New Features of Eli Version 4.4

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: 2008/06/25 18:19:23 $

Table of Contents

New Features of Eli Version 4.4 1

1 Eli can now run under Windows 3

2 Producing portable document files 5

3 Including indexes in LaTeX documents 7

4 New functionality for unparser generation . . . 9

5 Eli-generated code as a component 11

6 Name analysis for declarators as in C 13

7 Scope Properties without Ordering Restrictions 15

8 Better error reporting for known operators 17

9 Additional property access function 19

10 New error reporting for parser conflicts 21

10.1 Example 21
10.2 Help 23
10.3 Parsable 23

11 Using anything to access information 25

12 Simplified arithmetic on strings 27

Index 29

New Features of Eli Version 4.4

This document gives information about new facilities available in Eli version 4.4 and those modifications made since the previous distributed Eli version 4.3 that might be of general interest. Numerous corrections, improvements, and additions have been made without being described here.

1 Eli can now run under Windows

Eli 4.4.0 now runs reliably in the Cygwin Unix environment under Windows NT, 2000, and XP. Windows 3.1, 95, 98, and ME do not support pre-emption and true multi-threading. Eli is a multi-threaded program and therefore randomly deadlocks under these systems. Unfortunately, Eli is considerably slower under Cygwin than it is under Unix.

For full instructions on how to obtain Cygwin and implement Eli under it, see the file '`README.Windows`' in the top-level directory of the Eli 4.4.0 distribution.

2 Producing portable document files

Eli now uses pdftex and pdflatex to produce PDF files. The original source for the text may be TeX or LaTeX, FunnelWeb, or Texinfo.

A conversion from TeX, LaTeX, or Texinfo to PDF is obtained simply by requesting the `:pdf` product:

```
foo.tex :pdf
bar.tnf :pdf
```

FunnelWeb formatting is complicated by the `typesetter` option (see Section "Typesetting Documentation with Eli" in *FunnelWeb*). You must choose the correct weaver for the option and include it in the request. For example, suppose that your FunnelWeb file did not use the `typesetter` option. In that case, `fwTex` is the correct weaver:

```
MySpec.fw :fwTex :pdf
```

You will *not* be able to produce PDF files from FunnelWeb files that specify '`@p typesetter = html`'.

It is also possible to use a type-'`specs`' file to specify a collection of type-'`tnf`' files, all of which are to be formatted. Suppose that the collection is defined by '`Doc.specs`', that '`ps`' is the directory into which PostScript files are to be placed, and that '`pdf`' is the directory into which PDF files are to be placed. Here is a request that yields one PostScript file in directory '`ps`' for each type-'`tnf`' file whose name appears in '`Doc.specs`'. That PostScript file is formatted for two-sided printing, with each chapter starting on an odd-numbered page:

```
Doc.specs :ps >ps
```

Here is a request yielding one PDF file in directory '`pdf`' for each type-'`tnf`' file whose name appears in '`Doc.specs`'. The '`+single`' parameter means that the PDF file will be formatted for single-sided printing — chapters are not forced to begin on odd pages:

```
Doc.specs +single :pdf >pdf
```

Unfortunately, the names of the files in the '`ps`' directory will have the suffix '`.tnf.tnfps`' and the files in the '`pdf`' directory will have names ending in '`.tnf.tnfpdf`'. These names can be changed by a simple Bourne shell script with commands like the following:

```
for f in `ls *.tnfpdf`
do mv $f `basename $f .tnf.tnfpdf`.pdf
done
```

3 Including indexes in LaTeX documents

The Odin tex package has been updated to support automatic index generation. You can now process LaTeX documents that contain `\index` commands, as described in Section 4.5 of the LaTeX User's Guide & Reference Manual. The index generated from these commands is printed by a `\printindex` command appearing in the LaTeX document. No additional commands or derivations are involved.

Indexes are processed by `makeindex`, which is invoked automatically during a derivation to PostScript or PDF. The Odin `+index_flags` parameter can be used to provide command-line options to `makeindex` if desired. For example, the following derivation would supply the index style file '`MyStyle.ist`' to `makeindex` during the consgtruction of a PDF file from '`Doc.tex`':

```
Doc.tex +index_flags='-s' (MyStyle.ist) :pdf
```

See `makeindex` for a complete description of the available options. Note that in general the option list will contain both literal flags and file names. Any file names that do not specify a full path should be parenthesized.

4 New functionality for unparser generation

Textual representations of tree-structured data objects can now be produced in five standard forms: a generic functional notation (as with Eli 4.3.x), an XML description of the data object, C++ code that will re-build the data object, Java code that will re-build the data object, and code describing the graph to the daVinci graph visualization tool (assumes daVinci 2.1, found at http://www.informatik.uni-bremen.de/daVinci/). It is also possible for the user to specify their own notation for describing tree-structured data objects (see Section "Languages describing tree structure" in *Abstract Syntax Tree Unparsing*).

Default output for terminal symbols has now been provided. The default assumes that the internal representation for the terminal symbol is a string table index. Token processors establish the internal representations for terminal symbols; `mkidn` and `mkstr` result in string table indices, whereas `mkint` does not. If you want to unparse a tree whose terminal symbols are not represented by string table indices, you need to override the default output (see Section "Overriding PTG patterns" in *Abstract Syntax Tree Unparsing*).

5 Eli-generated code as a component

When you request an executable program from Eli, it is normally supplied with a main program. If you want to use the generated code as a part of a larger system, then the main program should *not* be supplied. Generation of the main program is controlled by the `+nomain` parameter. This parameter is not new in Eli 4.4.0, but its documentation has been improved (see Section "nomain — Omitting the main program" in *Products and Parameters Reference*).

In some cases, the larger system containing the Eli-generated component has facilities for analyzing input, and the tree described by the LIDO specification is built directly by invoking Mk functions (see Section "Computed Subtrees" in *LIDO - Reference Manual*). That means it is necessary to specify that Eli should generate no parser by giving the parameter `+parser=none` (see Section "parser — Choose the parser generator" in *Products and Parameters Reference*).

6 Name analysis for declarators as in C

A defining occurrence of an identifer may be part of *Declarator*, that is a larger construct which determines the type of the defined identifer, for example the definition of the array `a` in

```
int a[a+1];
```

Here `a[a+1]` is the `Declarator` and the first `a` is its defining occurrence. A pair of roles `NAMEDeclaratorWithId` and `NAMEIdInDeclarator` has been added to the module `CScope` to solve name analysis for such cases.

7 Scope Properties without Ordering Restrictions

The name analysis modules which support scopes being properties of program entities have been reconsidered. (They are used for example to bind identifier occurrences in qualified names.) A new module `ScopeProp` has been added to the three existing ones (`AlgScopeProp`, `CScopeProp`, `BuScopeProp`). `ScopeProp` fits to any of the basic scope rule modules, Alg-like, C-like, or bottom-up. It does *not* impose any ordering restriction that would require the definition of a member to occur before its qualified use. It is recommended to use this module instead of any of the other three. Even in case that such a restriction is intended, one can use this module and enforce that restriction by a check of the related positions. The three specific modules are kept in order not to invalidate existing specifications.

8 Better error reporting for known operators

The Oil operator identification functions return an invalid operator if the specified operator indication is not associated with an operator matching the context specified. This results in a possibly misleading error report, effectively stating that there is no such operator.

When an operator indication is associated with exactly one operator, it may be preferable to indicate specific mismatches between the requirements of the operator and the types produced by the context. This can be done by returning the only possible operator, which places appropriate constraints on the types of the operands and delivers a specific result type.

`OilNoOverload` takes two arguments. The first is the operator indication being identified and the second is the result of a normal Oil operator identification function. The value of the second argument is returned if it is valid. If the second argument is invalid, `OilNoOverload` checks whether the first argument is associated with exactly one operator. If so, then `OilNoOverload` returns that operator; otherwise it returns the invalid operator yielded by the second argument.

```
OilNoOverload( 'oi', OilIdOpTS2( 'rt', 'oi', 'ats1', 'ats2' ));
```

The arguments are:

'`oi`' The operator indication being identified.

'`rt`' The required result type.

'`ats1`' The set of possible first operand types.

'`ats2`' The set of possible second operand types.

9 Additional property access function

The access function `Has` has been added to the PDL library (see Section "Predefined query and update operations" in *Definition Table*). Here is the signature:

```
int HasName(DefTableKey key)
```

If `Has` is applied to a definition table key that has an associated `Name` property, then it yields 1; otherwise it yields 0.

Since `NoKey` represents an invalid entity that has no properties, applying `Has` to `NoKey` yields 0.

If a property is to be queried by `Has`, then `Has` must be added to that property's operation list. For example, suppose that we wantto be able to ask whether a definition table key has the `DefTableKey`-valued property `Proc`. Here is a declaration of the property:

```
Proc: DefTableKey [Has];
```

Given that declaration, the following query could be made in LIDO:

```
RULE: Expr ::= AppliedOccurrence '(' ArgList ')'
COMPUTE
  IF(NOT(HasProc(AppliedOccurrence.Key)),
    message(ERROR,"Not a procedure identifier",0,COORDREF));
END;
```

10 New error reporting for parser conflicts

We have a new default format for reporting parser conflicts: For each conflict, it provides an example of a derivation leading to each of the conflicting situations. Our hope is that it will be easier to determine the cause of the conflict with this information than with the simple printout of the state that was given previously.

10.1 Example

Here is a simple example (see Section "Explanation of the grammar for word classification" in *Guide for New Eli Users*):

```
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 .
```

Suppose that we make the grammar non-LALR by removing the brackets around set_body. Here is the result of applying the default :parsable:

```
Conflicting Derivations
=======================

************************************************************************
   *** shift-reduce conflict on: word

text EOF
set_defs
set_defs set_def
|        set_name set_body
|        word
|
set_def
set_name set_body
         .  [REDUCE] set_body ->  {word} ?

text EOF
set_defs
set_def
set_name set_body
         elements
         set_element
         . word  [SHIFT] set_element -> word .  ?

************************************************************************
   *** shift-reduce conflict on: word
```

```
text EOF
set_defs
set_defs set_def
|        set_name set_body
|        word
|
set_def
set_name set_body
         elements .  [REDUCE] set_body -> elements  {word} ?

text EOF
set_defs
set_def
set_name set_body
         elements
         set_element
         . word  [SHIFT] set_element -> word .  ?

************************************************************************
```

This output gives two examples in which the parser will be unable to decide what to do when it sees a `word`. Each example shows two conflicting derivations (a derivation is the reverse of the parser's reduction process, see Section "How the generated parser determines phrase structure" in *Syntactic Analysis Manual*).

Each derivation begins with `text EOF`. Succeeding lines are the result of rewriting a single nonterminal symbol by applying some production of the grammar. The result of a rewriting step is aligned with the symbol being rewritten. Thus we can rewrite `text` to `set_defs`, and then rewrite `set_defs` to `set_defs set_def`.

The lines adjacent to the vertical bar show how the lookahead symbol can be derived: `set_def` is rewritten to `set_name set_body`, and then `set_name` is rewritten to `word`.

Below the vertical bar, the main derivation continues by rewriting the symbol at the top of the bar. In this case, `set_defs` is rewritten as `set_def`, which is rewritten as `set_name set_body` in turn. The final line of the first derivation shows the action that the parser would take at that point: reducing an empty string to a `set_body` in the presence of the lookahead symbol `word`.

This first derivation of the first example shows that the parser could recognize an empty set body and consider that the lookahead symbol `word` is the name of the next set. You should convince yourself that the second derivation of the first example shows how the parser could consider the lookahead symbol `word` in this context to be the first element of the first set. This ambiguity is clearly the result of omitting the opening brace. Without that delimiter, there is no way to make the decision.

The second example also involves a `word` symbol. Here the question is whether the `word` is the name of the next set or whether it is the next element of the current set. This error is the result of omitting the closing brace. Again, without that delimiter, there is no way to make the decision.

10.2 Help

If you have conflicts in your grammar, the :help derivation will show you messages for a type-'`pgsconflict`' file. Each message specifies the type of conflict and the set of terminals causing the conflict. Clicking the help browser's `Edit` button while you are looking at the message screen will bring up your editor on the type-'`pgsconflict`' file, which contains the sample derivations that illustrate how the conflicts arise (see Section 10.1 [Example], page 21).

10.3 Parsable

The :parsable derivation will give you the new diagnostics by default. You can still obtain the state printout by supplying a `+pgsOpt` parameter to the derivation:

```
-> sets.specs +pgsOpt='S' :parsable <
```

Here the string `'S'` requests "standard" processing.

11 Using anything to access information

A `Table` is a sparse memory with a 32-bit address space (see Section "Mapping Arbitrary Values To Definition Table Keys" in *Abstract data types to be used in specifications*). Each element of the memory contains a `DefTableKey` value. This memory is used to implement a mapping from values of some arbitrary type to definition table keys, allowing an arbitrary set of properties to be associated with each value.

Any number of named tables can be instantiated, each with a specific type of value to be mapped. More than one table can map any given type of value. When a table is initialized, the user must provide two functions. One computes a 32-bit address from the value to be mapped, the other determines whether two values of the type to be mapped are identical. The first of these two functions often uses a general hashing operation (see Section "Computing a Hash Value" in *Solutions of Common Problems*).

12 Simplified arithmetic on strings

The Eli library contains a general package called `strmath` for carrying out computations on numeric values represented by strings (see Section "Character String Arithmetic" in *The Eli Library*). This package is difficult to use in the context of LIDO, partially because each computation overwrites the results of the previous computation. `StrArith` is a wrapper for `strmath` that stores all results uniquely in the string table and associates error reports with the node in which they occur (see Section "Character String Arithmetic" in *Solutions of Common Problems*).

Index

+

+nomain 11
+parser 11

A

AlgScopeProp 15

B

BuScopeProp 15

C

C 13
C declarator 13
C++ 9
computed subtrees 11
CScopeProp 15
Cygwin 3

D

daVinci 9

F

FunnelWeb 5

H

Has 19
Hash 25

J

Java 9

L

latex 5

M

Mk functions 11

O

OilNoOverload 17

P

PDF files 5

S

ScopeProp 15
StrArith 27

T

Table 25
tex 5
texinfo 5

W

Windows 3

X

XML 9