Tasks Related to Input Processing

W. M. Waite

University of Colorado
Boulder, Colorado 80309
USA

This manual is for the type analysis module library

Table of Contents

1 Eli Input Text

An Eli-generated processor treats its input as a single body of text. The default command line interface provided by Eli assumes that the generated processor will have one input file and no options. If no file name is specified on the command line, `stdin` is used as the input file. A developer can change the default command line interface by including specifications written in the CLP language (see Section "Specifying the command line interface" in *Command line processing*).

The input text consists of a sequence of *lines*, each terminated by a *newline* character. Lines are numbered, beginning with 1. In the simplest case, these lines are the content of a single file. In general, however, they may be drawn from an arbitrary collection of files, depending on the goals of the processor. For example, a C compiler will need to replace `#include` directives with the contents of the specified files. A Java compiler may need to combine a number of files describing classes into a single program. In this document, we explore ways of dealing with such situations using the `Include` module.

Error reporting in an Eli-generated processor is based on the line number of the input text in which the error occurred (see Section "Error Reporting" in *Frame Library Reference Manual*). If the text has been drawn from many files, that error report must be mapped to the corresponding line in the file containing the original text. The `CoordMap` module maintains a coordinate system relating the input text to its origins in the file system. This module is automatically instantiated when `Include` is used. (`CoordMap` is not intended to be used directly in specifications.)

2 Insert a File into the Input Text

Some processors need to insert the content of an arbitrary file at a specific point in the input text currently being processed. For example, consider the C preprocessor `cpp`, where the file inclusion command `#include "myfile"` inserts the content of file `myfile` at the position of that command in the input text. The Eli `Include` module supports such a requirement. It can be instantiated without generic parameters by

```
$/Input/Include.gnrc :inst
```

The `Include` module exports the token processor `InclDirective` (see Section "Token Processors" in *Lexical analysis specification*), allowing the developer to implement a file inclusion command as a lexical token (see *Lexical analysis specification*).

To implement the `cpp` file inclusion command as a lexical token, we add a scanner specification (see Section "Specifications" in *Lexical analysis specification*) to a type-`gla` file:

```
$#include\040+\"  (auxEOL)  [InclDirective]
```

This scanner specification begins with the regular expression `#include\040+\"`, which matches the beginning of the command (see Section "Regular Expressions" in *Lexical analysis specification*). This regular expression requires at least one space to follow the word `include`, but allows more spaces to precede the string giving the file name. The last character of the regular expression is the delimiter of the file name string.

The auxiliary scanner `auxEOL` (see Section "Auxiliary Scanners" in *Lexical analysis specification*) advances the input pointer beyond the end of the line of text containing the command (see Section "Available scanners" in *Lexical analysis specification*). This advance is vital, since the input file can only be changed at a line break.

The token processor `InclDirective` (see Section "Token Processors" in *Lexical analysis specification*) is exported by the `Include` module. It first extracts `mark`, the last character matched by the regular expression (in our case, `mark` is `"`). If `mark` is `<`, `InclDirective` assumes that the form of the string is `<...>`; otherwise it assumes that the string is delimited by `mark` characters.

Given the above assumption, `InclDirective` scans the text following the text matching the regular expression for the terminating delimiter. When the scan reaches a space, the terminating delimiter, or the end of the line, the file name is taken as the sequence of scanned characters up to, but not including, that space, terminating delimiter, or end-of-line. The module produces a warning if the terminating delimiter is end-of-line.

Finally, `InclDirective` seeks the specified file and switches the input pointer to that file, if it can be opened. When the end of that file is reached, the input pointer returns to the file containing the file inclusion command, at the beginning of the line following that command. If the specified file cannot be opened, an operating system report is issued and further input is take from the current file.

Recall that the lexical token was specified as:

```
$#include\040+\"  (auxEOL)  [InclDirective]
```

This specification identifies the lexical token as a *comment* rather than a *basic symbol* (see *Lexical analysis specification*). Because it is a comment, it can be used wherever a comment is allowed in the source text. It may be that the developer wishes to restrict file insertion

to specific grammatical constructs. In that case, the lexical token should be identified as a basic symbol by giving it a name:

```
Insertion: $#include\040+\"  (auxEOL)  [InclDirective]
```

The basic symbol name `Insertion` can then be used in the grammar just as any basic symbol name. Incorrectly-placed insertion points will be reported as syntax errors, but the contents of the files specified by the insertion commands will still be inserted at the specified points.

3 Specifying Include Directories

The `Include` module defines command line parameters to specify directories that might contain files to be included. It does this by exporting the following CLP specification (see Section "Specifying the command line interface" in *Command line processing*):

```
IncludeDirs "-I" joinedto strings
   "Directories to search for included files";
Source input
   "Primary input file";
```

The general form of a command line that can be recognized using this CLP specification is:

1. A program name, followed by
2. An arbitrary number of `-I` options, followed by
3. A single file name.

For example, suppose that the developer uses Eli to generate a processor called `DemoProc`. `DemoProc` accepts text written in a language called `Demo`, and `Demo` defines inclusion commands. A user has a `Demo` text file `DemoText` that might contain inclusion commands for files found in the directories `subDir` and `/usr/local/etc`. Here is the appropriate command line:

```
DemoProc -IsubDir -I/usr/local/etc DemoText
```

If a directory name does not begin with the character `/`, then that name is considered to be relative to the current directory. Thus, in this example, `subDir` must be a subdirectory of the current directory and `/usr/local/etc` is probably a system directory.

Please note that the CLP specification introduced by the `Include` module contains an input parameter (see Section "Input parameters" in *Command line processing*). If the `Include` module is instantiated in a context containing its own CLP specification with an input parameter, the input parameter in that CLP specification must be deleted. If the context contains any reference to the `name` of the deleted input parameter, that reference must be changed to reference `Source`.

4 Multiple files on the Command line

A sequence of input files can be specified by positional parameters on the command line if the `Include` module is instantiated as follows:

```
$/Input/Include.gnrc +referto=positionals :inst
```

In this case, the CLP specification exported by the `Include` module will be (see Section "Specifying the command line interface" in *Command line processing*):

```
IncludeDirs "-I" joinedto strings
   "Directories to search for included files";
Source input
   "Primary input file";
Sources positionals
   "Additional input files named on the command line";
```

The general form of a command line that can be recognized using this CLP specification is:

1. A program name, followed by
2. An arbitrary number of `-I` options, followed by
3. One or more file names.

Here is an example:

```
DemoProc -IsubDir -I/usr/local/etc DemoText AddedText
```

The effect of this command line is that the input text to `DemoProc` consists of the content of `DemoText` (with all inclusion commands expanded) followed by the content of `AddedText` (with all inclusion commands expanded). The CLP specification permits an arbitrary number of additional file names (see Section "Positional parameters" in *Command line processing*).

Do not specify additional positional parameters in type-`clp` files when using the `Include` module. If it is necessary to provide file names on the command line for purposes other than text input, use options to specify those names (see Section "Value options" in *Command line processing*).