New Features of Eli Version 4.1

Uwe Kastens

University of Paderborn
D-33098 Paderborn
FRG

A. M. Sloane

Department of Computing
School of Mathematics, Physics, Computing and Electronics
Macquarie University
Sydney, NSW 2109
Australia

W. M. Waite

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

$Revision: 4.4 $

Table of Contents

This document gives information about new facilities available in Eli version 4.1 and those modifications made since the previous distributed Eli version 4.0 that might be of general interest. Numerous corrections, improvements, and additions have been made without being described here. They shall just help users to solve their problem without taking notice of Eli's mechanism.

1 LIDO Language and Liga System

The modifications of the Lido language and the Liga system refer to the following topics: The use of symbol computations is further simplified by removal of certain restrictions, and by introduction of some new constructs. Some restrictions on bottom-up evaluation are removed. A mechanism has been installed such that messages given by the C compiler on the generated evaluator are related back to the Lido text.

None of the modifications invalidates existing Lido specifications.

For complete descriptions of the modified constructs see the Liga documentation: See Section "top" in *LIDO – Reference Manual*. See Section "top" in *LIDO - Computation in Trees*.

1.1 CONSTITUENTS Restrictions Removed

If a symbol computation contains a CONSTITUENTS construct, like

```
CLASS SYMBOL CS COMPUTE
  SYNT.c = f (CONSTITUENTS X.a WITH (t, f2, f1, f0));
END;
```

it was not allowed to inherit CS to a symbol that has a production with an empty right-hand side. It was also a violation if X were a CLASS symbol that is not inherited to any symbol. This was sometimes annoying, especially when using library modules.

Both restrictions have been removed. The result of the CONSTITUENTS is in such cases the result of the `f0`-function. A VOID CONSTITUENTS is replaced by nothing in such cases.

See Section "CONSTITUENT(S)" in *LIDO – Reference Manual*.

1.2 CHAIN in Empty Rules

It was a violation if a computation containing a TAIL chain access is inherited to a symbol that has a production with an empty right-hand side. A HEAD computation was dropped in such cases. This was sometimes annoying, especially when using library modules. This restriction has been removed.

Now, if a computation that contains a CHAINSTART, HEAD, or TAIL is inherited to a rule with an empty right-hand side, its effect is as if there was a symbol on the right-hand side that passes the chain unchanged.

See Section "CHAIN" in *LIDO – Reference Manual*.

1.3 Grammar Root Symbol

The

```
CLASS SYMBOL ROOTCLASS END;
```

is predefined without any computation. It is inherited to the particular root symbol of the tree grammar.

This facility is especially useful for writing grammar independent modules which need to associate computations to the grammar root. Module users then do not need to explicitly inherit a root role.

See Section "Predefined Entities" in *LIDO – Reference Manual*.

1.4 Right-Hand Side Access

The entities `TermFct`, `RhsFct`, and `RULENAME` are predefined to simplify symbol computations that access the right-hand side of rules they are inherited to.

See Section "Predefined Entities" in *LIDO – Reference Manual*.

A function `TermFct` is predefined to be used for systematic access of terminal values in computations. It is intended to be used in computations of `CLASS SYMBOL`s:

```
CLASS SYMBOL LeafNode COMPUTE
  SYNT.Ptg = TermFct ("ToPtg", TERM);
END;
SYMBOL LiteralExpr INHERITS LeafNode END;
```

If there were the following two rules for `LiteralExpr` that derive to different terminals, then the above computation is expanded as shown below:

```
RULE: LiteralExpr ::= IntNumber COMPUTE
  LiteralExpr.Ptg = ToPtgIntNumber (IntNumber);
END;
RULE: LiteralExpr ::= FloatNumber COMPUTE
  LiteralExpr.Ptg = ToPtgFloatNumber (FloatNumber);
END;
```

Suitable functions have to be defined for the calls constructed by prefixing the terminal name with the string given in the `TermFct` call.

A call of the predefined function `RhsFct (C_String, arguments ...)` is substituted by a call of a function whose name is composed of the `C_String` and and two numbers that indicate how many nonterminals and terminals are on the right-hand side of the rule that has (or inherits) this call. The remaining arguments are taken as arguments of the substituted call. E.g. in a rule `RULE: X ::= Id Y Id Z Id END;`, where `Y, Z` are nonterminals, and `Id` is a terminal, a call `RhsFct ("PTGChoice", a, b)` is substituted by `PTGChoice_2_3 (a, b)`. Usually, `RhsFct` will be used in symbol computations, having arguments that are obtained by the `RHS` construct and by a `TermFct` call.

`RULENAME` can be used in computations. It is replaced by the rule name as a string literal.

1.5 Bottom-up Evaluation

Liga's strategy for scheduling `BOTTOMUP` computations has been changed: The generated evaluator performs computations during the tree construction phase if and only if there are some computations marked `BOTTOMUP`.

See Section "Computations" in *LIDO – Reference Manual*.

Requesting `BOTTOM_UP` or `TREE_COMPLETE` in a `.ctl` file is now unnecessary and meaningless.

See Section "Order Options" in *LIGA - Control Language*.

Attributes are computed during the tree construction phase only if they are needed for `BOTTOMUP` computations. That strategy reduces the size of the tree in general.

Information about the results of Liga's analysis for `BOTTOMUP` can be obtained by deriving `OrdInfo`.

The restrictions on `BOTTOMUP` computations have been relaxed: The facility of subtrees being built by computations may be used together with some other computations being marked `BOTTOMUP`, unless computations in such generated subtrees are preconditions for `BOTTOMUP` computations.

Furthermore, chain productions may be introduced into the tree grammar, if necessary, without having corresponding chain productions in the concrete grammar, as long as they are not involved in `BOTTOMUP` computations. This situation is automatically checked in cooperation between Liga and the Maptool.

2 Specification Module Library

Many library modules provide some symbol role, e.g. `RootScope`, that has to be inherited to a tree grammar symbol which is usually the root of the tree grammar. These moduls are modified such that those root roles are automatically associated to `ROOTCLASS`, which stands for the tree grammar root. See Section 1.3 [Grammar Root Symbol], page 3.

All name analysis modules have been adapted to the modifications of the environment module. These changes should not invalidate existing uses of the modules.

See Section "Name Analysis Library" in *Specification Module Library: Name Analysis*.

2.1 Environment Module Enhancements

The environment module `envmod.[ch]` has been augmented by functions and macros that further support name analysis for object-oriented languages, i.e. name analysis with scopes that inherit from other scopes.

The module was also augmented by functions that return a binding instead of a definition table key. A binding is a triple `(int idn, Environment sc, DefTableKey key)`, for an identifier `idn` that is bound to `key` in the scope `sc`.

A full description of the interface of the module is given in the name analysis part of the module library documentation. (It has been moved there from its previous place in the documentation of Eli library routines.).

See Section "Environment Module" in *Specification Module Library: Name Analysis*.

Existing uses of the module should not be invalidated by these changes.

2.2 Module for Testing Name Analysis

A module is provided which augments the specified processor such that it produces output that makes the results of name analysis visible. For each identifier occurrence that has one of the identifier roles of the name analysis modules a line of the form

```
m in line 23 bound in line 4 of scope in line 3
```

is written to the standard output file. In general it is sufficient just to instantiate the module `ShowBinding` with the same instance parameter as used for the basic name analysis module.

See Section "Name Analysis Test" in *Specification Module Library: Name Analysis*.

2.3 Module `ChainPtg` obsolete

Module `ChainPtg` was obsolete with Version 3.6 of the Eli-System. This module has now been removed without replacement.

To collect PTG-Nodes for output, use `CONSTITUENTS`-Construct in Combination with the Pattern `Seq` of the new Module `PtgCommon`.

See Section "Using LIDO CONSTITUENTS" in *Pattern-Based Text Generator*, for details.

2.4 Module `Sort` added

A generic sorting module has been added to the library. This module can be instantiated for any data type, and sorts an array whose elements are of that data type. A user-supplied function defines the collating sequence, so that any arbitrary ordering is possible. The sort is done in place, so that the array after the sort is a permutation of the array before the sort.

See Section "Sorting Elements of an Array" in *Specification Module Library: Common Problems*, for details.

2.5 Module `Separator` added

This is a new PTG output module that allows separators to be inserted into the output stream depending on the last string printed and the next string to be printed.

See Section "Introduce Separators in PTG Output" in *Specification Module Library: Generating Output*, for details.

2.6 Instanciation of Modules with Filenames

To instanciate modules, that require filenames as their arguments from within a `.fw`-files previously required specification of the `.fw`-file generating it. This is now no longer required.

For example, to instanciate the `PreDefId`-Module with a filename `predef.d`, the following example now works within and outside from a `.fw` specification:

```
$/Name/PreDefId.gnrc +referto=(Predef.d) :inst
```

See Section "Predefined Identifiers" in *Specification Module Library: Name Analysis*, and Section "Operator Identification" in *Specification Module Library: Type Analysis*, for more information.

3 Command-line processing

The command-line processing tool has been rewritten to use a more flexible implementation technique. In the process some erroneous situations that weren't detected before are now detected.

Most notably, in previous versions when a positional parameter was specified no usage message was printed if the user did not provide a value for the parameter. This has been fixed.

Repeated boolean options are now a count of the number of times the option appears rather than a list of keys each of which have the value 1. This is a potential source of incompatibility with previous versions.

The new version also provides some new features, described in the following paragraphs.

Previously it was possible to have an integer or string value that was either the next argument or was joined to the option string. I.e., the following were possible

```
-#23
-m foo
```

It is now possible to have an option where the value can be either the next argument or can be joined to the option string. This is indicated by the keyword "with".

```
MacroPackage "-m" with strings "Load this macro package";
NumCols "-C" with int "Use this many colours";
NumZaps "-z" with ints "Zap this many times";
```

It is now possible to have more than one option string that invokes a particular option. Just list more than one string.

```
MacroPackage "-m" "mac" with strings "Load this macro package";
NumZaps "-z" "-Zap" "whammo" ints "Zap this many times";
```

Empty .clp files are now acceptable. The resulting processor will accept standard input, but nothing on the command line.

4 Monitoring

The Noosa system (invoked using the `:mon` product has undergone some major changes since the last release of Eli. Numerous small changes have been made to the user interface but the general appearance is the same.

The default font is not specified by Noosa any more. You now get whatever your Tk setup gives you by default, but you can set it yourself using the Noosa*Font resource in your .Xdefaults. More support for resources will be provided in future releases.

The previous version displayed the complete abstract syntax tree drawn in a conventional fashion. The new version also includes a tree display that allows a partial view of the tree and takes up much less space. It is designed for browsing of the tree and nodes can be selectively opened and closed. You can select which of the kinds of tree display you want using the Abstract Tree item in the Windows menu.

Eli and Noosa now have support for attribution monitoring so the nodes in the tree displays provide access to the node attributes. You can elect to see the values of attributes; they are displayed in the transcript window when next computed. Optionally you can also make the program stop when the value of an attribute is computed (a form of breakpoint).

The Windows menu has a new entry called Files. This brings up a window from which it is possible to display and edit files while you are monitoring (e.g., to fix bugs in your specs or to alter the process input). The Open menu item in the Noosa menu is no longer provided since its functionality is subsumed by the new window type. You can have as many file windows as you like. They support emacs-style key bindings and support searching.

Sensitive areas in the transcript are now always underlined and you just have to click on them with B1 (a'la netscape) to "open" them. What "opening" means depends on the kind of value. In the current version the sensitive areas are coordinates (or coordinate ranges), abstract tree nodes (Nodes), and pattern-based text generator nodes (PTGNodes). Opening a coordinate (or range) highlights that coordinate (or range) in the input window. Opening an abstract tree node highlights the node in an abstract tree window (if there are any). Opening a PTGNode causes the text expansion of that node to be printed at the bottom of the transcript window. Future versions of Noosa will support opening other types of values such as environments and definition table keys.

In this version of Eli the :mondbx and :mongdb products are not operational due to the new Noosa implementation. This situation will be remedied as soon as possible with the fixes being made available via an Eli patch.

5 Abstract Syntax Tree Unparsing

Parsing is the process of constructing a tree from a string of characters; *unparsing* is the reverse: constructing a string of characters from a tree. An Eli user can specify an arbitrary unparser by means of a combination of attribute computations and PTG (see *PTG: Pattern-Based Text Generator*) templates. For a large tree, this can be a tedious process.

Given a specification of the set of rules defining the tree, Eli can now generate the combinations of attribute computations and PTG templates needed to produce certain common unparsings. This information can be extracted and modified, or it can be automatically merged with the remainder of the specification to produce the output routines for the generated processor.

See *Abstract Syntax Tree Unparsing*, for details.

6 FunnelWeb Output Specification

A *non-product* output file, named by `@N`, is now available. A non-product output file is identical to a normal output file except that it is not included in the set of files making up the final product specification.

Non-product files are files that are used in the derivation of product components, but are not themselves components of the product. For example, consider the problem of making keywords case-insensitive but retaining case sensitivity in identifiers (see Section "Making Literal Symbols Case Insensitive" in *Lexical Analysis*). Here is a portion of a FunnelWeb file implementing such a processor:

```
@O@<nolit.gla@>==@{
identifier: C_IDENTIFIER
@}

@N@<keyword.gla@>==@{
$[a-z]+
@}

@O@<keyword.specs@>==@{
keyword.gla :kwd
@}
```

Note that the file '`keyword.gla`' can *not* form part of the final product specification. If it did, the specified processor would treat all completely lower case identifiers as comments! Nevertheless, file '`keyword.gla`' is necessary to specify the representation of the keywords in the *grammar* so that they can be extracted and processed separately (see Section "Making Literal Symbols Case Insensitive" in *Lexical Analysis*). Thus file '`keyword.gla`' is defined as a non-product file by using `@N` instead of `@O` when specifying its name and content.

7 Miscellaneous

7.1 Profile Support dropped

Support of Profiling in Eli had included only the definition of the commandline-switch when calling the C-Compiler. The documentation on profiling-Support in Eli was incomplete in large parts.

Since it is not possible, to automatically determine the correct commandline-switches for support of profiling in the C-Compiler, the support of profiling was dropped int the current version of Eli. If you need to use profiling, substitute the `+prof`-derivation option with `+define='-pg'`. For further information on the profiling-support that is built into Odin, refer to the odin reference manual available from `ftp://ftp.cs.colorado.edu/pub/odin`.

7.2 New Specification file-type eta

The type-'`.eta`'-files serve to assemble include-files just as the '`.phi`'-files. While the '`.phi`'-files can be used only, to introduce code into a '`.c`' or '`.h`'-file, the '`.eta`'-files assemble an include-file that can be used throughout the whole specification.

For a description of this file-type, see Section "Descriptive Mechanisms Known to Eli" in *Guide for New Eli Users*.

Index