New Features of Eli Version 4.7

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: 2009/10/15 21:17:47 $

Table of Contents

New Features of Eli Version 4.7

This document gives information about new facilities available in Eli version 4.7 and those modifications made since the previous distributed Eli version 4.6 that might be of general interest. Numerous corrections, improvements, and additions have been made without being described here.

1 Abstract Rule Names in Mapping Rules

Some abstract grammars have several rules with different names and identical signatures (this is relatively common in computed trees, see Section "Computed Subtrees" in *LIDO - Reference Manual*). If such rules are represented in an unambiguous manner in text, and that text is parsed, one must be able to map the disambiguated concrete rules into the appropriate abstract rules. Because the abstract rules have identical signatures, pattern matching won't work.

In order to solve this problem, the Maptool now accepts an optional rule name in a rule mapping (see Section "Specifying rule mappings" in *Syntactic Analysis*). A simple example might be a representation of dyadic expressions without explicit operators:

```
RULE Add: Expression ::= Expression Expression END;
RULE Mul: Expression ::= Expression Expression END;
...
```

Suppose that such nodes are represented in text by fully-parenthesized arithmetic expressions in the standard notation. Rule mappings specifying the rule names explicitly would then be needed to disambiguate the pattern match:

```
MAPRULE
Expression: '(' Expression '+' Expression ')' < $1 $2 >: Add .
Expression: '(' Expression '*' Expression ')' < $1 $2 >: Mul .
...
```

2 Change in the Token Processor Interface

Input character strings are converted into internal representations by token processors (see Section "Token Processors" in *Lexical Analysis*). A token processor should not alter the string that it is converting in any way, but the original interface specification did not enforce that restriction because C did not provide a const qualifier.

In many applications, it is important to apply token processors to literal string constants. Recent C++ compilers do not allow literal string constants to be passed to character string parameters that are not const-qualified. We therefore decided to alter the token processor interface by const-qualifying its character string parameter.

A string pointer that is *not* const-qualified can always be passed to a const-qualified parameter; only the reverse is prohibited by the C definition. Thus this change is transparent, unless you have defined your own token processors. In the simplest case, you need only add the const qualifier to the first parameter in your token processor's definition. If your token processor does, however, alter its argument string (for example, by planting a null character at the end) then it must be rewritten.

Index