Guide for New Eli Users

$Revision: 3.4 $

Compiler Tools Group
Department of Electrical and Computer Engineering
University of Colorado
Boulder, CO, USA
80309-0425

Table of Contents

A text processor is a program that takes a sequence of characters as input, computes some set of values based on that sequence, and then carries out some action determined by the computed values. A desk calculator program is therefore a text processor, as is a Pascal compiler and the input subsystem of a transaction processor.

The Eli system creates executable text processing programs from specifications. A user provides Eli with specifications that describe a particular text processing problem, and Eli creates a program to solve that problem. By making simple requests of Eli, the user can test the generated program, obtain an executable copy, or obtain a directory containing a source copy.

This manual is intended for the new Eli user. It begins with a general characterization of text processing problems. The emphasis is on a strategy for decomposing such problems and describing their components, but an indication of how Eli builds working software from these descriptions is also included.

The simplest way to understand a system is to see how it is used. We therefore present a concrete text processing problem and show, step by step, how a program to solve it is specified and produced. Each component of the specification is discussed, and exercises provided to lead you through the production process. The text of the specification is part of the Eli distribution.

As with any complex system, there are ways to customize Eli to fit particular working styles. We therefore describe some of the common things that you might want to do to change Eli's behavior in order to fit your working style.

Finally, we give an overview of the Eli documentation.

1 How Eli Creates a Text Processing Program

The program generated by Eli reads a file containing the text, examining it character-by-character. Character sequences are recognized as significant units or discarded, and relationships among the sequences are used to build a tree that reflects the structure of the text. Computations are then carried out over the tree, and the results of these computations determine the processor's output. Thus Eli assumes that the original text processing problem is decomposed into the problems of determining which character sequences are significant and which should be discarded, what tree structure should be imposed on significant sequences, what computations should be carried out over the resulting tree, and how the computed values should be encoded and output.

A user describes a particular text processing problem to Eli by specifying the characteristics of its subproblems. For example, the tree structure that should be imposed on significant character sequences is specified to Eli by providing a context-free grammar. Different subproblems are characterized in different ways, and they are specified by descriptions written in different languages. Those specifications are processed by a variety of tools, which verify their consistency and generate C-compatible C++ code to solve the specified subproblems.

Eli focuses the user's attention on specification by automating the process of tool invocation. It responds to a request for a piece of computed information, called a *derived object*, by invoking the minimum number of tools necessary to produce that information. Derived objects are automatically stored by Eli for later re-use, significantly improving the response time for subsequent requests.

This chapter provides an overview of the most important subproblems and how they can be described, summarizes the available descriptive mechanisms and explains the most common derived objects, and indicates how Eli is used to create and test programs.

1.1 How to Decompose a Text Processing Problem

There is considerable variability in the specific decompositions for particular text processing problems. For example, one subproblem of the compilation problem for many programming languages is *overload resolution*: how to decide that (say) a particular `+` operator means integer addition instead of real addition. Overload resolution would probably not be a subproblem of the problem of creating a PostScript version of a musical score from a description of the desired notes. This section briefly reviews five major subproblems common to a wide range of problems:

Syntactic analysis
: Determining the structure of an input text

Lexical analysis
: Recognizing character sequences

Attribution
: Computing values over trees

Property storage
: Maintaining information about entities

Text generation
: Producing structured output

Syntactic analysis determines the *phrase structure* of the input text. The phrase structure is described to Eli by a *context-free grammar*. Here is a context-free grammar describing the structure of a geographical position (see Section 'Context-Free Grammars and Parsing' in Syntactic Analysis):

```
Position: Latitude Longitude .
Latitude: NS Coordinate .
Longitude: EW Coordinate .
NS: 'N' / 'S' .
EW: 'E' / 'W' .
Coordinate: Integer Float .
```

This grammar has eight symbols (`Position`, `Latitude`, `Longitude`, `NS`, `EW`, `Coordinate` `Integer` and `Float`), four literals (`'N'`, `'S'`, `'E'` and `'W'`) and eight rules (`x: y / z .` is an abbreviation for the two rules `x: y .` and `x: z .`). Two of the symbols, `Integer` and `Float`, are not defined by the grammar. One of the symbols, `Position`, is not used in the definition of any other symbol.

Symbols that are not defined by the grammar are called *terminal* symbols; those defined by the grammar are called *nonterminal* symbols. The (unique) symbol not used in the definition of any other symbol is called the *axiom* of the grammar.

The entire input text is called a *sentence*, and corresponds to the axiom. Thus the input text `N41 58.8 W087 54.3` is a sentence corresponding to `Position`. A *phrase* is a portion of the sentence corresponding to a nonterminal symbol of the grammar. For example, `N41 58.8` is a phrase corresponding to `Latitude`, and `087 54.3` is a phrase corresponding to `Coordinate`. `54` and `58.8 W087` are *not* phrases because they do not correspond to any symbols. (`54` is a part of the phrase `087 54.3` corresponding to `Coordinate` and `58.8 W087` is made up of part of the phrase corresponding to `Latitude` and part of the phrase corresponding to `Longitude`.)

Lexical analysis is the process that examines the source program text, retaining significant character sequences and discarding those that are not significant. A character sequence is significant if it corresponds to a literal in the grammar or to a terminal symbol. In the sentence `N41 58.8 W087 54.3`, the significant character sequences are `N`, `41`, `58.8`, `W`, `087` and `54.3`. The spaces separating the numbers of a `Coordinate` and preceding the `Longitude` are not significant.

Eli obtains information about character sequences that correspond to literals directly from the grammar. The user must, however, provide descriptions of the character sequences corresponding to each terminal symbol. Those descriptions determine the *form* of the character sequences, but not their *content*. For example, the character sequences corresponding to the terminal symbol `Integer` might be described as sequences of one or more decimal digits, and those corresponding to the terminal symbol `Float` might be described as pairs of such sequences separated by a dot (see Section 'Regular Expressions' in Lexical Analysis):

```
Integer:  $[0-9]+
Float:    $[0-9]+"."[0-9]+
```

Computations may be specified over a tree that reflects the phrase structure of a sentence. That tree has one node corresponding to each phrase, with the root corresponding to the sentence. For example, the tree that reflects the structure of the sentence `N41 58.8 W087 54.3` has a node corresponding to the `Latitude` phrase `N41 58.8`. Children of a node correspond to the immediate component phrases of the phrase corresponding to that node. Thus the children of the the node corresponding to the `Latitude` phrase `N41 58.8` would correspond to the phrases `N` and `41 58.8`, because the immediate components of the `Latitude` phrase `N41 58.8` are an `NS` phrase (`N`) and a `Coordinate` phrase (`41 58.8`).

One or more values, called *attributes*, may be associated with each tree node. Computations over the tree may involve only attribute access, C constants, and function application. Here is an example:

```
RULE DegMin: Coordinate ::= Integer Float
COMPUTE
  Coordinate.minutes=Minutes(Integer, Float);
END;
```

According to the rule `DegMin`, `Coordinate` has a `minutes` attribute attached to it. The `minutes` attribute of `Coordinate` is computed by applying the function `Minutes` to the values representing `Integer` and `Float`, but how are those values obtained? Clearly they must depend on the character sequences corresponding to these terminal symbols.

A single integer value can be used to represent any terminal symbol. That value is determined by a *token processor* whose name is attached to the definition of the character sequences corresponding to the symbol, enclosed in brackets (see Section 'Token Processors' in Lexical Analysis):

```
Integer:  $[0-9]+            [mkstr]
Float:    $[0-9]+"."[0-9]+   [mkstr]
```

The processor `mkstr` is a library routine that stores the character sequence in an array and yields the sequence's index. `Minutes` can then use the index values to obtain the stored strings, convert them to numbers, and perform the necessary computation.

Often an input text describes some set of *entities* and the relationships among them. For example, a program in a conventional language may describe some set of constants, variables, types and procedures, and how these entities are related in the execution of an algorithm. Entities may be defined by one part of the input text and used by another. It is therefore necessary for computations over the tree representing the phrase structure of a sentence to be able to refer to entities and their properties at arbitrary points. Eli provides a *definition table* to meet this requirement (see Section 'Definition Table Design Criteria' in Definition Table).

Each entity can be represented by a unique *definition table key*, which allows access to arbitrary information about that entity. The Eli user specifies the information that might be stored, and possibly a set of *query* and *update* operations for that information. (Eli provides a standard query operation and a standard update operation that suffice for most purposes.)

Library modules are available for associating unique definition table keys with identifiers according to the scope rules of the input text, and for maintaining various kinds of information about entities. Definition table keys themselves can be stored as attributes, compared for equality, passed to query and update routines, and accessed either directly or remotely. A distinguished value, `NoKey`, represents the absence of a definition table key.

Output may be derived from arbitrary information in the input text. The elements of the output may be arranged in arbitrary ways based on computations over the tree representing the phrase structure of a sentence. It is therefore useful to be able to build up a complex output data structure piecemeal, combining components according to information gleaned from computation.

The program text generation facility allows the user to specify templates that describe output text fragments (see Section 'Pattern Specifications' in PTG: Pattern-Based Text Generator). "Holes" in these templates can be filled with text generated according to other templates. The result is a directed, acyclic graph in which each node represents a single template and the children of that node represent generated text fragments. Text at the leaves can be generated by arbitrary user-supplied routines. (A library module provides common leaf generation routines.)

Eli generates a set of functions, one per template, that are invoked during computations to build the directed, acyclic graph. These functions return values that can be stored as attributes, passed to text generation functions, and accessed either directly or remotely. A distinguished value, `PTGNULL`, represents the absence of a graph.

Printing functions are also provided by Eli to output the generated text on an arbitrary file (including the standard output unit). These functions accept a graph and perform a depth-first, left-to-right traversal. The text is output during the traversal, with text generated by a common subgraph being output once for each parent of that subgraph.

1.2 Descriptive Mechanisms Known to Eli

The Eli user describes the subproblems of a particular text processing problem. Eli derives code fragments from these descriptions, and combines them into a complete program. Each description is given in a notation that is ideally suited to the subproblem being described.

Subproblem descriptions are placed into files, each of which has a *type*. The type is indicated by the file name extension: '`foo.c`' is a type-'`c`' file. Eli recognizes type-'`c`' and type-'`h`' files as C program text and include files respectively. Here is a list of the other file types recognized by Eli:

'`specs`' A collection of object names, one per line.

'`con`' A description of the phrase structure of the input text. Each phrase may be associated with a computation to be carried out when that phrase is recognized. Eli generates a parser from these specifications. See Syntactic Analysis.

'`gla`' A description of character sequences, whether they are meaningful or not, and what (if any) computation should be carried out when they are recognized in the input text. Eli generates a scanner from these specifications. See Lexical Analysis.

'`lido`' A description of the structure of a tree and the computations to be carried out on that tree. Eli generates a tree-walking evaluator from these specifications. For a discussion on constructing computations in trees, see LIDO - Computation in Trees. For reference, see LIDO - Reference Manual.

'`ctl`' Constraints on evaluator generation. Eli uses these specifications to modify its behavior when constructing the routine that carries out the computations. See LIGA Control Language Reference Manual.

'ptg' A description of structured output text. Eli generates a set of output functions from these specifications. See Pattern-Based Text Generator.

'pdl' A definition of entities and their associated properties. Eli generates a definition table module from these specifications. See PDL Reference Manual.

'oil' A definition of possible tree node re-labeling. Eli generates a re-labeling module from these specifications. See OIL Reference Manual.

'clp' A description of the meanings of command line arguments. Eli generates a module that accesses command line arguments from these specifications. See CLP Reference Manual.

'map' A description of the relationship between the phrase structure of the input text and the structure of the tree over which computations are to be made. Eli uses this specification to determine the tree building actions that must be attached to rules of the parsing grammar. See Syntactic Analysis.

'sym' This is provided for backward compatibility with previous Eli releases for specifying symbolic equivalence classes. It is superseded by type-'map' files. See Section 'Specifying symbolic equivalence classes' in Syntactic Analysis.

'delit' Specifies literals appearing in a type-'con' file that are to be recognized by special routines. Each line of a type-'delit' file consists of a regular expression (see Section 'Regular Expressions' in Lexical Analysis) optionally followed by an identifier. The regular expression defines the literal to be recognized specially. A `#define` directive making the identifier a synonym for that literal's syntax code is placed in the generated file 'litcode.h'.

'str' Specifies initial contents of the identifier table. Each line of a type-'str' file consists of two integers and a sequence of characters. The first integer is the syntax code to be returned by `mkidn` (see Section 'Unique Identifier Management' in Library Reference Manual), and the second is the length of the character sequence. The integer representing the length is terminated by a single space. The character sequence begins immediately after this space, and consists of exactly the number of characters specified by the length.

'gnrc' Defines a generic module. Generic modules can be instantiated to yield collections of specifications that solve specific problems.

'fw' Combines a collection of strongly-coupled specifications with documentation describing their relationships. Eli splits these specifications according to their types and processes them individually. It can also create a printed document or on-line hypertext from a type-'fw' file.

'phi' Material to be included in some part of the generated processor. Specification files of this type should have a name consisting of three parts: `foo.bar.phi`. All the files whose names end in 'bar.phi' are concatenated together in arbitrary order to form a file named `bar.h`. An `#include` directive can then be used to incorporate '`bar.h`' into any generated file.

Any of these files can contain C-style comments and preprocessor directives such as `#include`, `#define` and `#ifdef`. The C preprocessor is applied to files of all types except type-'`fw`' *before* those files are examined. C-style comments and preprocessor directives appearing in type-'`fw`' files are passed unchanged to the files generated from the type-'`fw`' file.

Eli includes three pre-defined header files, which are usually generated from type-'`phi`' specifications, in specified places:

'`HEAD.h`'
: Included at the beginning of the main program, the tree constructor, and the attribute evaluator. This header file is used primarily to define abstract data types used in tree computation.

'`INIT.h`'
: Included at the beginning of the main program's executable code. '`INIT.h`' may contain declarations, but only if they appear at the beginning of compound statements that lie wholly within '`INIT.h`'. The content of '`INIT.h`' will be executed before any other code. Its primary purpose is to initialize abstract data types used in tree computation.

'`FINL.h`'
: Included at the end of the main program's executable code. '`FINL.h`' may contain declarations, but only if they appear at the beginning of compound statements that lie wholly within '`FINL.h`'. The content of '`FINL.h`' will be executed after all other code. Its primary purpose is to finalize abstract data types used in tree computation.

1.3 Common Derived Objects

Eli recognizes three kinds of object: a *file*, a *string* and a *list*. Examples of files are a specification file such as those mentioned in the previous section, an executable binary file, or an output file from a test run. A flag to a command is an example of a string. Lists are ordered sequences of objects, such as the arguments to a command or the Makefile, C files, and header files that implement a text processor.

Source objects can be created or modified directly by the user. They can be regular files, directories, or symbolic links. Source objects cannot be automatically recreated by Eli; they are the basic building blocks from which Eli creates all other objects. Every source object is given a type by Eli based on its host filename, and this type determines what derived objects can be produced from the source object.

The file type of a source file is the longest suffix of the file name that matches one of the source type suffixes listed in the last section. If no suffix match is found, the file type is empty.

Derived objects are objects that can be produced from source objects and other derived objects through the invocation of one or more tools. Tools are invoked only as needed to create a specified derived object. Eli automatically caches derived objects for re-use in future derivations. Derived objects are created and modified only by Eli itself, not by users.

A derived object is named by an *odin-expression* (see Section 'Referring to Objects' in User Interface). Lexically, an odin-expression is composed of a sequence of *identifier* and *operator* tokens, and is terminated by a newline character. An odin-expression can be continued on multiple lines by escaping each newline character with a backslash. This backslash (but not the newline) is deleted before the expression is parsed. Multiple odin-expressions can be specified on the same line by separating them with semicolon operators.

An identifier token is just a sequence of characters. The following characters must be escaped to be included in an identifier:

```
: + = ( ) / % ; ? $ < > ! # \ ' space tab newline
```

A single character can be escaped by preceding it with a backslash (e.g. `lost\+found`). A sequence of characters can be escaped by enclosing them in single quote marks (e.g. `'lost+found'`).

Unescaped *white space* characters (spaces, tabs, and newlines) are ignored during parsing except when they separate adjacent identifiers.

Here are a number of odin-expressions that name common objects derived from the same collection of specifications (all of the spaces are redundant). The identifier following a colon (`:`) is an *object type* (or *product*) that characterizes the properties of the derived object, while the identifier following a plus (`+`) is a *parameter type* that modifies those properties without changing the nature of the derived object. (For the characteristics of all of the products and parameters defined by Eli, see Products and Parameters.)

`sets.specs :exe`

> is the executable program generated by Eli from the specifications enumerated in `sets.specs`. It is a normal program for the machine on which it was generated, and is independent of Eli.

`sets.specs :source`
: is a set of C files, a set of header files, and a Makefile. The result of running `make` with this information is the executable program generated by Eli from the specifications enumerated in `sets.specs`.

`sets.specs :exe :help`
: is a browser session that helps to explain inconsistencies in the specifications enumerated in `sets.specs`. It provides cross-references to on-line documentation and allows you to invoke an editor on the proper files to make corrections.

`. +cmd=(sets.specs :exe) (input) :run`
: is the result of running the program generated by Eli from the specifications enumerated in `sets.specs` as a command with the file '`input`' from the current directory as an argument. The name of the directory (in this case `.`, the name of the current directory) in which the program is to be executed precedes the parameter that defines the command to be executed.

`sets.specs +monitor +arg=(input) :mon`
: is an interaction with the program generated by Eli from the specifications enumerated in `sets.specs`, as it processes the data file '`input`'. This interaction allows you to follow the execution at the level of your specifications, rather than at the level of the machine on which the program is running.

`sets.specs +debug :dbx`
: is an interaction with the program generated by Eli from the specifications enumerated in `sets.specs` using the symbolic debugger of the machine on which the program is running. It is useful when some of your specifications are written directly in C. (Replace `dbx` with `gdb` to use the GNU symbolic debugger.)

`sets.specs :gencode :viewlist`
: is an interactive shell executing in a directory containing all text files generated by Eli from the specifications enumerated in `sets.specs`. This facility is sometimes useful in diagnosing compiler errors due to type mismatches.

`sets.specs :exe :err >`
: is the raw set of reports generated by inconsistencies in the specifications enumerated in `sets.specs`, written to the screen. (It would be sent to your editor if you replaced `>` with `<`.) This display is sometimes useful if the reports are garbled by the `help` derivation.

`sets.specs :exe :warn >`
: is the raw set of reports generated by anomalies in the specifications enumerated in `sets.specs`, written to the screen. (It would be sent to your editor if you replaced `>` with `<`.) This display is sometimes useful if the reports are garbled by the `help` derivation.

1.4 How to Request Product Manufacture

Eli is invoked by giving the command `eli`. If you have never used Eli before, it will have to establish a *cache* (see Section 3.1 [Hints on Cache Management], page 29). This process is signaled by a long sequence of messages about installing packages, followed by a note that the packages have been compiled.

After printing an identifying banner, Eli writes the prompt `->` and waits for input. The interactive session can be terminated by responding to the `->` prompt with a `^D`, and you can browse the documentation by responding with a question mark.

Entering a derived object name in response to the `->` prompt constitutes a request to bring that derived object up to date with respect to all of the source objects on which it depends. Eli will carry out the minimum number of operations required to satisfy this request. When the next `->` prompt appears, the given object will be up to date.

Bringing an object up to date does not yield a copy of that object. To obtain a copy, you must add an output request. The precise form and effect of an output request depends on whether the object being output is a file object or a list object. All source objects are file objects; to find out the kind of a derived object, consult Products and Parameters.

Here are some examples of common output requests:

`sets.specs :parsable >`

: requests that the derived file object `sets.specs :parsable` be written to the standard output (normally the screen). Garbage will result if the derived object is not a text file.

`sets.specs :exe > trans`

: requests that the derived file object `sets.specs :exe` be written to file '`trans`'. The derived object must be a file object, but it need not be text. If the file '`trans`' does not exist, it will be created; if it does exist, it will be overwritten if its content differs from that of the derived object `sets.specs :exe`. If '`trans`' exists and is a directory, a file named `sets.specs.exe` will be written to that directory (see Section 'Extracting and Editing Objects' in User Interface).

`sets.specs :source > src`

: requests that the derived list object `sets.specs :source` be written to directory '`src`'. The directory '`src`' must exist. A file in '`src`' before the request will be overwritten only if it has the same name as one of the file objects in the list `sets.specs :source`, but different content. (Normally, 'src' would be either an empty directory or one that contains an earlier version of sets.specs :source.)

sets.con <
: requests that your current editor be invoked on the object sets.con

1.5 How to Invoke Unix Commands

While one is interacting with Eli, there are a number of situations in which one wishes to execute normal operating system commands. These commands can be executed with or without derived objects as arguments. We have already seen the most general form, a derived object that is an execution of an arbitrary command in an arbitrary directory (see Section 1.3 [Products], page 9). Although this facility is general enough to handle *any* command execution, it is cumbersome for simple commands.

The ! character introduces a host command line (see Section 'Unix Commands' in User Interface). If the first non-white space character following the ! is not :, ; or = then the rest of the line is treated as a single, escaped sequence of characters. This avoids the confusion resulting from interactions between the escape conventions of host commands and odin-expressions. A leading :, ;, = or whitespace can be included in the escaped sequence by preceding it with \.

If the name of a file object precedes the ! character, that object is brought up to date and the name of a file containing it is appended to the host command line.

Here are examples of some typical command invocations using !:

!ls
: lists the files in the current directory.

!mkdir src
: makes a new subdirectory of the current directory.

(sets.specs :exe) ! size
: provides information about the space used by the processor generated from sets.specs.

input +cmd=(sets.specs:exe) :stdout ! diff desired
: compares the file 'desired' with the result of applying the processor generated from sets.specs to the file 'input'.

2 Example of Eli Use

The example in this chapter illustrates how text processors are specified to Eli. Each section covers a major step in the development process, discussing the purpose of that step and then carrying it out for the example. A set of exercises is provided with each section. The purpose of these exercises is to familiarize you with the basic facilities that Eli provides for dealing with specifications, and how Eli is typically used.

All of the text used in the exercises can be obtained, and the exercises themselves can be carried out, using the facilities of Eli's system documentation browser described in the first set of exercises given below (see [Exercises], page 17).

2.1 Statement of the problem to be solved

You need to classify a collection of several thousand words. There are no duplicate words in any class, but a given word may belong to more than one class. The classes have arbitrary names, and no two classes may have the same name.

A C program will manipulate the words and classes. Because of the application, classes will be relatively fluid. The customer expects that new words and classes will be added, and the classification of existing words changed, on the basis of experience and changing requirements. Nevertheless, the system design requires that the data be built into the C program at compile time.

The system designers have settled on an internal representation of the data involving an array for each class containing the words in that class as strings, an array containing the class names as strings, and an array containing the sizes of the classes as integers. The number of classes is also given. All of these data items are to be specified in a single header file. Here is an example of such a header file for a simple classification:

```
int number_of_sets = 3;

char *name_of_set[] = {
"colors",
"bugs",
"verbs"};

int size_of_set[] = {
3,
5,
```

```
4};

char *set_of_colors[] = {
"red",
"blue",
"green"};

char *set_of_bugs[] = {
"ant",
"spider",
"fly",
"moth",
"bee"};

char *set_of_verbs[] = {
"crawl",
"walk",
"run",
"fly"};

char **values_of_set[] = {
set_of_colors,
set_of_bugs,
set_of_verbs};
```

Although the meaning of the internal representation is straightforward, it is quite clear that making the necessary alterations will be a tedious and error-prone process. Any change requires compatible modifications of several arrays. For example, moving a word from one class to another means not only cutting and pasting the word itself, but also changing the sizes of both classes.

It would be simpler and safer to define the classification with a notation ideally suited to that task, and generate the header file from that definition. Here is an example of an obvious notation, defining the classification represented by the header file given above:

```
colors{red blue green}
bugs{ant spider fly moth bee}
verbs{crawl walk run fly}
```

The remainder of this chapter discusses the specification and generation of a program that translates class descriptions into header files. This program must accept a class description, verify that class names are unique and that there is only one occurrence of any given word in a class, and then write a header file defining the appropriate data structure. Its specification is broken into four parts, each stored in a separate file:

`sets.con`
: A context-free grammar describing the structure of the input text.

`word.gla`
: A specification of the character sequences that are acceptable words, and how those character sequences should be represented internally, plus a specification of the character sequences to be ignored.

`symbol.lido`
: A specification of the context conditions on uniqueness of set names and elements within a single set.

`code.fw`
: A specification of the form of the output text and how it is constructed.

File '`sets.specs`' lists the names of these four files and contains requests to instantiate three library modules that support them.

Exercises

Invoke Eli and type `?`, followed by a carriage return. This request will begin a documentation browsing session. Use the browser's `goto` command to place yourself at node `(novice)tskex` (Browser commands are described in Section 'Some Advanced Info Commands' in Info.)

If you are using a computer with multiple-window capability, your documentation browsing session is independent of your Eli session, so you can simultaneously browse the documentation and make Eli requests. Otherwise you must terminate the document browsing session in order to make an Eli request. In that case, you might want to make a note of your current node (given in the highlighted status line) before exiting. When you begin a new session, you can then use the `g` command to go directly to that node.

1. Use the documentation browser's `run` command to obtain a copy of the complete specification. You will use this copy to do the exercises in the remainder of this chapter.
2. Verify that you have obtained all of the specification files by making the following Unix request via Eli (see Section 'Running Unix commands from Eli' in User Interface Reference Manual):

   ```
   -> !ls
   ```
3. Examine the file that is not a part of the specification by requesting Eli to display it on screen (see Section 'Copying to Standard Output' in User Interface Reference Manual):

   ```
   -> input>
   ```

2.2 Specifying the desired phrase structure

The first step in specifying a problem to Eli is to develop a context-free grammar that describes the phrase structure of the input text. This structure must reflect the desired semantics, and it must be possible for Eli to construct a parser from the grammar. Grammar development is a surprisingly difficult task. It is best to concentrate on the meaning of the tree structure as you are developing the grammar, and not try to economize by using the same symbols to describe constructs that look the same but have different meanings.

One possible description of the structure of the set definition text is (see Section 'How to describe a context-free grammar' in Syntactic Analysis):

```
text: set_defs .
set_defs: set_def / set_defs set_def .
set_def: set_name '{' set_body '}' .
set_name: word .
set_body: elements .
elements: set_element / elements set_element .
set_element: word .
```

Here each set definition is described as a `set_name` followed by a bracketed `set_body`. The text will be made up of an arbitrary number of such definitions. A `set_body`, in turn, consists of an arbitrary number of elements. The `set_name` and each `set_element` is a `word`.

This structure represents the semantics of the input text: Set names and set elements have different meanings, even though they are both written as words. Set bodies are significant units, even though they have the same form as any subset of themselves. The following specification would *not* reflect the semantics, even though it is simpler and describes the same input language:

```
text: set_defs .
set_defs: set_def / set_defs set_def .
set_def: word '{' elements '}' .
elements: word / elements word .
```

Exercises

To get information about whether Eli can construct a parser from a grammar without actually trying to build the whole program, use the `:parsable` product (see Products and Parameters Reference Manual). In order to be parsable, the grammar must satisfy the *LALR(1) condition*. If the LALR(1) condition is satisfied, `parsable` will indicate that fact. Otherwise it will say that the grammar is not LALR(1) and provide a listing of conflicts (see Section 'How to Resolve Parsing Conflicts' in Syntactic Analysis).

1. Tell Eli to explain what it is doing, and request verification that Eli can generate a parser. Append `>` to the derivation to tell Eli to copy the results on the screen.

   ```
   -> LogLevel=4
   -> sets.specs :parsable>
   ```

   When the process is complete, repeat the last request. To save keystrokes, you can scroll through the *history* (see Section 'The History Mechanism' in User Interface) using the up- and down-arrow keys.

   Explain the very different responses to the two requests for verification of parsability.
2. Request an editing session on the file '`sets.con`' (see Section 'Editing with the Copy Command' in Eli User Interface Reference Manual):

   ```
   -> sets.con<
   ```

   Delete `text` from the first rule of the grammar, leaving the colon and everything following it unchanged. Then request execution as before, scrolling through the history:

   ```
   -> sets.specs:parsable>
   ```

   Why was Eli's response so much shorter than before?
3. Obtain help diagnosing the error you created by deleting `text`:

   ```
   -> sets.specs :parsable :help
   ```

   This request will start a documentation browsing session. Follow the menu to the display for the file in error, and use the edit command to gain access to that file. Correct the error by inserting `text` before the colon, exit the editor, and then quit the browsing session. Use the Eli history mechanism to repeat the last request:

   ```
   -> sets.specs :parsable :help
   ```

   Explain Eli's response to this request. Why was no documentation browsing session started? Why was the response so much shorter than the response to the original request for derivation and execution of the processor?
4. Your request for help after fixing the error really didn't demonstrate that the grammar was parsable, because it didn't show you the result. Request the result:

   ```
   -> sets.specs :parsable>
   ```

   Explain Eli's response to this request. What steps were carried out to satisfy it? Why were these the only ones necessary?
5. Delete the braces `{ }` from the rule defining `set_def` in file '`sets.con`'. This change makes the entire input text nothing but a list of words, with no differentiation between set names and elements or between different sets. Eli will not be able to generate a parser from this grammar. Request verification of parsability to see the error report:

```
-> sets.specs :parsable >
```

A *shift-reduce conflict* is a situation in which the parser can't tell whether it should recognize a complete phrase or continue to add symbols to an incomplete phrase. In this example, the next word is either the name of a new set (and thus the current `elements` phrase is complete), or it is another set element (and thus belongs to the current `elements` phrase).

Add a comma as a separator between set elements, but do not re-introduce the braces. Do you think that Eli will be able to generate a parser? Briefly explain, and then verify your answer. (For a more complete treatment of conflict resolution, see Section 'How to Resolve Parsing Conflicts' in Syntactic Analysis.)

Restore the original specification, or change '`input`' to conform to your new specification, to ensure correct behavior for later exercises.

2.3 Nonliteral character sequences and comments

The terminal symbols of the grammar are the literal braces and the nonliteral symbol `word`. Eli can easily deduce that the braces are significant characters, but we must provide a definition of the significant character sequences that could make up a `word`. We must also describe how to capture the significant information in a word for further processing.

Eli normally assumes that white space characters (space, tab and newline) are not significant. If we want to provide a facility for commenting a classification then we must additionally define the form of a comment and specify that character sequences having this form are also not significant.

Here is one possible description of the non-literal character sequences and comments:

```
word:   $[a-zA-Z]+      [mkidn]
        C_COMMENT
```

The first line defines a `word` to be any sequence of one or more letters (see Section 'Regular Expressions' in Lexical Analysis). Whenever such a sequence is recognized in the input text, `mkidn` is invoked to capture the significant information represented by the sequence. This processor associates an integer with the recognized sequence, and arranges for that integer to become the value representing the character sequence. If two character sequences recognized as words are identical, `mkidn` will represent them with the same integer; distinct sequences are represented by different integers.

The second line of the specification does not begin with a symbol followed by `:`, which indicates that the character sequences it describes are not significant. It uses a *canned description* to describe character sequences taking the form of C comments (see Section 'Canned Symbol Descriptions' in Lexical Analysis). Thus any character sequence taking the form of a C comment will be ignored in the input text read by the generated program.

Exercises

1. Tell Eli to keep quiet about what it is doing, and then ask it to run the processor derived from your specification:

   ```
   -> LogLevel=2
   -> input +cmd=(sets.specs :exe) :stdout >
   ```

2. The sample input file does not contain either comments or errors. Introduce an error by inserting a digit into one of the words of the example and repeat your request:

   ```
   -> input<
   -> input +cmd=(sets.specs :exe) :stdout >
   ```

   Briefly explain Eli's response to this request.

3. Verify that the generated processor correctly handles C comments.
4. Change the specification to accept only words that begin with upper-case letters. Generate a translator and verify its correctness.
5. Change the specification to allow Ada comments instead of C comments. (Hint: see Section 'Canned Symbol Descriptions' in Lexical Analysis.) Generate a translator and verify its correctness.

2.4 Managing source text definitions

The statement of the problem requires that the names of the classes be unique, and that there be only one occurrence of a given word in a given class. This sort of condition is very common in translation problems. It involves recognition of regions and specific entities within those regions. For example, a `set_body` is a region and a `set_element` is a specific entity within that region. The check to be made is that no `set_element` appears more than once in any `set_body`.

Regions are defined by the grammar. Entities may be defined both by the grammar and by the values representing the terminal symbols: the grammar selects a particular kind of phrase, while the instances of this phrase are differentiated by the values of their terminal symbols. Some computation must be carried out over the region to verify the condition. This computation is standard, involving only the concepts of region and entity, so a *generic module* can be used to carry it out.

In order to use a generic module we must instantiate that module and connect the concepts it provides to the specification of our problem. Instantiation is handled in the '`sets.specs`' file:

```
$/Name/AlgScope.gnrc :inst
$/Prop/Unique.gnrc :inst
```

The `AlgScope` module provides the concept of nested regions containing entities distinguished by integer values (see Section 'Algol-like Basic Scope Rules' in MODLIB – Specification Module Library). The `Unique` module provides the concept of an error for entities that appear more than once in a region (see Section 'Check for Unique Object Occurrences' in MODLIB – Specification Module Library).

An attribute grammar fragment is used to connect the concepts provided by these two modules to the specification of the phrase structure:

```
ATTR Sym: int;

SYMBOL Entity INHERITS IdDefScope, Unique COMPUTE
  SYNT.Sym=TERM;
  IF(NOT(THIS.Unique),
     message(ERROR,"Multiply-defined word", 0, COORDREF));
END;

SYMBOL text INHERITS RootScope, RangeUnique END;
SYMBOL set_body INHERITS RangeScope, RangeUnique END;
SYMBOL set_name INHERITS Entity END;
SYMBOL set_element INHERITS Entity END;
```

The symbol `Entity`, which does not occur in the context-free grammar, is used to represent the concept of a word that must be unique within some region: a `set_name` must be unique over the region lying outside of all sets, while a `set_element` must be unique over the set in which it appears. `Entity` allows us to gather all of the characteristics of that concept in one place – a *symbol attribution* – and then use inheritance to associate those characteristics with the symbols that embody the concept.

An `Entity` appears in the input text as a word, which is represented internally by the value representing the terminal symbol `word`. That value was established by `mkidn` when the word was recognized (see Section 2.3 [GLA specification], page 20).

The two symbols inheriting `Entity`, `set_name` and `set_element`, are defined by rules that have terminal symbols on their right-hand sides. `TERM` can be used in a symbol computation to represent the value of a terminal symbol appearing on the right-hand side of any rule defining the given symbol (see LIDO Reference Manual). Thus `SYNT.Sym=TERM` sets the `Sym` attribute of the `Entity` to the value representing the terminal symbol defining that `Entity`. (`SYNT` means that the computation takes place in the *lower context* of the symbol, i.e. in the rules corresponding to the phrases `set_name: word` and `set_element: word`. See LIDO Reference Manual.)

The concept of a definition within a region is embodied in the symbol `IdDefScope` exported by the `AlgScope` module, while the concept that such a definition must be unique is embodied in the symbol `Unique`, exported by the `Unique` module. Thus `Entity` inherits from these two symbols.

If the `Unique` attribute of the `Entity` is false, then the `message` operation is invoked to output an error report. The report has severity `ERROR`, which indicates that a definite error has been detected and therefore no object code should be produced (see Section 'Source Text Coordinates and Error Reporting' in Library Reference Manual). It is placed at the source text coordinates (line and column) represented by `COORDREF`, and consists of the string `Multiply-defined word`. `COORDREF` always refers to the location of the leftmost character of the phrase corresponding to the rule in which it appears. Since this computation is inherited by `set_name` and `set_element`, it appears in rules corresponding to the phrases `set_name: word` and `set_element: word`. Any error report will therefore refer to the leftmost character of the multiply-defined word.

The `text` is the outermost region (`RootScope`), within which the set names are defined. Each set body is an inner region (`RangeScope`), within which the set elements are defined. Therefore `text` inherits the `RootScope` computations and `set_body` inherits the `RangeScope` computations.

A `set_name` must be unique within the `text` and a `set_element` must be unique within its `set_body`, so both `text` and `set_body` inherit the `RangeUnique` computations.

Exercises

1. Create a subdirectory '`src`' and then request source code for the set processor (see Section 'Running Unix commands from Eli' in User Interface Reference Manual):

   ```
   -> !mkdir src
   -> sets.specs :source >src
   ```

   Does '`src`' *really* contain a version of the translator that is independent of Eli? How can you be certain?

2. Try deriving source code without first creating a directory:

   ```
   -> sets.specs :source >foo
   ```

   What was the result? Can you explain what happened?

3. Request an executable version of the translator:

```
-> sets.specs :exe >sets.exe
```

Is this executable independent of Eli? How can you be certain?

2.5 Creating structured output text

At least two specifications, and sometimes more, are needed to create structured output text. The form of the text must be described, along with the source of its components. Eli generates a set of procedures from the description of the form of the text, one for each kind of component. The information about the components is then provided by a computation that invokes these procedures with appropriate arguments.

We have already seen how Eli allows us to break the specification into files that encapsulate individual tasks. So far, each of the tasks has required only one kind of specification. Here, however, we have a single task that requires at least two specifications. There is strong coupling between these specifications, however, so that a change to one will often involve a change to the other. The solution is to combine such related specifications into a single file of type '`fw`'. A type-'`fw`' file describes a set of specification files that Eli will separate as necessary, but which the user can manipulate as a single entity.

The variable atoms of the generated C code are integers (like the number of elements in a set) and strings (like the elements of a set). `LeafPtg` (see Section 'PTG Output of Leaf Nodes' in MODLIB – Specification Module Library) is a generic module that provides operations useful in writing such atoms, so it is instantiated in file '`sets.specs`':

```
$/Output/LeafPtg.gnrc :inst
```

Three related specifications are used to describe the creation of the C declarations. First the general form of the declarations is given in a special language called *PTG*, then there is an attribute grammar fragment that computes the individual components, and finally two C macros are needed to implement operations used in the computations. All three specifications are combined in a single type-'`fw`' file called '`code.fw`':

```
@O@<code.ptg@>@{
Table:
  "int number_of_sets = " $/*integer*/ ";\n\n"
  "char *name_of_set[] = {\n"
  $/*list of set names*/ "};\n\n"
```

```
  "int size_of_set[] = {\n"
  $/*list of set sizes*/ "};\n\n"
  $/*list of sets*/
  "char **values_of_set[] = {\n"
  $/*list of set representations*/ "};"

Set:
  "char *set_of_" $/*set name*/ "[] = {\n"
  $/*list of set elements*/ "};\n\n"

Seq:
  $ $

List:
  $ ",\n" $

Quoted:
  "\"" $ "\""

Name:
  "set_of_" $
@}

@O@<code.lido@>@{
ATTR Ptg: PTGNode;
SYMBOL Entity INHERITS IdPtg END;

SYMBOL text COMPUTE
  IF(NoErrors,
    PTGOut(
      PTGTable(
        PTGNumb(CONSTITUENTS set_name.Sym WITH (int, ADD, ARGTOONE, ZERO)),
        CONSTITUENTS set_name.Ptg WITH (PTGNode, PTGList, PTGQuoted, PTGNULL),
        CONSTITUENTS set_body.Size WITH (PTGNode, PTGList, PTGNumb, PTGNULL),
        CONSTITUENTS set_def.Ptg WITH (PTGNode, PTGSeq, IDENTICAL, PTGNULL),
        CONSTITUENTS set_name.Ptg WITH (PTGNode, PTGList, PTGName, PTGNULL))));
END;

ATTR Size: int;
SYMBOL set_body COMPUTE
  SYNT.Size=CONSTITUENTS set_element.Sym WITH (int, ADD, ARGTOONE, ZERO);
END;

SYMBOL set_def COMPUTE
  SYNT.Ptg=
    PTGSet(
      CONSTITUENT set_name.Ptg,
      CONSTITUENTS set_element.Ptg WITH (PTGNode, PTGList, PTGQuoted, PTGNull));
END;
@}
```

```
@O@<code.HEAD.phi@>@{
#include "err.h"
#define NoErrors (ErrorCount[ERROR]==0)
@}
```

The PTG specification, which is introduced by `@O@<code.ptg@>@{` and terminated by `@}`, is simply a collection of parameterized templates for output. Each template is given a name, and consists of a sequence of items that will be output in the given order. Quoted C strings are output as they stand, and each `$` stands for one parameter. A text fragment is constructed according to a particular template by invoking a function whose name is `PTG` followed by the template name. This function returns a value of type `PTGNode`, and must be provided with one argument of type `PTGNode` for each parameter.

To construct a text fragment according to the template named `Quoted`, for example, invoke `PTGQuoted` with one argument of type `PTGNode`. The result will be a value of type `PTGNode` that describes the text fragment `"`, followed by the text fragment described by the argument, followed by `"`.

The attribute grammar fragment, which is introduced by `@O@<code.lido@>@{` and terminated by `@}`, invokes the PTG functions with appropriate arguments. `PTGNumb` and `PTGName` are defined by the `LeafPtg` module (see Section 'PTG Output of Leaf Nodes' in MODLIB – Specification Module Library). They construct values of type `PTGNode` that describe the text fragment consisting of a single integer value or word respectively. These text fragments are then used in building larger text fragments, and so on.

The translated output should be produced only if no errors were detected by the translator. `NoErrors` is a user-defined macro that tests whether any reports of severity `ERROR` were issued. `NoErrors` must be defined as a C macro, in a file of type '`.HEAD.phi`'. This can be done by a segment of the FunnelWeb file introduced by `@O@<code.HEAD.phi@>@{` and terminated by `@}`.

The error module maintains an array `ErrorCount`, indexed by severity, each of whose elements specifies the number of reports issued at the corresponding severity (see Section 'Source Text Coordinates and Error Reporting' in Library Reference Manual). If the `ERROR` element of this array is `0`, then no reports of severity `ERROR` have been issued.

In addition to building the C declarations, the attribute grammar fragment computes the total number of sets and the total number of elements in each set:

```
ATTR Size: int;
```

```
SYMBOL set_body COMPUTE
  SYNT.Size=CONSTITUENTS set_element.Sym WITH (int, ADD, ARGTOONE, ZERO);
END;
```

`ADD`, `ARGTOONE` and `ZERO` are built-in functions of Eli (see Section 'Predefined Entities' in LIDO Reference Manual).

This computation visits nodes of the subtree rooted in the `set_body` node. If the node is a `set_element` node, function `ARGTOONE` is applied to the value of the `Sym` attribute to yield the integer value 1. If the node has no `set_element` descendants, then its descendants are not visited and the function `ZERO` is invoked with no arguments to yield the integer value 0. Otherwise the integers computed for the children of the node are combined pairwise in left-to-right order via the function `ADD`. See LIDO Reference Manual.

Exercises

1. Request the code generated by the processor from the specification:

   ```
   -> sets.specs :gencode :viewlist
   ```

   Find the files generated from `code.fw` and verify the content of `code.HEAD.phi`. (Hint: Reread the discussion of `code.fw`.)
2. Which file contains the definition of the function `PTGQuoted`? (Hint: Use grep, or create a '`tags`' file and then give the command `vi -t PTGQuoted`.)
3. Briefly explain the operation of `PTGQuoted`. How is the text fragment created by this function printed?
4. Use `lint` to search for anomalies in the collection of C routines. Are any of those found significant? Explain briefly.
5. Request an interactive debugging session:

   ```
   -> sets.specs +debug :dbx
   ```

   (If you prefer to use the GNU debugger gdb, simply replace `dbx` with `gdb`).

   Set breakpoints to stop the program in `PTGQuoted`, `_PrPTGQuoted` and `PTGOut`. Run the program with the name of file '`input`' as its command line argument. What is the order of the calls to these three routines? Explain briefly.
6. Redefine the output text so that each array value is indented, and the closing brace is at the beginning of a line. For example, the array of set names should look like this:

   ```
   char *name_of_set[] = {
           "colors",
           "bugs",
   ```

```
              "verbs"
      };
```

Generate a translator and verify its correctness.

3 Customizing Eli's Behavior

All derived objects are stored in a directory called the *derived object cache*, or simply the *cache*. The cache also contains a database that stores the *depends* relationship between the output and input files of a tool run, and the *contains* relationship between a list and its elements. Many of the ways of customizing Eli involve various aspects of the derived object cache.

Eli can also be used non-interactively, and can be customized by defining shortcuts for frequently-used derivations.

3.1 Hints on Cache Management

The default location for the cache is a directory named `.ODIN` in the user's home directory. A non-default cache location can be specified by the `$ODIN` environment variable, or with an option on the command line. The main reasons for specifying a non-default location for the cache are to share a common cache with other users, or to locate the cache on a local disk for more efficient access to derived files.

An Eli session is begun by giving the following command:

```
eli [ -c cache ] [ -r | -R ]
```

All of the command line arguments are optional, and all affect the cache:

`-c` *cache*
: Use the directory *cache* as the cache.

`-r`
: Reset the cache. This deletes all derived objects currently stored in the cache.

`-R`
: Reset the cache and upgrade all tools. This deletes all derived objects currently stored in the cache, and also installs the most recent versions of all tools.

Cache directories may also be deleted using normal Unix commands whenever they are not being used by active Eli sessions. If the specified cache does not exist when the `eli` command is given, then it will be created and the most recent versions of all tools installed.

There is no limit to the number of cache directories that may exist at one time. You might choose to have a separate cache for each project you are working on, or you might choose to have a single cache to hold information for all of your projects. If you choose multiple caches, each can be smaller than the cache you would use for all projects. When a project is complete, you can delete all the intermediate objects relating to it by deleting the cache directory for that project.

Cache contents are architecture dependent, so it is not possible to create a cache on one architecture and then use that same cache on a different architecture. In order to avoid this error, Eli creates a separate subdirectory of the cache directory for each *host* (not architecture) on which it is invoked. This behavior is unpleasant in a setting where there is a pool of hosts, all of which have the same architecture, running with a common file server. If the environment variable `ODINVIEW` is set, Eli uses the subdirectory name specified by that variable. (The subdirectory names can be anything; using the host name is simply convenient.)

The default inter-process communication mechanism for the odin cache manager process is TCP/IP. If TCP/IP is not available, set the environment variable `$ODIN_LOCALIPC` and Unix domain sockets will be used instead.

3.2 Hints on Session Management

There are two kinds of Eli sessions – interactive and non-interactive. Interactive sessions are used when the requests being made are ones that Eli can satisfy quickly, and actions by the user are necessary between requests. During initial development of a specification, when specification errors prevent Eli from completely satisfying the request, interactive sessions are very fruitful: The user makes a request, errors are reported, the user corrects the errors and makes the request again.

One important decision that must be made for either kind of session is the amount of information that should be provided to the user during that session. (Of course if the session is interactive, this decision can be changed during the session itself by making appropriate requests.) Eli is capable of describing at great length what it is doing at any given moment. Since the purpose of Eli is to suppress the details of the process needed to satisfy your request, you will probably not want Eli to report those details to you. The Eli variable `LogLevel` controls the level at which Eli describes the actions that it is taking. The default is `LogLevel=2`. For more information about the effect of different `LogLevel` values, give the Eli request `LogLevel=?`. (This is an example of an Eli help request, described in Section 'The Help Facility' in User Interface.)

The value of the environment variable `EDITOR` at the time the Eli session starts is the command that is invoked when the character `<` ends an input line. (If `EDITOR` is not defined when the Eli session starts then `vi` is assumed.) That value can be changed at any time during the session by assigning to the environment variable `EDITOR`:

```
-> EDITOR=!emacsclient
```

(Note the use of `!` to indicate that the assignment is to an environment variable rather than to an Eli variable.)

You may wish to make your selection of an editor dependent on some property of the environment. A typical situation is to use one editor when seated at a workstation and another when logged in remotely. In this case, create a script '`my_editor`' that tests the appropriate environment variables, decides what editor to use, and invokes it. Then set the value of the environment variable `EDITOR` to '`my_editor`'.

Users of Gnu Emacs who invoke Emacs only once per login session (i.e. in a window that is always present) can use the server capability of Emacs. To do this, execute the command `M-x server-start` in your Emacs session and use `emacsclient` as the value of the environment variable `EDITOR`. (You will also need to make sure that the `etc` directory in your Emacs distribution is on your `PATH`.) Once this is done, Eli editor invocations will use buffers in your Emacs session. A common way of utilizing this capability is to invoke Eli from a sub-shell of your Emacs (created using `M-x shell`).

Eli consults file '`Odinfile`' in the current directory for information about the task at hand. '`Odinfile`' is used to define one or more *targets*. Each target defines some product that can be requested, using the notation *target* `==` *odin-expression*. Here are examples of the three common kinds of target:

`mkhdr == sets.specs :exe`

: `mkhdr` is a *file target*. This line specifies that `mkhdr` should always be equal to the derived file object `sets.specs :exe`. If the command `eli mkhdr` is given in a directory with a file '`Odinfile`' containing this line, it will result in a non-interactive Eli session guaranteeing that file `mkhdr` in this directory is up to date. (The same effect can be obtained in an interactive session by responding to the `->` prompt with `mkhdr`.)

`%results == input +cmd=(mkhdr) :stdout`

: `%results` is a *virtual target*. A virtual target is simply a name for an odin-expression, and can be used wherever and odin-expression is required. If the command `eli '%results>'` is given in a directory with a file '`Odinfile`' containing this line, it will result in a non-interactive Eli session guaranteeing that the derived object `input +cmd=(mkhdr) :stdout` is up to date, and writing the content to the standard output. (The same effect can be obtained in an interactive session by responding to the `->` prompt with `%results>`.)

`%test ! == . +cmd=diff (%results) (result) :run`

`%test` is an *executable target*. An executable target is a target that is executable. If the command `eli %test` is given in a directory with a file '`Odinfile`' containing this line, it will result in a non-interactive Eli session guaranteeing that the derived object `input +cmd=(mkhdr) :stdout` (named `%results`) is up to date, and executing the `diff` command with this object and the file '`result`' from the current directory as arguments. Execution will take place in the current directory. (The same effect can be obtained in an interactive session by responding to the `->` prompt with `%test`.)

4 System Documentation

The Eli system documentation is divided into three basic groups:

Tutorial
: Strategies and examples for using Eli. The purpose of this material is to present simple techniques that work. Only points that we have found important for most users are covered.

Reference
: Detailed definitions of notation and behavior. The purpose of this material is to answer any question that might arise. There is a reference manual for each of the notations understood by Eli, including the language in which requests for processor construction are made. All of the products that can be requested, and all of the parameters that can be used to modify those requests, are the subject of a separate reference manual. Finally, there is a reference manual for the on-line documentation browser.

Administration
: Strategies for installing, configuring and maintaining Eli. The purpose of this material is to guide the person responsible for Eli at a particular installation.

All of the documentation is available both on-line and in printed form. Documents are stored on line as hypertext, and can be used to support the debugging phase of a project.

4.1 How On-line Documentation Supports Debugging

Two levels of debugging are necessary when using Eli:

1. The specifications you present to Eli may be inconsistent or ill-formed. In that case, Eli will provide error reports in the same way as any compiler. You must correct the specifications so that they are well-formed and consistent.
2. You have presented a correct specification to Eli, but this specification describes the wrong problem instance. Now you must determine how the problem instance you have described differs from the one you are really interested in, and change the specification accordingly.

On-line documentation for Eli can only provide support for level (1), because level (2) does not involve symptoms that can be diagnosed by Eli.

Eli presents error reports to a user only on request. The available requests are described in Section 'Diagnosing Specification Inconsistencies' in Products and Parameters Reference Manual.

One of these requests is `:help`. This request builds a new hypertext subtree containing the error reports, embedded in the text to which they refer. The files containing the errors are made accessible to the nodes describing those errors, so that the user can correct them directly.

To correct a file, move the browser to the node describing the errors in that file. Execute the browser's edit command and make whatever changes are necessary. Then exit the editor.

Error reports are also linked to the nodes of the on-line documentation describing the constructs in which the errors were detected. Thus the user is placed in an environment in which all of the information needed to diagnose the errors, and the tools needed to correct them, are immediately at hand.

Index

F

G

H

I

K

L

M

N

O

P

Q

R

S

T