New Features of Eli Version 4.3

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

$Revision: 3.7 $

Table of Contents

1 Specification Module Library 3
- 1.1 Name Analysis Library 3
- 1.2 Abstract Data Types 3
- 1.3 Property Library 4
- 1.4 Solutions of Common Problems 4

2 Lexical analysis 5
- 2.1 Detecting lexical errors explicitly 5
- 2.2 Scanning to, but not including, a newline 5
- 2.3 Auxiliary scanner and token processor definitions 5
- 2.4 Processing NUL characters during lexical analysis 5

3 Definition table 7

4 Oil 9

5 Change in tree parser naming conventions .. 11

6 FunnelWeb 13

7 Monitoring 15
- 7.1 Monitoring products 15
- 7.2 Main window command changes 15
- 7.3 Trees 15
- 7.4 Attributes 16
- 7.5 File and handlers windows 16
- 7.6 Monitoring user-defined types 16
- 7.7 Configuring Noosa 16

Index 17

This document gives information about new facilities available in Eli version 4.3 and those modifications made since the previous distributed Eli version 4.2 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 Specification Module Library

1.1 Name Analysis Library

Predefined Identifiers

The name analysis modules establish bindings of type `Bind`, which are triples of an identifier code, a definition table key, and an environment. The modules used for predefinitions `PreDefine` and `PreDefId` now provide three additional macros which allow to introduce and initialize `Bind` variables for predefined entities: `PreDefSymKeyBind`, `PreDefKeyBind`, and `PreDefBind`.

A C module `PreDefMod` is separated from the two predefinition modules. It provides the two function `PreDefine` and `PreDefineSym`. They now can be used directly e.g. in order to predefine entities in other environments than the outermost one. In that case `PreDefMod.specs` is used instead of instances of the modules `Predefine` and `PreDefId`. Existing uses of the `Predefine` and `PreDefId` are not affected by this modification.

Environment Module

Most of the macros that access components of structured values via pointers, e.g. `IdnOf`, `KeyOf`, `EnvOf`, are changed into functions to avoid multiple execution of side-effects caused by arguments of the macro calls.

1.2 Abstract Data Types

Lists in LIDO Specifications

In the module `LidoList` three alternative symbol roles are provided for tree nodes that carry list elements in construction or distribution of lists. They have different effects in cases where the element node can occur recursively in the tree. They handle the association between list elements and attribute of tree nodes in pre-order, post-order, or ignore recursive occurrences.

A condition attribute is introduced which decides at an element node whether it is considered for the list. By overriding its computation skipping of list elements can be controlled by non-trivial conditions.

Hence, the existing symbol role for that purpose, `FilterListElem`, is now outdated. It may be removed in a future version.

Linear Lists of Any Type

A function `AddToOrderedSet` has been added to be used for ordered lists without duplicates, which may implement sets.

A macro `SingleList` has been added especially to be used as the unary function of a CONSTITUENTS WITH clause.

The above facilities are added to both modules `List` and `PtrList`

Bit Sets of Arbitrary Length

A macro `ElemToBitSet` has been added especially to be used as the unary function of a CONSTITUENTS WITH clause.

1.3 Property Library

Map Objects to Integers

In the module `ObjCnt` the start value and the increment of the mapping can be modified by overriding of attributes.

1.4 Solutions of Common Problems

Counting Symbol Occurrences

In the module `Counter` the start value and the increment of the mapping can be modified by overriding of attributes.

2 Lexical analysis

There have been several additions involving auxiliary scanners and token processors: a new auxiliary scanner for reporting token errors, a header file defining the built-in auxiliary scanners and token processors, and a consolidation of NUL character processing.

2.1 Detecting lexical errors explicitly

Normally the scanner reports a lexical error when an input character cannot be the first character of any basic symbol. In other words, an error is signalled when the processor knows nothing about an input character. Sometimes, however, it is appropriate to recognize a specifc sequence of input characters as an invalid token.

A new auxiliary scanner called `lexerr` handles this situation. It reports that the scanned character sequence is not a token. It does not alter the initial classification, and does not compute a value. There is no source file for this token processor; it is a component of the scanner itself, but its interface is exported so that it can be used by other modules.

2.2 Scanning to, but not including, a newline

The auxiliary scanner `auxNoEOL` extends the character sequence matched by the associated pattern to the end of the current line, but does not include the terminating newline. It is useful in situations where a token must begin at the beginning of a line, and therefore has a regular expression whose first character is the newline. A token preceding token using `auxEol` to extend to the end of a line would absorb the newline, thus making it impossible to recognize the token beginning at the beginning of the next line.

2.3 Auxiliary scanner and token processor definitions

The header file '`$elipkg/Scan/ScanProc.h`', containing definitions of all of the auxiliary scanners available in the library, has been added. It should be included by any C program that uses auxiliary scanners from the library.

2.4 Processing NUL characters during lexical analysis

All of the auxiliary scanners that scan over a newline now invoke `auxNUL` when they detect an ASCII NUL just beyond that newline. An ASCII NUL just beyond a newline character signals the end of the current source buffer, and an operation is needed to refill the buffer. By invoking `auxNUL` whenever this condition arises, we have centralized the operation of refilling the buffer at one point. This means that if a specification requires some special action whenever the buffer is refilled, it can override `auxNUL`.

We strongly recommend that users adhere to this convention when they must write an auxiliary scanner that must scan over a newline. Here is a typical code sequence for such a scanner. The variable `p` is the scan pointer and `start` points to the beginning of the current token:

```
if (*p == '\0') {
  int current = p - start;
  TokenStart = start = auxNUL(start, current);
  p = start + current;
```

```
  StartLine = p - 1;
  if (*p == '\0') {
    /* Code to deal appropriately with end-of-file.
     * Some of the possibilities are:
     *   1. Output an error report and return p
     *   2. Simply return p
     *   3. Move to another file and continue
     ***/
  }
}
```

3 Definition table

A new operation called CloneKey is provided. It takes a single definition table key argument and returns a new key whose properties are initialised to the same values as the key argument. Property values are shallow-copied.

4 Oil

The OilList module provides computations that implement parameter lists and argument lists for operator identification of function calls. The elements are arranged in left-to-right order as they occur in the tree.

OilList is instantiated simply by mentioning its name:

```
$/oil/OilList.fw
```

It provides the computational roles ParameterListRoot, ParameterListElem, ArgumentListRoot, and ArgumentListElem to support type analysis, the the roles ArgumentDeListRoot and ArgumentDeListElem to support code generation.

The CLASS SYMBOL ParameterListElem is to be inherited by those SYMBOLs that contribute parameter types to the parameter list. The value of the attribute ParameterListElem.GivenType (of type tOilType) must be set by the user.

The CLASS SYMBOL ParameterListRoot is to be inherited by a SYMBOL that has all of the parameter SYMBOLs in its subtrees. The result of the computation is the attribute ParameterListRoot.ParameterList of type tOilArgSig, which specifies all of the parameter types. In order to obtain a complete signature for the function, the result type (ResType, for example) must be added: OilAddArgSig(ResType, ParameterListRoot.ParameterList)

The CLASS SYMBOL ArgumentListElem is to be inherited by those SYMBOLs that contribute arguments to the argument list. The value of the attribute ArgumentListElem.GivenType (of type tOilType) must be set by the user.

The CLASS SYMBOL ArgumentListRoot is to be inherited by a SYMBOL that has all of the argument SYMBOLs in its subtrees. The result of the computation is the attribute ArgumentListRoot.ArgumentList of type tOilSetSig, which specifies all of the argument types. It can be used in a call to OilIdOpTSn to identify a function whose parameter types could be the result of coercing the argument types.

The CLASS SYMBOL ArgumentDeListElem is to be inherited by those SYMBOLs that contribute arguments to the argument list. The result of the computation is the attribute ArgumentDeListElem.RequiredType of type tOilType, which specifies the type required by the function for that argument.

The CLASS SYMBOL ArgumentDeListRoot is to be inherited by a SYMBOL that has all of the argument SYMBOLs in its subtrees. The value of the attribute ArgumentDeListRoot.Operator (of type tOilOp) must be set by the user.

Please note that it is not sensible to try to test the value of ArgumentDeListElem.RequiredType against the a priori type of the argument expression for the purpose of error reporting. The reason is that if ArgumentDeListRoot.Operator is a valid operator then there is no error, and if ArgumentDeListRoot.Operator is an invalid operator then there is no information about the required types of the arguments.

5 Change in tree parser naming conventions

The conventions for naming types and routines created by the tree parser `tp` have been changed to avoid confusion with other Eli components and to shorten some names:

1. The type name `NODEPTR-TYPE` has been replaced by `TPNode`
2. The prefix for all function names is now `TP_` rather than `burm_`.
3. The name of the header file defining the prototypes of the generated functions is not '`tp_gen.h`' rather than '`xform.h`'.

6 FunnelWeb

The maximum number of lines in a FunnelWeb specification was increased to 65000.

7 Monitoring

The Noosa system (invoked using the `:mon` product has undergone many changes since the last release of Eli. Some relatively minor alterations have been made to the user interface. Numerous internal changes were also made, many aimed at improving the speed of Noosa.

The following section summarise the major user visible changes. See the See Section "top" in *Execution Monitoring Reference*, or the Noosa online help for more details.

7.1 Monitoring products

The `:mondbx` product for using Noosa in conjunction with Sun's dbx debugger is no longer supported. It was too hard to maintain and most users do not have access to this debugger anyway.

The `:mongdb` product for debugging with GNU's GDB is now fully supported and is working (in contrast with the situation at the Eli 4.2 release). Also, the `:mongdb` product now supports the `+arg` parameter for specifying the arguments to the program being debugged.

The monitoring derivations now allow more than one `+arg` parameter.

7.2 Main window command changes

The Token command now also displays the names of non-literal tokens.

Many Noosa menus can now be "torn off" so that they stay on the screen. Also, the Run, Continue and Kill commands now have keyboard accelerators Alt-r, Alt-c and Alt-k, respectively.

The main Noosa text windows (input and transcript) can now be searched and saved to files via the Noosa menu. The transcript can also be cleared.

The command to set event tracing filter regular expression now uses a popup dialog box.

7.3 Trees

The Noosa tree displays now come in four varieties: "just source" showing only the source tree, "separated computed" where each computed tree is shown in a separate window, "source and computed" where the source tree and computed trees are shown in the same window, and "incremental" where all trees are displayed together in a style which allows selective viewing.

All forms of tree display now support the Node command that allows you to go to the "most relevant" node for a selected coordinate in the input text window. Note that this command might not always produce expected results in the presence of computed trees where the coordinate ranges of non-related nodes can overlap.

The tree displays now have commands that allow their contents (either whole or visible) to be saved as PostScript.

7.4 Attributes

The tree displays now support two kinds of pop-up menu on the right mouse button. On symbol names the menu gives the attributes of the symbol. On rule names the menu gives attributes of the rule and terminal values that occur in that rule. In each case you can express an interest in seeing the value of the attribute or terminal, with the option of stopping execution when it is available.

Symbols and rules for which you have expressed an interest in one of their attributes are highlighted by underlining rather than by the extra graphic that was previously used.

Many values of complex types can now be browsed once they have been displayed in the transcript, including definition table keys, environments, PTG nodes, OIL types and typesets, and syntax tree nodes (NODEPTRs).

7.5 File and handlers windows

The Windows menu contains a File command that can be used to create windows in which you can edit arbitrary files.

The Handlers window in Noosa lets you set Tcl handlers for event types. There have been many improvements including the ability to rename handlers, to disable/enable them without deleting and to save them to files. Saved handlers can also be autoloaded.

7.6 Monitoring user-defined types

Eli now supports specifications of type '`tcl`' containing Tcl code. If any of these files are present in the user's specifications they are concatenated in an undefined order and loaded by Noosa on startup. Files supplied in this way can be used to provide extra monitoring support. The most likely situation where this will be useful is to enable Noosa to browse values of user-defined types. See the See Section "top" in *Execution Monitoring Reference*, for full details on how to do this.

7.7 Configuring Noosa

Many things such as the sizes of Noosa's windows and the colours used to highlight browsable values and selected tree nodes can now be configured using X resources.

Index

$

‘$elipkg/Scan/ScanProc.h’ 5

+

+arg and mongdb 15

A

AddToOrderedSet 3
ArgumentDeListElem 9
ArgumentDeListRoot 9
ArgumentListElem 9
ArgumentListRoot 9
Attribute monitoring and browsing 16
auxNoEOL 5
auxNUL 5

B

Bind 3

C

CloneKey 7
Counter 4

E

Editing files in Noosa 16
ElemToBitSet 3
Environment Module 3
Event tracing dialog box 15

F

FunnelWeb 13

L

lexerr 5
LidoList 3
Lines in a FunnelWeb file 13
List 3

M

mondbx is no longer supported 15
mongdb and +arg 15
mongdb is now working 15
Monitoring 15
Multiple +arg parameters 15

N

Node command 15
Noosa 15
Noosa handlers 16
Noosa keyboard accelerators 15
Noosa searching and saving 15
Noosa tearoff menus 15
Noosa tree displays 15
Noosa X resources 16

O

ObjCnt 4

P

ParameterListElem 9
ParameterListRoot 9
Postscript output from tree displays 15
PreDefBind 3
PreDefId 3
PreDefine 3
PreDefineSym 3
PreDefKeyBind 3
PreDefMod 3
PreDefSymKeyBind 3
PtrList 3

S

Searching and saving Noosa text 15
SingleList 3

T

Tcl specifications 16
Token names in Noosa 15

X

X resources for Noosa 16