New Features of Eli Version 4.8

Uwe Kastens

University of Paderborn
D-33098 Paderborn
FRG

A. M. Sloane

Department of Computing
Division of Information and Communication Sciences
Macquarie University
Sydney, NSW 2109
Australia

W. M. Waite

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

$Date: 2013/04/07 00:23:42 $

Table of Contents

New Features of Eli Version 4.8

This document gives information about new facilities available in Eli version 4.8 and those modifications made since the previous distributed Eli version 4.7 that might be of general interest. Numerous corrections, improvements, and additions have been made without being described here.

1 Refactored top level

The top level of the Eli system has been refactored to combine the primary input file open operation with the tree-building operation. With this refactoring, the driver of an Eli-generated processor first invokes a routine to parse the command line (see Section "top" in *Command Line Processing*). It then executes any code supplied via '`'m'.init`' files (see Section "Implementing Tree Computations" in *LIDO – Computations in Trees*).

At this point, the driver invokes the `TREEBUILD` routine. `TREEBUILD` opens the primary input file, and uses the generated scanner and parser to build the abstract syntax tree. If there were no reports above the `WARNING` severity level (see Section "Source Text Coordinates and Error Reporting" in *Library Reference*), then the driver invokes the `ATTREVAL` routine to perform the specified computation over the constructed tree. Finally, any code supplied via '`'m'.finl`' files is executed (see Section "Implementing Tree Computations" in *LIDO – Computations in Trees*).

This refactoring allows one to replace `TREEBUILD` with any code that builds a tree conforming to some LIDO definition (see Section "Tree Construction Functions" in *LIDO - Reference Manual*). That code will usually have its own input module, and may or may not access command line parameters (see Section "Accessing the command line" in *Command Line Processor*).

2 Support for parsers not generated by Eli

Eli has the ability to generate a complete text processor, including all of the tree computation needed for contextual analysis. It assumes, however, that the input language can be described by a reasonably consistent grammar. This is not the always the case, even for programming languages, and it may be that more ad-hoc methods are needed to construct a tree that describes the source text.

There are many tools other than Eli that one can use to create processors that scan and parse text, and they differ among themselves in strategy and power. All support mechanisms to build trees on the basis of the relationships implicit in the input text. Once the tree is built, however, most systems provide no further aid. The user is responsible for writing code in C or Java to process and transform the tree.

Eli now has the ability to interact with a scanner/parser developed using any arbitrary technology (see Section "Using Foreign parsers" in *Syntactic Analysis*). For example, the "foreign" analyzer might be a collection of C or C++ routines that could be defined by a '`specs`' file and invoked by the main program that Eli generates. Alternatively, it might be a main program that invokes a shared library to build and process the tree. In either case, Eli can generate code to interact with it and automate the tedious job of constructing tree computations.

3 Shared library product

The `:exe` product is the executable file derived from the specifications, to be run on the current machine (see Section "exe – Executable Version of the Processor" in *Products and Parameters*). When Eli-generated code is only a part of a program created by other means, it may be convenient to encode it as a shared library. This strategy could be used, for example, to incorporate Eli-generated code into a Java program via the Java Native Interface.

The `:so` product is a shared library file derived from the given specifications (see Section "so – Shared library Version of the Processor" in *Products and Parameters*). A shared library should not have a main program, and therefore it should be derived with the `+nomain` opition (see Section "nomain – Omitting the main program" in *Products and Parameters*).

When a shared library is used, the program using it normally requires that the shared library file have a specific name. Thus the `:so` product should be copied out of the cache into a file with the appropriate name (see Section "Extracting and Editing Objects" in *User Interface*). For example, suppose that we were deriving from '`MyProc.fw`' and the required name for the shared library file was '`YourLib.so`'. In that case, an appropriate derivation request might be:

```
MyProc.fw +nomain :so > YourLib.so
```

4 Reporting file opening errors

The format of a processor's command line can be specified in clp (see Section "Command Line Processor" in *Command Line Processor*). In addition to defining options and parameters, the designer can specify the form of an error report to be used when a file appearing on the command line cannot be opened.

In earlier versions of Eli, this message was only written when the processor could not open its primary input file. As a part of the refactoring to support foreign parsers, we defined a routine `ClpOpenError` to output the report on request (see Section "Reporting open errors" in *Command Line Processor*).

This report will be automatically made if the processor cannot open the file specified on the command line as its primary source of data (see Section "Input parameters" in *Command Line Processor*). User code can also make the report if it encounters an error while opening an arbitrary file.

Index