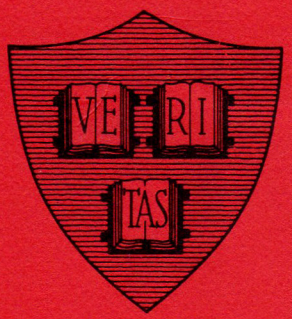
ECL PROGRAMMER'S MANUAL[*]

Center for Research in Computing Technology



Harvard University
Cambridge, Massachusetts 02138

ECL PROGRAMMER'S MANUAL[*]

23-74

CENTER FOR RESEARCH IN COMPUTING TECHNOLOGY
Harvard University
Cambridge, Massachusetts
December 1974

[*] This work was supported in part by the U. S. Air Force Electronic Systems Division under Contract F19628-71-C-0173 and by the Advanced Research Projects Agency under Contracts F19628-71-C-0174 and F19628-74-C-0083.

PREFACE

The ECL programming system is based on the programming language EL1 which is the work of Ben Wegbreit and is described in his doctoral dissertation Studies in Extensible Languages, Harvard University, June 1970.

This manual was written by Glenn Holloway, Judy Townley, Jay Spitzen and Ben Wegbreit.

The parsing algorithm is a modification of F. DeRemer's method. The parser production system was written by Pat Griffiths, Charles Prenner and Judy Townley.

The interpreter was written by Ben Wegbreit.

The storage management routines and garbage collector were written by William Conrad and Glenn Holloway.

The compiler for data type definitions was written by Ben Brosgol and William Conrad.

The facility for control of data type behavior described in section 4 was also implemented by William Conrad, who has rewritten and improved substantial portions of the system.

Steve German designed and implemented the syntax-directed list structure editor and describes it in section 5.

The compiler for procedures, the work of Glenn Holloway, is described in section 6 and more fully developed in his doctoral dissertation (forthcoming).

The utility routines described in sections 7 and 8 were written by Mark Davis, Glenn Holloway, George Mealy, Jay Spitzen, and Ben Wegbreit.

The non-deterministic control facility is the work of Jay Spitzen and is described as part of his doctoral dissertation Approaches to Automatic Programming, Harvard University, June 1974 and in Proceedings ACM 72.

Numerous individuals have made suggestions for the design and implementation of this system. These include Daniel Bobrow, Thomas E. Cheatham, Jr., and Robert Kierr. Their assistance is gratefully acknowledged.

Special thanks go to Terry Sack for her patience and diligence in producing this document.

December 1974

TABLE OF CONTENTS

Page

1. INTRODUCTION..... 1

1.1 AN OVERVIEW OF THE ECL PROGRAMMING SYSTEM..... 1

1.2 PURPOSE OF THIS MANUAL..... 2

2. INTRODUCTION TO EL1..... 3

2.1 FORMS..... 3

2.2 CONSTANTS AND THE BASIC DATA TYPES..... 3

2.3 VARIABLES..... 5

2.4 OPERATOR EXPRESSIONS..... 7

2.5 COMPOUND FORMS..... 13

2.5.1 Declarations..... 14

2.5.1.1 Local Variables..... 14
2.5.1.2 Initialization..... 16

2.5.2 Statements..... 17

2.6 ITERATION..... 18

2.7 MODE-VALUED FORMS..... 20

2.7.1 Mode-Valued Constants and Identifiers.... 20
2.7.2 VECTOR and SEQ..... 21
2.7.3 STRUCT..... 24
2.7.4 REF..... 27
2.7.5 PTR..... 28
2.7.6 Generic Modes..... 30
2.7.7 Summary..... 31

2.8 SELECTION..... 32

2.9 DATA GENERATION..... 34

2.9.1 Default Generation..... 34
2.9.2 Generation by SIZE..... 35
2.9.3 Generation from Components..... 36
2.9.4 Generation by Example..... 36
2.9.5 A Model for Generation..... 37

2.10 PROCEDURES..... 38

TABLE OF CONTENTS

2.11 BIND-CLASSES..... 43

2.12 CASE FORMS..... 46

2.13 COMMENTS..... 49

2.14 << AND RETURN..... 49

2.14.1 An Example..... 50
2.14.2 <<..... 50
2.14.3 RETURN..... 51

2.15 INPUT-OUTPUT..... 51

2.15.1 Ports..... 52

2.15.1.1 OPEN, DRAIN, and CLOSE..... 52
2.15.1.2 MAKEPF..... 53
2.15.1.3 End of File Handling..... 53
2.15.1.4 System Ports..... 54

2.15.2 Character Input/Output..... 55

2.15.2.1 INCHAR and OUTCHAR..... 55
2.15.2.2 LEX, PARSE, READ, and LOAD... 55
2.15.2.3 Printing..... 56
2.15.2.4 Formatting and Updating The Unparser..... 57

2.15.3 DUMPB and LOADB..... 57

2.16 STRONG PROCEDURE MODES..... 58

3. HOW TO USE ECL..... 63

3.1 GETTING INTO ECL..... 63

3.2 THE TOP-LEVEL ENVIRONMENT..... 63

3.3 CREATING AND EDITING PROGRAMS..... 66

3.4 SIMPLE INPUT/OUTPUT ROUTINES: READ AND PRINT... 67

3.5 ERRORS AND PROGRAM SUSPENSION..... 69

3.5.1 Execution Errors..... 69
3.5.2 Setting Breakpoints..... 70
3.5.3 User-Defined Error-Handling Routines..... 71
3.5.4 Emergency Measures..... 71

4. EXTENDED MODES AND THEIR BEHAVIOR..... 73

TABLE OF CONTENTS

4.1 EXTENDED MODES..... 74

4.1.1 Completion..... 74
4.1.2 Tags..... 74
4.1.3 The :: Operator..... 75
4.1.4 LIFT and LOWER..... 76

4.2 MODE BEHAVIOR..... 76

4.2.1 User-Defined Generation..... 78

4.2.1.1 Format of the UGF..... 78
4.2.1.2 When Is a User Generation Function Called?..... 79
4.2.1.3 SUPUGF..... 81
4.2.1.4 CONSTRUCT..... 82
4.2.1.5 An Example..... 83

4.2.2 Conversion..... 83
4.2.3 Assignment..... 84
4.2.4 Selection..... 85
4.2.5 Printing..... 86
4.2.6 Dimensions..... 87

4.2.6.1 Default Apparent Dimensions..... 87
4.2.6.2 User-Defined Dimension Functions 90

4.3 COMPLETE SEMANTICS OF DECLARATIONS..... 93

4.3.1 Declaration by Example..... 93
4.3.2 Declaration by Size..... 95
4.3.3 Declaration by Default..... 96
4.3.4 Declaration by Components..... 96
4.3.5 Replication and Naming..... 97

4.4 MODE DELETION..... 97

5. ECL LIST STRUCTURE EDITOR..... 102

5.1 GENERAL EDITOR INFORMATION..... 102

5.1.1 Command Interpretation..... 102
5.1.2 Input Modes..... 103
5.1.3 Editor Positioning..... 103
5.1.4 Substitution Operators..... 104

5.2 BASIC COMMANDS..... 106

5.2.1 Printing..... 106

5.2.1.1 Depth Limiting..... 107

TABLE OF CONTENTS

5.2.2 Insertion..... 107
5.2.3 Deletion..... 107
5.2.4 Positioning Commands..... 109
5.2.5 Search Commands..... 113
5.2.6 EXIT..... 115
5.2.7 Testing a Function During Editing..... 115

5.3 Q-REGISTER COMMANDS..... 115

5.3.1 Sharing of List Structure by Q-Register Commands..... 117

5.4 PATTERN MATCHING..... 117

5.4.1 Match Replacement Commands..... 120

5.5 ERROR DISCOVERY..... 124

5.5.1 Correcting Patterns..... 125

5.6 SPECIAL TECHNIQUES..... 125

5.6.1 Compress and Uncompress..... 125
5.6.2 Macro Editor Functions..... 126
5.6.3 Initialization Function..... 127
5.6.4 STATEMENT Command..... 127

5.7 INITEDIT and FLUSHEDIT..... 128

5.8 CHANGING THE NAMES OF EDITOR OPERATIONS..... 128

6. THE ECL COMPILER..... 129

6.1 PROCEDURE REPRESENTATION..... 129

6.2 FREE VARIABLES IN COMPILED PROCEDURES..... 131

6.2.1 Global Variables..... 132
6.2.2 Compile-time Macros..... 134
6.2.3 Other Free Variables..... 136

6.3 INTERPRETER/COMPILER COMPATIBILITY..... 138

6.4 USING THE COMPILER..... 143

6.4.1 A Typical Compilation..... 143
6.4.2 Compiler Input Variables..... 144
6.4.3 Compiler Routines..... 152
6.4.4 Creating Compiler Inputs..... 155

6.4.4.1 The Initial Scan..... 156
6.4.4.2 The Free Variable Scan..... 157
6.4.4.3 Editing the Command List..... 158

TABLE OF CONTENTS

6.4.5 Operating Instructions..... 160

6.4.5.1 Using Scan..... 160
6.4.5.2 Initial Compilation..... 161
6.4.5.3 Tinkering with Compiled Packages 161

7. BUILT-IN ROUTINES..... 163

7.1 GENERAL ROUTINES..... 163

7.2 ARITHMETIC AND TRIGONOMETRIC ROUTINES..... 165

7.3 LOGICAL AND RELATIONAL ROUTINES..... 167

7.4 OPERATOR-DEFINING ROUTINES..... 170

7.5 INPUT/OUTPUT ROUTINES..... 171

7.6 STRING AND CHARACTER HANDLING ROUTINES..... 176

7.7 DEBUGGING ROUTINES..... 177

7.8 ENVIRONMENT ROUTINES..... 180

7.9 PRIMITIVES FOR NON-DETERMINISTIC ALGORITHMS..... 181

7.10 MODE ROUTINES..... 183

7.11 CONTROL ROUTINES..... 184

8. SUPPORT PACKAGES..... 186

8.1 DEBUGGING ROUTINES..... 186

8.2 UNPARSING ROUTINES..... 188

8.3 METERING ROUTINES..... 190

8.4 HASHING ROUTINES..... 193

8.4.1 Using the Hashing Package..... 193
8.4.2 Technicalities..... 196

8.5 EXTENDED BINARY INPUT/OUTPUT..... 199

APPENDICES:

A: EL1 AUGMENTED GRAMMAR..... 200

A.1 Meta-Operators in the Augments..... 201
A.2 Examples of Internal Representation..... 203

TABLE OF CONTENTS

B: BUILT-IN DATA HANDLING ALGORITHMS..... 204

B.1 IMPLEMENTATION INDEPENDENT DATA ALGORITHMS.. 205
B.2 IMPLEMENTATION DEPENDENT DATA ALGORITHMS.... 219
B.3 PRIMITIVES USED BUT NOT MODELED..... 225

C: BUILT-IN MODES..... 227

C.1 COMMONLY USED MODES..... 227
C.2 OTHER MODES..... 227

D: RESERVED IDENTIFIERS AND SPECIAL SYMBOLS..... 229

E: ECL ERROR MESSAGES..... 230

F: MODE COMPATIBLITY FOR CONVERSION..... 233

INDEX..... 234

1. INTRODUCTION

1.1 AN OVERVIEW OF THE ECL PROGRAMMING SYSTEM

The ECL programming system has been designed as a tool for general applications programming. Its emphasis is on providing a powerful linguistic medium and convenient programming environment for the production of software spanning a large range of applications areas. In providing such an environment, three goals were considered primary:

(1) To allow problem-oriented description of algorithm, data, and control over a wide range of applications areas.

(2) To facilitate program construction and debugging.

(3) To facilitate smooth progression between initial program construction and the realization of a well-tuned final product.

ECL consists of a programming language, called EL1, and a system built around this language to realize these goals. The system allows on-line interactive construction, testing, and running of programs. It includes a language-oriented editor, an interpreter, and a fully compatible compiler. Utilities include a debugging package, a source-level frequency profile package, and a source program formatter.

So that it can span a wide range of applications areas, EL1 is an extensible language. Thus it provides a number of facilities for defining extensions so that the programmer can readily shape the language to the problem at hand, and progressively reshape the language as his understanding of the problem and its solution improves. Like the familiar notions of subroutine and macro definition, these extension facilities allow one to abstract significant aspects of a complex algorithm.

In addition to definition mechanisms provided by the language, the ECL programming system provides a number of other handles which the programmer can use to extend and tailor the environment in which he operates. Many of the system's facilities are written in the language and hence are open to modification by the programmer. These include the compiler, one of the editors, and most of the input/output and file system.

Extensibility alone is, however, not sufficient. Its counterpart -- contractibility -- is also required. Once running prototype programs have been produced, it must be possible to subject them to a sequence of contractions -- commitments to subsequent nonvariation -- to obtain a final

system optimal for the project requirements.

The ECL programming and the EL1 programming language have been designed to allow this. Programs can be run either by an interpreter or by a fully compatible compiler. Compiled and interpreted functions can call each other. Compilation can itself be progressively refined. The programmer is free to supply as much declarative information as he wishes (or knows) at a given time and the compiler will do the best it can with the information given.

In summary, the intended application of the ECL programming system is general applications programming. To this end, it has been designed to provide a powerful linguistic medium, a convenient environment for program construction and debugging, and a set of facilities for optimizing contractions to produce a final product.

1.2 PURPOSE OF THIS MANUAL

This manual is an introductory reference. It provides little or no motivation for the material it presents. It is not organized so as to lead the reader gently into ECL as a true primer would. Rather, it assumes that the reader, though perhaps ignorant of ECL, has some familiarity with programming languages and programming systems.

Section 2 describes the EL1 language in some detail, independent of the ECL system. It is an introduction in the sense that it proceeds from simple to more complicated notions, and usually defines terms before they are used. However, it is primarily organized to permit quick reference and easy comparison of EL1 with other languages.

Section 3 describes the interactive use of ECL. It assumes an understanding of section 2 as well as the ability to cope with TECO, the PDP-10 text editor.

Section 4 describes a sophisticated mode-behavior extension facility.

Sections 5 and 6 describe the use of the language-based list structure editor and ECL compiler, respectively.

Section 7 briefly describes each built-in routine of the system.

Section 8 describes a number of EL1 procedures, mostly utility packages of general applicability.

2. INTRODUCTION TO EL1

2.1 FORMS

EL1 programs are composed of basic units called forms. Examples of EL1 forms which have counterparts in most other programming languages are:

(1) constants such as 13 and TRUE,

(2) variables such as x and pressure,

(3) expressions composed of infix and prefix operations such as x+y, i-j*k, and -q1/q2,

(4) selections of components of compound objects such as b[i] and position[3*x],

(5) procedure calls like f(x) and foo(i,j+k,a[n]).

A form in EL1 is a syntactically complete unit, and each form represents a value. Forms may be combined according to the composition rules of the language to obtain larger forms. The remainder of this section describes EL1 by describing the individual forms which make up the language: how they are written and how they will be evaluated.

2.2 CONSTANTS AND THE BASIC DATA TYPES

Although the number of data types in EL1 is virtually unlimited, all are based on a relatively small number of primitive types, including:

(1) booleans: logical truth values;

(2) numbers: integers and reals;

(3) characters;

(4) references: pointers to data objects;

(5) the empty value.

A few other data types, although not technically primitive, are basic to the implementation and are also built-in. These include:

(6) modes: values which describe data types (in this manual, "mode" and "data type" are synonymous);

(7) symbols: which permit efficient representation of symbolic expressions;

(8) strings: arrays of characters;

(9) procedures: values which specify a transformation from a set of input values to a result, possibly with some side-effects.

Constant values for most of these built-in data types have explicit representations in the EL1 language:

(1) Boolean constants are written TRUE and FALSE.

(2) Integer constants are sequences of digits, such as

6 1596 6600

Real constants are digit sequences including exactly one radix point:

2.71828 .01745

Reals may also be written in 'scientific notation':

6.627E23 137E-2

where the suffix E followed by a signed integer N represents multiplication by a factor of ten to the power N: 137E-2 is the same as 1.37 .

(3) Character constants are denoted by a percent sign (%) followed by any character:

%Z %C %= %%

(4) The only reference constant is NIL. It represents a pointer to the empty value.

(5) The empty value is called NOTHING.

(6) The mode constants represent the built-in data types of the language: BOOL, INT, REAL, CHAR, REF, NONE, MODE, SYMBOL, STRING, ROUTINE. One mode constant, ANY, plays a special role in the language (see section 2.7.6) though there are no values with data type ANY.

(7) Symbol constants are sequences of characters enclosed in double quotation marks:

```
"ABC"      "1B16"      "@ ="
```

To include a double quotation mark in a symbol, one precedes it by a percent sign:

```
"%"Who are You?%", said the Caterpillar."
```

The same rule allows inclusion of percent sign:

```
"50%% overlap"
```

(8) String constants have the same form as symbol constants except that single quotation marks are used instead of double ones:

```
´Johann Sebastian Bach´
  ´Rake%´s Progress´
```

(9) Procedure constants are procedure definitions. They are not elementary constants since they are composed of other forms. For example:

```
EXPR(i:INT; INT) (i+1)
```

is a procedure constant which represents the transformation of an integer into its successor. The meaning of the syntax of procedures is explained in section 2.10.

2.3 VARIABLES

Variables give the programmer an abstract notation for the data his program manipulates. In EL1 every variable has a name, a mode, a scope, and a value. Only the value may change.

A variable name is called an identifier. EL1 has two kinds of identifiers:

(1) a sequence of consecutive letters (upper or lower case), digits, and backslashes (\), the first of which is a letter, for example

```
Igor  U235  real\matrix  FUM
```

(2) a sequence of characters from the set

. # $ * + - / < = > ? @ ← ^ & !

Identifiers may be of any length. Upper and lower case letters are not considered identical: cat and CAT are distinct names. The user's identifiers must not conflict with certain reserved words of the language. Appendix D contains a list of these reserved words.

Scope is the most subtle attribute of variables. Roughly speaking, the scope of a variable is the span of time over which it is defined. Some variables have meaning independent of the evaluation of a particular program. These are called top-level, or global variables. Many global variables are pre-defined as part of the language. The set of globals may be expanded or contracted by the user (see section 3.2). Other variables are created and destroyed dynamically by programs as they run. These are called local variables. Their scope lasts only as long as an activation of the form which creates them. Creation of local variables is discussed in section 2.5 and a more precise definition of local scope is given there. It should be emphasized here, however, that scope is no less a distinguishing feature of a variable than its name. Two variables may have the same name and may even be created by the same line of a program, and yet have different scope and different behavior.

A variable's mode describes the class of values the variable can assume. It is defined when the variable is created and remains fixed throughout its lifetime.

What can change, of course, is the variable's value, through a built-in operation called assignment. The format of an assignment is[*]

```
form <- form
```

When this expression is evaluated, the value of the right-hand form replaces that of the left-hand form. For example, suppose temp and count are variables of mode INT. The assignment

```
temp <- 1079
```

---

[*] Left-arrow (←) and the symbol <- are predefined in ECL to have identical meaning.

gives temp the value 1079; temp's value can be passed on to the variable count by

```
count <- temp
```

These assignments are legal because INT variables are being given INT values. In general, unless the modes of the two sides of an assignment agree, the evaluator of the program will signal an error. This kind of error is called a type fault. In certain cases, however, a right-hand value will be converted during assignment to match an expected mode. An INT will be converted to the equivalent REAL value on assignment to a REAL variable; a REAL will be rounded to the nearest INT, if necessary. Other permissible built-in conversions are listed in Appendix F.

The value of an assignment is the value of its left-hand form after the assignment has been completed. Thus the value of

```
count <- 4.9
```

(where count remains an INT) is the INT value 5, not the REAL value 4.9. Moreover, in the (useless but legal) event that an assignment is itself the left-hand form of another assignment, the value of a variable may change twice, for example

```
(count <- 2) <- 50
```

results in a value of 50 for count.

2.4 OPERATOR EXPRESSIONS

EL1 provides the programmer with a set of built-in operators on built-in data types, and with facilities for defining both new data types and new operations. Operator expressions provide a notation for computation which resembles the formula notation of standard algebra and logic.

An operator expression is either

(1) a prefix expression, written

```
identifier  form
```

(2) an infix expression, written

```
form  identifier  form
```

(3) a nofix expression, written

identifier

or (4) a matchfix expression, written

identifier1 form, form, ..., form identifier2

In a prefix expression, the identifier names an operation to be applied to the operand form, producing a value. For example, the identifier NOT is a built-in prefix operator which produces the logical negation of a BOOL value; if b has mode BOOL and value FALSE, then NOT b produces TRUE.

An infix expression is evaluated by applying the operator named by the identifier to the two operand forms, producing a value. An example is +, the built-in infix operator for adding two numbers. If delta is an INT variable with value 4, then 6.3 + delta has the REAL value 10.3.

A nofix expression is evaluated by calling the procedure named by the identifier with no arguments. For example, if f and g have the same procedure value, but f is nofix and g is a simple identifier (i.e. without fixity), then 1+f has the same meaning as 1+g().

A matchfix expression consists of a left-matchfix operator, a list of forms separated by commas, and a right-matchfix operator. Matchfix operators are defined in pairs -- a left operator is allowed syntactically only in combination with its associated right operator. A matchfix operator may not be used as a prefix, infix, or nofix operator and it may not be used as an identifier.

A matchfix expression is evaluated by applying the left operator to the expressions between the matching identifiers. Note that the binding of the left operator, for user defined matchfix pairs, will normally be established before the pair is given the matchfix property. For example, if "<" and ">" form a matchfix pair with "<" bound to the procedure QL (as, in fact, they will in the initial ECL environment), then <X,Y,Z> is equivalent to QL(X,Y,Z).

A particular identifier can be given certain combinations of operator properties simultaneously. Table 2.4-1 gives the allowed combinations. The built-in minus operator (-), for example, can be a prefix (negation) operator or an infix (difference) operator, and can also be used as a simple identifier. The meaning of a particular use is determined from context. In the expression -i-4, the

first appearance of minus is prefix, while the second is infix with arguments -i and 4. In - <- new\minus, minus acts as an ordinary identifier. When a conflict arises because of multiple operator properties, the meaning nearest the top of the left column of Table 2.4-1 is given precedence. Suppose Start is both a prefix and a nofix operator, for example. Then a use of Start will be treated as a nofix use only if no prefix interpretation is possible.

	Simple Identifier	Left MATCHFIX	NOFIX	PREFIX
INFIX	yes	no	yes	yes
PREFIX	yes	no	yes	
NOFIX	no	no		
Left MATCHFIX	no			

Table 2.4-1

The built-in operators fall into several categories. In the brief descriptions which follow, x and y represent numbers (INT or REAL), p and q represent boolean values, and v and w represent values of any mode.

(1) Arithmetic operators. These are written:

x + y	sum of x and y.
x - y	x less y.
- x	negative of x.
x * y	product of x and y.
x / y	x divided by y.

These operators use INT arithmetic and produce an INT result if and only if both operands are INT. Otherwise REAL arithmetic is used to yield a REAL result. The quotient of two INTs is obtained by truncating toward zero. That is, 5/3 becomes 1 and (-3)/2 is -1.

(2) Boolean operators. These are:

p AND q	the logical product or conjunction of p and q: TRUE if and only if both are TRUE.
p OR q	the logical sum or disjunction of p and q: TRUE unless both are FALSE.
NOT p	the logical negation of p: TRUE if and only if p is FALSE.

For AND and OR it is guaranteed that the second operand form will not be evaluated if the value of the first determines the outcome of the expression.

(3) Arithmetic relational operators. All are infix operators. They compare two numbers (INT or REAL) and produce a BOOL result.

x LT y	TRUE exactly when x is strictly less than y.
x LE y	TRUE exactly when x is less than or equal to y.
x GT y	TRUE exactly when x is strictly greater than y.
x GE y	TRUE exactly when x is greater than or equal to y.

If one but not both of the operands is an INT, it is converted to REAL before the comparison.

(4) General relational operators. These two operators test the equality of pairs of values which may be of any mode.

v = w	TRUE if v and w have the same modes and equal values (see section 7.3 for a more exact definition); FALSE otherwise.
v # w	The logical negation of v = w.

(5) Assignment, discussed in section 2.3, is an infix operation which can be applied to operands of any mode.

(6) Conditional execution. -> and +> are infix operators. The conditional is written

```
form -> form
```

or

```
form +> form
```

In a conditional, the left form is called the test or antecedent clause, and the right form is called the consequent clause. The consequent is evaluated exactly when either the test clause evaluates to TRUE and the operator is ->, or the

test clause evaluates to FALSE and the operator is +>. In these cases, the value of the conditional form is the value of the consequent; otherwise the value of the form is NOTHING. The operator -> may thus be read as a causation or implication sign, and P +> V is equivalent to (NOT P) -> V.

(7) "<" and ">" are a built-in matchfix pair with < bound to QL.

(8) Mark. "<<" (called a mark) is both a prefix and an infix operator. Its function is described in section 2.14.

In mathematics, notational conventions usually govern the interpretation of potentially ambiguous formulas. For example, $\sin x^2 + y$ probably means $\sin (x^2) + y$ and not $(\sin x)^2 + y$ or $\sin (x^2 + y)$. In any case, parentheses can be used to resolve ambiguities.

Similarly in EL1, any form may be replaced by the same form in parentheses, so the programmer can group sub-expressions explicitly. To retain the convenience of notational conventions, however, rules of precedence are used when an expression has not been completely broken into sub-expressions. The rules are:

(1) Prefix operators always take precedence over infix operators.

(2) Every infix operator is given a numeric precedence. Operations of higher precedence are evaluated first.

(3) Every infix operator is designated either left-associative or right-associative. When two or more infix operators in a sub-expression have the same precedence and are in conflict (i.e. appear to share operands) the conflict is resolved by a left-to-right scan through the sub-expression: each left-associative operator takes the operand immediately following as its right operand; each right-associative operator takes the rest of the sub-expression as its right operand.

Rule (1) means that, for example, -i-4 is interpreted as (-i)-4. The relational operators take precedence over AND, so by rule (2)

```
x GT y  AND v = w
```

means

```
(x GT y) AND (v = w)
```

Since * takes precedence over + and both take precedence over <-, the form

```
u <- x * y + z * a
```

is parenthesized

```
u <- ((x * y) + (z * a))
```

For precedences of all built-in infix operators, consult section 7.

The built-in operator * is left-associative. Thus by rule (3), a*b*c means (a*b)*c. Multiple assignments, however, associate to the right:

```
p <- q <- r <- FALSE
```

is evaluated as if it were

```
p <- (q <- (r <- FALSE))
```

Although rules (1) through (3) completely specify the association of operands with operators, they do not specify which operand of an operator will be evaluated first. Only for AND and OR have we said that one necessarily precedes the other. For the other built-in operators, the order of operand evaluation is specifically undefined. Different evaluators of the expression (a*b) + (c/d) are free to choose whether to multiply a by b before or after dividing c by d. As will be seen in section 7.4, the programmer can define his own operators, complete with precedence, and right or left associativity. He can likewise choose to explicitly evaluate the operands in a desired order or leave the evaluation implicit and the order consequently unspecified.

Finally, before considering more intricate forms, it is well to note an intrinsic difference in EL1, as in most programming languages, between the values represented by identifiers, like x, and those represented by constants or by expressions like y+1. It makes sense to assign a new value to x, e.g. x<-y+1; but an assignment like y+1<-x, although legal in EL1 and quite harmless, is useless. The value of x is persistent and reusable; a change made to x can have an effect on the later evaluation of the program. We call such values proper objects. We will see shortly that forms other than simple identifiers can represent proper objects. Forms like y+1, TRUE, or 1.414 which are

not useful targets for assignment are called pure values. All the built-in arithmetic, boolean, and relational operators produce pure values.

2.5 COMPOUND FORMS

When a computation must be performed which cannot be expressed using only built-in routines, it is often useful to evaluate a set of forms with control passing from one form to another according to some sequencing rule.

This need is satisfied by the compound form. There are two kinds of compound forms in EL1; the BEGIN block (described in the remainder of this subsection) and the REPEAT block (described in section 2.6).

To begin with an example, suppose one wishes to compute the square root of s, a REAL variable, to within an accuracy epsilon, where epsilon is also REAL. The following compound form computes this root by iterative approximation:

```
BEGIN
  DECL root:REAL BYVAL first;
  REPEAT
    abs(root*root-s) LT epsilon => root;
    root <- (root + s/root)/2.0;
  END;
END
```

This block uses a local variable named root to hold the successive approximations. As an initial approximation root is set to the value of a variable called first. Thereafter, the current value of root is tested for sufficient accuracy. If abs(root*root-s) LT epsilon, then the exit conditional (see section 2.5.2), the first line of the REPEAT body, succeeds, returning root as the value of the REPEAT and hence of the enclosing block. Otherwise the value of root is used to compute a better approximation and the REPEAT block is re-executed.

A BEGIN block consists of a sequence of declarations and statements separated by semicolons and surrounded by the delimiters BEGIN and END, which may be abbreviated [] and (], respectively.

2.5.1 Declarations

Declarations serve to establish any local variables needed within a block. They may be freely intermixed with the other statements of the block. The scope of the variables introduced in a declaration begins with the succeeding declaration or statement.

A declaration is syntactically a group of one or more sub-declarations, separate, but without separating tokens (like comma or semicolon) between them. Each sub-declaration consists of the word DECL followed by a list of identifiers (separated by commas), a colon(:), and a form whose value must be of data type MODE. A sub-declaration may optionally end with an initial value specification. There are two kinds of initial value specification. The first consists of a bind-class indicator from the set {BYVAL, FROM, LIKE, SHARED} followed by a single form. The second kind consists of either SIZE or OF followed by a list of one or more forms, separated by commas. The value of a declaration statement is the value of the last variable that it introduces.

Declarations will be used later in this manual to model other constructs in the language. A precise understanding of their evaluation is therefore very important.

2.5.1.1 Local Variables

A declaration is evaluated by first evaluating the mode-specifications and the initial-value forms (if present), then creating new variables whose name, mode and value are as specified. All names are added to the environment simultaneously, i.e., the variable names created by a component are not visible to other components of the declaration.

We say these variables are local to the compound form in which they are defined because they have meaning only while this block is being evaluated. The range of definition or scope of a local variable, extends from just after its declaration to the end of the block in which it is defined, and includes any forms which may be evaluated during the evaluation of that block. When control leaves a block, any variables defined within it vanish.

Within their scope, however, local variables supersede any "less local" variables with the same names. For example, suppose the block in the example above is embedded in another block in which the variable root has meaning:

```
BEGIN
  DECL chord:STRING BYVAL 'GBDF'
    DECL root:CHAR BYVAL %G;
   ...
  BEGIN
    DECL root:REAL BYVAL first;
    REPEAT
      abs(root*root-s) LT epsilon => root;
      root <- (root + s/root)/2.0;
    END;
     ...
  END
   ...
```

These two uses of root represent quite separate variables. During the square root computation the outer meaning of root is set aside; each use of root in the inner block refers to the REAL root, not the CHAR root. When the evaluation of the square root is finished, the outer meaning of root is restored: the CHAR variable with whatever value it had just before the inner block.

Variables used but not declared in a block are called free variables with respect to that block. In the square root block, for example, first, s, epsilon, abs are free variables. A free variable has the same meaning inside a block as it had just before control entered the block: either it is a local variable in some broader context or else it is a variable of global context, a so-called top-level variable. Top-level variables have meaning independent of the evaluation of any compound form. Most of the built-in routines in EL1, for example, are top-level routine-valued variables. The creation of global variables is described in section 3.2. The set of top-level and local variable definitions in effect at any point during a program's execution is called the environment of that point.

Note that an identifier may represent a free variable and a local within a single block. The declaration

```
DECL x:INT BYVAL x+1
```

uses the free variable x to compute an initial value for the local x being defined.

2.5.1.2 Initialization

A declaration may specify an initial value for local variables using any of the indicators BYVAL, LIKE, or SHARED, or it may contain no initial value specification at all. These four situations are called initialization by value, by coercion, by sharing, and by default, respectively. The indicator FROM, which is synonymous with BYVAL, may also be used.

Initialization by value guarantees that each new variable has a value independent of any proper object which exists just prior to the declaration. A copy of the value will be used to initialize the variable if the initial value form evaluates to a proper object.

Initialization by sharing, on the other hand, results in a sharing of values. For example, if k has mode BOOL and if i and j are created by

```
DECL i,j:BOOL SHARED k
```

then i, j, and k will share the same boolean value throughout the scope of i and j. Any change to one (for example, j <- NOT j) changes both of the others also. Initialization by sharing with a proper object makes an identifier a synonym for that object.

Initialization by coercion results in binding either by value or by sharing depending upon the result of the evaluation of the form following LIKE. This means

```
DECL x:REAL LIKE f(z)
```

will cause x to be initialized by value to f(z) whenever f(z) is a pure value, and to be shared when f(z) evaluates to a proper object of mode REAL. If f(z) is not REAL, then conversion to REAL takes place, producing a pure value and thus an initialization by value.

If no initial specification at all appears in the declaration, the declared variables are given default values corresponding to their modes. These values for the principal built-in data types are given in the following table.

Mode	Default Value
BOOL	FALSE
INT	0
REAL	0.0
CHAR	(ASCII NULL)
REF	NIL
NONE	NOTHING
MODE	NIL
SYMBOL	NIL
STRING	''
ROUTINE	NIL

Table 2.5-1 Default Values of Built-in Modes

Default values for compound data types (see section 2.7) are constructed from default values for their component modes.

Generally speaking, when an initial value is supplied for a local variable, this value's mode is expected to match the mode specified in the declaration. As with assignment, however, the evaluator will in certain cases automatically convert an initial value to match an expected mode (see Appendix F). Note that built-in conversion always produces a pure value, so that declaration by coercion acts exactly like declaration by value when the initial value must be converted. For example

```
DECL x:REAL;
DECL y:INT LIKE x
```

introduces two variables with two independent values, each zero.

2.5.2 Statements

A statement is either a form or an exit-conditional, written

```
form <arrow> form
```

where <arrow> is either => or #>. As with ordinary conditionals (section 2.4) the left form is called the test clause and the right form is called the consequent.

Ordinarily, a block is evaluated by evaluating its declarations and statements in sequence. When an exit-conditional statement is reached, its test clause is evaluated to a BOOL. If the result is TRUE and <arrow> is =>, or the result is FALSE and <arrow> is #>, then the consequent is evaluated, evaluation of the block is complete, and the value of the consequent becomes the value of the block. If the test clause and <arrow> do not agree as described, then the consequent is ignored and control proceeds to the next statement.

When the evaluation of a block is complete, its value is either the consequent of the terminating exit-conditional or the value of the last statement, if that is a simple form. If the last statement is a conditional whose test fails, the value of the block is NOTHING.

Note that it is reasonable for a block, whose value is a proper object with scope global to the block, to be the left-hand side of an assignment. For example,

```
[)p => a; b(] <- c+1
```

selects either a or b to receive the new value on the basis of the value of p.

2.6 ITERATION

Often in an algorithmic language it is necessary to evaluate a form repeatedly, possibly with an index variable changing for each repetition. This repetition can be indefinite or can be terminated by the occurrence of some condition such as an index exceeding a limit value or a boolean condition evaluating to TRUE. The iteration form is provided for this purpose.

The general format of the statement is the following:

```
FOR identifier <bndtyp> form BY form TO form
        REPEAT <blockbody> <sem> END
```

where <bndtyp> is FROM (or equivalently, BYVAL), LIKE, or SHARED, <blockbody> is a sequence of statements and declarations separated by semi-colons, and <sem> is an optional final semi-colon. Any of the following pairs may be omitted:

```
FOR identifier
<bndtyp> form
BY form
TO form
```

Only "REPEAT <blockbody> <sem> END" must always be present.

For example, the following fragment computes the sum of the positive integers less than or equal to n.

```
s <- 0;
FOR i FROM 1 BY 1 TO n REPEAT s <- s+i END;
```

In an iteration, the identifier given after the FOR is the index variable, and may be used explicitly in the body of the iteration. The index variable is a new variable of mode INT, which is local to the iteration. Its relation to other variables is determined by the <bndtyp> in the same way as for declarations (see section 2.5.1.2). If the FOR identifier part of the iteration is omitted, there is no named index variable, although the indexing may still be performed.

The forms which follow <bndtyp>, BY, and TO may be any forms which evaluate to integer values (not necessarily positive). They are the initial value of the index, the step size, and the limit, respectively. Each of the forms provided is evaluated only once. The step and limit values are also copied, so they cannot change during the iteration.

If the <bndtyp> form is omitted, a default initial value of 1 is used with <bndtyp> LIKE.

If the BY form is omitted, a default step size of 1 is used.

If the TO form is omitted, a default limit of infinity is used. In essence this means that the check (index < limit) always succeeds. The programmer is warned that in this case he should have another means of terminating the iteration since otherwise he may encounter overflow errors (the real world version of non-termination).

If one or more of FOR identifier, <bndtyp> form, BY form, or TO form are explicitly included in the iteration form, the following actions will be performed: the index will be initialized, it will be incremented by the step size after each iteration, and limit checking will be performed, using the default values given above for those values not explicitly provided. If none of these are explicitly included these actions will not be performed. The limit check is as follows: the iteration is terminated when

```
(index - limit)*step\size  GT  0
```

If this occurs initially, then the iteration is not executed. For example, in

```
FROM 10 TO -9 REPEAT form END
```

the form following the REPEAT will not be evaluated.

The following command

```
FOR i FROM 10 BY -1 TO -9 REPEAT x <- i END
```

will evaluate the form following REPEAT 20 times leaving x with the value -9 after the iteration terminates.

It may also be desirable to terminate an iteration upon the occurrence of some condition before the index has reached its limit. This facility is provided by means of the => and #> exit-conditionals. Each iteration allowed by the index checking will cause the statements of the REPEAT body to be executed in turn. If an exit-conditional P => V (or P #> V) is encountered whose antecedent P evaluates to TRUE (respectively, FALSE) then the iteration is immediately terminated with the evaluated consequent V as its result. If the value of the antecedent does not cause termination, the consequent is ignored and execution of the body continues. An iteration that is terminated by index checking will return the value of the last executed statement of its body as result and will return NOTHING if the body has not been executed at all.

2.7 MODE-VALUED FORMS

Those aspects of EL1 discussed thus far do not differ significantly from Algol 60 or, indeed, any other algorithmic language. Notation and syntax have been somewhat idiosyncratic but the underlying semantics and facilities provided have been quite conventional. However, having explained our notation for familiar concepts, we now have sufficient foundation to present the more innovative aspects of EL1. One of these is the mode-valued form.

2.7.1 Mode-Valued Constants and Identifiers

As mentioned in section 2.2, the term mode is used in EL1 to designate a certain class of objects: objects corresponding to the intuitive notion of data type. Seven modes are primitive: those denoted by the mode-valued constants INT, REAL, BOOL, CHAR, NONE, REF, and ANY.

Just as variables can be declared of type INT and thereby restricted to INT values, variables can be declared

of type MODE and restricted to MODE values. For example, consider

```
DECL m1, m2, complex, rational:MODE
```

The variables m1, m2, complex, and rational are mode-valued variables. Hence, it is legal to assign

```
m1 <- BOOL;
m2 <- m1
```

The variables m1 and m2 now have the same value as the constant BOOL. Hence, m2=BOOL is TRUE while m1=CHAR is FALSE.

Of themselves, mode constants and mode-valued variables are uninteresting. However, we next consider operators which take modes as operands and produce new modes. Six such mode-producing operators are pre-defined: SEQ, STRUCT, PTR, VECTOR, ONEOF, and PROC. The first five are discussed in the rest of this subsection. PROC will be described in section 2.16. From these, other mode-valued forms can be synthesized using conditional expressions, functional composition, and recursion.

2.7.2 VECTOR and SEQ

The operators VECTOR and SEQ are best introduced by means of an example. Consider the fragment

```
DECL triple:MODE
```

The variable triple has been declared to be an object whose value is a mode.

```
triple <- VECTOR(3, INT)
```

The variable triple now has a value--the mode integer-arrays-of-length-three. Hence, it is possible to write

```
DECL x, y:triple
```

The variables x and y are of mode triple.

As a consequence of the above EL1 declaration, x and y are objects having several of the properties of an Algol 60 array. They can be subscripted, e.g. x[2], y[i], y[f(x[i])] are well-formed. The result of the subscripting is a proper object of mode INT which may be changed by assignment. For example,

```
x[1] <- 10; x[2] <- 20;  x[3] <- 30;

y[1] <- x[3] - x[2]
```

The variable y can serve as operand for various operators--in particular, for the assignment operator. For example,

```
y <- x
```

is legal and assigns to y[1], y[2], and y[3] the values of x[1], x[2], and x[3], respectively. In general, assignment involves copying of values, not sharing. For example, if we next assign

```
x[2] <- 79
```

this does not change y[2] which remains 20.

We say that VECTOR(3, INT) is a form of mode MODE and that the class of this mode is row. An object (such as y) whose mode is of class row[*] has a number of properties.

(1) It is composed of an ordered collection of like mode (e.g., x consists of 3 components each having mode INT).

(2) Any one of these components may be selected by subscripting (e.g. x[1] is the first of the 3 INTs).

(3) The number of components may be determined by applying the function LENGTH to the object (e.g. LENGTH(x) is 3).

(4) The components of the row are of uniform dimensions. (The meaning of this property will become clearer below.)

It is frequently useful to define a mode of class row in which the number of components is not fixed at the time the mode is created. For example, the built-in mode STRING might have been defined by:

```
DECL STRING: MODE;
```

---

[*] It will be useful to abbreviate this notation and say, for example, "y is a row," meaning that y is an object whose mode is of class row.

```
STRING <- SEQ(CHAR)
```

This first declares STRING to be a variable of data type MODE and then assigns to STRING the mode intuitively described by array-of-any-number-of-characters. Since the number of components in a STRING is not predetermined, STRING is said to be length unresolved. This is not to say that objects of mode STRING have variable length, but rather that the mode STRING leaves the length open. A specific object of mode STRING will have some fixed length, but different STRINGs may have different lengths.

When a variable is created with mode STRING, it is necessary to resolve the mode (that is, determine how much storage space is to be allocated) either by specifying the length explicitly or by specifying an initial value, and hence a length, for the variable. For example,

```
DECL s:STRING SIZE n
```

declares s to be a STRING whose length is the value of n. Now s behaves like any other object whose mode is of class row: it has a fixed number of components which may be obtained by LENGTH(s); it may be subscripted; it may be changed by assignment.

The declaration

```
DECL new\s:STRING BYVAL s
```

creates a new string whose value is a copy of s. So LENGTH(s) = LENGTH(new\s) throughout the common scope of s and new\s.

To summarize, the operator VECTOR takes two arguments, an integer i and a mode m, and then produces the mode: row of length i each of whose components is of mode m. The operator SEQ takes one argument, a mode m, and then produces the mode: length unresolved sequence of components each of mode m. Both VECTOR and SEQ produce modes of class row.

Note that VECTOR(i, m) and SEQ(m) are always different modes. If r1 has mode SEQ(INT), r1 may not be directly assigned a value of mode VECTOR(10, INT) even if LENGTH(r1) is 10. Indeed they are stored differently in memory. An object of mode SEQ(m) requires an extra word of storage to indicate the length of that instance. Thus, for instance, a SEQ(BOOL) of length 5 requires one word plus 5 bits whereas an object of mode VECTOR(5, BOOL) takes exactly 5 bits.

In general, the arguments of SEQ and VECTOR may be arbitrary forms, provided that they evaluate to objects of the appropriate type. For example, consider

```
some\row <-
  VECTOR([) i GT j => i; 10 * j (],
         [) p(x) => INT; q(x) => triple; CHAR (]);
```

It should be noted that SEQ, VECTOR, and the other mode-valued operators, are treated as procedures: the arguments are evaluated, the body is executed, and a result whose data type is MODE is delivered. One consequence of this is that the output of SEQ or VECTOR can be used as the argument to a second evaluation of SEQ or VECTOR, e.g.

```
bool\matrix <- SEQ(SEQ(BOOL))
```

The mode bool\matrix corresponds to the notion of a two-dimensional array of booleans. Since length has been specified for neither evaluation of SEQ, bool\matrix is length unresolved with two unresolved dimensions. To generate a particular instance of mode bool\matrix (see section 2.9), we specify both dimensions, as in

```
DECL b:bool\matrix SIZE 5, 10
```

By virtue of the above declaration, b has the following properties:

(1) b is a row of 5 objects, each object being a row of 10 BOOLs.

(2) LENGTH(b) is 5.

(3) b can be subscripted, e.g., b[3], the result being a row of 10 BOOLs.

(4) LENGTH(b[i]) is 10 for i between 1 and 5.

(5) b[i] may be subscripted, e.g., b[i,j], the result being a BOOL

Repeated subscripting is written as x[j1, j2, ..., jn].

2.7.3 STRUCT

A row is subject to the restriction that all its components have the same mode and identical dimensions. A second class of modes, struct, allows composite objects whose components do not necessarily have the same mode. The built-in operator STRUCT takes as arguments a list of pairs of the form id:m where id is the name of the component and m is its mode. STRUCT delivers the mode which describes objects so constructed.

For example, the following definition might be used to represent a household fuse

```
fuse  <- STRUCT(amps:INT,
                manufacturer:VECTOR(10, CHAR),
                blown\flag:BOOL)
```

An object of mode fuse consists of three components:

(1) an INT
(2) a row of 10 CHARs
(3) a BOOL.

Since rows are homogeneous objects, it is appropriate to select their components by numerical subscripts (e.g., x[4]). However, structs are typically inhomogeneous and it is useful to refer to their components by symbolic names. The above definition specifies both the modes of the components of a fuse and the names of these components. That is,

(1) the INT is named "amps",
(2) the row of 10 CHARs is named "manufacturer",
(3) the BOOL is named "blown\flag".

Fuse having been defined, it can later be used in declaring variables, e.g.

```
DECL kitchen\fuse, basement\fuse:fuse
```

The variable kitchen\fuse is a fuse; hence, it has three components, one of which is an INT named "amps". A component may be selected by name qualification which is denoted by the object name, followed by a period, followed by the name of the component. For example,

```
kitchen\fuse.amps
```

Since this is an INT, it may be given an INT value

```
kitchen\fuse.amps <- 15
```

or used as an operand of an arithmetic expression

```
basement\fuse.amps <- kitchen\fuse.amps + 5
```

We have discussed two methods of selecting a component of a compound object: subscripting (e.g., x[i]) and name qualification (e.g., kitchen\fuse.amps). These may be intermixed. For example, kitchen\fuse.manufacturer is of mode VECTOR(10, CHAR); hence, it may be subscripted, e.g., kitchen\fuse.manufacturer[i], yielding a CHAR. If we define

```
fuse\box <- VECTOR(4, fuse)
  .
   .
    .
DECL central\control:fuse\box
```

then we may write

```
central\control[2].blown\flag
```

obtaining a BOOL, or

```
central\control[i].manufacturer[1]
```

obtaining a CHAR. In general, selection proceeds from left to right, obtaining successively lower-level components.

It is occasionally useful to compute which component of a struct is to be selected, as in the case of rows. This is done by subscripting. For example, consider

```
complex <- STRUCT(re:INT, im:INT)
  .
   .
    .
DECL z:complex
```

The form z[1] has precisely the same meaning as z.re and z[2] the same as z.im; z[i] is one of these two depending on the value of i.

In the discussion of rows, we introduced the notion of length unresolved modes resulting from forms such as SEQ(INT) which defer binding of the number of components. Because structs may have components whose types are such modes, it is possible to have length unresolved modes of class struct. For example,

```
shipment <- STRUCT(item:STRING, quantity:INT)
```

Since STRING is length unresolved, so is shipment. If a declared variable j6 of mode shipement is initialized by default, as in

```
DECL j6:shipment
```

then LENGTH(j6.item) is 0 and will remain so for the lifetime of the variable. To generate less vacuous shipment variables with default initialization we designate the SIZE, as in

```
DECL j6:shipment SIZE 16
```

which specifies that j6.item has length 16 (see section 2.9.2 for more details on SIZE).

2.7.4 REF

In section 2.2, we mentioned the mode constant REF but deferred explanation; we now remedy this omission. REF denotes a primitive mode which can be intuitively described as the set of objects which can point to arbitrary objects. Consider, for example,

```
DECL p1, p2: REF
```

The variables p1 and p2 can point to (reference) objects of any mode. Further, if p1 points to an object,

```
p2 <- p1
```

copies the value of p1 (essentially an address) into p2 so that p1 and p2 both point to the same object.

The question arises: how does one get p1 to point to an object in the first place? The form

```
p1 <- x
```

where x is some object, say an INT, does not work, for it is interpreted as: copy the value of x (an INT) into p1 (a REF) which does not achieve the desired result[*]. In general, there is no way to take a declared object x and obtain a pointer to it.

It is, however, possible to create a new object which is pointed to by a REF. For example, consider

```
p1 <- ALLOC(bool\matrix SIZE 5, 10)
```

The right-hand side of the assignment creates a new object of mode bool\matrix (dimension 5 by 10), allocates and returns a pointer to the object. The new object resides in what is called the heap. In fact the destination object of any pointer resides in the heap, and the value of such an object remains accessible so long as some pointer-valued variable exists, pointing to it. The above assignment operation copies this pointer into p1. Hence, p1 points to the bool\matrix. This bool\matrix differs from all objects

---

[*] In fact, mode checking performed on assignment will detect this as an illegal operation, for a REF cannot contain an INT value.

discussed thus far: it was created by an ALLOCation, not a DECLaration, and therefore has no name. It is designated only by means of a pointer. If p1 is assigned to p2, as in

```
p2 <- p1
```

then both p1 and p2 point to the same bool\matrix.

The link between a pointer P and an object Q to which it points is provided by a primitive function VAL, i.e., VAL(P) $\equiv$ Q. For example,

```
VAL(p1)
```

is the bool\matrix allocated earlier. In particular,

```
VAL(p1)[5]
```

is the 5th component of the bool\matrix, a row of 10 BOOLS, and

```
VAL(p1)[5, 3]
```

is a BOOL. Hence,

```
VAL(p1)[5, 3] <- TRUE
```

is a legal form which sets the (5,3) element of the matrix to TRUE.

Since p1 is a REF, it is unrestricted in the mode of objects it can address, and it is legal to assign

```
p1 <- ALLOC(bool\matrix SIZE 5, 10);

p2 <- p1;

p1 <- ALLOC(INT)
```

This leaves p2 pointing to the bool\matrix and p1 pointing to a single INT of value 0.

2.7.5 PTR

There are other pointers which are more restricted than REFs in the mode of objects they can reference. We next consider this type of pointer.

The built-in operator PTR takes a mode m as argument and produces the mode: the set of pointers restricted to address objects of type m. For example, consider

```
int\row <- SEQ(INT);
int\row\ptr <- PTR(int\row);
  .
   .
    .
DECL ip1, ip2:int\row\ptr
```

The variable ip1 is of class pointer (abbreviated "ptr") but its specific mode is int\row\ptr. It can point only to an int\row. Hence,

```
ip1 <- ALLOC(int\row SIZE n)
```

is legal, but

```
ip1 <- ALLOC(fuse\box)
```

is not. The right hand side of the latter form returns a pointer to a fuse\box; assignment of this to an int\row\ptr is a type fault.

As with REF, it is possible for two objects of mode int\row\ptr to refer to the same object, e.g. the assignment ip2 <- ip1 leaves ip1 and ip2 equal and VAL(ip1) identical to VAL(ip2).

We have thus far omitted discussion of pragmatics; however, a pragmatic note is unavoidable at this point lest it appear that pointer modes of restricted referent are an arbitrary whimsey. From considerations based only on semantic grounds, the charge is well-founded. The variable ip1 can be used for no purpose for which p1 (having mode REF) could not be used; p1 can point to any object to which ip1 can point, the assignment p1 <- ip1 being legal. Modes such as int\row\ptr are introduced for two reasons,

(1) In our implementation it is possible to use less storage for an int\row\ptr than for a REF since the REF will carry a type code as well as an address.

(2) Possibly more important is that tightly bound modes such as int\row\ptr allow more efficient (i.e., complete) compilation. For example, the compiler when confronted with the form

```
VAL(ip1)[k]
```

can determine that it is well-formed, that it involves subscripting a row of INTs, and that it yields an INT. Code generation is straightforward. The analogous form involving a REF

```
VAL(p1)[k]
```

could have any number of possible selection operations and result types, depending on what sort of object p1 references at the time the form is evaluated. Code compiled for this must reflect the uncertainty.

It is frequently useful to deal with pointers which can reference objects of more than one mode but which are not totally unrestricted. For this purpose, the operator PTR can be given more than one argument and when so invoked it produces a mode defining pointers which may reference objects of any of the specified types. The mode of such a pointer is said to be of sub-class _united pointer_. For example,

```
int\or\bool\ptr <- PTR(INT, BOOL)
```

Then if we declare

```
DECL q:int\or\bool\ptr;
```

q can point to INTs or BOOLs but to nothing else.

2.7.6 Generic Modes

For those cases in which a variable is to be bound to a value whose mode is not certain until the program is run, or in which automatic data type conversion is undesirable, EL1 provides a special class of modes, called _generic_ modes. Generic modes may only appear as the declared modes of local variables and formal parameters or as the result mode of an explicit procedure (see section 2.10 for a description of procedures). Each is actually a _set_ of modes representing the alternative mode possibilities for the variable or result. Generic modes are created by the built-in function ONEOF; for example,

```
scalar <- ONEOF(INT, REAL, complex)
```

creates a mode which matches actual values of type INT, REAL, or complex (defined in section 2.7.3). No automatic conversion takes place when an expected mode is of class generic. The actual value must agree in mode with one alternative of the generic mode or the evaluator will signal an error. For instance, if we create the following mode

```
flag <- ONEOF(INT, BOOL)
```

then the declaration

```
DECL f:flag LIKE 5.1
```

would produce an error.

Once an alternative has been chosen during a variable declaration or on procedure entry, the mode of the variable is fixed for its lifetime. That is, its mode is uncertain only before the variable is bound, not afterwards. Thus, there are, in fact, no values of generic mode. Nor may generic modes appear as component modes in the forms STRUCT, VECTOR, SEQ, PTR, or ONEOF.

There is a mode constant of class generic called ANY. When the expected mode is ANY, values of any mode whatever will be passed without conversion or type fault. Of course, a variable so declared is subject to the same restrictions that apply to other generic modes: after being given an initial value, the variable's mode is fixed.

As mentioned above a generic mode is like a set of modes. In comparing two modes m1 and m2, either of which may be generic, we often need to check for set inclusion of one by the other. El1 has a built-in routine called COVERS for this purpose. COVERS accepts two MODE arguments and produces a BOOL result. COVERS(m1, m2) is TRUE exactly when (1) m1 = m2, or (2) m1 is generic and m2 is among the alternatives of m1, or (3) both modes are generic and every alternative of m2 is an alternative of m1 as well. So COVERS(INT, INT), COVERS(ONEOF(REAL, m), REAL), and COVERS(ONEOF(u, v, w), ONEOF(w, u)) are all TRUE. But COVERS(CHAR, ONEOF(CHAR, STRING)) and COVERS(ONEOF(INT, scalar), ONEOF(INT, scalar, REAL)) are FALSE. ANY covers any mode whatever.

2.7.7 Summary

Section 2.7 has discussed five groups of mode-valued forms: constants, rows, structs, pointers, and generics. Table 2.7-1 summarizes this material.

Class	Sub-Class	Example
constants	--	INT, REF, ANY
row	resolved	VECTOR(5, CHAR)
	unresolved	SEQ(INT)
struct	resolved	STRUCT(a:INT, b:VECTOR(5, CHAR))
	unresolved	STRUCT(a:INT, b:SEQ(CHAR))
pointer	simple	PTR(SEQ(INT))
	united	PTR(SEQ(INT), BOOL)
generic	restricted	ONEOF(INT, REAL)
	universal	ANY

Table 2.7-1

Having such a diverse array of mode-forms leads to the issue of whether any flexibility is permitted in changing from one to another. Conversion is attempted whenever the mode of the object in hand is not covered by the mode desired. If conversion is possible, a new value is created with the desired mode. By design only a few very standard conversions are built into EL1; they include INT to REAL, REAL to INT, and conversions among pointer modes. A full list is given in Appendix F.

2.8 SELECTION

The operation of selection has been discussed as a peripheral issue in sections 2.7.2-6. In this section, we summarize this discussion and treat one additional point.

The operators SEQ (or VECTOR) and STRUCT produce modes of class row and struct, respectively--objects whose modes are compound and consist of components which may be designated by selection operations. There are two such operations: qualified naming and subscripting. The former is applicable only to structs. It is denoted by writing the name of the object, followed by a period, followed by the

name of the component (e.g., z.re), the name of the component having been specified in the mode definition. Subscripting is applicable to either rows or structs. It is denoted by writing the name of the object, followed by one or more forms separated by commas and enclosed in square brackets (e.g. x[i+j, k]). Each form inside the brackets is expected to yield ONEOF(INT, SYMBOL) to indicate the indexed components. Thus z.re and z["re"] yield the same result, with bracket selection evaluating its argument. The indices are used from left to right; each selects a component of the previous object, starting with the initial object.

The object on which selection is performed is not restricted to be an identifier: any form whose value is a row or struct may be employed. For example, it can be a prior selection (e.g., a[i].re), in which case the selection is carried out from left to right. It can be a function application which delivers a row or struct (e.g., f(x)[i]). In particular, it can be the application of VAL to a pointer; e.g., if p points to a bool\matrix, then

```
VAL(p)[i][j]   (or equivalently VAL(p)[i, j])
```

is a legal form having a BOOL value.

In connection with the last example, one additional point bears mentioning. Since such forms occur frequently, it is desirable to abbreviate their representation. An abbreviation is immediate if it is observed that the appearance of "VAL" is redundant. Given that p is a pointer, it is clear that p cannot itself be the object of a selection, for it has no components. Hence,

```
VAL(p)[i][j]
```

can be unambiguously abbreviated

```
p[i][j]
```

The latter is defined to be identical to the former. This scheme of abbreviation is, of course, completely general. For example, if f(x) returns a pointer to a row of structs each of which has a component named "tx" which is a pointer to a bool\matrix, then

```
f(x)[n].tx[i][j]
```

may be written with the same meaning as the explicit form

```
VAL(VAL(f(x))[n].tx)[i][j]
```

Note however, that at most one implicit use of VAL is permitted for each selection.

In the case of multiple or repeated selection, two formats are equivalent and equally permissible: x[i,j] has the same meaning as x[i][j]. Of course these formats may be combined as in x[i,j][k][l,m,n].

2.9 DATA GENERATION

Two forms are provided in EL1 for the purpose of creating and initializing objects of any mode. This process is called data generation. The two forms, called CONST and ALLOC, differ only in that the first creates a new instance of some data class, while the second does the same thing but returns a pointer to the new instance. Both CONST and ALLOC will be defined in terms of EL1 declarations in section 2.9.5. ALLOC's role as pointer-generator was mentioned in section 2.7.4 and 2.7.5. Now we will see in more detail how data may be created and copied.

2.9.1 Default Generation

Given a mode m, we can construct an object of mode m with the default initialization corresponding to that mode by writing

```
CONST(m)
```

or

```
ALLOC(m)
```

The first form evaluates to the object itself; the second returns a pointer to the new object. One point concerning the modes of these two expressions must be stressed: CONST(m) has mode m, while ALLOC(m) or any ALLOC form evaluates to the mode REF. This particular REF will be compatible with (i.e. convertible to) the mode PTR(m). The programmer should, however, be aware that a conversion step is required.

Examples of default generation are CONST(complex) which would create a complex number with real and imaginary parts equal to zero and ALLOC(INT) which would allocate space for an integer, initialize it to zero, then return a pointer to it. But suppose we wrote

```
CONST(SEQ(SEQ(BOOL)))
```

Since the default initial value for sequences is an empty sequence, this expression produces a rather vacuous object: the empty sequence of empty sequences of BOOLs.

2.9.2 Generation by SIZE

To generate data structures of length-unresolved mode without having each component with a length-unresolved mode assumed empty, we use a SIZE designator in either type of generator. For example

```
CONST(SEQ(SEQ(BOOL)) SIZE 2, 6)
```

creates a 2 by 6 boolean matrix, each of whose elements has the default initial value FALSE. If s is a struct mode defined by

```
s <- STRUCT(a:STRING,
            b:INT,
            c:SEQ(SEQ(INT)))
```

then ALLOC(s SIZE 4, 14, 2) constructs an object of mode s whose first component is a four character string and whose third component is a 14 by 2 INT matrix, all initialized by default.

When the SIZE designator is used, the specific dimensions provided correspond to length uncertainties inherent in the given mode. The sizes are given in the order in which the unresolved lengths occur in the definition of the mode, read left to right, top to bottom. In the previous example, there must be three dimension specifications, one for the first component and two for the third. The first size applies to the STRING, the second to the outer SEQ, the last to the inner SEQ.

To take a still more detailed example:

```
introw <- SEQ(INT);
matrix <- SEQ(SEQ(BOOL));
matrixrow <- SEQ(matrix);
comp <- STRUCT(a:introw, b:matrixrow, c:STRING)
 .
  .
   .
DECL x:comp BYVAL CONST(comp SIZE 10, 3, 20, 25, 6)
```

This results in the following:

(1) LENGTH(x.a) is 10

(2) LENGTH(x.b) is 3

(3) x.b[i] is a matrix, for i=1, 2, 3

(4) LENGTH(x.b[i]) is 20, for i=1, 2, 3

(5) LENGTH(x.b[i,j]) is 25, for appropriate i and j

(6) LENGTH(x.c) is 6

2.9.3 Generation from Components

Objects whose modes are of class row or struct may be generated from a list of the intended components. In this case, unspecified dimensions and all initial values will be inferred from the list given. The OF designator is used for this type of generation, as in

```
CONST(SEQ(INT) OF 4, i * j, -7)
```

This form produces a result equivalent to that of the following block:

```
BEGIN
  DECL ir:SEQ(INT) SIZE 3;
  ir[1] <- 4;
  ir[2] <- i * j;
  ir[3] <- -7;
  ir
END
```

A pointer to an object of mode s (defined in section 2.9.2) could be generated by

```
ALLOC(s OF 'AB',
          4,
          CONST(SEQ(SEQ(INT)) OF
                CONST(SEQ(INT) OF 1, 2),
                CONST(SEQ(INT) OF 3, 4)))
```

Notice that any form may appear in the list of components.

2.9.4 Generation by Example

Finally, an object may be generated using an existing object as an example. There are four different designators for generation by example, BYVAL, FROM, SHARED, and LIKE. The designator BYVAL (or its synonym FROM), used with either ALLOC or CONST specifies that the value of the form which

follows is to be copied, with automatic conversion to the target mode if necessary and permissible (see Appendix F for compatible mode pairs). For example,

```
pv <- ALLOC(SEQ(REAL) BYVAL vector2)
```

creates (and points to) a copy of vector2. That is, VAL(pv) = vector2 is TRUE but assignments like pv[i] <- 5.0 have no effect on vector2. The form

```
CONST(PTR(BOOL) BYVAL ALLOC(BOOL BYVAL TRUE))
```

produces a PTR(BOOL) since this mode is compatible with a REF whose VAL is a BOOL.

The designator or bind-class SHARED indicates that the value of the new object being generated is to be shared with the value of the example given. No conversion is permitted, that is, the mode of the example must be covered by the target mode. (See section 2.7.7 for a discussion of mode covering.) Thus

```
pv <- ALLOC(SEQ(REAL) SHARED vector2)
```

would create a new pointer pv which references the same value associated with vector2. Changes to either variable affect the other. Note, however, that a pointer can only share objects residing in the heap. It is not possible to obtain a pointer to a declared object (an object on the stack).

The third bind-class, LIKE, is a kind of hybrid. It will induce sharing if sharing is possible (that is, if the modes agree and the example is a proper object) and copying (as if the bind-class had been BYVAL) if not. The user should be aware that a degree of indeterminism is introduced into the program if LIKE is used. However, in the cases when it really doesn't matter, LIKE gives the compiler certain desirable flexibility.

2.9.5 A Model for Generation

Each type of generation we have described has a precise definition in terms of an EL1 declaration. Let us consider a generation of the form

```
CONST(M <designator> <initial value>)
```

where the <designator> may be "SIZE", "OF", "BYVAL", etc., and <initial value> specifies the appropriate set of initializing values for the corresponding <designator>. The precisely equivalent declaration has the form

```
DECL result:m <designator> <initial value>
```

Had the generator been ALLOC, the same translation is made, only the result of the declaration is copied into the heap (see section 2.7.4 for a definition of heap), and a REF is created pointing to the object in the heap.

2.10 PROCEDURES

A procedure is an EL1 form whose value is the text of a program. This program, called the procedure body, is executed whenever the procedure is applied to a set of input values. The application of a procedure is often called a procedure call , and the values supplied as inputs to the procedure are called arguments.

Here is a typical procedure call:

```
maximum(k, limit)
```

A procedure call consists of a form which designates a procedure value, followed by a list of forms which designate the arguments. The arguments are separated by commas and surrounded by parentheses. A procedure call is a form and it has a value, namely the value produced when the procedure body is evaluated.

Since each element of a procedure call may be an arbitrary form, elaborate procedure calls may be written:

```
[)i GT j => transform; revert(] (p(i))
```

More commonly, however, the procedure is specified by a simple procedure name, an identifier whose value is a procedure. Such identifiers may have mode ROUTINE, a built-in mode that describes the class of all EL1 procedure values, or they may have user-defined procedure modes. Creation and use of user-defined procedure modes are discussed in section 2.16.

Several ROUTINE-valued identifiers are built-in to EL1, and some of these have been mentioned in earlier sections (LENGTH, VAL). Others are listed in section 7. The programmer may extend the set of available procedures with an EL1 form called an explicit procedure. For example,

```
rem <-
  EXPR(k:INT BYVAL, j:INT BYVAL; INT) (k - (k/j) * j)
```

defines a new procedure, called rem, which produces, when called with two integer arguments, the remainder of the first after division by the second.

The explicit procedure is technically a constant of mode ROUTINE. EL1 programs are collections of these constants. The components of the explicit procedure are, from left to right:

(1) The special delimiter EXPR.

(2) A list of formal parameter descriptors, each consisting of an identifier, a form which must reduce to a mode, and a bind-class indicator, and each separated by a comma.

(3) The result type, a form which must evaluate to a mode. The procedure result must be compatible with this mode and will be returned via the bindclass LIKE. A result type of NONE may be omitted along with the preceding semi-colon.

(4) The procedure body, the form whose value becomes the value of a call of the procedure.

The formal parameter names listed in the procedure heading normally also appear in the body. Their role is to represent the actual arguments supplied as inputs to the procedure. When a procedure such as rem is called, e.g.,

```
rem(13, 4)
```

the arguments are put in correspondence with the formal parameters in left to right order before the procedure body is executed. The effect is as if the programmer had written the following compound form instead of the above procedure call:

```
BEGIN
  DECL k:INT BYVAL 13
    DECL j:INT BYVAL 4;
  DECL result:INT LIKE (k - (k/j) * j);
  result
END
```

When rem is called, the arguments are evaluated and checked for compatibility with the modes of the corresponding formal parameters. When necessary, an attempt will be made to convert an argument to agree with the formal parameter mode. REAL arguments to rem, for example, are rounded to the nearest integers. Then the formal identifiers are bound to the actual arguments, just as the local variables j and k would be in the block model. The procedure body is evaluated, producing a result. Finally, this result is checked for compatibility with the result type specified for the procedure. If it matches (or can be

converted to match) it becomes the value of the procedure call.

Note that in the block model of a procedure application, the binding of actual arguments to formal parameters (k and j in this example) takes place within a single declaration. Therefore the scopes of all the formal parameters begin together, after the evaluation of all actual arguments and before the evaluation of the procedure body. So no conflicts can occur between identifiers mentioned in the actual argument forms and those used as formal parameter names. Likewise the choice of the name result in the model is not significant. Its scope begins after evaluation of the procedure body.

Next, consider a slightly more complicated explicit procedure, a program to compute the greatest common divisor of two positive integers by Euclid's algorithm, using the remainder routine defined above:

```
gcd <-
  EXPR(m:INT BYVAL, n:INT BYVAL; INT)
  BEGIN
    DECL r:INT;
    REPEAT
      (r <- rem(m, n)) = 0 => n;
       m <- n;
       n <- r;
    END;
  END
```

This procedure uses its formal parameters as variables which may change in the course of computing the greatest common divisor. This technique is often convenient, but one may well ask what becomes of the actual arguments when such a procedure is applied. For example, suppose the statements

```
a <- 30;
b <- 21;
gcd(a, b);
```

are executed in sequence. Will the variables a and b be affected by assignments to their representatives m and n within the procedure body?

A model of the above procedure call shows that the answer is no.

```
a <- 30;
b <- 21;
BEGIN
  DECL m:INT BYVAL a
    DECL n:INT BYVAL b;
  DECL result:INT LIKE
    BEGIN ...  END;
  result;
END
```

Since the formal parameters of gcd have bind-class BYVAL and not SHARED or LIKE, they act as distinct variables, merely initialized by (copies of) the values of the arguments at the point the procedure is called. Binding by value allows the programmer to alter formal variables without affecting argument variables in the calling program.

On the other hand, one often writes a procedure expressly intended to have an effect on one or more of its arguments. Shared binding should be used in this situation, and changes made to the formal parameters inside the procedure body are reflected in the actual arguments which are bound shared.

An example of a procedure intended to leave its argument modified is the following routine to transpose the elements of a square matrix of REALs (defined as VECTOR(n, VECTOR(n, REAL))) about the main matrix diagonal:

```
transpose <-
  EXPR(a:square\matrix SHARED; NONE)
    FOR i TO LENGTH(a)
      REPEAT
        FOR j TO i-1
          REPEAT
            DECL temp:REAL BYVAL a[i,j];
            a[i,j] <- a[j,i];
            a[j,i] <- temp;
          END
      END
```

Since the bind-class indicator in the heading of the EXPR is SHARED, the formal parameter will be bound shared. Thus any proper object will be shared between the calling program and the activation of the routine:

```
transpose(mat1)
```

leaves the matrix mat1 transposed. Recall that proper objects need not be simple variable names. Suppose cube represents a three-dimensional array of mode VECTOR(n, square\matrix). Then cube[i] is a proper object and transpose(cube[i]) is a perfectly reasonable procedure

application, which transposes the ith "plane" of a cube.

If the bind-class indicator had been not SHARED but LIKE, the matrix mat1 would have been left transposed after the call transpose(mat1), since mat1 is a proper object of the expected mode. Had we called transpose with a pure value argument:

```
transpose(CONST(square\matrix OF
                CONST(VECTOR(2,REAL) OF 1.2, 3.4),
                CONST(VECTOR(2,REAL) OF 5.6, 7.8))
```

the result would be the same as if the bind-class indicator had been BYVAL. A matrix is generated, transposed, and forgotten. In this case, the call to transpose is harmless but pointless.

If no explicit bind-class indicator is included in the descriptor of a particular EXPR formal parameter, the parameter will be bound as if LIKE had been explicitly included.

Notice also that the value returned by a procedure may be a proper object. In the block models of procedure application given above, the last declaration, which models the treatment of procedure results, has bind-class LIKE. This is the case for all procedures. Thus if the value of the procedure body is a proper object in the scope of the procedure call, so will be the value of the call itself.

A simple example should make this clear. Let the procedure choose be defined by the assigment

```
choose <- EXPR(b:BOOL; INT) [] b => x; y []
```

The sequence

```
DECL x,y:INT;
choose(TRUE) <- 1492;
choose(FALSE) <- 1066
```

leaves x and y with values 1492 and 1066, respectively.

As mentioned earlier, the actual arguments to each procedure call will be checked against the modes of the formal parameters, as evaluated on entry to the procedure, and the result value will be checked against the formal result mode. In case of a mode mismatch, the evaluator will try to convert the value in hand to the expected mode. The situations in which such conversions are possible are listed in Appendix F. Basically, an INT may be converted to a REAL, a REAL may be rounded to an INT, a pointer may be converted to some compatible pointer mode (e.g. a PTR(INT)

may become a REF, though not a PTR(REAL)), and any value can be converted to mode NONE, i.e. replaced by the value NOTHING. The programmer must take care, however, that data objects intended to be shared between the calling program and a procedure, either via its arguments or its results, are not subjected to automatic conversion. Built-in conversion routines always create a pure value of the expected mode, even if the specified bind-class is LIKE. Suppose, for example, that a variable p has been declared to be a PTR(INT), and that within its scope the procedure

```
set\ptr <-
  EXPR(r:REF LIKE; NONE) (r <- ALLOC(INT BYVAL j*j))
```

is called with p as argument: set\ptr(p). Nothing happens to p. The proper object of mode PTR(INT) is converted into a pure value of mode REF, the expected mode for r. Thus the values of p and r will be distinct, and the assignment to r has no effect upon p. Had the bind-class been SHARED, then set\ptr(p) would have resulted in a sharing fault. The difficulty can be eliminated by changing the declared mode of the formal r to match p's, since it is clear that set\ptr only deals in PTR(INT)'s. Or set\ptr could return the new pointer as its value and the assignment could be made on the calling side:

```
p <- set\ptr()
```

2.11 BIND-CLASSES

The bind-class in the header of the definition of an explicit procedure specifies the relationship between the actual argument and the formal parameter. This has been touched upon in 2.10; here it is treated in detail.

There are five bind-classes in EL1: BYVAL, SHARED, LIKE, UNEVAL, and LISTED. As mentioned in section 2.6 another bind-class FROM is acceptable and is synonymous with BYVAL. To discuss the first three, it will be useful to use a standard example:

```
scale <-
  EXPR(x:REAL <class 1>, y:INT <class 2>; INT)
    BEGIN...END
```

For this example let <class 1> and <class 2> each be either BYVAL, SHARED, or LIKE.

BYVAL specifies that the formal parameter is to be bound by value, i.e. to a copy of the value of the corresponding actual argument. For example, consider

```
scale(aa, bb)
```

If <class 1> and <class 2> are BYVAL, the value of variable aa is obtained, copied and the formal parameter x is bound to this copy. Since the value of x is a copy of the value of aa, changes to x (e.g. by assignment) do not affect the value of aa and changes to aa do not affect the value of x. Since a copy is made, the value of the actual parameter can be converted to the type of the formal parameter if possible (c.f. Appendix F). For example, consider

```
scale(11, 12.6)
```

The first argument, the integer 11, is converted to a real; the second argument, the real 12.6, is converted to an integer by rounding.

SHARED specifies that the formal parameter is to be bound to the very same object as given by the value of the actual argument. For example, consider

```
scale(aa, bb)
```

If <class 1> and <class 2> are SHARED, the value of aa is obtained and formal parameter x is bound to this value. (Similarly, for bb and y.) Since aa and x are names for the same object, changes to aa change the value of x and changes to x change the value of aa. Since no copy is made, the value of the actual argument is never converted. If the mode of the actual argument disagrees with the mode of the formal parameter, then a sharing fault occurs. For example,

```
scale(11, 12)
```

causes a sharing fault for the first argument.

Coercion takes place if <class 1> and <class 2> are LIKE. That is, x is bound either to the value of aa or to a pure value depending upon whether the mode of aa matches the expected mode. If aa evaluates to a proper object of the expected mode, then x is bound to that object as if the bind-class were SHARED. If aa evaluates to a proper object of any other mode or to a pure value, the binding is as if it were BYVAL. In this case, changes to x do not affect any other variable. LIKE should be used when it is unknown whether type conversion will be necessary and a binding to a proper object is desired if possible but a pure value (possibly converted) will be acceptable. LIKE binding is the default and is assumed whenever a bind-class is omitted. If conversion between the actual argument mode and the formal parameter mode is defined, it will be carried out if necessary (as is the case with BYVAL).

The three classes BYVAL, SHARED, and LIKE can be used in specifying the initialization of a local declared variable (c.f. 2.5.1.2). The binding relation between local variable and initialized form in a declaration is the same as the relation between formal parameter and actual argument in a procedure call.

Two additional bind-classes, UNEVAL and LISTED, can appear as the binding specification for formal parameters, but not for declared variables. An example of these two classes may be useful:

```
foo <-
  EXPR(s:FORM <class 1>; FORM) BEGIN...END
```

Let <class 1> be either UNEVAL or LISTED.

UNEVAL specifies that the formal parameter is to be bound to the internal representation (for examples of the internal representation, see Appendix A; a complete definition of FORM appears in Appendix C) of the actual argument with evaluation of that argument inhibited. FORM is the type used in EL1 to represent such unevaluated expressions or abstract syntax. For example, in

```
foo(aa <- bb)
```

if <class 1> is UNEVAL then the assignment is not performed; s is bound to the internal representation of the first argument--a list of three elements: the assignment symbol, the symbol for aa, and the symbol for bb. (In LISP notation this is (<- aa bb).) Similarly, if the argument had been aa+bb*cc then s would be bound to a list of three elements: the symbol for +, the symbol for aa, and a sublist of three elements (* bb cc).

If we did not have the bindclass UNEVAL, then to prevent the evaluation of an argument, the built-in routine QUOTE would have to be applied to that argument at each point of call. QUOTE has the effect of returning its argument without evaluating it.

LISTED specifies that the formal parameter is bound to a list whose first element is the corresponding actual parameter and whose successive elements are the actual arguments remaining to the right. Evaluation is inhibited for all arguments included in the list. For example,

```
foo(aa<-bb, aa*(bb+cc))
```

If <class 1> is LISTED, then s is bound to the list ((<- aa bb) (* aa(+ bb cc))). Effectively, s has taken possession of its own actual argument and all arguments

remaining to the right. Hence, it is meaningful to use the bind-class LISTED only for the rightmost formal parameter.

A similar analogy exists between the bindclass LISTED and the built-in routine QL as between UNEVAL and QUOTE. That is, we could define QL as

```
EXPR(f:FORM LISTED; FORM) f
```

And as before, the bindclass LISTED provides the convenience of not having to apply QL to the list of arguments at every point of call, as would be the case if the formal parameter was specified to be simply a FORM.

UNEVAL and LISTED are useful bind-class types for certain modes other than FORM. Suppose we are writing a dispatching procedure which determines an action to be taken based on the keyword handed it. The header of that procedure might be

```
EXPR(KEY:SYMBOL UNEVAL)
```

The use of UNEVAL simplifies the call of such a procedure and that the mode of the actual argument must be exactly a SYMBOL supplies more specific information to the body of the procedure, and the compiler, than had it been FORM UNEVAL.

2.12 CASE FORMS

The following situations occur frequently in programming:

(1) We want to execute one of a number of conditional alternatives according to the values of a set of predicates (e.g., in a decision table).

(2) We want to execute the Ith action in a set of actions where I is some integer variable.

(3) We want to compute the result of a function of several generic variables in different ways depending on the actual types of the values of those variables.

All of these situations can be conveniently dealt with in EL1 by means of a uniform facility -- the CASE form. The syntax of the CASE form is

CASE <Relations> [<Form*>] <CaseStatement*> <SemiColon> END

where <Relations> is an optional parenthesized list of relation forms, <Form*> is a list of arguments,

<CaseStatement*> is a list of CASE statements, and <SemiColon> is an optional final semi-colon. Each CASE statement consists of a list of phrases (or <CaseLHS>'s), a conditional arrow (=> or #>), and a consequent (any form). Finally, a phrase consists of two parts (both optional) -- a bracketed list of match values and a predicate (any form). The first of these parts is called a <CaseLHS1> and the second a <CaseLHS2>. For example, the following is a valid CASE form:

```
CASE (COVERS, GT) [MD(X), I]
   [INT, 4], [REAL, 10] => R1;
   [INT, 10] => R2 ;
   [BOOL] P(I), [ONEOF(INT, REAL)]] Q(I) #> R3 ;
   TRUE => R4 ;
END
```

The value of a CASE form is the evaluated right-hand-side (consequent) of the first statement of the <CaseStatement*> whose <CaseLHS> 'matches' (in a sense to be described) the header of the CASE form. First, we define matching for the individual match phrases <CaseLHS1> and <CaseLHS2> of the <CaseLHS>. Let the header of the CASE form be

CASE(R1, ..., Rn)[A1, ..., An]

and let

[X1, ..., Xn]P

be a match phrase. The phrase succeeds if and only if

AND(R1'(X1',A1'), ..., Rn'(Xn',An')) AND P' = TRUE

where e' refers to the result of evaluating the expression e. A <CaseLHS> matches the CASE header exactly when either

(1) the operator of the containing statement is => and one of its match phrases succeeds, or

(2) the operator is #> and all its match phrases fail.

Observe that the syntax of a CASE form is more general than is accounted for by this explanation. The remaining cases are dealt with as follows:

(1) If a match phrase's <CaseLHS1> has fewer arguments than the <Form*> of the header, the excess header arguments are disregarded in checking the phrase.

(2) If the <CaseLHS1> has more arguments than the header, the excess <CaseLHS1> arguments are

disregarded.

(3) If <Relations> is non-empty, but there are not enough Ri's for some phrase, then <Relations> is filled out by repeating the last Ri.

(4) If <Relations> is empty, it defaults to (EQUAL). This means the initial binding of the built-in EQUAL -- subsequent user bindings do not affect these semantics.

(5) If either <CaseLHS1> or <CaseLHS2> is empty, the omitted part is assumed to succeed.

Finally, note that the interpretation of a CASE form avoids superfluous evaluations consistent with the above. That is, the header elements are evaluated exactly once; the consequents of statements that do not match are not evaluated; the result is returned as soon as a matching statement is found (without checking any statements which may follow); and checking of a <CaseLHS1> concludes as soon as a phrase succeeds (with the statement succeeding if the operator is => and failing if the operator is #>). If no statement matches, an error is indicated.

It will be best to illustrate this facility by means of some examples. First, consider a procedure for scalar addition. This may be written in a clear, concise format by means of the CASE statement as follows:

```
scalar\add <-
   EXPR(x:scalar, y:scalar; scalar)
      CASE (COVERS) [MD(x), MD(y)]
         [complex,complex] =>
           CONST(complex OF x.re + y.re, x.im + y.im);
         [complex] =>
            CONST(complex OF x.re + y, x.im);
         [ANY,complex] =>
            CONST(complex OF x + y.re, y.im);
         TRUE => x+y;
      END
```

Observe that in the list of relations, only one instance of COVERS is needed -- this is repeated for the y component.

As a second example, consider the problem of selecting one of a set of actions according to the value of an integer variable. This is done by the code

```
CASE[i]
   [1] => A1;
   [2] => A2;
     . . .
```

```
        TRUE => out\of\bounds(i);
    END
```

Since EQUAL is the default relation, we have omitted it from the header.

2.13 COMMENTS

Three operators are built into ECL which permit comments to appear wherever a form is acceptable. Their definitions are equivalent to

```
IE <- EXPR(a:ANY; ANY) (a);

/* <- IE;

*/ <- EXPR(f:FORM UNEVAL, a:ANY; ANY) (a);
```

/ is an infix operator; IE and / are each both infix and prefix. Some examples of their use follow.

```
DECL sigma:REAL /* 'Standard Deviation';


switch <- NOT switch IE 'Flip switch';


/* 'Here when all else fails';


'Exchange next two elements' */
   BEGIN
      t <- a[i];
      a[i] <- a[i+1];
      a[i+1] <- t;
   END;
```

2.14 << AND RETURN

Many of the statement types of EL1 -- the iteration, conditional, and CASE forms -- serve to provide the power and flexibility generally associated with labels and gotos (which are <u>not</u> included in EL1) while retaining clean semantics and encouraging structured, readable programs. In this section we describe a further facility of this sort, allowing a user to <u>mark</u> an arbitrary form and then to return from within its evaluation with a user supplied value as result.

2.14.1 An Example

Let X be an NxM dimensional rectangular array of elements of type STRUCT (KEY:INT, VALUE:INT). Suppose we are given an integer T such that a unique element of X has its KEY component equal to T. We wish to write EL1 code to determine the VALUE component of that element. A first attempt might be

```
FOR I TO N
   REPEAT
      FOR J TO M
         REPEAT  X[I,J].KEY=T => X[I,J].VALUE END;
   END
```

However this program is incorrect -- when the antecedent of the conditional holds, the value of the consequent is returned only from the inner iteration. In general, the outer iteration promptly increments I and continues, discarding the correct VALUE. We can correct this problem with << and RETURN as follows:

```
<< (FOR I TO N
      REPEAT
         FOR J TO M
            REPEAT X[I,J].KEY=T => RETURN(X[I,J].VALUE) END;
      END)
```

Now, when the antecedent is satisfied, the evaluation of the consequent RETURN(...) will cause the correct VALUE to be returned as the result of the dynamically most recent marked form, and that is the outer iteration as required.

2.14.2 <<

In the example above, << is used as a prefix operator with the form to be marked as its single operand. More generally, << serves as both a prefix and an infix operator. When used as a prefix operator, its single argument is the form to be marked. When used as an infix operator, the form to be marked is the right argument and the left argument associates a type and possibly a name with the mark. For example, in

```
INT << f
```

the type INT is associated with the marked evaluation of f. When this evaluation terminates, either normally or via a call on RETURN, the computed value is coerced to type INT and that coerced value is the result of the marked evaluation. If the coercion fails, a type fault occurs. A prefix use of << associates the type ANY with the marked

evaluation.

As an infix operator, << may also be used to explicitly associate a name with the marked evaluation. For example, in

```
<"FOO",INT> << f
```

the name "FOO", as well as the type INT, is associated with the marked evaluation of f. This name may be used by calls of RETURN to select this particular mark and avoid others.

Thus when << is infix, its left argument must be either of type MODE or else a list of two elements. In this latter case, the first element of the list must be evaluable to a SYMBOL and the second to a MODE. When no SYMBOL is explicitly specified, the default value for a SYMBOL (NIL) is associated with the mark. When the left argument to an infix use of << is ill-formed, a mark fault occurs.

2.14.3 RETURN

RETURN takes three arguments -- a VALUE of mode ANY, an ID of mode SYMBOL, and K an INT. (In the example above, ID is defaulted to NIL and K to 0.) RETURN looks for a dynamically enclosing marked evaluation as specified by ID and K. If K=0, then the most recent enclosing evaluation specified by ID is sought; otherwise the (K+1)st most recent such evaluation is sought. ID specifies an evaluation as follows. If ID is defaulted then any marked evaluation will do; otherwise, only evaluations marked with name ID are considered. If no mark as specified by ID and K encloses the call to RETURN, a return fault occurs. If such a mark does exist and has associated type M, then the evaluation of the marked form terminates with result CONST(M LIKE VALUE). Note that RETURN is an unusual function in that it usually does not return a result to its immediate caller, returning instead through some enclosing mark.

2.15 INPUT-OUTPUT

The ECL system provides a number of input-output facilities, many of which are implemented as built-in routines. Higher level input-output is provided by packages which may be loaded by the user during his session. It is suggested that only the first two subsections below be read the first time through this manual and that the remaining subsections be covered when the user requires use of more substantial facilities than those provided by the built-in routines.

2.15.1 Ports

The built-in mode PORT defines a data object used to associate the user's program with a source of input data or a sink for output data. The source or sink may be a terminal device (for instance, the user's terminal, the printer, or a connection established to another computer system), a file on an external storage device, or a temporary file held in the user's main storage (this latter type of port is called a pseudo-port).

2.15.1.1 OPEN, DRAIN, and CLOSE

In order to establish a PORT, the routine OPEN is normally used (see the following subsection for a discussion of pseudo-ports). The arguments of OPEN are:

(1) A symbol stating the information needed by the host operating system to establish operations between the port and a device. Examples are:

```
"TEMP"
"<ECL>UP"                (TENEX monitor only)
"RW.ECL[62,66]"          (DEC monitor only)
"LPT:"
"DSK:DEFALT.ECL"         (The default case)
```

(2) The direction of input-output -- either "IN" or "OUT", the former being the default.

(3) The type of input-output. "SYMBOLIC" is the default case and denotes ASCII character input-output. "BINARY" denotes packed binary input-output.

(4) An optional FORM which is to be evaluated at end of file (see section 2.15.1.3).

Missing arguments cause the corresponding defaults to be applied. For example:

```
P <- OPEN()
```

will open the port P using the disk file DEFALT.ECL in the user's file directory for symbolic input, and

```
P <- OPEN("X.TMP", NIL, "BINARY")
```

will open the file X.TMP for binary input.

At the termination of input-output, the port must be closed. The call

```
CLOSE(P)
```

will accomplish this. An attempt to open the same output file twice without closing it prior to the second OPEN will elicit the message "FILE NOT AVAILABLE", as will an attempt to open a non-existent input file.

The form of the file designator is idiosyncratic to the host operating system. The default extension "ECL" mentioned above, for instance, applies only to PDP-10 implementations of ECL.

The conversion routine PORT\STR takes a port as its argument and returns the associated file-name as a string. In the example above, PORT\STR(P) returns the string ´X.TMP´.

The routine DRAIN may be used to force any output data held in core buffers out to the device without closing the file. This procedure is often required in interactive use of a terminal device. The single argument of DRAIN is the port to be drained.

2.15.1.2 MAKEPF

There are cases in which short, temporary files may be stored in main storage, thus avoiding the overhead required to use external storage. MAKEPF opens a port for this purpose; the mode of its first argument should be a REF compatible with PTR(STRING) for symbolic input-output or with PTR(SEQ(INT)) for binary input-output. The following are examples:

```
P <- MAKEPF(ALLOC(STRING BYVAL ´Gosh;´));
Q <- MAKEPF(ALLOC(SEQ(INT) SIZE K),"OUT","BINARY");
R <- MAKEPF(S,"IN")
```

In each case, the result of the assignment is of mode PORT. Except for the first argument, the arguments of MAKEPF are the same as those for OPEN. Input from P will read the characters in the string ´Gosh;´; output to Q will place binary information into the allocated space; and input from R will take symbolic input from the object S.

End of file is reached, for both input and output, on any attempt to input or output beyond the extent of the user-allocated argument.

2.15.1.3 End of File Handling

Unless the user so directs, reaching end of file will result in a system error in the input-output routine being

used; such an error is non-continuable. The user may, however, supply a form to be evaluated at end of file in either or both of two ways:

The system-defined symbol EOF\TRAP may have a form assigned to it. For instance,

```
EOF\TRAP <-
  QUOTE(RETURN('End of file',"TOP\LEVEL"))
```

Use of RETURN in the end of file form in this example presumes that an outer form has been marked as follows:

```
QL("TOP\LEVEL",STRING) << (BEGIN ... END);
```

When end of file is reached the form EOF\TRAP is evaluated and in this example causes a return from the block with the value 'End of file'.

An end of file form may also be associated with an individual port as the fourth argument of OPEN or MAKEPF. When any input-output routine reaches end of file, the following actions occur (assume that the port P is used):

(1) The file is closed.

(2) EVAL(P.EOF). That is, if the user supplied a form when the file was opened or later stored one in the port, it is evaluated. This may cause a return of control to a higher level, as in the case of port Q in section 2.15.1.2. Otherwise,

(3) EVAL(EOF\TRAP). Again, control may not return to the input routine. If it does,

(4) The message "END OF FILE" is printed and an error break occurs.

Note that step (2) is port-specific, and step (3) represents only an opportunity to recover control for any end of file not having a port-specific action designated.

It should also be noted that the user may change the designated end of file action at various points in his program. This can be done by assigning a new form to the EOF component of the port. For example:

```
P.EOF <- QUOTE(FLAG <- TRUE)
```

2.15.1.4 System Ports

Four ports are permanently defined in the system. These are CIPORT and COPORT (command or top-level input and output ports), PIPORT and POPORT (primary input and output ports). Initially, and immediately after each RESET, CIPORT and COPORT are assigned the user-terminal input and output ports. PIPORT and POPORT have value NIL until another port-value is assigned to them. All input-output routines which take a port as an argument may default that argument, in which case PIPORT or POPORT is used, if non-NIL, otherwise the user-terminal input and output ports are used. The latter default is not used for binary input-output.

The system command interpreter uses CIPORT as a source of commands if at top level (i.e., not at an error break) and user-terminal input and output ports otherwise. Prompting arrows are printed on COPORT if at top level and the user terminal is being used for command input. Otherwise, prompting arrows and the break level are printed on the user-terminal input and output ports. Commands followed by an ALTMODE cause the result to be printed on COPORT.

Hence, the user may cause a command file to be read by assigning a new command input port to CIPORT. This will be effective only at top level, allowing the use of the terminal for responding to error breaks.

2.15.2 Character Input/Output

2.15.2.1 INCHAR and OUTCHAR

These routines input and output a single character at each call. INCHAR takes the input port as its argument, returning the character read. OUTCHAR takes the character to be output as its first argument and the output port as its second, returning NOTHING. For example, the following copies the file T.ECL to U.ECL:

```
BEGIN
  DECL P:PORT LIKE OPEN("T", NIL, NIL, QUOTE(RETURN()));
  DECL Q:PORT LIKE OPEN("U", "OUT");
  ANY << REPEAT OUTCHAR(INCHAR(P), Q) END;
  CLOSE(P);
  CLOSE(Q);
END;
```

Note that the second and third arguments to OPEN and MAKEPF default to "IN" and "SYMBOLIC", respectively.

2.15.2.2 LEX, PARSE, READ, and LOAD

LEX takes an input port as its argument, returning a lexeme. LEX returns an INT, REAL, CHAR, or STRING if the lexeme is one of these modes. If the lexeme is an identifier or delimiter (ALTMODE is treated as a delimiter), a SYMBOL is returned. If the lexeme is a SYMBOL constant, such as "ABC", a REF pointing to the symbol is returned. If the port is at end of file, LEX returns NOTHING.

PARSE uses LEX to attempt to read and parse a form from the input port. The second argument is the output port to which PARSE will write any error messages; if NIL, COPORT is used as the default port. PARSE returns an object with two fields, named CAR and CDR. Thus if we let X be the particular object which PARSE returns, then the result of PARSE may be interpreted as follows

X.CAR = "NOGO"	implies an error occurred,
X.CAR = "EOF"	implies the port was at end of file,
X.CAR = "SUCCESS"	if the parse was successful and the parsed form was terminated by a semicolon, or
X.CAR = "PSUCCESS"	if the form was terminated by an ALTMODE.

In the latter two cases, X.CDR is the form successfully parsed and in the first case X.CDR is the form in which an error occurred.

READ takes an input port as its argument and uses PARSE to input a form. In case of an error, the message is output to the user's terminal and the process is repeated. In case of success, EVAL is used to evaluate the form, and its value is returned by READ.

LOAD takes an input port as its argument and returns NOTHING. LOAD uses PARSE to read forms from the file until end of file is encountered. After reading each form, if no errors have occurred thus far, EVAL is used to evaluate the form.

2.15.2.3 Printing

The routines PFORM and PRINT take three arguments, the data object to be printed, the port to be used for output, and an integer used for controlling the length to which pointer chains will be followed. Both routines return their first argument.

PFORM expects as its argument an object of mode FORM. Its output is in a LISP-like format. (PFORM is invoked by PRINT if its first argument is a FORM). If the third argument, N, is greater than zero, list structure components deeper than N levels of nesting will be output as an elipsis. For example, if the output for N = 0 is

```
(A  (B  (C  D)))
```

then for N = 3 the output will be

```
(A  (B  ... ))
```

2.15.2.4 Formatting and Updating -- the Unparser

ECL has a collection of formatting routines, which are available to a user by issuing the command

```
LOADB "SYS:UP"
```

The main routine UNPARSE has the same argument structure as PRINT. Its purpose however is to produce output in a form that is easily read. UNPARSE also has an optional third argument that controls the level to which blocks and routine calls will be printed. A routine SET\UP is in the package which can be used to control the format. The current settings of the control variables are output as the result of a call on SET\UP with an empty argument list and appear in the form

```
WIDTH 65 INDENT 2 HEIGHT 55 COMMENT 40
```

If SET\UP is called with one or more integer arguments, the corresponding control variable (as indicated in the order above) is set to the value of the argument.

The unparser may also be used for formatting and updating an entire file. The routine UNPARSF takes its first argument, a SYMBOL, as the input file and its second argument, also a SYMBOL, as the name of a file it will create and use as destination for the formatted output. An optional third argument is a list of the forms whose current values are also to be output, possibly replacing existing assignments in the input file. UNPARSF2 is similar to UNPARSF only it assumes its third argument is a FORM bound to the list of forms to be updated.

2.15.3 DUMPB and LOADB

DUMPB is used to dump a set of data objects of any mode whatever, together with any mode definitions required for

reloading, as a binary output file. Sharing patterns are preserved among all the values named.

The first argument to DUMPB is a filename (the default extension used by both LOADB and DUMPB is "BIN"). The second is a list of identifiers; the objects output will be the top-level bindings of the corresponding symbols. The third argument is an integer stating the initial amount of space to be used by DUMPB for its internal hash table. The space actually used is returned on exit; DUMPB will expand its hash table if it becomes full.

The fourth argument to DUMPB is of mode FORM. LOADB, after loading the file, evaluates this form. The form is typically used for program initialization purposes.

LOADB takes a single argument, the symbolic name of the file to be loaded. LOADB is always present in the system. DUMPB must be loaded by the user issuing the command LOADB "SYS:DUMPB".

It should be noted that since DUMPB puts all necessary mode defining information in the output file, a user can get into trouble using old data files if subsequent changes have been made to the mode definitions in the program. The problem can be avoided by using routines available in a file called RW.BIN (see section 8.5 for a discussion of RW.BIN as well as several other routines which handle binary data).

2.16 STRONG PROCEDURE MODES

Section 2.10 suggests that the formal parameter modes, formal bindclasses, and formal result type given in the body of a procedure completely determine the disposition of actual arguments when the procedure is applied and the treatment of the result it returns. That is true if the nominal procedure mode, the mode to which the procedure is bound, gives no description of the procedure's arguments and its result. ROUTINE is such a nominal mode; it is said to be a weak procedure mode because it is general enough to describe any procedure value at all.

More specific ("stronger") nominal procedure modes are needed by the compiler, however, since it does not always have the body of a procedure in hand when compiling a call on it. To produce efficient procedure calls, it must know the number of arguments expected by the called procedure, their expected modes, and their bindclasses. It should also know what type of result the procedure can be expected to return.

Strong nominal procedure modes can be created in EL1 by a mode generator called PROC. PROC expressions look like EXPR headers without parameter names. So

```
        PROC(INT, SYMBOL UNEVAL, FORM LISTED; BOOL),
        PROC(; STRING),
    and PROC()
```

would be proper stong modes for the procedures

```
        EXPR(i:INT,s:SYMBOL UNEVAL,l:FORM LISTED;BOOL)...,
        EXPR(;STRING)...,
    and EXPR()...,
```

respectively. Note that i:INT in the first of these examples is an abbreviation for i:INT LIKE. The same default applies to the PROC expression: PROC(INT, ...) stands for PROC(INT LIKE, ...). Similarly, the third EXPR takes no arguments and has result mode NONE, and the corresponding mode, PROC() is equivalent to PROC(;NONE).

To take a complete example, consider a routine that distributes the effect of a caller-provided operator to each element of a REAL array:

```
distribute <-
  EXPR(array:SEQ(REAL) SHARED,
       operator:PROC(REAL, REAL; ARITH),
       constant:REAL)
    FOR i TO LENGTH(array)
      REPEAT
        array[i] <- operator(array[i], constant)
      END;
```

(ARITH is a built-in GENERIC mode that acts like ONEOF(INT, REAL). ARITH is the formal argument and result mode for the arithmetic SUBRs like +.) Then the call distribute(column\k, *, weight) will multiply each element of column\k by the value of weight. Or, if power is a procedure with header EXPR(b:REAL, p:ARITH; ARITH) that raises b to the p-th power, distribute(a, power, 2) will replace each member of array a by its square. If the nominal mode of a procedure is a strong (PROC) mode PM, it is used (both by the interpreter and the compiler) to determine the disposition of arguments when the procedure is called and to guarantee the mode of the result. The number of arguments bound will be that given by PM, with extras supplied by default or discarded as necessary. Expected argument modes and bindclasses will also be taken from PM.

Any valid EL1 procedure can be bound to any PROC mode; the conversion will be allowed even when the strong mode is in manifest disagreement with the procedure's formal mode

structure. In EL1, a routine's argument and result type forms are evaluated when it is applied, not when it is converted to one mode or another. Therefore, any mismatch is caught at calling time, not at the time the procedure is converted and bound. After the arguments have been prepared but before the procedure is entered, its formal argument modes are evaluated and checked for agreement with the nominal procedure mode PM. The bindclasses must also agree. The number of arguments can still be adjusted at this point, however, with extras generated or dropped to match the specifications of the procedure body. The procedure proper is then entered, and its value is coerced to the evaluated formal result mode given in the procedure specification. Before the result is returned to the caller, one further check is made: the formal result mode must agree with the nominal result mode given in PM.

The role of strong procedure modes in EL1 is mainly declarative, not imperative. The mode comparisons just described are to insure agreement, not to introduce new opportunities for type conversion. Ideally, the nominal mode should match the procedure's formal specification perfectly. The following definitions of mode and bindclass agreement, together with the checking rules given above, spell out the degree of mismatch tolerated during nominal mode validation.

We say that mode M1 agrees with mode M2 if either

(1) M2 is a non-GENERIC mode (or else an extended GENERIC mode, in the sense of section 4) and M1 = M2, or

(2) M2 is a non-extended, GENERIC mode and M2 covers M1.

Two bindclass symbols will be said to agree if they are identical, or if both are members of {"LIKE", "UNEVAL", "LISTED"}.

If a disagreement is found, an error occurs: "FORMAL/NOMINAL MISMATCH".

Look again at distribute, the example given above. When the call distribute(column\k, *, weight) is evaluated, operator, a PROC(REAL, REAL; ARITH) variable, is bound to *, the built-in product routine whose two formal parameter modes and formal result mode are of type ARITH. When operator is applied to array[i] and constant in the body of distribute, both arguments are coerced (albeit trivially) to REAL, the nominal mode of operator's two arguments. Before * is entered, the evaluator makes sure that each nominal argument mode (REAL) agrees with each formal argument mode (ARITH, which COVERS REAL) and that the bindclasses agree (all are "LIKE"). On exit, it must also be verified that

*'s formal result type (ARITH) matches operator's nominal result type (also ARITH).

Note that the formal result mode must agree with the nominal, and that a "FORMAL/NOMINAL MISMATCH" error can occur even though the mode of the actual value returned may agree with the nominal result mode. For example, if operator were given the mode PROC(REAL, REAL; REAL), then the built-in * procedure would not be a suitable binding: its result mode, ARITH, is broader than REAL, and so does not agree with REAL. Remember, strong nominal modes are mainly for making verifiable declarations, not for introducing type conversions. (As will be mentioned later, this restriction encourages programming practices that lead to the most efficient compilation of procedure applications.)

For similar reasons, a ROUTINE with the heading EXPR(p:PTR(INT, REAL)) should not be bound as a PROC(PTR(INT)). PTR(INT) does not agree with PTR(INT, REAL) even though any PTR(INT) could be converted to PTR(INT, REAL). Nor should an EXPR(t:REAL BYVAL) be bound to a PROC(REAL) variable; even though the programmer may not be concerned about protecting the actual argument, the bindclasses must agree.

There is some flexibility in the checking rules. For example, distribute's second argument need not be a binary operator. Let abs be a PROC(ARITH; ARITH) that returns the absolute magnitude of its operand. Then distribute(a, abs) will successfully replace each a[i] by abs(a[i]). The second operand supplied by the call to operator in distribute will be ignored in this case.

The fact that LIKE, UNEVAL, and LISTED are mutually agreeable bindclasses can also be very convenient. Suppose DUMP is a routine with the heading

```
EXPR(file:SYMBOL UNEVAL,
     values:FORM LISTED)
```

DUMP(TRIG, SIN, COS, ATAN) might create a file named "TRIG" and output the values of SIN, COS, and ATAN to it. Then another variable, say DUMP2, with nominal mode PROC(SYMBOL, FORM) could be bound to the same routine and yet take its arguments evaluated:

```
DUMP2(concat('TRIG', '.ECL'),
      append(<SIN>, other\fns))
```

Finally, a word about efficiency. As has been mentioned, strong nominal modes exist primarily to enhance the efficiency of compiled procedure application. Yet this

section has described several call-time checks needed just to validate the strong modes. In fact, for the most critical cases, those in which the callee is a compiled or built-in routine, call-time validation can usually be eliminated entirely. When a machine code routine, built-in or compiled, is converted to a PROC mode, it is checked for agreement with the new nominal mode. The conversion will succeed regardless of how the test turns out; mismatches will be reported only at call time. But if all the procedure's formal modes are constants (the compiler replaces many simple mode expressions by constants), if all modes and bindclasses agree with the nominal mode, and if the numbers of arguments are equal, then the representation used for the converted routine will indicate that no call-time validation is necessary.

Procedures are normally called much more often than they are converted from mode to mode. So it is to the programmer's advantage to have the nominal mode of a routine agree with the routine it modifies. Additional flexibility, such as allowing distribute's operator argument to be a PROC(REAL, REAL; REAL) but still take PROC(ARITH, ARITH; ARITH) values, could impose a great deal of call-time overhead as the price for superficial convenience.

3. HOW TO USE ECL

3.1 GETTING INTO ECL

Assuming you have logged in to the DECsystem-10 (with 10-50 monitor), you can enter ECL by typing, in response to the monitor's prompting dot, the command

```
.R ECL
```

(To enter ECL from a tenex system, type <ECL>ECL at monitor level.) ECL will respond with a title line identifying the current version of the system and will then type the prompt symbol, "->". This is ECL's signal that it is expecting input from the user.

A command may now be typed for evaluation by the ECL command interpreter. A command can be either:

(1) a form followed by an ALTMODE (indicated by $ in the examples below)

(2) a form followed by a semicolon (;)

(3) an ALTMODE

or

(4) a semicolon (;)

In cases (1) and (2), the form will be evaluated. In case (1) its value will be PRINTed and in case (2) this output will be suppressed. Cases (3) and (4) are present for convenience and completeness -- (3) will output NOTHING and (4) will not output. Since semicolon is not a line terminating character, some such character -- usually CARRIAGE-RETURN -- must follow cases (2) and (4).

In all cases, after the described action is taken, the command interpreter is ready to accept another command, which it signals by giving the prompt arrow. For example:

```
-> X <- 4*3+2;

-> Y <- X+1$
15

-> Y+1$
16
```

Syntax errors in input commands are signalled by an error message on the user's terminal. Commands containing syntax errors are not evaluated--the command interpreter simply waits for another command. (If a syntax error occurs

in the middle of a multi-line command, it may be necessary to type ^Z(CTRL-Z) to terminate the input of the incorrect command.)

```
-> X+1)$

X + 1 ??? ) ;

-> (X+1)$
15

->
```

The question marks in the error message indicate where the parse error was detected in the command.

3.2 THE TOP-LEVEL ENVIRONMENT

In ECL, variables may be used at two "levels":

(1) local to blocks (i.e., DECLared variables) or as formal parameters in routines;

and

(2) at the "top level", outside any block or routine.

There are two points we want to emphasize with respect to these two uses. The first concerns the special use of the assignment operator at the top level. Recall that when a form such as X<-F(Y) gets evaluated, the interpreter will normally check that the mode of X is appropriate before performing the assignment. However, if this form is given at command level, the interpreter behaves differently. If X has not been previously defined (by an earlier top-level assignment), then the mode of X, as well as the value of X, is determined by F(Y), and this mode cannot be changed. If X has been previously set, then it has a mode, and normal ECL assignment occurs. In short, the initial assignment to a variable at the top level has the effect of DECLaration, in that it fixes the mode of the variable. To illustrate:

```
-> X <- 8*7    /*  'FIXES THE MODE OF TOP-LEVEL
 X';

-> X <- "ABCDE";
TYPE FAULT

1:>RESET       /* 'SEE SECTION 3.5 FOR THE MEANING
                    OF "1:>"';

-> Y <- X+2   /* 'NOW Y IS ALSO AN INT';

-> Y <- [)DECL X:STRING SIZE 6;
          X <- 'HAMLET';
          17(]$
17

-> X$
56

->
```

The top-level binding of a variable exists independently of any block or routine; a variable defined at the top-level may be present for the duration of the user's session in ECL. Local or formal variables exist concurrently with the block or routine in which they are defined.

In general, the top-level binding of a variable X is used if and only if X is neither local to any dynamically enclosing block, nor a formal parameter of any dynamically enclosing procedure. To illustrate:

```
-> X <- 47;

-> P1 <- EXPR(X:BOOL; INT) [)P3(); X => -1; 5(]

-> P2 <- EXPR(;INT)[)X <- X+1(];

-> P3 <- EXPR(;BOOL)[)X <- NOT(X)(];

-> P1(FALSE)$
-1

-> P2();

-> X$
48

->
```

3.3 CREATING AND EDITING PROGRAMS

Commands may be issued to the ECL command interpreter either (1) directly from the user's terminal, or (2) through a prepared file of commands.

The editing facilities available at the user's terminal are as follows:

(1) RUBOUT will erase a single character; repeated RUBOUTs can erase characters until (but not including) the most recent "activation character". (Activation characters include CARRIAGE-RETURN, LINE-FEED, and ALTMODE.) The erased character will be echoed on the teletype, enclosed in back-slashes("\").

(2) CTRL-U will erase everything typed in since the most recent activation character (usually this means the current line).

It is recommended that the user prepare a separate file for commands longer than one or two lines. The reasons for this suggestion are that (1) there is no way of editing a previous line of a command being entered on the teletype and (2) if a syntax error is detected in a command, the whole command must be retyped.

Command files can be created and modified in ECL by calling the DECsystem-10 Text Editor (TECO) as a routine or by using the ECL list structure editor. (For a description of TECO, see the DECsystem-10 User's Handbook, book 8, or the PDP-10 Reference Handbook, book 4 and for a description of the ECL editor, see section 5.) To create a new file, using TECO from within ECL, call the routine MAKE with one argument--a SYMBOL--which will be the name of the file. If no extension is specified with the file name, the default is ".ECL".

For example,

```
-> MAKE("XANADU");
HTECO.XX.X
*IX <- 5; X <- X+1;
$$
*EX$$

->
```

Closing a file by EX will return control to ECL. Closing a file by EG will cause control to return to ECL, after which the file will be loaded (i.e. each command in the file will be parsed and evaluated in turn).

```
-> MAKE("HAMLET");
HTECO.XX.X
*IZ <- 14;
$$
*EG$$

-> Z$
14

->
```

Files may be edited by calling the routine TECO with one argument--a SYMBOL--with the same conventions as MAKE.

```
-> TECO("XANADU");
HTECO.XX.X
*LIX <- X+2;
$EG$$
-> X$
8
->
```

If the argument to MAKE or TECO is defaulted, the argument provided in the previous call to MAKE or TECO, if there has been one, will be assumed; otherwise "DEFALT.ECL" is taken as the argument.

Note, finally, that the user may accidentally destroy parts of a file or otherwise alter it unintentionally. If the error is noted before either exiting TECO or closing the output file, the teco command sequence

```
*^Z
$$
```

should avoid changing the original file.

3.4 SIMPLE INPUT/OUTPUT ROUTINES: READ AND PRINT

ECL provides a wide variety of input/output facilities; most of these will be discussed in a later section, though the two most basic routines will be covered here.

The built-in routine READ is used to accept a single command (a form followed by a semicolon or ALTMODE) and return its value. For example:

```
-> W <- []DECL X,Y:INT;
          REPEAT
             (X <- READ()) = 0 => Y;
             Y <- Y+X;
          END ();
34       /* 'Now X is 34';
X-30     /* 'Now X is 4';
2*X      /* 'And now X is 8';
0;

-> W$
46

->
```

In order to output the value of a form to a user's terminal, the system routine PRINT may be called with one argument (the value to be printed). For example,

```
-> PRINT(3.142-.718);
2.424

-> []PRINT("MADAM I"); PRINT(%');
     PRINT('M ADAM') ();
MADAM I'M ADAM


->
```

PRINT returns as its value the argument given for printing. To illustrate a use for this, as well as the editing features described in the preceding section:

```
-> MAKE("FIBSEQ");
HTECO.XX.X
*IM <- 0;
N <- 1;
FOR K TO 11
  REPEAT
    [) K=2*(K/2) => N; M (] <-
    [) PRINT(% ); PRINT(M+N) (];
  END;
$EG$$
 1 2 3 5 8 13 21 34 55 89 144

->
```

The above program prints the first eleven elements of the Fibonacci sequence.

3.5 ERRORS AND PROGRAM SUSPENSION

3.5.1 Execution Errors

Program errors may fall into two categories: syntax errors, detected by the parser (discussed in section 3.1) and execution errors detected during evaluation. When the latter occur, the ECL break package is invoked; this will print out (1) a descriptive message identifying the error (see Appendix E for a list of these messages), (2) a line identifying where the error occurred (unless it occurred at the top-level), and (3) a special prompting signal consisting of an integer followed by ":>".

For example:

```
-> X<-'ABCD';

-> S<-48;

-> T<-5;

-> F<-EXPR(S:STRING; CHAR)[)X[T]<-S[1](];

-> F('HAMLET')$
INVALID INDEX
F BROKEN
1:>
```

The error occurred because X has only 4 components. The message "F BROKEN" indicates that F was the most recent routine entered before the error. The integer 1 preceding ":>" indicates the break level on which the program has been interrupted. The break level is the number of breaks pending in the current environment. When the ":>" occurs, the user may type in commands just as he would at the top level; the difference is that the environment of the computation--the set of variables and their bindings--is the one current when the error occurred. The user in this way can examine (and change) the values of variables to determine why the error took place. The routine BT, described in section 8.1, is frequently of help in this situation. To continue with the above example:

```
1:> S$
HAMLET
1:> X$
ABCD
```

When the error is correctable, the user may continue the interrupted computation by calling the built-in routine CONT. CONT takes one argument: a form whose value will replace the value which caused the error.

```
1:> CONT(T-1);
H

-> X$
ABCH
```

An error may occur which cannot be corrected. In this case the user may return to the top level by executing the nofix procedure RESET or to level I (that is, the level at which 'I:>' is the prompt) by executing RETBRK(I).

```
-> Y<-14;

-> G<-EXPR(;CHAR)(Y[1]);

-> G();
CANT SELECT
G BROKEN
1:> Y$
14
1:> RESET;

->
```

3.5.2 Setting Breakpoints

The ECL break package can be invoked under circumstances other than error conditions. In particular, the routine BREAK may be called from a user routine in order to set a breakpoint at that spot (a valuable feature in debugging). BREAK takes one argument. Unless the argument is NOTHING, it will be printed on the user's teletype when BREAK is called.

```
-> P <- EXPR(X:INT; INT)
        [) X LT 0 => BREAK("X NEGATIVE");
           2*X(];

-> P(3)+100$
106

-> P(-3)+100$
X NEGATIVE BREAK
P BROKEN
1:> X$
-3
1:> CONT(X-7);
90

->
```

Note that CONT has been used to exit from the break package and to return to the interrupted computation. If an argument is provided to CONT, it will be the value returned by the call on BREAK in the program which was suspended. The user may have calls to BREAK automatically inserted in and removed from his programs by using the TRACE package described in section 8.1.

3.5.3 User-Defined Error-Handling Routines

An extensive error-handling facility is currently under development. A few ad hoc facilities are presently available, however. We have associated a SYMBOL with each of several commonly occurring error conditions. When one of these errors occurs, a form which has been bound to the corresponding SYMBOL is evaluated in the environment of the error. The errors which may be thus intercepted are

Condition	Symbol
End of File	EOF\TRAP
Undefined Procedure	UPROCF
File Not Available	FNA\E
Type Fault	TF\E
Invalid Index	IVX\E
Stack Overflow	STK\E

Some of these errors permit corrective action to be taken and execution to continue. For instance, it is possible to proceed from a type fault by issuing the command CONT(x) where x has a suitable value of the appropriate mode. Other conditions may require redoing some part of the computation prior to the error.

Two routines PEEK and POKE which are described in section 7.7 may also be useful in recovering from errors.

3.5.4 Emergency Measures

It is sometimes necessary to terminate execution of a program prematurely (because of an infinite loop, for example). In such cases the user should type CTRL-C twice: this will cause a return to the PDP-10 monitor. Following the monitor's prompting ".", the REENTER command should be typed. This will pass control back to the ECL break package, using which the user can examine or change variables, RESET, etc.

Manual intervention may also be useful when unwanted output is produced by an ECL program. In this case, typing CTRL-O will terminate the output while allowing the ECL program to continue its computation. Under some

circumstances, the prompt symbol "->" gets suppressed along with the unwanted output. If the system has not given the prompt symbol within a few seconds, type a short command (e.g. 1$) to obtain it.

If problems occur in returning from TECO to ECL, the TECO command CTRL-Z$$ will sometimes return control to ECL, leaving the edited file unchanged. If an EF command has been issued while in TECO, this will not work.

4. EXTENDED MODES AND THEIR BEHAVIOR

Section 2.7 describes how to produce new data types from old using the operators VECTOR, SEQ, STRUCT, ONEOF, and PTR. We now describe another operator, ::, that adds power to ECL's mode definition facility in two ways: (1) it permits a degree of self-reference in mode definitions, and (2) it allows the definer to control the treatment of data by ascribing behavior attributes to modes. The :: operator produces, from a given mode M, an extended mode with attributes, mainly in the form of user-defined mode behavior functions, that selectively supersede the properties of the underlying mode M. A couple of examples may help motivate the need for this capability.

Imagine representing a linked list of integers in the following way

```
ELEMENT <- STRUCT(VALUE:INT, LINK:REF);
INT\LIST <- PTR(ELEMENT)
```

Then

```
IL <- ALLOC(ELEMENT of 10, ALLOC(ELEMENT of 20))
```

produces a two element structure of the form

```
STRUCT (10, PTR(STRUCT(20, NIL)))
```

But note that the LINK component of an ELEMENT will always be used to point to ELEMENTs. Recalling the pragmatic discussion in section 2.7.5 one might want to narrow the definition, so that ELEMENT is defined as a

```
STRUCT(VALUE:INT, LINK:PTR(ELEMENT))
```

This definition cannot be achieved with the ordinary mode generators alone since it involves self-reference: ELEMENT cannot be defined until ELEMENT has been defined. To handle self-reference, ECL allows symbolic references to extended modes created by the :: operator to appear in mode definitions and permits resolution of these references to be delayed:

```
ELEMENT <- STRUCT(VALUE:INT, LINK:"INT\LIST*");
INT\LIST <- "INT\LIST*" :: PTR(ELEMENT);
```

When first defined, ELEMENT will be incomplete, since "INT\LIST*" will not have been associated with a mode. After the definition of INT\LIST, however, ELEMENT will be completed. Completion takes place automatically when an ELEMENT value is created or when the mode is used in

defining other modes. (See section 4.1.1 for details.)

Extended modes are also used to distinguish unique classes of data objects. Suppose we wish to add the arithmetic types "complex number" and "long real" to those already provided in the basic system. Though the two types are intended to behave differently, for example under an extended + operator, the representation used might be the same for both, e.g., VECTOR(2, REAL). Of course, if both COMPLEX and LONG\REAL were defined as VECTOR(2, REAL), there would be no way to distinguish COMPLEXs from LONG\REALs on the basis of mode. The :: operator provides a solution here too, in that it permits us to create distinct modes from the same underlying mode, e.g.:

```
COMPLEX <- "COMPLEX*" :: VECTOR(2, REAL);
LONG\REAL <- "LONG\REAL*" :: VECTOR(2, REAL);
```

Moreover, as will be seen in later sections, the treatment of extended modes during type conversion, value initialization, and in several other situations can be modified by the mode definer through the :: operator.

4.1 EXTENDED MODES

4.1.1 Completion

In the example above we defined a representation for a list of integers which involved deferred completion of a mode. The completion of the mode ELEMENT, and thus all substitutions of the actual mode for any symbolic reference, automatically takes place when an object of the mode ELEMENT is created. Also, if the mode is encountered in some subsequent mode definition, an attempt is made to complete it. A third way to cause completion of a mode is to call the built-in routine COMPLETE which takes a list of modes and attempts to complete each of them, as well as any embedded modes. COMPLETE returns a value, a boolean, indicating whether all of the modes in its argument list has actually been completed.

One additional concern when defining modes involving forward reference (often called recursive modes) is the requirement that every recursive chain contain at least one PTR level. Most implementations of these modes would use pointers; we simply choose to make the pointer explicit.

4.1.2 Tags

The symbolic reference occurring in an extended mode is called a tag or shortname. The association of a tag with an

extended mode is global and unique. The :: operator first decides whether its call represents a new definition or the modification of an existing mode. If its left operand is already a mode, M, or if the symbolic tag is already attached to a mode M, the call is a redefinition. In this case, :: checks that its right operand is equal to M's underlying mode. If not, the redefinition is not allowed ("ILL MODE REDEF'N"). The existing mode must be deleted before the new definition can take place. (See section 4.4 for a discussion of mode deletion.)

For obscure historical reasons, tags are treated as syntactic constants by the ECL parser. Thus, just as INT and REF are reserved tokens whose meaning is not alterable, and whose use as identifiers is proscribed, the identifier M becomes a reserved word after the operation

```
M :: SEQ(CHAR)
```

This convention can lead to awkward name conflicts, particularly when packages written by different programmers are merged. Since a mode tag may be any symbol value, however, one partial solution is to use tags that will not be recognized as single tokens during parsing and so cannot conflict with ordinary identifiers. The examples in this manual use mode tags that end with "*", as in "INT\LIST*". Of course, if two programmers use the same scheme for making up "funny" tags, they may still have conflicts when putting their programs together.

Two extended modes are considered equal exactly when their tags are equal (the same SYMBOL). Thus the tag is the key in determining equality of objects whose modes are extended.

4.1.3 The :: Operator

The :: operator is a built-in infix operator used with tags in defining extended modes. Its right operand is taken evaluated and coerced to MODE, becoming the underlying mode, or underlying representation. The left operand is taken unevaluated and interpreted as follows: if it is already an identifier or a mode, it is taken literally as the tag or expected mode, respectively. Otherwise it is evaluated and coerced to FORM and if the result is symbolic (technically, it points to an ATOM), this form is then taken as the tag. Otherwise it must be a list of forms. The first element is evaluated and coerced to SYMBOL to become the tag. The rest of the list specifies the mode's behavior attributes (described in section 4.2).

4.1.4 LIFT and LOWER

The two built-in routines LIFT and LOWER exist to facilitate the handling of objects of an extended mode. LOWER permits a user to look "below" the extended mode definition and deal with the object as if its mode were an underlying mode. The routines which define an extended mode's behavior (see section 4.2) must occasionally have this ability in order to avoid repeated invocations of a behavior routine, and possibly an infinite cycle of calls.

LOWER(X, NEW\MODE) returns an object whose value is X and whose mode is NEW\MODE. MD(X) must either be NEW\MODE itself (in which case LOWER simply returns X) or it must have been extended from NEW\MODE by one or more uses of ::. The second argument to LOWER may be omitted in which case the mode of the object returned is X's directly underlying mode. LIFT(X, NEW\MODE) returns the value of X treated as if its mode were NEW\MODE. NEW\MODE must either be MD(X) or an extension of MD(X), directly or through other extension levels. Both LIFT and LOWER return values shared with the input X; they do not make copies. If a type fault occurs in using either LIFT or LOWER, CONT(OBJ) will substitute a new value OBJ to be lifted or lowered.

In the definition of an extended mode, a field in the internal representation of the mode, the UR field, contains the underlying representation of the extended mode and is accessible to the user. That is, if MD(X) is extended,

```
MD(LOWER(X)) = MD(X).UR
```

4.2 MODE BEHAVIOR

A considerable amount of power and clarity of expression is available in EL1 through the use of mode behavior functions attached to extended modes. There are six behavior functions which can be defined controlling the generation, conversion, assignment, selection, printing, and dimensions of objects of the extended mode.

It is through the :: operator's left operand that we associate the behavior attributes with a mode. The left argument to :: may evaluate to a list of forms. In this case, the first element is taken as the tag. The elements after the first define the fundamental behavior of the extended mode and are interpreted according to a keyword scheme. Each of the elements consists of a keyword identifier and a value for the keyed attribute, in the format of a procedure application. The keywords for the six functions are, respectively, UGF, UCF, UAF, USF, UPF, and UDF for the attributes listed above.

Thus if we wish to ascribe assignment and selection functions to a mode for trees, we could say

```
TREE <- QL("TREE*", UAF(TREE\UAF),
                    USF (TREE\USF)) :: TREE\UNDERLYING
```

where TREE\UAF is a user-defined assignment function for objects of mode TREE and TREE\USF, the user selection function.

The forms associated with each keyword are not evaluated; they are taken literally and associated with the extended mode. When a behavior function is needed during the running of a program, the corresponding form must evaluate to a valid procedure. An empty value form for one of these keywords deletes the corresponding behavior function form. Deleting behavior functions is legal both on definition and redefinition with one exception: the presence of a UGF for a mode may not be changed by redefinition. The UGF attribute form itself may be changed, but not added to nor deleted from an existing mode. This is because definitions of modes that embed some mode M depend upon the knowledge of whether M has a UGF. As with the other restrictions on mode redefinition, this one can be circumvented in practice by expunging all references to the mode and defining it anew (see section 4.4).

Two additional attributes exist which can affect the behavior of extended modes, the SUPUGF (suppress user generation function) and DADV (default apparent dimension vector) features. SUPUGF is used to suppress calls on the UGF when the mode of the object provided matches the mode desired. (Details on SUPUGF are in section 4.2.1.3.) The DADV is fully described in section 4.2.6.1 below.

To summarize, the keyword/attribute pairs accepted by :: are as follows:

Keyword	Meaning	Attribute Mode	Evaluated/ Quoted	Redefinition Restricted
UGF	Generation Function	FORM	Quoted	Yes
UCF	Conversion Function	FORM	Quoted	No
UAF	Assignment Function	FORM	Quoted	No
USF	Selection Function	FORM	Quoted	No
UPF	Printing Function	FORM	Quoted	No
UDF	Dimension Function	FORM	Quoted	No
SUPUGF	Suppress UGF	BOOL	Evaluated	Yes
DADV	Default Apparent Dimension Vector	SEQ(INT)	Evaluated	Yes

4.2.1 User-defined Generation

4.2.1.1 Format of the UGF

The procedure taken to be the user's generation function (UGF) may be called whenever there is a need to generate an object of mode M, as in a call on CONST(M...), ALLOC(M...), DECL id: M..., or in binding actuals to formal parameters. The UGF expects three arguments: (1) the mode M, (2) a bindclass symbol, (3) a specification value. Though the purpose of the call is to produce an object whose mode is covered by M, this is not a strict requirement, since conversion follows generation whenever necessary. It suffices for the UGF's output to be convertible to mode M.

The bindclass symbol will never be NIL. Its value falls in one of three classes, corresponding to generation by example (bindclasses LIKE, SHARED, BYVAL, FROM), generation by size (bindclass SIZE), and generation from components (any other bindclass). Generation by default results in a call with bindclass SIZE and a default sequence of dimensions. The bindclass determines the type and meaning of the specification value as follows:

Bindclass	Specification Mode	Specification Value
"LIKE", "SHARED", "BYVAL", "FROM"	ANY	Prototype or initial value
"SIZE"	SEQ(INT)	Dimensions of expected object
Any other symbol	FORM	Unevaluated component forms

In the case of generation by example, the initial value arrives evaluated, but neither coerced nor copied. It is this value that, in the SHARED case, must have a location identical to that of the result of the generation process. "BYVAL" (or equivalently, "FROM") guarantees that the result of generation will be copied, if necessary, to produce a pure value -- thus the UGF need not bother.

In the "SIZE" case, the programmer may assume that the dimension vector provided via the sequence of integers has exactly the right length for objects of mode M. If the caller has specified too few dimensions or none at all (generation by default), the extras will come from the DADV for M (see section 4.2.6 below). If too many dimensions have been specified, all will be evaluated, but an error will occur to warn the user that some are being lost. CONTinuation will cause the given vector to be truncated and generation to continue. The dimension vector will be a pure

value and thus can be modified safely by the UGF.

The third bindclass is a catch-all case; it includes a built-in bindclass "OF" but also includes any symbol the user wishes to put in the place of "OF", thus providing a convenient mechanism for extending the information passed to a user's generation function. For this third case, the specification value is coerced to mode FORM and is guaranteed to be a pure value. When "OF" appears explicitly in the call, the form supplied is the list of unevaluated forms which appear after the bindclass. Before a symbol S may be used in place of "OF", it is necessary to execute POKE(OFBIND, "S"). This will destroy any other parse properties of S and, in particular, will prevent its use as an identifier.

4.2.1.2 When Is a User Generation Function Called?

To begin let us state the following general rule: a user's generation function for a mode M will be called, if so specified by the user, every time a new value is needed and M is the expected mode. That is, no object of an extended mode will be created without giving its UGF an opportunity to build or inspect it. Thus if M has a UGF, but SEQ(M) does not, and we encounter

```
DECL S:SEQ(M) SIZE 3
```

then the user's generation function for M will be called. This leads us however to the question of how many times it will be called and with what bindclass and specification, and to the concept of replication. But before we go into the explanation of replication, the reader should be forewarned that the following is an informal presentation. The examples used are explicit calls on DECL, but similar issues arise in all other instances of generation, whether from calls on CONST or ALLOC or implicit calls on DECL. For a complete explanation of when these implicit calls occur and a full understanding of the mechanics of declarations and other instances of generation, the reader should see section 4.3.

Replication of a value is performed whenever more than one instance of the value is required and each is constrained to be a distinct but "equivalent" copy of the value. Replication could therefore invoke (possibly several) calls on a user's generation function associated with the mode of the value. Replication occurs in three different ways: (1) row generation, (2) generation of more than one variable in a single declaration, and (3) assignment at top-level.

Considering the simplest case first, assignment at top-level produces a call on a UGF associated with the mode of the right operand, with bindclass "BYVAL" and specification value the right operand.

Next let us consider case (1), where we are generating a row of objects, through use of either VECTOR or SEQ. If we accept the general rule stated above, the primary issue becomes what form the call(s) on the user's generation function will take. Again the issue breaks down into three cases, depending on the bindclass of the statement prompting generation, whether it is (1) "SIZE" (or default), (2) "OF", "BYVAL", or "FROM", or (3) "LIKE" or "SHARED".

Suppose we encounter a statement of the form

```
DECL S:SEQ(M) SIZE 3
```

where, as before, SEQ(M) does not have a UGF but M does. Three calls are made on the UGF associated with M. The first is a call by default, and the remaining are calls with bindclass "BYVAL" and the result of the first call as the specification value.

In the case that "OF" is the bindclass as in

```
DECL S:SEQ(M) OF M1, M2, M3
```

there are again calls on the UGF for M for each of the components provided. The bindclass for each call is "BYVAL". That is, the first call would be given M1 as the specification value, the second, M2, etc. Similarly, if the initial declaration appeared as

```
DECL S:SEQ(M) BYVAL SM
```

there would be successive calls on the UGF for each of the components of SM.

If the bindclass is "LIKE" or "SHARED", no explicit call to create new objects of mode M is indicated so there are no calls on a user-defined generation function for M.

In each case, however, the system guarantees that all components of the sequence or vector have the same physical dimensions.

Suppose next that the declarations were of the form

```
DECL X, Y:M <Bindclass> <Specification\value>
```

where <Bindclass> is neither "LIKE" nor "SHARED", and M is an extended, non-generic mode with a generation function.

Then, as would be hoped, there are two calls on M's UGF. In the first call, the Bindclass and Specification\value appearing in the declaration are used to produce X. The second call results from the replication of the first value to produce a value for Y. So the Bindclass will be "BYVAL" and the Specification\value will be the value produced for X. Even in the case M is a generic mode, if the mode of the value for X has a user-defined generation function, that generation function will be called to produce replicas for the other variables created by the declaration.

One not so obvious source of a call on a user's generation function is during assignment. In system assignment, if there is a UGF associated with the mode of the left operand, it would be called during evaluation of the right operand. That is, we can imagine an operation of the following sort taking place

```
DECL <Right\value>:MD(<Left\operand>) BYVAL
  <Right\operand>
```

which would prompt a call on a UGF defined with the mode of the left operand.

Yet another unobvious instance in which a UGF can be called is in the RETURN from a marked form. The evaluation of an expression of the sort

```
QL(<Mark\name>, <Expected\mode>) << <Marked\form>
```

where the Marked\form contains a call on RETURN with value V, terminates with the result

```
CONST(Expected\mode LIKE V)
```

4.2.1.3 SUPUGF

As implied at the beginning of the previous section, there is a way to circumvent our general rule. The call on a user-supplied generation function for a mode M may be suppressed, only in the case of generation by example and only when M covers the mode of the specification value, if M has been given the SUPUGF property.

The SUPUGF property is associated with an extended mode in the same way the other behavior attributes are specified. That is, SUPUGF is one of the keywords which may appear in the list of forms as the left operand to the :: operator. The form associated with SUPUGF is evaluated and coerced to BOOL. If the value is TRUE, the UGF calls will be suppressed during generation by example unless a mode mismatch occurs. When the SUPUGF keyword appears but

without a value form (e.g., SUPUGF()), it is treated like SUPUGF(TRUE). If the SUPUGF attribute does not appear at all, the property comes from the underlying mode if that mode has a UGF; otherwise SUPUGF is taken to be TRUE.

The SUPUGF property of an extended mode may not be freely redefined. Its attribute value is carried up to (unextended) compound modes in which the extended mode is embedded, and therefore any changes to SUPUGF could create peculiar inconsistencies. To redefine the property, the extended mode, and all values of the mode, must be deleted (see section 4.4).

The reason for including the SUPUGF property is efficiency. With many data types, generation by example requires no special action if the expected and actual modes agree. In fact, if a mode M does not have the SUPUGF property, caution is advised in specifying the formal modes of the UGF for M. In particular, if the result mode is declared to be M, the UGF will be reinvoked while trying to return its result. To get around this problem without eliminating declarative information, we suggest defining a generic mode, say VALID\M, as follows: if M is already of class generic, let VALID\M be the deepest underlying mode in which M is based. Thus VALID\M would be ANY or a mode of the form ONEOF(M1, ..., MN). If M is not generic, let VALID\M be ONEOF(M). In either case using VALID\M in place of M would achieve the declarative effect without forcing M's UGF to be called.

4.2.1.4 CONSTRUCT

In writing mode extension packages it is often useful to be able to circumvent calling a UGF when the developer is certain of the validity of the object being created. Also, in generation by components ("OF"), if we already have the list of component forms (unevaluated), we may simply want to perform the generation without further ado.

For these reasons, we have a built-in routine called CONSTRUCT which takes three arguments, (1) the mode of the object being constructed, (2) the bindclass, a SYMBOL (evaluated), and (3) a specification value whose formal mode is ANY. CONSTRUCT works just like CONST except that the initial evaluation steps have already been performed. Thus, for instance, when the bindclass is "SIZE", CONSTRUCT coerces its third argument to SEQ(INT) and tailors its length to match the expected mode. For generation from components, the specification value is coerced to FORM for use as a list of unevaluated components. For the "BYVAL", "FROM", "LIKE", and "SHARED" cases, CONSTRUCT's third argument becomes the initial value. The object generated is

returned by CONSTRUCT.

CONSTRUCT may be used to avoid further invocations of a UGF associated with a mode M by calling CONSTRUCT with M.UR as first argument and bindclass and specification value as appropriate for the second and third arguments.

4.2.1.5 An Example

Suppose we want to extend ECL to include the notion of rational numbers. For this extension we propose that RATionals be represented by a pair of INTs (STRUCT(NUM:INT, DENOM:INT)), reduced to lowest terms, with the sign carried by the numerator and a unique representation of 0 as 0/1. The following is one way of encoding the constraints in a UGF for RATs.

```
RAT\UGF <- EXPR(M\:MODE, S\:SYMBOL, F\:ANY; RAT)
  CASE[S\]
    ["SIZE"] => LIFT(CONST(RAT.UR OF 0,1), RAT);
    ["OF"] =>
      BEGIN
        DECL T:RAT.UR LIKE CONSTRUCT(RAT.UR, "OF", F\);
        DECL D:INT SHARED TDENOM;
        D = 0 => BREAK("INVALID GEN OF RATIONAL");
        DECL N:INT SHARED T.NUM;
        D LT 0 -> [) N <- -N; D <- -D (];
        DECL G:INT LIKE GCD(N, D);
        G # 1 -> [)N <- N/G; D <- D/G(];
        LIFT(T, RAT);
      END;
    /* 'S\ IS SHARED, BYVAL, LIKE, OR FROM; WE TAKE THE';
    /* 'SUPUGF FLAG TO BE TRUE, THUS WILL GET TO THIS';
    /* 'POINT ONLY IF RAT DOES NOT COVER THE MODE OF F\';
    ["SHARED"] => BREAK("SHARING FAULT IN RATIONAL GEN");
    /* 'WE ASSUME F\ IS CONVERTIBEL TO INT';
    ["BYVAL"], ["LIKE"], ["FROM"] =>
      LIFT(CONST(RAT.UR OF F\, 1), RAT);
  END;
```

The \ as the last letter of the formal parameters of the procedure is merely an attempt to avoid name conflicts which may arise during evaluation of the third argument to CONSTRUCT.

4.2.2 Conversion

One of the ways in which a user is able to define the behavior of a class of objects is through the type of conversions that are permitted from the objects. This control is acquired via a user-defined conversion function

(UCF) associated with the mode specifying the class. A conversion function is called whenever the mode of an object in hand does not match the mode desired, for example in binding actual arguments to formals in a procedure call. A user-defined conversion function takes two arguments: (1) the object in hand, and (2) the desired mode, and is expected to return an object whose mode is compatible with that provided.

Suppose we have defined a mode SHORT\INT to be a VECTOR(9, BOOL) and wish an object of mode SHORT\INT to be accepted wherever an INT would be accepted. We might then make the following definitions (taking advantage of the built-in conversion from a VECTOR(n, BOOL) to an INT for $n \leq 36$)

```
SHORT\INT <- QL("SHORT\INT*",
                UCF(SHORT\INT\UCF)) ::
                  VECTOR(9, BOOL);
SHORT\INT\UCF <-
  EXPR(V:SHORT\INT, M:MODE; M)
    CONST(INT LIKE LOWER(V))
```

Thus for example if the desired mode is ONEOF(INT, REAL), the SHORT\INT would be converted to an INT and the modes would match.

Situations in which conversion would be attempted are more fully described in section 4.3.

4.2.3 Assignment

In order to insure the validity and integrity of all objects of a certain mode, the definer of the mode must be able to get control at the time any assignment is made to an object of the mode. Thus we permit the association of a user-defined assignment function (UAF) with an extended mode. The assignment function is called exactly when the binary operator <- is encountered with left operand an object of the extended mode. In writing a UAF, the definer should expect two values to be passed to the routine, the first is the left operand to <-, the second is the right operand. To be consistent with the built-in semantics of <-, the result of the assignment routine should be the value of the left operand after the assignment has been performed. This restriction is not enforced by the ECL system, however.

Consider our example (section 4.2.1.5) defining an extension to ECL for rational numbers represented as a STRUCT(NUM:INT, DENOM:INT). Suppose also that we want a unique representation of 0 as STRUCT(0,1) but would like to permit the user to make the simple assignment A\RATIONAL <-

0. We might make the following definitions:

```
RAT <- QL("RAT*",
          UAF(RAT\UAF)) :: STRUCT(NUM:INT, DENOM:INT);
RAT\UAF <-
  EXPR(L:RAT, R:ONEOF(RAT, INT); RAT)
     BEGIN
       MD(R) = RAT => LIFT(LOWER(L) <- LOWER(R), RAT);
       MD(R) = INT => LIFT(CONST(RAT.UR OF R, 1), RAT);
       BREAK("INVALID ASSIGNMENT TO RATIONAL");
     END
```

4.2.4 Selection

Suppose we want to add general sets to ECL and represent each as a linked list of cells, as in

```
SET\CELL <-
  "SET\CELL*" ::
    STRUCT(ELEMENT:REF, NEXT:PTR("SET\CELL*"))
```

where the elements of the set are represented as REFs. In dealing with the elements of a set, however, the notion of first element, second element, etc., might be useful. We acquire this power of expression by associating a user-defined selection function (USF) with our mode for sets, as in

```
SET <- QL("SET*", USF(SET\USF)) :: PTR("SET\CELL*")
```

A user-defined selection function takes as arguments (1) the evaluated object being selected from and (2) as many selectors as are needed to perform the selection. The "." operator expects only one selector, though "[" selection will pass as many arguments to the selection function as appear between the brackets. The selectors to "[" may be bound evaluated, UNEVALuated, or LISTED. However, if "." selection is anticipated, the selection function must take its second argument evaluated or a binding fault ("MODE-BIND CLASS MISMATCH") will occur. We therefore might define SET\USF as

```
SET\USF <- EXPR(S:SET, I:ONEOF(INT,SYMBOL) BYVAL; ANY)
  BEGIN
    MD(I) = SYMBOL => LOWER(S)[I]
    I LE 0 => BREAK("SELECTION FAULT FROM SET");
    DECL DUMMY:SET.UR BYVAL LOWER(S);
      REPEAT
        DUMMY = NIL => BREAK("SELECTION FAULT FROM SET");
        I = 1 => VAL(DUMMY.ELEMENT);
        DUMMY <- DUMMY.NEXT;
        I <- I-1;
      END;
  END
```

A built-in routine called SELECT exists which works just like "[" but takes two evaluated arguments, an object of mode ANY and a selector list of mode FORM. SELECT applies selectors from the list until it is exhausted. If a USF is found, it is given as much of the selector list as it needs. If a PTR is reached whose mode has no USF, SELECT dereferences one level only.

4.2.5 Printing

Another useful facility in defining mode extension packages is that of being able to print the objects of the extended mode in a clear, clean way. A user of the package would thereby not have to see the detailed data representations. As with the other behavior attributes, we associate a user-defined print function (UPF) with an extended mode through the :: operator.

Consider our earlier example of rational numbers which we represented as STRUCTs with two components. It would surely be cleaner to have the numbers appear in the customary mathematical notation of fractions rather than the ECL notation for STRUCTs. Thus we might add to our definitions

```
QL("RAT*", UPF(RAT\UPF)) :: STRUCT(NUM:INT, DENOM:INT)
```

and

```
RAT\UPF <-
  EXPR(R:RAT, P:PORT; RAT)
    BEGIN
      PRINT(R.NUM, P);
      PRINT("/", P);
      PRINT(R.DENOM, P);
      R;
    END
```

The first argument to a user-defined print function will always be the object to be printed, the second a PORT. To be consistent with system-defined printing, the result of the UPF should be the value of the object printed (but this restriction is not enforced). For an explanation of PORTs, see section 2.15.1 of the manual.

4.2.6 Dimensions

Imagine a mode called TREE which describes "complete N-ary trees of depth D", where N and D are arbitrary dimensions of a particular TREE. A typical EL1 definition for TREE might begin as follows, where LEAF is the mode of TREE terminal nodes.

```
TREE\BASE <- PTR(LEAF, SEQ("TREE*"));
TREE <- QL("TREE*", UGF(TREE\UGF)) :: TREE\BASE;
```

Given N actual N-ary TREE's, each of depth D - 1, it is easy to see how TREE\UGF could put together a TREE of depth D from them. But suppose the programmer simply wants a D-high N-tree constructed. As things stand, CONST(TREE SIZE D, N) won't work, since the number of dimensions passed to a data generator must correspond to the mode. TREE is a pointer mode and pointer's are length resolved, i.e. they need no dimensions supplied to their generators. The CONSTruction would fail with the message "GEN ER-BAD DOPE VECTS".

For extended modes like TREE, the notion of apparent dimensions is useful. The apparent dimension vector of an object X (DIMENSIONS(X)) is a SEQ(INT) such that CONSTRUCT(MD(X), "SIZE", DIMENSIONS(X)) produces an object of the same size as X. DIMENSIONS is a built-in EL1 routine. There are two extended mode attributes which may be used to create, in cooperation with the UGF, the appearance of dimensions not necessarily reflected in the structure of the underlying mode. These attributes are: (1) a default apparent dimension vector (DADV), and (2) a user-defined dimension function (UDF).

4.2.6.1 Default Apparent Dimensions

The DADV of a mode M gives the number of dimensions required to specify a value of mode M, and it provides default dimensions for use when explicit dimensions are omitted by the programmer. Unless M has a UGF, then the number of apparent dimensions (NAD) for a mode M must match that of M's underlying mode. User-defined DADV's may contain integers in any range. However, dimensions that reach built-in data generators must be non-negative and less

than 2^{18} or an error will occur ("GEN ER-BAD DOPE VECTS").

Redefinition of a DADV for a mode is not permitted. The reason is that the mode may have been embedded in other mode definitions; if so, its DADV may have been used in building those of the embedding modes. The restriction prevents obscure inconsistencies from arising. To change the user-defined DADV of a mode, one must first delete the mode and all references to it (see section 4.4).

The DADV attribute value appearing in the left operand to :: is evaluated and coerced to the mode SEQ(INT). If the DADV keyword appears with an empty attribute form (e.g., DADV( )), the resulting mode will be given the default apparent dimensions of its base mode (the lowest mode that underlies it).

Let us consider several examples which illustrate the usefulness of DADV's. Suppose INTROW is a function which builds a SEQ(INT) from a list of components:

```
INTROW <-
  EXPR(L\:FORM LISTED; SEQ(INT))
    CONSTRUCT(SEQ(INT), "OF", L\);
```

Then if we define a class of CHARacter buffers:

```
BUFFER <- QL("BUFFER*",
                DADV(INTROW(100)) :: SEQ(CHAR);
B <- CONST(BUFFER)
```

LENGTH(B) would return the value 100.

This definition provides a more reasonable default BUFFER length than zero, but one that may still be overridden by an explicit SIZE.

```
BIG\B <- CONST(BUFFER SIZE 1000)
```

would result in LENGTH(BIG\B) equal to 1000.

If we utter

```
SQUARE\MATRIX <-
  QL("S\M*",
     DADV(INTROW(10))) :: SEQ(SEQ(REAL));
```

we get an error message ("ILL MODE DEF'N"), since the number of apparent dimensions, the NAD, of SEQ(SEQ(REAL)) is 2, and INTROW(10) has length 1. When the mode is also given a UGF, however, the definition succeeds:

```
          SQUARE\MATRIX <-
            QL("S\M*",
               DADV(INTROW(10)),
               UGF(GEN\SQUARE\MATRIX)) :: SEQ(SEQ(REAL))
```

GEN\SQUARE\MATRIX presumably takes a single dimension D and produces a D by D array.

To consider yet another example

```
-> MATRIX\BASE <- SEQ(SEQ(BOOL));
-> MATRIX <- QL("MATRIX*",
                DADV(INTROW(0, -1)),
                UGF(GEN\MATRIX)) :: MATRIX\BASE;
-> GEN\MATRIX <-
     EXPR(EM\:MODE, BC\:SYMBOL, S\:ANY; ANY)
       BEGIN
         BC\ = "SIZE" =>
           LIFT(CONST(MATRIX\BASE SIZE
                      S\[1],
                      [) S\[2] LT 0 => S\[1]; S\[2] (]),
                MATRIX);
           .
            .
             .
       END;
-> OBLONG <- CONST(MATRIX SIZE 1, 2);
-> DIMENSIONS(OBLONG)$
SEQ(1, 2)
-> SQUARE <- CONST(MATRIX SIZE 5);
-> DIMENSIONS(SQUARE)$
SEQ(5, 5)
```

Had the "SIZE" case of GEN\MATRIX been simply

```
LIFT(CONSTRUCT(MATRIX\BASE, "SIZE", S\), MATRIX)
```

then CONST(MATRIX SIZE 5) would lead to

```
GEN ER-BAD DOPE VECTS
CONSTRUCT BROKEN
1:> S\$
SEQ(5, -1)
```

since negative dimensions are out of range for built-in generators.

The DADV explicitly given a mode M by its creator overrides the dimensions that would otherwise be transferred up from its underlying mode. Therefore, when M is embedded in another definition, such as SEQ(M), the DADV for M is used by the system to produce default apparent dimensions for the new, embedding mode. Let us define the physical

dimensions of any object to be the lengths of the SEQuences it contains, taken in order of their mention in its base mode definition. (See PHYSICAL\DIMENSIONS in Appendix B.) Then it is possible to have an object of non-extended mode whose physical and apparent dimensions are different.

Let SQUARE\MATRIX be as defined as before; then

```
-> TWO\SQUARES <-
     STRUCT(M1:SQUARE\MATRIX, M2:SQUARE\MATRIX);
-> T <- CONST(TWO\SQUARES SIZE 10,5);
-> PHYSICAL\DIMENSIONS(T)$
SEQ(10, 10, 5, 5)
```

4.2.6.2 User-defined Dimension Functions

So far, the examples in this section have avoided applying DIMENSIONS to an object whose mode has a non-standard DADV. In general, given an object X of mode M, DIMENSIONS cannot tell from the UGF for M and M's DADV alone what dimension input will cause the generation function to produce an object of the same "size" as X. Indeed, the notion "size of an object" is not strictly defined by EL1; for extended data types, "size" means whatever the definer chooses it to mean.

To maintain the appearance of dimensions different from those of the underlying representation, the user may associate a dimension function (UDF) with an extended mode. When applied to a value of mode M, the UDF for M should return a SEQ(INT) of length equal to the NAD for M. M's UDF will be called whenever DIMENSIONS is explicitly applied to a value of mode M, or when M is a component mode of an object whose dimensions are being extracted. If M has no UDF, a built-in algorithm (given in Appendix B) is applied. In either case, the resulting dimension vector is made to conform to M's DADV. If it is too short, it will be padded with DADV elements. If it is too long, an error ("GEN ER-BAD DOPE VECTS") occurs.

Continuing with our earlier example for SQUARE\MATRIX

```
-> DIMENSIONS(T)$
GEN ER-BAD DOPE VECTS
DIMENSIONS BROKEN
1:> RESET;
-> SQUARE\MATRIX\ORDER <-
     EXPR(S:SQUARE\MATRIX; SEQ(INT))
       CONST(SEQ(INT) OF LENGTH(S));
-> QL("S\M*", UDF(SQUARE\MATRIX\ORDER)) :: SEQ(SEQ(REAL));
-> DIMENSIONS(T)$
SEQ(10, 5)
```

We could also extend our definitions for TREEs

```
-> QL("TREE*", UDF(TREE\UDF)) :: TREE\BASE;
-> VALID\TREE <- ONEOF(TREE);
-> TREE\UDF <-
     EXPR(T:VALID\TREE; SEQ(INT))
       BEGIN
         DECL DV:SEQ(INT) SIZE 2;
         DECL P:TREE\BASE BYVAL LOWER(T);
         MD(VAL(P)) = LEAF => DV;
         DV[2] <- LENGTH(P);
         REPEAT
           DV[1] <- DV[1] + 1;
           MD(VAL(P <- P[1])) = LEAF => DV;
         END;
       END;
```

Now if V is bound to a TREE of depth 3, each of whose non-LEAF nodes has five descendents, then

```
-> DIMENSIONS(V)$
SEQ(3, 5)
```

Even if the mode definer expects no explicit calls to DIMENSIONS, he should provide a UDF with any mode for which the built-in algorithm would not be adequate, since the system may need to make implicit dimension measurements.

Consider the expression CONST(VECTOR(3, M) OF V). After the only explicit component, V, has been coerced (BYVAL) to mode M, DIMENSIONS (and therefore the UDF for M) will be used to measure it. The resulting dimension vector is passed to M's generation function to produce a second component by "SIZE". Finally the second component is replicated to produce the third. A check insures that the physical dimensions of all the components agree, since EL1 requires complete homogeneity of VECTORs and SEQuences. (See section 4.2.1.2 and Appendix B for more detail.)

The mode MATRIX defined earlier needed no UDF even though it possessed a non-standard DADV. Of course the NAD of MATRIX equals the NAD of the underlying mode MATRIX\BASE, and the physical and apparent dimensions correspond. The next example shows another use for apparent dimensions and also illustrates why a UDF is not strictly required, even when the DADV is larger than normal.

Suppose VARYING\STRING is the class of strings whose length may vary within a maximum specified when the string is generated. (A similar data type is represented in PL\I by character arrays having the VARYING attribute.) VARYING\STRING will have two apparent dimensions: the maximum length, and a second dimension used to set the

initial length of a string.

```
-> VS\BASE <- STRUCT(TEXT:STRING, LENGTH:INT);
-> VS\DEFAULT <- 100;
-> VARYING\STRING <- QL("V\S*",
                        UGF(VS\UGF),
                        DADV(INTROW(VS\DEFAULT, 0)),
                        SUPUGF(TRUE)) :: VS\BASE;
-> VS\UGF <-
     EXPR(EM\:MODE, BC\:SYMBOL, S\:ANY; VARYING\STRING)
       BEGIN
         BC\ = "SIZE" =>
           BEGIN
             DECL T:VS\BASE SIZE S\[1];
             T.LENGTH <- S\[2];
             LIFT(T, VARYING\STRING);
           END;
         BC\ = "BYVAL" AND MD(S\) = STRING =>
           BEGIN
             ASSERT(LENGTH(S\) LE VS\DEFAULT);
             DECL T:VS\BASE SIZE VS\DEFAULT;
             FOR I TO (T.LENGTH <- LENGTH(S\))
               REPEAT T.TEXT[I] <- S\[I] END;
             LIFT(T, VARYING\STRING);
           END;
          .
           .
            .
       END;
-> NAME <- CONST(VARYING\STRING);
-> NAME.LENGTH$
0
-> NAME <- 'GWENDOLYN';
-> NAME.LENGTH$
9
-> BIG <- CONST(VARYING\STRING SIZE 1000);
-> DIMENSIONS(BIG)$
SEQ(1000, 0)
-> TEN\NULLS <- CONST(VARYING\STRING SIZE 50,10);
-> DIMENSIONS(TEN\NULLS)$
SEQ(50, 0)
-> COMMITTEE <-
     CONST(VECTOR(7, VARYING\STRING) OF 'SAM', 'HOWARD');
-> COMMITTEE[1].LENGTH$
3
-> COMMITTEE[3].LENGTH$
0
```

Since the definer of VARYING\STRING attached no UDF to it, the DIMENSIONS of any VARYING\STRING consist of the length of its TEXT component, plus a second "dimension", always zero, taken from the DADV. Though unusual, this

scheme has the nice effect that the implicit components of COMMITTEE are built with zero LENGTH fields. The more obvious approach, a UDF for VARYING\STRING such as

```
EXPR(S:VARYING\STRING; SEQ(INT))
  INTROW(LENGTH(S.TEXT), S.LENGTH)
```

would leave COMMITTEE[3].LENGTH set to the length of one of the two explicit components. Incidentally, the choice of which explicit component will provide dimensions for the first implicit component is officially undefined, although particular evaluators (including SYSTEM\GENERATE in Appendix B) may use one rule consistently.

4.3 COMPLETE SEMANTICS OF DECLARATIONS

A thorough understanding of DECLaration semantics is the key to understanding the other generation and conversion situations in EL1, since these other constructs are usually defined in terms of DECLs.

The syntax for declarations is essentially as follows

```
<Decls>                  ::= <Declaration> <Decls>
                           | <Declaration>
<Declaration>            ::= DECL <Names> :
                                  <Expected\Mode>
                                  <Generation\Spec>
<Generation\Spec>        ::= ϵ
                           | <Example\Bindclass>
                             <Example\Form>
                           | <Size\Bindclass> <Dimensions>
                           | <Component\Bindclass>
                             <Components>
<Example\Bindclass>      ::= BYVAL | LIKE | SHARED | FROM
<Size\Bindclass>         ::= SIZE
<Component\Bindclass>    ::= OF
```

where <Expected\Mode> and <Example\Form> are FORMs, <Dimensions> and <Components> are FORMs LISTED, and <Names> is a list of identifiers.

4.3.1 Declaration by Example

We first obtain an expected mode EM by evaluating the Expected\Mode form and coercing the result to mode MODE. That is,

```
EM = [) DECL EM:MODE LIKE EVAL(Expected\Mode); EM (]
```

An initial value IV is obtained by evaluating the Example\Form. The evaluator may produce EM and IV in either order.

We next produce a "generated value" GV by applying EM's UGF to IV as follows [*]:

```
GV = BEGIN
       NOT HAS\UGF(EM) OR
         COVERS(EM, MD(IV)) AND SUPUGF(EM) => IV;
       UGF(EM)(EM, Bindclass, IV)
     END
```

SUPUGF stands for "SUPpress User-defined Generation Function"; it is a boolean attribute that can be associated with any extended mode by its definer (see section 4.2.1.3). Note that the default SUPUGF value is TRUE (do suppress if possible), but that suppression only occurs when MD(IV) covers EM.

Next, obtain a "converted value" CV from GV such that CV = CONVERT(GV, EM), where CONVERT is defined as

```
CONVERT =
  EXPR(V1:ANY, EM:MODE; ANY)
    BEGIN
      COVERS(EM, MD(V1)) => V1;
      DECL V2:ANY SHARED
        BEGIN
          HAS\UCF(MD(V1)) =>
            UCF(MD(V1))(V1, EM);
          V1;
        END;
      COVERS(EM, MD(V2)) => V2;
      DECL V3:ANY SHARED SYSTEM\CONVERT(V2, EM);
      COVERS(EM, MD(V3)) => V3;
      CONVERT(ERROR("TYPE FAULT"), EM);
    END;
```

SYSTEM\CONVERT (Appendix B) controls such built-in type transfers as INT to REAL conversion, conversion among pointer modes, and so on.

Finally derive a "bound value" BV from CV in a way that depends on the Bindclass. That is, BV is set to the result of the block

---

[*] Routines such as HAS\UGF, SUPUGF, and UGF, which are not part of the base language, are described or defined in Appendix B.

```
BV =
  BEGIN
    Bindclass = "LIKE" => CV;
    Bindclass = "BYVAL" OR
      Bindclass = "FROM" => PURIFY(CV);
    (Bindclass = "SHARED") */
      LOCATION(CV) = LOCATION(IV) => CV;
    REPEAT
      DECL CV2:ANY LIKE
        CONVERT(ERROR("SHARING FAULT"), EM);
      LOCATION(CV2) = LOCATION(IV) => CV2;
    END;
  END
```

where

```
PURIFY =
  EXPR(V:ANY, EM:MODE; ANY)
    BEGIN
      IS\GENERIC\MODE(EM) =>
        CONSTRUCT(MD(V), "BYVAL", V);
      HAS\UGF(EM) => PHYSICAL\PURIFY(V);
      SYSTEM\GENERATE(EM, "BYVAL", V)
    END;
```

and PHYSICAL\PURIFY simply produces a pure copy of an object, regardless of its components or their generation functions.

The idea is to guarantee that an object of mode M will never be created, even as a component of another object, without giving UGF(M) a chance to control the generation. IF EM is generic, UGF(MD(V)) must be given its chance. Otherwise, if EM has a UGF, it will already have been invoked to produce GV. Failing these, component-mode generation functions must be permitted to tidy things up as the copy is created.

LOCATION is a (fictional) routine that returns the address of a value; it cannot be written in EL1. ERROR performs error handling rites and may return a new value, for example the value passed to the CONTinue routine after an error break.

4.3.2 Declaration by Size

Again, produce mode EM from the Expected\Mode form. Obtain an initial dimension vector IDV from the Dimensions list by the equivalent of CONSTRUCT(SEQ(INT), "OF", Dimensions). (Section 4.2.1.4 describes the CONSTRUCT function.) Again, EM and IDV may be computed in either order.

Next, turn IDV into a "proper dope vector", DV, for mode EM by the algorithm

```
DV =
  BEGIN
    DECL DV:SEQ(INT) BYVAL DADV(EM);
    FOR I TO LENGTH(IDV)
      REPEAT
        I GT LENGTH(DV) => ERROR("GEN ER-BAD DOPE VECTS");
        DV[I] <- IDV[I];
      END;
    DV;
  END;
```

DADV stands for "Default Apparent Dimension Vector". More is said about apparent dimensions in section 4.2.6.1.

Finally, obtain a bound value BV from DV by

```
BV =
  BEGIN
    HAS\UGF(EM) =>
      PURIFY(CONVERT(UGF(EM)(EM, "SIZE", DV), EM));
    SYSTEM\GENERATE(EM, "SIZE", DV);
  END;
```

4.3.3 Declaration by Default

Produce the expected mode EM as usual, and build a default dimension vector DV = DADV(EM). Then let Bindclass be "SIZE" and proceed exactly as in section 4.3.2 above.

4.3.4 Declaration by Components

Obtain an expected mode EM from the Expected\Mode form as usual, and produce a bound value as follows:

```
BV =
  BEGIN
    HAS\UGF(EM) =>
      PURIFY(CONVERT(UGF(EM)(EM,
                             Bindclass,
                             Components), EM));
    SYSTEM\GENERATE(EM, Bindclass, Components);
  END;
```

Note that, if UGF(EM) exists, the Components list is transmitted without interpretation of its elements. The built-in algorithm (see SYSTEM\GENERATE in Appendix B) begins by evaluating the elements of the given list. For SEQuences, the number of elements determines the length of

the object to be built. For VECTORs and STRUCTs, additional implicit components will be supplied or extra element values thrown away to reach the right number. Each explicit component value is coerced BYVAL to the corresponding component mode. STRUCT implicit components are generated by default. VECTOR implicit components are obtained by first generating (by SIZE) a value with the dimensions of one of the explicit components, then making as many replicas as necessary. SEQuence and VECTOR components are checked for consistency of dimensions before the completed object is returned.

While SYSTEM\GENERATE is intended to reveal (by calls to CONSTRUCT and DIMENSIONS) which mode behavior functions can be called and with what arguments, it is not intended to guarantee the order of evaluation, except that implied by data dependencies. An evaluator is free to process the components in reverse, for example, or even in parallel.

4.3.5 Replication and Naming

The bound value BV obtained by one of the bindclass-dependent methods given in 4.3.1 through 4.3.4 above becomes the binding of the first variable in the Names list of the sub-declaration. If Names contains other identifiers, bound values must be obtained for each. If the Bindclass is "LIKE" or "SHARED", all the bindings are simply shared with the original BV. Otherwise, each additional value is created by replication of a preceding value. Replication of any object X is equivalent to CONSTRUCT(MD(X), "BYVAL", X). So user-defined mode behavior functions may even play a role at this stage of the declaration process.

After all sub-declarations have been handled, names are associated with bound values for the whole declaration. The value of the declaration is the value of the last name bound.

4.4 MODE DELETION

In many ways, ECL assumes that "a mode is forever", that once it has been defined and completed, its attributes will not be changed. The compiler, for example, assumes that a mode known at compile-time will have the same definition at run-time. As has been mentioned in earlier sections, certain kinds of mode redefinition are not allowed unless the existing mode and all references to it have been deleted. An ECL package called LSMFT exists to aid the user in locating and expunging mode references.

When a mode and its tag symbol are completely unreferenced in the current environment, and there are no other features of the tag that make it worth saving (such as a top level binding, a property list, or syntactic attributes), then the garbage collector will delete the mode. LSMFT helps the user put one or more modes into this featureless state. It contains the following routines:

LM (Locate Modes) accepts a set S of modes and returns a set S* consisting of S plus all other modes whose definitions depend on one or more members of S.

LG (Locate Globals) also takes a set of modes as input. Its result is a set of names of top level variables, each of which is bound either (1) to a mode in the input set, or (2) to a value of one of the input modes, or (3) to a pointer to such a value.

FG (Flush Globals) simply deletes the top level bindings of a list of global names.

FM (Flush Modes) attempts to expunge a set of modes. It saves their names (in STRING form, to elude the garbage collector), deletes (with suitable warning messages) any features of the tags themselves that would inhibit reclamation, deletes its own input set, and calls RECLAIM. After garbage collection, FM uses the saved mode names to determine whether the corresponding modes have been deleted. It creates a new set, containing any members of the original input that have not been successfully flushed, rebinds its input variable to the new set, and also returns that set as its value.

FA (Flush All) combines calls on the routines above to try to delete references to a set of modes, and then flush the modes themselves. Using FM, FA replaces its input by the set of modes that will not go away, and also returns that set.

Here is an example of the use of these routines. Suppose a programmer has LOADed a file containing the definitions

```
RECORD <-
  <"RECORD*", UPF(PRINT\RECORD)> ::
    STRUCT(NAME:STRING, AGE:INT);

AGE\RECORDS <- CONST(SEQ(RECORD) SIZE 100);
```

After some testing, he realizes that the NAME component of RECORD should have mode PTR(STRING) so that names of different length can be entered in AGE\RECORDS. Using TECO, he corrects the file. But when he tries to reload it, an

error break occurs:

```
ILL MODE REDEF'N
:: BROKEN
1:>
```

The redefinition of an extended mode cannot change its underlying mode. To recover, the user RESETs control to top level and, with LSMFT loaded, tries to flush the old definition of RECORD:

```
-> T <- LM <RECORD>$
(SEQ("RECORD*") RECORD*)
-> FM T$
(SEQ("RECORD*") RECORD*)
-> RECORD$
RECORD*
```

The attempt fails because AGE\RECORDS and RECORD are still bound at top level, the former to an object whose mode depends on RECORD, the latter to the mode itself.

```
-> T2 <- LG T$
(AGE\RECORDS RECORD)
-> FG T2;
-> RECORD$
NOTHING
-> FM T$
NIL
-> T$
NIL
```

This time the modes are deleted, and the file can be reloaded. Note that FM takes its argument SHARED and replaces it by the mode list that is also its value.

The definition of FA is

```
FA <- EXPR(L1:MODE\LIST SHARED; MODE\LIST)
      [) L1 <- LM(L1); FG(LG(L1)); FM(L1) (];
```

So in the example just given, the command T <- FA <RECORD> would have deleted the superseded mode and all references to it. FA should be used cautiously, however. It is usually helpful to know exactly what global values are being deleted and what dependent modes must be flushed to permit a given mode to go away. Hence, as the example shows, the individual routines comprising FA are available as prefix operators. Whether FA or the individual routines are used, it is wise both to print the results and to save them in temporary variables. This habit will minimize confusion in case modes cannot be flushed although global references have been deleted.

LSMFT is not completely effective at finding data that reference modes to be deleted. It does not trace pointers from global variables beyond one level, nor does it trace the local environment at all. If feasible, the user should RESET to top level before using the package. When a group of modes refuses to disappear, a look at the dependent mode set produced by LM and at the global names returned by LG will often suggest other data that are hanging onto the modes. Failure to use distinctive mode tags can greatly increase the difficulty of flushing modes. Not only can extended mode "constants" become buried in list structure output by the parser, but the appearance of tag tokens in completely unrelated contexts can lead to "sticky" modes. For example, in the definition of FA above, "L1" is used as a formal parameter name, so the routine itself contains references to that symbol. If a programmer has used "L1" as a mode tag, e.g. L <- L1::SEQ(UENCE), it will not be possible to flush L using FA because its references prevent the tag, and hence the mode, from being RECLAIMed. The best practice is to pick mnemonic tags that are not lexical tokens and then avoid mentioning them unless absolutely necessary.

The routines in LSMFT deal with two kinds of lists, called NAME\LIST and MODE\LIST. Both are extensions of the built-in mode FORM. When a NAME\LIST is required (by FG), a valid input is a list each of whose elements is either an identifier (taken literally), a nofix expression (from which the operator name will be extracted), or some other form that yields, when evaluated, either a symbol (used directly) or a mode (whose symbolic tag is used).

When a MODE\LIST is required (by LM, LG, FM, and FA), a valid input is a list each element of which is either an identifier that tags an existing mode, or a form whose value is a mode or a symbolic mode tag (from which the mode will be derived). FM and FA take their MODE\LIST arguments SHARED so that the input variables can be reset. In these cases, the actual argument supplied must either be a MODE\LIST already (the result of an LM call, perhaps) or a FORM. That is, because of sharing, a REF of PTR(DTPR) would not do.

Under the definitions

```
NOFIX("TECO");
M <- "M*"::INT
```

the list < FOO, TECO, "FUM", M, EVAL("M"), INT, PTR(M) > coerces to a valid NAME\LIST equivalent to that produced from < "FOO", "TECO", "FUM", "M", "M*", "INT", "PTR(%"M*%")" >. The list < "M*", M, EVAL("M"), INT, PTR(REAL) > would coerce to an acceptable MODE\LIST. Each of the first three

elements represents the mode M. (The MODE\LIST produced during coercion will be two elements shorter than the input, since the UGF for MODE\LIST removes duplicates.)

A sixth routine, FLUSH\LSMFT, is provided in the package. It deletes the global bindings made by LSMFT when it is loaded, and restores the original values.

5. ECL LIST STRUCTURE EDITOR

The List Structure Editor gives an ECL user greater power and flexibility than he can possibly obtain by editing text files. Although the Editor is oriented toward operations on the internal representation used by ECL instead of external text representations, it has been designed to be used easily even by those with little knowledge of the ECL system. The List Structure Editor is especially suitable for debugging because it enables the user to make program changes quickly and without the necessity of reloading files. The List Structure Editor has a great advantage over text editors in that it contains commands which are specifically relevant to the syntax of the EL1 language. The user can easily perform complicated transformations on the program being edited, often with a single command. Because the Editor operates in the ECL program environment, it is possible to test the program being edited without exiting from the Editor. In some applications of ECL, an additional benefit of editing in core is that the structure of the material being edited is not needlessly destroyed. After an ECL debugging session with the Editor, a symbolic text file of the functions which have been edited can be generated using the function UNPARSF (see section 2.15.2.4).

5.1 GENERAL EDITOR INFORMATION

The file containing the Editor may be called in by issuing a LOADB"SYS:EDIT" command. The Editor can then be used to edit any ECL form. To start the Editor, evaluate

EDIT(<form>);

The argument <form> is taken unevaluated and then evaluated by the Editor to obtain the form to be edited, and the unevaluated argument is saved between applications of the Editor so that evaluation of

EDIT();

re-evaluates and edits the last non-defaulted argument. The Editor accumulates a list, bound to the identifier EDFNS at top level, of all the identifiers given as arguments.

5.1.1 Command Interpretation

The Editor types "*" to indicate its willingness to accept a new command. Any syntactically valid EL1 expression which is typed in will be evaluated. As in the ECL top-level environment, commands may be terminated either by a semicolon or by an alt-mode, and the Editor prints the value of commands terminated by alt-mode, but if the value of a command terminated by alt-mode is NOTHING, the Editor prints the current statement of the program instead. Some commands are evaluated specially, according to rules which differ slightly from the normal evaluation of ECL

expressions. Commands of the form <identifier>+<form> and <identifier>-<form> are evaluated as <identifier>(+<form>) and <identifier>(-<form>). A command consisting of just an <identifier> results in the evaluation of <identifier>(1) if <identifier> is the name of an Editor function with one integer argument. For example, the commands

```
L + 1;
L(1);
```

and

```
L;
```

are all equivalent. If Editor functions which do not take numeric arguments are invoked by the <identifier> command format, the Editor will type "!" and wait for the argument to be typed in. When the !-type-in mode is used with functions which have more than one argument, after each argument has been typed, the Editor will prompt again for the next entry until all the arguments have been supplied.

Assignment commands of the form

```
<id> <- <form>
```

will create a top-level binding if <id> evaluates to NOTHING.

5.1.2 Input Modes

The Editor accepts input from the teletype in two distinct modes. When the Editor prompts for input with "*", it is waiting for a command, which can be any syntactic EL1 form. When the Editor prompts with "!", it is in a special input mode used to supply text arguments to functions. When in !-mode, one or more groups, each consisting of one or more statements separated by semicolons, may be typed. The groups are separated by alt-modes. Whenever an alt-mode is typed, the statements in the group are parsed and saved. Then the Editor prompts again with "!" for another group of statements, which will be appended to the end of those previously parsed. Typing an extra alt-mode after the last statement informs the Editor that the entire argument has been entered. If a syntax error is detected in !-mode, an error message is printed and the Editor prompts with "!" for more input. Because any groups of statements which have already been parsed are saved, only the group containing the svntax error must be re-typed.

5.1.3 Editor Positioning

For convenience in explaining the operation of the Editor, certain phrases are used to describe the portion of an ECL form on which the Editor can operate at a given moment, and to explain how the Editor shifts its focus of attention within the material it is editing. The phrase 'current statement' generally refers to that subexpression of the material being edited, on which, at a given time, the

Editor operates. The current statement may be a subexpression corresponding to the EL1 syntactic unit <statement>, but in general, any subexpression of the material being edited can become the current statement. The current statement can always be printed by issuing a P(1) print command. The current statement can be unparsed by an alt-mode or T(1) command, but when it does not correspond to a complete EL1 expression, the results of unparsing may be misleading.

It is possible to position the pointer after the last statement of a block. In this position, commands to print the current statement are ignored.

When a block is entered using the B block-positioning command, the first statement of the block becomes the current statement. The body of a REPEAT expression may also be entered with a B command, and the statements within the REPEAT are treated like block statements.

The processes of entry into and exit from blocks are nested. Therefore, the phrase 'current block' refers to the last block entered by a B command, or other subexpression entered by a Z command, from which the Editor has not yet exited.

The phrase 'current pointer position' is closely related to the idea of the current statement and denotes an imaginary position between the current statement and the statement which preceeds the current statement.

5.1.4 Substitution Operators

Functions which operate on text or patterns (such as the insert command) do not evaluate their arguments, but the Editor provides facilities which modify arguments before an editing operation is applied. When the Editor is loaded, the identifiers .., $, @, and // are declared prefix operators.

Expressions in an argument of the form

.. <form>

are replaced by the value of <form>. If the entire argument is prefixed by .., it will be evaluated and the value will be used by the Editor function. The .. operator can be used to supply reserved-word identifiers in arguments to functions. The argument

abc(.. "BYVAL", 6)

becomes

abc(BYVAL, 6)

when the .. substitution operator is processed. The resulting argument could not have been supplied directly to the Editor, because it is syntactically invalid. Although

this example is not intended to represent typical usage, the importance of supplying reserved-word identifiers in Editor arguments will become apparent in the section on pattern matching.

The Editor provides registers for temporary storage of forms, called Q-registers after TECO. Expressions in arguments of the form

$ <q>

where <q> is any identifier, are replaced by a copy of the contents of register <q>. Section 5.3 gives more information about Q-registers.

The identifier ! is a special operator of no arguments. When the operator ! is encountered in an argument, the Editor enters !-input mode and waits for input from the teletype. When the input is completed, it is checked for the presence of substitution operators, and if any are found, they are interpreted. The resulting form is substituted into the original argument, replacing the occurrence of !.

The @ operator works somewhat differently from the other substitution operators. Expressions of the form

@ <form>

cause <form>, which must be non-atomic (with one exception, explained later), to be concatenated as a list between the right and left contexts of the expression @ <form>. The argument

BEGIN @ f(x.s, 2); TRUE END

becomes

BEGIN f; x.s; 2; TRUE END

when the @ operator is processed. If there is no right or left context, as in

@ (a + b)

the @ operator will not be processed unless the argument is given to the insert function or is to be inserted by the pattern matcher in an R command, in which case the context used is the context of the point of insertion. (The pattern matcher will not spread replacements at the point of insertion in commands other than the R command.)

Substitution operators can be composed by specifying a substitution within the argument of a substitution operator. Substitutions are always executed in order from innermost to outermost, making nested substitutions possible. The argument

.. $ reg

is processed by first substituting a copy of the contents of Q-register reg for $ reg, and then replacing the entire argument by the result of evaluating the copy of the contents of the register.

It is now possible to explain the exception to the rule that the argument to @ must be non-atomic. If the result of argument processing in the argument to @ is a Q-register name <q>, the @ operator will use a copy of the contents of register <q>. Thus

@ <q>

is interpreted as a shorthand for

@ $ <q>

The operator // simply quotes its argument. Expressions of the form

// <form>

are replaced by <form> without interpretation of substitution operators in <form>.

Substitution operators never alter the list structure of the argument being modified. All substitutions are performed by creating a copy of the list structure of the original argument, with appropriate changes. Even if no substitutions are required, a copy of the original argument is made to insure that later editing changes will not alter the original argument (this prevents difficulties in macro-editor functions). Note that when Q-registers are substituted into an argument by $, a copy of the register contents is made, but when evaluated expressions are substituted by .., the result of the evaluation is not copied. Substitutions are not permitted in numeric <n> or <q> identifier arguments.

5.2 BASIC COMMANDS

5.2.1 Printing

T(<n>);

The notation "<n>", when used to represent the argument to an Editor function, indicates that any ECL expression which evaluates to an integer may be given as the argument to the function. T unparses onto the user's terminal n statements beginning with the current statement. If n is 0, prints all statements in current block.

PQ; <q>$
PQ(<q>);

The notation "<q>" is used to indicate that the argument to a function is an identifier used as Q-register name. PQ unparses the contents of register q.

P(<n>);

Like T except statements are printed by the ECL PRINT routine instead of UNPARSE.

5.2.1.1 Depth Limiting

T ... <int>;
P ... <int>;

The identifiers T and P, followed by the infix operator ... and an integer, limit the depth of printing for the current statement; the printlevel is temporarily set to <int> and then the current statement is printed.

T / <int>;
P / <int>;

These two commands cause depth-limited printing of the current block.

5.2.2 Insertion

I; <statement 1>; <statement 2>; <statement m>$
I(<form 1>, <form 2>, <form m>);

The I command inserts statements into the current block. Statements are inserted at the current pointer position (before current statement). If the pointer is positioned after the last statement, new statements will be inserted at the end of the block. After insertion, the pointer is positioned after the last statement inserted. When I is called with functional arguments, they are taken unevaluated and each argument is inserted as a new statement.

```
I(a, b, c);
```

inserts three statements a; b; c; at the current pointer position. The following command

```
I($ <q>);
```

inserts the contents of register <q> as a single statement, while

```
I(@ <q>);
```

causes each list element of the form in register <q> to become a new statement.

5.2.3 Deletion

K(<n>);

If n is positive, delete next n statements of current block. If n is negative, delete preceding n statements of current block. If the last statement in the current block is deleted, the pointer is positioned after the remaining statements.

The following protocol contains examples of the I, K and printing commands. The B command is explained in the next section. Its effect in the protocol is to make the first statement of the block the current statement. The ZJ command moves the pointer to the end of the current block. Text typed by the user has been underlined to distinguish it from text typed by the Editor.

```
*$
[) a; b; c; d; e (];
*B;
*I;
!new(statement); qq$  ! $
*$
a;
*B - 0;
*T(99);
new(statement);
qq;
a;
b;
c;
d;
e;
*ZJ;
*I;
! f$  ! g$  ! $
*T(-99);
new(statement);
qq;
a;
b;
c;
d;
e;
f;
g;
*$
*K - 3;
*T - 50;
new(statement);
qq;
a;
b;
c;
d;
*$
*
```

The ! substitution operator is very useful when making long insertions, because it simplifies the nesting of complex statements and limits the consequences of a syntax error. In the following example, the user types ! as the first argument to a function to be able to enter that

argument separately. Then, when inserting a block, the @ spreading operator is used in conjunction with ! to allow a sequence of statements to be inserted. When a svntax error occurs in the next statement, only that statement must be repeated.

```
* I;
! F(!, TRUE)$  ! $
! BEGIN a = 3 => p.j; p[x] END$  ! $
* T-1;
f([) a = 3 => p.j; p[x] (], TRUE);
* I;
! BEGIN DECL N:INT SHARED e.g; @ !  END$  ! $
! f(n, 0 => TRUE$

 [) f ( n , 0  ???  => TRUE ;
  ?SYNTAX - REENTER LAST GROUP OF STATEMENTS
! f(n, 0) => TRUE$  ! n * q + 55$  ! $
* T-2;
f([) a = 3 => p.j; p[x] (], TRUE);
[) DECL N:INT SHARED e.g; f(n, 0) => TRUE; n * q + 55 (];
*
```

5.2.4 Positioning Commands

L(<n>);

If n is positive, moves pointer forward over n statements in current block. If n is negative, moves pointer back n statements in current block. If n is positive and too big, pointer is positioned after the last statement of the block. If n is zero, does nothing.

ZJ;

The ZJ command positions the pointer at the end of the current block, after the last statement.

B(<n>);

The B command is used to move the pointer into and out of blocks, CASE bodies, and REPEAT bodies. If n is positive, B searches for a point in the program n blocks deeper into the block structure than the current pointer position. Search starts from the current pointer position, and proceeds in print order. If search succeeds, the pointer is positioned before the first statement of the last block or REPEAT expression entered. If the search fails, the pointer is returned to the position it pointed to before the B command was given. If n is negative, B returns to a point -n blocks higher in the program. If n is negative and too large, the pointer is returned to the outermost block. If n is zero, B positions the pointer before the first statement in current block.

J;

J returns the statement pointer to the position it was in before the last command (including J) which changed the current block position. Successive J commands flip the pointer back and forth between statements in two different blocks.

Note: The Z command and integer commands are directly related to the internal list structure of EL1 forms. New users are advised to skip the descriptions of these commands and to rely mainly on L, B, and search commands for positioning.

Z(<q>);
Z;
Z(<non-atomic pattern>);

The command Z(<q>) positions the Editor at the top of the form in register q. The command Z; positions the Editor at the top of (immediately inside of) the current statement. If Z is given a non-atomic argument, it is treated as a pattern to be matched against the current statement. The subexpression of the current statement which matches the pattern is entered. (Pattern matching is explained in section 5.4.)

In both types of Z command, the first list element of a form (the form in register q for Z(<q>); or the current statement in the case of Z;) becomes the current statement. Other list elements of the form can be reached by L commands, and in general, the Editor handles any form entered by a Z command as though it had entered a block and the list elements of the form are handled as though they were block statements.

The B(<n>) command with n negative is used to return from entries into forms by Z; commands and Z commands with patterns. The processes of entry by positive n B commands and Z commands are nested and negative B commands provide a common exit mechanism. When a positive B or Z command is given, the current statement in the current block is saved at the top of a push-down list, and negative B commands restore the current statement from the top of the push-down list.

Z(<q>) commands set a mark on the push-down list which prevents B commands and search commands from exiting the Q-register. Marked expressions are exited with the command UP. The MARK command described below makes it possible to mark expressions other than Q-registers, to permit the range of searches to be restricted.

Warning: Z commands, except those with pattern arguments, temporarily modify the internal list structure of the expression entered. These modifications are normally invisible to the user, and they are reversed when the expression is exited. Expressions entered by a Z command, or forms containing subexpressions entered by a Z command, will not evaluate properly unless the evaluation is requested through an E command (see section 5.2.7). The EXIT command should always be used to leave the Editor, because it insures that all expressions will be restored.

UP(<n>);

UP pops the push-down list until the nth marked expression from the top, not counting the current level, is reached, or the push-down list becomes empty. When UP is used at the level of a marked expression, that expression is exited and then the push-down list is popped until the nth mark above the original level is reached.
UP(0) is defined as:

$$\lim_{n \to 0} B(-1/n)$$

MARK(<n>);

If n = 1, the current level of the push-down list is marked. If n = 0, the current level is unmarked. An error is returned if MARK is used at the top level of an expression in a Q-register.

It is possible to delete all the statements of a block or all the list elements of an expression entered with Z, by use of the K command. When attempting to exit such a block or list with a B command, the Editor will enter a break. Continuing from the break will cause the expression being exited to become a one-element list of the NIL pointer.

The following sample protocol illustrates the use of the B and L commands.

```
*$
EXPR( )
   [) a; b; [) c; [) d; e (]; [) f; g (]; h (]; i; j; k (];
*B;
*$
a;
*B + 2;
*$
d;
*B - 1;
*$
[) d; e (];
*L;
*B;
```

```
*$
f;
*B - 2;
*$
[) c; [) d; e (]; [) f; g (]; h (];
*L + 99;
*$
*L - 1;
*$
k;
*
```

The next protocol demonstrates the use of the Z command. First, the two statements being edited are unparsed. Then the P command is used to print the statements without unparsing. When the first Z command is given, the first list element of the current statement becomes the current statement. Then the L command is used to shift the Editor two positions to the right in the statement. Then the user jumps into the block in the second statement, moves right one line, and uses Z to position within the current statement. A B command moves the pointer to its original position outside of both blocks, and the second statement of the inner block is re-entered by a Z command with a pattern argument. Note that infix operator expressions are represented internally as if they were common procedure calls. For example:

```
                    a + 1
```

is stored in the same way as the procedure call

```
                   +(a, 1)
```

```
*$
BEGIN
   a <- c AND switch(arg1, arg2);
   waldo("zilch", [) q => a; other(m) (]);
END;
*B;
*P(99);
(<- a (AND c (switch arg1 arg2)));
(waldo "zilch" (BEGIN (=> q a) (other m)));
*$
a <- c  AND switch(arg1, arg2);
*Z;
*$
<-;
*L + 2;
*P;
(AND c (switch arg1 arg2));
*$
c AND switch(arg1, arg2);
*B - 1;
*L;
*B;
*$
```

```
q => a;
*L;
*$
other(m);
*P;
(other m);
*Z;
*P(50);
other;
m;
*B - 3;
*Z(... other);
*$
other;

<int>;
```

Integers are interpreted as positioning commands when they are given as typed input to the Editor. A positive value causes the <int> th list element of the current statement to become the current statement. A negative value causes the <int> th list element from the end of the current statement to become the current statement. For example, the command

```
- 1;
```

jumps to the last list element of the current statement. The effect of a non-zero integer is equivalent to a Z; command followed by an L command. The command

```
0;
```

is equivalent to

```
B(- 1);
```

5.2.5 Search Commands

Search commands are executed as a two-step process. The first step is a pattern search which attempts to locate an expression of the program which matches a pattern. If the pattern search is successful, the Editor statement pointer is re-positioned in front of the most directly enclosing statement which contains the expression located during the first stage. If the first stage is unsuccessful, the pointer position is not changed by the search command, and an error message is printed.

```
S; <pattern>$
S(<pattern>);
```

Beginning in the current statement, S searches the program in print order to find the first expression which matches the pattern. Whenever the end of a block or statement entered by a Z command is reached, the search continues in the remainder of the enclosing block, and the search is discontinued only when the end of the program is

reached, or if the statement pointer is positioned inside a Q-register, when the end of the expression in the register is reached. In the second stage of the command, the B and L functions are used to move the pointer to the most directly enclosing statement containing the located expression. (If the expression is a statement, the pointer is positioned at the expression itself.)

NS; <n>$ <pattern>$
NS(<n>, <pattern>);

NS is like S except that the first phase of the search continues until the nth expression in print order which matches the pattern is found. If fewer than n matching expressions are encountered, the search fails and the statement pointer is not moved by the command.

The following protocol illustrates the search commands. The pattern supplied to S in response to its ! prompt instructs the pattern matcher to locate exit conditional statements.

```
*$
EXPR(f:SYMBOL)
 BEGIN
  DECL last:INT;
  DECL seen:snp BYVAL ALLOC(sss);
  piport <- OPEN(f);
  l <- und();
  maplist(l,
          EXPR(x:FORM; NONE)
           BEGIN
            DECL evx:ANY LIKE EVAL(PFORM(x, COPORT));
            DECL obj:ANY LIKE und();
            PRINT(ff, COPORT);
            MD(evx) = NONE => x.TLB <- ALLOC(ANY BYVAL obj);
            evx <- obj;
           END);
  CLOSE(piport);
  piport <- pi;
 END;
*B;
*S(CLOSE)$
CLOSE(piport);
*B();
*S(obj)$
DECL obj:ANY LIKE und();
*NS(3, <-)$
piport <- pi;
*B - 1;
*S;
! ? =>\ $   ! $
*$
MD(evx) = NONE => x.TLB <- ALLOC(ANY BYVAL obj);
```

*

5.2.6 EXIT

EXIT;

The EXIT command returns control to the calling environment of the Editor.

5.2.7 Testing a Function During Editing

E(<form>);
E; <form>$

The E command allows the function being edited to be tested without exiting from the Editor. E removes the Editor's temporary internal structural modifications from the expression being edited, and then evaluates <form>. While it is possible to give any form as a direct command to the Editor, commands which attempt to access the expression being edited may evaluate improperly unless the evaluation is requested through an E command.

5.3 Q-REGISTER COMMANDS

Q-registers are used for temporary storage of forms. Any EL1 identifier can be used as a Q-reqister name. When the Editor is entered, no Q-registers are in use. New registers are created by Editor commands which attempt to store information in a previously non-existent Q-register.

XFQ; <q>$ <form>$
XFQ(<q>, <form>);

Stores the <form> in register q. Both arguments to XFQ are taken unevaluated, but substitution operators are interpreted in the <form> argument.

XZQ; <q>$ <form>$
XZQ(<q>, <form>);

Equivalent to the sequence:

```
XFQ(<q>, <form>);
Z(<q>);
```

XLQ; <q>$ <n>$
XLQ(<q>, <n>);

For positive or negative integers n, stores a list of n statements from the current pointer position into register

q. A list of all the statements in the current block is stored if n is zero. XLQ does not move the pointer position or alter the program being edited.

XQ; <q>$ <statement 1>; <statement 2>; <statement m>$
XQ(<q>); <statement 1>; <statement 2>; <statement m>$

Loads a sequence of statements into register q.

WQ(<q>);

Register q ceases to be a Q-register.

WQ;

Deletes all Q-registers.

Q; <q>$
Q(<q>);

A function which returns as its value the contents of register q.

In the following protocol, the XLQ and insert commands are used to re-order the second and third statements of the block.

```
*$
EXPR(f:FORM; FORM)
   BEGIN
      DECL a:FORM;
      MD(VAL(f)) # ATOM => f;
      (a <- getx(f, "xyz")) = NIL => convert(f);
      nlistp(a) => a;
      a.CAR;
   END;
*B;
*L + 2;
*XLQ(line, -1);
*PQ(line);
MD(VAL(F)) # ATOM => F
*K - 1;
*L;
*$
nlistp(a) => a;
*I($ line);
*B - 0;
*T(9);
DECL a:FORM;
(a <- getx(f, "xyz")) = NIL => convert(f);
MD(VAL(f)) # ATOM => f;
nlistp(a) => a;
a.CAR;
*
```

5.3.1 Sharing of List Structure by Q-Register Commands

List structure set into a Q-register by the pattern matcher is shared with the current statement, so that editing changes in either the register or the statement will affect both. The XLQ command copies the dotted pairs connecting the statements entered, but does not copy the statements themselves. The $ substitution operator always copies the register contents before insertion into the modified argument.

5.4 PATTERN MATCHING

The pattern-matching commands of the Editor allow the user to locate, save, modify, and replace program subexpressions without extensive use of positioning commands and without requiring detailed knowledge of the internal representation of EL1 forms. The pattern matcher is invoked by the Editor commands which take pattern arguments. Any EL1 form or any sequence of EL1 statements can be used as a pattern.

The pattern-matching apparatus of the Editor consists of two components: the pattern test and the pattern search. The pattern test may be thought of as a predicate which when given a pattern and a subexpression of the program being edited, decides whether the subexpression matches the pattern. Editor patterns are defined so that there must be an essentially one-to-one correspondence between the pattern and the program subexpression for the two to match. When the pattern test succeeds, two types of special actions may be taken, if they are specified by the pattern. First, the pattern may cause certain subexpressions of the program expression which satisfied the pattern test to be saved in Q-registers. After the register loading operations are completed, certain subexpressions of the matching program expression may be replaced by other expressions specified by the pattern.

The pattern search is a procedure which attempts to find an expression in the current statement which satisfies the pattern test. The search first tests the entire current statement, and then scans in print order over the current statement, trying every possible subexpression until a match is found. The pattern search is distinct from the Editor's search comands, and does not move the statement pointer while searching. An expression will be said to 'match a pattern' if it satisfies the pattern test for the pattern. An expression will 'contain a matching subexpression' if the pattern search can locate a subexpression which satisfies the pattern test.

Most occurrences of substitution operators in a pattern are executed before the pattern is given to the pattern matcher. The only exception is that some patterns have subexpressions known as replacement expressions, which are intended to be inserted into the program at a place to be determined by the pattern matcher. Substitutions are performed in replacement expressions after pattern matching.

Forms which do not point to dotted pairs match if the objects they point to are equal (as determined by ECL EQUAL). In general, list structure is matched by matching of consecutive items in the pattern with the corresponding points in the form being tested. For example, the pattern

```
a <- thing(1, 2)
```

matches the second argument in the procedure call

```
f(q, a <- thing(1, 2), [] z => TRUE; FALSE (])
```

The prefix operator ? defines patterns which match general syntactic categories. ?B, for example, matches any EL1 block. The complete list of patterns which can be defined by ? is

?A	ALLOC expression
?AT	atomic form
?B	block
?C	CONST expression
?D	declaration statement
?E	EXPR
?F	any form
?CA	CASE
?I	iteration
?ID	identifier
?L	non-atomic form
?S	[selection
?SQ	. selection
? =>\#	exit conditional
?<mode>	any constant of type <mode>

Referring to the previous example, the pattern

```
a <- ?F
```

also matches the second argument in

```
f(q, a <- thing(1, 2), [] z=> TRUE; FALSE (])
```

and the pattern

```
?L; ?B
```

matches the list consisting of the second and third arguments to f.

When the Editor is loaded, the identifiers <* and *> are made prefix operators and infix operators with priority 5. <* and *> may be thought of as 'assignment' operators. Both take a pattern as their left-hand argument, and form patterns which match anything that matches their left-hand argument. But when the pattern test succeeds, <* and *>

have special side effects: *> loads a Q-register and <* alters the program being edited.

The *> operator takes an identifier as its right-hand operand. The pattern

```
<pat> *> <q>
```

saves the program expression which matched <pat> in register <q>. The pattern

```
<pat> <* <replacement>
```

replaces the program expression which matched <pat> with the form <replacement>. The replacement expression can be any EL1 form, and if it contains substitution operators, they are interpreted after the pattern test has succeeded and after Q-registers have been loaded by occurrences of the *> operator in the pattern. The pattern

```
?STRING *> s1
```

saves a string in register s1, and

```
(?INT *> u) <* ..($u + 1)
```

locates an integer, saves it in u, and replaces it by the original value plus one.

The <* and *> operators are used as prefix operators only for the purpose of abbreviating two commonly used patterns. The pattern

```
<* <replacement>
```

is equivalent to

```
?F <* <replacement>
```

and

```
*> <q>
```

is equivalent to

```
?F *> <q>
```

The identifier ... has special significance in patterns and may be used either as an identifier or as a prefix operator. Patterns of the form

```
... <pat>
```

match any list whose first list element matches <pat>. When used in a pattern as the last list element of the pattern or as the last list element of a sublist of the pattern, the identifier ... causes the pattern test to bypass the remaining list elements in the expression being tested. The pattern

```
b + 17; ...
```

matches from the second argument through the third argument in the procedure call

```
func(a, b + 17, d + b * 5)
```

and is equivalent to the pattern

```
... (b + 17)
```

The pattern

```
... (+)
```

matches

```
b + 17
```

because of the internal representation of infix expressions.

The pattern matcher has two operators which perform unanchored pattern matching. They are >> and **, both infix operators with priority 10. Patterns of the form

** (<pat1>, <pat2>, . . . , <patn>)

match any expression which matches <pat1> and has list elements which match <pat2> to <patn>. (The order of <pat2> to <patn> is not significant.) Thus

?I ** .. "TO"

matches iterations with TO clauses and

?B ** ?D

matches blocks with declaration statements. Patterns of the form

>> (<pat1>, <pat2>, . . . , <patn>)

match any expression which matches <pat1> and contains proper subexpressions which match <pat2> to <patn>. The patterns <pat2> to <patn> are not matched against the entire expression which matches <pat1> but they are matched against all its subexpressions. The >> operator is useful for specifying a context in which to make an editing change. If the current statement is

NOT f(CAR(CAR(p.x)), CDR(CAR(p.x)))

then the second p can be changed to the contents of register a1 by processing the pattern

CDR(?F) >> (p <* $a1)

Patterns of the form

<pat>

match any expression which does not match <pat>. When an expression prefixed by ## is used as an argument other than the first argument to ** or >>, no list element or subexpression, respectively, of the expression which matched the first argument, may match the expression prefixed by ##. The pattern

?B >> ## ?B

matches any block which does not contain another block and the pattern

?I ** ## .. "BY"

matches any iteration which does not have an increment specification.

5.4.1 Match Replacement Commands

R; <pattern>$
R(<pattern>);

R attempts to match the pattern with some subexpression of the current statement. Q-registers are loaded and replacements are executed as directed by the pattern. The value of the function R is TRUE if the pattern search succeeds, and FALSE otherwise. If the pattern fails in an R; command, an error message is printed.

The @ spreading operator can be used in an R command to add statements to a block or new list elements to a list. Assume the current statement is of the form

```
a <- QL(f1, f2, . . . , fn)
```

where f1 to fn are arbitrary forms. If register b contains a form, the command

```
R(QL <* @ QL($ b));
```

adds it as the first argument of QL. Since the replacement has insufficient syntactic context for the @ operator to be processed, spreading occurs after the replacement has been inserted into the program. After processing the pattern, the statement would be

```
a <- QL(f0, f1, f2, . . . , fn)
```

where f0 is a copy of the contents of register b.

```
<pattern> <* <replacement>;
<pattern> *> <q>
```

The <* and *> operators may be used directly in editor commands. This usage is interpreted as an abbreviation for an R command. For example, the command

```
x <* y;
```

changes the first x in the current statement to y.

```
RA; <pattern>$
RA(<pattern>, <routine>);
```

RA is similar to R but it maps the pattern over the entire current statement in print order. Replacements specified by the pattern are executed wherever the pattern matches a subexpression of the current statement. The routine argument is optional. Immediately after each successful match, <routine> is called with one argument: the list structure form which matched the pattern. The routine can be an Editor macro and is permitted to alter the list structure of its argument. The value of RA is TRUE if the pattern matches at least one expression in the current statement, and FALSE otherwise.

```
RAF; <pattern>$
RAF(<pattern>, <routine>);
```

RAF is similar to RA but it does not attempt to match the pattern against subexpressions of an expression which has matched the pattern. This is necessary in cases where the pattern modifies the matching expression and creates new subexpressions which match the pattern. For example, to change all instances of

```
id9
```

to

```
id9.p
```

use the command

```
RAF(id9 <* id9.p);
```

The following protocol contains examples of the replacement commands. First the bind class LIKE in the EXPR heading, which appears in the list structure of the EXPR but is not printed by UNPARSE is changed to BYVAL. Then R is used to change q to q * 5. All occurrences of 1 are next changed to 111. The fourth command demonstrates the ! substitution operator. The initial argument to R only indicates that the pattern must match an assignment expression, that the pattern for the left-hand argument will be specified later, and that any right-hand argument will match. Then the Editor prompts for input to complete the pattern. This time, the user specifies that the left-hand argument can be any form, that it should be saved in register x, and that it will be replaced, but the replacement expression will be supplied later. The Editor has a complete pattern and finds that it does match a part of the program. When it prompts for input the second time, it has already done the pattern matching and loaded register x. The entire command could have been specified in the initial pattern, but using ! to break the input into smaller pieces simplifies the planning needed to construct patterns.

```
*$
EXPR(q:INT)
   BEGIN
      DECL r:BOOL LIKE l[q];
      FOR n FROM z(q, 5) REPEAT l[n] <- w[q <- q - 1].a END;
      r => look(elsewhere);
      xx(l);
   END;
*P;
(EXPR ((q INT LIKE)) NONE (BEGIN (DECL ((r) BOOL LIKE ([ l
 q))) (FOR FOR n FROM (z q 5) REPEAT (<- ([ l n) (. ([ w (<-
 q (- q 1))) a))) (=> r (look elsewhere)) (xx l)));
*R(.. "LIKE" <* .. "BYVAL");
*$
EXPR(q:INT BYVAL)
   BEGIN
      DECL r:BOOL LIKE l[q];
      FOR n FROM z(q, 5) REPEAT l[n] <- w[q <- q - 1].a END;
      r => look(elsewhere);
      xx(l);.
   END;
*R( l[q <* (q*5)] );
*$
EXPR(q:INT BYVAL; NONE)
   BEGIN
      DECL r:BOOL LIKE l[q * 5];
      FOR n FROM z(q, 5) REPEAT l[n] <- w[q <- q - 1].a END;
      r => look(elsewhere);
      xx(l);
```

```
   END;
*RA(l <* lll);
*$
EXPR(q:INT BYVAL; NONE)
   BEGIN
      DECL r:BOOL LIKE lll[q * 5];
      FOR n FROM z(q, 5)
         REPEAT lll[n] <- w[q <- q - 1].a END;
      r => look(elsewhere);
      xx(lll);
   END;
*R(! <- ...);
! (*> x) <* ! $   ! $
! s($x).a $   ! $
*$
EXPR(q:INT BYVAL; NONE)
   BEGIN
      DECL r:BOOL LIKE lll[q * 5];
      FOR n FROM z(q, 5)
         REPEAT s(lll[n]).a <- w[q <- q - 1].a END;
      NOT r => look(elsewhere);
      xx(lll);
   END;
*
```

RA can be used to perform macro expansions. The LISP function CONS, defined in ECL by the assignment

```
CONS <- EXPR(A:FORM, B:FORM; FORM) ALLOC(DTPR OF A, B)
```

serves as a simple example. All expressions of the form

```
CONS(<form1>, <form2>)
```

where <form1> and <form2> are arbitrary EL1 expressions, can be expanded to

```
ALLOC(DTPR OF <form1>, <form2>)
```

by the command

```
RA(CONS(*>A, *>B) <* ALLOC(DTPR OF $A, $B))
```

```
MBD; <form>$
MBD(<form>);
```

MBD alters the current statement by replacing it with a new expression containing one or more subexpressions which are copies of the original statement. Typical uses of MBD include enclosing a statement in a block, and imbedding the current statement as an argument in a function application. The argument to MBD can be any form. The form is modified by the normal substitution operators, but additionally, all occurrences of the identifier @@ are replaced by copies of the current statement. Then the current statement is deleted and the new expression is inserted in its place. Thus

```
MBD(BEGIN @@ END)
```

encloses the current statement in a block.

```
XTR <xpat>$
XTR(<xpat>);
```

The XTR command replaces the current statement by one of its subexpressions. The argument to XTR is a pattern which is processed specially as follows. If xpat contains the identifier @@, then @@ may match any form and the expression which matches @@ becomes the current statement. If xpat contains an expression of the form

```
<@ <pat>
```

then this expression matches anything which matches pat, and the matching expression becomes the current statement. If xpat contains neither @@ nor <@, then it is a regular pattern, and the expression it matches will be extracted. For example, if the current statement is

```
[) d(5) (] AND b.c -> d([) a => NIL; %b (])
```

then

```
XTR(d(@@));
```

changes the current statement to

```
5
```

but

```
XTR(d(<@ ?B));
```

extracts the block

```
[) a => NIL; %b (]
```

while

```
XTR(?B);
```

extracts the first block

```
[) d(5) (]
```

5.5 ERROR DISCOVERY

When the Editor detects an error in a command, it terminates execution of the command, prints the message "huh?" and accepts a new command. Of the commands which take numeric arguments <n>, only NS and B give an error message if there is not a sufficient number of statements to permit execution of the command. The R, RA, RAF, MBD, XTR, and Z; commands give an error message if the pointer is positioned after the last statement. Print commands P and T; commands given with the pointer positioned after the last statement have no effect on the Editor. In general, a command containing an error which is detected by the Editor will not cause any change in the status of the Editor or the material being edited. In particular, positioning and searching commands have no effect on the pointer position if they fail. A pattern-matching command which fails can, however, alter the contents of Q-registers referenced by the *> operator. This feature can be useful for diagnostic purposes; after execution of a pattern-matching command which fails, Q-registers will contain whatever they were last matched with before pattern matching was stopped.

If execution of the Editor is terminated abnormally, immediately evaluate

```
EDIT();
```

and the Editor will reposition at the top of the last form it was called to edit and will accept further editing commands or the EXIT command. Q-register values will not be lost if this procedure is used.

If ECL enters a break during editing, evaluation of

```
EDCONT;
```

will cause the Editor to accept a new command.

The Editor aborts command execution when it detects an error by evaluating

```
RETURN(NOTHING, "EDERROR")
```

User-defined error-recovery procedures can be implemented with << by evaluating Editor commands in a marked expression.

5.5.1 Correcting Patterns

It is often the case that when a pattern-matching command fails, the reason for its failure can be quickly corrected by other commands or by a slight change in the pattern. The Editor temporarily saves all pattern arguments to enable them to be re-used or changed without being re-typed. The identifiers RPAT, RAPAT, SPAT, XPAT, and ZPAT are bound variables of mode FORM, local to EDIT. Each pattern argument to the function R is assigned to RPAT after all substitutions in the pattern have been processed (but before substitutions in replacement expressions). RAPAT and ZPAT save pattern arguments to RA and Z, respectively, XPAT saves patterns from XTR, and SPAT saves patterns from S and NS commands. Thus

```
S(.. RPAT)
```

searches for an expression matching the last pattern argument given to R and

```
R(.. ZPAT <* $a)
```

replaces the expression matching the last pattern argument to Z by the contents of register a. To edit a pattern, load it into a register and enter the register with XZQ, and make editing changes. The register must be exited with a B command before the pattern can be re-applied.

5.6 SPECIAL TECHNIQUES

5.6.1 Compress and Uncompress

```
C(<q>);
C(<n>);
```

The command C(<q>); adds a level of list structure to the contents of register q by getting the contents, and setting the register to a new one-element list made from the old contents. The previous contents of the register becomes the CAR of the contents of the register.
The command C(<n>); for positive or negative n, removes n statements from the current block and replaces them by a single statement which is a list of the n old statements. After executing the C(<n>); command, the pointer is positioned in front of the new statement.

U(<q>);
U;

The command U(<q>); removes the top level of list structure from the contents of register q. If the register does not contain list structure, or if the CDR of the contents of the register is not NIL, the uncompress operation is not performed.

The U; command deletes the current statement and replaces it with a sequence of statements which are the successive list elements of the old statement. After execution of the U; command, the pointer is positioned in front of the first new statement.

Several examples will explain the usefulness of the U and C commands. Assume the current statement is a block, and it is necessary to eliminate the block by incorporating its statements directly into the current block in place of the current statement. First give a U; command. This has the effect of concatenating the statements from the block at the current statement directly into the current block. After the U; command, the current statement will be the atom BEGIN, because BEGIN is the first list element of an ECL block. At this point, giving a K; command will complete the editing task. For an example of the C; command, assume that the current statement is a form f, and it is necessary to change the current statement into a procedure call with f as one of its arguments. This can be accomplished by inserting the procedure name into the current block before f, inserting any other arguments before or after f in the order in which they should appear in the procedure call, positioning the pointer at the statement which is the procedure name, and giving the command C(<n>+1); where n is the number of arguments in the procedure call.

5.6.2 Macro Editor Functions

In accordance with the philosophy of extensible languages in general and ECL in particular, the Editor admits a wide range of user defined functional extensions. The basic principle of Editor extensibility is that the

Editor's commands are bound identifiers which can be referenced as free variables by a form which is evaluated in the environment of the Editor. It should be noted that by calling Editor functions directly, the user bypasses the Editor's command interpreter, so that the special evaluation conventions for commands do not apply. The following example is intended only to suggest possible techniques. The top-level assignment

```
psearch <-
   EXPR(pred:PROC(FORM; BOOL))
      BEGIN
         DECL exp:FORM;
         << (REPEAT
                RA(? F,
                 EXPR(F:FORM)
                      pred(exp <- f) -> RETURN());
                L(1);
             END);
         S(.. exp);
      END;
```

defines a function which may be used as an Editor macro. The command psearch searches in the current block for an expression with list structure such that pred is satisfied. Then, it positions the Editor at the smallest statement containing this expression. If no expression satisfies pred, then RA will be attempted at a non-existent statement after the last statement, and an error return will be given.

5.6.3 Initialization Function

The function EDIT takes an optional second argument of mode ROUTINE. If EDIT is called with the second argument not defaulted, the routine is called in the environment of the Editor before commands are accepted. This feature can be used either to perform special initialization before editing, or to implement completely automatic editing functions which enter and exit the Editor without user intervention.

5.6.4 STATEMENT Command

```
STATEMENT;
```

The command STATEMENT(); returns as its result the current statement of the program being edited. This value may be used in user macro extensions to the Editor.

5.7 INITEDIT AND FLUSHEDIT

INITEDIT and FLUSHEDIT are defined at top-level when the Editor is loaded. The routine INITEDIT establishes the parsing fixities of the special operators used by the Editor. INITEDIT is called during loading of the Editor but it may also be used at other times. FLUSHEDIT removes all of the Editor's top-level bindings, mode definitions and operator fixities.

5.8 CHANGING THE NAMES OF EDITOR OPERATORS

The routine EDOPCHANGE of type

EXPR(OP:SYMBOL, NEWNAME:SYMBOL; NONE)

is defined at top-level when the Editor is loaded. The argument OP specifiès the operator which is to be redefined, and NEWNAME is the symbol which is to be used instead of the original. Eg:

EDOPCHANGE("..", "!+")

causes the Editor to recognize !+ as the evaluation substitution operator. The user is responsible for both removing the Editor's assignment of operator fixity, and defining fixity of the new name (if desired). The argument OP always refers to the original name of the operator, and not to its current binding if it has been changed.

6. THE ECL COMPILER

6.1 PROCEDURE REPRESENTATION

In section 2.10, the EL1 explicit procedure form EXPR(...) ... is called a constant of mode ROUTINE. When the interpreter reaches an EXPR expression, it simply converts the form to ROUTINE and returns it without evaluating any of the forms embedded within it. These constituent forms, the formal mode expressions and the procedure body, are evaluated when the procedure is applied.

ROUTINE is a built-in pointer mode, equivalent structurally to the united pointer mode

PTR(DTPR, CEXPR, SUBR, VCEXPR, VSUBR)

DTPR ("dotted pair") is the pair structure used to build the list representation of EL1 programs used by ECL's interpreter. The other types of data that a ROUTINE value can point to are machine code structures. A CEXPR ("compiled EXPR") is an object built by the compiler to hold compiled code and data. It is a STRUCT with components that describe the formal parameters (FORMALS), formal result mode (RETYPE), pointer data (DATAB), and the code body (BODY) of the routine. The CEXPR BODY also includes non-pointer data and relocation information needed when the machine code is moved around in storage. SUBR is the mode of built-in, hand-written machine code subroutines. PRINT, for example, is a built-in ROUTINE, and MD(VAL(PRINT)) is SUBR. SUBRs are similar to CEXPRs without pointer data (DATAB) tables and without relocation information; in general, SUBRs will not work if they have been copied. VCEXPR and VSUBR are simple extensions of the corresponding "V-less" modes: VCEXPR::CEXPR and VSUBR::SUBR. Their special role in the conversion and calling of machine language procedures is described below.

Procedure values are defined as united pointers in ECL for two practical reasons. First, pointers are convenient values to pass around; they are fixed in size and much less bulky than the code they address. Second, the use of a united pointer representation allows routine variables to switch freely between list structure and machine language procedures.

Strong procedure modes (see section 2.16) are also united pointer modes, with the same VAL alternatives as ROUTINE. Physically, any procedure mode, regardless of the argument and result structure it describes, is capable of representing any procedure value. Differences among strong procedure modes are behavioral rather than physical. Two

procedure values may refer to the same code, and yet, because they have different nominal modes, they may act differently when being called or converted.

Section 2.16 states rules for mode and bindclass agreement and tells how those rules are used, when a strongly typed procedure is called, to validate the binding between the procedure and its nominal mode. It also mentions that when a machine language routine, compiled or built-in, can be shown at the time of its conversion to agree with its nominal mode, the representation of the converted procedure is used to signal that no further call-time validation is necessary.

This is where the "V-modes", VCEXPR and VSUBR, come in. V stands for "validate"; when a strongly typed procedure refers to a VCEXPR or VSUBR, validation must occur during each call, just as it would for list structure routines. If the code object is referred to as a CEXPR or SUBR, no call-time validation takes place, the formal parameter mode forms are not evaluated, the formal result mode is not checked, and the number of arguments provided is assumed correct.

The trick is that, since VCEXPR is defined as VCEXPR::CEXPR, and similarly for VSUBR, the same code may be addressed as a CEXPR by a procedure whose mode agrees with it and as a VCEXPR by another procedure for which agreement is not certain. The choice of code type is controlled by the built-in pointer conversion algorithm. (See section 4.3 and the definition of SYSTEM\CONVERT in Appendix B.) When called upon to convert a source value S to a strong procedure mode PM, the algorithm first makes sure S is a pointer to a DTPR, CEXPR, SUBR, VCEXPR, or VSUBR, or else is NIL. If not, a type fault occurs, just as it might during ordinary conversion to a united pointer mode. Otherwise, the conversion succeeds, but not immediately. If S is NIL or points to a DTPR, no further checking occurs at conversion time. A NIL procedure is caught as an "UNDEFINED PROC" during its call. List structure routines (built from DTPRs) always undergo nominal/formal correspondence checks at call-time, since (unlike machine language routines) they are assumed to be modifiable after binding.

Finally, suppose MD(VAL(S)) is CEXPR or VCEXPR. If the number of formal parameters equals the number specified by PM, if every formal argument mode and the formal result mode of the code object is constant, if every nominal argument mode from PM agrees with the corresponding formal mode (in the sense of section 2.16), and the formal result mode agrees with the nominal result mode, and all the bindclasses agree, then conversion yields a procedure value that refers to VAL(S) as a CEXPR. It can be called without further

nominal/formal validation. If any of these conditions is not met, the converted PM value still points to VAL(S), but as a VCEXPR. Its call may be slower.

The case in which VAL(S) is a SUBR or VSUBR is treated similarly.

Note that the compiler can replace many formal mode expressions, e.g. STRUCT(R:PTR(STRING), L:INT), by constants when it produces the code object. Note also that nominal/formal mode agreement is not a symmetric relationship. Formal argument modes may be broader than nominal argument modes. The nominal result mode may be broader than the formal. But not conversely in either case.

6.2 FREE VARIABLES IN COMPILED PROCEDURES

The compiler's treatment of free variables is the key to the level of efficiency achieved by compilation. It is also the source of most of the confusion experienced by new users of the compiler.

A variable is free with respect to a procedure if its use lies outside the scope of any locally declared variable (including formal parameters) with the same name. Remember that the scope of a local variable begins after the entire declaration that creates it. So in

```
DECL X:INT LIKE X
  DECL Y:REAL LIKE X+1
```

X has two free uses. Remember also that while the formal result mode expression of a procedure is evaluated within the scopes of its formal parameters, the modes of the formal parameters themselves are not. For example, in

```
EXPR(M:MODE, Z:M; M) ...
```

the use of M as Z's declared mode is free, but the use of M as the result type is not; it refers to the routine's first argument.

When an explicit procedure is encountered within a routine being compiled, it begins a fresh environment, independent of any local declarations in force. In EL1 there is not necessarily any connection between the environment in which a procedure is defined and the meanings of its free variables. So, for example, in

```
DECL F:PROC(INT; INT) LIKE FACTORIAL;
DECL PF:PROC(INT) LIKE
  EXPR(N:INT) PRINT(F(N));
```

the use of F in the body of PF is considered to be free.

A large proportion of the identifiers used in a procedure are typically free variables with respect to it. After all, nearly every reference to an EL1 primitive is a free variable use. Free variables are, by definition, expressions about which no declarative information appears in the procedure text. In principle, the identifier + may be dynamically re-bound by a caller to some value wholly unrelated to the built-in arithmetic addition routine. It is evident that the programmer must have some way of making declarations to the compiler about free variables. So a means has been provided for ascribing values or other properties to free variables. The categories into which frees can fall are described in the remainder of this section; the mechanics of classifying them are discussed in sections 6.4 and 6.5.

But first a word of caution. Unlike local declarations in EL1, which are both imperative and declarative, global declarations to the compiler are taken on faith, without either compile-time or run-time attempts to verify their accuracy or determine whether they alter the meaning of the programs being compiled. For example, in specifying the mode of a free variable, the programmer must be precise and certain that his declaration always obtains. Otherwise, the compiler may assume an incorrect representation, with disastrous results.

6.2.1 Global Variables

Most free variables refer not to locally declared names in the enclosing environment, but to top-level, or global, variables. In many cases, these values are not "variable" at all; they remain fixed during the execution of the program. Whether they vary or not, however, access to global values is always more efficient when the compiled program can avoid searching the local environment for them.

The compiler recognizes two kinds of global names: constant names and shared value names. Free names declared constant are simply replaced by their top-level values at compile-time, just as if the constant had appeared literally in the text. So names may be used instead of literal constants (e.g. PI instead of 3.141592) for mnemonic reasons without loss of efficiency. For many data types, EL1 provides no way of writing literal constants. Names declared constant during compilation also serve this purpose.

The compiler assumes unless told otherwise the names of all built-in routines are to be treated as constants with

their corresponding ROUTINEs as values. This default assumption can be broken by making any explicit declaration about a built-in name, including a constant declaration. It is not enough simply to rebind the name during compilation, however.

When a name declared constant is found to have a built-in routine (actually any SUBR pointer) as its value, it is treated just like the built-in name itself. Calls on the operators AND and OR, for example, are normally compiled directly into machine instructions instead of subroutine calls. The programmer can accord the same treatment to the identifiers & and V by assigning

```
& <- AND;
V <- OR;
```

at top level, and then declaring & and V constant.

In general, the invariance of a constant value is only assumed to hold for the object itself, not for any other data that may be accessible from it through pointers. As with manifest constants, compiled code will create a pure copy of the object itself when necessary to protect the original. But objects linked to it by pointers will not usually be considered part of the constant.

A notable exception is the case of mode constants. Mode constants can reach the compiler in several ways: either indirectly, as the modes of declared global values or other free variables, or directly, through constant declarations and even through their inclusion by ECL's parser in program text (see section 4.1.2). Knowledge of mode properties is so critical to good compilation that all features of a constant mode are assumed to remain "frozen" throughout the running of the program. User-provided forms that produce extended mode behavior functions, for example, are often compiled directly "in-line" when needed, even though the :: operator can in principle be used to change them before or during execution. If the programmer has some reason for retaining the variability of a mode M, he should not use M as the declared mode of a free variable (see 6.2.2 below). If M is extended, its tag should be something besides "M", to prevent its becoming a syntactic constant. And, of course, M should not be declared constant in this case, though it may be a shared value. The user is cautioned, however, to develop his programs with a view to "freezing" as many modes as possible before compiling a production version.

Like names declared constant, a shared value name represents a single top-level variable whenever it is used free, so it need not be looked up dynamically. The

difference, of course, is that shared values must be assumed to vary. Shared global variables are an efficient and convenient means of interprocedure communication. In a compiled package, the names themselves can be eliminated, saving space and avoiding name conflicts, since code objects point directly to the shared values. They are not appropriate, of course, in certain recursive algorithms, where each activation may need to create a distinct environment for use (free) by subordinate routines, or in programs like the ECL editor, that present users an augmented environment using local declarations. But when use of shared globals is appropriate, it is to be encouraged.

One less than obvious use for shared value declarations is to describe procedures defined at top-level. Typically, top level procedures are constants during a program's execution. They can be changed between compilation and use, however. The compilation process itself alters procedure values when it replaces list structure by machine code. And recompilation of parts of packages requires switching back to the source version. To be sure that these changes are properly reflected throughout the package, global routines should be shared, not declared constant.

6.2.2 Compile-time Macros

The liberal use of procedure definitions to achieve clarity and maintainability is a healthy programming practice. To encourage it, the compiler provides two kinds of macro substitution facility. Each permits the replacement of a procedure application at compile-time by another expression that may be more efficient. Either scheme may alter the meaning of the program unless used with care.

Macro substitution takes place when the name of a procedure being applied is a free variable that has been declared a macro. (Treatment of the same variable in other contexts than procedure application depends on whatever other attributes it may have.) A substituted macro name has an associated EXPR form. The macro call is replaced by a copy of the EXPR body in which each use of a formal parameter is replaced by the corresponding actual argument expression. The result is then compiled in line.

CAUTION: This call-by-name handling of substituted macro arguments violates the semantic definition of EL1 if a macro formal parameter is used more than once in the body and the corresponding actual argument has side effects. Consider the absolute magnitude routine:

```
ABS <-
  EXPR(X:ARITH; ARITH)
    [) X LT 0 => -X; X (]
```

In some respects it is a good candidate to become a substituted macro: it is non-recursive, and its body is short enough that, in many cases, less code will be needed to compile it in line than to compile the procedure call. However, the expansion of ABS(A <- A+1) would be

```
[) (A <- A+1) LT 0 => -(A <- A+1); (A <- A+1) (]
```

which clearly does not preserve the original meaning. To become a suitable substituted macro, ABS would need a local declaration such as DECL X:ARITH LIKE X at the beginning of its body.

The substituted macro facility is intended only as a stopgap, awaiting a more rigorous procedure back-substitution and optimization scheme. It is perhaps most appropriate for isolating simple but representation-dependent access algorithms from the programs that use them. For example, the predicate that determines whether a mode has a user-defined generation function (see section 4) might be declared a substituted macro:

```
HAS\UGF <- EXPR(M:MODE; BOOL)
             BEGIN
               DECL U:PTR(SEQ(REF)) LIKE M.UFN;
               U # NIL AND U[5] # NIL
             END
```

A computed macro name has an associated PROC(FORM, SYMBOL; FORM) value. Whenever a macro call is reached, the compiler applies the corresponding macro procedure to two arguments, the list of actual arguments and the macro name itself. The form returned is then compiled in-line. For example, suppose one has written routines called PLUS and TIMES. Each takes a variable number of arguments and expects each argument to evaluate to a number. PLUS computes the sum of its arguments; TIMES, the product. In order to have calls such as TIMES(Z, PLUS(X, A[I], Y), 10) replaced at compile time by infix expressions, in this case Z*((X+(A[I]+Y))*10), one can attach the following computed macro to both PLUS and TIMES:

```
EXPR(L:FORM, OP:SYMBOL; FORM)
  BEGIN
    L = NIL => QUOTE(0);
    L.CDR = NIL => L.CAR;
    LIST([) OP = "PLUS" => "+"; "*" (],
         L.CAR,
         CONS(OP, L.CDR));
```

```
END
```

where LIST produces a list with the values of its arguments as elements, and CONS(X, Y) makes a list from the new item X and the old list Y: CONS("PLUS", <A, B>) is (PLUS A B). Note that this macro may produce a form whose compilation invokes the macro again.

6.2.3 Other Free Variables

Free variables that are not built-in names, declared global, or declared and used as macros, must be identified dynamically by the compiled routine in the environment of its call. "Dynamic identification" in ECL means searching the name stack, the stack of local bindings currently in force, for the most recent binding of the variable. If none is found, then the top-level binding is used. (Every name has a top-level binding; those not set explicitly have the default value NOTHING.)

Since dynamic identification is a time-consuming process, the compiler normally produces code to look up the free variables used in a routine all at once, just after the routine is entered. The bindings thus created are added to the name stack, and for the rest of the routine they are treated just like local parameters, which need not be looked up when they are referenced.

When free variables are identified with local bindings (stack entries) of the environment, this scheme of lookup-on-entry is perfectly valid. Since nothing can alter the mode, size, or location of locally bound values in the environment enclosing a routine during its activation, and since the number and order of those locals also remain fixed, there is no need to reidentify each free variable each time it is used.

A problem arises when the free variable is identified with a top-level binding. While the lookup-on-entry scheme is consistent with the model of an "outermost block", enclosing all procedure activations and including declarations for each top-level variable, ECL is more flexible than that model suggests. Top-level bindings can be created and destroyed dynamically. So free name identifications made at the beginning of a compiled routine and not renewed at each use can be invalidated within the body. The following statements will, when interpreted, load the ECL unparser if it is not resident, and then use it to print a form F:

```
UNPARSE = NOTHING -> LOADB"SYS:UP";
UNPARSE(F);
```

LOADB establishes top-level bindings destructively; in this case, if UNPARSE is initially unbound, LOADB leaves it bound to a procedure value at top level. If the program containing these statements is compiled, and the lookup-on-entry method is used to identify UNPARSE, the program can die trying to apply the value NOTHING to the form F. The rebinding of UNPARSE globally is not reflected in the pseudo-local binding used by the compiled routine.

So a class of free names called rebound names is defined. When a free variable is declared a member of this class, the compiler will make sure that it is re-identified whenever it is used if its binding in the environment could possibly have changed. In the example just given, UNPARSE should be declared a rebound name.

Finally, when it is known that an identifier will have a particular mode in all its free uses, the association of name and mode can be declared to the compiler. The declared mode may be generic, e.g., ONEOF(STRING, SYMBOL), to indicate that one alternative may apply throughout one activation, another during another activation. But the modes so declared must be complete and accurate. They are not ordinarily checked at run-time; the compiler is quite trusting on this point. For example, the correct mode of UNPARSE is PROC(FORM, PORT, INT). It is not ROUTINE, and, although UNPARSE can be coerced to ROUTINE, free variable mode declaration does not lead to coercion. If UNPARSE is a rebound name because it may be unbound when first referenced, then even the correct PROC mode is not broad enough. The proper declared mode in this case is ONEOF(NONE, PROC(FORM, PORT, INT)). Unless NONE is declared a possible alternative mode for UNPARSE, the compiler treats the test UNPARSE = NOTHING as tautologically FALSE.

Though the compiler normally relies on the accuracy of free variable mode declarations, the user can require that modes so declared be verified when the variables are used. The declared mode must cover the actual mode; again, no mode conversion will take place. Mode verification is expensive, since the tests may be scattered through the body of the routine, even when the variable itself is looked up and bound on entry. If the programmer feels the need for free mode validation as a permanent feature, he is usually better advised to add a declaration such as DECL V:M SHARED V to his program and to omit any compiler declaration about V's mode. However, the mode verification option can be quite useful when a newly compiled routine is not working properly, and erroneous free variable mode declarations are suspected.

If the user makes no declarations about a free variable other than a built-in routine, it will be looked up on entry

to the procedure that uses it and assumed to have potentially any mode.

6.3 INTERPRETER/COMPILER COMPATIBILITY

Genuine, irreconcilable incompatibilities between the treatment of a program by ECL's interpreter and its compiler are few, and for the most part, obvious. Programs that modify their own list representation probably will not survive compilation. Neither will those that depend on details of their own stack records (as divulged by PEEK); the compiler tries to simplify stack bookkeeping, but it may also add stack entries not present during interpretation, such as name stack entries for free variables and others for temporary results. One result is that the picture of an environment printed by the backtrace program BT will be markedly different when a BREAK occurs in a compiled routine from that shown at the same point when the routine is interpreted. Indicators for some blocks will appear as "CBLOCK" (with no accompanying statement, of course) while others will have disappeared altogether. Most names of entered procedures will have disappeared, as may some local variable names. Clearly, one should test a procedure thoroughly before compiling it.

"Reconcilable" incompatibilities arise because of default assumptions made in an effort to simplify the compilation process for the user. The user must be aware of these assumptions, of course, so he will know when to override them. Some have already been mentioned in section 6.2. Built-in routine names are assumed to refer to built-in routines unless the programmer explicitly says otherwise. Global variable and macro declarations are made at the user's own risk; though potential sources of incompatibility, they can simply be omitted to revert to proper semantics. Declarations of the modes of dynamic (i.e., not necessarily global) free variables are likewise the user's to control; if in doubt, he can omit them or require that they be checked at run-time. And finally, there is the important assumption of fixed identification for dynamic free variables, leading to a single lookup of each at routine entry. This assumption can be overridden by declaring names <u>rebound</u>.

Furthermore, for each declaration type that overrides some expedient assumption, there is also a compiler switch, or "toggle", that overrides the assumption for <u>all</u> free names, so that the user need not ferret out and name every free variable explicitly in order to request compatible compilation. For details of how to make declarations and set toggles, see section 6.4.

Another expedient, related to the use of dynamic free variables and employed by the compiler unless overridden, is the dropping of names of local parameters from the environment of a compiled procedure. Compiled code for a procedure refers directly to name stack entries of its locals without needing their names. Name-dropping saves both space and time, and the knowledge that local names can be discarded in compiled packages may encourage use of names chosen for descriptive clarity, not space efficiency. However, if local names are to be used free by routines called within their scope, or if they may appear in forms passed to EVAL by the compiled routine or one of its subroutines, the names should be retained. Retention of local names is controlled by compiler declaration and toggle.

Unfortunately, not every phrase of a program being compiled can be rendered as machine code. Sometimes, it is necessary to leave part of the program in its list structure format for possible interpretation at run-time. In these cases, the variable references in the uncompiled forms may require retention of the corresponding names. The compiler is unable to decide, in general, which forms will ultimately be evaluated and in what environments. So it does <u>not</u> alter its local name-dropping practices on the basis of forms it cannot compile in line. It does, however, provide some help in finding names that must be kept.

From the point of view of the compiler, expressions fall into three categories: (1) those it compiles into machine instructions, (2) those it cannot compile lest the program treat them as list-structure data instead of executable text, and (3) those that are execute-only data. Expressions in category (3) are forms that, if they are used at all, will simply be passed to EVAL. They can be compiled, provided the results are still forms, and that each is equivalent to the original under EVAL. To achieve the best performance from compiled programs, and to deal correctly with local name retention declarations, the programmer should have some idea of conditions under which the compiler is able to translate an expression into machine language.

Obviously, explicitly quoted forms and lists, e.g., QUOTE([) P => E1; E2 (]) and QL(A, B, C+D), are shielded from compilation in nearly every context. (The exceptions are given in the discussion of THUNK and THUNKLIST below.) The special formal parameter bindclasses, UNEVAL and LISTED, are simply convenient ways of achieving the same kind of shielding from evaluation and hence, compilation.

When the compiler reaches a procedure application and it cannot determine the strong mode of the routine being

applied, it has to leave the whole actual argument list uncompiled. Not only might some arguments be bound UNEVALuated of LISTED, but the number of expected arguments might be less than the number given; EL1's semantics say that extras will not be evaluated.

The uncertainties of incompletely specified application extend to some data generation and selection expressions as well. Consider E0[E1, ..., En]. If the mode of the initial object E0 or that of some partial result of selection, E0[E1, ..., Ek], for k<n, is not known, then the remaining selector forms are not compilable, since they may become arguments to a user-defined selection function. Generation by components gives rise to a similar situation; in the declaration DECL V:E0 OF E1, ..., En, and in analogous CONST and ALLOC expressions, E1 through En will be left uncompiled when the expected mode represented by E0 is not known to the compiler. If the mode produced by E0 at run-time has a user-defined generation function, it will expect to be given a list of unevaluated component forms. Generation by SIZE and by example (see section 4.3) do not have this property. In both cases specification values are compiled, regardless of whether the expected mode is fixed at compile time.

Now consider the compiler's handling of a call on the built-in pointer mode generator, PTR. If possible, the call is evaluated at compile-time and replaced by a mode constant. Occasionally, however, code will have to be generated to call the PTR routine at run-time. Since PTR accepts a variable number of arguments, it has one formal parameter, bound LISTED. It extracts the element forms of the input list and uses EVAL to turn them into the actual arguments it needs. So, although PTR expects a list of forms, the compiler knows that each can be compiled; since it will just be EVALuated, its particular representation is not significant.

In this and similar situations, the compiler creates a type of form called a thunk. A thunk is an execute-only form, one whose representation is irrelevant to the program as long as it EVALuates properly. Given an EL1 expression E, an equivalent thunk is produced by embedding E in an explicit procedure application:

E ≡ (EXPR(; ANY) E)( )

Now the EXPR can be compiled; the only part of the thunk left in list format is the application expression (i.e., the compiled EXPR applied to an empty argument list). The PTR routine, then, is given a list of thunks produced from its actual argument forms. It proceeds to evaluate them, indifferent to their representation.

The programmer can specify the same treatment for other quoted forms and lists using the built-in modes THUNK and THUNKLIST. Both are simple mode extensions: THUNK::FORM and THUNKLIST::PTR(DTPR). Whenever an expression known at compile-time is converted to THUNK or to some mode extension of THUNK, the compiler will feel free to replace the expression by an equivalent thunk. Expressions that are simple identifiers, or constants, or are already in thunk form, will be left as they are, since nothing will be gained by compiling them. Likewise, when a list constant is to be converted to THUNKLIST or an extension of THUNKLIST, the compiler may substitute another list whose elements are "thunked" renditions of the original.

To take an example, let PRINT\MANY be a procedure that prints the values of any number of expressions:

```
PRINT\MANY <-
  EXPR(L:THUNKLIST LISTED)
    BEGIN
      DECL T: FORM BYVAL L;
      REPEAT
        T = NIL => /* 'DONE ... EXIT LOOP AND ROUTINE';
        PRINT(EVAL(T.CAR));
        T <- T.CDR;
      END;
    END;
```

A typical use of PRINT\MANY might read

```
PRINT\MANY('THE AVERAGE AGE IS:',
           BEGIN
             DECL S: REAL;
             (FOR I TO N REPEAT S <- S+AGES[I] END)/N;
           END,
           NEWLINE);
```

Under the ECL interpreter, PRINT\MANY would work just as well if its argument specification were L:FORM LISTED. But since the input list elements are simply EVALuated, not used as data themselves, the programmer chooses THUNKLIST. When the compiler reaches the call of PRINT\MANY shown above it examines each actual argument as a potential thunk. The first argument is a constant; nothing will be gained by enclosing it in an EXPR and compiling. The same is true of the third argument, NEWLINE. The middle argument, however, is replaced by a thunk, and a new list of the three arguments is prepared for presentation to PRINT\MANY at run-time.

The treatment of THUNK and THUNKLIST is another type of extra-linguistic agreement between the compiler and its user. Like free variable declarations, it can lead to

interpreter/compiler incompatibilities if misused. Special handling of these modes can be disabled by a compiler toggle.

In addition to the obvious speed advantage of "thunking", there is a side-benefit. Global values, both implicit and declared, are dissociated from the top-level names by which they are addressed in the list structure representation. Returning to the use of PRINT\MANY given earlier, not that four names are free with respect to that expression: N, AGES, +, and NEWLINE. If the arguments were not treated as thunks, each of these names would require a binding in the environment at run-time, even though they may be declared global. Thunks, on the other hand, always isolate global free variable values from the corresponding names. Even the third actual argument to PRINT\MANY, NEWLINE will be modified to refer directly to the compile-time top-level value if NEWLINE is a global name. The result of this isolation is that most of the global names used in program specification and interpretive debugging can be dropped from the compiled package. So one can enjoy the efficiency of global variables and constants and still avoid top-level name conflicts when unrelated programs are loaded together.

Unfortunately, thunking does not also help eliminate "local free variable" names. For example, in

```
PRINT\ROSTER <-
  EXPR(NAMES:SEQ(SYMBOL))
    FOR I TO LENGTH(NAMES)
      REPEAT PRINT\MANY(I, % , NAMES[I], NEWLINE) END;
```

the local variables I and NAMES are used free when PRINT\MANY evaluates the elements of its argument list. Those names must therefore be declared as retained local names, whether or not PRINT\MANY accepts a THUNKLIST. To help the user figure out which names must be retained, both locally and globally, the compiler generates a list of the tokens found free within each expression compiled. For this purpose a free token is defined as an identifier (other than the elementary syntactic keywords like BEGIN and DECL) that the compiler is unable to associate with a specific local or global binding. For example, assembling PRINT\MANY is a shared global value, and NEWLINE is a global constant, the free token list for PRINT\ROSTER, defined above, would consist of NAMES and I, since, although local to PRINT\ROSTER, they are free with respect to thunks within it. Another list kept by the compiler contains the constant forms it encounters but cannot compile, even as thunks. If PRINT\MANY took a FORM LISTED instead of a THUNKLIST LISTED, then the free token list for PRINT\ROSTER would be empty, but its list of uncompiled constant forms would contain the

whole actual argument list to PRINT\MANY. So the user can discern, not only that I and NAMES must be retained, but that NEWLINE must be defined in the run-time environment as well.

6.4 USING THE COMPILER

From the user's point of view the compiler is not a single, monolithic procedure. It is a collection of routines whose functions range from processing of compiler declarations, to control of the translation itself, to the creation of binary output files. Most compiler inputs are not arguments to these routines but bindings to global variables in the compilation environment. The usual method of compiling a package is to prepare a file of ECL commands to create the compiler inputs, load the necessary source files, then load and call the various components of the compiler. A program called SCAN has been provided to help build such a command file. The programmer need not necessarily adopt this method, however. After becoming familiar with the compiler, he may choose a scheme that better suits his style. He might, for example, maintain the compiler inputs in his source file, along with a small routine of his own to drive compilation.

6.4.1 A Typical Compilation

A typical compilation involves the following steps:

(1) The user's source program is LOADed, and the necessary compiler input variables are set. These are lists of procedures to be compiled, declarations about free variables, toggle settings, and the like.

(2) The main control module of the compiler, called DRIVER.BIN, is also loaded. DRIVER defines all the compiler routines normally accessible to the user, although the translator itself is contained in three other files, PASS1.BIN, PASS2.BIN, and PASS3.BIN.

(3) A routine named DECLARE\FREES is called to process free variable declarations and other compiler inputs. DECLARE\FREES inspects the values of all input variables and makes appropriate default bindings where necessary. It converts declarative information into the tabular format used by the translator proper. For each top-level procedure to be compiled that does not have a strong nominal mode (one generated by PROC), DECLARE\FREES tries to strengthen the mode using the EXPR header. Though quite dumb about deciding whether an arbitrary mode form may safely be evaluated at compile-time, it often succeeds anyway, since

it takes global constant declarations into account and since most formal mode expressions involve only simple applications of the mode generators.

(4) A routine named LOAD\COMPILER is called to load and initialize the translator modules PASS1.BIN through PASS3.BIN. (In overlay mode, described below, LOAD\COMPILER has no effect; the individual passes are loaded as needed.)

(5) COMPILE\ROUTINES is applied to the list of names of top-level procedures to be compiled. Each undergoes three translation phases. PASS1 analyzes the routine and produces an intermediate tree representation. From this tree PASS2 generates symbolic machine code in the form of list structure consistent with ECL's internal syntax. PASS3 assembles the code into CEXPR objects. To conserve space the input to each phase is normally deleted when the phase is complete. This includes the source routine. Besides managing these passes, COMPILE\ROUTINES collects and prints statistics, and will optionally produce a listing file containing the machine code generated for one or more of the routines compiled.

(6) FLUSH\COMPILER is invoked to delete the translator passes from storage. In overlay mode, this routine is unnecessary, but it should be included so that switching to and from overlay mode will be simple.

(7) Finally, DUMP\PACKAGE generates two binary files called the BIG file and the BIN file. The latter contains the programs and other structures accessible from the names that are to be available at top-level to the user of the package. The BIG file is intended to permit restoration of the environment just after compilation. It includes the bindings of compiler input variables, all the user's global variables, including the newly compiled routines, and the tables built by DECLARE\FREES for the translator. The BIG file is used for selective recompilation of parts of a package, and can also be helpful in debugging when the programmer has been overoptimistic about discarding "superfluous" information from the production (BIN) file.

6.4.2 Compiler Input Variables

Nearly all inputs to the compiler are transmitted via assignments to a number of global variables. Though inelegant, this scheme permits easy alteration of declarations and switches during and after compilation, which is particularly useful for recompilation of selected routines. The names chosen are rather long not only for suggestive and mnemonic reasons, but in the hope of avoiding conflicts with user variable names. DECLARE\FREES announces

any conflicts it can recognize.

The heading of each of the descriptions below gives the name of the variable, along with its expected mode and, in parenthesis, the default value used if necessary. There may be a further description of the expected value of the variable as well. In these descriptions, the term "list" means a FORM that points to a chain of DTPR's linked through their CDR fields. A "name" is either a quoted symbol or a form that looks like an identifier when unparsed. <SAM, "WALDO", RESET, INT> is an acceptable name list, even though the third element is actually a nofix procedure application (the procedure name will be extracted), and the fourth is a mode constant (the symbolic tag will be extracted). A "mode-name" is a name that is either a symbolic mode tag or is bound at top level to a mode at compile-time. An "EXPR-expression" is a form that can safely be evaluated by the compiler and that produces an explicit (list-structure) procedure value. Analogously, a "PROC(FORM, SYMBOL; FORM)-expression" is one that can be coerced, on evaluation, to such a PROC value. A "pair" is a list of two elements; the parser produces a pair, consisting of the procedure form and the argument form, when it recognizes a monadic procedure application. So <COUNT(INT), COORDINATES(VECTOR(3, REAL))> is a proper "list of name(MODE-expression) pairs".

ROUTINE\NAMES FORM (NIL) name list

Specifies the top-level procedures to be compiled. Each name in this list should ordinarily be in SHARED\NAMES as well. (See the discussion of shared global values in section 6.2.1.)

DECLARE\FREES will try, if necessary, to strengthen the mode of each routine named. While strengthening permits more efficient compilation of procedure calls, it may also break sharing patterns, and of course, by changing the modes of some variables, it may affect the meaning of the program. If FIDO has the definition

```
FIDO <- EXPR(S:SEQ(M) SHARED; ANY) ...
```

in the source file, then the test MD(FIDO) = ROUTINE succeeds before FIDO is compiled. Once FIDO has had its mode strengthened to PROC(SEQ("M*") SHARED; ANY) or the like, the same test would fail. Plan ahead.

If global routines are shared by other data structures, strengthening can also interfere. Suppose the source file contains

```
DISPATCH\TABLE <-
  CONST(SEQ(REF) OF
          "FIRST\CASE".TLB,
          "SECOND\CASE".TLB,
          "ALL\THE\REST".TLB)
```

Here the programmer has tabulated references to procedure values, hoping to have the values themselves updated automatically by the compiler. This idea is sound, except for those routines that must be strengthened. A new mode means a new top-level binding; DISPATCH\TABLE is left pointing to the old.

The paragraph below on PRELUDE\FORM suggests a way to recover from this problem. The way to avoid it is to use strong nominal modes in the first place.

CONST\NAMES FORM (NIL) name list

Names of the global variables to be replaced, when found free, by their compile-time values as constants. (See section 6.2.1.)

SHARED\NAMES FORM (NIL) name list

Global names whose compile-time values are to be shared by the compiled routines that use them as free variables. (See section 6.2.1.)

SMACRO\PAIRS FORM (NIL)
list of name(EXPR-expression) pairs

Names and definitions of substituted macros. (See section 6.2.2.)

CMACRO\PAIRS <-FORM (NIL)
list of name(PROC(FORM, SYMBOL; FORM)-expression) pairs

Names and definitions of computed macros. The expression part of each pair should be coercible to PROC(FORM, SYMBOL; FORM) when evaluated by DECLARE\FREES. This procedure will be used by the translator to process macro calls on the corresponding name. (See section 6.2.2.)

MDKN\PAIRS <-FORM (NIL)
list of name(MODE-expression) pairs

Declared modes for free variables used dynamically. (See section 6.2.3.) MDKN stnads for "mode known".

VERIFY\NAMES <-FORM (NIL) name list

Names of dynamic free variables mentioned in MDKN\PAIRS whose declared modes will be verified at each use. Coercion to the expected mode is not implied. Either the declared mode covers the actual or an error ("ACTUAL/NOMINAL MODE MISMATCH") results. (See section 6.2.3.)

VERIFY\ALL BOOL (FALSE)

When TRUE, all names in MDKN\PAIRS will have their declared modes verified at each use.

REBOUND\NAMES FORM (NIL) name list

Names of variables that may be rebound (not just assigned a new value, but given a completely new binding, with a new location and usually a new mode) at top level during the execution of the compiled program. These variables will be dynamically re-identified (looked up in the current environment) each time they are used free. (See section 6.2.3.)

RETAIN\NAMES FORM (NIL) name list

Local variable names to be retained in compiled routines. (See section 6.3.)

RETAIN\ALL BOOL (TRUE)

When TRUE, all local variable names of the source program will be retained in the compiled version. Note that the compiler's default action is to retain all such names. Nevertheless, the number of local names that need to be kept is rarely large. It is to the user's advantage to ferret them out, list them in RETAIN\NAMES, and then set RETAIN\ALL to FALSE.

GUESS\MODE BOOL (FALSE)

When TRUE, the compiler may assume that no data mode relationships can develop at run-time that will result in an execution error. For example, if F is a FORM variable, and the compiler encounters the assignment F <- F.CDR, it can

assume (when GUESS\MODE is TRUE) that F points to a DTPR before the assignment is evaluated.

The GUESS\MODE flag compensates in part for the compiler's inability to recognize a thoroughly debugged program that correctly accounts for all feasible states of its data. Use it cautiously.

DONT\THUNK BOOL (FALSE)

When TRUE, the compiler will give no special treatment to expression or list constants bound to the modes THUNK, THUNKLIST, or their extensions. (See section 6.3.)

LIBRARY\AREA SYMBOL ("" at Harvard; "<CONRAD>" at USC-ISI)

Indicates where the files PASS1.BIN through PASS3.BIN are to be found. If LIBRARY\AREA is set to "DTA4:", for example, the modules would be loaded from that device. On a TENEX system, the setting "<CONRAD>" means the translator lives in CONRAD's directory. The value "" for LIBRARY\AREA designates whichever DSK: area the user is connected to as the library.

PACKAGE\NAMES FORM (union of all other NAMES lists)
name list

Names to be made accessible in the compiled package (BIN file). Normally, the package names are those the package user will need to mention. Top-level names used in expressions left unevaluated by the compiler, however, must also be left accessible. (See section 6.3 and the description of SHOW\QUOTATIONS, below.)

If PACKAGE\NAMES is initially unbound or NIL, DECLARE\FREES merges ROUTINE\NAMES, SHARED\NAMES, CONST\NAMES, and SAVED\NAMES (see below) to create the default.

BIN\NAME SYMBOL ("DEFALT.BIN")

File designator of the production binary version of the compiled package.

BIN\SIZE INT (1)

The initial number of hash table pages to be used in dumping the BIN file. BIN\SIZE becomes the third argument

in a call to DUMPB (see sections 2.15.3 and 7.5); the final hash table size (in pages) is printed after the file has been created. Dumping and subsequent loading will be fastest and require least available memory if BIN\SIZE is well matched to the package. Hash table size is a function of the amount of pointer data, not necessarily the overall size of the package. DUMPB never contracts its hash table, so BIN\SIZE should be chosen conservatively.

INIT\FORM FORM (NIL)

Form to be evaluated by LOADB after the contents of the BIN file have been loaded, and all package names have been bound. The best practice is usually to consolidate initialization code in one routine, defined in the source file and included in PACKAGE\NAMES. If this routine is named INIT\MYSTUFF, for example, then INIT\FORM can be defined by INIT\FORM <- QUOTE(INIT\MYSTUFF( )).

PRELUDE\FORM FORM (NIL)

Form to be evaluated at the end of DECLARE\FREES, after declarations have been processed. At that point, strengthening of procedure modes is complete and the members of ROUTINE\NAMES have the bindings they will keep throughout compilation. So PRELUDE\FORM can be used to rebuild data structures that must refer to these bindings. Returning to the DISPATCH\TABLE example given in the description of ROUTINE\NAMES, one could simply include the whole definition of the table in PRELUDE\FORM. Since DISPATCH\TABLE is itself most likely a shared value, however, it is important not to rebind the table after declarations have been processed. One relatively clean way of satisfying these constraints is to define the table as follows in the source file:

```
FLUSH(DISPATCH\NAMES, DISPATCH\TABLE) /*
  'in case sizes have changed since last LOAD';
DISPATCH\NAMES <-
  CONST(SEQ(SYMBOL) OF "FIRST\CASE", "SECOND\CASE",
                       "ALL\THE\REST");
DISPATCH\TABLE <-
  CONST(SEQ(REF) SIZE LENGTH(DISPATCH\NAMES));
PRELUDE\FORM <-
  QUOTE(FOR I TO LENGTH(DISPATCH\NAMES)
          REPEAT
            DISPATCH\TABLE[I] <- DISPATCH\NAMES[I].TLB;
          END);
EVAL(PRELUDE\FORM);
```

DISPATCH\NAMES will fade harmlessly away when the BIN file is created (unless it is deliberately mentioned in PACKAGE\NAMES). To repeat, this sort of effort is necessary only when the user (1) wishes to let the compiler strengthen procedures instead of using PROC modes explicitly, and (2) wants to build structures that share routine bindings.

POSTLUDE\FORM FORM (NIL)

A form evaluated by DUMP\PACKAGE before the creation of the BIN file. POSTLUDE\FORM serves a purpose similar to that of PRELUDE\FORM. Since it is executed after compilation, it is the place to build data structures that embed compiled procedure values directly (as opposed to references to the shared routines that the compiler updates).

EXTENDED\MODES FORM (NIL) mode-name list

Modes whose user-defined mode behavior function forms (see section 4.2) are to be evaluated and replaced by direct references to the resulting procedure values. The replacement is performed by DUMP\PACKAGE just before it creates the BIN file. Most behavior function "forms" will actually be simple identifiers that are members of ROUTINE\NAMES. The idea is to replace these names by the corresponding compiled values, lest the names be forced unnecessarily into PACKAGE\NAMES. If a mode has a behavior function form that cannot correctly be evaluated in the global environment just after compilation, the mode should be left out of EXTENDED\MODES.

SHOW\FREES BOOL (FALSE)

When TRUE, COMPILE\ROUTINES prints the list of free tokens found in compiled expressions, including thunks, but not including forms left uncompiled. (See section 6.3.)

SHOW\QUOTATIONS BOOL (FALSE)

When TRUE, COMPILE\ROUTINES prints the forms left uncompiled in each routine it translates. (See section 6.3.)

BIG\NAME SYMBOL (NIL)

File designator of the BIG file, the file intended to preserve the environment immediately after compilation.

(See section 6.4.1.) If BIG\NAME is NIL, no BIG file is dumped.

BIG\SIZE INT (1)

Initial hash table size (in pages) to be passed to DUMPB when DUMP\PACKAGE creates the BIG file. (See the discussion of BIN\SIZE above.)

SAVED\NAMES FORM (NIL) name list

Names of global variables that are neither SHARED\NAMES nor CONST\NAMES, but that should nevertheless be included in the BIG file for use during partial recompilations. DISPATCH\NAMES, used in the PRELUDE\FORM example above, might be such a datum: not part of the package itself, but used to build (or rebuild) another value, DISPATCH\TABLE, that is.

LISTING\NAMES FORM (NIL) name list

Names of the routines for which COMPILE\ROUTINES is to produce a listing file of symbolic machine code. No listing file is created if LISTING\NAMES is NIL.

LST\NAME SYMBOL ("DEFALT.LST")

File designator for the listing file optionally produced by COMPILE\ROUTINES.

OVERLAY\MODE BOOL (FALSE on TENEX; TRUE at Harvard)

When TRUE, each translator pass is applied to all routines being compiled before the next begins. If QUICK\FLUSH is also TRUE, each pass is flushed from memory when no longer needed. In non-overlay mode, each routine is completely translated before the next is started.

QUICK\FLUSH BOOL (FALSE on TENEX; TRUE at Harvard)

Used in overlay mode to indicate that each translator pass can be deleted immediately after being used.

LOAD\EARLY BOOL (TRUE on TENEX; FALSE at Harvard)

When TRUE, enables LOAD\COMPILER to load all three translator phases. Otherwise, overlay mode disables early loading.

6.4.3 Compiler Routines

Section 6.1 gives the general purposes of the major routines defined in DRIVER. This section adds details that should aid the reader in interpreting compile-time messages, in handling errors, and hopefully in finding ways of using the compiler that most suit his needs and style.

DECLARE\FREES CEXPR( )

Looks at the binding of each compiler input in the top level environment. If an input variable is unbound (has value NOTHING), it is rebound to the appropriate default. If it is bound to an object that cannot be coerced to the mode the compiler expects the variable to have, it is still rebound, but the message "REBINDING: <variable>" will be printed.

DECLARE\FREES then processes the lists of names and pairs that comprise the programmer's instructions to the compiler. As it does so, it also edits the lists so that they can be used by other parts of the compiler without further checking. For instance, the elements of a name list will all be left as simple identifiers, even though other forms (quoted symbols, mode constants, and nofix expressions) are permissible inputs. If an input element is not among these alternatives, DECLARE\FREES announces "BAD NAME: <erroneous form>" and asks whether to "REPLACE OR IGNORE" the miscreant. If the user responds by typing R, he will be asked for a replacement symbol; if he types I, the offending element will be deleted from the list. Whenever DECLARE\FREES is awaiting input, the user can force an escape to top level by typing CTRL-Z.

Treatment of the members of ROUTINE\NAMES is particularly important. Each must be bound at top level to an explicit procedure (EXPR). If the value found does not appear to be a valid EXPR, the message "CANT COMPILE: <routine name>" is printed, and the user is permitted either to replace the name (e.g., if it has been misspelled) or ignore it (i.e., remove the element from ROUTINE\NAMES).

If the procedure named has a strong nominal mode, it is left unaltered. Otherwise DECLARE\FREES tries to strengthen the mode of the routine. It examines each formal argument mode and the result mode to determine whether they can safely be evaluated. A "safe" argument mode expression is a

straightforward composition of the mode generators (STRUCT, SEQ, VECTOR, PTR, ONEOF, ::, and PROC) with manifest constants and free names that are declared CONST\NAMES. A safe result mode is similar, but cannot include a free name that is one of the formal parameter names. If all the header mode forms are safe, they are used to create a proper strong mode for the procedure, and it is rebound to that mode. If only the formal result mode fails to meet the safety criterion, a strong mode is created with ANY in place of a specific result type. For example, a routine headed EXPR(M:MODE; M) would be strengthened to PROC(MODE; ANY). When a member of ROUTINE\NAMES is weakly typed and cannot be strengthened by DECLARE\FREES' crude algorithm, the message "CANT STRENGTHEN": <routine name>" will appear. This is an admission, not an admonition, however. Compilation proceeds without interruption.

CONST\NAMES are dealt with before ROUTINE\NAMES so that declared constants can be used in strengthening. SHARED\NAMES are handled after ROUTINE\NAMES so that the routine values shared will include those generated by strengthening. (As mentioned in section 6.2.1, the ROUTINE\NAMES of a package should nearly always be declared shared values as well.)

Processing MDKN\PAIRS, DECLARE\FREES checks that the first element of each is a name, it evaluates the second pair element and requires it to be a COMPLETE mode. If one of these conditions fails, the pair is excised from the list with the announcement "IGNORING MDKN\PAIR: <erroneous pair>". Similar validation and rejection of malformed pairs take place for SMACRO\PAIRS and CMACRO\PAIRS, except that instead of modes, the property expressions are expected to produce explicit procedures (substituted macros) and PROC(FORM, SYMBOL; FORM) values (computed macros), respectively. Incidentally, it is reasonable and acceptable for an SMACRO name to appear as both elements of a member of SMACRO\PAIRS (e.g., <..., CADDR(CADDR), ...>), even when the same name is in ROUTINE\NAMES. Remember that a macro is expanded only when its name is the entire procedure part of an applicative expression. For the other contexts in which a routine value can be used, it may be advisable to compile the routines being used as substituted macros.

Similar validation is applied to SAVED\NAMES, EXTENDED\MODES, and PACKAGE\NAMES. If PACKAGE\NAMES is NIL, it is replaced by the union of ROUTINE\NAMES, SHARED\NAMES, CONST\NAMES, and SAVED\NAMES. Finally, PRELUDE\FORM is evaluated, completing the precompilation preparations.

LOAD\COMPILER CEXPR( )

Loads the translator passes, PASS1.BIN, PASS2.BIN, and PASS3.BIN, from the directory or device specified by LIBRARY\AREA, unless LOAD\EARLY is FALSE and OVERLAY\MODE is TRUE.

COMPILE\ROUTINES CEXPR(NAMES:FORM, FLUSH\NOT:BOOL)

Translates the top-level routines mentioned in NAMES from list-structure representation to machine language. If OVERLAY\MODE is TRUE, PASS1 (analysis) is applied to all routines before PASS2 (code generation) begins for any; likewise PASS3 (code assembly) is deferred until PASS2 has been applied to each. Since each translator pass will be loaded when needed but not resident, this order achieves an overlaying of the phases. When OVERLAY\MODE and QUICK\FLUSH are both TRUE, each pass is deleted after being used for all routines. If OVERLAY\MODE is FALSE, each member of NAMES is translated completely before the next is started.

Unless FLUSH\NOT is TRUE, each routine in NAMES is set to NIL as soon as it is no longer needed. This makes for a smaller compilation environment; it does not affect the sharing of routines. At the end of translation, the same routine value receives the compiled version.

COMPILE\ROUTINES prints a summary giving the amount of computation time, in seconds, required to compile each routine. If garbage collection occurs during translation of a routine, the extra time is printed separately, next to the time for compilation alone. If PASS\TIMES is TRUE, timing information appears for the individual passes. If SHOW\FREES is TRUE, a list of the free tokens in compiled expressions is included in the summary. If SHOW\QUOTATIONS is TRUE, the forms left uncompiled are also printed.

If LISTING\NAMES is non-NIL, COMPILE\ROUTINES opens the file designated by LST\NAME and UNPARSEs the symbolic machine code for the routines named in LISTING\NAMES.

COMPILE\ROUTINE CEXPR(A:ANY; ANY)

Translates its argument from list form to machine code, loading the passes if necessary and flushing them after use if OVERLAY\MODE and QUICK\FLUSH are TRUE. No summary is printed; no listing file is created. The input value A is not modified. The result is coerced to the mode of the input.

FLUSH\COMPILER CEXPR( )

Deletes all three phases of the translator (PASS1, PASS2, and PASS3), but not the routines used to drive compilation (i.e., those summarized in this section).

DUMP\PACKAGE CEXPR( )

Optionally creates the BIG file, then evaluates POSTLUDE\FORM, then dumps the BIN file for the package.

If BIG\NAME is non-NIL, DUMPB is called to create a binary file whose name is the value of BIG\NAME, whose hash table size estimate is BIG\SIZE, and whose contents are the user's globally declared values, plus the members of SAVED\NAMES, plus the values of the compiler input variables and the tables created by DECLARE\FREES for use by the translator. By reloading (using LOADB) DRIVER and the BIG file, the user restores the essential features of the environment just after compilation but just before dumping of his package. He can tinker with parameter settings, test parts of his program while all global variables are still bound at top level, replace some compiled routines by source versions, and, if he wishes, he can call COMPILE\ROUTINES to translate the new versions of those routines. Finally, another call to DUMP\PACKAGE will dump new BIG and BIN files reflecting the new state of the package.

The BIN file, the production binary version of the package, is created by a call to DUMPB equivalent to

DUMPB(BIN\NAME, PACKAGE\NAMES, BIN\SIZE, <init-form>)

where <init-form> represents the value of INIT\FORM.

FLUSH\DRIVER CEXPR( )

Calls FLUSH\COMPILER, then deletes the driving routines of the compiler, those described in this section and defined in DRIVER.BIN. Compiler input variables are not flushed, however.

6.4.4 Creating Compiler Inputs

SCAN is a program that reads an ECL source file and aids the user in preparing a compilation command file. SCAN builds values for the major compiler input variables, ROUTINE\NAMES, SHARED\NAMES, and so on. It begins with whatever settings these variables have in the environment of its call. Then it loads the input file and uses definitions in it to augment the initial compiler directives. Next, it builds a list of commands to initialize compiler inputs and

drive compilation. The user is permitted to examine and modify these commands using the list structure editor. At his option, the package may then be rescanned in light of his modifications. Finally, the command list is written on a file designated by the user; to compile his package, he simply LOADs the command file so created.

6.4.4.1 The Initial Scan

SCAN's first argument is the file designator symbol for the input file; its second is the designator for the output command file. It begins by creating its own local bindings for ROUTINE\NAMES, SHARED\NAMES, and the other lists and forms that are the principal inputs to the compiler. These local bindings are initialized to their counterparts in the current environment, if they exist, but changes made during scanning only affect the local copies.

Each form in the input file is evaluated. Particular attention is given to those that represent top-level variable definitions. If the right side of a definition is an explicit procedure, the name being defined is included in both ROUTINE\NAMES and SHARED\NAMES. If the right side of a definition is a default value of its type, the variable on the left is assumed to be a shared global variable; its name is included in SHARED\NAMES. Otherwise (i.e., for non-procedure, non-default, top-level definitions) the name is included in CONST\NAMES. CAUTION: it should be clear that these and other guesses made by SCAN may be dead wrong. The programmer must consider each of them carefully and correct them when necessary.

If, during the initial scan of the input file, a call to LOAD another file is encountered, SCAN will type:

LOAD <file or port> ... SCAN, LOAD, OR IGNORE?

If the user responds with S, the file will be scanned as if it were part of the input file. If he types L, the file will be loaded but not scanned, and if he types I the LOAD will be ignored.

Special treatment is also given uses of the :: operator during the initial scan. If the extended mode being defined appears to have non-trivial behavioral properties (specifically, if the left operand to :: is a list), the symbolic mode tag will be added to EXTENDED\MODES. In other words, SCAN assumes the user wants mode behavior function forms rebound to constants by DUMP\PACKAGE. (See sections 6.4.2 and 6.4.3.) If not, he should edit the mode tags out of EXTENDED\MODES.

If a syntax error occurs, the message "SCAN ABORTED" will be typed when the simulated LOAD is finished, and SCAN will proceed no further.

Finally, a list of commands is constructed. One command loads the user's source file, named by SCAN's first argument. Several others are definitions for the principal compiler input variables. Lists and forms have the definitions absorbed from the enclosing environment and amended during the initial scan. BIG\NAME and BIN\NAME are bound to file designators derived from the source file name. Near the end of the list, there is a command to load DRIVER.BIN. The last command is a BEGIN-block containing the compiler routine calls that will compile and dump the package. The first statement of this block closes LDPORT, the port used by LOAD to read the command file itself. This conserves storage and makes it easier to use SAVE and RESTORE (see section 7.8) during compilation if necessary.

Next begins the iterative, interactive process of correcting and augmenting this initial command list.

6.4.4.2 The Free Variable Scan

SCAN searches each procedure named in ROUTINE\NAMES for instances of free variables other than those built-in to ECL or already included in CONST\NAMES, SHARED\NAMES, MDKN\PAIRS, SMACRO\PAIRS, or CMACRO\PAIRS. The name of each routine is printed, along with the names of any undeclared free variables ("loose frees") found. SCAN's analysis is faster but less thorough than the compiler's. To SCAN, a token is a loose free name unless it is globally declared or appears to be covered by a local declaration. SCAN is not able to distinguish compilable text from list structure data. So it may describe tokens as free variable names when they are not actually variables at all. And it may miss some names that appear to be local but are actually free with respect to thunks or other executable data. The compiler itself, through the SHOW\FREES and SHOW\QUOTATIONS options, is more accurate.

Next, the names of loose frees are compared with names of variables declared locally in the package to see whether a unique, safely evaluatable mode form can be associated with the free name. (Safe mode expressions are described in the discussion of DECLARE\FREES given in section 6.4.3.) If a unique mode is found, the name(MODE-expression) pair is added to MDKN\PAIRS. Note again, however, that this is just a guess; if inaccurate, it must be corrected by the programmer.

After all the package routines have been searched, the new value of MDKN\PAIRS is edited into the command list. An extra line is also included among the commands, one that will not appear in the command file ultimately created from the command list. It has the form

$$\begin{aligned} &\text{LOOSE\FREES}(id_1(\text{USED IN}(routine_1, \ldots, routine_k),\\ &\qquad\qquad \text{COULD BE}(mode_1, \ldots, mode_m)),\\ &\qquad\qquad \vdots\\ &\qquad id_n(\ldots)) \end{aligned}$$

That is, the available information about undeclared free variables is summarized for the user's reference. Where meaningful and possible, he should classify the identifiers mentioned as LOOSE\FREES by including them in other lists. In any case, however, there is no need to edit the LOOSE\FREES line.

6.4.4.3 Editing the Command List

The ECL Editor is next applied to the command list so that the programmer can examine and correct guesses made by SCAN. He can add inputs left out or left blank; particular attention should be given the definition of PACKAGE\NAMES, for example. And he can add commands needed to prepare the compilation environment, such as calls to LOAD or LOADB that fetch needed auxiliary files.

The editor signals its readiness by typing an asterisk. In addition to the editor's usual repertoire, SCAN provides several routines to make editing of the compilation commands easier. All except RESCAN are designed to deal with list-variable definitions of the form:

```
<variable-name> <- QL(...);
```

In the descriptions that follow, the phrase "current line" means the expression currently under the editor's attention pointer, the expression printed when the user types ALT-MODE.

ADD CEXPR(L:FORM LISTED)

Looks for an application of QL in the current line and adds the members of L to the arguments of QL. For example:

```
* S(MDKN\PAIRS)$
MDKN\PAIRS <- QL(COUNT(INT));
* ADD(NUMBER(ONEOF(REAL, COMPLEX)), B(BOOL))$
MDKN\PAIRS <- QL(NUMBER(ONEOF(REAL, COMPLEX)),
```

```
                        B(BOOL),
                        COUNT(INT));
```

REMOVE CEXPR(L:FORM LISTED)

For each member E of list L, removes the first QL argument of the current line that contains E as a subexpression. Continuing the example started above:

```
* REMOVE(NUMBER, COUNT(INT))$
MDKN\PAIRS <- QL(B(BOOL));
```

AT CEXPR(S:SYMBOL UNEVAL,
P:PROC(FORM LISTED),
L:FORM LISTED)

Searches the command list for an assignment whose left side is the value of S and whose right side is a QL application, then applies P to the list L. For example:

```
* AT(SHARED\NAMES, ADD, WHISTLE, GONG)$
SHARED\NAMES <- QL(WHISTLE, GONG, BELL, ALARM);
```

AT2 CEXPR(S:SYMBOL, P:PROC(FORM LISTED),
L:FORM)

Identical to AT, but with evaluated first and third arguments.

SC CEXPR(L:FORM LISTED)

REMOVEs the members of L from the definition of SHARED\NAMES and ADDs them to CONST\NAMES. SC is equivalent to

```
[) AT2("SHARED\NAMES", REMOVE, L);
   AT2("CONST\NAMES", ADD, L) (]
```

CS CEXPR(L:FORM LISTED)

REMOVEs the members of L from CONST\NAMES and adds them to SHARED\NAMES.

RESCAN CEXPR( )

Leaves the editor, evaluates the modified commands that define free variable categories, then repeats the free variable scan (section 6.4.4.2) for the members of (the new) ROUTINE\NAMES. When the rescan is complete, the lines of the command list specifying MDKN\PAIRS and LOOSE\FREES will be updated and the editor reentered.

Ideally, after use of RESCAN, the programmer should find no LOOSE\FREES other than those he knows to be irrelevant tokens and those he has decided to leave unclassified (dynamic variables, for example, whose mode is not predictable but that will not be REBOUND during execution, in the sense of section 6.2.3).

The editor EXIT command will cause the command list (without the LOOSE\FREES line) to be written on the file designated by SCAN's second argument.

6.4.5 Operating Instructions

At Harvard, the files SCAN.BIN, DRIVER.BIN, PASS1.BIN, PASS2.BIN, and PASS3.BIN reside on the DECtape in bin 61. At USC-ISI, they are in directory <CONRAD>.

6.4.5.1 Using SCAN

SCAN is used only to create a compilation command file; it need not be present during the compilation. To use SCAN, enter ECL and LOADB "SCAN" (at Harvard), or LOADB"<CONRAD>SCAN" (at USC-ISI). At Harvard, where EDIT.BIN and UP.BIN are not part of the resident ECL system, SCAN loads these files from "SYS:" if necessary. Next, preset any compiler input variables, such as SHARED\NAMES or INIT\FORM, that SCAN is expected to "absorb" from its environment. If these settings are stored in the source file, for example, then LOAD the file. Finally, evaluate

SCAN(<source-designator>, <command-file-designator>)

For example, SCAN("SAM", "SAMDRV") will create a file called SAMDRV.ECL to be used in compiling SAM.ECL.

One line in SCAN's output file will be monitor dependent. It is the command that loads DRIVER, the primary compiler module. At Harvard, SCAN produces LOADB"DRIVER"; at USC-ISI, LOADB"<CONRAD>DRIVER". If a command file created under one system is to be used under the other, this line must be edited. Normally, no other changes are necessary.

6.4.5.2 Initial Compilation

During compilation, DRIVER and PASS1 through PASS3 must be available. At Harvard, the default setting for LIBRARY\AREA is the blank symbol "", meaning the compiler files are to be found in the user's DSK: area. If PASS1 through PASS3 are elsewhere, he should edit a different setting of LIBRARY\AREA into the compilation command file. If DRIVER is also somewhere else, the line LOADB"DRIVER" must also be corrected. At USC-ISI, LIBRARY\AREA defaults to "<CONRAD>".

Harvard compilations normally run in overlay mode: default settings of OVERLAY\MODE, QUICK\FLUSH, and LOAD\EARLY are TRUE, TRUE, and FALSE, respectively. On a TENEX system, it is more efficient to use non-overlay mode, so these three default settings are inverted.

To initiate compilation, enter ECL and LOAD the command file.

6.4.5.3 Tinkering with Compiled Packages

From a BIG file, one can often produce an updated version of the package (BIG and BIN files) without resorting to total recompilation. One can adjust values of global variables, alter compiler lists and parameters (e.g., PACKAGE\NAMES, INIT\FORM, or POSTLUDE\FORM), supersede compiled routines with list code, and, optionally, compile the new versions.

An example should suggest how the DRIVER routines can be used to make such patches. Suppose SAM.ECL has been compiled, producing SAM.BIG and SAM.BIN. The programmer wants to update two procedures, SIMPLIFY and INTEGRATE, and to recompile the former, leaving INTEGRATE in interpreted form for the time being. After modifying the source file, he extracts the new definitions of SIMPLIFY and INTEGRATE and makes a file called PATCH.ECL containing just their definitions. After entering ECL, he issues the following commands:

```
LOADB"SAM.BIG";
LOAD"PATCH";
LOADB"DRIVER";
LOAD\COMPILER( );
COMPILE\ROUTINES(QL(SIMPLIFY));
FLUSH\COMPILER( );
DUMP\PACKAGE( );
```

Note that it was not necessary to re-apply DECLARE\FREES since the results of its preprocessing are included in the

BIG file. Also, in OVERLAY\MODE, LOAD\COMPILER and FLUSH\COMPILER have no effect. If the patches do not require any recompilation, of course, the calls to these routines and to COMPILE\ROUTINES are unnecessary, and PASS1 through PASS3 need not even be available.

Patching, of course, has its limits. One cannot expect declarations modified after compilation to take effect in existing machine code. With global variable declarations, the compile-time mode and location of the global value are effectively part of the declaration. While shared variables may safely be assigned a new value during patching, no global variable of the package should be destructively rebound, e.g., to change its mode or length.

A subtle problem arises when a compiled routine is being superseded by another that has a different formal parameter or formal result mode structure. Reassignment and recompilation may take place normally. But unless the new definition agrees with the original nominal mode in the sense detailed in section 2.16, a "FORMAL/NOMINAL MISMATCH" error will arise when the new routine is called. A new nominal mode is needed. But changing the mode of a shared global variable requires a full recompilation.

7. BUILT-IN ROUTINES

This section describes routines which are built-in to the ECL system. Those labelled "SUBR" are permanently embedded in the basic system. Others, labelled "CEXPR" (for compiled-EXPR), are written in the format of compiled procedures and those labelled NSUBR are not looked up dynamically but rather cause control to jump directly to the machine code associated with the top-level identifiers. Some of these are not loaded with the basic system, but may be read-in using LOAD or LOADB (see section 7.5). The non-resident routines are so indicated before their descriptions, and names of the files in which they may be found are given.

For each built-in routine, the following information is given:

(1) the top-level name to which the routine is bound.

(2) the type of routine, i.e. SUBR or CEXPR;

(3) argument modes and their bind-classes;

(4) the mode of the result;

(5) a brief description of the routine;

(6) parsing properties of the routine name. If it can be used as a prefix (or nofix) operator, the word PREFIX (or NOFIX) appears below its description. If it is infix, both the word INFIX and a priority appear. (Priorities specify the relative binding strength of operators--highest being tightest--and run from 1 to 255).

7.1 GENERAL ROUTINES

IE SUBR(A:ANY; ANY)

This permits the inclusion of comments in EL1 programs. See section 2.13 for details.

PREFIX, INFIX 253

/* SUBR(A:ANY; ANY)

Identical in meaning to IE.

PREFIX, INFIX 253

*/ SUBR(A:FORM UNEVAL, B:ANY; ANY)

Permits insertion of a comment as the left hand operand. See section 2.13 for details.

INFIX 253

<- or ← NSUBR(X:ANY, Y:ANY; ANY)

Assigns the value of the object Y to the object X.

INFIX 50

EVAL SUBR(X:FORM; ANY)

Evaluates X and returns its value.

MD SUBR(X:ANY; MODE)

The mode of X (i.e. its data type).

LENGTH SUBR(X:ANY; INT)

If X is a struct or a row then the number of components in X is returned. If X is a pointer, returns the LENGTH of VAL(X). If X is not compound, returns 1 (except that LENGTH(NOTHING) is 0).

VAL SUBR(X:ANY; ANY)

If X is not a pointer, issues type fault. Otherwise returns the object pointed to by X.

QUOTE SUBR(X:FORM UNEVAL; FORM)

The value of QUOTE(f) is f, for any form f in the language.

QL SUBR(L:FORM LISTED; FORM)

Returns L unevaluated. QL(a, b, c) is the list (a b c).

< SUBR(L;FORM LISTED; FORM)

Identical in meaning to QL.

RECLAIM SUBR(N:INT, M:MODE, B:BOOL, F:INT)

Invokes a garbage collection which attempts to obtain at least N words to hold objects of mode M. If the desired storage cannot be obtained, a storage-allocation fault occurs. If successful, RECLAIM returns the number of free words available for objects of mode M. B becomes the value of an ECL internal flag which, when TRUE, inhibits the release of wholly free pages to the PDP-10 monitor. This feature is used to avoid repeated garbage collections by preallocating large blocks of storage. F controls the frequency of compactifications. The system default is 1 compactifying GC per 5 GCs. Calling RECLAIM with F#0 will change this as follows -- if F GT 0, a CGC occurs every F GCs; if F LT 0, CGCs are suppressed.

7.2 ARITHMETIC AND TRIGONOMETRIC ROUTINES

The following routines use the built-in mode ARITH which is defined as ONEOF(INT, REAL) (see Appendix C).

+ SUBR(X:ARITH, Y:ARITH; ARITH)

The arithmetic sum of X and Y. The result is an INT if both X and Y are INT; otherwise it is REAL.

INFIX 175

\- SUBR(X:ARITH, Y:ONEOF(INT,REAL,NONE); ARITH)

If the argument Y is omitted or has mode NONE the result is the negative of X. Otherwise the result is X less Y and is an INT unless either X or Y is REAL.

PREFIX, INFIX 175

* SUBR(X:ARITH, Y:ARITH; ARITH)

The arithmetic product of X and Y. The result is an INT if both X and Y are INT; otherwise it is REAL.

INFIX 200

/ SUBR(X:ARITH, Y:ARITH; ARITH)

X divided by Y. If both arguments are INT, the result is an INT and is truncated toward zero. Otherwise, the result is REAL.

INFIX 200

SUM SUBR(X:ARITH, Y:ARITH; ARITH)

Identical in meaning to +

DIFF SUBR(X:ARITH, Y:ONEOF(INT,REAL,NONE); ARITH)

Identical in meaning to -

PRODUCT SUBR(X:ARITH, Y:ARITH; ARITH)

Identical in meaning to *

QUOTIENT SUBR(X:ARITH, Y:ARITH; ARITH)

Identical in meaning to /

NOTE: The routines SIN, COS, ATAN, and LN are not resident in the basic system. To load them, type LOADB("SYS:TRANS");

SIN CEXPR(X:REAL; REAL)

Trigonometric sine of X. Argument is expressed in radians.

COS CEXPR(X:REAL; REAL)

Cosine of X.

ATAN CEXPR(X:REAL; REAL)

Arctangent of X. Result is expressed in radians.

LN CEXPR(X:REAL; REAL)

Natural logarithm of X.

EXP CEXPR(X:REAL; REAL)

Inverse of LN.

7.3 LOGICAL AND RELATIONAL ROUTINES

AND SUBR(L:FORM LISTED; BOOL)

The elements of L are evaluated in turn, and each value is converted to BOOL. If it is FALSE, then AND immediately returns FALSE. If it is TRUE, the next element is considered. If the list L is exhausted before a FALSE value is encountered, then AND returns TRUE. If a value not convertible to BOOL is encountered, a type fault occurs.

INFIX 125

OR SUBR(L:FORM LISTED; BOOL)

The elements of L are evaluated in turn, and each value is converted to BOOL. If it is TRUE, then OR immediately returns TRUE. If it is FALSE, the next element is considered. If the list L is exhausted before a TRUE value is encountered, then OR returns FALSE. If a value not convertible to BOOL is encountered, a type fault occurs.

INFIX 100

NOT SUBR(X:BOOL; BOOL)

The logical negation of X.

PREFIX

= SUBR(X:ANY, Y:ANY; BOOL)

TRUE for objects of primitive mode iff X and Y are identical values (except that if X is REAL and Y is INT then TRUE iff CONST(REAL LIKE Y) = X or similarly for INT X and REAL Y). TRUE of pointers iff X and Y point to the same object. TRUE of structs and rows iff (MD(X) = MD(Y) AND LENGTH(X) = LENGTH(Y) and for all I from 1 to LENGTH(X), X[I] = Y[I].

INFIX 150

EQUAL SUBR(X:ANY, Y:ANY; BOOL)

Identical in meaning to =.

SUBR(X:ANY, Y:ANY; BOOL)

Defined as NOT(X=Y).

INFIX 150

GT SUBR(X:ARITH, Y:ARITH; BOOL)

TRUE iff X is arithmetically greater than Y.

INFIX 150

GE SUBR(X:ARITH, Y:ARITH; BOOL)

TRUE iff X is arithmetically greater than or equal to Y.

INFIX 150

LE SUBR(X:ARITH, Y:ARITH; BOOL)

TRUE iff X is arithmetically less than or equal to Y.

INFIX 150

LT SUBR(X:ARITH, Y:ARITH; BOOL)

TRUE iff X is arithmetically less than Y.

INFIX 150

The following routines are available in the file MACH.BIN; they allow bit manipulation of PDP-10 machine words represented as INTs. All are equivalent to the corresponding PDP-10 machine language instructions. LSH and LROT shift or rotate left if B GT 0 and right if B LT 0.

LAND CEXPR(A:INT, B:INT; INT)

LOR CEXPR(A:INT, B:INT; INT)

LEQV CEXPR(A:INT, B:INT; INT)

LNOT CEXPR(A:INT; INT)

LSH CEXPR(A:INT, B:INT; INT)

LROT CEXPR(A:INT, B:INT; INT)

7.4 OPERATOR-DEFINING ROUTINES

NOFIX SUBR(S:SYMBOL; NONE)

Causes S to be a nofix operator. Example: after executing NOFIX("LOGOUT"), use of the identifier LOGOUT will be interpreted as if (before executing NOFIX) the code had read LOGOUT().

PREFIX SUBR(S:SYMBOL; NONE)

Causes S to be a prefix operator.

INFIX SUBR(S:SYMBOL, N:INT, F:BOOL; NONE)

Causes S to be an infix operator with priority N. (If N is zero, it acts like N=254). If the flag F is TRUE, then S is right associative; otherwise, S is left associative. Example: if & is a right associative operator, then x & y & z is treated as x & (y & z).

MATCHFIX SUBR(L:SYMBOL, R:SYMBOL; NONE)

L and R become a pair of matchfix brackets. Since a matchfix bracket has no other fixity, some routine should be bound to L prior to the call on MATCHFIX. Nesting of bracket expressions may require liberal use of spaces. For example, < <A, B>, <C, D> > is interpreted correctly but in <<A, B>, <C, D>> the double brackets is taken as a single identifier.

FLUSHFIX SUBR(S:SYMBOL; NONE)

Causes S to be no longer an operator of any sort, i.e. flushes the effect of any previous NOFIX, MATCHFIX, PREFIX, or INFIX calls with S.

7.5 INPUT/OUTPUT ROUTINES

OPEN SUBR(F:SYMBOL, D:SYMBOL, T:SYMBOL, V:FORM; PORT)

Opens the file F, in direction D, for I/O of type T. F must be a file designator of the form:

dev:name.ext[proj, prog]

Any portion of F may be omitted with the following defaults:

dev - DSK

name - DEFALT

ext - ECL

proj - user's

prog - user's

D must be either "IN", or "OUT". (If D is missing, the default is "IN").

T must be either "SYMBOLIC" or "BINARY". (If T is missing, the default is "SYMBOLIC").

V is an optional FORM which is evaluated when end-of-file F is reached.

Having opened the file as specified by <F, D, T>, OPEN returns a port which can be used to access this file.

CLOSE SUBR(P:PORT; NONE)

Closes the port P. Subsequent attempts to do I/O through P will cause an end-of-file fault.

DRAIN SUBR(P:PORT; NONE)

Forces any remaining data held in core buffers of port P to be output to the device associated with P without closing the file.

MAKEPF SUBR(X:REF, D:SYMBOL, T:SYMBOL, V:FORM; PORT)

Makes a pseudo-file from the STRING or SEQ(INT) pointed to by X. If X is a PTR(STRING) then the pseudo-file is open for symbolic I/O; if X points to a SEQ(INT) then the pseudo-file is open for binary I/O. D specifies the direction ("IN" or "OUT"). If the direction is OUT, MAKEPF zeroes the buffer pointed to by X. Subsequent I/O operations using the port returned by MAKEPF will read from or write into the pseudo-file. T indicates whether the data is "BINARY" or "SYMBOLIC", the latter being the default. V is an optional FORM which will be evaluated at end-of-file.

NOTE: All I/O routines that take a port argument may default that argument. In such cases, the port used is as follows:

(a) the primary input or output port, if that is non-NIL, otherwise

(b) the teletype input or output port, which will never be NIL.

NOTE: The teletype input and output ports are always open for symbolic I/O. Attempts to perform binary I/O on them will cause an I/O fault.

OUTCHAR SUBR(C:CHAR, P:PORT; NONE)

Writes the character C on port P. P must be open for symbolic output.

INCHAR SUBR(P:PORT; CHAR)

Reads one CHAR from the port P. P must be open for symbolic input.

OUTOBJ SUBR(N:ANY, P:PORT; ANY)

OUTOBJ assumes its first argument the data object N has mode NONE, INT, REAL, CHAR, BOOL, or a sequence or vector of any of those modes (this includes the mode STRING) and that its second argument has been opened for binary output. The output is packed and is the internal binary representation of the data object. A sequence is not preceded in the file by its length.

In the PDP-10 implementation of ECL, the number of bits output for each of the basic modes listed above is 0, 36, 36, 7, and 1 respectively. Output of a BOOL followed by an INT, for instance, will result in exactly 37 bits being placed into the output file. When the file is closed, the last word will be padded if necessary with binary zeroes.

The result returned by OUTOBJ is the value of the data object output.

INOBJ SUBR(M:MODE, P:PORT; M)

INOBJ reads from port P which it assumes has been opened for binary input. Since no mode or length information is placed in the output file by OUTOBJ, the action of INOBJ is governed entirely by the mode given as its first argument. INOBJ generates an object of that mode in the heap and then copies enough bits from the file to fill the object. In the case of the mode STRING, for instance, a string of length zero will be generated, and nothing will be input from the file!

Although INOBJ generates in the heap, the object itself is returned, not a pointer to it. Heap generation is used to avoid the possibility of stack overflow.

READ SUBR(P:PORT; ANY)

Reads, parses, and evaluates a form from port P. In case of syntax error, an error message is typed on the user's terminal and READ accepts a new command from P.

PRINT SUBR(X:ANY, P:PORT, LVL:INT; ANY)

Prints X onto port P in symbolic format and returns X as result. LVL can be used to limit printing as follows. An INT suppression depth D is computed by

```
[) LVL = 0 AND MD(...)=INT => ... ; LVL (]
```

where ... is a distinguished SYMBOL. If D LE 0 then printing is unrestricted. If D GT 0 then the printing of forms at depth dreater than D from the root of X will be suppressed and "..." substituted. Note that if ... has an INT value, then a call such as PRINT (FOO, P, -1) will override that value but PRINT(FOO, P, 0) will not. Implicit calls to PRINT, e.g., via top level commands terminated with ALTMODE, are governed by the binding of "...".

PFORM SUBR(X:FORM, P:PORT, LVL:INT; ANY)

Prints X onto P in LISP-like notation. A depth limit is computed in the same way as with PRINT. X is the result of the call.

LEX SUBR(P:PORT; ANY)

Reads one lexeme from the port P, and returns:

(1) an INT, REAL, STRING, or CHAR, if the lexeme is a constant of one of these modes,

(2) a SYMBOL, if the lexeme is an identifier or delimiter (e.g. (], FIE, or //),

(3) a REF which points to a SYMBOL if the lexeme is a SYMBOL constant like "ABC",

(4) the value NOTHING, if the PORT P is at end-of-file.

PARSE SUBR(P:PORT, P2:PORT; DTPR)

Attempts to read and parse a form from port P. If unsuccessful, returns the dotted pair (NOGO.NIL) after emitting an error message to port P2 (or to the command output port if P2 is NIL). If parsing is successful, and the form was terminated with a semicolon, PARSE returns the dotted pair (SUCCESS . form). If successful and the form was terminated with an ALT MODE, then PARSE returns (PSUCCESS.form). When port P reaches end-of-file, PARSE returns (EOF.NIL).

LOAD SUBR(P:REF; NONE)

P should be either a PORT or a SYMBOL. Let Q be P in the former case, OPEN(P) in the latter case. Repeatedly executes the following cycle (until end-of-file is encountered):

(1) call PARSE to read and parse one command from port Q,

(2) call EVAL to evaluate the command if the parse has been error-free so far.

PREFIX

LOADB SUBR(FILE:SYMBOL; ANY)

FILE, with default extension ".BIN", is read into memory. FILE will have been produced with DUMPB. If a fourth argument was provided to DUMPB that argument is evaluated after loading and the result of that evaluation is LOADB's result.

PREFIX

DUMPB CEXPR(FILE:SYMBOL,
NAMES:FORM,
HTSZ:INT,
F:FORM UNEVAL;
INT)

To use DUMPB, call LOADB "SYS:DUMPB". FILE specifies the output file (with default extension ".BIN"). NAMES is a list of top level variables and MODES whose values are to be output. Sharing patterns are preserved among these values. Values and parsing properties of the names are restored when the file is LOADBed. HTSZ, if present, is an initial size in pages for DUMPB's hash table. The final size of this table is DUMPB's result. F will be evaluated immediately after the binary file has been LOADBed.

7.6 STRING AND CHARACTER HANDLING ROUTINES

BASIC\STR SUBR(X:BASIC; STRING)

Converts X to a STRING.

HASH SUBR(X:STRING; SYMBOL)

Converts X to a SYMBOL.

CHAR\INT SUBR(C:CHAR; INT)

The INT value corresponding to the CHAR C in ASCII representation.

INT\CHAR SUBR(N:INT; CHAR)

The CHAR value corresponding to the INT N in ASCII representation.

REAL\STR SUBR(X:REAL, D:INT, F:BOOL; STRING)

If F is FALSE then X is converted to a STRING representing X in scientific notation with D significant digits (default is 8) and a two digit decimal exponent. The result is rounded properly. If F is TRUE then the STRING represents X as a fixed decimal number with D fractional digits. To convert X to an INT, use built-in REAL to INT coercion.

PORT\STR SUBR(P:PORT, F:BOOL; STRING)

The filename associated with P is returned as a STRING. If F and P is associated with a network connection then the local socket number is the result. Since PORT is an alternative of BASIC, BASIC\STR will also do this conversion.

7.7 DEBUGGING ROUTINES

BREAK SUBR(S:ANY; ANY)

Suspends execution of the program. A message

<mess> BREAK

<fn> BROKEN

is typed out on the user's terminal, where <mess> is the value of S and <fn> is the surrounding function. The state of the computation is preserved. A prompt character is output to the user's terminal and further commands will be accepted.

CONT SUBR(X:ANY; NONE)

Causes program execution to resume from the point where a break occurred, with X as the value of the break.

RESET SUBR(; NONE)

Flushes all non-global context.

RETBRK SUBR(N:INT; NONE)

Returns control to the N-th break level. N=0 brings control to the top level.

ASSERT SUBR(X:FORM UNEVAL; BOOL)

ASSERT is used to make assertions which will be checked if ASSERT\FLAG is TRUE. If the Boolean variable ASSERT\FLAG is not TRUE, then X is ignored and ASSERT returns TRUE. Otherwise, X is evaluated to obtain a boolean value. If this value is TRUE, then ASSERT returns TRUE. If this value is FALSE, then an assertion fault occurs.

STEP SUBR(N:INT, C:ANY; NONE)

Routine to be used at breakpoints to perform (essentially) a CONT(C) and break again after finishing the current statement and evaluating N more statements. For example,

```
X <- BREAK()+5;
PRINT("NEXT")
```

If the command

```
STEP(0,1);
```

is executed at the indicated BREAK, then X is assigned the value 6 and another BREAK occurs before the PRINT statement is evaluated.

DDT SUBR(; NONE)

Jumps into DDT. Note: this is not available on most versions of the system. To get back from DDT to ECL, give the DDT command GOBAK $G.

PEEK SUBR(KEY:FORM UNEVAL, KEYMOD:ANY; ANY)

PEEK is used to determine the current settings of certain system parameters. The first argument, an identifer, supplemented for some of the parameters by a second argument, specifies the parameter to be examined. Only the first six characters of the KEY are necessary. The following table indicates the permitted KEY-KEYMOD combinations along with their meanings. GC refers to the garbage collector, CGC indicates compactifying GC and FGC, "fast" GC (actually no GC at all, rather simply allocation of a free page). The down-counter

indicates when it is time to do a CGC.

KEY	KEYMOD	MEANING
CGCCOUNT	--	Down-counter for CGC frequency
CGCFREQUENCY	N:INT	Do a CGC every N regular GCs
CGCDENSITY	N:INT	Do a CGC if used space drops below N% (1% to 100%)
FREEPAGES	N:INT	Number of pages of core allocatable without intervening GC
FREECOUNT	N:INT	Down-counter for FREEPAGES
GCTIME	--	Time in MS for GC + CGC
CGCTIME	--	Time in MS for CGC
FGCNUMBER	--	Number of FGCs
GCNUMBER	--	Number of GCs and CGCs
GCMESSAGE	N:INT	If N=1, GC messages are output
OPENCGC	N:INT	If N=1, CGC allowed on OPEN
RUNTIME	--	Time in MS since entering ECL
CORESIZE	--	Current number of pages for low segment
MAXSIZE	--	Largest low segment size so far
NEXTATOM	X:SYMBOL	The next SYMBOL after X. The first SYMBOL is accessed by CONST(SYMBOL)
CSRECORD	N:INT	Nth most recent CS record
NSVALUE	N:INT	SHARED value of Nth most recent local variable
NSMODE	N:INT	MODE of Nth most recent local variable
NSNAME	N:INT	Name (a SYMBOL) of Nth most recent local variable
OFBIND	X:SYMBOL	Returns TRUE or FALSE according as X has been set to have bindclass properties like "OF"

POKE SUBR(KEY:FORM UNEVAL, KEYMOD:ANY, NEWVALUE:ANY; ANY)

POKE is used to modify certain system parameters. The meanings of KEY and KEYMOD are the same as for PEEK. The third argument specifies the new value of the parameter. When a modifier is not needed, POKE takes its second argument as the new value. Only CGCOUNT, CGFREQUENCY, CGCDENSITY, FREEPAGES, FREECOUNT, GCMESSAGE, OPENCGC, and OFBIND may be

changed by POKE.

7.8 ENVIRONMENT ROUTINES

SAVE SUBR(P:REF, B:BOOL; NONE)

A PORT Q is obtained from P as with LOAD. The complete environment of the call on SAVE is written as a file on Q with default extension ".LOW". A compactifying garbage collection takes place before the environment is saved unless B is TRUE. If a break occurs during a call on SAVE because of open files, CONT( ) may be used to close a file and continue with the SAVE. CONT( ) may be used repeatedly to close several files.

RESTORE SUBR(P:REF, D:BOOL; NONE)

A PORT Q is obtained from P as with LOAD. The environment file previously written to this PORT by SAVE (default extension ".LOW") is reinstated and computation continues from the point of the SAVE. RESTORE does not return to its caller. If the version of the ECL System has changed, a warning is issued -- beware that results are unpredictable in this situation. If D is TRUE then the referenced file is deleted from the storage medium it resides on.

STACKS SUBR(K:INT, V:INT, N:INT; NONE)

Replaces current stack triple with a new triple providing K pages (512 words) of control stack, V pages of value stack, and N pages of name stack. The value of K (which may not be defaulted) is used for any defaulted arguments.

FLUSH SUBR(L:FORM LISTED; NONE)

L must be a list of variable names. For each variable name on L, the top-level binding of that variable is flushed (i.e. set to NOTHING).

7.9 PRIMITIVES FOR NON-DETERMINISTIC ALGORITHMS

The NDA routines are not resident in the basic system. They may be loaded by typing LOADB("SYS:NDA").

TAG CEXPR(X:FORM UNEVAL; SYMBOL)

A call on TAG labels a point in the program's dynamic execution sequence. A subsequent "branch" to a tag-point restores the program's stack environment to that extant at the time the tag-point was created (possibly modified by normal assignments, after the tag-point's creation, to variables local to blocks dynamically enclosing the tag-point). The argument to TAG yields the symbolic "tag-value" thereafter associated with the tag-point as follows: if VAL(x) has the mode ATOM then a SYMBOL s such that VAL(s)=VAL(x) is the tag-value; otherwise if s=EVAL(x) has the mode SYMBOL then s is the tag-value; otherwise a type fault occurs. The tag-value s may subsequently be used in referring to the tag-point thus created, and is the value returned by the creating call of TAG. A block or procedure that contains a tag-point is said to be "tagged" and an exit from such a block or procedure is a "tagged exit". (Note that this containment need not be immediate--i.e. if a block B is tagged and B' contains B, then B' is also tagged).

PREFIX

NASSIGN CEXPR(X1:ANY, X2:ANY; ANY)

NASSIGN, like <-, changes the value of an existing object. However it also records the location of the object X1 and its old value for use by FAIL. When not dynamically preceded by an existing tag-point, an NASSIGNment is equivalent to the corresponding ASSIGNment. (Note however that NASSIGN may not be used to introduce and implicitly declare new top level objects.)

<= CEXPR(X1:ANY, X2:ANY; ANY)

Identical in meaning to NASSIGN.

INFIX 50

FAIL CEXPR(X:FORM UNEVAL, I:INT, F:FORM; BOOL)

X and I specify at most one tag-point. I must be a nonnegative integer. If X is NIL or defaulted, then the Ith most recent tag-point is sought; otherwise a tag-value s is obtained from X as described under TAG (see above) and the Ith most recent tag-point with tag-value s is sought ('0th most recent' means 'most recent'). If no existing tag-point satisfies these conditions, FAIL returns TRUE and takes no other action. Having isolated a tag-point, FAIL partially restores the stack environment of that tag-point as follows:

(1) Portions of the current stack environment added since the tag-point are deleted.

(2) The effect of the most recent tagged exit since the tag-point is reversed by restoring the portion of the stack environment of the tag-point deleted when the exit occurred. Note that a tagged block or procedure may have been exited by calling RETFROM or doing a non-local GOTO as well as in the normal way.

(3) The effect of the most recent NASSIGN call since the tag-point is reversed by restoring the old value of the left hand side.

(4) Parts (2) and (3) are repeated for each tagged exit or NASSIGNment since the tag-point (most recent ones first). The resulting stack environment is consistent with control being inside the original call on TAG prior to the creation of the tag-point and following the processing of TAG's argument.

(5) The form F is evaluated in this environment. The effect of a call on RETURN during this evaluation is that of such a call occurring within TAG. If the evaluation does not cause an exit from TAG, control passes to TAG which continues as usual. Note that this will result in the recreation of the tag-point and the return of the tag-value to TAG's calling environment.

UNTAG CEXPR(X:FORM UNEVAL, I:INT; BOOL)

UNTAG, like FAIL, isolates a tag-point (having no effect apart from a return of FALSE if no tag-point is found). It then modifies the current environment, with respect to subsequent FAIL's, to be as though (i) that and all succeeding tag-points had never been created and (ii) all calls on NASSIGN since that tag-point's creation had actually been calls on ASSIGN. It then returns TRUE.

7.10 MODE ROUTINES

:: SUBR(X:FORM UNEVAL, Y:MODE; MODE)

X must evaluate either to a symbol to be used as a shortname or to a list of user-defined mode functions. Details of the usage of this operator are given in section 4.1.3.

INFIX 175

LOWER SUBR(M:ANY, NEWM:MODE; ANY)

Returns the object M of mode NEWM if NEWM is explicitly provided, otherwise of the mode found in the UR field of the DDB of the mode of M. This is further described in section 4.4.

LIFT SUBR(M:ANY, NEWM:MODE; ANY)

Returns object M with mode NEWM. This is further described in section 4.1.4.

COVERS SUBR(A:MODE, B:MODE; BOOL)

Returns TRUE if mode A covers mode B. Mode A is said to cover mode B if and only if:

(1) A=B, or

(2) A=ANY, or

(3) A is of class generic and B is one of its alternatives, or

(4) A is of class generic and B is of class generic and all of B's alternatives are alternatives of A.

COMPLETE SUBR(F:FORM LISTED; BOOL)

Attempts to complete (see section 4.1.1) each mode in the list F of modes. Returns TRUE iff each mode is already complete or can be completed at this time, and FALSE otherwise.

CONSTRUCT SUBR(M:MODE, BC:SYMBOL, F:ANY; M)

Works just like CONST except the arguments are all evaluated. See section 4.2.1.4 for a description.

SELECT SUBR(V:ANY, L:FORM; ANY)

V is the object from which selection is made; L is the list of zero or more selector forms. The result is the component produced by successive application of the selectors to V. If L is NIL on entry and no USF is found for the object mode, a type fault will occur. SELECT has the same effect as [selection.

7.11 CONTROL ROUTINES

<< SUBR(M:FORM UNEVAL,F:FORM UNEVAL; ANY)

The FORM M must be either NIL, a MODE, or else a two element list whose first element is a quoted symbol and whose second element is a MODE. A mark record is created on the control stack and F evaluated within its scope. If there is no RETURN to the mark M, then the result of F's evaluation is coerced to the MODE associated with M and this yields the result of <<. For details and examples, see section 2.14.

PREFIX, INFIX 10

RETURN SUBR(V:ANY, N:SYMBOL, K:INT; NONE)

The value V is returned as the result of the Kth most recent mark record with name N. If no such record exists, there is a mark fault. For details and examples, see section 2.14.

8. SUPPORT PACKAGES

Support packages are collections of procedures which have been defined in EL1 and which are sufficiently useful to merit inclusion in this manual. Each subsection below describes a package found in a single file. LOADing that file makes the procedures available (along with a number of auxiliary procedures.)

8.1 DEBUGGING ROUTINES

File Name: TRACE.ECL

TRACE EXPR(FN:FORM UNEVAL, N:VECTOR(4,BOOL), INF:FORM, OUTF:FORM)

FN must be an identifier--the name of a procedure to be traced. As specified by N, that procedure may be modified to print out trace information, to call BREAK, or to perform some other user specified action as it is entered and as it exits. INF and OUTF indicate, respectively, the action to be taken on entering and exiting a traced procedure. Automatic conversion between INT and VECTOR(4, BOOL) permits several conditions to be specified simultaneously by N. Thus the call TRACE(P, 5) will result in N[3]=N[1]=TRUE and N[2]=N[4]=FALSE. The possibilities are as follows:

N[4] -> the names and values of the arguments are printed on each entry to FN
N[3] AND INF=NIL -> a call to BREAK (with typeout "CBREAK BROKEN") occurs just after entry to FN
N[2] -> the result computed by FN is printed prior to FN's exit
N[1] AND OUTF=NIL -> a call to BREAK (with typeout "CBREAK BROKEN") occurs just before exit from FN
INF # NIL -> INF is evaluated prior to exectuion of FN's body
OUTF # NIL -> OUTF is evaluated prior to exit from FN

In any case, the name of the procedure is printed at each entry ("FN CALLED") and exit ("FN RETURNING") along with a count of the depth of recursion of the invocation.

When a BREAK created by TRACE has been reached, the user may choose to proceed through several successive entries of the function before breaking there again. He may do so by executing CONT(n), where n is the number of BREAKs to be skipped. TRACE information will still be printed on routine entry. Other traced routines are not affected. CONT() is equivalent to CONT(0) at a TRACE-generated break.

UNTRACE EXPR(F:FORM UNEVAL)

F must be an identifier--the name of a procedure which is no longer to be traced. The original definition of F is restored.

NOTE: The top-level variable TRACEDFNS is a list of the names of all TRACEd procedures. TRACE adds names and UNTRACE removes names from this list.

UNTRACEALL EXPR( )

Calls UNTRACE on all elements of the list TRACEDFNS.

FLUSHTRACE EXPR( )

Deletes the TRACE package.

File Name: BT.BIN

BT CEXPR(TXTNM:INT, LOCNM:INT, FRMNM:INT, BEGNM:INT)

Prints on the user's command output port (COPORT) a backtrace of procedures, block, iterations, and marks that have been entered but not exited. Print order is most recent context first. TXTNM is the number of contexts corresponding to block activations for which the current statement will be printed (using UNPARSE if it is resident). In printing nested statements, '***' will be printed in place of an inner statement when printing an outer statement. Abs(LOCNM) local variable names and modes are printed; if LOCNM LT 0, then the values of the locals are printed as well. FRMNM

limits the total number of contexts to be backtraced and BEGNM specifies an initial number of contexts for which printing is to be omitted. If a backtraced context is an iteration statement for which control is nested within the header, this will be indicated by the typeout "FOR <PREFIX>". If a backtraced context is a marked evaluation with name ID and type M, the mark will be printed as "<< ID M".

NOTE: In the event of errors (e.g., stack overflow), do not use RESET or RETBRK. Instead, call RECOVER -- a local procedure of BT -- to exit from BT.

8.2 UNPARSING ROUTINES

File Name: UP.BIN

UNPARSE CEXPR(F:FORM, P:PORT, LVL:INT)

Converts the internal representation of the form F into a character string and outputs that string to the port P. The character string includes sufficient spaces, tabs, and line-feeds to make the rendering fairly readable. In particular, BEGIN-END blocks are indented, unneeded parentheses in sequences of binary operators are suppressed, and small expressions are kept on a single line where possible. A depth limit N is computed by

```
[)LVL=0 AND MD(...) = INT => ... ; LVL(]
```

If N is positive then text at a nesting depth greater than N (i.e., nesting of blocks and routine calls) will be elided if printing such text in full would require added vertical space. If N is non-positive then no elision occurs. If the third argument is defaulted but ... is bound to an integer, that integer is used. If a negative integer is supplied (either from LVL or ...), printing is not suppressed.

UNPARSF CEXPR(FI:SYMBOL, FO:SYMBOL, UPDATE:FORM LISTED)

Parses successive commands from the input file FI and unparses the resulting form onto the output file FO. FI and FO are expected to be either NIL or file designators (see description of OPEN, section 7.5). If either is NIL "TTY:"

will be assumed. The remaining arguments, comprising the list UPDATE, should all be identifiers. For each of these whose value is a FORM, ROUTINE (only if an EXPR), BASIC, or system mode (if the UR field is NIL), an assignment command is added to FO whose evaluation, when FO is subsequently loaded, will recreate that value. Appropriate fixity declarations are also added to the file for any EXPR being added. This updating facility is thus used to add new routines and constants to an existing file. Beware that any value of class PTR is unparsed as NIL. UNPARSF gives special treatment to the identifier "^". When ^; is encountered either as a top-level command in the input file or as a statement other than the last statement of a BEGIN or REPEAT block, UNPARSF inserts a page separator after the ^; in the output file.

UNPARSF2 CEXPR(FI:SYMBOL, FO:SYMBOL, UPDATE:FORM LISTED)

Like UNPARSF except that the update list is provided by the user as a single argument, e.g., UNPARSF2("X1", "X2", QL(A,B,C)).

UNPARSP CEXPR(PI:PORT, PO:PORT, UPDATE:FORM, F:BOOL)

Unparses from PI -- a PORT open for symbolic input. PO is not closed on exit and thus this routine may be used to concatenate files or to add definitions at the beginning of a file. On parse errors, the flag F is set to TRUE.

UNPARSFM CEXPR(F:FORM, P:PORT, LVL:INT, IND:INT)

Like UNPARSE except that no terminating semi-colon is printed and all lines of output except the first are indented IND spaces.

SET\UP CEXPR(W:INT, I:INT, H:INT, C:INT; FORM)

Sets the page frame size for subsequent unparser calls to W spaces wide by H spaces high. The incremental indentation is set to I. Comment forms using /* and */ are treated specially in unparsing according to the value of C. Multi-line string constants in these forms will be aligned at the first non-blank character of each line and comments with infix */ will be formatted with the string constant beginning in column C. If C LT 0 then comment formatting is inhibited. Any zero or defaulted arguments to

SET\UP cause no change in the corresponding parameter. SET\UP returns a "keyed" list of current values, e.g., initially this result is

(WIDTH 65 INDENT 2 HEIGHT 55 COMMENT 40)

FLUSHUP CEXPR( )

All unparsing routines and top-level bindings are deleted from the environment.

8.3 METERING ROUTINES

File name: METER.BIN

This package of routines provides a means for making various measurements on programs and functions. A count of the number of times a function is used can be kept, or the frequency of execution of each statement can be counted. The cumulative execution time of functions and programs can also be measured. In addition, there are routines to reinitialize or remove counters, and to print their values. The timing is accurate to about 33 msec.

METER EXPR(F:FORM, M\S:INT)

Takes a form and inserts counters specified by M\S into the list-structure representation of the form. The form must be an EXPR.

M\S	Purpose
1	CPU time(msec.)
2	frequency counter for entire form
3	1 & 2
4	frequency counter for each statement in the form (recursively)
5	1 & 4
11	1 with RETURN checking
13	3 with RETURN checking
15	5 with RETURN checking

If timing is specified , exit from the function must not be made by non-local RETURNs. For settings 11, 13, and 15, METER checks each RETURN and prints an error message if the matching << is not in the function. However, the function will be METERed! If the RETURN is illegal, you must UNMETER the function or the

timing for all functions will be erroneous.

The checking can take considerable time with large routines, but it can be avoided if you are certain there are no illegal RETURNs by using settings 1, 3, or 5.

TIME EXPR(F:FORM;INT)

Prints and returns the time counter for F in milliseconds. This is the time spent in F and all "untimed" functions called by F. The time spent in a "timed" function called by F is automatically subtracted from F's TIME, as is the time spent in the garbage collector.

COUNT EXPR(F:FORM; INT)

Prints and returns the frequency counter for the form F. Can be used on forms METERed with M\S values of 2, 3, 4, or 5.

TIMECOUNT EXPR(F:FORM; VECTOR(3, INT))

Prints and returns the time, frequency count and average time per call for the form F.

In order to see the frequency counts for all statements in a function, it is necessary to print the entire form. If you like to read list-structure you can type <function name>$. The frequency count associated with each statement will be printed in parentheses before that statement. For a more readable print-out, use the UNPARSEing routines described in section 8.2. The unparser prints the frequency count in angle brackets on the same line as the associated statement, as shown below. Some additional BEGIN-END blocks will have been added to the function by METER in order to conform to some ECL syntax constraints, so do not be disturbed if the METERed version differs from the original.

The following UNPARSEd listings show a function before being METERed, and after being METERed with M\S = 5, and applied to the integer 12.

```
EXPR(I:[) INT (]; BOOL)
   BEGIN
      DECL K:INT;
      OR(I = K, I GT 13, I / 3 * 3 = I) ->
         K <- I / 3;
      TO 15 REPEAT I <- I - K END;
      I LT 0;
   END;

EXPR(I:[) <1>; INT (]; BOOL)
   (TIME(83,
    BEGIN
    <1>; BEGIN
               DECL K:INT;
         <1>; OR([) <1>; I = K (],
              [) <1>; I GT 13 (],
              [) <1>; I / 3 * 3 = I (]) ->
              [) <1>; K <- I / 3 (];
         <1>; TO 15
              REPEAT <15>; I <- I - K END;
         <1>; I LT 0;
         END;
    END));
```

REMETER EXPR(F:FORM)

Sets all counters in the form F to zero.

UNMETER EXPR(F:FORM)

Removes all metering artifacts from F (including the additional blocks), leaving F the same as it was before being METERed.

TIMESTACK EXPR(N:INT)

Allows the user to pick the size of the timing stack. One location is used for each level of call to a function being timed. The stack size is initially set to 50. TIMESTACK calls RESET, thereby returning control to the top level.

METERF EXPR(S:SYMBOL, L:FORM, M\S:INT)

Opens and loads the command file specified by S, METERing each EXPR in the file as specified by M\S. A list of the functions metered is assigned to L.

The following routines take a form which is a list of function names and perform the implied action on each function: METERL, REMETERL, UNMETERL, TIMEL, COUNTL, TIMECOUNTL, and UNPARSEL.

In addition, TIMEL (COUNTL) prints the percent of time spent in this function (percent of calls to this function) relative to the total for the list of functions. It also returns the total time (number of calls) for the functions on the list. TIMECOUNTL prints TIMECOUNT information plus these two percentages for each function, and returns a VECTOR(3,INT) of total time, total frequency and average time per call.

Note: If you are finished with the entire package, call FLUSHMETER() to remove all METER.BIN functions. In using this package one usually applies METER or METERF several times in the beginning of a session to add counters to functions, and from then on uses only REMETER and UNMETER to modify and remove counters. If you are not going to use the functions METERF and METER for the remainder of a session, you can save space by evaluating

```
FLUSH(METER,METERF)
```

8.4 HASHING ROUTINES

8.4.1 Using the Hashing Package

The ECL hash coding facility consists of a built-in mode, HASHTABLE, and a group of routines for building and using hash tables. In TENEX ECL the hashing routines are built into the system; at Harvard they must be obtained by evaluating LOADB "SYS:HASH".

Hash coding is a technique for associative data storage and retrieval. Hash tables pair data items with key values. Both the keys and the data may be of any length-resolved mode, though all keys for a given table must have the same mode, as must all data values. Given a hash table H and a key value K of the appropriate mode, the programmer can determine whether a datum D is stored in H under key K. If so, he can read and/or modify D, or he can delete it from the table. If not, he can make a new entry in H for K. There are also routines to create hash tables, and to expand or contract them.

It is sometimes useful to build hash tables holding no data other than the key values themselves; the presence or absence of a particular key is often sufficient "data".

ECL's facility allows for this possibility by permitting the key and data entries to be identical if the user so specifies when creating the table.

The function which hashes a key value to produce the internal table index from which the search for a matching key entry begins is built into the system. However, the user has two means of controlling the lookup process. First, he can provide a key transformation function, called the key deriver. The key deriver is useful for isolating the representative feature of the input key value to be used in the search for a matching table entry. The key deriver is applied to the input key before the initial hash probe is computed, and it is applied to each table key entry before comparison with the (derived) input key during lookup. Thus, for example, if the (raw) key mode is REF and the key deriver is the function VAL, effective key values of any mode can be entered in one table by providing raw keys which are pointers to them.

The second way the user can control lookup is by providing the key comparison function itself. Normally, ECL's EQUAL operator is used to compare (derived) input keys with (derived) table keys. The comparator may be superseded by any binary operator capable of accepting key values and returning a BOOL. The user-provided comparison function might, for example, test two list structures for equality to arbitrary depth.

Each hash table has an associated load factor, L, expressed as an integer percentage between 1 and 100. Whenever an attempt is made to fill the table more than L per cent full, table overflow occurs. Ordinarily, overflow will cause the table to be expanded by one-half its current size before the new entry is inserted. However, the user may provide a routine to be called in case of overflow. The overflow routine will be passed a HASHTABLE and an INT approximately equal to 1.5 times the current table size.

The hashing package contains the following routines.

```
MAKEHASH  CEXPR(KEY\MODE:MODE,
                DATA\MODE:MODE,
                TABLE\SIZE:INT,
                LOAD\FACTOR:INT,
                DERIVER:ANY,
                COMPARATOR:ANY,
                OVERFLOW:ANY;
                HASHTABLE)
```

MAKEHASH builds hash tables. KEY\MODE and DATA\MODE should be length-resolved and non-GENERIC, but they are otherwise unrestricted. If DATA\MODE is NIL, MAKEHASH assumes that there will be no data values in the hash table, only key values. TABLE\SIZE may be any non-negative integer. LOAD\FACTOR may lie between 0 and 100; however, a zero LOAD\FACTOR is assumed to indicate default and is translated to a value of 80. The user-provided key deriver, key comparison function, and table overflow function have formal mode ANY so that strong PROCedure modes can be preserved, if the user supplies them. NOTHING or NIL are permissible default indicators in these argument positions. If the key deriver is defaulted, the raw input and table keys are used for probing. As mentioned above, the default COMPARATOR is EQUAL, and the default overflow handler is REHASH (see below).

```
FINDHASH  CEXPR(TABLE:HASHTABLE,
                KEY:ANY,
                NOCREATE\FOUND:BOOL;
                ANY)
```

FINDHASH looks up the data value for KEY in TABLE. If an entry is found it is returned as the result of the call to FINDHASH. The data value returned will be identical to (shared with) the table entry so that it can be modified. If no matching key is found in TABLE, the input value of NOCREATE\FOUND determines what FINDHASH does. If it is TRUE, no new entry is made; FINDHASH returns a pure (unshared) default value of the data mode for TABLE. If NOCREATE\FOUND is FALSE on entry, but KEY has not yet been entered, a new entry is created. Its key field is filled with the raw KEY; its data field (initialized to the default value of its mode) is returned shared, to be filled by the calling program. If TABLE has only key entries (DATA\MODE argument to MAKEHASH was NIL), the result of a call to FINDHASH which creates a new entry will be the value KEY itself.

NOCREATE\FOUND is also an output variable of FINDHASH. It is always set TRUE if a match for KEY is found in TABLE, and FALSE otherwise.

FLUSHASH CEXPR(TABLE:HASHTABLE, KEY:ANY)

FLUSHASH calls FINDHASH to determine whether KEY has an entry in TABLE. If so, FLUSHASH removes it. If not, FLUSHASH does nothing.

REHASH CEXPR(TABLE:HASHTABLE, TABLE\SIZE:INT BYVAL; HASHTABLE)

REHASH expands or contracts TABLE as necessary to make its new capacity TABLE\SIZE. A zero TABLE\SIZE is taken as a default indicator; in this case TABLE is rehashed but not changed in size. If the new TABLE\SIZE would shrink TABLE so far that its load factor would be exceeded, no rehashing takes place, and an 'ILLEGAL ARG' break is generated instead.

REHASH calls FINDHASH for each occupied entry of table. REHASH may itself be called internally from FINDHASH for either of two reasons. First, REHASH is the default routine for handling table overflow; it is called to expand a table if there is no user-provided overflow routine, or if the user-provided routine fails to enlarge the table. Second, since key values may include pointer data, the integrity of a hash table may be disturbed when objects are moved around in the heap. This can happen either during compactifying garbage collection or as the result of a hash table's having been output and then reloaded. When FINDHASH discovers that a table has been so compromised, it applies REHASH to the table before starting its probe. Suitable precautions are taken against unbounded recursive loops between FINDHASH and REHASH.

8.4.2 Technicalities

A HASHTABLE has the following components:

KEYS:REF	Points to a SEQuence of key values.
DATA:REF	Points to a SEQuence of data values.
KEYFN:REF	Points to the key deriver. NIL when there is no deriver.
EQFN:REF	Points to the comparison function, either the EQUAL SUBR or a user-provided routine.
OVFLOWFN:REF	Points to the user-provided table overflow handler; NIL when none is supplied.
LINK:HWD	Used by the garbage collector.
LOADF:HWD	Load factor for the table.

COUNT:INT The number of occupied table entries.
CASENM:HWD Field used to designate tables treated specially by FINDHASH.
KEYBIT:BOOL TRUE if the raw input key is to be transformed by the key deriver.
TABBIT:BOOL TRUE if table key values are to be transformed by the key deriver.
ZIPBIT:BOOL TRUE if the table entry for the default key value is occupied.
CGCBIT:BOOL TRUE if the table needs to be rehashed before the next attempt to use it.
RHSBIT:BOOL TRUE unless table validity cannot depend on the positions of objects in the heap.

[KEYS, DATA] When MAKEHASH is given a NIL DATA\MODE, it creates a table whose KEYS and DATA fields both point to the same sequence of items.

In general, the KEYS and DATA fields point to sequences of length TABLE\SIZE + 2, where TABLE\SIZE is the capacity specified in the call to MAKEHASH or the last call to REHASH for the table. The two extra entries are to allow for special handling of the default key value and to accomodate a search algorithm which requires at least one vacant table entry at all times. The actual condition for overflow of a hash table H, therefore, is that

```
(H.COUNT + 1) * 100 GT
  (LENGTH(H.KEYS)=2)*CONST(INT BYVAL H.LOADF)
```

just before a new entry is to be made.

[KEYFN, EQFN, OVFLOWFN] Each user-provided function is verified by MAKEHASH to be a pointer to an EXPR, a CEXPR, or a SUBR. The REF in the table will point to the user-supplied code-pointer. That way, strong PROCedure modes are not lost, as they would be by coercion to mode ROUTINE or REF. When the value provided is already in the heap, it will be shared. Otherwise, of course, it must be copied into the heap so that it can be pointed at. Note that while the KEYFN and OVFLOWFN fields are left NIL by default, the EQFN field gets a direct pointer to the EQUAL SUBR.

Because user-defined functions can be called during the lookup process, there is the possibility that a table will be invalidated by a call to REHASH while it is being searched. To

guard against this, FINDHASH increments a counter, PROBE\LEVEL, at the beginning of the lookup, and decrements it at the end. REHASH zeroes the same counter each time it is called. If FINDHASH discovers that PROBE\LEVEL is negative after being decremented, and if the lookup has been unsuccessful, PROBE\LEVEL is cleared and the whole lookup restarted. If the problem recurs, FINDHASH causes a (continuable) error break with the message 'REPEATED INVALIDATION DURING HASHING.' The same error can be generated by REHASH if compactifying garbage collection repeatedly forces rehashing to be restarted.

[CASENM] Two types of hash tables are given special treatment because they occur frequently and can be handled more efficiently than hash tables in general. Both cases cover tables with no key deriver and for which EQUAL is the comparison function. The first (CASENM = 1) includes tables with keys which are halfwords, e.g. FORM's, HWD's, or PTR(M)'s. The second (CASENM = 2) covers fullword keys, e.g. INT's, REF's, and PTR(BOOL, CHAR)'s. A mode M describes a halfword object if NOT M.WDFLG AND (CONST(INT BYVAL M.SZ) = 18). M describes a fullword object if M.WDFLG AND M.LRFLG AND (CONST(INT BYVAL M.SZ) = 1). At present, all other types of tables have CASENM = 0.

[KEYBIT, TABBIT] MAKEHASH sets both KEYBIT and TABBIT if a KEYFN is provided. The user may clear one or the other if he wishes. It might, for example, be most efficient for table key derivation to be handled inside the user-supplied comparison routine. If so, TABBIT may be set to FALSE after the table is created. The table key, possibly transformed by the deriver, will be the first argument to the comparison function.

[ZIPBIT] Unoccupied table entries are distinguished by key fields which carry the default value of the key mode. In order to permit this default value itself to be used as a key, the first entry in every hash table is reserved for this value. ZIPBIT will be set TRUE whenever this default key entry is occupied.

8.5 EXTENDED BINARY INPUT/OUTPUT

File Name:RW.BIN

This package, loaded by LOADB"SYS:RW", is an ECL extension containing several routines which may be used for binary input-output of data objects of any mode not containing embedded pointers. This restriction would exclude data of modes MODE and SYMBOL, except for the fact that a special file representation is adopted for these two cases, consisting of a character string. User mode functions are not handled, however.

The routine ROPEN takes two arguments and returns a port of mode RWPORT. The first argument is the file name and the second the end of file form. The file is opened and initialized for use by RDOBJ. WOPEN takes just a filename and opens a RWPORT for use by WROBJ.

The arguments to WROBJ are the data object to be output and the RWPORT to be used. In the case of an object whose mode is length-unresolved, the object is preceded in the file by information describing its dimensions.

The arguments to RDOBJ are the mode of the next object in the input file and its RWPORT. As in the case of INOBJ, heap generation is used.

The mode RWPORT has a user assignment function which is essentially WROBJ and a user conversion function which is essentially RDOBJ. This means that if P and Q are RWPORTs, the assignments

```
P <- X ;
Y <- Q ;
```

are equivalent to

```
WROBJ(X,P) ;
Y <- INOBJ(Y,Q) ;
```

Note, however, that if the mode of Y is length-unresolved, the dimension vector for Y must match that in the file or an error break will occur.

If a symbol is placed in the output file, RDOBJ will return the same symbol. If a mode is placed in the output file, the mode, except for any user mode functions present at output time, will be defined by RDOBJ, provided that all component modes are then defined.

The RWPORT must be closed by use of the routine RWCLOSE.

```
<Command>          ::=  <Form> ALT                              @ <Form>
                      | <Form> ;                                @ <Form>
                      | ;                                       @ NIL
                      | ALT                                     @ NIL

<Form>             ::=  <PrefixForm>                            @ <PrefixForm>
                      | <PrefixForm> INFIXOP <Form>             @ (INFIXOP <PrefixForm> <Form>)
                      | EXPR (<Formal*> <ResultType>) <Form>    @ (EXPR <Formal*> <ResultType> <Form>)
                      | <For> <Initial> <Step> <Limit>
                        REPEAT <Statement*> <SemiColon> END     @ (FOR)+<For>+<Initial>+<Step>+<Limit>+<Statement*>

<For>              ::=  FOR ID                                  @ (FOR ID)
                      | ε                                       @ NIL

<Initial>          ::=  BINDCLASS <Form>                        @ (BINDCLASS <Form>)
                      | ε                                       @ NIL

<Limit>            ::=  TO <Form>                               @ (TO <Form>)
                      | ε                                       @ NIL

<Step>             ::=  BY <Form>                               @ (BY <Form>)
                      | ε                                       @ NIL

<PrefixForm>       ::=  PREFIXOP <PrefixForm>                   @ (PREFIXOP <PrefixForm>)
                      | <FunctionCall>                          @ <FunctionCall>

<FunctionCall>     ::=  <FunctionCall> (<Form*>)                @ <FunctionCall> & <Form*>
                      | <LimitedForm>                           @ <LimitedForm>

<LimitedForm>      ::=  PROC (<ProcArgument*> <ResultType>)     @ (PROC <ProcArgument*> <ResultType>)
                      | CONSTANT                                @ CONSTANT
                      | ID                                      @ ID
                      | <CompoundForm>                          @ <CompoundForm>
                      | <FunctionCall> . ID                     @ (. <FunctionCall> ID)
                      | <FunctionCall> [<Form*>]                @ ([])+(<FunctionCall>)+<Form*>
                      | CONST (<Form> <Init>)                   @ CONST & <Form> & <Init>
                      | STRUCT (<Component+>)                   @ STRUCT & <Component+>
                      | LEFTMATCHOP <Form*> RIGHTMATCHOP        @ LEFTMATCHOP & <FORM*>

<CompoundForm>     ::=  BEGIN <Statement*> <SemiColon> END      @ BEGIN & <Statement*>
                      | (<Form>)                                @ <Form>
                      | CASE <Relations> [<Form*>]
                        <CaseStatement*> <SemiColon> END        @ CASE & <Relations> & <Form*> & <CaseStatement*>

<Statement*>       ::=  <Statement>                             @ (<Statement>)
                      | <Statement*>; <Statement>               @ <Statement*>+(<Statement>)
                      | ε                                       @ NIL

<Statement>        ::=  <Form>                                  @ <Form>
                      | <Form> => <Form>                        @ (=> <Form> <Form>)
                      | <Decl+>                                 @ <Decl+>

<Decl+>            ::=  DECL <ID+>: <Form> <Init>               @ (DECL (<ID+> <Form> <Init>))
                      | <Decl+> DECL <ID+>: <Form> <Init>       @ <Decl+>+((<ID+> <Form> <Init>))

<CaseStatement*>   ::=  <CaseLHS> => <Form>                     @ ((=>  [] & <CaseLHS>  <Form>))
                      | <CaseStatement*>; <CaseLHS> => <Form>   @ <CaseStatement*>+((=>  [] & <CaseLHS>  <Form>))
                      | ε                                       @ NIL

<CaseLHS>          ::=  <CaseLHS1> <CaseLHS2>                   @ (<CaseLHS1> <CaseLHS2>)
                      | <CaseLHS>, <CaseLHS1> <CaseLHS2>        @ <CaseLHS>+(<CaseLHS1> <CaseLHS2>)

<CaseLHS1>         ::=  [<Form*>]                               @ <Form*>
                      | ε                                       @ NIL

<CaseLHS2>         ::=  <Form>                                  @ <Form>
                      | ε                                       @ NIL

<Component+>       ::=  ID: <Form>                              @ ((ID <Form>))
                      | <Component+>, ID:<Form>                 @ <Component+>+((ID <Form>))

<Form*>            ::=  <Form+>                                 @ <Form+>
                      | ε                                       @ NIL

<Form+>            ::=  <Form>                                  @ (<Form>)
                      | <Form+>, <Form>                         @ <Form+>+(<Form>)

<Formal*>          ::=  <Formal+>                               @ <Formal+>
                      | ε                                       @ NIL

<Formal+>          ::=  ID: <Form> BINDCLASS                    @ ((ID <Form> BINDCLASS))
                      | <Formal+>, ID: <Form> BINDCLASS         @ <Formal+>+((ID <Form> BINDCLASS))

<ID+>              ::=  ID                                      @ (ID)
                      | <ID+>, ID                               @ <ID+>+(ID)

<Init>             ::=  BINDCLASS <Form+>                       @ BINDCLASS & <Form+>
                      | ε                                       @ NIL

<ProcArgument*>    ::=  <ProcArgument+>                         @ <ProcArgument+>
                      | ε                                       @ NIL

<ProcArgument+>    ::=  <Form> BINDCLASS                        @ ((<Form> BINDCLASS))
                      | <ProcArgument+>, <Form> BINDCLASS       @ <ProcArgument+>+((<Form> BINDCLASS))

<Relations>        ::=  (<Form*>)                               @ <Form*>
                      | ε                                       @ NIL

<ResultType>       ::=  ; <Form>                                @ <Form>
                      | ε                                       @ NONE

<SemiColon>        ::=  ;                                       @ NIL
                      | ε                                       @ NIL
```

Note that in this grammar the symbols INFIXOP, PREFIXOP, ID, LEFTMATCHOP, RIGHTMATCHOP, and BINDCLASS are not to be taken literally but, rather, stand for the classes of symbols with the appropriate lexical properties (as set by MATCHFIX, PREFIX, etc.). Note also that we have, for convenience of presentation, used BINDCLASS instead of its individual elements which have somewhat more restricted syntactic properties than the grammar indicates. Specifically, only FROM, LIKE, SHARED, and BYVAL are permitted in REPEAT headers, OF and SIZE are not permitted in <ProcArgument+>, and UNEVAL and LISTED are not permitted in declarations and CONSTructions. Though only CONST appears in the grammar, ALLOC may be used in its place.

A.1 Meta-Operators in the Augments

Augments perform one of four actions on internal list structure:

(1) <name> signifies direct promotion.

```
<LimitedForm> ::= <CompoundForm>   @ <CompoundForm>
```

means that the list structure specified by <CompoundForm> is to be associated with <LimitedForm>.

(2) The list structure may be given explicitly.

```
<LimitedForm> ::= <FunctionCall> . ID
                  @ (. <FunctionCall> ID)
```

means that the list structure to be associated with <LimitedForm> is a list of three elements, the symbol ".", the list structure specified by <FunctionCall>, and the list structure specified by ID. (Note: In all cases, the list structure specified by a terminal of the grammar, such as ID, is the atomic form for that particular instance; in the case of ID, it is the actual identifier name.)

(3) The list structure may be obtained by CONSing list structure together, using the & infix operator.

```
<Init> ::= BINDCLASS <Form+> @ BINDCLASS & <Form+>
```

means that the list structure to be associated with <Init> is obtained by CONSing BINDCLASS onto the head of the list structure specified by <Form+>.

(4) The list structure may be obtained by concatenating list structure together, using the + infix operator.

<Form+> ::= <Form+>, <Form> @ <Form+>+(<Form>)

means that the new list structure to be associated with <Form+> is obtained by appending the list presently specified by <Form+> to the explicit list (<Form>). For example, if <Form+> is (1 2 3) and (<Form>) is (X), then the resulting <Form+> is (1 2 3 X). We further specify the actions of + when the operands are NIL.

NIL + (a) = (a) + NIL = (a)
NIL + NIL = NIL

A.2 Examples of Internal Representation

Form	Internal Representation
`a + b`	`(+ a b)`

`BEGIN`
`  DECL X:INT BYVAL 2;`
`  TRUE => X;`
`END`
	`(BEGIN`
	`  (DECL ((X) INT BYVAL 2))`
	`  (=> TRUE X))`

`X[5]`	`([ X 5)`
`FOO.LHS`	`(. FOO LHS)`
`FUNCT(ARG1, ARG2, ARG3)`	`(FUNCT ARG1 ARG2 ARG3)`
`FOR I FROM 10 BY 2 TO 20` `  REPEAT F END`	`(FOR FOR I FROM 10 BY 2` `  TO 20 REPEAT F)`

`EXPR(X:INT, Y:REAL BYVAL;BOOL)`
`  (X-Y)`
	`(EXPR((X INT LIKE)`
	`  (Y REAL BYVAL))`
	`  BOOL`
	`  (- X Y))`

`CASE[COVERS, GT] (MD(A), B)`
`  [INT, 5]P => R1`
`  [REAL, 1]Q, [ANY, 2] => R2;`
`  TRUE => R3;`
`END`
	`(CASE (COVERS GT) ((MD A) B)`
	`  (=> ([] (INT 5) P) R1)`
	`  (=> ([] (REAL 1) Q`
	`  (ANY 2) ())`
	`  R2) ( )`
	`  (=> ([] () TRUE) R3))`

`<< BEGIN`
`  DECL X:INT;`
`  X<-Y;`
`  Y<-0;`
`  RETURN(X);`
`  END`
	`(<< (BEGIN`
	`  (DECL ((X) INT))`
	`  (<- X 0)`
	`  (<- Y X)`
	`  (RETURN X)))`

APPENDIX B: BUILT-IN DATA HANDLING ALGORITHMS

This appendix gives precise specifications for ECL's built-in data manipulation algorithms. It should be used in conjunction with section 4 to obtain an accurate picture of when and how extended mode attributes are used. Insofar as possible, this description is not dependent on the existing implementation of ECL. To make the points of dependence clear, however, the appendix is in three sections. Routines in section B.1 are implementation independent; they should be equally valid in any implementation of EL1. Those in section B.2 reflect particular representation choices in ECL. They would be different for an implementation with a different representation of modes, for example. Section B.3 describes functions used in B.1 and B.2 but not expressible in EL1 without circularity or other obfuscation.

These model algorithms describe ECL only in an abstract sense: they are alternative realizations of the same semantics. They can be trusted to describe, for example, which behavior functions will be called during data generation, how often, and with what arguments. They correctly model error situations, including the disposition of values returned through CONT or STEP following a break. But they should not be construed as a commitment to a particular order of evaluation, except where that is obvious from data dependences.

Routines whose names are prefixed by "PHYSICAL\" disregard extended mode attributes and deal only with machine representations of data. Those prefixed by "SYSTEM\" are the built-in or default algorithms that take over in the absence of a user-defined function for a particular situation. "SYSTEM\" routines nevertheless respect all extended data properties, and so they may cause user-defined functions to be invoked during their own activations.

B.1 IMPLEMENTATION INDEPENDENT DATA ALGORITHMS

```
' - - -  SYSTEM\GENERATE  - - -
!
! BUILT-IN DATA GENERATION ALGORITHM.  USED IN SECTION 4.3.
!
! ARGUMENTS:
!
!   EM    EXPECTED MODE.
!   BC    BINDCLASS SYMBOL ("BYVAL", "SIZE", OR "OF").
!   SPEC  SOURCE VALUE WHEN BC = "BYVAL",
!         DIMENSION VECTOR (SEQ(INT)) WHEN BC = "SIZE",
!         EXPLICIT COMPONENT FORM LIST WHEN BC = "OF".
!
! ASSUMPTIONS:
!
!   WHEN BC = "BYVAL", MD(SPEC) = EM.
!
!   WHEN BC = "SIZE", MD(SPEC) = SEQ(INT) AND SPEC  IS  A  PROPER
!   DIMENSION VECTOR FOR EM: LENGTH(SPEC) = NAD(EM).
!
!   WHEN BC = "OF", MD(SPEC) = FORM AND SPEC IS A  LIST  (PERHAPS
!   NULL) OF EXPLICIT COMPONENT FORMS.
!
!   NO OTHER BINDCLASS WILL OCCUR.
!
! METHOD:
!
!   ERRONEOUS AND TRIVIAL CASES ARE  HANDLED  FIRST:  USE  OF  AN
! INCOMPLETABLE  MODE  AND THE ATTEMPT TO GENERATE A NON-COMPOUND
! OBJECT FROM COMPONENTS ARE ERRORS; GENERATION  OF  NON-COMPOUND
! OBJECTS "BYVAL" AND BY "SIZE" ARE SIMPLE SINCE NO MODE BEHAVIOR
! FUNCTIONS ARE INVOLVED.
!
!   OTHERWISE, THE OBJECT IS PRODUCED BY A RECURSIVE  SUBROUTINE,
! SG2.   THE  I-TH  NESTED  CALL  UPON  SG2  GENERATES  THE  I-TH
! COMPONENT, CREATES THE OBJECT  WITH  ELEMENTS  ABOVE  THE  I-TH
! FILLED  IN,  INSERTS  THAT  COMPONENT AND RETURNS THE PARTIALLY
! FILLED OBJECT.
!
!   NESTED INVOCATIONS OF SG2 SHARE A PHYSICAL  DIMENSION  VECTOR
! (PDV) FOR THE EXPECTED MODE.  AS COMPONENT VALUES ARE PRODUCED,
! THEIR PHYSICAL DIMENSIONS ARE EXTRACTED AND USED TO  FILL  PDV.
! THE  DEEPEST  CALL  TO SG2 USES PDV TO BUILD A SKELETON OBJECT,
! WHICH IS THEN  FILLED  AS  IT  IS  RETURNED  THROUGH  ENCLOSING
! ACTIVATIONS.
!
';

^;
```

```
!
!   SG2 RECEIVES THE OBJECT SPECIFICATION VIA SEVERAL ARGUMENTS:
! A SOURCE VALUE (SV) FOR THE "BYVAL" CASE, A COMPONENT LIST (CL)
! FOR "OF", AND AN APPARENT DIMENSION VECTOR (ADV) TAILORED TO
! MODE EM AND USED BOTH WHEN BC = "SIZE" AND SOMETIMES WHEN BC =
! "OF". EACH COMPONENT IS PRODUCED IN ONE OF FOUR WAYS: WHEN SV
! IS GIVEN [BC = "BYVAL"], ITS COMPONENTS ARE REPLICATED. IF CL
! IS GIVEN [BC = "OF"], THEN ITS ELEMENTS ARE EVALUATED, WHILE
! THEY LAST, AND COERCED TO THE APPROPRIATE COMPONENT MODES.
! WHEN CL IS EMPTY [BC = "SIZE" OR BC = "OF"], ADV PROVIDES
! DIMENSIONS FOR COMPONENT GENERATION BY SIZE. FINALLY, IN
! CERTAIN CASES OF ROW GENERATION, COMPONENTS WILL BE PRODUCED BY
! REPLICATION OF EARLIER COMPONENTS. FOR THIS PURPOSE, EACH
! RECURSIVE ACTIVATION OF SG2 IS PASSED A REFERENCE TO THE LAST
! COMPONENT (LC) CREATED.
!
!   THE MAIN ROUTINE SETS UP INITIAL CONDITIONS FOR SG2 BASED ON
! THE BINDCLASS BC AND ON WHETHER OR NOT EM IS A SEQUENCE MODE.
! THE SOURCE VALUE, COMPONENT LIST, AND DIMENSION INPUTS ARE
! INITIALIZED IN THE OBVIOUS WAY:
!
! SV   SET TO SPEC WHEN BC = "BYVAL"; OTHERWISE, TO NOTHING.
! CL   SET TO SPEC WHEN BC = "OF"; OTHERWISE, TO NIL.
! ADV  SET TO SPEC WHEN BC = "SIZE"; OTHERWISE, TO DADV(EM).
!
! FOUR INTEGER PARAMETERS TO SG2 ALSO DEPEND ON THE
! BINDCLASS/MODE-CLASS BREAKDOWN:
!
! CLI  COMPONENT LIST INDEX. SHARED BY ALL SG2 CALLS. BEGINS AT
!      ZERO AND IS INCREMENTED BY ONE FOR EACH EXPLICIT ELEMENT
!      IN CL. WHEN BC = "OF" AND EM IS A SEQUENCE MODE, CLI IS
!      SHARED WITH PDV[1] SO THAT THE FIRST PHYSICAL DIMENSION IS
!      FILLED IN AUTOMATICALLY AS THE COUNT IS MADE. OTHERWISE
!      CLI IS AN INDEPENDENT INTEGER.
!
! NC   NUMBER OF COMPONENTS. IF EM IS A VECTOR OR STRUCT MODE,
!      NC IS SET TO THE INTRINSIC OBJECT LENGTH. IF
!      IS\SEQ\MODE(EM), NC IS SHARED WITH PDV[1]. WHEN BC =
!      "SIZE", THEIR COMMON VALUE IS SPEC[1], THE FIRST SUPPLIED
!      DIMENSION. WHEN BC = "BYVAL" THEY ARE SET TO THE LENGTH
!      OF THE SOURCE VALUE. OTHERWISE, NC AND PDV[1] ARE SHARED
!      WITH CLI, THE COMPONENT FORM COUNTER.
!
! ADVI(PDVI) POINTER TO ADV(PDV). ALWAYS INDEXES THE ELEMENT
!      LAST READ(FILLED). STARTS AT ZERO UNLESS IS\SEQ\MODE(EM).
!      IN THE SEQUENCE CASE, STARTS AT ONE SINCE THE FIRST
!      DIMENSION OF A SEQUENCE IS ITS LENGTH, WHICH IS HANDLED
!      SPECIALLY THROUGH NC.
!
;

;
```

```
SYSTEM\GENERATE <-
 EXPR(EM:MODE, BC:SYMBOL, SPEC:ANY; ANY)
  BEGIN
   NOT COMPLETE(EM) =>
    FATAL\ERROR("UNRESOLVED MODE IN MODE DEF");
   NOT IS\COMPOUND\MODE(EM) =>
    BEGIN
     BC = "BYVAL" => PHYSICAL\PURIFY(SPEC);
     BC = "SIZE" => PHYSICAL\GENERATE(EM);
     FATAL\ERROR("CAN'T GEN OBJ");
    END;
   DECL SV:ANY LIKE [) BC = "BYVAL" => SPEC; NOTHING (];
   DECL CL:FORM BYVAL [) BC = "OF" => SPEC; NIL (];
   DECL ADV:SEQ(INT) BYVAL [) BC = "SIZE" => SPEC; DADV(EM) (];
   DECL PDV:SEQ(INT) SIZE NPD(EM);
   DECL CLI:INT SHARED
    [) IS\SEQ\MODE(EM) AND BC = "OF" => PDV[1]; 0 (];
   DECL NC:INT SHARED
    BEGIN
     IS\SEQ\MODE(EM) #> NUMBER\COMPONENTS(EM);
     BC = "OF" => CLI;
     PDV[1] <- [) BC = "SIZE" => SPEC[1]; LENGTH(SV) (];
    END;
   DECL DVI:INT LIKE [) IS\SEQ\MODE(EM) => 1; 0 (];
   SG2(EM, NC, 1, SV, CL, CLI, ADV, DVI, PDV, DVI, NOTHING);
  END;

'- - -  PHYSICAL\PURIFY  - - -
!
! MAKES A COPY OF OBJECT V WITHOUT PERMITTING  USER-DEFINED  MODE
! BEHAVIOR FUNCTIONS TO GAIN CONTROL.
!
! USED IN SECTION 4.3.1 AND IN SYSTEM\GENERATE, ABOVE.
!
';

PHYSICAL\PURIFY <-
 EXPR(V:ANY; ANY)
  LIFT(PHYSICAL\GENERATE(BASE(MD(V)),
                         PHYSICAL\DIMENSIONS(V)) <-
       LOWEST(V),
      MD(V));

^;
```

```
´ - - -  SG2  - - -
!
! SUBROUTINE OF SYSTEM\GENERATE ...  CREATES ONE COMPONENT VALUE,
! C,  RECURSIVELY  BUILDS THE REQUIRED OBJECT, V, WITH COMPONENTS
! BEYOND C FILLED IN, THEN INSERTS C AND RETURNS V.
!
! ARGUMENTS:
!
!   EM     EXPECTED MODE (ASSUMED COMPLETE AND COMPOUND).
!   NC     NUMBER OF COMPONENTS
!          (SHARED WITH CLI FOR SEQS IF BC = "OF").
!   CI     INDEX OF CURRENT COMPONENT (STARTS AT 1).
!   SV     SOURCE VALUE
!   CL     EXPLICIT COMPONENT LIST (POSSIBLY EMPTY).
!   CLI    NUMBER OF EXPLICIT COMPONENTS SO FAR (STARTS AT 0).
!   ADV    APPARENT DIMENSION VECTOR (LENGTH = NAD(EM)).
!   ADVI   INDEX OF LAST ADV ELEMENT USED.
!   PDV    PHYSICAL DIMENSION VECTOR (LENGTH = NPD(EM)).
!          (PDV[1] IS SHARED WITH CLI FOR SEQS WHEN CL #  NIL.)
!   PDVI   INDEX OF LAST PDV ELEMENT FILLED.
!   LC   LAST COMPONENT VALUE CREATED.
!
! SPECIAL CASES:
!
!   IF THE EXPLICIT COMPONENT LIST IS LONGER THAN NECESSARY,  THE
! EXTRA ELEMENTS ARE EVALUATED BUT DROPPED WITHOUT COERCION.
!
!   IF EM IS A  VECTOR  MODE,  AND  THERE  IS  A  COMPONENT  LIST
! INITIALLY,  BUT  IT  IS SHORTER THAN NEEDED, THE FIRST IMPLICIT
! COMPONENT IS  CREATED  BY  SIZE  FROM  THE  DIMENSIONS  OF  THE
! PREVIOUS  (I.E.  LAST  EXPLICIT)  ONE.  OTHERS  ARE  THEN
! REPLICATED.  THE  TRANSITION  IS  EFFECTED  BY  PUTTING  THE
! DIMENSIONS  OF  THE  LAST  EXPLICIT  COMPONENT  INTO  ADV  AND
! RESETTING ADVI TO 0.
!
´;

^;
```

```
SG2 <-
 EXPR(EM:MODE,
      NC:INT,
      CI:INT,
      SV:ANY,
      CL:FORM BYVAL,
      CLI:INT SHARED,
      ADV:SEQ(INT) SHARED,
      ADVI:INT BYVAL,
      PDV:SEQ(INT) SHARED,
      PDVI:INT BYVAL,
      LC:ANY;
      ANY)
  BEGIN
   CL = NIL AND CI GT NC =>
    ´ - - -
    !
    ! NO MORE EXPLICIT COMPONENTS TO EVALUATE (CL = NIL), AND  NO
    ! MORE IMPLICIT COMPONENTS TO GENERATE (CI GT NC).  BUILD AND
    ! RETURN A SKELETON OBJECT WITH THE DIMENSIONS  DEDUCED  FROM
    ! GIVEN COMPONENTS OR SIZE SPECIFICATIONS.
    !
    ´ */ PHYSICAL\GENERATE(EM, PDV);
   DECL CM:MODE LIKE
    ´ - - -
    !
    ! TAKE MODE OF THE CI-TH COMPONENT FROM EM UNLESS CI  EXCEEDS
    ! THE  LENGTH  OF  THE  OBJECT  TO BE GENERATED (SEE "SPECIAL
    ! CASES" ABOVE).
    !
    ´ */ [) CI GT NC => ANY; COMPONENT\MODE(EM, CI) (];
   ^;
```

```
DECL C:ANY LIKE
  - - -
 !
 ! GENERATE THE CI-TH COMPONENT:
 !
 !   IF THERE IS AN EXPLICIT SOURCE VALUE, REPLICATE THE CI-TH
 !   COMPONENT.
 !
 !   IF THERE IS AN EXPLICIT COMPONENT FORM, EVAL AND COERCE
 !   IT TO CM; SKIP THE PART OF ADV TO WHICH IT CORRESPONDS.
 !
 !   OTHERWISE, IF EM IS A STRUCT MODE, OR IF THIS IS THE
 !   FIRST IMPLICIT COMPONENT OF A ROW, GENERATE BY SIZE FROM
 !   THE NEXT SEGMENT OF ADV.
 !
 !   IF THIS IS A LATER (CI GT CLI + 1) IMPLICIT ROW
 !   COMPONENT, REPLICATE THE LAST COMPONENT GENERATED.
 !
 ! NOTE THAT IN EACH CASE A CALL TO CONSTRUCT OCCURS. SO
 ! UGF(CM) MAY BE INVOKED, OR A RECURSIVE CALL ON
 ! SYSTEM\GENERATE MAY TAKE PLACE.
 !
  */
  BEGIN
   SV # NOTHING =>
    CONSTRUCT(CM, "BYVAL", LOWEST(SV)[CI]);
   CL # NIL =>
    BEGIN
     ADVI <- ADVI + NAD(CM);
     CONSTRUCT(CM, "BYVAL", EVAL(CL.CAR));
    END;
   IS\ROW\MODE(EM) AND CI GT CLI + 1 =>
    CONSTRUCT(CM, "BYVAL", LC);
   DECL CADV:SEQ(INT) SIZE NAD(CM);
   FOR I TO LENGTH(CADV)
    REPEAT CADV[I] <- ADV[ADVI <- ADVI + 1] END;
   CONSTRUCT(CM, "SIZE", CADV);
  END;
^;
```

```
    CL # NIL ->
      - - -
     !
     ! THIS COMPONENT COMES FROM THE LIST CL.  ADVANCE CL AND CLI.
     ! CHECK  FOR  THE SPECIAL CASE OF TRANSITION FROM EXPLICIT TO
     ! IMPLICIT VECTOR COMPONENTS.  (SEE "SPECIAL CASES" ABOVE.)
     !
      */
      BEGIN
       CLI <- CLI + 1;
       CL <- CL.CDR;
       CL = NIL AND CI LT NC AND IS\VECTOR\MODE(EM) #> NOTHING;
       ADV <- DIMENSIONS(C);
       ADVI <- 0;
      END;
    PDVI LT LENGTH(PDV) ->
      - - -
     !
     ! PDV NOT YET COMPLETE.  MEASURE THE COMPONENT JUST GENERATED
     ! AND FILL THE CORRESPONDING SEGMENT OF PDV.
     !
      */ INSERT\DIMENSIONS(PHYSICAL\DIMENSIONS(C), PDV, PDVI);
    DECL V:ANY LIKE
      - - -
     !
     ! BUILD THE OBJECT AND FILL IN COMPONENTS ABOVE THE CI-TH.
     !
      */ SG2(EM, NC, CI + 1, CL, CLI, ADV, ADVI, PDV, PDVI, C);
    CI GT NC =>
      - - -
     !
     ! IGNORE EXTRA COMPONENTS.  (SEE "SPECIAL CASES" ABOVE.)
     !
      */ V;
     - - -
    !
    ! FINALLY,  WITHOUT  FURTHER  RESORT  TO  USER-DEFINED  MODE
    ! FUNCTIONS,  CHECK  THAT  THE  PHYSICAL  SIZES  OF  C AND ITS
    ! DESTINATION AGREE, THEN INSERT IT AND RETURN THE OBJECT.
    !
     */ -;
    DECL VI:ANY SHARED LOWEST(V)[CI];
    PHYSICAL\DIMENSIONS(VI) # PHYSICAL\DIMENSIONS(C) =>
     FATAL\ERROR("GEN ER-BAD DOPE VECTS");
    LOWEST(VI) <- LOWEST(C);
    V;
   END;

;
```

```
´ - - -  DIMENSIONS   - - -
!
! MODEL FOR THE  BUILT-IN  DIMENSIONS  ROUTINE  (SECTION  4.2.6).
! EXTRACTS AN APPARENT DIMENSION VECTOR FROM ANY OBJECT V.
!
! INITIALIZE DV TO THE DEFAULT DIMENSIONS OF THE OBJECT MODE, MV,
! AND  DEFINE DVI AS AN INDEX IN DV.  DVI STARTS AT 0 AND POINTS,
! NORMALLY, TO  THE  LAST  DIMENSION  INSERTED.   IF,  AFTER  ALL
! DIMENSIONS HAVE BEEN COLLECTED, DVI IS LEFT POINTING BEYOND DV,
! A NON-FATAL ERROR OCCURS TO INDICATE THAT SOME ARE BEING LOST.
!
! CASES:
!
!   IF UDF(MV) EXISTS, APPLY IT TO V, INSERT THE  RESULT  IN  DV,
!   AND QUIT.
!
!   OTHERWISE, IF V IS A SEQUENCE, PLUG IN ITS  LENGTH.   IF  ITS
!   LENGTH  IS  ZERO, THEN QUIT, LEAVING THE OTHER DIMENSIONS SET
!   TO DEFAULT VALUES.
!
!   IF V IS A NON-EMPTY ROW, PLUG IN DIMENSIONS  OF  ONE  OF  ITS
!   COMPONENTS.
!
!   IF V IS A STRUCT, INSERT  DIMENSIONS  OF  EACH  COMPONENT  IN
!   TURN.
!
!   OTHERWISE, RETURN THE DEFAULT DIMENSIONS.
!
´;

DIMENSIONS <-
 EXPR(V:ANY; SEQ(INT))
  BEGIN
   DECL MV:MODE LIKE MD(V);
   DECL DV:SEQ(INT) BYVAL DADV(MV);
   DECL DVI:INT;
   BEGIN
    HAS\UDF(MV) => INSERT\DIMENSIONS(UDF(MV)(V), DV, DVI);
    IS\SEQ\MODE(MV) AND
     INSERT\DIMENSION(LENGTH(V), DV, DVI) = 0 =>
     /* ´EMPTY SEQUENCE .. DONT MEASURE COMPONENT´;
    FOR CI
     TO BEGIN
         IS\STRUCT\MODE(MV) => LENGTH(V);
         IS\ROW\MODE(MV) => 1;
         0;
        END
     REPEAT
      INSERT\DIMENSIONS(DIMENSIONS(LOWEST(V)[CI]), DV, DVI);
     END;
   END;
   DVI GT LENGTH(DV) -> ERROR("GEN ER-BAD DOPE VECTS");
   DV;
  END;
```

```
´ - - -  INSERT\DIMENSION  - - -
!
! PLUG A COMPONENT DIMENSION, CD, INTO THE NEXT AVAILABLE SLOT OF
! DIMENSION VECTOR DV, AND LEAVE THE INDEX DVI POINTING TO THE
! NEW ELEMENT. IF NO SLOT IS AVAILABLE, LEAVE DVI POINTING
! BEYOND DV. IN ANY CASE, RETURN CD AS RESULT.
!
´;

INSERT\DIMENSION <-
 EXPR(CD:INT, DV:SEQ(INT) SHARED, DVI:INT SHARED; INT)
  [) (DVI <- DVI + 1) GT LENGTH(DV) => CD; DV[DVI] <- CD (];

´ - - -  INSERT\DIMENSIONS  - - -
!
! PLUG EACH ELEMENT OF COMPONENT DOPE VECTOR CDV INTO DV AS
! DESCRIBED ABOVE.
!
´;

INSERT\DIMENSIONS <-
 EXPR(CDV:SEQ(INT), DV:SEQ(INT) SHARED, DVI:INT SHARED)
  FOR I TO LENGTH(CDV)
   REPEAT INSERT\DIMENSION(CDV[I], DV, DVI) END;

^;
```

```
´ - - -  PHYSICAL\DIMENSIONS  - - -
!
! EXTRACT A PHYSICAL DIMENSION VECTOR FROM ANY OBJECT (V).  THE
! PHYSICAL DIMENSIONS OF AN OBJECT ARE THE LENGTHS OF THE
! SEQUENCE OBJECTS EMBEDDED WITHIN IT, TAKEN IN THE ORDER OF
! THEIR MENTION IN ITS BASE MODE DEFINITION.
!
´;

PHYSICAL\DIMENSIONS <-
 EXPR(V:ANY; SEQ(INT))
  BEGIN
   DECL PDV:SEQ(INT) SIZE NPD(MD(V));
   DECL PDVI:INT;
   PD2(V, PDV, PDVI);
   PDV;
  END;

´ - - -  PD2  - - -
!
! FILL IN ELEMENTS OF PDV FROM PDVI+1 THROUGH PDVI+NPD(V) WITH
! THE PHYSICAL DIMENSIONS OF THE VALUE V.  LEAVE PDVI UPDATED TO
! INDEX THE LAST ELEMENT INSERTED (IF ANY).
!
´;

PD2 <-
 EXPR(V:ANY, PDV:SEQ(INT) SHARED, PDVI:INT SHARED)
  BEGIN
   DECL BMV:MODE LIKE BASE(MD(V));
   DECL LPDVI:INT LIKE PDVI + NPD(BMV);
   IS\SEQ\MODE(BMV) AND
    INSERT\DIMENSION(LENGTH(V), PDV, PDVI) = 0 =>
    ´ - - -
    !
    ! EMPTY SEQUENCE FOUND.  LEAVE ZERO DIMENSIONS CORRESPONDING
    ! TO ITS COMPONENT MODE.
    !
    ´ */ PDVI <- LPDVI;
   FOR CI
    REPEAT
     PDVI = LPDVI => /* ´END OF PDV SEGMENT FOR V´;
     PD2(LOWER(V, BMV)[CI], PDV, PDVI);
    END;
  END;

^;
```

```
´ - - -  SYSTEM\CONVERT  - - -
!
! BUILT-IN DATA CONVERSION ALGORITHM.
!
! ARGUMENTS:
!
!   V     VALUE TO BE CONVERTED.
!   EM    EXPECTED MODE.
!
! EFFECT:
!
!   EITHER RETURNS A CONVERTED PURE VALUE OF MODE EM (SUCCESS) OR
! ELSE  THE  ORIGINAL  VALUE  V  (FAILURE).  THE ONLY OCCASION ON
! WHICH AN ERROR CAN ARISE IN SYSTEM\CONVERT IS DURING TRUNCATION
! OF AN INTEGER TO A BYTE.
!
!   THERE ARE THREE MAIN KINDS OF  BUILT-IN  CONVERSION:  TRIVIAL
! CONVERSION  TO  MODE  NONE  (USEFUL  FOR PROCEDURES INTENDED TO
! RETURN  NO  VALUE),  CONVERSION  AMONG   POINTER   MODES,   AND
! CONVERSION OF NUMBERS.
!
! USE OF SYSTEM\CONVERT IS DESCRIBED IN SECTION 4.3.1.
!
´;

SYSTEM\CONVERT <-
 EXPR(V:ANY, EM:MODE; ANY)
  BEGIN
   EM = NONE => NOTHING;
   IS\PTR\MODE(EM) AND IS\PTR\MODE(MD(V)) =>
    ´ - - -
    !
    ! POINTER CONVERSION CASES:
    !
    !   CONVERSION OF NIL TO ANY POINTER MODE SUCCEEDS.
    !
    !   OTHERWISE,  THE  OBJECT  (VV)  ADDRESSED  BY  THE  SOURCE
    !   POINTER  (V)  MUST  HAVE  A  MODE  (MVV) THAT IS AMONG THE
    !   ALTERNATIVE TARGET MODES OF EM.  IF NOT, CONVERSION FAILS
    !   AND V IS RETURNED UNCHANGED.
    !
    !   FINALLY, THOUGH SUCCESS IS ASSURED, ONE SPECIAL CHECK  IS
    !   MADE.   IF  EM  IS  A PROC MODE AND V POINTS TO A MACHINE
    !   CODE   OBJECT,   THE   FORMAL   PARAMETER   AND    RESULT
    !   SPECIFICATIONS  OF  THE  CODE  ARE COMPARED WITH EM.  THE
    !   "VAL" MODE OF THE CONVERTED POINTER IS CHOSEN TO  REFLECT
    !   THE  OUTCOME.   CEXPR AND SUBR ARE USED WHEN VERIFICATION
    !   IS SUCCESSFUL.  VCEXPR AND VSUBR MEAN IT WILL HAVE TO  BE
    !   REPEATED WHEN THE CODE IS CALLED.
    !
    ´ */
```

```
    BEGIN
     DECL   ...NY LIKE VAL(V);
     DECL MVV:MODE BYVAL MD(VV);
     V = NIL OR IS\ALTERNATIVE(MVV, EM) #> V;
     V = NIL OR NOT IS\PROC\MODE(EM) OR MVV = DTPR =>
      CREATE\POINTER(EM, V);
      - - -
     !
     ! MVV MUST BE CEXPR, SUBR, VCEXPR, OR VSUBR.  CHECK THE
     ! NOMINAL/FORMAL CORRESPONDENCE BETWEEN EM AND THE CODE.
     ! RECALL THAT VCEXPR = VCEXPR::CEXPR AND VSUBR =
     ! VSUBR::SUBR.  NOTE THAT THE CONVERTED RESULT WILL POINT
     ! TO VAL(V), EVEN THOUGH ITS "VAL" MODE MAY NOT BE
     ! MD(VAL(V)).
     !
      */
     DECL LVV:ONEOF(CEXPR, SUBR) LIKE LOWER(VV);
     CREATE\POINTER(EM,
                    BEGIN
                     NOMINAL\FORMAL\MATCH(EM, LVV) => LVV;
                     LIFT(LVV,
                          BEGIN
                           MD(LVV) = CEXPR => VCEXPR;
                           VSUBR;
                          END);
                    END);
    END;
     - - -
   !
   ! NUMERIC CONVERSION CASES:
   !
   !   INT VALUES MAY BE CONVERTED TO REAL, OR TO SUB-WORD MODES,
   !   AS DEFINED BY THE IS\BYTE\MODE PREDICATE.
   !
   !   REAL NUMBERS MAY BE ROUNDED TO THE NEAREST INT.
   !
   !   SUB-WORD FIELDS MAY BE EXPANDED TO BECOME INTS.
   !
    */ -;
   EM = INT =>
    BEGIN
     MD(V) = REAL => ENTIER(V + 5.0E-1);
     IS\BYTE\MODE(MD(V)) => BYTE\TO\INT(V);
     V;
    END;
   EM = REAL => [) MD(V) = INT => FLOAT(V); V (];
   IS\BYTE\MODE(EM) AND MD(V) = INT => INT\TO\BYTE(V, EM);
   V;
  END;

^;
```

```
´ - - -  ERROR, FATAL\ERROR  - - -
!
! MODEL  RECOVERABLE  AND  NON-RECOVERABLE  ERROR  HANDLING,
! RESPECTIVELY.  NOTE  THAT  ERROR  RETURNS EITHER THE RESULT OF
! EVALUATING A USER-PROVIDED TRAP FORM OR ELSE THE VALUE RETURNED
! FROM THE BREAK (VIA CONT OR STEP).
!
! IN THE SPECIAL  CASE  OF  AN  UNDEFINED  PROCEDURE  ERROR,  THE
! OFFENDING PROCEDURE FORM WILL BE BOUND LOCALLY TO "UPROCN".
!
´;

ERROR <-
 EXPR(E:SYMBOL; ANY)
  BEGIN
   DECL TAG:SYMBOL;
   FOR I BY 2 TO LENGTH(TRAP\TAGS)
    REPEAT E = TRAP\TAGS[I] => TAG <- TRAP\TAGS[I + 1] END;
   TAG # NIL AND MD(EVAL(TAG)) = FORM => EVAL(EVAL(TAG));
   PRINT(E);
   E = "UNDEFINED PROC: " -> PRINT(UPROCN);
   BREAK();
  END;

FATAL\ERROR <-
 EXPR(E:SYMBOL)
  [) ERROR(E); REPEAT BREAK(´CANT CONTINUE´) END (];

TRAP\TAGS <-
 CONST(SEQ(SYMBOL) OF
       "END OF FILE",
       "EOF\TRAP",
       "UNDEFINED PROC: ",
       "UPROCF",
       "FILE NOT AVAIL",
       "FNA\E",
       "TYPE FAULT",
       "TF\E",
       "INVALID INDEX",
       "IVX\E");

^;
```

```
´ - - -  BASE, IS\BASE\MODE, LOWEST  - - -
!
! PEEL AWAY LAYERS OF MODE EXTENSION.  BASE(M)  IS  THE  DEEPEST
! UNDERLYING  MODE  FROM  WHICH  M  IS DEFINED USING ZERO OR MORE
! APPLICATIONS OF THE :: OPERATOR.  IS\BASE\MODE IS  A  PREDICATE
! TRUE  ONLY OF MODES NOT PRODUCED BY ::.  LOWEST(V) PRODUCES THE
! DEEPEST UNDERLYING REPRESENTATION OF THE VALUE V.
!
´;

BASE <-
 EXPR(M:MODE BYVAL; MODE) REPEAT M.UR = NIL => M; M <- M.UR
 END;

IS\BASE\MODE <- EXPR(M:MODE; BOOL) M = BASE(M);

LOWEST <- EXPR(V:ANY; ANY) LOWER(V, BASE(MD(V)));

´ - - -  MODE\AGREES, BINDCLASS\AGREES  - - -
!
! THESE PREDICATES ENCODE  THE  AGREEMENT  REQUIRED  BETWEEN  THE
! FORMAL  SPECIFICATIONS  OF  A  PROCEDURE AND THE STRONG NOMINAL
! MODE TO WHICH IT IS BOUND.  SEE SECTIONS  2.16,  6.1,  AND  THE
! DEFINITION OF NOMINAL\FORMAL\MATCH IN SECTION B.2.
!
´;

MODE\AGREES <-
 EXPR(M1:MODE, M2:MODE; BOOL)
  M1 = M2 OR IS\BASE\MODE(M2) AND COVERS(M2, M1);

BINDCLASS\AGREES <-
 EXPR(BC1:SYMBOL, BC2:SYMBOL; BOOL)
  BC1 = BC2 OR
   CASE[BC1]
    ["LIKE"], ["UNEVAL"], ["LISTED"] =>
     CASE[BC2]
      ["LIKE"], ["UNEVAL"], ["LISTED"] => TRUE;
      TRUE => FALSE;
     END;
    TRUE => FALSE;
   END;

^;
```

B.2 IMPLEMENTATION DEPENDENT DATA ALGORITHMS

```
´ - - -  NOMINAL\FORMAL\MATCH - - -
!
! ARGUMENTS:
!
!   PM    A STRONG PROCEDURE MODE.
!   CODE  A MACHINE CODE OBJECT.
!
! RETURNS TRUE IFF THE FORMAL SPECIFICATIONS OF CODE  AGREE  WITH
! THE PROC MODE PM, IN THE SENSE DESCRIBED IN SECTION 6.1.
!
! USED BY SYSTEM\CONVERT.
!
´;

NOMINAL\FORMAL\MATCH <-
 EXPR(PM:MODE, CODE:ONEOF(CEXPR, SUBR, VCEXPR, VSUBR); BOOL)
  BEGIN
   DECL PD:PDESC LIKE VAL(PM.PROCD);
   DECL NA:SEQ(FDS) LIKE PD.FORMALS;
   DECL FA:SEQ(FDS) LIKE CODE.FORMALS;
   LENGTH(NA) # LENGTH(FA) => FALSE;
   (FOR I TO LENGTH(NA)
    REPEAT
     MODE\AGREES(NA[I].TYPE, FA[I].TYPE) AND
      BINDCLASS\AGREES(NA[I].SYM, FA[I].SYM) #> FALSE;
     TRUE;
    END) AND MODE\AGREES(CODE.RETYPE, PD.RETYPE);
  END;

^;
```

```
´ - - - - - -
!
! MODE CLASS PREDICATES.
!
´;

IS\ROW\MODE <- EXPR(M:MODE; BOOL) M.CLASS = "ROW";

IS\VECTOR\MODE <-
 EXPR(M:MODE; BOOL) M.CLASS = "ROW" AND M.D.LENGTH GT 0;

IS\SEQ\MODE <-
 EXPR(M:MODE; BOOL) M.CLASS = "ROW" AND M.D.LENGTH LE 0;

IS\STRUCT\MODE <- EXPR(M:MODE; BOOL) M.CLASS = "STRUCT";

IS\COMPOUND\MODE <-
 EXPR(M:MODE; BOOL) M.CLASS = "ROW" OR M.CLASS = "STRUCT";

IS\PTR\MODE <- EXPR(M:MODE; BOOL) M.CLASS = "PTR";

IS\GENERIC\MODE <- EXPR(M:MODE; BOOL) M.CLASS = "GENERIC";

´ - - -
!
! MODES CREATED BY PROC( ... ) ARE ALWAYS OF CLASS "PTR". THEY
! ARE DISTINGUISHED BY THEIR NON-NIL PROCD FIELD, A POINTER TO AN
! OBJECT OF MODE PDESC THAT DESCRIBES THE NOMINAL MODES AND
! BINDCLASSES PASSED AS ARGUMENTS TO PROC.
!
´;

IS\PROC\MODE <- EXPR(M:MODE; BOOL) M.PROCD # NIL;

^;
```

```
´ - - -  IS\ALTERNATIVE  - - -
!
! MEMBERSHIP PREDICATE FOR MODES THAT DEFINE SETS OF
! ALTERNATIVES.
!
! ARGUMENTS:
!
!   M     A NON-GENERIC MODE.
!   SM    A GENERIC MODE [ONEOF(M1,...,MK)] OR POINTER MODE
!         [PTR(M1,...,MK)], FOR K GE 1.
!
! RETURNS TRUE IFF M IS AMONG THE ALTERNATIVES M1,...,MK.
!
´;

IS\ALTERNATIVE <-
 EXPR(M:MODE, SM:MODE; BOOL)
  BEGIN
   MD(VAL(SM.D)) = DDB => M = SM.D;
   DECL ALTERNATIVES:SEQ(FORM) LIKE VAL(SM.D);
   FOR I TO LENGTH(ALTERNATIVES)
    REPEAT M = ALTERNATIVES[I] => TRUE; FALSE END;
  END;

´ - - -  IS\BYTE\MODE  - - -
!
! TRUE EXACTLY WHEN M IS THE KIND OF PARTIAL-WORD MODE FOR WHICH
! CONVERSIONS TO AND FROM INT ARE BUILT-IN: VECTOR(K, BOOL), FOR
! K NOT EXCEEDING THE SIZE OF THE HOST MACHINE WORD.
!
´;

IS\BYTE\MODE <-
 EXPR(M:MODE; BOOL)
  IS\VECTOR\MODE(M) AND COMPONENT\MODE(M) = BOOL AND
   NUMBER\COMPONENTS(M) LE MACHINE\WORD\SIZE;

MACHINE\WORD\SIZE <- 36;

^;
```

```
´ - - -  NPD  - - -
!
! NUMBER OF PHYSICAL DIMENSIONS FOR AN OBJECT OF MODE M.
!
! NPD IS THE  NUMBER  OF  POTENTIALLY  DIFFERENT  SEQUENCE  (SEQ)
! LENGTHS  IN  THE PHYSICAL REPRESENTATION OF AN M-VALUE, WITHOUT
! REGARD FOR ANY USER-ASCRIBED PROPERTIES.
!
´;

NPD <- EXPR(M:MODE; INT) [] M.LRFLG => 0; M.SZ (];

´ - - -  NAD  - - -
!
! NUMBER OF APPARENT DIMENSIONS FOR AN OBJECT OF MODE M.
!
! IF  M  HAS  A  NON-STANDARD  APPARENT  DIMENSION  SET,  EITHER
! DIRECTLY,  THROUGH  USER-EXTENTION,  OR  INDIRECTLY,  THROUGH
! COMPOSITION OF MODES THAT HAVE NON-STANDARD DIMENSIONS, NAD  IS
! THE  LENGTH  OF  ITS  DIMENSION  VECTOR.  OTHERWISE,  NAD(M) =
! NPD(M).  (SEE SECTION 4.2.6.)
!
´;

NAD <-
 EXPR(M:MODE; INT)
  [) M.DADV = NIL => NPD(M); LENGTH(M.DADV) (];

´ - - -  DADV  - - -
!
! DEFAULT APPARENT DIMENSION VECTOR FOR MODE M.
!
´;

DADV <-
 EXPR(M:MODE; SEQ(INT))
  BEGIN
   M.DADV = NIL => CONST(SEQ(INT) SIZE NPD(M));
   VAL(M.DADV);
  END;

^;
```

```
´ - - -  NUMBER\COMPONENTS  - - -
!
! RETURNS THE NUMBER OF IMMEDIATE COMPONENTS OF AN OBJECT OF MODE
! M.  M IS ASSUMED TO BE ONE OF THE COMPOUND MODES FOR WHICH THIS
! NUMBER IS FIXED, NAMELY A VECTOR OR STRUCT.
!
´;

NUMBER\COMPONENTS <-
 EXPR(M:MODE; INT)
  [) M.CLASS = "ROW" => M.D.LENGTH; LENGTH(M.D) (];

´ - - -  COMPONENT\MODE  - - -

! RETURNS THE MODE OF THE I-TH COMPONENT OF  AN  M-VALUE.   M  IS
! ASSUMED  TO  BE  A  COMPOUND  MODE.  IF IT IS NOT A STRUCT, THE
! ARGUMENT I IS IGNORED.
!
´;

COMPONENT\MODE <-
 EXPR(M:MODE, I:INT; MODE)
  [) M.CLASS = "ROW" => M.D.TYPE; M.D[I].TYPE (];

^;
```

```
´ - - - - - -
!
! BEHAVIOR FUNCTION PREDICATES AND SELECTORS.  (SEE SECTION 4.2.)
!
`;

HAS\UCF <- EXPR(M:MODE; BOOL) M.UFN # NIL AND M.UFN[UCFN] # NIL;

UCF <- EXPR(M:MODE; ANY) EVAL(M.UFN[UCFN]);

UCFN <- 1;

HAS\UAF <- EXPR(M:MODE; BOOL) M.UFN # NIL AND M.UFN[UAFN] # NIL;

UAF <- EXPR(M:MODE; ANY) EVAL(M.UFN[UAFN]);

UAFN <- 2;

HAS\USF <- EXPR(M:MODE; BOOL) M.UFN # NIL AND M.UFN[USFN] # NIL;

USF <- EXPR(M:MODE; ANY) EVAL(M.UFN[USFN]);

USFN <- 3;

HAS\UPF <- EXPR(M:MODE; BOOL) M.UFN # NIL AND M.UFN[UPFN] # NIL;

UPF <- EXPR(M:MODE; ANY) EVAL(M.UFN[UPFN]);

UPFN <- 4;

HAS\UGF <- EXPR(M:MODE; BOOL) M.UFN # NIL AND M.UFN[UGFN] # NIL;

UGF <- EXPR(M:MODE; ANY) EVAL(M.UFN[UGFN]);

UGFN <- 5;

SUP\UGF <- EXPR(M:MODE; BOOL) M.SUPUGF;

HAS\UDF <- EXPR(M:MODE; BOOL) M.UFN # NIL AND M.UFN[UDFN] # NIL;

UDF <- EXPR(M:MODE; ANY) EVAL(M.UFN[UDFN]);

UDFN <- 6;

^;
```

B.3 PRIMITIVES USED BUT NOT MODELED

```
´ - - -  PHYSICAL\GENERATE  - - -
!
! BUILDS AN OBJECT OF MODE EM WITH PHYSICAL DIMENSIONS PDV,
! DISREGARDING THE POSSIBLE USER-EXTENDED BEHAVIOR OF EM OR ITS
! COMPONENT MODES.  EACH NON-COMPOUND SUB-COMPONENT OF THE RESULT
! IS INITIALIZED TO THE DEFAULT VALUE FOR ITS BASE MODE.
!
! USED IN SYSTEM\GENERATE, SG2, AND PHYSICAL\PURIFY.
!
´;

PHYSICAL\GENERATE <- EXPR(EM:MODE, PDV:SEQ(INT); ANY) ...;

´ - - -  CREATE\POINTER  - - -
!
! ARGUMENTS:
!
!   EM    EXPECTED POINTER MODE.
!   HV    TARGET VALUE, ASSUMED IN THE HEAP.
!
! CREATES A NEW VALUE OF MODE EM POINTING TO HV, WITHOUT INVOKING
! MODE BEHAVIOR FUNCTIONS AND WITHOUT COPYING HV.
!
! USED IN SYSTEM\CONVERT.
!
´;

CREATE\POINTER <- EXPR(EM:MODE, HV:ANY; EM) ...;

 ;
```

```
' - - -  INT\TO\BYTE  - - -
!
! ARGUMENTS:
!
!   V     INTEGER TO BE CONVERTED.
!   EM    A BYTE MODE (A VECTOR(K, BOOL), FOR K NOT EXCEEDING THE
!         SIZE, IN BITS, OF THE HOST MACHINE WORD).
!
! CREATES A NEW VALUE OF MODE EM, INITIALIZED TO THE RIGHTMOST  K
! BITS OF V, WITHOUT INVOKING MODE BEVAVIOR FUNCTIONS.  CAUSES AN
! ERROR ("EXPECTED MODE-VALUE MISMATCH") IF ANY NON-ZERO BITS  OF
! V   ARE   EXCLUDED.    THE   INTENT   IS   TO   ENSURE   THAT   N   =
! BYTE\TO\INT(INT\TO\BYTE(N,EM)).
!
';

INT\TO\BYTE <- EXPR(V:INT, EM:MODE; EM) ...;

' - - -  BYTE\TO\INT  - - -
!
! ARGUMENT:
!
!   V     A  BYTE  VALUE,  I.E.   AN  OBJECT   WHOSE   UNDERLYING
!         REPRESENTATION  HAS  MODE  VECTOR(K,  BOOL),  FOR  K LE
!         MACHINE\WORD\SIZE.
!
! RETURNS AN INTEGER WHOSE LOW ORDER K BITS ARE  COPIED  FROM  V,
! AND WHOSE REMAINING BITS ARE ZERO.
!
';

BYTE\TO\INT <- EXPR(V:ANY; INT) ...;

' - - -  ENTIER  - - -
!
! TRUNCATES R TO THE NEAREST INTEGER TOWARD ZERO.
!
';

ENTIER <- EXPR(R:REAL; INT) ...;

' - - -  FLOAT  - - -
!
! PRODUCES THE REAL NUMBER MACHINE REPRESENTATION FOR I.
!
';

FLOAT <- EXPR(I:INT; REAL) ...;
```

APPENDIX C: BUILT-IN MODES

The following identifiers occur as built-in mode-valued constants in the system. Their values are as indicated below and are not subject to change by the user. No guarantee is intended, however, that their values will remain as indicated as the modes are subject to change from time to time by the ECL development group. The NAME field of the identifiers is the associated tag and the UR field is NIL.

C.1 COMMONLY USED MODES

```
ARITH   ==  ARITH::ONEOF(INT, REAL);

BASIC   ==  BASIC::ONEOF(NONE, BOOL, CHAR, INT, REAL,
                         MODE, SYMBOL, STRING, PORT);

DTPR    ==  DTPR::STRUCT(CAR:FORM,
                         CDR:FORM);

FORM    ==  FORM::PTR(INT, REAL, REF,
                      DDB, ATOM, DTPR);

MODE    ==  MODE::PTR(DDB);

ROUTINE ==  ROUTINE::PTR(DTPR,   CEXPR,   SUBR,   VCEXPR,
                         VSUBR);

STRING  ==  STRING::SEQ(CHAR);

SYMBOL  ==  SYMBOL::PTR(ATOM);
```

C.2 OTHER MODES

```
ATOM == ATOM::STRUCT(TLB:REF,
                     SBLK:PTR(SBLOCK),
                     LINK:HWD);

CEXPR == CEXPR::STRUCT(BODY:SEQ(INT),
                       FORMALS:SEQ(FDS),
                       RETYPE:FORM,
                       DATAB:SEQ(REF));

SUBR == SUBR::STRUCT(BODY:SEQ(INT),
                     PRMD:MODE,
                     RETYPE:MODE,
                     FORMALS:SEQ(FDS));
```

```
FDS == FDS::STRUCT(TYPE:FORM,
                   SYM:SYMBOL);

HWD == HWD::VECTOR(18,BOOL);

SBLOCK == SBLOCK::STRUCT(SINFO:HWD,
                         PLIST:FORM,
                         RMTCH:FORM,
                         CONSTF:FORM);

STRTE == STRTE::STRUCT(JUNK:HWD,
                       TYPE:MODE,
                       LENGTH:INT);

DDB == DDB::STRUCT(SFLG:BOOL,
                   HFLG:BOOL,
                   EPFLG:BOOL,
                   WDFLG:BOOL,
                   LRFLG:BOOL,
                   FINFLG:BOOL,
                   SAFLG:BOOL,
                   SUPUGF:BOOL,
                   EUGFLG:BOOL,
                   CYCFLG:BOOL,
                   PFLG:BOOL,
                   BIFLG:BOOL,
                   QMWFLG:BOOL,
                   ANYFLG:BOOL,
                   NAME:SYMBOL,
                   PROCD:PDPTR,
                   CLASS:SYMBOL,
                   UR:MODE,
                   DADV:PTR(SEQ(INT)),
                   STRTB:PTR(SEQ(STRTE)),
                   UFN:PTR(SEQ(REF)),
                   SFN:HWD,
                   AFN:HWD,
                   GENFN:HWD,
                   TRFN:HWD,
                   BYPT:HWD,
                   SZ:HWD,
                   D:REF);
```

APPENDIX D: RESERVED IDENTIFIERS AND SPECIAL SYMBOLS

SYMBOL	TYPE	SYMBOL	TYPE	SYMBOL	TYPE
ALLOC	t	LISTED	t	UNEVAL	t
AND	io	LOAD	po	<ALT>	t
ANY	c	LOADB	io	"	l
ARITH	c	LT	io	#	io
ARITHNONE	c	MODE	c	#>	t
ATOM	c	NIL	c	%	l
BASIC	c	NONE	c	'	l
BEGIN	t	NOT	po	(	t
BOOL	c	NOTHING	c	(]	t
BY	t	OF	t	)	t
BYVAL	t	OR	io	*	io
CEXPR	c	PDESC	c	*/	io
CHAR	c	PDPTR	c	+	io
CONST	t	PORT	c	+>	io
DDB	c	PROC	t	,	t
DECL	t	PSTKE	c	-	ipo
DTPR	c	PSTRW	c	->	io
END	t	REAL	c	.	t
EXPR	t	REF	c	/	io
FALSE	c	REPEAT	t	/*	ipo
FDS	c	RESET	no	:	t
FHB	c	ROUTINE	c	::	io
FHB1	c	SBLOCK	c	;	t
FOR	t	SHARED	t	<	lmo
FORM	c	SIZE	t	<-	io
FROM	t	STRTE	c	<=	io
GE	io	STRUCT	t	=	io
GT	io	SUBR	c	=>	t
HASHTABLE	c	SYMBOL	c	>	rmo
HWD	c	SYSTEM	c	[	t
IE	ipo	TAG	po	[)	t
INT	c	THUNK	c	]	t
LE	io	THUNKLIST	c		
LIKE	t	TO	t		
		TRUE	c		

Key:
- t = grammar terminal
- io = infix operator
- c = constant
- ipo = infix and prefix operator
- po = prefix operator
- lmo = left match operator
- rmo = right match operator
- no = nofix operator
- l = lexical reserved character

APPENDIX E: ECL ERROR MESSAGES

The following describes some of the most commonly encountered error messages.

Parse Error Messages

These are described in section 3.1

Execution Error Messages

Currently these errors are non-recoverable. All of them except ILL MEM REF leave you at break level at the point at which the error occured.

UNDEFINED PROC: <procedure form>

A form being applied as a procedure does not yield a procedure value. The offending form is printed.

TYPE FAULT

The expected mode of a value and the mode of the actual value provided are incompatible and no conversion is possible. Type faults often occur during procedure calls, either while actual arguments are being bound to the corresponding formal arguments, or when the result mode is checked against the procedure's formal result type. Another common cause of type faults is the attempt to assign an object a value of incompatible mode, as in number <- FALSE, when number already has mode INT.

INVALID INDEX

An attempt has been made to select a non-existent component from an object, such as x[11] when x has only 10 components.

CANT SELECT

An attempt has been made to select from an object with no components, such as 5[1].

FILE NOT AVAILABLE

An attempt has been made to perform an OPEN or some other input-output operation upon a file which is non-existent or upon a device (such as a tape drive) which is not available. A file can not be open for output on two ports at the same time. The port on which it is already open must be closed before it can be reopened. Note that RECLAIM automatically closes inaccessible ports.

ILL MEM REF

Technically, this is caused by an attempt to access an address in memory which is not legal for the user's job. This error is often in the ECL system and not in the user's program. Fill out an ECL Trouble Report and provide a minimal protocol which causes the error to occur. (A minimal protocol is a step by step set of instructions to follow starting from .R ECL which involves as few ECL commands as possible.)

CANT GEN OBJ

An attempt has been made to generate an object of a mode whose components are not all defined.

UNDEFINED MODE

An attempt has been made to create an object of a mode which has not been defined. For example,

```
BEGIN
   DECL M:MODE;
   DECL N:M;
      .
      .
      .
END
```

<NAME> STACK OVERFLOW
<VALUE> STACK OVERFLOW
<CONTROL> STACK OVERFLOW

One of the evaluator´s stacks has overflowed. The first occurrence of this message is a warning only. Part of each stack is held in reserve until the first overflow occurs. Then the stacks are automatically extended, so the user may resume computation by typing CONT(). When an extended stack overflows, the message CONTEXT LOST appears along with the regular overflow message. At this point the computation cannot be CONTinued. The STACKS routine(section 4.8) may be used to reapportion or enlarge the stacks. Both STACKS and RESET leave the stacks in ´unextended´ state.

APPENDIX F: MODE COMPATIBILITY FOR CONVERSION

The following expected mode/actual value pairs will (in the absence of user defined conversion functions) invoke automatic data type conversion:

Expected Mode	Actual Value
NONE	any value
INT	a REAL
REAL	an INT
PTR(..., m, ...)	any pointer to an m
REF	any pointer
INT	VECTOR(n, BOOL) for n LE 36

INDEX

! (editor command) 105

. . . . . 10, 168
(editor command) 120
#> 17, 20

$ (editor command) 104

* 9, 166
** (editor command) 119
*/ 49, 164
*> (editor command) 118

+ 9, 165
+> 10

- 9, 166
-> 10

.. (editor command) 104
... (editor command) 107, 119

/ 9, 166
/* 49, 163
// (editor command) 104, 106

:: 75, 183

< 11
<* (editor command) 118
<- 6, 164
<< 49, 184
<= 182
<@ (editor command) 124

= 10, 168
=> 17, 20

> 11
>> (editor command) 119

? (editor command) 118

@ (editor command) 104, 105, 121
@@ (editor command) 123

ALLOC 34
alt-mode (editor command) 102

AND 9, 167
ANY 31
apparent dimensions 87
ARITH 227
arithmetic operators 9
ASSERT 178
assignment 6
assignment function 84
ATAN 167
ATOM 227

B (editor command) 109
BASIC 227
BASIC\STR 176
BEGIN 13
BIG\NAME (compiler input) 150
BIG\SIZE (compiler input) 151
bind-class 14, 43
BIN\NAME (compiler input) 148
BIN\SIZE (compiler input) 148
block 13
BOOL 4
Boolean operators 9
BREAK 70, 177
break level 69
BT 187
BY 18
BYVAL 14, 16, 36, 43

C (editor command) 125
CASE 46
CEXPR 129, 163, 227
CHAR 4
CHAR\INT 176
CIPORT 54
CLOSE 52, 171
CMACRO\PAIRS (compiler input) 146
coercion 16, 44
command 63
comments 49
compiler toggles 138
COMPILE\ROUTINE (compiler routine) 154
COMPILE\ROUTINES (compiler routine) . . . 154
COMPLETE 74, 184
completion 74
compound form 13
computed macro 135
conditional 10, 17
consequent 17
CONST 34
constant 3, 132
CONSTRUCT 82, 184
CONST\NAMES (compiler input) 146

CONT 69, 177
conversion 32, 42, 233
conversion function 83
COPORT 54
COS 167
COUNT 191
COVERS 31, 183
CTRL-C 71
CTRL-O 71
CTRL-U 66
CTRL-Z 64, 72

DADV 87, 222
data type 3
DDB 228
DDT 178
DECL 14
declaration 13, 93
DECLARE\FREES (compiler routine) 152
default generation 34
default value 17
DIFF 166
dimension function 90
DIMENSIONS 212
DONT\THUNK (compiler input) 148
DRAIN 52, 172
DTPR 129, 227
DUMPB 57, 175
DUMP\PACKAGE (compiler routine) 155

E (editor command) 115
EDCONT (editor command) 125
EDIT 102
EDIT (editor command) 125, 127
editing 66
Editor 102
Editor macro 127
EDOPCHANGE (editor command) 128
END 13
end of file handling 53
environment 15
EOF\TRAP 53
EQUAL 168
ERROR 217
Error Messages 230
error-handling 71
EVAL 164
execution error messages 230
execution errors 69
EXIT (editor command) 115
exit-conditional 17, 20
EXP 167
explicit procedure 38

EXPR 39
extended mode 73
EXTENDED\MODES (compiler input) 150

FA (LSMFT command) 98
FAIL 182
FALSE 4
FDS 227
FG (LSMFT command) 98
FINDHASH 195
FLUSH 181
FLUSHASH 196
FLUSHEDIT (editor command) 128
FLUSHFIX 170
FLUSH\COMPILER (compiler routine) 154
FLUSH\DRIVER (compiler routine) 155
FLUSH\LSMFT 101
FLUSH\UP 190
FM (LSMFT command) 98
FOR 18
FORM 227
form 3
formal parameter 39
free variable 15, 131
free variable mode declaration 137
FROM 14, 18, 36

GE 10, 169
generation 34
generation by example 36
generation function 78
GENERIC 46
generic mode 30
global variable 6, 132
GT 10, 169
GUESS\MODE (compiler input) 147

HASH 176
hash coding 193
HASHTABLE 196
HAS\UGF 224
heap 27
HWD 228

I (editor command) 107
identifier 5
IE 49, 163
INCHAR 55, 173
INFIX 170
infix expression 7
INITEDIT (editor command) 128
initialization by coercion 16
initialization by default 16

initialization by sharing 16
initialization by value 16
INIT\FORM (compiler input) 149
INOBJ 173
input-output 51
INT 4
INT\CHAR 176
iteration 18

J (editor command) 110

K (editor command) 107

L (editor command) 109
LAND 169
LE 10, 169
left-association 11
LENGTH 22, 164
length unresolved 23
LEQV 169
LEX 174
LG (LSMFT command) 98
LIBRARY\AREA (compiler input) 148
LIFT 76, 183
LIKE 14, 16, 36, 44
limit check 19
LISTED 45, 139
LISTING\NAMES (compiler input) 151
LM (LSMFT command) 98
LN 167
LNOT 169
LOAD 55, 175
LOADB 58, 175
LOAD\COMPILER (compiler routine) 153
LOAD\EARLY (compiler input) 151
local name retention 139
local variable 6, 14
LOR 169
LOWER 76, 183
LOWEST 218
LROT 169
LSH 169
LSMFT 97
LST\NAME (compiler input) 151
LT 10, 169

MAKEHASH 194
MAKEPF 53, 172
MAKEPF, Pseudoport 53
MARK (editor command) 111
mark fault 51
MATCHFIX 170
matchfix expression 8

MBD (editor command) 123
MD 164
MDKN\PAIRS (compiler input) 146
METER 190
METERF 192
metering routines 190
mode 3, 4, 6, 227
mode behavior function 76
mode deletion 97
MODE-BIND CLASS MISMATCH 85
mode-valued form 20

NAD 87, 222
name qualification 25
name stack 136
NASSIGN 181
NIL 4
NOFIX 170
nofix expression 8
nominal procedure mode 58
NONE 4
NOT 9, 168
NOTHING 4
NPD 222
NS (editor command) 114
NSUBR 163

OF 14, 36
ONEOF 30
OPEN 52, 171
operator 7
OR 9, 168
OUTCHAR 55, 172
OUTOBJ 173
OVERLAY\MODE (compiler input) 151

P (editor command) 106
PACKAGE\NAMES (compiler input) 148
PARSE 55, 175
pattern-matching 117
PEEK 178
PFORM 56, 174
physical dimensions 89
PHYSICAL\DIMENSIONS 214
PHYSICAL\PURIFY 207
PIPORT 54
POKE 179
POPORT 54
PORT 52
PORT\STR 177
POSTLUDE\FORM (compiler input) 150
PQ (editor command) 106
precedence 11

PREFIX 170
prefix expression 7
PRELUDE\FORM (compiler input) 149
PRINT 56, 68, 174
print function 86
procedure 4, 38
procedure body 39
procedure call 38
procedure modes 58, 129
PRODUCT 166
proper object 12
PTR 28
pure value 13

Q (editor command) 116
Q-registers 115
QL 165
QUICK\FLUSH (compiler input) 151
QUOTE 165
QUOTIENT 166

R (editor command) 120
RA (editor command) 121
RAF (editor command) 121
RAPAT (editor command) 125
RDOBJ 199
READ 55, 67, 174
REAL 4
REAL\STR 176
rebound names 137
REBOUND\NAMES (compiler input) 147
RECLAIM 165
REF 4, 27
reference 3
REHASH 196
relational operators 10
REMETER 192
REPEAT 18
replication 79, 97
RESET 70, 177
RESTORE 180
result type 39
RETAIN\ALL (compiler input) 147
RETAIN\NAMES (compiler input) 147
RETBRK 70, 177
RETURN 49, 185
return fault 51
right-association 11
ROPEN 199
ROUTINE 4, 38, 129, 227
ROUTINE\NAMES (compiler input) 145
row 22
RPAT (editor command) 125

RUBOUT 66
RWCLOSE 199

S (editor command) 113
SAVE 180
SAVED\NAMES (compiler input) 151
SBLOCK 228
scope 6, 14
SELECT 86, 184
selection 32
selection function 85
SEQ 21
SET\UP 57, 189
SHARED 14, 16, 36, 44
shared value 132
SHARED\NAMES (compiler input) 146
sharing fault 44
SHOW\FREES (compiler input) 150
SHOW\QUOTATIONS (compiler input) 150
SIN 167
SIZE 14, 35
SMACRO\PAIRS (compiler input) 146
SPAT (editor command) 125
STACKS 180, 232
statement 17
STATEMENT (editor command) 127
STEP 178
STRING 4, 5, 23, 227
STRUCT 24
SUBR 129, 163
subscripts 25
substituted macro 134
substitution operator 104
SUM 166
SUPUGF 81
SUP\UGF 224
SYMBOL 4, 227
syntax errors 63
SYSTEM\CONVERT 215
SYSTEM\GENERATE 205

T (editor command) 106
TAG 181
tag 74
test clause 17
THUKLIST 141
THUNK 141
thunk 140
TIME 191
TIMECOUNT 191
TIMESTACK 192
TO 18
top-level assignment 64

top-level binding 65
top-level environment 64
top-level variable 6, 15
TRACE 186
TRUE 4
type fault 7, 230

U (editor command) 126
UAF 84
UCF 83
UDF 90
UGF 78
UNEVAL 45, 139
united pointer 30
UNMETER 192
UNPARSE 57, 188
UNPARSF 188
UNPARSF2 189
UNPARSFM 189
UNPARSP 189
UNTAG 183
UNTRACE 187
UNTRACEALL 187
UP (editor command) 111
UPF 86
USF 85

VAL 28, 164
variable 5
VCEXPR 129
VECTOR 21
VERIFY\ALL (compiler input) 147
VERIFY\NAMES (compiler input) 147
VSUBR 129

WOPEN 199
WQ (editor command) 116
WROBJ 199

XFQ (editor command) 115
XLQ (editor command) 115
XPAT (editor command) 125
XQ (editor command) 116
XTR (editor command) 124
XZQ (editor command) 115

Z (editor command) 110
ZJ (editor command) 109
ZPAT (editor command) 125

HARVARD UNIVERSITY
CENTER FOR RESEARCH IN COMPUTING TECHNOLOGY

ECL TROUBLE REPORT

Please attach a minimal protocol exhibiting the difficulty you've encountered. Protocols should begin with an .R ECL and be as short as possible. Where appropriate, try to save any files or other data necessary to reproduce the protocol.

ECL Version: (___-___-___)

Description of Problem_____

_____

_____

_____

_____

_____

_____

_____

Name_____Date_____

Organization_____

Position_____

Street_____

City_____State_____Zip Code_____

AIKEN COMPUTATION LABORATORY
Cambridge, Mass. 02138