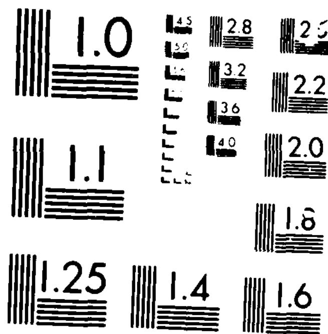
AD-A169 382 IMPROVING PRODUCTIVITY IN THE DEVELOPMENT OF LARGE SOFTWARE SYSTEMS E-L REFERENCE MANUAL(U) SOFTWARE OPTIONS INC CAMBRIDGE MA 15 MAR 86 N00014-85-C-0710 1/1

UNCLASSIFIED F/G 9/2 NL


1.0
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

AD-A169 382

E-L

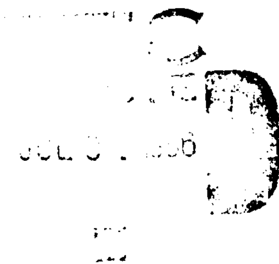
Reference Manual

(Version 0.0)

Contract No. N00014-85-C-0710

March 15, 1986

Software Options, Inc.
22 Hilliard Street
Cambridge, Mass 02138



DTIC FILE COPY

86 6 30 098

Progress Report

on

Improving Productivity in the Development of Large Software Systems

N00014-85-C0710

(Phase 1)

Submitted to

Scientific Officer
Program Manager, Information Sciences Division
Office of Naval Research
800 North Quincy Street
Arlington, Virginia 22217

Attention Dr. Charles J. Hoiland, Code 433

By

Software Options, Inc.
22 Hilliard Street
Cambridge, Mass. 02138

March 14, 1986

Overview

The documents included in this package constitute the progress report covering Phase 1 of the project *Improving Productivity in the Development of Large Software Systems*. The deliverables for Phase 1, as outlined in section 8 of the proposal, fall into the categories of environment, language framework, and code generator. The last of these categories is discussed in the document *Coagulating Code Generator—Prototype Implementation.*

The first two categories have a more indirect relationship with the other three documents in this report. It has been our contention from the outset that there must be a close connection between the designs of an environment and language. To quote from the proposal, "the difficulty is that environments have usually been afterthoughts to the language design" and "the existence of a powerful environment ... affects the nature of the language". As our research progressed during Phase 1, we became increasingly convinced of the tightness of the connection, to the point of choosing the name E-L, for environment and language, to describe our design.

The importance that we feel about this issue is reflected in the documents that we have written describing our research. Because of the seamlessness of environment and language, there are not separate documents describing each. Rather, each of the documents has a mixture of the concerns of both. What may seem like language in one document may seem more like environment in another. Ultimately, the distinction is so blurred that it becomes terminological rather than substantive.

The forms that we have chosen for the report on our research also reflect the process of the research as well as its results. Because of our concern for the way that the system is actually used, the *E-L Reference Manual* is written in a somewhat tutorial style. There is an implicit acknowledgement that E-L, like any system, must be learned. Describing the system from the point of view of the novice is particularly valuable in revealing what is important in the environment, and this document implicitly describes much of our design for the environment, particularly in Chapters 7 and 14. The *E-L Design Rationale* turns out to be the document that we anticipated in the proposal. The remaining document, the *E-L Definition*, was not originally planned. We discuss this document here at greater length than the other documents, because it is an additional deliverable, not described in the proposal. In some respects, this document resembles a cross between the traditional reference manual and a formal semantic definition of a language. However, we have deliberately omitted the words "formal" and "semantic", because we have not intended to produce a dry formal description. Rather, this document is something that we expect a substantial percentage of E-L programmers to use, as they become more sophisticated in their use of the system. The *E-L Definition* is more than just an extended reference manual. It also includes what would be considered rationale material in other language descriptions. This is motivated by the desire to convey the issues that arise when doing an extension. Much of what looks "built-in" is in fact based on E-L's extension mechanisms, and when a programmer is considering an extension, it is natural that he should know the principles that governed the design of "built-in extensions". Further, there are several themes running through the *E-L Definition* that address issues of the environment, and especially, the blurring of language and environment.

Table of Contents

Part I

1. Overview
   - .1 Documentation
   - .2 Getting started
   - .3 Syntax
   - .4 Declarations
   - .5 Errors

2. Types
   - .1 Introduction
   - .2 Literals
   - .3 Constructors
   - .4 Constants and variables
   - .5 Types and functions

3. Elementary Constructs
   - .1 Numbers
   - .2 Logical expressions
   - .3 Text
   - .4 Input and output

4. Compound Values
   - .1 Tuples
   - .2 Arrays
   - .3 Records

5. Flow of Control
   - .1 Functions
   - .2 Conditionals
   - .3 Iteration
   - .4 Premature exit
   - .5 Case expressions

6. Functions
   - .1 Mutual recursion
   - .2 Functions as values
   - .3 Functions and assignment
   - .4 Multiple results
   - .5 Pure functions

7. Program Preparation
   - .1 Documents
   - .2 Editing
   - .3 Breakouts
   - .4 Annotations
   - .5 Testing

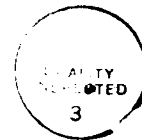


Table of Contents, *cont.*

Part II

8. Types Revisited
  .1 Types as values
  .2 Declaring literals and constructors
  .3 Base type
  .4 Choose types
  .5 Union types
  .6 Infinite unions
  .7 Expandable unions
  .8 Cyclic types

9. Function Families
  .1 Introduction
  .2 Equality
  .3 Well-defined functions
  .4 Generation
  .5 Declaring families
  .6 Summary

10. Conversion
  .1 Laws
  .2 Conversion and equality
  .3 Extensions
  .4 Addition
  .5 Zero

11. Flow of Control Revisited
  .1 Writing iterators
  .2 "Input" from iterators
  .3 "Output" to iterators
  .4 Premature exit revisited
  .5 Continuing iterations

12. Assignment
  .1 Places
  .2 Operations on places
  .3 Conversion on places
  .4 Extensions

13. Compound Values Revisited
  .1 Selection syntax
  .2 Types and laws
  .3 Extensions
  .4 Slices
  .5 Sparse arrays

14. Larger Programs
  .1 Packages
  .2 Multi-document programs
  .3 Measuring performance
  .4 Testing Revisited
  .5 Syntax extensions

Chapter 1

Overview

1.1 Documentation

This manual is the entry point to E-L. It is intended for programmers, not for people for whom E-L would be an introduction to computing. For the intended audience, there is the advantage that fundamental notions do not have to be explained, and the disadvantage that E-L nomenclature will inevitably overlap with that of other languages, while the precise technical meaning that E-L and other languages assign to the same words will often be different. We have tried to point out similarities and contrasts with other languages, but beware.

The style of this manual is more narrative than topical. The aim is to get you as quickly as possible to the point where you can program in E-L. Constructs that are analogous to those of other languages are left to your intuition. No single subject is covered in its entirety in any one place. This not only prevents telling you too much detail when you don't need it, it also makes it easier to understand the interdependence of different facets of E-L.

When you first start using E-L, it is inevitable that you will use it in the same style that you have used your previous programming language(s)—you will speak E-L with an accent. While on the one hand, this manual is organized to allow you to do that as quickly as possible, it also tries to encourage you to program in the E-L style. The whole purpose of E-L is to support a programming style that increases productivity, not only by producing working programs faster, but by improving the "maintainability" of your programs. This is done by supporting programming at very generic levels, and achieving efficiency indirectly. E-L helps you accomplish this by providing tools for analysis and transformation, tools for specifying program derivation, a documentation facility that goes far beyond comments, as well as the more traditional environment capabilities of packages, version control, multiple user access, and the like.

There are several other major documents for E-L. One is the E-L Definition. Its organization is entirely different. While the reference manual is based upon what you will need to write, the Definition is organized around "what is really going on". Once you become familiar with the surface of E-L, questions about details are most easily answered by looking in the Definition. You will also find there summary descriptions of various topics, and a few "proofs" of certain implementations.

The E-L Rationale has the purpose that its name implies—it explains why E-L is designed the way it is. Although this document may be useful in giving you further insight into the E-L mentality, it really has no information that you need to program in E-L. There are also implementation documents for various pieces of the E-L system. These are primarily for the benefit of implementors of E-L, but also, we hope, exemplify documentation in the E-L style.

1.2 Getting started

To experiment, get into E-L. You will be asked for the name of a document. For present purposes, just hit carriage return. An edit window, the program box, will pop up, and the cursor will land in it. The only reasonable thing for you to do at this point is to type in a program. Try the following:

```
write(read()^2)
```

You have just written a complete program. It reads a number, squares it, and prints the result, as you shall soon see.

After you have finished editing, you may run the program by selecting "test". This will flash up a run window associated with this run of the program, and the cursor lands in the

input section of the window. Type in a number, and you will see your input, followed by the output, appear in the transcript part of the window. Because the program exits immediately after printing one number, the input section of the window is closed, and the cursor lands in the header. A finished run window will likely not hold your interest for long. A double click will delete it, and the cursor will reappear in your program box. You may edit the text to try other programs. This is the standard way of experimenting with E-L.

1.3 Syntax

The purpose of the syntax of any language is to structure the characters that ultimately make up a program. Like other languages that you know, structure in E-L is first introduced by grouping the characters into lexemes. As a general rule, lexeme breaks occur at spaces, new lines, and between sufficiently dissimilar characters. You should be aware that the character "`-`" does not cause a lexeme break, thus allowing the use of hypenated names. A minus sign requires space around it. Without going into more details here,

```
a old-value(1.3e7
```

has lexemes `a`, `old-value`, `(`, and `1.3e7`.

The syntactic organization of E-L above the lexical level is called its *phrase structure*. The simplest phrases are those that consist of a single lexeme. More complicated phrases of E-L arise from only one construction, with a few variants:

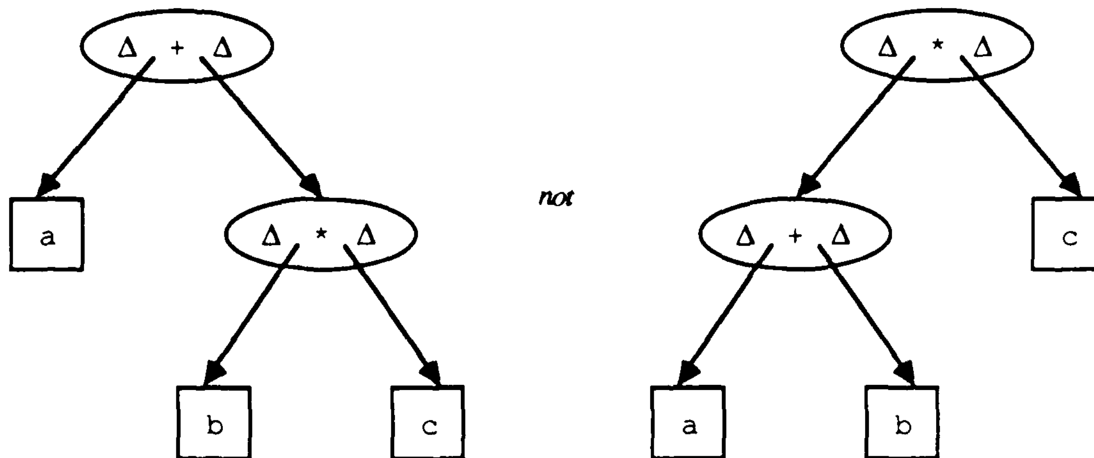
```
[op0] phrase1  op1 . . . opk-1 phrasek [opk]
```

An *op* here is not necessarily a function—for syntactic purposes, a comma and a parenthesis are *op*s. The brackets mean that an *op* is optional. Certain *op*s like **`If, then, else`**, and **`For`** are presented in boldface to aid the user in perceiving the structure of a phrase containing them. Certain phrase types have names for easy reference. Letting Δ denote an arbitrary phrase, they are:

`op0`	nofix ($k = 0$, no subphrases)
`op0` Δ	prefix, e.g. `not b`
Δ `op1`	postfix, e.g. `3!`
Δ `opi` Δ	infix, e.g. `x + y`
`op0` Δ `op1`	matchfix, e.g. `< 0 >`
Δ `op1` Δ `op2`	parenfix, e.g. `f(x)`

It may be helpful to imagine a tree generated by the phrase structure. Lexemes have one node type, say a rectangle, containing the lexeme. Phrases have another node type, say an oval, containing the phrase operators, and have the subphrases as descendants.

The grammar just given is highly ambiguous, because it admits many different phrase structures for one string. This ambiguity is resolved by the parsing convention that a phrase is extended as far as possible, innermost phrases first, and by precedence among *op*s. As with lexical conventions, you do not need full knowledge at this point. The precedences have been established to follow usual and sensible conventions. For example, `a + b * c` would result in:



We say that * is stickier than +. Grouping of phrases can be accomplished by using parentheses. There is no **begin ... end** construction, the equivalent being achieved by parenthesization. To within knowledge of particular phrases and precedence relationships, you now know the entire E-L grammar.

Well, almost. There is a two-dimensional aspect to E-L that lies between the lexical and phrase structure of the grammar. Many languages use indentation in their presentation, and some have utilities that will format a program with indentations, according to its phrase structure. E-L turns this situation around. Since you don't want to look at an unindented program, neither does E-L. Instead, it relies upon indentation to provide abbreviations for grouping by parenthesization. For example, consider the following statement.

```
If b then (x <- 1; y <- 2) else (f(x,y); g(y,x))
```

This is equivalent to the following indented version:

```
If b then
  x <- 1; y <- 2
else
  f(x,y); g(y,x)
```

This convention allows for succinct presentation of code, but it means that carriage return is not equivalent to space, as it is in some languages. Moreover, note that there is no closing **fi** or **endif**. The **If then else** statement ends whenever there is text following the **else** at the same indentation level as the **else**, or as in the above fragment, when there is simply no more code.

If the white space at the beginning of a line goes beyond the normal amount for indentation, then the second line is in some way a continuation of the first. For example:

```
If x gt 0 and
     y lt 0 then
  z <- 1
elseif y gt 0 then
  z <- z + 1;
  w <- x
```

1.4 Declarations

Certain phrases in E-L are *declarations*. As an example, we will use the declaration that introduces constants:

```
Let phrase1 be phrase2 in phrase3
```

The principal *op*s for this phrase are **Let**, **be**, and **in**. The *scope* of this declaration is *phrase3*. By convention, the **in** can be replaced by a semi-colon, and *phrase3* can occur on the next line, not indented:

```
Let phrase1 be phrase2;
phrase3
```

More generally, a declaration will have the form:

```
op0 phrase1 . . . phrasek-1 in phrasek Δ
```

Because the scopes of declarations are often more than a line, you will more often use:

```
op0 phrase1 . . . phrasek-1;
phrasek
```

From now on in this manual, you will never see an **in** in a declaration, and the term "declaration" will tend to refer to the part of the declaration before **in** *phrasek* Δ. This part of the declaration may take several lines, but you may always use parentheses, explicitly or implicitly via indentation, to effect the desired grouping.

```
Let phrase1 be
  a very complicated phrase2
  taking several lines;
phrase3
```

1.5 Errors

It is possible to make errors when programming. These fall roughly into the following categories.

- Syntax errors—your characters don't come together to yield phrase structure.
- Analyzable errors—your phrase structure is sensible, but some analyzer detects a problem with the program.
- Runtime errors—some problem arises in executing your program (these are called *faults*).
- Algorithmic errors—your program runs without detectable error, but you don't like the results.

These categories are not completely distinct. Syntax errors blur into analyzable errors in part because the phrase structure is more flexible than anything that can be executed, and some analysis is done on the way from phrase structure to executable program. For example, references to undefined symbols are detected as part of this translation. It is a matter of

nomenclature whether these are thought of as syntax errors or analyzable errors.

Analyzable errors shade into runtime errors because of the nature of some analyzers. They perform "weak" or "symbolic" executions, in order to learn something about the program, and the nature of the errors that they detect are more akin to runtime errors than to syntax errors. You can view running the program simply as analysis by the "strongest" execution.

There is also overlap between runtime and algorithmic errors. For example, an infinite loop might be thought of as an algorithmic error, but you notice it while running the program, and deal with it in the same way that you deal with other runtime errors (click the break pedal in the header of the run window to stop running and look around). It is sometimes possible to view an algorithmic error as an indication of insufficient analysis. If you can specify some aspect of what you want a program to do, and execute it symbolically to determine if it meets the specification, you have a lot more confidence in your program: it is at least partially correct on an entire class of inputs, rather than the few, or even many, test cases that you could try.

Errors that E-L detects are associated with a window. For example, syntax errors are reported in the edit window, and faults in the run window. E-L does a lot behind your back, so there will be windows that you don't see unless some error is detected. You will be notified about these in the E-L window. Errors usually have a brief description, and ways of making more detailed inquiries.

Chapter 2

Types

2.1 Introduction

Types in E-L are analogous to types in other programming languages. They are central to disambiguating generic operations like +, and determining where implicit conversion occurs. Like many languages, types in E-L supply information about the values with which they are associated, and are thus useful in catching certain kinds of errors and in analysis and transformation. But E-L has more powerful techniques for this purpose than types (see section 7.4), so types should not be forced unnaturally into playing this role.

Unlike the situation in most other languages, types in E-L are themselves values. To understand their use in E-L, it is necessary and sufficient to understand type literals and operations that use or produce types. In forming your mental image of types in E-L, it is best not to think of them as a subset of values (though all values with the same type are indeed a subset of values), nor, as in some languages, a set of functions (although they are related to that also). Rather, types provide connections between a given value and other related values, including types and functions. Remember, types and functions are values in E-L.

This manual will return to the subject of types often. This chapter is only to get you started. Once you can write simple functions, types will become much more interesting.

2.2 Literals

Several types are built in to E-L. These include classical types found in many programming languages. This chapter is not the place for a discussion of the particular properties of any one type; rather, they are discussed in their natural place throughout this manual.

The built-in types are technically type *literals*. You will later see how to introduce your own literals. Like all names in E-L, type literals have lexical scope. If you reuse the name in some other way, say as the name of a function parameter, the scope of the inner use does not see the literal, only the parameter. It is not a good idea to cover up built-in type names with inner definitions of the same names, simply because your program becomes harder to read—these commonly occurring names have fixed meanings for E-L programmers. For a list of these names, select "?" in an E-L window, and inspect the window that results.

2.3 Type constructors

From the point of view of types as values, a type constructor is a function that yields a type as its result. All of the results of a type constructor share certain characteristic properties of that constructor. For example, the type constructor `array` has two arguments: a tuple of `integers`, and a `type`. A value whose type is `array(<3>,real)` is indexable by integers 1 through 3, with indexed values being `reals` (see Chapter 4 on compound objects). Many type constructors have a type as at least one of their arguments, but this is not strictly necessary.

As with type literals, certain type constructors are built-in to E-L, and you may define your own. The built-in constructors will be described at their natural places throughout this manual; a list may be found beginning by a "?" in the E-L window. You will see how to define your own in Section 8.2.

2.4 Constants and variables

There are simple declarations to introduce new names. To introduce a constant named x:

```
Constant x:integer is v;
```

The value *v* will be converted to `integer` before x is established. Conversion will be discussed in more detail in Chapter 10. For now, it is enough to know that conversion can "fail"; if so, it causes a Constant Fault (see section 1.5). If conversion "succeeds", the name x is established, with initial value the integer obtained from *v*. As will be formalized later, conversion always succeeds when the target type is the same as that of the value being converted. The scope of x is the scope of this declaration, and its value will always be its initial value.

To introduce a variable, use:

```
Variable x:integer is v;
```

Throughout this scope, the type of x is `integer`. This has about the same effect as a **Constant** declaration, except of course that x may vary.

A variable or constant may also be declared without specifying a type, for example:

```
Variable x is 0;
```

In a typeless declaration, the type of x takes on the type of the initializing expression. For better or worse, you will never have a fault in this kind of declaration.

Once a variable is extablished, it retains the same type throughout its lifetime, whether or not a type was mentioned in its declaration. You may assign to the variable, as in the statement `x <- 2`, but this changes the value of the variable, not the type. Rather, the value produced by the righthand side of the assignment is converted to the type of the variable being assigned to. Again, conversion can fail; if so, there is an Assignment Fault. Otherwise, the assignment, like declaration, has the effect you would expect.

2.5 Types and functions

The general form for declaring a function is:

```
Function name(formals-list) -> (result-type) is body;
```

The **is** is almost never seen, because it disappears in the multiple line form:

```
Function name(formals-list) -> (result-type)
  body;
```

In other words, the entire body must be indented. The semicolon convention applies in *body*. If the function has no parameters, you may omit the `(formals-list)` part of the header; you may also omit the `-> (result-type)` part, but more on that later. You may not omit *body*.

The *name* following **Function** is the name of the new function that this declaration is declaring. The scope of *name* is not only the scope of the declaration (see section 1.4), but also *body*. The *formals-list* is a comma-separated list of name-type pairs, with a colon separating the name and the type. For example:

```
Function f(x:integer, y:real) -> (boolean)
  body;
```

The identifiers x and y are the formal parameters for f; their scope is the body of f only, not the scope of the declaration. The types in a function header work analogously to a **Constant** declaration. If types are given, the corresponding parameter in a call on f must convert to the type in the header. If conversion fails, there is an Argument Fault. If there is no type associated with the formal, then no conversion is done.

The -> (*result-type*) part of the declaration controls what happens at function exit. If present, the value produced by *body* is converted to the indicated type; conversion failure causes a Result Fault. If this part of the declaration is missing, then it is assumed that the type of the result can be inferred from the body of the function, and no conversion is necessary. The notation -> () indicates that no result is returned.

Chapter 3

Elementary Constructs

3.1 Numbers

The main arithmetic types you will use are `integer` and `real`. They have the usual operators of +, −, *, and /. These operators have precedences so that they group according to the standard mathematical conventions:

`a - b - c*d/e` is the same as `(a - b) - ((c*d)/e)`

They work the same as in other languages, except perhaps for / on integers: if the division is not exact, / will return a `rational` number. The `rationals` also have the standard four arithmetic functions defined on them. To obtain the numerator or denominator of a `rational`, use dot notation:

`(-2/4).num = -1`   `(-2/4).denom = 2`

The numerator and denominator are always given in lowest terms, with the numerator carrying the sign. The function `floor` is defined on `rational` and `real`, returning the greatest `integer` part in the number, according to the standard mathematical definition:

`floor(x) ≤ x < floor(x) + 1`

NB: `floor(-1/2) = -1`. The common pattern `floor(x/y)` may also be written `x//y`, so for positive `integers`, the operator // may replace what you are accustomed to writing as /; the situation for negative integers is less standard, so beware. The functions `ceiling` and `round` are also defined on `real` and `rational`. They obey similar rules:

```
ceiling(x) - 1 < x ≤ ceiling(x)
round(x) - 1/2 < x ≤ round(x) + 1/2
```

In addition to the usual relations = and #, the types `integer`, `real`, and `rational` all have ordering relations defined: `le`, `lt`, `ge`, and `gt`. These operators are less sticky than those for arithmetic operations, so that there is reasonable grouping: `a+b le c*d` is the same as `(a+b) le (c*d)`.

The type `complex` is also available, and the four arithmetic operations are defined on this types, but the ordering relations are not. You may obtain the real and imaginary parts of numbers of these types using dot notation. If *z* is an expression yielding one of these types, then:

`z.re` is the real part of *z*   `z.im` is the imaginary part of *z*

You may construct a complex number by giving its real and imaginary parts: `complex(0,1)` is the square root of -1.

You cannot assign to the real and imaginary parts of a complex variable separately, nor the numerator and denominator of a `rational`. These types and the operations on them are defined solely by their role as numbers, and are to be used in that way.

In addition to the basic operations defined above, the types `real` and `complex` have a

standard set of more complicated functions associated with them, whose names pretty well describe them.

```
sqrt  exp   log
sin   cos   tan   sec   csc   cot
asin  acos  atan  asec  acsc  acot
sinh  cosh  tanh  sech  csch  coth
asinh acosh atanh asech acsch acoth
```

Any of these functions applied to a `real` will yield a `real` result when possible. Applying `sqrt` to a negative `real` yields a `complex` with positive imaginary part.

As in many languages, an `integer` in E-L will implicitly convert to a `real`, for example, on function entry. In an analogous way, an `integer` will also convert to a `rational`, and any `integer`, `rational`, or `real` will convert to `complex`. However, unlike some languages (but like Common Lisp), conversion in the other direction is partial: `reals` and `rationals` that are exact integers will convert to `integer` (use `floor`, `ceiling`, or `round` to get an integer near a given `real` or `rational`), and `complexes` with an imaginary part of zero will convert to `real`. The motivation for this convention, as well as other properties of conversion, will be discussed in Chapter 10.

3.2 Logical connectives

The type for logical values in E-L is `boolean`, with constants `true` and `false` having their obvious meanings. The main logical connectives that you will use are `and`, `or`, and `not`. They are less sticky than relational operators, so that `a = b and c = d` groups as you would expect: `(a = b) and (c = d)`. Also, `not` is stickier than `and`, and `and` is stickier than `or`, so `not a and b or c` groups as `((not a) and b) or c`. This is the convention in most programming languages.

The `and` and `or` operations have "short-circuit evaluation", meaning that they evaluate their arguments from left to right, and stop evaluation as soon as the answer becomes clear. Thus a condition like

```
x # 0 and f(x) / x gt 1
```

will never divide by 0, because the second condition will not be evaluated when the first one is false.

There are two other logical connectives that are occasionally useful:

x `xor` *y* means either *x* or *y*, but not both
x `eqv` *y* means *x* and *y* have the same `boolean` value

There is no question of short-circuiting here, because these functions require looking at both operands before the yielding the result. These expressions are not quite equivalent to their equivalents in terms of `not`, `and`, and `or`, only because their arguments are guaranteed to be evaluated only once.

3.3 Text

Characters in E-L have type `char`; they are denoted using \ in programs, for example, `\a`. Non-printing characters are indicated by spelling out the full name after the \, as in `\newline`. Except for the newline character, the built-in standard naming for characters follows official ASCII.

The built-in type `symbol` *describes tuples of characters. Symbol literals are denoted in*

programs using single quotes: `'abc'`. To include special characters inside `symbols`, surround the name for the character by \: `"\newline\\newline\"`. To include backslash and single quote in `symbols`, use `\\` and `\'`, respectively. In writing programs, symbol literals may not extend over one line. The concatenation operator `&` may be useful.

3.4 Input and output

You have already seen the input and output functions `read` and `write` in section 1.2. These functions are provided specifically for the purpose of transacting with the run window, and are based on the more basic functions `input` and `output`. The first argument of both of these functions is a `stream`. The second argument of `input` is an action to be performed when there is no more input; to use this in a program, you will probably want to consult section 5.4. The second argument of `output` is the value to be sent to the `stream`.

To obtain a `stream` to transact with a particular file, you can use one of the following functions:

```
open-input-stream(symbol)

open-output-stream(symbol)
```

The name given by *symbol* is a file name in the syntax of the host system. To indicate the end of transacting with a `stream` in particular, to indicate that the data sent to an `output-stream` is to become the data in the file, use:

```
close(stream)
```

Chapter 4

Compound Values

4.1 Tuples

The most basic compound object in E-L is a tuple, which is simply a value consisting of several other values. To create tuples, use angle brackets:

```
<5, \a, <1, 2>>
```

This is a tuple whose components are the integer 5, the character "a", and the tuple <1, 2>. To obtain a component of a tuple, use the square bracket notation:

```
<3.14, 2.718, 1.618>[2] = 2.718
```

There is built-in type constructor tuple associated with tuples. Its parameters are the types associated with each component. A type for the first tuple above is

```
tuple(integer,  char, tuple(integer, integer))
```

Conversion from one tuple type to another will succed if the tuples are the same length and conversion succeeds on the components.

The type constructors pair, triple, and ktuple are useful in specifying tuples where all the components have the same type:

pair(*t*) is inter-convertible with tuple(*t*, *t*)

triple(*t*) is inter-convertible with tuple(*t*, *t*, *t*)

ktuple(*k*, *t*) is inter-convertible with tuple(*t*, ... , *t*) (for *k* copies of t)

More generically,

ntuple(*t*) means ktuple(*k*, *t*) for some *k*

any-tuple means any tuple at all

To create a ktuple(*k*, *t*) with all components having the same value, use

```
ktuple(k, t)(value)
```

where *value* is convertible to type *t*. There are other ways to create tuples that we will see in connection with other constructs. The types any-tuple and ntuple usually appear only in function headers, so directly creating values with such types is not an issue.

The components of a tuple cannot be modified by assignment and thus tuples provide the means for modelling compound constants. For example, given the declarations

```
Constant t is <1, 2>;

Variable u is <1, 2>;
```

we may select either component of `t` or `u` and assign a new pair to `u`, but neither component of `t` or `u` can be modified individually.

4.2 Arrays

Arrays in E-L provide the means for modelling compound (homogeneous) values with state. As with tuples, there are several type constructors associated with arrays, having varying degrees of specificity. The most specific is:

```
array(dimensions, t)
```

where *`dimensions`* must convert to `ntuple(integer)`, and *t* is a type.

Arrays are introduced by an **Array** declaration, having the following general form:

```
Array name[dimensions]:type initially initialization;
```

As in a **Constant** or **Variable** declaration, the *`:type`* is optional, and if missing, is inferred from *`initialization`*. A common form for *`initialization`* is a single value, used as the initial value for each component of the array:

```
Array A[3, 5] initially 0;
```

Just as *`:type`* may be dropped, you may also drop `[`*`dimensions`*`]` when it is evident from *`values`*:

```
Array A initially <C, <1, 4, 9, 16, 25>, 3>;
```

Suppose `C` is an array of integers with dimensions <5>, and values at the time of this declaration of 2, 3, 5, 7, 11. Then `A` would be initialized to:

2	3	5	7	11
1	4	9	16	25
3	3	3	3	3

`A` can be thought of as a handle on the fifteen values `A[1, 1]`, `A[1, 2]`, ..., `A[3, 5]`. `A` cannot be modified, but its components may. For example, to set a component of the above array `A` to 0, say:

```
A[1, 2] <- 0
```

This means the same thing that it means in other languages (with perhaps differences in syntax): the value stored in the 1, 2 position of `A` is now 0. Moreover, when passing an array as a parameter, and the function changes an element, the change is reflected in the caller. For example, suppose that after either of the above declarations, the next statement is

```
clear-array(A)
```

You could write the function to do this as follows:

```
Function clear-array(B:array(<3, 5>, integer))
  For i in 1 to 3 do
    For j in  1 to 5 do
      B[i, j] <- 0;
```

(The reason that the `to` is not boldface is that it is not, strictly speaking, part of the **For in do** construct; see section 5.3.) After `clear-array(A)`, `A[i, j]` will be 0, for $1 \leq i \leq 3$ and $1 \leq j \leq 5$.

A function to clear a 3 by 5 integer array does not have wide applicability. In fact the easiest way to change all the values of an array is

```
A <<- initialization
```

The general rule is that *`initialization`* can be anything found in an the like-named position of **Array** declaration. If it is a value, each component of `A` is set to that value, so `A <<- 0` clears `A`. If *`initialization`* can be interpreted as a way to enumerate values (for example, a tuple), those values are stored in `A`. The multiple assignment operator **<<-** has much broader use than just arrays. .

Still, there is the problem that you would like to write functions that work on arrays with different sizes, and perhaps even different dimensions. In the header of such functions, it is desirable to give as precise a type as the function is intended for. The **?** construct seen in the example below is a convenient way both to state the specific type and to obtain the length of the array along various axes:

```
Function clear-array(B:array(<?m, ?n>, integer))
  For i in 1 to m do
    For j in 1 to n do
      B[i, j] <- 0;
```

The rule of section 2.5 about the conversion of actual parameters to the type of the formal still holds, but when **?** is in a type specification, the specification is actually something other than a type; see D4.4 and D6.4 (these are shorthand references to chapter and section of the E-L Definition. Nevertheless, there is conversion to "some type of the given form". However, unlike the situation for tuples, conversion does not proceed component-wise, because of the fact that arrays are collections of places to put things. An `array(<4>, integer)` is not an unfailing instance of an `array(<4>, real)`, because when you go to store a `real`, the place where you store it wants an `integer`, and conversion may not always be possible. There does not have to be identity of element types in arrays for conversion to take place, but that is the way to view things until section 12.3. For most programming, identity of types will suffice. Identity of dimensions is required in any case.

4.3 Records

A record is analogous to a tuple, in that its components may have heterogeneous types, and it is analogous to an array in that it is a collection of places to store values. Types for records are constructed as follows:

```
record(name1:t1, . . . , namek:tk)
```

The *`namei`* are the names of the fields and the *`ti`* are the types. A record may be declared similarly to an array:

```
Record r[x:real, y:real] initially (0.0, 1.0);
```

As with other declarations, you can omit the `:ti` parts, if they can be inferred from the initialization. You can access and modify the components using dot selection and ordinary <-:

```
r.x <- r.x * 1.618
```

Passing records as parameters works like arrays: assignment to a component of a formal changes the value in the caller, and you may not assign directly to the name introduced in a **`Record`** declaration.

Chapter 5

Flow of Control

5.1 Functions

In addition to their role as encapsulations of algorithmic patterns, functions provide the backbone for the flow of control constructs of E-L. Function call and return are the most direct ways that functions interact with flow of control, but there are other connections.

You have already seen how to define functions in section 2.5, and have implicitly seen how to call them:

```
function(argument1, ... , argumentk)
```

There are other ways to call functions, adopted from standard mathematical notation, and exemplified in section 1.3 (prefix, infix, postfix). All function calls behave the same, whether they are written in the standard way, or with some kind of fixity. Moreover, there is only one kind of function definition in E-L. Unlike some languages, there is no syntactic distinction between functions that do and do not return a result. (In E-L, functions yield zero, one, or more results, but that is not a flow of control issue; see section 6.3). There is also no syntactic distinction between functions that are or are not "pure" (see section 7.5). The issue of purity indeed arises in E-L, but it is not treated by syntactic variants of function definition or call.

5.2 Conditionals

The conditional in E-L is an if-then-else construct.

If *cond1* **then** *expr1 {***elseif** *condi* **then** *expri } [***else** *exprk]*

The italicized curly braces mean that there may be zero or more of the **elseif** clauses, and the italicized square brackets mean that the **else** clause is optional. The semantics are probably obvious; if not, see D5.1.

5.3 Iteration

The simplest iteration construct is the indefinite repeat

Repeat *body*

The associated *body* is evaluated over and over, until some flow of control construct, such as **Exit-function** (see section 5.4) causes an escape. Like **else**, **Repeat** is less sticky than semicolon:

```
Repeat read(); test(); adjust()
```

has a body consisting of three function calls.

Another simple iteration construct is the while-do:

While *condition* **do** *expression*

The stickiness of **do** is the same as that of **else** or **Repeat**. The semantics are as follows. On each iteration, the *condition* is evaluated. If the result is true, *expression* is evaluated and

iteration continues. If the result is false, iteration ceases.

At this point in the discussion of iteration in most languages, you would expect to see a description of a mechanism for stepping through a finite contiguous set of integers. In E-L, there is a more general mechanism for iterating over a set of values; iterating over integers uses this general mechanism:

```
For parameters in expression1 do expression2
```

The *parameters* introduce new names for use in *expression2*. On each iteration of the loop, these names are supplied with the values for that iteration; *expression1* specifies the set of values to be iterated over. As an example:

```
For i in 1 to k do expression
```

As we said in section 4.2, the `to` is not part of the **For in do** syntax, merely one way to describe a set of values to iterate over.You will see in section 11.1 how to write your own iterators.

5.4 Premature exit

It is occasionally convenient to exit a function "prematurely", yielding a supplied value. The general form of the statement is:

```
Exit-function {expression|,}
```

The curly brace notation with the vertical bar means a list of *expressions* (the class before the vertical bar), separated by comma (the class after the vertical bar). This allows the possiblity of an empty list, which means to exit the enclosing function with no results; one or more results are also allowed (see section 6.3). A classic example of premature exit is looking for a particular value to arise in an iteration.

```
Function search(values, v) -> boolean
  For x in values
    If x = v then Exit-function true;
  false;
```

The expressions of **Exit-function** become the result(s) of the innermost **Function** containing the call. This example has two other notable properties. First, the arguments have no declared types. This is quite common when writing highly generic functions, that is, ones that work over a large set of types. Second, an iteration is evidently something that can be passed as a parameter, and then used in a **For** loop. The only iteration construct you have seen thus far is `to`. The result of the `search(1 to 5, 3)` is `true`. And `search(1 to 0, ...)` will be `false`, no matter what the second argument.

5.5 Case expressions

The case expression allows a concise way to dispatch to one of several possibilities. Its simplest form looks for an alternative according to the equality of two expressions:

```
Case expression0 of
  expression1 => alternative1;
  ...
  expressionk => alternativek;
  [otherwise alternativem]
```

The result of this expression is obtained by finding the first *i* for which *expression0* = *expressioni*, and evaluating *alternativei*. If no such *i* is found, the result is obtained by evaluating *alternativem*, if the **otherwise** clause is present; if not, there is a Case Fault.

A more flexible form of the case expression allows an arbitrary test on each of the *expressioni*. The scope of the parameter in the header is limited to *expression0*.

```
Case(parameter) expression0 of
  expression1 => alternative1;
  ...
  expressionk => alternativek;
  [otherwise alternativem]
```

In this form of the case expression, *expression0* must yield a boolean. The result is obtained by finding the first *i* for which *expression0* is true when *expressioni* is substituted for *parameter*. The **otherwise** clause is handled as above. An example of this is a trichotomy on sign:

```
Case(test) test(expression, 0) of
  gt => (the alternative for positive expression);
  lt => (the alternative for negative expression);
  =  => (the alternative for zero expression)
```

The case expression may be generalized still further by allowing several expressions preceding each => (each list of the same length), and a *parameter* list of the same length.

Chapter 6

Functions

This manual assumes that you already know something about functions in other languages (known variously as procedures or subroutines) and has implicitly introduced E-L functions in sections 2.5 and 5.1. This chapter covers several topics in more depth.

6.1 Mutual recursion

Ordinary function definition allows for recursion, because the function name introduced is visible to the body of the function (see section 2.5). To define several functions, all of whose definitions depend upon each other in a mutually recursive way, use the plural:

```
Functions
   f(x,y)
      (body of f, involving f and g);
   g(x,y)
      (body of g, involving f and g);
```

The scopes of `f` and `g` each include both the bodies of `f` and `g`, as well as the scope of the declaration. The scope of the `x` and `y` formals of `f` is naturally the body of `f`, and nothing else. The `x` and `y` that are formals of `g` are a different `x` and `y`.

6.2 Functions as values

As you have already been warned, functions are values in E-L. There is a simple way to produce a function value without declaring a function:

```
lambda(formals)[ -> results] body
```

The *`formals`* phrase has the same form as in a function header. It specifies the names of formals and their types, perhaps using **?** (see section 4.3). The *`results`* part of the header specifies the types of the results. The function body is an ordinary E-L expression. The scope of the introduced names is exactly *`body`*. A lambda expression can occur anywhere an expression can occur. Its value is the function that it specifies.

Function declarations and lambda expressions are the ultimate source of functions in E-L; they provide the "constants" from which other function expressions can be built. We now discuss some operators that produce function values. The most important of these is *restriction*, from the mathematical notion of restriction:

> Given a function f of n arguments, a position i between 1 and n, and a value v, the restriction of f to v at the ith position is a function of $n-1$ arguments, given by inserting v after the $i-1^{st}$ argument and applying f to the (now) n arguments.

In E-L, restriction is written thus:

```
restrict(f, i, v)
```

Usually, one of the following operators is more convenient:

f | *v* means `restrict(f, 1, v)`

f || *v* means `restrict(f, 2, v)`

f ||| *v* means `restrict(f, 3, v)`

The operators |, ||, and ||| are built-in. If you want |||| or higher, you can define it yourself. For example:

```
Constant ||||| is restrict || 5
```

Think about it.

Another function-producing operator that is useful is composition, denoted in mathematics by $\circ$, and in E-L by o. Mathematically, $\circ$ is given by

$$(f \circ g)(x) = f(g(x))$$

The same rule governs the built-in operator o.

6.3 Functions and assignment

It is sometimes useful for a function to perform one or more direct assignments on behalf of a caller. The ability to do this is used much less often in E-L than in most other languages, because E-L allows functions to return multiple results (see the next section). It is not good E-L style to use the constructs described in this section merely to return several results, as would be necessary in some languages.

A legitimate example of a function that does assignments for the caller is that for exchanging the values of two places. There is a built-in function in E-L for this, called <->, but you could write it as follows:

```
Function <->(shared v, shared w)
  Constant t is v; v <- w; w <- t
```

The **shared** marker specifies a property of the names `v`, `w`, and <-> in their respective scopes, so that `x <-> y` works as expected: it exchanges the values of `x` and `y`. The effect is that of "var" in PASCAL or "in out" in Ada.

- Assignment to a **shared** formal effects assignment to the corresponding actual parameter.

- The value of a **shared** formal parameter is the value of the actual parameter at the time of reference (*not* the time of call).

These rules are not mathematically precise, but they should be sufficient for understanding **shared** formals. There are more details in D4.

The **shared** marker may be used in a **Let be** declaration in the same way that it is used in a **Function** heading (there will be further discussion of **Let be** declarations below). Such a declaration is typically used to provide a short name of some element of a compound object that will be both used and assigned:

```
Let shared pivot:real be A[i, j]
```

Assignments to pivot are the same as assignments to A[i, j], *for the values of* i *and* j *at the time of the declaration of* pivot.

Even more rarely than you use a **shared** formal, you will want a **shared** result. As an example, suppose that in connection with some non-deterministic program, you wish to modify one variable or another, choosing randomly between them. You could define:

```
Function either(shared x,  shared y) -> (shared)
  If random-real() lt .5 then x else y
```

You would then be able to increment either x or y randomly.

```
Let shared c be either(x,y);                    (*)
c <- c + 1
```

As you would expect from looking at this fragment, either is called only once, yielding the variable to be updated. This is *not* equivalent to:

```
either(x, y) <- either(x, y) + 1
```

Here, it is possible for x to get the value y + 1. A more concise way to achieve the effect of (*) is to define an operator <-+, which increments its first argument by its second.

```
either(x, y) <-+ 1
```

You can implement <-+ using **shared**, <-, and +; in section 14.5 we will describe how to establish <-+ as an infix operator with the appropriate precedence.

In using a function with a **shared** formal, it is necessary to supply it with an argument that can reasonably be interpreted as a place where values can be stored. If you write x <-> 3, you will get a syntax error because there is no way to interpret 3 as such a place. The term "**shared** position" will describe arguments to a function that has a **shared** formal at that position, and to a result of a function if the result is **shared**. Clearly, the left argument of <- is a **shared** position. The rule of thumb is that the only syntactically legal constructs in a **shared** position are:

- variables (which may always be interpreted as a place)
- applications of functions whose result is **shared**.

This is only a rule of thumb, because certain constructs are transparent to the **shared** marker. For example, the result of either is shared, and its body is an **If then else**. In this situation, the syntactic limitations then apply to the two alternatives. If *expr1; expr2* is in a **shared** position, then *expr2* is in a **shared** position.

All this may make you wonder how c in the either example, which in some sense must be a place, could be an operand to +. Again, the syntactic effect of **shared** is felt:

- If a name introduced with **shared**, a variable, or an application of a function whose result is **shared** appears in an un**shared** position, an implicit "contents" operation is supplied.

Like the first rule, this rule filters through **If then else** and ;. The right argument of <- is an un**shared** position. The two rules of this section will suffice for most programming. For the full story, see D4.

In all of the examples of this section, **shared** formals and results have not had types associated with them. To do so, you can say, for example:

```
Function f(shared x:real) -> (shared real)
```

The rule of section 2.5 about argument and result conversion still holds, but it must be understood that the target type of conversion is not **real**, but a type that describes a place to store **reals**. Without going into the details of types here, suffice it to say that supplying a variable whose type is **real**, or the application of function whose result is **real shared**, will always work. Anything else is a more delicate matter (see section 12.3).

The **shared** marker is called a *bindclass*. There are other bindclasses, among them **variable**. When used in the header of a function, the **variable** bindclass allows assignments to the formal parameters, but unlike **shared**, assignments to the formal are guaranteed *not* to affect the actuals, and when a function has a **variable** bindclass in a certain position, it is legal to supply constants for actuals at that position. You may also use **variable** in a **Let be** declaration. The declaration "**Variable** x **is** 0" is equivalent to:

```
Let variable x be 0
```

You may assign directly to the formal parameter only when it is modified by a **shared** or **variable** bindclass. If you do not specify a bindclass, there is an implicit **constant** bindclass, and you may insert this if you wish to emphasize the point. The declaration "**Constant** x **is** 0" is equivalent to:

```
Let constant x be 0
```

As in function headers, the **constant** bindclass is the default in **Let be** declarations, so that the above declaration is also equivalent to:

```
Let x be 0
```

6.4 Multiple results

Functions in E-L can return multiple results. The types of the results may be specified by a list of types after the -> in a function header

```
Function sign-mag(x:integer) -> (boolean, integer)
  If x lt 0
    (true, -x)
  else
    (false, x);
```

Conversion to the result types is done result by result.

One way that multiple results are used is in a **Let be** declaration.

```
Let sign, mag be sign-mag(...)
```

When a **Let be** declaration is used to distribute multiple results to several names, it is possible to mix bindclasses, and in fact, they must be specified individually for each name—in the above example, both `sign` and `mag` have the default bindclass, **constant**. The following example could not be used with the `sign-mag` function, because that function does not return a **shared** result, but it might be reasonable in other situations:

```
Let shared x, y be ...
```

The name y has the default bindclass, **constant**, *not* **shared**.

An application of a function that yields multiple results can appear in the argument list of another application. When this happens, all of the results become arguments to the outer application, as if they occurred with commas between them. For example, suppose we define

```
Function combine-sign-mag(sign:boolean, x:integer) -> integer
  If sign then -x else x
```

Then `combine-sign-mag(sign-mag(...))` is the identity function on values of type `integer`; similarly, `sign-mag(combine-sign-mag(...)` is the identity on pairs consisting of a `boolean` and a non-negative `integer`.

6.5 Pure functions

A function is pure if it obeys the following two properties.

- A function is *side-effect-free* if it does not change the contents of any place.

Assignment is of course not side-effect-free. The second property of interest is:

- A function is *memory-less* if its effect (including side-effects and results) when called on constants does not depend upon the value stored in any place.

Selection from an array is not memory-less.

- A function is *pure* if it is side-effect-free and memory-less.

Functions like + and = are pure.

Chapter 7

Program Preparation

7.1 Documents

Large programs require documentation, and E-L is designed to make this documentation easier to prepare and of more use than systems you may be accustomed to. Perhaps the best introduction to documentation in the E-L style is to call attention to two items that you will *not* find in E-L. First, there is no such thing as an ".E-L file" containing E-L code, analogous to a .PAS file containing PASCAL code, or similarly for other languages. Second, there is no comment statement in E-L like that found in other languages.

To write an E-L program, you prepare a *document*, using the E-L word processor. In a well documented program, most of the document will be English. Scattered through the document will be E-L fragments, mostly declarations of functions. The facilities for pulling these fragments togother are powerful, and, with a few exceptions, you are not obligated to structure these fragments in any particular way. Instead, you should focus on explaining your program in the clearest possible way. This may be top down, bottom up, inside out, utilities first or last or interleaved—whatever seems most natural for your problem.

One of the characteristics of the E-L word processor that E-L relies on is the ability to edit a document "on the surface", in the style of a what-you-see-is-what-you-get system, *as well as* the ability to edit the "underlying text" of a document, in the style of Scribe or T$_E$X. Commands to the word processor (necessarily in the underlying text) are set off by the backslash, and arguments are in braces, in the style of T$_E$X. Context should make it clear that these commands are not part of the E-L syntax.

7.2 Editing

When you begin editing a new document, the editor is in "English mode". The details of using the word processor may be found in its user's manual. You will find there all you need to know about chapters and sections, displays, symbolic cross-referencing, indexing, multiple views, and the like. For present purposes, what you need to know is how to get into "E-L mode". To enter E-L mode, type ↑L. Once in E-L mode, the editor is sensitive to E-L syntax. For example, when you hit return, the cursor returns to the expected indentation point. A backspace will move it left (to leave an indentation level), or a tab will move it right (to indent further). You can also "keep on typing", and watch as the editor formats your program. Once you have finished typing in an E-L fragment, you can leave E-L mode by again typing ↑L. When you move your cursor into an existing E-L fragment, the editor automatically enters E-L mode; when the cursor is an English fragment, it is automatically in English mode.

A useful feature of E-L mode is the ability to select *phrases*, in the technical sense of section 1.3. If the cursor is sitting on a lexeme, a click—or ↑K— will select the lexeme. A second click will select the next largest phrase beginning at the lexeme, and so on until the largest phrase beginning at the lexeme is selected. Another click deselects. For example, suppose the cursor is on the a.

```
(a - b*c + d)          (after one click)
(a - b*c + d)          (after two clicks)
(a - b*c + d)          (after three)
(a - b*c + d)          (after four)
(a - b*c + d)          (after five)
```

A selected phrase may be cut, pasted, moved, exchanged, and so on, just as selected portions of text in English mode.

7.3 Breakouts

You are strongly encouraged to write small functions in E-L, but sometimes, you will want to present a function in even smaller fragments. A motivating factor is that an E-L fragment cannot be split across a page boundary in a document. A convenient way to present the definition of a function piecemeal is to use "breakouts". These are visually easy to spot, because they are in a different font from standard E-L.

```
Function f(x:integer, y:integer) -> integer
  If p(x)
    g(y)
  else
    (handle the case in which f depends on both x and y)
```

Then, in some other part of the document, you can see:

```
(handle the case in which f depends on both x and y)
  h(x, y)
```

To achieve this, you type in E-L mode until you get to the point where you want the breakout, and then type ↑B. You will be queried for the key, which will presumably be shorter than the long description that appears; in the above example, you might choose *fxy* for the key. If you have never used this key before, you will be queried for the description; it is at this point that you type in the description (this is automatically in English mode), exclusive of parentheses, and this will appear at the point you typed ↑B. If you have used the key before, you will immediately get the description text associated with the key. To edit an existing description, put the cursor on one of its appearances, and simply edit (you will be in English mode). Both appearances will change. To find out the name of the key associated with a breakout, look at the text underlying the description.

When a breakout appears as the header of an E-L fragment, the rest of the breakout is the "definition" of the breakout. An embedded breakout is a "reference". When E-L extracts a program from a document, it substitutes the definition of a breakout for the reference, with indentation adjusted to the level of the reference. Breakouts are not a macro facility. When your document is complete, every breakout key should have exactly one definition and one reference (with the exception of one key that we discuss below). The purpose of breakouts is to allow you to think of and present a program in bits and pieces.

In section 1.2, you responded to the document query of the E-L window with a carriage return, and placed an expression in a temporary window. In large programs, you will supply the name of an actual document to E-L, where the document contains the "main program". You specify a main program by giving a definition of a breakout whose key is ***main*** (this is a key that your document should not reference). The description of the breakout is anything you like, and the E-L fragment is whatever you want for your main program. What you will see in your document is something like:

```
(This is the main program for the ZYX system)
  print-output(analyze(query-the-user()))
```

Assume that all keys except ***main*** have one definition and one reference. Starting from the fragment with the key ***main***, it is possible to back-substitute fragments until there are no more breakouts. The only fragments that are not back-substituted into the main program are those that do not have a breakout as the header. When this occurs, E-L has some choice in how it pulls the program together. Suppose that an unlabeled fragment is the declaration of a function. It is then likely that your main program—that is, the fully back-substituted version of it—has the name of the

function as an undeclared identifier. If so, then E-L will incorporate the unlabeled fragment that declares the identifier into the program, using the general rule that a declaration is placed to have the smallest possible scope consistent with covering undefined references to it. In the process of incorporating your fragments, the presence of mutual reference may cause several **Function** declarations to be turned into one **Functions** declaration (see section 6.1). You may want to understand in detail where your fragments have been placed. To do so, use `\declaration-structure{`*`key`*`}` as the underlying text; on the surface you will see who is declared inside what, beginning with the declaration whose header has the key *key*.

If your document contains the definitions of all the non-built-in functions that you use, you have a complete program. In Chapter 14, we will return to multi-document programs, libraries, and related issues.

7.4 Annotations

Annotations are a way to associate information with a program, without including information directly in the program text. There are several *annotation spaces* that come with E-L, and you will later see how to add to these. Some annotations you provide, others are supplied by tools. Annotations provide a uniform mechanism for the many different kinds of information that you and tools will want to specify or use.

In annotating E-L fragments, the annotated object will always be a phrase, in the usual sense of section 1.3. You may thus annotate something as small as a lexeme, up to the largest phrase you have. The phrases that are reasonable to annotate depend upon the particular annotation space. To illustrate the use of annotations in a concrete way, we will discuss here the "check" annotation spaces. In this case the annotations are themselves E-L code, to check that certain conditions hold. Suppose that you expect a certain condition to hold at the beginning of the evaluation of a given phrase, and that for purposes of program development, you want to actually check that the condition holds, but for the production program, you do not want the check to be made. The way to do this in E-L is to use the `entry-check` annotation space. First, using the technique of section 7.2, select the phrase whose entry condition you want to check (to make an entry check for a function, select the entire body). Then, type ↑A, and you will be prompted for the name of the annotation space (in this case, `entry-check`), and for the key, to name the annotation. To define the annotation, put the cursor at the point in the document where you want the annotation to appear, and with no text actively selected, type ↑A. If you have just selected a phrase for annotation, the editor knows the key of interest to you; if the editor cannot figure out which annotation you want, it will ask for the key. When you are putting an entry into the `entry-check` space, the ↑A will automatically put you into E-L mode. An annotation will look something like:

```
Annotation(entry-check) reference to the annotated phrase
   x # 0
```

The first line of this annotation is set up by the editor. The *reference* part allows you to easily locate the phrase being annotated (you have some control over this, but live with the default for a while). The second line of this annotation has to be supplied by you. The scope of variables in a check annotation is the scope of the annotated phrase. To terminate an annotation, again type ↑A.

Annotations are connected by the key you name, even though you don't see the key in the document. If you change the annotated phrase, the annotation persists, although this may change what you see in the *reference* part of an annotation header. If you delete an annotated phrase entirely, the *reference* will become blank. If you then annotate another phrase with the same key, the *reference* will be revised. If you annotate several phrases with the same key, the *reference* will indicate all such phrases.

The meaning of entry checks is that if you are debugging, the check is made at the point just before evaluating the annotated phrase (by the way, this means that you can have entry checks only on phrases that corresponding to actual evaluations—if comma is the main "op" in your phrase,

there will be complaints). If the result of the check is `true`, execution continues; otherwise, you will get a Check Fault. There is also an `exit-check` annotation space, that works in the same way except that the test is made after the evaluation of the phrase. If the evaluated phrase yields no results, it is written just like an entry check. If there is a result, the `exit-check` can look at the result, by means of an expression we will discuss later.

Another built-in annotation space is named `comment`. This is E-L's substitute for the comment statement. The annotation here is English text. It is not examined by any tool other than the word processor. There are other built-in annotation spaces that you will encounter later in this manual.

7.5 Testing

When you are trying out a new program, one of the first things that you may want to do is to set a breakpoint (not to be confused with the breakouts described earlier). Select the phrase that you want breakpointed, and type ↑P (for "pause"—↑B is for "break*out*"). There is a menu for selecting various options on your breakpoint. Typing "return" gives the default: your program will pause just before executing the selected phrase and just after.

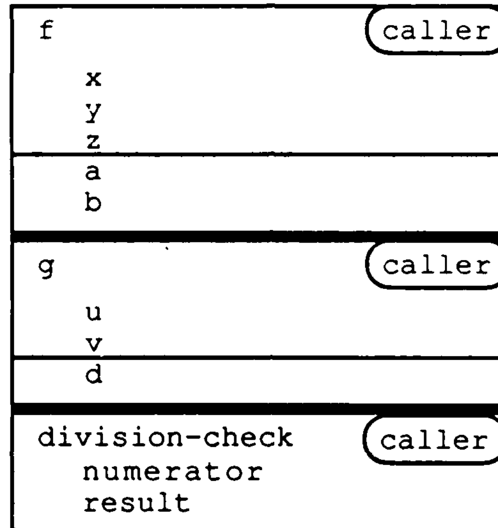
When your program stops, either at a breakpoint or because of a fault, a window associated with the stop will appear; the "run" window (see section 1.2) is not affected. The header of the stop window will contain a message associated with the stop. Most of the window is used to display the stack, which you may use to determine what state your program was in when the stop occurred. The arrows on the side of the window allow you to scroll through the stack as if it were an ordinary document. The stack is a sequence of frames, corresponding to function entries. The frame for the main program will be at the top of the window. The first item in a frame is the name that appears in the **Function(s)** declaration. Clicking on this name will open a window on the document containing the function definition (if one is not already open), and will put the cursor in that window at the declaration of the function, so that you see not only the function itself, but also the surrounding English text. Just to the right of the name of the function is the "caller" button. Clicking this will move the cursor in your document window to the code that applied the function, and will highlight the application responsible for the frame. The next items in the frame will be the arguments to the function, in their natural order. Clicking a name once will display its type; clicking twice will display its value. If the function has any local variables, they will be separated from the arguments by a line, and will otherwise be listed exactly as the arguments. Their types and values may be examined in the same way. The last frame in the stack corresponds to the handler for the fault or breakpoint. Clicking its caller button will highlight the phrase responsible for the stop. In a breakpoint, this will be the breakpointed phrase. In an argument fault, the highlighted phrase will be the argument that did not convert to the type expected by the function. The arguments and locals in any handler's frame may be examined to determine information of interest about the stop.

After double clicking an argument or local, you may edit its value, thereby replacing the old. You may change the values of "constants" in this way, as well as variables. It is sometimes useful to use this facility to change values of locals in a handler's frame, for example the counter in a breakpoint. Many handlers have a local named **result**. In a recoverable stop, its value replaces what would have been the caller's result when you resume from the stop (click the resume button on the stop window). If you wish to return a value other than the one set up, just edit **result**.

As an example of this, let us consider a function **f** that applies a function **g**, which divides by zero. The source might look something like:

```
Function g(u:..., v:...)
  Let d be 1
  Let e be d/0
  ...
Function f(x:..., y:..., z:...)
  Let a be ...
  Let b be g(a, a)
  Let c be g(a, b)
  ...
```

When the division by 0 occurs, the last three frames will look like this:



If you click the caller button for `division-check`, you will see `g` in its edit window with the guilty application highlighted:

```
Function g(u:..., v:...)
Let d be 1
Let e be d/0
...
```

Clicking the caller button for **g** locates the appropriate application in **f**:

```
Function f(x:..., y:..., z:...)
  Let a be ...
  Let b be g(a, a)
  Let c be g(a, b)
  ...
```

Chapter 8

Types Revisited

8.1 Types as values

We have already said that types are values, and have strongly implied that every value in E-L has a type. Since `integer` and `real` are values, it is reasonable to ask what their type is. The answer is, their type is `type`. Naturally, the type of `type` is also `type`. All this may take a little getting used to, but as we shall see later in this chapter, there are several situations in which a `type` value is treated like any other value, rather than as something special.

8.2 Introducing literals and constructors

You are by now familiar with several built-in type literals. It is useful to be able to introduce your own. The header for the declaration is quite simple:

Type *name*

Like all declarations in E-L, this hides outer declarations of *name*, and within the scope of this declaration, *name* is a type literal (a constant whose type is `type`), unless hidden by inner declarations.

You have already seen the type `array(<...>,...)` in section 4.2. The `array` in this expression is a built-in *type-constructor*. In the same spirit as type literals, you may introduce your own. The declaration is:

Type-constructor *name0(name1:type1, ... , namek:typek)*

The name of the constructor is *name0*. The other names are names of the components. To use a type constructor, you pretend that it is a function, as with `array`. The resulting value is a `type`. If you print such a value (for example, by exploring the run window), the type will appear in the same form that you write it, but with constants filled in. For example, consider the last version of `clear-array` in section 4.2, the one with the **?**s in the function header. If you happen to interrupt a program while it is in `clear-array`, and look at the type of `B`, it will be `array(<3,5>,integer)` if `clear-array` was called on the array `A` earlier in that section.

A type that is the result of a type constructor can be decomposed into the pieces that were used to construct it. In most programming, the best way to do this is indirectly, with the **?** technique that was illustrated in section 4.2. It may also be done directly by looking at various fields of the type. The `constructor` field of a type gives the constructor, and the *namei* field of a type, where *namei* appeared in the definition of the type constructor, gives the value of the corresponding field. The declaration of the built-in constructor `array` begins:

```
Type-constructor array(dimensions:ntuple(integer),
                       elm-type:type)
```

(An argument to a type constructor is sometimes a `type`, but as we see in this example, is sometimes not a `type`. This is one of the situations in which a `type` value is on a par with other values.) A type that is the result of type constructor can be decomposed into the constructor and arguments to the type constructor. Let `t = array(<5>, integer)`. Then:

```
t.constructor = array

t.dimensions = <5>

t.elm-type = integer
```

The `constructor` field of a `type` literal, whether built-in or defined by you, is `nil`. For any type that is the result of a constructor, the `parameters` field gives the labeled tuple made from the arguments to the constructor. For obvious reasons, it is not a good idea to use `constructor` or `parameters` as the names of fields of a type constructor that you declare.

8.3 Base type

The values of a new type are often based on another type. This latter type is specified as part of the declaration for the new type. For example, let us declare the type `mod-2`, thinking of the integers modulo 2, and let us base this type on `boolean`:

```
Type mod-2
  Based-on boolean
```

When you introduce a type constructor and specify a base type, you implicitly specify the base type for every type in the new set of types. Sometimes, the base type is independent of the parameters of the constructor, as in:

```
Type-constructor modulo(modulus:integer)
  Based-on integer
```

This evidently intends to represent the integers modulo *n* by `integers`. It is also possible to parameterize the base type by the parameters of the type constructor:

```
Type-constructor triangular-array(n:integer, elm-type:type)
  Based-on array(<n * (n+1) / 2>, elm-type)
```

When you start writing functions for your new types, you will do so using the base type.

A type must include exactly one base type declaration, unless it is declared to be an expandable union (see section 8.6).

8.4 Sets as types

E-L has set constructing operators, `{` and `}`, for producing fixed, finite sets. For example:

```
{0, \0, '0'}
```

specifies a set whose values are the `integer` 0, the `character` 0, and the `symbol` whose single character is 0.

The same notation can be used to define a type whose values are limited to a fixed, finite set of values. For example:

```
Variable x:{0, \0, '0'} be ...
```

introduces `x` with such a type. The rule for converting a value to such a set type is of course that it equals (via =) one of the elements of the set, and converting from such a type begins by taking the type of the particular value in hand and continuing conversion from there.

8.5 Finite unions

The type producer `union` (technically *not* a constructor) yields a type that describes all values of several possible types.

```
Function f(x:union(char, symbol))
```

The point of declaring `x` to have this type is that `f` may be presented with a `char` or a `symbol`, but not just any arbitrary value.

For a type t = `union` $(t_1, \ldots, t_k)$, the various t_i are called *variants* of t. The fundamental rule for union types is that a value v will convert to a union type t if and only if it will convert to one of the variants of t. Thus, the rule for conversion of an actual parameter to the type of the formal (section 2.5) still governs entry to `f` in the above example.

Suppose that `union` is used in variable declaration:

```
Variable z:union(char, symbol) is ...;
```

Assignments to `z` work exactly as described in 2.4. They do not change the type of `z`— throughout its scope, that type continues to be `union(char, symbol)`. Further, conversion to the `union` type is performed. Because any `char` will convert to the type of `z`, an assignment like `z <- \a` will not fault; since `symbol` will convert to this type, assignments like `z <- 'abc'` will not fault; thus you can freely assign to a variable having a union type any value that converts to a variant of the union.

Since the type of a parameter or variable does not change, the question arises, how to get back the type of what went into the union? First, it is almost never necessary to worry about this. When you write a function with a `union` type parameter, it is likely that you don't really care which of the variants occurs, and you will treat the parameter uniformly, usually by passing it to another function that takes an argument at least as generic. For example, in the function `f` at the beginning of this section, it would be reasonable to find `print(x)`. The use of function families (see the next chapter) to discriminate among union alternatives provides another mechanism that avoids having to ask explicitly, "what went into the union"?

When you really do care, there is a convenient case construct designed especially for the purpose. Inside `f`, you might write:

```
Type-case x of
   char => (whatever you want to do for chars)
   symbol => (whatever you want to do for symbols)
```

There are yet other means, which are described in D3.1.

8.6 Infinite unions

By its very nature, you can list only a finite set of alternatives with `union`. But suppose that you want a much larger union. For example, we have already seen `any-tuple`; this is an infinite union of certain kinds of types. Suppose you want to define `any-modulo`, meaning `modulo(`*n*`)` for any *n*. This is possible not because E-L understands a naming convention, but by using an infinite union. The type you want is:

```
Constant any-modulo is union(n) modulo(n);
```

Here, **union** is in boldface in order to distinguish it from `union` used above. The **union** notation has the following general syntax:

```
union(parameters) expression
```

The *parameters* may list any number of names, and these may have associated types. The rules for a legal *expression* are given formally in D1.4, but you will always be safe if you stick to expressions that are formed by combinations of type constructors, `union` and **union**, constants, and names mentioned in the *parameters*.

The scope of names in the *parameters* is limited to *expression*. Further, the resulting type is independent of the names you choose. Imagine replacing the **union** by **lambda**, so that the infinite union becomes a function. A "variant" of the infinite union is any type that may be obtained by applying the lambda expression to a set of values. With this understanding of "variant", the fundamental rule for infinite unions is analogous to the rule for finite unions: A value will convert to an an infinite union *t* if and only if it will convert to one of the variants of *t*. The largest union of all is `all`, a built-in constant equal to **union**`(t:type)t`, in other words, the union over all types. Use it sparingly.

8.7 Expandable unions

In introducing a new type or type constructor, we saw that one possibility is to base the new type on an existing type. Another possibility is to declare the new type to be an expandable union, meaning that it will have some finite set of variants, but that these will be declared later.

```
Type t
  Expandable-union;
```

In other declarations within the scope of `t`, types can be declared to be variants of `t`.

```
Type ti
  Variant-of t;
```

In a scope where an expandable union *t* and variants $t_1, \ldots, t_k$ have been declared, their behavior is exactly as if t = `union` $(t_1, \ldots, t_k)$.

8.8 Cyclic types

To adequately specify certain types, it is necessary for the definitions to be self-referencing. For example, a list is either `nil`, or it is:

```
record(goods:elm-type, next:(a list))
```

To specify such a type, use **cyclic**, with syntax like that for **union**:

```
cyclic(parameters) expression
```

The *expression* has the same restrictions as the one for **union** (and we now add **cyclic** to the list of constructions that are safe in such *expressions*). For a realistic example, we would base the type constructor `list` on a cyclic type:

```
Type-constructor list(elm-type:type)
  Based-on
    cyclic(t)
      union({nil}, record(goods:elm-type, next:t));
```

Note the use of a set of a single element to give a type to indicate the end of a list.

To obtain several types that are mutually self-referencing, use one name in *parameters* for each such type, and have *expression* return one result for each of the types. A **Let be** declaration may be used to capture the results. For example, to define bi-partite trees with alternate nodes holding `integer` and `boolean` labels, you can use:

```
Let tree1, tree2 be
  cyclic(t1, t2)
    (record(goods:integer, left:t2, right:t2),
     record(datum:boolean, left:t1, right:t1));
```

Chapter 9

Function Families

9.1 Introduction

Most programming languages have some notion of a *family* of functions in that they provide several distinct functions—the family members—that have the same name. Examples include =, +, and <-. One reason for providing function families is to avoid having to introduce a new name for each family member. Furthermore, members of a given family are, in a strong sense, "doing the same thing". E-L has a number of function families started for you, such as = and +. You may also start your own families, and you may extend any family by adding new members.

The E-L philosophy about function families is that the family members are "doing the same thing" in a formalizable and useful way. In particular, analysis and transformation of E-L constructs is based on the properties of a family and not on properties detected in the implementations of the various members of the family. Thus when you extend a built-in family you must do this in a way that is consistent with the assumptions that are made about that family. (We provide an example of such assumptions in section 9.2, where we discuss the = family.) When you define your own family you will establish the properties that are to be assumed about its members, and it is expected that you will adhere to them when defining extensions.

A function family is characterized by a set of *parameters*. For example, the equality family has a single parameter, the type of the values that are being compared. When a family is extended, it is extended for particular values of its parameters.

One form of the declaration for extending a function family is:

Extend *family* **for** *parameters* **with** *member;*

where we are extending the family named *family* with a new *member* for the particular *parameters* given. Suppose we want to extend the = family for mod-2s. We would use something like the following **Extend** declaration.

Extend = **for** mod-2 **with** (code to test equality on two booleans)

An **Extend** declaration may stand on its own, or it may occur as a sub-declaration of a type or type constructor declaration. Within such a declaration, several conventions apply. First, the type being defined, or the result of the type constructor being defined, may be referred to as **$** in the *paramters*, and, in the case of a one parameter family, the **for** clause may be omitted, meaning the same thing as **for $** . Second, certain arguments of *member* will be treated be of the base type, not the type being declared. This is important because all we may have in the body of *member* are the functions defined on the base type; after all, in an extension we are in the process of extending those functions to the new type. Third, if *member* returns a value of the base type, it may be considered to have the type **$.** The particular argument and result positions that are treated in this way depend upon the family definition, about which we say more in section 9.5.

Because you may want to think of an **Extend** declaration as a function definition for *member* and because very often *member* would be a lambda expression, we offer the following convenient alternate notation for an **Extend** declaration:

Extend *family(formals) [-> (results)]* **for** *parameters* **with**
body;

Thus, we could define the mod-2 extension in the following way:

```
Type mod-2
  Based-on boolean;
  Extend = for $ with =;
  Extend print(x) -> () with
    print(if x then 1 else 0);
    print('mod 2');
```

The first extension says that the = family is extended with a member that allows us to compare two mod-2 values using the equality function for booleans. In the body of the print family extension, x, has the base type, boolean.

In extending a family for types formed with the type constructor, say modulo, a problem arises. You could say:

```
Extend family for modulo(3) with (whatever);
```

But this extends *family* for only one modulo type. You may want to extend the entire set of types at once. This is done with the **?** notation:

```
Extend family for modulo(?n) with (some function depending upon n);
```

The scope of n does not extend beyond the **with** clause. In a sub-declaration of a type constructor definition, the **?**n is not needed because we can use the parameter in the header of the type constructor definition.

```
Type-constructor modulo(modulus:integer)
  Based-on integer;
  Extend print(x) -> () with
    print(x);
    print(' mod ');
    print(modulus);
```

9.2 Equality

The prototypical function family is equality, denoted in E-L by the traditional symbol =. Equality is a single parameter family, whose paramter is the type of its arguments. The equality function that arrives with E-L obeys the traditional rules of the mathematical definition of an equivalence relation.

EQ1 For any value v, $v = v$ is true.

EQ2 For any values v and w, $v = w$ if and only if $w = v$.

EQ3 For any values v, w, and x, if $v = w$ and $w = x$, then $v = x$.

The traditional properties of reflexivity, commutativity, and transitivity must be augmented in a programming language context by the additional requirement:

EQ0 The function = is pure and yields a boolean.

Programs with = are analyzed and transformed according to these rules.

When you define a new type or type constructor, you may want to define equality for it. An argument can be made that equality is among the first things to think about when you define new types, along with generation (see the next section). Equality is often most conveniently extended in

a sub-declaration of the type or type constructor declaration, in which case the formals of the function named in the **Extend** are of the base type. The function you supply must determine whether the two values would be equal if they were considered to be of the newly declared type. With may one-parameter families, like equality, the new family member is essentially the same as the family member for the base type. Because the situation arises frequently, there is a shorthand notation for it. In defining the new type `mod-2`, we could have extended = in the following way, using the **Induce** sub-declaration:

```
Type mod-2
  Based-on boolean;
  Induce =;
```

Later, we will see examples in which the equality function is more interesting.

Let $=_t$ be the equality family member for type *t*. It must obey EQ0-EQ3, where *v*, *w*, and *x* range over values of type *t*. If you extend equality in this way, then = will continue to work as advertised.

The opportunity to supply an equality extension allows the possibility of doing so incorrectly. You are justified in asking, "what happens if an equality extension doesn't obey the axioms?" The answer lies in the fact that the analysis and transformation machinery relies on the axioms of equality, not on your implementation. If you do it wrong, your program may be incorrectly analyzed or transformed—the practical effect is that it may behave differently when transformed.

9.3 Well-defined functions

Most functions that you will write have the following property.

> A function *f* is *well-defined*, if for two sets of arguments $v_1, \ldots, v_n$ and $w_1, \ldots, w_n$ satisfying the condition $v_i = w_i$ for $i = 1, \ldots, n$, occurrences of $f(v_1, \ldots, v_n)$ may be replaced by $f(w_1, \ldots, w_n)$ without affecting the computation.

This says that $f(v_1, \ldots, v_n) = f(w_1, \ldots, w_n)$, and if *f* has side-effects, that the side-effects must be the same. If *f* is pure, this reduces to the usual mathematical definition of well-defined. The purpose of this definition is to connect equality with transformation. One of the most basic transformations is the substitution of one value that is equal to another.

Well-definedness is not a vacuous property. There are several functions built in to E-L that are not well-defined. For example, the function that takes a value and produces the corresponding value as a base type is not well-defined, because values may be equal when considered as objects of a given type and have different values as base types. This document says nothing of such functions, because it tries to encourage programming at a well-defined level. Even so, the potential for difficulties arises. For example, suppose that you extend both = and + in sub-declarations of a type declaration. Because you are implementing functions by operations on underlying types, your + may not be well-defined, and, as you will see in section 10.4, this is indeed something to worry about for +. Writing well-defined functions is often the crucial step in providing a layer of abstraction. An implementation must necessarily operate on the underlying representations, but if the functions provided are well-defined, analysis and transformation can be done at the higher level, well separated from the implementations.

9.4 Generation

In this section we explain how to generate values of a type that you have introduced. This is done with the `generate` family of functions. Calls on `generate` have the following form:

generate(*target-type*, (the values from which the new value is to be generated))

There is a short-hand notation for calling generate, which you have already seen in the context of complex numbers. The above may also be written:

target-type((the values from which the new value is to be generated))

Thus, complex(1.1, 2.3) is the same as generate(complex, 1.1, 2.3). Generally the latter form is used in the case of a simple type, but is avoided when the expression becomes awkward visuallv. Like all function families in E-L, the generate family obeys certain rules. But not many:

GEN0 The function generate is pure and well-defined.

GEN1 The type of the result of generate is the first argument.

This family is parameterized by two types: the type of the value to be generated, and the type of the value from which generation can occur. The most general form for extending the family is:

```
Extend generate for target-type, source-type with member;
```

When you extend generate with a sub-declaration of a type declaration, one of the parameters will be **$**, the new type being declared. If this is the *target-type*, *member* must produce a value of the base type and its argument will be *source-type*. If the **$** is the source type, the second argument of the member function will take a value of the base type, and the result must have type equal to the other parameter. We might want to produce mod-2 values from either a boolean or an integer and use mod-2 values to produce integers:

```
Type mod-2
  Based-on boolean;
  Induce =;
  Extend generate for $, integer with is-odd;
  Extend generate for $, boolean with identity;
  Extend generate(b) for integer, $ with
    If b then 1 else 0;
```

The function is-odd tests an integer for being odd, and identity returns its argument.

Whenever **$** is the source type for a generate extension, some thought must be given to whether the extension is well-defined. When equality is induced, any function that is itself well-defined will suffice, but if equality for the new type causes values that differ as base type to become equal as the new type, you must make sure that the member returns values that are equal when given base type values that will be equal as new types.

When declaring a new type, the = and generate extensions must be designed jointly. Consider types of the form modulo(*n*), which we have based on integer, and for which we specified induced equality (in section 2). Both these decisions were made with the assumption that whenever a value has type modulo(*n*), the integer representative is restricted to some interval of length *n*, say between 0 and *n*–1. Thus, for modulo, we have a more interesting generation function:

```
Type-constructor modulo(modulus:integer)
  Based-on integer;
```

```
Induce =;
Extend generate for integer with remainder || modulus;
```

(The function remainder yields the remainder of dividing its first argument by its second. See section 6.2 for ||.) The correctness of = for modulo(*n*) relies upon the fact that generate produces representatives with values in a suitable range. This same rule must be followed by other functions that produce values of the base type.

As a general rule, there is a trade-off between the complexity of the equality and generate functions. Here we have a simple equality function and a more complicated generation function. We might also have chosen an induced generation function; then we would have had to say something different about equality:

```
Extend =(x, y) with remainder(x - y, modulus) = 0;
```

The reason we saw no trade-off for mod-2 was that the example was too simple. The case of modulo(*n*) is the typical one.

The generate family is not the only way to produce values for a type. It may be natural in some cases to use specifically named functions to generate values in particular ways. These are easily declared by the following kind of sub-declaration:

```
Generator name is function;

Generator name(arguments)[ -> (results)] body;
```

In both cases, the result must be convertible to the base type.

9.5 Declaring families

There are quite general ways of declaring families, and there are also abbreviations for declaring families that work in simple but commonly used ways. The header for a family declaration has the same form as the header for a function declaration:

```
Family new-family[(arguments)][ -> (results)]
```

Rather than a body, however, a family declaration has one or more sub-declarations, and it is in these sub-declarations that the abbreviations appear. One of the most common kinds of family to write is one to provide access to an attribute of a value. Such families have a single argument, the type of which determines which family member to call, and otherwise are unconstrained. To define such a family, all you need is a one member sub-declaration:

```
Access
```

The *(arguments)* may be omitted when there is an **Access** sub-declaration. In access families the member provided in an extension will be given an argument of base type. For an example of such a family, we again turn to mod-2 and modulo(*n*). We might have chosen not to extend generate so that integers could be produced from these types, but instead declared:

```
Family integer-rep
  Access;
```

Then, the type declarations would have read:

```
Type mod-2
  (previous sub-declarations, except extending generate for integer, $)
```

```
    Extend integer-rep(b) with
      If b then 1 else 0;

  Type-constructor modulo(modulus:integer)
    (previous sub-declarations, except extending generate for integer, $)
    Extend integer-rep with identity;
```

Access functions have a nearly minimal axiomatization:

ACC0 Any access function is pure and well-defined.

If the equality function for a type is induced, and there is no conversion between the types for which it is extended, access family members can be written with little thought. If the equality function is non-trivial, then care must be taken to ensure that an access function is well-defined, even on a single type. In the above examples, equality is induced, but we will introduce conversion in the next chapter, and we should examine at that point whether `integer-rep` will still be well-defined.

A more general form of family declaration is used to set up "standard" families. In this case the header of the declaration will have **?**-tagged identifiers that implicitly describe the parameterization of the family. For example, a famly might be introduced by:

```
  Family family(v1:?t, v2:?t) -> (boolean)
    Standard;
```

Since `t` is the only identifier tagged by **?**, this is a one parameter family. In a multiple parameter family the parameters may optionally be listed after the sub-declaration keyword:

```
  Family family( ... ?t1 ... ?t2 ... )
    Standard t2, t1;
```

The order of parameters is used to determine which family member to call, where preference is given to the parameter earlier in the list. If the parameter list is omitted in the multi-parameter case, the order used is that of the first appearance of the identifier in the header.

The sub-declaration rule for standard families is obtained by looking at the appearances of the parameters in the header. If any of these occur as arguments to type constructors, there is no sub-declaration rule for the corresponding parameter of the family. But if all occurrences of a parameter *?t* appear as the type of the argument, as in the equality declaration above, or as a result of the family member, the sub- declaration rule for that parameter position is that arguments are changed from the declared type to the base type before calling the member supplied in an **Extend** declaration, and the result is changed from the base type to the declared type. The sub-declaration rule for equality given in section 9.2 would thus be a special case of the sub-declaration rule for standard families, and the declaration of equality as a standard family.

The two kinds of declarations for families in this section suffice for most familes, but they are special cases of a more general family declaration that is described in D2.4. Chapter 2 of the Definition details exactly how the family declarations and family extensions connect up.

9.6 Summary

This chapter has introduced the notion of function family and has discussed two of the most fundamental function families, equality and generation, as well as the useful notion of an access function. You will see many other function families in this manual, and each one is accompanied by a set of rules that governs the behavior of the family.

This chapter has also introduced the notion of well-definedness, a property we will refer to often. This property is important because it provides a technical criterion for determining whether

you have provided a true encapsulation of some data type. If the functions on your data type are well-defined, then any program using them can also use other representations of the objects, and other algorithms to supply the operations. If the functions you supply are not well-defined, then you have just defined some functions and hung them on the dispatching machinery that function families provide. Your program may work, but it will be harder to understand and modify.

Chapter 10

Conversion

10.1 Laws

The generic conversion function for E-L is `convert`. Conversion should not be confused with generation. While generation always requires you to write something explicitly, conversion is done implicitly—and you will seldom use `convert` explicitly.

The first argument to `convert` is any value at all; the second is the type to which the first value is to be converted. At various points, we have used the terms "succeed" and "fail" for `convert`. These are given meaning via the third argument. This argument is not evaluated at the time of calling `convert`, instead, `convert` chooses when to evaluate it. To say that `convert` *succeeds* means that it never evaluates its third argument. In this case, `convert` yields the converted value. To say that `convert` *fails* means that it does evaluate its third argument. When evaluated, the argument never returns. Typically, it exits or causes a fault:

```
w <- convert(v, t, Exit-function ... with ...)
```

In other words: try to convert v to the type t; if this succeds, store the result in w, if it fails, exit the function.

Built-in `convert` obeys several rules, of which the first are:

CV0 The function `convert` is pure and well-defined.

CV-TP The type of `convert(`v`, `t`, ...)` is t, if convert succeeds.

In other words, conversion gives you the type you expect. Further, converting a value to its own type is a no-op:

CV1 If the type of v is t, then `convert(`v`, `t`, ...)` $= v$, and such an application always succeeds.

The next rule for conversion says that the result of conversion can always be recovered by conversion in the other direction.

CV2 Let v_1 have type t_1 and v_2 have type t_2.

If `convert(`v_1`, `t_2`, ...)` $= v_2$ then `convert(`v_2`, `t_1`, ...)` $= v_1$.

(Implicitly, if the first `convert` succeeds, so does the second.)

An important consequence of this rule is that if the second argument of `convert` is fixed, distinct values in the first argument are mapped, if at all, to distinct values. For example, while an `integer` always converts to a `real`, the only `real`s that convert to `integer`s are those that are an exact integer (and they clearly map to distinct integers). Thus the rules in section 3.1 about conversion among `integer`, `real`, `complex`, etc., are special cases of a general rule.

The final rule for `convert` says that a path of conversion implies, and equals, direct conversion.

CV3 Let v and w be values, t_1 and t_2 be types.

If w = `convert(convert(`v`, `t_1`, ...), `t_2`, ...)`
then w = `convert(`v`, `t_2`, ...)`

(But not necessarily conversely.)

These may seem like very strong rules for `convert`, and that is the intent. Because conversion is done implicitly, it is something that must be very tightly constrained, or you cannot tell what your program does by looking at it. Among other reasons, these rules are chosen because they allow equality to perform reasonable implicit conversions, yet still obey the traditional rules (see the next section). They also allow `convert` to play an important role in function families.

10.2 Conversion and equality

Conversion supplies the criterion for cross-type equality:

EQ-CV Let v_1 and v_2 be any two values, and let t_1 be the type of v_1. Then v_1 = v_2 if and only if v_1 = `convert(`v_2`, `t_1`, ...)`.

This rule holds of built-in equality and conversion. Further, if conversion extensions follow the rules of this and the previous section, then the laws of equality will continue to hold. This rule has such stature that we might have made it the fundamental rule for `convert`. Then reflexivity (EQ1, section 9.2) requires CV1, symmetry (EQ2) requires CV2, and transitivity (EQ3) requires CV3.

10.3 Extensions

The function `convert` is essentially a family of functions, whose members are denoted $cvt[t_1, t_2]$. When you introduce new types, you have the ability to supply conversion for them.

```
Extend convert for t1, t2 with cvt[t1, t2];
```

Each $cvt[t_1, t_2]$ takes two arguments, corresponding to the first and third arguments of `convert`. When extending conversion as a sub-declaration of a type declaration, use **$** (see section 9.1).

```
Extend convert for $, t2 with cvt-out-of;
    or
Extend convert for t1, $ with cvt-into;
```

In the first these cases, the first argument of *cvt-out-of* is given an argument of base type, and must produce a t_2. In the second of these cases, *cvt-into* has a first argument of type t_1, and must produce a result of the base type. The official family member $cvt[t_1, t_2]$ includes the transactions that carry the base type to or from the new type, and thus include slightly more than *cvt-out-of* and *cvt-into*. We now give a simple example not using **$**.

```
Extend convert(x, fail) for mod-2, modulo(2) with
  modulo(2)(integer-rep(x));
```

In other words, generate a `modulo(2)` from the `integer-rep` of the `mod-2`. Similarly:

```
Extend convert(x, fail) for modulo(2), mod-2 with
  mod-2(integer-rep(x));
```

Recall that the use of `modulo(2)` as a function means to call its generation function; similarly for `mod-2`. Observe that these conversions leave `integer-rep` well-defined.

We now discuss the relationship between `convert` and its family members—given several extensions to `convert`, what happens when `convert` is applied to arguments v and t? The fundamental rule is:

CV-EXT Given a value v and a type t. Then w = `convert(`v`,` t`, ...)` if and only if there exist $v_0, \ldots, v_n$ and t_{l1}, t_{l2}, for $l = 1, \ldots, n$ such that

$v = v_0$,
t_{l1} is the type of v_{l-1},
t_{l2} is the type of v_l,
v_l = $cvt[t_{l-1}, t_l]$`(`v_{l-1}`, ...)` for $l = 1, \ldots, n$,
t is the type of w, and
$v_n = w$

In order for `convert` to work as advertised, the family members must obey certain rules, and `convert` will follow certain rules in applying family members. The most obvious are these:

CV0' Each $cvt[t_1, t_2]$ must be pure and well-defined.

CV-TP' When applied by `convert`, the first argument of $cvt[t_1, t_2]$ will have type t_1. The type of the result must be t_2, assuming the call succeeds.

There is no need to carry over CV1 for family members, because `convert` itself guarantees the rule (this is the $n = 0$ case in CV-EXT1). Instead,

CV1' In extending `convert`, the type parameters t_1 and t_2 must be distinct.

CV2' Let $cvt[t_1, t_2]$ be a family member, and let v_1 have type t_1.

If $cvt[t_1, t_2]$`(`v_1`, ...)` = v_2 then `convert(`v_2`,` t_1`, ...)` = v_1.

(And `convert` always succeeds in this situation.)

In piecing together several family members to perform a conversion, there is the issue that several conversion paths are possible, and it is necessary that they all arrive at the same answer. The analogue to CV3 guarantees this, but it requires something stronger, namely that $cvt[t_1, t_2]$ yield *all* conversions from t_1 to t_2, not just some of them.

CV3' Let $cvt[t_1, t_2]$ be a family member, and let v have type t_1. Suppose that `convert(`v`,` t_2`, ...)` succeeds, yielding w. Then $cvt[t_1, t_2]$`(`v`, ...)` succeeds, and yields w.

Suppose there are types t_1, t_2, and t_3, with conversions from t_1 to t_2, t_2 to t_3, and t_1 to t_3.

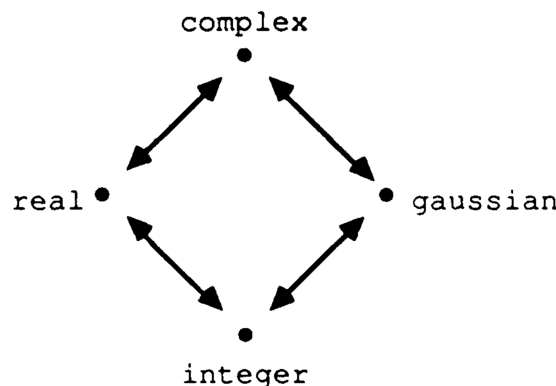
CV3' requires that conversion directly from t_1 to t_3 include all conversions that can go through t_2, but allows the possibility that conversion directly from t_1 to t_3 may include conversions that cannot go throught t_2. But it is often the case that conversion from t_1 to t_3 is *exactly* the same as conversion from t_1 to t_2 followed by conversion from t_2 to t_3. It is not only not necessary to define conversion directly from t_1 to t_3 when all you want is composition, as a matter of style, it is better not to. The only conceivable reason to do so is that there is some efficiency to be gained. But E-L has ample tools for manipulating programs, and it is likely that much of the efficiency to be gained can be done so by applying standard optimization techniques to the composition of the two functions. Subtle optimizations missed by standard techniques can be obtained by more closely specifying the derivation process. This is a preferred way of obtaining the efficiency, because it is less likely to introduce errors than programming the optimized composition directly, and there will be less work to do if conversions from t_1 to t_2 or t_2 to t_3 are changed.

There is one final rule for extensions to `convert`.

> An extension to `convert` done in conjunction with a new type may not affect conversion relationships of previously defined types.

You may *not*, for example, change conversion relationships of built-in types. The motivation for this restriction is that it greatly aids analysis; for a more detailed discussion, see the *E-L Design Rationale*.

To provide an example of an interesting extension to `convert`, we look at several types for which conversion exists—`integer`, `real`, and `complex`—and consider adding the type `gaussian`, which will be like `complex`, but with `integer` components. The conversion functions defined for these types are depicted in the diagram:



The conversion function for the pair `integer`, `complex` is not defined directly. If such a conversion is required, `convert` does a two step conversion, from `integer` to `real` to `complex`. Suppose you add conversions between `integer` and `gaussian` and `gaussian` and `complex`, and back, for each pair. You will find that for the natural conversions among these objects, it does not matter if `convert` goes from `integer` to `complex` via `real`, as it did before the extension, or via `gaussian`, which is now allowed. Further, both CV3' and CV4' are satisfied.

This section has discussed extensions to `convert` only for specific types, such as `mod-2` or `modulo(2)`, not for sets of types, as you might wish to do when defining certain type constructors. It is possible to have extensions with the parameters involving ?-tagged identifiers, for example, `modulo(?i)`, and for the identifiers to be used in the family member. The rules for such extensions are quite similar to those given here for types, but involve sufficient technicalities to be postponed to D3.

10.4 Addition

The symbol + is ubiquitous in mathematics. By nearly universal convention, and in E-L, + obeys the following rules:

PLS1 For any values v and w, $v + w = w + v$.

PLS2 For any values v, w and x, $(v + w) + x = w + (v + x)$.

To the classical rules for +, we will add

PLS0 The function + is pure and well-defined.

Like =, the + family is parameterized by a single type, so you can extend it with:

```
Extend + for t with plst;
```

The family member pls_t is given two arguments of type t, and must yield a result of type t. A family member must obey PLS0-PLS2. As a sub-declaration, you may say merely

```
Extend + with plst;
```

In this case, the family member is given arguments whose type is the base type of t, and must yield a value with same type. For example:

```
Type mod-2
  Based-on boolean;
  Induce =;
  Extend + with exclusive-or;
```

In a sub-declaration, pls_t must obey PLS0-PLS2 only up to $=_t$.

A discussion of + is not the main point of this section. Rather, + illustrates a technique in forming families of functions. Just as equality can take arguments of differing type and produce a reasonable result, it makes sense for + to take arguments of different type. For example, given a `real` r and a `gaussian` g, both can be converted to `complex`, where + is defined. This is what is done in order to yield a result for + (assuming merely that you have provided conversion from `gaussian` to `complex`. In general, the generic operator + tries to convert its operands to a common type t, and then uses pls_t. Even though you need to worry about + for only one type at a time, the + family does not require identical types for both operands.

In order for this strategy to provide an operator that obeys PLS0-PLS2, there must be a proper relationship between pls_t for different types t. It is captured in the following definition.

We say that functions f and g are *compatible* if and only if

$$v_i = w_i \text{ for } i = 1, \ldots, k \text{ implies } f(v_1, \ldots, v_k) = g(w_1, \ldots, w_k)$$

The following rule concerns the interaction of + and `convert`:

PLS-CV For types t_1 and t_2 where pls_{t1} and pls_{t2} are defined, they are compatible.

For example, let pls_i and pls_c be the built-in + on the types `integer` and `complex`. It is clear

that they are compatible in the above sense. Let $pls_{\texttt{mod-2}}$ be the extension to + defined above. Then $pls_{\texttt{i}}$ and $pls_{\texttt{mod-2}}$ are also compatible, for a nearly vacuous reason: because we have not defined conversion between `mod-2` and `integer`, the hypothesis part of the compatibility condition is always false, so the condition is always true. This is one more way in which E-L constrains the use of `convert`. You cannot just decide that it would be nice if the values of two types, or even subsets of the values of two types, had conversion between them. You must think about function families defined on the two types, and whether conversion will interfere with their compatibility.

If + is given two arguments that cannot be converted to a common type *t* for which pls_t is defined, there is a Conversion Fault. We state that relationship between + and its family members:

PLS-EXT1 For all *t* where pls_t exists, + agrees with pls_t on values of type *t*.

PLS-EXT2 For all *v* and *w*, $v + w$ produces a value (as opposed to faulting) if and only if *v* and *w* can be converted to a type *t* where pls_t exists.

10.5 Zero

Just as + is used in mathematics and in E-L to name a function that is associative and commutative, the symbol 0 is used to denote the identity for +.

PLS-ZERO For any value *v*, $v + 0 = v$.

The same notation convenience is available in E-L, for a small price. If you wish to use 0 to denote the identiy for pls_t, you must make **Extend** the `convert` family so that the `integer` 0 and the "zero" for your type *t* inter-convert. The "zero"s of two types may well be the only members of the types that successfully convert.

The point of of this remark is to allow you to write very generic functions in terms of + and `0`, and to be able to use these functions for types where you extend + and `zero`, and in situations where + requires conversion. Not only can you write generically with `0`, programs are analyzed and transformed generically. The rule $v + 0 = v$ has the same status as other rules for function families.

Chapter 11

Flow of Control Revisited

11.1 Writing iterators

An important aspect of E-L is that it provides the means for you to write your own iterators. Thus far we have seen one built-in iterator, *i* to *k*, typically used in a **For in do** construct like

```
For i in 1 to n do A[i] <- i;
```

An iteration such as this can be thought of as two cooperating activities—a *producer* and a *consumer*. In the above, the producer, `1 to n`, produces the consecutive integer values `1, ..., n`. When a new value is produced, it is delivered to the consumer—the body of the iteration—with the value `i` playing the role of a communications channel in the sense that the delivery of a new value to the consumer is accomplished by `i` being bound to that value when the body is executed.

Certain types have natural iterations associated with them. For example, in section 8.8 we saw the type constructor, `list`. If `t` is a type and `L` a `list(t)`, you can define what is meant by

```
For e : t in L do
   (some expression involving e);
```

Here the producer must consider the successive elements in the list and deliver the "goods" associated with each to the consumer. The consumption is accomplished by evaluating (some expression involving e) with e bound to the "goods" produced. An iteration such as the above is, in actuality, mapped into a call on the family named `iterate`. `iterate` takes two arguments corresponding respectively to the producer and the consumer.

The extension to `iterate` in the context of defining the type constructor named `list` could be written:

```
Type-constructor list(elm-type:type)
  Based-on (see section 8.8);
  Extend iterate(variable L,
                deliver : function(e : elm-type) -> ()) with
    While L # nil do
      deliver(L.goods);
      L <- L.next;
```

In an iteration of the form

```
For i in j to k do body;t
```

the situation is somewhat different. In the list example, the first argument was the `list` over which we were to iterate. Here the first argument to `iterate` is not a data object but a "producer", `j to k`.

There is a type producing function named `iteration` and a special syntax that makes it easy to write iterators like `to`. The type of `to` is `iteration(integer)` and, though `to` happens to be built-in, it could be defined as follows:

```
Iterator(integer) to(first:integer, last:integer)
```

```
Variable i is first;
While i le last do
  deliver(i); i <- i + 1;
```

Here the phrase **Iterator**(integer) indicates that to has the type iteration(integer), that is, to is an iterator delivering integers.

The interpretation of a call on the iterate family when the first argument has the type iterator(t) is to call the first argument with deliver bound to its second argument. Thus in this example, each call, deliver(i), executes the body of the iteration with the iteration parameter bound to the value of i. The scope of deliver is local to the **Iterator** declaration.

In general, the type producing function iteration takes parameters that are types $t_1, ..., t_k$, and describes values that are iterations delivering those types.

For more on iteration, see D5.2.

11.2 "Input" from iterators

Let us consider the problem of writing the most general possible merge algorithm, where by "merge", we mean producing an ordered series from two ordered series. The first question to answer is what, precisely, is meant by "ordered series"? In view of the general mechanism for iteration, it would be sensible if "ordered series" meant simply an iteration with the additional property that the elements produced are in increasing order, for some ordering. For example, suppose that the lists of the previous section were constructed with "goods" in increasing order. If $list_1$ and $list_2$ were two such lists, we would like to be able to write:

```
For x in ordered-merge(list1, list2) do ...
```

From our discussion of iteration, we at least know the form that the declaration for `ordered-merge` should take.

```
Iterator ordered-merge(values1, values2)
   (body for ordered-merge);
```

Because we want this function to take as arguments anything that one can iterate over, and since it is possible to iterate over any type that has an extension of `iterate`, the function header quite deliberately has no types.

The difficulty in writing the body of this iterator is that it is necessary to be able to get an arbitrary number of values from one iteration, while suspending the other. This requires a new notion—we must turn an iteration into a value that can be stepped to obtain the next value in the iteration; these are called "input processes". If the type obtained by stepping is always fixed types $t_1, \ldots, t_k$, the type of the input process is `input-process(`$t_1, \ldots, t_k$`)`. Such a value may be generated from a value whose type is `iteration(`$t_1, \ldots, t_k$`)`. For $k = 1$, this includes any type whose extension of `iterate` produces value of type t_1. If the type obtained by stepping is not necessarily constant, the type for the input process is `any-input-process`. Values of this type may of course be generated from values with type `any-iteration`, and for $k = 1$, from *any* type with an extension of `iterate`. Since `ordered-merge` is being written as generically as possible, it uses the generic input process.

```
(body for ordered-merge)
   Let process1 be any-input-process(values1);
   Let process2 be any-input-process(values2);
   (continuation of body)
```

To get values out of these processes, use the built-in `input`. Given a first argument that is an input process, the result is the next value that would be delivered by the iteration used to generate the process. This immediately raises the issue, what happens at the end of iteration? The second argument of `input` is evaluated only at the end of iteration, and like the third argument of `convert`, it is an error for the evaluation to return to `input`. Instead, there must be an exit to some other point.

```
(continuation of body)
   Let v1 be
      input(process1, (what to at the end of process1));
   (second continuation of body)
```

The second argument of `input` here is evaluated only when `values1` has no values at all. In that case, the sequence of values delivered by `ordered-merge` is exactly the sequence of values

delivered by `process2`. We could, of course, write a simple loop to deliver these values, but this situation comes up several times, and we encapsulate it in the function `in`:

```
(what to do at the end of process1)
  in(process2, deliver); Exit-iterator
```

The reason that `deliver` can be passed along as an argument will become evident in D5.2. The next statement is similar except it must also `deliver(v1)` if `values2` is empty.

```
(second continuation of body)
  Let v2 be
    input(
      process2,
      (deliver(v1); in(process1, deliver); Exit-iterator));
  (the main loop for ordered-merge)
```

The rest is relatively straight-forward.

```
(the main loop for ordered-merge)
  Repeat
    If v1 lt v2
      deliver(v1);
      v1 <-
        input(
          process1,
          (deliver(v2); in(process2, deliver); Exit-iterator))
    else
      deliver(v2);
      v2 <-
        input(
          process2,
          (deliver(v1); in(process1, deliver); Exit-iterator))
```

This concludes the body of `ordered-merge`.

Other applications of input processes are ways of providing parallel or interleaved iteration:

```
For x1, x2 in parallel(i1, i2) do ...

For x in interleave(i1, i2) do ...
```

You may supply implementations for these iterators, and decide what to do if the argument iterations are of different length.

11.3 "Output" to functions of iterations

Just as we considered merging as the motivating example of the last section, we will consider summing as the example of this section. We can write the very generic function:

```
Function sum(values)
  Variable s is 0;
  For v in values do s <- s + v;
  s
```

The argument `values` can be any value for which iteration is defined, provided that it delivers

values for which + is defined (for the generic use of 0, see section 10.5). For example, the expression `sum(list)` yields the sum of all the elements of a *list*.

The point of this section is not the compact notation that is possible for summation or other similar functions, but rather, that functions of an iteration can be controlled from the "outside". To illustrate this, we will write a function `sum-sumsq` whose argument is an iteration and whose results are the sum of the values of the iteration and the sum of their squares. The type constructor `output-process` and the type `any-output-process` have an obvious analogy with the input versions of the previous section.

```
Function sum-sumsq(values)
  Let sum-process be any-output-process(sum);
  Let sumsq-process be any-output-process(sum);
  (do the summing);
  (yield the results);
```

Doing the summing is what you would expect.

```
(do the summing)
  For v in values do
    output(v, sum-process); output(v^2, sumsq-process)
```

The idea is that each copy of `sum` accumulates its total. The remaining problem is how to get the answers. This is related to informing the argument of `sum` that the iteration is at an end. The built-in `stop-output` is used for this purpose. It has one argument, which must be an output process. Its result is the result of the function from which the output process was generated. Thus:

```
(yield the results)
  stop-output(sum-process), stop-output(sumsq-process)
```

It may seem silly to encapsulate an idea as trivial as summing. After all, `sum-sumsq` could be written as easily by initializing variables `s` and `ssq` to 0, and using ordinary statements like `ssq <- ssq + v^2`. Part of the purpose has been merely to provide an easily understood example of the use of output processes. But it might also be noted that even a trivial encapsulation has its practical aspects. Suppose that it is necessary to compute the sum and sum of squares of tens of thousands of floating point numbers, a circumstance that can arise in a large regression problem, among others. In summing this many floating point numbers, round-off error becomes an issue, and there are sophisticated summation algorithms that try to minimize round-off error. If the program isolates the generation of the numbers to be summed (say by iterating over an array) from the technique of summing (by writing summation routines whose argument is an iteration), it is much easier to experiment with various summation techniques, and their implementations are not tied to particular data structures, so are more re-usable.

Both input and output processes, and the operators for them described in this and the previous section, are special cases of a general inter-process communication mechanism, which is discussed in D5.3.

11.4 Premature exit revisited

In section 5.4, the **`Exit-function`** construct was introduced, and you have seen analogues like **`Exit-iterator`**. These are the usual but not universal means of doing a premature exit. It is sometimes helpful to exit not from the innermost function seen from a certain point, but from a function more distant. This can by naming the function to be exited:

```
Exit name [with values]
```

The *name* here must be the name of a declared **Function**, **Iterator**, or similar declared object, and *values* are returned as the results. Where there are no results, the **with** clause may be omitted.

To exit a construct within a function, you set up a "catch":

```
Catch(name[(t1, ... , tk)]) (body)
```

The *name* introduced hides other names in the usual way. Its scope is the body. The evaluation of **Exit** *name* **with** *values* within the body yields the *values* as the result of the catch construct. If the types are present, the results are converted to the specified types.

In the exit constructs we have defined thus far, it is possible to leave only lexical scopes. It can also be useful to exit to a point that can't be seen from the exit. This is possible because "catches" are values that can be passed as arguments, so the value can appear where the actual catch cannot be seen. Such values have types obtained via the type constructor catch, whose single argument is a type. The following describes a value to which exits may occur with values of types $t_1, \ldots, t_k$.

catch($t_1, \ldots, t_k$)

The union of all such types is any-catch. A function that exits to a catch that it cannot see will have the following general structure:

```
Function f(..., lost-data-catch:catch(t1, ... , tk), ...)
  ...
     ...Exit last-data-catch with values
  ...
```

Catch values arise in only a few ways. The name of a **Catch** expression, within its scope, is a catch value, a constant, if you will. The **Exit with** syntax allows an abuse of notation by letting you put the name of a function where a catch value really belongs, but in other contexts, you must use catch-point(*name*) to obtain the catch value associated with the function named by *name*. This expression must occur within the scope of *name*, and produces the catch value for the current invocation of the function. Since catches are values, they may be passed about like other values. For example, a catch-valued argument may be passed along as an argument to another function taking a catch-value.

11.5 Continuing iterations

If you do not read this section, you will find yourself writing loops where the body is a catch construct:

```
For ... in ... do
  Catch(c)
    (body with Exit c)
```

The header of the loop might also be a **While do** or a **Repeat**. It is simpler and more readable to say:

```
For ... in ... do
  (body with Continue)
```

The same construct may be used in **While do** and **Repeat** loops. The **Continue**

statement needs no arguments to continue the inner-most loop. To continue outer loops, you must name them. This is done by replacing the **do** with <<*name*>>, or simply adding that construct to a **Repeat**.

```
For ... in ... <<for-name>>
  While ... <<while-name>>
    Repeat <<repeat-name>>
      ... body containing statements like:
          ...Continue for-name
          ...Continue while-name
          ...Continue repeat-name   (equivalently, Continue)
```

Chapter 12

Assignment

The purpose of this chapter is to explain how assignment "really works". Part of the reason for doing this is to clarify the kind of conversion that occurs when passing **shared** arguments and results. But the real payoff is that this knowledge will enable you to extend some of the most basic function families in E-L, providing a mechanism for certain types of encapsulation that is otherwise unobtainable.

12.1 Places

To return to the example of section 6.3, consider

```
Function f(shared x:real) -> (shared real)
```

As we said, the type that is involved in the parameter conversion is not `real`, rather it is:

```
place(real)
```

The built-in type constructor `place` takes a single type parameter. Intuitively, `place(`*t*`)` describes places where values of type *t* can be stored. The type `any-place` includes all such places.

12.2 Operations on places

The real message of this chapter is that places are values in E-L, just as surely as types or functions. They may be passed as arguments and results just like other values, and they may be incorporated into data structures. So far, you have learned how to deal in place values only in connection with the **shared** bindclass. But there is great power in being able to deal with them as bona fide values. First, it is necessary to describe the basic operations on places.

On several occasions, we have said, "assignment works as you expect". This section will give the precise rules, in terms of two functions defined on places. The first of these functions is almost, but not quite, <-. The symbol <- is syntactically marked as having a **shared** left argument. The symbol `assign` is the same function, but has no special syntactic property. It is more convenient to axiomatize in terms of `assign`. The first rule is:

ASGN0 The function `assign` is well-defined and memory-less.

In interpreting the term "well-defined" here, remember that the first argument of **assign** is a place, and the equality being talked about is that of places. To emphasize this point, a consequence of ASGN0 is:

> If p_1 and p_2 are places for which $p_1 = p_2$, then `assign(`p_1`, ...)` may be replaced by `assign(`p_2`, ...)` without affecting the computation.

Note that `assign` is not pure, only memory-less (depends only on the values of its arguments). The whole purpose of assignment, after all, is side-effect.

The second function is not related to any name that you have yet seen. Recall the rule in section 6.4 about a **shared** variable or function result in an un**shared** position: an implicit contents operation is supplied. To describe how places and assignment interrelate, it is necessary to make

this function explicit. It is called `contents`. Its single argument must be a place.

CNT0 The function `contents` is well-defined and side-effect free.

In other words, taking the `contents` of equal places depends only on the place, so long as the state does not change, and `contents` never changes the state. But `contents` is not memory-less; its purpose is to produce a value from the current state of a place.

There are two rules relating `assign` and `contents`. One says that you can change what you want to change, and the second says that you will not change what you don't want to change.

ASGN-CNT1 For any place *p* and any value *v*, immediately after `assign(`*p*`,` *v*`)`, the predicate `contents(`*p*`) =` *v* is `true`.

ASGN-CNT2 For places p_1 and p_2, with p_1 # p_2, and any value *v*, if `contents(`p_1`)` = *v* immediately before `assign(`p_2`, ...)`, then `contents(`p_1`) =` *v* immediately afterward.

When we said earlier that "assignment does what you expect", these rules are what we assumed that you expected.

Using the `assign` and `contents` functions, we can finally give precise meanings to the terms "memory-less" and "side-effect free". We said that intuitively, a memory-less function was one that depended only upon its arguments, and not on the state. This is made more formal by viewing "memory-less" as describing a subset of functions, and then postulating the following relationship between `assign` and functions in this subset.

ASGN-ML Let *f* be a memory-less function. Then

```
(assign(...); f(...))
```

is equivalent to

```
(Let v1 ... vk be f(...); assign(...); v1, ..., vk)
```

where the v*i* are not free in the original expression.

Just as "memory-less" signifies a relationship between a function and `assign`, the term "side-effect free" signifies a relationship between a function and `contents`.

CNT-SEF Let *f* be a side-effect free function. Then

```
(f(...); contents(...))
```

is equivalent to

```
(Let v be contents(...); f(...); v)
```

where v is not a free variable of the original expression.

12.3 Conversion of places

Now you find out about conversion of places. Reconsider the function f defined in section 1. Suppose that you pass in integer variable i to f, as in f(i), so that you are implicitly requesting a conversion from place(integer) to place(real). What to do?

E-L tries not to jump to conclusions about this. After all, maybe you know what you are doing. If you happen to know that f only *looks* at its shared parameter, and doesn't try to set it, there should be no difficulty. As long as (the by now explicit) contents function can look at the converted place, obtain an integer value, and convert it to a real, all the laws for places are satisfied.

Suppose, though, that f *does* assign directly to its formal argument. To see what should happen here, suppose you say i <- *r*, for some real-valued expression *r*. So long as *r* is an exact integer value, the assignment will complete. Under the principle that behavior inside a function should not be that much different than if the function were expanded away, assign looks at the converted place, sees the place(integer) underneath, and tries to assign to it. If that works, fine; if not, maybe you *don't* know what you're doing. There is nothing to do but fault.

A dual situation occurs if you pass a complex-valued variable to f, requesting a conversion from place(complex) to place(real). In this case, assign will never fault, but the possibility exists for contents. And if you pass a boolean-valued variable to f, f cannot do either operation, since boolean and real have no interconversion whatsoever.

While E-L does no fussing about place convertibility at the time of call, it anticipates potential problems with place conversion in other ways. If you manage to run your program before E-L has had time to snoop around, you may indeed first become aware of the problem because of a fault. But static type analysis will detect a potential problem before it arises. If there is a conversion from place(integer) to place(real), and f has assignments to the formal, you will be warned, unless the analyzer can figure out that the real value will always convert to an integer. Dually for place(complex) to place(real). If there is conversion from place(boolean) to place(real), the analyer will look askance at any reference to this formal; if there are no references to the formal, E-L will point *that* out, regardless of conversions.

When the types of two places are completely inter-convertible, then conversion of places will never cause any faults. This is quite convenient when your program has different types, signifying different representations, and you wish to use functions with **shared** parameters.

12.4 Families

Both assign and contents are function families, and of course, you may extend them. Like other function families, your extensions must obey family rules, in this case, those of section 2. To illustrate the use of this capability on a simple example, we will show how to implement the inter-convertibility of places, as if it were not already built-in. You will find other extensions to these families in the next chapter.

From the description of the previous section, it is clear that a converted place must contain the place from which it was converted, and have access to the type that it is known by in its converted state. Consider converting from place(t1) to place(t2). Using this nomenclature, define:

```
Type-constructor converted-place(t2:type)
  Variant-of place(t2);
  Based-on any-place;
  Induce =, generate, assign;
  Extend contents(p) with
    convert(
      contents(p),
      t2,
      fault('Cannot convert contents of converted place'));
  Extend convert for any-place, $ with identity;
```

Note that converted places are equal when their underlying places are equal, and that conversion amounts to clothing the place in a different type. While we do not do so here, it can be shown that all the properties for `assign` and `contents` hold.

Chapter 13

Compound Values Revisited

You have already seen the basic compound values provided by E-L: tuples, arrays and records. Here, the purpose is to understand selection from compound values, with the main purpose of being able to extend selection. As with assignment, in order to extend selection, it is necessary to get underneath the surface syntax of brackets and dots. The payoff is also similar, because the extensions allow some very elegant encapsulations.

13.1 Selection syntax

The selection constructs `cv[...]` and `cv.s` have somewhat the same position as `<-`. They are convenient surface syntax; they, or slight variants, have a long tradition in other languages; and, they are unsuitable vehicles for properly formalizing the operations that they suggest, and thus unsuitable for thinking about extension. There are two functions related to selection having a role analogous to `contents` or `assign`. They rarely appear in surface syntax, but provide the proper conceptual devices. The first of these is `select`, which may be thought of as being brought into existence whenever bracket or dot appear in an unshared position (cf. section 6.4).

what you see (un**shared**) — what you get

```
cv[i1, ... , ik]        select(cv, <i1, ... , ik>)
cv.s                    select(cv, 's')
```

The second function is `pselect`, which has a similar role in a **shared** position.

what you see (**shared**) — what you get

```
cv[i1, ... , ik]        pselect(cv, <i1, ... , ik>)
cv.s                    pselect(cv, 's')
```

The `p` is for place, as you shall see. To give an example, including assignment:

`A[i, j] <- B[k, l]` in the surface syntax becomes

```
assign(pselect(A, <i, j>), select(B, <k, l>))
```

Next, we have to understand the `pselect` and `select` functions.

13.2 Types and laws

The type `any-compound` is an expandable union, where each variant must have a `select` function. This is a very generic type, for there is only one rule for `select` :

SEL0 The function `select` is a well-defined, side-effect free function of two arguments, the first of which is a `any-compound` (the second is intuitively the index, and is arbitrary). The result is arbitrary.

The type `compound-place` is also an expandable union, where each variant must have a `pselect` function, whose purpose is to obtain a place from a compound place.

PSEL0 The function `pselect` is well-defined, pure, and has two arguments, the first of which is a `compound-place`. The result of `pselect` is an `any-place`.

There is one additional requirement on `pselect`, of great interest to analysis machinery:

PSEL1 The function `pselect` is 1-1 on its second argument, that is, for any cp a compound place, and values *v* and *w*.

If *v* # *w* then `pselect(`*cp*`, `*v*`)` # `pselect(`*cp*`, `*w*`)`

Finally, `compound-place` is a variant of `compound`, and there is the following relation between the two functions.

SEL-PSEL For any compound place *cp* and any value *v*,

`select(`*cp*`, `*v*`) = contents(pselect(`*cp*`, `*v*`))`

Selection fits together in the way that you expect.

13.3 Extensions

As you have been warned, `select` and `pselect` are function families. When you want to use the bracket and dot syntax in some new way, you have to think about defining a new variant of `compound` or `compound-place`, and you have to think about implementing `select` or `pselect`. Since the SEL-PSEL rule fixes `select` for `compound-place`, it is preferable to *implement only the* `pselect` *function* for `compound-places`, for reasons analogous to those for not implementing redundant conversions (see section 10.3).

```
Type name
  Variant-of compound;          (or compound-place)
  Based-on (whatever);
  Extend select with ...;       (resp., pselect)
```

You will find that in writing members of the `select` and `pselect` families, it is quite natural to use `select` and `pselect` in the surface syntax. But these are probably the only places.

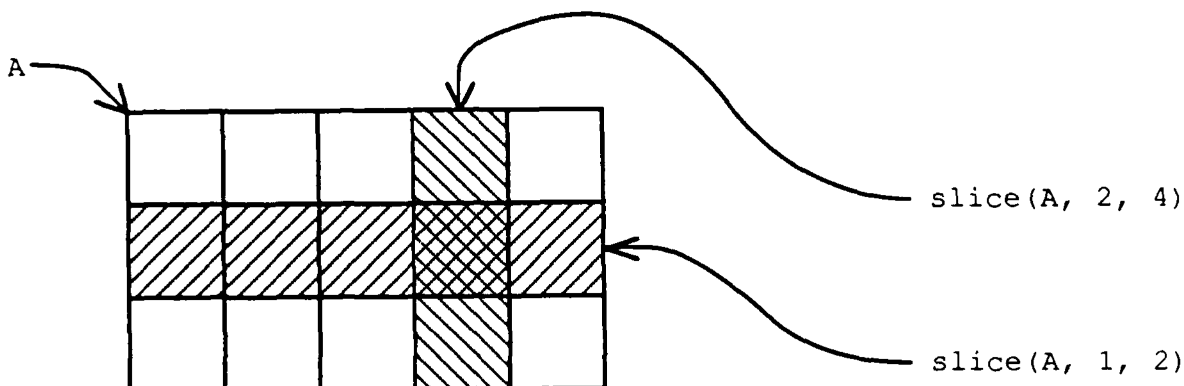
13.4 Slices

The purpose of this section is three-fold. First, it introduces the notion of a *slice*, a useful notion for programming on its own terms. Second, the implementation of the machinery associated with slices provides a good example of selection extension. Third, the entire exercise emphasizes the notion that arrays are values themselves, quite apart from the values stored in their places.

First, slices on their own terms. Suppose you have:

```
Array A(3, 5) initially ...
```

The function `slice` takes three arguments: an array, an "axis" (in this case, 1 or 2) and an index, i.e., along the indicated axis. The following picture illustrates two applications:



The result of `slice` is thus an array itself, whose constituent places are places in the array `A`. It can appear anywhere that an array with its dimensions and element type can appear. For instance, to clear the above column, write:

```
slice(A, 2, 4) <<- 0
```

The depicted row can be cleared similarly.

The slice machinery is built-in to E-L, by the following definitions. The idea is to define a type constructor that is a variant of `array`. The new type constructor `slice-array` is based on a record for holding the array, the axis, and the index.

```
Type-construc )r slice-array(dimensions:ntuple(integer),
                           elm-type:type)
  Based-on
    tuple(A:any-array, axis:integer, index:integer);
  Variant-of array(dimensions, elm-type);
  Induce generate, =;
  (selection for slices);
```

This takes care of the type definitions, and all that remains are `slice` and selection for slices. Both are easy, but require the use of simple tuple-manipulation functions. The first is `contract-tuple(t,i)`, which yields a tuple one shorter than t, differing from it by the deletion of the ith component. This is used in:

```
Function slice(A:array(?dimensions, ?elm-type),
               axis:integer,
               index:integer)
  generate(
    slice-array(contract-tuple(dimensions, axis), elm-type),
    <A, axis, index>);
```

Thus, `slice(A, 2, 4)` has type `slice-array(<3>, integer)`. To implement selection, we need `expand-tuple(t, p, i)`, which returns a tuple one longer than t, the value i occupying the pth position and subsequent components moved over.

```
(selection for slices)
  Extend pselect(sb, selector) -> (place(elm-type))
    pselect(sb.A, expand-tuple(selector, sb.axis, sb.i));
```

Note the `pselect` in surface syntax, and the implicit possible recursive use of `pselect-slice`.

This example shows that the representation for an array in E-L can depart rather drastically from the usual notion of contiguous locations in memory. All that is required for a variant of `array` is that its `pselect` function obey the rules of section 2, where selectors are tuples of integers of the right length, with components in the right range. The array `A` declared at the beginning of this section has a type that is more specific than `array(<3,5>, integer)`. The default type for new arrays indicates to built-in selection that the array is laid out in the usual way. You will not find the specific type constructor named in this manual, because you should program at the level of `array` and slices. If you are worried about the fact that a slice is laid out contiguously and want to exploit that for speed of selection, you can handle that with analysis (which way is this array sliced most often?) and transformation (maybe the indices should be permuted so that slices are contiguous). But those considerations should be separate from the statement of your algorithm.

On an entirely different matter, slice allows a "multi-dimensional" iteration over arrays, each iteration yielding a succession of slices of smaller dimension. The following member of the iterate family is built-in.

```
Extend iterate(A:array(?dimensions, ?t))
  For i in 1 to dimensions[1]
    deliver(slice(A, 1, i));
```

In iterating over a 1-dimensional array, 0-dimensional arrays are produced. These are convertible to places in the obvious way.

13.5 Sparse arrays

In a sparse array, there is some special value, say 0, which takes less space than non-special values. In practical situations, sparse arrays are usually implemented with list structure or by hashing, which are not of interest here. The purpose of this section is to show how array notation and abstractions can be used at one level of a program, and be supported by data structure like lists or hash tables at another. What we will do here is to describe the machinery that connects the array notation with the representing data structure. Here the type of that data structure will be called `table(`t_1`, `t_2`)`, where t_1 is the type of an index, and t_2 is the type of the datum associated with an index. We will later describe operations required on values of this type. It is useful to define not only the type constructor for a sparse array, but also one for its base.

```
Type-constructor sparse-array-base (k:integer, elm-type:type)
  Based-on
    tuple(tbl:(ktuple(k, integer), elm-type),
          default:elm-type);
  (defintions for sparse-array-base);
Type-constructor sparse-array
                  (dimensions:ntuple(integer),
                  elm-type:type)
  Based-on
    sparse-array-base(length(dimensions), elm-type);
  Variant-of array(dimensions, elm-type);
  Induce;
  (generation for sparse arrays);
  (selection for sparse arrays);
```

Because `sparse-array` is a variant of `array`, selection is `pselect`. The interesting part about `pselect` is that it should not do anything to affect the table. Remember that `pselect` returns a place. If `contents` is applied to this place, the table is not disturbed. So what does `pselect` do? It must return something to which both `contents` and `assign` can be applied, and work as required. To see what is going, it is easiest to first consider how sparse array places

must behave.

```
Type-constructor sparse-array-place
                    (k:integer, elm-type:type)
  Based-on
    tuple(s-a:sparse-array-base(k, elm-type),
          index:ktuple(k, integer));
  (contents  for sparse array places);
  (assignment for sparse array places);
```

In other words, a sparse array place is merely a tuple holding the sparse array and the index. This allows an obvious implementation:

```
(contents for sparse array places)
  Extend contents(s-a-p) -> (place(elm-type)) with
    Catch default
      lookup(s-a-p.s-a.table, s-a-p.index,
             Exit default with s-a-p.s-a.default-value);
```

Evidently, `lookup` is a function on tables. The third argument to `lookup` is evaluated only when the first argument is not found in the table. In the above function, the result is the result of `lookup` if the index is found in the table; otherwise, the result is the default value associated with the sparse array.

Two other functions on tables are `delete` and `enter`, used in the extension of `assign`, which after all, must change the table.

```
(assignment for sparse array places)
  Extend assign(s-a-p, value) with
    If value = s-a-p.s-a.default-value
      delete(sap.a.t, sap.index)
    else
      enter(sap.a.t, sap.index);
```

Now that we have sparse array places, we can define `pselect` for sparse arrays: just generate a sparse array place.

```
(selection for sparse arrays)
  Extend pselect(sa, index) with
    check-index(index, sa.dimensions)
    generate(sparse-array-place, <sa.index>);
```

Arrays, not tables, take on the burden of index checking. You may supply `check-index`.

Chapter 14

Larger Programs

14.1 Packages

A package is the natural unit of encapsulation in E-L. As in other languages, a package in E-L consists of a number of declarations, some of which are for "private" use by the package, and some of which are for "public" use. The declared identifiers may be constants, including types and functions, or variables. An unusual aspect of E-L packages is that the may be parameterized, for example, by a type or the size of a table.

The syntax for declaring a package is much like that of other declarations, a header and body:

```
Package [pkg][(formals)]
  [initialization;]
  [Export {name [as public-name]|,} |
   Export-all [except private-names] ];
  [clean-up];
```

The *initialization* part of the body contains both public and private declarations (there is no distinction between the two here), as well as executable code that is be performed when the package is imported. The **Export** statement lists the public names of the package, optionally indicating a different name for public use; **Export-all** means to export all of the identifiers defined in the *initialization*, with the exception of any explicitly listed names. The *clean-up* part of the package is executed when the scope of an import statement is finished executing. This part is often empty, but may include actions such as closing files.

To import a package and not change any of its names, you say:

```
Import pkg[(actuals)];
```

The *actuals* are required only when the package has parameters. The names in the **Export** list for the package are introduced into the scope of the **Import** declaration. If you want to change names as you import, you say:

```
Import pkg[(actuals)] changing new-name1 to new-name1, ...
```

A package may occur in any number of **Import** declarations.

When a package is declared with no package name, there is an implicit **Import** immediately after the package definition. It is usually in connection with nameless packages that the **as** option is used in an **Export** declaration.

An important aspect of the semantics of packages is that they may be declared anywhere that a declaration can occur, and that the functions declared within a package have the lexical scope of the package declaration, not that of the **Import** declaration. For example, consider:

```
Constant x is ...;
Package p
  Function f(...)
    (a body that references x);
  ...
  Export f;
Let x be ...;
Import p;
f(...)
```

When `f` is applied, an `x` in its body refers to the constant `x`, not the variable `x`. For a complete description of the semantics of packages, see D6.3.

You saw in section 7.3 that a document can define a main program. That section also described how various E-L fragments are pulled together to become a program, either explicitly via the use of breakouts, or implicitly in the attempt to provide declarations for undeclared identifiers. This section has introduced packages, and has emphasized that packages may be declared anywhere in the program, and that the functions they contain have the lexical scope of the package declaration. Many packages, however, use only built-in functions, and thus do not need to be declared in any particular lexical scope. In a sense, they can be declared at the lexical scope that defines built-ins, i.e., the same lexical scope at which a main program is defined. Such packages may be imported by any program, and so are called *universal*. When you write a package that is intended to be universal, use a breakout header with the key ***universal***. Like ***main***, this key is not referenced by a breakout—such a reference would indicate a point at which to declare it. (Of course, you may **`Import`** a universal package anywhere, but that is a reference to the package, not to the key.)

14.2 Multi-document programs

As programs become larger, it becomes unwieldy to have everything about them in a single document. And by their very nature, universal packages cannot appear in every document that **`Import`**s them. So there are simple ways to make multi-document programs.

We first consider cross-document breakouts. To create a reference to a breakout that is defined in another document, you do nothing different than in creating a reference to any other breakout. However, rather than defining the breakout, you include the following underlying command somewhere in your document (typically, near the end):

```
\cite-breakout{key}{document}
```

This will cause the named document to appear in your bibliography, and the reference itself will be tagged with a bibliographic citation, in whatever style your document uses.

In a document that defines a breakout used in another document, you must make the breakout available outside the document. Like the reference to a cross-document breakout, there is nothing special about the point of definition of the breakout. However, somewhere in your document (typically, near the beginning) you include the following underlying command:

```
\citable-breakout{key}
```

The list of citable breakouts appears in the table of contents part of your document, along with lists of tables and figures.

A second form of cross-document reference concerns those fragments without a breakout header. These are automatically incorporated into a program (see section 7.3). You may indicate that you want undefined names from one document to be resolved, if possible, with names from another document. By analogy with the cross availability of breakouts, use the following underlying commands in the respective documents:

```
\cite-loose-fragments{document}

\citable-loose-fragments
```

There will be entries made automatically in the bibliography and contents parts of the two documents.

The final means of cross-document reference concerns universal delcarations. You have already seen universal packages, but you may in fact use the key ***universal*** in any breakout that has a declaration that refers only to built-ins, and you may use such declarations from any number of programs—unlike loose fragments, which are intended to be used in only one place, even if it is not completely specified. References to identifiers that are universal declarations in another document are like references to any other symbol. To indicate documents to search for universal declarations, you say:

```
\cite-universals{document}
```

The bibliography of your document will indicate, for each documented cited in the above way, the identifiers from that document that are actually referenced in the citing document. Universal declarations are automatically available for cross-document use, and appear in the contents part of their defining document.

14.3 Measuring performance

When you start to worry about the performance of your program, you will want to know where the time goes. You may measure the performance of your program by measuring frequencies or by measuring time spent in the porgram. To measure either of these quantities, set the flag in the options part of the E-L window, and run the program. After running the program, you may discard the data you have gathered (it is to allow you this option that you make the choice of what to do *after* a run), you may save the data, or you may use the data to increment a set of already saved data. Performance data is samed as an annotation space.

Now that you have the data, what can you do with it? Since it is in an annotation space, one thing that you can do is use the editor to look at it (see section 7.4). You may look at the frequency of any executable phrase. Times, on the other hand, are associated only with the names of declared funcions. In either case, looking at performance data with the editor is tedious, and you will probably want to use one of the more indirect means discussed below.

A nice way to present frequency information is provided by putting the following command in the set up portion of your document:

```
\profile{frequency-space}
```

The `frequency--space` argument is the name of the annotation space in which you deposited frequency counts. The effect of this is to place a frequency profile in the margin alongside your program fragments. This is necessarily summary information, because a line of code may not have all its pieces executed with the same frequency. However, it allows you to see at a glance where your program is busiest, and just as important, where your program has not yet been executed—if the porfile bar on a line is blank, all the line has frequency zero. The following underlying text places in the document a summary of profile information, including a scale for interpreting the frequency scale in the margin:

```
\profile-summary
```

Aside from being available for display, frequency data is of interest to the optimizing compiler. In fact, it will not run unless there is data in the annotation space `official-frequencies`.

The information about how much time is spent in each function is necessarily a listing of functions and their corresponding times. The only variation in presentation is in the order in which

the pairs are listed. The underlying text that results in a list sorted by decreasing order of times is:

```
\times-in-decreasing-order{times-space}
```

The may also be listed alphabetically or according to the structure of the program.

```
\functions-and-times{times-space}

\declaration-structure-with-times{times-space}
```

14.4 Testing revisited

The testing of larger and more complicated programs is facilitated by E-L in several ways. To test a universal declarations, respond to the E-L document query with the name of the document containing the declarations. A program box will pop up, as in section 1.2 where you responded to the query with a carriage return, but here the program box will already have some text in it. First, there will be one **Import** declaration for each universal package in your document.

```
Import pkg1;
...
Import pkgn;
```

Then, following the **Import** declarations (there will be none, if your document makes only universal declarations of other kinds), there will be a bibliography citing the document that you have indicated to the E-L query. The cursor will be just after the last **Import** declaration, so you can begin typing in statements that test some aspect of the universal declarations. As described in section 1.2, after you type in the desired statement, you can run, and if desired, repeat the edit and run cycle.

You may suspect that a program box is nothing more than an ordinary document, and that is indeed the case. Whenever you specify a document that does not have a main program, or if you specify no document (as in section 1.2), your program box is set up as an edit window on a document with a breakout whose key is ***main***, and the cursor is left inside this breakout. In the case described above, your document has a `\cite-universals` in its underlying text, and the breakout for ***main*** may already have some text in it, namely the **Import** declaration(s). The descriptor for this key is blank. All the editor commands are available on this window; if you like, you can save it as an ordinary document, once you give it a name.

When you are testing a package, you may want to directly test some private functions of the package—you might like to change your **Export** statement that lists specific functions to an **Export-all** statement. This and other temporary changes can be done with the "patching" facility, which is based on an annotation space. "Patches" are changes to your program that persist only as long as the current E-L session. To patch a phrase, you select it (see section 7.2), select "patch" from the edit menu (usually bound to ↑P), and type in the temporary replacement. This will appear in a different size or color from your permanent text, so that you know what your patches are. To see the permanent text that is patched, select it and then type ↑P. You will be given the option of unpatching, making the patch permanent, or exchanging the patch with its permanent text. When you quit an E-L session and there are active patches, you will be asked to confirm their obliteration.

14.5 Syntax extensions

Syntactic extensions to E-L may be done in several ways, with simple techniques available to add constructs of certain standard styles, and more elaborate statements necessary as the extensions become more idiosyncratic. The description of any of this machinery should be preceded by the warning that syntactic extension should be used sparingly; too much can easily be an obstacle to

reading a program, rather than a help.

The most common syntactic extensions, and the ones that are easiest to make, are the ones that provide "operators", including nofix, prefix, postfix and infix (see section 1.3 for explanation of the terminology). In each of these cases, the phrase has only one *op* that is not missing. By definition, an *operator* is the *op* from one of the phrases that has a function definition attached, and the meaning of the phrase is the application of the function. Because operator definitions are so closely tied to the function definition, their syntax is indicated as part of the **Function**, **Family**, or **Iterator** declaration (and persists only in the scope of the introduced name). The simplest operator to introduce is nofix, which by its nature must be associated with a function of no arguments.

```
Function time {nofix} () -> (integer) read-clock();
```

Thus the program may appear on the surface as if `time` is a variable, for example, in expressions such as `time + 10`. But because syntax is washed out before analysis is begun, no analyzer would make the mistake of deducing that `time - time` is equal to `0`.

The other three operators are slightly more complicated to introduce, because it is necessary to indicate their precedence (again, review section 1.3, if necessary). The easiest way to specify these precedences is by picking another operator whose precedences are to be used for the new operator. For example, if the operator * were already declared, one could say:

```
Family / {infix like *} (x, y) ...
```

Another common way to specify precedences is by indicating that *op* is a little bit stickier than another operator (specified by **above**) or a little bit less sticky than another operator (**below**). For example, it is reasonable that factorial is stickier than *, e.g., we want `3 * x !` to be the same as `3 * (x !)`, but a little less sticky than ^, i.e., we want `x ^ 2 !` to be `(x ^ 2) !` (this is of course open to argument and confusion, some of the dangers of syntactic extension). To indicate the above convention, we can say:

```
Function ! {postfix above *} ( ... ) ...
```

If there is no previous precedence between * and ^, this is equivalent to

```
Function ! {postfix below ^} ( ... ) ...
```

In defining an **infix** operator, there are actually two precedences to worry about, left and right. These are generally almost the same, but only "almost", because the grammar must specify how to group the operator next to itself. This is its associativity:

left-associative ... *op* ... *op* ... means (... *op* ...) *op* ...

right-associative
... *op* ... *op* ... means ... *op* (... *op* ...)

For obvious reasons, + and – are left associative, so would be introduced like:

```
Family + {infix left-associative} (x, y) ...
```

E-L follows the usual programming language convention that ^ is right associative.

The precedence hierarchy often divides into large groups; classically, all of the logical operators are less sticky than relational operators, which are in turn less sticky than arithmetic operators. The boundaries between these classes can be declared, and then used in **above** and **below** clauses.

The built-in classes are defined thus:

```
Precedence logical-relational;
Precedence relational-arithmetic above logical-relational;
```

The prefix **not** has high stickiness for a logical connective, and so is introduced must below the logical-relational boundary:

```
Function not {prefix below logical-relational} (b) ...
```

The **Precedence** declaration without an **above** or **below** clause is necessary to start the process of declaring precedences—one has to start somewhere.

Matchfix phrases represent a slightly new phenomenon, in that they involve two lexemes; the standard built-in example is < >, the tuple maker. There is also the absence of an old phenomenon, because there is no precedence associated with matchfix, due to null operators at the end of a phrase (see section 1.3). You may imagine the following definition as built-in:

```
Function make-tuple {matchfix < >} ( ... ) ...
```

Thus, make-tuple is also a built-in, and it is available for your use. Because of the convenience of the < > notation, the only time you would use make-tuple would be when you wanted its function value, for example, to pass as an argument.

In all cases where you have associated special syntax with the name of a function (or family or iterator), that special syntax filters through **Export** and **Import** statements. The syntax persists if the names change (using the **as** and **changing to** options of the respective statements). Further, syntax may be added in an **Import** statement, using the same { } notation used in headers. For example, if you say:

```
Package p
  Function op1 {nofix} () ...
  Function op2() ...
  ...
  Export op1, op2;
```

Now, suppose that p is imported with an ordinary **Import** p declaration. Then in the scope of this declaration, the symbols op1 andop2 will be available, and will have the same syntax as in the package declaration. Even in an import declaration changing the names, the syntactic properties of op1 and op2 are felt. That is, unless you write something like the following:

```
Import p changing op1 to f1 {standard}, op2 to f2 {nofix};
```

In the scope of this declaration, f1 refers to the function op1 defined in the package, but the **standard** syntax descriptor gives f1 "standard" syntax, in other words, you must say f1() in the scope of this import declaration to achieve the effect of saying op1 within p. In a sense then, omitting **{ }** from a function declaration is the same as saying **{standard}**. But without adding this syntax information to the import declaration, f1 would have inherited the nofix properties of op1. In the same way, f2 in the scope of the import declaration has the effect of op2() within p.

The syntactic extensions described above suffice for most purposes. The next step up in complexity of use is a **Phrase** declaration, the purpose of which is to allow the introduction of a new phrase into the grammar of its scope. The declaration must be accompanied by a description of what is to be done with the new kind of phrases. A common use of this facility is to give English-like syntax to an ordinary function application. For example, suppose you are writing a

program to control automatic test equipment, and want to be able to apply a voltage across certain terminal, for a certain amount of time, monitoring the amperage, and taking certain actions of the amperage is too high or too low. You could supply all this information as arguments to a function, but there is so much of it that readabiliity suffers. It would be preferable to say:

```
Apply voltage across a, b duration c seconds
      Amperage above d causes e
      Amperage below f causes g;
```

So if this notion is really natural in your programming world, you can declare syntax for it.

```
Phrase [name]
  {Apply} {voltage} {across} a {,} b{duration} c {seconds}
  {Amperage} {above} d {causes} e
  {Amperage} {below} f {causes} g {above  semicolon}
becomes
  apply-voltage(a, b, c, d, lambda() e, f, lambda() g);
```

Each bold brace surrounds an *op* in your phrase. Note the use of **{,}** as part of the syntax of your phrase, and that some of your *op*s do not allow phrases between them. Recall from section 1.3 that the first and last *op* may be missing. A missing *op* in a **Phrase** declaration must have its braces just like any other; instead of putting in an *op*, you put in a specification for the precedence at that end of the phrase, using **above**, **below**, or **like**.

The only reason to name a phrase is to be able to mention it in **Export** and **Import** statements. In keeping with the usual flavor of these statements, the syntax defined by a **Phrase** declaration is exported by an **Export-all** statement, unless its *name* is specifically mentioned in **except** clause. The syntax defined by a **Phrase** declaration is exported by an **Export** statement only when its *name* occurs in the list to be exported.

END

DTIC

7 - 86