLY and CALL INDIRECTLY. A function is called DIRECTLY if it occurs as car-of-form in a normal evaluation context. A function is called INDIRECTLY if its name appears in a context which does not imply its immediate evaluation, for example (SETQ Y (LIST (FUNCTION FOO) 3))⁷. In addition, CALL FOR EFFECT (where the value of the function is not used) is distinguished from CALL FOR VALUE..

BIND	The BIND relation between functions and variables includes both variables bound as arguments and those bound in an internal <u>prog</u> or <u>lambda</u> .	*
USE AS A FIELD	Masterscope notes all uses of record field names within <u>fetch</u> , <u>replace</u> or <u>create</u> expressions.	*
FETCH	Use of a field within a <u>fetch</u> expression.	+
REPLACE	Use of a record field name within a <u>replace</u> or <u>create</u> expression.	+
USE AS A RECORD	Masterscope notes all uses of record names within <u>create</u> or <u>type?</u> expressions. ⁸	*
CREATE	Use of a record name within a <u>create</u> expression.	+
USE AS A PROPERTY NAME	Masterscope notes the property names used in <u>getprop</u> , <u>putprop</u> , <u>getlist</u> , etc. expressions if the name is quoted. E.g. if a function contains a form (GETPROP X (QUOTE INTERP)), then that function USES INTERP AS A PROPERTY NAME.	*
USE AS A CLISP WORD	Masterscope notes all iterative statement operators and user defined clisp words as being used as a clisp word.	+
CONTAIN	Files contain functions, records, and variables. This relation is not stored in the database but is computed using the file package.	+

⁷ The distinction is whether or not the compiled code of the caller would contain a direct call to the callee. Note that an occurrence of (FUNCTION FOO) as the functional argument to one of the built-in mapping functions which compile open is considered to be a direct call.

⁸ Additionally, in X:FOO.FIE, FOO is used as a record name.

DECLARE AS LOCALVAR
DECLARE AS SPECVAR

Masterscope notes internal "calls" to DECLARE from within functions.

Note: Masterscope uses "templates" for functions, as described on page 20.16, to decide which relations hold. For example, the information that sort SMASHes its first argument is contained in the template for SORT. Masterscope initially contains templates for most system functions which set variables, test their arguments, or perform destructive operations. The user may change existing templates or insert new ones in Masterscope's tables via the settemplate function, described on page 20.18.

The following abbreviations are recognized: FREE=FREELY, LOCAL=LOCALLY, PROP=PROPERTY, REF=REFERENCE. Also, the words A, AN and NAME (after AS) are "noise" words and may be omitted.

SETS

A "set" is a collection of things (functions, variables, etc.). A set is specified by a set phrase, consisting of a *determiner* (e.g., ANY, WHICH, WHO) followed by a *type* (e.g., FUNCTIONS, VARIABLES) followed by a *specification* (e.g., IN MYFNS, @ SUBRP). The determiner, type and specification may be used alone or in combination. For example, ANY FUNCTIONS IN MYFNS, ANY @ SUBRP, VARIABLES IN GLOBALVARS, and WHO are all acceptable set phrases. Set specifications, types and determiners are explained below:

Set specifications

'atom	The simplest way to specify a set consisting of a single thing is by the name of that thing. For example, in the command WHO CALLS 'ERROR, the function <u>error</u> is referred to by its name. Although the ' can be left out, to resolve possible ambiguities names should usually be quoted; e.g., WHO CALLS 'CALLS will return the list of functions which call <u>calls</u> .
'list	Sets consisting of several atoms may also be referred to by naming the atoms; e.g. the command WHO USES '(A B) will call the editor on all expressions where the variables A or B are used freely.
IN expression	Similarly, the user can give a LISP expression to be <u>evaluated</u> , and have the value treated as a (list of) the elements of a set. For example, IN GLOBALVARS specifies the list of variables in the value of <u>globalvars</u> .
@ predicate	A set may also be specified by giving a predicate which the elements of that set must satisfy. <u>predicate</u> is either a function name, a LAMBDA expression, or an expression in terms of the variable X. The specification @ <u>predicate</u> represents all atom for which the value of <u>predicate</u> is non-NIL. For example, @ EXPRP specifies all those atoms which have expr definitions; @ (STRPOS1 X CLISPCHARRAY) specifies those atoms which contain clisp characters. The universe to be searched is either

determined by the context within the command (e.g., in WHO IN FOOFNS CALLS ANY NOT @ GETD, the predicate is only applied to functions which are called by any functions in the list FOOFNS), or in the extreme case, the universe defaults to the entire set of things which have been noticed by Masterscope, as in the command WHO IS @ EXPRP.

LIKE atom

atom may contain alt-modes; it is used as a pattern to be matched (as in the editor). For example, WHO LIKE /R\$ IS CALLED BY ANY would find both /RPLACA and /RPLNODE.

+
+
+

A set may also be specified by giving a relation its members must have with the members of another set:

relationING set

relationING is used here generically to mean any of the relation words in the present participle form (possibly with a modifier), e.g., USING, SETTING, CALLING, BINDING. relationING set specifies the set of all objects which have that relation with some element of set. For example, CALLED BY X specifies the set of functions called by the function X; USING ANY IN FOOVARS FREELY specifies the set of functions which uses freely any variable in the value of FOOVARS.

relationED BY set

This is similar to the relationING construction. For example, CALLED BY ANY IN FOOFNS represents the set of functions which are called by any element of FOOFNS; USED FREELY BY ANY CALLING ERROR is the set of variables which are used freely by any function which also calls the function error. IN may be used instead of BY, i.e., relationED IN set is allowed.

+
+

Note: sets may also be specified with "relative clauses" introduced by the word THAT, e.g. THE FUNCTIONS THAT BIND 'X.

blocktype OF functions

These phrases allow the user to ask about BLOCKS declarations on files (see Section 18). blocktype is one of LOCALVARS, SPECVARS, GLOBALVARS, ENTRIES, BLKFNS, BLKAPPLYFNS, or RETFNS. These phrases denote the names which are declared to be blocktype in any blocks declaration which contain the any of functions (a "set" of functions). The function can either be the block name or just one of the functions in the block. In the second construct, all names which are declared to be blocktype on any of the given files (a "set" of files) are denoted. For example, WHICH ENTRIES OF ANY CALLING 'Y BIND ANY GLOBALVARS ON 'FOO.

+
+
+
+
+
+
+
+
+
+
+
+
+

FIELDS OF set

set is a set of records. This denotes the field names of those records.

+
+

KNOWN	The set of all functions which have been analyzed. For example, the command WHO IS KNOWN will print out the list of functions which have been analyzed.
* THOSE	The set of things printed out by the last Masterscope question. For example, following the command WHO IS USED FREELY BY PARSE, the user could ask WHO BINDS THOSE to find out where those variables are bound.
ON PATH pathoptions	Refers to the set of functions which <i>would be</i> printed by the command SHOW PATHS <u>pathoptions</u> . For example, IS FOO BOUND BY ANY ON PATH TO 'PARSE tests if FOO might be bound "above" the function PARSE. <u>pathoptions</u> are explained in detail on page 20.15.

Determiners

Set phrases may be preceded by a *determiner*. A determiner is one of the words THE, ANY, WHO or WHICH. The "question" determiners (WHO and WHICH) are only meaningful in some of the commands, namely those that take the form of questions. ANY and WHO (or WHOM) can be used alone; they are "wild-card" elements, e.g., the command WHO USES ANY FREELY, will print out the names of all (known) functions which use any variable freely. If the determiner is omitted, ANY is assumed; e.g. the command WHO CALLS '(PRINT PRIN1 PRIN2) will print the list of functions which call *any* of print, prin1, prin2. THE is also allowed, e.g. WHO USES THE RECORD + FIELD FIELDX.

Types

A set phrase may also have a *type*; that is, a set may contain either functions, variables, files, record names, record field names or property names. The type is used by Masterscope in a variety of ways when interpreting the set phrase:

- (1) Set types are used to disambiguate possible parsings. For example, both commands WHO SETS ANY BOUND IN X OR USED BY Y and WHO SETS ANY BOUND IN X OR CALLED BY Y have the same general form. However, the first case is parsed as WHO SETS ANY (BOUND BY X OR USED BY Y) since both BOUND BY X and USED BY Y refer to variables; while the second case as WHO SETS ANY BOUND IN (X OR CALLED BY Y), since CALLED BY Y and X must refer to functions.⁹
- (2) The type is used to determine the modifier for USE: FOO USES WHICH RECORDS is equivalent to FOO USES WHO AS A RECORD FIELD.
- (3) The interpretation of CONTAIN depends on the type of its object: the command WHAT FUNCTIONS ARE CONTAINED IN MYFILE prints the list of functions in MYFILE; WHAT RECORDS ARE ON MYFILE prints the list of records.

⁹ Note that parentheses may be used to group phrases.

- (4) The implicit "universe" in which a set expression is interpreted depends on the type: ANY VARIABLES @ GETD is interpreted as the set of all variables which have been noticed by Masterscope (i.e., bound or used in any function which has been analyzed) that also have a definition. ANY FUNCTIONS @ (NEQ (GETTOPVAL X) 'NOBIND) is interpreted as the set of all functions which have been noticed (either analyzed or called by a function which has been analyzed) that also have a top-level value.

The type may be determined by the context within the command (e.g., in CALLED BY ANY ON FOO, the set ANY ON FOO is interpreted as meaning the *functions* on FOO since only functions can be CALLED), or the type may be given explicitly by the user (e.g., FUNCTIONS ON FILE). The following types are recognized: FUNCTIONS, VARIABLES, FILES, PROPERTY NAMES, RECORDS, FIELDS, I.S.OPRS.¹⁰

*
*
*

CONJUNCTIONS

Sets may be joined by the conjunctions AND and OR or preceded by NOT to form new sets. AND is always interpreted as meaning "intersection"; OR as "union", while NOT means "complement". For example, the set CALLING X AND NOT CALLED BY Y specifies the set of all functions which call the function X but are not called by Y.

*
*
*
*

Masterscope's interpretation of AND and OR follow LISP conventions rather than the conventional English interpretation. For example "calling X and Y" would, in English, be interpreted as the intersection of (CALLING X) and (CALLING Y); but Masterscope interprets CALLING X AND Y as CALLING ('X AND 'Y); which is the null set. Only sets may be joined with conjunctions: joining modifiers, as in USING X AS A RECORD FIELD OR PROPERTY NAME, is not allowed; in this case, the user must say USING X AS A RECORD FIELD OR USING X AS A PROPERTY NAME

As described above, the type of sets is used to disambiguate parsings. The algorithm used is to first try to match the type of the phrases being joined and then try to join with the longest preceding phrase. In any case, the user may group phrases with parentheses to specify the manner in which conjunctions should be parsed.

COMMANDS

The normal mode of communication with Masterscope is via "commands". These are sentences in the Masterscope command language which direct Masterscope to answer questions or perform various operations. The syntax of Masterscope commands is described below:

ANALYZE set

Analyze the functions in set (and any functions called by them) and include the information gathered in the database.¹¹

¹⁰ or abbreviations FNS, VARS, PROPNAMEs or the singular forms FUNCTION, FN, VARIABLE, VAR, FILE, PROPNAMe, RECORD, FIELD. Note that most of these types correspond to built-in "file package types". (See Section 14).

*
*
*

¹¹ Masterscope will not re-analyzing a function if it thinks it already has valid information about that function in its database. The user may use the command REANALYZE set to force re-analysis. For example, this would be necessary if the user had disabled or subverted the file package, e.g. performed putd's to change the definition of functions.

Note that whenever a function is referred to in a command as a "subject" of one of the relations, it is automatically analyzed; the user need not give an explicit ANALYZE command. Thus, WHO IN MYFNS CALLS FIE will automatically analyze the functions in MYFNS if they have not already been analyzed.

Note also that only expr definitions will be analyzed; that is, Masterscope will not analyze compiled code. If there is no in-core definition for a function (either in the definition cell or an EXPR property), Masterscope will attempt to read in the definition from a file.¹² If necessary, the definition will be dwimified before analysis.

ERASE set Erase all information about the functions in set from the database. ERASE by itself clears the entire database.

SHOW PATHS pathoptions Displays a tree of function calls. pathoptions are described on page 20.15.

set relation set
set IS|ARE set

This command has the same format as an English sentence with a subject (the first set), a verb (the relation or IS or ARE), and an object (the second set). Any of the sets within the command may be preceded by the question determiners WHICH or WHO (or just WHO alone). For example, WHICH FUNCTIONS CALL X prints the list of functions that call the function X. relation may be one of the relation words in present tense (CALL, BIND, TEST, SMASH, etc.) or used as a passive (e.g., WHO IS CALLED BY WHO).¹³

The interpretation of the command depends on the number of question elements present:

- (1) If there is *no* question element, the command is treated as an assertion and Masterscope returns either T or NIL, depending on whether that assertion is true. Thus, ANY IN MYFNS CALL HELP will print T if any function in MYFNS call the function help, and NIL otherwise.
- (2) If there is *one* question element, Masterscope returns the list of items for which the assertion would be true. For example MYFN BINDS WHO USED FREELY BY YOURFN prints the list of variables bound by MYFN which are also used freely by YOURFN.
- (3) If there are *two* question elements, Masterscope will print a doubly indexed list:

←. WHO CALLS WHO IN /FNS ↗

¹² Files which have been explicitly mentioned previously in some command are searched first. If the definition cannot be found on any of those files, Masterscope looks among the files on filelst for a definition. If a function is found in this manner, Masterscope will print a message "(reading from filename)". If no definition can be found at all, Masterscope will print a message "fn can't be analyzed". If the function previously was known, the message "fn disappeared!" is printed.

¹³ Other variants are allowed, e.g. WHO DOES X CALL, IS FOO CALLED BY FIE, etc.

RECORDSTATEMENT --	/RPLNODE
RECORDECL1 --	/NCONC, /RPLACD, /RPLNODE
RECREDECLARE1 --	/PUTHASH
UNCLISPTRAN --	/PUTHASH, /RPLNODE2
RECORDWORD --	/RPLACA
RECORD1 --	/RPLACA, /SETTOPVAL
EDITREC --	/SETTOPVAL

*
*
*
*
*
*
*

EDIT WHERE set relation set [- editcoms]

(WHERE may be omitted.) The first set refers to a set of functions. The EDIT command calls the editor on each expression where the relation actually occurs.¹⁴ For example, EDIT WHERE ANY CALL ERROR will call editf on each (analyzed) function which calls error stopping within a TTY: at each call to error. editcoms, if given, are a list of commands passed to editf to be performed at each expression. For example, EDIT WHERE ANY CALLS MYFN DIRECTLY - (SW 2 3) P will switch the first and second arguments to MYFN in every call to MYFN and print the result. EDIT WHERE ANY ON MYFILE CALL ANY NOT @ GETD will call the editor on any expression involving a call to an undefined function. Note that EDIT WHERE X SETS Y will point only at those expressions where Y is actually set, and will skip over places where Y is otherwise mentioned.

*

SHOW WHERE set relation set

Like the EDIT command except merely prints out the expressions without calling the editor.

EDIT set [- editcoms]

calls editf on each function in set. editcoms, if given, will be passed as a list of editor commands to be executed. For example EDIT ANY CALLING FN1 - (R FN1 FN2) will replace FN1 by FN2 in those functions that call FN1.

+
+
+
+

DESCRIBE set

Prints out the BIND, USE FREELY and CALL information about the functions in set. For example, the command DESCRIBE PRINTARGS might print out:

```
PRINTARGS[N;FLG]
binds   : TEM,LST,X
calls   : MSRECORDFILE,SPACES,PRIN1
called by: PRINTSENTENCE,MSHELP,CHECKER
```

showing that printargs has two arguments, n and flg, binds internally the variables tem, lst and x, calls msrecordfile, spaces and prin1 and is called by printsentence, mshelp, and checker.

CHECK set

checks for various anomolous conditions (mainly in the compiler declarations) for the files in set (if set is not given, filelst is used). For example, this command will warn about variables which are

+
+
+

¹⁴ Currently one cannot EDIT WHERE a file CONTAINS a datum, nor where one function CALLS another SOMEHOW.

+ bound but never referenced, functions in BLOCKS declarations which aren't on the file containing the declaration, variables declared as LOCALFREEVARS but which are used freely in contexts where they are not bound, functions declared as ENTRIES but not in the block, variables which may not need to be SPECVARS because they are not used freely below the places where they are bound, etc.

+ FOR variable set i.s.tail

This command provides a way of combining CLISP iterative statements with Masterscope. An iterative statement will be constructed in which var is iteratively assigned to each element of set, and then the iterative statement tail i.s.tail is executed. For example, FOR X CALLED BY FOO WHEN CCODEP DO (PRINT (ARGLIST X)) will print out the argument list of all of the compiled functions which are called by FOO.

HELP

Prints out the summary of Masterscope command syntax as shown on the next page. Optional elements are shown in brackets []; alternatives are separated with vertical bars | or are listed on separate lines; words in angle brackets <> are "meta-objects"; other lower-case words are "noise words" and may be omitted.

Note: any command may be followed by OUTPUT filename to send output to the given file rather than the terminal, e.g. WHO CALLS WHO OUTPUT CROSSREF.

This completes the presentation of the Masterscope command language.

a <command> is:

```
[RE]ANALYZE <functions>
ERASE <functions>
show PATHS <pathoptions>
<set> {<relation>!IS!ARE} <set>
EDIT where <functions> [<relation> <set>] [- <edit commands>]
SHOW where <functions> <relation> <set>
CHECK <files>
FOR <variable> <set> <i.s. opr> <expression>
```

a <set> is (at least one of):

a determiner + a type + a specification

THE	FUNCTIONS	[']{atom!list}
ANY	VARIABLES	@ <pred>
WHICH	PROPERTY NAMES	IN <expression>
WHO	RECORDS	<relation>ING <set>
	FIELDS	<relation>ED {BY!IN} <set>
	FILES	THAT <relation> <set>
		LIKE <edit-pattern>
		ON <files>

FIELDS OF <records>
<blockword> {ON <files>}!OF <functions>}

<functions>, <files>, etc. are <set>s whose type is implied.

a <relation> is a verb and optional modifier:

verbs:	modifiers (anywhere after the verb):
CALL	... SOMEHOW FOR EFFECT FOR VALUE
BIND	
USE	... AS a {RECORD!PROPERTY!record FIELD} name
USE	... AS a CLISP word
USE	... FREELY!LOCALLY
SET	... FREELY!LOCALLY
SMASH	... FREELY!LOCALLY
TEST	... FREELY!LOCALLY
REFERENCE	... FREELY!LOCALLY
FETCH	
REPLACE	*
CREATE	<blockword>: ENTRIES, GLOBALVARS, FREEVARS,
CONTAIN	SPECVARS, LOCALFREEVARS, BLKFNS or BLOCKFNS

<pathoptions>:	abbreviations & synonyms:
FROM <functions>	FNS = FUNCTIONS PROPS = PROPERTIES
TO <functions>	VARS = VARIABLES
AVOIDING <functions>	(& singular FN, VARIABLE, etc)
NOTRACE <functions>	FREE = FREELY LOCAL = LOCALLY
SEPARATE <functions>	AMONG = AVOIDING NOT
LINELENGTH <number>	

<sets> may be joined by AND or OR or preceded by NOT.
Any command can be followed by OUTPUT <filename>.

20.1.2 PATHS

In trying to work with large programs, the user can lose track of the hierarchy which defines his program structure. Masterscope can aid the user by providing a map showing the calling structure of a set of functions, via the SHOW PATHS command. The SHOW PATHS command prints out a tree structure showing which functions call which other functions. For example, the command SHOW PATHS FROM MSPARSE will print out the structure of Masterscope's parser:

```

1. MSPARSE  MSINIT MSMARKINVALID
2.      |      MSINITH MSINITH
3.      |      MSINTERPRET MSRECORDFILE
4.      |      MSPRINTWORDS
5.      |      PARSECOMMAND GETNEXTWORD CHECKADV
6.      |      |      PARSERELATION {a}
7.      |      |      PARSESET {b}
8.      |      |      PARSEOPTIONS {c}
9.      |      |      MERGECONJ GETNEXTWORD {5}
10.     |      |      GETNEXTWORD {5}
11.     |      |      FIXUPTYPES SUBJTYPE
12.     |      |      |      OBJTYPE
13.     |      |      FIXUPCONJUNCTIONS MERGECONJ {9}
14.     |      |      MATCHSCORE
15.     |      MSPORTSENTENCE
----- overflow - a
16. PARSERELATION GETNEXTWORD {5}
17.      |      CHECKADV
----- overflow - b
18. PARSESET PARSESET
19.      |      GETNEXTWORD {5}
20.      |      PARSERELATION {6}
21.      |      SUBPARSE GETNEXTWORD {5}
----- overflow - c
22. PARSEOPTIONS GETNEXTWORD {5}
23.      |      PARSESET {19}
24.

```

Figure 20-3

Figure 20-3 displays that the function msparse calls msinit, msinterpret, and msprintsentence. msinterpret in turn calls msrecordfile, msprintwords, parsecommand, getnextword, fixuptypes, and fixupconjunctions. The numbers in braces {} after a function name are backward references: they indicate that the tree for that function was expanded on a previous line. The lowercase letters in braces are *forward* references: they indicate that the tree for that function will be expanded below, since there is no more room on the line. The vertical bar is used to keep the output aligned.

Path options

The SHOW PATHS command takes the form: SHOW PATHS followed by some combination of the following *path options*:

FROM set	Display the function calls from the elements of <u>set</u> .
TO set	Display the function calls leading <i>to</i> elements of <u>set</u> . If TO is given before FROM (or no FROM is given), the tree is "inverted" and a message, (<i>inverted tree</i>) is printed to warn the user that if FN1 appears after FN2 it is because FN1 is <i>called by</i> FN2.
When both FROM and TO are given, the first one indicates a set of functions which are to be displayed while the second restricts the paths that will be traced; i.e., the command SHOW PATHS FROM X TO Y will trace the elements of the set CALLED SOMEHOW BY X AND CALLING Y SOMEHOW.	
If TO is not given, TO KNOWN OR NOT @ GETD is assumed; that is, only functions which have been analyzed or which are undefined will be included. ¹⁵	
AVOIDING set	Do not display any function in <u>set</u> . AMONG is recognized as a synonym for AVOIDING NOT. For example, SHOW PATHS TO ERROR AVOIDING ON FILE2 will not display (or trace) any function on FILE2.
NOTRACE set	Do not trace from any element of <u>set</u> . NOTRACE differs from AVOIDING in that a function which is marked NOTRACE will be printed, but the tree beyond it will not be expanded; the functions in an AVOIDING set will not be printed at all. For example, SHOW PATHS FROM ANY ON FILE1 NOTRACE ON FILE2 will display the tree of calls emanating from FILE1, but will not expand any function on FILE2.
SEPARATE set	Give each element of <u>set</u> a separate tree. Note that FROM and TO only insure that the designated functions will be displayed. SEPARATE can be used to guarantee that certain functions will begin new tree structures. SEPARATE functions are displayed in the same manner as overflow lines; i.e., when one of the functions indicated by SEPARATE is found, it is printed followed by a forward reference (a lower-case letter in braces) and the tree for that function is then expanded below.
LINELENGTH n	Resets <u>linelength</u> to n before displaying the tree. The linelength is used to determine when a part of the tree should "overflow" and be expanded lower.

¹⁵ Note that Masterscope will analyze a function while printing out the tree if that function has not previously been seen and it currently has an expr definition; thus, any function which *can be* analyzed will be displayed.

20.1.3 AFFECTING MASTERSCOPE ANALYSIS

Masterscope will analyze the expr definitions of functions and note in its database the relations that function has with other functions and with variables. To perform this analysis, Masterscope uses *templates* which describe the behavior of functions.

A template is a pattern of a function's evaluation. A template is a list structure containing any of the following atoms:

PPE	If an expression appears in this location, there is most likely a parenthesis error. ¹⁶
NIL	The expression occurring at this location is not evaluated. For example, the template for QUOTE is (NIL . PPE).
SET	A variable appearing at this place is set. E.g., the template for SETQQ is (SET NIL . PPE).
SMASH	The value of this expression is smashed. For example, the template for DREVERSE is (SMASH . PPE).
* TEST	This expression is used as a predicate (that is, the only use of the value of the expression is whether it is NIL or non-NIL.) For example, the template for NULL is (TEST . PPE).
PROP	The value of this expression is used as a property name. If the expression is of the form (QUOTE atom), Masterscope will note that <u>atom</u> is USED AS A PROPERTY NAME. For example, the template for <u>getprop</u> is (EVAL PROP . PPE).
FUNCTION	The expression at this point is used as a functional argument. For example, the template for MAPC is (SMASH FUNCTION FUNCTION . PPE). ¹⁷
EVAL	The expression at this location is evaluated (but not set, smashed, tested, used as a functional argument, etc.).
* RETURN	The value of the function (of which this is the template) is the value of this expression.
+ EFFECT	The expression at this location is evaluated, but the value is not used. For example, the template for PROGN is (... EFFECT RETURN).
+ FETCH	An atom at this location is a field which is fetched.

* 16 Masterscope notes this as a "call" to the function "ppe" (lowercase). Note that, when Masterscope finds a possible parenthesis error in the course of analyzing a function definition, rather than printing the usual ".", it prints out a "?" instead.

17 Actually, Masterscope distinguishes between functional arguments to functions which "compile open" from those that do not. For the latter (e.g. SORT and APPLY), the token in the template is FUNCTIONAL rather than FUNCTION.

REPLACE	An atom at this location is a field which is replaced.	+
RECORD	An atom at this location is used as a record name.	+
CREATE	An atom at this location is a record which is created.	+
BIND	An atom at this location is a variable which is bound.	+
CALL	An atom at this location is a function which is called.	+
CLISP	An atom at this location is used as a clisp word.	+

In addition to the above atoms which occur in templates, there are some "special forms" which are lists keyed by their car.

.. template	Any part of a template may be preceded by the atom .. (two periods) which specifies that the template should be repeated an indefinite number ($n \geq 0$) of times to fill out the expression. For example, the template for COND might be (.. (TEST .. EFFECT RETURN)) while the template for SELECTQ is (EVAL .. (NIL .. EFFECT RETURN) RETURN).	+
(BOTH template template)	analyze the current expression twice, using the each of the templates in turn.	+
(IF expression template template)	evaluate <u>expression</u> at analysis time (the variable EXPR will be bound to the expression which corresponds to the IF), and if the result is non-NIL, use the first <u>template</u> , otherwise the second. If <u>expression</u> is a literal atom, it is <u>apply'd</u> to EXPR. For example, (IF LISTP (RECORD FETCH) FETCH) specifies that if the current expression is a list, then the first element is a record name and the second element a field name, otherwise it is a field name.	+
(@ exprform templateform)	evaluate <u>exprform</u> giving <u>expr</u> , evaluate <u>templateform</u> giving <u>template</u> . Then analyze <u>expr</u> with <u>template</u> . @ lets the user compute on the fly both a template and an expression to analyze it with. The forms can use the variable EXPR, which is bound to the current expression.	+
(MACRO . macro)	<u>macro</u> is interpreted in the same way as a compiler macro (see Section 18) ¹⁸ and the resulting form is analyzed.	+

Some examples of templates:

¹⁸ Additionally, the template for a function may be the atom MACRO itself, in which case, Masterscope will use the MACRO property of the function itself. This is useful when analyzing code which contains calls to user-defined compiler macros. If the user changes a macro property (e.g. by editing it) of an atom which has template=MACRO, Masterscope will mark any function which used that macro as needing to be reanalyzed.

function	template
AND MAPCAR	(TEST .. RETURN) (EVAL FUNCTION FUNCTION)

Templates may be changed and new templates defined via the function:

- + **gettemplate[fn]** returns current template of fn.
- + **settemplate[fn;template]** Changes the template for the function fn and returns the old value. If any functions in the database are marked as calling fn, they will be marked as needing re-analysis.

20.1.4 DATA BASE UPDATING

Masterscope is interfaced to the editor and file package so that it notes whenever a function has been changed, either through editing or loading in a new definition. Whenever a command is given which requires knowing the information about a specific function, if that function has been noted as being changed, the function is automatically re-analyzed before the command is interpreted. If the command requires that all the information in the database be consistent (e.g., the user asks WHO CALLS X) then *all* functions which have been marked as changed are re-analyzed.

20.1.5 MASTERSCOPE ENTRIES

- + **calls[fn;usedatabase]** fn can be a function name, a definition, or a form. calls will also work on compiled code. calls returns a list of four elements: a list of all the functions called by fn,¹⁹ a list of all the variables bound in fn, a list of all the variables used freely in fn, and a list of the variables used globally in fn. For the purpose of calls, variables used freely which are on globalvars or have a property GLOBALVAR value T are considered to be used globally. If usedatabase is NIL (or fn is not a litatom), calls will perform a one-time analysis of fn. Otherwise (i.e. if usedatabase is non-NIL and fn a function name), calls will use the information in Masterscope's database (fn will be analyzed first if necessary).

- + **callscode[fn]** The sub-function of calls which analyzes compiled code. callscode returns a list of five elements: a list of all the functions called via "linked" function calls, a list of all functions called regularly, a list

¹⁹ Functions called via "linked" calls from compiled code are indicated by semicolons packed around their name; e.g. calls[MASTERSCOPE] might return ((;MASTERSCOPEEXEC; ;MSINTERPRET; ;PRINT; HELP) --). This feature can be suppressed by setting nopackcallsflg to T.

of variables bound in fn, a list of variables used freely, and a list of variables used globally.

freevars[fn;usedatabase]

Equivalent to caddr[calls[fn;usedatabase]]. Returns the list of variables used freely within fn.

masterscope[command]

Top level entry to Masterscope. If command is NIL, will enter into a userexec in which the user may enter commands. If command is not NIL, the command is interpreted and masterscope will return the value that would be printed by the command. Note that only the question commands return meaningful values.

setsynonym[newphrase;meaning]

Defines a new synonym for Masterscope's parser. Both newphrase and meaning are lists of words; anywhere newphrase is seen in a command, meaning will be substituted. For example, setsynonym[GLOBALS ;(VARS IN GLOBALVARS OR @(GETPROP X 'GLOBALVAR))] would allow the user to refer with the single word GLOBALS to the set of variables which are either in globalvars or have a GLOBALVAR property.

The following functions are provided for users who wish to write their own routines using Masterscope's database:

parserelation[relation]

relation is a relation phrase; e.g., parserelation[(USE FREELY)]. parserelation returns an internal representation for relation. For use in conjunction with getrelation.

getrelation[item;relation;inverted]

relation is an internal representation as returned by parserelation (if not, getrelation will first perform parserelation[relation]); item is an atom. getrelation returns the list of all atoms which have the given relation to item. For example, getrelation[X ;(USE FREELY)] returns the list of variables that X uses freely. If inverted is T, the inverse relation is used; e.g. getrelation[X ;(USE FREELY) ;T] returns the list of functions which use X freely.

If item is NIL, getrelation will return the list of atoms which have relation with any other item; i.e., answers the question WHO relationS ANY. Note that getrelation does not check to see if item has been analyzed, or that other functions that have been changed have been re-analyzed.

testrelation[item;relation;item2;inverted]

equivalent to memb[item2;getrelation[item;relation;inverted]], that is, tests if item and item2 are related via relation. If item2 is NIL, the call is equivalent to not>null[getrelation[item;relation;inverted]], i.e., testrelation tests if item has the given relation with any other item.

+ maprelation[relation;mapfn]	Calls the function <u>mapfn</u> on every pair of items related via <u>relation</u> . If <u>nargs[mapfn]</u> is 1, then <u>mapfn</u> is called on every item which has the given <u>relation</u> to <u>any</u> other item.
+ updatefn[fn;evenifvalid]	Equivalent to the command ANALYZE 'fn' ; that is, <u>updatefn</u> will analyze <u>fn</u> if <u>fn</u> has not been analyzed before or if it has been changed since the time it was analyzed. If <u>evenifvalid</u> is set, <u>updatefn</u> will re-analyze <u>fn</u> even if Masterscope thinks it has a valid analysis in the database.
+ updatechanged[]	Performs <u>updatefn[fn]</u> on every function which has been marked as changed.
* msmarkchanged[fn]	Mark that <u>fn</u> has been changed and needs to be reanalyzed.
dumpdatabase[fnlst]	Dumps the current Masterscope database on the current output file in a <u>loadable</u> form. If <u>fnlst</u> is not <u>NIL</u> , <u>dumpdatabase</u> will only dump the information for the list of functions in <u>fnlst</u> . ²⁰ The variable <u>databasecoms</u> is initialized to <u>((E (DUMPDATABASE)))</u> ; thus, the user may merely perform <u>makefile[DATABASE.extent]</u> to save the current Masterscope database. If a Masterscope database already exists when a DATABASE file is loaded, the database on the file will be merged with the one in core. ²¹

20.1.6 ERROR MESSAGES

When the user gives Masterscope a command, the command is first parsed, i.e. translated to an internal representation, and then the internal representation is interpreted. If a command cannot be parsed, e.g. if the user typed **SHOW WHERE CALLED BY X**, the message "Sorry, I can't parse that!" is printed and an error is generated. If the command is of the correct form but cannot be interpreted (e.g., the command **EDIT WHERE ANY CONTAINS ANY**) Masterscope will print the message "Sorry, that isn't implemented!" and generate an error. If the command requires that some functions having been analyzed (e.g., the command **WHO CALLS X**) and the database is empty, Masterscope will print the message "Sorry, no functions have been analyzed!" and generate an error.

20.1.7 NOTICING CHANGES THAT REQUIRE RECOMPILING

- + When a record declaration, iterative statement operator or compiler macro is changed, and

²⁰ The databasefns package (section 24) provides a more convenient way of saving data bases along with the source files which they correspond to.

²¹ Note that functions whose definitions are different from their definition when the database was made must be REANALYZEd if their new definitions are to be noticed.

Masterscope has "noticed" a use of that declaration or macro (i.e. it is used by some function known about in the data base) Masterscope will alert the user about those functions which might need to be re-compiled (e.g. they do not currently have expr definitions).²² The functions which need recompiling are added to the list needunsave and a message is printed out: The functions fn1, fn2,... use compiler macros which have changed. Call UNSAVEFNS() to load and/or unsave them. The function:

unsavefns()	Uses <u>loadfns</u> or <u>unsavedef</u> to make sure that all functions in the list <u>needunsave</u> have expr definitions, and then sets <u>needunsave</u> to NIL.
-------------	--

20.1.8 IMPLEMENTATION NOTES

Masterscope keeps a database of the relations noticed when functions are analyzed. The relations are intersected to form "primitive relationships" such that there is little or no overlap of any of the primitives. For example, the relation SET is stored as the union of SET LOCAL and SET FREE. The BIND relation is divided into BIND AS ARG, BIND AND NOT USE, and SET LOCAL, SMASH LOCAL, etc. Splitting the relations in this manner reduces the size of the database considerably, to the point where it is reasonable to maintain a Masterscope database for a large system of functions during a normal debugging session.

Each primitive relationship is stored in a pair of hash-tables, one for the "forward" direction and one for the "reverse". For example, there are two hash tables, USE AS PROPERTY and USED AS PROPERTY. To retrieve the information from the database, Masterscope performs unions of the hash-values. For example, to answer FOO BINDS WHO Masterscope will look in all of the tables which make up the BIND relation. The "internal representation" returned by parserelation is just a list of dotted pairs of hash-tables. To perform getrelation requires only mapping down that list, doing gethash's on the appropriate hash-tables and unioning the result.

Hash tables are used for a variety of reasons: storage space is smaller; it is not necessary to maintain separate lists of which functions have been analyzed (a special table, DOESN'T DO ANYTHING is maintained for functions which neither call other functions nor bind or use any variables); and accessing is relatively fast. Within any of the tables, if the hash-value would be a list of one atom, then the atom itself, rather than the list, is stored as the hash-value. This also reduces the size of the database significantly.

This completes the section on Masterscope.

²² Extra functions may be noticed; for example if FOO contains a (fetch (REC X) --), and some declaration other than REC which contains an X is changed, Masterscope will still think that FOO needs to be loaded/unsaved.

20.2 HELPSYS²³

Helpsys provides yet another form of on-line assistance to the user, namely in the area of documentation of Interlisp. Helpsys uses the Interlisp Reference Manual as a data base, and answers simple questions about Interlisp by presenting the appropriate passages from the manual. Inquiries are made in the form of simple sentences, void of punctuation, terminated with a carriage-return. The following sample session illustrates the use of Helpsys. User inputs are underlined.

```

<HELP SYS()
Type ??? <CR> for assistance
!TELL ME ABOUT EVAL
Subcategories available:
1. function description
2. break command
3. cross referenced subject
4. ERRORSET {cross referenced subject}
Which one ? FUNCTION DESCRIPTION           {user typed '1a'}
8
eval[x]                                eval evaluates the expression x and returns this
                                         value i.e., eval provides a way of calling the
                                         interpreter.
-----
8
In INTERLISP-10, eval is a subr so that the "name" x does not
actually appear on the stack.
-----
```

Note that eval is itself a lambda type function, so its argument is first evaluated, e.g.,

```

<SET(FOO (ADD1 3))
(ADD1 3)
<(EVAL FOO)
4
<EVAL(FOO) or (EVAL (QUOTE FOO))
(ADD1 3)
```

```

!REDO
Subcategories available:
1. function description
2. break command
3. cross referenced subject
4. ERRORSET {cross referenced subject}
Which one ? BREAK COMMAND                {user typed '2a'}
-----
```

```

... from section 15
EVAL          Same as GO or OK except that the break is
```

²³ Helpsys was written by R. L. Walker II.

maintained after the evaluation. The user can then interrogate the value of the break which is bound on the variable !value, and continue with the break. Typing GO or OK following EVAL will not cause reevaluation but another EVAL will. EVAL is a useful command when the user is not sure whether or not the break will produce the correct value and wishes to be able to do something about it if it is wrong.

!TELL ME ABOUT UB

(UB as a break command from section 15)

UB

unbreaks brkfn, e.g.,

(FOO BROKEN)
:UB
FOO
:

!WHAT ARE THE ARGS FOR BREAK1

Valid arguments for BREAK1 are: [BRKEXP;BRKWHEN;BRKFN;BRKCOMS;BRKTYPE]

!TELL ME ABOUT THE LAST ARGUMENT OF BREAK1

=ARGUMENT

Subcategories available:

1. function description
2. topic description
3. cross referenced subject
4. TRACE {cross referenced subject}
5. FAULTEVAL {cross referenced subject}
6. HELP {cross referenced subject}

Which one ? FUNCTION DESCRIPTION

{user typed '1'}

break1[brkexp;brkwhen;brkfn;brkcoms;brktype]

is an nlambda. brktype is NIL for user breaks, INTERRUPT for control-H breaks, and ERRORX for error breaks.

!TELL ME ABOUT THE 2ND ARG OF CHANGEPROP

(CHANGEPROP as a function description from section 7)

changeprop[x;prop1;prop2]

Changes name of property prop1 to prop2 on property list of x, (but does not affect the value of the property).

!TELL ME ABOUT ERROR IN ARRAYSIZE

(ARRAYSIZE as a function description from section 10)
arraysize[a]

Generates an error, ARG NOT ARRAY, if a is not an array.

WHAT IS THE VALUE OF MAPC?

(MAPC as a function description from section 11)
mapc[mapx;mapfn1;mapfn2]

The value of mapc is NIL. mapc compiles open.

TELL ME ABOUT SPLICE?

(SPLICE, type of read-macro, from section 14)
(2) SPLICE

The result (which should be a list or NIL) is nconc'd into the input list, e.g., if ! is defined by (SPLICE (LAMBDA NIL (APPEND FOO))), and the value of foo is (A B C), when the user inputs (X ! Y), the result will be (X A B C Y).

TELL ME ABOUT CLISP\$FLG?

1. CLISPFLG
2. CLISPIFTRANFLG
3. CLISPIFYENGLSHFLG
4. CLISPIFYPACKFLG
5. CLISPIFYPRETTYFLG
6. CLISPRETRANFLG

Which one? CLISPIFYPRETTYFLG

{user typed '5'}

Subcategories available:

1. prettydef variable/parameter
2. CLISP variable/parameter

Which one? CLISP VARIABLE/PARAMETER

{user typed '2'}

clispifyprettyflg if non-NIL, causes prettyprint to CLISPIFY selected function definitions before printing them according to the following interpretations

131

of clispifyprettyflg:

131

Another way to inform prettyprint to clispify functions is for the function to have a CLISP declaration containing the word CLISPIFY.

ALL

all functions

T,EXPRS functions currently defined as
20.24

exprs

CHANGES

functions marked as having
been changed

a list

a member of that list

clispifyprettyflg is (temporarily) reset to T
when makefile is called with the option
CLISPIFY, and reset to CHANGES when the file
being dumped has the property FILETYPE value

132

CLISP. clispifyprettyflg is initially NIL.

132

If clispifyprettyflg is non-NIL, and the only transformation
performed by DWIM are well formed CLISP transformations, i.e., no
spelling corrections, the function will not be marked as changed,
since it would only have to be re-clispified and re-prettyprinted
when the file was written out.

!TELL ME ABOUT COROUTINES AND GENERATORS?
=coroutines% and% generators

(coroutines and generators as a cross referenced subject from section
12)

5

12.7 Coroutines and Generators

5

Designed and implemented by D.G. Bobrow, who also did the
documentation. Early versions of the Conniver possibilites-list
package were written by Henry Thompson. Daryle Lewis found and
corrected a number of bugs, and wrote the compiler macros that go
with the package.

This section describes an application of the spaghetti stack facility to
provide mechanisms for creating and using simple generators (with and
without CLISP, Section 23), generalized coroutines, and Conniver style
possibility lists.

!MORE?

coroutines and generators ...continued from section 12

A generator is like a subroutine except that it retains information
about previous times it has been called. Some of this state may be data

(for example, the seed in a random number generator), and some may be in program state (as in a recursive generator which finds all the atoms in a list structure). For example, if listgen is defined using defineq as:

```
(LISTGEN (L)
  (IF L THEN (PRODUCE L:1) (LISTGEN L::1)))
```

we can use the function generator (described below) to create a generator that uses listgen to produce the elements of a list one at a time, e.g.,

```
GR←(GENERATOR (LISTGEN '(A B C))
```

creates a generator, which can be called by

```
(GENERATE GR)
```

to produce as values on successive calls, A, B, C. When generate (not generator) is called the first time, it simply starts evaluating (LISTGEN '(A B C)). produce gets called from listgen, and pops back up to generate with the indicated value after saving the state. When generate gets called again, it continues from where the last produce left off. This process continues until finally listgen completes and

12.15

More ? No

10K2
thank-you
←

20.26

Index for Section 20

	Page Numbers
AMONG (Masterscope path option)	20.15
ANALYZE (Masterscope command)	20.9
AVOIDING (Masterscope path option)	20.15
BIND (Masterscope relation)	20.5
BIND (Masterscope template)	20.17
BOTH (Masterscope template)	20.17
BY (Masterscope set specification)	20.7
CALL DIRECTLY (Masterscope relation)	20.5
CALL FOR EFFECT (Masterscope relation)	20.5
CALL FOR VALUE (Masterscope relation)	20.5
CALL INDIRECTLY (Masterscope relation)	20.5
CALL SOMEHOW (Masterscope relation)	20.4
CALL (Masterscope relation)	20.4
CALL (Masterscope template)	20.17
CALLS[FN;USEDATABASE]	20.18
CALLSCCODE[FN]	20.18
CHECK (Masterscope command)	20.11
CLISP (Masterscope template)	20.17
conjunctions (in Masterscope)	20.9
CONTAIN (Masterscope relation)	20.5
CREATE (Masterscope relation)	20.5
CREATE (Masterscope template)	20.17
DECLARE AS LOCALVAR (Masterscope relation)	20.6
DECLARE AS SPECVAR (Masterscope relation)	20.6
DESCRIBE (Masterscope command)	20.11
determiners (in Masterscope)	20.8
DUMPDATABASE[FNLST]	20.20
EDIT WHERE (Masterscope command)	20.11
EDIT (Masterscope command)	20.11
EFFECT (Masterscope template)	20.16
ERASE (Masterscope command)	20.10
EVAL (Masterscope template)	20.16
FETCH (Masterscope relation)	20.5
FETCH (Masterscope template)	20.16
FIELDS OF (Masterscope set specification)	20.7
FOR (Masterscope command)	20.12
FREELY (use in Masterscope)	20.5
FREEVARS[FN;USEDATABASE]	20.19
FROM (Masterscope path option)	20.15
FUNCTION (Masterscope template)	20.16
GETRELATION[ITEM;RELATION;INVERTED]	20.19
GETTEMPLATE[FN]	20.18
HELP (Masterscope command)	20.12
helpsys package	20.22-27
IF (Masterscope template)	20.17
IN (Masterscope set specification)	20.6
KNOWN (Masterscope set specification)	20.8
LIKE (Masterscope set specification)	20.7
LINELENGTH (Masterscope path option)	20.15
LOCALLY (use in Masterscope)	20.5
MACRO (Masterscope template)	20.17
MAPRELATION[RELATION;MAPFN]	20.20
Masterscope	20.1-22
MASTERSCOPE[COMMAND]	20.19
Masterscope commands	20.9-12
MSMARKCHANGED[NAME;TYPE;NEWFLG]	20.20

	Page Numbers
MSPRINTFLG (Masterscope parameter)	20.3
NEEDUNSAVE (Masterscope paramater)	20.21
NIL (Masterscope template)	20.16
NOPACKCALLSFLG (Masterscope parameter)	20.18
NOTRACE (Masterscope path option)	20.15
ON PATH (Masterscope set specification)	20.8
OUTPUT (Masterscope command)	20.12
PARSERELATION[RELATION]	20.19
path options (in Masterscope)	20.15
paths (in Masterscope)	20.14-15
PPE (Masterscope template)	20.16
PROP (Masterscope template)	20.16
REANALYZE (Masterscope command)	20.9
RECORD (Masterscope template)	20.17
REFERENCE (Masterscope relation)	20.5
relations (in Masterscope)	20.4-6
REPLACE (Masterscope relation)	20.5
REPLACE (Masterscope template)	20.17
RETURN (Masterscope template)	20.16
SEPARATE (Masterscope path option)	20.15
set specifications (in Masterscope)	20.6-8
SET (Masterscope relation)	20.4
SET (Masterscope template)	20.16
sets (in Masterscope)	20.6-8
SETSYNONYM[NEWPHRASE;MEANING]	20.19
SETTEMPLATE[FN;TEMPLATE]	20.18
SHOW PATHS (Masterscope command)	20.10
SHOW WHERE (Masterscope command)	20.11
SMASH (Masterscope relation)	20.4
SMASH (Masterscope template)	20.16
SORRY, I CAN'T PARSE THAT (error message)	20.20
SORRY, NO FUNCTIONS HAVE BEEN ANALYZED (error message)	20.20
SORRY, THAT ISN'T IMPLEMENTED (error message)	20.20
templates (in Masterscope)	20.16-18
TEST (Masterscope relation)	20.5
TEST (Masterscope template)	20.16
TESTRELATION[ITEM;RELATION;ITEM2;INVERTED]	20.19
THOSE (Masterscope set specification)	20.8
TO (Masterscope path option)	20.15
types (in Masterscope)	20.8-9
UNSAVEFNS[FNS]	20.21
UPDATECHANGED[]	20.20
UPDATEFN[FN;EVENIFVALID]	20.20
USE AS A CLISP WORD (Masterscope relation)	20.5
USE AS A FIELD (Masterscope relation)	20.5
USE AS A PROPERTY (Masterscope relation)	20.5
USE AS A RECORD (Masterscope relation)	20.5
USE (Masterscope relation)	20.4
' (Masterscope set specification)	20.6
. (printed by Masterscope)	20.3
. (use in Masterscope)	20.3
.. (Masterscope template)	20.17
? (printed by Masterscope)	20.16
@ (Masterscope set specification)	20.6
@ (Masterscope template)	20.17

SECTION 21

MISCELLANEOUS¹

21.1 MEASURING FUNCTIONS

time[timex;timen;timetyp]

is an nlambda function. It executes the computation timex, and prints out the number of conses and computation time. Garbage collection time is subtracted out.

```
←TIME((LOAD (QUOTE PRETTY) (QUOTE PROP])
FILE CREATED 1-AUG-78 14:56:12
PRETTYCOMS
collecting lists
582, 10291 free cells
13169 CONSES
29.484 SECONDS
PRETTY
```

If timen is greater than 1 (timen=NIL equivalent to timen=1), time executes timex timen number of times and prints out number of conses/timen, and computation time/timen. This is useful for more accurate measurement on small computations, e.g.

```
←TIME((COPY (QUOTE (A B C))) 10)
30/10 = 3 CONSES
.055/10 = .0055 SECONDS
(A B C)
```

¹ Some of the functions in this section are TENEX/TOPS-20 or implementation dependent. They may not be provided in other implementations of Interlisp.

If timetype is 0, time measures and prints total *real* time as well as computation time, e.g.

```
+TIME((LOAD (QUOTE PRETTY) (QUOTE PROP)) 1 0]
FILE CREATED 7-MAY-71 12:47:14
```

```
GC: 8
582, 10291 FREE WORDS
PRETTYFNS
PRETTYVARS
3727 CONSES
11.193 SECONDS
27.378 SECONDS, REAL TIME
PRETTY
```

If timetyp = 3, time measures and prints garbage collection time as well as computation time, e.g.

```
+TIME((LOAD (QUOTE PRETTY) (QUOTE PROP)) 1 3]
FILE CREATED 7-MAY-71 12:47:14
```

```
GC: 8
582, 1091 FREE WORDS
PRETTYFNS
PRETTYVARS
3727 CONSES
10.597 SECONDS
1.487 SECONDS, GARBAGE COLLECTION TIME
PRETTY
```

Another option is timetype=T, in which case time measures and prints the number of pagefaults.

The value of time is the value of the last evaluation of timex.

date[-]²

obtains date and time, returning it as single string in format "dd-mm-yy hh:mm:ss", where dd is day, mm is month, yy year, hh hours, mm minutes, ss seconds, e.g., "14-MAY-71 14:26:08".

idate[d]

d is a date and time string. Value of idate is d converted to a number such that if date1 is before (earlier than) date2, then idate[date1] < idate[date2]. If d=NIL, idate returns idate[date[]].

+ **gdate[date;formatbits;strptr]**

Interlisp-10 function for obtaining time-date formatted string, date is in internal date-and-time format. If NIL, current time and date

² In Interlisp-10, date will accept formatbits as an argument, which can be used to specify other formats, e.g. day of week, time zone, etc., as described in the JSYS manual.

is used, i.e. value of idate[]. formatbits is 36 bit quantity to be passed to TENEX/TOPS 20 time-date conversion routines (see JSYS manual.) For example, formatbits=-1 gives a "long" date, e.g. "FRIDAY, JUN 16, 1978, 23:41:52-PDT".³ If formatbits=NIL, defaults to a value which will produce the same format as that of date[], i.e. "dd-mm-yy hh:mm:ss". strptr is an optional string pointer to be reused.⁴

clock[n]

for n=0 current value of the time of day clock i.e., number of milliseconds since last system start up.

for n=1 value of the time of day clock when the user started up this Interlisp, i.e., difference between clock[0] and clock[1] is number of milliseconds (real time) since this Interlisp was started.

for n=2 number of milliseconds of *compute* time since user started up this Interlisp (garbage collection time is subtracted off).

for n=3 number of milliseconds of compute time spent in garbage collections (all types).⁵

dismiss[n]

In Interlisp-10, dismisses program for n milliseconds, during which time program is in a state similar to an I/O wait, i.e., it uses no CPU time. Can be aborted by control-D, control-E, or control-B.

conscount[n]

conscount[] returns the number of conses since Interlisp started up. If n is not NIL, resets conscount to n.

boxcount[type;n]

In Interlisp-10, number of boxing operations (see Section 13) since Interlisp started up. If type=NIL, returns number of large integer boxes; type=FLOATING, returns number of floating boxes.⁶ If n is not NIL, resets the corresponding counter to n.

³ The dateformat package (Section 24) provides a convenient way of specifying the format bits in terms of keywords.

⁴ In this case, the string characters are stored in an internal scratch string, macscratchstring, so that a subsequent call to gdate will overwrite the characters returned by this one. Note that this internal scratch string is also used by several other functions in this section.

⁵ In Interlisp-10, this number is directly accessible via the COREVAL GCTIM.

⁶ In Interlisp-10, these counters are directly accessible via the COREVALS IBOXCN and FBOXCN.

gctrp[-]

number of conses to next garbage collection of lists, i.e., number of list words not in use. Note that an intervening garbage collection of another type could collect as well as allocate additional list words. See Section 3.

gctrp[n] can be used to cause an interrupt when value of gctrp[] = n, see Section 10.

pagefaults[]

In Interlisp-10, number of page faults since Interlisp started up.

logout[]

returns control to operating system.⁷ In Interlisp-10, a subsequent CONTINUE command will enter the Interlisp-10 program, return NIL as the value of the call to logout, and continue the computation exactly as if nothing had happened, i.e., logout is a programmable control-C.

logout[] will not affect the state of any open files.

21.2 BREAKDOWN⁸

Time gives analyses by computation. Breakdown is available to analyze the breakdown of computation time (or any other measureable quantity) function by function. The user calls breakdown giving it a list of functions of interest. These functions are modified so that they keep track of the "charge" assessed to them. The function brkdwnresults gives the analysis of the statistic requested as well as the number of calls to each function.⁹ Sample output is shown below.

```

←BREAKDOWN(SUPERPRINT SUBPRINT COMMENT1)
(SUPERPRINT SUBPRINT COMMENT1)
←PRETTYDEF((SUPERPRINT) FOO)
FOO.;3
* ←BRKDWNRESULTS()
FUNCTIONS    TIME      # CALLS    PER CALL      %
SUPERPRINT   8.261      365       0.023        20
SUBPRINT     31.910     141       0.226        76
COMMENT1     1.612      8         0.201        4
TOTAL        41.783     514       0.081
NIL

```

The procedure used for measuring is such that if one function calls other and both are 'broken down', then the time (or whatever quantity is being measured) spent in the inner function is *not*

⁷ In Interlisp-10, if Interlisp was started as a subsidiary fork (see subsys, page 21.7), control is returned to the higher fork.

⁸ breakdown was written by W. Teitelman, and extended by L. P. Deutsch.

⁹ + brkdwnresults takes an optional argument, returnvaluesflg, which if non-NIL, causes brkdwnresults not to print the results, but instead to return them in the form of a list of elements of the form (name # calls value), e.g. for the example shown above, this list would be ((SUPERPRINT 365 8261)(SUBPRINT 141 31910)(COMMENT1 8 1612)).

charged to the outer function as well.¹⁰

To remove functions from those being monitored, simply unbreak the functions, thereby restoring them to their original state. To add functions, call breakdown on the new functions. This will not reset the counters for any functions not on the new list. However breakdown[] can be used for zeroing the counters of all functions being monitored.

To use breakdown for some other statistic, before calling breakdown, set the variable brkdwntype to the quantity of interest, e.g., TIME, CONSES, etc, or a list of such quantities. Whenever breakdown is called with brkdwntype not NIL, breakdown performs the necessary changes to its internal state to conform to the new analysis. In particular, if this is the first time an analysis is being run with a particular statistic, a measuring function will be defined, and the compiler will be called to compile it.¹¹ The functions being broken down will then be redefined to call this measuring function. When breakdown is through initializing, it sets brkdwntype back to NIL. Subsequent calls to breakdown will measure the new statistic until brkdwntype is again set and a new breakdown performed. Sample output is shown below:

```

←SET(BRKDWNTYPE (TIME CONSES))
(TIME CONSES)
←BREAKDOWN(MATCH CONSTRUCT)
(MATCH CONSTRUCT)
←FLIP((A B C D E F G H C Z) (.. $1 .. #2 ..) (.. #3 ..))
(A B D E F G H Z)
←RESULTS()
FUNCTIONS      TIME      # CALLS      PER CALL      %
MATCH          0.036        1           0.036       54
CONSTRUCT      0.031        1           0.031       46
TOTAL          0.067        2           0.033
FUNCTIONS      CONSES      # CALLS      PER CALL      %
MATCH          32           1           32.000      40
CONSTRUCT      49           1           49.000      60
TOTAL          81           2           40.500
NIL

```

The value of brkdwntype is used to search the list brkdwntypes for the information necessary to analyze this statistic. The entry on brkdwntypes corresponding to brkdwntype should be of the form (type form function), where form computes the statistic, and function (optional) converts the value of form to some more interesting quantity, e.g. (TIME (CLOCK 2) (LAMBDA (X) (FQUOTIENT X 1000))) measures computation time and reports the result in seconds instead of milliseconds. If brkdwntype is not defined on brkdwntypes, an error is generated. brkdwntypes currently contains entries for TIME, CONSES, PAGEFAULTS, BOXES, and FBOXES.

¹⁰ breakdown will not give accurate results if a function being measured is not returned from normally, e.g., a lower retfrom (or error) bypasses it. In this case, all of the time (or whatever quantity is being measured) between the time that function is entered and the time the next function being measured is entered will be charged to the first function.

¹¹ The measuring functions for TIME, CONSES, BOXES, and FBOXES have already been compiled.

MORE ACCURATE MEASUREMENT

Occasionally, a function being analysed is sufficiently fast that the overhead involved in measuring it obscures the actual time spent in the function. If the user were using time, he would specify a value for timen greater than 1 to give greater accuracy. A similar option is available for breakdown. The user can specify that a function(s) be executed a multiple number of times for each measurement, and the average value reported, by including a number in the list of functions given to breakdown, e.g., BREAKDOWN(EDITCOM EDIT4F 10 EDIT4E EQP) means normal breakdown for editcom and edit4f but executes (the body of) edit4e and eqp 10 times each time they are called. Of course, the functions so measured must not cause any harmful side effects, since they are executed more than once for each call. The printout from results will look the same as though each function were run only once, except that the measurement will be more accurate.

- + Another way of obtaining more accurate measurement is to expand the call to the measuring function in-line. If the value of brkdwncmplg is non-NIL (initially NIL), then whenever a function is broken-down, it will be redefined to call the measuring function, and then recompiled.
- + The measuring function is expanded in-line via an appropriate macro. In addition, whenever brkdwnotype is reset, the compiler is called for *all* functions for which brkdwncmplg was set at the time they were originally broken-down, i.e. the setting of the flag at the time a function is broken-down determines whether the call to the measuring code is compiled in-line.

21.3 INTERFORK COMMUNICATION IN INTERLISP-10

The functions described below permit two forks (one or both of them Interlisp-10) to have a common area of address space for communication by providing a means of assigning a block of storage *guaranteed not to move during garbage collections*.

getblk[n]

Creates a block n pages in size (512 words per page). Value is the address of the first word in the block, which is a multiple of 512 since the block will always begin at a page boundary. If not enough pages are available, generates the error **ILLEGAL OR IMPOSSIBLE BLOCK**.

Note: the block can be used for storing unboxed numbers only.

To store a number in the block, the following function could be used:

```
[SETBLOCK (LAMBDA (START N X) (CLOSER (IPLUS (LOC START) N) X))]
```

Some boxing and unboxing can be avoided by making this function compile open via a substitution macro.

Note: getblk should be used sparingly since several unmovable regions of memory can make it difficult or impossible for the garbage collector to find a contiguous region large enough for expanding array space.

relblk[address;n]

releases a block of storage beginning at address and extending for n

pages. Causes an error ILLEGAL OR IMPOSSIBLE BLOCK if any of the range is not a block. Value is address.

21.4 SUBSYS¹²

This section describes a function, subsys, which permits the user to run a TENEX/TOPS 20 subsystem, such as SNDMSG, SRCCOM, TECO, or even another Interlisp, from inside of an Interlisp without destroying the latter. In particular, SUBSYS(EXEC) will start up a lower exec, which will print the operating system herald, followed by @. The user can then do anything at this exec level that he can at the top level, without affecting his superior Interlisp. For example, he can start another Interlisp, perform a sysin, run for a while, type a control-C returning him to the lower exec, RESET, do a SNDMSG, etc. The user exits from the lower exec via the command QUIT,¹³ which will return control to subsys in the higher Interlisp. Thus with subsys, the user need not perform a sysout to save the state of his Interlisp in order to use a TENEX/TOPS 20 capability which would otherwise clobber the core image. Similarly, subsys provides a way of checking out a sysout file in a fresh Interlisp without having to commandeer another terminal or detach a job.

While subsys can be used to run any subsystem directly, without going through an intervening exec, this procedure is not recommended. The problem is that control-C always returns control to the next highest *EXEC*. Thus if the user is running an Interlisp in which he performs SUBSYS(LISP), and then types control-C to the lower Interlisp, control will be returned to the exec above the first Interlisp. If the user elects to call a subsystem directly, he must therefore know how it is normally exited and always exit from it that way.¹⁴

Starting a lower exec does not have this disadvantage, since it can *only* be exited via QUIT or POP, i.e., the lower exec is effectively "errorset protected" against control-C.

subsys[file/fork;incomfile;outcomfile;entrypointflg]

If file/fork=EXEC, starts up a lower exec, otherwise runs <SUBSYS>system, e.g. subsys[SNDMSG], subsys[TECO] etc. subsys[] is same as subsys[EXEC]. Control-C always returns control to next higher *exec*. Note that more than one Interlisp can be stacked, but there is no backtrace to help you figure out where you are.

incomfile and outcomfile provide a way of specifying files for input and output. incomfile can also be a string, in which case a temporary file is created, and the string printed on it.

entrypointflg may be START, REENTER, or CONTINUE. NIL is equivalent to START, except when file/fork is a handle (see below) in which case NIL is equivalent to CONTINUE.

¹² subsys was written by J.W. Goodwin and modified by D. C. Lewis. It is TENEX/TOPS 20 dependent and may not be available in implementations of Interlisp other than Interlisp-10.

¹³ POP on TOPS-20.

¹⁴ Interlisp is exited via the function logout, TECO via the command ;H, SNDMSG via control-Z, and EXEC via QUIT.

The value of subsys is a large integer which is a handle to the lower fork. The lower fork is *not* reset unless the user specifically does so using kfork, described below.¹⁵ If subsys is given as its first argument the value of a previous call to subsys,¹⁶ it continues the subsystem run by that call. For example, the user can do (SETQ SOURCES (SUBSYS TECO)), load up the TECO with a big source file, massage the file, leave TECO with ;H, run Interlisp for awhile (possibly including other calls to subsys) and then perform (SUBSYS SOURCES) to return to TECO, where he will find his file loaded and even the TECO pointer position preserved.

Note that if the user starts a lower EXEC, in which he runs an Interlisp, control-C's from the Interlisp, then QUIT from the EXEC, if he subsequently continues this EXEC with subsys, he can reenter or continue the Interlisp.

Note also that calls to subsys can be stacked. For example, using subsys, the user can run a lower Interlisp, and within that Interlisp, yet another, etc., and ascend the chain of Interlisps using logout, and then descend back down again using subsys.

For convenience, subsys[T] continues the last subsystem run.

SNDMSG, LISP, TECO, and EXEC, are all LISPMACROS which perform the corresponding calls to subsys. CONTIN is a LISPMACRO which performs subsys[T], thereby continuing the last subsys.¹⁷

kfork[fork] accepts a value from subsys and kills it (RESET in TENEX terminology). If subsys[fork] is subsequently performed, an error is generated. kfork[T] kills all outstanding forks (from this Interlisp).

21.5 MISCELLANEOUS OPERATING SYSTEM FUNCTIONS

fildir[filegroup;-] filegroup is a file group descriptor, i.e., it can contain stars. fildir returns a list of the files which match filegroup, a la the DIRECTORY command, e.g., (FILDIR (QUOTE *.COM;0)).

loadav[] returns current load average as a floating point number (this number is the first of the three printed by the SYSTAT command).

erstr[ern;-] ern is an error number from a JSYS fail return. ern=NIL means the most recent error. erstr returns the operating system error diagnostic as a string.

¹⁵ The fork is also reset when the handle is no longer accessible, i.e., when nothing in the Interlisp system points to it. Note that the fork is accessible while the handle remains on the history list.

¹⁶ Must be the exact same large number, i.e., eq. Note that if the user neglects to set a variable to the value of a call to subsys, (and has performed an intervening call so that subsys[T] will not work), he can still continue this subsystem by obtaining the value of the call to subsys for the history list using the function valueof, described in Section 22.

¹⁷ The EXEC lispmacro is defined to save its value on lastexec so that subsequent EXEC commands will restart the same exec.

jsys[n;ac1;ac2;ac3;resultac]	loads (unboxed) values of <u>ac1</u> , <u>ac2</u> , and <u>ac3</u> into appropriate accumulators, and executes JSYS number <u>n</u> . ¹⁸ If <u>ac1</u> , <u>ac2</u> , or <u>ac3</u> = <u>NIL</u> , 0 is used. Value of <u>jsys</u> is the (boxed) contents of the accumulator specified by <u>resultac</u> , i.e., 1 means <u>ac1</u> , 2 means <u>ac2</u> , and 3 means <u>ac3</u> , with <u>NIL</u> equivalent to 1. Compiles open if <u>n</u> is itself a small integer, and <u>resultac</u> is a small integer, or <u>NIL</u> . ¹⁹	+ + +
username[a]	If <u>a</u> = <u>NIL</u> , returns login directory name; if <u>a</u> = <u>T</u> , returns connected directory name; if <u>a</u> is a number, <u>username</u> returns the user name corresponding to that user number. In all cases, the value is a string.	+ + +
usernumber[a;-]	If <u>a</u> = <u>NIL</u> , returns login user number; if <u>a</u> = <u>T</u> , returns connected user number; if <u>a</u> is a literal atom or string, <u>usernumber</u> returns the number of the corresponding user, or <u>NIL</u> if no such user exists. ²⁰	+ + +
hostname[hostn]	returns hostname as a string for host number <u>hostn</u> , e.g. "PARC-MAXC2", "BBN-TENEXD", etc. If <u>hostn</u> = <u>NIL</u> , local host is used. If local host is not an arpanet host, value is <u>NIL</u> . Value is also <u>NIL</u> if <u>hostn</u> is not a valid host number.	+ + +
hostnumber[]	returns host numer of local host, or <u>NIL</u> , of local host is not an arpanet host.	+ +
systemtype[]	for Interlisp-10, returns either TENEX or TOPS 20.	+ +
tenex[str;fileflg]	Starts up a lower exec (without a message) using <u>subsys</u> , and then if <u>fileflg</u> = <u>NIL</u> unread <u>str</u> , followed by "QUIT" ²¹ (using <u>bksysbuf</u> , described in Section 14). The value of <u>tenex</u> is <u>T</u> if all of <u>str</u> is actually processed/read by the lower exec, <u>NIL</u> if the user control-C's and manually QUIT's back to Interlisp.	+ + + + +

-
- ¹⁸ If the JSYS causes a trap, the message TRAP AT LOCATION nnnnnn is printed by the operating system, followed by JSYS ERROR: and the operating system diagnostic. The user is then talking to the operating system exactly as though control-C had been typed. If the user then continues using the CONTINUE command, an Interlisp error is generated, JSYS ERROR, and control then proceeds the same as for any other flavor of error, i.e. unwinds to last errorset or goes into a break as described in Section 16.
- ¹⁹ The cjsys package (Section 24) enables calling jsyses by their corresponding name, rather than their number.
- ²⁰ On TOPS-20, there is a difference between the user number, which is associated with the job, and the directory number, which is associated with the file system. Therefore, on TOPS-20, usernumber takes an extra argument which, if T, says to return the directory number rather than the user number.
- ²¹ "POP" for Interlisp on TOPS-20.

+ If fileflg=T, tenex passes the string as the second argument to subsys, instead of unread it. This has the advantage that str can be of any length, and also that typeahead will not interfere with the call to the lower exec. The disadvantage is that tenex cannot tell whether the commands to the lower exec terminated successfully, or were aborted. Thus, if fileflg=T, the value of tenex is always T.

- + For example, listfiles (Section 14) is implemented using tenex, with fileflg=NIL, so listfiles can tell if listings actually were completed. The lispmacro SY, which does a SYSTAT, is implemented as TENEX["SY";T], so that the user can type ahead.

+ MANIPULATING TENEX FILE DIRECTORIES FROM INTERLISP-10

- + The following function allows the user to conveniently specify and/or program a variety of directory operations:

- + directory[filegroup;commands;defaulttext;defaultvers;-]²²

filegroup is either [1] NIL (which is equivalent to *.*;*); or [2] an atom which can contain \$'s or *'s (equivalent) which match any number of characters²³ or ?'s which match a single character, or else [3] filegroup is a list of the form (filegroup + filegroup), (filegroup - filegroup), or (filegroup * filegroup),²⁴ e.g., (T\$ + \$L) will match with any file beginning with T or ending in L, (T\$ - *.COM) matches all files that begin with T and are not .COM files.

+ For each file that matches, each command in commands is executed with the following interpretation:

- + @ fn

apply fn to the JFN for each file; if fn returns NIL, abort command processing for this file.²⁵

- + P

print file name.

- + PAUSE

wait for user to type any char (good for display if you want to ponder).

- + PROMPT mess

prompts with mess; if user responds with No, abort command processing for this file.

- + SIZE

print file size.

22 directory was written by L.M. Masinter.

23 not necessarily trailing characters, e.g., F\$1 matches FOO1 and FIE1.

24 OR can be used for +, and AND for *.

+ 25 If fn is a function of two arguments, it will be passed the name of the file as its second argument.

TRIMTO n	deletes all but <u>n</u> versions of file (<u>n</u> > 0).	+
OUT file	directs output to <u>file</u> .	+
COLLECT	adds file on value list. In this case the value of <u>directory</u> will be the list of files (complete file names) collected.	+
DATE	prints date the file was last written.	+
DELETE	deletes file.	+
	The value of <u>directory</u> is NIL if no COLLECT command is specified, otherwise the list of files "collected".	+
DELETED	prints out those files that have been deleted.	+
UNDELETE	undeletes the indicated files that have been deleted.	+

directory uses dircommands to correct spelling, which also provides a way of defining abbreviations and synonyms (see Section 17 on spelling lists). Currently the following abbreviations are recognized:

TI	same as DATE
DEL	same as DELETED
DEL?	same as PROMPT "delete?" DELETE
COLLECT?	same as PROMPT "?" COLLECT

There is also a lispmacro DIR which calls the function directory:

DIR group commands	calls the function <u>directory</u> with (P . commands) as the command list and * and * as the default extension and default version respectively.
---------------------------	--

For example, to DELVER only those files which you ok, do DIR group PROMPT "?" TRIMTO 1.

21.6 JFN FUNCTIONS IN INTERLISP-10

JFN stands for job file number. It is an integral part of the TENEX file system and is described in [Murl], and in somewhat more detail in the TENEX JSYS manual. In Interlisp-10, the following functions are available for direct manipulation of JFNs:

opnjfn[file;access]	returns the JFN for <u>file</u> . If <u>file</u> not open, generates a FILE NOT OPEN error. <u>access</u> =NIL, INPUT, OUTPUT, or BOTH as described in discussion of <u>openp</u> . For example, (JSYS 51Q (OPNJFN FILE) BYTE) will write a byte on a file,
----------------------------	---

while (JSYS 50Q (OPNJFN FILE) NIL NIL 2) will read one byte.

gtjfn[file;ext;v;flags]

sets up a "long" call to GTJFN (see JSYS manual). file is a file name possibly containing control-F and/or < esc >. ext is the default extension, v the default version (overriden if file specifies extension/version, e.g., FOO.COM;2). flags is as described on page 17, section 2 of JSYS manual. file and ext may be strings or atoms; v and flags must be numbers. Value is JFN, or NIL on errors.

rljfn[jfn]

releases jfn. rljfn[-1] releases all JFN's which do not specify open files. Value of rljfn is T.

jfns[jfn;ac3;strptr]

converts jfn (a small number) to a file name. ac3 is either NIL, meaning format the file name as would openp or other Interlisp-10 file functions, or else is a number, meaning format according to JSYS manual. The value of jfns is atomic except where enough options are specified by ac3 to exceed atom size. In this case, the value is returned as a string.

+ strptr is an optional string pointer to be reused. In this case, the string characters are stored in an internal scratch string, macscratchstring, so that a subsequent call to jfns will overwrite the characters returned by this one. The value of jfns when strptr is supplied is always a string.

The following function is available in Interlisp-10 for specialized file applications:

openf[file;x]

opens file. x is a number whose bits specify the access and mode for file, i.e., x corresponds to the second argument to the TENEX JSYS OPENF (see JSYS Manual). Value is full name of file.

The first argument to openf can also be a number, which is then interpreted as a JFN. openf does not affect the primary input or output file settings, and does not check whether the file is already open - i.e., the same file can be opened more than once, possibly for different purposes.

- + Note that for almost all applications the function openfile (Section 14) provides a more convenient
- + (and implementation independent) way of opening files.

21.7 PMAP PACKAGE²⁶

This facility allows paged access to files in Interlisp-10. It manages a set of paging buffers as a least-recently-used queue, with each buffer being a full-page block (see getblk, Section 3). Facilities are provided for allocating and deallocating buffers, locking down pages, mapping a given page of the file into core, and getting the in-core location to which a given word of the file has been mapped. Any number of files can be mapped in at one time.

The following scenario illustrates the use of these facilities: The user first opens the file (or files) that he wants to access, by page-mapping using any of the ordinary file-opening functions. Then, to examine a particular word in one of the files, the user simply gives the word number and the file's name to the function mapword, which returns a pointer to the in-core location that that word is mapped to (i.e. the address as an unboxed number). When he has finished processing, the user simply closes the file (e.g. using closef) and the buffers are automatically unmapped.

The basic functions are:

addbuffer[temp;errorflg]

Initially, a single buffer is allocated, so that page-mapping may be done without further initialization. More buffers can be allocated by addbuffer, which may help to avoid thrashing. addbuffer attempts to allocate a single new buffer, and returns T if successful. If there is not enough space to allocate a new buffer, then if errorflg is NIL, addbuffer simply returns NIL. Otherwise, addbuffer causes an error UNABLE TO ALLOCATE PMAP BUFFER.

If temp=T, the buffers are allocated on a "temporary" basis: allocation takes place via a resetsave whose restoration form will de-allocate the buffers.

mapbuffercount[onlyunlocked] value is the number of buffers currently allocated. If onlyunlocked=T, counts only unlocked buffers; otherwise, counts all buffers. Thus, to insure that at least 3 (unlocked) buffers are allocated, the user could perform (while (MAPBUFFERCOUNT T) 1t 3 do (ADDBUFFER NIL T)).

mappage[page# ;file]

the primitive function for mapping in pages from file into the queue of buffers. page# is a page number in file. The value of mappage is a pointer to the word in core at which the first word of the page is located, which will always be at a page-boundary (i.e. the pointer will print as #xxx000).

If file is NIL, the value of defaultmapfile is used. file may also be a fork handle (i.e. a value of subsys), in which case the specified page from that fork will be mapped in.

26 The PMAP package was written by R. M. Kaplan and L. M. Masinter.

- + mappage searches the buffers to see if the given page for the given file has already been mapped in. If so, it returns the core address to which it was previously mapped. Otherwise, it replaces the previous contents of the least-recently-used buffer with the specified file page. It is important to note that the contents of a given core buffer are not guaranteed across calls to mappage, unless the page has been locked down via lockmap. mappage compiles open, and in the case where the desired page is already in the buffer it is quite efficient.
- + mappage will allocate an additional buffer if no unlocked buffers are available (and the desired page is not already mapped in).
- + mapword[fileadr;file] like mappage, except that it allows the specification of a word-address in file, not just a page number. mapword determines what page that address is on, maps that page into a buffer (using mappage), and returns a pointer into the middle of the buffer where the indicated word appears. The rest of the words on the same file page appear at the appropriate wordoffsets from the value returned by mapword.
- + wordoffset[ptr;n] If ptr is a pointer into a buffer as returned by mappage or mapword, wordoffset[ptr;n] returns a pointer to the nth following word. For example, mapword could be written as an wordoffset of a mappage. wordoffset compiles open.
- + wordcontents[ptr] Returns the contents of the word at ptr as an integer. For example, wordcontents[mapword[10;file]] will return the value stored in word 10 of a (binary) file. wordcontents[ptr] is equivalent to openr[loc[ptr]]. wordcontents compiles open.
- + setwordcontents[ptr;n] Sets the contents of the word pointed to by ptr to be the number n. Interpreted, setwordcontents checks that ptr actually is a pointer as returned by mappage or mapword. setwordcontents compiles open with no error checks.
- + clearmap[file;pages;release] file specifies a file or fork as for mappage, or it is T. pages is a single page number or a list of page numbers. clearmap unmaps any of those pages that are currently mapped in,²⁷ making those buffers available for other mappings. file=T means all files; pages= NIL means all pages. Thus clearmap[T] will completely clear the buffers.

+ ²⁷ whether or not they are currently locked, i.e. clearmap takes precedence over lockmap.

	If <u>release</u> =T, then not only will the buffers containing the specified pages be unmapped, but the buffers themselves will be released, i.e. returned to the Interlisp storage manager.	+ + +
lockmap[ptr]	For those situations in which a program needs prolonged access to a particular file page, <u>lockmap</u> can be used to prevent <u>mappage</u> from shifting or unmapping the contents of the given core page. <u>ptr</u> is a pointer into a mapped page (i.e. a value of <u>mapword</u> or <u>mappage</u>). <u>lockmap</u> locks the indicated page in core until a corresponding <u>unlockmap</u> has been performed. ²⁸ Value is <u>ptr</u> .	+ + + + + +
unlockmap[ptr;flg]	<u>ptr</u> is a pointer into a mapped page. <u>unlockmap</u> removes the most recent lock for that page if <u>flg</u> =NIL, and all locks if <u>flg</u> =T.	+ +

21.8 TYPESCRIPT FILES

A typescript file is a "transcript" of all of the input and output on a terminal. The following function enables transcript files for Interlisp.

dribble[filename;appendflg;]²⁹ Opens filename and begins recording the typescript. Value is old typescript file if any, otherwise NIL. If appendflg=T, the typescript will be appended to the end of filename.³⁰ dribble[] closes the typescript file.³¹

dribble processes a line buffer at a time. Thus, the typescript produced is somewhat neater than that generated by TELNET because it does *not* show characters that were erased via control-A or control-Q. Note that the typescript file is *not* included in the list of files returned by openp[], nor will it be closed by a call to closeall or closef. Only dribble[] closes the typescript file.

dribblefile[] returns name of current typescript file, if any, otherwise NIL.

21.9 DISPLAY TERMINALS

The value of the variable displaytermflg indicates whether the user is running on a display terminal or not. displaytermflg is used in various places in the system, e.g., prettyprint, helpsys, etc.

²⁸ If a page has been locked twice, it must be unlocked twice before it is available for reuse.

²⁹ dribble was written by D. C. Lewis.

³⁰ dribble also takes an extra argument, thawedflg. If thawedflg=T, the file will be opened in "thawed" mode.

³¹ Only one typescript file can be active at any one point; i.e., dribble[file1] followed by dribble[file2] will cause file1 to be closed.

primarily to decide how much information to present to the user (more on a display terminal than on a hard copy terminal.) displaytermflg is initialized to the value of displaytermpl[], whenever Interlisp is (re)-entered, and after returning from a sysout.

displaytermpl[]

value is T if user is on a display terminal, NIL otherwise. In Interlisp-10, displaytermpl is defined to invoke the appropriate jsys to check the user's terminal type.

21.10 GAINSPACE

For users with large programs and data bases, the user may sometimes find himself in a situation where he needs to obtain more space, and is willing to pay the price of eliminating some or all of the context information that the various user-assistance facilities such as the programmer's assistant, file package, CLISP, etc., have accumulated during the course of his session. The following function is available for this purpose.

gainspacepl[]

walks the user through a menu, prompting him at each point, and allowing him to specify what can be discarded and what should be retained, e.g.:

GAINSPACE()

purge history lists ? Yes

purge everything, or just the properties, e.g., SIDE, LISPXPRINT, etc.?
just the properties

discard definitions on property lists ? Yes

discard old values of variables ? Yes

erase filepkg information ? No

erase properties ? No

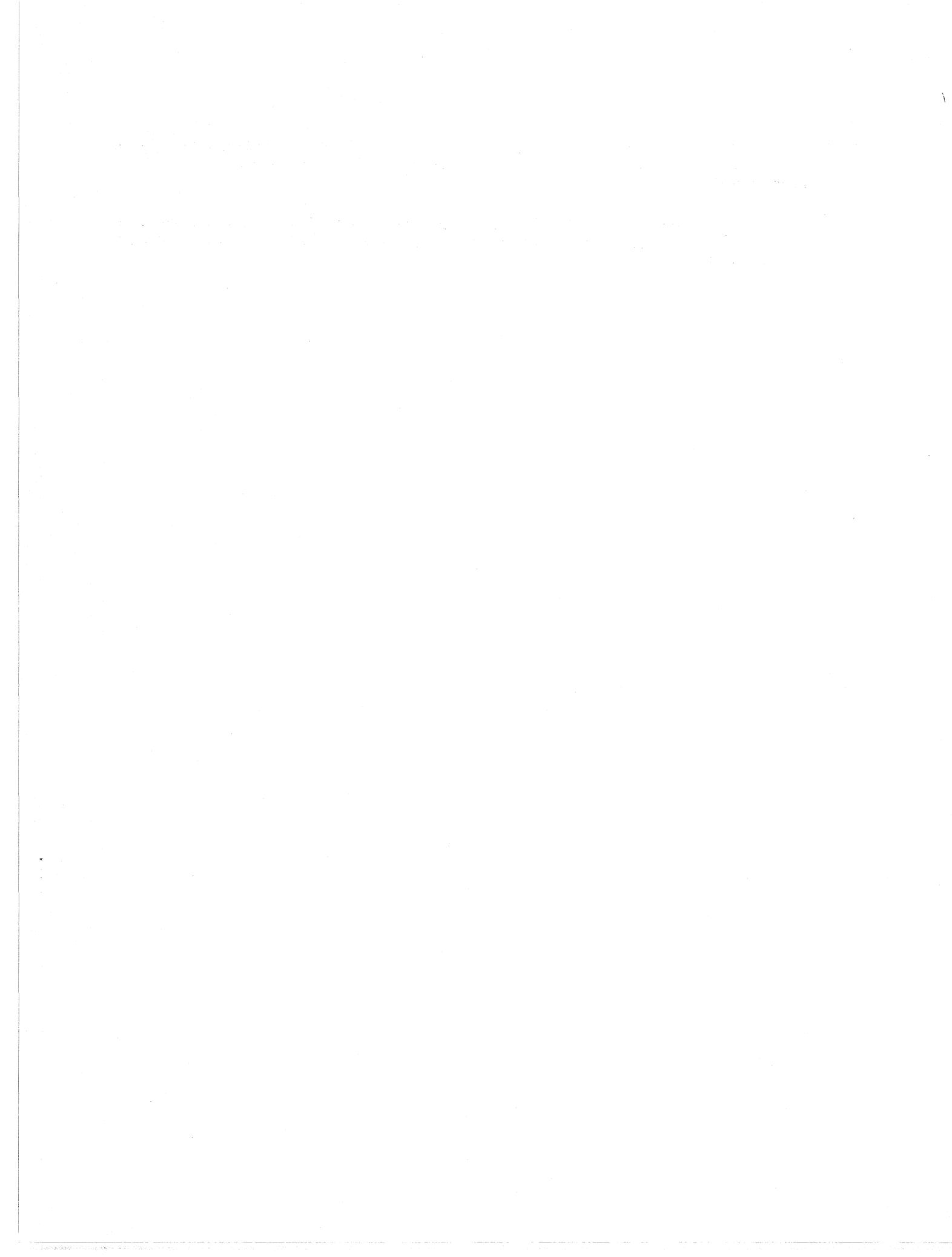
etc.

gainspace is driven by the list gainspaceforms. Each element on gainspaceforms is of the form (precheck message form keylst). If precheck, when evaluated, returns NIL, gainspace skips to the next entry. For example, the user will not be asked whether or not to discard edita initialization if he has not initialized edita. Otherwise, askuser (Section 17) is called with the indicated message and the (optional) keylst. If the user responds No, i.e., askuser returns N, gainspace skips to the next entry. Otherwise, form is evaluated with the variable response bound to the value of askuser, e.g., in the above example, if the user had responded with Everything, instead of Yes, to the "purge history lists" question, he would not have been asked the second question, because the form for this entry checks to see whether the value of response is Y or E.

The "erase properties" entry on gainspaceforms is driven by a list smashpropsmenu. Each element on this list is of the form (message . props). The user is prompted with message (by askuser), and if he responds Yes, props is added to the list smashprops. The "discard definitions on property lists" and "discard old values of variables" entries on gainspaceforms also add to smashprops. The user will not be prompted for any entry on smashpropsmenu for which all of the corresponding properties are already on smashprops. smashprops is initially set to the value of smashpropslst. This permits the user to specify in advance those properties which he always wants to be discarded, and not be asked about them subsequently.

After finishing all the entries on gainspaceforms, gainspace checks to see if the value of smashprops is non-NIL, and if so, does a mapatoms, i.e., looks at every atom in the system, and erases the indicated properties.

Note that the user can change the entries on gainspaceforms or smashpropsmenu, and/or add new entries to either, so that gainspace can also be used to purge structures that the user's programs have accumulated.



Index for Section 21

	Page Numbers
ADDBUFFER[TEMP;ERRORFLG]	21.13
BKSYSBUF[X] SUBR	21.9
BOXCOUNT[TYPE;N] SUBR	21.3
BREAKDOWN[FNS] NL*	21.4-6
BRKDWNRESULTS[RETURNVALUESFLG]	21.4
BRKDWNTYPE (system variable/parameter)	21.5
BRKDWTYPES (system variable/parameter)	21.5
CLEARMAP[FILE;PAGES;RELEASE]	21.14
CLOCK[N] SUBR	21.3
CONSCOUNT[N] SUBR	21.3
CONTIN (prog. asst. command)	21.8
CONTINUE (TENEX command)	21.4
control-B	21.3
control-C	21.7
control-D	21.3
control-E	21.3
COREVAL[X] NL	21.3
DATE[FORMATBITS] SUBR	21.2
DIR (prog. asst. command)	21.11
DIRECTORY[FILEGROUP;COMMANDS;DEFALTEXT; DEFAULTVERS;LISPXPRNTFLG]	21.10
DISMISS[N]	21.3
DISPLAYTERMFLG (system variable/parameter)	21.15
DISPLAYTERMP[]	21.16
DРИBBLE[FILE;APPENDFLG;THAWEDFLG]	21.15
DРИBBLEFILE[] SUBR	21.15
ERSTR[ERN]	21.8
EXEC (prog. asst. command)	21.8
FILDIR[FILEGROUP;FORMATFLG]	21.8
FILE NOT OPEN (error message)	21.11
fork handle	21.8
forks	21.6
GAINSPACE[]	21.16
GAINSPACEFORMS (system variable/parameter)	21.16
GCTR[P] SUBR	21.4
GDATE[DATE;FORMATBITS;STRPTR]	21.2
GETBLK[N] SUBR	21.6
GTJFN[FILE;EXT;V;FLAGS]	21.12
HOSTNAME[HOSTN]	21.9
HOSTNUMBER[]	21.9
IDATE[D]	21.2
ILLEGAL OR IMPOSSIBLE BLOCK (error message)	21.6-7
interfork communication	21.6
JFN	21.11-12
JFNS[JFN;AC3;STRPTR]	21.12
JSYS[N;AC1;AC2;AC3;RESULTAC] SUBR	21.9
JSYS	21.8,11-12
JSYS ERROR (error message)	21.9
KFORK[FORK]	21.8,8
LISPXMACROS (prog. asst. variable/parameter)	21.8
LOADAV[]	21.8
LOCKMAP[PTR]	21.15
LOGOUT[] SUBR	21.4,8
MACSCRATCHSTRING (system variable/parameter)	21.3,12
MAPBUFFERCOUNT[ONLY]	21.13
MAPPAGE[PAGE#;FILE]	21.13

Page
Numbers

MAPWORD[FILEADR;FILE]	21.14
OPENF[FILE;X] SUBR	21.12
OPNJFN[FILE;TYPE] SUBR	21.11
page mapping	21.13-15
PAGEFAULTS[]	21.4
pmap package	21.13-15
QUIT (TENEX command)	21.7,8
RELBLK[ADDRESS;N] SUBR	21.6
RESULTS[RETURNVALUESFLG]	21.6
RLJFN[JFN]	21.12
running other subsystems from within Interlisp ..	21.7
SETWORDCONTENTS[PTR;N]	21.14
SUBSYS[FILE/FORK;INCOMFILE;OUTCOMFILE; ENTRYPOINTFLG]	21.7,8
SYSTEMTYPE[] SUBR	21.9
TELNET[CONNECTION;TYPE;SKT;BUTTONCHAR]	21.15
TENEX	21.11-12
TENEX[STR;FILEFLG]	21.9
TIME[FORM;CNT] NL	21.1,2
TRAP AT LOCATION (error message)	21.9
typescript files	21.15
UNABLE TO ALLOCATE PMAP BUFFER (error message) ..	21.13
UNBREAK[X] NL*	21.5
UNLOCKMAP[PTR]	21.15
USERNAME[A]	21.9
USERNUMBER[A;FLG]	21.9
VALUEOF[X] NL*	21.8
WORDCONTENTS[PTR]	21.14
WORDOFFSET[PTR;N]	21.14

SECTION 22

THE PROGRAMMER'S ASSISTANT¹

22.1 INTRODUCTION

The central idea of the programmer's assistant is that the user, rather than talking to a passive system which merely responds to each input and waits for the next, is instead addressing an active intermediary, namely his assistant. Normally, the assistant is invisible to the user, and simply carries out the user's requests. However, since the assistant remembers what the user has told him, the user can instruct him to repeat a particular operation or sequence of operations, with possible modifications, or to undo the effect of certain specified operations. Like DWIM, the programmer's assistant is not implemented as a single function or group of functions, but is instead dispersed throughout much of Interlisp.² Like DWIM, the programmer's assistant embodies a philosophy and approach to system design whose ultimate goal is to construct a programming environment which would "cooperate" with the user in the development of his programs, and free him to concentrate more fully on the conceptual difficulties and creative aspects of the problem he is trying to solve.

EXAMPLE

The following dialogue, taken from an actual session at the console, gives the flavor of the programmer's assistant facility in Interlisp. The user is about to edit a function loadf, which contains several constructs of the form (PUTD FN2 (GETD FN1)). The user plans to replace each of these by equivalent MOVD expressions.

¹ The programmer's assistant was designed and implemented by W. Teitelman. It is discussed in [Tei4].

² Some of the features of the programmer's assistant have been described elsewhere, e.g., the UNDO command in the editor, the file package, etc.

```

←EDITF(LOADFF) [1]
=LOADF
EDIT
*PP
[LAMBDA (X Y)
[COND
  ((NULL (GETD (QUOTE READSAVE)))
   (PUTD (QUOTE READSAVE)
         (GETD (QUOTE READ]
        (PUTD (QUOTE READ)
              (GETD (QUOTE REED))))
      (NLSETQ (SETQ X (LOAD X Y))))
     (@UTD (QUOTE READ)
           (GETD (QUOTE READSAVE)))
    X]

*F PUTD (1 MOVD) [2]
*3 (XTRR 2) [3]
=XTR
*OP [4]
=0 P
(MOVD (QUOTE READSAVE) (QUOTE READ))
*(SW 2 3) [5]
*
```

At [1], the user begins to edit loadf.³ At [2] the user finds PUTD and replaces it by MOVD. He then shifts context to the third subexpression, [3], extracts its second subexpression, and ascends one level [4] to print and result. The user now switches the second and third subexpression [5], thereby completing the operation for this PUTD. Note that up to this point, the user has not directly addressed the assistant. The user now requests that the assistant print out the operations that the user has performed, [6], and the user then instructs the assistant to REDO FROM F, [7], meaning repeat the entire sequence of operations 15 through 20. The user then prints the current expression, and observes that the second PUTD has now been successfully transformed.

```

*?? FROM F [6]
15. *F PUTD
16. *(1 MOVD)
17. *3
18. *(XTR 2)
19. *0
20. *(SW 2 3)

*REDO FROM F [7]
*P
(MOVD (QUOTE REED) (QUOTE READ))
*
```

³ We prefer to consider the programmer's assistant as the moving force behind this type of spelling correction (even though the program that does the work is part of the DWIM package). Whereas correcting @PRINT to PRINT, or XTRR to XTR does not require any information about what *this* user is doing, correcting LOADFF to LOADF clearly required noticing when this user defined loadf.

The user now asks the assistant to replay the last three steps to him, [8]. Note that the entire REDO FROM F operation is now grouped together as a single unit, [9], since it corresponded to a single user request. Therefore, the user can instruct the assistant to carry out the same operation again by simply saying REDO. This time a problem is encountered [10], so the user asks the assistant what it was trying to do [11].

*?? FROM -3

[8]

19. *0
 20. *(SW 2 3)
 21. REDO FROM F
 *F PUTD
 *(1 MOVD)
 *3
 *(XTR 2)
 *0
 *(SW 2 3)

[9]

*REDO

PUTD ?

[10]

*?? -1

[11]

22. REDO
 *F PUTD
 *(1 MOVD)
 *3
 *(XTR 2)
 *0

The user then realizes the problem is that the third PUTD is misspelled in the definition of LOADF (see page 22.2). He therefore instructs the assistant to USE @UTD FOR PUTD, [12], and the operation now concludes successfully.

*USE @UTD FOR PUTD
 *P
 (MOVD (QUOTE READSAVE) (QUOTE READ))
 *↑ PP
 [LAMBDA (X Y)
 [COND
 ((NULL (GETD (QUOTE READSAVE)))
 (MOVD (QUOTE READ)
 (QUOTE READSAVE])
 (MOVD (QUOTE REED)
 (QUOTE READ))
 (NLSETQ (SETQ X (LOAD X Y)))
 (MOVD (QUOTE READSAVE)
 (QUOTE READ))
 X]
 *OK
 LOADF
 ←

[12]

An important point to note here is that while the user *could* have defined a macro to execute this operation, the operation is sufficiently complicated that he would want to try out the individual

steps before attempting to combine them. At this point, he would already have executed the operation once. Then he would have to type in the steps again to define them as a macro, at which point the operation would only be repeated once more before failing. Then the user would have to repair the macro, or else change @UTD to PUTD by hand so that his macro would work correctly. It is far more natural to decide *after* trying a series of operations whether or not one wants them repeated or forgotten. In addition, frequently the user will think that the operation(s) in question will never need be repeated, and only discover afterwards that he is mistaken, as occurs when the operation was incorrect, but salvageable:

```
*P
(LAMBDA (STR FLGCQ VRB) **COMMENT** (PROG & & LP1 & LP2 & &))
*-1 -1 P
(RETURN (COND &))
*(-2 ((EQ BB (QUOTE OUT)) BB)] [1]
*P
(RETURN (& BB) (COND &))
*UNDO
(-2 --) UNDONE
*2 P
(COND (EXPANS & & T))
*REDO EQ
*P
(COND (& BB) (EXPANS & & T))
*
```

Here the operation was correct, [1], but the context in which it was executed, [2], was wrong.

This example also illustrates one of the most useful functions of the programmer's assistant: its UNDO capability. In most systems, if a user suspected that a disaster might result from a particular operation, e.g., an untested program running wild and chewing up a complex data structure, he would prepare for this contingency by saving the state of part or all of his environment before attempting the operation. If anything went wrong, he would then back up and start over. However, saving/dumping operations are usually expensive and time consuming, especially compared to a short computation, and are therefore not performed that frequently, and of course there is always the case when disaster strikes as a result of a "debugged" or at least innocuous operation, as shown in the following example:

```
←(MAPC ELTS (FUNCTION (LAMBDA (X) (REMPROP X (QUOTE MORPH)] [1]
NIL
←UNDO [2]
MAPC UNDONE.
←USE ELEMENTS FOR ELTS [3]
NIL
←
```

The user types an expression which removes the property MORPH from every member of the list ELTS [1], and then realizes that he meant to remove that property only from those members of the list ELEMENTS, a much shorter list. In other words, he has deleted a lot of information that he actually wants saved. He therefore simply reverses the effect of the MAPC by typing UNDO [2], and then does what he intended via the USE command [3].

22.2 OVERVIEW

The programmer's assistant facility is built around a memory structure called the "history list." The

history list is a list of the information associated with each of the individual "events" that have occurred in the system, where each event corresponds to one user input.⁴ For example, (XTR 2) ([3] on page 22.2) is a single event, while REDO FROM F ([7] on page 22.2) is also a single event, although the latter includes executing the operation (XTR 2), as well as several others.

Associated with each event on the history list is its input and its value, plus other optional information such as side-effects, formatting information, etc. If the event corresponds to a history command, e.g., REDO FROM F, the input corresponds to what the user would have had to type to execute the same operation(s), although the user's actual input, i.e., the history command, is saved in order to clarify the printout of that event ([9] on page 22.3). Note that if a history command event combines several events, it will have more than one value:

⁴ For various reasons, there are *two* history lists: one for the editor, and one for lispx, which processes inputs to evalqt and break, see page 22.34.

```

←(LOG (ANTILOG 4))
4.0
←USE 4.0 40 400 FOR 4
4.0
40.0
ARG NOT IN RANGE
400

←USE -40.0 -4.00007 -19.
-40.0
-4.00007
-19.0
←USE LOG ANTILOG FOR ANTILOG LOG IN -2 AND -1
4.0
40.0
400.0
4.00007
19.0
←??
4. USE LOG ANTILOG FOR ANTILOG LOG IN -2 -1
←(ANTILOG (LOG 4.0))
4.0
←(ANTILOG (LOG 40))
40.0
←(ANTILOG (LOG 400))
400.0
←(ANTILOG (LOG -40.0))
40.0
←(ANTILOG (LOG -4.00007))
4.00007
←(ANTILOG (LOG -19.0))
19.0
3. USE -40.0 -4.00007 -19.0
←(LOG (ANTILOG -40.0))
-40.0
←(LOG (ANTILOG -4.00007))
-4.00007
←(LOG (ANTILOG -19.0))
-19.0
2. USE 4.0 40 400 FOR 4
←(LOG (ANTILOG 4.0))
4.0
(LOG (ANTILOG 40.0))
40.0
←(LOG (ANTILOG 400))
1. ←(LOG (ANTILOG 4))
4.0

```

As new events occur, existing events are aged, and the oldest event is "forgotten." For efficiency, the storage used to represent the forgotten event is cannibalized and reused in the representation of the new event, so the history list is actually a ring buffer. The size of this ring buffer is a system

parameter called the 'time-slice'.⁵ Larger time-slices enable longer "memory spans," but tie up correspondingly greater amounts of storage. Since the user seldom needs really "ancient history," and a NAME and RETRIEVE facility is provided for saving and remembering selected events, a relatively small time slice such as 30 events is more than adequate, although some users prefer to set the time slice as large as 100 events.

Events on the history list can be referenced in a number of ways. The output on page 22.8 shows a printout of a history list with time-slice 16. The numbers printed at the left of the page are the event numbers. More recent events have higher numbers; the most recent event is event number 52, the oldest and about-to-be-forgotten event is number 37.⁶ At this point in time, the user can reference event number 51, RECOMPILE(EDIT), by its event number, 51; its relative position, -2 (because it occurred two events back from the current time), or by a "description" of its input, e.g., (RECOMPILE (EDIT)), or (& (EDIT)), or even just EDIT. As new events occur, existing events retain their absolute event numbers, although their relative positions change.

Similarly, descriptor references may require more precision to refer to an older event. For example, the description RECOMPILE would have sufficed to refer to event 51 had event 52, also containing a RECOMPILE, not intervened. Event specification will be described in detail later.

⁵ Initially 30 events. The time-slice can be changed with the function changeslice, page 22.40.

⁶ When the event number of the current event is 100, the next event will be given number 1. (If the time slice is greater than 100, the "roll-over" occurs at the next highest hundred, so that at no time will two events ever have the same event number. For example, if the time slice is 150, event number 1 follows event number 200.)

```

←??
52. ... HIST UNDO
    ←RECOMPILE(HIST)
    ST
    HIST.COM
    ←RECOMPILE(UNDO)
    ST
    UNDO.COM
51. ←RECOMPILE(EDIT)
    ST
    EDIT.COM
50. ←LOGOUT]

49. ←MAKEFILES]
    (EDIT UNDO HIST)
48. ←EDITF(UNDOLISPX)
    UNDOLISPX
47. REDO GETD
    ←GETD(FIE)
    (LAMBDA (X) (MAPC X (F/L (PRINT X))))
46. ←UNDO
    FIE
45. ←GETD(FIE)
    (LAMBDA (X) (MAPC X (FUNCTION (LAMBDA (X) (PRINT X)))))
44. ←FIE]
    NIL
43. ←DEFINEQ((FIE (LAMBDA (X) (MAPC X (F/L (PRINT X))))))
    (FIE)
42. REDO GETD
    ←GETD(FIE)
    (LAMBDA (Y) Y)
41. ←UNDO
    MOVD
40. REDO GETD
    ←GETD(FIE)
    (LAMBDA (X) X)
39. ←MOVD(FOO FIE)
    FIE
38. ←DEFINEQ((FOO (LAMBDA (X) X)))
    (FOO)
37. ←GETD(FIE)
    (LAMBDA (Y) Y)

```

The most common interaction with the programmer's assistant occurs at the top level evalqt, or in a break, where the user types in expressions for evaluation, and sees the values printed out. In this mode, the assistant acts much like a standard LISP evalqt, except that before attempting to evaluate an input, the assistant first stores it in a new entry on the history list. Thus if the operation is aborted or causes an error, the input is still saved and available for modification and/or reexecution. The assistant also notes new functions and variables to be added to its spelling lists to enable future corrections. Then the assistant executes the computation (i.e., evaluates the form or applies the function to its arguments), saves the value in the entry on the history list

corresponding to the input, and prints the result,⁷ followed by a prompt character to indicate it is again ready for input.⁸

If the input typed by the user is recognized as a history command, the assistant takes special action.⁹ Commands such as UNDO, ??, NAME, and RETRIEVE are immediately performed. Commands that involved reexecution of previous inputs, e.g., REDO and USE, are achieved by computing the corresponding input expression(s) and then *unreading* them. The effect of this unreading operation is to cause the assistant's input routine, lispread, to act exactly as though these expression were typed in by the user. Except for the fact that these inputs are not saved on new and separate entries on the history list, but associated with the history command that generated them, they are processed exactly as though they had been typed.

The advantage of this implementation is that it makes the programmer's assistant a callable facility for other system packages as well as for users with their own private executives. For example, break1 accept user inputs, recognizes and executes certain break commands and macros, and interprets anything else as Interlisp expressions for evaluation. To interface break1 with the programmer's assistant required three small modifications to break1: (1) input was to be obtained via lispread instead of read; (2) instead of calling eval or apply directly, break1 was to give those inputs it could not interpret to lisp, and (3) any commands or macros handled by break1, i.e., not given to lisp, were to be stored on the history list by break1 by calling the function historysave, a part of the assistant package.

Thus when the user typed in a break command, the command would be stored on the history list as a result of (3). If the user typed in an expression for evaluation, it would be evaluated as before, with the expression and its value both saved on the history list as a result of (2). Now if the user entered a break and typed three inputs: EVAL, (CAR !VALUE), and OK, at the next break, he could achieve the same effect by typing REDO FROM EVAL. This would cause the assistant to unread the three expressions EVAL, (CAR !VALUE), and OK. Because of (1), the next "input" seen by break1 would then be EVAL, which break1 would interpret. Next would come (CAR !VALUE), which would be given to lisp to evaluate, and then would come OK, which break1 would again process. Thus, by virtue of *unreading*, history operations will work even for those inputs not interpretable by lisp, in this case, EVAL and OK.

The net effect of this implementation of the programmer's assistant is to provide a facility which is easily inserted at many levels, and embodies a consistent set of commands and conventions for talking about past events. This gives the user the subjective feeling that a single agent is watching everything he does and says, and is always available to help.

⁷ using showprint (Section 14), so that if the value of sysprettyflg = T, the value will be prettyprinted. +

⁸ The function that accepts a user input, saves the input on the history list, performs the indicated computation or history command, and prints the result, is lisp. lisp is called by evalqt and break1, and in most cases, is synonymous with "programmer's assistant." However, for various reasons, the editor saves its own inputs on a history list, carries out the requests, i.e., edit commands, and even handles undoing independently of lisp. The editor only calls lisp to execute a history command, such as REDO, USE, etc. Therefore we use the term assistant (loosely) when the discussion applies to features shared by evalqt, break and the editor, and the term lisp when we are discussing the specific function. +

⁹ If the user defines a function by the same name as a history command, a warning message is printed to remind him that the history command interpretation will take precedence for type-in. +

22.3 EVENT SPECIFICATION

All history commands use the same conventions and syntax for indicating which event or events on the history list the command refers to, even though different commands may be concerned with different aspects of the corresponding event(s), e.g., side-effects, value, input, etc. Therefore, before discussing the various history commands in the next section, this section describes the types of event specifications currently implemented. All examples refer to the history list on page 22.8.

An event address identifies one event on the history list. It consists of a sequence of "commands" for moving an imaginary cursor up or down the history list, much in the manner of the arguments to the @ command in break (see Section 15). The event identified is the one "under" the imaginary cursor when there are no more commands. (If any command fails, an error is generated and the history command is aborted.)

The commands are interpreted as follows:

- n ($n \geq 1$)** move forward n events, i.e., in direction of increasing event number. If given as the first "command," n specifies the event with event number n .
- n ($n \leq -1$)** move backward $-n$ events.
- \leftarrow atom** specifies an event whose *function* matches atom (i.e., for *apply* format only), e.g., whereas FIE would refer to event 47, \leftarrow FIE would refer to event 44. Similarly, ED\$¹⁰ would specify event 51, whereas \leftarrow ED\$ event 48.
- \leftarrow** next search is to go forward instead of backward, (if given as the first "command", next search begins with last, i.e., oldest, event on history list), e.g., \leftarrow LAMBDA refers to event 38; MAKEFILES \leftarrow RECOMPILE refers to event 51.
- F** next object is to be searched for, regardless of what it is, e.g., F -2 looks for an event containing a -2.
- =** next object (presumably a pattern) is to be matched against *values*, instead of inputs, e.g., = UNDO refers to event 49; 45 = FIE refers to event 43; \leftarrow = LAMBDA refers to event 37.
- ** specifies the event last located.

¹⁰ i.e., ED <esc>.

SUCHTHAT pred

specifies an event for which pred, a function of two arguments, when given the input portion of the event as its first argument, and the event itself as its second argument, returns true. E.g., SUCHTHAT (LAMBDA (X Y) (MEMB (QUOTE *ERROR*) Y)) specifies an event in which an error occurred.¹¹

atom

where atom is the name of a command defined via the NAME command (page 22.21),¹² specifies the event(s) defining atom.

pat

anything else specifies an event whose input contains an expression that matches pat as described in Section 9.¹³

Note: each search skips the current event, i.e., each command always moves the cursor. For example, if FOO refers to event n, FOO FIE will refer to some event before event n, even if there is a FIE in event n.

An event specification specifies one or more events:

FROM #1 THRU #2
#1 THRU #2

the sequence of events from the event with address #1 through event with address #2,¹⁴ e.g., FROM GETD THRU 49 specifies events 47, 48, and 49. #1 can be more recent than #2, e.g., FROM 49 THRU GETP specifies events 49, 48, and 47 (note reversal of order).

FROM #1 TO #2
#1 TO #2

Same as THRU but does not include event #2.

FROM #1

Same as FROM #1 THRU -1, e.g., FROM 49 specifies events 49, 50, 51, and 52.

THRU #2

Same as FROM -1 THRU #2, e.g., THRU 49 specifies events 52, 51, 50, and 49. Note reversal of order.

TO #2

Same as FROM -1 TO #2.

¹¹ See page 22.34 for discussion of the format of events on the history list.

¹² If FOO is such a command, but the user wants to specify the event containing FOO, he can still use the event specification F FOO.

¹³ The matching is performed by the function historymatch (page 22.40), which is initially defined to call editfindp but can be advised or redefined for specialized applications.

¹⁴ i.e., the symbol #1 corresponds to all words between FROM and THRU in the event specification, and #2 to all words from THRU to the end of the event specification. For example, in FROM FOO 2 THRU FIE -1, #1 is (FOO 2), and #2 is (FIE -1).

#1 AND #2 AND ... AND #n

i.e., a sequence of event specifications separated by AND's, e.g., FROM 47 TO LOGOUT would be equivalent to 47 AND 48 AND MAKEFILES.

ALL #1

specifies all events satisfying #1, e.g., ALL LOAD, ALL SUCHTHAT FOO.

empty

i.e., nothing specified, same as -1, unless last event was an UNDO, in which case same as -2.¹⁵

@@ \$

\$ is an event specification and interpreted as above, but with respect to the archived history list, as specified on page 22.23.

If no events can be found that satisfy the event specification, spelling correction on each word in the event specification is performed using lispfindsplst as a spelling list, e.g., REDO 3 THRUU 6 will work correctly. If the event specification still fails to specify any events after spelling correction, an error is generated.

22.4 HISTORY COMMANDS

All history commands can be input as either lists, or as lines (see readline, Section 14, and also page 22.35).

\$ is used to denote an event specification. Unless specified otherwise, \$ omitted is the same as \$=-1, e.g., REDO and REDO -1 are the same.

REDO \$

redoing the event or events specified by \$, e.g., REDO FROM -3 redoing the last three events.

REDO \$ N TIMES

redoing the event or events specified by \$ N times, e.g., REDO 10 TIMES redoing the last event ten times.

REDO \$ WHILE form

redoing the specified events as long as the value of form is true. form is evaluated before each iteration so if its initial value is NIL, nothing will happen.

REDO \$ UNTIL form

same as REDO \$ WHILE (NOT form).

¹⁵ For example, if the user types (NCONC FOO FIE), he can then type UNDO, followed by USE NCONC1.

REPEAT ϕ same as REDO & WHILE T, i.e., the event(s) are repeated until an error occurs, or user types control-E or control-D.

REPEAT ϕ WHILE/UNTIL form
same interpretation as REDO.

For all history commands that perform multiple repetitions, the variable redocnt is initialized to 0 and incremented each iteration. If the event terminates gracefully, i.e., is not aborted by an error or control-D, the number of iterations is printed.

USE exprs FOR args IN ϕ substitutes exprs for args in ϕ , and redoing the result, e.g.,
USE LOG ANTILOG FOR ANTILOG LOG IN -2 AND -1.
Substitution is done by esubst, Section 9, and is carried out as described below. exprs and args can include non-atomic members.

USE $exprs_1$ FOR $args_1$ AND ... AND $exprs_n$ FOR $args_n$ IN ϕ
More general form of USE command. See description of substitution algorithm below.

Every USE command involves three pieces of information: the expressions to be substituted, the arguments to be substituted for, and an event specification, which defines the expression (input) in which the substitution takes place.¹⁶

Any expression to be substituted can be preceded by a !, meaning that the expression is to be substituted as a *segment*, e.g., LIST(A B C) followed by USE ! (X Y Z) FOR B will produce (A X Y Z C), and USE ! NIL FOR B will produce (A C).

If args are omitted, i.e., the form of the command is USE exprs IN ϕ , or just USE exprs (which is equivalent to USE exprs IN -1), and the event referred to was itself a USE command, the arguments and expression substituted into are the same as for the indicated USE command. In effect, this USE command is thus a continuation of the previous USE command. For example, on page 22.6, when the user types (LOG (ANTILOG 4)), followed by USE 4.0 40 400 FOR 4, followed by

USE -40.0 -4.00007 -19., the latter command is equivalent to
USE -40.0 -4.00007 -19. FOR 4 IN -2.

If args are omitted and the event referred to was *not* a USE command, substitution is for the operator in that command, i.e., if a lispx input, the name of the function, if an edit command, the name of the command. For example ARGLIST(FF) followed by USE CALLS is equivalent to USE CALLS FOR ARGLIST.

If IN ϕ is omitted, but args are specified, the first member of args is used for ϕ , e.g., USE PUTD

¹⁶ The USE command is parsed by a small finite state parser to distinguish the expressions and arguments. For example, USE FOR FOR AND AND AND FOR FOR will be parsed correctly.

FOR @UTD is equivalent to USE PUTD FOR @UTD IN F @UTD.¹⁷

If the USE command has the same number of expressions as arguments, the substitution procedure is straightforward,¹⁸ i.e., USE X Y FOR U V means substitute X for U and Y for V, and is equivalent to USE X FOR U AND Y FOR V. However, the USE command also permits distributive substitutions, i.e., substituting several expressions for the same argument. For example, USE A B C FOR X means first substitute A for X then substitute B for X (in a new copy of the expression), then substitute C for X. The effect is the same as three separate USE commands. Similarly, USE A B C FOR D AND X Y Z FOR W is equivalent to USE A FOR D AND X FOR W, followed by USE B FOR D AND Y FOR W, followed by USE C FOR D AND Z FOR W. USE A B C FOR D AND X FOR Y¹⁹ also corresponds to three substitutions, the first with A for D and X for Y, the second with B for D, and X for Y, and the third with C for D, and again X for Y. However, USE A B C FOR D AND X Y FOR Z is ambiguous and will cause an error. Essentially, the USE command operates by proceeding from left to right handling each "AND" separately. Whenever the number of expressions exceeds the number of expressions available, the expressions multiply.²⁰

FIX &

puts the user in the editor looking at a copy of the input(s) for &. Whenever the user exits via, OK, the result is unread and reexecuted exactly as with REDO.

FIX is provided for those cases when the modifications to the input(s) are not of the type that can be specified by USE, i.e., not substitutions. For example:

```
←(DEFINEQ FOO (LAMBDA (X) (FIXSPELL SPELLINGS2 X 70])
INCORRECT DEFINING FORM
FOO
←FIX
EDIT
*P
(DEFINEQ FOO (LAMBDA & &))
*(LI 2)
*OK
(FOO)
←
```

¹⁷ The F is inserted to handle correctly the case where the first member of args is a number, e.g., USE 4.0 4.0 400 FOR 4. Obviously the user means find the event containing a 4 and perform the indicated substitutions, whereas USE 4.0 40 400 FOR 4 IN 4 would mean perform the substitutions in event *number 4*.

¹⁸ Except when one of the arguments and one of the expressions are the same, e.g., USE X Y FOR Y X, or USE X FOR Y AND Y FOR X. This situation is noticed when parsing the command, and handled correctly.

¹⁹ or USE X FOR Y AND A B C FOR D.

²⁰ Thus USE A B C D FOR E F means substitute A for E at the same time as substituting B for F, then in another copy of the indicated expression, substitute C for E and D for F. Note that this is also equivalent to USE A C FOR E AND B D FOR F.

The user can also specify the edit command(s) to lispX, by typing - followed by the command(s) after the event specification, e.g., FIX - (LI 2). In this case, the editor will not type EDIT, or wait for an OK after executing the commands.

IMPLEMENTATION OF REDO, USE, AND FIX

The input portion of an event is represented internally on the history list simply as a linear sequence of the expressions which were read. For example, an input in apply format is a list consisting of two expressions, and an input in eval format is a list of just one expression.²¹ Thus if the user wishes to convert an input in apply format to eval format, he simply moves the function name inside of the argument list:

```

←MAPC(FOOFNS (F/L (AND (EXPRP X) (PRINT X]
NIL
←EXPRP(F001)
T
←FIX MAPC
EDIT
*p
(MAPC (FOOFNS &) "<c.r.>")22
*(MOVE 1 TO BEFORE 2 1)
*p
((MAPC FOOFSN &) "<c.r.>")
*OK
F001
FIE2
FUM
NIL
←

```

By simply converting the input from two expressions to one expression, the desired effect, that of mapping down the list that was the *value* of foofns, was achieved.

REDO, **USE**, and **FIX** all operate by obtaining the input portion of the corresponding event, processing the input (except for **REDO**), and then storing it on the history list as the input portion of a new event. The history command completes operating by simply unreadring the input. When the input is subsequently "reread", the event which already contains the input will be retrieved and used for recording the value of the operation, saving side-effects, etc., instead of creating a new event. Otherwise the input is treated exactly the same as if it had been typed in directly.

If ℓ specifies more than one event, the inputs for the corresponding events are simply concatenated into a linear sequence, with special markers (called pseudo-carriage returns) representing carriage returns²³ inserted between each input to indicate where new lines start. The result of this

²¹ For inputs in eval format, i.e., single expressions, **FIX** calls the editor so that the current expression is that input, rather than the list consisting of that input - see the example on the preceding page. However, the entire list is actually being edited. Thus if the user typed ↑ P in that example, he would see ((DEFINEQ FOO &) "<c.r.>").

²² "< c.r. >" denotes a pseudo-carriage return and is explained later.

²³ The value of the variable histstr0 is used to represent a carriage return. For readability, this value is the string "< c.r. >". Note that since the comparison is made using eq, this marker will never be confused with a string that was typed in by the user.

concatenation is then treated as the input referred to by $\$$. For example, when the user typed REDO FROM F ([7] on page 22.2) the inputs for the corresponding six events were concatenated to produce:

(F PPUTD "<c.r.>" (1 MOVD) "<c.r.>" 3 "<c.r.>" (XTR 2) "<c.r.>"
0 "<c.r.>" (SW 2 3)). Similarly, if the user had typed USE PPUTD FOR @UTD IN 15 THRU 20, the above list would have been constructed, and then PPUTD substituted for @UTD throughout it.

The same convention is used for representing multiple inputs when a USE command involves sequential substitutions. For example, if the user types GETD(FOO) and then USE FIE FUM FOR FOO, the input sequence that will be constructed is (GETD (FIE) "<c.r.>" GETD (FUM)), which is the result of substituting FIE for FOO in (GETD (FOO)) concatenated with the result of substituting FUM for FOO in (GETD (FOO)).

Once such a multiple input is constructed, it is treated exactly the same as a single input, i.e., the input sequence is recorded in a new event, and then unread, exactly as described above. When the inputs are "reread," the "pseudo-carriage-returns" are treated by lispread and readline exactly as real carriage returns, i.e., they serve to distinguish between apply and eval formats on inputs to lisp, and to delimit line commands to the editor. Note that once this multiple input has been entered as the input portion of a new event, that event can be treated exactly the same as one resulting from type in. In other words, no special checks have to be made when *referencing* an event, to see if it is simple or multiple. Thus, when the user types REDO following REDO FROM F, ([10] page 22.3) REDO does not even notice that the input retrieved from the previous event is (F PPUTD "<c.r.>" ... (SW 2 3)) i.e., a multiple input, it simply records this input and unreads it. Similarly, when the user then types USE @UTD FOR PPUTD on this multiple input, the USE command simply carries out the substitution, and the result is the same as though the user had typed USE @UTD FOR PPUTD IN 15 THRU 20.

In sum, this implementation permits $\$$ to refer to a single simple event, or to several events, or to a single event originally constructed from several events (which may themselves have been multiple input events, etc.) without having to treat each case separately.

HISTORY COMMANDS APPLIED TO HISTORY COMMANDS

Since history commands themselves do *not* appear in the input portion of events (although they are stored elsewhere in the event), they do not interfere with or affect the searching operations of event specifications. In effect, history commands are invisible to event specifications.²⁴ As a result, history commands themselves cannot be recovered for execution in the normal way. For example, if the user types USE A B C FOR D and follows this with USE E FOR D, he will not produce the effect of USE A B C FOR E (but instead will simply cause E to be substituted for D in the last event containing a D). To produce this effect, i.e., USE A B C FOR E, the user should type USE D FOR E IN USE. The appearance of the word REDO, USE or FIX in an event address specifies a search for the corresponding *history* command. (For example, the user can also type UNDO REDO.) It also specifies that the text of the history command itself be treated as though it were the input. However, *the user must remember that the context in which a history command is reexecuted is that of the current history, not the original context*. For example, if the user types USE FOO FOR FIE IN -1, and then later types REDO USE, the -1 will refer to the event before the REDO, not before the USE.

²⁴ With the exception described below under "History Commands that Fail".

HISTORY COMMANDS THAT FAIL

The one exception to the statement that "history commands are invisible to event specifications" occurs when a history command fails to produce any input. For example, suppose the user types USE LOG FOR ANTILOG AND ANTILOG FOR LOGG, causing lispX to respond LOGG ?. Since the USE command did not produce any input, the user can repair it by typing USE LOG FOR LOGG (i.e., does not have to specify IN USE). This latter USE command will invoke a search for LOGG, which *will* find the bad USE command. lispX then performs the indicated substitution, and unread USE LOG FOR ANTILOG AND ANTILOG FOR LOG. In turn, this USE command invokes a search for ANTILOG, which, *because it was not typed in but reread*, ignores the bad USE command which was found by the earlier search for LOGG, and which is still on the history list. In other words, *history commands that fail to produce input are visible to searches arising from event specifications typed in by the user, but not to secondary event specifications.*

In addition, if the most recent event is a history command which failed to produce input, a secondary event specification will effectively back up the history list one event so that relative event numbers for that event specification will not count the bad history command. For example, suppose the user types USE LOG FOR ANTILOG AND ANTILOG FOR LOGG IN -2 AND -1, and after lispX types LOGG ?, the user types USE LOG FOR LOGG. He thus causes the command USE LOG FOR ANTILOG AND ANTILOG FOR LOG IN -2 AND -1 to be constructed and unread. In the normal case, -1 would refer to the last event, i.e., the "bad" USE command, and -2 to the event before it. However, in this case, -1 refers to the event before the bad USE command, and the -2 to the event before that. In short, the caveat that "the user must remember that the context in which a history command is reexecuted is that of the current history, not the original context" does not apply if the correction is performed immediately.

MORE HISTORY COMMANDS

RETRY $\$$ similar to REDO except sets helpclock so that any errors that occur while executing $\$$ will cause breaks.

... vars similar to USE except substitutes for the (first) *operand*.

For example, EXPRP(FOO) followed by ... FIE FUM is equivalent to USE FIE FUM FOR FOO. See also event 52 on page 22.8.

?? $\$$ prints history list. If $\$$ is omitted, ?? prints the entire history list, beginning with most recent events. Otherwise ?? prints only those events specified in $\$$ (and in the order specified), e.g., ?? -1, ?? 10 THRU 15, etc.

?? commands are not entered on the history list, and so do not affect relative event numbers. In other words, an event specification of -1 typed following a ?? command will refer to the event immediately preceding the ?? command.

?? will print the history command, if any, associated with each event as shown at [9] on page 22.3 and page 22.6. Note that these history commands are not preceded by prompt characters,

indicating they are not stored as input.²⁵

?? prints multiple input events under one event number (see page 22.6).

Since events are initially stored on the history list with their value field equal to bell (control-G), if an operation fails to complete for any reason, e.g., causes an error, is aborted, etc., its "value" will be bell. This is the explanation for the blank line in event 2, page 22.6, and event 50, page 22.8.

- + **??** resets the value of the variable **it** (see page 22.26) to be the value of the last event printed.
- + **??** is implemented via the function printhistory, page 22.44, which can also be called directly by the user. Printing is performed via the function showprin2 (Section 14), so that if the value of sysprettyflg=T, events will be prettyprinted.

UNDO \emptyset

undoes the side effects of the specified events. For each event undone, **UNDO** prints a message: e.g., RPLACA UNDONE, REDO UNDONE etc. If nothing is undone because nothing was saved, **UNDO** types NOTHING SAVED. If nothing was undone because the event(s) were already undone, **UNDO** types ALREADY UNDONE. If \emptyset is empty, **UNDO** searches back for the last event that contained side effects, was not undone, and itself was not an **UNDO** command.²⁶ ²⁷

UNDO $\emptyset : x_1 \dots x_n$

Each x_i refers to a message printed by DWIM in the event(s) specified by \emptyset . The side effects of the corresponding DWIM corrections, and only those side effects, are undone.

²⁵ **REDO**, **RETRY**, **USE**, ..., and **FIX** commands, i.e., those commands that reexecute previous events, are not stored as inputs, because the input portion for these events are the expressions to be "reread". The history commands **UNDO**, **NAME**, **RETRIEVE**, **BEFORE**, and **AFTER** are recorded as inputs, and **??** prints them exactly as they were typed.

²⁶ Note that the user can undo **UNDO** commands themselves by specifying the corresponding event address, e.g., **UNDO -7** or **UNDO UNDO**.

²⁷ **UNDO**ing events in the reverse order from which they were executed is guaranteed to restore all pointers correctly, e.g., to undo all effects of last five events, perform **UNDO THRU -5**, not **UNDO FROM -5**. Undoing out of order may have unforeseen effects if the operations are *dependent*. For example, if the user performed (**NCONC1 FOO FIE**), followed by (**NCONC1 FOO FUM**), and then undoes the (**NCONC1 FOO FIE**), he will also have undone the (**NCONC1 FOO FUM**). If he then undoes the (**NCONC1 FOO FUM**), he will cause the **FIE** to reappear, by virtue of restoring **FOO** to its state before the execution of (**NCONC1 FOO FUM**). For more details, see page 22.33.

For example, if the message PRINTT [IN FOO] -> PRINT were printed, UNDO : PRINTT or UNDO : PRINT would undo the correction.²⁸

\$²⁹ is a special form of the USE command for conveniently specifying *character* substitutions. In addition, it has a number of useful properties in connection with events that involve errors.

\$ x FOR y IN & equivalent to USE \$x\$ FOR \$y\$ IN &

For example, the user types MOVD(FOO FOOSAVE T), he can then type \$ FIE FOR FOO IN MOVD to perform MOVD(FIE FIESAVE T). Note that USE FIE FOR FOO would perform MOVD(FIE FOOSAVE T).

An abbreviated form of \$ is available:

\$ y x IN & same as \$ x FOR y IN &, i.e., y's are changed to x's. can also be written as \$ y TO x, \$ y = x, or \$ y -> x.

\$ does event location the same as the USE command, i.e., if IN & is not specified, \$ searches for y.³⁰

After \$ finds the event, it looks to see if an error was involved in that event, and if the indicated character substitution can be performed in the object of the error message, called the offender.³¹ If so, \$ assumes the substitution refers to the offender, performs the indicated character substitution in the offender only, and then substitutes the result for the original offender throughout the event. For example, the user types (PRETTYDEF FOOFNS 'FOO FOOOVARS) causing a U.B.A. FOOOVARS error message. The user can now type \$ OO O, which will change FOOOVARS to FOOVARS, but *not* change FOOFNS or FOO.

If an error did occur in the specified event, the user can also omit specifying the object of the substitution, y, in which case the offender itself is used. Thus, the user could have corrected the above example by simply typing \$ FOOVARS. Since esubst is used for performing the substitution, i.e. the editors R command (see Section 9), \$ can be used in x to refer to the characters in y. For example, if the user types LOAD(PRSTRUC PROP), causing the error FILE NOT FOUND PRSTRUC, he can request the file to be loaded from LISP's directory by simply typing \$ <LISP>\$. This is equivalent to performing (R PRSTRUC <LISP>\$) on the event, and therefore replaces PRSTRUC by <LISP>PRSTRUC.

²⁸ Some portions of the messages printed by DWIM are strings, e.g., the message FOO UNSAVED is printed by printing FOO and then " UNSAVED". Therefore, if the user types UNDO : UNSAVED, the DWIM correction will not be found. He should instead type UNDO : FOO or UNDO : \$UNSAVED\$ (<esc>UNSAVED<esc>, see R command in editor, Section 9).

²⁹ \$ is the way that <esc> is echoed in Interlisp-10. Rather than writing <esc> throughout in the discussion that follows, \$ is used instead, since this is what the user will see when he types <esc>.

³⁰ However, unlike USE, \$ can only be used to specify one substitution at a time.

³¹ Whenever an error occurs, the offender is automatically saved on that event's entry in the history list, under the property *ERROR*.

Note that \$ also works for events in the editor. For example, if the user types (MOVE COND 33 2 TO BEFORE HERE), and the editor types 33 ?, the user can type \$ 3, causing 3 to be substituted for 33 in the MOVE command.

Note also that \$ never *searches* for an error. Thus, if the user types LOAD(PRSTRUC PROP) causing a FILE NOT FOUND error, types CLOSEALL(), and then types \$ <LISP>\$, lispx will complain that there is no error in CLOSEALL(). In this case, the user would have to type \$ <LISP>\$ IN LOAD, or \$ PRS <LISP>PRS (which would cause a search for PRS).

Note also that \$ operates on *input*, not on programs. If the user types FOO(), and within the call to FOO gets a U.D.F. CONDD error, he *cannot* repair this by \$ COND. lisp will type CONDD NOT FOUND IN FOO().

- + DWIM ¶
 - + this command is an instruction to the programmer's assistnat to reexecute the indicated event, only to "try harder" as described below.
 - + The basic idea behind the DWIM command is that the user has typed something that (a) is not what he meant to type, and (b) he feels that the p.a. has sufficient information to infer what it was that he did mean to type. For example, suppose the user had typed GETP(FOO MACCRO), which returned NIL. There is no "error" in this event: it is perfectly reasonable for getp to be given as its second argument a property not on the property list of a particular atom, or in fact, a property not on the property list of *any* atom. Thus there is no justification for DWIM or the programmer's assistant to perform any corrections or transformations in this event. Nevertheless, most users would recognize that what was meant was GETP(FOO MACRO). Thus if the user types DWIM GETP (or just DWIM if the GETP was the previous event), the p.a. would correct MACCRO to MACRO and reexecutes the event.
 - + Another example is if the user types SET(INITIALS RMK:), which performs the indicated assignment, and returns RMK:. This sets the value of the new, previously unbound variable initials, whereas the user intended to set the time stamp package parameter, initials (see Section 9). The user now types DWIM, the programmer's assistant undoes the indicated event, corrects INITIALS to INITIALS, and then reexecutes the event.
 - + The DWIM command is implemented by associating with various functions certain declarative information about their arguments, e.g. for getp, the p.a. knows that atm, its first argument, "should" have a non-NIL property list, and prop, its second argument, either should be one of the property names on that property list, or else a member of sysprops. When the p.a. is given a DWIM command, it searches the indicated event looking for a function for which one of these declarations is not satisfied.³² It then tries to "coerce" the corresponding argument so that the declaration is true, e.g. if told that the argument should have a property list, the p.a. would attempt spelling correction using the spelling lists userwords, spellings2, and spellings1, specifying fn=GETPROPLIST.
 - + The p.a. currently knows about the following information in the context of the DWIM command:
 - + (1) getd, fntyp, and movd should be given as a first argument an atom which has a definition;

+ 32 looking at quoted arguments only, e.g. if the user types (`GETP X 'MACCRO`), the p.a. would not try to do anything to `X` (or its value). The DWIM command works only on what is typed by the user.

- (2) getp, getprop, getproplist, put, and putprop should given as a first argument an atom which has a non-NIL property list, and as a second argument, a property which is either on the property list of that atom, or a member of sysprops; +
- (3) set, setq, and sctqq should be given a variable that has a top level value or is bound; +
- (4) loadfns and editf should be given the name of a function that is contained on one of the files on filelst. +

When the DWIM command is typed, and any of these conditions are not true, and the programmer's assistant can make them be true by appropriate spelling correction(s), the corrected event will be reexecuted. Otherwise, if all declarations are satisfied or the p.a. is unable to coerce an argument to satisfy one of these conditions, there is still one more thing that it can try: if an error occurred in the indicated event, the p.a. will pack \$\$ (two <esc>s) on the end of the offender, substitute the result for the offender throughout the event, and then reexecute the event. For those events in which the editor's pattern match is being employed (see Section 9), e.g. history commands, edit commands, etc., this has the effect of specifying a search for an atom that is "close" to the offender in the sense used by the spelling corrector (see pattern type 6b, Section 9). For example, suppose the user types UNDO LOADD, to which the p.a. responds LOADD ? because there was no event on the history list containing the atom LOADD. The user then types DWIM, and the p.a. is able to find an event containing LOAD. Similarly, the user types to the editor (MOVE 3 2 TO AFTER CONDD 1), and gets the error message CONDD ? because the find command failed to find CONDD. A DWIM command will cause the edit command (MOVE 3 2 TO AFTER CONDD\$\$ 1) to be executed, which will search for an atom that is "close" to CONDD, e.g. COND.

If all of these procedures fail, the p.a. types "Unable to figure out what you meant in:" followed by the indicated event. +

NAME atom \$ saves the event(s) (including side effects) specified by \$ on the property list of atom (under the property **HISTORY**) e.g., NAME FOO 10 THRU 15. NAME commands are undoable.

Commands defined by NAME can also be typed in directly as though they were built-in commands, e.g., FOO\$ is equivalent to REDO FOO.³³

Commands defined by NAME can also be parameterized, i.e., be defined to take arguments:

NAME name (args) : \$	<u>args</u> are interpreted the same as the arguments for a USE command. See page 22.13. When <u>name</u> is invoked, the argument values are substituted for <u>args</u> using the same substitution algorithm as for USE.
or	
NAME name ... args ... : \$	

For example, following the event (PUTD 'FOO (COPY (GETP 'FIE 'EXPR))), the user types NAME MOVE FOO FIE : PUDT. Then typing MOVE TEST1 TEST2 would cause (PUTD 'TEST1 (COPY (GETP 'TEST2 'EXPR))) to be executed, i.e., would be equivalent to typing USE TEST1 TEST2 FOR FOO FIE IN MOVE. Typing MOVE A B C D would cause two PUDT's to be executed. Note that !'s and \$'s can also be employed the same as with USE. For example, if following

³³ However, if FOO is the name of a variable, it would be evaluated, i.e., FOO\$ would return the value of FOO.

`<PREPINDEX(<MANUAL>14LISP.XGP)
<FIXFILE(<MANUAL>14LISP.XGPIDX)`

the user performed NAME FOO \$14\$: -2 AND -1, then FOO \$15\$ would perform the indicated two operations with 14 replaced by 15.³⁴

RETRIEVE atom

Retrieves and reenters on the history list the events named by atom. Causes an error if atom was not named by a NAME command.

For example, if the user performs NAME FOO 10 THRU 15, and at some time later types RETRIEVE FOO, 6 new events will be recorded on the history list (whether or not the corresponding events have been forgotten yet). Note that RETRIEVE does *not* reexecute the events, it simply retrieves them. The user can then REDO, UNDO, FIX, etc. any or all of these events. Note that the user can combine the effects of a RETRIEVE and a subsequent history command in a single operation, e.g., REDO FOO is equivalent to RETRIEVE FOO, followed by an appropriate REDO.³⁵ Note that UNDO FOO and ?? FOO are permitted.

BEFORE atom

undoes the effects of the events named by atom.

AFTER atom

undoes a BEFORE atom.

BEFORE/AFTER provide a convenient way of flipping back and forth between two states, namely that state *before* a specified event or events were executed, and that state *after* execution. For example, if the user has a complex data structure which he wants to be able to interrogate before and after certain modifications, he can execute the modifications, name the corresponding events with the NAME command, and then can turn these modifications off and on via BEFORE or AFTER commands.³⁶ Both BEFORE and AFTER are no-ops if the atom was already in the corresponding state; both generate errors if atom was not named by a NAME command.

Note: since UNDO, NAME, RETRIEVE, BEFORE, and AFTER are recorded as inputs they can be referenced by REDO, USE, etc. in the normal way. However, the user must again remember that the context in which the command is reexecuted is different than the original context. For example, if the user types NAME FOO DEFINEQ THRU COMPILE, then types ... FIE, the input that will be reread will be NAME FIE DEFINEQ THRU COMPILE as was intended, but both DEFINEQ and COMPILE, will refer to the most recent event containing those atoms, namely the event consisting of NAME FOO DEFINEQ THRU COMPILE!

³⁴ NAME FOO \$ is equivalent to NAME FOO : \$. In either case, if FOO is invoked with arguments, an error is generated.

³⁵ Actually, REDO FOO is better than RETRIEVE followed by REDO since in the latter case, the corresponding events would be entered on the history list twice, once for the RETRIEVE and once for the REDO.

³⁶ The alternative to BEFORE/AFTER for repeated switching back and forth involves UNDO, UNDO of the UNDO, UNDO of that etc. At each stage, the user would have to locate the correct event to undo, and furthermore would run the risk of that event being "forgotten" if he did not switch at least once per time-slice.

ARCHIVE \$

records the events specified by \$ on a permanent history list. This history list can be referenced by preceding a standard event specification with @@, e.g., ?? @@ prints the archived history list, REDO @@ -1 will recover the corresponding event from the archived history list and redo it, etc.

The user can also provide for automatic archiving of selected events by appropriately defining archivefn, or by putting the property *ARCHIVE*, value T, on the event. Events that are referenced by history commands are automatically marked for archiving in this fashion. For more details, see page 22.28.

FORGET \$

permanently erases the record of the side effects for the events specified by \$. If \$ is omitted, forgets side effects for entire history list.

FORGET is provided for users with space problems. For example, if the user has just performed sets, rplacas, rplacds, putd, remprops, etc. to release storage, the old pointers would not be garbage collected until the corresponding events age sufficiently to drop off the end of the history list and be forgotten. FORGET can be used to force immediate forgetting (of the side-effects only). FORGET is not undoable (obviously).

REMEMBER \$

instructs the file package to "remember" the events specified by \$. These events will be marked as changed objects of file package type EXPRESSIONS.³⁷ For example, following:

```
←MOVD?(DEFILE /DEFILE)
DEFILE
←REMEMBER -1
(MOVD? (QUOTE DELFILE) (QUOTE /DEFILE))
←
```

if the user calls files?, makefiles, or cleanup, the command (P (MOVD? (QUOTE DELFILE) (QUOTE /DEFILE))) will be constructed by the file package and added to the fileCOMS indicated by the user.³⁸

22.5 MISCELLANEOUS FEATURES AND COMMANDS

PL atom

(property list) prints out the property list of atom in a nice format, with printlevel reset to (2 . 3), e.g.

```
←PL +
```

³⁷ which can be written out via the file package command P

³⁸ unless the user has already explicitly added the corresponding expression to some P command himself. Note that "remembering" an event like PUT(FOO CLISPTYPE expression) will *not* result in a (PROP CLISPTYPE FOO) command, because this will save the current (at the time of the makefile) value for the CLISPTYPE property, which may or may not be expression. Thus, even if there is a PROP command which saves the CLISPTYPE property for FOO in some fileCOMS, remembering this event will still require a (P (PUT 'FOO 'CLISPTYPE 'expression)) command to appear.

+ CLISPTYPE: 12
 + ACCESSFNS: (PLUS IPLUS FPLUS)³⁹

+ PB atom (print bindings) prints value of atom with printlevel reset to (2 . 3). If atom is not bound, does not attempt spelling correction or generate an error.⁴⁰

? ¢ For use following an error. If ¢ is not specified, searches back on history list for last event that contains an error. The programmer's assistant then attempts to "analyze" the nature and cause of the error using the context information that was automatically saved at the time of the error.

For example, suppose the function foo contains the expression (ELT X 3), and foo is called with x equal to (#526234), i.e., a list of an array, causing the error ARG NOT ARRAY. If the user then types ?, the programmer's assistant will respond with:

because elt requires that A (its first argument) be an array, but in (ELT X 3) {in FOO}, the value of X is (#526234)

The programmer's assistant contains built-in information about many of the more common errors.⁴¹ In the event that it is unable to analyze the error, the programmer's assistant simply calls helpsys (Section 20) to present the user with information from the Interlisp manual relating to this error message.

; allows the user to type a line of text without having the programmer's assistant process it. Useful when linked to other users, or to annotate a dribble file (Section 21).

EXEC in Interlisp-10, calls subsys (Section 21) to descend to lower **exec**.⁴²

+ 39 PL is implemented via the function printprops.

+ 40 PB is also a break command (Section 15). As a break command, it ascends the stack, and, for each frame in which atom is bound, prints the frame name and value of atom. If typed in to the programmer's assistant when not at the top level, e.g. in the editor, a lower userexec, etc., PB will also ascend the stack as it does with a break. However, as a programmer's assistant command, it is primarily used to examine the top level value of a variable that may or may not be bound, or to examine a variable whose value is a large S-expression. PB is implemented via the function printbindings.

41 At some point in the future, we hope to extend this facility to enable the programmer's assistant to propose corrections to certain simple errors, e.g., argument reversal, leaving out an argument, etc.

* 42 Rather than start up a new fork each time the user types EXEC, the EXEC command will save the old fork handle upon return from an EXEC command, and, if the fork handle is still active, reuse it for the next EXEC command, i.e. an EXEC followed by another EXEC is equivalent to an EXEC followed by a CONTIN.

CONTIN

in Interlisp-10, performs subsys[T] (see Section 21) to continue last call to subsys.

TYPE-AHEAD

is a command that allows the user to type-ahead an indefinite number of inputs.

The assistant responds to TYPE-AHEAD with a prompt character of >. The user can now type in an indefinite number of lines of input, under errorset protection. The input lines are saved and unread when the user exits the type-ahead loop with the command \$GO (< esc > GO). While in the type-ahead loop, ?? can be used to print the type-ahead, FIX to edit the type-ahead, and \$Q (< esc > Q) to erase the last input (may be used repeatedly). For example:

```
<TYPE-AHEAD
>SYSOUT(TEM)
>MAKEFILE(EDIT)
>BRECOMPILE((EDIT WEDIT))
>F
>$Q
\&F
>$Q
\BRECOMPILE
>LOAD(WEDIT PROP)
>BRECOMPILE((EDIT WEDIT))
>F
>MAKEFILE(BREAK)
>LISTFILES(EDIT BREAK)
>SYSOUT(CURRENT)
>LOGOUT]
>??
>SYSOUT(TEM)
>MAKEFILE(EDIT)
>LOAD(WEDIT PROP)
>BRECOMPILE((EDIT WEDIT))43
>F
>MAKEFILE(BREAK)
>LISTFILES(EDIT BREAK)
>SYSOUT(CURRENT)
>LOGOUT]
>FIX
EDIT
*(R BRECOMPILE BCOMPL)
*p
((LOGOUT) (SYSOUT &) (LISTFILES &) (MAKEFILE &) (F) (BCOMPL &)
(LOAD &) (MAKEFILE &) (SYSOUT &))
*(DELETE LOAD)
*OK
>$GO
```

The TYPE-AHEAD command may be aborted by \$STOP (< esc > STOP); control-E simply aborts the current line of input.

⁴³ Note that type-ahead can be addressed to the compiler, since it uses lispread for input. Type-ahead can also be directed to the editor, but type-ahead to the editor and to lisp cannot be intermixed.

\$BUFS (< esc > BUFS) is a command for recovering the input buffers.

Whenever an error occurs in executing a lisp input or edit command, or a control-E or control-D is typed, the input buffers are saved and cleared. The \$BUFS command is used to restore the input buffers, i.e., its effect is exactly the same as though the user had retyped what was "lost." For example:

```
*(-2 (SETQ X (COND ((NULL Z) (CONS      (user typed control-E)
  *P
  (COND (& &) (T &))
  *2
  *$BUFS
  (-2 (SETQ X (COND ((NULL Z) (CONS
```

and user can now finish typing the (-2 ..) command.

Note: the type-ahead does not have to have been seen by Interlisp, i.e., echoed, since the system buffer is also saved.

Input buffers are not saved on the history list, but on a free variable. Thus, only the contents of the input buffer as of the last clearbuf can ever be recovered. However, input buffers cleared at evalqt are saved independently from those cleared by break or the editor. The procedure followed when the user types \$BUFS is to recover first from the local buffer, otherwise from the top level buffer.⁴⁴ Thus the user can lose input in the editor, go back to evalqt, lose input there, then go back into the editor, recover the editor's buffer, etc. Furthermore, a buffer cleared at the top can be recovered in a break, and vice versa.

valueof[x] is an nlambda function for obtaining the value of a particular event, e.g., (VALUEOF -1),⁴⁵ (VALUEOF ←FOO -2).

The value of an event consisting of several operations is a list of the values for each of the individual operations.

Note: the value field of a history entry is initialized to bell (control-G). Thus a value of bell indicates that the corresponding operation did not complete, i.e., was aborted or caused an error (or else returned bell).

+ IT the value of it is always the value of the last event executed, i.e. valueof[-1], e.g.

⁴⁴ The local buffer is stored on lispbufs; the top level buffer on toplispbufs. The forms of both buffers are (CONS (LINBUF) (SYSBUF)) (see Section 14). Recovery of a buffer is destructive, i.e., \$BUFS sets the corresponding variable to NIL. If the user types \$BUFS when both lispbufs and toplispbufs are NIL, the message NOTHING SAVED is typed, and an error generated.

⁴⁵ Although the input for valueof is entered on the history list before valueof is called, valueof[-1] still refers to the value of the expression immediately before the valueof input, because valueof effectively backs the history list up one entry when it retrieves the specified event. Similarly, (VALUEOF FOO) will find the first event before this one that contains a FOO.

```
←(SQRT 2)
1.414214
←(SQRT IT)
1.189207
```

If last event was a multiple event, e.g. REDO -3 THRU -1, it is set to value of the last of these events. Following a ?? command, it is set to value of corresponding event. In other words, in all cases, it is set to the last value printed on the terminal.

control-U⁴⁶

when typed in at any point during an input being read by lispread, permits the user to edit the input before it is returned to the calling function.

This feature is useful for correcting mistakes noticed in typing *before* the input is executed, instead of waiting till after execution and then performing an UNDO and a FIX. For example, if the user types (DEFINEQ (FOO (LAMBDA (X) (FIXSPELL X and at that point notices the missing left parenthesis, instead of completing the input and allowing the error to occur, and then fixing the input, he can simply type control-U,⁴⁷ finish typing normally, whereupon the editor is called on (FOO (LAMBDA (X) (FIXSPELL X --]), which the user can then repair, e.g., by typing (LI 1). If the user exits from the editor via OK, the (corrected) expression will be returned to whoever called lispread exactly as though it had been typed.⁴⁸ If the user exits via STOP, the expression is returned so that it can be stored on the history list. However it will *not* be executed. In other words, the effect is the same as though the user had typed control-E at exactly the right instant.

prompt#flg

is a flag which when set to T causes the current event number to be printed before each ←, : and * prompt characters. See description of promptchar, page 22.38.

prompt#flg is initially NIL.

promptcharforms

list of forms to be executed each time promptchar is called. See description of promptchar.

promptcharforms initially contains the expression (CHECKNIL) which checks to see if car or cdr of NIL have been clobbered, or NIL or T have been reset or rebound, and if so, restores them and prints a warning message.

historysaveforms

list of forms to be executed each time historysave is called. See description of historysave.

⁴⁶ control-N for Interlisp on TOPS-20.

⁴⁷ Control-U can be typed at any point, even in the middle of an atom; it simply sets a variable checked by lispread.

⁴⁸ Control-U also works for calls to readline, i.e., for line commands.

+ Note that promptcharforms and historysaveforms together enable bracketing each interaction with the user, e.g. to measure how long he takes to respond, use a different readable or terminal table, etc.

+ resetforms list of forms to be evaluated at each RESET, i.e. when user types control-D, calls function reset, or types control-C followed by START.

archivefn allows the user to specify events to be automatically archived.

When archivefn is set to T, and an event is about to drop off the end of the history list and be forgotten, archivefn is called giving it as its first argument the input portion of the event, and as its second argument, the entire event.⁴⁹ If archivefn returns T, the event is archived. For example, some users like to keep a record of all calls to load. Defining archivefn as: (LAMBDA (X Y) (EQ (CAR X) (QUOTE LOAD))) will accomplish this. Note that archivefn must be both set and defined. archivefn is initially NIL and undefined.

The user can also specify that a particular event be archived when it is about to drop off the end of the history list by putting the property *ARCHIVE*, value T, on the event, e.g., by means of an appropriately defined lispxuserfn (see below). One use of this feature is that the system automatically adds the *ARCHIVE* property to all events that are referenced by history commands.⁵⁰ Thus once an event is redone, it is guaranteed to be saved.

lispxmacros provides a macro facility for lispx.

lispxmacros allows the user to define his own lispx commands. It is a list of elements of the form (command def). Whenever command appears as the first expression on a line in a lispx input, the variable lispxline is bound to the rest of the line, the event is recorded on the history list, and def is evaluated, and its value stored as the value of the event. Similarly, whenever command appears as car of a form in a lispx input, the variable lispxline is bound to cdr of the form, the event recorded, and def is evaluated.⁵¹ (See page 22.44 for an example of a lispxmacro). RETRIEVE, BEFORE, and AFTER are implemented as lispxmacros. In addition in Interlisp-10, LISP, SNDMSG, TECO, and EXEC are lispxmacros which perform the corresponding calls to subsys (Section 21), and CONTIN is a lispxmacro which performs (SUBSYS T). SY is a lispxmacro which performs the SYSTAT command. Finally, DIR is a lispxmacro which calls the function directory, e.g., DIR *.COM;* lists all compiled files. (For more details, see Section 21.)

lispxhistorymacros provides a macro facility for history commands.

lispxhistorymacros allows the user to define his own history commands. The format of

⁴⁹ In case archivefn needs to examine the value of the event, its side effects, etc. See page 22.34 for discussion of the format of history lists.

⁵⁰ unless archiveflg=NIL. archiveflg is initially T.

+ ⁵¹ An element of the form (command NIL def) is interpreted to mean bind lispxline and evaluate def as described above, except do not save the event on the history list.

lispxhistorymacros is the same as that of lispxmacros, except that the result of evaluating def is treated as a list of expressions to be *unread*, exactly as though the expressions had been retrieved by a REDO command, or computed by a USE command.⁵² Note that returning NIL means nothing else is done. This provides a mechanism for defining lispx commands which are executed for effect only, and do not really have values, per se. For example, the ? command is implemented as a lispxhistorymacro.

lispxuserfn provides a way for a user function to process selected inputs.

When lispxuserfn is set to T, it is applied⁵³ to all inputs not recognized as one of the commands described above. If lispxuserfn decides to handle this input, it simply processes it (the event was already stored on the history list before lispxuserfn was called), sets lispxvalue to the value for the event, and returns T. lispx will then know not to call eval or apply, and will simply store lispxvalue into the value slot for the event, and print it. If lispxuserfn returns NIL, lispx proceeds by calling eval or apply in the usual way. Thus by appropriately defining (and setting) lispxuserfn, the user can with a minimum of effort incorporate the features of the programmer's assistant into his own executive (actually it is the other way around).

The following output illustrates such a coupling.⁵⁴

```

**SETQ(ALTFORM (MAPCONC NASDIC (F/L (GETP X 'ALTFORMS) [1]
=NASDICT
(AL26 BE7 C056 C057 C060 C13 H3 MN54 NA22 SC46 S34 TI44)
*(GIVE ME LINES CONTAINING COBALT) [2]
SAMPLE PHASE CONSTIT. CONTENT UNIT CITATION TAG
S10002 OVERALL C056 40.0 DPM/KG D70-237 0
          C13 8.8 DEL D70-228 0
          H3 314.0 DPM/KG
          MN54 28

**GETP(COBALT ALTFORMS) [3]
(C056 C057 C060 C13 H3 MN54 NA22 SC46 S34 TI44)
**UNDO MAPCONC [4]
SETQ UNDONE.
**REDO GETP [5]
(C056 C057 C060)
**REDO COBALT [6]
SAMPLE PHASE CONSTIT. CONTENT UNIT CITATION TAG
S10002 OVERALL C057 40.0 DPM/KG D70-237 0
S10003 OVERALL CO 15.0 DPM/KG D70-203 0
          14.1 D70-216
          C056 43.0 DPM/KG D70-237 0
          C057 43.0 DPM/KG D70-241 0
          C060 1.0

**USE MANGANESE FOR COBALT

```

⁵² See page 22.15 for discussion of implementation of REDO, USE, and FIX.

⁵³ Like archivefn, lispxuserfn must be both set and defined.

⁵⁴ The output is from the Lunar Sciences Natural Language Information System developed for the NASA Manned Spacecraft Center by William A. Woods of Bolt Beranek and Newman Inc., Cambridge, Mass.

The user is running under his own executive program which accepts requests in the form of sentences, which it first parses and then executes. The user first "innocently" computes a list of all ALTERNATIVE-FORMS for the elements in his system [1]. He then inputs a request in sentence format [2] expecting to see under the column CONSTIT. only cobalt, CO, or its alternate forms, CO56, CO57, or CO60. Seeing C13, H3, and MN54, he aborts the output, and checks the property ALTFORMS for COBALT [3]. The appearance of C13, H3, MN54, he aborts the output, and checks the property ALTFORMS for COBALT [3]. The appearance of C13, H3, MN54 et al, remind him that the mapconc is destructive, and that in the process of making a list of the ALTFORMS, he has inadvertently strung them all together. Recovering from this situation would require him to individually examine and correct the ALTFORMS for each element in his dictionary, a tedious process. Instead, he can simply UNDO MAPCONC, [4] check to make sure the ALTFORM has been corrected [5], then redo his original request [6] and continue. The UNDO is possible because the first input was executed by lisp; the (GIVE ME LINES CONTAINING COBALT) is possible because the user defined lispuserfn appropriately; and the REDO and USE are possible because the

(GIVE ME LINES CONTAINING COBALT) was stored on the history list before it was transmitted to lispuserfn and the user's parsing program.

lispuserfn is a function of two arguments, x and line, where x is the first expression typed, and line the rest of the line, as read by readline (see page 22.35). For example, if the user types FOO(A B C), x=FOO, and line=((A B C)); if the user types (FOO A B C), x=(FOO A B C), and line=NIL; and if the user types FOO A B C, x=FOO and line=(A B C).

Thus in the above example, lispuserfn would be defined as:

```
[LAMBDA (X LINE)
  (COND
    ((AND (NULL LINE)
          (LISTP X))
     (SETQ LISPXVALUE (PARSE X)))
    T)]
```

Note that since lispuserfn is called for each input (except for p.a. commands), it can also be used to monitor some condition or gather statistics.

In addition to saving inputs and values, lisp saves most system messages on the history list, e.g., FILE CREATED --, (fn REDEFINED), (var RESET), output of TIME, BREAKDOWN, STORAGE, DWIM messages, etc. When printhistory prints the event, this output is replicated. This facility is implemented via the functions lispxprint, lispxprinl, lispxprin2, lispxspaces, lispxterpri, lispxtab, and lispxprintdef.⁵⁵ ⁵⁶ In addition to performing the corresponding output operation, these

+ 55 i.e. to perform output operations from user programs so that the output will appear on the history list, the program needs simply to call the corresponding lisp printing function. All of these functions have an optional argument, nodoflg, which if T, says to store the output on the history list as though it had been printed, but not to do the actual printing.

* 56 The function userlispxprint is available to permit the user to define additional lispprinting functions for already existing printing functions. The user can define a lispprinting function by simply giving it the definition of userlispxprint, e.g., MOVD(USERLISPXPRINT LISPXPRINTDEF), as long as the new function name is formed by adding "LISPX" to the front of the name of an existing printing function, and that this function takes three or fewer arguments, and the second argument be the file name. userlispxprint is defined to look back on the stack, find the name of the calling function, strip off the leading "LISPX", perform the appropriate saving information, and then call the function to do the actual printing.

functions store an appropriate expression on the history event under the property ***LISPxPRINT***.⁵⁷ This expression is used by printhistory to reproduce the output.

Lispx also performs spelling corrections using lispcoms, a list of its commands, as a spelling list whenever it is given an unbound atom or undefined function, i.e., before attempting to evaluate the input.⁵⁸

22.6 UNDOING

The UNDO capability of the programmer's assistant is implemented by requiring that each operation that is to be undoable be responsible itself for saving on the history list enough information to enable reversal of its side effects. In other words, the assistant does not "know" when it is about to perform a destructive operation, i.e., it is not constantly checking or anticipating. Instead, it simply executes operations, and any undoable changes that occur are automatically saved on the history list by the responsible function.⁵⁹ The operation of UNDOing, which involves recovering the saved information and performing the corresponding inverses, works the same way, so that the user can UNDO an UNDO, and UNDO that etc.

At each point, until the user specifically requests an operation to be undone, the assistant does not know, or care, whether information has been saved to enable the undoing. Only when the user attempts to undo an operation does the assistant check to see whether any information has been saved. If none has been saved, and the user has specifically named the event he wants undone, the assistant types **NOTHING SAVED**. (When the user simply types UNDO, the assistant searches for the last undoable event, ignoring events already undone as well as UNDO operations themselves.)

This implementation minimizes the overhead for undoing. Only those operations which actually make changes are affected, and the overhead is small: two or three cells of storage for saving the information, and an extra function call. However, even this small price may be too expensive if the operation is sufficiently primitive and repetitive, i.e., if the extra overhead may seriously degrade the overall performance of the program.⁶⁰ Hence not every destructive operation in a program should necessarily be undoable; the programmer must be allowed to decide each case individually.

Therefore for each primitive destructive operation, we have implemented *two* separate functions, one which always saves information, i.e., is always undoable, and one which does not, e.g., /rplaca

⁵⁷ unless lispprintflg is **NIL**.

⁵⁸ Lispx is also responsible for rebinding helpclock, used by breakcheck, Section 16, for computing the amount of time spent in a computation, in order to determine whether to go into a break if and when an error occurs.

⁵⁹ When the number of changes that have been saved exceeds the value of #undosaves (initially set to 50), the user is asked if he wants to continue saving the undo information for this event. The purpose of this feature is to avoid tying up large quantities of storage for operations that will never need to be undone. The interaction is handled by the same routines used by DWIM, so that the input buffers are first saved and cleared, the message typed, then the system waits dwinwait seconds, and if there is no response, assumes the default answer, which in this case is NO. Finally the input buffers are restored. See page 22.41 for details.

⁶⁰ The rest of the discussion applies only to Lispx; the editor handles undoing itself in a slightly different fashion, as described on page 22.45.

and rplaca, /remprop and remprop.⁶¹ In the various system packages, the appropriate function is used. For example, break uses /putd and /remprop so as to be undoable, and DWIM uses /rplaca and /rplacd, when it makes a correction.⁶² Similarly the user can simply use the corresponding / function if he wants to make a destructive operation in his own program undoable. When the / function is called, it will save the undo information in the current event on the history list.

However, all operations that are *typed in* to lisp are made undoable, simply by substituting the corresponding undoable function for any destructive function throughout the input.⁶³ For example, on page 22.7, when the user typed (MAPCONC NASDIC (F/L ...)) it was (/MAPCONC NASDIC (F/L ...)) that was evaluated. Since the system cannot know whether efficiency and overhead are serious considerations for the execution of an expression in a user *program*, the user must decide, e.g., call /mapconc if he wants the operation undoable. However, expressions that are typed-in rarely involve iterations or lengthy computations *directly*. Therefore, if all primitive destructive functions that are immediately contained in a type-in are made undoable, there will rarely be a significant loss of efficiency. Thus lisp scans all user input before evaluating it, and substitutes the corresponding undoable function for all primitive destructive functions. Obviously with a more sophisticated analysis of both user input and user programs, the decision concerning which operations to make undoable could be better advised. However, we have found the configuration described here to be a very satisfactory one. The user pays a very small price for being able to undo what he types in, and if he wishes to protect himself from malfunctioning in his own programs, he can have his program specifically call undoable functions, or go into testmode as described next.

TESTMODE

Because of efficiency considerations, the user may not want certain functions undoable after his program becomes operational. However, while debugging he may find it desirable to protect himself against a program running wild, by making primitive destructive operations undoable. The function testmode provides this capability by temporarily making everything undoable.

testmode[flg]

testmode[T] redefines all primitive destructive functions⁶⁴ with their corresponding undoable versions and sets testmodeflg to T. testmode[] restores the original definitions, and sets testmodeflg to NIL.⁶⁵

Note that setq's are *not* undoable, even in testmode. To make the corresponding operation undoable in testmode, set or rplaca should be used.

- * 61 The "slash" functions that are currently implemented can be found as the value of /fns.
- 62 The effects of the following functions are always undoable (regardless of whether or not they are typed in): define, defineq, defc (used to give a function a compiled code definition), deflist, load, savedef, unsavedef, break, unbreak, rebreak, trace, breakin, unbreakin, changename, editfns, editf, editv, editp, edite, editl, esubst, advise, unadvise, readadvise, plus any changes caused by DWIM.
- 63 The substitution is performed by the function lisp/, described on page 22.42.
- 64 i.e., the "slash" functions; see footnote on page 22.32.
- 65 testmode will have no effect on compiled mapconc's, since they compile open with frplacd's.

UNDOING OUT OF ORDER

/rplaca and /rplacd operate by saving the pointer that is to be changed and its original contents (i.e., /rplaca saves car and /rplacd saves cdr). Undoing /rplaca and /rplacd simply restores the pointer. Thus, if the user types (RPLACA FOO 1), followed by (RPLACA FOO 2), then undoes both events by undoing the most recent event first, then undoing the older event, FOO will be restored to its state before either rplaca operated. However if the user undoes the first event, then the second event, (CAR FOO) will be 1, since this is what was in car of FOO before (RPLACA FOO 2) was executed. Similarly, if the user performs (NCONC1 FOO 1) then (NCONC1 FOO 2), undoing just (NCONC1 FOO 1) will remove both 1 and 2 from FOO. The problem in both cases is that the two operations are not "independent." In general, operations are always independent if they affect different lists or different sublists of the same list.⁶⁶ Undoing in reverse order of execution, or undoing independent operations, is always guaranteed to do the "right" thing. However, undoing dependent operations out of order may not always have the predicted effect.

SAVESET

Setq's are made undoable on type in by substituting a call to saveset (described in detail on page 22.40), whenever setq is the name of the function to be applied, or car of the form to be evaluated.⁶⁷ In addition to saving enough information on the history list to enable undoing, saveset operates in a manner analogous to savedef when it resets a top level value, i.e., when it changes a top level binding from a value other than NOBIND to a new value that is not equal to the old one. In this case, saveset saves the old value of the variable being set on the variable's property list under the property VALUE, and prints the message (variable RESET). The old value can be restored via the function unset,⁶⁸ which also saves the current value (but does not print a message). Thus unset can be used to flip back and forth between two values.

rpaq and rpaqq are implemented via calls to saveset. Thus old values will be saved and messages printed for any variables that are reset as the result of loading a file.⁶⁹ Calls to set and setqq appearing in type in are also converted to appropriate calls to saveset.

For top level variables, saveset also adds the variable to the appropriate spelling list, thereby noticing variables set in files via rpaq or rpaqq, as well as those set via type in.

⁶⁶ Property list operations, (i.e., put, addprop and remprop) are handled specially so that they are always independent, even when they affect the same property list. For example, if the user types PUT(FOO FIE1 FUM1) then PUT(FOO FIE2 FUM2), then undoes the first event, the FIE2 property will remain, even though CDR(FOO) may have been NIL at the time the first event was executed.

⁶⁷ setq is made undoable by substituting savesetq, setqq by savesetqq, both of which are implemented in terms of saveset.

⁶⁸ Of course, UNDO can be used as long as the event containing this call to saveset is still active. Note however that the old value will remain on the property list, and therefore be recoverable via unset, even after the original event has been forgotten.

⁶⁹ To complete the analogy with define, saveset will not save old values on property lists if dflg=T, e.g., when load is called with second argument T, (however, the call to saveset will still be undoable), and when dflg=ALLPROP, the value is stored directly on the property list under property VALUE (the latter applies only to calls from rpaqq and rpaq).

UNDONLSETQ AND RESETUNDO

The function undonlsetq provides a limited form of backtracking: if an error occurs under the undonlsetq, all undoable side effects executed under the undonlsetq are undone. resetundo, for use in conjunction with resclst and resetsave (Section 5), provides a more general undo capability in that the user can specify that the side effects be undone after the specified computation finishes, is aborted by an error, or by a control-D. undonlsetq and resetundo are described in detail on page 22.43.

22.7 FORMAT AND USE OF THE HISTORY LIST

There are currently two history lists, lispxhistory and edithistory. Both history lists have the same format, and in fact, each use the same function, historysave, for recording events, and the same set of functions for implementing commands that refer to the history list, e.g., historyfind, printhistory, undosave, etc.⁷⁰

Each history list is a list of the form (l event# size mod), where l is the list of events with the most recent event first, event# is the event number for the most recent event on l, size is the size of the time-slice, i.e., the maximum length of l, and mod is the highest possible event number (see footnote on page 22.7). lispxhistory and edithistory are both initialized to (NIL 0 30 100). Setting lispxhistory or edithistory to NIL is permitted, and simply disables all history features, i.e., lispxhistory and edithistory act like flags as well as repositories of events.

Each individual event on l is a list of the form (input id value . props), where input is the input sequence for the event, as described on page 22.15-16, id the prompt character, e.g., ,, :, *,⁷¹ and value is the value of the event, and is initialized to bell.⁷²

props is a property list, i.e., of the form (property value property value --). props can be used to associate arbitrary information with a particular event. Currently, the properties SIDE, *ARCHIVE*, *GROUP*, *HISTORY*, *PRINT*, USE-ARGS, ...ARGS, *ERROR*, *CONTEXT* and *LISPXPRINT* are being used. The value of property SIDE is a list of the side effects of the event. (See discussion of undosave, page 22.41, and undolisp, page 22.42.) The *HISTORY* and *GROUP* properties are used for commands that reexecute previous events, i.e., REDO, RETRY, USE, ..., and FIX. The value of the *HISTORY* property is the history command itself, i.e., what the user actually typed, e.g., REDO FROM F, and is used by the ?? command for printing the event. The value of the property *PRINT* is also for use by the ?? command, when special formatting is required, for example, in printing events corresponding to the break commands OK, GO, EVAL, and ?=. USE-ARGS and ...ARGS are used to save the arguments and expression for the corresponding history command. *ERROR* and *CONTEXT* are used to save information when errors occur for subsequent use by the \$ and ? commands. The property *ARCHIVE* on an event causes the event to be automatically archived when it "falls off the end" of the history list (see page 22.28). *LISPXPRINT* is used to record calls to lispxprint, lispxprinl, et al, (see page 22.30).

⁷⁰ A third history list, archivelst, is used when events are archived, as described on page 22.23. It too uses the same format.

⁷¹ id is one of the arguments to lisp and to historysave. A user can call lisp giving it any prompt character he wishes (except for *, since in certain cases, lisp must use the value of id to tell whether or not it was called from the editor.) For example, on page 22.29, the user's prompt character was **.

⁷² On edithistory, this field is used to save the side effects of each command.

When lisp is given an input, it calls historysave to record the input in a new event.⁷³ Normally, historysave returns as its value the new event. lisp binds lisphist to the value of historysave, so that when the operation has completed, lisp knows where to store the value, namely in caddr of lisphist.⁷⁴ lisphist also provides access to the property list for the current event. For example, the / functions are all implemented to call undosave, which simply adds the corresponding information to lisphist under the property SIDE, or if there is no property SIDE, creates one, and then adds the information.

After binding lisphist, lisp executes the input, stores its value in caddr of lisphist, prints the value, and returns.

When the input is a REDO, RETRY, USE, ..., or FIX command, the procedure is similar, except that the event is also given a *GROUP* property, initially NIL, and a *HISTORY* property, and lisp simply unread the input and returns. When the input is "reread", it is historysave, not lisp, that notices this fact, and finds the event from which the input originally came.⁷⁵ historysave then adds a new (input id value . props) entry to the *GROUP* property for this event, and returns this entry as the "new event." lisp then proceeds exactly as when its input was typed directly, i.e., it binds lisphist to the value of historysave, executes the input, stores the value in caddr of lisphist, prints the value, and returns. In fact, lisp never notices whether it is working on freshly typed input, or input that was reread. Similarly, undosave will store undo information on lisphist under the property SIDE the same as always, and does not know or care that lisphist is not the entire event, but one of the elements of the *GROUP* property. Thus when the event is finished, its entry will look like:

```
(input1 id1 value1 SIDE sidel)
  (input2 id2 value2 SIDE side2)
    ...
))
```

⁷⁶

This implementation removes the burden from the function calling historysave of distinguishing between new input and reexecution of input whose history entry has already been set up.⁷⁷

22.8 LISPX AND READLINE

lisp is called with the first expression typed on a line as its first argument, lispxx.

If this is *not* a list, lisp *always* does a readline, and treats lispxx plus the line as the input for the

⁷³ The commands ??, FORGET, TYPE-AHEAD, \$BUFS, and ARCHIVE are executed immediately, and are not recorded on the history list.

⁷⁴ Note that by the time it completes, the operation may no longer correspond to the most recent event on the history list. For example, all inputs typed to a lower break will appear later on the history list.

⁷⁵ If historysave cannot find the event, for example if a user program unread the input directly, and not via a history command, historysave proceeds as though the input were typed.

⁷⁶ In this case, the value field is not being used; valueof instead collects each of the values from the *GROUP* property, i.e., returns mapcar[listget[event; *GROUP*]; CADDR]. Similarly, undo operates by collecting the SIDE properties from each of the elements of the *GROUP* property, and then undoing them in reverse order.

⁷⁷ Although we have not yet done so, this implementation, i.e., keeping the various "sub-events" separate with respect to values and properties, also permits constructing commands for operating on just one of the sub-events.

event, and stores it accordingly on the history list.⁷⁸ Then it decides what to do with the input, i.e., if it is not recognized as a command, a lispmacro, or is processed by lispuserfn, call eval or apply.⁷⁹ readline normally is terminated either by (1) a carriage return that is not preceded by a space, or (2) a list that is terminated by a], or (3) an unmatched) or], which is not included in the line. However, when called from lisp, readline operates differently in two respects:

- (1) If the line consists of a single) or], readline returns (NIL) instead of NIL, i.e., the) or] is included in the line. This permits the user to type FOO) or FOO], meaning call the function' FOO with no arguments, as opposed to FOO] (FOOcarriage-return), meaning evaluate the variable FOO.
- (2) If the first expression on the line is a list that is not preceded by any spaces, the list terminates the line regardless of whether or not it is terminated by]. This permits the user to type EDITF(FOO) as a single input.

Note that if any spaces are inserted between the atom and the left parentheses or bracket, readline will assume that the list does not terminate the line. This is to enable the user to type a line command such as USE (FOO) FOR FOO. In this case, a carriage return will be typed after (FOO) followed by "..." as described in Section 14. Therefore, if the user accidentally puts an extra space between a function and its arguments, he will have to complete the input with another carriage return, e.g.,

```
+EDITF (FOO)
...
EDIT
*
```

22.9 FUNCTIONS

lisp[lispxx;lispid;lispxxmacros;lispxxuserfn]⁸⁰

lisp is like eval/apply. It carries out a single computation, and returns its value. The first argument, lispxx is the result of a single call to lispread. lisp will call readline, if necessary as described on page 22.35. lisp prints the value of the computation, as well as saving the input and value on lispxhistory.⁸¹

If lispxx is a history command, lisp calls historysave, executes the command, and returns the value of historysave.

⁷⁸ If lispxx is a list car of which is LAMBDA or NLAMBDA, lisp calls lispread to obtain the arguments.

⁷⁹ If the input consists of one expression, eval is called; if two, apply; if more than two, the entire line is treated as a single form and eval is called.

⁸⁰ lispid corresponds to id on PAGEREF L147. Lisp also has a fifth argument, lispflg, which is used by the E command in the editor.

⁸¹ Note that the history is *not* one of the arguments to lisp, i.e., the editor must bind (reset) lispxhistory to edithistory before calling lisp to carry out a history command.
Lisp will continue to operate as an eval/apply function if lispxhistory is NIL. Only those functions and commands that involve the history list will be affected.

If the value of the fourth argument, lispxxmacros, is not NIL, it is used as the lispx macros, otherwise the top level value of lispxxmacros is used. If the value of the fifth argument, lispxxuserfn, is not NIL, it is used as lispxxuserfn. In this case, it is not necessary to both set and define lispxxuserfn as described on page 22.29.

The overhead for a call to lispx (in Interlisp-10) is approximately 17 milliseconds, of which 12 milliseconds are spent in maintaining the spelling lists. In other words, in Interlisp, the user pays 17 more milliseconds for each eval or apply input over a conventional LISP executive, in order to enable the features described in this chapter.

userexec[lispqid;lispxxmacros;lispxxuserfn]

repeatedly calls lispx under errorset protection specifying lispxxmacros and lispxxuserfn, and using lispqid (or ← if lispqid=NIL) as a prompt character. Userexec is exited via the lispmacro OK, or else with a retfrom.

lispxread[file;rdtbl]

is a generalized read. The value of readbuf is a list of expressions that have been unread.⁸² If readbuf=NIL, lispxread performs read[file;rdtbl], which it returns as its value. (If the user types control-U during the call to read, lispxread calls the editor and returns the edited value.) *

If readbuf is not NIL, lispxread "reads" the next expression on readbuf, i.e., essentially returns

$$(\text{PROG1} (\text{CAR READBUF}) (\text{SETQ READBUF} (\text{CDR READBUF}))).^{83}$$

readline, described in Section 14, also uses this generalized notion of reading. When readbuf is not NIL, readline "reads" expressions from readbuf until it either reaches the end of readbuf, or until it reads a pseudo-carriage return (see page 22.15). In both cases, it returns a list of the expressions it has "read". (The pseudo-carriage return is not included in the list.) *

When readbuf is not NIL, both lispxread and readline actually obtain their input by performing (APPLY* LISPXREADFN FILE), where lispxreadfn is initially set to READ. Thus, if the user wants lispx, the editor, break, et al to do their reading via a different input function, e.g., uread, he simply sets lispxreadfn to the name of that function (or an appropriate LAMBDA expression).

⁸² Note: the user should only add expressions to readbuf by using the function lispunread, since it knows about the format of readbuf. *

⁸³ Except that pseudo-carriage returns, as represented by the value of histstr0, are ignored, i.e., skipped. Lispread also sets rereadflg to NIL when it reads via read, and sets rereadflg to the value of readbuf when rereading. *

lispxreadp[flg]

is a generalized readp. If flg=T, lispxreadp returns T if there is any input waiting to be "read", a la lispxread. If flg=NIL, lispxreadp returns T only if there is any input waiting to be "read" on this line. In both cases, leading spaces are ignored, i.e., skipped over with readc, so that if only spaces have been typed, lispxreadp will return NIL.

lispxunread[1st;-]

unreads 1st, a list of expressions to be read.

promptchar[id;flg;history]

prints the prompt character id.

promptchar will not print anything when the next input will be "reread", i.e., readbuf is not NIL. promptchar will also not print when readp[]=T, unless flg is T.

Thus the editor calls promptchar with flg=NIL so that extra *'s are not printed when the user types several commands on one line. However, evalqt calls promptchar with flg=T, since it always wants the ← printed (except when "rereading").

Finally, if prompt#flg is T and history is not NIL, promptchar prints the current event number (of history) before printing id.

- + The value of promptcharforms is a list of expressions to be evaluated by promptchar before, and if, it does any printing. If promptchar is going to print something, it first maps down promptcharforms evaluating each expression under an errorset. These expressions can be conditioned on the values of history, id, and promptstr, which is what promptchar will print before id, if anything, e.g. when prompt#flg is T, promptstr will be the event number. The expressions on promptcharforms can be executed for effect, e.g. to change the shape of a cursor, update a clock, check for mail, etc. or to change what promptchar is about to print by resetting id and/or promptstr.⁸⁴ After promptcharforms have been evaluated, promptstr is printed if it is (still) non-NIL, and then id is printed, if it is (still) non-NIL.

lispxeval[lispxform;lispqid]

evaluates lispxform (using eval) the same as though it were typed in to lispq, i.e., the event is recorded, and the evaluation is made undoable by substituting the slash functions for the corresponding destructive functions, as described on page 22.32. lispxeval returns the value of the form, but does not print it.

historiesave[history;id;input1;input2;input3;props]

records one event on history. If input1 is not NIL, the input is of the form (input1 input2 . input3). If input1 is NIL, and input2 is not NIL, the input is of the form (input2 . input3). Otherwise, the input is just input3.

⁸⁴ promptcharforms initially contains the expression (CHECKNIL) which checks to see if car or cdr of NIL have been clobbered, or NIL or T have been reset or rebound, and if so, restores them and prints a warning message.

historysave creates a new event with the corresponding input, id, value field initialized to bell, and props. If the history has reached its full size, the last event is removed and cannibalized.

The value of historysave is the new event. However, if rereadflg is not NIL, and the most recent event on the history list contains the history command that produced this input, historysave does not create a new event, but simply adds an (input id bell . props) entry to the *GROUP* property for that event and returns that entry. See discussion on page 22.35.

historysaveforms is a list of expressions that are evaluated under errorset protection each time historysave creates a new event, i.e. each time there was an interaction with the user, but not when performing an operation that is being redone. The expressions on historysaveforms are presumably executed for effect, and can use the value of history and id, as well as the value of event, which is the current, about to be executed, event, i.e. the value historysave is going to return.

lispxstorevalue[event;value] used by lispx for storing the value of an event. Can be advised by user, e.g. to watch for particular values or perform other monitoring functions.

lispxfind[history;line;type;backup;-] line is an event specification, type specifies the format of the value to be returned by lispxfind, and can be either ENTRY, ENTRIES, COPY, COPIES, INPUT, or REDO. lispxfind parses line, and uses historyfind to find the corresponding events. lispxfind then assembles and returns the appropriate structure.

lispxfind incorporates the following special features:

- 1) if backup=T, lispxfind interprets line in the context of the history list *before* the current event was added. This feature is used, for example, by valueof, so that (VALUEOF -1) will not refer to the valueof event itself;
- 2) if line=NIL and the last event is an UNDO, the next to the last event is taken. This permits the user to type UNDO followed by REDO or USE;
- 3) lispxfind recognizes @@, and substitutes archivelst for history (see page 22.12); and
- 4) lispxfind recognizes @, and retrieves the corresponding event(s) from the property list of the atom following @.

historyfind[lst;index;mod;eventadr;-]

searches lst and returns the tails of lst beginning with the event corresponding to eventadr. lst, index, and mod are as described on page 22.34. eventadr is an event address, as described on page 22.10-12, e.g., (43), (-1), (FOO FIE), (LOAD ← FOO), etc. If historyfind cannot find x, it generates an error.

+ **historymatch[input;pat;event]** used by historyfind for "matching" when eventadr specifies a pattern. Matches pat against input, the input portion of the history event event. Initially defined as editfindp[input;pat;T], but can be advised or redefined by the user.

entry #[hist;x]

hist is a history list, i.e., of the form described on page 22.34. x is one of the events on hist, i.e., (MEMB X (CAR HIST)) is true. The value of entry # is the event number for x.

valueof[x]

is an nlambda, nospread function for obtaining the value of the event specified by x, e.g., (VALUEOF -1), (VALUEOF LOAD 1), etc. valueof returns a list of the corresponding values if x specifies a multiple event.

changeslice[n;history;-]

changes time-slice for history to n. If history is NIL, changes both edithistory and lispvhxhistory.

Note: the effect of *increasing* a time-slice is gradual: the history list is simply allowed to grow to the corresponding length before any events are forgotten. *Decreasing* a time-slice will immediately remove a sufficient number of the older events to bring the history list down to the proper size. However, changeslice is undoable, so that these events are (temporarily) recoverable. Thus if the user wants to recover the storage associated with these events without waiting n more events for the changeslice event to be forgotten, he must perform a FORGET command.

saveset[name;value;topflg;flg]

an undoable set (see page 22.33). saveset scans the pushdown list looking for the last binding of name, sets name to value, and returns value.

If the binding changed was a top level binding, name is added to spellings3 (see Section 17). Furthermore, if the old value was not NOBIND, and was also not equal to the new value, saveset calls the file package to update the necessary file records. Then, if dfnflg is not equal to T, saveset prints (name RESET), and saves the old value on the property list of name, under the property VALUE. If flg=NOPRINT, saveset saves the old value, but does not print the message. This option is used by unset.

If topflg=T, saveset operates as above except that it does not scan the pushdown list but goes right to name's value cell, e.g., rpaqq[x;y] is simply saveset[x;y;T]. When topflg is T, and dfnflg is ALLPROP and the old value was not NOBIND, saveset simply stores value on the property list of name under the property VALUE, and returns value. This option is used for loading files without disturbing the current value of variables (see Section 14).

If flg=NOSAVE, saveset does *not* save the old value on the property list, nor does it add name to spellings3. However, the call to saveset is still undoable. This option is used by /set. If

+

`flg=NOSTACKUNDO`, `saveset` is undoable only if the binding being changed is a top-level binding, i.e. this says when resetting a variable that has been rebound, don't bother to make it undoable. This option is used by `rpaq`, `rpaqq`, and `addtovar`.

+
+
+

`unset[name]`

if `name` does not contain a property `VALUE`, `unset` generates an error. Otherwise `unset` calls `saveset` with `name`, the property value, `topflg=T`, and `flg=NOPRINT`.

`undosave[undoform;:]`⁸⁵

if `lispvhist` is not `NIL` (see discussion on page 22.35), and `get[lispvhist;SIDE]` is not equal to `NOSAVE`, `undosave` adds `undoform` to the value of the property `SIDE` on `lispvhist`, creating a `SIDE` property if one does not already exist. The form of `undoform` is `(fn . args)`,⁸⁶ i.e., `undoform` is undone by performing `apply[car[undoform];cdr[undoform]]`. For example, if the definition of `FOO` is `def`, `/putd[FOO;newdef]` will cause a call `undosave` with `undoform = (/PUTD FOO def)`.

*

`car` of the `SIDE` property is the number of "undosaves", i.e., length of `cdr` of the `SIDE` property, which is the list of `undoforms`. Each call to `undosave` increments this count, and adds `undoform` to the front of the list, i.e., just after the count. When the count reaches the value of `#undosaves` (initially 50),⁸⁷ `undosave` prints a message asking the user if he wants to continue saving.⁸⁸ If the user answers NO or defaults, `undosave` discards the previously saved information for this event, and makes `NOSAVE` be the value of the property `SIDE`, which disables any further saving for this event. If the user answers YES, `undosave` changes the count to -1, which is then never incremented, and continues saving.⁸⁹

*

`/rplnode[x;a;d]`

Undoably performs `rplaca[x;a]` and `rplacd[x;d]`. Value is `x`. Generates an error, `ILLEGAL ARG`, if `x` is not a list. The principle advantage of `/rplnode` is that when `x` is a list, `/rplnode` saves its undo information as `cons[x;cons[car[x];cdr[x]]]`, i.e., `(x originalcar . originalcdr)`, and therefore requires only 3 cells of storage, instead of the 8 that would be required for a `/rplaca` and a

⁸⁵ `Undosave` has a second optional argument, `histentry`, which can be used to specify `lispvhist`. If both `histentry` and `lispvhist` are `NIL`, `undosave` is a no-op.

+

⁸⁶ Except for `/rplnode`, as described below.

*

⁸⁷ `#undosaves=NIL` is equivalent to `#undosaves=infinity`.

+

⁸⁸ If `#undosaves` is negative, when the count reaches `|#undosaves|`, `undosave` simply stops saving without printing any messages or interacting with the user.

+

⁸⁹ `load` initializes the count on `SIDE` to -1, so that regardless of the value of `#undosaves`, no message will be printed, and the `load` will be undoable.

/rplacd that saved their information as described earlier.⁹⁰

/rplnode has a BLKLIBRARYDEF.

/rplnode2[x;y] same as /rplnode[x;car[y];cdr[y]].

Note: for consistency, there are definitions for both rplnode and rplnode2, although their primary reason for existence is the undoable versions.

new/fn[fn] After the user has defined /fn, new/fn performs the necessary housekeeping operations to make fn be undoable.

For example, the user could define /radix as
 $(\text{LAMBDA } (\text{X}) (\text{UNDOSAVE} (\text{LIST} (\text{QUOTE} /RADIX) (\text{RADIX X}))))$ and then perform new/fn[radix], and radix would then be undoable when typed in or in testmode.

lispx/[x;fn;vars] performs the substitution of / functions for destructive functions. If fn is not NIL, it is the name of a function, and x is its argument list. If fn is NIL, x is a form. In both cases, lispx/ returns x with the appropriate substitutions. Vars is a list of bound variables (optional).

lispx/ incorporates information about the syntax and semantics of Interlisp expressions. For example, it does not bother to make undoable operations involving variables bound in x. It does not perform substitution inside of expressions car of which is NLAMBDA, i.e., has argtype 1 or 3 (unless car of the form has the property INFO value EVAL, as described in Section 20). For example, (BREAK PUTD) typed to lispx, will break on putd, not /putd. Similarly, substitution *should* be performed in the arguments for functions like mapc, rptq, etc., since these contain expressions that will be evaluated or applied. For example, if the user types mapc[(FOO1 FOO2 FOO3);PUTD] the putd must be replaced by /putd.

undolispx[line] line is an event specification. undolispx is the function that executes UNDO commands by calling undolispxl on the appropriate entry(s).

undolispxl[event;flg;-] undoes one event. The value of undolispxl is NIL if there is nothing to be undone. If the event is already undone, undolispxl

⁹⁰ Actually, /rplaca and /rplacd also use this format for saving their undo information when their first arguments are lists. However, if both a /rplaca and /rplacd are to be performed, it is still more efficient to use /rplnode (3 cells versus 6 cells).

prints ALREADY UNDONE and returns T.⁹¹ Otherwise, undolisp1 undoes the event, prints a message, e.g., SETQ UNDONE, and returns T.

Undoing an event consists of mapping down (cdr of) the property value for SIDE, and for each element, applying car to cdr, and then marking the event undone by attaching (with /attach) a NIL to the front of its SIDE property. Note that the undoing of each element on the SIDE property will usually cause undosaves to be added to the *current lispxhist*, thereby enabling the effects of undolisp1 to be undone.

undonlsetq[form;-]

is an nlambda function similar to nlsetq. undonlsetq evaluates form, and if no error occurs during the evaluation, returns list[eval[form]] and passes the undo information from form (if any) upwards.⁹² If an error does occur, the value of undonlsetq is NIL, and any changes made by / functions during the evaluation of form are undone.

undonlsetq compiles open.

undonlsetq will operate even if lispxhistory or lispxhist are NIL, or if #undosaves is or has been exceeded for this event, or is exceeded while under the scope⁹³ of the undonlsetq.

resetundo[x;stopflg]

For use in conjunction with resetlst (Section 5). resetundo[] initializes the saving of undo information and returns a value which when given back to resetundo undoes the intervening side effects.

For example, (RESETLST (RESETSAVE (RESETUNDO)) . forms) will undo the side effects of forms on normal exit, or if an error occurs or a control-D is typed. Note that (UNDOLSETQ form) could be written as:

⁹¹ If flg=T and the event is already undone, or is an undo command, undolisp1 takes no action and returns NIL. Undolisp uses this option to search for the last event to undo. Thus when line=NIL, undolisp simply searches history until it finds an event for which undolisp1 returns T, i.e., undolisp performs (SOME (CDAR LISPXHISTORY) (F/L (UNDOLISPC1 X T)))

⁹² Actually, undonlsetq does not rebind lispxhist, so that any undo information is stored directly on the history event, exactly as though there were no undonlsetq. Instead, undonlsetq simply marks the state of the undo information when it starts, so that if an error occurs, it can then know how much to undo. The purpose of this is so that if the user control-D's out of the undonlsetq, the event is still undoable.

⁹³ Caution must be exercised in using coroutines or other non-standard means of exiting while under an undonlsetq. See discussion under CHANGING AND RESTORING SYSTEM STATE in Section 5.

```
(RESETLST (RESETSAVE (RESETUNDO)
  (AND (EQ RESETSTATE 'ERROR)
    (RESETUNDO OLDVALUE)))
  . forms)
```

If stopflg=T, resetundo stops accumulating undo information it is saving on x.⁹⁴

For example,

```
(RESETLST (SETQ FOO (RESETUNDO))
  (RESETSAVE NIL (LIST 'RESETUNDO FOO))
  (ADVISE --)
  (RESETUNDO FOO T)
  . forms)
```

would cause the advice to be undone, but *not* any of the side effects in forms.

printhistory[history; line; skipfn; novalues; file]

line is an event specification. printhistory prints the events on history specified by line, e.g., (-1 THRU -10). Printing is performed via the function showprin2, so that if the value of sysprettyflg=T, events will be prettyprinted. skipfn is an (optional) functional argument that is applied to each event before printing. If its value is true, the event is skipped, i.e., not printed. If novalues=T, or novalues applied to the corresponding event is true, the value is not printed.⁹⁵

For example, the following lispmacro will define ?? as a command for printing the history list while skipping all "large events" and not printing any values.

```
(??' (PRINTHISTORY LISPXHISTORY LISPXLINE
  (FUNCTION (LAMBDA (X)
    (IGREATERP (COUNT (CAR X)) 5)))
  T T))
```

⁹⁴ Note that this has no bearing on the saving of undo information on higher resetundo's, or on being able to undo the entire event.

⁹⁵ For example, novalues is T when printing events on edithistory.

22.10 THE EDITOR AND THE PROGRAMMER'S ASSISTANT

As mentioned earlier, all of the remarks concerning "the assistant" apply equally well to user interactions with evalqt, break or the editor. The differences between the editor's implementation of these features and that of lispX are mostly obvious or inconsequential. However, for completeness, this section discusses the editor's implementation of the programmer's assistant.

The editor uses promptchar to print its prompt character, and lispXread, lispXreadp, and readline for obtaining inputs. When the editor is given an input, it calls historysave to record the input in a new event on its history list, edithistory.⁹⁶ Edithistory follows the same conventions and format as lispXhistory. However, since edit commands have no value, the editor uses the value field for saving side effects, rather than storing them under the property SIDE.

The editor recognizes and processes the four commands DO, !E, !F, and !N commands which refer to previous events on edithistory. The editor also processes UNDO itself, as described below. All other history commands⁹⁷ are simply given to lispX for execution, after first binding (resetting) lispXhistory to edithistory. The editor also calls lispX when given an E command as described in Section 9.⁹⁸

The major implementation difference between the editor and lispX occurs in undoing. Edithistory is a list of only the last n commands, where n is the value of the time-slice. However the editor provides for undoing *all* changes made in a single editing session, even if that session consisted of more than n edit commands. Therefore, the editor saves undo information independently of the edithistory on a list call undolst, (although it also stores each entry on undolst in the field of the corresponding event on edithistory.) Thus, the commands UNDO, !UNDO, and UNBLOCK, are not dependent on edithistory,⁹⁹ i.e., UNDO specifies undoing the last command on undolst, even if that event no longer appears on edithistory. The only interaction between UNDO and the history list occurs when the user types UNDO followed by an event specification. In this case, the editor calls lispXfind to find the event, and then undoes the corresponding entry on undolst. Thus the user can only undo a *specified* command within the scope of the edithistory. (Note that this is also the only way UNDO commands themselves can be undone, that is, by using the history feature, to specify the corresponding event, e.g., UNDO UNDO.)

The implementation of the actual undoing is similar to the way it is done in lispX: each command that makes a change in the structure being edited does so via a function that records the change on a variable. After the command has completed, this variable contains a list of all the pointers that

⁹⁶ Except that the atomic commands OK, STOP, SAVE, P, ?, PP and E are not recorded. In addition, number commands are grouped together in a single event. For example, 3 3 -1 is considered as one command for changing position.

⁹⁷ as indicated by their appearance on historycoms, a list of the history commands. editdefault interrogates historycoms before attempting spelling correction. (All of the commands on historycoms are also on editcoms and editcomsl so that they can be corrected if misspelled in the editor.) Thus if the user defines a lispXmacro and wishes it to operate in the editor as well, he need simply add it to historycoms. For example, RETRIEVE is implemented as a lispXmacro and works equally well in lispX and the editor.

⁹⁸ In this case, the editor uses the fifth argument to lispX, lispXflg, to specify that any history commands are to be executed by a recursive call to lispX, rather than by unread. For example, if the user types E REDO in the editor, he wants the last event on lispXhistory processed as lispX input, and not to be unread and processed by the editor.

⁹⁹ and in fact will work if edithistory=NIL, or even in a system which does not contain lispX at all.

have been changed and their original contents. Undoing that command simply involves mapping down that list and restoring the pointers.

22.11 STATISTICS

The programmer's assistant keeps various statistics about system usage, e.g., number of lisp inputs, number of undo saves, number of calls to editor, number of edit commands, number of p.a. commands, cpu time, console time, et al. These can be viewed via the function lispstats.

- * lispstats[returnvaluesflg] prints statistics. If returnvaluesflg=T, returns the statistics as a list of elements of the form (value . explanation).

The user can add his own statistics to the lisp statistics via the function addstats.

addstats[statlst] no spread, nlambda. Statlst is a list of elements of the form (statistic-name . message), e.g., (EDITCALLS CALLS TO EDITOR) (UNDOSTATS CHANGES UNDONE), etc. statistic-name is set to the cell in an unboxed array, where the corresponding statistic will be stored. This statistic can then be incremented by lispwatch.

lispwatch[stat;n] increments stat by n (or 1 if n=NIL). lispwatch has a BLKLIBRARYDEF.

The user can save his statistics for loading into a new system by performing MAKEFILE(DUMPSTATS). After the file DUMPSTATS is loaded, the statistics printed by lispstats will be the same as those that would be printed following the makefile.

22.12 GREETING AND USER PROFILES

Many of the features of Interlisp are parameterized to allow the user to adjust the system to his or her own tastes. Among the more commonly adjusted parameters are prompt# flg, dwinwait, changeslice, #rpars, lowercase, archivefn, #undosaves, fltfmt, etc. In addition, the user can modify the action of system functions in ways not specifically provided for by using advise (Section 19).

- * In order to encourage this procedure, and to make it as painless and automatic as possible, the p.a. includes a facility for both a site-profile and a user-defined profile. When Interlisp is first run, it looks for a file on the LISP directory, with name INIT, extension LISP, and if one is found, loads the file. This provides a way of setting defaults for a particular community of users, patching bugs, etc. Interlisp then looks on the user's login directory for a file with name INIT, extension LISP, and if one is found, loads that. In both cases, the loads are performed "silently" by rebinding prettyheader (Section 14) to NIL. The p.a. then prints a greeting, e.g., "HELLO, WARREN." or

"GOOD AFTERNOON, DANNY.", etc.¹⁰⁰

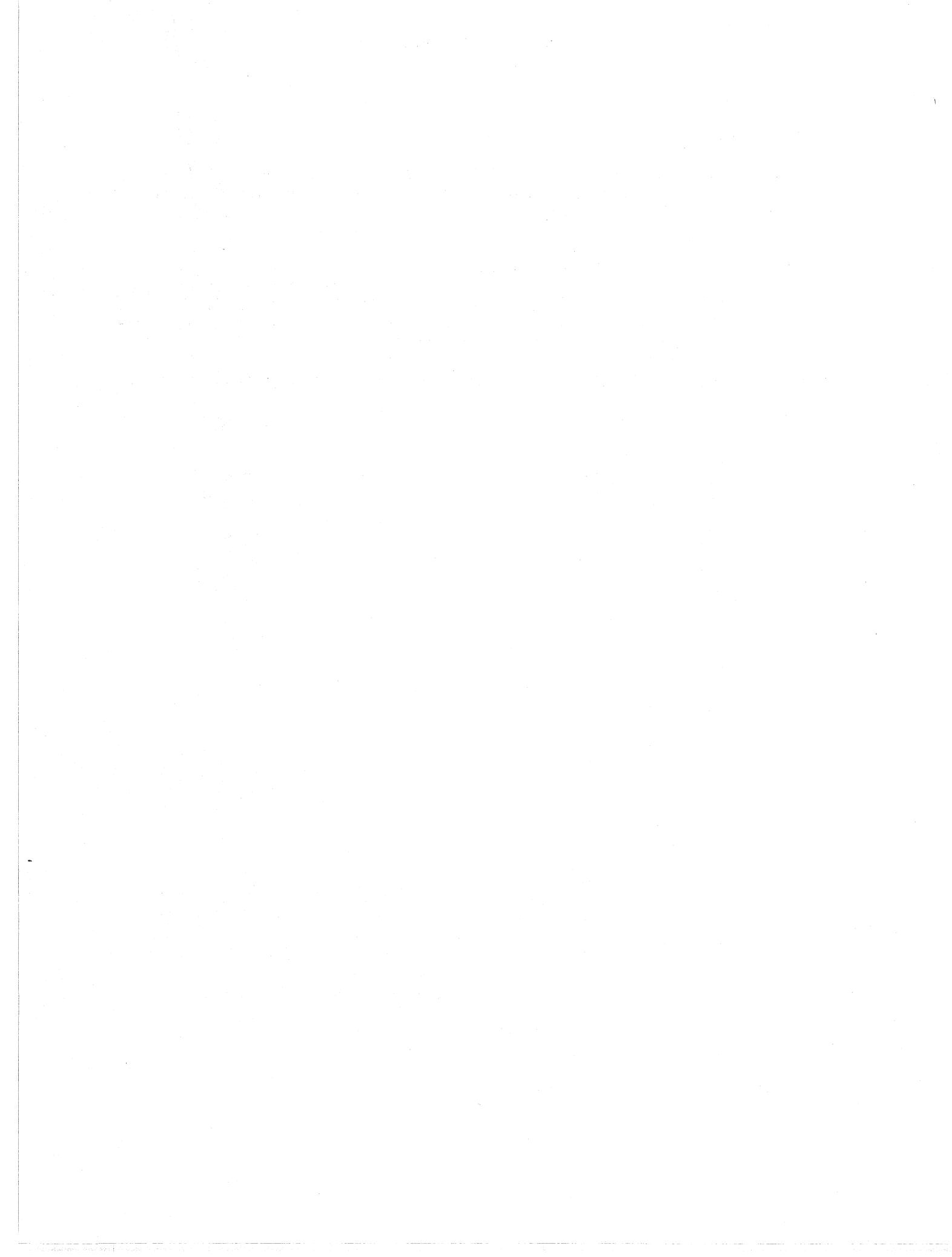
Greeting (i.e., the initialization) is undoable, and is stored as a separate event on the history list. The user can also specifically invoke the greeting operation via the function greet, for example, if he wishes to effect another user's initialization.

greet[name]

performs greeting for user whose username is name, or if name=NIL, for login name (see username and usernumber, Section 21), i.e., when Interlisp first starts up, it performs greet[]]. Before greet performs the indicated initialization, it first undoes the effects of the previous greeting.^{101 102}

greet also sets the variable username to the name for which the greeting was performed.

-
- 100 The p.a. uses the value of firstname in greeting the user. firstname should be set in the user's INIT.LISP file. In addition, the value of greetdates can be used to specify special greeting messages for various dates. greetdates is a list of elements of the form (datestring . string), e.g. ("25-DEC" . "Merry Christmas"). The user can add entries to this list in his/her INIT.LISP file, e.g. by using a prettydef command like (ADVARS (GREETDATES ("21-FEB" . "Happy Birthday"))). On the specified date, the p.a. will use the indicated salutation. If the user wishes his functions to be time stamped (see Section 9) with his initials when edited, he should also include a prettydef command (ADVARS (INITIALSLST (username . initials))).
- 101 The side effects of the greeting operation are stored on a global variable as well as the history list, thus enabling the previous greeting to be undone even if it is no longer on the history list.
- 102 In addition, makesys is advised to undo the effects of the previous greeting, thereby returning the system to a pristine state.



Index for Section 22

	Page Numbers
ADDSTATS[STATLST] NL*	22.46
AFTER (prog. asst. command)	22.22,18,28
ALL (in event specification)	22.12
ALLPROP (as value of DFNFLG)	22.40
ALREADY UNDONE (printed by system)	22.18,43
AND (in event specification)	22.12
AND (in USE command)	22.13
ARCHIVE (prog. asst. command)	22.23
ARCHIVEFLG (prog. asst. variable/parameter)	22.28
ARCHIVEFN (prog. asst. variable/parameter)	22.28,23
ARCHIVELST (prog. asst. variable/parameter)	22.34,39
BEFORE (prog. asst. command)	22.22,18,28
bell (in history event)	22.26,18,34,39
BLKLIBRARYDEF (property name)	22.42,46
CHANGESLICE[N;HISTORY;L]	22.40,7
CHECKNIL[]	22.38
CLEARBUF[FILE;FLG] SUBR	22.26
CONTIN (prog. asst. command)	22.25,28
CONTINUE SAVING? (printed by system)	22.41,31
control-D	22.26
control-E	22.26
control-G (use in history list)	22.26,18
control-N (TOPS-20)	22.27
control-U	22.27,37
DFNFLG (system variable/parameter)	22.33,40
DIR (prog. asst. command)	22.28
DO (edit command)	22.45
DWIM (prog. asst. command)	22.20-21
DWIMWAIT (dwim variable/parameter)	22.31
E (edit command)	22.45
EDITDEFAULT	22.45
EDITHISTORY (editor variable/parameter)	22.34,45,36,44
ENTRY#[HIST;X]	22.40
ESUBST[NEW;OLD;EXPR;ERRORFLG;CHARFLG]	22.13
event address	22.10-11
event number	22.7,10,17,27,40
event specification	22.10-12,16-17
EXEC (prog. asst. command)	22.24,28
EXPRESSIONS (file package type)	22.23
F (in event address)	22.10
FIRSTNAME (system variable/parameter)	22.47
FIX (prog. asst. command)	22.14,15,18
FOR (in USE command)	22.13
FORGET (prog. asst. command)	22.23,40
format and use of history list	22.34-35
FROM (in event specification)	22.11
GREET[NAME;FLG]	22.47
GREETDATES (system variable/parameter)	22.47
greeting and user profiles	22.46
HELP CLOCK (system variable/parameter)	22.17,31
history commands	22.9-23
history commands applied to history commands	22.16
history commands that fail	22.17
history list	22.4,5-12,34-35
HISTORYCOMS (editor variable/parameter)	22.45
HISTORYFIND[LST;INDEX;MOD;EVENTADDRESS; LISPXFINDFLG]	22.39,39

Page
Numbers

HISTORYMATCH[INPUT;PAT;EVENT]	22.40
HISTORYSAVE[HISTORY;ID;INPUT1;INPUT2;INPUT3; PROPS]	22.38,9,27,34-35,39,45
HISTORYSAVEFORMS (prog. asst. variable/parameter)	22.27
HISTSTRO (prog. asst. variable/parameter)	22.15,37
ILLEGAL ARG (error message)	22.41
implementation of REDO, USE, and FIX	22.15-16
IN (in USE command)	22.13
IT (prog. asst. variable/parameter)	22.26,18
LISPX[LISPXX;LISPXID;LISPXXMACROS;LISPXXUSERFN; LISPXFGLG]	22.9,35-36,13,15-17,25, 28-32,34,37-38,45
LISPXCOMS (prog. asst. variable/parameter)	22.31
LISPXEVAL[LISPXFORM;LISPXID]	22.38
LISPXFIND[HISTORY;LINE;TYPE;BACKUP;QUIETFLG]	22.39,45
LISPXFINDSPLST (prog. asst. variable/parameter) ..	22.12
LISPXHIST (prog. asst. variable/parameter)	22.35,41,43
LISPXHISTORY (prog. asst. variable/parameter) ...	22.34,36,43,45
LISPXHISTORYMACROS (prog. asst. variable/parameter)	22.28
LISPXLINE (prog. asst. variable/parameter)	22.28
LISPXXMACROS (prog. asst. variable/parameter)	22.28,37
LISPXPRINT[X;Y;Z;NODOFLG]	22.30,34
LISPXPRINTDEF[EXPR;FILE;LEFT;DEF;TAIL;NODOFLG] ..	22.30
LISPXPRINTFLG (system variable/parameter)	22.31
lispxprinting functions	22.30
LISPXPRIN1[X;Y;Z;NODOFLG]	22.30
LISPXPRIN2[X;Y;Z;NODOFLG]	22.30
LISPXREAD[FILE;RDTBL]	22.37,9,16,25,27,36,45
LISPXREADFN (prog. asst. variable/parameter)	22.37
LISPXREADP[FLG]	22.38,45
LISPXSPACES[X;Y;Z;NODOFLG]	22.30
LISPXSTATS[RETURNVALUESFLG]	22.46
LISPXSTOREVALUE[EVENT;VALUE]	22.39
LISPXTAB[X;Y;Z;NODOFLG]	22.30
LISPXTERPRI[X;Y;Z;NODOFLG]	22.30
LISPXUNREAD[LST;EVENT]	22.38
LISPXUSERFN (prog. asst. variable/parameter)	22.29,30,36-37
LISPXWATCH[STAT;N]	22.46
LISPX/[X;FN;VARS]	22.42,32
NAME (prog. asst. command)	22.21,18,22
NEW/FN[FN]	22.42
NLSETQ[NLSETX] NL	22.43
NO VALUE SAVED: (error message)	22.41
NOBIND	22.33,40
NOSAVE	22.41
NOTHING SAVED (printed by system)	22.31,18
PB (prog. asst. command)	22.24
PL (prog. asst. command)	22.23
PRINTBINDINGS[AT;POS]	22.24
PRINTHISTORY[HISTORY;LINE;SKIPFN;NOVALUES;FILE] ..	22.44,18,30-31
PRINTPROPS[AT]	22.24
programmer's assistant	22.1-36
programmer's assistant and the editor	22.45
programmer's assistant commands	22.9-27
prompt character	22.9,27,38
PROMPTCHAR[ID;FLG;HISTORY]	22.38,27,45

	Page Numbers
PROMPTCHARFORMS (prog. asst. variable/parameter)	22.27,38
PROMPT#FLG (prog. asst. variable/parameter)	22.27,38
pseudo-carriage return	22.15
READBUF (prog. asst. variable/parameter)	22.37-38
READLINE[RDTBL;LINE;LISPXFLG]	22.35,12,16,27,30,36-37, 45
REDO N TIMES (prog. asst. command)	22.12
REDO (prog. asst. command)	22.12,15,18
REDOCNT (prog. asst. variable/parameter)	22.13
REMEMBER (prog. asst. command)	22.23
REPEAT (prog. asst. command)	22.13
REREADFLG (prog. asst. variable/parameter)	22.37,39
RESET (printed by system)	22.33,40
RESETFORMS (system variable/parameter)	22.28
RESETUNDO[X;STOPFLG]	22.43,34
restoring input buffers	22.26
RETRIEVE (prog. asst. command)	22.22,18,28
RETRY (prog. asst. command)	22.17,18
RPAQ[X;Y] NL	22.33
RPAQQ[X;Y] NL	22.33
SAVESET[NAME;VALUE;TOPFLG;FLG]	22.33,40
SAVESETQ[SETQX] NL*	22.33
SAVESETQQ[SETQX;SETQY] NL	22.33
SHOWPRINT[X;FILE;RDTBL]	22.9
SHOWPRIN2[X;FILE;RDTBL]	22.18,44
SIDE (property name)	22.35,41,43,45
spelling correction	22.12,31
spelling lists	22.12,31
SPELLINGS3 (dwim variable/parameter)	22.40
statistics	22.46
SUBSYS[FILE/FORK;INCOMFILE;OUTCOMFILE; ENTRYPOINTFLG]	22.28
SUCHTHAT (in event address)	22.11
SY (prog. asst. command)	22.28
SYSPRETTYFLG (system variable/parameter)	22.9,18,44
TESTMODE[FLG]	22.32
TESTMODEFLG (prog. asst. variable/parameter)	22.32
THRU (in event specification)	22.11
time-slice (of history list)	22.7,40
TO (in event specification)	22.11
TYPE-AHEAD (prog. asst. command)	22.25
UNDO (edit command)	22.45
UNDO (prog. asst. command)	22.18,12,33,42,45
undoing	22.31,4,32-33,40-43,45
undoing DWIM corrections	22.18,18
undoing out of order	22.33,18
undoing (in editor)	22.45
UNDOLISPX[LINE]	22.42
UNDOLISPX1[EVENT;FLG;DWIMCHANGES]	22.42
UNDOLST (editor variable/parameter)	22.45
UNDONE (printed by system)	22.18,43
UNDONLSETQ[FORM] NL	22.43,34
UNDOSAVE[UNDOFORM;HISTENTRY]	22.41,35
unreading	22.9,15,38
UNSET[NAME]	22.41,33
UNTIL (use with REDO)	22.12
USE (prog. asst. command)	22.13-14,15,18

Page
Numbers

USEREXEC[LISPXID;LISPXXMACROS;LISPXXUSERFN]	22.37
USERLISPXPRINT[X;FILE;Z;NODOFLG]	22.30
USERNAME (prog. asst. variable/parameter)	22.47
USE-ARGS (property name)	22.34
VALUE (property name)	22.33, 40-41
VALUEOF[X] NL*	22.26, 40, 35
WHILE (use with REDO)	22.12
! (use in history commands)	22.13
!E (edit command)	22.45
!F (edit command)	22.45
!N (edit command)	22.45
"<c.r.>" (use in history commands)	22.15, 37
#UNDOSAVES (prog. asst. variable/parameter)	22.31, 41, 43
\$ (<esc>) (prog. asst. command)	22.19-20
\$BUFS (<esc>BUFS) (prog. asst. command)	22.26
ARCHIVE (property name)	22.28, 23
ERROR (property name)	22.34, 19
GROUP (property name)	22.34, 35, 39
HISTORY (property name)	22.34, 35
LISPXPRINT (property name)	22.31, 34
PRINT (property name)	22.34
... (printed following a carriage-return)	22.36
... (prog. asst. command)	22.17, 18
/ functions	22.32, 42
/RPLNODE[X;A;D]	22.41, 42
/RPLNODE2[X;Y]	22.42
; (prog. asst. command)	22.24
<esc> (prog. asst. command)	22.19-20
<esc>BUFS (prog. asst. command)	22.26
= (in event address)	22.10
? (prog. asst. command)	22.24
?? (prog. asst. command)	22.17, 18
@ (in event specification)	22.39
@@ (in event specification)	22.12, 23, 39
\ (in event address)	22.10
↔ (in event address)	22.10

SECTION 23

CLISP - CONVERSATIONAL LISP¹

23.1 INTRODUCTION

The syntax of LISP is very simple, in the sense that it can be described concisely, but not in the sense that LISP programs are easy to read or write! This simplicity of syntax is achieved by, and at the expense of, extensive use of explicit structuring, namely grouping through parenthesization. Unlike many languages, there are no reserved words in LISP such as IF, THEN, AND, OR, FOR, DO, BEGIN, END, etc., nor reserved characters like +, -, *, /, =, <, etc.² This eliminates entirely the need for parsers and precedence rules in the LISP interpreter and compiler, and thereby makes program manipulation of LISP programs straightforward. In other words, a program that "looks at" other LISP programs does not need to incorporate a lot of syntactic information. For example, a LISP interpreter can be written in one or two pages of LISP code ([McC1], pp. 70-71). It is for this reason that LISP is by far the most suitable, and frequently used, programming language for writing programs that deal with other programs as data, e.g., programs that analyze, modify, or construct other programs.

However, it is precisely this same simplicity of syntax that makes LISP programs difficult to read and write (especially for beginners). 'Pushing down' is something programs do very well, and people do poorly. As an example, consider the following two "equivalent" sentences:

"The rat that the cat that the dog that I owned chased caught ate the cheese."

versus

"I own the dog that chased the cat that caught the rat that ate the cheese."

Natural language contains many linguistic devices such as that illustrated in the second sentence above for minimizing embedding, because embedded sentences are more difficult to grasp and understand than equivalent non-embedded ones (even if the latter sentences are somewhat longer). Similarly, most high level programming languages offer syntactic devices for reducing apparent depth and complexity of a program: the reserved words and infix operators used in ALGOL-like languages simultaneously delimit operands and operations, and also convey meaning to the programmer. They are far more intuitive than parentheses. In fact, since LISP uses parentheses (i.e., lists) for almost all syntactic forms, there is very little information contained in the parentheses

¹ CLISP was designed and implemented by W. Teitelman. It is discussed in [Tei5].

² except for parentheses (and period), which are used for indicating structure, and space and end-of-line, which are used for delimiting identifiers.

for the person reading a LISP program, and so the parentheses tend mostly to be ignored: the meaning of a particular LISP expression for people is found almost entirely in the words, not in the structure. For example, the following expression

(COND (EQ N 0) 1) (T TIMES N FACTORIAL ((SUB1 N)))

is recognizable as FACTORIAL even though there are five misplaced or missing parentheses. Grouping words together in parentheses is done more for LISP's benefit, than for the programmer's.

CLISP is designed to make Interlisp programs easier to read and write by permitting the user to employ various infix operators, IF-THEN-ELSE statements, FOR-DO-WHILE-UNLESS-FROM-TO-etc. expressions, which are automatically converted to equivalent Interlisp expressions when they are first interpreted. For example, FACTORIAL could be written in CLISP:

(IF N=0 THEN 1 ELSE N*(FACTORIAL N-1))

Note that this expression would become an equivalent COND after it had been interpreted once, so that programs that might have to analyze or otherwise process this expression could take advantage of the simple syntax.

There have been similar efforts in other LISP systems, most notably the MLISP language at Stanford [Smil]. CLISP differs from these in that it does not attempt to *replace* the LISP syntax so much as to *augment* it. In fact, one of the principal criteria in the design of CLISP was that users be able to freely intermix LISP and CLISP without having to identify which is which. Users can write programs, or type in expressions for evaluation, in LISP, CLISP, or a mixture of both. In this way, users do not have to learn a whole new language and syntax in order to be able to use selected facilities of CLISP when and where they find them useful.

CLISP is implemented via the error correction machinery in Interlisp (see Section 17). Thus, any expression that is well-formed from Interlisp's standpoint will never be seen by CLISP (i.e., if the user defined a function IF, he would effectively turn off that part of CLISP). This means that interpreted programs that do not use CLISP constructs do not pay for its availability by slower execution time. In fact, the Interlisp interpreter does not "know" about CLISP at all. It operates as before, and when an erroneous form is encountered, the interpreter calls an error routine which in turn invokes the Do-What-I-Mean (DWIM) analyzer which contains CLISP. If the expression in question turns out to be a CLISP construct, the equivalent Interlisp form is returned to the interpreter. In addition, the original CLISP expression, is modified so that it *becomes* the correctly translated Interlisp form. In this way, the analysis and translation are done only once.

Integrating CLISP into the Interlisp system (instead of, for example, implementing it as a separate preprocessor) makes possible Do-What-I-Mean features for CLISP constructs as well as for pure LISP expressions. For example, if the user has defined a function named GET-PARENT, CLISP would know not to attempt to interpret the form (GET-PARENT) as an arithmetic infix operation. (Actually, CLISP would never get to see this form, since it does not contain any errors.) If the user mistakenly writes (GET-PRAENT), CLISP would know he meant (GET-PARENT), and not (DIFFERENCE GET PRAENT), by using the information that PRAENT is not the name of a variable, and that GET-PARENT is the name of a user function whose spelling is "very close" to that of GET-PRAENT. Similarly, by using information about the program's environment not readily available to a preprocessor, CLISP can successfully resolve the following sorts of ambiguities:

- 1) (LIST X*FACT N), where FACT is the name of a variable, means (LIST (X*FACT) N).

- 2) (`LIST X*FACT N`), where `FACT` is *not* the name of a variable but instead is the name of a function, means (`LIST X*(FACT N)`), i.e., `N` is `FACT`'s argument.
- 3) (`LIST X*FACT(N)`), `FACT` the name of a function (and not the name of a variable), means (`LIST X*(FACT N)`).
- 4) cases (1), (2) and (3) with `FACT` misspelled!

The first expression is correct both from the standpoint of CLISP syntax and semantics and the change would be made without the user being notified. In the other cases, the user would be informed or consulted about what was taking place. For example, to take an extreme case, suppose the expression (`LIST X*FCCT N`) were encountered, where there was both a function named `FACT` and a variable named `FCT`. The user would first be asked if `FCCT` were a misspelling of `FCT`. If he said YES, the expression would be interpreted as (`LIST (X*FCT) N`).³ If he said NO, the user would be asked if `FCCT` were a misspelling of `FACT`, i.e., if he intended `X*FCCT N` to mean `X*(FACT N)`. If he said YES to this question, the indicated transformation would be performed. If he said NO, the system would then ask if `X*FCCT` should be treated as CLISP, since `FCCT` is not the name of a (bound) variable.⁴ If he said YES, the expression would be transformed, if NO, it would be left alone, i.e., as (`LIST X*FCCT N`). Note that we have not even considered the case where `X*FCCT` is itself a misspelling of a variable name, e.g., a variable named `XFCT` (as with `GET-PRAENT`). This sort of transformation would be considered after the user said NO to `X*FCCT N -> X*(FACT N)`. The graph of the possible interpretations for (`LIST X*FCCT N`) where `FCT` and `XFCT` are the names of variables, and `FACT` is the name of a function, is shown in Figure 23-1 below.

-
- 3 Through this discussion, we speak of CLISP or DWIM asking the user. Actually, if the expression in question was typed in by the user for immediate execution, the user is simply informed of the transformation, on the grounds that the user would prefer an occasional misinterpretation rather than being continuously bothered, especially since he can always retype what he intended if a mistake occurs, and ask the programmer's assistant to UNDO the effects of the mistaken operations if necessary. For transformations on expressions in user programs, the user can inform CLISP whether he wishes to operate in CAUTIOUS or TRUSTING mode. In the former case (most typical) the user will be asked to approve transformations, in the latter, CLISP will operate as it does on type-in, i.e., perform the transformation after informing the user.
 - 4 This question is important because many Interlisp users already have programs that employ identifiers containing CLISP operators. Thus, if CLISP encounters the expression `A/B` in a context where either `A` or `B` are not the names of variables, it will ask the user if `A/B` is intended to be CLISP, in case the user really does have a free variable named `A/B`.

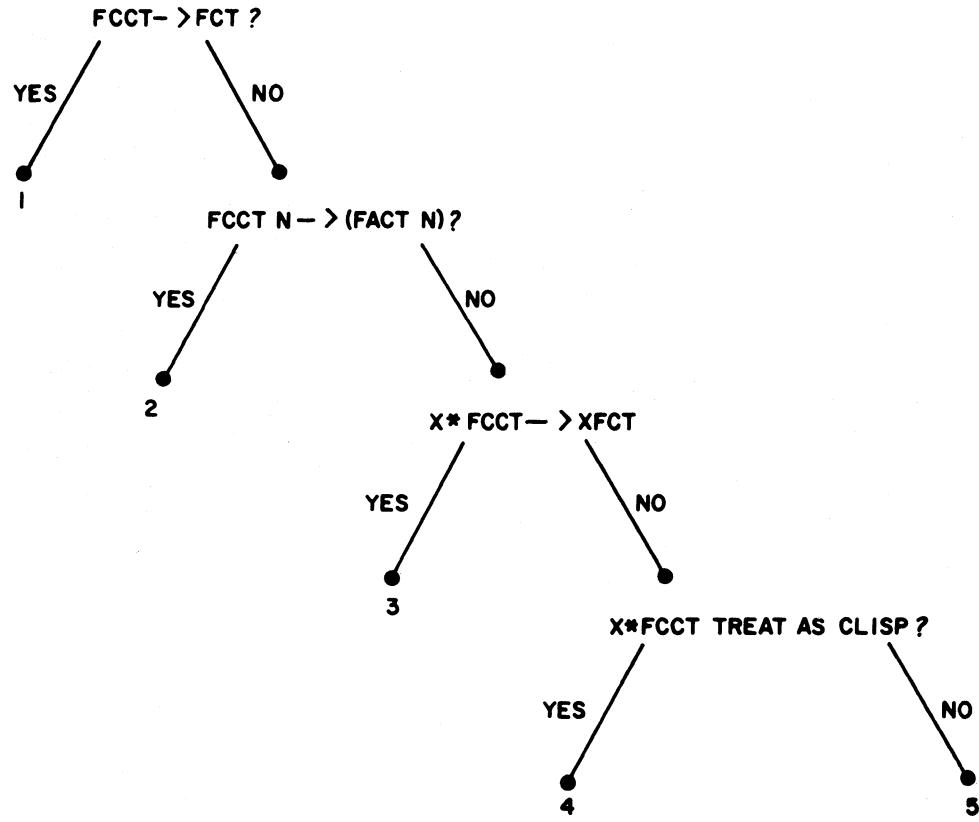


Figure 23-1

The final states for the various terminal nodes shown in the graph are:

- 1: (LIST (TIMES X FCT) N)
- 2: (LIST (TIMES X (FACT N)))
- 3: (LIST XFCT N)
- 4: (LIST (TIMES X FCCT) N)
- 5: (LIST X*FCCT N)

CLISP can also handle parentheses errors caused by typing 8 or 9 for "(" or ")". (On most terminals, 8 and 9 are the lower case characters for "(" and ")", i.e., "(" and "8" appear on the same key, as do ")" and "9".) For example, if the user writes $N*8FACTORIAL N-1$, the parentheses error can be detected and fixed before the infix operator * is converted to the Interlisp function TIMES. CLISP is able to distinguish this situation from cases like $N*8*X$ meaning (TIMES N 8 X), or $N*8X$, where $8X$ is the name of a variable, again by using information about the programming environment. In fact, by integrating CLISP with DWIM, CLISP has been made sufficiently tolerant of errors that almost everything can be misspelled! For example, CLISP can successfully translate the definition of FACTORIAL:

```
(IFF N=0 THENN1 ESLE N*8FACTTORIALNN-1)
```

to the corresponding COND, while making 5 spelling corrections and fixing the parenthesis error.⁵

This sort of robustness prevails throughout CLISP. For example, the iterative statement permits the user to say things like:⁶

```
FOR OLD X FROM M TO N DO (PRINT X) WHILE (PRIMEP X)
```

However, the user can also write OLD (X←M), (OLD X←M), (OLD (X←M)), permute the order of the operators, e.g., DO PRINT X TO N FOR OLD X←M WHILE PRIMEP X, omit either or both sets of parentheses, misspell any or all of the operators FOR, OLD, FROM, TO, DO, or WHILE, or leave out the word DO entirely! And, of course, he can also misspell PRINT, PRIMEP, M or N!⁷

CLISP is well integrated into the Interlisp system. For example, the above iterative statement translates into an equivalent Interlisp form using PROG, COND, GO, etc.⁸ When the interpreter

⁵ CLISP also contains a facility for converting from Interlisp back to CLISP, so that after running the above incorrect definition of FACTORIAL, the user could "CLISPIFY" the now correct LISP version to obtain (IF N=0 THEN 1 ELSE N*(FACTORIAL N-1)).

⁶ This expression should be self explanatory, except possibly for the operator OLD, which says X is to be the variable of iteration, i.e., the one to be stepped from N to M, but X is not to be rebound. Thus when this loop finishes execution, X will be equal to N+1.

⁷ In this example, the only thing the user could not misspell is the first X, since it specifies the *name* of the variable of iteration. The other two instances of X could be misspelled.

⁸

```
(PROG NIL
  (SETQ X M)
  $$LP(COND
    ((OR (IGREATERP X N) (NOT (PRIMEP X)))
     (RETURN)))
    (PRINT X)
    (SETQ X (ADD1 X))
    (GO $$LP))
```

subsequently encounters this CLISP expression, it automatically obtains and evaluates the translation.⁹ Similarly, the compiler "knows" to compile the translated form. However, if the user PRETTYPRINTS his program, at the corresponding point in his function, PRETTYPRINT "knows" to print the original CLISP. Similarly, when the user edits his program, the editor keeps the translation invisible to the user. If the user modifies the CLISP, the translation is automatically discarded and recomputed the next time the expression is evaluated.

In short, CLISP is not a language at all, but rather a system. It plays a role analogous to that of the programmer's assistant (Section 22). Whereas the programmer's assistant is an invisible intermediary agent between the user's console requests and the Interlisp executive, CLISP sits between the user's programs and the Interlisp interpreter.

Only a small effort has been devoted to defining the core syntax of CLISP. Instead, most of the effort has been concentrated on providing a facility which "makes sense" out of the input expressions using context information as well as built-in and acquired information about user and system programs. It has been said that communication is based on the intention of the speaker to produce an effect in the recipient. CLISP operates under the assumption that what the user said was *intended* to represent a meaningful operation, and therefore tries very hard to make sense out of it. The motivation behind CLISP is not to provide the user with many different ways of saying the same thing, but to enable him to worry less about the *syntactic* aspects of his communication with the system. In other words, it gives the user a new degree of freedom by permitting him to concentrate more on the problem at hand, rather than on translation into a formal and unambiguous language.

23.2 CLISP SYNTAX

Throughout CLISP, a non-atomic form, i.e., a list, can always be substituted for a variable, and vice versa, without changing the interpretation. For example, if the value of (FOO X) is A, and the value of (FIE Y) is B, then (LIST (FOO X)+(FIE Y)) has the same value as (LIST A+B). Note that the first expression consists of a list of *four* elements: the atom "LIST", the list "(FOO X)", the atom "+", and the list "(FIE X)", whereas the second expression, (LIST A+B), consists of a list of only *two* elements: the atom "LIST" and the atom "A+B". Since (LIST (FOO X)+(FIE Y)) is indistinguishable from (LIST (FOO X)_+(FIE Y)) because spaces before or after parentheses have no effect on the Interlisp READ program,¹⁰ to be consistent, extra spaces have no effect on atomic operands either. In other words, CLISP will treat (LIST A+_B), (LIST A_+B), and (LIST A_+_B) the same as (LIST A+B).

23.3 INFIX OPERATORS

CLISP recognizes the arithmetic infix operators +, -, *, /, and \uparrow . These are converted to IPLUS,

⁹ See page 23.22, for discussion of how translations are stored.

¹⁰ CLISP does not use its own special READ program because this would require the user to explicitly identify CLISP expressions, instead of being able to intermix Interlisp and CLISP.

IDIFFERENCE (or in the case of unary minus, IMINUS), ITIMES, IQUOTIENT, and EXPT.¹¹ The usual precedence rules apply (although these can be easily changed by the user),¹² i.e., * has higher precedence than + so that A+B*C is the same as A+(B*C), and both * and / are lower than ↑ so that 2*X↑2 is the same as 2*(X↑2). Operators of the same precedence group from left to right, e.g., A/B/C is equivalent to (A/B)/C. Minus is binary whenever possible, i.e., except when it is the first operator in a list, as in (-A) or (-A), or when it immediately follows another operator, as in A*-B.^{13 14}

Note that grouping with parentheses can always be used to override the normal precedence grouping, or when the user is not sure how a particular expression will parse.

CLISP also recognizes as infix operators =, GT, LT, GE, and LE,¹⁵ as well as various predicates, e.g., MEMBER, AND, OR, EQUAL, etc.¹⁶ AND is higher than OR, e.g., (X OR Y AND Z) is the same as (X OR (Y AND Z)), and both AND and OR are lower than the other infix operators, e.g., (X AND Y EQUAL Z) is the same as (X AND (Y EQUAL Z)). All of the infix predicates have lower precedence than Interlisp forms, i.e., (FOO X GT FIE Y) is the same as ((FOO X) GT (FIE Y)), since it is far more common to apply a predicate to two forms, than to use a Boolean as an argument to a function, e.g., (FOO (X GT (FIE Y))). However, again, the user can easily change this.

Note that only single character operators, e.g., +, ←, =, etc., can appear in the interior of an atom. All other operators must be set off from identifiers with spaces. For example, XLTY will not be recognized as CLISP.¹⁷

- 11 The I in IPLUS denotes integer arithmetic, i.e., IPLUS converts its arguments to integers, and returns an integer value. Interlisp also contains floating point arithmetic functions as well as mixed arithmetic functions (see Section 13). Floating point arithmetic functions are used in the translation if one or both of the operands are themselves floating point numbers, e.g., X+1.5 translates as (FPLUS X 1.5). In addition, CLISP contains a facility for declaring which type of arithmetic is to be used, either by making a global declaration, or by separate declarations about individual functions or variables. See section on declarations, page 23.24.
- 12 The complete order of precedence for CLISP operators is given in Figure 23-2, page 23.9.
- 13 There are some do-what-I-mean features associated with Unary minus, as in (LIST -X Y). See section on operation, page 23.51.
- 14 Note that + in front of a number will disappear when the number is read, e.g., (FOO X +2) is indistinguishable from (FOO X 2). This means that (FOO X +2) will not be interpreted as CLISP, or be converted to (FOO (IPLUS X 2)). Similarly, (FOO X -2) will not be interpreted the same as (FOO X-2). To circumvent this, always type a space between the + or - and a number if an infix operator is intended, e.g., write (FOO X + 2).
- 15 Greater Than, Less Than, Greater than or Equal to, and Less than or Equal to, respectively. GT, LT, GE, and LE are all affected by the same declarations as + and *, with the initial default to use IGREATERP and ILESSP.
- 16 Currently the complete list is MEMBER, MEMB, FMEMB, ILESSP, IGREATERP, LESSP, GREATERP, FGTP, EQ, NEQ, EQP, EQUAL, OR, and AND. New infix operators can be easily added, as described in the section on CLISP internal conventions, page 23.53. Spelling correction on misspelled infix operators is performed using clispinfixsplst as a spelling list.
- 17 In some cases, DWIM will be able to diagnose this situation as a run-on spelling error, in which case after the atom is split apart, CLISP will be able to perform the indicated transformation.

: is an infix operator used in CLISP for extracting substructures from lists.¹⁸ e.g., X:3 specifies the 3rd element of X, (FOO Y)::2 specifies the second tail of (FOO Y), i.e., (CDDR (FOO Y)), and Z:1:2 is the second element of the first element of Z, or (CADAR Z). Negative numbers may be used to indicate position counting from the end of a list, e.g., X:-1 is the last element of X, or (CAR (LAST X)), X::-1 is the last tail, i.e., (LAST X).¹⁹

\leftarrow is used to indicate assignment, e.g., X \leftarrow Y translates to (SETQ X Y).²⁰ In conjunction with : and ::, \leftarrow can also be used to perform a more general type of assignment, namely one involving structure modification. For example, X:2 \leftarrow Y means make the second element of X be Y, in Interlisp terms (RPLACA (CDR X) Y).²²²³ Negative numbers can also be used, e.g., X:-2 \leftarrow Y.²⁴ \leftarrow is also used to indicate assignment in record operations, page 23.25, and pattern match operations (Section 24).

\leftarrow has different precedence on the left from on the right. On the left, \leftarrow is a "tight" operator, i.e., high precedence, so that A+B \leftarrow C is the same as A+(B \leftarrow C). On the right, \leftarrow has broader scope so that A \leftarrow B+C is the same as A \leftarrow (B+C).

On typein, \$ \leftarrow form (< esc > \leftarrow form) is equivalent to set the "last thing mentioned".²⁵ For example, immediately after examining the value of LONGVARIABLENAME, the user could set it by typing \$ \leftarrow followed by a form.

23.4 PREFIX OPERATORS

CLISP recognizes ' and ~ as prefix operators. ' means QUOTE when it is the first character in an identifier, and is ignored when it is used in the interior of an identifier. Thus, X='Y means (EQ X (QUOTE Y)), but X=CAN'T means (EQ X CAN'T), not (EQ X CAN) followed by

18 The record facility, page 23.25, provides another way of extracting substructures by allowing the user to assign names to the various parts of the structure and then retrieve from or store into the corresponding structure by name. The pattern match facility (Section 24) also can be used to extract substructure. : is also used to indicate both record and pattern match operations.

19 The interpretation of negative numbers can be explained neatly in terms of edit commands: :-n returns what would be the current expression after executing the command -n, and ::-n returns what would be the current expression after executing -n followed by UP.

20 If x does not have a value, and is not the name of one of the bound variables of the function in which it appears, spelling correction is attempted. However, since this may simply be a case of assigning an initial value to a new free variable, DWIM will always ask for approval before making the correction.

21 Note that an atom of the form X+Y, appearing at the top level of a PROG, will not be recognized as an assignment statement because it will be interpreted as a PROG label by the Interlisp interpreter, and therefore will not cause an error, so DWIM and CLISP will never get to see it. Instead, one must write (X \leftarrow Y).

22 Note that the *value* of this operation is the value of rplaca, which is the corresponding *node*.

23 The user can indicate he wants /rplaca and /rplacd used (undoable version of rplaca and rplacd, see Section 22), or frplaca and frplacd (fast versions of rplaca and rplacd, see Section 5), by means of declarations (page 23.24). The initial default is for rplaca and rplacd.

24 which translates to (RPLACA (NLEFT X 2) Y).

25 i.e., is equivalent to (SETQ lastword form). See Section 17.

(QUOTE T). This enables users to have variable and function names with ' in them (so long as the ' is not the first character).

Following ', all operators are ignored for the rest of the identifier, e.g., '*A means (QUOTE *A), and 'X=Y means (QUOTE X=Y), not (EQ (QUOTE X) Y).²⁶

On typein, '\$ (i.e., '< esc >') is equivalent to (QUOTE value-of-lastword) (see Section 17). For example, after calling prettyprint on LONGFUNCTION, the user could move its definition to FOO by typing (MOVD '\$ 'FOO).²⁷

~ means NOT. ~ can negate a form, as in ~(ASSOC X Y), or ~X, or negate an infix operator, e.g., (A ~GT B) is the same as (A LEQ B). Note that ~A=B means (EQ (NOT A) B).

Order of Precedence of CLISP operators

,	
:	
← (left precedence)	28
- (unary), ~	29
↑	
*, /	
+, - (binary)	
← (right precedence)	
=	
Interlisp forms	
LT, GT, EQUAL, MEMBER, etc.	
AND	
OR	
IF, THEN, ELSEIF, ELSE	
iterative statement operators	

Figure 23-2

-
- 26 To write (EQ (QUOTE X) Y), one writes Y='X, or 'X_=Y. This is one place where an extra space does make a difference.
 - 27 Not (MOVD \$ 'FOO), which would be equivalent to (MOVD LONGFUNCTION 'FOO), and would (probably) cause a U.B.A. LONGFUNCTION error, nor MOVD(\$ FOO), which would actually move the definition of \$ to FOO, since DWIM and the spelling corrector would never be invoked.
 - 28 ← has a different left and right precedence, e.g., A+B+←C+D is the same as A+(B+←(C+D)). In other words, ← has minimal scope on the left and maximal scope on the right.
 - 29 When ~ negates an operator, e.g., ~=, ~LT, the two operators are treated as a single operator whose precedence is that of the second operator. When ~ negates a function, e.g., (~FOO X Y), it negates the whole form, i.e., (~(~FOO X Y)).

23.5 CONSTRUCTING LISTS - THE < AND > OPERATORS³⁰

Angle brackets are used in CLISP to indicate list construction. The appearance of a "<" corresponds to a "(" and indicates that a list is to be constructed containing all the elements up to the corresponding '>'. For example, <A B <C>> translates to (LIST A B (LIST C)). ! can be used to indicate that the next expression is to be inserted in the list as a *segment*, e.g., <A B ! C> translates to (CONS A (CONS B C)) and <! A ! B C> to (APPEND A B (LIST C)). !! is used to indicate that the next expression is to be inserted as a segment, and furthermore, all list structure to its right in the angle brackets is to be physically attached to it, e.g., <!! A B> translates to (NCONC1 A B), and <!!A !B !C> to (NCONC A (APPEND B C)).³¹³² Note that <, !, !!, and > need not be separate atoms, for example, <A B ! C> may be written equally well as < A B !C >. Also, arbitrary Interlisp or CLISP forms may be used within angle brackets. For example, one can write <FOO←(FIE X) ! Y> which translates to (CONS (SETQ FOO (FIE X)) Y). CLISPIFY converts expressions in cons, list, append, nconc, nconcl, /nconc, and /nconcl into equivalent CLISP expressions using <, >, !, and !!.

Note: brackets differ from other CLISP *operators*. For example, <A B 'C> translates to (LIST A B (QUOTE C)) even though following ', all *operators* are ignored for the rest of the identifier.³³ Note however that <A B ' _C> D> is equivalent to (LIST A B (QUOTE C)) D).

- + Section 23.17 describes how the user can define his own bracketing operators.

23.6 IF, THEN, ELSE

CLISP translates expressions employing IF|THEN|ELSEIF|ELSE into equivalent conditional expressions. The segment between IF|ELSEIF and the next THEN corresponds to the predicate of a COND clause, and the segment between THEN and the next ELSE|ELSEIF as the consequent(s). ELSE is the same as ELSEIF T THEN.

IF, THEN, ELSE, and ELSEIF are of lower precedence than all infix and prefix operators, as well as Interlisp forms, so that parentheses can be omitted between IF|ELSEIF, and THEN.³⁴ For example, (IF FOO X Y THEN --) is equivalent to (IF (FOO X Y) THEN --).³⁵ Similarly, CLISP treats (IF X THEN FOO X Y ELSE --) as equivalent to (IF X THEN (FOO X Y) ELSE --) because it does not "make sense" to evaluate a variable for effect. In other words, even if FOO were also the name of a variable, (COND (X FOO X Y)) doesn't make sense. Essentially, CLISP determines whether the segment between THEN and the

30 The <, > operator was written by P.C. Jackson.

31 Not (NCONC (APPEND A B) C), which would have the same value, but would attach C to B, and not attach either to A.

32 The user can indicate /nconc or /nconcl be used instead of nconc and nconcl by declarations.

33 Only if a previous unmatched < has been seen, e.g., (PRINT 'A>B) will print the atom A>B.

34 IF, THEN, ELSE, and ELSEIF can also be misspelled. Spelling correction is performed using clispifwordsplst as a spelling list.

35 If FOO is the name of a variable, IF FOO THEN -- is translated as (COND (FOO --)) even if FOO is also the name of a function. If the functional interpretation is intended, FOO should be enclosed in parentheses, e.g., IF (FOO) THEN --. Similary for IF -- THEN FOO ELSEIF --.

next ELSE|ELSEIF corresponds to one form or several and acts accordingly.³⁶ Thus, (IF -- THEN (FOO X) Y ELSE --) corresponds to a clause with two consequents. Similarly, (IF -- THEN FOO←X Y ELSE --) corresponds to a clause with two consequents, and is equivalent to (IF -- THEN (FOO←X) Y ELSE --).³⁷

23.7 ITERATIVE STATEMENTS

The following is an example of a CLISP iterative statement:

```
(WHILE X←(READ)~='STOP DO (PRINT (EVAL X)))
```

This statement says "READ an expression and set X to it. If X is not equal to the atom STOP, then evaluate X, print the result, and iterate."³⁸

The i.s. (iterative statement) in its various forms permits the user to specify complicated iterative statements in a straightforward and visible manner. Rather than the user having to perform the mental transformations to an equivalent Interlisp form using PROG, MAPC, MAPCAR, etc., the system does it for him. The goal was to provide a robust and tolerant facility which could "make sense" out of a wide class of iterative statements. Accordingly, the user should not feel obliged to read and understand in detail the description of each operator given below in order to use iterative statements.

Currently, the following i.s. operators are implemented: FOR, BIND, OLD, IN, FROM, TO, BY, WHEN, WHILE, UNTIL, REPEATWHILE, REPEATUNTIL, UNLESS, COLLECT, JOIN, DO, SUM, COUNT, ALWAYS, NEVER, THEREIS, AS, FIRST, FINALLY, EACHTIME. Their function is explained below. New operators can be defined as described on page 23.19. Misspellings of operators are recognized and corrected.³⁹ The order of appearance of operators is never important.⁴⁰ CLISP scans the entire statement before it begins to construct the equivalent Interlisp form.

³⁶ occasionally interacting with the user to resolve ambiguous cases.

³⁷ To write the equivalent of a singleton cond clause, i.e., a clause with a predicate but no consequent, write either nothing following the THEN, or omit the THEN entirely, e.g., (IF (FOO X) THEN ELSEIF --) or (IF (FOO X) ELSEIF --), meaning if (FOO X) is not NIL, it is the value of the cond.

³⁸ The statement translates to:

```
(PROG ($$VAL)
      $$LP(COND
            ((EQ (SETQ X (READ)) (QUOTE STOP))
             (RETURN $$VAL)))
            (PRINT (EVAL X)))
      $$ITERATE (GO $$LP))
```

³⁹ using the spelling list clispforwordsplst.

⁴⁰ DWIM and CLISP are invoked on iterative statements because car of the i.s. is not the name of a function, and hence generates an error. If the user defines a function by the same name as an i.s. operator, e.g., WHILE, TO, etc., the operator will no longer have the CLISP interpretation when it appears as car of a form, although it will continue to be treated as an i.s. operator if it appears in the interior of an i.s. To alert the user, a warning message is printed, e.g., (WHILE DEFINED, THEREFORE DISABLED IN CLISP).

DO form	specifies what is to be done at each iteration. DO with no other operator specifies an infinite loop. If some explicit or implicit terminating condition is specified, the value of the i.s. is NIL. Translate to MAPC or MAP whenever possible.
COLLECT form	like DO, except specifies that the value of <u>form</u> at each iteration is to be collected in a list, which is returned as the value of the i.s. when it terminates. Translates to MAPCAR, MAPLIST or SUBSET whenever possible. ⁴¹
JOIN form	like DO, except that the values are NCONCed. Translates to MAPCONC or MAPCON whenever possible. ⁴²
SUM form	like DO, except specifies that the values of <u>form</u> at each iteration be added together and returned as the value of the i.s., e.g., (FOR I FROM 1 TO 5 SUM I ²) is equal to 1+4+9+16+25. ⁴³
COUNT form	like DO, except counts number of times that <u>form</u> is true, and returns that count as its value.
ALWAYS form	like DO, except returns T if the value of <u>form</u> is non-NIL for all iterations (returns NIL as soon as the value of <u>form</u> is NIL), e.g., (FOR X IN Y ALWAYS (ATOM X)) is the same as (EVERY Y (FUNCTION ATOM)).
NEVER form	like ALWAYS, except returns T if the value of <u>form</u> is <i>never</i> true, i.e., NEVER <u>form</u> is the same as ALWAYS ~ <u>form</u> .
THEREIS form	returns the first value of the i.v. for which <u>form</u> is non-NIL, e.g., (FOR X IN Y THEREIS NUMBERP) returns the first number in Y, and is equivalent to (CAR (SOME Y (FUNCTION NUMBERP))). ⁴⁴

⁴¹ when COLLECT translates to a PROG, e.g., a WHILE operator appears in the iterative statement, the translation employs an open tconc using two pointers similar to that used by the compiler for compiling mapcar. To disable this translation, perform clidisable[FCOLLECT].

⁴² /NCONC, /MAPCONC, and /MAPCON are used when the declaration UNDOABLE is in effect.

⁴³ iplus, fplus, or plus will be used for the translation depending on the declarations in effect.

⁴⁴ THEREIS returns the i.v. instead of the tail (as does the function some) in order to provide an interpretation consistent with statements such as (FOR I FROM 1 TO 10 THEREIS --), where there is no tail. Note that (SOME Y (FUNCTION NUMBERP)) is equivalent to (FOR X ON Y THEREIS (NUMBERP (CAR X))).

DO, COLLECT, JOIN, SUM, ALWAYS, NEVER, and THEREIS are examples of a certain kind of i.s. operator called an i.s.type. The i.s.type specifies what is to be done at each iteration. Its operand is called the body of the iterative statement. Each i.s. must have one and only one i.s.type.

FOR var specifies the variable of iteration, or i.v., which is used in conjunction with IN, ON, FROM, TO, and BY. The variable is rebound for the scope of the i.s., except when modified by OLD as described below.

FOR vars vars a list of variables, e.g., FOR (X Y Z) IN --. The first variable is the i.v., the rest are dummy variables. See BIND below.

OLD var indicates var is *not* to be rebound, e.g., (FOR OLD X FROM 1 TO N DO -- UNTIL --).

BIND var, vars used to specify dummy variables, e.g., FOR (X Y Z) IN -- is equivalent to FOR X BIND (Y Z) IN --. BIND can be used without FOR. For example, in the i.s. shown on page 23.11, X could be made local by writing (BIND X WHILE X<-(READ)~='STOP...').

Note: FOR, OLD, and BIND variables can be initialized by using ←, e.g., (FOR OLD (X←form) BIND (Y←form)...).

IN form specifies that the i.s. is to iterate down a list with the i.v. being reset to the corresponding element at each iteration. For example, FOR X IN Y DO -- corresponds to (MAPC Y (FUNCTION (LAMBDA (X) --))). If no i.v. has been specified, a dummy is supplied, e.g., IN Y COLLECT CADR is equivalent to (MAPCAR Y (FUNCTION CADR)).

ON form same as IN except that the i.v. is reset to the corresponding *tail* at each iteration. Thus IN corresponds to MAPC, MAPCAR, and MAPCONC, while ON corresponds to MAP, MAPLIST, and MAPCON.

Note: for both IN and ON, form is evaluated before the main part of the i.s. is entered, i.e. *outside* of the scope of any of the bound variables of the i.s. For example, (FOR X BIND Y←(LIST 1 2 3) IN Y --) will *not* map down (1 2 3). + + + +

IN OLD var specifies that the i.s. is to iterate down var, with var itself being reset to the corresponding tail at each iteration, e.g., after (FOR X IN OLD L DO -- UNTIL --) finishes, L will be some tail of its original value. + + + +

IN OLD (var←form) same as IN OLD var, except var is first set to value of form.

ON OLD var	same as IN OLD <u>var</u> except the i.v. is reset to the current value of <u>var</u> at each iteration, instead of to car[<u>var</u>].
ON OLD (var+form)	same as ON OLD <u>var</u> , except <u>var</u> is first set to value of <u>form</u> .
+ INSIDE form	like IN, except treats first non-list, non-NIL tail as the last element of the iteration, e.g., INSIDE '(A B C D . E) iterates five times with the i.v. set to E on the last iteration. Similarly, INSIDE 'A is equivalent to INSIDE '(A), i.e., will iterate once.
+ + +	
WHEN form	provides a way of excepting certain iterations. For example, (FOR X IN Y COLLECT X WHEN NUMBERP X) collects only the elements of Y that are numbers.
UNLESS form	same as WHEN except for the difference in sign, i.e., WHEN Z is the same as UNLESS ~Z.
WHILE form	provides a way of terminating the i.s. WHILE <u>form</u> evaluates <u>form</u> <i>before</i> each iteration, and if the value is NIL, exits.
UNTIL form	Same as WHILE except for difference in sign, i.e., WHILE <u>form</u> is equivalent to UNTIL ~ <u>form</u> .
UNTIL n	n a number, equivalent to UNTIL (i.v. GT n).
REPEATWHILE form	same as WHILE except the test is performed after the evalution of the body, but before the i.v. is reset for the next iteration.
REPEATUNTIL form	same as UNTIL, except the test is performed after the evaluation of the body.
REPEATUNTIL n	n a number, equivalent to REPEATUNTIL (i.v., GT n).
FROM form	is used to specify an initial value for a numerical i.v. The i.v. is automatically incremented by 1 after each iteration (unless BY is specified). If no i.v. has been specified, a dummy i.v. is supplied and initialized, e.g., (COLLECT SQRT FROM 2 TO 5) returns (1.414 1.732 2.0 2.236).
TO form	is used to specify the final value for a numerical i.v. If FROM is not specified, the i.v. is initialized to 1. If no i.v. has been specified, a

dummy i.v. is supplied and initialized. If BY is not specified, the i.v. is automatically incremented by 1 after each iteration.⁴⁵ When the i.v. is definitely being *incremented*, i.e., either BY is not specified, or its operand is a positive number, the i.s. terminates when the i.v. exceeds the value of form⁴⁶ e.g., (FOR X FROM 1 TO 10 --), is equivalent to (FOR X FROM 1 UNTIL (X GT 10) --).

Similarly, when the i.v. is definitely being decremented the i.s. terminates when the i.v. becomes *less* than the value of form (see description of BY).

BY form (with IN/ON) If IN or ON have been specified, the value of form determines the *tail* for the next iteration, which in turn determines the value for the i.v. as described earlier, i.e., the new i.v. is car of the tail for IN, the tail itself for ON. In conjunction with IN, the user can refer to the current tail within form by using the i.v. or the operand for IN/ON, e.g., (FOR Z IN L BY (CDDR Z) ...) or (FOR Z IN L BY (CDDR L) ...). At translation time, the name of the internal variable which holds the value of the current tail is substituted for the i.v. throughout form. For example, (FOR X IN Y BY (CDR (MEMB 'FOO (CDR X))) COLLECT X) specifies that after each iteration, cdr of the current tail is to be searched for the atom FOO, and (cdr of) this latter tail to be used for the next iteration.

BY form (without IN/ON)

If IN or ON have not been used, BY specifies how the i.v. itself is reset at each iteration. If FROM or TO have been specified, the i.v. is known to be numerical, so the new i.v. is computed by adding the value of form (which is reevaluated each iteration) to the current value of the i.v., e.g., (FOR N FROM 1 TO 10 BY 2 COLLECT N) makes a list of the first five odd numbers.

If form is a positive number,⁴⁷ the i.s. terminates when the value of the i.v. *exceeds* the value of TO's operand. If form is a negative number, the i.s. terminates when the value of the i.v. becomes *less* than TO's operand, e.g., (FOR I FROM N TO M BY -2 UNTIL (I LT M) ...). Otherwise, the terminating condition for each iteration depends on the value of form for that iteration: if form < 0, the test is whether the i.v. is less than TO's operand, if form > 0 the test is whether the i.v. exceeds TO's operand, otherwise if form=0, the i.s. terminates unconditionally.

⁴⁵ except when both the operands to TO and FROM are numbers, and TO's operand is less than FROM's operand, e.g., FROM 10 TO 1, in which case the i.v. is decremented by 1 after each iteration. In this case, the i.s. terminates when the i.v. becomes *less* than the value of form.

⁴⁶ form is evaluated only once, when the i.s. is first entered, and its value bound to a temporary variable against which the i.v. is checked each iteration. If the user wishes to specify an i.s. in which the value of the boundary condition is recomputed each iteration, he should use WHILE or UNTIL instead of TO.

⁴⁷ form itself, not its value, which in general CLISP would have no way of knowing in advance.

If FROM or TO have not been specified and form is not a number, the i.v. is simply reset to the value of form after each iteration, e.g., (FOR I FROM N BY M ...) is equivalent to (FOR I←N BY (IPLUS I M) ...).

FIRST form

form is evaluated once before the first iteration, e.g., (FOR X Y Z IN L -- FIRST (FOO Y Z)), and FOO could be used to initialize Y and Z.

FINALLY form

form is evaluated after the i.s. terminates. For example, (FOR X IN L BIND Y←0 DO (IF ATOM X THEN Y←Y+1) FINALLY (RETURN Y)) will return the number of atoms in L.

EACHTIME form

form is evaluated at the beginning of each iteration before, and regardless of, any testing. For example, consider (FOR I FROM 1 TO N DO (... (FOO I) ...) UNLESS (... (FOO I) ...) UNTIL (... (FOO I) ...)). The user might want to set a temporary variable to the value of (FOO I) in order to avoid computing it three times each iteration. However, without knowing the translation, he would not know whether to put the assignment in the operand to DO, UNLESS, or UNTIL, i.e., which one would be executed first. He can avoid this problem by simply writing EACHTIME J←(FOO I).

AS var

is used to specify an iterative statement involving more than one iterative variable, e.g., (FOR X IN Y AS U IN V DO --) corresponds to map2c. The i.s. terminates when any of the terminating conditions are met, e.g., (FOR X IN Y AS I FROM 1 TO 10 COLLECT X) makes a list of the first ten elements of Y, or however many elements there are on Y if less than 10.

The operand to AS, var, specifies the new i.v. For the remainder of the i.s., or until another AS is encountered, all operators refer to the new i.v. For example, (FOR I FROM I TO N1 AS J FROM 1 TO N2 BY 2 AS K FROM N3 TO 1 BY -1 --) terminates when I exceeds N1, or J exceeds N2, or K becomes less than 1. After each iteration, I is incremented by 1, J by 2, and K by -1.

OUTOF form

for use with generators (Section 12). On each iteration, the i.v. is set to successive values returned by the generator. The i.s. terminates when the generator runs out.

+ **DECLARE:** decl

inserts decl immediately following the prog variable list in the translation, or, in the case that the translation is a mapping function rather than a prog, immediately following the argument list of the lambda expression in the translation. For example (FOR X IN Y (DECLARE: (LOCALVARS X)) --). Several **DECLARE:**'s can appear in the same i.s.; the declarations are inserted in the order they appear.

DECLARE decl same as DECLARE:⁴⁸

ORIGINAL i.s.opr operand

i.s.opr will be translated using its original, built-in interpretation, independent of any user defined i.s. operators. See section on "Defining New Iterative Statement Operators" below.

MISCELLANEOUS

1. Lowercase versions of all i.s. operators are equivalent to the uppercase, e.g., (for X in Y ...).
2. Each i.s. operator is of lower precedence than all Interlisp forms, so parentheses around the operands can be omitted, and will be supplied where necessary, e.g., BIND (X Y Z) can be written BIND X Y Z, OLD (X-form) as OLD X-form, WHEN (NUMBERP X) as WHEN NUMBERP X, etc.
3. RETURN or GO may be used in any operand. (In this case, the translation of the iterative statement will always be in the form of a PROG, never a mapping function.) RETURN means return from the i.s. (with the indicated value), *not* from the function in which the i.s appears. GO refers to a label elsewhere in the function in which the i.s. appears, except for the labels \$\$LP, \$\$ITERATE, and \$\$OUT which are reserved, as described in 6 below.
4. In the case of FIRST, FINALLY, EACHTIME, DECLARE: or one of the i.s.ops, e.g., DO, COLLECT, SUM, etc., the operand can consist of more than one form, e.g., COLLECT (PRINT X:1) X:2, in which case a PROGN is supplied.
5. Each operand can be the name of a function, in which case it is applied to the (last) i.v.^{49 50 51} e.g., FOR X IN Y DO PRINT WHEN NUMBERP, is the same as FOR X IN Y DO (PRINT X) WHEN (NUMBERP X). Note that the i.v. need not be explicitly specified, e.g., IN Y DO PRINT WHEN NUMBERP will work.
6. While the exact form of the translation of an iterative statement depends on which operators are present, a PROG will always be used whenever the i.s. specifies dummy variables, i.e., if a BIND operator appears, or there is more than one variable specified by a FOR operator, or a GO, RETURN, or a reference to the variable \$\$VAL appears in any of the operands. When a PROG is used, the form of the translation is:

⁴⁸ Note that since declare is also the name of a function, DECLARE cannot be used as an i.s. operator when it appears as car of a form, i.e. as the first i.s. operator in an iterative statement. However, declare (lower-case version) *can* be the first i.s. operator.

⁴⁹ For i.s.ops, e.g., DO, COLLECT, JOIN, the function is always applied to the first i.v. in the i.s., whether explicitly named or not. For example, (IN Y AS I FROM 1 TO 10 DO PRINT) prints elements on Y, not integers between 1 and 10.

⁵⁰ Note that this feature does not make much sense for FOR, OLD, BIND, IN, or ON, since they "operate" before the loop starts, when the i.v. may not even be bound.

⁵¹ In the case of BY in conjunction with IN, the function is applied to the current *tail* e.g., FOR X IN Y BY CDDR ..., is the same as FOR X IN Y BY (CDDR X)... See page 23.15.

```
(PROG variables
  {initialize}
  $$LP {eachtime}
  {test}
  {body}
  $$ITERATE
  {aftertest}
  {update}
  (GO $$LP)
  $$OUT {finalize}
  (RETURN $$VAL))
```

where {test} corresponds to that portion of the loop that tests for termination and also for those iterations for which {body} is not going to be executed, (as indicated by a WHEN or UNLESS); {body} corresponds to the operand of the i.s.opr, e.g., DO, COLLECT, etc.; {aftertest} corresponds to those tests for termination specified by REPEATWHILE or REPEATUNTIL; and {update} corresponds to that part that resets the tail, increments the counter, etc. in preparation for the next iteration. {initialize}, {finalize}, and {eachtime} correspond to the operands of FIRST, FINALLY, and EACHTIME, if any.

Note that since {body} always appears at the top level of the PROG, the user can insert labels in {body}, and go to them from within {body} or from other i.s. operands, e.g., (FOR X IN Y FIRST (GO A) DO (FOO) A (FIE)).⁵² The user can also go to \$\$LP, \$\$ITERATE, or \$\$OUT, or explicitly set \$\$VAL.

ERRORS IN ITERATIVE STATEMENTS

An error will be generated and an appropriate diagnostic printed if any of the following conditions hold:

1. Operator with null operand, i.e., two adjacent operators, as in FOR X IN Y UNTIL DO --
2. Operand consisting of more than one form (except as operand to FIRST, FINALLY, or one of the i.s.oprs), e.g., FOR X IN Y (PRINT X) COLLECT --.
3. IN, ON, FROM, TO, or BY appear twice in same i.s.
4. Both IN and ON used on same i.v.
5. FROM or TO used with IN or ON on same i.v.
6. More than one i.s.type, e.g., a DO and a SUM.

In 3, 4, or 5, an error is not generated if an intervening AS occurs.

If an error occurs, the i.s. is left unchanged.

⁵² However, since {body} is dwimified as a list of forms, the label(s) should be added to the dummy variables for the iterative statement in order to prevent their being dwimified and possibly "corrected", e.g., (FOR X IN Y BIND A FIRST (GO A) DO (FOO) A (FIE)).

If no DO, COLLECT, JOIN or any of the other i.s.ops are specified, CLISP will first attempt to find an operand consisting of more than one form, e.g., FOR X IN Y (PRINT X) WHEN ATOM X, and in this case will insert a DO after the first form. (In this case, condition 2 is not considered to be met, and an error is not generated.) If CLISP cannot find such an operand, and no WHILE or UNTIL appears in the i.s., a warning message is printed: NO DO, COLLECT, OR JOIN: followed by the i.s.

Similarly, if no terminating condition is detected, i.e., no IN, ON, WHILE, UNTIL, TO, or a RETURN or GO, a warning message is printed⁵³: POSSIBLE NON-TERMINATING ITERATIVE STATEMENT: followed by the i.s. However, since the user may be planning to terminate the i.s. via an error, control-E, or a retfrom from a lower function, the i.s. is still translated.

DEFINING NEW ITERATIVE STATEMENT OPERATORS

The following function is available for defining new or redefining existing iterative statement operators:

i.s.opr[name;form;others;evalflg]

name is the name of the new i.s.opr. If form is a list, name will be a new i.s.type, and form its body.⁵⁴

For example, for COLLECT, form would be (SETQ \$\$VAL (NCONC1 \$\$VAL BODY))
For SUM, form would be (\$\$VAL+\$\$VAL+BODY),⁵⁵ others would be (FIRST \$\$VAL+0).
For NEVER: (IF BODY THEN \$\$VAL+NIL (GO \$\$OUT)),⁵⁶ and for
THEREIS: (IF BODY THEN \$\$VAL+I.V. (GO \$\$OUT)).

others is an (optional) list of additional i.s. operators and operands which will be added to the i.s. at the place where name appears. If form is NIL, name is a new i.s.opr defined entirely by others.

In both form and others, \$\$VAL can be used to reference the value to be returned by the i.s., I.V. to reference the current i.v., and BODY to reference name's operand.⁵⁷

⁵³ unless the value of clisp.i.s.gag is T. clisp.i.s.gag is initially NIL.

⁵⁴ The i.s.type is the i.s.opr that specifies what is to be done at each iteration, e.g., performing an operation (DO), collecting values on a list (COLLECT), adding numbers (SUM), searching for a particular condition (THEREIS), etc. Each i.s. can have one and only one i.s. type.

⁵⁵ \$\$VAL+BODY is used instead of (IPLUS \$\$VAL BODY) so that the choice of function used in the translation, i.e., iplus, fplus, or plus, will be determined by the declarations then in effect.

⁵⁶ (IF BODY THEN RETURN NIL) would exit from the i.s. immediately and therefore not execute the operations specified via a FINALLY (if any).

⁵⁷ In other words, the current i.v. will be substituted for all instances of I.V. and name's operand will be substituted for all instances of BODY throughout form and others.

If evalflg is T, form and others are evaluated at translation time, and their values used as described above.⁵⁸

If name was previously an i.s.opr and is being redefined, the message (name REDEFINED) will be printed (unless dfnflg=T), and all expressions using the i.s.opr name that have been translated will have their translations discarded.

Examples:

- (1) To define RCOLLECT, a version of COLLECT which uses cons instead of nconc and then reverses the list of values:

```
i.s.opr[RCOLLECT;($$VAL←(CONS BODY $$VAL));
          (FINALLY (RETURN (DREVERSE $$VAL)))]
```

- (2) To define TCOLLECT, a version of COLLECT which uses tconc:

```
i.s.opr[TCOLLECT;(TCONC $$VAL BODY);
          (FIRST $$VAL←(CONS) FINALLY (RETURN (CAR $$VAL)))]
```

- (3) To define PRODUCT: i.s.opr[PRODUCT;(\$\$VAL←\$\$VAL*BODY);(FIRST \$\$VAL+1)]

- (4) To define UPTO, a version of TO whose operand is evaluated only once:

```
i.s.opr[UPTO;NIL;(BIND $$FOO←BODY TO $$FOO)].
```

- (5) To redefine TO so that instead of recomputing form each iteration, a variable is bound to the value of form, and then that variable is used:

```
i.s.opr[TO;NIL;(BIND $$END FIRST $$END←BODY ORIGINAL TO $$END)]
```

Note the use of ORIGINAL to redefine TO in terms of its original definition.⁵⁹

i.s.opr can also be used to define synonyms for already defined i.s. operators by calling i.s.opr with form an atom, e.g., i.s.opr[WHERE;WHEN] makes WHERE be the same as WHEN. Similarly, following i.s.opr[ISTHERE;THEREIS], one can write (ISTHERE ATOM IN Y), and following i.s.opr[FIND;FOR] and i.s.opr[SUCHTHAT;THEREIS], one can write (FIND X IN Y SUCHTHAT X MEMBER Z).⁶⁰

+ 58 lstvars is a list of dummy variable names used by the iterative statement translator. If the user wishes to obtain a dummy variable for use in translation, and be sure it does not clash with a dummy variable already used by some other i.s. operators, he can use car of lstvars, and reset lstvars to cdr[lstvars].

+ 59 ORIGINAL is intended for use in redefining built-in operators, since their definitions are not accessible, and hence not directly modifiable. Thus if the operator had been defined by the user via i.s.opr, ORIGINAL would not obtain its original definition. In this case, one presumably would simply modify the i.s.opr definition.

60 In the current system, WHERE is synonymous with WHEN, SUCHTHAT and ISTHERE with THEREIS, and FIND with FOR.

If form is the atom MODIFIER, then name is defined as an i.s.opr which can immediately follow another i.s. operator (i.e., an error will not be generated, as described previously). name will not terminate the scope of the previous operator, and will be stripped off when dwimify is called on its operand. OLD is an example of a MODIFIER type of operator. The MODIFIER feature allows the user to define i.s. operators similar to OLD, for use in conjunction with some other user defined i.s.opr which will produce the appropriate translation

+
+
+
+
+
+

For convenience, there is a file package command, I.S.OPRS, which dumps i.s.ops, e.g., (I.S.OPRS PRODUCT UPTO) as a file package command will print suitable expressions so that these iterative statement operators will be (re)defined when the file is loaded.

This completes the description of iterative statements.

23.8 ENGLISH PHRASES

CLISP also recognizes a limited but expandable set of english-like constructions of the form "A is B", e.g., FOO IS A NUMBER, Z IS NOT A STRING, (CDDR X) ISN'T A TAIL OF Y. Both subject and relation can be "distributed", e.g., X AND Y ARE ATOMIC is equivalent to X IS ATOMIC AND Y IS ATOMIC. Similarly, Z IS AN ARRAY OR A LIST is equivalent to Z IS AN ARRAY OR Z IS A LIST, and A AND B ARE NUMBERS AND LESS THAN 5 AND GT 0 is equivalent to the conjunction of the indicated six predicates. These constructions are translated to the corresponding LISP expressions when they are run or dwimified. In addition, clispify will convert LISP forms into "english" when clispifyenglshflg is T.

Clisp currently knows about the following unary relations in their singular and plural forms: ARRAY, ATOM, ATOMIC, FLOATING POINT NUMBER, LIST, LITATOM, LITERAL ATOM, NEGATIVE, NIL (i.e., X IS NIL), NULL, NUMBER, SMALL INTEGER, SMALL NUMBER, STRING; and the following binary relations in their singular and plural forms: EQ TO, EQUAL TO, GEQ, GREATER THAN, GT, LESS THAN, LT, MEMB OF, MEMBER OF, TAIL OF. All relationships can be negated with either NOT, N, or N'T, e.g., X IS ~LESS THEN Y, A AND B AREN'T ATOMIC. New relations can be defined via the function newisword.

newisword[sing;plu;form;vars] sing is the singular form of the new english construct, plu the plural without the subject. form is the form the singular construct translates to, and vars the parameters.

For example, "SMALL INTEGER" could be defined by newisword[(X IS A SMALL INTEGER); (ARE SMALL INTEGERS); (SMALLP X); (X)] and "TAIL OF" by newisword[(X IS A TAIL OF Y); (ARE TAILS OF Y); (TAILP X Y); (X Y)].

23.9 CLISP TRANSLATIONS

The translation of infix operators and IF|THEN|ELSE statements are handled in CLISP by replacing the CLISP expression with the corresponding Interlisp expression, and discarding the original CLISP, because (1) the CLISP expression is easily recomputable (by clispify),⁶¹ and (2) the Interlisp expressions are simple and straightforward. In addition to saving the space required to retain both the CLISP and the Interlisp, another reason for discarding the original CLISP is that it may contain errors that were corrected in the course of translation, e.g., the user writes FOO←FOOO:1, N*8FOO X), etc. If the original CLISP were retained, either the user would have to go back and fix these errors by hand, thereby negating the advantage of having DWIM perform these corrections, or else DWIM would have to keep correcting these errors over and over.

Where (1) or (2) are not the case, e.g., with iterative statements, pattern matches, record expressions, etc.⁶² the original CLISP is retained (or a slightly modified version thereof), and the translation is stored elsewhere, usually in clispararray, a hash array.⁶³⁶⁴ The interpreter automatically checks this array using gethash when given a form car of which is not a function.⁶⁵ Similarly, the compiler performs a gethash when given a form it does not recognize to see if it has a translation, which is then compiled instead of the form. Whenever the user changes a CLISP expression by editing it, the editor automatically deletes its translation (if one exists), so that the next time it is evaluated or dwimified, the expression will be retranslated.⁶⁶ The function ppt and the edit commands PPT and CLISP: are available for examining translations, see page 23.60. Similarly, if prettytranflg is T, prettyprint will print the translations instead of the corresponding CLISP expression.⁶⁷

⁶¹ Note that clispify is sufficiently fast that it is practical for the user to configure his Interlisp system so that all expressions are automatically clispified immediately before they are presented to him. For example, he can define an edit macro to use in place of P which calls clispify on the current expression before printing it. Similarly, he can inform prettyprint to call clispify on each expression before printing it, etc.

⁶² The handling of translations for IF|THEN|ELSE statements is determined by the value of clispiftranflg. If T, the translations are stored elsewhere, and the (modified) CLISP retained as described below. If NIL, the corresponding COND replaces the IF|THEN|ELSE expression. The initial value of clispiftranflg is NIL.

⁶³ The actual storing of the translation is performed by the function clisptran, page 23.57.

⁶⁴ The user can also indicate that he wants the original clisp retained by embedding it in an expression of the form (CLISP . clisp-expression), e.g., (CLISP X:5:3) or (CLISP < A B C ! D >). In such cases, the translation will be stored remotely as described in the text. Furthermore, such expressions will be treated as clisp even if infix and prefix transformations have been disabled by setting clispflg to NIL, as described on page 23.56. In other words, the user can instruct the system to interpret as clisp infix or prefix constructs only those expressions that are specifically flagged as such. The user can also include clisp declarations by writing (CLISP declarations . form), e.g., (CLISP (CLISP: FLOATING) ...). These declarations will be used in place of any clisp declarations in the function definition. Note this feature provides a way of including clisp declarations in compiler macro definitions.

+
+
+
+

⁶⁵ CLISP translations can also be used to supply an interpretation for function objects, as well as forms, either for function objects that are used openly, i.e., appearing as car of form, function objects that are explicitly applied, as with arguments to mapping functions, or function objects contained in function definition cells. In all cases, if car of the object is not LAMBDA or NLAMBDA, the interpreter and compiler will check clispararray.

⁶⁶ If the value of clispretranflg is T, dwimify will also (re)translate any expressions which have translations stored remotely. The initial value of clispretranflg is NIL.

⁶⁷ Note that the user can always examine the translation himself by performing (GETHASH expression CLISPARRAY).

If clisparray is NIL,⁶⁸ translations are implemented instead by replacing the CLISP expression by an expression of the form (CLISP%_ translation . CLISP-expression),⁶⁹ e.g., (FOR X IN Y COLLECT (CAR X)) would be replaced by (CLISP%_ (MAPCAR Y (FUNCTION CAR)) FOR X IN Y COLLECT (CAR X)). Both the editor and prettyprint know about CLISP%_ expressions and treat them specially by suppressing the translations: Prettyprint prints just the CLISP (unless prettytranflg=T, as described below), while the editor makes the translation completely invisible, e.g., if the current expression were the above CLISP%_ expression, F MAPCAR would fail to find the MAPCAR, and (3 ON) would replace IN with ON, i.e., the editor operates as though both the CLISP%_ and the MAPCAR were not there. As with translations implemented via clisparray, if the CLISP expression is changed by editing it, the translation is automatically deleted.

CLISP%_ expressions will interpret and compile correctly: CLISP%_ is defined as an nlambda nospread function with an appropriate compiler macro. Note that if the user sets clisparray to NIL, he can then break, trace, or advise CLISP%_ to monitor the evaluation of iterative statements, pattern matches, and record operations. This technique will work even if clisparray was not NIL at the time the expressions were originally translated, since setting clisparray to NIL will effectively delete the translations, thereby causing the CLISP expressions to be retranslated when they are first encountered. Note that if the user only wishes to monitor the CLISP in a certain function, he can accomplish this by embedding its definition in (RESETVAR CLISPARRAY NIL *).

If a CLISP%_ expression is encountered and clisparray is *not* NIL, the translation is transferred to the hash array, and the CLISP%_ expression replaced by just the CLISP. Setting prettytranflg to CLISP%_ causes prettyprint to print CLISP expressions that have been translated in the form of (CLISP%_ translation . CLISP-expression), even if the translation is currently stored in clisparray. These two features together provide the user with a way of dumping CLISP expressions together with their translations so that when reloaded (and run or dwimified), the translations will automatically be transferred to clisparray.

In summary, if prettytranflg=NIL, only the CLISP is printed (used for producing listings). If prettytranflg=T, only the translation is printed (used for exporting programs to systems that do not provide CLISP, and to examine translations for debugging purposes).⁷⁰ If prettytranflg=CLISP%_, an expression of the form (CLISP%_ translation . CLISP) is printed, (used for dumping both CLISP and translations). The preferred method of storing translations is in clisparray, so that if any CLISP%_ expressions are converted while clisparray is not NIL, they will automatically be converted so as to use clisparray. If clisparray=NIL, they will be left alone, and furthermore, new translations will be implemented using CLISP%_ expressions.

⁶⁸ clisparray is initially NIL, and #clisparray is its size. The first time a translation is performed, a hash array of this size is created. Therefore to disable clisparray, both it and #clisparray should be set to NIL.

⁶⁹ CLISP%_ is an atom consisting of the six characters C, L, I, S, P, and *space*, which must be preceded by the escape character % in order for it to be included as a part of an identifier. The intent was to deliberately make this atom hard to type so as to make it unlikely that it would otherwise appear in a user's program or data, since the editor and prettyprint treat it very specially, as described above.

⁷⁰ Note that makefile will reset prettytranflg to T, using resetvar, when called with the option NOCLISP.

23.10 DECLARATIONS

Declarations are used to affect the choice of Interlisp function used as the translation of a particular operator. For example, A+B can be translated as either (IPLUS A B), (FPLUS A B), or (PLUS A B), depending on the declaration in effect. Similarly X:1↔Y can mean (RPLACA X Y), (FRPLACA X Y), or (/RPLACA X Y), and <!!A B> either (NCONC1 A B) or (/NCONC1 A B). The table below gives the declarations available in CLISP, and the Interlisp functions they indicate. *The choice of function on all CLISP transformations are affected by these declarations, i.e., iterative statements, pattern matches, record operations, as well as infix and prefix operators.*

The user can make (change) a global declaration by calling the function CLISPDEC and giving it as its argument a list of declarations, e.g., (CLISPDEC (QUOTE (FLOATING UNDOABLE))). Changing a global declaration does not affect the speed of subsequent CLISP transformations, since all CLISP transformation are table driven (i.e., property list), and global declarations are accomplished by making the appropriate internal changes to CLISP at the time of the declaration. If a function employs *local* declarations (described below), there will be a slight loss in efficiency owing to the fact that for each CLISP transformation, the declaration list must be searched for possibly relevant declarations.

Declarations are implemented in the order that they are given, so that later declarations override earlier ones. For example, the declaration FAST specifies that FRPLACA, FRPLACD, FMEMB, and FLAST be used in place of RPLACA, RPLACD, MEMB, and LAST; the declaration RPLACA specifies that RPLACA be used. Therefore, the declarations (FAST RPLACA RPLACD) will cause FMEMB, FLAST, RPLACA, and RPLACD to be used.

The initial global declaration is INTEGER and STANDARD.

TABLE OF DECLARATIONS

<u>Declaration</u>	<u>Interlisp functions to be used</u>
INTEGER or FIXED	IPLUS, IMINUS, IDIFFERENCE, ITIMES, IQUOTIENT, ILESSP, IGREATERP
FLOATING	FPLUS, FMINUS, FDIFFERENCE, FTIMES, FQUOTIENT, LESSP, FGTP
MIXED	PLUS, MINUS, DIFFERENCE, TIMES, QUOTIENT, LESSP, GREATERP
FAST	FRPLACA, FRPLACD, FMEMB, FLAST, FASSOC
UNDOABLE	/RPLACA, /RPLACD, /NCONC, /NCONC1, /MAPCONC, /MAPCON
STANDARD	RPLACA, RPLACD, MEMB, LAST, ASSOC, NCONC, NCONC1, MAPCONC, MAPCON
RPLACA, RPLACD, /RPLACA, ...	corresponding function

LOCAL DECLARATIONS

The user can also make declarations affecting a selected function or functions by inserting an expression of the form (CLISP: . declarations) immediately following the argument list, i.e., as CADDR of the definition. Such local declarations take precedence over global declarations. Declarations affecting selected variables can be indicated by lists, where the first element is the name of a variable, and the rest of the list the declarations for that variable. For example, (CLISP: FLOATING (X INTEGER)) specifies that in this function integer arithmetic be used for computations involving X, and floating arithmetic for all other computations.⁷¹ The user can also make local record declarations by inserting a record declaration, e.g., (RECORD --), (ARRAYRECORD --), etc., in the local declaration list. Local record declarations override global record declarations for the function in which they appear. Local declarations can also be used to override the global setting of certain DWIM/CLISP parameters effective only for transformations within that function, by including in the local declaration an expression of the form (variable = value), e.g., (PATVARDEFAULT = QUOTE).

The CLISP: expression is converted to a comment of a special form recognized by CLISP. Whenever a CLISP transformation that is affected by declarations is about to be performed in a function, this comment will be searched for a relevant declaration, and if one is found, the corresponding function will be used. Otherwise, if none are found, the global declaration(s) currently in effect will be used.

Local declarations are effective in the order that they are given, so that later declarations can be used to override earlier ones, e.g., (CLISP: FAST RPLACA RPLACD) specifies that FMEMB, FLAST, RPLACA, and RPLACD be used. An exception to this is that declarations for specific variables take precedence of general, function-wide declarations, regardless of the order of appearance, as in (CLISP: (X INTEGER) FLOATING).

Clispify also checks the declarations in effect before selecting an infix operator to ensure that the corresponding CLISP construct would in fact translate back to this form. For example, if a FLOATING declaration is in effect, clispify will convert (FPLUS X Y) to X+Y, but leave (IPLUS X Y) as is. Note that if (FPLUS X Y) is CLISPIFYed while a FLOATING declaration is under effect, and then the declaration is changed to INTEGER, when X+Y is translated back to Interlisp, it will become (IPLUS X Y).

23.11 THE RECORD PACKAGE⁷²

The advantages of "data-less" or data-structure-independent programming have long been known: more readable code, fewer bugs, the ability to change the data structure without having to make major modifications to the program, etc. The record package in CLISP both encourages and facilitates this good programming practice by providing a uniform syntax for creating, accessing and storing data into many different types of data structures, e.g. those employing arrays, list structures, association lists, hash links, etc., and combinations thereof, as well as removing from the user the

⁷¹ "involving" means where the variable itself is an operand. For example, with the declaration (FLOATING (X INTEGER)) in effect, (FOO X)+(FIE X) would translate to FPLUS, i.e., use floating arithmetic, even though X appears somewhere inside of the operands, whereas X+(FIE X) would translate to IPLUS. If there are declarations involving *both* operands, e.g., X+Y, with (X FLOATING) (Y INTEGER), whichever appears first in the declaration list will be used.

⁷² The record package was written by L. M. Masinter.

task of writing the various routines themselves. The user declares (once) the data structure(s) used by his programs, and thereafter indicates the manipulations of the data in a data-structure-independent manner. The record package automatically computes from the declaration(s) the corresponding Interlisp expressions necessary to accomplish the indicated access/storage operations. The user can change his data structure simply by changing the corresponding declaration(s), and his program automatically (re)adjusts itself to the new conventions.

The user informs the record package about the format of his data structures by making a record *declaration*. A record declaration defines a *record*, i.e., a data structure. The record declaration is a description of the record, associating names with its various parts, or *fields*. For example, the record declaration (RECORD MSG (ID (FROM TO) . TEXT)) describes a data structure called MSG, which contains four fields: ID, FROM, TO, and TEXT. The user can then reference these fields by name, either to retrieve their contents or to store new data into them, by using the : operator followed by the field name. For example, for the above record declaration, X:FROM would be equivalent (and translate) to (CAADDR X), and Y:TO \leftarrow Z to (CAR (RPLACA (CDADR Y) Z)).⁷³ Record operations are implemented by replacing expressions of the form X:FOO by (fetch FOO of X), and X:FOO \leftarrow Y by (replace FOO of X with Y) and then storing the translation elsewhere, usually in a hash array, as described on page 23.22. CLISP also recognizes expressions input in this form; both lower and upper case are acceptable.⁷⁴

The fields of a record can be further broken down into sub-fields by *subdeclarations* within the record, e.g.,

(RECORD NODE (POSITION . LABEL) (RECORD POSITION (XLOC . YLOC))) would permit the user to refer to POSITION, or to its subfields XLOC and YLOC.

Note that what the record declaration is really doing is specifying the *data-paths* of the structure, and thereby specifying how the corresponding access/storage operations are to be carried out. For example, the above declaration of NODE says the XLOC of a NODE is to be found as the CAR of its POSITION, which is the CAR of the NODE itself. Hence, N:XLOC \leftarrow 30 is achieved by performing (CAR (RPLACA (CAR N) 30)).

Note also that when the user writes N:XLOC, he is implicitly saying the N is an instance of the record NODE, or at least is to be treated as such for this particular operation. In other words, the interpretation of N:*field* never depends on the value of N. The record package does not provide any facility which uses *run-time* checks to determine data paths, nor is there any error checking other than that provided by Interlisp itself. For example, if N happened to be an array, N:YLOC would still compute (CDAR N).

The user can also *create* new data structures using a record declaration as a guide or template. Initial values for the contents of each field can be specified in the CREATE expression, defaulted to values specified in the record declaration, or CREATE can be instructed to use an existing datum as a model, i.e. to obtain the field values for the new datum from the corresponding fields of an

⁷³ or /RPLACA or FRPLACA, depending on the CLISP declaration in effect. Note that the *value* of X:TO \leftarrow Z is Z. In general, the value of a replacement record operation is the same as the value stored into the field. In this case, the Interlisp-10 compiler will eliminate the CAR if the value of X:TO \leftarrow Z is not actually used, e.g. if the replacement is a statement in a PROG.

+ ⁷⁴ FFETCH and FREPLACE are versions which insure FAST CLISP declarations will be in effect, /REPLACE insures UNDOABLE declarations, e.g. using the declaration for the record MSG shown above, (FREPLACE TO OF Y WITH Z) would translate as (CAR (FRPLACA (CDADR Y) Z)), and (/REPLACE TO OF Y WITH Z) would use /RPLACA.

existing datum. For example, with the above declaration of NODE, (CREATE NODE USING FOO XLOC \leftarrow 10 LABEL \leftarrow 'L1) translates to (CONS (CONS 10 (CDAR FOO)) (QUOTE L1)).

The record package also provides a facility for allowing the user to *test* if a datum is an instance of a given record via a TYPE? expression, as explained below.

RECORD (used to specify elements and tails of a list structure) is just one of several record-types currently implemented. For example, the user can specify a property list format by using the record type PROPRECORD, or that fields are to be associated with parts of the data structure via hash links by using the record-type HASHLINK, or that an entirely new data type be allocated (as described in section 3) by using the record-type DATATYPE. These are described in detail below.

As with all DWIM/CLISP facilities, the record package contains many do-what-I-mean features, spelling correction on field names, record types, etc. In addition, the record package includes a RECORDS file package command for dumping record declarations,⁷⁵ as well as the appropriate modifications to the file package (Section 14), so that files? and cleanup will inform the user about records that need to be dumped.

RECORD DECLARATIONS

A record declaration is an expression of the form

(record-type record-name fields . {record tail})

This expression is evaluated to effect the corresponding declaration.⁷⁶

1. record-type specifies the "type" of data being described by the record declaration, and thereby implicitly specifies the data paths, i.e., how the corresponding access/storage operations are performed. record-type currently is either RECORD, TYPERECORD, ARRAYRECORD, ATOMRECORD, ASSOCRECORD, PROPRECORD, DATATYPE, HASHLINK, ARRAYBLOCK or ACCESSFNS. RECORD and TYPERECORD are used to describe list structures, DATATYPE to describe user data-types, ARRAYRECORD to describe arrays, ATOMRECORD to describe (the property list of) atoms, PROPRECORD to describe lists in property list format, and ASSOCRECORD to describe association list format. HASHLINK can be used with any type of data; it simply specifies the data path to be a hash-link. ACCESSFNS is also type-less; the user specifies the data-paths in the record declaration itself, as described below.
2. record-name is a literal atom used to identify the record declaration for dumping to files via the RECORDS file package command, creating instances of the record via CREATE, and testing via TYPE?. DATATYPE and TYPERECORD declarations also use record-name to identify the

⁷⁵ The file package command INITRECORDS can be used to write expressions on a file that will, when loaded, perform whatever initialization/allocation is necessary for the indicated records, but not to write out, and hence cause to be read back in, the record declarations themselves. This facility is useful for building systems on top of Interlisp, in which the implementor may want to eliminate the record declarations from a production version of the system, but the initialization for these records must still be done.

⁷⁶ Local record declarations are performed by including an expression of this form in the CLISP declaration for that function (page 23.25), rather than evaluating the expression itself.

data structure (as described below).⁷⁷

For subdeclarations, record-name specifies the parent field that is being elaborated.

3. fields describes the structure of the record. Its exact interpretation varies with the record-type:

RECORD	<u>fields</u> is a list structure whose non-NIL literal atoms are taken as field-names to be associated with the corresponding elements and tails of a list structure. NIL can be used as a place marker to fill an unnamed field, e.g., (A NIL B) describes a three element list, with B corresponding to the third element. A number may be used to indicate a sequence of NILs, e.g. (A 4 B) is interpreted as (A NIL NIL NIL NIL B).
TYPERECORD	Similar to RECORD except that <u>record-name</u> is also used as an indicator in CAR of the datum to signify what "type" of record it is. CREATE will insert an extra field containing <u>record-name</u> at the beginning of the structure, and the translation of the access and storage functions will take this extra field into account. ⁷⁸ For example, for (TYPERECORD MSG (ID (FROM TO) . TEXT)), X:FROM translates as (CAADDR X), not (CAADR X).
ASSOCRECORD	<u>fields</u> is a list of literal atoms. The fields are stored in a-list format; i.e., ((fieldname . value) (fieldname . value) ...). Accessing is performed with <u>assoc</u> , ⁷⁹ storing with <u>putassoc</u> .
PROPRECORD	<u>fields</u> is a list of property names. The fields are stored in "property list" format; i.e., (fieldname value fieldname value ...). Accessing is performed with <u>listget</u> , storing with <u>listput</u> . Both ASSOCRECORD and PROPRECORD are useful for defining data structures in which it is often the case that many of the fields are NIL. A CREATE for these record types only stores those fields which are non-NIL. ⁸⁰
ARRAYRECORD	<u>fields</u> is a list of field-names that are associated with the corresponding elements of the array. NIL can be used as a place marker for an unnamed field (element). Positive integers can be used as abbreviation for the corresponding number of NILs. For example, (ARRAYRECORD (ORG DEST NIL ID 3 TEXT)) describes an eight element array, with ORG corresponding to the first element, ID to the fourth, and TEXT to the eighth.
HASHLINK	<u>fields</u> is either just <u>field-name</u> , i.e. an atom, or a list interpreted as

⁷⁷ For some top-level declarations, record-name is optional, e.g., (RECORD (ID (FROM TO) . TEXT)) is acceptable. However, if record-name is omitted, the user cannot specify the record by name, e.g., in CREATE expressions, or when using the RECORDS file package command.

⁷⁸ This type-field is used by the record package in the translation of TYPE? expressions.

⁷⁹ or fassoc, depending on current CLISP declarations.

⁸⁰ However, with the declaration (PROPRECORD FIE (H I J)) the expression (CREATE FIE) would still construct (H NIL), since a later operation of X:J←T could not possibly change the instance of the record if it were NIL.

(field-name arrayname arraysize). arrayname indicates the hash-array to be used; if not given, SYSHASHARRAY is used. For example, (HASHLINK (CLISP CLISPARRAY)) would permit the user to obtain the CLISP translation of X by simply writing X:CLISP. arraysize is used for initializing the hash array: if arrayname has not been initialized at the time of the declaration, it will be set to (LIST (HARRAY (OR arraysize 100))). HASHLINKs are useful as subdeclarations to other records to add additional fields to already existing data-structures.

DATATYPE

specifies that a new user data type with type name record-name be allocated via declaredatatype (see Section 3).⁸¹ When a DATATYPE declaration is given for the first time, the system allocates storage space and a type number for that data type. Thus, unlike other record-types, the records of a DATATYPE declaration are represented with a completely new Interlisp type, and not in terms of other existing types.⁸² fields is a list of field specifications, where each specification is either fieldname or (fieldname fieldtype). If fieldtype is omitted (or fieldtype=POINTER) then the field can contain a pointer to any arbitrary Interlisp datum. Other options for fieldtype are:

BITS <u>n</u>	field contains an <u>n</u> -bit unsigned integer.
BETWEEN <u>n1</u> <u>n2</u>	a generalization of BITS. field may contain an integer x, such that x is greater than or equal to <u>n1</u> and less than or equal to <u>n2</u> . Enough bits are allocated to store a number between 0 and (<u>n2-n1</u>); <u>n1</u> is appropriately added or subtracted when the field is accessed or stored.
INTEGER or FIXP	field contains a full word signed integer.
FLOATING or FLOATP	field contains a full word floating point number.
FLAG	field is a one bit field that "contains" T or NIL. +

For example, the declaration

```
(DATATYPE MSG ((FLG BITS 12) TEXT (CNT BETWEEN 10 25)
               HEAD (DATE BITS 18) (PRIO FLOATP)
               (READ? FLAG) ))
```

would define a data type MSG which occupies (in Interlisp-10) three words of storage with two pointer fields (one word), a full word floating point number, fields for an 18, 12, and 4 bit unsigned integer, and a flag (one bit), with 1 bit

⁸¹ Since the data type must be set up at run-time, the RECORDS file package command will dump a declaredatatype expression as well as the DATATYPE declaration itself. The INITRECORDS command will dump only the declaredatatype expression. +

⁸² For this reason, DATATYPE declarations should be used with caution within local declarations, since a new and different data type is allocated for each one with a different name.

left over.⁸³

- + **ARRAYBLOCK** similar to a DATATYPE declaration, except that the objects it creates and manipulates are arrays. As with DATATYPE's, the actual order of the fields of the ARRAYBLOCK may be shuffled around in order to satisfy garbage collector constraints.

+ For example,

```
(ARRAYBLOCK FOO ((F1 INTEGER) (F2 FLOATING) (F3 POINTER)
                  (F4 BETWEEN -30 -2) (F5 BITS 12)
                  (F6 FLAG) ))
```

- ACCESSFNS** fields is a list of elements of the form (field-name accessdef setdef), i.e. for each fieldname, the user specifies how it is to be accessed and set. accessdef should be a function of one argument, the datum, and will be used for accessing. setdef⁸⁴ should be a function of two arguments, the datum and the new value, and will be used for storing. Alternatively, accessdef and/or setdef may be a LAMBDA expression or a form written in terms of variables DATUM and (in the case of setdef) NEWVALUE.⁸⁵ For example, given the declaration

```
[ACCESSFNS ((FIRSTCHAR (NTHCHAR DATUM 1)
              (RPLSTRING DATUM 1 NEWVALUE))
            (RESTCHARS (SUBSTRING DATUM 2))]
```

X:FIRSTCHAR←Y would translate to (RPLSTRING X 1 Y). Since no setdef is given for the RESTCHARS field, attempting to perform X:RESTCHARS←Y would generate an error, REPLACE UNDEFINED FOR FIELD. Note that ACCESSFNS do not have a CREATE definition. However, the user may supply one in the {defaults and/or subdeclarations} of the declaration, as described below. Attempting to CREATE an ACCESSFNS record without specifying a create definition will cause an error CREATE NOT DEFINED FOR THIS RECORD.

- ATOMRECORD** fields is a list of property names, e.g., (ATOMRECORD (EXPR CODE MACRO BLKLIBRARYDEF)). Accessing is performed with gelprop, storing with putprop.⁸⁶

- + In addition to the above built-in record types, users can declare their own record types by performing the following steps:

⁸³ Fields are allocated in such a way as to optimize the storage used and not necessarily in the order specified. To store this information in a conventional RECORD list structure, e.g., (RECORD MSG (FLG TEXT CNT DATE PRIO . HEAD)), would take 5 words of list space and up to three number boxes (for FLG, DATE, and PRIO).

⁸⁴ setdef may be omitted, in which case, no store operations are allowed.

⁸⁵ accessdef and setdef can also be a property list which specify FAST, STANDARD and UNDOABLE versions of the ACCESSFNS forms, e.g. [ACCESSFNS LITATOM ((DEF (STANDARD GETD FAST FGETD) (STANDARD PUTD UNDOABLE /PUTD]) means if FAST declaration is in effect, use FGETD for fetching, if UNDOABLE, use /PUTD for saving.

⁸⁶ as with ACCESSFNS, CREATE is not initially defined for ATOMRECORD records.

- (1) add the new record-type to the value of clisprecordtypes; +
- (2) perform movd[RECORD;record-type], i.e. give the record-type the same definition as that of record; +
- (3) put on the property list of the record-type, under the property **USERRECORDTYPE**, the name of a function which will return the translation. Whenever a record declaration of type record-type is encountered, this function will be passed the record declaration as its argument, and should return a *new* record declaration which the record package will then use in its place. +

4. {record-tail} is optional. It may contain expressions of the form:

- (1) **field-name** ← **form** allows the user to specify within the record declaration the default value to be stored in field-name by a **CREATE** (if no value is given within the **CREATE** expression itself). Note that form is evaluated at **CREATE** time, not when the declaration is made.

- (2) (**CREATE** **form**)⁸⁷ (re)defines the manner in which **CREATE** of this record should be performed. This provides a way of specifying how **ACCESSFNS** should be created or overriding the usual definition of **CREATE**. If form contains the field-names of the declaration as variables, the forms given in the **CREATE** will be substituted in. For example,
 (RECORD C (A . D)) and
 (ACCESSFNS C ((A CAR RPLACA)(D CDR RPLACD))
 (CREATE(CONS A D)))
 are equivalent.⁸⁸

If the word **DATUM** appears in the create form, the *original* create definition is inserted. This effectively allows the user to "advise" the create.

- (3) (**INIT** **form**) specifies that form should be evaluated when the record is

⁸⁷ CREATE may also be given as recordname ← form, e.g. C ← (CONS A D).

⁸⁸ This facility allows the use of data-structures not specified by one of the built-in record types. For example, one possible representation of a data-structure is to store the fields in *parallel* arrays, especially if the number of instances required is known, and they do not need to be garbage collected. Thus, to implement a data structure called **LINK** with two fields **FROM** and **TO**, one would have two arrays **FROMARRAY** and **TOARRAY**. The representation of an "instance" of the record would be an integer which is used to index into the arrays. This can be accomplished with the declaration:

```
[ACCESSFNS LINK
  ((FROM (ELT FROMARRAY DATUM) (SETA FROMARRAY DATUM NEWVALUE))
   (TO (ELT TOARRAY DATUM) (SETA TOARRAY DATUM NEWVALUE)))
  (CREATE (PROG1 (SETQ LINKCNT (ADD1 LINKCNT))
            (SETA FROMARRAY LINKCNT FROM)
            (SETA TOARRAY LINKCNT TO)))
```

To CREATE a new **LINK**, a counter is incremented and the new elements stored (although the create form given the declaration should actually include a test for overflow).

declared.⁸⁹

(4) a subdeclaration

i.e., a record declaration of any of the above types. The record-name of a subdeclaration must be either the record-name of its immediately superior declaration or one of the superior's field-names. Instead of identifying the declaration as with top level declarations, the record-name of a subdeclaration identifies the parent field or record that is being described by the subdeclaration. Subdeclarations can be nested to an arbitrary depth.^{90 91}

(5) (TYPE? form)⁹²

(Re)defines the manner in which TYPE? expressions are to be translated. form may either be an expression in terms of DATUM or a function of one argument.

(6) (SUBRECORD name . defaults)

name must be a field that appears in the current declaration and the name of another record. This says that, for the purposes of create expressions, substitute the top-level declaration of name for the SUBRECORD form, adding on any defaults specified.

For example: given (RECORD B (E F G)), then (RECORD A (B C D) (SUBRECORD B)) would be treated like (RECORD A (B C D) (RECORD B (E F G))), for the purposes of translating CREATE expressions.

CREATE

Record operations can be applied to arbitrary structures, i.e., structures created directly by user programs can be manipulated in a data-independent manner using record declarations. However, to be completely data-independent, new data should be created using the same declarations that define its data paths. This can be done by means of an expression of the form (CREATE record-name . {assignments}). A CREATE expression translates into an appropriate Interlisp form using cons, list, puthash, array, etc., that creates the new datum with the

+ 89 form will also be dumped by the INITRECORDS file package command.

90 Note that, in a few cases, it makes sense for a given field to have more than one subdeclaration. For example, in (RECORD (A . B) (PROPRECORD B (FOO FIE FUM)) (HASHLINK B C)) B is elaborated by both a PROPRECORD and a HASHLINK. Similarly, (RECORD (A B) (RECORD A (C D)) (RECORD A (FOO FIE))) is also acceptable, and essentially "overlays" (FOO FIE) and (C D), i.e. X:FOO and X:C would be equivalent. In such cases, the *first* subdeclaration is the one used by CREATE.

+ 91 Giving (RECORD name1 name2) is a simple way of defining a *synonym* for the field name1.

+ 92 TYPE? may also be given as record name @ form, e.g. C @ LISTP.

various fields initialized to the appropriate values.⁹³ {assignments} is optional and may contain expressions of the following form:

field-name \leftarrow form	specifies initial value for <u>field-name</u> . ⁹⁴
USING form	specifies that for all fields not given a value by (1), the value of the corresponding field in <u>form</u> is to be used.
COPYING form	like USING except the corresponding values are copied (with <u>copyall</u>).
REUSING form	like USING, except that wherever possible, the corresponding <u>structure</u> in <u>form</u> is used. ⁹⁵
SMASHING form	a new instance of the record is not created at all; rather, the value of <u>form</u> is used and smashed.

If the value of a field is neither explicitly specified, nor implicitly specified via USING, REUSING or COPYING, the default value in the declaration is used, if any, otherwise NIL.⁹⁶ For example, following (RECORD A (B C D) D \leftarrow 3),
 (CREATE A B \leftarrow T USING X) translates as (LIST T (CADR X) (CADDR X)),
 (CREATE A B \leftarrow T COPYING X)) as [LIST T(COPYALL (CADR X)) (COPYALL (CADDR X)).
 (CREATE A B \leftarrow T REUSING X) as (CONS T (CDR X)), and
 (CREATE A B \leftarrow T) as (LIST T NIL 3).

TYPE?

The record package allows the user to test if a given datum "looks like" an instance of a record.

⁹³ CREATE is not defined as a function. Instead, DWIM calls the appropriate function in the record package giving it the entire CREATE expression as an argument. The translation of the CREATE expression, i.e., the Interlisp form which is evaluated to construct the datum, is then stored elsewhere, as with iterative statements and pattern matches.

⁹⁴ The record package goes to great pains to insure that the order of evaluation in the translation is the same as that given in the original create expression if the side effects of one expression might affect the evaluation of another. For example, given the declaration (RECORD CONS (CAR . CDR)), the expression (CREATE CONS CDR \leftarrow X CAR \leftarrow Y) will translate to (CONS Y X), but (CREATE CONS CDR \leftarrow (FOO) CAR \leftarrow (FIE)) will translate to ((LAMBDA (\$\$1) (CONS (PROGN (SETQ \$\$1 (FOO)) (FIE)) \$\$1))) because, for example, FOO might set some variables used by FIE.

⁹⁵ Note that (CREATE record REUSING form ...) does not itself do any destructive operations on the value of form. The distinction between USING and REUSING is that (CREATE record REUSING form ...) will incorporate as much as possible of the old data structure into the new one being created, while (CREATE record USING form ...) will create a completely new data structure, with only the contents of the fields re-used. For example, CREATE REUSING a PROPRECORD just conses the new property names and values onto the list, while CREATE USING copies the top level of the list. Another example of this distinction occurs when a field is elaborated by a subdeclaration: USING will create a new instance of the sub-record, while REUSING will use the old contents of the field (unless some field of the subdeclaration is assigned in the CREATE expression.)

⁹⁶ For BETWEEN fields in DATATYPE records, n1 is used; for other non-pointer fields zero is used.

This can be done via an expression of the form (`TYPE? record-name form`). `TYPE?` is mainly intended for declarations involving record-type DATATYPE or TYPERECORD. For DATATYPES, the `TYPE?` check is exact; i.e. the `TYPE?` expression will return non-NIL only if the value of form is an instance of the record named by record-name. For TYPERECORDs, the `TYPE?` expression will check that the value of form is a list beginning with record-name. For ARRAYRECORDS, it checks that the value is an array of the correct size. For PROPRECORDs and ASSOCRECORDs, a `TYPE?` expression will make sure that the value of form is a property/association list with property names among the field-names of the declaration.

Attempting to execute a `TYPE?` expression for a record of type ACCESSFNS, HASHLINK or RECORD will cause an error, `TYPE? NOT IMPLEMENTED FOR THIS RECORD`. The user can (re)define the interpretation of `TYPE?` expressions for a particular declaration by inclusion of an expression of the form (`TYPE? com`) in the declaration, as described on page 23.32.

DATA-PATHS

The user may also elaborate a field by declaring that field name in a *separate* record declaration (as opposed to an embedded subdeclaration). For example, the declarations

(`RECORD MSG (ID (FROM TO) . TEXT)`) and (`RECORD TEXT (HEADER . TXT)`) also subdivide TEXT into two subfields. The user may then specify `X:MSG.HEADER` to achieve the interpretation "X is a MSG, retrieve its HEADER".⁹⁷ The notation `X:NAME1.NAME2` is interpreted to mean "find a path from the record with name NAME1 to the field named NAME2", as opposed to `X:NAME1:NAME2`, where NAME1 and NAME2 are interpreted independently.

The central point of separate declarations is that the (sub)record is *not* tied to another record (as with embedded declarations), and therefore can be used in many different contexts. For example, one might additionally have a declaration

(`RECORD REPLY (TEXT TO . RESPONSE)`).

In this case, one could specify `X:REPLY.HEADER` to mean that X is a REPLY, and to retrieve (CAAR X). In general, the user may specify as a data-path a chain of record/field names, e.g., `X:MSG.TEXT.HEADER.SUBHEAD...` etc.,⁹⁸ where there is some path from each record to the next in the chain. Only as much of the path as is necessary to disambiguate it needs to be specified. For example, with the above declarations of MSG, TEXT and REPLY, the path `X:MSG.HEADER` is unambiguous (it must go through TEXT); however, `X:TEXT` is not,⁹⁹ as this could mean that X is either a MSG or a REPLY.¹⁰⁰ The record package interprets a data path by performing a tree search among all current declarations for a path from each name to the next, considering first local declarations (if any) and then global ones.

⁹⁷ `X:HEADER` by itself is interpreted to mean that X is an instance of TEXT, and translates as (`CAR X`).

⁹⁸ Translation of expressions involving data paths are handled by replacing the expression by a `fetch` or `replace` statement with the fields given in a list; e.g. `X:FOO.FIE.A` and `X:FOO.FIE.A+Y` are replaced by the expression (`fetch (FOO FIE A) X`) and (`replace (FOO FIE A) of X with Y`) respectively, with the translation stored elsewhere. Input of this form is also acceptable.

⁹⁹ Note that if a field has an *identical* interpretation in two declarations, e.g. if the field TEXT occurred in the same location within the declarations of MSG and REPLY, it would *not* be considered ambiguous.

¹⁰⁰ In this case, the message AMBIGUOUS RECORD FIELD is printed and an error is generated. If a data-path rather than a single field is ambiguous, (e.g. if there were yet another declaration (`RECORD TO (NAME . HEADER)`) and the user specified `X:MSG.HEADER`), the error AMBIGUOUS DATA PATH is generated.

CHANGING RECORD DECLARATIONS

The user may edit (or delete) global record declarations with the function

editrec[editrecx] nlambda, nospread function, similar to editf or editv. editrec calls the editor on a copy of all declarations in which car[editrecx] is the record-name or a field name. On exit, it redeclares those that have changed and undeclares any that have been deleted. If car[editrecx] is NIL, *all* declarations are edited.

Records can also be declared local to a particular function by using a CLISP declaration, as described on page 23.25; all local record declarations override global ones. In addition, a local declaration of the form (RECORDS A B C) is equivalent to having copies of the global declarations A, B, and C in the local declaration.

For both global and local records, the translation is computed using all CLISP declarations in effect as described on page 23.24, e.g., if the declaration UNDOABLE is in effect, /RPLACA, /RPLACD, /PUTHASH, etc. will be used.

When the user redeclares a global record, the translations of all expressions involving that record or any of its fields are automatically deleted,¹⁰¹ and thus will be recomputed using the new information. If the user changes a *local* record declaration, or changes some other CLISP declaration, e.g., STANDARD to FAST, and wishes the new information to affect record expressions already translated, he must make sure the corresponding translations are removed, usually either by CLISPIFYING or applying the !DW edit macro.

OTHER FUNCTIONS

reclook[recordname;-;-;-] returns the entire declaration for the record named recordname.¹⁰² +
NIL if no record declaration with name recordname.¹⁰² +

fieldlook[fieldname] returns the list of declarations in which fieldname is the name of a field. +
+
+

recordfieldnames[recordname] returns the list of fields declared in record recordname, recordname may either be a name or an entire declaration. +
+
+

¹⁰¹ from clisparray. If the user is not using this method for storing translations, i.e., is instead using the CLISP% method (page 23.23), those expressions already translated will remain as they are. (There is no practical way to locate them.)

¹⁰² Note that the record package maintains internal state about current record declarations; performing destructive operations (e.g. nconc) on the value of reclook may leave the record package in an inconsistent state. To change a record declaration, use editrec.

- + recordaccess[field;value;type;newvalue;dec]

type is one of (FETCH FFETCH REPLACE FREPLACE /REPLACE) or their lowercase equivalents. type=NIL means FETCH. If type corresponds to a fetch operation, i.e. is FETCH, or FFETCH, recordaccess performs (type field OF value). If type corresponds to a replace, recordaccess performs (type field OF value WITH newvalue). dec is an optional declaration; if given, field is interpreted as a field name of that declaration.¹⁰³

23.12 CHANGETRAN¹⁰⁴

- + A very common programming construction consists of assigning a new value to some datum that is a function of the current value of that datum. Some examples of such read-modify-write sequences include:

- + (SETQ X (IPLUS X 1)) Incrementing a counter
- + (SETQ X (CONS Y X)) Pushing an item on the front of a list
- + (PROG1 (CAR X) (SETQ X (CDR X))) Popping an item off a list

- + It is easier to express such computations when the datum in question is a simple variable as above than when it is an element of some larger data structure, for instance, the car of some list:

- + (RPLACA X (IPLUS (CAR X) 1))
- + (RPLACA X (CONS Y (CAR X)))
- + (PROG1 (CAAR X) (RPLACA X (CDAR X)))
- + (SETA A N (IPLUS (ELT A N) 1)))

- + The difficulty in expressing (and reading) modification idioms is in part due to the well-known asymmetry of smashing versus accessing operations on structures: rplaca is used to smash what car would return, seta corresponds to elt, and so on. The CLISP operator \leftarrow and the record package : operator combine to provide a more uniform mode of expression:

- + X:1 \leftarrow (IPLUS X:1 1)
- + X:1 \leftarrow (CONS Y X:1)
- + (PROG1 (CAR X:1) (X:1 \leftarrow (CDR X:1)))

- + The \leftarrow operator also helps in the seta-elt example, because elt and seta are defined to be setfn-accessfn pairs:

- + (ELT A N) \leftarrow (IPLUS (ELT A N) 1)

- + CLISP allows the same expression to be used for both the accessing and the smashing components of these statements, but it is still not obvious that they all involve a simple notion of structure modification. The user must read these statements very carefully to notice that the X:1 and (ELT A N) expressions appear twice (e.g., the second expression might have been (ELT A M)),

¹⁰³ Note that recordaccess is relatively inefficient, although it is better than constructing the equivalent form and performing an EVAL.

¹⁰⁴ Changetran was designed by R. M. Kaplan.

and he must type (and edit) the same expressions twice, which can be cumbersome if the expressions are large.

The Changetran facility is designed to provide a more satisfactory notation in which to express certain common (but user-extensible) structure modification operations. Changetran defines a set of CLISP words that encode the kind of modification that is to take place, e.g. pushing on a list, adding to a number, etc. More important, the expression that indicates the datum whose value is to be modified needs to be stated only once. Thus, the "change word" ADD is used to increase the value of a datum by the sum of a set of numbers. Its arguments are an expression denoting the datum, and a set of items to be added to its current value. For example, (ADD X:3 (FOO)) is equivalent to X:3+X:3+(FOO).

The datum expression in this and all other changewords must be an expression that can appear to the left of the CLISP \leftarrow operator, e.g. a variable, a : expression, an accessfn form, or any of the corresponding dwimified expressions **fetch**, CAR, LAST, etc., e.g. (ADD (ELT A N) 1). For ADD and for all other changewords, the lower-case version may also be specified. Except for POP (see below), the value of all built-in changeword forms is defined to be the new value of the datum. Finally, if the datum expression is a complicated form involving subsidiary function calls (e.g. (ELT (FOO X) (FIE Y))), changetran goes to some lengths to make sure that those subsidiary functions are evaluated only once (it binds local variables to save the results), even though they logically appear in both the setting and accessing parts of the translation. Thus, in thinking about both efficiency and possible side effects, the user can rely on the fact that the forms will be evaluated only as often as they appear in the expression.

CHANGE WORDS RECOGNIZED BY CHANGETRAN

The following is a list of those change words recognized by changetran:

(ADD datum item₁ item₂ ...) Adds the specified items to the current value of the datum, stores the result back in the datum location.¹⁰⁵

(PUSH datum item₁ item₂ ...) Conses the items onto the front of the current value of the datum, and stores the result back in the datum location. Equivalent to (datum \leftarrow < item₁ item₂ ... ! datum >). For example (PUSH X A B) would translate as [SETQ X (CONS A (CONS B X)].

(PUSHNEW datum item) Like PUSH (with only one item) except that the item is not added if it is already fmemb of the datum's value.¹⁰⁶

(PUSHLIST datum item₁ item₂ ...) Similar to PUSH, except that the items are appended in front of the current value of the datum. Equivalent to (datum \leftarrow < ! item₁ ! item₂ ... ! datum >).

¹⁰⁵ The translation will use IPLUS, PLUS, or FPLUS according to the CLISP declarations in effect.

¹⁰⁶ Note that, whereas [CAR (PUSH X 'FOO] will always be FOO, [CAR (PUSHNEW X 'FOO] might be something else if FOO already existed in the middle of the list.

- + (POP datum) Returns car of the current value of the datum after storing its cdr into the datum. Equivalent to [PROG1 (CAR datum) (datum \leftarrow (CDR datum)]. The current value is computed only once even though it is referenced twice. Note that this is the only built-in changeword for which the value of the form is not the new value of the datum.
- + (SWAP form1 form2) sets form1 to form2 and vice versa, i.e. is equivalent to (form1 \leftarrow (PROG1 form2 form2 \leftarrow form1)), except that care is taken to make sure that expressions are not reevaluated unnecessarily.
- + (CHANGE datum form) This is the most flexible of all change words, since it enables the user to provide an arbitrary form describing what the new value should be, but it still highlights the fact that structure modification is to occur, and still enables the datum expression to appear only once. (CHANGE datum form) is equivalent to (datum \leftarrow form'), where form' is constructed from the form in the CHANGE statement by substituting the datum expression for every occurrence of the atom DATUM. For example, (CHANGE X:FOO (DATUM*5)) is equivalent to (X:FOO \leftarrow (ITIMES X:FOO 5)). CHANGE is useful for expressing modifications that are not built-in and are not sufficiently common to justify defining a user-changeword. As for other changeword expressions, the user need not repeat the datum-expression and need not worry about multiple evaluation of the accessing form.

23.13 CLISPIFY

Clispyf converts Interlisp expressions to CLISP. Note that the expression given to clispyf need not have originally been input as CLISP, i.e., clispyf can be used on functions that were written before CLISP was even implemented. Clispyf is cognizant of declaration rules as well as all of the precedence rules.¹⁰⁷ For example, clispyf will convert (IPLUS A (ITIMES B C)) into A+B*C, but (ITIMES A (IPLUS B C)) into A*(B+C).¹⁰⁸ Clispyf converts calls to the six basic mapping functions, MAP, MAPC, MAPCAR, MAPLIST, MAPCONC, and MAPCON, into equivalent iterative statements. It also converts certain easily recognizable internal PROG loops to the corresponding i.s. For example,

... label (COND (pred ... forms ... (GO label))) ...
 becomes
 ... label (WHILE pred DO ... forms ...) ... 109

¹⁰⁷ clispyf is table driven exactly the same as CLISP, so that if the user changes any precedence, or defines new operators, clispyf "automatically" knows about it.

¹⁰⁸ clispyf also knows how to handle expressions consisting of a mixture of Interlisp and CLISP, e.g., (IPLUS A B*C) is converted to A+B*C, but (ITIMES A B+C) to (A*(B+C)). clispyf handles such cases by first dwimifying the expression.

¹⁰⁹ clispyf can convert all iterative statements input in CLISP back to CLISP, regardless of how complicated the translation was, because the original CLISP is saved.

Clispify is not destructive to the original Interlisp expression, i.e., clispify produces a new expression without changing the original.¹¹⁰ Clispify will not convert expressions appearing as arguments to NLAMBDA functions.¹¹¹

The value of various global parameters affect the operation of clispify:

cl:flg

The user can disable the : transformation by setting the variable cl:flg to NIL. This will prevent clispify from constructing any expression employing a : infix operator, e.g., (CADR X) will not be transformed to X:2. When cl:flg is T, clispify will convert to : notation only when the argument is atomic or a simple list (a function name and one atomic argument). If cl:flg is ALL, clispify will convert to : expressions whenever possible. The initial value of cl:flg is T.

cremparsflg

Clispify will remove parentheses in certain cases from simple forms, where "simple" means a function name and one or two atomic arguments. For example, (COND ((ATOM X) --)) will CLISPIFY to (IF ATOM X THEN --). However, if cremparsflg is set to NIL, clispify will produce (IF (ATOM X) THEN --). Note that regardless of the setting of this flag, the expression can be input in either form. The initial value of cremparsflg is T.

clispifypackflg

clispifypackflg affects the treatment of infix operators with atomic operands. If clispifypackflg is T, clispify will pack these into single atoms, e.g., (IPLUS A (ITIMES B C)) becomes A+B*C. If clispifypackflg is NIL, no packing is done, e.g., the above becomes A+_B_*_C. The initial value of clispifypackflg is T.

clispifyenglshflg

If T, causes clispify to convert LISP forms to english phrases when possible, e.g., (MEMBER X Y) -> X IS A MEMBER OF Y.. See page 23.21.

funnyatomlst

Suppose the user has variables named A, B, and A*B. If clispify were to convert (ITIMES A B) to A*B, A*B would not translate back correctly to (ITIMES A B), since it would be the name of a variable, and therefore would not cause an error. The user can prevent this from happening by adding A*B to the list funnyatomlst. Then, (ITIMES A B) would clispify to A_*_B.

Note that A*B's appearance on funnyatomlst would *not* enable DWIM/CLISP to decode A*B+C as

¹¹⁰ The new expression may however contain some "pieces" of the original, since clispify attempts to minimize the number of CONSES by not copying structure whenever possible.

¹¹¹ Except for those functions whose INFO property is or contains the atom EVAL, such as nlsetq, resetlst, etc. clispify also contains built in information enabling it to process special forms such as prog, selectq, etc. If the INFO property is or contains the atom LABELS, clispify will never create an atom (by packing) at the top level of the expression. For example, prog is handled in this fashion.

(IPLUS A*B C); funnyatomlist is used only by clispify. Thus, if an identifier contains a CLISP character, it should always be separated (with spaces) from other operators. For example, if X* is a variable, the user should write (SETQ X* form) in CLISP as X* ← form, not X*← form. However, in general, it is best to avoid use of identifiers containing CLISP character operators as much as possible.

clispifyprettyflg

If T, causes prettyprint to clispify all expressions before printing them (but not to redefine any functions). clispifyprettyflg is temporarily reset to T, using resetvar, when makefile is called with the option CLISPIFY, or when the file in question has property FILETYPE with value CLISP on its property list. clispifyprettyflg is initially NIL.

In addition to the above controls, disabling a CLISP operator (see cldisable, page 23.59) will also disable the corresponding CLISPIFY transformation. Thus, if ← is "turned off", A+B will not transform to (SETQ A B), nor vice versa.

- + clispifyuserfn
- + If T, causes clispifyuserfn, a function of one argument, to be called on the form (list) to be clispified, when form is not otherwise recognized by clispify. If a non-NIL value is returned, this value is treated as the clispified version of the form.

23.14 DWIMIFY

Dwimify is effectively a preprocessor for CLISP. Dwimify operates by scanning an expression as though it were being interpreted, and for each form that would generate an error, calling DWIM to "fix" it.¹¹² Thus the user will see the same messages, and be asked for approval in the same situations, as he would if the expression were actually run. If DWIM is unable to make a correction, no message is printed, the form is left as it was, and the analysis proceeds.

Dwimify knows exactly how the interpreter works. It knows the syntax of progs, selectqs, lambda expressions, sets, et al. It knows that the argument of nlambdas are not evaluated.¹¹³ It also knows how variables are bound.¹¹⁴ In the course of its analysis of a particular expression, dwimify builds a list of the bound variables from the LAMBDA expressions and PROGS that it encounters. It uses this list for spelling corrections. Dwimify also knows not to try to "correct" variables that are on this list since they would be bound if the expression were actually being run. However, note that dwimify cannot, a priori, know about variables that are used freely but would be bound in a higher function if the expression were evaluated in its normal context. Therefore, dwimify will try

¹¹² Thus dwimify performs *all* DWIM transformations, not just CLISP transformations, i.e., it does spelling correction, fixes 8-9 errors, handles F/L, etc.

¹¹³ The user can inform dwimify that an NLAMBDA function *does* evaluate its arguments (presumably by direct calls to eval), by including on its property list the property INFO with value EVAL or a list which contains the atom EVAL.

- + ¹¹⁴ The user can inform dwimify that a particular function or construct binds variables by including the atom BINDS on the INFO property for car of the form. In this case, dwimify assumes that cdr of the form is the variable list, i.e. a list of atoms, or lists of the form (var value). lambda, nlambda, prog, and resetvars are handled in this fashion.

to "correct" these variables.¹¹⁵ Similarly, Dwimify will attempt to correct forms for which car is undefined, even when the form is not in error from the user's standpoint, but the corresponding function has simply not yet been defined.

Dwimify will also inform the user when it encounters an expression with too *many* arguments,¹¹⁶ because such an occurrence, although does not cause an error in the Interlisp interpreter, nevertheless is frequently symptomatic of a parenthesis error, e.g. the user wrote (CONS (QUOTE FOO X)) instead of (CONS (QUOTE FOO) X). Dwimify will print

POSSIBLE PARENTHESIS ERROR IN
(QUOTE FOO X)
TOO MANY ARGUMENTS (MORE THAN 1).

Dwimify will also check to see if a prog label contains a clisp character,¹¹⁷ and if so, will alert the user by printing the message SUSPICIOUS PROG LABEL, followed by the label. The prog label will *not* be treated as clisp.

Note that in most cases, an attempt to transform a form that is already as the user intended will have no effect (because there will be nothing to which that form could reasonably be transformed). However, in order to avoid needless calls to DWIM or to avoid possible confusion, the user can inform Dwimify *not* to attempt corrections or transformations on certain functions or variables by adding them to the list nofixfnslst or nofixvarslst respectively.^{118 119}

Dwimify and Dwimifyfns (used to Dwimify several functions) maintain two internal lists of those functions and variables for which corrections were unsuccessfully attempted. These lists are initialized to nofixfnslst and nofixvarslst. Once an attempt is made to fix a particular function or variable, and the attempt fails, the function or variable is added to the corresponding list, so that on subsequent occurrences (within this call to Dwimify or Dwimifyfns), no attempt at correction is made. For example, if FOO calls FIE several times, and FIE is undefined at the time FOO is dwimified, Dwimify will not bother with FIE after the first occurrence. In other words, once Dwimify "notices" a function or variable, it no longer attempts to correct it.¹²⁰ Moreover, once Dwimify "notices" such functions or variables, it subsequently treats them the same as though they were actually defined or set.

¹¹⁵ Dwimify rebinds fixspelldefault to N, so that if the user is not at the terminal when dwimifying (or compiling), spelling corrections will not be performed.

¹¹⁶ unless dwimcheck#argsflg=NIL (initially T).

¹¹⁷ unless dwimcheckproglabelsflg=NIL (initially T), or the label is a member of nofixvarslst.

¹¹⁸ Note that the user could achieve the same effect by simply setting the corresponding variables, and giving the functions dummy definitions.

¹¹⁹ Dwimify will never attempt corrections on global variables, i.e., variables that are a member of the list globalvars, or have the property GLOBALVAR with value T, on their property list. Similarly, Dwimify will not attempt to correct variables declared to be LOCALFREEVARS or SPECVARS in block declarations or via DECLARE expressions in the function body. The user can also declare variables that are simply used freely in a function by using the USEDFREE declaration.

¹²⁰ Dwimify and Dwimifyfns also "notice" free variables that are set in the expression being processed.

Note that these internal lists are local to each call to dwimify and dwimifyfns, so that if a function containing FOO0, a misspelled call to FOO, is dwimified before FOO is defined or mentioned, if the function is dwimified again after FOO has been defined, the correction will be made.

Note that the user can undo selected transformations performed by dwimify, as described in Section 22.

COMPIILING CLISP

Since the compiler does not know about CLISP, in order to compile functions containing CLISP constructs, the definitions must first be dwimified. The user can automate this process in several ways:

- 1) If the variable dwimifycompflg is T, the compiler will always dwimify expressions before compiling them. dwimifycompflg is initially NIL.
- 2) If a file has the property FILETYPE with value CLISP on its property list, tcompl, bcompl, recompile, and brecompile will operate as though dwimifycompflg is T and dwimify all expressions before compiling.
- 3) If the function definition has a CLISP declaration (see page 23.24), including a null declaration, i.e., just (CLISP:), the definition will be automatically dwimified before compiling.

Note: compileuserfn (Section 18) is defined to call dwimify on iterative statements, IF-THEN statements, and fetch, replace, and match expressions, i.e., any CLISP construct which can be recognized by its car of form. Thus, if the only CLISP constructs in a function appear inside of iterative statements, IF statements, etc., the function does not have to be dwimified before compiling.

- + If dwimify is ever unsuccessful in processing a CLISP expression, it will print the error message + UNABLE TO DWIMIFY followed by the expression, and go into a break.¹²¹ The user can then + either:
 - + (1) type OK to the break, which will cause the compiler to try again, e.g. the user could define + some missing records while in the break, and then continue; or
 - + (2) type ↑, which will cause the compiler to simply compile the expression as is, i.e. as though clisp + had not been enabled in the first place; or
 - + (3) return an expression to be compiled in its place by using the RETURN break command.

Note: tcompl, bcompl, recompile, and brecompile all scan the entire file before doing any

¹²¹ unless dwimessgag=T. In this case, the expression is just compiled as is, i.e. as though clisp had not been enabled.

compiling, and take note of the names of all functions that are defined in the file as well as the names of all variables that are set by adding them to nofixfnslst and nofixvarslist, respectively. Thus, if a function is not currently defined, but is defined in the file being compiled, when dwimify is called before compiling, it will not attempt to interpret the function name as CLISP when it appears as car of a form. Dwimify also takes into account variables that have been declared to be GLOBALVARS, LOCALVARS, or SPECVARS, either via block declarations or DECLARE expressions in the function being compiled, and does not attempt spelling correction on these variables. The declaration USEDFREE is also available to declare variables simply used freely in a function. These variables will also be left alone by dwimify. Finally, nospellflg (see page 23.57) is reset to T when compiling functions from a file (as opposed to from their in-core definition) so as to suppress spelling correction.

23.15 THE PRINTOUT PACKAGE

Interlisp provides many facilities for controlling the format of printed output. By executing various sequences of prin1, prin2, tab, terpri, spaces, printnum, and printdef, almost any effect can be achieved. The printout package implements a compact language for specifying complicated sequences of these elementary printing functions. It makes fancy output formats easy to design and simple to program.

OVERVIEW AND EXAMPLES

PRINTOUT is a CLISP word (like for and if) for interpreting a special printing language in which the user can describe the kinds of fancy printing he wants to achieve. The description is translated by dwimify machinery to the appropriate sequence of prin1, tab, etc. before it is evaluated or compiled. Printout printing descriptions have the following general form:

(printout file printcom₁ printcom₂ ... printcom_n)

The file parameter is evaluated to obtain the name of the file to which the output from this specification is directed, and each printcom in the sequence is a command that indicates a printing event to be executed. The various ways of realizing a printcom are defined below. The following examples give a general flavor of how printout is used.

Example 1: Suppose the user wanted to print out on his terminal the values of three variables, x, y, and z, separated by spaces and followed by a carriage return. This could be done by:

```
(PRIN1 X T)
(SPACES 1 T)
(PRIN1 Y T)
(SPACES 1 T)
(PRIN1 Z T)
(TERPRI T)
```

or by the more concise printout form:

```
(PRINTOUT T X , Y , Z T)
```

Here the first T specifies output to the terminal, the commas cause single spaces to be printed, and the final T specifies a terpri. The variable names are not recognized as special printout commands, so they are printed using prin1 by default.

- + Example 2. Suppose the values of x and y are to be pretty-printed lined up at position 10, preceded by identifying strings. If the output is to go to the primary output file, the user could write either:

```
+      (PRIN1 "X =")
+      (PRINTDEF X 10 T)
+      (TERPRI )
+      (PRIN1 "Y =")
+      (PRINTDEF Y 10 T)
+      (TERPRI)
```

- + or the equivalent:

```
+      (printout NIL "X =" 10 .PPV X T "Y =" 10 .PPV Y T)
```

- + Since strings are not recognized as special commands, "X =" is also printed with prin1 by default.
- + The positive integer means tab to position 10, where the .PPV command causes the value of x to be prettyprinted as a variable. By convention, special atoms used as printout commands are prefixed with a period. The T causes a carriage return, so the y information is printed on the next line.

- + Example 3. As a final example, suppose that the value of x is an integer and the value of y is a floating-point number. x is to be printed rightFlushed in a field of width 5 beginning at position 15, and y is to be printed in a field of width 10 also starting at position 15 with 2 places to the right of the decimal point. Furthermore, suppose that the variable names are to appear in the font named BOLDFONT and the values in font SMLLFONT. The program in ordinary Lisp that would accomplish these effects is too complicated to include here. With printout, one could write:

```
+      (PRINTOUT NIL .FONT BOLDFONT "X =" 15 .FONT SMLLFONT .I5 X T
+                  .FONT BOLDFONT "Y =" 15 .FONT SMLLFONT
+                  .F10.2 Y T .FONT BOLDFONT)
```

- + The .FONT commands print on the file the font-changing instructions appropriate for a multi-font output device (see Section 14). The .I5 command sets up a FIX format for a call to the function printnum (see Section 14) to print X in the desired format. The .F10.2 specifies a FLOAT format for printnum.

+ THE PRINTOUT FORM

- + There are two parts to the expression that is the Lisp translation of a printout form. The first part sets up a context in which printing is to be done in the case that a non-NIL file parameter is specified. In this case, the previous primary output file is remembered and the specified file is made primary during the execution of any subsequent printing. When the printout form is completed, the earlier primary file is restored.¹²² The second part of a printout translation contains

+ 122 i.e. if a non-NIL file parameter is specified, the printout must effectively be embedded in a resetform (see Section 5).
+ For this reason, if the user is going to be performing several calls to printout specifying the same, non-NIL file, it
+ will be more efficient to embed them all in a single resetform which changes the primary output file, and then
+ specify file=NIL in the printout expressions themselves.

the Lisp expressions which actually carry out the list of printout commands. Notice that the printout commands are strung together, one after the other without punctuation, as cddr of the printout form. Some commands occupy a single position in this list, but many commands expect to find arguments following the command name in the list.

The printout commands fall into several logical groups: one set deals with horizontal and vertical spacing, another group provides controls for certain formatting capabilities (font changes and subscripting), while a third set is concerned with various ways of actually printing items. Finally, there is a command that permits escaping to a simple Lisp evaluation in the middle of a printout form.

HORIZONTAL SPACING COMMANDS

The horizontal spacing commands provide convenient ways of calling tab and spaces. In the following descriptions, n stands for a literal positive integer.

<u>n</u>	used for absolute spacing. It results in a <u>tab</u> to position <u>n</u> (literally, a (TAB n)). If the line is currently at position <u>n</u> or beyond, the file will be positioned at <u>n</u> on the next line.
.TAB pos	specifies <u>tab</u> to position (the value of) <u>pos</u> . This is one of several commands whose effect could be achieved by simply escaping to Lisp, and executing the corresponding form. It is provided as a separate command so that the <u>printout</u> form is more concise and is prettyprinted more compactly. Note that .TAB n and n, where <u>n</u> is an integer, are equivalent.
.TAB0 pos	like .TAB except might result in zero spaces. ¹²³
-n	Negative integers indicate relative (as opposed to absolute) spacing. Translates as (SPACES n).
, , , ,	Provide a short-hand way of specifying 1, 2 or 3 spaces, i.e., these commands are equivalent to -1, -2, and -3, respectively.
.SP distance	Translates as (SPACES distance). Note that .SP n and -n, where <u>n</u> is an integer, are equivalent.
.RESET	Resets the current line by causing a carriage-return to be printed without a line-feed. Useful for overprinting, or for regaining

¹²³ i.e. the call to tab specifies minspaces=0.

- + control of a line on which characters have been printed in a variable pitched font.
- +

+ VERTICAL SPACING COMMANDS

- + Vertical spacing is obtained by calling terpri or printing form-feeds. The relevant commands are:

- + **T** Translates as (TERPRI). This command is functionally equivalent to the integer command 0: they both move to position 0 (= column 1) of the next line. Note, to print the letter T, use the string "T".
- +
- +
- +

- + **.SKIP** lines Equivalent to a sequence of lines (TERPRI)'s. The **.SKIP** command allows for skipping large constant distances and for computing the distance to be skipped.
- +
- +

- + **.PAGE** Puts a form-feed (control-L) out on the file. Care is taken to make sure that Interlisp's view of the current line position is correctly updated.
- +
- +

+ SPECIAL FORMATTING CONTROLS

- + There are a small number of commands for invoking some of the formatting capabilities of multi-font output devices. The available commands are:

- + **.FONT** fontspec Puts out a control sequence that causes a change to font fontspec (the association between fontspec and a specific font must be defined in the user's font profile, as described in Section 14). fontspec may be a font-name variable or an expression that evaluates to the value of a font-name variable. fontspec may also be a positive integer n, which is taken as an abbreviated reference to the font named **FONTn** (e.g. 1 = > **FONT1**).
- +
- +
- +
- +
- +

- + **.SUP** Specifies superscripting. All subsequent characters are printed above the base of the current line. Note that this is absolute, not relative: a **.SUP** following a **.SUP** is a no-op.
- +
- +

- + **.SUB** Specifies subscripting. Subsequent printing is below the base of the current line. As with superscripting, the effect is absolute.
- +

- + **.BASE** Moves printing back to the base of the current line. Un-does a previous **.SUP** or **.SUB**; a no-op, if printing is currently at the base.
- +
- +

PRINTING SPECIFICATIONS

The value of any expression in a printout form that is not recognized as a command itself or as a command argument is printed using prinl by default. For example, title strings can be printed by simply including the string as a separate printout command, and the values of variables and forms can be printed in much the same way. Note that a literal integer, say 51, cannot be printed by including it as a command, since it would be interpreted as a TAB; the desired effect can be obtained by using instead the string specification "51", or the form (QUOTE 51).

For those instances when prinl is not appropriate, e.g. the user wants prin2 instead, or he wants list structures to be prettyprinted, the following commands are available:

.P2 thing Causes thing to be printed using prin2; translates as (PRIN2 thing). +

.PPV thing Causes thing to be prettyprinted at the current line position via printdef. The call to printdef specifies that thing is to be printed as a variable, not a function, so that nothing special is done for SELECTQ, PROG, etc. +

.PPF thing Prettyprints thing as part of a function definition. SELECTQ, PROG, etc. do receive special treatment. +

.PPVTL thing Causes thing to be prettyprinted as a tail, that is, without the initial and final parentheses if it is a list. Useful for prettyprinting sublists of a list whose other elements are formatted with other commands. +

.PPFTL thing Prettyprints thing as a tail, and as part of a function definition. +

Paragraph format +

Interlisp's prettyprint routines are designed to display the structure of S-expressions, but they are not really suitable for formatting unstructured text. If a list is to be printed as a textual paragraph, its internal structure is less important than controlling its left and right margins, and the indentation of its first line. The .PARA and .PARA2 commands allow these parameters to be conveniently specified.

.PARA lmarg rmarg list Prints list in paragraph format, using prinl. Translates as (PRINTPARA lmarg rmarg list) (see page 23.50). Example: (PRINTOUT T 10 .PARA 5 -5 LST) will print the elements of lst as a paragraph with left margin at 5, right margin at (LINELENGTH)-5, and the first line indented to 10. +

.PARA2 lmarg rmarg list Print as paragraph using prin2 instead of prinl. Translates as (PRINTPARA lmarg rmarg list T). +

+ **Right-flushing**

- + Two commands are provided for printing simple expressions flushed-right against a specified line position, using the function flushright. They take into account the current position, the number of characters in the print-name of the expression, and the position the expression is to be flush against, and then print the appropriate number of spaces to achieve the desired effect. Note that this might entail going to a new line before printing. Note also that right-flushing of expressions longer than a line (e.g. a large list) makes little sense, and the appearance of the output is not guaranteed.

+ **.FR pos expr**

Flush-right using prinl. The value of pos determines the position that the right end of expr will line up at. As with the horizontal spacing commands, a negative position number means $|pos|$ columns from the current position, a positive number specifies the position absolutely. $pos=0$ specifies the right-margin, i.e. is interpreted as (LINELENGTH).

+ **.FR2 pos expr**

Flush-right using prin2 instead of prinl.

+ **Centering**

- + Commands for centering simple expressions between the current line position and another specified position are also available. As with right flushing, centering of large expressions is not guaranteed.

+ **.CENTER pos expr**

Centers expr between the current line position and the position specified by the value of pos. A positive pos is an absolute position number, a negative pos specifies a position relative to the current position, and 0 indicates the right-margin. Uses prinl for printing.

+ **.CENTER2 pos expr**

Centers using prin2 instead of prinl.

+ **Numbering**

- + The following commands provide FORTRAN-like formatting capabilities for integer and floating-point numbers. Each command specifies a printing format and a number to be printed.
- + The format specification translates into a format-list for the function printnum, which is described in Section 14.

+ **.Iformat number**

Specifies integer printing. Translates as a call to the function printnum with a FIX format-list constructed from format. The atomic format is broken apart at internal periods to form the format-list. For example, .I5.-8.T yields the format-list (FIX 5 -8 T), and the command sequence (... .I5.-8.T FOO ...) will translate as (PRINTNUM '(FIX 5 -8 T) FOO). It will cause the value of foo to be printed with radix -8 right_FLUSHED in a field of width 5, with 0's used for padding on the left.

Internal NIL's may be omitted, e.g. the commands .I5..T and .I5.NIL.T are equivalent.

.F format number Specifies floating-number printing. Like the .Iformat command, except translates with a FLOAT format-list.

.N format number The .I and .F commands specify calls to printnum with quoted format specifications. The .N command translates as (PRINTNUM format number), i.e., it permits the format to be the value of some expression. Note that, unlike the .I and .F commands, format is a separate element in the command list, not part of an atom beginning with .N.

ESCAPING TO LISP

There are many reasons for taking control away from printout in the middle of a long printing expression. Common situations involve temporary changes to system printing parameters (e.g. linelength), conditional printing (e.g. print foo only if fie is T), or lower-level iterative printing within a higher-level print specification.

form the escape command. form is an arbitrary Lisp expression that will be evaluated within the context established by the printout form, i.e. form can assume that the primary output file has been set up appropriately.

USER-DEFINED COMMANDS

The collection of commands and options outlined above is aimed at fulfilling all common printing needs. However, certain applications might have other, more specialized printing idioms, so a facility is provided whereby the user can define his own commands. To do this, he must make entries on the global list printoutmacros to define how his new commands are to be translated. printoutmacros is an a-list whose elements are of the form (command-name translation-function). Whenever command-name appears in command position in the sequence of printout commands (as opposed to an argument position of another command), translation-function is applied to the tail of the command-list. After inspecting as much of the tail as necessary, the function must return a list whose car is the translation of the user-defined command and its arguments, and whose cdr is the list of commands still remaining to be translated in the normal way.

For example, suppose the user wanted to define a command "?", which will cause its single argument to be printed with prinl only if it is not NIL. He must enter (? ?TRAN) on printoutmacros, and define the function ?tran as follows:

```
(LAMBDA (COMS)
  (CONS (SUBST COMS:2 'ARG
    '(PROG ((TEMP ARG))
      (AND TEMP (PRIN1 TEMP))))
    COMS::2))
```

- + Note that ?tran does not do any printing itself; it returns a form which, when evaluated in the proper context, will perform the desired action. This form should direct all printing to the primary output file.

+ SPECIAL PRINTING FUNCTIONS

- + The paragraph printing commands are translated into calls on the function printpara, which may also be called directly:

- + printpara[lmarg;rmarg;list;p2flg;parenflg;file]

+ Prints list on file in line-filled paragraph format with its first element beginning at the current line position and ending at or before rmarg, and with subsequent lines appearing between lmarg and rmarg. If p2flg is non-NIL, prints elements using prin2, otherwise prin1. If parenflg is non-NIL, then parentheses will be printed around the elements of list.

+ If lmarg is zero or positive, it is interpreted as an absolute column position. If it is negative, then the left margin will be at $|lmarg| + (POSITION)$. If lmarg=NIL, the left margin will be at $(POSITION)$, and the paragraph will appear in block format.

+ If rmarg is positive, it also is an absolute column position (which may be greater than the current (LINELENGTH)). Otherwise, it is interpreted as relative to (LINELENGTH), i.e., the right margin will be at $(LINELENGTH) + |rmarg|$. Example: (TAB 10) (PRINTPARA 5 -5 LST T) will prin2 the elements of lst in a paragraph with the first line beginning at column 10, subsequent lines beginning at column 5, and all lines ending at or before (LINELENGTH)-5.

+ The current (LINELENGTH) is unaffected by printpara, and upon completion, file will be positioned immediately after the last character of the last item of list. printpara is a no-op if list is not a list.

- + The right-flushing and centering commands translate as calls to the function flushright:

- + flushright[pos;x;min;p2flg;centerflg;file]

+ If centerflg=NIL, prints x rightFlushed against position pos on file; otherwise, centers x between the current line position and pos. Makes sure that it spaces over at least min spaces before printing by doing a terpri if necessary; min=NIL is equivalent to min=1. A positive pos indicates an absolute position, while a negative pos signifies the position which is $|pos|$ to the right of the current line position. pos=0 is interpreted as (LINELENGTH), the right margin.

23.16 CLISP OPERATION

CLISP is a part of the basic Interlisp system. Without any special preparations, the user can include CLISP constructs in programs, or type them in directly for evaluation (in eval or apply format), then, when the "error" occurs, and DWIM is called, it will destructively¹²⁴ transform the CLISP to the equivalent Interlisp expression and evaluate the Interlisp expression. User approval is not requested, and no message is printed.¹²⁵

However, if a CLISP construct contains an error, an appropriate diagnostic is generated, and the form is left unchanged. For example, if the user writes (LIST X+Y*), the error diagnostic MISSING OPERAND AT X+Y* IN (LIST X+Y*) would be generated. Similarly, if the user writes (LAST+EL X), CLISP knows that ((IPLUS LAST EL) X) is not a valid Interlisp expression, so the error diagnostic MISSING OPERATOR IN (LAST+EL X) is generated. (For example, the user might have meant to say (LAST+EL*X).) Note that if LAST+EL were the name of a defined function, CLISP would never see this form.

Since the bad CLISP transformation might not be CLISP at all, for example, it might be a misspelling of a user function or variable, DWIM holds all CLISP error messages until after trying other corrections. If one of these succeeds, the CLISP message is discarded. Otherwise, if all fail, the message is printed (but no change is made).¹²⁶ For example, suppose the user types (R/PLACA X Y). CLISP generates a diagnostic, since ((IQUOTIENT R PLACA) X Y) is obviously not right. However, since R/PLACA spelling corrects to /RPLACA, this diagnostic is never printed.

If a CLISP infix construct is well formed from a syntactic standpoint, but one or both of its operands are atomic and not bound,¹²⁷ it is possible that either the operand is misspelled, e.g., the user wrote X+YY for X+Y, or that a CLISP transformation operation was not intended at all, but that the entire expression is a misspelling. For example, if the user has a variable named LAST-EL, and writes (LIST LAST-ELL). Therefore, CLISP computes, but does not actually perform, the indicated infix transformation. DWIM then continues, and if it is able to make another correction, does so, and ignores the CLISP interpretation. For example, with LAST-ELL, the transformation LAST-ELL -> LAST-EL would be found.

If no other transformation is found, and DWIM is about to interpret a construct as CLISP for which one of the operands is not bound, DWIM will ask the user whether CLISP was intended, in this case by printing LAST-ELL TREAT AS CLISP ?¹²⁸

¹²⁴ CLISP transformations, like all DWIM corrections, are undoable.

¹²⁵ This entire discussion also applies to CLISP transformation initiated by calls to DWIM from dwimify.

¹²⁶ Except that CLISP error messages are not printed on type-in. For example, typing X+*Y will just produce a U.B.A. X+*Y message.

¹²⁷ For the purpose of dwimifying, "not bound" means no top level value, not on list of bound variables built up by dwimify during its analysis of the expression, and not on nofixvarslist, i.e., not previously seen.

¹²⁸ If more than one infix operator was involved in the CLISP construct, e.g., X+Y+Z, or the operation was an assignment to a variable already noticed, or treatasclispflg is T (initially NIL), the user will simply be informed of the correction, e.g., X+Y+Z TREATED AS CLISP. Otherwise, even if DWIM was enabled in TRUSTING mode, the user will be asked to approve the correction.

The same sort of procedure is followed with 8 and 9 errors. For example, suppose the user writes `FOO8*X` where `FOO8` is not bound. The CLISP transformation is noted, and DWIM proceeds. It next asks the user to approve `FOO8*X -> FOO (*X)`. (For example, this would make sense if the user has (or plans to define) a function named `*X`.) If he refuses, the user is asked whether `FOO8*X` is to be treated as CLISP. Similarly, if `FOO8` were the name of a variable, and the user writes `FOO08*X`, he will first be asked to approve `FOO08*X -> FOO0 (*XX)`¹²⁹ and if he refuses, then be offered the `FOO08 -> FOO8` correction.

CLISP also contains provision for correcting misspellings of infix operators (other than single characters), IF words, and i.s. operators. This is implemented in such a way that the user who does not misspell them is not penalized. For example, if the user writes `IF N=0 THEN 1 ELSE N*(FACT N-1)` CLISP does *not* operate by checking each word to see if it is a misspelling of IF, THEN, ELSE, or ELSEIF, since this would seriously degrade CLISP's performance on *all* IF statements. Instead, CLISP assumes that all of the IF words are spelled correctly, and transforms the expression to `(COND ((ZEROP N) 1 ELSE N*(FACT N-1)))`. Later, after DWIM cannot find any other interpretation for ELSE, and using the fact that this atom originally appeared in an IF statement, DWIM attempts spelling correction, using `(IF THEN ELSE ELSEIF)` for a spelling list. When this is successful, DWIM "fails" all the way back to the original IF statement, changes ELSE to ELSE, and starts over. Misspellings of AND, OR, LT, GT, etc. are handled similarly.

CLISP also contains many Do-What-I-Mean features besides spelling corrections. For example, the form `(LIST +X Y)` would generate a MISSING OPERATOR error. However, `(LIST -X Y)` makes sense, if the minus is unary, so DWIM offers this interpretation to the user. Another common error, especially for new users, is to write `(LIST X*FOO(Y))` or `(LIST X*FOO Y)`, where `FOO` is the name of a function, instead of `(LIST X*(FOO Y))`. Therefore, whenever an operand that is not bound is also the name of a function (or corrects to one), the above interpretations are offered.

23.17 CLISP INTERACTION WITH USER

Syntactically and semantically well formed CLISP transformations are always performed without informing the user. Other CLISP transformations described in the previous section, e.g., misspellings of operands, infix operators, parentheses errors, unary minus - binary minus errors, all follow the same protocol as other DWIM transformations (Section 17). That is, if DWIM has been enabled in TRUSTING mode, or the transformation is in an expression typed in by the user for immediate execution, user approval is not requested, but the user is informed.¹³⁰ However, if the transformation involves a user program, and DWIM was enabled in CAUTIOUS mode, the user will be asked to approve. If he says NO, the transformation is not performed. Thus, in the previous section, phrases such as "one of these (transformations) succeeds" and "the transformation `LAST-ELL -> LAST-EL` would be found" etc., all mean if the user is in CAUTIOUS mode and the error is in a program, the corresponding transformation will be performed only if the user approves (or defaults by not responding). If the user says NO, the procedure followed is the same as though the transformation had not been found. For example, if `A*B` appears in the function `FOO`, and `B` is not bound (and no other transformations are found) the user would be asked

¹²⁹ The 8-9 transformation is tried before spelling correction since it is empirically more likely that an unbound atom or undefined function containing an 8 or a 9 is a parenthesis error, rather than a spelling error.

¹³⁰ However, in certain situations, DWIM will ask for approval even if DWIM is enabled in TRUSTING mode. For example, the user will always be asked to approve a spelling correction that might also be interpreted as a CLISP transformation, as in `LAST-ELL -> LAST-EL`.

A*B [IN FOO] TREAT AS CLISP?

131

If the user approved, A*B would be transformed to (ITIMES A B), which would then cause a U.B.A. B error in the event that the program was being run (remember the entire discussion also applies to DWIMIFYing). If the user said NO, A*B would be left alone.¹³²

23.18 CLISP INTERNAL CONVENTIONS

Note: the reader can skip this section and proceed to "Function and Variables" (page 23.56), unless he wants to add new operators, or modify the action of existing ones (other than by making declarations).

CLISP is almost entirely table driven by property lists for the corresponding infix or prefix operators. Thus it is relatively easy to add new infix or prefix operators or change old ones, simply by adding or changing selected property values.¹³³

CLISPTYPE

The property value of the property CLISPTYPE is the precedence number of the operator:¹³⁴ higher values have higher precedence, i.e., are tighter. Note that the actual value is unimportant, only the value relative to other operators. For example, CLISPTYPE for :, ↑, and * are 14, 6, and 4 respectively. Operators with the same precedence group left to right, e.g., / also has precedence 4, so A/B*C is (A/B)*C.

An operator can have a different left and right precedence by making the value of CLISPTYPE be a dotted pair of two numbers, e.g., CLISPTYPE of ← is (8 . -12). In this case, car is the left precedence, and cdr the right, i.e., car is used when comparing with operators on the left, and cdr with operators on the right. For example, A*B←C+D is parsed as A*(B←(C+D)) because the left precedence of ← is 8, which is higher than that of *, which is 4. The right precedence of ← is -12, which is lower than that of +, which is 2.

Cisp bracket operators are defined by having the value of the CLISPTYPE property be the atom BRACKET for both left and right brackets, e.g. for both < and >. See discussion of CLISPBRACKET below.

+
+
+
+

¹³¹ The waiting time on such interactions is three times as long as for simple corrections, i.e., 3*dwimwait.

¹³² If the value of clisphelpflg=NIL (initially T), the user will not be asked to approve any cisp transformation. Instead, in those situations where approval would be required, the effect is the same as though the user had been asked and said No.

+
+
+

¹³³ There is some built in information for handling minus, :, ', and ~, i.e., the user could not himself add such "special" operators, although he can disable or redefine them.

¹³⁴ Unless otherwise specified, the property is stored on the property list of the operator.

If the CLISPTYPE property for any operator is removed, the corresponding CLISP transformation is disabled, as well as the inverse CLISPIFY transformation.

UNARYOP

The value of property UNARYOP must be T for unary operators or brackets. The operand is always on the right, i.e., unary operators or brackets are always prefix operators.

BROADSCOPE

The value of property BROADSCOPE is T if the operator has lower precedence than Interlisp forms, e.g., LT, EQUAL, AND, etc. For example, (FOO X AND Y) parses as ((FOO X) AND Y). If the BROADSCOPE property were removed from the property list of AND, (FOO X AND Y) would parse as (FOO (X AND Y)).

LISPFN

The value of the property LISPFN is the name of the function to which the infix operator translates. For example, the value of LISPFN for \uparrow is EXPT, for ' QUOTE, etc. If the value of the property LISPFN is NIL, the infix operator itself is also the function e.g., AND, OR, EQUAL.

SETFN

If FOO has a SETFN property FIE, then (FOO --) \leftarrow X translates to (FIE -- X). For example, if the user makes ELT be an infix operator, e.g. #, by putting appropriate CLISPTYPE and LISPFN properties on the property list of # then he can also make # followed by \leftarrow translate to SETA, e.g., X#N \leftarrow Y to (SETA X N Y), by putting SETA on the property list of ELT under the property SETFN. Putting (ELT) (i.e., list[ELT])) on the property list of SETA under property SETFN will enable SETA forms to CLISPIFY back to ELT's.

CLISPINFIX

The value of this property is the CLISP infix to be used in CLISPIFYing. This property is stored on the property list of the corresponding Interlisp function, e.g., the value of property CLISPINFIX for EXPT is \uparrow , for QUOTE is ' etc.

- + CLISPWORD
- +
- +
- +
- +
- +
- +
- +

appears on property list of clisp operators which can appear as car of a form, such as FETCH, REPLACE, IF, iterative statement operators, etc. Value of property is of the form (keyword . name), where name is the lowercase version of the operator, and keyword is its type, e.g. FORWARD, IFWORD, RECORDWORD, etc. keyword can also be the name of a function; in which case when the atom appears as car of a form, the function is applied to the form and the result taken as the correct form.¹³⁵

+ 135 In this case, the function should either physically change the form, or call clisptran to store the translation.

Global declarations operate by changing the corresponding LISPFN and CLISPINFIX properties.

clispchars is a list of single character operators that can appear in the interior of an atom. Currently these are: +, -, *, /, ↑, ~, ', =, ←, :, <, and >.

clispchararray is a bit table of the characters on **clispchars** used for calls to strpos (see Section 10). **clispchararray** is initialized by performing (SETQ CLISPCHARARRAY (MAKEBITTABLE CLISPCHARS)).

clispinfixsplst is a list of infix operators used for spelling correction.

As an example, suppose the user wants to make | be an infix character operator meaning OR. He performs:

```
←(PUT (QUOTE |) (QUOTE CLISPTYPE))
  (GETP (QUOTE OR) (QUOTE CLISPTYPE)))
←PUT(| LISPFN OR)
←PUT(| BROADSCOPE T)
←PUT(OR CLISPINFIX |)
←SETQ(CLISPCHARS (CONS (QUOTE |) CLISPCHARS))
←SETQ(CLISPCHARARRAY (MAKEBITTABLE CLISPCHARS))
```

CLISPBRACKET Used for defining CLISP bracket operators. This property must be on the property list of both the left and right brackets, and also on the property list of any of the functions that expressions involving the brackets translate to (to enable clispifying). For the latter, the property value is simply the left bracket, e.g. <. For the brackets themselves, car of the property value is the left bracket, cadr the right bracket, and cddr a list in property list format. The following properties are recognized:

DWIMIFY function to be called to dwimify the construct: If the bracket is a unary operator, the function will be applied to the list consisting of the (dwimified) segment between the matching brackets.

If the bracket is not a unary operator, the function will be applied to the (dwimified) expression to the left of the bracket and the (dwimified) segment between the brackets. (See example below.)

- + CLISPIFY (optional) function to be called when clispifying.¹³⁶
- + SEPARATOR (optional) character to be split from any atoms but otherwise unprocessed, e.g. for < and >, the separator is !
- + For example, to define { } as a bracket which translates to elt in the case of a single expression, and multi-elt for more than one:

```

+   ↪PUT({ CLISP TYPE BRACKET)
+   ↪PUT() CLISP TYPE BRACKET)
+   ↪PUT({ CLISPBRACKET ({ }) SEPARATOR , DWIMIFY FOO]
+   ↪PUT() CLISPBRACKET ({ }) SEPARATOR , DWIMIFY FOO]
+   ↪DEFINEQ[(FOO (A LST)
+             (IF (MEMB ', LST)
+                 THEN <'MULTI-ELT A (FOR X IN LST COLLECT X WHEN X~='),>
+                 ELSE <'ELT A I LST>]
+   ↪(NCONC CLISPCHARS '({ }),)
+   ↪(SETQ CLISPCHARARRAY (MAKEBITTABLE CLISPCHARS))

```

- + Then, X:1{N-1} will dwimify to (ELT (CAR X) (SUB1 N))
+ and Z{N,M} will dwimify to (MULTI-ELT Z N M)
- + To enable clispifying,

```

+   ↪PUT(ELT CLISPBRACKET {})
+   ↪PUT(MULTI-ELT CLISPBRACKET {})

```

- + Then, (MULTI-ELT (CADR A) (IQUOTIENT (SUB1 N) 2)) I) will clispify to
+ A:2{(N-1)/2,I}.

23.19 CLISP FUNCTIONS AND VARIABLES

clispflg if set to NIL, disables all CLISP infix or prefix transformations (but does not affect IF/THEN/ELSE statements, or iterative statements).

If clispflg=TYPE-IN, CLISP transformations are performed only on expressions that are typed in for evaluation, i.e. not on user programs.

If clispflg=T, CLISP transformations are performed on all expressions.

+ 136 If the CLISPIFY property is not present, clispifying will consist of, for unary brackets, clispifying cdr of the form, inserting the separator if any between each element, and surrounding the result with the brackets. For brackets that are not unary operators, cadr of the form is clispified and placed to the left of the brackets, and cddr of the form treated as above.

The initial value for clispflg is T. clispifying anything will cause clispflg to be set to T.

clisparray	hash array used for storing translations. <u>clisparray</u> is checked by <u>faulteval</u> and <u>faultapply</u> on erroneous forms before calling DWIM, and by the compiler.
clisptran[x;tran]	gives <u>x</u> the translation <u>tran</u> . If <u>clisparray</u> is not NIL, uses hashing scheme, otherwise uses CLISP% scheme. See page 23.22-24. ¹³⁷
nofixfnslst	list of functions that <u>dwimify</u> will not try to correct. See page 23.41.
nofixvarslst	list of variables that <u>dwimify</u> will not try to correct. See page 23.41.
nospellflg	If <u>nospellflg</u> is T, <u>dwimify</u> will not perform any spelling corrections. The initial value of <u>nospellflg</u> is NIL. <u>nospellflg</u> is reset to T when compiling functions whose definitions are obtained from a file, as opposed to being in core.
clisphelpflg	if NIL, <u>dwimify</u> will not ask the user for approval of any cisp transformations. Instead, in those situations where approval would be required, the effect is the same as though the user had been asked and said No.
dwimify[x;quietflg;l]	dwimifies <u>x</u> , i.e., performs all corrections and transformations that would be performed if <u>x</u> were run, and prints the result unless <u>quietflg</u> =T. If <u>x</u> is an atom and <u>l</u> is NIL, <u>x</u> is treated as the name of a function, and its entire definition is dwimified. Otherwise, if <u>x</u> is a list or <u>l</u> is not NIL, <u>x</u> is the expression to be dwimified. If <u>l</u> is not NIL, it is the edit push-down list leading to <u>x</u> , and is used for determining context, i.e., what bound variables would be in effect when <u>x</u> was evaluated, whether <u>x</u> is a form or sequence of forms, e.g., a cond clause, etc. ¹³⁸

137 clisptran is called for all cisp translations, via a non-linked, external function call, i.e. it can be advised. +

138 If x is an iterative statement and l is NIL, dwimify will also print the translation, i.e., what is stored in the hash array. *

- * dwimifyfns[fns] nlambda, nospread. Dwimifies each function on fns. If fns is a list of only one element, car[fns] is evaluated. If its value is a list, this list is used, e.g. dwimifyfns[FOOFNS]. If car[fns] is atomic, and its value is *not* a list, and car[fns] is the name of a known file, dwimifyfns will operate on filefnslst[car[fns]], e.g. dwimifyfns[FOO.LSP] will dwimify every function in the file FOO.LSP.
- * Every 30 seconds, dwimifyfns prints the name of the function it is processing, a la prettyprint.
- * Value is a list of the functions dwimified.

- dwimifycompflg if T, dwimify is called before compiling an expression. See page 23.42.

- + dwimcheck#argsflg if T, causes dwimify to check for too many arguments in a form.

- + dwimcheckproglabelsflg if T, causes dwimify to check whether a prog label contains a clisp character.

- + dwimessgag if T (initially NIL), suppresses all dwimify error messages.

- clispdec[declst] puts into effect the declarations in declst. clispdec performs spelling corrections on words not recognized as declarations. clispdec is undoable.

- clispify[x;l] clispifies x. If x is an atom and l is NIL, x is treated as the name of a function, and its definition (or EXPR property) is clispified. After clispify has finished, x is redefined (using /PUTD) with its new CLISP definition. The value of clispify is x. If x is atomic and not the name of a function, spelling correction is attempted. If this fails, an error is generated.

If x is a list, or l is not NIL, x itself is the expression to be clispified. If l is not NIL, it is the edit push-down list leading to x and is used to determine context as with dwimify, as well as to obtain the local declarations, if any. The value of clispify is the clispified version of x.

See earlier section on CLISPIFY for more details.

clispifyfns[fn]	nlambda, nospread. Like <u>dwimifyfns</u> except calls <u>clispify</u> instead of <u>dwimify</u> . *
cldisable[op]	disables <u>op</u> , e.g., cldisable[-] makes - be just another character. ¹³⁹ <u>cldisable</u> can be used on all CLISP operators, e.g., infix operators, prefix operators, iterative statement operators, etc. <u>cldisable</u> is undoable.
clispiftranflg	affects handling of translations of IF THEN ELSE statements. If T, the translations are stored elsewhere, and the (modified) CLISP retained. If NIL, the corresponding COND expression, replaces the CLISP. <u>clispiftranflg</u> is initially NIL. See page 23.22.
clispretranflg	If T, informs <u>dwimify</u> to (re)translate all expressions which have remote translations, either in hash array or using CLISP%. Initially NIL.
cl:flg	affects <u>clispify</u> 's handling of forms beginning with <u>car</u> , <u>cdr</u> , ... <u>cddddr</u> , as well as pattern match and record expressions. See page 23.39.
cremparsflg	affects <u>clispify</u> 's removal of parentheses from "small" forms. See page 23.39.
clispifpackflg	if T, informs <u>clispify</u> to pack operator and atomic operands into single atoms; if NIL, no packing is done. See page 23.39.
clispifyenglshflg	if T, informs <u>clispify</u> to convert LISP expressions to english phrases when possible. See page 23.21.
clispifyprettyflg	if non-NIL, causes <u>prettyprint</u> to CLISPIFY selected function definitions before printing them according to the following interpretations of <u>clispifyprettyflg</u> : ¹⁴⁰ + + +

¹³⁹ Simply removing a character operator from clispchars will prevent it from being treated as a clisp operator when it appears as part of an atom, but it will continue to be an operator when it appears as a separate atom, e.g. (FOO + X) vs FOO+X.

¹⁴⁰ Another way to inform prettyprint to clispify functions is for the function to have a CLISP declaration containing the word CLISPIFY.

ALL	all functions
T, EXPRS	functions currently defined as exprs
CHANGES	functions marked as having been changed
a list	a member of that list

clispifyprettyflg is (temporarily) reset to T when makefile is called with the option CLISPIFY, and reset to CHANGES when the file being dumped has the property FILETYPE value CLISP. clispifyprettyflg is initially NIL.¹⁴¹

+ clispifyuserfn	if T, causes <u>clispifyuserfn</u> to be called on each form (list) not otherwise recognized by <u>clispify</u> . If a non-NIL value is returned, it is treated as the clispified form.
prettytranflg	If T, causes <u>prettyprint</u> to print translations instead of CLISP expressions. This is useful for creating a file for compilation, or for exporting to a LISP system that does not have CLISP. <u>prettytranflg</u> is (temporarily) reset to T when <u>makefile</u> is called with the option NOCLISP. If <u>prettytranflg</u> is CLISP%, both the CLISP and translations are printed in appropriate form. For more details, see page 23.23. <u>prettytranflg</u> is initially NIL.
PPT	is both a function and an edit macro for prettyprinting translations. It performs a PP after first resetting <u>prettytranflg</u> to T, thereby causing any translations to be printed instead of the corresponding CLISP.
CLISP:	edit macro that obtains the translation of the correct expression, if any, from <u>clisparray</u> , and calls <u>edite</u> on it.
funnyatomlst	list of identifiers containing CLISP operators. Used by <u>clispify</u> to avoid accidentally constructing a user identifier, e.g., (ITIMES A B) should not become A*B if A*B is the name of a PROG variable. See page 23.39.
CL	edit macro. Replaces current expression with CLISPIFYed current expression. Current expression can be an element or tail.

¹⁴¹ If clispifyprettyflg is non-NIL, and the only transformation performed by DWIM are well formed CLISP transformations, i.e., no spelling corrections, the function will *not* be marked as changed, since it would only have to be re-clispified and re-prettyprinted when the file was written out.

DW

edit macro. DWIMIFY's current expression, which can be an element (atom or list) or tail.

Both CL and DW can be called when the current expression is either an element or a tail and will work properly. Both consult the declarations in the function being edited, if any, and both are undoable.

lowercase[flg]

If flg=T, lowercase makes the necessary internal modifications so that clispify will use lower case versions of AND, OR, IF, THEN, ELSE, ELSEIF, and all i.s. operators. This produces more readable output. Note that the user can always type in *either* upper or lower case (or a combination), regardless of the action of lowercase.

If flg=NIL, clispify will use uppercase versions of AND, OR, et al. The value of lowercase is its previous "setting". Lowercase is undoable. The initial setting for lowercase is T.

Index for Section 23

	Page Numbers
ACCESSFNS (record package)	23.30-31
ADD (change word)	23.37
ALWAYS (clisp iterative statement operator)	23.12
AMBIGUOUS DATA PATH (error message)	23.34
AMBIGUOUS RECORD FIELD (error message)	23.34
ARRAYBLOCK (record package)	23.30
ARRAYRECORD (record package)	23.28
AS (clisp iterative statement operator)	23.16
assignments (in clisp)	23.8
ASSOCRECORD (record package)	23.28
ATOMRECORD (record package)	23.30
BETWEEN (record field type)	23.29
BIND (clisp iterative statement operator)	23.13
BINDS (as value of INFO property)	23.40
BITS (record field type)	23.29
BROADSCOPE (property name)	23.54
BY (clisp iterative statement operator)	23.15,14,17
CAUTIOUS (DWIM mode)	23.3,52
CHANGE (change word)	23.38
CHANGETRAN (in record package)	23.36-38
changing record declarations	23.35
CL (edit command)	23.60
CLDISABLE[OP]	23.59
CLISP	23.1,2-61
(CLISP declarations . form)	23.22
CLISP interaction with user	23.52
CLISP internal conventions	23.53
CLISP operation	23.51-52
CLISPPARRAY (clisp variable/parameter)	23.57,22,60
CLISPBRACKET (property name)	23.55
CLISPCHARARRAY (clisp variable/parameter)	23.55
CLISPCHARS (clisp variable/parameter)	23.55
CLISPDEC[DECLST]	23.58,24
CLISPFLG (clisp variable/parameter)	23.56
CLISPFORWORDSPLST (clisp variable/parameter)	23.11
CLISHELPFLG (clisp variable/parameter)	23.57,53
CLISPIFTRANFLG (clisp variable/parameter)	23.59,22
CLISPIFWORDSPLST (clisp variable/parameter)	23.10
CLISPIFY[X;L]	23.38,58,25,39-40
CLISPIFY (makefile option)	23.40,60
CLISPIFYENGLSHFLG (clisp variable/parameter)	23.59,21,39
CLISPIFYFNS[FNS] NL*	23.59
CLISPIYPACKFLG (clisp variable/parameter)	23.59,39
CLISPIYPRETTYFLG (prettyprint variable/parameter)	23.59
CLISPIFYUSERFN (clisp variable/parameter)	23.60,40
CLISPINFIX (property name)	23.54
CLISPINFIXSPLST (clisp variable/parameter)	23.55,7
CLISPI.S.GAG (clisp variable/parameter)	23.19
clisprecordtypes	23.31
CLISPRETRANFLG (clisp variable/parameter)	23.59,22
CLISPTRAN[X;TRAN]	23.57
CLISPTYPE (property name)	23.53-54
CLISWORD (property name)	23.54
CLISP%	23.23,60
CLISP: (edit command)	23.60,22
CLREMPARSFLG (clisp variable/parameter)	23.59,39

Page
Numbers

CL:FLG (clisp variable/parameter)	23.59,39
COLLECT (clisp iterative statement operator)	23.12
COMPILEUSERFN (use by clisp)	23.42
compiling CLISP	23.42
constructing lists (in clisp)	23.10
control-L	23.46
COPYALL[X]	23.33
COPYING (record package)	23.33
COUNT (clisp iterative statement operator)	23.12
CREATE NOT DEFINED FOR THIS RECORD (error message)	23.30
CREATE (record package)	23.31-33
DATATYPE (record package)	23.29
data-paths (in records in clisp)	23.34,26
DATUM (use in changetran constructs)	23.38
declarations (in clisp)	23.24,8,10,25
DECLARE (clisp iterative statement operator)	23.17
DECLAREDATATYPE	23.29
DECLARE: (clisp iterative statement operator) ...	23.16
DEFINED, THEREFORE DISABLED IN CLISP (error message)	23.11
defining new iterative statement operators	23.19-21
disabling CLISP operators	23.59
DO (clisp iterative statement operator)	23.12
DW (edit command)	23.61
DWIMCHECKPROGLABELSFLG (dwim variable/parameter) ..	23.58,41
DWIMCHECK#ARGSFLG (dwim variable/parameter)	23.58,41
DWIMESSGAG (dwim variable/parameter)	23.58,42
DWIMIFY[X;QUIETFLG;L]	23.40,57,41-42,51,58
dwimify (printed by dwim)	23.42
DWIMIFYCOMPFLG (compiler variable/parameter)	23.58,42
DWIMIFYFNS[FNS] NL*	23.58,41
EACHTIME (clisp iterative statement operator) ...	23.16,17
EDITREC[EDITRECX] NL*	23.35
english phrases in clisp	23.21
errors in iterative statements	23.18
FETCH (use in records in clisp)	23.26
FFETCH (use in records in clisp)	23.26
FIELDLOOK[FIELDNAME]	23.35
FILETYPE (property name)	23.40,42,60
FINALLY (clisp iterative statement operator) ...	23.16,17
FIRST (clisp iterative statement operator)	23.16,17
FIXP (record field type)	23.29
FIXSPELLDEFAULT (dwim variable/parameter)	23.41
FLAG (record field type)	23.29
FLOATING (record field type)	23.29
FLOATP (record field type)	23.29
FLUSHRIGHT[POS:X;MIN;P2FLAG;CENTERFLAG;FILE] ..	23.50
FOR (clisp iterative statement operator)	23.13
FREPLACE (use in records in clisp)	23.26
FROM (clisp iterative statement operator)	23.14,15-16
FUNNYATOMLST (clisp variable/parameter)	23.60,39
GETHASH[ITEM;HARRAY] SUBR	23.22
global variables	23.41
GLOBALVAR (property name)	23.41
GLOBALVARS (compiler variable/parameter)	23.41
GO (use in iterative statement in clisp)	23.17
HASHLINK (record package)	23.28

Page
Numbers

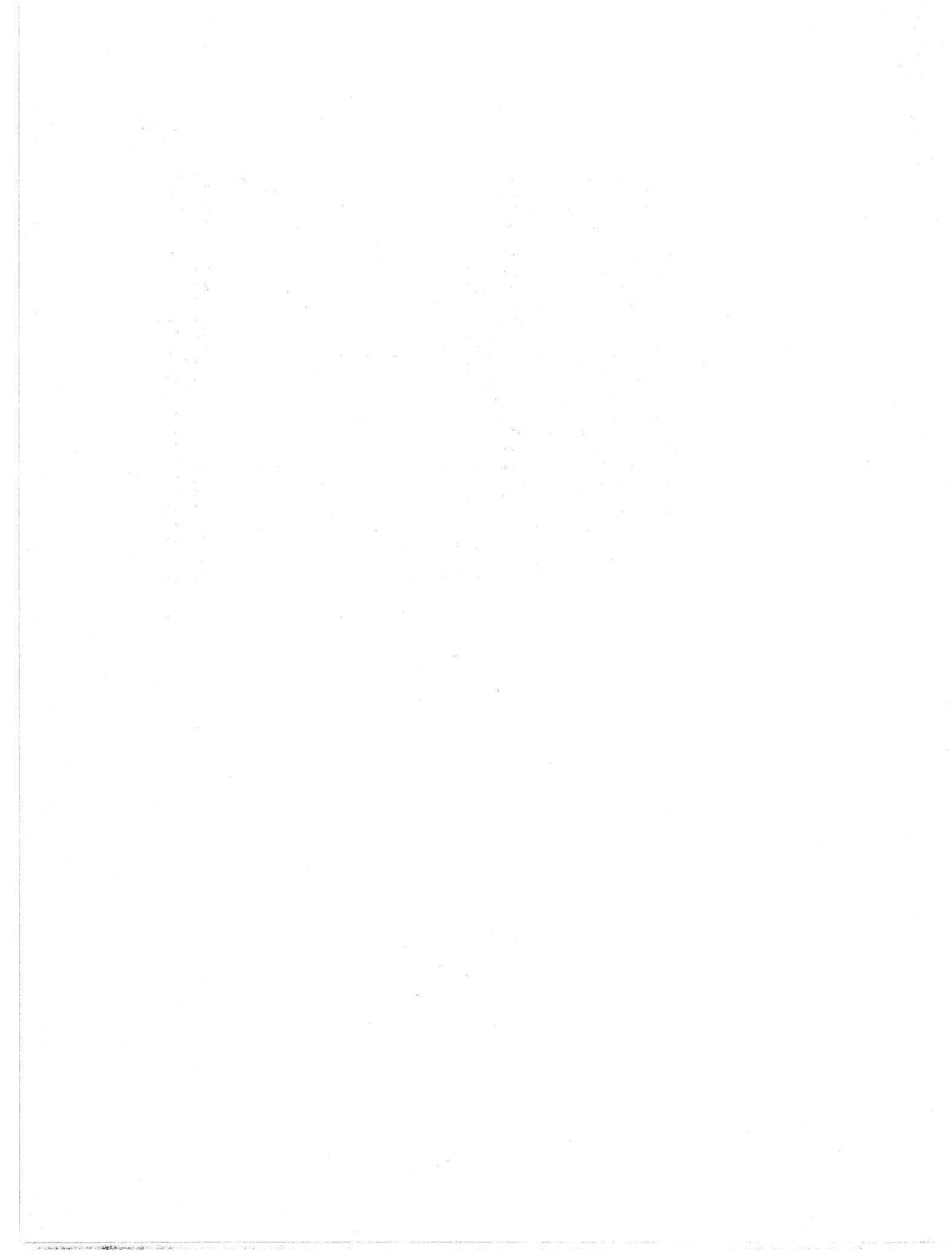
IF-THEN-ELSE statements	23.10
IN (clisp iterative statement operator)	23.13,15,17
infix operators (in clisp)	23.6-8
INFO (property name)	23.39,40
INIT (use in record declarations)	23.31
INITRECORDS (file package command)	23.27
INSIDE (clisp iterative statement operator)	23.14
INTEGER (record field type)	23.29
iterative statements (in clisp)	23.11-21
I.S.OPR[NAME;FORM;OTHERS;EVALFLG]	23.19
i.s.ops	23.11-21
I.S.OPRS (file package command)	23.21
i.s.type	23.13,19
JOIN (clisp iterative statement operator)	23.12
LABELS (use with clispify)	23.39
LASTWORD (dwim variable/parameter)	23.8
LISPFN (property name)	23.54
LISTGET[LST;PROP]	23.28
LISTPUT[LST;PROP;VAL]	23.28
local record declarations (in clisp)	23.25
lower case in clisp	23.61
LOWERCASE[FLG]	23.61
makefile and clisp	23.60,23
MISSING OPERAND (dwim error message)	23.51
MISSING OPERATOR (clisp error message)	23.51
MODIFIER (use with iterative statement operators)	23.21
n (n a number, printout command)	23.45
NEVER (clisp iterative statement operator)	23.12
NEWISWORD[SING;PLU;FORM;VARS]	23.21
NO DO, COLLECT, OR JOIN (error message)	23.19
NOCLISP (makefile option)	23.23,60
NOFIXFNSLST (dwim variable/parameter)	23.57,41,43
NOFIXVARSLST (dwim variable/parameter)	23.57,41,43,51
NOSPELLFLG (dwim variable/parameter)	23.57
OLD (clisp iterative statement operator)	23.13,5,14
ON (clisp iterative statement operator)	23.13,15
order of precedence of CLISP operators	23.9
ORIGINAL (clisp iterative statement operator) ...	23.17,20
OUTOF (clisp iterative statement operator)	23.16
POINTER (record field type)	23.29
POP (change word)	23.38
POSSIBLE NON-TERMINATING ITERATIVE STATEMENT (error message)	23.19
POSSIBLE PARENTHESIS ERROR (error message)	23.41
PPT[X] NL*	23.60,22
PPT (edit command)	23.22,60
precedence rules (for CLISP operators)	23.7
prefix operators (in clisp)	23.8
PRETTYTRANFLG (clisp variable/parameter)	23.60,22-23
printout package	23.43-51
PRINTOUTMACROS (printout parameter)	23.49
PRINTPARA[LMARG;RMARG;LIST;P2FLAG;PARENFLAG;FILE].	23.50
PROPRECORD (record package)	23.28
PUSH (change word)	23.37
PUSHLIST (change word)	23.37
PUSHNEW (change word)	23.37
PUTASSOC[KEY;VAL;ALST]	23.28

Page
Numbers

RELOOK[RECNAME;TL;LOCALDEC;PARENT;ERROR]	23.35
record declarations (in clisp)	23.27,25,28-32
record package (in clisp)	23.25-36
RECORD (record package)	23.28
RECORDACCESS[FIELD;DATUM;DEC;TYPE;NEWVALUE]	23.36
RECORDFIELDNAMES[RECORDNAME]	23.35
RECORDS (file package command)	23.27,29
record-type (record package)	23.27
REPEATUNTIL (clisp iterative statement operator) ..	23.14
REPEATWHILE (clisp iterative statement operator) ..	23.14
REPLACE UNDEFINED FOR FIELD (error message)	23.30
REPLACE (use in records in clisp)	23.26
RETURN (use in iterative statement in clisp)	23.17
REUSING (record package)	23.33
SETFN (property name)	23.54
spelling correction	23.7,10-11,55
spelling lists	23.7,10-11,55
structure modification (the changetran package) ..	23.37
subdeclarations (record package)	23.32
SUM (clisp iterative statement operator)	23.12
SUSPICIOUS PROG LABEL (error message)	23.41
SWAP (change word)	23.38
T (printout command)	23.46
THEREIS (clisp iterative statement operator)	23.12
TO (clisp iterative statement operator)	23.14,15-16
translations (in clisp)	23.22-23
TREAT AS CLISP ? (printed by dwim)	23.51
TREATASCLISPFLG (clisp variable/parameter)	23.51
TREATED AS CLISP (printed by dwim)	23.51
TRUSTING (DWIM mode)	23.3,51-52
TYPERECORD (record package)	23.28
TYPE? NOT IMPLEMENTED FOR THIS RECORD (error message)	23.34
TYPE? (record package)	23.33,28,34
TYPE? (use in record declarations)	23.32
UNABLE TO DWIMIFY (error message)	23.42
UNARYOP (property name)	23.54
undoing DWIM corrections	23.42
UNLESS (clisp iterative statement operator)	23.14
UNTIL (clisp iterative statement operator)	23.14
USEDFREE (clisp declaration)	23.41,43
USERRECORDTYPE (property name)	23.31
USING (record package)	23.33
WHEN (clisp iterative statement operator)	23.14
WHERE (clisp iterative statement operator)	23.20
WHILE (clisp iterative statement operator)	23.14
~ (clisp operator)	23.9
! (use with <,> in clisp)	23.10
!! (use with <,> in clisp)	23.10
# (printout command)	23.49
\$ (<esc>) (in clisp)	23.8-9
' (clisp operator)	23.8
. (printout command)	23.45
-n (n a number, printout command)	23.45
.BASE (printout command)	23.46
.CENTER (printout command)	23.48
.CENTER2 (printout command)	23.48

Page
Numbers

.F (printout command)	23.49
.FONT (printout command)	23.46
.FR (printout command)	23.48
.FR2 (printout command)	23.48
.I (printout command)	23.48
.N (printout command)	23.49
.PAGE (printout command)	23.46
.PARA (printout command)	23.47
.PARA2 (printout command)	23.47
.PPF (printout command)	23.47
.PPFTL (printout command)	23.47
.PPV (printout command)	23.47
.PPVTL (printout command)	23.47
.P2 (printout command)	23.47
.RESET (printout command)	23.45
.SKIP (printout command)	23.46
.SP (printout command)	23.45
.SUB (printout command)	23.46
.SUP (printout command)	23.46
.TAB (printout command)	23.45
.TAB0 (printout command)	23.45
/REPLACE	23.26
: (clisp operator)	23.8
<, > (use in clisp)	23.10
← operator (in clisp)	23.8-9
← (use in record declarations)	23.31



SECTION 24

LISPUSERS PACKAGES

This chapter contains packages which are of sufficient utility that they would otherwise be included as part of the Interlisp system, except for virtual address space limitations. In Interlisp-10, these packages normally reside on either the directory <LISP> or <LISPUSERS>.

24.1 PATTERN MATCH COMPILER¹

The pattern match compiler provides within CLISP a fairly general pattern match facility. The purpose of this pattern match facility is to make more convenient the specifying of certain tests that would otherwise be clumsy to write (and not as intelligible), by allowing the user to give instead a pattern which the datum is supposed to match. Essentially, the user writes "Does the (expression) X look like (the pattern) P?" For example, X:(& 'A -- 'B) asks whether the second element of X is an A, and the last element a B. The implementation of the matching is performed by computing (once) the equivalent Interlisp expression which will perform the indicated operation, and substituting this for the pattern, and *not* by invoking each time a general purpose capability such as that found in FLIP or PLANNER. For example, the translation of X:(& 'A -- 'B)

is:
(AND (EQ (CADR X) (QUOTE A)) (EQ (CAR (LAST X)) (QUOTE B))). Thus the CLISP pattern match facility is really a Pattern Compiler, and the emphasis in its design and implementation has been more on the efficiency of object code than on generality and sophistication of its matching capabilities. The goal was to provide a facility that could and would be used even where efficiency was paramount, e.g., in inner loops. As a result, the CLISP pattern match facility does not contain (yet) some of the more esoteric features of other pattern match languages, such as repeated patterns, disjunctive and conjunctive patterns, recursion, etc. However, the user can be confident that what facilities it does provide will result in Interlisp expressions comparable to those he would generate by hand.²

The syntax for pattern match expressions is form:pattern, where pattern is a list as described below. As with iterative statements, the translation of patterns, i.e., the corresponding Interlisp expressions,

¹ The pattern match compiler was written by L. M. Masinter. It can be loaded from the file MATCH.COM, or, since the entries have a FILEDEF property, (see Section 17) simply using a pattern match construct will cause the file to be loaded automatically.

² Wherever possible, already existing Interlisp functions are used in the translation, e.g., the translation of (\$ 'A \$) uses MEMB, (\$ ('A \$) \$) uses ASSOC, etc.

are stored in `clisparray`, a hash array, as described in Section 23.5. The original expression, `form:pattern`, is replaced by an expression of the form (`MATCH form WITH pattern`). CLISP also recognizes expressions input in this form.

If `form` appears more than once in the translation, and it is not either a variable, or an expression that is easy to (re)compute, such as `(CAR Y)`, `(CDDR Z)`, etc., a dummy variable will be generated and bound to the value of `form` so that `form` is not evaluated a multiple number of times. For example, the translation of `(FOO X):($ 'A $)` is simply `(MEMB (QUOTE A) (FOO X))`, while the translation of `(FOO X):('A 'B --)` is:

```
[PROG ($$2) (RETURN
  (AND (EQ (CAR (SETQ $$2 (FOO X)))
            (QUOTE A))
        (EQ (CADR $$2) (QUOTE B)).
```

In the interests of efficiency, the pattern match compiler assumes that all lists end in `NIL`, i.e., there are no LISTP checks inserted in the translation to check tails. For example, the translation of `X:('A & --)` is `(AND (EQ (CAR X) (QUOTE A)) (CDR X))`, which will match with `(A B)` as well as `(A . B)`. Similarly, the pattern match compiler does not insert LISTP checks on elements, e.g., `X:(('A --) --)` translates simply as `(EQ (CAAR X) (QUOTE A))`, and `X:((\$1 \$1 --) --)` as `(CDAR X)`.³ Note that the user can explicitly insert LISTP checks himself by using `@`, as described on page 24.3, e.g., `X:((\$1 \$1 --)@LISTP --)` translates as `(CDR (LISTP (CAR X)))`.

PATTERN ELEMENTS

A pattern consists of a list of pattern elements. Each pattern element is said to match either an element of a data structure or a segment. (cf. the editor's pattern matcher, "--" matches any arbitrary segment of a list, while & or a subpattern match only one element of a list.) Those patterns which may match a segment of a list are called *segment* patterns; those that match a single element are called *element* patterns.

ELEMENT PATTERNS

There are several types of element patterns, best given by their syntax:

³ The insertion of LISTP checks for *elements* is controlled by the variable `patlistpcheck`. When `patlistpcheck` is T, LISTP checks are inserted, e.g., `X:(('A --) --)` translates as:

`(EQ (CAR (LISTP (CAR (LISTP X)))) (QUOTE A))`

`patlistpcheck` is initially NIL. Its value can be changed within a particular function by using a local declaration, as described in Section 23.10.

<u>PATTERN</u>	<u>MEANING</u>
\$1, or &	matches an arbitrary element of a list
'expression	matches only an element which is equal to the given expression e.g., 'A, ⁴ '(A B).
=form	matches only an element which is <u>equal</u> to the value of form, e.g., =X, =(REVERSE Y).
= =form	same as =, but uses an <u>eq</u> check instead of <u>equal</u> .
atom	treatment depends on setting of <u>patvardefault</u> . If <u>patvardefault</u> is ' or QUOTE, same as 'atom. If <u>patvardefault</u> is = or EQUAL, same as =atom. If <u>patvardefault</u> is == or EQ, same as ==atom. If <u>patvardefault</u> is ← or SETQ, same as atom←&. <u>patvardefault</u> is initially '. ⁵

Note: numbers and strings are always interpreted as though patvardefault were =, regardless of its setting. Eq, memb, and assoc are used for comparisons involving small integers.

(pattern ₁ ... pattern _n) n ≥ 1	matches a list which matches the given patterns, e.g., (& &), (-- 'A).
element-pattern@fn	matches an element if the element-pattern matches it, and fn (name of a function or a LAMBDA expression) applied to that element returns non-NIL, e.g., &@NUMBERP matches a number, ('A --)@FOO matches a list whose first element is A, and for which FOO applied to that list is non-NIL. ⁶

⁴ eq, memb, and assoc are automatically used in the translation when the quoted expression is atomic, otherwise equal, member, and sassoc.

⁵ patvardefault can be changed within a particular function by using a local declaration, as described in Section 23.10.

⁶ For "simple" tests, the function-object is applied before a match is attempted with the pattern, e.g., ((-- 'A --)@LISTP --) translates as (AND (LISTP (CAR X)) (MEMB (QUOTE A) (CAR X))), not the other way around. fn may also be a form in terms of the variable @, e.g., &@EQ @ 3) is equivalent to -3.

+

+

*

matches any arbitrary element. If the entire match succeeds, the element which matched the * will be returned as the value of the match.

Note: normally, the pattern match compiler constructs an expression whose value is guaranteed to be non-NIL if the match succeeds and NIL if it fails. However, if a * appears in the pattern, the expression generated will either return NIL if the match fails, or *whatever matched the ** even though that may be NIL. For example, X:(A * --) translates as (AND (EQ (CAR X) (QUOTE A)) (CADR X)).

~element-pattern

matches an element if the element is *not* matched by element-pattern, e.g., ~'A, ~=X, ~(-- 'A --).

+ (*ANY* element-pattern element-pattern ...) matches if any of the contained patterns match.

SEGMENT PATTERNS

\$, or --

matches any segment of a list (including one of zero length).

The difference between \$ and -- is in the type of search they generate. For example, X:(\$ 'A 'B \$) translates as (EQ (CADR (MEMB (QUOTE A) X)) (QUOTE B)), whereas X:(-- 'A 'B \$) translates as: [SOME X (FUNCTION (LAMBDA (\$\$2 \$\$1) (AND (EQ \$\$2 (QUOTE A)) (EQ (CADR \$\$1) (QUOTE B))))]. Thus, a paraphrase of (\$ 'A 'B \$) would be "Is the element following the *first* A a B?", whereas a paraphrase of (-- 'A 'B \$) would be "Is there *any* A immediately followed by a B?" Note that the pattern employing \$ will result in a more efficient search than that employing --. However, (\$ 'A 'B \$) will not match with (X Y Z A M N O A B C), but (-- 'A 'B \$) will.

Essentially, once a pattern following a \$ matches, the \$ never resumes searching, whereas -- produces a translation that will always continue searching until there is no possibility of success. However, if the pattern match compiler can deduce from the pattern that continuing a search after a particular failure cannot possibly succeed, then the translations for both -- and \$ will be the same. For example, both X:(\$ 'A \$3 \$) and (-- 'A \$3 --) translate as (CDDDR (MEMB (QUOTE A) X)), because if there are not three elements following the first A, there certainly will not be three elements following subsequent A's, so there is no reason to continue searching, even for --. Similarly, (\$ 'A \$ 'B \$) and (-- 'A -- 'B --) are equivalent.

\$2, \$3, etc.

matches a segment of the given length. Note that \$1 is not a segment pattern.

!element-pattern

matches any segment which the given element pattern would match as a list. For example, if the value of FOO is (A B C) !=FOO will match the segment ... A B C ... etc. Note that !* is permissible and means Value-of-match \leftarrow \$, e.g., X:(\$ 'A !*) translates to (CDR (MEMB (QUOTE A) X)).

Note: since ! appearing in front of the last pattern specifies a match with some *tail* of the given

expression, it also makes sense in this case for a ! to appear in front of a pattern that can only match with an atom, e.g., (\$2 ! 'A) means match if cddr of the expression is the atom A. Similarly, X:(\\$! 'A) translates to (EQ (CDR (LAST X)) (QUOTE A)).

!atom

treatment depends on setting of patvardefault. If patvardefault is ' or QUOTE, same as !atom (see above discussion). If patvardefault is = or EQUAL, same as !=atom. If patvardefault is == or EQ, same as !==atom. If patvardefault is ← or SETQ, same as atom←\$.

The atom "." is treated *exactly* like !.⁷ In addition, if a pattern ends in an atom, the "." is first changed to !, e.g., (\$1 . A) and (\$1 ! A) are equivalent, even though the atom "." does not explicitly appear in the pattern.

Segment-pattern@function-object

matches a segment if the segment-pattern matches it, and the function object applied to the corresponding segment (as a list) returns non-NIL, e.g., (\$@CDDR 'D \$) matches (A B C D E) but not (A B D E), since CDDR of (A B) is NIL.

Note: an @ pattern applied to a segment will require *computing* the corresponding structure (with lendiff) each time the predicate is applied (except when the segment in question is a tail of the list being matched).

ASSIGNMENTS

Any pattern element may be preceded by a variable and a '←', meaning if the match succeeds (i.e., everything matches), the variable given is to be set to *what matches* that pattern element. For example, if X = (A B C D E), X:(\\$2 Y←\\$3) will set Y to (C D E). *Assignments are not performed until the entire match has succeeded*. Thus, assignments cannot be used to specify a search for an element found earlier in the match, e.g., X:(Y←\\$1 =Y --)⁸ will *not* match with (A A B C ...).⁹ This type of match is achieved by using place-markers, described below.

⁷ With one exception, namely '!' preceding an assignment does not have the special interpretation that ! has preceding an assignment (see page 24.6). For example, X:('A . FOO←'B) translates as: (AND (EQ (CAR X) (QUOTE A)) (EQ (CDR X) (QUOTE B)) (SETQ FOO (CDR X))), but X:('A ! FOO←'B) translates as:

```
(AND (EQ (CAR X) (QUOTE A))
     (NULL (CDDR X))
     (EQ (CADR X) (QUOTE B))
     (SETQ FOO (CDR X))).
```

⁸ The translation of this pattern is:

```
(COND ((AND (CDR X) (EQUAL (CADR X) Y))
        (SETQ Y (CAR X)))
      T)).
```

The AND is because if Y is NIL, the pattern should match with (A NIL), but not with just (A). The T is because (CAR X) might be NIL.

⁹ unless, of course, the value of Y was A before the match started.

If the variable is preceded by a !, the assignment is to the *tail* of the list as of that point in the pattern, i.e., that portion of the list matched by the remainder of the pattern. For example, if X is (A B C D E), X:(\$!Y \leftarrow 'C 'D \$) sets Y to (C D E), i.e., cddr of X. In other words, when ! precedes an assignment, it acts as a modifier to the \leftarrow , and has no effect whatsoever on the pattern itself, e.g., X:('A 'B) and X:('A !FOO \leftarrow 'B) match identically, and in the latter case, FOO will be set to CDR of X.

Note: * \leftarrow pattern-element and !* \leftarrow pattern-element are acceptable, e.g., X:(\$ 'A * \leftarrow ('B --) --) translates as:

```
[PROG ($$2) (RETURN
  (AND (EQ (CAADR (SETQ $$2 (MEMB (QUOTE A) X)))
           (QUOTE B))
        (CADR $$2))]
```

PLACE-MARKERS

Variables of the form #n, n a number, are called place-markers, and are interpreted specially by the pattern match compiler. Place-markers are used in a pattern to mark or refer to a particular pattern element. Functionally, they are used like ordinary variables, i.e., they can be assigned values, or used freely in forms appearing in the pattern, e.g., X:(#1 \leftarrow \$1 =(ADD1 #1)) will match the list (2 3). However, they are not really variables in the sense that they are not bound, nor can a function called from within the pattern expect to be able to obtain their values. For convenience, regardless of the setting of patvardefault, the first appearance of a defaulted place-marker is interpreted as though patvardefault were \leftarrow . Thus the above pattern could have been written as X:(#1 =(ADD1 #1)). Subsequent appearances of a place-marker are interpreted as though patvardefault were =. For example, X:(#1 #1 --) is equivalent to X:(#1 \leftarrow \$1 =#1 --), and translates as (AND (CDR X) (EQUAL (CAR X) (CADR X))).¹⁰

REPLACEMENTS

Any pattern element may be *followed* by a " \leftarrow " and a form, meaning if the match succeeds, the part of the data that matched is to be *replaced* (e.g., with RPLACA or RPLACD)¹¹ with the value of <form>. For example, if X =(A B C D E), X:(\$ 'C \$1 \leftarrow Y \$1) will replace the third element of X with the value of Y. As with assignments, *replacements are not performed until after it is determined that the entire match will be successful*.

Replacements involving segments splice the corresponding structure into the list being matched, e.g., if X is (A B C D E F) and FOO is (1 2 3), after the pattern ('A \$ \leftarrow FOO 'D \$) is matched with X, X will be (A 1 2 3 D E F), and FOO will be eq to CDR of X, i.e., (1 2 3 D E F).

Note that (\$ FOO \leftarrow FIE \$) is ambiguous, since it is not clear whether FOO or FIE is the pattern element, i.e., whether \leftarrow specifies assignment or replacement. For example, if patvardefault is =,

¹⁰ Just (EQUAL (CAR X) (CADR X)) would incorrectly match with (NIL).

¹¹ The user can indicate he wants rplaca and rplacd used, or frplaca and frplacd, by means of declarations. The initial default is for rplaca and rplacd.

this pattern can be interpreted as ($\$ \text{ FOO} \leftarrow \text{FIE} \$$), meaning search for the value of FIE, and if found set FOO to it, or ($\$ =\text{FOO} \leftarrow \text{FIE} \$$) meaning search for the value of FOO, and if found, store the value of FIE into the corresponding position. In such cases, the user should disambiguate by not using the patvardefault option, i.e., by specifying ' or =.

RECONSTRUCTION

The user can specify a value for a pattern match operation other than what is returned by the match by writing after the pattern $=>$ followed by another form, e.g., $X:(\text{FOO} \leftarrow \$ \text{ 'A } --) => (\text{REVERSE } \text{FOO})$,¹² which translates as:

```
[PROG ($$2) (RETURN
  (COND ((SETQ $$2 (MEMB (QUOTE A) X))
         (SETQ FOO (LDIFF X $2)))
        (REVERSE FOO)).
```

Place-markers in the pattern can be referred to from within form, e.g., the above could also have been written as $X:(\text{!#1 } \text{'A } --) => (\text{REVERSE } \text{!#1})$. If $->$ is used in place of $=>$, the expression being matched is also *physically changed* to the value of form. For example, $X:(\text{#!1 } \text{'A } \text{#!2}) -> (\text{CONS } \text{#!1 } \text{#!2})$ would remove the second element from X, if it were equal to A.

In general, form1:pattern->form2 is translated so as to compute form2 if the match is successful, and then smash its value into the first node of form1. However, whenever possible, the translation does not actually require form2 to be computed in its entirety, but instead the pattern match compiler uses form2 as an indication of what should be done to form1. For example, $X:(\text{#!1 } \text{'A } \text{#!2}) -> (\text{CONS } \text{#!1 } \text{#!2})$ translates as $(\text{AND } (\text{EQ } (\text{CADR } X) (\text{QUOTE A})) (\text{RPLACD } X (\text{CDDR } X)))$.

EXAMPLES

$X:(-- \text{ 'A } --)$ -- matches any arbitrary segment. 'A matches only an A, and the 2nd -- again matches an arbitrary segment; thus this translates to $(\text{MEMB } (\text{QUOTE A}) X)$.

$X:(-- \text{ 'A })$ Again, -- matches an arbitrary segment; however, since there is no -- after the 'A, A must be the last element of X. Thus this translates to: $(\text{EQ } (\text{CAR } (\text{LAST } X)) (\text{QUOTE A}))$.

$X:(\text{'A } \text{'B } -- \text{ 'C } \$3 --)$ CAR of X must be A, and CADR must be B, and there must be at least three elements after the first C, so the translation is:
 $(\text{AND } (\text{EQ } (\text{CAR } X) (\text{QUOTE A}))
 (\text{EQ } (\text{CADR } X) (\text{QUOTE B}))
 (\text{CDDDR } (\text{MEMB } (\text{QUOTE C}) (\text{CDDR } X))))$

¹² The original CLISP is replaced by an expression of the form (MATCH form1 WITH pattern => form2). CLISP also recognizes expressions input in this form.

X:(('A 'B) 'C Y←\$1 \$) Since ('A 'B) does not end in \$ or --, (CDDAR X) must be NIL.

```
(COND
  ((AND (EQ (CAAR X) (QUOTE A))
        (EQ (CADAR X) (QUOTE B))
        (NULL (CDDAR X))
        (EQ (CADR X) (QUOTE C))
        (CDDR X)))
   (SETQ Y (CADDR X))
  T))
```

X:(#1 'A \$ 'B 'C #1 \$) #1 is implicitly assigned to the first element in the list. The \$ searches for the first B following A. This B must be followed by a C, and the C by an expression equal to the first element.

```
[PROG ($$2) (RETURN
  (AND (EQ (CADR X) (QUOTE A))
    (EQ [CADR (SETQ $$2 (MEMB (QUOTE B) (CDDR X)
      (QUOTE C)))
    (CDDR $$2)
    (EQUAL (CADDR $$2) (CAR X)]
```

X:(#1 'A -- 'B 'C #1 \$) Similar to the pattern above, except that -- specifies a search for *any* B followed by a C followed by the first element, so the translation is:

```
[AND (EQ (CADR X) (QUOTE A))
  (SOME (CDDR X) (FUNCTION (LAMBDA ($$2 $$1)
    (AND (EQ $$2 (QUOTE B))
      (EQ (CADR $$1) (QUOTE C))
      (CDDR $$1)
      (EQUAL (CADDR $$1) (CAR X)]
```

This concludes the description of the pattern match compiler.

24.2 EDITA¹³

Edita is an editor for arrays. However, its most frequent application is in editing compiled functions (which are also arrays in Interlisp-10), and a great deal of effort in implementing Edita, and most of its special features, are in this area. For example, Edita knows the format and

¹³ Edita was written by W. Teitelman, and modified by D. C. Lewis. That portion of Edita relating to compiled code may or may not be available in implementation of Interlisp other than Interlisp-10. Edita is contained on the file EDITA.COM. Edita also has a FILEDEF property so that the user can simply call Edita and the file will be automatically loaded.

conventions of Interlisp-10 compiled code, and so, in addition to decoding instructions a la DDT,¹⁴ Edita can fill in the appropriate COREVALS, symbolic names for index registers, references to literals, linked function calls, etc. The following output shows a sequence of instructions in a compiled function first as they would be printed by DDT, and second by Edita.

466716/	PUSH 16,LISP&KNIL	3/	PUSH PP,KNIL
466717/	PUSH 16,LISP&KNIL	4/	PUSH PP,KNIL
466720/	HRRZ 1,-12(16)	5/	HRRZ 1,-10(PP) 15
466721/	CAME 1,LISP&KNIL	6/	CAME 1,KNIL
466722/	JRST 466724	7/	JRST 9
466723/	HRRZ 1,@467575	8/	HRRZ 1,@'BRKFILE
466724/	PUSH 16,1	9/	PUSH PP,1
466725/	LISP&IOFIL,,467576	10/	PBIND 'BRKZ
466726/	-3,-3	11/	-524291
466727/	HRRZ 1,-14(16)	12/	HRRZ 1,-12(PP)
466730/	CAMN 1,467601	13/	CAMN 1,'OK
466731/	JRST 466734	14/	JRST 17
466732/	CAME 1,467602	15/	CAME 1,'STOP
466733/	JRST 466740	16/	JRST 21
466734/	PUSH 16,467603	17/	PUSH PP,'BREAK1
466735/	PUSH 16,467604	18/	PUSH PP,'(ERROR!)
466736/	LISP&FILEN,,467605	19/	CCALL 2,'RETEVAL
466737/	JRST 467561	20/	JRST 422
466740/	CAME 1,467606	21/	CAME 1,'GO
466741/	JRST 466754	22/	JRST 33
466742/	HRRZ 1,@-12(16)	23/	HRRZ 1,@-10(PP)
466743/	PUSH 16,1	24/	PUSH PP,1

Therefore, rather than presenting Edita as an array editor with some extensions for editing compiled code, we prefer to consider it as a facility for editing compiled code, and point out that it can also be used for editing arbitrary arrays.

OVERVIEW

To the user, Edita looks very much like DDT with Interlisp-10 extensions. It is a function of one argument, the name of the function to be edited.¹⁶ Individual registers or cells in the function may be examined by typing their address followed by a slash,¹⁷ e.g.

6/ HRRZ 1,-10(PP)

-
- ¹⁴ DDT is one of the oldest debugging systems still around. For users unfamiliar with it, let us simply say that Edita was patterned after it because so many people are familiar with it.
 - ¹⁵ Note that Edita prints the addresses of cells contained in the function relative to the origin of the function.
 - ¹⁶ An optional second argument can be a list of commands for Edita. These are then executed exactly as though they had come from the teletype.
 - ¹⁷ Underlined characters were typed by the user. Edita uses its own read program, so that it is unnecessary to type a space before the slash or to type a carriage return after the slash.

The slash is really a command to edita to open the indicated register.¹⁸ Only one register at a time can be open, and *only open registers can be changed*. To change the contents of a register, the user first opens it, types the new contents, and then closes the register with a carriage-return,¹⁹ e.g.

7/ CAME 1, '↑ CAMN 1, '↑

If the user closes a register without specifying the new contents, the contents are left unchanged. Similarly, if an error occurs or the user types control-E, the open register, if any, is closed without being changed.

INPUT PROTOCOL

Edita processes all inputs not recognized as commands in the same way. If the input is the name of an instruction (i.e., an atom with a numeric OPD property), the corresponding number is added to the input value being assembled,²⁰ and a flag is set which specifies that the input context is that of an instruction.

The general form of a machine instruction is (opcode ac , @ address (index)) as described in Section 18. Therefore, in instruction context, edita evaluates all atoms (if the atom has a COREVAL property, the value of the COREVAL is used), and then if the atom corresponds to an ac,²¹ shifts it left 23 bits and adds it to the input value, otherwise adds it directly to the input value, but performs the arithmetic in the low 18 bits.²² Lists are interpreted as specifying index registers, and the value of car of the list (again COREVALs are permitted) is shifted left 18 bits. Examples:

```
PUSH PP, KNIL
HRRZ 1, -10(PP)
CAME 1, 'GO
JRST 33 ORG 23
```

The user can also specify the address of a literal via the ' command, see page 24.13. For example, if the literal " UNBROKEN" is in cell 85672, HRRZ 1, " UNBROKEN" is equivalent to HRRZ 1, 85672.

When the input context is *not* that of an instruction, i.e., no OPD has been seen, all inputs are

¹⁸ Edita also converts absolute addresses of cells within the function to relative address on input. Thus, if the definition of foo begins at 85660, typing 6/ is *exactly* the same as typing 85666/.

¹⁹ Since carriage-return has a special meaning, edita indicates the balancing of parentheses by typing a space.

²⁰ The input value is initially 0.

²¹ i.e., if a "," has not been seen, *and* the value of the atom is less than 16, *and* the low 18 bits of the input value are all zero.

²² If the absolute value of the atom is greater than 1000000Q, full word arithmetic is used. For example, the indirect bit is handled by simply binding @ to 2000000Q.

²³ Edita cannot in general know whether an address field in an instruction that is typed in is relative or absolute. Therefore, the user must add ORG, the origin of the function, to the address field himself. Note that edita would print this instruction, JRST 53 ORG, as JRST 53.

evaluated (the value of an atom with a COREVAL property is the COREVAL.) Then numeric values are simply added to the previous input value; non-numeric values *become* the input value.²⁴

The only exception to the entire procedure occurs when a register is open that is in the pointer region of the function, i.e., literal table. In this case, atomic inputs are *not* evaluated. For example, the user can change the literal FOO to FIE by simply opening that register and then typing FIE followed by carriage-return, e.g.

'FOO/ FOO FIE\r

Note that this is equivalent to 'FOO/ FOO (QUOTE FIE)\r

EDITA COMMANDS AND VARIABLES

\ (carriage-return) If a register is open and an input was typed, store the input in the register and close it.²⁵

If a register is open and nothing was typed, close the register without changing it.

If a register is not open and input was typed, type its value.

ORG Has the value of the address of the first instruction in the function. i.e., loc of getd of the function.

/ Opens the register specified by the low 18 bits of the quantity to the left of the /, and types its contents. If nothing has been typed, it uses the last thing typed by Edita, e.g.,

35/ JRST 53 / CAME 1, 'RETURN / RETURN

If a register was open, / closes it without changing its contents.

After a / command, Edita returns to that state of no input having been typed.

tab (control-I) Same as carriage-return, followed by the address of the quantity to the left of the tab, e.g.,

35/ JRST 53 tab
53/ CAME 1, 'RETURN

Note that if a register was open and input was typed, tab will change the open register before closing it, e.g.,

²⁴ Presumably there is only one input in this case.

²⁵ If the register is in the unboxed region of the function, the unboxed value is stored in the register.

35/ JRST 53 JRST 54 tab
54/ JRST 70 2
35/ JRST 54

. (period) has the value of the address of the current (last) register examined.

line-feed same as carriage-return followed by (ADD1 .)/ i.e. closes any open register and opens the *next* register.

↑ same as carriage-return followed by (SUB1 .)/

\$Q (<esc>Q) has as its value the last quantity typed by edita e.g.

35/ JRST 53 \$Q 12
./ JRST 54

LITS has as value the (relative) address of the first literal.

BOXED same as LITS

\$ (dollar) has as value the relative address of the last literal in the function.

= Sets radix to -8 and types the quantity to the left of the = sign, i.e., if anything has been typed, types the input value, otherwise, types \$Q, e.g.

35/ JRST 54 =254000241541Q
JRST 54=25400000066Q

Following =, radix is restored and edita returns to the no input state.

OK leave edita

? return to "no input" state. ? is a "weak" control-E, i.e., it negates any input typed, but does not close any registers.

address1, address2/ prints²⁶ the contents of registers address1 through address2. . is set to address2 after the completion.

'x

corresponds to the ' in LAP. The next expression is read, and if it is a small number, the appropriate offset is added to it. Otherwise, the literal table is searched for x, and the value of 'x' is the (absolute) address of that cell. An error is generated if the literal is not found, i.e., ' cannot be used to *create* literals.

-:atom

defines atom to an address

- (1) the value of \$Q if a register is open,
- (2) the input if any input was typed, otherwise
- (3) the value of ".".²⁷

For example:

```
35/ JRST 54 :FOO2
:FIE2
FIE/ JRST FOO.=35
```

Edita keeps its symbol tables on two free variables, usersyms and symlst. Usersyms is a list of elements of the form (name . value) and is used for *encoding* input, i.e., all variables on usersyms are bound to their corresponding values during evaluation of any expression inside Edita. Symlst is a list of elements of the form (value . name) and is used for *decoding* addresses. Usersyms is initially NIL, while symlst is set to a list of all the corevals. Since the : command adds the appropriate information to both these two lists, new definitions will remain in effect even if the user exits from Edita and then reenters it later.

Note that the user can effectively define symbols without using the : command by appropriately binding usersyms and/or symlst before calling Edita. Also, he can thus use different symbol tables for different applications.

\$W (<esc>W)

search command.

Searching consists of comparing the object of the search with the contents of each register, and printing those that match, e.g.,

```
HRRZ @ $W2
8/ HRRZ 1,@'BRKFILE
23/ HRRZ 1,@-10(PP)
28/ HRRZ 1,@-12(PP)
```

²⁶ output goes to file, initially set to T. The user can also set file (while in Edita) to the name of a disc file to redirect the output. (The user is responsible for opening and closing file.) Note that file only affects output for the address1, address2/ command.

²⁷ Only the low 18 bits are used and converted to a relative address whenever possible.

The \$W command can be used to search either the unboxed portion of a function, i.e., instructions, or the pointer region, i.e., literals, depending on whether or not the object of the search is a number. If any input was typed before the \$W, it will be the object of the search, otherwise the next expression is read and used as the object.²⁸ The user can specify a starting point for the search by typing an address followed by a "," before calling \$W, e.g., 1, JRST \$W. If no starting point is specified, the search will begin at 0 if the object is a number, otherwise at LITS, the address of the first literal.²⁹ After the search is completed, ":" is set to the address of the last register that matched.

If the search is operating in the unboxed portion of the function, only those fields (i.e., instruction, ac, indirect, index, and address) of the object that contain one bits are compared.³⁰ For example, HRRZ @ \$W will find all instances of HRRZ indirect, regardless of ac, index, and address fields. Similarly, 'PRINT \$W will find all instructions that reference the literal PRINT.³¹

If the search is operating in the pointer region, a "match" is as defined in the editor. For example, \$W (&) will find all registers that contain a list consisting of a single expression.

\$C (<esc>C) like \$W except only prints the first match, then prints the number of matches when the search finishes.

EDITING ARRAYS

Edita is called to edit a function by giving it the name of the function. Edita can also be called to edit an array by giving it the array as its first argument,³² in which case the following differences are to be noted:

1. decoding - The contents of registers in the unboxed region are boxed and printed as numbers, i.e., they are never interpreted as instructions, as when editing a function.
2. addressing convention - Whereas 0 corresponds to the first instruction of a function, the first element of an array by convention is element number 1.

²⁸ Note that inputs typed before the \$W will have been processed according to the input protocol, i.e., evaluated; inputs typed after the \$W will not. Therefore, the latter form is usually used to specify searching the literals, e.g., \$W FOO is equivalent to (QUOTE FOO) \$W.

²⁹ Thus the only way the user can search the pointer region for a number is to specify the starting point via ":".

³⁰ Alternately, the user can specify his own mask by setting the variable mask (while in Edita), to the appropriate bit pattern.

³¹ The user may need to establish instruction context for input without giving a specific instruction. For example, suppose the user wants to find all instructions with ac=1 and index=PP. In this case, the user can give & as a pseudo-instruction, e.g., type & 1, (PP).

³² the array itself, *not* a variable whose value is an array, e.g., (EDITA FOO), not EDITA(FOO).

3. input protocols - If a register is open, lists are evaluated, atoms are not evaluated (except for \$Q which is always evaluated). If no register is open, all inputs are evaluated, and if the value is a number, it is added to the "input value".
4. left half - If the left half of an element in the pointer region of an array is not all 0's or NIL, it is printed followed by a ;, e.g.

10/ (A B) ; T

Similarly, if a register is closed, either its left half, right half, or both halves can be changed, depending on the presence or absence, and position of the ; e.g.

<u>10/ (A B) ; T</u>	<u>B ; 2</u>	changes left
<u>/ B ; T</u>	<u>NIL 2</u>	changes right
<u>/ B ; NIL</u>	<u>A ; C 2</u>	changes both

If ; is used in the unboxed portion of an array, an error will be generated.

The \$W command will look at both halves of elements in the pointer region, and match if either half matches. Note that \$W A ; B is not allowed.

This concludes the discussion of Edita.

24.3 PRINTING REENTRANT AND CIRCULAR LIST STRUCTURES

CIRCLPRINT³³

Horribleprint (Section 14) is designed primarily for dumping circular or reentrant list structures (as well as other data structures for which read is not an inverse of print) so that they can be read back in by Interlisp. The circlprint package is designed for printing circular or reentrant structures so that the user can look at them and understand them.

A reentrant list structure is one that contains more than one occurrence of the same (eq) structure. For example, tconc (Section 6) makes use of reentrant list structure so that it does not have to search for the end of the list each time it is called. Thus, if x is a list of 3 elements, (A B C), being constructed by tconc, the reentrant list structure used by tconc for this purpose is:

³³ CIRCLPRINT was written by P. C. Jackson. It is contained on the file CIRCLPRINT.COM.

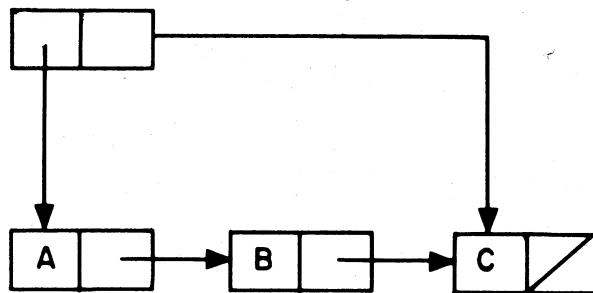


FIGURE 24-1

This structure would be printed by print as ((A B C) C). Note that print would produce the same output for the non-reentrant structure:

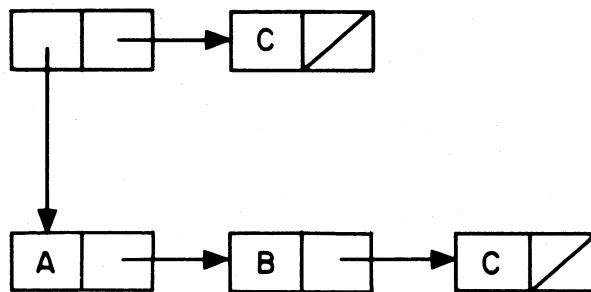


FIGURE 24-2

In other words, print does not indicate the fact that portions of the structure in Figure 24-1 are identical. Similarly, if print is applied to a circular list structure (a special type of reentrant structure) it will never terminate.

For example, if print is called on the structure:

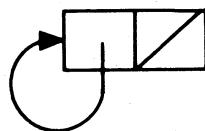


FIGURE 24-3

it will print an endless sequence of left parentheses, and if applied to:

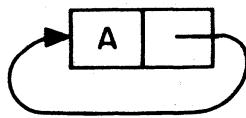


FIGURE 24-4

will print a left parenthesis followed by an endless sequence of A's.

The function circlprint described below produces output that will exactly describe the structure of any circular or reentrant list structure. This output may be in either single or double-line formats. Below are a few examples of the expressions that circlprint would produce to describe the structures discussed above.

expression in Figure 24-1:

single-line:

$((A\ B\ *1*\ C)\ \{1\})$

double-line:

$((A\ B\ C)\ \{1\})$
1

expression in Figure 24-3:

single-line:

$(*1*\ \{1\})$

double-line:

$(\{1\})$
1

expression in Figure 24-4:

single-line:

$(*1*\ A\ .\ \{1\})$

double-line:

$(A\ .\ \{1\})$
1

The more complex structure:

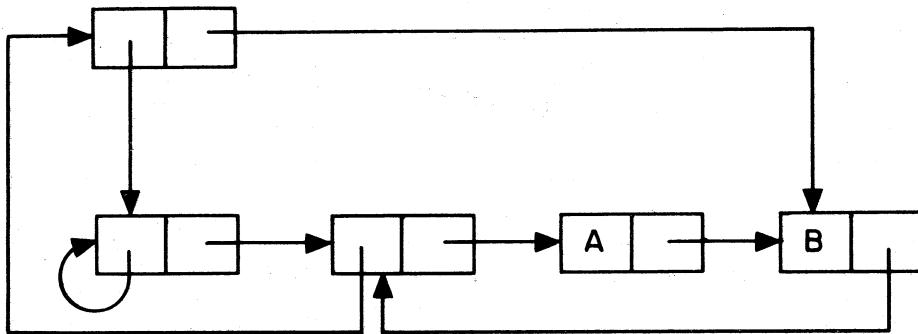


FIGURE 24-5

is printed as follows:

single-line:

(*2* (*1* {1} *3* {2} A *4* B . {3}) . {4})

double-line:

(({1} {2} A B . {3}) . {4})

21 3 4

In both formats, the reentrant nodes in the list structure are labeled by numbers. (A reentrant node is one that has two or more pointers coming into it.) In the single-line format, the label is printed between asterisks at the beginning of the node (list or tail) that it identifies. In the double-line format, the label is printed below the beginning of the node it identifies. An occurrence of a reentrant node that has already been identified is indicated by printing its label in brackets.

circlprint[list;printflg;rlknt]

prints an expression describing list. If printflg=NIL, double-line format is used, otherwise single-line format. circlprint first calls circlmark[list;rlknt], and then calls either rlprin1[list] or rlprin2[list], depending on the value of printflg (T or NIL, respectively). Finally, rlrestore[list] is called, which restores list to its unmarked state. Value is list.

circlmark[list;rlknt]

marks each reentrant node in list with a unique number, starting at rlknt+1 (or 1, if rlknt is NIL). Value is (new) rlknt.

Marking list physically alters it. However, the marking is performed undoably. In addition, list can always be restored by specifically calling rlrestore.

rlprin1[list]

prints an expression describing list in the single-line format. Does not restore list to its uncircmarked state. list must previously have been circlmarked or an error is generated.

rlprin2[list] same as rlprin1, except that the expression describing list is printed in the double-line format.

rlrestore[list] physically restores list to its original, unmarked state.

Note that the user can mark and print several structures which together share common substructures, e.g., several property lists, by making several calls to circlmark, followed by calls to rlprin1 or rlprin2, and finally to rlrestore.

circlmaker[list] list may contain labels and references following the convention used by circlprint for printing reentrant structures in single line format, e.g., (*1* . {1}). circlmaker performs the necessary rplaca's and rplacd's to make list correspond to the indicated structure. Value is (altered) list.

circlmaker1[list] Does the work for circlmaker. Uses free variables labelst and reflst. labelst is a list of dotted pairs of labels and corresponding nodes. reflst is a list of nodes containing references to labels not yet seen. Circlmaker operates by initializing labelst and reflst to NIL, and then calling circlmaker1. It generates an error if reflst is not NIL when circlmaker1 returns. The user can call circlmaker1 directly to "connect up" several structures that share common substructures, e.g., several property lists.

PRINTL³⁴

The printl package uses a different scheme than circlprint to present circular structures in an easily readable format. printl uses indentation a la prettyprint to make it easier for the user to see the nesting of list structure, and prints index numbers for the beginning and ends of expressions so that the user can find what is referred back to easily.³⁵ The following example illustrates the use of printl:

³⁴ PRINTL was written by M. J. Kay. It is contained on the file PRINTL.COM.

³⁵ Note that printl does not provide an output format which can be read back in to reconstruct the original list structure; it is intended primarily as a debugging aid.

```

+
+ 32←(PRINTL (NCONC (SETQQ X (A B C D)) X))
+   1: (A B C D . {1}) :1
+
+ NIL
+
+ 33←(PRINTL (LIST X (CDR X) (CDDR X) (CDDDR X])
+   1: ((A B C D . {2}) {3} {4} {5}) :1
+
+ NIL
+
+ 34←(PRINTL (LIST X (CONS 'P (CDR X)) (CONS 'Q (CDDR X))
+ (CONS 'R (CDDDR X])
+   1: ((A B C D . {2}) :2
+   6: (P . {3})) :6
+   7: (Q . {4})) :7
+   8: (R . {5})) :1
+
+ NIL
+
+ 35←USE LIST FOR CONS
+   1: ((A B C D . {2}) :2
+   6: (P {3})) :6
+   8: (Q {4})) :8
+   10: (R {5})) :1
+
+ NIL

```

+ printl uses the following algorithm: Each list node that is printed (car or cdr) is assigned a number.
+ The second and subsequent appearances of this list node are represented simply by printing the
+ number corresponding to the node in {} brackets. Every line on which the representation of a list
+ begins shows the corresponding number of the *first* such list, i.e. this number corresponds to the
+ first open parenthesis on the line. Similarly, to the right of every line on which a list ends is
+ printed the number that corresponds to the *last* close parenthesis on the line. The numbers for
+ those list nodes which do not correspond to the first open parentheses or the last close parentheses
+ on a line can be obtained by simply counting from the last numbered parenthesis. For example, in
+ the line

```

+
+   1: ((A B C D . {2}) {3} {4} {5}) :1
+
+ 2 is (A B C D), 3 is (B C D), 4 is (C D), and 5 is (D).

```

+ printl[item;depth;lmarg;rmarg;file]
+ prints an item which is known to be, or suspected of being a
+ circular list structure, in the form described above. depth controls
+ the depth of recursion in the car direction and defaults to the value
+ of the variable printdepth (initially 4). Elements of the structure at
+ this depth are printed as '{--}'.

+ lmarg is the left margin. If NIL, lmarg defaults to position[file]. rm
+ is the position at which the righthand column of numbers will be
+ printed. If NIL, rm defaults to (LINELENGTH)-5.

+ Printing is to file, which is opened if necessary.

+ **PRNTL args**
+ lispmacro that performs (PRINTL . args) provided car[args] is not
+ a number. If it is, or if args=NIL, the item to be printed is taken
+ to be the last event on the history list with a non-null value. Thus
+ PRNTL 6 will print the last non-null value with depth=6.

24.4 TRANSOR³⁶

INTRODUCTION

transor is a LISP-to-LISP translator intended to help the user who has a program coded in one dialect of LISP and wishes to carry it over to another. The user loads transor along with a file of transformations. These transformations describe the differences between the two LISPs, expressed in terms of Interlisp editor commands needed to convert the old to new, i.e. to edit forms written in the source dialect to make them suitable for the target dialect. transor then sweeps through the user's program and applies the edit transformations, producing an object file for the target system. In addition, transor produces a file of translation notes, which catalogs the major changes made in the code as well as the forms that require further attention by the user. Operationally, therefore, transor is a facility for conducting massive edits, and may be used for any purpose which that may suggest.

Since the edit transformations are fundamental to this process, let us begin with a definition and some examples. A transformation is a list of edit commands associated with a literal atom, usually a function name. transor conducts a sweep through the user's code, until it finds a form whose car is a literal atom which has a transformation. The sweep then pauses to let the editor execute the list of commands before going on. For example, suppose the order of arguments for the function tconc must be reversed for the target system. The transformation for tconc would then be: ((SW 2 3)). When the sweep encounters the form (TCONC X (FOO)), this transformation would be retrieved and executed, converting the expression to (TCONC (FOO) X). Then the sweep would locate the next form, in this case (FOO), and any transformations for foo would be executed, etc.

Most instances of tconc would be successfully translated by this transformation. However, if there were no second argument to tconc, e.g. the form to be translated was (TCONC X), the command (SW 2 3) would cause an error, which transor would catch. The sweep would go on as before, but a note would appear in the translation listing stating that the transformation for this particular form failed to work. The user would then have to compare the form and the commands, to figure out what caused the problem. One might, however, anticipate this difficulty with a more sophisticated transformation: ((IF (## 3) ((SW 2 3)) ((-2 NIL))), which tests for a third element and does (SW 2 3) or (-2 NIL) as appropriate. It should be obvious that the translation process is no more sophisticated than the transformations used.

This documentation is divided into two main parts. The first describes how to use transor assuming that the user already has a complete set of transformations. The second documents transorset, an interactive routine for building up such sets. transorset contains commands for writing and editing transformations, saving one's work on a file, testing transformations by translating sample forms, etc.

Two transformations files presently exist for translating programs into Interlisp. < LISP > SDS940.XFORMS is for old BBN LISP (SDS 940) programs, and < LISP > LISP16.XFORMS is for Stanford AI LISP 1.6 programs. A set for LISP 1.5 is planned.

³⁶ TRANSOR was written by J. W. Goodwin. It is contained on the file TRANSOR.COM.

USING TRANSOR

The first and most exasperating problem in carrying a program from one implementation to another is simply to get it to read in. For example, SRI LISP uses / exactly as Interlisp uses %, i.e. as an escape character. The function prescan exists to help with these problems: the user uses prescan to perform an initial scan to dispose of these difficulties, rather than attempting to transor the foreign sourcefiles directly.

prescan copies a file, performing character-for-character substitutions. It is hand-coded and is much faster than either readc's or text-editors.

prescan[file;charlst] Makes a new version of file, performing substitutions according to charlst. Each element of charlst must be a dot-pair of two character codes, (OLD . NEW).

For example, SRI files are prescan'ed with charlst = ((37 . 47) (47 . 37)), which exchanges slash (47) and percent-sign (37).

The user should also make sure that the treatment of doublequotes by the source and target systems is similar. In Interlisp, an unmatched double-quote (unless protected by the escape character) will cause the rest of the file to read in as a string.

Finally, the lack of a STOP at the end of a file is harmless, since transor will suppress END OF FILE errors and exit normally.

TRANSLATING

transor is the top-level function of the translator itself, and takes one argument, a file to be translated. The file is assumed to contain a sequence of forms, which are read in, translated, and output to a file called file.TRAN. The translation notes are meanwhile output to file.LSTRAN. Thus the usual sequence for bring a foreign file to Interlisp is as follows: prescan the file; examine code and transformations, making changes to the transformations if needed; transor the file; and clean up remaining problems, guided by the notes. The user can now make a pretty file and proceed to exercise and check out his program. To export a file, it is usually best to transor it, then prescan it, and perform clean-up on the foreign system where the file can be loaded.

transor[sourcefile] Translates sourcefile. Prettyprints translation on file.TRAN: translation listing on file.LSTRAN.

transorform[form] Argument is a LISP form. Returns the (destructively) translated form. The translation listing is dumped to the primary output file.

transorfns[fnlst] Argument is a list of function names whose interpreted definitions are destructively translated. Listing to primary output file.

transform and transorfns can be used to translate expressions that are already in core, whereas transor itself only works on files.

THE TRANSLATION NOTES

The translation notes are a catalog of changes made in the user's code, and of problems which require, or may require, further attention from the user. This catalog consists of two cross-indexed sections: an index of forms and an index of notes. The first tabulates all the notes applicable to any form, whereas the second tabulates all the forms to which any one note applies. Forms appear in the index of forms in the order in which they were encountered, i.e. the order in which they appear on the source and output files. The index of notes shows the name of each note, the entry numbers where it was used, and its text, and is alphabetical by name. The following sample was made by translating a small test file written in SRI LISP.

LISTING FROM TRANSORING OF FILE TESTFILE.;7
DONE ON 1-NOV-71 20:10:47

INDEX OF FORMS

1. APPLY/EVAL at

```
[DEFINEQ
  (FSET (LAMBDA &
    (PROG ...3...
      (SETQ Z (COND
        ((ATOM (SETQ --))
         (COND
           ((ATOM (SETQ Y (NLSETQ "(EVAL W)")))
            --)
           --))
         --))
      -- )]
```

2. APPLY/EVAL at

```
[DEFINEQ
  (FSET (LAMBDA &
    (PROG ...3...
      (SETQ Z (COND
        ((ATOM (SETQ --))
         (COND
           ((ATOM (SETQ --))
            "(EVAL (NCONS W))")
           --))
         --))
      -- )]
```

3. MACHINE-CODE at

```
[DEFINEQ
  (LESS1 (LAMBDA &
    (PROG ...3...
      (COND
        ...2...
        ((NOT (EQUAL (SETQ X2 "(OPENR (MAKNUM & -))"
          )
          --))
        --))
      -- )]
```

4. MACHINE-CODE at

```
[DEFINEQ
  (LESS1 (LAMBDA &
    (PROG ...3...
      (COND
        ...2...
        ((NOT (EQUAL & (SETQ Y2
          "(OPENR (MAKNUM & --)))))
        --))
      -- )]
```

INDEX OF NOTES

APPLY/EVAL at 1, 2.

TRANSOR will translate the arguments of the APPLY or EVAL expression, but the user must make sure that the run-time evaluation of the arguments returns a BBN-compatible expression.

MACHINE-CODE at 3, 4.

Expression dependent on machine-code. User must recode.

The translation notes are generated by the transformations used, and therefore reflect the judgment of their author as to what should be included. Straightforward conversions are usually made without comment; for example, the DEFPROP's in this file were quietly changed to DEFINEQ's. transor found four noteworthy forms on the file, and printed an entry for each in the index of forms, consisting of an entry number, the name of the note, and a printout showing the precise location of the form. The form appears in double-quotes and is the last thing printed, except for closing parentheses and dashes. An ampersand represents one non-atomic element not shown, and two or more elements not shown are represented asn..., where n is the number of elements. Note that the printouts describe expressions on the output file rather than the source file; in the example, the DEFPROP's of SRI LISP have been replaced with DEFINEQ's.

ERRORS AND MESSAGES

transor records its progress through the source file by teletype printouts which identify each expression as it is read in. Progress within large expressions, such as a long DEFINEQ, is reported every three minutes by a printout showing the location of the sweep.

If a transformation fails, transor prints a diagnostic to the teletype which identifies the faulty transformation, and resumes the sweep with the next form. The translation notes will identify the form which caused this failure, and the extent to which the form and its arguments were compromised by the error.

If the transformation for a common function fails repeatedly, the user can type control-H. When the system goes into a break, he can use transorset to repair the transformation, and even test it out (see TEST command, page 24.26). He may then continue the main translation with OK.

TRANSORSET

To use transorset, type transorset() to Interlisp. transorset will respond with a + sign, its prompt character, and await input. The user is now in an executive loop which is like evalqt with some extra context and capabilities intended to facilitate the writing of transformations. transorset will thus progress apply and eval input, and execute history commands just as evalqt would. Edit commands, however, are interpreted as additions to the transformation on which the user is currently working. transorset always saves on a variable named currentfn the name of the last function whose transformation was altered or examined by the user. currentfn thus represents the function whose transformation is currently being worked on. Whenever edit commands are typed to the + sign, transorset will add them to the transformation for currentfn. This is the basic mechanism for writing a transformation. In addition, transorset contains commands for printing out a transformation, editing a transformation, etc., which all assume that the command applies to currentfn if no function is specified. The following example illustrates this process.

```
←TRANSORSET()
+FN TCONC
TCONC
+(SW 2 3)
```

[1]

[2]

```

+TEST (TCONC A B) [3]
P
(TCONC B A)
+TEST (TCONC X) [4]
TRANSLATION ERROR: FAULTY TRANSFORMATION
TRANSFORMATION: ((SW 2 3)) [5]
OBJECT FORM: (TCONC X)

1. TRANSFORMATION ERROR AT [6]
  "(TCONC X)"

(TCONC X)
+(IF (## 3) ((SW 2 3)) ((-2 NIL]) [7]
+SHOW
TCONC
  [(SW 2 3)
   (IF (## 3)
     ((SW 2 3))
     ((-2 NIL])
TCONC
+ERASE [9]
TCONC
+REDO IF [10]
+SHOW
TCONC
  [(IF (## 3)
    ((SW 2 3))
    ((-2 NIL])
TCONC
+TDST
=TEST [11]
(TCONC NIL X)
+

```

In this example, the user begins by using the FN command to set currentfn to TCONC [1]. He then adds to the (empty) transformation for tconc a command to switch the order of the arguments [2] and tests the transformation [3]. His second TEST [4] fails, causing an error diagnostic [5] and a translation note [6]. He writes a better command [7] but forgets that the original SW command is still in the way [8]. He therefore deletes the entire transformation [9] and redoing the IF [10]. This time, the TEST works [11].

TRANSORSET COMMANDS

The following commands for manipulating transformations are all lispxmacros which treat the rest of their input line as arguments. All are undoable.

FN

Resets currentfn to its argument, and returns the new value. In effect FN says you are done with the old function (as least for the moment) and wish to work on another. If the new function already has a transformation, the message (OLD TRANSFORMATIONS) is printed, and any editcommands typed in will be added to the end of the existing commands. FN followed by a carriage return will return the value of currentfn without changing it.

SHOW	Command to prettyprint a transformation. SHOW followed by a carriage return will show the transformation for <u>currentfn</u> , and return <u>currentfn</u> as its value. SHOW followed by one or more function names will show each one in turn, reset <u>currentfn</u> to the last one, and return the new value of <u>currentfn</u> .
EDIT	Command to edit a transformation. Similar to SHOW except that instead of prettyprinting the transformation, EDIT gives it to <u>edite</u> . The user can then work on the transformation until he leaves the editor with OK.
ERASE	Command to delete a transformation. Otherwise similar to SHOW.
TEST	Command for checking out transformations. TEST takes one argument, a form for translation. The translation notes, if any, are printed to the teletype, but in an abbreviated format which omits the index of notes. The value returned is the translated form. TEST saves a copy of its argument on the free variable <u>testform</u> , and if no argument is given, it uses <u>testform</u> , i.e. tries the previous test again.
DUMP	Command to save your work on a file. DUMP takes one argument, a filename. The argument is saved on the variable <u>dumpfile</u> , so that if no argument is provided, a new version of the previous file will be created.
EXIT	<u>transorset</u> returns NIL.

THE REMARK FEATURE

The translation notes are generated by those transformations that are actually executed via an editmacro called REMARK. REMARK takes one argument, the name of a note. When the macro is executed, it saves the appropriate information for the translation notes, and adds one entry to the index of forms. The location that is printed in the index of forms is the editor's location when the REMARK macro is executed.

To write a transformation which makes a new note, one must therefore do two things: define the note, i.e. choose a new name and associate it with the desired text; and call the new note with the REMARK macro, i.e. insert the edit command (REMARK name) in some transformation. The NOTE command, described below, is used to define a new note. The call to the note may be added to a

transformation like any other edit command. Once a note is defined, it may be called from as many different transformations as desired.

The user can also specify a remark with a new text, without bothering to think of a name and perform a separate defining operation, by calling REMARK with more than one argument, e.g. (REMARK text-of-remark). This is interpreted to mean that the arguments are the text. transorset notices all such expressions as they are typed in, and handles naming automatically; a new name is generated³⁷ and defined with the text provided, and the expression itself is edited to be (REMARK generated-name). The following example illustrates the use of REMARK.

```

←TRANSORSET()
+NOTE GREATERP/LESSP (BBN'S GREATERP AND LESSP ONLY [1]
TAKE TWO ARGUMENTS, WHEREAS SRI'S FUNCTIONS TAKE AN
INDEFINITE NUMBER. AT THE PLACES NOTED HERE, THE SRI CODE
USED MORE THAN TWO ARGUMENTS, AND THE USER MUST RECODE.] GREATERP/LESSP
+FN GREATERP
GREATERP
+(IF (IGREATERP (LENGTH (##))3) NIL ((REMARK GREATERP/LESSP] [2]
+FN LESSP
LESSP
+REDO IF [3]
+SHOW
LESSP
[(IF (IGREATERP (LENGTH (##))
3)
NIL
((REMARK GREATERP/LESSP]
LESSP
+FN ASCII
(OLD TRANSFORMATIONS)
ASCII
+(REMARK ALTHOUGH THE SRI FUNCTION ASCII IS IDENTICAL [4]
TO THE BBN FUNCTION CHARACTER, THE USER MUST MAKE SURE THAT
THE CHARACTER BEING CREATED SERVES THE SAME PURPOSE ON BOTH
SYSTEMS, SINCE THE CONTROL CHARACTERS ARE ALL ASSIGNED
DIFFERENTLY.] +SHOW [5]
ASCII
((1 CHARACTER)
(REMARK ASCII:))
ASCII
+NOTE ASCII: [6]
EDIT
*NTH -2
*P
... ASSIGNED DIFFERENTLY.)
*(2 DIFFERENTLY.)
OK
ASCII:
+

```

³⁷ The name generated is the value of currentfn suffixed with a colon, or with a number and a colon.

In this example, the user defines a note named GREATERP/LESSP by using the NOTE command [1], and writes transformations which call this note whenever the sweep encounters a GREATERP or LESSP with more than two arguments [2-3]. Next, the implicit naming feature is used [4] to add a REMARK command to the transformation for ASCII, which has already been partly written. The user realizes he mistyped part of the text, so he uses the SHOW command to find the name chosen for the note [5]. Then he uses the NOTE command on this name, ASCII:, to edit the note [6].

NOTE

First argument is note name and must be a literal atom. If already defined, NOTE edits the old text; otherwise it defines the name, reading the text either from the rest of the input line or from the next line. The text may be given as a line or as a list. Value is name of note.

The text is actually stored.³⁸ as a comment, i.e. a * and %% are added in front when the note is first defined. The text will therefore be lower-cased the first time the user DUMPs (see Section 14).

DELNOTE

Deletes a note completely (although any calls to it remain in the transformations).

CONTROLLING THE SWEEP

transor's sweep searches in print-order until it finds a form for which a transformation exists. The location is marked, and the transformation is executed. The sweep then takes over again, beginning from the marked location, no matter where the last command of the transformation left the editor. User transformations can therefore move around freely to examine the context, without worrying about confusing the translator. However, there are many cases where the user wants his transformation to guide the sweep, usually in order to direct the processing of special forms and FEXPR's. For example, the transformation for QUOTE has only one objective: to tell the sweep to skip over the argument to QUOTE, which is (presumably) not a LISP form. NLAM is an editmacro to permit this.

NLAM

An atomic editmacro which sets a flag which causes the sweep to skip the arguments of the current form when the sweep resumes.

Special forms such as cond, prog, selectq, etc., present a more difficult problem. For example, (COND (A B)) is processed just like (FOO (A B)): i.e. after the transformation for cond finishes, the sweep will locate the "next form," (A B), retrieve the transformation for the function A, if any, and execute it. Therefore, special forms must have transformations that preempt the sweep and direct the translation themselves. The following two atomic editmacros permit such transformations to process their forms, translating or skipping over arbitrary subexpressions as desired.

³⁸ On the global list usernotes.

DOTHIS	Translates the editor's current expression, treating it as a single form.
DOTHESE	Translates the editor's current expression, treating it as a list of forms.

For example, a transformation for setq might be (3 DOTHIS).³⁹ This translates the second argument to a setq without translating the first. For cond, one might write (1 (LPQ NX DOTHESE)), which locates each clause of the COND in turn, and translates it as a list of forms, instead of as a single form.

The user who is starting a completely new set of transformations must begin by writing transformations for all the special forms. To assist him in this and prevent oversights, the file <LISP> SPECIAL.XFORMS contains a set of transformations for LISP special forms, as well as some other transformations which should also be included. The user will probably have to revise these transformations substantially, since they merely perform sweep control for Interlisp, i.e. they make no changes in the object code. They are provided chiefly as a checklist and tutorial device, since these transformations are both the first to be written and the most difficult, especially for users new to the Interlisp editor.

When the sweep mechanism encounters a form which is not a list, or a form car of which is not an atom, it retrieves one of the following special transformations.

NLISTPCOMS	Global value is used as a transformation for any form which is not a list.
------------	--

For example, if the user wished to make sure that all strings were quoted, he might set nlistpcoms to ((IF (STRINGP (##)) ((ORR ((QUOTE))) (MBD QUOTE)))) NIL)).

LAMBDACOMS	Global value is used as a transformation for any form, <u>car</u> of which is not an atom.
------------	--

These variables are initialized by <LISP> SPECIAL.XFORMS and are saved by the DUMP command. nlistpcoms is initially NIL, making it a NOP. lambdacoms is initialized to check first for open LAMBDA expressions, processing them without translation notes unless the expression is badly formed. Any other forms with a non-atomic car are simply treated as lists of forms and are always mentioned in the translation notes. The user can change or add to this algorithm simply by editing or resetting lambdacoms.

This completes the discussion of TRANSOR.

³⁹ Recall that a transformation is a list of edit commands. In this case, there are two commands, 3 and DOTHIS.

+ 24.5 INDEXING AND CROSS REFERENCING FILES

+ MULTIFILEINDEX⁴⁰

- + Many systems built in Interlisp consist of a number of symbolic source files. Finding one's way around in the listings for these various files can be very tedious, even for the implementor of the system. The multifileindex package is an attempt to help users deal with this problem by creating a listing of an entire system or set of files, including an alphabetized table of contents containing entries for each function on any of the files. Information is also included for other entities in the files such as records, blocks, and properties. The function multifileindex implements this mechanism.
- + multifileindex first creates an alphabetized table of contents, indicating the name of the entity, the file that it belongs to, and its type (property, variable (set or saved), record, block, etc.) in a columnar format. If the entity is the name of a function, then the information also includes a unique index in the listing for the function, its function type, and its arguments. The files are then printed with each function being preceded by its indexnumber right-justified on the line. Header information is placed at the top of each page, and the pages are numbered. After the files have been listed, they are (undoably) removed from notlistedfiles.

+ multifileindex[filenamelst;mapfile;newpageflg]

+ filenamelst can either be a list of file names, or an atom. If filenamelst is NIL, multifileindex returns immediately. If it is T, filelst is used. multifileindex requires that the COMS for all the files being listed need to be loaded.⁴¹ If a particular file is not loaded,⁴² then its fileCOMS will not be set. In this case, multifileindex will do a loadvars on the file to load the fileCOMS. (Note that this may have other side effects.)

+ mapfile is the output file. If mapfile is NIL, it defaults to value the of printer (initially LPT:). If newpageflg=T, each function in the listing will be placed on a page by itself.

- + The value of linesperpage determines the number of lines per page, and is initially 58. The value of filclineLength determines the width of the page. The following four parameters affect how the columns are placed. The value of multifileindexcols indicates how the other three are to be interpreted. These other variables are multifileindexnamecol, multifileindexfilecol, and multifileindextypecol (initially 0, 26 and 41, respectively). If multifileindexcols is the atom FLOATCOLS (its initial value), then an attempt is made to fit the columns onto the page in a way that maximizes the amount of space for the type information (the amount of space allocated for the type field must be at least 45% of filclineLength in this case). If multifileindexcols is either T or

+ ⁴⁰ multifileindex was written by J. J. Vittal. It is contained on the file MULTIFILEINDEX.COM.

+ ⁴¹ multifileindex assumes if fileCOMS is set, then all nested or indirect COMS are also set.

+ ⁴² or it was loaded with the SYSLOAD option to load.

`FIXCOLS`, then the value of the other variables are treated as absolute column positions on the page. If `multifileindexcols` is either `NIL` or `FIXFLOATCOLS`, the columns will be floated, but will not be any smaller than the column positions defined by the other variables.

SINGLEFILEINDEX⁴³

`singlefileindex` is a package for giving the user an alphabetical function index on the front of each lisp file listed by Interlisp. This package is similar to the `multifileindex` package described above, except that `singlefileindex` provides a table of contents for functions only, and operates on one file at a time. However, `singlefileindex` is much simpler and faster than `multifileindex` and is useful every time a file is made.

The first page gives the filename, time of creation, and the time of the listing. Following that (on possibly more than one page) are n columns of function names and index numbers, where the index number indicates the function's linear occurrence within the file. The number of columns is determined by the length of the longest function name, as well as by the number of functions in the file as described below. The file is then printed with the filename and page number at the top of every page, and each function is preceded by its index number right-justified on the page.

When the `singlefileindex` package is first loaded, it redefines `listfiles1` (Section 14) so that all files listed by `listfiles` will be listed using `singlefileindex`.⁴⁴ Note that the file being indexed does not have to be loaded, or even noticed in the file package sense.

`singlefileindex[file;outputfile;newpageflg]`

`file` is the lisp source file. `outputfile` is the destination file. If `outputfile=NIL`, then the value of `printer` (initially `LPT:`) is used. `newpageflg=T` means each function will be printed on a new page. The value of `filelinelength` determines the position of the index numbers, as well as the placement of the columns. The value of `linesperpage` (initially 58) determines the number of lines per page.

24.6 DATABASEFNS⁴⁵

`databasefns` is a very small package whose purpose is to make the construction and maintenance of `Masterscope` databases an essentially automatic process. It modifies `makefile`, `load`, and `loadfrom` to behave in the following way:

A database will be maintained automatically for any file (containing functions) whose file name has the property `DATABASE` with value `YES`. Whenever such a file is dumped via `makefile`, `masterscope` will analyse any new or changed functions on the file, and a database for all of the

⁴³ `singlefileindex` was written by M. D. Yonke. It is contained on the file `SINGLEFILEINDEX.COM`.

⁴⁴ with both `outputfile` and `newpageflg` being `NIL`.

⁴⁵ The `databasefns` package was written by R. M. Kaplan. It is contained on the file `DATABASEFNS.COM`.

- + functions on the file will be written on a separate file whose name is of the form `file.DATABASE`.
- + Whenever a file which has a `DATABASE` property with value YES is loaded via `load` or `loadfrom`,
- + then the corresponding `.DATABASE` file, if any, is also loaded. The database will not be dumped
- + or loaded if the value of the `DATABASE` property for the file is NO.⁴⁶
- + If the `DATABASE` property is not YES or NO, then for `makefile`, `load`, and `loadfrom` will ask the user whether he wants automatic database maintenance.⁴⁷ Thus when a file is dumped for the first time, the user will be asked "Do you want a Masterscope Database for this file?". Similarly, if the user loads a file which has an associated database, the user will be asked "load database for <file>?".
- + The above interactions may be controlled via the global variables `savedbflg` and `loaddbflg`. When a file which has neither a YES or NO database property is being dumped, `makefile` will assume (and store) a YES value if the value of `savedbflg` is YES, and a NO value if `savedbflg` is NO. The user will be queried only if `savedbflg` is ASK (its initial value). Similarly, if `loaddbflg` is YES, `load` and `loadfrom` will automatically load an existing `.DATABASE` file for a file which does not have a YES or NO value for its `DATABASE` property. The database will not be loaded if `loaddbflg` is NO, and the user will be interrogated as described above if `loaddbflg` is ASK (its initial value).
- + The user can dump and restore databases explicitly via the following functions:
- + `dumpdb[file]` dumps a database for `file` then sets the `DATABASE` property to YES, so that database maintenance for `file` will subsequently be automatic.
- + `loaddb[file]` loads the file `file.DATABASE` if one exists. After the database is loaded, the `DATABASE` property for `file` is set to YES, so that maintenance will thereafter be automatic.
- + Database files include the date and full filename of the file to which they correspond. `loaddb` will print out a warning message if it loads a database that does not correspond to the in-core version of the file.
- + Note that `loaddb` is the only approved way of loading a database: Attempting to load a database file will cause an error.

+ 24.7 LAMBDATRAN⁴⁸

- + The purpose of this package is to facilitate defining new `lambda` words in such a way that a variety of other system packages will respond to them appropriately. A `lambda` word is a word that can

+ ⁴⁶ The `DATABASE` property is considered to be NO if the file is loaded with `ldflg = SYSLOAD`.

+ ⁴⁷ The user's answer will be stored on the `DATATBASE` property so that he will not be asked again.

+ ⁴⁸ The lambdatran package was written by R. M. Kaplan. It is contained on the file `LAMBDATRAN.COM`.

appear as car of a function definition, like LAMBDA and NLAMBDA. New lambda words are useful because they enable the user to define his own conventions about such things as the interpretation of arguments, and to build in certain defaults about how values are returned. For example, the decl package (page 24.53) defines DLAMBDA as a new lambda word with unconventional arguments such as the following:

```
(DLAMBDA ((A FLOATP) (B FIXP) (RETURNS SMALLP)) (FOO A B))
```

In order for such an expression to be executable and compilable, a mechanism must be provided for translating this expression to an ordinary LAMBDA or NLAMBDA, with the special behavior associated with the arguments built into the function body. The lambdatran package accomplishes this via an appropriate entry on dwinuserforms (see Section 17) that computes the translation.

Besides executing and compiling, Interlisp applies a number of other operations to function definitions (e.g. breaking, advising), many of which depend on the system being able to determine certain properties of the function, such as the names of its arguments, their number, and the type of the function (EXPR, FEXPR, etc.). The lambdatran package also provides new definitions for the functions fntyp, arglst, nargs, and argtype which can be told how to compute properties for the user's lambda-words.

A new lambda-word is defined in the following way:

1. Add the lambda-word itself (e.g. the atom DLAMBDA) to the list lambdasplst. This suppresses attempts to correct the spelling of the lambda-word.
2. Add an entry for the lambda-word to the association-list lambdatrancns, which is a list of elements of the form: (Lambda-word Tranfn Fntyp Arglist), where

lambda-word is the name of the lambda-word (e.g. DLAMBDA).

tranfn is a function of one argument that will be called whenever a real definition is needed for the lambda-word definition. Its argument is the lambda-word definition, and its value should be a conventional LAMBDA or NLAMBDA expression which will become the translation of the lambda-word form. The free variable faultfn is bound to the name of the function in which the lambda-word form appeared (or TYPE-IN if the form was typed in).

fntyp determines the function-type of a definition beginning with lambda-word. It is consulted if the definition does not already have a translation from which the function type may be deduced. If fntyp is one of the atoms EXPR, FEXPR, EXPR*, FEXPR*, then all definitions beginning with lambda-word are assumed to have that type. Otherwise, fntyp is a function of one argument that will be applied to the lambda-word definition. Its value should be one of the above four function types.

arglist determines the argument list of the definition if it has not already been translated (if it has, the arglist is simply the arglist of the translation). It is also a function of one argument, the lambda-word definition, and its value should be the list of arguments for the function (e.g. (A B) in the DLAMBDA example

+ above). If the lambda-word definition is ill-formed and the argument list cannot be computed, the function should return T. If an arglist entry is not provided in the lambdatranfns element, then the argument list defaults to the second element of the definition.

+ As an example, the lambdatranfns entry for DLAMBDA is (DLAMBDA DECL EXPR DLAMARGLIST), where decl and dlamarglist are functions of one argument.

+ Note: if the lambda-word definition has an argument list with argument names appearing either as literal atoms or as the first element of a list, the user should also put the property INFO with value BINDS on the property list of the lambda-word in order to inform dwimify (Section 23) to take notice of the names of the arguments when dwimifying.

+ 24.8 PERMSTATUS⁴⁹

+ The function permstatus defined in this package can be used in conjunction with the whenclose package (Section 14) to make a file "permanently" open in the sense that as much of its status as possible will be restored when a sysout is resumed. This includes its access mode, file-pointer position, bytesize, and any pages mapped in by the pmap package. The desired effect is achieved by saying (WHENCLOSE filename 'STATUS 'PERMSTATUS) after the file has been opened.

+ Note that the permanency of files is not guaranteed in that files may be deleted or renamed, or their contents changed, despite their permanent attribute in some sysout. When restarting a sysout, a warning message will be printed if the file cannot be found or restored. However, permstatus will not be able to detect that the contents of a file have been modified since the sysout was created. Note also that "permanent" files will still be closed by closef, and will not be immune to closecall or to closing on end-of-file errors unless the appropriate whenclose attributes for closeall and EOF are also established.

+ 24.9 WHEREIS⁵⁰

+ This package extends the function whereis (see Section 14) such that, when asked about a given name as a function, whereis will consult not only the commands of files that have been noticed by the file package (Section 14) but also a hashfile database (page 24.47) that associates function names with filenames.

+ whereis[name;type;files;-] behaves exactly like the definition in Section 14 unless type=FNS (or NIL) and files=T. In this case, whereis will consult, in addition to the files on filelst, the hashfile that is the value of whereis.hash (initially <KLISPUSER>WHEREIS.HASH).

+ Note: most system functions call whereis with files=T, so loading this package automatically makes the information contained in the whereis database available throughout the system.

+ ⁴⁹ The permstatus package was written by R. M. Kaplan. It is contained on the file PERMSTATUS.COM.

+ ⁵⁰ The whereis package was written by L. M. Masinter. It is contained on the file WHEREIS.COM.

Information may be added to a whereis hashfile by explicitly calling the following function:

whereisnotice[filegroup;newflg] inserts the information about all of the functions on the files in filegroup into the whereis data base contained on (the value of) whereis.hash. filegroup is given as a filegroup argument to directory (see Section 21), so &, \$, etc. may be used. If newflg=T, a new version of whereis.hash will be created containing the database for the functions specified in filegroup.

Note the whereis package requires the hash package (page 24.47). Loading WHEREIS.COM will also load HASH.COM, if it has not already been loaded.

24.10 CJSYS⁵¹

This package provides assistance to Interlisp-10 users who wish to make direct calls on the operating system (via JSYSes). It also makes the coding of certain common assemble constructions more convenient. The package defines the following functions:

js[jsysname;ac1;ac2;ac3;result] nlambda function. All arguments are evaluated except for jsysname. Like jsys (see Section 21), loads the unboxed values of ac1, ac2, and ac3 into the appropriate registers, and executes the JSYS jsysname. js differs from jsys in that the JSYS may be indicated by its *symbolic* name, not just by its number.⁵² js also generates slightly cleaner code than jsys. js also differs from jsys in that:

(a) if any argument is supplied as NIL, then it is not loaded at all, i.e. the corresponding ac will contain garbage. (jsys loads the ac with 0.)

(b) if result is NIL, then no value is loaded (interpreted, js returns the string "garbage result from JS").

(c) result can be T, meaning return T if the JSYS skips, NIL if not.

Because of these differences, caution must be exercised in turning jsys calls into js calls.

Examples: (JS BIN (OPNJFN FILE) NIL NIL 2) returns the value of AC2 after doing a BIN from the JFN of file. (JS BOUT (OPNFJN FILE) 3) sends a control-C to file. The value of this js call is garbage.

⁵¹ The cjsys package was written by L. M. Masinter. It is contained on the file CJSYS.COM.

⁵² The symbolic JSYS name is looked up on the list jsyses, an a-list of (jsysname jsysnumber #skips). If any entry is not found, then the file STENEX.MAC (or SYS:MONSYMS.MAC for Tops-20) is scanned.

+ **xwd[n1;n2]** returns (**LOGOR (LLSH n1 18) (LOGAND n2 77777Q)**), i.e. the word with n1 in the left half and n2 in the right.

+ **bit[bit#;word]** lambda no-spread. If word is not specified, bit simply returns a number with bit bit# set to 1 and all other bits 0. If word is given, then bit is a predicate that returns T if bit# is set in word. Bits are numbered from left to right.

+ Examples: (BIT 32) is 8 (=10Q), (BIT 32 8) is T.

+ **jsyserror[errorn]** lambda function. Returns the TENEX/TOPS-20 error number for errorn. For example, (JSYSERROR GJFX23) is 600103Q. jsyserror compiles open as a constant.

+ This package also defines the following assemble macros:

+ **(JS jsysname)** can be used in assemble statements instead of (JSYS jsysnumber)

+ **(CV expr)** expands to (CQ (VAG (FIX expr))), which unboxes expr to AC1.

+ **(CV2 expr)** expands to (CQ2 (VAG (FIX expr))), which unboxes expr to AC2, saving AC1.

+ 24.11 SCRATCHLIST

+ This facility helps in writing programs that wish to reuse a scratch list to collect together some result.

+ There are two functions (both of which compile open):

+ **scratchlist[lst;x₁;x₂;...;x_n]** lambda, nospread. scratchlist sets up a context in which the value of lst is used as a "scratch" list. The expressions x₁, x₂, ... x_n are evaluated in turn. During the course of evaluation, any value passed to addtoscratchlist will be saved, reusing cons cells from the value of lst. If the value of lst is not long enough, new cons cells will be added onto its end. If the value of lst is NIL, the entire value of scratchlist will be "new" (i.e. no cons cells will be reused).

+ **addtoscratchlist[value]** For use under calls to scratchlist. value is added on to the end of the value being collected by scratchlist. When scratchlist returns, its value is a list containing all of the things that addtoscratchlist has added.

24.12 NOBOX⁵³

This package contains facilities for subverting the normal manner of dynamically allocating and collecting cons cells, large integer boxes, and floating boxes in Interlisp-10 by using static, compile-time allocation. Storage allocation is controlled by allocating the memory for temporary results (e.g. a list that will be thrown away or a floating number that will not exist outside a local computational context) at compile-time or load-time. This "static" storage will be reused whenever the given line of code is re-executed. Because functions which use these facilities may exhibit bizarre behaviour if they are called recursively or if values escape outside of them, these facilities must be used with extreme caution, and should be reserved for those cases where the normal method of storage allocation and garbage collection is not workable or practical. Note: compiled functions need no run-time support for these facilities, i.e. NOBOX does not have to be loaded to execute compiled code.

CONS CELLS

The function cbox may be used to avoid allocation of cons cells. When run interpreted, cbox is exactly equivalent to the function cons. Compiled, cbox operates like cons, except that the cons cell returned is constructed (once) at compile or load time. New values for car and cdr are smashed into the cell at each execution.

The function lbox performs an analogous role for list. When run interpreted, lbox is exactly equivalent to list. Compiled, the corresponding cons cells are allocated at compile or load time. For example, (LBOX A B C) will cause a 3-element static list to be included with a compiled function's literals. Each time the corresponding compiled code is executed, those three cells will be returned containing the current values of the variables a, b, and c.

lbox allocates as many cells as there are arguments in the corresponding form, i.e. the number of scratch cells is determined at compile time. The iterative statement operator SCRATCHCOLLECT enables avoiding conses when the length of a list is not known at compile-time. SCRATCHCOLLECT is used in iterative statements exactly as COLLECT. Each time it is executed, it reuses the cells that it returned on previous executions, which it remembers as an internal scratch list. The length of this scratch list is always the length of the longest value that was ever returned; new cells are allocated whenever the scratch list runs out, and they are permanently remembered.

The SCRATCHCOLLECT i.s.opr and the function scratchlist (page 24.36) have similar applications. With scratchlist, the user makes explicit the origin of the list getting smashed, while with the SCRATCHCOLLECT i.s.opr, the scratch list is hidden (and there is a different scratch-list for each occurrence of the i.s.opr).

NUMBER BOXES

The functions ibox, fbox, and nbox, and the record declarations IBOX and FBOX are provided to improve the efficiency of arithmetic computations. They permit information to be given to the Interlisp-10 compiler that will inhibit the allocation (and subsequent collection) of number boxes

⁵³ The NOBOX package was written by R. M. Kaplan with assistance from B. A. Sheil and M. Kay. It is contained on the file NOBOX.COM.

- + needed for holding temporary results of numeric computations.⁵⁴ In addition, access time to variable-values that are known to be large integers or floating point numbers is improved.
- + The records IBOX and FBOX essentially describe the structure of large integer and floating point boxes respectively. IBOX consists of a single field, called I, which corresponds to the actual contents of the large integer box. FBOX consists of a single field, called F, which corresponds to the contents of the floating point box. For example, the user can create a large integer box containing a given value and assign it to x by saying (SETQ X (CREATE IBOX I ← form)). Even if the value of form is a *small* integer, the result will be stored in a new, large number box. This seeming inefficiency is important because if *some* values of form might be large, making *all* values large means that the compiler can be told how to treat all references to x without generating run-time tests to discover how to do the unboxing. Thus, wherever the value of x is to be referenced, the user simply writes X:I (or (fetch I of X)). In compiling this expression, the compiler generates a single MOVE instruction without any type-testing whatsoever. The user can reuse that number box by saying X:I←(FOO), which is equivalent to, but much more efficient than, (SETN X (FOO)). In other words, once it is known that x is bound to a large integer, the suffix :I can be used in all number-contexts to inform the compiler of that fact.
- + The facilities described so far do nothing to suppress the creation of unnecessary boxes; indeed, the (CREATE IBOX --) will produce boxes for small numbers that would not be allocated otherwise. The functions (not records) ibox, fbox, and nbox are used to suppress unnecessary boxing of temporaries. Effectively, they cause "constant" or "static" boxes of the appropriate type to be allocated and stored in a function's literals when a function is compiled or loaded. Those boxes can be used (and reused) to hold temporary results.
- + ibox and fbox can be called with 0 or 1 arguments. If no arguments are specified (as opposed to a single argument whose value is NIL), then the value of the function is a large-integer or floating number box which is allocated statically. For example, these might be used to construct an initial binding for a variable into which temporary values will be stored using the :I or :F assignments. For example:

 - + (PROG ((X (IBOX))) (X:I←(FOO)) ...).
 - + If an argument is specified for ibox or fbox, then a static box of the appropriate type will be allocated at compile- or load-time, and the value of the argument will be stored in that box whenever the IBOX statement is executed. For example, suppose the user wanted to set a file pointer to 1 past a given byte position. The expression

 - + (SETFILEPTR FILE (ADD1 POS))
 - + would generate a new number box on each execution for which pos happened to be a large number. That box would be passed into setfileptr and then returned as its value. Since the value is not saved, the box would be thrown away, to be collected later. The expression

 - + (SETFILEPTR FILE (IBOX (ADD1 POS)))
 - + would store the desired position in a constant box, and no allocations would take place.

⁵⁴ In the latter respect, these duplicate some of what setn (Section 13) does, except that they are more convenient to use and are executed with less run-time checking (i.e. setn will never smash random memory locations).

As another example, consider a complicated integer expression whose value must be saved in a variable to be used a little further down in a program:

```
(X←(IPLUS 2000 (ITIMES FOO (IQUOTIENT FUM 5))))  
...  
(Z←(IPLUS X (GETFILEPTR FILE)))
```

The Interlisp-10 compiler is smart enough to suppress the boxing inside the (IPLUS 2000 &) expression, but it will generate a box when it comes to do the setq (\leftarrow). This box can be suppressed by writing

```
(X←(IBOX (IPLUS 2000 (ITIMES FOO (IQUOTIENT FUM 5)))))
```

Furthermore, since it is known that x is bound to a large integer, the z assignment can be speeded up by writing

```
(Z←(IPLUS X:I (GETFILEPTR FILE)))
```

The function fbox behaves the same as ibox, except that it traffics in constant floating boxes. Note that if the argument of ibox is FLOATP, then it will be FIXed; if the argument of fbox is FIXP, it will be FLOATed.

The function nbox is a generic function for copying unknown values into constant number boxes. It allocates two constant boxes, one integer and one floating, and stores the value of its argument in the one compatible with the value's type. nbox is useful if the argument value is a constant number box (but one of unknown type) that needs to be copied (see caution (2) below).

CAUTIONS

There are some dangers in using these facilities. The user of this package should be particularly aware of the following:

(1) The F and I fields aim at efficiency more than validity. This means that they *do not check the type* of the pointer that they smash into. For example, if x is bound to NIL, the expression X:I←Z will clobber car and cdr of NIL! The user must be very careful that the arguments given for replacing do indeed point to cells that unboxed numbers can be smashed into. Note: the DECL package (page 24.53) can be used to generate the replaces, iboxes, fboxes automatically in a safe and efficient way.

(2) obox, lbox, SCRATCHCOLLECT, ibox, and fbox all allocate constant boxes, and those boxes will be reused (i.e. smashed with new values) every time the code containing that function call is executed. If that box is saved in a variable or data-structure (e.g. by a setq) as a way of preserving the value it contains, and then the code is re-executed, the value that was saved will be smashed. Thus, the user must beware of using constant boxes to save information in loops or recursions that can get back to the same statement. In these situations, the values must be copied into other cells, perhaps a constant associated with some other line of code, or into cells allocated in the ordinary way. The user must also be careful about returning a constant box as the value of a function, since the caller might unknowingly save the value and re-invoke the box-returner.

(3) because the constant boxes are allocated only in compiled code, *these functions will work quite differently compiled and interpreted*. Side effects which occur because of inadvertent smashing of shared structures will only occur when running compiled definitions and will not be detectable when running interpreted.

+ 24.13 DATEFORMAT⁵⁵

+ dateformat is a small file (one function) which provides assistance for constructing format bits for
+ the ODTIM JSYS (output date/time) as required by date and gdate. dateformat is given a set of
+ keywords (listed below) and it returns a number suitable as a parameter to date and gdate.
+ dateformat is an NLAMBDA NOSPREAD function. The keywords are given below (usually in pairs)
+ and can be thought of as switches (i.e. turn on or off a particular format feature). The default for
+ each pair is indicated by an "[*]" after the keyword. If no keyword is given for a particular pair,
+ the default is used.⁵⁶ The variable dateformat.keys is a list of the keywords used for spelling
+ correction.

+ The keywords are:

+ DATE [*]	Include the date information.
+ NO.DATE	Don't include the date information.
+ NAME.OF.MONTH [*]	Show the month as the name of month.
+ NUMBER.OF.MONTH	Show the month as the number of the month.
+ MONTH.LONG	If the name of the month was requested, spell it out.
+ MONTH.SHORT [*]	If the name of the month was requested, abbreviate it.
+ YEAR.LONG	Four digit year, e.g. 1978.
+ YEAR.SHORT [*]	Two digit year, e.g. 78.
+ DAY.OF.WEEK	Include the day of the week in the date information.
+ NO.DAY.OF.WEEK [*]	Don't include the day of the week in the date information.
+ DAY.LONG	If the day of the week was included, spell it out.
+ DAY.SHORT [*]	If the day of the week was included, abbreviate it.
+ DASHES [*]	Separate the <day>, <month>, and <year> fields with dashes.
+ SLASHES	Separate the <day>, <month>, and <year> fields with slashes.
+ SPACES	Separate the <day>, <month>, and <year> fields with spaces.
+ USA FORMAT	The order <month> <day> <year>.
+ EUROPE FORMAT [*]	The order <day> <month> <year>.
+ LEADING.SPACES [*]	Keep the <day> field two characters long.
+ NO.LEADING.SPACES	<day> field can be one character for dates earlier than the 10th.
+ TIME [*]	Include the time information.
+ NO.TIME	Don't include the time information.
+ TIME.ZONE	Include the time zone in the time specification.
+ NO.TIME.ZONE [*]	Don't include the time zone in the time specification.

+ 55 The dateformat package was written by M. Yonke. It is contained on the file DATEFORMAT.COM.

+ 56 The variable dateformat.default is the number used as the initial value to work with. Therefore, to switch any of
+ these defaults, set the variable dateformat.default to be the value of a call to dateformat with the appropriate keys.

SECONDS [*]	Include the seconds.	+
NO.SECONDS	Don't include the seconds.	+
CIVILIAN.TIME	12 hour time (with AM or PM).	+
MILITARY.TIME [*]	24 hour time.	+

24.14 EXEC⁵⁷

This file contains a set of lispmacros (Section 22) which resemble features of the TENEX EXEC. It also defines functions that provide certain EXEC capabilities for Interlisp programs, e.g. changing the connected directory, detaching the job, etc.

EXEC LISPXMACROS

DA	prints out current time and date.	+
LD	prints <u>systat</u> information, just like the LD subsystem. Jobs are sorted in inverse order of CPU utilization. ⁵⁸	+
LD username	prints information for that user only.	+
LD ALL	like LD, but includes system jobs.	+
DET	detaches the current job.	+
QU	does a logout(). Does not go on history list.	+
LINK user	mimics the exec link command. If <u>user</u> has multiple jobs logged in, asks which tty to link to. ⁵⁹	+
BR	breaks links.	+

⁵⁷ The EXEC package was written by L. M. Masinter, D. C. Lewis and J. J. Vittal. It is contained on the file EXEC.COM. The Exec package uses the passwords package. Loading EXEC.COM will load PASSWORDS.COM if it has not already been loaded. Note: some of the facilities described below will work correctly only on TENEX systems, others only on TOPS-20. The system will inform the user when he attempts to use a facility not supported by his particular operating system.

⁵⁸ SY and WHE is a synonym for LD.

⁵⁹ TALK is a synonym for LINK.

- + CONN dir pwd connects to the directory dir. If the password pwd is not given and is required, conn will prompt. dir can be abbreviated; if omitted, it defaults to the user's login directory. If pwd is given in command line, it is removed from the history list so that ?? will not print it out. Password prompting is handled by getpassword from the passwords package (page 24.44).

- + NDIR filegroup prints the files in filegroup in a multi-column format.

- + DEL filegroup deletes specified files. Uses directory. Note that if <esc> is specified, *all* files that match will be deleted. This command is undoable.

- + UND filegroup undeletes the specified files (undoably).

- + DELVER filegroup deletes all but 1 version of the filegroup specified. Uses directory (Section 21), so filegroup may utilize any of the options allowed for directory filegroup specifications.

- + EXP dir expunges directory dir.⁶⁰

- + TY file outfile bytesize copies file to outfile, or T if outfile is not given. Assumes that the bytes of file are bytesize bits wide (bytesize=NIL defaults to 7). Suppresses blank lines and control character sequences used to indicate font changes.⁶¹

- + DSK dir days prints out disk allocation and usage for the directory dir using dskstat. Also prints total size of files untouched in days days (90 if days not specified).

- + FI like the EXEC filestat command, prints out status of all currently assigned JFNS for the current job.

- + FI jfn prints information for jfn only.

+ 60 If the user does not have access to dir, a message is printed.

+ 61 SEE is a synonym for TY.

EXEC FUNCTIONS

job#[]	returns the job number for the logged in job.	+
tty#[]	returns the teletype-number of the current job.	+
detach[]	detaches the current job.	+
-detachedp[]	a predicate that returns T if the current program is running detached.	+
linktotty[tty#]	generates a two-way link between the controlling terminal of the user's job and <u>tty#</u> . Returns T if the link was successful, otherwise prints an error message and returns NIL.	+
linktouser[user]	links the controlling terminal to a terminal associated with <u>user</u> . Generates an error if the user is not logged in or not attached. If <u>user</u> has more than one attached job, then a systat of his jobs is printed, and the user is asked to provide the proper tty number for the job. Returns T if successful.	+
breaklinks[]	breaks all links to the user's controlling terminal.	+
cndir[dir;password]	Implements the <u>conn</u> command.	+
/delfile[file]	undoable version of <u>delfile</u> .	+
/undelfile[file]	undeletes a single file (undoably).	+
expunge[dir]	expunges directory <u>dir</u> . On TENEX, <u>dir</u> is ignored, and the connected directory is expunged. On TOPS20, if the user does not have access to <u>dir</u> , a message is printed.	+
copyallbytes[fromfile;tofile;bytesize]	implements the <u>sec</u> command.	+
dskstat[dir;printifover;printsyst;printdel;printold]	prints disk usage statistics for directory <u>dir</u> (either a name or number). <u>printifover=NIL</u> means always print <u>printifover=T</u> means only print if <u>dir</u> is over allocation <u>printifover</u> a number means only print if <u>dir</u> has more than that many pages in use.	+

- + printsys=T means print system disk statistics too.
- + printdel=T means print total size of deleted files for dir (this is slow).
- + printold=T or a number means print total size of files untouched in 90 (or printold) days.

- + memstat[pg1:pgn;fork] prints the status of the memory pages pg1 (0 if pgn1=NIL) to pgn (the last page of memory if NIL) in fork fork. fork is either NIL, meaning the current fork, or a fork handle.

+ 24.15 PASSWORDS⁶²

- + getpassword[directoryname] prompts the user for the password for the given directory. The user's response is not echoed. getpassword remembers the password so that it need not ask again; however, saved information is cleared before sysout, so that the sysout contains no passwords.

+ 24.16 TELNET⁶³

- + This package makes it possible to interact with connections created via the net package (described below) without leaving Interlisp.

- + telnet[connection;type;skt;-] in addition, all typeout is included in the dribble file. It permits connections to ARPANET hosts (a la TELNET). connection may be an instance of a CONNECTION record (as created by makenewconnection, see below). Alternatively, the arguments to makenewconnection may be specified in the call to telnet (i.e. if connection is a litatom, telnet uses makenewconnection[connection;type;skt] for the connection). In any case, telnet returns the connection as an instance of the CONNECTION record, so that it is possible to telnet back.

+ 24.17 FTP⁶⁴

- + The ftp package makes it possible to deal with files at other hosts on the Arpa network almost as if they were files on the user's local machine, i.e. the files can be opened via infile, outfile, openfile.

+ 62 The passwords package was written by L. M. Masinter. It is contained on the file **PASSWORDS.COM**.

+ 63 The telnet package was written by L. M. Masinter. It is contained on the file **TELNET.COM**. Since the telnet package uses the net package, loading **TELNET.COM** will also load **NET.COM** unless it has already been loaded.

+ 64 The ftp package was written by L. M. Masinter. It is contained on the file **FTP.COM**. The ftp package uses the net and passwords packages. Loading **FTP.COM** will load **NET.COM** and **PASSWORDS.COM** if they are not already loaded.

read from and printed to by the ordinary reading and printing functions, and closed in the standard way.

Files on remote hosts are designated by including the host name between curly brackets, {}, at the front of the ordinary file name. Since curly brackets are illegal characters in regular file names, a BAD FILE NAME error is generated. This error is intercepted by an entry on errortypest (see Section 16) which then establishes the appropriate network connections.⁶⁵ For example, infile[BBN-D]<LEWIS>INIT.LISP will open the file <LEWIS>INIT.LISP on the host BBN-D and make it be the primary input file. The user could then say read[] to obtain the first expression on that file. The ftp package extends the functions packfilename, unpackfilename, and filenamefield so that they will associate the curly bracket syntax with the new file field HOST. Thus, packfilename[HOST ; BBND ; NAME ; INIT] will return {BBND}INIT.

Remote files have certain properties that limit how they may be used:

- (1) randaccessp is NIL for such files, and setfileptr may not be applied to them. This means, for example, that functions and variables may not be loaded from such files via loadfns.
- (2) the open bytesize of a remote file may not be changed (e.g. by setfileinfo). This means that Interlisp-10 compiled files may not be loaded from remote hosts.
- (3) The remote host may close the connection spontaneously (e.g. because of a timeout if the file is not referenced for some length of time, or because of a crash). If this happens, the next attempt at reading or writing on the file will generate a FILE DATA ERROR. Note: it is unwise to keep a remote file open for long periods of time.⁶⁶

When the connection for the remote file is first established, a password for the remote machine/directory may be required. The user will be asked to supply one via the passwords package (page 24.44). Alternatively, if the host name has on its property list the property LOGIN with value of the form (name password account), then the indicated name, password, and account will be used to log the user into the remote host.⁶⁷

ftp[host;file;access;user;password;account;bytesize]

opens a network connection to the ftp server at host. If access=INPUT or OUTPUT, ftp works like openfile; value is a literal atom of the form {host}file which can then be used as a file name by all Interlisp input and output functions, e.g. read, print, copybytes, etc.⁶⁸ For example,

⁶⁵ Note: it is fairly expensive to open a network connection as compared with the time to open a local file, e.g. an order of magnitude slower.

⁶⁶ For input files, these limitations may be skirted conveniently in the following way: if a colon appears between the last character of the host name and the right curly bracket (e.g. {BBND:}<LEWIS>INIT.LISP), then the remote file will be copied to a temporary *local* file when it is opened, and all subsequent references will be to that local file.

⁶⁷ If the value is of the form (name NIL account), then getpassword[name] will be used for the password. If the account field is NIL, no account will be supplied to the remote host. If no LOGIN property is supplied, ANONYMOUS will be used as the user name.

⁶⁸ In reality, this "file" is a network connection to the host's ftp server. This "file" has a whenclose attribute (Section 14) associated with it so that when Interlisp closes the file, the correct terminating sequence will be performed.

- + ftp[SU-AI YUMYUM%[P,DOC%] INPUT] will allow the Stanford Restaurant Guide to be read. Note that file must satisfy the file name conventions of the remote host.
- + If access= DIRECTORY, then ftp will print on the terminal the names of all files which match file, e.g. ftp[PARC-MAXC2 <NETLISP>*.SAV DIRECTORY].
- + user, password, and account are used for logging in to the remote host. If not supplied, the values are obtained from the LOGIN property (if any) as described above. bytesize is the byte size in which to open the connection. bytesizes of 7, 8, 16, 32 and 36 are supported. bytesize=NIL defaults to 7.

+ 24.18 NET⁶⁹

- + This package contains functions for establishing ARPANET connections from an Interlisp-10 job.
- + A connection is described by and is an instance of the record CONNECTION. The only fields of interest to the user in this record are IN and OUT, which are guaranteed to be car and cadr, respectively. IN is a file name which can be read from, OUT a file name which can be printed to.
- + makenewconnection[host;type;skt;scratchconn;waitflg]
 - + makes a connection to host. For type=ARPA, host is the name of the host to which the connection is to be made.⁷⁰ For skt=NIL (the normal case), the connection will be to the telnet server of host; connections to other servers can be made by supplying the appropriate value for skt.
 - + The value of makenewconnection is a connection. If waitflg is non-NIL, makenewconnection waits until its request for connection is acknowledged. Otherwise, checkconnection must be called on the result before it is used (this allows additional processing to be done while waiting for the remote host to respond).
 - + If scratchconn is non-NIL, it is a scratch connection which is reused.
- + For example, (MAKENEWCONNECTION 'BBND) makes an ARPA connection to BBND, (MAKENEWCONNECTION 'SU-AI 'ARPA 'FINGER) makes a connection to the Stanford whereis service.
- + closeconnection[connection] Closes the given connection and replaces the IN and OUT fields with NIL.

+ ⁶⁹ The net package was written by L. M. Masinter. It is contained on the file NET.COM.

+ ⁷⁰ Other values of type are supplied when makenewconnection is called from arpauser or arpaserver, described below.

checkconnection[connection]	Checks to make sure that the given connection is still open (e.g. it hasn't been closed remotely). If the connection is valid, <u>connection</u> is returned. If the connection is in an in-between state, i.e. in the process of being opened or closed, <u>checkconnection</u> waits to see what happens before returning. Otherwise the connection is cleaned up (as if a <u>closeconnection</u> were performed) and <u>checkconnection</u> returns NIL.	+ + + + + + +
netserver[arpa#;waitflg]	Initiates a "server" connection. This is a connection which will talk to a "user" connection. If <u>waitflg</u> is non-NIL, waits for a user to connect; if <u>waitflg</u> =NIL, returns immediately (and <u>checkconnection</u> must be called on the connection before the connection is actually used). <u>arpa#</u> defaults to 0.	+ + + + + + +
netuser[host;user;arpa#;waitflg]	Initiates the other half of an arpa connection. <u>arpa#</u> defaults to 0 and must be the same as the argument given the corresponding call to <u>netserver</u> . <u>user</u> must be the USERNUMBER (directory number) under which the server job is logged in.	+ + + + + + +
forceout[connection/file]	Normally, characters sent to the "OUT" of a connection are buffered locally. The function <u>forceout</u> can be used to force partially filled packets of bytes to be sent across the connection. The argument to <u>forceout</u> can either be the <u>connection</u> record or the OUT filename.	+ + + + + + +

24.19 HASH -- A HASH-CODED DICTIONARY FACILITY⁷¹

The hash package is an Interlisp-10 facility that permits information associated with string or atom "keys" to be stored on and retrieved from files. The information (or "values") associated with the keys in a file may be numbers, strings, or arbitrary s-expressions. The associations are maintained by a hashing scheme that minimizes the number of page-maps it takes to access a value from its key.

A hashfile may contain information other than key-value associations. The user may print on the file using ordinary printing functions (e.g. prin1, printdef), and he may also store non-character

⁷¹ The hash package, implemented by R. Kaplan, is an extension of previous versions written by L. Masinter and W. van Melle. It is contained on the file HASH.COM.

+ information (e.g. binary data) formatted to suit his particular applications. This information is
+ stored in regions of the file distinct from the hash index. The hash index can be used to locate
+ non-hash information, if the necessary file addresses are stored as hash values.

+ A hashfile is created by the function createhashfile:

+ createhashfile[file; valuetype; itemlength; #entries]

A new version of file is opened and initialized as a hashfile.
valuetype is one of NUMBER, STRING, EXPR, SMALLEXPRESS, or
SYMBOLTABLE, interpreted as follows:

NUMBER	The values are 24-bit unsigned integers.
STRING	The values are strings with less than 128 characters.
EXPR	The values are arbitrary s-expressions ⁷² .
SMALLEXPRESS	The values are arbitrary s-expressions such that (NCHARS value T HASHFILERDTBL) is less than 128. Storing and retrieving is more efficient than for the more general EXPR <u>valuetype</u> .
SYMBOLTABLE	The values are 24-bit unsigned numbers, as for <u>valuetype</u> NUMBER, except that the numbers are treated as the addresses of "symbols" located on non-hash pages in the file. See the discussion of symbol-tables below.

The other arguments to createhashfile are optional. itemlength is the user's estimate of the average number of characters in the entries he expects to store in the hashfile (= the average key length plus the average number of characters in the values for valuetype STRING or SMALLEXPRESS). #entries is an estimate of the total number of key-value associations he is likely to store. These two arguments determine how many pages in the file will be initially allocated as hash-pages; accurate estimates can reduce the number of times that the file must be rehashed as information is stored in it. If these arguments are not given, reasonable defaults are supplied.

After being initialized, file is left open and createhashfile returns as its value a *hashfile datum*, a handle on the hashfile that may be used as an argument for most of the functions described below.

+ openhashfile[file; access]

Re-opens the previously existing hashfile file. access may be INPUT (or NIL), in which case file is opened for reading only, or BOTH, in which case file is open for both input and output. Causes an error NOT A HASHFILE, if file is not recognized as a hashfile.

If access is BOTH and file is a hashfile open for reading only,

+ 72 The values are stored by printing them in the file with hashfilerdtbl, initially ORIG.

<u>openhashfile</u>	attempts to close it and re-open it for writing. Otherwise, if <u>file</u> designates an already open hashfile, <u>openhashfile</u> is a no-op.	+
	The value of <u>openhashfile</u> is a hashfile datum.	+
hashfilep[x]	Returns <u>x</u> if <u>x</u> is a hashfile datum (i.e., a value returned by <u>createhashfile</u> or <u>openhashfile</u>). If <u>x</u> is NIL, returns <u>syshashfile</u> if it is a hashfile datum. If <u>x</u> is the name of an open hashfile, returns the corresponding hashfile datum. Otherwise, returns NIL.	+
	The following functions require an open hashfile as an argument, i.e. an object for which <u>hashfilep</u> is non-NIL.	+
puthashfile and gethashfile	are the basic functions for storing and retrieving values in an open hashfile:	+
puthashfile[key;value;hashfile]	Puts <u>value</u> in <u>hashfile</u> , indexed under <u>key</u> . If <u>value</u> is NIL, any previous entry for <u>key</u> is deleted.	+
gethashfile[key;hashfile]	Returns the value corresponding to <u>key</u> in <u>hashfile</u> . For files of <u>valuetype</u> STRING, NUMBER, or SYMBOLTABLE, the value returned by <u>gethashfile</u> is temporary in that any subsequent calls to <u>hash</u> or <u>pmap</u> function may smash it. <u>concat</u> or <u>mkatom</u> must be applied if the value is a string, or <u>iplus</u> if it is a number, in order to make the value permanent.	+
hashfileprop[hashfile;prop]	returns the value of the <u>prop</u> property of <u>hashfile</u> . The recognized <u>props</u> and the values returned are:	+
	VALUETYPE one of NUMBER, STRING, etc.	+
	NAME the full name of the file.	+
	ACCESS BOTH if file is open for writing, INPUT if it is read-only.	+
hashfilename[hashfile]	same as hashfileprop[hashfile;NAME] .	+
closchashfile[hashfile]	same as close[hashfileprop[hashfile;NAME]] .	+
The function hashstatus can be used as a STATUS function for the whenclose package (Section 14) to restore the state of a hashfile when a sysout is resumed. If hashstatus is used, the permstatus package (page 24.34) must also be loaded.		+
The functions in the following group operate on all the keys in a hashfile:		+

- + **maphashfile[hashfile;mapfn]** For each entry in hashfile, performs mapfn[key;gethashfile[key;hashfile]]. If mapfn is a function of only one argument, performs mapfn[key] thereby avoiding the call to gethashfile needed to obtain the value. key (also value for STRING, NUMBER, and SYMBOLTABLE files) is temporary, as for gethashfile.
- + **rehashfile[hashfile]** After many insertions and deletions much of the space in a hashfile may be unusable. rehashfile reclaims that space by rehashing all the keys. The information on non-hash pages in the file is not altered or moved (except that the pname-pointers in a SYMBOLTABLE file are updated. See below.).
- + **copyhashfile[hashfile;newname;fn;vtype]** calls createhashfile to open newname as a hashfile, with valuetype, itemlength and #entries determined by examining the open hashfile hashfile. Then maps through all the keys in hashfile, doing the equivalent of:
 - + **puthashfile[key;gethashfile[key;hashfile];newhashfile]** for each key. In essence, copyhashfile copies the hash portion of hashfile to newname.
 - + If fn is given, then it is applied to the successive values of hashfile, the old hashfile, and the new hashfile, and the value returned is used as the value in the new file. In effect,
 - + **puthashfile[key;fn[gethashfile[key;hashfile];hashfile;newhashfile];newhashfile]** is evaluated for each key. Thus, the user can intervene as each key is processed in order to copy information associated with the key that resides on non-hash pages.
 - + For example, an EXPR file could be implemented by printing the full s-expressions in a NUMBER file's printing region (see below) and storing their byte-positions as hash values. Instead of reading an s-expression into internal data structures before writing it out to the new file, a fn could be given that transferred the s-expression to the new file more efficiently, via copybytes. The function would return the byte-position on the new file where the expression ended up.⁷³.
 - + If fn is given, then vtype, if specified, is a temporary valuetype (NUMBER, STRING, etc.) to be used during copying. This permits the user to force the valuetype of both files to one more suited for fn, e.g. SMALLEXPR to STRING or EXPR to NUMBER, as in the example. vtype does not affect the permanent valuetype of either file.

+ 73 Actually, this is the way EXPR files are copied if fn is not specified

hashfilesplst[hashfile] +
 Returns a "generator" for the keys in hashfile that is acceptable as +
 an argument to fixspell. Thus, (FIXSPELL BADWORD 70 +
 (HASHFILESPST hashfile)) will spelling correct a word using the +
 keys in hashfile. +

lookuphashfile[key;value;hashfile;calltype] +
 a generalized entry for inserting and retrieving values; provides +
 certain options not available with gethashfile or puthashfile. +
calltype is one of RETRIEVE, DELETE, REPLACE, INSERT or a +
 list of any combination of them. lookuphashfile looks up key in +
hashfile. If key is found, then if calltype is or contains RETRIEVE, +
 the old value is returned from lookuphashfile; otherwise returns T. +
 If calltype is or contains DELETE, the value associated with key is +
 deleted from the file. If calltype is or contains REPLACE, the old +
 value is replaced with value. +

If key is not found, returns NIL. In addition, if calltype is or +
 contains INSERT, lookuphashfile inserts value as the value +
 associated with key. +

Examples: +

To either return an old value or insert a new value in the file if one does not already exist, +
 perform lookuphashfile[key;newvalue;hashfile;(INSERT RETRIEVE)]. The value returned will be +
 NIL if newvalue was inserted, or the old value if key was found. +

To merely check whether key exists in the file without actually retrieving its value (which may be +
 expensive for the more general valuetypes), perform lookuphashfile[key;NIL;hashfile;NIL] +

The function puthashfile is defined as: +

```
(LAMBDA (KEY VALUE HASHFILE)
  (if VALUE=NIL
      then (LOOKUPHASHFILE KEY NIL HASHFILE 'DELETE)
      else (LOOKUPHASHFILE KEY VALUE HASHFILE '(INSERT REPLACE))
            VALUE]) +
```

And gethashfile is defined as: +

```
(LAMBDA (KEY HASHFILE) (LOOKUPHASHFILE KEY NIL HASHFILE 'RETRIEVE)) +
```

UNSTRUCTURED PAGES AND SYMBOL TABLES

The non-hash information in a hash-file may be formatted as printed character strings or binary +
 data. Printed information resides in a file's "printing region", while binary data is stored on +
 "unstructured pages". +

Unstructured pages in a file are allocated and deallocated by the hash package so that they do not +
 encroach on hash or printing pages. Other than that, the user has complete freedom to map them +
 in via mapin or locate for arbitrary reading and writing. The primitive operations are: +

- + `getpage[hashfile;n]` returns the page number of a free page in hashfile. If n is given, then the user is guaranteed that the page returned is the first of n contiguous pages all of which are free.

- + `delpage[page#;hashfile]` removes page page# from hashfile. page# should be the number of an unstructured page, either a value of getpage or within the block of free pages guaranteed by getpage. The contents of the page in the file are lost, and the page itself becomes available for re-allocation either by getpage or internally as a hash page. If page# happens to be the number of a hash page, the hashing information will be destroyed.

- + Unstructured pages are available on hashfiles so that the user can link hash keys to data in special formats. For example, the user might associate lists of properties with a key by writing the properties on an unstructured page, and then storing the file address of the properties as the value of the key in a NUMBER file. The properties could be retrieved by applying locate to `gethashfile[key;hashfile]`.

- + A SYMBOLTABLE hashfile provides an additional feature that makes it possible to implement arbitrary file-resident symbol processing systems. The user may store the data to be associated with a key on unstructured pages, and he can then link the file address to the key via puthashfile, as described above. The difference between a NUMBER and SYMBOLTABLE file is that for a SYMBOLTABLE, the hash package also stores the reverse link from the file address to the key. This makes it possible to obtain a "print-name" for an address on an unstructured page, via the function getpname:

- + `getpname[fileadr;hashfile]` returns a temporary string containing the characters of the key whose hash value is the 24-bit unsigned fileadr. Causes an error if hashfile is not a SYMBOLTABLE file.

- + The hash package automatically updates the print-name information for the file address if the key is relocated by rehashing, and it destroys the back-link if the value for the key is deleted. A SYMBOLTABLE file imposes one restriction on the way unstructured pages are treated: If a file address is stored as a hash-value for some key, then the right-most 24 bits of the word at that location in the file are reserved for the use of the hash mechanism.⁷⁴ The user must not write into it.

- + With these primitives, a list-processing system with a 24-bit non-resident address space is easy to build. The user is responsible for allocating "atoms" on unstructured pages, and updating the "atom hash table" with puthashfile. The second (and subsequent) words after an atom address may be used to store the atom's "property list", containing other atom addresses, or other addresses interpreted as pointers to "cons" cells. These can also be allocated on unstructured pages. It is a simple matter to implement the equivalent of car, cdr, rplaca, and rplacd.

+ 74 The left-most 12 bits are available and can be used for a number of applications, e.g. to store type-bits.

THE PRINTING REGION

Hashfiles are organized so that it is always permissible to print at the end of the file with ordinary Interlisp output functions. That is, the file is arranged so that the hash and unstructured pages are always located before the end-of-file for sequential reading and writing. This is accomplished by creating the file with the end-of-file some number of free pages past the last hash or unstructured page. When all free pages below the end-of-file have been used, the end-of-file is moved so that there are again a reservoir of free pages before it.

Thus, the printing region may shift as a result of calls to getpage or puthashfile, and the user cannot rely on the output from two different printing operations being located at adjacent positions in the file. The expressions he prints cannot be retrieved by successive calls to standard reading functions. Instead, the user should record the byte position of each printed expression as a hash value or on an unstructured page so that he may use setfileptr to position the file properly. If he does change the file's byte-pointer, he must be sure to reset it to the end-of-file (e.g. setfileptr[file;-1]) before more printing is done.

24.20 THE DECL PACKAGE⁷⁵

INTRODUCTION

The Decl package extends Interlisp to allow the user to declare the types of variables and expressions appearing in functions. It provides a convenient way of constraining the behavior of programs when the generality and flexibility of ordinary Interlisp is either unnecessary, confusing, or inefficient.

The Decl package provides a simple language for declarations, and augments the interpreter and the compiler to guarantee that these declarations are always satisfied. The declarations make programs more readable by indicating the type, and therefore something about the intended usage, of variables and expressions in the code. They facilitate debugging by localizing errors that manifest themselves as type incompatibilities. Finally, the declaration information is available for other purposes: compiler macros can consult the declarations to produce more efficient code; coercions for arguments at user interfaces can be automatically generated; and the declarations will be noticed by the Masterscope function analyzer.

The declarations interpreted by the Decl package are in terms of a set of declaration types called decltypes, each of which specifies a set of acceptable values and also (optionally) other type specific behavior. The Decl package provides a set of facilities for defining decltypes and their relations to each other, including type valued expressions and a comprehensive treatment of union types.

The following description of the Decl package is divided into three parts. First, the syntactic extensions which permit the concise attachment of declarations to program elements are discussed. Second, the mechanisms by which new decltypes can be defined and manipulated are covered. Finally, some additional capabilities based on the availability of declarations are outlined.

⁷⁵ The Decl package was designed and implemented by R. M. Kaplan and B. A. Sheil, with the assistance of W. Teitelman and L. M. Masinter. It is contained on the file DECL.COM. The Decl package requires the LAMBDA TRAN package, which will automatically be loaded with Decl if it is not already present.

+ USING DECLARATIONS IN PROGRAMS

- + Declarations may be attached to the values of arbitrary expressions and to LAMBDA and prog variables throughout (or for part of) their lexical scope. The declarations are attached using constructs that resemble the ordinary Interlisp LAMBDA, prog, and progn, but which also permit the expression of declarations. The following examples illustrate the use of declarations in programs.

- + Consider the factorial function discussed in Section 2:

```
+      [LAMBDA (N)
+        (COND
+          ((EQ N 0) 1)
+          (T (ITIMES N (FACT (SUB1 N]
```

- + Obviously, this function presupposes that n is a number, and the run-time checks in itimes and sub1 will cause an error if this is not so. For instance, FACT(T) will cause an error and print the message NON-NUMERIC ARG T. By defining FACT as a DLAMBDA, the Decl package analog of LAMBDA, this presupposition can be stated directly in the code:

```
+      [DLAMBDA ((N NUMBERP))
+        (COND
+          ((EQ N 0) 1)
+          (T (ITIMES N (FACT (SUB1 N]
```

- + With this definition, FACT(T) will *not* result in a NON-NUMERIC ARG T error when the body of the code is executed. Instead, the numberp declaration will be checked when the function is first entered, and a *declaration fault* will occur. Thus, the message that the user will see will not dwell on the offending value T, but instead give a symbolic indication of what variable and declaration were violated, as follows:

```
+      DECLARATION NOT SATISFIED
+      ((N NUMBERP) BROKEN)
+      :
```

- + The user is left in a break from which the values of variables, e.g. n, can be examined to determine what the problem is.

- + The function fact also makes other presuppositions concerning its argument, n. For example, fact will go into an infinite recursive loop if n is a number less than zero. Although the user could program an explicit check for this unexpected situation, such coding is tedious and tends to obscure the underlying algorithm. Instead, the requirement that n not be negative can be succinctly stated by declaring it to be a subtype of NUMBERP which is restricted to non-negative numbers. This can be done by adding a SATISFIES clause to n's type specification:

```
+      [DLAMBDA ([N NUMBERP (SATISFIES (NOT (MINUSP N))]
+        (COND
+          ((EQ N 0) 1)
+          (T (ITIMES N (FACT (SUB1 N]
```

- + The predicate in the SATISFIES clause will be evaluated after n is bound and found to satisfy numberp, but before the function body is executed. In the event of a declaration fault, the SATISFIES condition will be included in the error message. For example, FACT(-1) would result in:

```
+      DECLARATION NOT SATISFIED
```

```
((N NUMBERP (SATISFIES (NOT (MINUSP N))) BROKEN)
```

The DLAMBDA construct also permits the type of the value that is returned by the function to be declared by means of the pseudo-variable RETURNS. For example, the following definition specifies that fact is to return a positive integer:

```
[DLAMBDA ([N NUMBERP (SATISFIES (NOT (MINUSP N))
    [RETURNS FIXP (SATISFIES (IGREATERP VALUE 0)])
    (COND
        ((EQ N 0) 1)
        (T (ITIMES N (FACT (SUB1 N))
```

After the function body is evaluated, its value is bound to the variable value and the RETURNS declaration is checked. A declaration fault will occur if the value is not satisfactory. This prevents a bad value from propagating to the caller of fact, perhaps causing an error far away from the source of the difficulty.

Declaring a variable causes its value to be checked not only when it is first bound, but also whenever that variable is reset by setq within the DLAMBDA. In other words, the type checking machinery will not allow a declared variable to take on an improper value. An iterative version of the factorial function illustrates this feature in the context of a dprog, the Decl package analog of prog:

```
(DLAMBDA ([N NUMBERP (SATISFIES (NOT (MINUSP N))
    [RETURNS FIXP (SATISFIES (IGREATERP VALUE 0)])
    [DPROG ([TEMP 1 FIXP (SATISFIES (IGREATERP TEMP 0)]
        [RETURNS FIXP (SATISFIES (IGREATERP VALUE 0)])
        LP (COND ((EQ N 0) (RETURN TEMP)))
        (SETQ TEMP (ITIMES N TEMP))
        (SETQ N (SUB1 N))
        (GO LP]
```

dprog declarations are much like DLAMBDA declarations, except that they also allow an initial value for the variable to be specified. In the above example, temp is declared to be a positive integer throughout the computation and n is declared to be non-negative. Thus, a bug which caused an incorrect value to be assigned by one of the setq expressions would cause a declaration failure. Note that the RETURNS declaration for a dprog is also useful in detecting the common bug of omitting an explicit RETURN.

DLAMBDA

The Decl package version of a LAMBDA expression is an expression beginning with the atom DLAMBDA. Such an expression is a function object that may be used in any context where a LAMBDA expression may be used. It resembles a LAMBDA expression except that it permits declaration expressions in its argument list, as illustrated in the examples given earlier. Each element of the argument list of a DLAMBDA may be a literal atom (as in a conventional LAMBDA)

- + or a list of the form (name type . extras).⁷⁶
- + name fulfills the standard function of a parameter, i.e. providing a name to which the value of the corresponding argument will be bound.
- + type is either a Decl package type name or type expression. When the DLAMBDA is entered, its arguments will be evaluated and bound to the corresponding argument names, and then, after *all* the argument names have been bound, the declarations will be checked. The type checking is delayed so that SATISFIES predicates can include references to other variables bound by the same DLAMBDA. For example, one might wish to define a function whose two arguments are not only both required to be of some given type, but are also required to satisfy some relationship (e.g., that one is less than the other).
- + extras allows some additional properties to be attached to a variable. One such property is the accessibility of name outside the current lexical scope. Accessibility specifications include the atoms LOCAL or SPECIAL, which indicate that this variable is to be compiled so that it is either a localvar or a specvar, respectively. This is illustrated by the following example:

```
+ [DLAMBDA ((A LISTP SPECIAL)
+           (B FIXP LOCAL))
+           ...]
```

- + A more informative equivalent to the SPECIAL key word is the USEDIN form, the tail of which can be a list of the other functions which are expected to have access to the variable:⁷⁷

```
+ [DLAMBDA ((A LISTP (USEDIN FOO FIE))
+           (B FIXP LOCAL))
+           ...]
```

- + extras may also include a comment in standard format, so that descriptive information may be given where a variable is bound:

```
+ [DLAMBDA ((A LISTP (USEDIN FOO FIE) (* This is an important variable))
+           (B FIXP LOCAL))
+           ...]
```

- + As mentioned earlier, the value returned by a DLAMBDA can also be declared, by means of the pseudo-variable RETURNS. The RETURNS declaration is just like other DLAMBDA declarations, except (1) in any SATISFIES predicate, the value of the function is referred to by the distinguished name value; and (2) it makes no sense to declare the return value to be LOCAL or SPECIAL.

⁷⁶ Strictly, this would require a declaration with a SATISFIES clause to take the form (N (NUMBERP (SATISFIES --)) --) (page 24.62). However, due to the frequency with which this construction is used, it may be written without the inner set of parentheses, e.g. (N NUMBERP (SATISFIES --) --).

⁷⁷ USEDIN is mainly for documentation purposes, since there is no way for such a restriction to be enforced.

DPROGS

Just as DLAMBDA resembles LAMBDA, dprog is analogous to prog. As for an ordinary prog, a variable binding may be specified as an atom or a list including an initial value form. However, a dprog binding also allows type and extras information to appear following the initial value form. The format for these augmented variable bindings is (name initialvalue type . extras). Note that the only difference between a dprog binding and a DLAMBDA binding is that the second position is interpreted as the initial value for the variable.⁷⁸ The same rules apply for the interpretation of the type information for dprogs as for DLAMBDA, and the same set of optional extras can be used. dprogs may also declare the type of the value they return, by specifying the pseudo-variable RETURNS.

Just as for a DLAMBDA, type tests in a dprog are not asserted until *after* all the variables have been bound, thus permitting predicates to refer to other variables being bound by this dprog. If NIL appears as the initial value for a binding (i.e. the atom NIL actually appears in the code, not simply an expression which evaluates to NIL) the initial type test will be suppressed, but subsequent type tests, e.g. following a setq, will still be performed.

A common construct in Lisp is to bind and initialize a prog variable to the value of a complicated expression in order to avoid recomputing it, and then to use this value in initializing other prog variables, e.g.

```
[PROG ((A expression))
      (RETURN (PROG ((B (... A ...)) (C (... A ...)))
                    ...)]
```

The ugliness of such constructions in conventional Lisp often tempts the programmer to loosen the scoping relationships of the variables by binding them all at a single level and using setq's in the body of the prog to establish the initial values for variables that depend on the initial values of other variables, e.g.

```
[PROG ((A expression) B C)
      (SETQ B (... A ...))
      (SETQ C (... A ...))
      ...]
```

In the Decl package environment, this procedure undermines the protection offered by the type mechanism by encouraging the use of uninitialized variables. Therefore, the dprog offers a syntactic form to encourage more virtuous initialization of its variables. A dprog variable list may be segmented by occurrences of the special atom THEN, which causes the binding of its variables in stages, so that the bindings made in earlier stages can be used in later ones, e.g.

```
[DPROG ((A (LENGTH FOO) FIXP LOCAL)
        THEN (B (SQRT A) FLOATP)
        THEN (C (CONS A B) LISTP))
        ...]
```

Each stage is carried out as a conventional set of dprog bindings (i.e., simultaneously, followed by the appropriate type testing). This layering of the bindings permits one to gradually descend into a

⁷⁸ Thus, if the user wishes to supply a type declaration for a variable, an initial value must be specified.

- + inner scope, binding the local names in a very structured and clean fashion, with initial values
- + type-checked as soon as possible.

+ DECLARATIONS IN ITERATIVE STATEMENTS

- + The clisp iterative statement (Section 23) provides a very useful facility for specifying a variety of progs that follow certain widely used formats. The Decl package allows declarations to be made for the scope of an iterative statement via the declare clisp i.s.opr. declare can appear as an operator anywhere in an iterative statement, followed by a list of declarations, for example:

+ (for J from 1 to 10 declare (J FIXP) do ...)

- + Note that declare declarations do not *create* bindings, but merely provide declarations for existing bindings. For this reason, an initial value cannot be specified and the form of the declaration is the same as that of DLAMBDAs, namely (name type . extras).⁷⁹

+ DECLARING A VARIABLE FOR A RESTRICTED LEXICAL SCOPE

- + The Decl package also permits declaring the type of a variable over some restricted portion of its existence. For example, suppose the variable x is either a fixed or floating number, and a program branches to treat the two cases separately. On one path x is known to be fixed, whereas on the other it is known to be floating. The Decl package dprogn construct can be used in such cases to state the type of the variable along each path. dprogn is exactly like progn, except that the second element of the form is interpreted as a list of DLAMBDA format declarations. These declarations are added to any existing declarations in the containing scope, and the composite declaration⁸⁰ is considered to hold throughout the lexical scope created by the dprogn. Thus, our example becomes:

+ (if (FIXP X) then (DPROGN ((X FIXP)) ... stuff ...) + else (DPROGN ((X FLOATP)) ... otherstuff ...))

- + Like dprog and DLAMBDA, the value of a dprogn may also be declared, using the pseudo-variable RETURNS.

- + dprogn may be used not only to restrict the declarations of local variables, but also to declare variables which are being used freely. For example, if the variable a is used freely inside a function but is known to be fixp, this fact could be noted by enclosing the body of the function in (DPROGN ((A FIXP FREE)) ... body ...). Instead of FREE, the more specific construction (BOUNDDIN function₁ function₂ ...) can be used. This not only states that the variable is used freely but also gives the names of the functions which might have provided this binding.⁸¹

+ ⁷⁹ Note that variables bound *outside* of the scope of the iterative statement, i.e. a variable used freely in the i.s., can also be declared using this construction. Such a declaration will only be in effect for the scope of the iterative statement.

+ ⁸⁰ created using the ALLOF type expression, page 24.62.

+ ⁸¹ Like USEDIN declarations, FREE and BOUNDDIN declarations cannot be checked, and are provided for documentation purposes only.

Since the dprogn form introduces another level of parenthesization, which results in the enclosed forms being prettyprinted indented, the Decl package also permits such declarations to be attached to their enclosing DLAMBDA or dprog scopes by placing a DECL expression, e.g. (DECL (A FIXP (BOUNDIN FUM))), before the first executable form in that scope. Like dprogn's, DECL declarations use DLAMBDA format.

DECLARING THE VALUES OF EXPRESSIONS

The Decl package allows the value of an arbitrary form to be declared with the Decl construct THE. A THE expression is of the form (THE type . forms), e.g. (THE FIXP (FOO X)). forms are evaluated in order, and the value of the *last* one is checked to see if it satisfies type, a type name or type expression. If so, its value is returned, otherwise a declaration fault occurs.

ASSERTIONS

The Decl package also allows for checking that an arbitrary predicate holds at a particular point in a program's execution, e.g. a condition that must hold at function entry but not throughout its execution. Such predicates can be checked using an expression of the form (ASSERT form₁ form₂ ...), in which each form_i is either a list (which will be evaluated) or a variable (whose declaration will be checked). Unless all elements of the ASSERT form are satisfied, a declaration fault will take place.

ASSERTing a variable provides a convenient way of verifying that the value of the variable has not been improperly changed by a lower function. Although a similar effect could be achieved for predicates by explicit checks of the form (OR predicate (SHOULDNT)), ASSERT also provides the ability both to check that a variable's declaration is currently satisfied and to remove its checks at compile time without source code modification (see page 24.60).

USING TYPE EXPRESSIONS AS PREDICATES

The Decl package extends the Record package TYPE? construct (Section 23) so that it accepts decltypes, as well as record names, e.g. (TYPE? (FIXP (SATISFIES (ILESSP VALUE 0)))) expr). Thus, a TYPE? expression is exactly the same as a THE expression except that, rather than causing a declaration fault, TYPE? is a predicate which determines whether or not the value satisfies the given type.

ENFORCEMENT

The Decl package is a "soft" typing system - that is, the data objects themselves are not inherently typed. Consequently, declarations can only be enforced *within the lexical scope in which the declaration takes place*, and then only in certain contexts. In general, changes to a variable's value such as those resulting from side effects to embedded structure (e.g., rplaca, setn, etc.) or free variable references from outside the scope of the declaration cannot be, and therefore are not, enforced.

Declarations *are* enforced i.e. checked, in three different situations: when a declared variable is bound to some value or rebound with setq or setqq, when a declared expression is evaluated, and when an ASSERT expression is evaluated. In a binding context, the type check takes place *after* the binding, including any user-defined behavior specified by the type's binding function. Any

- + failure of the declarations causes a break to occur and an informative message to be printed. In that break, the name to which the declaration is attached (or value if no name is available) will be bound to the offending value. Thus, in the FACT(T) example on page 24.54, n would be bound to T. The problem can be repaired either by returning an acceptable value from the break via the RETURN command, or by assigning an acceptable value to the offending name and returning from the break via an OK or GO command. The unsatisfied declaration will be reasserted when the computation is continued, so an unacceptable value will be detected.⁸²
- + The automatic enforcement of type declarations is a very flexible and powerful aid to program development. It does, however, exact a considerable run-time cost because of all the checking involved.⁸³ As a result, it is usually desirable to remove the declaration enforcement code when the system is believed to be bug-free and performance becomes more central. Setting the value of the variable compileignoredcl to T (initially NIL) instructs the compiler not to insert declaration enforcement tests in the compiled code. More selective removal can be achieved by setting compileignoredcl to a list of function names. Any function whose name is found on this list is compiled without declaration enforcement. Finally, declaration enforcement may be suppressed selectively by file using the IGNOREDECL file package command. If this appears in a file's file commands, it redefines the value of compileignoredcl to cdr of the IGNOREDECL command for the compilation of this file only.

+ DECLTYPES

- + A Decl package type, or decltype, specifies a subset of data values to which values of this type are restricted. For example, a "positive number" type might be defined to include only those values that are numbers and greater than zero. A type may also specify how certain operations, such as assignment or binding (see page 24.64), are to be performed on variables declared to be of this type.
- + The inclusion relations among the sets of values which satisfy the different types define a natural partial ordering on types, bound by the universal type ANY (which all values satisfy) and the empty type NONE (which no value satisfies). Each type has one or more *supertypes* (each type has at least ANY as a supertype) and one or more *subtypes* (each type has at least NONE as a subtype). This structure is important to the user of Decl as it provides the framework in which new types are defined. Typically, much of the definition of a new type is defaulted, rather than specified explicitly. The definition will be completed by inheriting attributes which are shared by all its immediate supertypes.
- + An initial set of decltypes which defines the Interlisp built-in datatypes and a few other commonly used types is provided. Thereafter, new decltypes are created in terms of existing ones using the type expressions described below. For conciseness, such new types can be associated with literal atoms using the function decltype (page 24.63).

+ 82 With this exception, assignments to variables from within the break are not considered to be in the scope of the declarations that were in effect when the break took place, and so are not checked.

+ 83 Factors of two to ten in running speed are not uncommon, especially where low level, frequently used functions employ type declarations.

PREDEFINED TYPES

Some commonly used types, such as the Interlisp built-in data types, are already defined when the Decl package is loaded. These types, indented to show subtype-supertype relations, are:

ANY							
ARRAYP	ATOM	FUNCTION	LST ⁸⁴	STACKP	STRINGP		+
HARRAYP	LITATOM		ALIST ⁸⁵				+
READTABLEP	NIL		LISTP				+
	NUMBERP						+
	FIXP						+
	LARGEPP						+
	SMALLP						+
	FLOATP						+
			NONE				+

Note that the definition of LST causes NIL to have multiple supertypes, i.e. LITATOM and LST, reflecting the duality of NIL as an atom and a (degenerate) list.

In addition, declarations made using the Record package (Section 23) also define types which are attached as subtypes to an appropriate existing type (e.g., a typerecord declaration defines a subtype of LISTP, a datatype declaration a subtype of ANY, etc.) and may be used directly in declaration contexts.

TYPE EXPRESSIONS

Type expressions provide convenient ways for defining new types in terms of modifications to, or compositions of one or more, existing types.

(MEMQ value ₁ ... value _n)	specifies a type whose values can be any one of the fixed set of elements {value ₁ ... value _n }. For example, the status of a device might be represented by a datum restricted to the values BUSY and FREE. Such a "device status" type could be defined via (MEMQ BUSY FREE). The new type will be a subtype of the narrowest type which all of the alternatives satisfy (e.g., the "device status" type would be a subtype of LITATOM). The membership test uses <u>eq</u> if this supertype is LITATOM; <u>equal</u> otherwise. Thus, lists, floating point numbers, etc., can be included in the set of alternatives.	+
---	---	---

(ONEOF type ₁ ... type _n)	specifies a type which is the union of two or more other types. For	+
--	---	---

⁸⁴ LST is defined as either LISTP or NIL, i.e. a list or NIL. The name LST is used, because the name LIST is treated specially by clisp.

⁸⁵ ALIST is defined as either NIL, or a list of elements each of which is of type LISTP.

+ example, the notion of a possibly degenerate list is something that
+ is either LISTP or NIL. Such a type can be (and the built in type
+ LST in fact is) defined simply as (ONEOF NIL LISTP). A union
+ data type becomes a supertype of all of the alternative types
+ specified in the ONEOF expression, and a subtype of their lowest
+ common supertype. The type properties of a union type are taken
+ from its alternative types if they all agree, otherwise from the
+ supertype.

+ (ALLOF type₁ ... type_n)
+ specifies a type which is the intersection of two or more other
+ types. For example, a variable may be required to satisfy both
+ FIXP and also some type which is defined as
+ (NUMBERP (SATISFIES predicate)). The latter type will admit
+ numbers that are not FIXP, i.e. floating point numbers; the former
+ does not include predicate. Both restrictions can be obtained by
+ using the type
+ (ALLOF (NUMBERP (SATISFIES predicate)) FIXP).⁸⁶

+ (aggregate OF element)
+ specifies a type which is an aggregate of values of some other type
+ (e.g., list of numbers, array of strings, etc.). aggregate must be a
+ type which provides an EVERYFN property (page 24.64). The
+ EVERYFN is used to apply an arbitrary function to each of the
+ elements of a datum of the aggregate type, and check whether the
+ result is non-NIL for each element. element may be any type
+ expression. For example, the type "list of either strings or atoms"
+ can be defined as (LISTP OF (ONEOF STRINGP ATOM)). The
+ type test for the new type will consist of applying the type test for
+ element to each element of the aggregate type using the EVERYFN
+ property. The new type will be a subtype of its aggregate type.⁸⁷

+ (type (SATISFIES form₁ ... form_n))
+ specifies a type whose values are a subset of the values of an
+ existing type. The type test for the new type will first check that
+ the base type is satisfied, i.e. that the object is a member of type,
+ and then evaluate form₁ ... form_n. If each form returns a non-NIL
+ value, the type is satisfied.

+ The value that is being tested may be referred to in form₁ ... form_n
+ by either (a) the variable name if the type expression appears in a
+ binding context such as DLAMBDA or dprog (b) the distinguished
+ atom elt for a SATISFIES clause on the elements of an aggregate
+ type, or (c) the distinguished atom value, when the type expression
+ is used in a context where no name is available (e.g., a RETURNS
+ declaration). For example, one might declare the program variable

+ 86 When a value is tested, the component type tests are applied from left to right.

+ 87 The built-in aggregate types are arrayp, listp, lst, and stringp (and their subtypes).

a to be a negative integer via (FIXP (SATISFIES (MINUSP A))), or declare the value of a DLAMBDA to be of type ((ONEOF FIXP FLOATP) (SATISFIES (GREATERP VALUE 25))). Note that more than one SATISFIES clauses may appear in a single type expression attached to different alternatives in a ONEOF type expression, or attached to both the elements and the overall structure of an aggregate. For example,

```
[LISTP OF [FIXP (SATISFIES (ILEQ ELT (CAR VALUE)
                               (SATISFIES (ILESSP (LENGTH VALUE) 7))]
```

specifies a list of less than 7 integers each of which is no greater than the first element of the list.

(SHARED type)

Specifies a subtype of type with default binding behavior, i.e. the binding function (see page 24.64), if any, will be suppressed.⁸⁸ For example, if the type FLOATP were redefined so that DLAMBDA and dprog bindings of variables that were declared to be FLOATP copied their initial values (e.g., to allow setns to be free of side effects), then variables declared (SHARED FLOATP) would be initialized in the normal fashion, without copying their initial values.

NAMED TYPES

Although type expressions can be used in any declaration context, it is often desirable to save the definition of a new type if it is to be used frequently, or if a more complex specification of its behavior is to be given than is convenient in an expression. The ability to define a named type is provided by the function decltype.

decltype[typename; type; prop₁; val₁; ...; prop_n; val_n]

Nilambda, nospread function. typename is a literal atom, type is either the name of an existing type or a type expression, and prop₁, val₁, ..., prop_n, val_n is a specification (in property list format) of other attributes of the type. decltype derives a type from type, associates it with typename, and then defines any properties specified with the values given.

The following properties are interpreted by the Decl package.⁸⁹ Each of these properties can have as its value either a function name or a LAMBDA expression.

⁸⁸ As no predefined type has a binding function, this is of no concern until the user defines or redefines a type to have a binding function.

⁸⁹ Actually, any property can be attached to a type, and will be available for use by user functions via the function getdecltypeprop, described below.

+ TESTFN fn will be used by the Decl package to test whether a given value satisfies this type. The type is considered satisfied if fn applied to the item is non-NIL. For example, one might define the type INTEGER with TESTFN fixp.⁹⁰

+ EVERYFN fn specifies a mapping function which can apply a functional argument to each "element" of an instance of this type, and which will return NIL unless the result of every such application was non-NIL. fn must be a function of two arguments: the aggregate and the function to be applied. For example, the EVERYFN for the built-in type LISTP is every. As described on page 24.62, the Decl package uses the EVERYFN property of the aggregate type to construct a type test for aggregate type expressions. In fact, it is the presence of an EVERYFN property which allows a type to be used as an aggregate type.⁹¹⁹²

+ BINDFN fn is used to compute from the initial value supplied for a DLAMBDA or dprog variable of this type, the value to which the variable will actually be initialized. fn must be a function of one argument which will be applied to the initial value,⁹³ and which should produce another value which is to be used to make the binding. For example, a BINDFN could be used to bind variables of some type so that new bindings are copies of the initial value. Thus, if FLOATP were given the BINDFN fplus, any variable declared FLOATP would be initialized with a new floating box, rather than sharing with that of the original initial value.⁹⁴

+ SETFN fn is used for performing a setq or setqq of variables of this type. fn is a function of two arguments, the name of the variable, and its new value. A SETFN is typically used to avoid the allocation of storage for intermediate results. Note that the SETFN is *not* the mechanism for the enforcement of type compatibility, which is

+ 90 Typically, the TESTFN for a type is derived from its type expression, rather than specified explicitly. The ability to specify the TESTFN is provided for those cases where a predicate is available that is much more efficient than that which would be derived from the type expression. For example, the type SMALLP is defined to have the function smallp as its TESTFN, rather than (LAMBDA (DATUM) (AND (NUMBERP DATUM) (FIXP DATUM) (SMALLP DATUM))) as would be derived from the subtype structure.

+ 91 Note that a type's EVERYFN is *not* used in type tests for that type, but only in type tests for types defined by OF expressions (page 24.62) which used this type as the aggregate type. For example, every is not used in determining whether some value satisfies the type LISTP.

+ 92 The Decl package never applies the EVERYFN of a type to a value without first verifying that the value satisfies that type.

+ 93 For a dprog binding, fn will be applied to no arguments if the initial value is lexically NIL.

+ 94 The BINDFN, if any, associated with a type may be suppressed in a declaration context by creating a subtype with the type expression operator SHARED, as described on page 24.63.

checked *after* the assignment has taken place. Also note that not all functions which can change values are affected: in particular, set and setn are not.

Manipulating named types

decltype is a file package type (Section 14). Thus all of the operations relating to file package types, e.g. getdef, putdef, editdef, deldef,⁹⁵ showdef, etc., can be performed on decltypes.

The file package command, DECLTYPES, is provided to dump named decltypes symbolically. They will be written as a series of decltype forms which will specify only those fields which differ from the corresponding field of their supertype(s). If the type depends on any unnamed types, those types will be dumped (as a compound type expression), continuing up the supertype chain until a named type is found. Care should be exercised to ensure that enough of the named type context is dumped to allow the type definition to remain meaningful.

The functions getdecltypeprop and setdecltypeprop, defined analogously to the property list functions for atoms, allow the manipulation of the properties of named types. Setting a property to NIL with setdecltypeprop removes it from the type.

RELATIONS BETWEEN TYPES

The notion of equivalence of two types is not well defined. However, type equivalence is rarely of interest. What is of interest is type *inclusion*, i.e. whether one type is a supertype or subtype of another. The predicate covers can be used to determine whether the values of one type include those of another.

covers[hi;lo]

is T if hi can be found on some (possibly empty) supertype chain of lo; else NIL. Thus, covers[FIXP;declof[4]]=T, even though the decltype of 4 is SMALLP, not FIXP. The extremal cases are the obvious identities: covers[ANY;anytype] = covers[anytype;NONE] = covers[x;x] for any type x = T.

covers allows declaration based transformations of a form which depend on elements of the form being of a certain type to express their applicability conditions in terms of the weakest type to which they apply, without explicit concern for other types which may be subtypes of it. For example, if a particular transformation is to be applied whenever an element is of type NUMBERP, the program which applies that transformation does not have to check whether the element is of type SMALLP, LARGEP, FIXP, FLOATP, etc., but can simply ask whether NUMBERP covers the type of that element.

The elementary relations among the types, out of which arbitrary traversals of the type space can be constructed, are made available via:

⁹⁵ Deleting a named type could possibly invalidate other type definitions that have the named type as a subtype or supertype. Consequently, the deleted type is simply unnamed and left in the type space as long as it is needed.

- + **subtypes[type]** returns the list of types which are *immediate* subtypes of type.
- + **supertypes[type]** returns the list of types which are *immediate* supertypes of type.

+ THE DECLARATION DATABASE

- + One of the primary uses of type declarations is to provide information that other systems can use to interpret or optimize code. For example, one might choose to write all arithmetic operations in terms of general functions like plus and times and then use variable declarations to substitute more efficient, special purpose code at compile time based on the types of the operands. To this end, a data base of declarations is made available by the Decl package to support these operations.

- + **declof[form]** returns the type of form in the current declaration context.⁹⁶ If form is an atom, declof will look up that atom directly in its database of current declarations. Otherwise, declof will look on the property list of car[form] for a DECLOF property, as described below. If there is no DECLOF property, declof will check if car[form] is one of a large set of functions of known result type (e.g., the arithmetic functions). Failing that, if car[form] has a MACRO property, declof will apply itself to the result of expanding (with expandmacro, Section 18) the macro definition. Finally, if form is a Lisp program element that declof "understands" (e.g., a cond, prog, selectq, etc.), declof applies itself recursively to the part(s) of the contained form which will be returned as value.

- + **DECLOF val** allows the specification of the type of the values returned by a particular function. The value of the DECLOF property can be either a type, i.e. a type name or a type expression, or a list of the form (FUNCTION fn), where fn is a function object. fn will be applied (by declof) to the form whose car has this DECLOF property on its property list. The value of this function application will then be considered to be the type of the form.

- + As an example of how declarations can be used to automatically generate more efficient code, consider an arithmetic package. Declarations of numeric variables could be used to guide code generation to avoid the inefficiencies of Interlisp's handling of arithmetic values. Not only could the generic arithmetic functions be automatically specialized, as suggested above, but by redefining the BINDFN and the SETFN properties for the types FLOATP and LARGEFP to re-use storage in the

⁹⁶ The "current declaration context" is defined by the environment at the time that declof is called. Code reading systems, such as the compiler and the interpreter, keep track of the lexical scope within which they are currently operating, in particular, which declarations are currently in effect. Note that (currently) declof does *not* have access to any global data base of declarations. For example, declof does not have information available about the types of the arguments of, or the value returned by, a particular function, unless it is currently "inside" of that function. However, the DECLOF property (described below) can be used to inform declof of the type of the value returned by a particular function.

appropriate contexts (i.e., when the new value can be determined to be of the appropriate type), +
tremendous economies could be realized by not allocating storage to intermediate results which +
must later be reclaimed by the garbage collector. The Decl package has been used as the basis for +
several such code optimizing systems.

DECLARATIONS AND MASTERSCOPE

The Decl package notifies Masterscope about type declarations and defines a new Masterscope +
relation, TYPE, which depends on declarations. Thus, the user can ask questions such as +

WHO USES MUMBLE AS A TYPE?
DOES FOO USE FIXP AS A TYPE?

and so on.

Index for Section 24

	Page Numbers
ADDTOSCRATCHLIST[VALUE]	24.36
ANY (in Decl package)	24.60
ASSERT (in Decl package)	24.59
assignments (in pattern match compiler)	24.5
BIT[BIT#;WORD]	24.36
BOUNDDIN (in Decl package)	24.58
BOXED (edita parameter)	24.12
BR (exec command)	24.41
BREAKLINKS[]	24.43
carriage-return (edita command)	24.11, 10
CBOX[X;Y]	24.37
CHECKCONNECTION[CONNECTION]	24.47
CIRCLMAKER[LIST]	24.19
CIRCLPRINT[LIST;PRINTFLG;RLKNT]	24.18, 17
cjsys package	24.35-36
CLISPARRAY (clisp variable/parameter)	24.2
CLOSECONNECTION[CONNECTION]	24.46
CLOSEHASHFILE[HASHFILE]	24.49
CNDIR[DIR;PASSWORD]	24.43
COMPILEIGNOREDECL (Decl package parameter)	24.60
CONN (exec command)	24.42
control-E (typed to edita)	24.10
COPYALLBYTES[FROMFILE;TOFILE;BYTESIZE]	24.43
COPYHASHFILE[HASHFILE;NEWNAME;FN;VTYPE]	24.50
COREVAL (property name)	24.10-11
COVERS[HI;LO]	24.65
CREATEHASHFILE[FILE;VALUETYPE;ITEMLENGTH; #ENTRIES]	24.48
curly brackets (use with ftp package)	24.45
current declaration context	24.66
CURRENTFN (transor variable)	24.24
CV (assemble macro)	24.36
CV2 (assemble macro)	24.36
DA (exec command)	24.41
databasefns package	24.31-32
DATEFORMAT[KEYWORD1;KEYWORD2;...;KEYWORDn]	24.40
dateformat package	24.40-41
DDT[] SUBR	24.9
Decl package	24.53-68
DECL (in Decl package)	24.59
declaration fault (in Decl package)	24.54
DECLARATION NOT SATISFIED (error message)	24.54
declarations (in pattern match compiler)	24.6
DECLOF[FORM]	24.66
DECLOF (property name)	24.66, 66
DECLTYPE[TYPENAME;TYPEEXPRESSION;PROP1;VAL1;...; PROPn;VALn]	24.63
DECLTYPES (file package command)	24.65
decltypes (in Decl package)	24.53
DEL (exec command)	24.42
DELNOTE (transor command)	24.28
DELPAGE[PAGE#;HASHFILE]	24.52
DELVER (exec command)	24.42
DET (exec command)	24.41
DETACH[]	24.43
DETACHEDP[]	24.43

Page
Numbers

DLAMBDA (in Decl package)	24.55,54
DOTHESE (transor command)	24.29
DOTHIS (transor command)	24.29
DPROG (in Decl package)	24.55,57
DSK (exec command)	24.42
DSKSTAT[DIR;PRINTIFOVER;PRINTSYS;PRINTDEL; PRINTOLD]	24.43
DUMP (transorset command)	24.26
DUMPDB[FILE]	24.32
DWIMUSERFORMS (dwim variable/parameter)	24.33
EDIT (transorset command)	24.26
EDITA[EDITARRY;COMS]	24.8-15
editing arrays	24.8-15
editing compiled code	24.8-15
element patterns (in pattern match compiler)	24.2-4
ERASE (transorset command)	24.26
exec package	24.41-44
EXIT (transorset command)	24.26
EXP (exec command)	24.42
EXPUNGE[DIR]	24.43
FBOX[N]	24.38
FBOX (record declaration)	24.38
FI (exec command)	24.42
FILE DATA ERROR (error message)	24.45
FILE (edita parameter)	24.13
FN (transorset command)	24.25
FORCEOUT[CONNECTION/FILE]	24.47
FREE (in Decl package)	24.58
FTP[HOST;FILE;ACCESS;USER;PASSWORD;ACCOUNT; BYTESIZE]	24.45
ftp package	24.44
generators for spelling correction	24.51
GETDECLTYPEPROP[TYPE;PROP]	24.65
GETHASHFILE[KEY;HASHFILE]	24.49
GETPAGE[HASHFILE;N]	24.52
GETPASSWORD[DIRECTORYNAME]	24.44
GETPNAME[FILEADR;HASHFILE]	24.52
hash package	24.47-53
HASHFILENAME[HASHFILE]	24.49
HASHFILEP[X]	24.49
HASHFILEPROP[HASHFILE;PROP]	24.49
HASHFILESPLST[HASHFILE]	24.51
HOST (as a file name field)	24.45
IBOX[N]	24.38
IBOX (record declaration)	24.38
IGNOREDECL (file package command)	24.60
JOB#[]	24.43
JS[JSYSNAME;AC1;AC2;AC3;RESULT]	24.35
JS (assemble macro)	24.36
JSYSERROR[ERRORN]	24.36
JSYSES (system variable/parameter)	24.35
LAMBDAACOMS (transor parameter)	24.29
LAMBDAASPLST (dwim variable/parameter)	24.33
Lambdatran package	24.32-34
LAMBDATRANFNS (lambdatran parameter)	24.33
LBOX[X1;X2;...;Xn]	24.37
LD (exec command)	24.41

	Page Numbers
line-feed (edita command)	24.12
LINK (exec command)	24.41
LINKTOTTY[TTY#]	24.43
LINKTOUSER[USER]	24.43
listp checks (in pattern match compiler)	24.2
LITS (edita parameter)	24.12
LOADDB[FILE]	24.32
LOADDBFLG (system variable/parameter)	24.32
LOCAL (in Decl package)	24.56
LOGIN (property name)	24.45
LOOKUPHASHFILE[KEY;VALUE;HASHFILE;CALLTYPE]	24.51
machine instructions	24.10
MAKENEWCONNECTION[HOST;TYPE;SKT;SCRATCHCONN; WAITFLG]	24.46
MAPHASHFILE[HASHFILE;MAPFN]	24.50
MASK (edita parameter)	24.14
MATCH (use in pattern match in clisp)	24.2
MEMSTAT[PG1;PGN;FORK]	24.44
MULTIFILEINDEX[FILENAMELST;MAPFILE;NEWPAGEFLG] ..	24.30
multifileindex package	24.30-31
NBOX[N]	24.38-39
NDIR (exec command)	24.42
net package	24.46-47
NETSERVER[ARPA#;WAITFLG]	24.47
NETUSER[HOST;USER;PUP#;WAITFLG]	24.47
NLAM (transor command)	24.28
NLISTPCOMS (transor parameter)	24.29
nobox package	24.37-40
NONE (in Decl package)	24.60
NOT A HASHFILE (error message)	24.48
NOTE (transor command)	24.28,26
OK (edita command)	24.12
OPD (property name)	24.10
OPENHASHFILE[FILE;ACCESS]	24.48
ORG (edita parameter)	24.11
passwords package	24.44
PATLISTPCHECK (in pattern match compiler)	24.2
pattern match compiler	24.1-8
PATVARDEFAULT (in pattern match compiler)	24.3,5-6
permanent files	24.34
permstatus package	24.34
place-markers (in pattern match compiler)	24.6
PRESCAN[FILE;CHARLST;PRESCANFN]	24.22
printing circular lists	24.15-21
PRINTL[ITEM;DEPTH;LMARG;RMARG;FILE]	24.20
PRNTL (prog. asst. command)	24.20
PUTHASHFILE[KEY;VALUE;HASHFILE]	24.49
QU (exec command)	24.41
reconstruction (in pattern match compiler)	24.7
REHASHFILE[HASHFILE]	24.50
REMARK (transor command)	24.26
replacements (in pattern match compiler)	24.6
RETURNS (in Decl package)	24.56,55,58
SATISFIES (in Decl package)	24.54
SAVEDBFLG (system variable/parameter)	24.32
SCRATCHCOLLECT (clisp iterative statement operator)	24.37

Page
Numbers

SCRATCHLIST[LST;X1;X2;...;Xn]	24.36
scratchlist package	24.36
segment patterns (in pattern match compiler)	24.4-5
SETDECLTYPEPROP[TYPE;PROP;VAL]	24.65
SHOW (transorset command)	24.26
SINGLEFILEINDEX[FILE;OUTPUTFILE;NEWPAGEFLG]	24.31
singlefileindex package	24.31
SPECIAL (in Decl package)	24.56
spelling correction on hash files	24.51
SUBTYPES[TYPE]	24.66
subtypes (in Decl package)	24.60
SUPERTYPES[TYPE]	24.66
supertypes (in Decl package)	24.60
SYMLST (edita parameter)	24.13
tab (edita command)	24.11
TELNET[CONNECTION;TYPE;SKT;BUTTONCHAR]	24.44
telnet package	24.44
TEST (transorset command)	24.26
THE (in Decl package)	24.59
THEN (in Decl package)	24.57
translation notes (in transor package)	24.22-24
transor package	24.21-29
transor sweep	24.28
TRANSORSET[]	24.24,21
TTY#[]	24.43
TY (exec command)	24.42
type declarations	24.53-68
TYPE (Masterscope relation)	24.67
UND (exec command)	24.42
USEDIN (in Decl package)	24.56,58
USERSYMS (edita parameter)	24.13
WHENCLOSE[FILENAME;PROP1;VAL1;...;PROFn;VALn] * ..	24.34,49
WHEREIS[NAME;TYPE;FILES]	24.34
whereis package	24.34,35
WHEREIS.HASH (system variable/parameter)	24.34
XWD[N1;N2]	24.36
{ } (use with ftp package)	24.45
~ (in pattern match compiler)	24.4
! (in pattern match compiler)	24.4-6
#n (n a number, in pattern match compiler)	24.6
\$ (dollar) (edita parameter)	24.12
\$ (dollar) (in pattern match compiler)	24.4
\$C (<esc>C) (edita command)	24.14
\$n (in pattern match compiler)	24.4
\$Q (<esc>Q) (edita command)	24.12
\$W (<esc>W) (edita command)	24.13,14-15
\$1 (in pattern match compiler)	24.3
& (in pattern match compiler)	24.3
' (edita command)	24.13,10
' (in pattern match compiler)	24.3
* (in pattern match compiler)	24.4
. (edita command)	24.10
-- (in pattern match compiler)	24.4
-> (in pattern match compiler)	24.7
. (edita parameter)	24.12
. (in pattern match compiler)	24.5
/ (edita command)	24.11,10

Page
Numbers

/DELFILE[FILE]	24.43
/UNDELFILE[FILE]	24.43
: (edita command)	24.13
; (edita command)	24.15
= (edita command)	24.12
= (in pattern match compiler)	24.3
== (in pattern match compiler)	24.3
=> (in pattern match compiler)	24.7
? (edita command)	24.12
@ (edita command)	24.10
@ (in pattern match compiler)	24.3,5
↑ (edita command)	24.12
← (in pattern match compiler)	24.5

APPENDIX 1

CONTROL CHARACTERS

Several control characters are available to the user for communicating directly to Interlisp, i.e., not through the read program. These characters are enabled by Interlisp as interrupt characters, so that Interlisp immediately "sees" the characters, and takes the corresponding action as soon as possible. For example, control characters are available for aborting or interrupting a computation, changing the printlevel, etc. This section summarizes the action of these characters, and references the appropriate section of the manual where a more complete description may be obtained. Section 16 describes how these interrupt characters can be disabled and/or redefined, as well as how the user can define his own new interrupt characters.

CONTROL CHARACTERS AFFECTING THE FLOW OF COMPUTATION

1. control-H (interrupt) at next non-linked function call, Interlisp goes into a break. Section 16.
 2. control-B (break) computation is stopped, stack backed up to the last function call, and a break occurs. Section 16.
 3. control-E (error) computation is stopped, stack backed up to the last errorset, and NIL returned as its value. Section 16.
 4. control-D (reset) computation is stopped, control returns to evalqt.
 5. control-C In Interlisp-10, computation is stopped, control returns to the operating system. Program can always be continued without any ill effect with CONTINUE command.

If typed during a garbage collection the action of control-B, control-E, and control-D is postponed until the garbage collection is completed.

Typing control-E and control-D causes Interlisp to clear and save the input buffers. Their contents can usually be recovered via the \$BUFS (`<esc>BUFS`) command, as described in Section 22.

I/O CONTROL CHARACTERS

1. < del >¹ clears terminal input buffer. For example, < del > would be used if the user typed ahead while in a garbage collection and then changed his mind. Section 2. A bell is rung when the buffer has been cleared, so that the user will know when he may begin typing again.

Note: a sudden burst of noise on a telephone line frequently causes Interlisp to receive a , since the code for is 177Q, i.e., all 1's. This causes Interlisp to (mistakenly) clear the input buffer and ring a bell. If Interlisp seems to be typing many spurious bells, it is a good indication that you have a bad connection.

1 Control-Z for Interlisp-10 on TOPS-20.

2. control-O clears teletype output buffer, Sections 2 and 14.
3. control-P changes printlevel. Section 14.
4. control-A²,Q,³W line editing characters, Sections 2 and 14.⁴
5. control-R causes Interlisp to retype the input line, useful after several control-A's, e.g., if the user types
`←DEFINEQ((LAMDA\A\DBA\Acontrol-R, Interlisp
 types
 DEFINEQ((LAMB.`
6. control-V on input from the terminal, control-V followed by A, B, ... Z inputs the corresponding control character, otherwise is a no-op. The control-V is not passed to the line buffer; the transformation takes place before that. Thus ABC↑VD followed by two control-A's erases the control-D and the C. ↑V takes precedence over , i.e., ↑V inputs a control-C, ↑VC inputs a C.

MISCELLANEOUS

1. control-T (time) prints total execution time for program, as well as its status, e.g.,

```
*      ←RECLAIM()
*
*      collecting lists
*      COLLECTING AT 26310, LOAD 0.1, UTIL 37.8%
*      7819, 10379 FREE CELLS
*      10379
*      ←LOAD(FOO) 10 WAIT IN READ IN *EVAL QT* IN LISPXBLOCK, LOAD
*      0.1, UTIL 57.6%
*      RUNNING IN EVAL IN *LOAD* IN LOAD, LOAD 0.1, UTIL 8.6%
```

2. control-S⁶ (storage) change minfs. Section 10.
3. control-U⁷ if typed in the middle of an expression that is being typed to evalqt, break1 or the editor, will cause the editor to be called on the expression when it is finished being read. See Section 22.

+ ² < del > for Interlisp-10 on TOPS-20.

+ ³ Control-U for Interlisp-10 on TOPS-20.

⁴ Control-A, Q, R, V, and W are not interrupt characters, since their effect does not take place when they are typed, but when they are *read*. Section 14 describes how these pseudo-interrupt characters can also be disabled and/or redefined. Note that control-A, Q, R, V, and W have their special effect only on input from the terminal. On input from files, they are treated the same as any other character.

+ ⁵ Users who prefer the format used by the TENEX or TOPS-20 exec, i.e. RUNNING AT --, LOAD --, UTIL --, USED -- IN --, may disable the Interlisp Control-T by performing interruptchar[20].

+ ⁶ Control-X for Interlisp-10 on TOPS-20.

+ ⁷ Control-N for Interlisp-10 on TOPS-20.

- | | | |
|---------------------------|--|---|
| 4. line-feed | edit command. Aborts printout and moves to next expression and prints it. Section 9. | + |
| 5. control-X ⁸ | edit command. Aborts printout and moves to previous expression and prints it. Section 9. | + |
| 6. control-Z ⁹ | edit command. Aborts printout and moves to last expression of current expression and prints it. Section 9. | + |
| 7. control-Y | read macro. Returns result of evaluating next expression read as though it had been typed. Section 14. | + |

The function performed by <line-feed>, <control-X>, <control-Z>, and <control-Y>. may be easily assigned to other control characters via the function settermchars described in Section 14. +

⁸ Control-A for Interlisp-10 on TOPS-20. +

⁹ Control-L for Interlisp-10 on TOPS-20. +

Index for Section A1

	Page Numbers
bell (printed by system)	A1.1
belts (printed by system)	A1.1
CONTINUE (TENEX command)	A1.1
control characters	A1.1, 2-4
control-A	A1.2
control-A (TOPS-20) (edit command)	A1.3
control-B	A1.1
control-C	A1.1
control-D	A1.1
control-E	A1.1
control-H	A1.1
control-L (TOPS-20) (edit command)	A1.3
control-N (TOPS-20)	A1.2
control-O	A1.2
control-P	A1.2
control-Q	A1.2
control-R	A1.2
control-S	A1.2
control-T	A1.2
control-U	A1.2
control-U (TOPS-20)	A1.2
control-V	A1.2
control-W	A1.2
control-X (edit command)	A1.3
control-X (TOPS-20)	A1.2
control-Y (as a read-macro)	A1.3
control-Z (edit command)	A1.3
control-Z (TOPS-20)	A1.1
interrupt characters	A1.2
line-feed (edit command)	A1.3
SETTERMCHARS[NEXTCHAR;BKCHAR;LASTCHAR; UNQUOTECHAR;2CHAR;PPCHAR]	A1.3
\$BUFS (<esc>BUFS) (prog. asst. command)	A1.1
	A1.1
 (TOPS-20)	A1.2
<esc>BUFS (prog. asst. command)	A1.1

Index of Functions

	Page Numbers
ABS[X]	13.7
ADDBUFFER[TEMP;ERRORFLG]	21.13
ADDPROP[ATM;PROP;NEW;FLG]	7.2
ADDSPELL[X;SPLST;N]	17.18,20
ADDSTATS[STATLST] NL*	22.46
ADDTOCOMS[COMS;NAME;TYPE]	14.75
ADDTOFILE[NAME;TYPE;FILE;NEAR]	14.75
ADDTOFILES?[NOASKSTR]	14.67
ADDTOSCRATCHLIST[VALUE]	24.36
ADDTOVAR[VAR;X1;X2;...;Xn] NL*	5.6
ADD1[X]	13.2
ADIEU[VAL##] *	12.17
ADVISE[FN;WHEN;WHERE;WHAT]	19.4,3,5
ADVISEDUMP[X;FLG]	19.6
ALPHORDER[A;B]	6.8
AND[X1;X2;...;Xn] FSUBR*	5.11
ANTILOG[X]	13.7
APPEND[X1;X2;...;Xn] *	6.1
APPLY[FN;ARGS] SUBR	8.7; 2.3; 11.1; 16.1,
APPLY*[FN;ARG1;ARG2;...;ARGn] SUBR*	18.14
ARCCOS[X;RADIANSFGLG]	8.8; 2.3; 11.1; 16.1,
ARCSIN[X;RADIANSFGLG]	18.14
ARCTAN[X;RADIANSFGLG]	13.8
ARCTAN2[X;Y;RADIANSFGLG]	13.8
ARG[VAR;M] FSUBR	13.8
ARGLIST[FN]	8.9; 4.2
ARGTYPE[FN] SUBR	8.4; 2.2; 8.1-3; 15.9
ARRAY[N;P;V] SUBR	8.3,1-3
ARRAYBEG[A] SUBR	10.8; 3.5
ARRAYP[X] SUBR	10.9
ARRAYSIZE[A]	10.9
ARRAYTYP[ARRAY]	10.9
ASKUSER[WAIT;DEFAULT;MESS;KEYLST;TYPEAHEAD; [ISPXRNTFLG;OPTIONSLST;FILE]]	17.22,29,23-28,30
ASSOC[KEY;ALST]	5.13
ATOM[X] SUBR	5.9
ATTACH[X;L]	6.3
AU-REVOIR[VAL##] *	12.17
BACKTRACE[IPOS;EPOS;FLAGS;FILE;PRINTFN] SUBR	12.11
BAKTRACE[IPOS;EPOS;SKIPFN;FLAGS;FILE]	15.19
BCOMPL[FILES;CFILE;NOBLOCKSFLG]	18.22; 14.65; 18.19-21, 23
BIT[BIT#;WORD]	24.36
BKLINBUF[X] SUBR	14.34
BKSYSBUF[X] SUBR	14.34; 21.9
BLIPSCAN[BLIPTYP;IPOS] SUBR	12.4
BLIPVAL[BLIPTYP;IPOS;FLG] SUBR	12.4
BLKAPPLY[FN;ARGS] SUBR	18.14
BLKAPPLY*[FN;ARG1;ARG2;...;ARGn] SUBR*	18.14
BLOCKCOMPILE[BLKNAME;BLKFNS;ENTRIES;FLG]	18.19,20
BOUNDP[VAR]	8.7
BOXCOUNT[TYPE;N] SUBR	21.3
BREAK[X] NL*	15.16,1,5,17

Page
Numbers

BREAKCHECK[ERRORPOS;ERXN]	16.3,2,4-5,9-10; 17.10
BREAKDOWN[FNS] NL*	21.4-6
BREAKIN[FN;WHERE;WHEN;BRKCOMS] NL	15.16-17,1-2,5,15,18
BREAKLINKS[]	24.43
BREAKREAD[TYPE]	15.18
BREAKO[FN;WHEN;COMS;BRKFN;TAIL]	15.15,16-18
BREAK1[BRKEXP;BRKWHEN;BRKFN;BRKCOMS;BRKTYPE] NL	15.3,14,1-2,4-13,15-17, 16.1-2,4,10; 17.21
BRECOMPILE[FILES;CFILE;FNS;NOBLOCKSFLG]	18.23; 14.65-66, 18.19-22
BRKDWNRESULTS[RETURNVALUESFLG]	21.4
CALLS[FN;USEDATABASE]	20.18
CALLSCCODE[FN]	20.18
CAR[X] SUBR	5.1
CBOX[X;Y]	24.37
CCODEP[FN] SUBR	8.3; 3.13-14; 8.1-3
CDR[X] SUBR	5.1
CHANGECALLERS[OLD;NEW;TYPES;FILES;METHOD]	14.71
CHANGEFONT[FONTCLASS]	14.52,50
CHANGENAME[FN;FROM;TO]	9.64; 15.18
CHANGEPROP[X;PROP1;PROP2]	7.3
CHANGESLICE[N;HISTORY;L]	22.40,7
CHARACTER[N] SUBR	10.3
CHCON[X;FLG;RDTBL] SUBR	10.3
CHCON1[X] SUBR	10.3
CHECKCONNECTION[CONNECTION]	24.47
CHECKNIL[]	22.38
CHOOZ[XWORD;REL;SPLST;TAIL;FN;TIEFLG;NDBLS;CLST]..	17.16
CIRCLMAKER[LIST]	24.19
CIRCLPRINT[LIST;PRINTFLG;RLKNT]	24.18,17
CLDISABLE[OP]	23.59
CLEANPOSLST[PLST]	12.17
CLEANUP[FILES] NL*	14.66
CLEARBUF[FILE;FLG] SUBR	14.34; 22.26
CLEARMAP[FILE;PAGES;RELEASE]	21.14
CLEARSTK[FLG] SUBR	12.11
CLISPDEC[DECLST]	23.58,24
CLISPIFY[X;L]	23.38,58; 14.65; 23.25, 39-40
CLISPIFYFNS[FNS] NL*	23.59
CLISPTRAN[X;TRAN]	23.57
CLOCK[N] SUBR	21.3
CLOSEALL[]	14.5,10
CLOSECONNECTION[CONNECTION]	24.46
CLOSEF[FILE]	14.5
CLOSEF?[FILE]	14.5
CLOSEHASHFILE[HASHFILE]	24.49
CLOSER[A;X] SUBR	10.15
CLRHASH[HARRAY] SUBR	7.4
CNDIR[DIR;PASSWORD]	24.43
COMMENT1[L;INBLOCKFLG]	14.80
COMPARE[NAME1;NAME2;TYPE;SOURCE1;SOURCE2]	14.71
COMPAREDEFS[NAME;TYPE;SOURCES]	14.72
COMPARELISTS[X;Y]	6.9
COMPILE[X;FLG]	18.5
COMPILEFILES[FILES] NL*	14.66

Page
Numbers

COMPILE1[FN;DEF;COREFLG]	18.5
COMPSET[FILE;FLG;FILES]	18.2
CONCAT[X1;X2;...;Xn] SUBR*	10.5; 3.6; 10.8
COND[C1;C2;...;Cn] FSUBR*	5.3; 4.3
CONS[X;Y] SUBR	5.1; 3.5,9
CONSCOUNT[N] SUBR	5.1; 10.14; 21.3
CONSTANT[X]	18.11
CONTROL[FLG;RDTBL] SUBR	14.33; 2.4; 14.12,14,32
COPY[X]	6.4
COPYALL[X]	6.4; 23.33
COPYALLBYTES[FROMFILE;TOFILE;BYTESIZE]	24.43
COPYARRAY[AR]	10.10
COPYBYTES[SRCFIL;DSTFIL;START;END]	14.9
COPYDEF[OLD;NEW;TYPE;SOURCE;OPTIONS]	14.70
COPYHASHFILE[HASHFILE;NEWNAME;FN;VTYPE]	24.50
COPYREADTABLE[RDTBL] SUBR	14.24
COPYSTK[POS1;POS2] SUBR	12.11
COPYTERMTABLE[TTBL] SUBR	14.29
COREVAL[X] NL	21.3
COROUTINE[CALLPTR##;COROUTPTR##;COROUTFORM##; ENDFORM##] NL	12.15
COS[X;RADIANSFGLG]	13.7
COUNT[X]	6.6
COUNTDOWN[X;N]	6.7
COVERS[HI;LO]	24.65
CREATEHASHFILE[FILE;VALUETYPE;ITEMLENGTH; #ENTRIES]	24.48
DATE[FORMATBITS] SUBR	21.2
DATEFORMAT[KEYWORD1;KEYWORD2;...;KEYWORDn]	24.40
DCHCON[X;SCRATCHLIST;FLG;RDTBL]	10.3
DDT[] SUBR	24.9
DECLAREDATATYPE[TYPENAME;FIELDSPECS;FLG]	3.7; 23.29
DECLARE:[X] NL*	14.59
DECLOF[FORM]	24.66
DECLTYPE[TYPENAME;TYPEEXPRESSION;PROP1;VAL1;...; PROPh;VALn]	24.63
DEFAULTMAKENEWCOM[NAME;TYPE;LISTNAME;FILE]	14.72
DEFERREDCONSTANT[X]	18.12
DEFEVAL[TYPE;FN]	8.7
DEFINE[X;TYPE-IN]	8.5; 2.5
DEFINEQ[X1;X2;...;Xn] NL*	8.5; 2.5-6
DEFLIST[L;PROP]	7.3
DEFPRINT[TYPE;FN]	14.22
DELDEF[NAME;TYPE]	14.70
DELETECONTROL[TYPE;MESSAGE;TTBL]	14.31
DELFILE[FILE]	14.5
DELFROMCOMS[COMS;NAME;TYPE]	14.76
DELFROMFILES[NAME;TYPE;FILES]	14.75
DELPAGE[PAGE#;HASHFILE]	24.52
DETACH[]	24.43
DETACHEDP[]	24.43
DIFFERENCE[X;Y]	13.6
DIRECTORY[FILEGROUP;COMMANDS;DEFAULTTEXT; DEFAULTVERS;LISPXRNTFLG]	21.10
DISMISS[N]	21.3
DISPLAYTERMP[]	21.16

	Page Numbers
DMPHASH[ARRAYNAME1;...;ARRAYNAME _n] NL*	7.5
DOBE[]	14.18
DOCOLLECT[ITEM;LST]	6.3
DREMOVE[X;L]	6.3
DREVERSE[L]	6.4
DRIBBLE[FILE;APPENDFLG;THAWEDFLG]	21.15
DRIBBLEFILE[] SUBR	21.15
DSKSTAT[DIR;PRINTIFOVER;PRINTSYS;PRINTDEL; PRINTOLD]	24.43
DSUBLIS[ALST;EXPR;FLG]	6.5
DSUBST[NEW;OLD;EXPR]	6.4,5
DUMMYFRAMEP[POS]	12.7
DUMPDATABASE[FNLST]	20.20
DUMPDB[FILE]	24.32
DUNPACK[X;SCRATCHLIST;FLG;RDTBL]	10.3
DWIM[X]	17.17,3
DWIMIFY[X;QUIETFLG;L]	17.17; 23.40,57,41-42, 51.58
DWIMIFYFNS[FNS] NL*	23.58,41
DWIMLOADFNS?[]	17.16
ECHOCONTROL[CHAR;MODE;TTBL]	14.30
ECHOMODE[FLG;TTBL] SUBR	14.30
EDITA[EDITARRY;COMS]	24.8-15
EDITCALLERS[ATOMS;FILES;COMS]	9.60
EDITDATE[OLDDATE;INITLS]	9.59
EDITDATE?[COMMENT]	9.59
EDITDEF[NAME;TYPE;SOURCE;EDITCOMS]	14.70
EDITE[EXPR;COMS;ATM;TYPE;IFCHANGEDFN]	9.61,1,61
EDITF[NAME;COM1;COM2;...;COMn] NL*	9.58,1,59,61
EDITFINDP[X;PAT;FLG]	9.63
EDITFNS[NAME:COM1;COM2;...;COMn] NL*	9.59,60
EDITFPAT[PAT;FLG]	9.63
EDITL[L;COMS;ATM;MESS;EDITCHANGES]	9.62
EDITLOADFNS? [FN;STR;ASKFLG;FILES]	9.64
EDITLO[L;COMS;MESS;EDITLFLG]	9.63
EDITP[NAME;COM1;COM2;...;COMn] NL*	9.61,1
EDITREC[EDITRECX] NL*	23.35
EDITUSERFN[COM]	9.56
EDITV[NAME;COM1;COM2;...;COMn] NL*	9.61,1
EDIT4E[PAT;X;CHANGEFLG]	9.63
ELT[A;N] SUBR	10.9; 3.5; 16.7
ELTD[A;N] SUBR	10.10; 3.5
ENDCOLLECT[LST;TAIL]	6.3
ENDFILE[FILE]	14.42
ENTRY#[HIST;X]	22.40
ENVAPPLY[FN;ARGS;APOS;CPOS;AFLG;CFLG] SUBR	12.9
ENVEVAL[FORM;APOS;CPOS;AFLG;CFLG] SUBR	12.9
EQ[X;Y] SUBR	5.10; 2.2
EQLENGTH[X;N]	6.6
EQMEMB[X;Y]	5.12
EQP[X;Y] SUBR	5.10; 13.3,5; 3.3, 13.1,4
EQUAL[X;Y] SUBR	5.10; 2.2; 13.1
EQUALALL[X;Y]	5.11
EQUALN[X;Y;DEPTH]	5.11
ERROR[MESS1;MESS2;NOBREAK]	16.10,4,6,9

	Page Numbers
ERRORMESS[U]	16.11,4
ERRORN[] SUBR	16.11,4
ERRORSET[FORM;FLG] SUBR	16.11; 7.5; 16.3-4,10, 17.10
ERRORSTRING[N] SUBR	16.11
ERRORX[ERXM]	16.10
ERROR![] SUBR	16.10; 15.5; 16.10
ERSETQ[ERSETX] NL	16.12; 5.5; 16.11
ERSTR[ERN]	21.8
ESCAPE[FLG] SUBR	14.14,14
ESUBST[NEW;OLD;EXPR;ERRORFLG;CHARFLG]	9.63; 6.5; 22.13
EVAL[X] SUBR	8.6; 2.3,6; 4.1; 16.11
EVALA[X;A] SUBR	8.8
EVALQT[LISPID]	2.3; 5.7; 15.4
EVALV[X;POS] SUBR	12.10
EVERY[EVERYX;EVERYFN1;EVERYFN2]	5.11
EXPANDMACRO[FORM;QUIETFLG]	18.11
EXPRP[FN] SUBR	8.3,1-2,4
EXPT[M;N]	13.7
EXPUNGE[DIR]	24.43
FASSOC[KEY;ALST]	5.13; 2.2
FAULTAPPLY[FAULTFN;FAULTARGS]	16.1; 17.3,10,14; 18.17
FAULTEVAL[FAULTX] NL*	16.1,6; 17.3,10,14
FBOX[N]	24.38
FCHARACTER[N] SUBR	10.3
FDIFFERENCE[X;Y]	13.5
FETCHFIELD[DESCRIPTOR;DATUM]	3.8
FFILEPOS[PATTERN;FILENAME;FILESTART;FILEEND;SKIP; TAIL;CASEARRAY]	14.9
FGETD[X]	8.2
FGREATERP[X;Y] SUBR	13.5
FIELDLOOK[FIELDNAME]	23.35
FILDIR[FILEGROUP;FORMATFLG]	21.8
FILECOMS[FILE;X]	14.76
FILECOMSLST[FILE;TYPE;FLG]	14.76
FILECREATED[X] NL*	14.79,60; 18.5
FILEDATE[FILE;CFLG]	14.79
FILEFNSLST[FILE]	14.76
FILENAMEFIELD[FILENAME;FIELDNAME]	14.6
FILEPKGCHANGES[N] *	14.67
FILEPKGCOM[COMMNAME;PROP1;VAL1;...;PROPN;VALn] *	14.73
FILEPKGTYPE[TYPE;PROP1;VAL1;...;PROPN;VALn] *	14.72
FILEPOS[PATTERN;FILENAME;FILESTART;FILEEND;SKIP; TAIL;CASEARRAY]	14.8
FILES?[]	14.67,64
FINDCALLERS[ATOMS;FILES]	9.61
FINDFILE[FILE;NSFLG]	17.21
FIX[X]	13.4
FIXEDITDATE[EXPR]	9.59
FIXP[X]	13.4
FIXSPELL[XWORD;REL;SPLST;FLG;TAIL;FN;TIEFLG; DONTMOVETOFLG;CLST;APPROVALFLG]	17.18,19,21
FLAST[X]	6.5; 2.2
FLENGTH[X]	6.6; 2.2
FLESSP[X;Y]	13.5
FLOAT[X]	13.6

Page
Numbers

FLOATP[X] SUBR	13.6
FLTfmt[FORMATBITS]	14.35; 3.4
FLUSHRIGHT[POS;X;MIN;P2FLAG;CENTERFLAG;FILE]	23.50
FMAX[X1;X2;...;Xn]	13.6
FMEMB[X;Y]	5.12; 2.2
FMIN[X1;X2;...;Xn]	13.6
FMINUS[X]	13.5
FNCHECK[FN;NOERRORFLG;SPELLFLG;PROPFLG;TAIL]	17.20; 8.4; 17.21
FNTH[X;N]	6.6; 2.2
FNTYP[FN]	8.3; 4.2; 8.1-2,4,6
FONTNAME[NAME]	14.52
FONTSET[NAME]	14.52
FORCEOUT[CONNECTION/FILE]	24.47
FPLUS[X1;X2;...;Xn] SUBR*	13.5
FQUOTIENT[X;Y] SUBR	13.5
FRAMESCAN[ATOM;POS] SUBR	12.8
FREEVARS[FN;USEDATABASE]	20.19
FREMAINDER[X;Y] SUBR	13.5
FRPLACA[X;Y] SUBR	5.2
FRPLACD[X;Y] SUBR	5.2
FRPLNODE[X;A;D]	5.2
FRPLNODE2[X;Y]	5.2
FRPTQ[N;FORM1;FORM2;...;FORMn] NL*	8.8
FTIMES[X1;X2;...;Xn] SUBR*	13.5
FTP[HOST;FILE;ACCESS;USER;PASSWORD;ACCOUNT; BYTESIZE]	24.45
FULLNAME[X;RECOG]	14.3
FUNCTION[FN;ENV] NL	11.1; 12.10; 11.3-4, 18.13
GAINSPACE[]	21.16
GCD[X;Y]	13.5
GCGAG[MESSAGE]	10.11
GCMESS[MESSAGE#;STRING] SUBR	10.12
GCTRPN[N] SUBR	10.14; 21.4
GDATE[DATE;FORMATBITS;STRPTR]	21.2
GENERATE[HANDLE;VAL]	12.14
GENERATOR[FORM##;COMVAR##] NL	12.14
GENSYM[CHAR]	10.3; 3.2; 10.4; 15.15, 18.13; 19.3-4
GEQ[X;Y]	13.6
GETATOMVAL[ATM] SUBR	5.6
GETBLK[N] SUBR	21.6; 16.7
GETBRK[RDTBL] SUBR	14.14
GETCOMMENT[X;DESTFL;DEF]	14.45
GETCONTROL[TTBL]	14.33
GETD[X] SUBR	8.2; 2.3,5; 8.1,6
GETDECLTYPEPROP[TYPE;PROP]	24.65
GETDEF[NAME;TYPE;SOURCE;OPTIONS]	14.69
GETDELETECONTROL[TYPE;TTBL]	14.31
GETDESCRIPTORS[TYPENAME]	3.8
GETECHOMODE[TTBL]	14.31
GETEOFPTR[FILE] SUBR	14.8
GETFIELDSPECS[TYPENAME]	3.8
GETFILEINFO[FILE;ATTRIB]	14.4
GETFILEMAP[FILE;FL]	14.41
GETFILEPTR[FILE] SUBR	14.7

Page
Numbers

GETHASH[ITEM;HARRAY] SUBR	7.4; 23.22
GETHASHFILE[KEY;HASHFILE]	24.49
GETLIS[X;PROPS]	7.3
GETPAGE[hashfile;n]	24.52
GETPASSWORD[DIRECTORYNAME]	24.44
GETPNAME[FILEADR;HASHFILE]	24.52
GETPROP[ATM;PROP]	7.1
GETPROPLIST[ATM]	7.1; 2.3
GETRAISE[TTBL]	14.32
GETREADTABLE[RDTBL] SUBR	14.24
GETRELATION[ITEM;RELATION;INVERTED]	20.19
GETSEPR[RDTBL] SUBR	14.14
GETSYNTAX[CH;TABLE]	14.25
GETTEMPLATE[FN]	20.18
GETTERMTABLE[TTBL] SUBR	14.29
GETTOPVAL[ATM] SUBR	5.5; 2.3
GETTYPEDESCRIPTION[TYPE]	10.13
GLC[X] SUBR	10.5,8
GNC[X] SUBR	10.5,8
GO[X] FSUBR*	5.5
GREATERP[X;Y] SUBR	13.6
GREET[NAME;FLG]	22.47
GTJFN[FILE;EXT;V;FLAGS]	21.12
HARRAY[N] SUBR	7.4; 10.9
HARRAYP[X]	7.4; 10.9
HARRAYSIZE[HARRAY]	7.4; 10.9
HASDEF[NAME;TYPE;SPELLFLG]	14.72
HASHFILENAME[hashfile]	24.49
HASHFILEP[X]	24.49
HASHFILEPROP[hashfile;PROP]	24.49
HASHFILESPLST[hashfile]	24.51
HCOPYALL[X]	6.4; 14.23
HELP[MESS1;MESS2]	16.10
HERALD[STRING] SUBR	3.11
HISTORYFIND[LST;INDEX;MOD;EVENTADDRESS; LISPXFINDFLG]	22.39,39
HISTORYMATCH[INPUT;PAT;EVENT]	22.40
HISTORYSAVE[HISTORY;ID;INPUT1;INPUT2;INPUT3; PROPS]	22.38,9,27,34-35,39,45
HOSTNAME[HOSTN]	21.9
HOSTNUMBER[]	21.9
HPRINT[EXPR;FILE;UNCIRCULAR]	14.23
HREAD[FILE]	14.23
IBOX[N]	24.38
IDATE[D]	21.2
IDIFFERENCE[X;Y]	13.2
IEQP[N;M] SUBR	13.3
IGEQ[X;Y]	13.3
IGREATERP[X;Y] SUBR	13.3
ILEQ[X;Y]	13.3
ILESSP[X;Y]	13.3
IMAX[X ₁ ;X ₂ ;...;X _n] *	13.3
IMIN[X ₁ ;X ₂ ;...;X _n] *	13.3
IMINUS[X]	13.2
INFILE[FILE] SUBR	14.1,7

Page
Numbers

INFILECOMS?[NAME;TYPE;COMS;ONFILETYPE]	14.75
INFILEP[FILE] SUBR	14.2,2
INPUT[FILE] SUBR	14.1
INREADMACROP[] SUBR	14.28
INTERRUPT[INTFN;INTARGS;INTYPE]	16.2; 10.14
INTERRUPTABLE[FLG] SUBR	16.13
INTERRUPTABLEP[] SUBR	16.13
INTERRUPTCHAR[CHAR;TYP/FORM;HARDFLG]	16.12
INTERSECTION[X;Y]	6.7
IOFILE[FILE] SUBR	14.4,7-8
IPLUS[X1;X2;...;Xn] SUBR*	13.2
IQUOTIENT[X;Y] SUBR	13.3
IREMAINDER[X;Y] SUBR	13.3
ITIMES[X1;X2;...;Xn] SUBR*	13.2
I.S.OPR[NAME;FORM;OTHERS;EVALFLG]	23.19
JFNS[JFN;AC3;STRPTR]	21.12
JOB#[]	24.43
JS[JSYSNAME;AC1;AC2;AC3;RESULT]	24.35
JSYS[N;AC1;AC2;AC3;RESULTAC] SUBR	21.9
JSYSERROR[ERRORN]	24.36
KFORK[FORK]	21.8,8
KWOTE[X]	5.3
LAST[X]	6.5
LASTC[FILE] SUBR	14.15
LASTN[L;N]	6.6
LBOX[X1;X2;...;Xn]	24.37
LCONC[PTR;X]	6.2,3
LDIFF[X;Y;Z]	6.7
LDIFFERENCE[X;Y]	6.7
LENGTH[X]	6.6
LEQ[X;Y]	13.7
LESSP[X;Y]	13.6
LINBUF[FLG] SUBR	14.34
LINELENGTH[N] SUBR	14.35,48; 2.2; 3.5
LINKTOTTY[TTY#]	24.43
LINKTOUSER[USER]	24.43
LISPX[LISPXX;LISPXID;LISPXXMACROS;LISPXXUSERFN; LISPXFLG]	22.9,35-36; 9.41,48, 17.3,8-9,21; 22.13, 15-17,25,28-32,34, 37-38,45
LISPXEVAL[LISPXFORM;LISPXID]	22.38
LISPXFIND[HISTORY;LINE;TYPE;BACKUP;QUIETFLG]	22.39,45
LISPXPRINT[X;Y;Z;NODOFLG]	22.30,34
LISPXPRINTDEF[EXPR;FILE;LEFT;DEF;TAIL;NODOFLG]	22.30
LISPXPRI1[X;Y;Z;NODOFLG]	22.30
LISPXPRI2[X;Y;Z;NODOFLG]	22.30
LISPXREAD[FILE;RDTBL]	22.37,9,16,25,27,36,45
LISPXREADP[FLG]	22.38,45
LISPXSPACES[X;Y;Z;NODOFLG]	22.30
LISPXSTATS[RETURNVALUESFLG]	22.46
LISPXSTOREVALUE[EVENT;VALUE]	22.39
LISPXTAB[X;Y;Z;NODOFLG]	22.30
LISPXTERPRI[X;Y;Z;NODOFLG]	22.30

Page
Numbers

LISPXUNREAD[LST;EVENT]	22.38
LISPXWATCH[STAT;N]	22.46
LISPX/[X;FN;VARS]	22.42,32
LIST[X1;X2;...;Xn] SUBR*	6.1; 3.5
LISTFILES[FILES] NL*	14.66,64-65
LISTFILES1[FILE]	14.66
LISTGET[LST;PROP]	5.13; 23.28
LISTGET1[LST;PROP]	5.13
LISTP[X] SUBR	5.10; 2.3
LISTPUT[LST;PROP;VAL]	5.13; 23.28
LISTPUT1[LST;PROP;VAL]	5.13
LITATOM[X] SUBR	5.9
LLSH[N;M] SUBR	13.4
LOAD[FILE;LDFLG;PRINTFLG]	14.38; 2.9; 18.5
LOADAV[]	21.8
LOADBLOCK[FN;FILE;LDFLG]	14.40
LOADCOMP[FILE;LDFLG]	14.40
LOADCOMP?[FILE;LDFLG]	14.41
LOADDB[FILE]	24.32
LOADDEF[NAME;TYPE;SOURCE]	14.71
LOADEFS[FNS;FILE]	14.40
LOADFNS[FNS;FILE;LDFLG;VARS]	14.39
LOADFROM[FILE;FNS;LDFLG]	14.40; 18.8
LOADVARS[VARS;FILE;LDFLG]	14.40
LOAD?[FILE;LDFLG;PRINTFLG]	14.39
LOC[X] SUBR	13.10,10
LOCKMAP[PTR]	21.15
LOG[X]	13.7
LOGAND[X1;X2;...;Xn] SUBR*	13.4
LOGOR[X1;X2;...;Xn] SUBR*	13.4
LOGOUT[] SUBR	21.4; 2.3; 21.8
LOGXOR[X1;X2;...;Xn] SUBR*	13.4
LOOKUPHASHFILE[KEY;VALUE;HASHFILE;CALLTYPE]	24.51
LOWERCASE[FLG]	23.61
LRSH[N;M]	13.4
LSH[N;M] SUBR	13.4
LSUBST[NEW;OLD;EXPR]	6.4,5
L-CASE[X;FLG]	10.4; 9.50
MAKEBITTABLE[L;NEG;A]	10.7
MAKEFILE[FILE;OPTIONS;REPRINTFNS;SOURCEFILE]	14.64-66; 17.21; 18.8
MAKEFILES[OPTIONS;FILES]	14.66
MAKEKEYLST[LST;DEFAULTKEY;LCASEFLG]	17.29
MAKENEWCOM[NAME;TYPE;LISTNAME;FILE]	14.76
MAKENEWCONNECTION[HOST;TYPE;SKT;SCRATCHCONN; WAITFLG]	24.46
MAKESYS[FILE] SUBR	3.11
MAP[MAPX;MAPFN1;MAPFN2]	11.2
MAPATOMS[FN] SUBR	10.4
MAPBUFFERCOUNT[ONLY]	21.13
MAPC[MAPX;MAPFN1;MAPFN2]	11.2
MAPCAR[MAPX;MAPFN1;MAPFN2]	11.2
MAPCON[MAPX;MAPFN1;MAPFN2]	11.2
MAPCONC[MAPX;MAPFN1;MAPFN2]	11.2
MAPDL[MAPDLFN;MAPDLPOS]	12.12
MAPHASH[ARRAY;MAPFN]	7.5
MAPHASHFILE[HASHFILE;MAPFN]	24.50

Page
Numbers

MAPLIST[MAPX;MAPFN1;MAPFN2]	11.2
MAPPAGE[PAGE#;FILE]	21.13
MAPRELATION[RELATION;MAPFN]	20.20
MAPPRINT[LST;FILE;LEFT;RIGHT;SEP;PFN;LSPXPRNTFLG]..	11.3
MAPWORD[FILEADR;FILE]	21.14
MAP2C[MAPX;MAPY;MAPFN1;MAPFN2]	11.3
MAP2CAR[MAPX;MAPY;MAPFN1;MAPFN2]	11.3
MARKASCHANGED[NAME;TYPE;NEWFLG]	14.67
MASTERSCOPE[COMMAND]	20.19
MAX[X1;X2;...;Xn] *	13.7
MEMB[X;Y]	5.12
MEMBER[X;Y]	5.12
MEMSTAT[PG1;PGN;FORK]	24.44
MERGE[A;B;COMPAREFN]	6.8
MERGEINSERT[NEW;LST;ONEFLG]	6.8
MIN[X1;X2;...;Xn] *	13.7
MINFS[N;TYPE]	10.13; 3.10-11
MINUS[X] SUBR	13.6
MINUSP[X] SUBR	13.3,5
MISSPELLED?[XWORD;REL;SPLST;FLG;TAIL;FN]	17.18,21
MKATOM[X] SUBR	10.5; 3.2-4
MKLIST[X]	6.3
MKSTRING[X] SUBR	10.4; 3.6,9; 10.8
MKSWAP[X]	3.14
MKSWAPP[FNAME;CDEF]	3.14
MKUNSWAP[X]	3.14
MOVED[FROM;TO;COPYFLG]	8.3
MOVED?[FROM;TO;COPYFLG]	8.3
MOVEITEM[TOFILE;NAME;TYPE;FROMFILE]	14.76
MSMARKCHANGED[NAME;TYPE;NEWFLG]	20.20
MULTIFILEINDEX[FILENAMELST;MAPFILE;NEWPAGEFLG]	24.30
NARGS[FN]	8.4,1-3
NBOX[N]	24.38-39
NCHARS[X;FLG;RDTBL] SUBR	10.3; 14.8
NCONC[X1;X2;...;Xn] SUBR*	6.1,2
NCONC1[LST;X]	6.1,2
NCREATE[TYPENAME;FROM]	3.8
NEGATE[X]	5.10
NEQ[X;Y]	5.10
NETSERVER[ARPA#;WAITFLG]	24.47
NETUSER[HOST;USER;PUP#;WAITFLG]	24.47
NEWISWORD[SING;PLU;FORM;VARS]	23.21
NEW/FN[FN]	22.42
NILL[]	5.10
NLEFT[L;N;TAIL]	6.5
NLISTP[X]	5.10; 2.2
NLSETQ[NLSETX] NL	16.12; 5.5; 16.11, 22.43
NOT[X] SUBR	5.10
NOTANY[SOME _X ;SOMEFN1;SOMEFN2]	5.12
NOTE[VAL;LSTFLG]	12.17
NOTEVERY[EVERY _X ;EVERYFN1;EVERYFN2]	5.12
NTH[X;N]	6.6
NTHCHAR[X;N;FLG;RDTBL] SUBR	10.3
NTYP[DATUM] SUBR	10.11
NULL[X] SUBR	5.10

Page
Numbers

NUMBERP[X] SUBR	5.9
NUMFORMATCODE[FORMAT;SMASHCODE]	14.22
OPENF[FILE;X] SUBR	21.12
OPENFILE[FILE;ACCESS;RECOG;BYTESIZE; MACHINE.DEPENDENT.PARAMETERS]	14.3
OPENHASHFILE[FILE;ACCESS]	24.48
OPENP[FILE;ACCESS] SUBR	14.4,2
OPENR[A] SUBR	10.15
OPENJFN[FILE;TYPE] SUBR	21.11
OR[X1;X2;...;Xn] FSUBR*	5.11
OUTFILE[FILE] SUBR	14.2,7-8
OUTFILEP[FILE] SUBR	14.3,2
OUTPUT[FILE] SUBR	14.1
PACK[X] SUBR	10.2; 3.2-4,9
PACKC[X] SUBR	10.3
PACKFILENAME[FIELDNAME1;FIELDCONTENTS1; FIELDNAME _n ;FIELDCONTENTS _n] *	14.6
PACK*[X] SUBR*	10.2
PAGEFAULTS[]	21.4
PARSERELATION[RELATION]	20.19
PEEKC[FILE;RDtbl] SUBR	14.14,33
PF[FN;FROMFILES;TOFILE] NL*	14.44
PF*[FN;FROMFILES;TOFILE] NL*	14.44
PLUS[X1;X2;...;Xn] SUBR*	13.6
POSITION[FILE;N] SUBR	14.36
POSSIBILITIES[FORM##] NL	12.16
PP[X] NL*	14.43
PPT[X] NL*	23.60,22
PP*[X] NL*	14.44
PRESCAN[FILE;CHARLST;PRESCANFN]	24.22
PRETTYCOMPRINT[X] NL	14.79
PRETTYDEF[PRTTYFNS;PRTTYFILE;PRTTYCOMS; REPRINTFNS;SOURCEFILE;CHANGES]	14.78,64; 19.6
PRETTYPRINT[FNS;PRETTYDEFLG;FNSLST]	14.43; 2.9
PRINT[X;FILE;RDtbl] SUBR	14.17; 3.1,5-6; 14.18
PRINTBELLS[]	17.4; 14.18
PRINTBINDINGS[AT;POS]	15.8; 22.24
PRINTDATE[FILE;CHANGES]	14.79
PRINTDEF[EXPR;LEFT;DEF;TAILFLG;FNSLST]	14.79,49,80
PRINTFNS[X;PRETTYDEFLG]	14.79
PRINTHISTORY[HISTORY;LINE;SKIPFN;NOVALUES;FILE]	22.44,18,30-31
PRINTL[ITEM;DEPTH;LMARG;RMARG;FILE]	24.20
PRINTLEVEL[CARN;CDRN] SUBR	14.18; 2.2; 3.5
PRINTNUM[FORMAT;NUMBER;FILE]	14.20
PRINTPARA[LMARG;RMARG;LIST;P2FLAG;PARENFLAG;FILE]	23.50
PRINTPROPS[AT]	22.24
PRIN1[X;FILE] SUBR	14.17; 3.2,5-6; 14.18
PRIN2[X;FILE;RDtbl] SUBR	14.17; 3.1,5-6; 14.18
PRIN3[X;FILE] SUBR	14.17
PRIN4[X;FILE;RDtbl] SUBR	14.17
PRODUCE[VAL]	12.14
PROG[VARLST;E1;E2;...;En] FSUBR*	5.4
PROGN[X1;X2;...;Xn] FSUBR*	4.3; 5.4
PROG1[X1;X2;...;Xn] FSUBR*	5.4
PROMPTCHAR[ID;FLG;HISTORY]	22.38,27,45

	Page Numbers
PROPNAMES[ATM]	7.3
PSTEP[]	18.26
PSTEPN[N]	18.32
PUTASSOC[KEY;VAL;ALST]	5.13; 23.28
PUTD[FN;DEF] SUBR	8.2; 2.3,5; 8.1-2
PUTDEF[NAME;TYPE;DEFINITION]	14.70
PUTDQ[FN;DEF] NL	8.2
PUTDQ?[FN;DEF] NL	8.2
PUTHASH[ITEM;VAL;HARRAY] SUBR	7.4
PUTHASHFILE[KEY;VALUE;HASHFILE]	24.49
PUTPROP[ATM;PROP;VAL]	7.2
PUTPROPS[ATM;PROP1;VAL1;...;PROPn;VALn] NL*	7.2
QUOTE[X] NL*	5.2
QUOTIENT[X;Y] SUBR	13.6
RADIX[N] SUBR	14.34; 2.2; 3.3; 14.12, 17
RAISE[FLG;TTBL] SUBR	14.32
RAND[LOWER;UPPER]	13.8
RANDACCESSP[FILE]	14.8
RANDSET[X]	13.8
RATEST[X] SUBR	14.14
RATOM[FILE;RDTBL] SUBR	14.12,13,33
RATOMS[A;FILE;RDTBL]	14.13
READ[FILE;RDTBL;FLG] SUBR	14.11,12,33
READC[FILE] SUBR	14.14,33
READFILE[FILE]	14.42
READLINE[RDTBL;LINE;LISPXFLG]	14.15; 22.35; 9.56, 14.16; 22.12,16,27,30, 36-37,45
READMACROS[FLG] SUBR	14.28
READP[FILE;FLG] SUBR	14.15
READTABLEP[RDTBL] SUBR	14.24
READVISE[X] NL*	19.5; 14.58; 19.6
REALFRAMEP[POS;INTERPFLG]	12.7
REALSTKNTH[N;POS;INTERPFLG;OLDPOS]	12.8
REBREAK[X] NL*	15.18,15
RECLAIM[TYPE]	10.11; 3.10
RECLOOK[RECNAME;TL;LOCALDEC;PARENT;ERROR]	23.35
RECOMPILE[PFILE;CFILE;FNS]	18.7; 14.65-66; 18.5,8, 21
RECORDACCESS[FIELD;DATUM;DEC;TYPE;NEWVALUE]	23.36
RECORDFIELDNAMES[RECORDNAME]	23.35
REHASH[OLDAR;NEWAR] SUBR	7.5
REHASHFILE[HASHFILE]	24.50
RELBLK[ADDRESS;N] SUBR	21.6; 16.7
RELINK[FN;UNLINKFLG]	18.17
RELSTK[POS] SUBR	12.11,12
RELSTKP[X]	12.11
REMAINDER[X;Y] SUBR	13.6
REMOVE[X;L]	6.3
REMPROP[ATM;PROP]	7.2
REMPROPLIST[ATM;PROPS]	7.2
RENAME[OLD;NEW;TYPES;FILES;METHOD]	14.71
RENAMEFILE[OLD;NEW]	14.5
REPLACEFIELD[DESCRIPTOR;DATUM;NEWVALUE]	3.8

Page
Numbers

RESET[] SUBR	16.11
RESETBUFS[FORM1;FORM2;...;FORMn] NL*	14.34
RESETFORM[RESETFORM;FORM1;FORM2;...;FORMn] NL*	5.8
RESETLST[RESETX] NL*	5.7
RESETREADTABLE[RDTBL;FROM] SUBR	14.24
RESETSAVE[RESETX] NL*	5.7
RESETTERMTABLE[TTBL;FROM] SUBR	14.29
RESETUNDO[X;STOPFLG]	22.43; 5.9; 22.34
RESETVAR[VAR;NEWVALUE;FORM] NL	5.8; 18.5
RESETVARS[VARSLST;E1;E2;...;En] FSUBR*	5.8
RESULTS[RETURNVALUESFLG]	21.6
RESUME[FROMPTR;TOPTR;VAL] SUBR	12.15
RETAPPLY[POS;FN;ARGS;FLG;INTERNALFLG]	12.10
RETEVAL[POS;FORM;FLG;INTERNALFLG]	12.9; 15.4; 17.10
RETFROM[POS;VAL;FLG] SUBR	12.10; 15.4; 16.4
RETTO[POS;VAL;FLG] SUBR	12.10
RETURN[X] SUBR	5.5
REVERSE[L]	6.4
RLJFN[JFN]	21.12
RPAQ[X;Y] NL	5.6; 14.38; 22.33
RPAQQ[X;Y] NL	5.6; 14.38,78; 22.33
RPLACA[X;Y]	5.2
RPLACD[X;Y]	5.1
RPLNODE[X;A;D]	5.2
RPLNODE2[X;Y]	5.2
RPLSTRING[X;N;Y] SUBR	10.5,8
RPT[RPTN;RPTF]	8.8
RPTQ[N;FORM1;FORM2;...;FORMn] NL*	8.8
RSH[N;M]	13.4
RSTRING[FILE;RDTBL] SUBR	14.12,13
SASSOC[KEY;ALST]	5.13
SAVEDEF[NAME;TYPE;DEFINITION]	8.6; 14.70
SAVEPUT[ATM;PROP;VAL]	7.2
SAVESET[NAME;VALUE;TOPFLG;FLG]	22.33,40
SAVESETQ[SETQX] NL*	22.33
SAVESETQQ[SETQX;SETQY] NL	22.33
SCODEP[FN] SUBR	3.14; 8.3,3
SCRATCHLIST[LST;X1;X2;...;Xn]	24.36
SEARCHPDL[SRCHFN;SRCHPOS]	12.12
SELECTQ[X;CLAUSE1;CLAUSE2;...;CLAUSEn;DEFAULT] NL*	5.3,4
SEPRCASE[CLISPFLG]	14.9
SET[X;Y] SUBR	5.5
SETA[A;N;V]	10.9; 3.5; 16.7
SETARG[VAR;M;X] FSUBR	8.9
SETATOMVAL[ATM;VAL] SUBR	5.6
SETBLIPVAL[BLIPTYP;IPOS;N;VAL] SUBR	12.4
SETBRK[LST;FLG;RDTBL] SUBR	14.13
SETD[A;N;V]	10.10; 3.5
SETDECLTYPEPROP[TYPE;PROP;VAL]	24.65
SETERRORN[NUM;MESS] SUBR	16.11
SETFILEINFO[FILE;ATTRIB;VALUE]	14.5
SETFILEPTR[FILE;ADR] SUBR	14.8,36
SETINITIALS[]	14.37
SETLINELENGTH[N]	14.35
SETN[VAR;X] FSUBR	13.10,8-9
SETPROPLIST[ATM;LST]	7.1; 2.3

Page
Numbers

SETQ[X;Y] FSUBR*	5.5
SETQQ[X;Y] NL	5.5
SETREADMACROFLG[FLG] SUBR	14.28
SETREADTABLE[RDTBL;FLG] SUBR	14.24
SETSBSIZE[N] SUBR	3.14; 16.8
SETSEPR[LST;FLG;RDTBL] SUBR	14.13
SETSTKARG[N;POS;VALUE] SUBR	12.8
SETSTKARGNAME[N;POS;NAME] SUBR	12.8
SETSTKNAME[POS;NAME] SUBR	12.7
SETSYNONYM[NEWPHRASE;MEANING]	20.19
SETSYNTAX[CH;CLASS;TABLE]	14.25
SETTEMPLATE[FN;TEMPLATE]	20.18
SETTERMCHARS[NEXTCHAR;BKCHAR;LASTCHAR; UNQUOTECHAR;2CHAR;PPCHAR]	14.36; 9.14; A1.3
SETTERMTABLE[TTBL] SUBR	14.29
SETTOPVAL[ATM;VAL] SUBR	5.6; 2.3
SETTYPEDESCRIPTION[TYPE;STRING]	10.13
SETWORDCONTENTS[PTR;N]	21.14
SHOULDNT[]	16.10
SHOWDEF[NAME;TYPE;FILE]	14.70
SHOWPRINT[X;FILE;RDTBL]	14.18; 15.8-9,19; 22.9
SHOWPRIN2[X;FILE;RDTBL]	14.18; 22.18,44
SIN[X;RADIANSLFG]	13.7
SINGLEFILEINDEX[FILE;OUTPUTFILE;NEWPAGEFLG]	24.31
SKOR[XWORD;TWORD;NCX;NCT;FLG]	17.16-17
SKREAD[FILE;REREADSTRING]	14.16
SMALLP[N]	13.4; 3.3; 13.1
SMARTARGLIST[FN;EXPLAINFLG;TAIL]	8.4
SMASHFILECOMS[FILE]	14.76
SOME[SOME[X];SOMEFN1;SOMEFN2]	5.12
SORT[DATA;COMPAREFN]	6.7
SPACES[N;FILE] SUBR	14.17
SPELLFILE[FILE;NOPRINTFLG;NSFLG]	17.21; 14.2; 16.7,10
SQRT[N]	13.7
STACKP[X] SUBR	12.11
STKAPPLY[POS;FN;ARGS;FLG;INTERNALFLG]	12.9
STKARG[N;POS] SUBR	12.8; 15.8
STKARGNAME[N;POS] SUBR	12.8
STKARGS[POS;NARGS]	12.9
STKEVAL[POS;FORM;FLG;INTERNALFLG]	12.9; 15.8
STKNAME[POS] SUBR	12.7
STKNARGS[POS] SUBR	12.8
STKNTH[N;IPOS;OPOS] SUBR	12.7
STKNTHNAME[N;POS] SUBR	12.7
STKPOS[FRAMENAME;N;IPOS;OPOS] SUBR	12.6
STKSCAN[VAR;IPOS;OPOS] SUBR	12.8
STORAGE[FLG;GCFLG]	10.14
STREQUAL[X;Y] SUBR	10.4
STRINGP[X] SUBR	5.9; 10.4
STRPOS[PAT;STRING;START;SKIP;ANCHOR;TAIL]	10.6; 14.8
STRPOSL[A;STR;START;NEG]	10.7
SUBATOM[X;N;M]	10.6
SUBLIS[ALST;EXPR;FLG]	6.5
SUBPAIR[OLD;NEW;EXPR;FLG]	6.5
SUBRP[FN] SUBR	8.3,1-3
SUBSET[MAPX;MAPFN1;MAPFN2]	11.2
SUBST[NEW;OLD;EXPR]	6.4,5

Page
Numbers

SUBSTRING[X;N;M;OLDPTR] SUBR	10.5; 3.6; 10.8
SUBSYS[FILE/FORK;INCOMFILE;OUTCOMFILE; ENTRYPOINTFLG]	21.7,8; 22.28
SUBTYPES[TYPE]	24.66
SUB1[X]	13.2
SUPERTYPES[TYPE]	24.66
SWPARRAY[N;P;V] SUBR	3.14
SWPARRAYP[X] SUBR	3.14; 10.9
SYNTAXP[CHARCODE;CLASS;TABLE]	14.25
SYSBUF[FLG] SUBR	14.34
SYSIN[FILE] SUBR	14.37; 2.9
SYSOUT[FILE] SUBR	14.36; 2.9; 14.38
SYSOUTP[FILE]	14.38
SYSTEMTYPE[] SUBR	21.9
TAB[POS;MINSPACES;FILE]	14.17
TAILP[X;Y]	5.12
TAN[X;RADIANSF LG]	13.8
TCOMPL[FILES]	18.5; 14.65; 18.6-7, 21-22
TCONC[PTR;X]	6.2,3
TELNET[CONNECTION;TYPE;SKT;BUTTONCHAR]	24.44; 21.15
TENEX[STR;FILEFLG]	21.9
TERMTABLEP[TTBL] SUBR	14.29
TERPRI[FILE] SUBR	14.17
TESTMODE[FLG]	22.32
TESTRELATION[ITEM;RELATION;ITEM2;INVERTED]	20.19
TIME[FORM;CNT] NL	21.1,2
TIMES[X1;X2;...;Xn] SUBR*	13.6
TRACE[X] NL*	15.16,1,5,13,17
TRANSORSET[]	24.24,21
TRYNEXT[PLST##;ENDFORM##;VAL##] NL	12.17
TTY#[]	24.43
TYPENAME[DATUM] SUBR	10.10
TYPENAMEFROMNUMBER[N]	10.11
TYPENAMEP[DATUM;TYPENAME]	10.11
TYPENUMBERFROMNAME[NAME]	10.11
TYPEP[DATUM;N]	10.11
TYPESOF[NAME;POSSIBLETYPES;IMPOSSIBLETYPES]	14.72
UNADVISE[X] NL*	19.5,4,6
UNBREAK[X] NL*	15.17,15,18; 21.5
UNBREAKIN[FN]	15.18,18
UNBREAKO[FN;TAIL]	15.18
UNDOLISPX[LINE]	22.42
UNDOLISPX1[EVENT;FLG;DWIMCHANGES]	22.42
UNDONLSETQ[FORM] NL	22.43,34
UNDOSAVE[UNDOFORM;HISTENTRY]	22.41,35
UNION[X;Y]	6.7
UNLOCKMAP[PTR]	21.15
UNMARKASCHANGED[NAME;TYPE]	14.67
UNPACK[X;FLG;RDTBL] SUBR	10.2
UNPACKFILENAME[FILENAME]	14.6
UNSAVEDEF[NAME;TYPE;DEF]	8.6; 14.71; 17.12-13
UNSAVEFNS[FNS]	20.21
UNSET[NAME]	22.41,33
UPDATECHANGED[]	20.20

Page
Numbers

UPDATEFILES[PRLST;FLST]	14.63
UPDATEFN[FN;EVENIFVALID]	20.20
USERDATATYPES[]	3.9
USEREXEC[LISPXID;LISPXXMACROS;LISPXXUSERFN]	22.37
USERLISPXPRINT[X;FILE;Z;NODOFLG]	22.30
USERNAME[A]	21.9
USERNUMBER[A;FLG]	21.9
U-CASE[X]	10.4; 9.50
U-CASEP[X]	10.4
VAG[X] SUBR	13.11.10
VALUEOF[X] NL*	22.26,40; 21.8; 22.35
VARIABLES[POS]	12.8; 15.9
VARS[FN;USEDATABASE]	14.39
VIRGINFN[FN;FLG]	15.19
WAITFORINPUT[FILE]	14.15
WHENCLOSE[FILENAME;PROP1;VAL1;...;PROPn;VALn] * ..	14.10; 24.34,49
WHEREIS[NAME;TYPE;FILES]	14.67; 24.34
WIDEAPER[FLG]	14.48
WORDCONTENTS[PTR]	21.14
WORDOFFSET[PTR;N]	21.14
WRITEFILE[X;FILE]	14.42
XWD[N1;N2]	24.36
ZEROP[X]	13.3
##[COMS] NL*	9.42,19
/DELFILE[FILE]	24.43
/RPLNODE[X;A;D]	22.41,42
/RPLNODE2[X;Y]	22.42
/UNDELFILE[FILE]	24.43

Index of Variables & Parameters

	Page Numbers
ABBREVLST (prettydef)	14.46,48
ADDSPELLFLG (dwim)	17.20,8,18
ADVINFOOLST (system)	19.5
ADVISEDFNS (system)	19.4-5
AFTERSYSOUTFORMS (system)	14.37,35
ALAMS (compiler)	18.4
APPROVEFLG (dwim)	17.3-6,19,21
ARCHIVEFLG (prog. asst.)	22.28
ARCHIVEFN (prog. asst.)	22.28,23
ARCHIVELST (prog. asst.)	22.34,39
ASKUSERTTBL (dwim)	17.23
BAKTRACECLST (break)	15.19
BEFORESYSOUTFORMS (system)	14.37
BLKAPPLYFNS (compiler)	18.14,19,21
BLKLIBRARY (compiler)	18.14,21
BREAKCOMSLST (break)	15.13
BREAKDELIMITER (break)	15.8
BREAKMACROS (break)	15.13-14
BREAKRESETFORMS (break)	15.14; 14.29
BRKCOMS (break)	15.13,7-8,14-15
BRKDWNNTYPE (system)	21.5
BRKDWNNTYPES (system)	21.5
BRKEXP (break)	15.4,5,8,10-12,14-15, 16.1,3
BRKFILE (break)	15.13
BRKFN (break)	15.14,6,15
BRKINFOOLST (break)	15.17-18
BRKTYPE (break)	15.14
BRKWHEN (break)	15.14-15
BROKENFNS (break)	15.15,17; 17.21
BUILDMAPFLG (system)	14.41; 18.7
CHANGECHAR (prettydef)	14.49; 9.25
CHANGESARRAY (system)	9.24
CLEANUOPTIONS (file package)	14.66
CLEARSTKLST (system)	12.11
CLISPARRAY (clisp)	23.57,22,60; 24.2
CLISPCHARRAY (clisp)	23.55
CLISPCHARS (clisp)	23.55
CLISPFLG (clisp)	23.56
CLISPFORWORDSPPLST (clisp)	23.11
CLISPHELPFLG (clisp)	23.57,53
CLISPIFTRANFLG (clisp)	23.59,22
CLISPIFWORDSPPLST (clisp)	23.10
CLISPIFYENGLSHFLG (clisp)	23.59,21,39
CLISPIFYPACKFLG (clisp)	23.59,39
CLISPIFYPRETTYFLG (prettydef)	23.59; 14.49,65
CLISPIFYUSERFN (clisp)	23.60,40
CLISPINFIXSPLST (clisp)	23.55,7
CLISPI.S.GAG (clisp)	23.19
CLISPRETRANFLG (clisp)	23.59,22
CLREMPARSFLG (clisp)	23.59,39
CL:FLG (clisp)	23.59,39
COMMENTFLG (prettydef)	14.48,44-45
COMMENTLINELENGTH (prettydef)	14.51

Page
Numbers

COMPILEHEADER (compiler)	18.5
COMPILETYPELST (compiler)	8.7; 18.4, 12
COMPILEUSERFN (compiler)	18.4, 9
COMPILE.EXT (compiler)	18.6
COREVALS (system)	18.27
COUTFILE (compiler)	18.33
DECLARETAGSLST (prettydef)	14.59
DEFAULTINITIALS (editor)	9.59
DFNFLG (system)	8.5; 5.6; 8.6; 14.38, 22.33, 40
DIRECTORIES (system)	17.21
DISPLAYTERMFLG (system)	21.15
DONTCOMPILEFNS (compiler)	18.6, 8, 21
DONTMOVETOPFLG (dwim)	17.7, 9
DWIMCHECKPROGLABELSFLG (dwim)	23.58, 41
DWIMCHECK#ARGSFLG (dwim)	23.58, 41
DWIMESSGAG (dwim)	23.58, 42
DWIMFLG (dwim)	17.20; 9.56, 59, 61, 17.3
DWIMIFYCOMPFLG (compiler)	18.5; 23.58; 18.6, 22, 23.42
DWIMLOADFNSFLG (dwim)	17.20, 16
DWIMUSERFORMS (dwim)	17.20, 11, 13-15; 24.33
DWIMWAIT (dwim)	17.20, 4-5; 22.31
EDITCHARACTERS (system)	14.36
EDITCOMSA (editor)	9.56, 57; 17.11, 14
EDITCOMSL (editor)	9.56, 57; 17.13-14
EDITHISTORY (editor)	22.34, 45, 36, 44
EDITLOADFNSFLG (editor)	9.59
EDITQUIETFLG (editor)	9.15
EDITRDTBL (system)	14.23
ENTRIES (compiler)	18.21
ERRORTYPELST (system)	16.9; 14.2
FASTYPEFLG (dwim)	17.17
FILELINELENGTH (file package)	14.64, 48
FILELST (file package)	14.63, 66; 17.21
FILEPKGCOMSPLST (prettydef)	14.60
FILEPKGFLG (file package)	14.54
FILEPKGTYPES (file package)	14.61
FILEPKG.SCRATCH (file package)	14.72
FILERDTBL (system)	14.16, 38, 42, 78
FIRSTCOL (prettydef)	14.48
FIRSTNAME (system)	22.47
FIXSPELLDEFAULT (dwim)	17.20, 4; 23.41
FIXSPELLREL (dwim)	17.20, 19
FONTCHANGEFLG (prettydef)	14.51
FONTDEFS (prettydef)	14.52
FONTDEFSVARS (prettydef)	14.52
FONTEXCAPECHAR (prettydef)	14.51
FONTPROFILE (prettydef)	14.51
FUNNYATOMLST (clisp)	23.60, 39
GAINSPACEFORMS (system)	21.16
GENNUM (system)	10.4

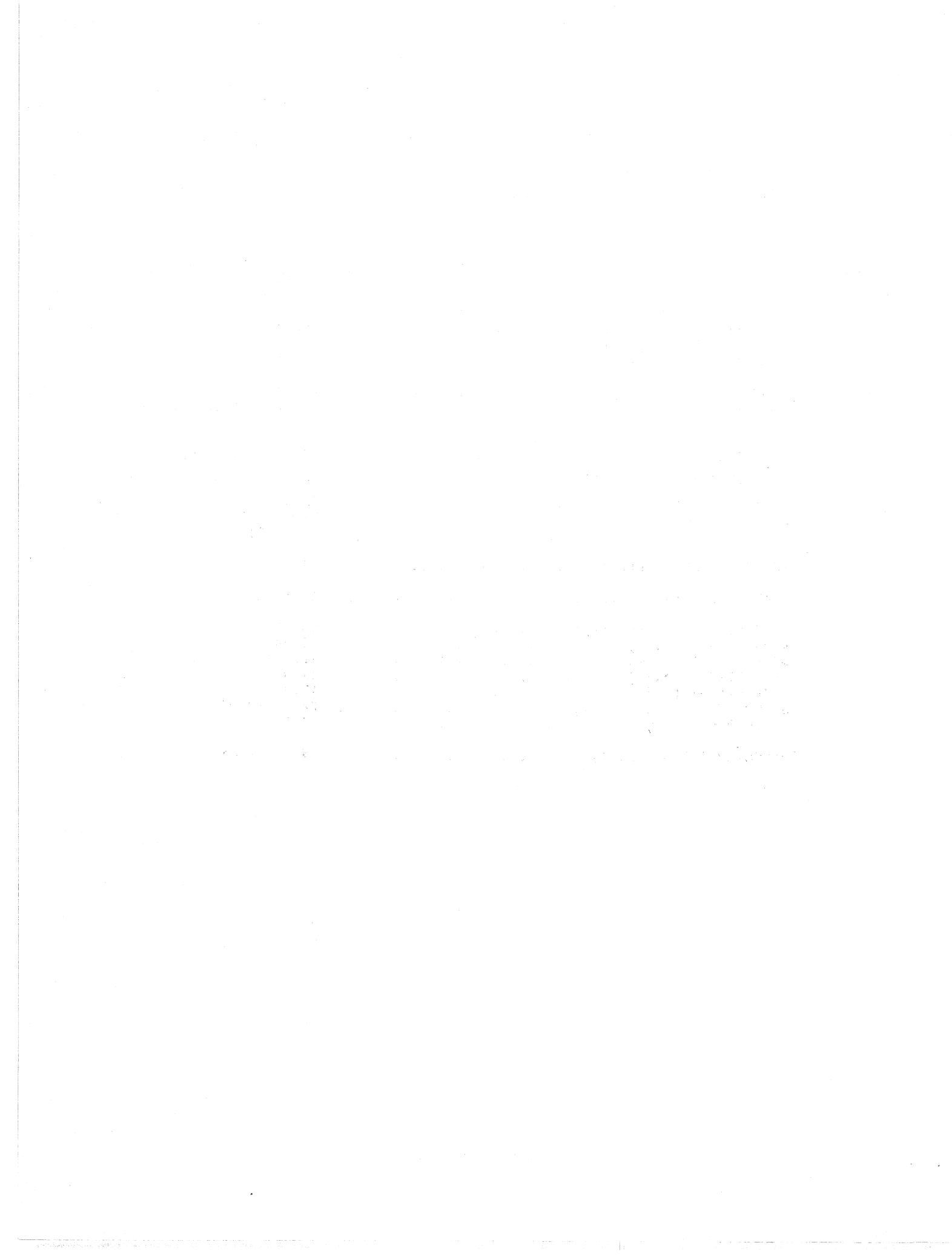
Page
Numbers

GLOBALVARS (compiler)	18.4, 21; 23.41
GREETDATES (system)	22.47
HELP CLOCK (system)	16.4; 22.17, 31
HELPDEPTH (system)	16.3-4
HELPFLAG (system)	16.3, 2, 5
HELPTIME (system)	16.3-4
HERALDSTRING (system)	14.37; 3.12
HISTORYCOMS (editor)	22.45
HISTORYSAVEFORMS (prog. asst.)	22.27
HISTSTRO (prog. asst.)	22.15, 37
INITIALS (editor)	9.59
INITIALSLST (editor)	9.59
IT (prog. asst.)	22.26, 18
JSYSES (system)	24.35
LAMBDA SPLST (dwim)	17.12; 8.4; 17.13-14, 24.33
LAMS (compiler)	18.3, 6
LAPFLG (compiler)	18.2
LASTAIL (editor)	9.11, 17, 63
LASTPOS (break)	15.6, 7-9, 11
LASTWORD (dwim)	17.9; 9.61; 17.18, 21, 23.8
LCASELST (prettydef)	14.46
LCFIL (compiler)	18.2-3
LINKEDFNS (system)	18.17
LINKFNS (compiler)	18.17, 21-22
LISPXCOMS (prog. asst.)	14.58; 22.31
LISPXFINDSPLST (prog. asst.)	22.12
LISPXHIST (prog. asst.)	22.35, 41, 43
LISPXHISTORY (prog. asst.)	22.34, 36, 43, 45
LISPXHISTORYMACROS (prog. asst.)	22.28
LISPXLINIE (prog. asst.)	22.28
LISPXMACROS (prog. asst.)	22.28; 21.8; 22.37
LISPXPRINTFLG (system)	22.31
LISPXREADFN (prog. asst.)	22.37; 14.15
LISPXUSERFN (prog. asst.)	22.29, 30, 36-37
LISTFILESTR (file package)	14.51, 66
LOADDBFLG (system)	24.32
LOADEDFILELST (file package)	14.63
LOADOPTIONS (system)	14.39
LOCALFREEVARS (compiler)	18.21
LOCALVARS (compiler)	18.18
LPARKEY (dwim)	17.5
LSTFIL (compiler)	18.2
MACSCRATCHSTRING (system)	21.3, 12
MAKEFILEFORM (file package)	14.66
MAKEFILEOPTIONS (file package)	14.65
MAKEFILEREMAKEFLG (file package)	14.77, 65
MAKESYSDATE (system)	3.11
MARKLST (editor)	9.22, 63
MAXLEVEL (editor)	9.16, 18
MAXLOOP (editor)	9.44

Page
Numbers

MAX.FLOAT (system)	13.6
MAX.INTEGER (system)	13.3
MIN.FLOAT (system)	13.6
MIN.INTEGER (system)	13.3
MKSWAPSIZE (overlay)	3.14
MODEL33FLG (dwim)	17.16
NILCOMS (file package)	14.68
NLAMA (compiler)	18.3
NLAML (compiler)	18.3
NLSETQGAG (system)	16.4, 11
NOBREAKS (break)	15.17
NOCLEARSTKLST (system)	12.11
NOFIXFNSLST (dwim)	23.57; 14.40; 23.41, 43
NOFIXVARSLST (dwim)	23.57; 14.40; 23.41, 43, 51
NOLINKFNS (compiler)	18.17, 21-22
NORMALCOMMENTSFLG (system)	14.45
NOSPELLFLG (dwim)	17.20; 23.57
NOSWAPFNS (overlay)	3.14
NOTCOMPILEDFILES (file package)	14.64, 66
NOTLISTEDFILES (file package)	14.64, 66
OKREEVALST (dwim)	17.6
OLDVALUE (system)	5.9
PLVLFILEFLG (system)	14.19
PRETTYEQUIVLST (prettydef)	14.49
PRETTYFLG (prettydef)	14.49, 64
PRETTYHEADER (prettydef)	14.79
PRETTYLCOM (prettydef)	14.48
PRETTYPRINTMACROS (prettydef)	14.49
PRETTYPRINTTYPEMACROS (prettydef)	14.49
PRETTYTABFLG (prettydef)	14.43
PRETTYTRANFLG (clisp)	23.60; 14.65; 23.22-23
PRINTMSG (system)	16.9
PROMPTCHARFORMS (prog. asst.)	22.27, 38
PROMPT#FLG (prog. asst.)	22.27, 38
PRXFLG (system)	10.1
READBUF (prog. asst.)	22.37-38
RECOMPILEDEFAULT (compiler)	18.8, 23
REDOCNT (prog. asst.)	22.13
REREADFLG (prog. asst.)	22.37, 39
RESETFORMS (system)	22.28; 14.35
RETFNS (compiler)	18.14, 19, 21
RPARKEY (dwim)	17.5
RUNONFLG (dwim)	17.19-20
SAVEDBFLG (system)	24.32
SHALL I LOAD (system)	17.13
SPECVARS (compiler)	18.18; 5.8; 18.19, 21
SPELLINGS1 (dwim)	17.8, 9, 14, 18
SPELLINGS2 (dwim)	17.8, 9, 13-14, 18
SPELLINGS3 (dwim)	17.8, 9, 12, 18; 22.40
STRF (compiler)	18.2, 3, 5
SVFLG (compiler)	18.2-3

	Page Numbers
SYSHASHARRAY (system)	7.4-5
SYSLINKEDFNS (system)	18.17
SYSOUTDATE (system)	14.37
SYSOUTFILE (system)	14.37
SYSOUTGAG (system)	14.37
SYSOUT.EXT (system)	14.37
SYSPRETTYFLG (system)	14.18; 15.8-9, 19; 22.9, 18.44
SYSPROPS (system)	7.3; 14.57
TESTMODEFLG (prog. asst.)	22.32
TREATASCLISPFLG (clisp)	23.51
TTYLINELENGTH (system)	14.35
UCASELST (prettydef)	14.46
UNDOLST (editor)	9.54, 47, 55, 63; 22.45
UNFIND (editor)	9.23, 17, 27-28, 30-33, 47-48, 53, 63
UPFINDFLG (editor)	9.29, 17-18
USEMAPFLG (system)	14.42
USERMACROS (editor)	9.46; 14.58
USERNAME (prog. asst.)	22.47
USERWORDS (dwim)	17.9; 9.59, 61; 17.18, 20-21
WHEREIS.HASH (system)	24.34
!VALUE (break)	15.5, 14
#CAREFULCOLUMNS (prettydef)	14.48
#RPARS (prettydef)	14.48
#SPELLINGS1 (dwim)	17.9
#SPELLINGS2 (dwim)	17.9
#SPELLINGS3 (dwim)	17.9
#UNDOSAVES (prog. asst.)	22.31, 41, 43
#USERWORDS (dwim)	17.9
COMMENTFLG (prettydef)	14.44; 9.41



MASTER INDEX

Names of functions are in upper case, followed by their arguments enclosed in square brackets [], e.g. ASSOC[KEY;ALST]. The FNTYP for SUBRs is printed in full; for other functions, NL indicates an NLAMBDA function, and * a nospread function, e.g. LISTFILES[FILES] NL* indicates that LISTFILES is an NLAMBDA nospread function. Words in upper case not followed by square brackets are other INTERLISP words (system parameters, property names, messages, etc.). Words and phrases in lower case are not formal INTERLISP words but are general topic references. Where there are multiple references for a single entry, the primary reference(s) appears first, and in bold-italic font.

	Page Numbers
(A e1 ... em) (edit command)	9.26,9
ABBREVLST (prettyprint variable/parameter)	14.46,48
ABS[X]	13.7
AC (in a lap statement)	18.28
AC (in an assemble statement)	18.32
access chain	12.5
ACCESSFNS (record package)	23.30-31
active frame	12.5
AC1	18.32,24,26
ADD (change word)	23.37
ADDBUFFER[TEMP;ERRORFLG]	21.13
ADDPROP[ATM;PROP;NEW;FLG]	7.2
addressable files	14.7-9
ADDSPELL[X;SPLST;N]	17.18,20
ADDSPELLFLG (dwim variable/parameter)	17.20,8,18
ADDSTATS[STATLST] NL*	22.46
ADDTOCOMS[COMS;NAME;TYPE]	14.75
ADDTOFILE[NAME;TYPE;FILE;NEAR]	14.75
ADDTOFILES?[NOASKSTR]	14.67
ADDTOSCRATCHLIST[VALUE]	24.36
ADDTOVAR[VAR;X1;X2;...;Xn] NL*	5.6
ADDVARS (file package command)	14.56
ADD1[X]	13.2
ADIEU[VAL##] *	12.17
advice	19.1,3
ADVICE (file package command)	14.58; 19.6
ADVICE (property name)	19.5-6
ADVINFOLST (system variable/parameter)	19.5
ADVISE[FN;WHEN;WHERE;WHAT]	19.4,3,5
ADVISE (file package command)	14.58; 19.6
ADVISED (property name)	19.4; 8.5
ADVISEDFNS (system variable/parameter)	19.4-5
ADVISEDUMP[X;FLG]	19.6
advising	19.1-6
ADV-PROG	19.3-4
ADV-RETURN	19.3-4
ADV-SETQ	19.3-4
AFTER (as argument to advise)	19.3-4,1
AFTER (as argument to breakin)	15.16,5,17
AFTER (in INSERT command) (in editor)	9.27
AFTER (in MOVE command) (in editor)	9.31
AFTER (prog. asst. command)	22.22,18,28
AFTERSYSOUTFORMS (system variable/parameter)	14.37,35
ALAMS (compiler variable/parameter)	18.4

	Page Numbers
ALIAS (property name)	15.15,18
alink	12.5
ALISTS (file package command)	14.56
ALL (in event specification)	22.12
ALLPROP (as value of DFNFLG)	14.57
ALONE (type of read-macro)	8.5; 5.6; 14.38; 22.40
ALPHORDER[A;B]	14.27
ALREADY UNDONE (printed by system)	6.8
ALWAYS (clisp iterative statement operator)	22.18,43
ALWAYS (type of read-macro)	23.12
AMAC (property name)	14.27
AMBIGUOUS DATA PATH (error message)	18.24-25
AMBIGUOUS RECORD FIELD (error message)	23.34
AMBIGUOUS (printed by dwim)	23.34
AMONG (Masterscope path option)	17.7
ANALYZE (Masterscope command)	20.15
AND[X1;X2;...;Xn] FSUBR*	20.9
AND (in event specification)	5.11
AND (in USE command)	22.12
ANTILOG[X]	22.13
ANY (in Decl package)	13.7
APPEND[X1;X2;...;Xn] *	24.60
APPLY[FN;ARGS] SUBR	6.1
apply format	8.7; 2.3; 11.1; 16.1,
APPLY*[FN;ARG1;ARG2;...;ARGn] SUBR*	18.14
apply format	2.3
APPROVEFLG (dwim variable/parameter)	8.8; 2.3; 11.1; 16.1,
APPROVEFLG (dwim corrections)	18.14
ARCCOS[X;RADIANSLG]	17.3-6,19
ARCCOS: ARG NOT IN RANGE (error message)	17.3-6,19,21
ARCHIVE (prog. asst. command)	13.8
ARCHIVEFLG (prog. asst. variable/parameter)	13.8
ARCHIVEFN (prog. asst. variable/parameter)	22.23
ARCHIVELST (prog. asst. variable/parameter)	22.28
ARCSIN[X;RADIANSLG]	22.28,23
ARCSIN: ARG NOT IN RANGE (error message)	22.34,39
ARCSIN: ARG NOT IN RANGE (error message)	13.8
ARCTAN[X;RADIANSLG]	13.8
ARCTAN2[X;Y;RADIANSLG]	13.8
ARG[VAR;M] FSUBR	8.9; 4.2
ARG NOT ARRAY (error message)	16.7; 10.9-10
ARG NOT LIST (error message)	16.5; 5.2,13; 6.3
ARG NOT LITATOM (error message)	16.6; 4.1; 5.4-6, 7.1-3; 8.2; 14.3
ARGLIST[FN]	8.4; 2.2; 8.1-3; 15.9
ARGNAMES (property name)	8.5
ARGS NOT AVAILABLE (error message)	8.4
ARGS (break command)	15.9,6
ARGTYPE[FN] SUBR	8.3,1-3
argument evaluation	4.1-2
argument list	8.1; 4.1
arithmetic functions	13.2-8
AROUND (as argument to advise)	19.3,4
AROUND (as argument to breakin)	15.16,5,17
ARRAY[N;P;V] SUBR	10.8; 3.5
array functions	10.8-10

	Page Numbers
array header	3.5; 10.8
array pointer	3.5
ARRAY (file package command)	14.59
ARRAYBEG[A] SUBR	10.9
ARRAYBLOCK (record package)	23.30
ARRAYP[X] SUBR	5.10; 10.9; 3.14
ARRAYRECORD (record package)	23.28
arrays	3.5, 1.9-10; 5.10
ARRAYS FOULED (error message)	3.10
ARRAYS FULL (error message)	16.7; 10.8
ARRAYSIZE[A]	10.9
ARRAYTYP[ARRAY]	10.9
AS (clisp iterative statement operator)	23.16
ASKUSER[WAIT;DEFAULT;MESS;KEYLST;TYPEAHEAD; LISPXPRNTFLG;OPTIONSLST;FILE]	17.22, 29, 23-28, 30
ASKUSERTTBL (dwim variable/parameter)	17.23
ASSEMBLE	18.24-27; 4.2; 13.10, 18.31-33
ASSEMBLE macros	18.25
ASSEMBLE statements	18.24-26
ASSERT (in Decl package)	24.59
assignments (in clisp)	23.8
assignments (in pattern match compiler)	24.5
ASSOC[KEY;ALST]	5.13
association list	12.1
ASSOCRECORD (record package)	23.28
ATOM[X] SUBR	5.9
atom hash table	10.10
ATOM HASH TABLE FULL (error message)	16.6
ATOM TOO LONG (error message)	16.6; 10.2, 6
ATOMRECORD (record package)	23.30
atoms	3.1, 9
ATTACH[X;L]	6.3
ATTEMPT TO BIND NIL OR T (error message)	16.8; 4.1; 5.4
ATTEMPT TO RPLAC NIL (error message)	16.5; 5.2; 6.3; 7.1
ATTEMPT TO SET NIL (error message)	16.5; 5.5
ATTEMPT TO USE ITEM OF INCORRECT TYPE (error message)	16.8
AUTOCOMPLETEFLG (askuser option)	17.28
AU-REVOIR[VAL##] *	12.17
AVOIDING (Masterscope path option)	20.15
a-list	8.8
A000n (gensym)	10.4
(B e1 ... em) (edit command)	9.26, 9
BACKTRACE[IPOS;EPOS;FLAGS;FILE;PRINTFN] SUBR	12.11
backtrace	15.8; 2.6; 12.2-3, 15.9, 19
backtracking	12.4
BAD ARGUMENT - FASSOC (error message)	5.13; 2.2
BAD ARGUMENT - FGETD (error message)	8.2
BAD ARGUMENT - FLAST (error message)	6.5; 2.2
BAD ARGUMENT - FLENGTH (error message)	6.6; 2.2
BAD ARGUMENT - FMEMB (error message)	5.12; 2.2
BAD ARGUMENT - FNTH (error message)	6.6; 2.2
BAD FILE NAME (error message)	16.8
BAD FILE PACKAGE COMMAND (error message)	14.60

	Page Numbers
BAD PROG BINDING (compiler error message)	18.35
BAD SETQ (compiler error message)	18.34
BAD SYSOUT FILE (error message)	16.7
BAKTRACE[IPOS;EPOS;SKIPFN;FLAGS;FILE]	15.19
BAKTRACELST (break variable/parameter)	15.19
basic frame	12.5, 1.8
BCOMPL[FILES;CFILE;NOBLOCKSFLG]	18.22; 14.65; 18.19-21, 23
BEFORE (as argument to advise)	19.3, 4
BEFORE (as argument to breakin)	15.16, 5, 17
BEFORE (in MOVE command) (in editor)	9.31
BEFORE (prog. asst. command)	22.22, 18, 28
BEFORESYSOUTFORMS (system variable/parameter) ...	14.37
bell (in history event)	22.26, 18, 34, 39
bell (printed by system)	10.13; 14.19; 16.2, A1.1
..... bells (printed by dwim before an interaction)	17.4
..... bells (printed by system)	A1.1
(BELOW com x) (edit command)	9.20
(BELOW com) (edit command)	9.21
BETWEEN (record field type)	23.29
(BF pattern T) (edit command)	9.18
BF (edit command)	9.18, 8
(BI n m) (edit command)	9.34, 6
(BI n) (edit command)	9.34
BIND (clisp iterative statement operator)	23.13
BIND (Masterscope relation)	20.5
BIND (Masterscope template)	20.17
(BIND . coms) (edit command)	9.46
bindings in a basic frame	12.8
BINDS (as value of INFO property)	23.40
BIT[BIT#;WORD]	24.36
BITS (as a field specification)	3.8
BITS (record field type)	23.29
(BK n) (n a number, edit command)	9.13
BK (edit command)	9.12, 7, 13
BKLINBUF[X] SUBR	14.34
BKSYSBUF[X] SUBR	14.34; 21.9
blink	12.5
blip functions	12.4
blips	12.4
BLIPSCAN[BLIPTYP;IPOS] SUBR	12.4
BLIPVAL[BLIPTYP;IPOS;FLG] SUBR	12.4
BLKAPPLY[FN;ARGS] SUBR	18.14
BLKAPPLYFNS (compiler variable/parameter)	18.14, 19.21
BLKAPPLY*[FN;ARG1;ARG2;...;ARGn] SUBR*	18.14
BLKLIBRARY (compiler variable/parameter)	18.14, 21
BLKLIBRARYDEF (property name)	18.14; 22.42, 46
block compiler	18.19-23
block compiling	18.13-23
block declarations	18.20; 14.58; 18.21-22
block library	18.14
BLOCKCOMPILE[BLKNAME;BLKFNS;ENTRIES;FLG]	18.19, 20
BLOCKED (printed by editor)	9.55
BLOCKS (file package command)	14.58; 18.20-21
(BO n) (edit command)	9.34, 6
BOTH (Masterscope template)	20.17

Page
Numbers

BOTTOM (as argument to advise)	19.3-4
BOUNDBIN (in Decl package)	24.58
BOUNDP[VAR]	8.7
box	13.10
BOXCOUNT[TYPE;N] SUBR	21.3
boxed numbers	13.1
BOXED (edita parameter)	24.12
boxing	13.1,2,8-10
Boyer-Moore fast string searching algorithm	14.9
BR (exec command)	24.41
BREAK[X] NL*	15.16,1,5,17
break characters	14.26,13-15,33
break commands	15.5-13
break expression	15.4,11
BREAK INSERTED AFTER (typed by breakin)	15.17
break package	15.1-21
BREAK WITHIN A BREAK (printed by system)	15.4,14
BREAK (error message)	16.6
BREAKCHECK[ERRORPOS;ERXN]	16.3,2,4-5,9-10; 17.10
BREAKCOMSLST (break variable/parameter)	15.13
BREAKDELIMITER (break variable/parameter)	15.8
BREAKDOWN[FNS] NL*	21.4-6
BREAKIN[FN;WHERE;WHEN;BRKCOMS] NL	15.16-17,1-2,5,15,18
breaking clisp expressions	15.1
BREAKLINKS[]	24.43
BREAKMACROS (break variable/parameter)	15.13-14
BREAKREAD[TYPE]	15.18
BREAKRESETFORMS (break variable/parameter)	15.14; 14.29
BREAK0[FN;WHEN;COMS;BRKFN;TAIL]	15.15,16-18
BREAK1[BRKEXP;BRKWHEN;BRKFN;BRKCOMS;BRKTYPE] NL	15.3,14,1-2,4-13,15-17, 16.1-2,4,10; 17.21 18.23; 14.65-66, 18.19-22
BRECOMPILE[FILES;CFILE;FNS;NOBLOCKSFLG]	15.13,7-8,14-15
BRKCOMS (break variable/parameter)	21.4
BRKDWNRESULTS[RETURNVALUESFLG]	21.5
BRKDWNTYPE (system variable/parameter)	21.5
BRKDWNTYPES (system variable/parameter)	15.4,5,8,10-12,14-15, 16.1,3
BRKEXP (break variable/parameter)	15.13
BRKFILE (break variable/parameter)	15.14,6,15
BRKFN (break variable/parameter)	15.15,17-18
BRKINFO (property name)	15.17-18
BRKINFOLST (break variable/parameter)	15.14
BRKTYPE (break variable/parameter)	15.14-15
BRKWHEN (break variable/parameter)	23.54
BROADSCOPE (property name)	16.3
(BROKEN) (printed by system)	15.3
BROKEN (printed by system)	15.15; 8.5
BROKEN (property name)	15.15,17; 17.21
BROKEN-IN (property name)	15.17; 8.5; 15.18
BT (break command)	15.8; 2.6; 15.6
BTV (break command)	15.9,6
BTVI (break command)	15.9
BTV* (break command)	15.9,6
BTV+ (break command)	15.9
BUILDMAPFLG (system variable/parameter)	14.41; 18.7

	Page Numbers
BY (clisp iterative statement operator)	23.15,14,17
BY (in REPLACE command) (in editor)	9.27
BY (Masterscope set specification)	20.7
C (in an assemble statement)	18.26
C (makefile option)	14.65
CALL DIRECTLY (Masterscope relation)	20.5
CALL FOR EFFECT (Masterscope relation)	20.5
CALL FOR VALUE (Masterscope relation)	20.5
CALL INDIRECTLY (Masterscope relation)	20.5
CALL SOMEHOW (Masterscope relation)	20.4
CALL (Masterscope relation)	20.4
CALL (Masterscope template)	20.17
CALLS[FN;USEDATABASE]	20.18
CALLSCCODE[FN]	20.18
CAN'T BE BOTH AN ENTRY AND THE BLOCK NAME (compiler error message)	18.34,20
CAN'T - AT TOP (printed by editor)	9.12,3
CAP (edit command)	9.50
CAR[X] SUBR	5.1
car of NIL	2.3
carriage-return	2.4; 3.1; 14.11-12, 15-17,30
carriage-return (edita command)	24.11,10
CAUTIOUS (DWIM mode)	17.3,17,21; 23.3,52
CBOX[X;Y]	24.37
CCODEP[FN] SUBR	8.3; 3.13-14; 8.1-3
CDR[X] SUBR	5.1
CEXPR (function type)	4.2; 8.3
CEXPR* (function type)	4.2; 8.3-4
CFEXPR (function type)	4.2; 8.3-4
CFEXPR* (function type)	4.2; 8.3-4
CHANGE (change word)	23.38
(CHANGE @ TO ...) (edit command)	9.27
CHANGECALLERS[OLD;NEW;TYPES;FILES;METHOD]	14.71
CHANGECHAR (prettyprint variable/parameter)	14.49; 9.25
CHANGEFONT[FONTCLASS]	14.52,50
CHANGENAME[FN;FROM;TO]	9.64; 15.18
CHANGEPROP[X;PROP1;PROP2]	7.3
CHANGESARRAY (system variable/parameter)	9.24
CHANGESLICE[N;HISTORY;L]	22.40,7
CHANGETRAN (in record package)	23.36-38
changing record declarations	23.35
CHARACTER[N] SUBR	10.3
character atoms	10.2
character codes	10.3
CHARDELETE (syntax class)	14.30-31
CHCON[X;FLG;RDTBL] SUBR	10.3
CHCON1[X] SUBR	10.3
CHECK (Masterscope command)	20.11
CHECKCONNECTION[CONNECTION]	24.47
CHECKNIL[]	22.38
CHOOZ[XWORD;REL;SPLST;TAIL;FN;TIEFLG;NDBLS;CLST]..	17.16
CIRCLMAKER[LIST]	24.19
CIRCLPRINT[LIST;PRINTFLG;RLKNT]	24.18,17
cjsys package	24.35-36
CL (edit command)	9.52; 23.60

	Page Numbers
CLDISABLE[OP]	23.59
CLEANPOSLST[PLST]	12.17
CLEANUP[FILES] NL*	14.66
CLEANUOPTIONS (file package variable/parameter)	14.66
CLEARBUF[FILE;FLG] SUBR	14.34; 22.26
clearing input buffer	2.4; 14.19, 32
clearing output buffer	2.4; 14.19
CLEARMAP[FILE;PAGES;RELEASE]	21.14
CLEARSTK[FLG] SUBR	12.11
CLEARSTKLST (system variable/parameter)	12.11
clink	12.5
CLISP	23.1; 11.3; 17.11-14, 23.2-61
CLISP and compiler	18.4-5
(CLISP declarations . form)	23.22
CLISP interaction with user	23.52
CLISP internal conventions	23.53
CLISP operation	23.51-52
CLISP (Masterscope template)	20.17
CLISPARRAY (clisp variable/parameter)	23.57,22,60; 24.2
CLISPBRACKET (property name)	23.55
CLISPCHARRAY (clisp variable/parameter)	23.55
CLISPCHARS (clisp variable/parameter)	23.55
CLISPDEC[DECLST]	23.58,24
CLISPFLG (clisp variable/parameter)	23.56
CLISPFONT	14.50
CLISPFORWORDSPLST (clisp variable/parameter)	23.11
CLISPHELPFLG (clisp variable/parameter)	23.57,53
CLISPIFTRANFLG (clisp variable/parameter)	23.59,22
CLISPIFWORDSPLST (clisp variable/parameter)	23.10
CLISPIFY[X;L]	23.38,58; 14.65; 23.25, 39-40
CLISPIFY (makefile option)	14.65; 23.40,60
CLISPIFYENGLSHFLG (clisp variable/parameter)	23.59,21,39
CLISPIFYFNS[FNS] NL*	23.59
CLISPIFYPACKFLG (clisp variable/parameter)	23.59,39
CLISPIFYPRETTYFLG (prettyprint variable/parameter)	23.59; 14.49,65
CLISPIFYUSERFN (clisp variable/parameter)	23.60,40
CLISPINFIX (property name)	23.54
CLISPINFIXSPLST (clisp variable/parameter)	23.55,7
CLISPI.S.GAG (clisp variable/parameter)	23.19
clisprecordtypes	23.31
CLISPRETRANFLG (clisp variable/parameter)	23.59,22
CLISPTRAN[X;TRAN]	23.57
CLISPTYPE (property name)	23.53-54
CLISPWORD (property name)	23.54; 17.12
CLISP%	23.23,60
CLISP: (edit command)	23.60,22
CLOCK[N] SUBR	21.3
CLOSEALL[]	14.5,10
CLOSECONNECTION[CONNECTION]	24.46
CLOSEF[FILE]	14.5
CLOSEF?[FILE]	14.5
CLOSEHASHFILE[HASHFILE]	24.49
CLOSER[A;X] SUBR	10.15
closing and reopening files	14.9-11
CLREMPARSFLG (clisp variable/parameter)	23.59,39

	Page Numbers
CLRHASH[HARRAY] SUBR	7.4
CL:FLG (clisp variable/parameter)	23.59,39
CNDIR[DIR;PASSWORD]	24.43
CNTRLV (syntax class)	14.30
CODE (property name)	8.6; 3.14
COLLECT (clisp iterative statement operator)	23.12
collecting (printed by system)	10.11
COM (as suffix to file name)	18.6,22
commands that move parentheses (in editor)	9.33-35
comment pointers	9.53; 14.45
COMMENT USED FOR VALUE (compiler error message) ..	18.34
COMMENTFLG (prettyprint variable/parameter)	14.48,44-45
COMMENTFONT	14.50
COMMENTLINELENGTH (prettyprint variable/parameter)	14.51
comments (in listings)	14.44-46
COMMENT1[L;INBLOCKFLG]	14.80
compacting	3.10
COMPARE[NAME1;NAME2;TYPE;SOURCE1;SOURCE2]	14.71
COMPAREDEFS[NAME;TYPE;SOURCES]	14.72
COMPARELISTS[X;Y]	6.9
COMPILE[X;FLG]	18.5
compiled code	10.8
compiled file	18.6,7
compiled functions	4.2
COMPILED ON	18.5
COMPILEFILES[FILES] NL*	14.66
COMPILEHEADER (compiler variable/parameter)	18.5
COMPILEIGNOREDECL (Decl package parameter)	24.60
compiler	18.1; 4.2; 18.2-36
compiler error messages	18.33-36
compiler functions	18.5,6-8,19-20,22-23
compiler macros	18.10-12
compiler printout	18.33
compiler questions	18.2-3
compiler structure	18.23
COMPILETYPELST (compiler variable/parameter)	8.7; 18.4,12
COMPILEUSERFN (compiler variable/parameter)	18.4,9
COMPILEUSERFN (use by clisp)	23.42
COMPILE.EXT (compiler variable/parameter)	18.6
COMPILE1[FN;DEF;COREFLG]	18.5
compiling by datatype	18.12
compiling CLISP	23.42; 18.4-5
compiling files	18.5,7,22
compiling FUNCTION	18.13
compiling NLAMBDAbs	18.3-4
COMPLETEON (askuser option)	17.27
COMPSET[FILE;FLG;FILES]	18.2
computed macros	18.10
(COMS x1 ... xn) (edit command)	9.43
COMS (file package command)	14.57
(COMSQ . coms) (edit command)	9.43
CONCAT[X1;X2;...;Xn] SUBR*	10.5; 3.6; 10.8
COND[C1;C2;...;Cn] FSUBR*	5.3; 4.3
cond clause	5.3
CONFIRMF LG (askuser option)	17.26
conjunctions (in Masterscope)	20.9
CONN (exec command)	24.42

	Page Numbers
CONS[X;Y] SUBR	5.1; 3.5,9
cons algorithm	5.1
CONSCOUNT[N] SUBR	5.1; 10.14; 21.3
CONSTANT[X]	18.11
constants in compiled code	18.11
constructing lists (in clisp)	23.10
CONTAIN (Masterscope relation)	20.5
context switching	12.6
CONTIN (prog. asst. command)	22.25; 21.8; 22.28
CONTINUE SAVING? (printed by system)	22.41,31
CONTINUE WITH T CLAUSE (printed by dwim)	17.6
CONTINUE (TENEX command)	2.3,9; 21.4; A1.1
continuing an edit session	9.48-49
CONTROL[FLG;RDTBL] SUBR	14.33; 2.4; 14.12,14,32
control chain	12.5
control character echoing	14.30
control characters	A1.1; 2.3-4; A1.2-4
control-A	2.4; 14.32,11-14,29-31, 33; A1.2
control-A (TOPS-20) (edit command)	9.14; A1.3
control-B	16.2,3,5-6; 21.3; A1.1
control-C	2.3; A1.1; 21.7
control-D	2.3; A1.1; 9.47, 14.34-35; 15.4,14, 16.2,4,11; 18.4-5, 21.3; 22.26
control-E	16.2; 9.2; 14.34; 15.4, 17; 16.10; 17.4-5,10, 21.3; 22.26; A1.1
control-E (typed to edita)	24.10
control-F (in file name)	14.2
control-G (use in history list)	22.26,18
control-H	16.2; 10.14; 14.34, 15.14; A1.1
control-L	23.46
control-L (TOPS-20) (edit command)	9.14; A1.3
control-N (TOPS-20)	2.4; 22.27; A1.2
control-O	2.4; 14.19; A1.2
control-P	14.19,34; 15.9; A1.2
control-Q	2.4; 14.32,11-14,29-31, 33; A1.2
control-R	A1.2; 14.30
control-S	10.13; 14.34; A1.2
control-T	A1.2
control-U	2.4; 22.27,37; A1.2
control-U (TOPS-20)	2.4; 14.30; A1.2
control-V	A1.2; 2.4; 14.11,13-14, 30
control-W	2.4; 14.12; A1.2
control-X (edit command)	9.14; 14.36; A1.3
control-X (TOPS-20)	10.13; A1.2
control-Y (as a read-macro)	14.27,36; A1.3
control-Z (edit command)	9.14; 14.36; A1.3
control-Z (TOPS-20)	2.4; 14.14,32; A1.1
COPY[X]	6.4
copy	6.4,1,5
COPY (DECLARE: option)	14.59

	Page Numbers
COPYALL[X]	6.4; 23.33
COPYALLBYTES[FROMFILE;TOFILE;BYTESIZE]	24.43
COPYARRAY[AR]	10.10
COPYBYTES[SRCFIL;DSTFIL;START;END]	14.9
COPYDEF[OLD;NEW;TYPE;SOURCE;OPTIONS]	14.70
COPYHASHFILE[HASHFILE;NEWNAME;FN;VTYPE]	24.50
COPYING (record package)	23.33
COPYREADTABLE[RDTBL] SUBR	14.24
COPYSTK[POS1;POS2] SUBR	12.11
COPYTERMTABLE[TTBL] SUBR	14.29
COPYWHEN (DECLARE: option)	18.7; 14.59
COREVAL[X] NL	21.3
COREVAL (property name)	18.27,28-29; 24.10-11
COREVALS	18.27
COREVALS (system variable/parameter)	18.27
COROUTINE[CALLPTR##;COROUTPTR##;COROUTFORM##; ENDFORM##] NL	12.15
coroutines	12.15,4.13
COS[X;RADIANSLG]	13.7
COUNT[X]	6.6
COUNT (clisp iterative statement operator)	23.12
COUNTDOWN[X;N]	6.7
COUTFILE (compiler variable/parameter)	18.33
COVERS[HI;LO]	24.65
CQ (in an assemble statement)	18.26
CREATE NOT DEFINED FOR THIS RECORD (error message)	23.30
CREATE (Masterscope relation)	20.5
CREATE (Masterscope template)	20.17
CREATE (record package)	23.31-33
CREATEHASHFILE[FILE;VALUETYPE;ITEMLENGTH; #ENTRIES]	24.48
CTRLV (syntax class)	14.30
curly brackets (use with ftp package)	24.45
current declaration context	24.66
current expression (in editor)	9.2,3,6,8-14,16-23
CURRENTFN (transor variable)	24.24
CV (assemble macro)	24.36
CV2 (assemble macro)	24.36
D (edit command)	9.54
DA (exec command)	24.41
DATA TYPES FULL (error message)	16.8
databasefns package	24.31-32
DATATYPE (record package)	23.29
data-paths (in records in clisp)	23.34,26
data-types	3.1,2-7,9
DATE[FORMATBITS] SUBR	21.2
DATEFORMAT[KEYWORD1;KEYWORD2;...;KEYWORDn]	24.40
dateformat package	24.40-41
DATUM OF INCORRECT TYPE (error message)	3.8
DATUM (use in changetran constructs)	23.38
DCHCON[X;SCRATCHLIST;FLG;RDTBL]	10.3
DDT[] SUBR	24.9
debugging	15.1; 2.6; 12.2
Decl package	24.53-68
DECL (in Decl package)	24.59
declaration fault (in Decl package)	24.54

Page
Numbers

DECLARATION NOT SATISFIED (error message)	24.54
declarations (in clisp)	23.24,8,10,25
declarations (in pattern match compiler)	24.6
DECLARE	18.10; 14.58; 18.21
DECLARE AS LOCALVAR (Masterscope relation)	20.6
DECLARE AS SPECVAR (Masterscope relation)	20.6
DECLARE (clisp iterative statement operator)	23.17
DECLAREDATATYPE[TYPENAME;FIELD_SPECS;FLG]	3.7; 23.29
DECLARETAGSLST (prettyprint variable/parameter)	14.59
DECLARE:[X] NL*	14.59
DECLARE: (clisp iterative statement operator) ...	23.16
DECLARE: (file package command)	14.58,59
DECLARE: (in compiler)	18.6,6
DECLOF[FORM]	24.66
DECLOF (property name)	24.66,66
DECLTYPE[TYPENAME;TYPEEXPRESSION;PROP1;VAL1;...; PROPN;VALn]	24.63
DECLTYPES (file package command)	24.65
decltypes (in Decl package)	24.53
DEFAULTFONT	14.50
DEFAULTINITIALS (editor variable/parameter)	9.59
DEFAULTMAKENEWCOM[NAME;TYPE;LISTNAME;FILE]	14.72
DEFERREDCONSTANT[X]	18.12
DEFEVAL[TYPE;FN]	8.7
DEFINE[X;TYPE-IN]	8.6; 2.5
DEFINED, THEREFORE DISABLED IN CLISP (error message)	23.11
DEFINEQ[X1;X2;...;Xn] NL*	8.5; 2.5-6
defining file package commands	14.73-75
defining file package types	14.72-73
defining new iterative statement operators	23.19-21
DEFLIST[L;PROP]	7.3
DEFPRINT[TYPE;FN]	14.22
DEL (exec command)	24.42
DELDEF[NAME;TYPE]	14.70
DELETE (edit command)	9.26,9,24
(DELETE . @) (edit command)	9.28
DELETECHAR (syntax class)	14.30,29
DELETECONTROL[TYPE;MESSAGE;TTBL]	14.31
DELETELINE (syntax class)	14.30,29
DELFILE[FILE]	14.5
DELFROMCOMS[COMS;NAME;TYPE]	14.76
DELFROMFILES[NAME;TYPE;FILES]	14.75
DELNOTE (transor command)	24.28
DELPAGE[PAGE#;HASHFILE]	24.52
DELVER (exec command)	24.42
DESCRIBE (Masterscope command)	20.11
DESTINATION IS INSIDE EXPRESSION BEING MOVED (printed by editor)	9.31
destructive functions	6.3-4
DET (exec command)	24.41
DETACH[]	24.43
DETACHEDP[]	24.43
determiners (in Masterscope)	20.8
DFNFLG (system variable/parameter)	8.6; 5.6; 8.6; 14.38, 22.33,40
DIFFERENCE[X;Y]	13.6

	Page Numbers
DIFFERENT EXPRESSION (printed by editor)	9.55
DIR (prog. asst. command)	21.11; 22.28
DIRECTORIES (system variable/parameter)	17.21
DIRECTORY[FILEGROUP;COMMANDS;DEFAULTTEXT; DEFAULTVERS;LISPXPRNTFLG]	21.10
disabling CLISP operators	23.59
DISMISS[N]	21.3
DISPLAYTERMFLG (system variable/parameter)	21.15
DISPLAYTERMP[]	21.16
DLAMBDA (in Decl package)	24.55,54
DMPHASH[ARRAYNAME1;...;ARRAYNAMEn] NL*	7.5
DO (clisp iterative statement operator)	23.12
DO (edit command)	9.52; 22.45
DOBE[]	14.18
DOCOLLECT[ITEM;LST]	6.3
DOCOPY (DECLARE: option)	14.59
DOEVAL@COMPILE (DECLARE: option)	18.7; 14.59
DOEVAL@LOAD (DECLARE: option)	14.59
DONTCOMPILEFNS (compiler variable/parameter)	18.6,8,21
DONTCOPY (DECLARE: option)	18.7; 14.59
DONTEVAL@COMPILE (DECLARE: option)	14.59
DONTEVAL@LOAD (DECLARE: option)	14.59
DONTMOVETOPFLG (dwim variable/parameter)	17.7,9
dot notation	2.1
DOTHESE (transor command)	24.29
DOTHIS (transor command)	24.29
dotted pair	5.1
DPROG (in Decl package)	24.55,57
DREMOVE[X;L]	6.3
DREVERSE[L]	6.4
DRIBBLE[FILE;APPENDFLG;THAWEDFLG]	21.15
DRIBBLEFILE[] SUBR	21.15
DSK (exec command)	24.42
DSKSTAT[DIR;PRINTIFOVER;PRINTSYS;PRINTDEL; PRINTOLD]	24.43
DSUBLIS[ALST;EXPR;FLG]	6.5
DSUBST[NEW;OLD;EXPR]	6.4,5
DUMMYFRAMEP[POS]	12.7
DUMP (transor set command)	24.26
DUMPDATABASE[FNLST]	20.20
DUMPDB[FILE]	24.32
dumping circular lists	14.22
dumping unusual data structures	14.22
DUNPACK[X;SCRATCHLIST;FLG;RDTBL]	10.3
DW (edit command)	9.53; 17.18; 23.61
DWIM[X]	17.17,3
DWIM	17.1-30; 2.6; 16.1
DWIM interaction with user	17.3
DWIM variables	17.15
DWIM (prog. asst. command)	22.20-21
DWIMCHECKPROGLABELSFLG (dwim variable/parameter)	23.58,41
DWIMCHECK#ARGSFSLG (dwim variable/parameter)	23.58,41
DWIMESSGAG (dwim variable/parameter)	23.58,42
DWIMFLG (dwim variable/parameter)	17.20; 9.56,59,61, 17.3
DWIMIFY[X;QUIETFLG;L]	17.17; 23.40,57,41-42, 51,58

Page
Numbers

dwimify (printed by dwim)	23.42
DWIMIFYCOMPFLG (compiler variable/parameter)	18.5; 23.58; 18.6,22, 23.42
DWIMIFYFNS[FNS] NL*	23.58,41
DWIMLOADFNSFLG (dwim variable/parameter)	17.20,16
DWIMLOADFNS?[]	17.16
DWIMUSERFORMS (dwim variable/parameter)	17.20,11,13-15; 24.33
DWIMWAIT (dwim variable/parameter)	17.20,4-5; 22.31
(E x T) (edit command)	9.42
(E x) (edit command)	9.42
E (edit command)	9.41,7,42; 22.45
E (file package command)	14.57
E (in a floating point number)	3.4; 14.12
E (in an assemble statement)	18.26
E (use in comments)	14.49
EACHTIME (clisp iterative statement operator)	23.16,17
ECHOCONTROL[CHAR;MODE;TTBL]	14.30
echoing	14.30
ECHOMODE[FLG;TTBL] SUBR	14.30
edit chain	9.3,5,8-14,16-23
edit commands that search	9.14-22
edit commands that test	9.43
edit macros	9.45-46
EDIT WHERE (Masterscope command)	20.11
EDIT (break command)	15.10,6,11-12
EDIT (Masterscope command)	20.11
EDIT (printed by editor)	9.62
EDIT (transorset command)	24.26
EDITA[EDITARRY;COMS]	24.8-15
EDITCALLERS[ATOMS;FILES;COMS]	9.60
EDITCHARACTERS (system variable/parameter)	14.36
EDITCOMSA (editor variable/parameter)	9.56,57; 17.11,14
EDITCOMSL (editor variable/parameter)	9.56,57; 17.13-14
EDITDATE[OLDATE;INITLS]	9.59
EDITDATE?[COMMENT]	9.59
EDITDEF[NAME;TYPE;SOURCE;EDITCOMS]	14.70
EDITDEFAULT	9.56,57-58; 17.3, 22.45
EDITE[EXPR;COMS;ATM;TYPE;IFCHANGEDFN]	9.61,1,61
EDITF[NAME;COM1;COM2;...;COMn] NL*	9.58,1,59,61
EDITFINDP[X;PAT;FLG]	9.63
EDITFNS[NAME;COM1;COM2;...;COMn] NL*	9.59,60
EDITFPAT[PAT;FLG]	9.63
EDITHISTORY (editor variable/parameter)	22.34,45,36,44
editing arrays	24.8-15
editing compiled code	9.64; 15.18; 24.8-15
EDITL[L;COMS;ATM;MESS;EDITCHANGES]	9.62
EDITLOADFNSFLG (editor variable/parameter)	9.59
EDITLOADFNS? [FN;STR;ASKFLG;FILES]	9.64
EDITL0[L;COMS;MESS;EDITLFLG]	9.63
EDITP[NAME;COM1;COM2;...;COMn] NL*	9.61,1
EDITQUIETFLG (editor variable/parameter)	9.15
EDITTRACEFN	9.64,65
EDITRDTBL (system variable/parameter)	14.23
EDITREC[EDITRECX] NL*	23.35
EDITUSERFN[COM]	9.56

	Page Numbers
EDITV[NAME;COM1;COM2;...;COMn] NL*	9.61,1
EDIT-SAVE (property name)	9.47-48
EDIT4E[PAT;X;CHANGEFLG]	9.63
EF (edit command)	9.50
EFFECT (Masterscope template)	20.16
element patterns (in pattern match compiler)	24.2-4
ELT[A;N] SUBR	10.9; 3.5; 16.7
ELTD[A;N] SUBR	10.10; 3.5
(EMBED @ IN ...) (edit command)	9.31
END OF FILE (error message)	16.6; 14.7,11
ENDCOLLECT[LST;TAIL]	6.3
ENDFILE[FILE]	14.42
end-of-line	14.11; 3.1; 14.8,17
english phrases in clisp	23.21
ENTRIES (compiler variable/parameter)	18.21
entries (to a block)	18.13,19-20
ENTRY#[HIST;X]	22.40
ENVAPPLY[FN;ARGS;APOS;CPOS;AFLG;CFLG] SUBR	12.9
ENVEVAL[FORM;APOS;CPOS;AFLG;CFLG] SUBR	12.9
EOL (syntax class)	14.30
EP (edit command)	9.50
EQ[X;Y] SUBR	5.10; 2.2
eq	2.2
EQLENGTH[X;N]	6.6
EQMEMB[X;Y]	5.12
EQP[X;Y] SUBR	5.10; 13.3,5; 3.3, 13.1,4
EQUAL[X;Y] SUBR	5.10; 2.2; 13.1
equal	2.2
EQUALALL[X;Y]	5.11
EQUALN[X;Y;DEPTH]	5.11
ERASE (Masterscope command)	20.10
ERASE (transerset command)	24.26
ERROR[MESS1;MESS2;NOBREAK]	16.10,4,6,9
error correction	17.1-30
error handling	16.1-12
error number	16.4
error types	16.4-9
ERROR (error message)	16.6
ERRORMESS[U]	16.11,4
ERRORN[] SUBR	16.11,4
errors in compiler	18.33-36
errors in iterative statements	23.18
errors (in editor)	9.2
ERRORSET[FORM;FLG] SUBR	16.11; 7.5; 16.3-4,10, 17.10
ERRORSTRING[N] SUBR	16.11
ERRORTYPELIST (system variable/parameter)	16.9; 14.2
ERRORX[ERXM]	16.10
ERROR![] SUBR	16.10; 15.5; 16.10
ERSEQ[ERSETX] NL	16.12; 5.5; 16.11
ERSTR[ERN]	21.8
ESC (type of read-macro)	14.27
ESCAPE[FLG] SUBR	14.14,14
escape character	14.11; 2.4; 3.1
ESCAPE (syntax class)	14.25
ESCQUOTE (type of read-macro)	14.27

	Page Numbers
ESUBST[NEW;OLD;EXPR;ERRORFLG;CHARFLG]	9.63; 6.5; 22.13
EV (edit command)	9.50
EVAL[X] SUBR	8.6; 2.3,6; 4.1; 16.11
eval format	2.3
EVAL (break command)	15.5,13-14,17; 16.2-3
EVAL (edit command)	9.42
EVAL (Masterscope template)	20.16 , 8.8
EVALA[X;A] SUBR	2.3; 5.7; 15.4
EVALQT[LISPXID]	12.10
EVALV[X;POS] SUBR	18.7; 14.59
EVAL@COMPILE (DECLARE: option)	18.7; 14.59
EVAL@COMPILEWHEN (DECLARE: option)	14.59
EVAL@LOAD (DECLARE: option)	14.59
EVAL@LOADWHEN (DECLARE: option)	22.10-11
event address	22.7,10,17,27,40
event number	22.10-12,16-17
event specification	5.11 9.44
EVERY[EVERYX;EVERYFN1;EVERYFN2]	24.41-44
(EXAM . x) (edit command)	22.24; 21.8; 22.28
exec package	24.26
EXEC (prog. asst. command)	24.42
EXIT (transerset command)	18.11
EXP (exec command)	17.28
EXPANDMACRO[FORM;QUIETFLG]	17.27
EXPLAINDELIMITER (askuser option)	4.2; 8.3
EXPLAINSTRING (askuser option)	8.5,6; 9.58-60; 14.38, 17.12-13; 18.5,14
EXPR (function type)	22.23
EXPR (property name)	8.3,1-2,4 4.1 4.2; 8.3-4
.....	13.7
EXPRESSIONS (file package type)	24.43
EXPRP[FN] SUBR	9.30
exprs	
EXPR* (function type)	
EXPT[M;N]	
EXPUNGE[DIR]	
(EXTRACT @1 from . @2) (edit command)	
(F pattern N) (edit command)	9.17
(F pattern n) (n a number, edit command)	9.17
(F pattern T) (edit command)	9.17
(F pattern) (edit command)	9.18
F pattern (edit command)	9.17
F (edit command)	9.17,4-5
F (in event address)	22.10
F (response to compiler question)	18.1-3
false	5.3
FASSOC[KEY;ALST]	5.13; 2.2
fast symbolic dump	14.49
FAST (makefile option)	14.64
FASTYPEFLG (dwim variable/parameter)	17.17
FAULT IN EVAL (error message)	16.6
FAULTAPPLY[FAULTFN;FAULTARGS]	16.1; 17.3,10,14; 18.17
FAULTEVAL[FAULTX] NL*	16.1,6; 17.3,10,14
FBOX[N]	24.38
FBOX (record declaration)	24.38
FC	14.57
FCHARACTER[N] SUBR	10.3

Page
Numbers

FDIFFERENCE[X;Y]	13.5
FETCH (Masterscope relation)	20.5
FETCH (Masterscope template)	20.16
FETCH (use in records in clisp)	23.26
FETCHFIELD[DESCRIPTOR;DATUM]	3.8
FEXPR (function type)	4.2; 8.3-4
FEXPR* (function type)	4.2; 8.3-4
FFETCH (use in records in clisp)	23.26
FFILEPOS[PATTERN;FILENAME;FILESTART;FILEEND;SKIP; TAIL;CASEARRAY]	14.9
FGETD[X]	8.2
FGREATERP[X;Y] SUBR	13.5
FI (exec command)	24.42
FIELDLOOK[FIELDNAME]	23.35
FIELDS OF (Masterscope set specification)	20.7
FILDIR[FILEGROUP;FORMATFLG]	21.8
file attributes	14.4-5
file command list	14.56
FILE DATA ERROR (error message)	24.45
file maps	14.41-42
file names	14.2,3,6-7
FILE NOT COMPATIBLE (error message)	14.36
FILE NOT FOUND (error message)	16.7; 14.2-3,38
FILE NOT OPEN (error message)	16.6; 14.2,5; 21.11
file package	14.54-81
file package commands	14.56-61,73-75
file package functions	14.64-69,75-76
file package types	14.55-56,61-62,72-73
file pointer	14.7-8
FILE SYSTEM RESOURCES EXCEEDED (error message)	16.7; 14.3
FILE WONT OPEN (error message)	14.3
FILE WON'T OPEN (error message)	16.6; 14.1-2
FILE (edita parameter)	24.13
FILE (property name)	14.63-64
FILECHANGES (property name)	14.63,64,77
FILECOMS[FILE;X]	14.76
fileCOMS (in file package)	14.39,54,63-65,76
FILECOMSLST[FILE;TYPE;FLG]	14.76
FILECREATED[X] NL*	14.79,60; 18.5
FILEDATE[FILE;CFLG]	14.79
FILEDATES (property name)	14.63,64,77,79
FILEDEF (property name)	17.12,14
FILEFNSLST[FILE]	14.76
FILEGROUP (property name)	14.66
FILELINELENGTH (file package variable/parameter)..	14.64,48
FILEELST (file package variable/parameter)	14.63,66; 17.21
FILEMAP DOES NOT AGREE WITH CONTENTS OF file-name (error message)	14.42
FILEMAP (property name)	14.41
FILENAMEFIELD[FILENAME;FIELDNAME]	14.6
FILEPKGCHANGES[N] *	14.67
FILEPKGCOM[COMNAME;PROP1;VAL1;...;PROPN;VALn] * ..	14.73
FILEPKGCOMS (file package command)	14.58,60
FILEPKGCOMSPLST (prettyprint variable/parameter)..	14.60
FILEPKGFLG (file package variable/parameter)	14.54
FILEPKGTYPE[TYPE;PROP1;VAL1;...;PROPN;VALn] * ..	14.72
FILEPKGTYPES (file package variable/parameter) ..	14.61

Page
Numbers

FILEPKG.SCRATCH (file package variable/parameter).	14.72
FILEPOS[PATTERN;FILENAME;FILESTART;FILEEND;SKIP; TAIL;CASEARRAY]	14.8
FILERDTBL (system variable/parameter)	14.16, 38, 42, 78
files	14.1-11
FILES?[]	14.67, 64
FILETYPE (property name)	14.64; 18.6; 23.40, 42, 60
filevar (in file package)	14.60, 39, 76
FILE: (compiler question)	18.2
FINALLY (clisp iterative statement operator)	23.16, 17
FINDCALLERS[ATOMS;FILES]	9.61
FINDFILE[FILE;NSFLG]	17.21
FIRST (as argument to advise)	19.3-4
FIRST (clisp iterative statement operator)	23.16, 17
FIRST (DECLARE: option)	18.7
FIRST (type of read-macro)	14.27
FIRSTCOL (prettyprint variable/parameter)	14.48
FIRSTNAME (system variable/parameter)	22.47
FIX[X]	13.4
FIX format (in printnum package)	14.20
FIX (prog. asst. command)	22.14, 15, 18
fixed number of arguments	4.1
FIXEDITDATE[EXPR]	9.59
FIXP[X]	13.4
FIXP (as a field specification)	3.7
FIXP (record field type)	23.29
FIXSPELL[XWORD;REL;SPLST;FLG;TAIL;FN;TIEFLG; DONTMOVETOPFLG;CLST;APPROVALFLG]	17.18, 19, 21
FIXSPELLDEFAULT (dwim variable/parameter)	17.20, 4; 23.41
FIXSPELLREL (dwim variable/parameter)	17.20, 19
FLAG (record field type)	23.29
FLAST[X]	6.5; 2.2
FLENGTH[X]	6.6; 2.2
FLESSP[X;Y]	13.5
FLOAT[X]	13.6
FLOAT format (in printnum package)	14.20
floating point arithmetic	13.5-6
floating point numbers	3.4, 1, 3, 9; 13.1, 3, 9, 14.12
FLOATING (record field type)	23.29
FLOATP[X] SUBR	13.6
FLOATP (as a field specification)	3.8
FLOATP (record field type)	23.29
FLTfmt[FORMATBITS]	14.35; 3.4
FLUSHRIGHT[POS;X;MIN;P2FLAG;CENTERFLAG;FILE]	23.50
FMAX[X1;X2;...;Xn]	13.6
FMEMB[X;Y]	6.12; 2.2
FMIN[X1;X2;...;Xn]	13.6
FMINUS[X]	13.5
FN (transorset command)	24.25
FNCHECK[FN;NOERRORFLG;SPELLFLG;PROPFLG;TAIL]	17.20; 8.4; 17.21
FNS (file package command)	14.56
FNTH[X;N]	6.6; 2.2
FNTYP[FN]	8.3; 4.2; 8.1-2, 4, 6
(fn1 IN fn2)	15.15, 18; 19.4
(fn1 NOT FOUND IN fn2)	15.15

	Page Numbers
fn1-IN-fn2	15.15, 18; 19.4
font package	14.49-52
FONTCHANGEFLG (prettyprint variable/parameter) ..	14.51
FONTDEFS (prettyprint variable/parameter)	14.52
FONTDEFSVARS (prettyprint variable/parameter)	14.52
FONTEXCAPECHAR (prettyprint variable/parameter) ..	14.51
FONTNAME[NAME]	14.52
FONTPROFILE (prettyprint variable/parameter)	14.51
FONTSET[NAME]	14.52
FOR (clisp iterative statement operator)	23.13
FOR (in INSERT command) (in editor)	9.27
FOR (in USE command)	22.13
FOR (Masterscope command)	20.12
FORCEOUT[CONNECTION/FILE]	24.47
FORGET (prog. asst. command)	22.23, 40
fork handle	21.8
forks	21.6
format and use of history list	22.34-35
format characters	14.25
format of keylst (for askuser)	17.24
FPLUS[X1;X2;...;Xn] SUBR*	13.5
FQUOTIENT[X;Y] SUBR	13.5
frame extension	12.5
frame name	12.5
frames	12.5
FRAMESCAN[ATOM;POS] SUBR	12.8
FREE (in Decl package)	24.58
FREELY (use in Masterscope)	20.5
FREEVARS[FN;USEDATABASE]	20.19
free-list	3.9-10
FREMAINDER[X;Y] SUBR	13.5
FREPLACE (use in records in clisp)	23.26
FROM (clisp iterative statement operator)	23.14, 15-16
FROM (in event specification)	22.11
FROM (in EXTRACT command) (in editor)	9.30
FROM (Masterscope path option)	20.15
FRPLACA[X;Y] SUBR	5.2
FRPLACD[X;Y] SUBR	5.2
FRPLNODE[X;A;D]	5.2
FRPLNODE2[X;Y]	5.2
FRPTQ[N;FORM1;FORM2;...;FORMn] NL*	8.8
(FS . patterns) (edit command)	9.18
FSUBR (function type)	4.2; 8.3-4
FSUBR* (function type)	4.2; 8.3-4
FTIMES[X1;X2;...;Xn] SUBR*	13.5
FTP[HOST;FILE;ACCESS;USER;PASSWORD;ACCOUNT; BYTESIZE]	24.45
ftp package	24.44
full file name	14.2-3
FULLNAME[X;RECOG]	14.3
FUNARG	11.3-4, 1.5; 18.13
FUNARG (function type)	8.3
FUNCTION[FN;ENV] NL	11.1; 12.10; 11.3-4, 18.13
function definition and evaluation	8.1-9
function definition cell	8.1; 2.3, 5; 18.15
function objects	11.4; 16.1

Page
Numbers

function types	4.1-2
FUNCTION (Masterscope template)	20.16
functional arguments	11.1; 2.3; 8.8; 18.13
FUNNYATOMLST (clisp variable/parameter)	23.60,39
F/L (as a dwim construct)	17.12
(F= expression flg) (edit command)	9.18
GAINSPACE[]	21.16
GAINSPACEFORMS (system variable/parameter)	21.16
garbage collection	3.9-11; 10.8,11,13-14
garbage collection message	10.11-12
GCD[X;Y]	13.5
GCGAG[MESSAGE]	10.11
GCMESS[MESSAGE#;STRING] SUBR	10.12
GCTR P[N] SUBR	10.14; 21.4
GCTR P (printed by system)	10.14
GDATE[DATE;FORMATBITS;STRPTR]	21.2
generalized NTH command (in editor)	9.21,33,40
GENERATE[HANDLE;VAL]	12.14
GENERATOR[FORM##;COMVAR##] NL	12.14
generator handle	12.14
generators	12.13
generators for spelling correction	17.9; 24.51
GENNUM (system variable/parameter)	10.4
GENSYM[CHAR]	10.3; 3.2; 10.4; 15.15, 18.13; 19.3-4
GEQ[X;Y]	13.6
GETATOMVAL[ATM] SUBR	5.6
GETBLK[N] SUBR	21.6; 16.7
GETBRK[RDTBL] SUBR	14.14
GETCOMMENT[X;DESTFL;DEF]	14.45
GETCONTROL[TTBL]	14.33
GETD[X] SUBR	8.2; 2.3,5; 8.1,6
GETD (edit command)	9.53
GETDECLTYPEPROP[TYPE;PROP]	24.65
GETDEF[NAME;TYPE;SOURCE;OPTIONS]	14.69
GETDELETECONTROL[TYPE;TTBL]	14.31
GETDESCRIPTORS[TYPENAME]	3.8
GETECHOMODE[TTBL]	14.31
GETEOFPTR[FILE] SUBR	14.8
GETFIELDSPECS[TYPENAME]	3.8
GETFILEINFO[FILE;ATTRIB]	14.4
GETFILEMAP[FILE;FL]	14.41
GETFILEPTR[FILE] SUBR	14.7
GETHASH[ITEM;HARRAY] SUBR	7.4; 23.22
GETHASHFILE[KEY;HASHFILE]	24.49
GETLIS[X;PROPS]	7.3
GETPAGE[HASHFILE;N]	24.52
GETPASSWORD[DIRECTORYNAME]	24.44
GETPNAME[FILEADR;HASHFILE]	24.52
GETPROP[ATM;PROP]	7.1
GETPROPLIST[ATM]	7.1; 2.3
GETRAISE[TTBL]	14.32
GETREADTABLE[RDTBL] SUBR	14.24
GETRELATION[ITEM;RELATION;INVERTED]	20.19
GETSEPR[RDTBL] SUBR	14.14
GETSYNTAX[CH;TABLE]	14.25

	Page Numbers
GETTEMPLATE[FN]	20.18
GETTERMTABLE[TTBL] SUBR	14.29
GETTOPVAL[ATM] SUBR	5.5; 2.3
GETTYPEDESCRIPTION[TYPE]	10.13
GETVAL (edit command)	9.42
GET* (edit command)	9.53; 14.45
GLC[X] SUBR	10.5,8
global variables	18.5; 23.41
GLOBALVAR (property name)	18.4; 23.41
GLOBALVARS (compiler variable/parameter)	18.4,21; 23.41
GLOBALVARS (file package command)	14.59
GNC[X] SUBR	10.5,8
GO[X] FSUBR*	5.5
(GO label) (edit command)	9.19
GO (break command)	15.5,4,6,13-14, 16.2-3
GO (use in iterative statement in clisp)	23.17
GREATERP[X;Y] SUBR	13.6
GREET[NAME;FLG]	22.47
GREETDATES (system variable/parameter)	22.47
greeting and user profiles	22.46
GTJFN[FILE;EXT;V;FLAGS]	21.12
 handle	3.12
HARRAY[N] SUBR	7.4; 10.9
HARRAYP[X]	7.4; 10.9
HARRAYSIZE[HARRAY]	7.4; 10.9
HASDEF[NAME;TYPE;SPELLFLG]	14.72
hash arrays	3.1
hash link functions	7.4-5
hash links	7.3-5
hash overflow	7.5
hash package	24.47-53
HASH TABLE FULL (error message)	16.7; 7.5
HASHFILENAME[HASHFILE]	24.49
HASHFILEP[X]	24.49
HASHFILEPROP[HASHFILE;PROP]	24.49
HASHFILESPLST[HASHFILE]	24.51
HASHLINK (record package)	23.28
hash-address	7.3
hash-array	7.3-5
hash-item	7.3-5
hash-link	7.3-5
hash-value	7.3-5
HCOPYALL[X]	6.4; 14.23
HELP[MESS1;MESS2]	16.10
HELP (Masterscope command)	20.12
HELPCLK (system variable/parameter)	16.4; 22.17,31
HELPDEPTH (system variable/parameter)	16.3-4
HELPFLAG (system variable/parameter)	16.3,2,5
helpsys package	20.22-27
HELPTIME (system variable/parameter)	16.3-4
HELP! (printed by system)	16.10
HERALD[STRING] SUBR	3.11
HERALDSTRING (system variable/parameter)	14.37; 3.12
HERE (in edit command)	9.28
history commands	22.9-23

Page
Numbers

history commands applied to history commands	22.16
history commands that fail	22.17
history list	22.4; 9.48,54; 22.5-12, 34-35
HISTORYCOMS (editor variable/parameter)	22.45
HISTORYFIND[LST;INDEX;MOD;EVENTADDRESS; LISPXFINDFLG]	22.39,39
HISTORYMATCH[INPUT;PAT;EVENT]	22.40
HISTORYSAVE[HISTORY;ID;INPUT1;INPUT2;INPUT3; PROPS]	22.38,9,27,34-35,39,45
HISTORYSAVEFORMS (prog. asst. variable/parameter). HISTSTRO (prog. asst. variable/parameter)	22.27
HORRIBLEVARS (file package command)	22.15,37 14.59,23
HOST (as a file name field)	24.45
HOSTNAME[HOSTN]	21.9
HOSTNUMBER[]	21.9
HPRINT[EXPR;FILE;UNCIRCULAR]	14.23
HREAD[FILE]	14.23
 (I c x1 ... xn) (edit command)	9.42
IBOX[N]	24.38
IBOX (record declaration)	24.38
IDATE[D]	21.2
IDIFFERENCE[X;Y]	13.2
IEQP[N;M] SUBR	13.3
(IF x coms1 coms2) (edit command)	9.43
(IF x coms1) (edit command)	9.44
(IF x) (edit command)	9.43
IF (Masterscope template)	20.17
IFPROP (file package command)	14.57,60
IF-THEN-ELSE statements	23.10
IGEQ[X;Y]	13.3
IGNOREDECL (file package command)	24.60
IGNOREMACRO (in compiler)	18.10
IGREATERP[X;Y] SUBR	13.3
ILEQ[X;Y]	13.3
ILESSP[X;Y]	13.3
ILLEGAL ARG (error message)	16.7; 8.2; 10.2; 12.8, 14.3,30; 22.41
 ILLEGAL DATA TYPE NUMBER (error message)	16.8
ILLEGAL DATA TYPE (error message)	3.8
ILLEGAL EXPONENTIATION (error message)	13.7
ILLEGAL GO (compiler error message)	18.35
ILLEGAL INSTRUCTION (error message)	16.5
ILLEGAL OR IMPOSSIBLE BLOCK (error message)	16.7; 21.6-7
ILLEGAL READTABLE (error message)	16.8; 14.24,30
ILLEGAL RETURN (compiler error message)	18.34
ILLEGAL RETURN (error message)	16.5; 5.5
ILLEGAL STACK ARG (error message)	16.6; 12.6
ILLEGAL TERMINAL TABLE (error message)	16.8; 14.29-30
IMAX[X1;X2;...;Xn] *	13.3
IMIN[X1;X2;...;Xn] *	13.3
IMINUS[X]	13.2
IMMED (type of read-macro)	14.28
IMMEDIATE (type of read-macro)	14.28
implementation of REDO, USE, and FIX	22.15-16
implementation of structure modification commands (in editor)	9.25-26

Page
Numbers

implicit progn	4.3; 12.2-3
IN (clisp iterative statement operator)	23.13,15,17
IN (in EMBED command) (in editor)	9.31
IN (in USE command)	22.13
IN (Masterscope set specification)	20.6
IN (printed by system)	16.3
INCORRECT DEFINING FORM (error message)	8.5
incorrect number of arguments	4.2
indefinite number of arguments	4.2
INFILE[FILE] SUBR	14.1,7
INFILECOMS?[NAME;TYPE;COMS;ONFILETYPE]	14.75
INFILEP[FILE] SUBR	14.2,2
infix operators (in clisp)	23.6-8
INFIX (type of read-macro)	14.26
INFO (property name)	23.39; 4.1; 23.40
INIT (use in record declarations)	23.31
INITIALS (editor variable/parameter)	9.59
INITIALSLST (editor variable/parameter)	9.59
INITRECORDS (file package command)	14.58; 23.27
INPUT[FILE] SUBR	14.1
input buffer	14.32; 10.13; 14.15,19, 34; 15.14; 16.2,4
input functions	14.11-16
inputting numbers	14.12
input/output	14.1-81
INREADMACROP[] SUBR	14.28
(INSERT ... AFTER . @) (edit command)	9.27
(INSERT ... FOR . @) (edit command)	9.27
INSIDE (clisp iterative statement operator)	23.14
INSTRUCTIONS (in compiler)	18.10
integer arithmetic	13.2-5
INTEGER (record field type)	23.29
integers	3.3
interfork communication	21.6
interpreter	8.6; 16.1
INTERRUPT[INTFN;INTARGS;INTYPE]	16.2; 10.14
interrupt characters	16.12; 2.4; A1.2
INTERRUPTABLE[FLG] SUBR	16.13
INTERRUPTABLEP[] SUBR	16.13
INTERRUPTCHAR[CHAR;TYP/FORM;HARDFLG]	16.12
INTERRUPTED BEFORE (printed by system)	16.2
INTERSECTION[X;Y]	6.7
IN? (break command)	15.11,6,12; 16.1
IOFILE[FILE] SUBR	14.4,7-8
IPLUS[X1;X2;...;Xn] SUBR*	13.2
IQUOTIENT[X;Y] SUBR	13.3
IREMINDER[X;Y] SUBR	13.3
IT (prog. asst. variable/parameter)	22.26,18
iterative statements (in clisp)	23.11-21
ITIMES[X1;X2;...;Xn] SUBR*	13.2
I.S.OPR[NAME;FORM;OTHERS;EVALFLG]	23.19
i.s.ops	23.11-21
I.S.OPRS (file package command)	14.58; 23.21
i.s.type	23.13,19
JFN	21.11-12
JFNS[JFN;AC3;STRPTR]	21.12

Page
Numbers

JOB#[]	24.43
JOIN (clisp iterative statement operator)	23.12
JOINC (edit command)	9.51
JS[JSYSNAME;AC1;AC2;AC3;RESULT]	24.35
JS (assemble macro)	24.36
JSYS[N;AC1;AC2;AC3;RESULTAC] SUBR	21.9
JSYS	14.35; 21.8,11-12
JSYS ERROR (error message)	16.5; 21.9
JSYSERROR[ERRORN]	24.36
JSYSES (system variable/parameter)	24.35
Keyboard layouts	17.5,16
KEYLST (askuser option)	17.26
KEYSTRING (askuser option)	17.27
KFORK[FORK]	21.8,8
KNOWN (Masterscope set specification)	20.8
KWOTE[X]	5.3
LABELS (use with clispify)	23.39
LAMBDA	4.1,2-3; 8.2-3,6
LAMBDACOMS (transor parameter)	24.29
LAMBDAFONT	14.50
LAMBDAPLST (dwim variable/parameter)	17.12; 8.4; 17.13-14, 24.33
Lambdatran package	24.32-34
LAMBDATRANFNS (lambdatran parameter)	24.33
LAMS (compiler variable/parameter)	18.3,6
LAP	18.27,2,23,28-29
LAP macros	18.29,25
LAP op-defs	18.24-25
LAP statements	18.27-29
LAPFLG (compiler variable/parameter)	18.2
large integers	3.3,1,9; 5.10; 13.1,9
LAST[X]	6.5
LAST (as argument to advise)	19.3-4
LASTAII (editor variable/parameter)	9.11,17.63
LASTC[FILE] SUBR	14.15
LASTN[L;N]	6.6
LASTPOS (break variable/parameter)	15.6,7-9,11
LASTVALUE (property name)	9.48
LASTWORD (dwim variable/parameter)	17.9; 9.61; 17.18,21, 23.8
LBOX[X1;X2;...;Xn]	24.37
(LC . @) (edit command)	9.19
LCASELST (prettyprint variable/parameter)	14.46
LCFIL (compiler variable/parameter)	18.2-3
(LCL . @) (edit command)	9.19
LCONC[PTR;X]	6.2,3
LD (exec command)	24.41
LDIFF[X;Y;Z]	6.7
LDIFFERENCE[X;Y]	6.7
LDIFF: NOT A TAIL (error message)	6.7
LEFTBRACKET (syntax class)	14.25
LEFTPAREN (syntax class)	14.25
LENGTH[X]	6.6
LEQ[X;Y]	13.7
LESSP[X;Y]	13.6

	Page Numbers
(LI n) (edit command)	9.34,6
LIKE (Masterscope set specification)	20.7
LINBUF[FLG] SUBR	14.34
line buffer	14.32,34
LINEDELETE (syntax class)	14.30-31
LINELENGTH[N] SUBR	14.35,48; 2.2; 3.5
LINELENGTH (Masterscope path option)	20.15
line-buffering	14.32; 2.4; 14.12,14-15, 33
line-feed	14.11; 3.1; 14.17
line-feed (edit command)	9.14; 14.36; A1.3
line-feed (edita command)	24.12
LINK (exec command)	24.41
linked function calls	18.15-18
LINKEDFNS (system variable/parameter)	18.17
LINKFNS (compiler variable/parameter)	18.17,21-22
LINKTOTTY[TTY#]	24.43
LINKTOUSER[USER]	24.43
LISP FN (property name)	23.54
LISPX[LISPXX;LISPXD;LISPXXMACROS;LISPXXUSERFN; LISPXFLG]	22.9,35-36; 9.41,48, 17.3,8-9,21; 22.13, 15-17,25,28-32,34, 37-38,45
LISPXCOMS (prog. asst. variable/parameter)	14.58; 22.31
LISPXEVAL[LISPXFORM;LISPXID]	22.38
LISPXFIND[HISTORY;LINE;TYPE;BACKUP;QUIETFLG]	22.39,45
LISPXFINDSPLST (prog. asst. variable/parameter)	22.12
LISPXHIST (prog. asst. variable/parameter)	22.35,41,43
LISPXHISTORY (prog. asst. variable/parameter)	22.34,36,43,45
LISPXHISTORYMACROS (prog. asst. variable/parameter)	22.28
LISPXLINE (prog. asst. variable/parameter)	22.28
LISPXMACROS (file package command)	14.58
LISPXMACROS (prog. asst. variable/parameter)	22.28; 21.8; 22.37
LISPXPRINT[X;Y;Z;NODOFLG]	22.30,34
LISPXPRINTDEF[EXPR;FILE;LEFT;DEF;TAIL;NODOFLG]	22.30
LISPXPRINTFLG (system variable/parameter)	22.31
lispxprinting functions	22.30
LISPXPRIN1[X;Y;Z;NODOFLG]	22.30
LISPXPRIN2[X;Y;Z;NODOFLG]	22.30
LISPXREAD[FILE;RDTBL]	22.37,9,16,25,27,36,45
LISPXREADFN (prog. asst. variable/parameter)	22.37; 14.15
LISPXREADP[FLG]	22.38,45
LISPXSPACES[X;Y;Z;NODOFLG]	22.30
LISPXSTATS[RETURNVALUESFLG]	22.46
LISPXSTOREVALUE[EVENT;VALUE]	22.39
LISPXTAB[X;Y;Z;NODOFLG]	22.30
LISPXTERPRI[X;Y;Z;NODOFLG]	22.30
LISPXUNREAD[LST;EVENT]	22.38
LISPXUSERFN (prog. asst. variable/parameter)	22.29,30,36-37
LISPXWATCH[STAT;N]	22.46
LISPX/[X;FN;VARS]	22.42,32
LIST[X ₁ ;X ₂ ;...;X _n] SUBR*	6.1; 3.5
list manipulation and concatenation	6.1-9
list nodes	3.5,9
LIST (makefile option)	14.65

	Page Numbers
LIST (property name)	8.6
LISTFILES[FILES] NL*	14.66,64-65
LISTFILESTR (file package variable/parameter)	14.51,66
LISTFILES1[FILE]	14.66
LISTGET[LST;PROP]	5.13; 23.28
LISTGET1[LST;PROP]	5.13
LISTING? (compiler question)	18.1-2
LISTP[X] SUBR	5.10; 2.3
listp checks (in pattern match compiler)	24.2
LISTPUT[LST;PROP;VAL]	5.13; 23.28
LISTPUT1[LST;PROP;VAL]	5.13
lists	3.4; 2.3; 3.1; 5.10
LITATOM[X] SUBR	5.9
literal atoms	3.1,2; 5.9; 10.7, 14.12
.....	24.12
LITS (edita parameter)	13.4
LLSH[N;M] SUBR	9.34,6
(LO n) (edit command)	14.38; 2.9; 18.5
LOAD[FILE;LDFLG;PRINTFLG]	21.8
LOADAV[]	14.40
LOADBLOCK[FN;FILE;LDFLG]	14.40
LOADCOMP[FILE;LDFLG]	14.41
LOADCOMP?[FILE;LDFLG]	24.32
LOADDB[FILE]	24.32
LOADDBFLG (system variable/parameter)	14.71
LOADDEF[NAME;TYPE;SOURCE]	14.63
LOADEDFILEST (file package variable/parameter)	14.40
LOADEFS[FNS;FILE]	14.39
LOADFNS[FNS;FILE;LDFLG;VARS]	14.40; 18.8
LOADFROM[FILE;FNS;LDFLG]	14.39
LOADOPTIONS (system variable/parameter)	14.40
LOADVARS[VARS;FILE;LDFLG]	14.39
LOAD?[FILE;LDFLG;PRINTFLG]	13.10,10
LOC[X] SUBR	23.25
local record declarations (in clisp)	5.4
local variables	24.56
LOCAL (in Decl package)	18.21
LOCALFREEVARS (compiler variable/parameter)	20.5
LOCALLY (use in Masterscope)	18.18
LOCALVARS (compiler variable/parameter)	14.59
LOCALVARS (file package command)	9.19,43
location specification (in editor)	9.11
LOCATION UNCERTAIN (printed by editor)	21.15
LOCKMAP[PTR]	13.7
LOG[X]	13.4
LOGAND[X1;X2;...;Xn] SUBR*	24.45
LOGIN (property name)	13.4
LOGOR[X1;X2;...;Xn] SUBR*	21.4; 2.3; 21.8
LOGOUT[] SUBR	13.4
LOGXOR[X1;X2;...;Xn] SUBR*	24.51
LOOKUPHASHFILE[KEY;VALUE;HASHFILE;CALLTYPE]	10.4
lower case	14.45-46
lower case comments	23.61
lower case in clisp	14.32
lower case input	9.50
(LOWER x) (edit command)	9.50
LOWER (edit command)	9.50

Page
Numbers

LOWERCASE[FLG]	23.61
(LP . coms) (edit command)	9.44
LPARKEY (dwim variable/parameter)	17.5
(LPQ . coms) (edit command)	9.44
LRSH[N;M]	13.4
LSH[N;M] SUBR	13.4
LSTFIL (compiler variable/parameter)	18.2
LSUBST[NEW;OLD;EXPR]	6.4,5
L-CASE[X;FLG]	10.4; 9.50
M (edit command)	9.45
machine instructions	18.27,28-29; 24.10
MACRO (Masterscope template)	20.17
MACRO (property name)	18.10,9,12
MACRO (type of read-macro)	14.26
MACROCHARS (askuser option)	17.28
MACROS (file package command)	14.59
macros (in compiler)	18.10-12
macros (in editor)	9.45-46
macrotran package	18.12; 17.15; 18.24
MACSCRATCHSTRING (system variable/parameter)	21.3,12
(MAKE argname exp) (edit command)	9.54
MAKEBITTABLE[L;NEG;A]	10.7
MAKEFILE[FILE;OPTIONS;REPRINTFNS;SOURCEFILE]	14.64-66; 17.21; 18.8
makefile and clisp	23.60,23
MAKEFILEFORM (file package variable/parameter)	14.66
MAKEFILEOPTIONS (file package variable/parameter)	14.65
MAKEFILEREMAKEFLG (file package variable/parameter)	14.77,65
MAKEFILES[OPTIONS;FILES]	14.66
(MAKEFN (fn . actualargs) arglist n1 n2) (edit command)	9.53
MAKEKEYLST[LST;DEFAULTKEY;LCASEFLG]	17.29
MAKENEWCOM[NAME;TYPE;LISTNAME;FILE]	14.76
MAKENEWCONNECTION[HOST;TYPE;SKT;SCRATCHCONN; WAITFLG]	24.46
MAKESYS[FILE] SUBR	3.11
MAKESYSDATE (system variable/parameter)	3.11
manipulating file names	14.6-7
MAP[MAPX;MAPFN1;MAPFN2]	11.2
MAPATOMS[FN] SUBR	10.4
MAPBUFFERCOUNT[ONLY]	21.13
MAPC[MAPX;MAPFN1;MAPFN2]	11.2
MAPCAR[MAPX;MAPFN1;MAPFN2]	11.2
MAPCON[MAPX;MAPFN1;MAPFN2]	11.2
MAPCONC[MAPX;MAPFN1;MAPFN2]	11.2
MAPDL[MAPDLFN;MAPDLPOS]	12.12
MAPHASH[ARRAY;MAPHFN]	7.5
MAPHASHFILE[HASHFILE;MAPFN]	24.50
MAPLIST[MAPX;MAPFN1;MAPFN2]	11.2
MAPPAGE[PAGE#;FILE]	21.13
MAPRELATION[RELATION;MAPFN]	20.20
MAPPRINT[LST;FILE;LEFT;RIGHT;SEP;PFN;LSPXPRNTFLG]..	11.3
MAPWORD[FILEADR;FILE]	21.14
MAP2C[MAPX;MAPY;MAPFN1;MAPFN2]	11.3
MAP2CAR[MAPX;MAPY;MAPFN1;MAPFN2]	11.3
margins (for prettyprint)	14.79

	Page Numbers
(MARK atom) (edit command)	9.22
MARK (edit command)	9.22
MARKASCHANGED[NAME;TYPE;NEWFLG]	14.67
MARKLST (editor variable/parameter)	9.22, 63
MASK (edita parameter)	24.14
Masterscope	20.1-22
MASTERSCOPE[COMMAND]	20.19
Masterscope commands	20.9-12
MATCH (use in pattern match in clisp)	24.2
MAX[X1;X2;...;Xn] *	13.7
MAXLEVEL (editor variable/parameter)	9.16, 18
MAXLOOP EXCEEDED (printed by editor)	9.44
MAXLOOP (editor variable/parameter)	9.44
MAX.FLOAT (system variable/parameter)	13.6
MAX.INTEGER (system variable/parameter)	13.3
(MBD e1 ... em) (edit command)	9.30
MEMB[X;Y]	5.12
MEMBER[X;Y]	5.12
MEMSTAT[PG1;PGN;FORK]	24.44
MERGE[A;B;COMPAREFN]	6.8
MERGEINSERT[NEW;LST;ONEFLG]	6.8
meta-LISP notation	2.1
MIN[X1;X2;...;Xn] *	13.7
MINFS[N;TYPE]	10.13; 3.10-11
MINUS[X] SUBR	13.6
MINUSP[X] SUBR	13.3, 5
MIN.FLOAT (system variable/parameter)	13.6
MIN.INTEGER (system variable/parameter)	13.3
MISSING OPERAND (dwim error message)	23.51
MISSING OPERATOR (clisp error message)	23.51
MISSPELLED?[XWORD;REL;SPLST;FLG;TAIL;FN]	17.18, 21
mixed arithmetic	13.6-7
MKATOM[X] SUBR	10.5; 3.2-4
MKLIST[X]	6.3
MKSTRING[X] SUBR	10.4; 3.6, 9; 10.8
MKSWAP[X]	3.14
MKSWAPP[FNAME;CDEF]	3.14
MKSWAPSIZE (Overlay variable/parameter)	3.14
MKUNSWAP[X]	3.14
MODEL33FLG (dwim variable/parameter)	17.16
MODIFIER (use with interative statement operators)	23.21
MOVD[FROM;TO;COPYFLG]	8.3
MOVD?[FROM;TO;COPYFLG]	8.3
(MOVE @1 TO com . @2) (edit command)	9.31-33
MOVEITEM[TOFILE;NAME;TYPE;FROMFILE]	14.76
MSMARKCHANGED[NAME;TYPE;NEWFLG]	20.20
MSPRINTFLG (Masterscope parameter)	20.3
MULTIFILEINDEX[FILENAMELST;MAPFILE;NEWPAGEFLG]	24.30
multifileindex package	24.30-31
MULTIPLY DEFINED TAG (compiler error message) ...	18.35
MULTIPLY DEFINED TAG, ASSEMBLE (compiler error message)	18.35
MULTIPLY DEFINED TAG, LAP (compiler error message)	18.35
(N e1 ... em) (edit command)	9.24
(n e1 ... em) (n a number, edit command)	9.24, 4
(n) (n a number, edit command)	9.24, 4

Page
Numbers

n (n a number, edit command)	9.11,2
n (n a number, printout command)	23.45
NAME (prog. asst. command)	22.21,18,22
NAMES RESTORED (printed by system)	15.19
NAMESCHANGED (property name)	15.15
NARGS[FN]	8.4,1-3
NBOX[N]	24.38-39
NCHARS[X;FLG;RDTBL] SUBR	10.3; 14.8
NCONC[X1;X2;...;Xn] SUBR*	6.1,2
NCONC1[LST;X]	6.1,2
NCREATE[TYPENAME;FROM]	3.8
NDIR (exec command)	24.42
NEEDUNSAVE (Masterscope paramater)	20.21
NEGATE[X]	5.10
NEGATE (edit command)	9.51
NEQ[X;Y]	5.10
net package	24.46-47
NETSERVER[ARPA#;WAITFLG]	24.47
NETUSER[HOST;USER:PUP#;WAITFLG]	24.47
NEVER (clisp iterative statement operator)	23.12
NEW (makefile option)	14.65
NEWISWORD[SING;PLU;FORM;VARS]	23.21
NEW/FN[FN]	22.42
(NEX x) (edit command)	9.21
NEX (edit command)	9.21
NIL	2.2
NIL (edit command)	9.52,43
NIL (Masterscope template)	20.16
NIL (use in block declarations)	18.22
NILCOMS (file package variable/parameter)	14.68
NILL[]	5.10
NLAM (transor command)	24.28
NLAMA (compiler variable/parameter)	18.3
NLAMBDA	4.1,2-3; 8.2-3
NLAML (compiler variable/parameter)	18.3
NLEFT[L;N;TAIL]	6.5
NLISTP[X]	5.10; 2.2
NLISTPCOMS (transor parameter)	24.29
NLSETQ[NLSETX] NL	16.12; 5.5; 16.11, 22.43
NLSETQGAG (system variable/parameter)	16.4,11
NO BINARY CODE GENERATED OR LOADED (compiler error message)	18.35
(NO BREAK INFORMATION SAVED)	15.18
NO DO, COLLECT, OR JOIN (error message)	23.19
NO FILE PACKAGE COMMAND FOR (error message)	14.58
NO LONGER INTERPRETED AS FUNCTIONAL ARGUMENT (compiler error message)	18.34
NO PROPERTY FOR (error message)	14.57
NO USERMACRO FOR (error message)	14.58
NO VALUE SAVED: (error message)	22.41
NOBIND	3.3; 16.1; 2.6, 5.6; 8.7; 9.61; 12.10, 14.38; 22.33,40
nobox package	24.37-40
NOBREAKS (break variable/parameter)	15.17
NOCASEFLG (askuser option)	17.27

	Page Numbers
NOCLEARSTKLST (system variable/parameter)	12.11
NOCLISP (makefile option)	14.65; 23.23,60
NOECHOFLG (askuser option)	17.27
NOESC (type of read-macro)	14.27
NOESCQUOTE (type of read-macro)	14.27
NOFIXFNSLST (dwim variable/parameter)	23.57; 14.40; 23.41,43
NOFIXVARSLST (dwim variable/parameter)	23.57; 14.40; 23.41,43, 51
NOLINKDEF	18.17
NOLINKFNS (compiler variable/parameter)	18.17,21-22
NONE (in Decl package)	24.60
NONE (syntax class)	14.30
NONIMMED (type of read-macro)	14.28
NONIMMEDIATE (type of read-macro)	14.28
NON-ATOMIC CAR OF FORM (compiler error message) ..	18.34
NON-NUMERIC ARG (error message)	16.6; 13.2,5-6; 16.3
NOPACKCALLSFLG (Masterscope parameter)	20.18
NORAISe (TENEX command)	14.32
NORMALCOMMENTSFLG (system variable/parameter) ..	14.45
NOSAVE	22.41
NOSPELLFLG (dwim variable/parameter)	17.20; 23.57
nospread functions	4.2; 8.1
NOSWAPFNS (Overlay variable/parameter)	3.14
NOT[X] SUBR	5.10
NOT A BINDABLE VARIABLE (compiler error message) ..	18.35
NOT A FUNCTION (error message)	8.5-6; 19.4
NOT A HASHFILE (error message)	24.48
NOT BLOCKED (printed by editor)	9.55
(NOT BROKEN)	15.18
NOT COMPILEABLE (compiler error message)	18.34,5,21
NOT DUMPED (error message)	14.65
NOT EDITABLE (error message)	9.59,61
NOT FOUND (compiler error message)	18.34
NOT FOUND (error message)	14.66
(NOT FOUND) (typed by break)	15.7
(NOT FOUND) (typed by breakin)	15.16-17
(NOT FOUND) (value of unsavedef)	8.6
NOT FOUND, SO IT WILL BE WRITTEN ANEW (error message)	14.79
NOT IN FILE - USING DEFINITION IN CORE (compiler error message)	18.33
NOT ON BLKFNS (compiler error message)	18.34,14,20
NOT ON FILE, COMPILING IN CORE DEFINITION (compiler error message)	18.21
(NOT PRINTABLE)	14.43
NOTANY[SOMEEx;SOMEFN1;SOMEFN2]	5.12
NOTCOMPILEDFILES (file package variable/parameter) ..	14.64,66
NOTE[VAL;LSTFLG]	12.17
NOTE (transor command)	24.28,26
NOTEVERY[EVERYx;EVERYFn1;EVERYFn2]	5.12
NOTE: BRKEXP NOT CHANGED. (typed by break)	15.11
NOTFIRST (DECLARE: option)	18.7
(NOTHING FOUND)	8.6
NOTHING SAVED (printed by editor)	9.55
NOTHING SAVED (printed by system)	22.31,18
noticing files	14.63
NOTLISTEDFILES (file package variable/parameter) ..	14.64,66

	Page Numbers
NOTRACE (Masterscope path option)	20.15
NOT-FOUND:	14.39
NTH[X;N]	6.6
(NTH n) (n a number, edit command)	9.14
(NTH x) (edit command)	9.21
NTHCHAR[X;N;FLG;RDTBL] SUBR	10.3
NTYP[DATUM] SUBR	10.11
NULL[X] SUBR	5.10
null string	10.5
null-check	2.2; 6.5-7
NUMBERP[X] SUBR	5.9
numbers	13.1; 5.9; 13.2-11, 14.12
NUMFORMATCODE[FORMAT;SMASHCODE]	14.22
(NX n) (n a number, edit command)	9.12
NX (edit command)	9.12,6
OCCURRENCES (printed by editor)	9.44
octal	14.12; 3.3,5; 14.17
OK TO REEVALUATE (printed by dwim)	17.6
OK (break command)	15.5,4,11,13-14, 16.2-3
OK (edit command)	9.47,51,62
OK (edita command)	24.12
OKREEVALST (dwim variable/parameter)	17.6
OLD (clisp iterative statement operator)	23.13,5,14
OLDVALUE (system variable/parameter)	5.9
ON PATH (Masterscope set specification)	20.8
ON (clisp iterative statement operator)	23.13,15
OPCODE (in a lap statement)	18.28
OPCODE? - ASSEMBLE (compiler error message)	18.35,24
OPD (property name)	18.25,24,28-29; 24.10
open functions	18.9-10
open macros	18.10
OPENF[FILE;X] SUBR	21.12
OPENFILE[FILE;ACCESS;RECOG;BYTESIZE; MACHINE.DEPENDENT.PARAMETERS]	14.3
OPENHASHFILE[FILE;ACCESS]	24.48
opening files	14.1-4
OPENP[FILE;ACCESS] SUBR	14.4,2
OPENR[A] SUBR	10.15
OPENJFN[FILE;TYPE] SUBR	21.11
OR[X1;X2...;Xn] FSUBR*	5.11
order of precedence of CLISP operators	23.9
(ORF . patterns) (edit command)	9.18
ORG (edita parameter)	24.11
ORIG (used as a readable)	14.24
ORIGINAL (break command)	15.13
ORIGINAL (clisp iterative statement operator) ...	23.17,20
ORIGINAL (file package command)	14.60
(ORIGINAL . coms) (edit command)	9.45-46
(ORR coms1 ... comsn) (edit command)	9.44
OUTFILE[FILE] SUBR	14.2,7-8
OUTFILEP[FILE] SUBR	14.3,2
OUTOF (clisp iterative statement operator)	23.16; 12.15
OUTPUT[FILE] SUBR	14.1
output buffer	14.19

	Page Numbers
OUTPUT FILE? (compiler question)	18.1,3
output functions	14.17-19
OUTPUT (Masterscope command)	20.12
overflow	13.2,5
overlays	3.12-14
(P m n) (edit command)	9.40
(P m) (edit command)	9.40
P (edit command)	9.40,2
P (file package command)	14.57
PACK[X] SUBR	10.2; 3.2-4,9
PACKC[X] SUBR	10.3
PACKFILENAME[FIELDNAME1;FIELDCONTENTS1;...; FIELDNAMEn;FIELDCONTENTSn] *	14.6
PACK*[X] SUBR*	10.2
page	3.9
page mapping	21.13-15
PAGEFAULTS[]	21.4
parentheses counting (by READ)	14.12,32
PARENTHESIS ERROR (error message)	5.2
PARSERELATION[RELATION]	20.19
passwords package	24.44
path options (in Masterscope)	20.15
paths (in Masterscope)	20.14-15
PATLISTPCHECK (in pattern match compiler)	24.2
pattern match compiler	24.1-8
pattern match (in editor)	9.14,15-16,63
(pattern .. @) (edit command)	9.21
PATVARDEFAULT (in pattern match compiler)	24.3,5-6
PB (break command)	15.8
PB (prog. asst. command)	22.24
PEEKC[FILE;RDTBL] SUBR	14.14,33
permanent files	24.34
permstatus package	24.34
PF[FN;FROMFILES;TOFILE] NL*	14.44
PF*[FN;FROMFILES;TOFILE] NL*	14.44
PL (prog. asst. command)	22.23
place-markers (in pattern match compiler)	24.6
PLUS[X1;X2;...;Xn] SUBR*	13.6
PLVLFILFLG (system variable/parameter)	14.19
pmap package	21.13-15
pnames	3.2,1,9; 10.1-3,7
pointer	3.1
POINTER (as a field specification)	3.7
POINTER (record field type)	23.29
POP (change word)	23.38
POSITION[FILE;N] SUBR	14.36
POSSIBILITIES[FORM##] NL	12.16
possibilities lists	12.16
POSSIBLE NON-TERMINATING ITERATIVE STATEMENT (error message)	23.19
POSSIBLE PARENTHESIS ERROR (error message)	23.41
PP[X] NL*	14.43
PP (edit command)	9.40,2
PPE (Masterscope template)	20.16
PPT[X] NL*	23.60,22
PPT (edit command)	9.41; 23.22,60

Page
Numbers

PPV (edit command)	9.41; 14.80
PP*[X] NL*	14.44
PP* (edit command)	9.41
precedence rules (for CLISP operators)	23.7
predicates	5.9; 2.2
prefix operators (in clisp)	23.8
PRESCAN[FILE;CHARLST;PRESCANFN]	24.22
PRETTYCOMFONT	14.50
PRETTYCOMPRINT[X] NL	14.79
PRETTYDEF[PRTTYFNS;PRTTYFILE;PRTTYCOMS; REPRINTFNS;SOURCEFILE;CHANGES]	14.78,64; 19.6
PRETTYEQLVLST (prettyprint variable/parameter) ..	14.49
PRETTYFLG (prettyprint variable/parameter)	14.49,64
PRETTYHEADER (prettyprint variable/parameter) ..	14.79
PRETTYLCOM (prettyprint variable/parameter)	14.48
PRETTYPRINT[FNS;PRETTYDEFLG;FNSLST]	14.43; 2.9
prettyprinting by system functions	14.18
PRETTYPRINTMACROS (prettyprint variable/parameter)	14.49
PRETTYPRINTTYPEMACROS (prettyprint variable/parameter)	14.49
PRETTYTABFLG (prettyprint variable/parameter) ..	14.43
PRETTYTRANFLG (clisp variable/parameter)	23.60; 14.65; 23.22-23
primary input file	14.1,5,11
primary output file	14.1,5,17
primary readable	14.23,11,17,30
primary terminal table	14.29-30
PRINT[X;FILE;RDTBL] SUBR	14.17; 3.1,5-6; 14.18
print name	10.1
PRINTBELLS[]	17.4; 14.18
PRINTBINDINGS[AT;POS]	15.8; 22.24
PRINTDATE[FILE;CHANGES]	14.79
PRINTDEF[EXPR;LEFT;DEF;TAILFLG;FNSLST]	14.79,49,80
PRINTFNS[X;PRETTYDEFLG]	14.79
PRINTHISTORY[HISTORY;LINE;SKIPFN;NOVALUES;FILE]	22.44,18,30-31
printing circular lists	24.15-21
printing numbers	14.20-22
PRINTL[ITEM;DEPTH;LMARG;RMARG;FILE]	24.20
printlevel	14.18-19
PRINTLEVEL[CARN;CDRN] SUBR	14.18; 2.2; 3.5
PRINTMSG (system variable/parameter)	16.9
PRINTNUM[FORMAT;NUMBER;FILE]	14.20
printnum package	14.20-22
printout package	23.43-51
PRINTOUTMACROS (printout parameter)	23.49
PRINTPARA[LMARG;RMARG;LIST;P2FLAG;PARENFLAG;FILE].	23.50
PRINTPROPS[AT]	22.24
PRIN1[X;FILE] SUBR	14.17; 3.2,5-6; 14.18
PRIN2[X;FILE;RDTBL] SUBR	14.17; 3.1,5-6; 14.18
prin2-pnames	10.1-3
PRIN3[X;FILE] SUBR	14.17
PRIN4[X;FILE;RDTBL] SUBR	14.17
private pages	3.11
PRNTL (prog. asst. command)	24.20
PRODUCE[VAL]	12.14
PROG[VARLST;E1;E2;...;En] FSUBR*	5.4
PROG label	5.4
PROGN[X1;X2;...;Xn] FSUBR*	4.3; 5.4

	Page Numbers
programmer's assistant	22.1-36
programmer's assistant and the editor	22.45
programmer's assistant commands	22.9-27
PROG1[X1;X2;...;Xn] FSUBR*	5.4
prompt character	2.3,5-6; 9.1; 15.3, 22.9,27,38
PROMPTCHAR[ID;FLG;HISTORY]	22.38,27,45
PROMPTCHARFORMS (prog. asst. variable/parameter) ..	22.27,38
PROMPTCONFIRMFLG (askuser option)	17.27
PROMPTON (askuser option)	17.27
PROMPT#FLG (prog. asst. variable/parameter)	22.27,38
PROP (as value of DFNFLG)	8.5
PROP (file package command)	14.57,60
PROP (Masterscope template)	20.16
PROP (printed by editor)	9.58
proper tail	5.12
property list	7.1; 2.3; 7.2-3
property name	7.1,3
property value	7.1,3
PROPNAMES[ATM]	7.3
PROPRECORD (record package)	23.28
PROPS (file package command)	14.57
PROTECTION VIOLATION (error message)	16.8
PRXFLG (system variable/parameter)	10.1
pseudo-carriage return	22.15
PSTEP[]	18.26
PSTEPN[N]	18.32
PUSH (change word)	23.37
pushdown list	12.2; 2.6; 4.2; 12.1, 3-4
PUSHLIST (change word)	23.37
PUSHNEW (change word)	23.37
PUTASSOC[KEY;VAL;ALST]	5.13; 23.28
PUTD[FN;DEF] SUBR	8.2; 2.3,5; 8.1-2
PUTDEF[NAME;TYPE;DEFINITION]	14.70
PUTDQ[FN;DEF] NL	8.2
PUTDQ?[FN;DEF] NL	8.2
PUTHASH[ITEM;VAL;HARRAY] SUBR	7.4
PUTHASHFILE[KEY;VALUE;HASHFILE]	24.49
PUTPROP[ATM;PROP;VAL]	7.2
PUTPROPS[ATM;PROP1;VAL1;...;PROPn;VALn] NL*	7.2
Q (edit command)	9.54
Q (following a number)	14.12; 3.3; 14.17,34
QU (exec command)	24.41
QUIT (TENEX command)	21.7,8
QUOTE[X] NL*	5.2
QUOTIENT[X;Y] SUBR	13.6
R (edit command)	9.38,5
radix	10.1-2
RADIX[N] SUBR	14.34; 2.2; 3.3; 14.12, 17
RAISE[FLG;TTBL] SUBR	14.32
(RAISE X) (edit command)	9.50
RAISE (edit command)	9.50
RAISE (TENEX command)	14.32

Page
Numbers

RAND[LOWER;UPPER]	13.8
RANDACCESSP[FILE]	14.8
random numbers	13.8
RANDSET[X]	13.8
RATEST[X] SUBR	14.14
RATOM[FILE;RDTBL] SUBR	14.12, 13, 33
RATOMS[A;FILE;RDTBL]	14.13
(RC x y) (edit command)	9.39
RC (makefile option)	14.65
(RC1 x y) (edit command)	9.39
READ[FILE;RDTBL;FLG] SUBR	14.11, 12, 33
READBUF (prog. asst. variable/parameter)	22.37-38
READC[FILE] SUBR	14.14, 33
READFILE[FILE]	14.42
reading from strings	14.11
READLINE[RDTBL;LINE;LISPXFLG]	14.15; 22.35; 9.56, 14.16; 22.12, 16, 27, 30, 36-37, 45
.....	
READMACROS[FLG] SUBR	14.28
READP[FILE;FLG] SUBR	14.15
READTABLEP[RDTBL] SUBR	14.24
readtables	14.23, 11, 17, 24-29
READVICE (property name)	19.5-6
READVISE[X] NL*	19.5; 14.58; 19.6
read-macro characters	14.25-29
READ-MACRO CONTEXT ERROR (error message)	16.8; 14.28
read-macro options	14.27
REALFRAMEP[POS;INTERPFLG]	12.7
REALSTKNTH[N;POS;INTERPFLG;OLDPOS]	12.8
REANALYZE (Masterscope command)	20.9
REBREAK[X] NL*	15.18, 15
RECLAIM[TYPE]	10.11; 3.10
RECLOOK[RECNAME;TL;LOCALDEC;PARENT;ERROR]	23.35
RECOMPILE[PFILE;CFILE;FNS]	18.7; 14.65-66; 18.5.8, 21
.....	
RECOMPILEDDEFAULT (compiler variable/parameter) ..	18.8, 23
reconstruction (in pattern match compiler)	24.7
record declarations (in clisp)	23.27, 25, 28-32
record package (in clisp)	23.25-36
RECORD (Masterscope template)	20.17
RECORD (record package)	23.28
RECORDACCESS[FIELD;DATUM;DEC;TYPE;NEWVALUE]	23.36
RECORDFIELDNAMES[RECORDNAME]	23.35
RECORDS (file package command)	14.58; 23.27, 29
record-type (record package)	23.27
(REDEFINED) (printed by system)	8.5; 14.38
REDEFINE? (compiler question)	18.3
REDO N TIMES (prog. asst. command)	22.12
REDO (prog. asst. command)	22.12, 15, 18
REDOCNT (prog. asst. variable/parameter)	22.13
REFERENCE (Masterscope relation)	20.5
REHASH[OLDAR;NEWAR] SUBR	7.5
REHASHFILE[HASHFILE]	24.50
relations (in Masterscope)	20.4-6
RELBLK[ADDRESS;N] SUBR	21.6; 16.7
releasing stack pointers	12.12
RELINK[FN;UNLINKFLG]	18.17

Page
Numbers

relinking	18.17-18
relocation information (in arrays)	3.5
RELSTK[POS] SUBR	12.11,12
RELSTKP[X]	12.11
REMAINDER[X;Y] SUBR	13.6
REMAKE (makefile option)	14.65
remaking a file	14.77
REMARK (transor command)	24.26
REMEMBER (prog. asst. command)	22.23; 14.67
REMOVE[X;L]	6.3
REMPROP[ATM;PROP]	7.2
REMPROPLIST[ATM;PROPS]	7.2
RENAME[OLD;NEW;TYPES;FILES;METHOD]	14.71
RENAMEFILE[OLD;NEW]	14.5
REPACK (edit command)	9.51
(REPACK @) (edit command)	9.51
REPEAT (prog. asst. command)	22.13
REPEATUNTIL (clisp iterative statement operator) ..	23.14
REPEATWHILE (clisp iterative statement operator) ..	23.14
REPLACE UNDEFINED FOR FIELD (error message)	23.30
REPLACE (Masterscope relation)	20.5
REPLACE (Masterscope template)	20.17
REPLACE (use in records in clisp)	23.26
(REPLACE @ WITH ...) (edit command)	9.27
REPLACEFIELD[DESCRIPTOR;DATUM;NEWVALUE]	3.8
replacements (in pattern match compiler)	24.6
REREADFLG (prog. asst. variable/parameter)	22.37,39
RESET[] SUBR	16.11
RESET	5.7
RESET (printed by system)	22.33,40
RESETBUFS[FORM1;FORM2;...;FORMn] NL*	14.34
RESETFORM[RESETFORM;FORM1;FORM2;...;FORMn] NL* ..	5.8
RESETFORMS (system variable/parameter)	22.28; 14.35
RESETLST[RESETX] NL*	5.7
RESETREADTABLE[RDTBL;FROM] SUBR	14.24
RESETSAVE[RESETX] NL*	5.7
RESETTERMTABLE[Ttbl;FROM] SUBR	14.29
RESETUNDO[X;STOPFLG]	22.43; 5.9; 22.34
RESETVAR[VAR;NEWVALUE;FORM] NL	5.8; 18.5
RESETVARS[VARSLST;E1;E2;...;En] FSUBR*	5.8
restoring input buffers	22.26
RESULTS[RETURNVALUESFLG]	21.6
RESUME[FROMPTR;TOPTR;VAL] SUBR	12.15
RETAPPLY[POS;FN;ARGS;FLG;INTERNALFLG]	12.10
RETEVAL[POS;FORM;FLG;INTERNALFLG]	12.9; 15.4; 17.10
RETFNS (compiler variable/parameter)	18.14,19,21
RETFROM[POS;VAL;FLG] SUBR	12.10; 15.4; 16.4
RETRIEVE (prog. asst. command)	22.22,18,28
RETRY (prog. asst. command)	22.17,18
RETTO[POS;VAL;FLG] SUBR	12.10
RETURN[X] SUBR	5.5
RETURN (askuser option)	17.27
RETURN (break command)	15.6; 2.9; 15.4,14,
.....	16.1,3
RETURN (Masterscope template)	20.16
RETURN (use in iterative statement in clisp)	23.17
RETURNS (in Decl package)	24.56,55,58

Page
Numbers

RETYPE (syntax class)	14.30
reusing stack pointers	12.13
REUSING (record package)	23.33
REVERSE[L]	6.4
REVERT (break command)	15.12
(RI n m) (edit command)	9.34, 6
RIGHTBRACKET (syntax class)	14.25
RIGHTPAREN (syntax class)	14.25
RLJFN[JFN]	21.12
(RO n) (edit command)	9.34, 6
root name of the file	14.54
RPAQ[X;Y] NL	5.6; 14.38; 22.33
RPAQQ[X;Y] NL	5.6; 14.38, 78; 22.33
RPARKEY (dwim variable/parameter)	17.5
RPLACA[X;Y]	5.2
RPLACD[X;Y]	5.1
RPLNODE[X;A;D]	5.2
RPLNODE2[X;Y]	5.2
RPLSTRING[X;N;Y] SUBR	10.5, 8
RPT[RPTN;RPTF]	8.8
RPTQ[N;FORM1;FORM2;...;FORMn] NL*	8.8
RSH[N;M]	13.4
RSTRING[FILE;RDTBL] SUBR	14.12, 13
RUN (TENEX command)	3.11
running other subsystems from within Interlisp ..	21.7
RUNONFLG (dwim variable/parameter)	17.19-20
run-on spelling corrections	17.19, 4
(R1 x y) (edit command)	9.39
(S var . @) (edit command)	9.23
S (response to compiler question)	18.2-3
SASSOC[KEY;ALST]	5.13
SATISFIES (in Decl package)	24.54
SAVE EXPRS? (compiler question)	18.3
SAVE (edit command)	9.47, 49, 62
SAVEDBFLG (system variable/parameter)	24.32
SAVEDDEF[NAME;TYPE;DEFINITION]	8.6; 14.70
SAVEPUT[ATM;PROP;VAL]	7.2
SAVESET[NAME;VALUE;TOPFLG;FLG]	22.33, 40
SAVESETQ[SETQX] NL*	22.33
SAVESETQQ[SETQX;SETQY] NL	22.33
SCODEP[FN] SUBR	3.14; 8.3, 3
SCRATCHCOLLECT~ (clisp iterative statement operator)	24.37
SCRATCHLIST[LST;X1;X2;...;Xn]	24.36
scratchlist package	24.36
search algorithm (in editor)	9.16-17
searching files	14.8-9
searching strings	10.6-7
SEARCHING... (typed by breakin)	15.17
SEARCHPDL[SRCHFN;SRCHPOS]	12.12
second pass (of the compiler)	18.23
segment patterns (in pattern match compiler)	24.4-5
SELECTQ[X;CLAUSE1;CLAUSE2;...;CLAUSEn;DEFAULT] NL*	5.3, 4
SEPARATE (Masterscope path option)	20.15
separator characters	14.26, 13-15, 33
SEPRCASE[CLISPFLG]	14.9

Page
Numbers

SET[X;Y] SUBR	5.5
set specifications (in Masterscope)	20.6-8
SET (Masterscope relation)	20.4
SET (Masterscope template)	20.16
SETA[A;N;V]	10.9; 3.5; 16.7
SETARG[VAR;M;X] FSUBR	8.9
SETATOMVAL[ATM;VAL] SUBR	5.6
SETBLIPVAL[BLIPTYP;IPOS;N;VAL] SUBR	12.4
SETBRK[LST;FLG;RDTBL] SUBR	14.13
SETD[A;N;V]	10.10; 3.5
SETDECLTYPEPROP[TYPE;PROP;VAL]	24.65
SETERRORN[NUM;MESS] SUBR	16.11
SETFILEINFO[FILE;ATTRIB;VALUE]	14.5
SETFILEPTR[FILE;ADR] SUBR	14.8, 36
SETFN (property name)	23.54
SETINITIALS[]	14.37
SETLINELENGTH[N]	14.35
SETN[VAR;X] FSUBR	13.10, 8-9
SETPROPLIST[ATM;LST]	7.1; 2.3
SETQ[X;Y] FSUBR*	5.5
SETQ (in an assemble statement)	18.26
SETQQ[X;Y] NL	5.5
SETREADMACROFLG[FLG] SUBR	14.28
SETREADTABLE[RDTBL;FLG] SUBR	14.24
sets (in Masterscope)	20.6-8
SETSBSIZE[N] SUBR	3.14; 16.8
SETSEPR[LST;FLG;RDTBL] SUBR	14.13
SETSTKARG[N;POS;VALUE] SUBR	12.8
SETSTKARGNAME[N;POS;NAME] SUBR	12.8
SETSTKNAME[POS;NAME] SUBR	12.7
SETSYNONYM[NEWPHRASE;MEANING]	20.19
SETSYNTAX[CH;CLASS;TABLE]	14.25
SETTEMPLATE[FN;TEMPLATE]	20.18
SETTERMCHARS[NEXTCHAR;BKCHAR;LASTCHAR; UNQUOTECHAR;2CHAR;PPCHAR]	14.36; 9.14; A1.3
SETTERMTABLE[TTBL] SUBR	14.29
SETTOPVAL[ATM;VAL] SUBR	5.6; 2.3
SETTYPEDESCRIPTION[TYPE;STRING]	10.13
SETWORDCONTENTS[PTR;N]	21.14
SHALL I LOAD (system variable/parameter)	17.13
shallow binding	12.1; 5.6; 18.4
shared pages	3.11
shared system	3.11
sharing	3.11
SHOULD BE A SPECVAR (compiler error message)	18.34
SHOULDN'T []	16.10
SHOULDN'T HAPPEN (error message)	16.10
SHOW PATHS (Masterscope command)	20.10
SHOW WHERE (Masterscope command)	20.11
SHOW (transorset command)	24.26
(SHOW . x) (edit command)	9.44
SHOWDEF[NAME;TYPE;FILE]	14.70
SHOWPRINT[X;FILE;RDTBL]	14.18; 15.8-9, 19; 22.9
SHOWPRIN2[X;FILE;RDTBL]	14.18; 22.18, 44
SIDE (property name)	22.35, 41, 43, 45
SIN[X;RADIANSLFG]	13.7
SINGLEFILEINDEX[FILE;OUTPUTFILE;NEWPAGEFLG]	24.31

	Page Numbers
singlefileindex package	24.31
single-stepping a program	9.42
SKOR[XWORD;TWORD;NCX;NCT;FLG]	17.16-17
SKREAD[FILE;REREADSTRING]	14.16
small integers	3.3,1; 5.10; 13.1
SMALLP[N]	13.4; 3.3; 13.1
SMARTARGLIST[FN;EXPLAINFLG;TAIL]	8.4
SMASH (Masterscope relation)	20.4
SMASH (Masterscope template)	20.16
SMASHFILECOMS[FILE]	14.76
SOME[SOMEX;SOMEFN1;SOMEFN2]	5.12
SORRY, I CAN'T PARSE THAT (error message)	20.20
SORRY, NO FUNCTIONS HAVE BEEN ANALYZED (error message)	20.20
SORRY, THAT ISN'T IMPLEMENTED (error message)	20.20
SORT[DATA;COMPAREFN]	6.7
SP (in an assemble statement)	18.26
space	3.1
SPACES[N;FILE] SUBR	14.17
spaghetti stacks	12.4-13
SPECIAL (in Decl package)	24.56
SPECVARS (compiler variable/parameter)	18.18; 5.8; 18.19,21
SPECVARS (file package command)	14.59
SPELLFILE[FILE;NOPRINTFLG;NSFLG]	17.21; 14.2; 16.7,10
spelling completion	17.7
spelling correction	17.7; 9.56-57,59,61, 14.59-60; 15.13, 17.8-10; 22.12,31, 23.7,10-11,55
spelling correction on file names	17.21
spelling correction on hash files	24.51
spelling correction protocol	17.4-5
spelling corrector	17.7,2,8-10,16-17 17.8; 9.56-57,59, 14.59-60; 15.13; 17.9, 12-14; 22.12,31, 23.7,10-11,55
spelling lists	17.8,9,14,18 17.8,9,13-14,18 17.8,9,12,18; 22.40
SPELLINGS1 (dwim variable/parameter)	14.26
SPELLINGS2 (dwim variable/parameter)	9.51
SPELLINGS3 (dwim variable/parameter)	4.2; 8.1
SPICE (type of read-macro)	4.2
(SPLITC x) (edit command)	13.7
spread functions	13.7
spreading arguments	13.7
SQRT[N]	14.48
SQRT OF NEGATIVE VALUE (error message)	2.4
square brackets (inserted by prettyprint)	6.9
square brackets (use in input)	18.1-3
SRCCOM	12.6
ST (response to compiler question)	12.6-12
stack descriptor	16.5
stack functions	16.5; 12.12
STACK OVERFLOW IN GC - COMPUTATION LOST (error message)	12.5
STACK OVERFLOW (error message)	12.6
stack pointer	12.6
STACK POINTER HAS BEEN RELEASED (error message)	

Page
Numbers

STACK PTR HAS BEEN RELEASED (error message)	16.7	
STACKP[X] SUBR	12.11	
statistics	22.46	
STF (response to compiler question)	18.2-3	
STKAPPLY[POS;FN;ARGS;FLG;INTERNALFLG]	12.9	
STKARG[N;POS] SUBR	12.8; 15.8	
STKARGNAME[N;POS] SUBR	12.8	
STKARGS[POS;NARGS]	12.9	
STKEVAL[POS;FORM;FLG;INTERNALFLG]	12.9; 15.8	
STKNAME[POS] SUBR	12.7	
STKNARG\$[POS] SUBR	12.8	
STKNTH[N;IPOS;OPOS] SUBR	12.7	
STKNTHNAME[N;POS] SUBR	12.7	
STKPOS[FRAMENAME;N;IPOS;OPOS] SUBR	12.6	
STKSCAN[VAR;IPOS;OPOS] SUBR	12.8	
STOP (at the end of a file)	14.38,42 9.47,51,62; 15.16	
STOP (edit command)	10.14 3.9	
STORAGE[FLG;GCFLG]	16.7	
storage allocation	10.4	
STORAGE FULL (error message)	18.2,3,5 3.7,1,6,9; 10.7	
STREQUAL[X;Y] SUBR	10.4-7 3.7,1,9; 10.5,7-8	
STRF (compiler variable/parameter)	10.7-8	
string characters	14.25 5.9; 10.4	
string functions	3.6; 5.9; 14.11	
string pointers	10.6; 14.8	
string storage	10.7 9.24-40	
STRINGDELIM (syntax class)	23.37	
STRINGGP[X] SUBR	10.6	
strings	23.32	
STRPOS[PAT;STRING;START;SKIP;ANCHOR;TAIL]	6.5	
STRPOSL[A;STR;START;NEG]	6.5	
structure modification commands (in editor)	4.2; 8.3-4	
structure modification (the changetran package)	8.6	
SUBATOM[X;N;M]	8.3,1-3	
subdeclarations (record package)	8.1; 2.2	
SUBLIS[ALST;EXPR;FLG]	4.2; 8.3-4	
SUBPAIR[OLD;NEW;EXPR;FLG]	11.2	
SUBR (function type)	6.4,5	
SUBR (property name)	18.11	
SUBRP[FN] SUBR	10.5; 3.6; 10.8	
subrs	21.7,8; 22.28	
SUBR* (function type)	24.66	
SUBSET[MAPX;MAPFN1;MAPFN2]	24.60	
SUBST[NEW;OLD;EXPR]	13.2	
substitution macros	22.11	
SUBSTRING[X;N;M;OLDPTR] SUBR	23.12	
SUBSYS[FILE/FORK;INCOMFILE;OUTCOMFILE; ENTRYPOINTFLG]	24.66	
SUBTYPES[TYPE]	supertypes (in Decl package)	24.60
SUB1[X]	(SURROUND @ IN ...) (edit command)	9.31
SUCHTHAT (in event address)	SUSPICIOUS PROG LABEL (error message)	23.41

	Page Numbers
SVFLG (compiler variable/parameter)	18.2-3
(SW n m) (edit command)	9.39-40
SWAP (change word)	23.38
(SWAP @1 @2) (edit command)	9.40
SWAPBLOCK TOO BIG FOR BUFFER (error message)	16.8
SWAPC (edit command)	9.51
swappable array	3.12
swapping buffer	3.13
SWPARRAY[N;P;V] SUBR	3.14
SWPARRY[P] SUBR	3.14; 10.9
SY (prog. asst. command)	22.28
symbolic file input	14.38-42
SYMLST (edita parameter)	24.13
synonyms	17.7
syntax classes	14.24-30
SYNTAXP[CHARCODE;CLASS;TABLE]	14.25
SYSBUFP[FLG] SUBR	14.34
SYSHASHARRAY (system variable/parameter)	7.4-5
SYSSIN[FILE] SUBR	14.37; 2.9
SYSLINKEDFNS (system variable/parameter)	18.17
SYSLOAD (load option)	14.38,39; 17.13
SYSOUT[FILE] SUBR	14.36; 2.9; 14.38
SYSOUTDATE (system variable/parameter)	14.37
SYSOUTFILE (system variable/parameter)	14.37
SYSOUTGAG (system variable/parameter)	14.37
SYSOUTP[FILE]	14.38
SYSOUT.EXT (system variable/parameter)	14.37
SYSPRETTYFLG (system variable/parameter)	14.18; 15.8-9,19; 22.9, 18.44
SYSPROPS (system variable/parameter)	7.3; 14.57
SYSTEMFONT	14.50
SYSTEMTYPE[] SUBR	21.9
 T FIXED (printed by dwim)	17.6
T (printout command)	23.46
TAB[POS;MINSPACES;FILE]	14.17
tab (edita command)	24.11
tail of a list	5.12
TAILP[X;Y]	5.12
TAN[X;RADIANSF]	13.8
TCOMPL[FILES]	18.5; 14.65; 18.6-7, 21-22
TCONC[PTR;X]	6.2,3
TELNET[CONNECTION;TYPE;SKT;BUTTONCHAR]	24.44; 21.15
telnet package	24.44
TEMPLATES (file package command)	14.58
templates (in Masterscope)	20.16-18
TENEX[STR;FILEFLG]	21.9
TENEX	2.1,3,5,9; 3.1,4,11, 13.10; 14.8; 21.11-12
terminal	9.41; 14.1,5,11-12,15, 32.44
terminal initiated breaks	16.2
terminal syntax classes	14.30
terminal tables	14.29-33
TERMTABLEP[TTBL] SUBR	14.29
TERPRI[FILE] SUBR	14.17

Page
Numbers

TEST (edit command)	9.55
TEST (Masterscope relation)	20.5
TEST (Masterscope template)	20.16
TEST (transorset command)	24.26
TESTMODE[FLG]	22.32
TESTMODEFLG (prog. asst. variable/parameter)	22.32
TESTRELATION[ITEM;RELATION;ITEM2;INVERTED]	20.19
THE (in Decl package)	24.59
THEN (in Decl package)	24.57
THEREIS (clisp iterative statement operator)	23.12
THOSE (Masterscope set specification)	20.8
THRU (edit command)	9.35-38
THRU (in event specification)	22.11
TIME[FORM;CNT] NL	21.1,2
time stamp package	9.59; 8.5
TIMES[X1;X2;...;Xn] SUBR*	13.6
time-slice (of history list)	22.7,40
TO (clisp iterative statement operator)	23.14,15-16
TO (edit command)	9.35-38
TO (in event specification)	22.11
TO (Masterscope path option)	20.15
too few arguments	4.2
too many arguments	4.2
TOO MANY FILES OPEN (error message)	16.6
TOO MANY USER INTERRUPT CHARACTERS (error message)	16.8
top level value	5.6
TOP (as argument to advise)	19.3-4
TRACE[X] NL*	15.16,1,5,13,17
translation notes (in transor package)	24.22-24
translations (in clisp)	23.22-23
transor package	24.21-29
transor sweep	24.28
TRANSORSET[]	24.24,21
TRAP AT LOCATION (error message)	16.5; 21.9
TREAT AS CLISP ? (printed by dwim)	23.51
TREATASCLISPFLG (clisp variable/parameter)	23.51
TREATED AS CLISP (printed by dwim)	23.51
true	2.2; 5.3
TRUSTING (DWIM mode)	17.3,2,17; 23.3,51-52
TRYNEXT[PLST##;ENDFORM##;VAL##] NL	12.17
TTYLINELENGTH (system variable/parameter)	14.35
TTY#[]	24.43
TTY: (edit command)	9.49,44,47; 15.16
TTY: (printed by editor)	9.49
TY (exec command)	24.42
type declarations	24.53-68
type description	10.13
type names	10.10-11
type numbers	10.10-11
TYPE (Masterscope relation)	24.67
typed definitions	14.55-56
TYPENAME[DATUM] SUBR	10.10
TYPENAMEFROMNUMBER[N]	10.11
TYPENAMEP[DATUM;TYPENAME]	10.11
TYPENUMBERFROMNAME[NAME]	10.11
TYPEP[DATUM;N]	10.11
TYPERECORD (record package)	23.28

	Page Numbers
types (in Masterscope)	20.8-9
typescript files	21.15
TYPESOF[NAME;POSSIBLETYPES;IMPOSSIBLETYPES]	14.72
TYPE-AHEAD (prog. asst. command)	22.25
TYPE? NOT IMPLEMENTED FOR THIS RECORD (error message)	23.34
TYPE? (record package)	23.33,28,34
TYPE? (use in record declarations)	23.32
 U (value of ARGLIST)	8.4
UB (break command)	15.6
UCASELST (prettyprint variable/parameter)	14.46
UGLYVARS (file package command)	14.59,23
UNABLE TO ALLOCATE PMAP BUFFER (error message) ..	21.13
UNABLE TO DWIMIFY (error message)	23.42
UNADVISE[X] NL*	19.5,4,6
UNADVISED (printed by system)	15.19
UNARYOP (property name)	23.54
UNBLOCK (edit command)	9.55
unbound atom	16.1; 8.7; 17.10-14
unboxed numbers	13.10
unboxed numbers (in arrays)	3.5; 10.8
unboxing	13.1,2,10
UNBREAK[X] NL*	15.17,15,18; 21.5
(UNBREAKABLE)	15.17
UNBREAKIN[FN]	15.18,18
UNBREAKO[FN;TAIL]	15.18
UNBROKEN (printed by system)	15.19
UNBROKEN (typed by advise)	19.4
UNBROKEN (typed by compiler)	18.5
UND (exec command)	24.42
undefined function	16.1; 17.10-14
UNDEFINED OR ILLEGAL GO (error message)	16.5; 5.5
UNDEFINED TAG (compiler error message)	18.35
UNDEFINED TAG, ASSEMBLE (compiler error message) ..	18.35
UNDEFINED TAG, LAP (compiler error message)	18.35
UNDEFINED USER INTERRUPT (error message)	16.12
UNDO (edit command)	9.54,7; 22.45
UNDO (prog. asst. command)	22.18; 17.3; 22.12,33, 42,45
undoing	22.31,4,32-33,40-43,45
undoing DWIM corrections	22.18; 23.42
undoing out of order	22.33,18
undoing (in editor)	9.54,7,24,55; 22.45
UNDOLISPX[LINE]	22.42
UNDOLISPX1[EVENT;FLG;DWIMCHANGES]	22.42
UNDOLST (editor variable/parameter)	9.54,47,55,63; 22.45
UNDONE (printed by editor)	9.54
UNDONE (printed by system)	22.18,43
UNDONLSETQ[FORM] NL	22.43,34
UNDOSAVE[UNDOFORM;HISTENTRY]	22.41,35
UNFIND (editor variable/parameter)	9.23,17,27-28,30-33, 47-48,53,63
 UNION[X;Y]	6.7
UNLESS (clisp iterative statement operator)	23.14
UNLOCKMAP[PTR]	21.15
UNMARKASCHANGED[NAME;TYPE]	14.67

	Page Numbers
UNPACK[X;FLG;RDTBL] SUBR	10.2
UNPACKFILENAME[FILENAME]	14.6
unreading	22.9, 15, 38
UNSAVED (printed by dwim)	17.12-13
UNSAVEDEF[NAME;TYPE;DEF]	8.6; 14.71; 17.12-13
UNSAVEFNS[FNS]	20.21
UNSET[NAME]	22.41, 33
UNTIL (clisp iterative statement operator)	23.14
UNTIL (use with REDO)	22.12
UNUSUAL CDR ARG LIST (error message)	16.7
UP (edit command)	9.10, 9, 11, 17, 28
UPDATECHANGED[]	20.20
UPDATEFILES[PRLST;FLST]	14.63
UPDATEFN[FN;EVENIFVALID]	20.20
updating files	14.63
UPFINDFLG (editor variable/parameter)	9.29, 17-18
USE AS A CLISP WORD (Masterscope relation)	20.5
USE AS A FIELD (Masterscope relation)	20.5
USE AS A PROPERTY (Masterscope relation)	20.5
USE AS A RECORD (Masterscope relation)	20.5
USE (Masterscope relation)	20.4
USE (prog. asst. command)	22.13-14, 15, 18
USED AS ARG TO NUMBER FN? (compiler error message)	18.34
USED BLKAPPLY WHEN NOT APPLICABLE (compiler error message)	18.34
USEDFREE (clisp declaration)	23.41, 43
USEDIN (in Decl package)	24.56, 58
USEMAPFLG (system variable/parameter)	14.42
USER BREAK (error message)	16.8
user defined printing	14.22
user interrupt characters	16.12; 2.4
USERDATATYPES[]	3.9
USEREXEC[LISPXID;LISPXXMACROS;LISPXXUSERFN]	22.37
USERFONT	14.50
USERLISPXPRINT[X;FILE;Z;NODOFLG]	22.30
USERMACROS (editor variable/parameter)	9.46; 14.58
USERMACROS (file package command)	14.58; 9.46, 56
USERNAME[A]	21.9
USERNAME (prog. asst. variable/parameter)	22.47
USERNUMBER[A;FLG]	21.9
USERRECORDTYPE (property name)	23.31
USERSYMS (edita parameter)	24.13
USERWORDS (dwim variable/parameter)	17.9; 9.59, 61; 17.18, 20-21
USE-ARGS (property name)	22.34
USING (record package)	23.33
U-CASE[X]	10.4; 9.50
U-CASEP[X]	10.4
U.B.A. breaks	15.9
U.B.A. (error message)	16.1; 2.6; 16.3; 17.10
U.D.F. breaks	15.10
U.D.F. T FIX? (printed by dwim)	17.5
U.D.F. T (printed by dwim)	17.6
U.D.F. (error message)	16.1, 3; 17.2, 10
VAG[X] SUBR	13.11, 10
value cell	12.1; 3.2; 5.6; 16.1

	Page Numbers
value of a break	15.4; 16.1
value of a property	7.1
VALUE OUT OF RANGE EXPT (error message)	13.7
VALUE (property name)	5.6; 22.33,40-41
VALUEOF[X] NL*	22.26,40; 21.8; 22.35
variable bindings	12.1; 2.6; 11.3-4, 12.2-4
VARIABLES[POS]	12.8; 15.9
VARS[FN;USEDATABASE]	14.39
VARS (file package command)	14.56
VARTYPE (property name)	14.62
version numbers	14.2
VIRGINFN[FN;FLG]	15.19
WAITFORINPUT[FILE]	14.15
WHEN (clisp iterative statement operator)	23.14
WHENCLOSE[FILENAME;PROP1;VAL1;...;PROPN;VALn] *	14.10; 24.34,49
whenclose package	14.9-11
WHERE (clisp iterative statement operator)	23.20
WHEREIS[NAME;TYPE;FILES]	14.67; 24.34
whereis package	24.34; 9.64; 14.40,67, 69; 24.35
WHEREIS.HASH (system variable/parameter)	24.34
WHILE (clisp iterative statement operator)	23.14
WHILE (use with REDO)	22.12
WIDEPAPER[FLG]	14.48
WITH (in REPLACE command) (in editor)	9.27
WITH (in SURROUND command) (in editor)	9.31
WORDCONTENTS[PTR]	21.14
WORDOFFSET[PTR;N]	21.14
WORLD (as argument to RELINK)	18.17
WRITEFILE[X;FILE]	14.42
(XTR . @) (edit command)	9.29
XWD[N1;N2]	24.36
ZEROP[X]	13.3
[.....	3.1
[,] (inserted by prettyprint)	14.48
{ } (use with ftp package)	24.45
(change character)	14.49; 9.25
~ (clisp operator)	23.9
~ (in pattern match compiler)	24.4
! (in pattern match compiler)	24.4-6
! (use in history commands)	22.13
! (use with <,> in clisp)	23.10
!E (edit command)	9.52; 22.45
!EVAL (break command)	15.5
!F (edit command)	9.52; 22.45
!GO (break command)	15.6,14
!IN (edit command)	9.52; 22.45
!NX (edit command)	9.13

Page
Numbers

!OK (break command)	15.5, 14
!UNDO (edit command)	9.55
!VALUE (break variable/parameter)	15.5, 14
!VALUE (use with advising)	19.1, 3
!! (use with <,> in clisp)	23.10
!O (edit command)	9.12
"	3.6, 1; 14.11-12, 14
"" (use in askuser)	17.29
"<c.r.>" (use in history commands)	22.15, 37
# (followed by a number)	14.17; 3.5; 10.9
# (printout command)	23.49
#CAREFULCOLUMNS (prettyprint variable/parameter)	14.48
#n (n a number, in pattern match compiler)	24.6
#RPARS (prettyprint variable/parameter)	14.48
#SPELLINGS1 (dwim variable/parameter)	17.9
#SPELLINGS2 (dwim variable/parameter)	17.9
#SPELLINGS3 (dwim variable/parameter)	17.9
#UNDOSAVES (prog. asst. variable/parameter)	22.31, 41, 43
#USERWORDS (dwim variable/parameter)	17.9
##[COMS] NL*	9.42, 19
## (in INSERT, REPLACE, and CHANGE commands)	9.28
## (printed by system)	14.32; 2.4; 14.11, 31, 33
#0	12.12
\$ (dollar) (edita parameter)	24.12
\$ (dollar) (in pattern match compiler)	24.4
\$ (<esc>)	14.2
\$ (<esc>) (in clisp)	23.8-9
\$ (<esc>) (in edit pattern)	9.8, 15
\$ (<esc>) (in R command) (in editor)	9.38
\$ (<esc>) (in spelling correction)	17.7, 18
\$ (<esc>) (prog. asst. command)	22.19-20
\$ (<esc>, in R command) (in editor)	9.39
\$ (<esc>, use in askuser)	17.28
\$BUFS (<esc>BUFS) (prog. asst. command)	22.26; 9.5; A1.1
\$C (<esc>C) (edita command)	24.14
\$n (in pattern match compiler)	24.4
\$Q (<esc>Q) (edita command)	24.12
\$W (<esc>W) (edita command)	24.13, 14-15
\$\$ (two <esc>s) (in edit pattern)	9.15
\$1 (in pattern match compiler)	24.3
% (escape character)	14.11; 2.4; 3.1, 6, 14.12-14, 17, 33
% (use in comments)	14.46
%% (use in comments)	14.45-46
& (in edit pattern)	9.8, 14
& (in pattern match compiler)	24.3
& (printed by editor)	9.2
& (printed by system)	14.19
& (use in askuser)	17.28
'	17.10
' (as a read-macro)	14.26; 2.5

Page
Numbers

' (clisp operator)	23.8
' (edita command)	24.13,10
' (in a lap statement)	18.28
' (in pattern match compiler)	24.3
' (Masterscope set specification)	20.6
(.....	3.1
()	3.4
)	3.1
* (as a prettyprint macro)	14.45
* (as a read-macro)	14.45
* (in a lap statement)	18.29
* (in an assemble statement)	18.26
* (in MBD command) (in editor)	9.30
* (in pattern match compiler)	24.4
* (printed by editor)	9.1
* (use in comments)	14.44,48
* (use in file package command)	14.60
(* . x) (edit command)	9.53
ANY (in edit pattern)	9.14
ARCHIVE (property name)	22.28,23
ARGVAL (as a blip on the stack)	12.4
*ARG1 (as a blip on the stack)	12.3
ERROR (property name)	22.34,19
FN (as a blip on the stack)	12.4
FORM (as a blip on the stack)	12.4
GROUP (property name)	22.34,35,39
HISTORY (property name)	22.34,35
LISPXPRINT (property name)	22.31,34
PRINT (property name)	22.34
TAIL (as a blip on the stack)	12.4
BREAK (in backtrace)	15.8
COMMENT (printed by editor)	9.40
COMMENT (printed by system)	14.44
COMMENTFLG (prettyprint variable/parameter) ..	14.44; 9.41
EDITOR (in backtrace)	15.8
TOP (in backtrace)	15.8
***** (in compiler error messages)	18.33
, (edita command)	24.10
, (printout command)	23.45
- (in argument list, in manual)	2.3
(-n e1 ... em) (n a number, edit command)	9.24,4
-n (n a number, edit command)	9.11,2
-n (n a number, printout command)	23.45
-- (in edit pattern)	9.8,15
-- (in pattern match compiler)	24.4
-- (printed as part of a list structure)	14.19
-- (printed by editor)	9.2
-> (break command)	15.10
-> (in pattern match compiler)	24.7
-> (printed by dwim)	17.4,2-3,5
-> (printed by editor)	9.39

Page
Numbers

. notation	3.4
. (edita parameter)	2.1
. (in a floating point number)	24.12
. (in pattern match compiler)	3.4
. (printed by Masterscope)	24.5
. (use in Masterscope)	20.3
.BASE (printout command)	20.3
.CENTER (printout command)	23.46
.CENTER2 (printout command)	23.48
.F (printout command)	23.48
.FONT (printout command)	23.49
.FR (printout command)	23.46
.FR2 (printout command)	23.48
.I (printout command)	23.48
.N (printout command)	23.49
.PAGE (printout command)	23.46
.PARA (printout command)	23.47
.PARA2 (printout command)	23.47
.PPF (printout command)	23.47
.PPFTL (printout command)	23.47
.PPV (printout command)	23.47
.PPVTL (printout command)	23.47
.P2 (printout command)	23.47
.RESET (printout command)	23.45
.SKIP (printout command)	23.46
.SP (printout command)	23.46
.SUB (printout command)	23.45
.SUP (printout command)	23.46
.TAB (printout command)	23.45
.TAB0 (printout command)	23.45
.. (edit command)	9.21
.. (in edit pattern)	9.15
.. (Masterscope template)	20.17
.. (in edit pattern)	9.15-16
.. (printed by dwim)	17.3-4
.. (printed by editor)	9.9-10
.. (printed following a carriage-return)	14.16; 22.36
.. (prog. asst. command)	22.17,18
/ functions	22.32,42
/ (edita command)	24.11,10
/DELFILE[FILE]	24.43
/REPLACE	23.26
/RPLNODE[X;A;D]	22.41,42
/RPLNODE2[X;Y]	22.42
/UNDELFILE[FILE]	24.43
0 (edit command)	9.12,3
(2ND . 0) (edit command)	9.20
(3RD . 0) (edit command)	9.20
7 (instead of ')	17.11
8 (instead of left parenthesis)	17.5; 9.57; 17.1,11, 13-14

Page
Numbers

9 (instead of right parenthesis)	17.5,1,11,13
(: e1 ... em) (edit command)	9.26,9
: (clisp operator)	23.8
: (edita command)	24.13
: (printed by system)	15.3; 2.6
; (edita command)	24.15
; (prog. asst. command)	22.24
	2.4; 14.14,32; A1.1
 (TOPS-20)	2.4; 14.30; A1.2
<esc> (in spelling correction)	17.7,18
<esc> (prog. asst. command)	22.19-20
<esc>BUFS (prog. asst. command)	22.26; A1.1
<,> (use in clisp)	23.10
= (break command)	15.9
= (edita command)	24.12
= (in a lap statement)	18.28
= (in event address)	22.10
= (in pattern match compiler)	24.3
= (printed by dwim)	17.4-5
= (printed by editor)	9.8
= (use with @ break command)	15.6
=E (printed by editor)	9.57
=EDITF (printed by editor)	9.61
=EDITP (printed by editor)	9.59
=EDITV (printed by editor)	9.59
== (in edit pattern)	9.15
== (in pattern match compiler)	24.3
=> (in pattern match compiler)	24.7
? (edit command)	9.40,2
? (edita command)	24.12
? (printed by dwim)	17.4-5
? (printed by editor)	9.2
? (printed by Masterscope)	20.16
? (printed by system)	16.3
? (prog. asst. command)	22.24; 2.5
?= (break command)	15.6-8
?= (edit command)	9.41; 2.5
?? (prog. asst. command)	22.17,18
@ (break command)	15.6,7,11
@ (edita command)	24.10
@ (in a lap statement)	18.28
@ (in event specification)	22.39
@ (in pattern match compiler)	24.3,5
@ (location specification) (in editor)	9.19
@ (Masterscope set specification)	20.6
@ (Masterscope template)	20.17
(@1 THRU) (edit command)	9.37
(@1 THRU @2) (edit command)	9.35
(@1 TO) (edit command)	9.37
(@1 TO @2) (edit command)	9.35

	Page Numbers
@@ (in event specification)	22.12,23,39
(\ atom) (edit command)	9.22
\ (edit command)	9.23,8,27
\ (in event address)	22.10
\ (printed by system)	2.4; 14.11,31-32
\P (edit command)	9.23,8,41
\\ (printed by system)	14.12; 2.4
] (use in input)	2.4; 3.1; 14.15
↑ (break command)	15.5,14; 16.1-2,4
↑ (edit command)	9.12,3
↑ (edita command)	24.12
↑ (use in comments)	14.46
← operator (in clisp)	23.8-9
(← pattern) (edit command)	9.20
← (edit command)	9.22
← (in event address)	22.10
← (in pattern match compiler)	24.5
← (printed by system)	2.3,5; 15.3
← (use in record declarations)	23.31
↔ (edit command)	9.22

