Abstract Syntax Tree Unparsing

$Revision: 2.17 $

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

Table of Contents

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.

A so-called “pretty-printer” is an example of a processor that incorporates an unparser: It reads arbitrarily-formatted text, builds a tree representing the text’s structure, and then unparses that tree using appropriate formatting rules to lay out the text in a standard way.

An unparser is also used to produce a textual representation of a tree-structured data object. One example of such a textual representation is the XML file used to transmit a data object over the Internet; another is a Java program that can be executed to re-build the object.

Arbitrary unparsers can be specified by means of a combination of attribute computations and PTG (see Section “Pattern Specifications” in *PTG: Pattern-Based Text Generator*) patterns. Writing these specifications by hand is a tedious process for a large tree grammar.

Given a specification of the LIDO rules defining a tree grammar, Eli can derive the specifications of certain common unparsings. The result is a FunnelWeb file (see Section “Introduction” in *FunnelWeb*) that is used directly to produce output routines for a generated processor. Each of the common unparsings has certain characteristics that must be understood to use it effectively.

Although a pre-packaged unparsing may suffice for almost all of the rules of a particular tree grammar, a user may need to make a few changes in structure or representation. The unparser generator provides facilities for specifying such changes, while retaining the bulk of the generated attribute computations and PTG patterns.

Finally, an unparser must be derived from a specification of the tree grammar to be unparsed together with specifications of any changes in representation. The resulting FunnelWeb file must either be extracted or incorporated into the derivation of the processor using it.

1 Using an Unparser

In order to make the discussion concrete, we will use a trivial expression language as an example. That language is defined by the FunnelWeb file ‘`example.fw`’:

```
@O@<example.lido@>==@{
RULE PlusExp: Expression ::=    Expression '+' Expression    END;
RULE StarExp: Expression ::=    Expression '*' Expression    END;
RULE Parens:  Expression ::=    '(' Expression ')'           END;
RULE IdnExp:  Expression ::=    Identifier                   END;
RULE IntExp:  Expression ::=    Integer                      END;

RULE CallExp: Expression ::=    Identifier '(' Arguments ')' END;
RULE ArgList: Arguments  LISTOF Expression                   END;
@}

@O@<example.gla@>==@{
Identifier: C_IDENTIFIER
Integer:    C_INT_DENOTATION
@}

@O@<example.con@>==@{
Axiom: Expression .
Expression: Expression '+' Term / Term .
Term: Term '*' Factor / Factor .
Factor:
  Identifier /
  Integer /
  Identifier '(' Arguments ')' /
  '(' Expression ')' .
Arguments: Expression // ',' .
@}

@O@<example.map@>==@{
MAPSYM
Expression ::= Term Factor .
@}
```

The non-literal terminal symbols `Integer` and `Identifier` are defined by canned descriptions. According to their definitions (see Section “Available Descriptions” in *Lexical Analysis*), `C_IDENTIFIER` uses the token processor `mkidn` to establish the internal value of the terminal symbol, and `C_INT_DENOTATION` uses `mkstr`.

The concrete grammar and mapping specification together provide three precedence levels of `Expression` (`Expression`, `Term` and `Factor`) to disambiguate the dyadic computation rules. Parentheses can be used to override the defined precedence and association, and their presence is reflected in the tree.

Finally, the concrete grammar specifies the comma as the argument separator in procedure calls. This fact is *not*, however, explicit in the tree.

How could a pretty-printer for this language be constructed? Eli is capable of generating an unparser specification from the FunnelWeb file given above. That specification is, itself, a FunnelWeb file. These two FunnelWeb files, plus some additional information, provide a complete definition of the pretty-printer. Suppose that the additional information will be provided by a FunnelWeb file '`Add.fw`'. Here is a type-'`specs`' file that will define the pretty-printer:

```
example.fw
example.fw :idem
Add.fw
```

The first line describes the file defining the language, the second describes the textual unparser derived from the language definition (see Chapter 4 [Deriving an Unparser], page 19), and the last describes the file defining the additional information. A processor is derived from this type-'`specs`' file in the usual way (see Section "exe – Executable Version of the Processor" in *Products and Parameters Reference*).

'`Add.fw`' must specify three additional pieces of information to complete the pretty-printer:

- The formatting strategy.
- Which node of the tree to print.
- The argument list separators.

Our sample language involves only expressions, so the formatting strategy can be specified by instantiating a library module used to format C programs:

```
$/Output/C_Separator.fw
```

The generated unparser specification establishes a type-`PTGNode` value for the `IdemPtg` attribute of each of the tree nodes. Because this value was created with the help of the library module defined by '`C_Separator.fw`' it should be written to the standard output stream by applying the routine `Sep_Out` to it:

```
SYMBOL Axiom COMPUTE Sep_Out(THIS.IdemPtg); END;
```

Finally, we can ensure that commas separate the arguments by overriding the generated `IdemPtg` computation for the argument list. The complete FunnelWeb file '`Add.fw`' would be:

```
@O@<Add.specs@>==@{
$/Output/C_Separator.fw
@}

@O@<Add.lido@>==@{
SYMBOL Axiom COMPUTE Sep_Out(THIS.IdemPtg); END;

RULE: Arguments LISTOF Expression
COMPUTE
  Arguments.IdemPtg=
    CONSTITUENTS Expression.IdemPtg SHIELD Expression
      WITH (PTGNode, PTGArgSep, IDENTICAL, PTGNull);
END;
@}
```

```
@O@<Add.ptg@>==@{
ArgSep: $ { "," [Separator] } $
@}
```

The `[Separator]` function call insertion in the pattern allows the module defined by ‘`C_Separator.fw`’ to provide appropriate layout characters between the elements of an expression to implement the desired formatting.

2 Available Kinds of Unparser

Eli is capable of generating specifications for the following kinds of unparsers:

Textual	Print a “source text” representation of the tree. This kind of unparser is most useful for processors that solve a “source-to-source” translation problem, such as pretty-printing or language extension.
Structural	Print a “structural” representation of the tree. This kind of unparser is most useful for debugging applications, and for processors that output textual representations of tree-structured data objects.

2.1 Textual unparser

A textual unparser creates source text which, when parsed, results in the tree that was unparsed. For example, the pretty-printer described above accepted a sentence in the expression language and built a tree. It then unparsed that tree to produce an equivalent sentence in the expression language that was formatted in a particular way. If the resulting sentence were parsed according to the rules of the expression language, the result would be the tree from which it had been created.

Consider the tree representing the following sentence in the expression language:

```
a((b+c)*d, e)
```

None of the terminal symbols `( ) + *` is stored explicitly in the tree. Thus the textual unparser must reconstruct these terminal symbols from the LIDO rules defining the tree nodes.

The terminal symbol `,` doesn't appear in any of the LIDO rules, and therefore it cannot be automatically reconstructed by the textual unparser. Additional information must be provided by the user to insert it into the unparsed text. This is a common consequence of using `LISTOF` productions.

Our definition of the tree grammar for the expression language contains the following rule:

```
RULE Parens: Expression ::= ’(’ Expression ’)’ END;
```

The purpose of this rule is to support the unparser by retaining information about the presence of parentheses used to override the normal operator precedence. Such parentheses result in a `Parens` node in the tree, and the unparser can then use this LIDO rule to reconstruct the parentheses.

`Parens` is an example of a *chain rule*, in which the left-hand non-terminal symbol appears exactly once as the only non-terminal symbol on the right-hand side. Such chain rules are often eliminated from a tree grammar, because they have no significance to the computations it supports. If a textual unparser is to be generated, however, then either a chain rule must be in the tree grammar or there must be additional information that allows the unparser to reconstruct its terminal symbols.

One important aspect of the textual form of the program that is missing from the LIDO rules is how to separate the basic symbols. For example, consider a `CallExp` in the expression language:

```
a(b, c)
```

There is no information in the LIDO rules about whether a space should precede and/or follow a `(` or `,`. Spacing is important for making the text readable, however, and cannot simply be ignored.

2.1.1 Computations for plain productions

A generated textual unparser defines the following computation (two attributes are used to simplify overriding, see Section 3.2 [Changing `IdemPtg` computations], page 16):

```
ATTR IdemPtg, IdemOrigPtg: PTGNode;

CLASS SYMBOL IdemReproduce COMPUTE
  SYNT.IdemOrigPtg=
    RuleFct("PTGIdem_", RHS.IdemPtg, TermFct("PTGIdem_"));
  SYNT.IdemPtg=THIS.IdemOrigPtg;
END;
```

The class symbol `IdemReproduce` is inherited by every non-terminal symbol appearing on the left-hand side of a plain production. For example, it is inherited by `Expression` and `Axiom` in the textual unparser specification generated from the expression language definition.

This computation invokes a function specific to the LIDO rule and, if the rule contains any instances of non-literal terminal symbols, a function specific to each. For example, the effect of `Expression` inheriting `IdemReproduce` is to carry out computations at the `StarExp` and `CallExp` rules that are equivalent to the following rule computations:

```
RULE StarExp: Expression ::= Expression '*' Expression
COMPUTE
  Expression[1].IdemOrigPtg=
    PTGIdem_StarExp(Expression[2].IdemPtg,Expression[3].IdemPtg);
  Expression[1].IdemPtg=Expression[1].IdemOrigPtg;
END;

RULE CallExp: Expression ::= Identifier '(' Arguments ')'
COMPUTE
  Expression[1].IdemOrigPtg=
    PTGIdem_CallExp(Arguments.IdemPtg,PTGIdem_Identifier(Identifier));
  Expression[1].IdemPtg=Expression[1].IdemOrigPtg;
END;
```

(For details about `RuleFct` and `TermFct`, see Section "Predefined Entities" in *LIDO - Reference Manual*.)

Here are several PTG patterns appearing in a textual unparser generated from the expression language definition that illustrate how the PTG functions are specified:

```
Idem_StarExp: $1  "*" [Separator]  $2
Idem_IdnExp:  $1  [Separator]
Idem_CallExp: $2  [Separator] "(" [Separator]  $1  ")" [Separator]
```

Notice how these patterns reconstruct the terminal symbols `*`, `(`, and `)`.

The different orders of the indexed insertion points in the patterns `Idem_StarExp` and `Idem_CallExp` are due to the definition of the computation above. `Idem_StarExp` has two non-terminal children, which appear in order as the arguments of the generated rule function. `Idem_CallExp`, on the other hand, has a non-literal terminal as its first child and a non-terminal as its second. The non-literal terminal arguments of the generated rule function follow the non-terminal arguments.

`Separator` is the function that makes the decision about how to place layout characters (see Section "Introduce Separators in PTG Output" in *Tasks related to generating output*). A call to `Separator` is inserted into every pattern after each terminal symbol, both literal and non-literal. This allows a decision to be made about layout characters between each pair of terminal symbols.

The separator module provides the following output functions, which must be used instead of the corresponding PTG output functions (see Section "Output Functions" in *PTG: Pattern-based Text Generator*):

```
PTGNode Sep_Out(PTGNode root);
PTGNode Sep_OutFile(char *filename, PTGNode root);
PTGNode Sep_OutFPtr(FILE *fptr, PTGNode root);
```

The module library contains two modules that implement different strategies for selecting layout characters:

'`Sp_Separator.fw`'
: A single space is used as a separator regardless of the context.

'`C_Separator.fw`'
: Reasonable separator placement rules for C program text: a newline is added after any of `; { }`, no separator is added after any of `( [ . ++ --`, no separator is added before any of `[ ] , . ; ++ --`, and a single space added in all other cases.

If none of the available modules is satisfactory, then you must create your own. The simplest approach is to modify one from the library. Here is a sequence of Eli requests that will extract '`C_Separator.fw`' as file `My_Separator.fw`, make `My_Separator.fw` writable, and initiate an editor session on it:

```
-> $elipkg/Output/C_Separator.fw > My_Separator.fw
-> My_Separator.fw !chmod +w
-> My_Separator.fw <
```

In order to change the decision about what (if any) separator is to be inserted in a given context, you need to change the function called `Sep_Print`. `Sep_Print` (see Section "Introduce Separators in PTG Output" in *Specification Module Library: Generating Output*) has three arguments: a pointer to the file to which the separator is to be written, a pointer to the string that immediately precedes the separator, and a pointer to the string that immediately follows the separator.

`Sep_Print` must decide whether a separator is appropriate between the two strings given by its second and third arguments, and if so then what that separator should be. If a separator is required, `Sep_Print` must write that separator to the file. `Sep_Print` must not modify the strings passed to it.

2.1.2 Computations for LISTOF productions

A generated textual unparser defines the following computation for a `LISTOF` production named '`r`' with left-hand side '`X`' and elements '`Y | Z`' (two attributes are used to simplify overriding, see Section 3.2 [Changing `IdemPtg` computations], page 16):

```
ATTR IdemPtg, IdemOrigPtg: PTGNode;

CLASS SYMBOL IdemReproduce_X COMPUTE
  SYNT.IdemOrigPtg=
    PTG_r(
      CONSTITUENTS (Y.IdemPtg, Z.IdemPtg) SHIELD (Y, Z)
      WITH (PTGNode, PTGIdem_2r, PTGIdem_1r, PTGNull));
  SYNT.IdemPtg=THIS.IdemOrigPtg;
END;
```

The class symbol `IdemReproduce_X` is inherited by the non-terminal symbol `X`. For example, in the textual unparser specification generated from the expression language, there is such a rule with `r` being `ArgList`, `X` being `Arguments`, and `Y` being `Expression`. There is no `Z` in that case:

```
CLASS SYMBOL IdemReproduce_Arguments COMPUTE
  SYNT.IdemOrigPtg=
    PTG_ArgList(
      CONSTITUENTS (Expression.IdemPtg) SHIELD (Expression)
      WITH (PTGNode, PTGIdem_2ArgList, PTGIdem_1ArgList, PTGNull));
  SYNT.IdemPtg=THIS.IdemOrigPtg;
END;
```

`Arguments` inherits `IdemReproduce_Arguments` in the textual unparser specification generated from the expression language definition.

The computation for the class symbol invokes three functions specific to the LIDO rule. Here are the three PTG patterns specifying those functions for the `ArgList` rule:

```
Idem_ArgList:  $
Idem_2ArgList: $ $
Idem_1ArgList: $
```

PTG patterns for other `LISTOF` productions will differ from these only in the pattern names.

2.2 Structural unparser

A structural unparser creates a textual description of the tree in terms of rule names and non-literal terminal symbols. For example, the sentence '`a(b,c)`' in the expression language could be unparsed as the XML file:

```
<rule_000>
  <CallExp>
    a
    <ArgList>
      <IdnExp>b</IdnExp>
      <IdnExp>c</IdnExp>
    </ArgList>
  </CallExp>
</rule_000>
```

The entire sentence is output as a `rule_000` because the LIDO rule defining `Axiom` was generated, and was given the name `rule_000` by Eli. The single child of this node is a `CallExp` with two components, the non-literal terminal symbol '`a`' and an `ArgList` made up of two `IdnExp` nodes.

Appropriate layout, with meaningful line breaks and indentation, is important for a human trying to understand the output of a structural unparser. This formatting depends only on structure, however, not on the content of the output.

Structural unparser generators producing both simple descriptions of trees and descriptions in several standard languages are available. It is also possible for a user to create an unparser generator that describes the tree in a language of their own choosing.

2.2.1 Computations for plain productions

A generated structural unparser defines the following computation (two attributes are used to simplify overriding, see Section 3.2 [Changing `IdemPtg` computations], page 16):

```
ATTR IdemPtg, IdemOrigPtg: PTGNode;

CLASS SYMBOL IdemReproduce COMPUTE
  SYNT.IdemOrigPtg=
    RuleFct("PTGIdem_", RHS.IdemPtg, TermFct("PTGIdem_"));
  SYNT.IdemPtg = THIS.IdemOrigPtg;
END;
```

The class symbol `IdemReproduce` is inherited by every non-terminal symbol appearing on the left-hand side of a plain production. For example, it is inherited by `Expression` and `Axiom` in the structural unparser specification generated from the expression language definition.

This computation invokes a function specific to the LIDO rule and, if the rule contains any instances of non-literal terminal symbols, a function specific to each. For example, the effect of `Expression` inheriting `IdemReproduce` is to carry out computations at the `StarExp` and `CallExp` rules that are equivalent to the following rule computations:

```
RULE StarExp: Expression ::= Expression '*' Expression
COMPUTE
  Expression[1].IdemOrigPtg=
    PTGIdem_StarExp(Expression[2].IdemPtg,Expression[3].IdemPtg);
  Expression[1].IdemPtg=Expression[1].IdemOrigPtg;
END;

RULE CallExp: Expression ::= Identifier '(' Arguments ')'
COMPUTE
  Expression[1].IdemOrigPtg=
    PTGIdem_CallExp(Arguments.IdemPtg,PTGIdem_Identifier(Identifier));
  Expression[1].IdemPtg=Expression[1].IdemOrigPtg;
END;
```

(For details about `RuleFct` and `TermFct`, see Section "Predefined Entities" in *LIDO - Reference Manual*.)

Here are several PTG patterns from a structural unparser generated from the expression language definition that illustrate how those functions are specified:

```
Idem_StarExp:
  "<StarExp>" [BP_BeginBlockI]
    [BP_BreakLine] $1 [BP_BreakLine] $2 [BP_BreakLine]
  [BP_EndBlockI] "</StarExp>"
Idem_IdnExp:
  "<IdnExp>" [BP_BeginBlockI]
    [BP_BreakLine] $1 [BP_BreakLine]
  [BP_EndBlockI] "</IdnExp>"
Idem_CallExp:
  "<CallExp>" [BP_BeginBlockI]
    [BP_BreakLine] $2 [BP_BreakLine] $1 [BP_BreakLine]
  [BP_EndBlockI] "</CallExp>"
```

These patterns are the ones generated if the output is to be an XML file (see Section 2.2.3 [Languages describing tree structure], page 13).

The different orders of the indexed insertion points in the patterns `Idem_StarExp` and `Idem_CallExp` are due to the definition of the computation above. `Idem_StarExp` has two non-terminal children, which appear in order as the arguments of the generated rule function. `Idem_CallExp`, on the other hand, has a non-literal terminal as its first child and a non-terminal as its second. The non-literal terminal arguments of the generated rule function follow the non-terminal arguments.

Generated structural unparsers use the block print module (see Section "Typesetting for Block Structured Output" in *Tasks related to generating output*) to provide layout. The generated PTG patterns invoke functions of this module to mark potential line breaks and the boundaries of logical text blocks. The block print module provides the following output functions, which must be used instead of the corresponding PTG output functions (see Section "Output Functions" in *PTG: Pattern-based Text Generator*):

```
PTGNode BP_Out(PTGNode root);
PTGNode BP_OutFPtr(FILE *fptr, PTGNode root);
PTGNode BP_OutFile(char *filename, PTGNode root);
```

Note that the textual representation of the *children* of every node is considered to be a logical text block. A line break can occur before each child. The effect of this specification is to keep the textual representation of a node on a single line if that is possible. Otherwise, the sequence of children is written one per line, indented from the name of the block's rule.

2.2.2 Computations for LISTOF productions

A generated structural unparser defines the following computation for a `LISTOF` production named '`r`' with left-hand side '`X`' and elements '`Y | Z`' (two attributes are used to simplify overriding, see Section 3.2 [Changing `IdemPtg` computations], page 16):

```
ATTR IdemPtg, IdemOrigPtg: PTGNode;

CLASS SYMBOL IdemReproduce_X COMPUTE
```

```
    SYNT.IdemOrigPtg=
      PTG_r(
        CONSTITUENTS (Y.IdemPtg, Z.IdemPtg) SHIELD (Y, Z)
        WITH (PTGNode, PTGIdem_2r, PTGIdem_1r, PTGNull));
    SYNT.IdemPtg=THIS.IdemOrigPtg;
  END;
```

The symbol `IdemReproduce_X` is inherited by the non-terminal symbol `X`. For example, in the structural unparser specification generated from the expression language, there is such a rule with `r` being `ArgList`, `X` being `Arguments`, and `Y` being `Expression`. There is no `Z` in that case:

```
  CLASS SYMBOL IdemReproduce_Arguments COMPUTE
    SYNT.IdemOrigPtg=
      PTG_ArgList(
        CONSTITUENTS (Expression.IdemPtg) SHIELD (Expression)
        WITH (PTGNode, PTGIdem_2ArgList, PTGIdem_1ArgList, PTGNull));
    SYNT.IdemPtg=THIS.IdemOrigPtg;
  END;
```

`Arguments` inherits `IdemReproduce_Arguments` in the structural unparser specification generated from the expression language definition.

The computation for the class symbol invokes three functions specific to the LIDO rule. Here are the three PTG patterns specifying those functions for the `ArgList` rule:

```
  Idem_ArgList:
    "<ArgList>" [BP_BeginBlockI]
     [BP_BreakLine] $ [BP_BreakLine]
    [BP_EndBlockI] "</ArgList>"
  Idem_2ArgList: $ { [BP_BreakLine] } $
  Idem_1ArgList: $
```

PTG patterns for other `LISTOF` productions will differ from these only in the rule name.

2.2.3 Languages describing tree structure

By default, a structural unparser generator uses a generic functional representation to describe the tree. Here's the default representation of the sentence '`a(b,c)`' in the expression language:

```
  rule_000(CallExp(a,IdnExp(b),IdnExp(c)))
```

(Recall that the entire sentence is output as a `rule_000` because the LIDO rule defining `Axiom` was generated, and was given the name `rule_000` by Eli.)

Four other standard representations are available:

`XML`	Generates an unparser that produces an XML representation of the tree, and a separate DTD file defining the possible structures.
`CPP`	Generates an unparser that produces C++ code to build the tree, and a separate module defining the set of C++ classes used.
`Java`	Generates an unparser that produces Java code to build the tree, and a separate package defining the set of Java classes used.

It is also possible to build structural unparser generators for other application languages by modifying existing generator specifications. All unparser generators have the same general organization: they analyze the tree grammar and produce class symbol computations and PTG patterns to output any tree defined by that grammar. Much of the analysis is common, with differences appearing only in the final output of the generated FunnelWeb file.

The unparser generator specifications available in the library are:

'`$/Unparser/Analysis.fw`'
: Analysis of the input text that defines the tree grammar. Common attribute computations supporting a wide range of unparsers.

'`$/Unparser/Idem.fw`'
: Attribute computations specific to textual unparsers.

'`$/Unparser/Tree.fw`'
: Attribute computations specific to the generic functional representation.

'`$/Unparser/Xml.fw`'
: Attribute computations specific to XML files and the associated DTD file.

'`$/Unparser/Cpp.fw`'
: Attribute computations specific to C++ code and the associated module definition.

'`$/Unparser/Java.fw`'
: Attribute computations specific to Java code and the associated package definition.

Suppose that you wanted to create an unparser generator that would produce Modula-3 code to build the tree, and a separate interface file defining the tree structure. Because Modula-3 is quite similar to Java in its structure, you might start by modifying '`$/Unparser/Java.fw`' from the library. Here is a sequence of Eli requests that will extract '`Java.fw`' as file `Modula-3.fw`, make `Modula-3.fw` writable, and initiate an editor session on it:

```
-> $elipkg/Unparser/Java.fw > Modula-3.fw
-> Modula-3.fw !chmod +w
-> Modula-3.fw <
```

After suitable modification, '`Modula-3.fw`' could be combined with the library specification '`$/Unparser/Analysis.fw`' to define the new unparser generator. Thus you might create a file called '`M3.specs`' with the following content:

```
Modula-3.fw
$/Unparser/Analysis.fw
```

The unparser generator could then be derived from '`M3.specs`' as usual (see Section "exe – Executable Version of the Processor" in *Products and Parameters Reference*):

```
-> M3.specs :exe
```

3 Changing Structure or Representation

The computation of `IdemPtg` in a given context can be decomposed into two tasks: collecting the `IdemPtg` attribute values from the children, and combining those values into a representation of the current context. Methods for attribute value collection depend on the tree grammar, and are embodied in LIDO computations. Methods for combining values, on the other hand, depend on the desired form of the unparsed text. They are embodied in PTG patterns.

There are two ways to override the output defined by the `IdemPtg` attribute at a given node:

1. Override the PTG pattern associated with that node
2. Override the computation of the `IdemPtg` attribute in the associated rule

The first method should be used when the change is simply one of format (adding constant strings, changing the order of the components, or omitting components). When it is necessary to add significant content to the unparsed representation of a node, then the second method should be used. Any arbitrary computation yielding an object of type `PTGNode` can be carried out, using any information at the processor's disposal. (Such a solution usually *also* requires overriding of the pattern.)

3.1 Overriding PTG patterns

The generated unparser specification contains a PTG pattern for each non-literal terminal symbol and each LIDO rule in the definition of the tree grammar. Each pattern name is the name of the construct (non-literal terminal or rule), preceded by a prefix followed by an underscore. The default prefix is `Idem`.

All of the non-literal terminal symbols are represented by patterns of the following form (‘`name`’ is the non-literal terminal symbol):

```
Idem_‘name’: [PtgOutId $ int]
```

This pattern is a single function call insertion (see Section “Function Call Insertion” in *PTG: Pattern-based Text Generator*). `PtgOutId` is a function exported by the PtgCommon module (see Section “Commonly used Output patterns for PTG” in *Tasks related to generating output*). Its argument is assumed to be a string table index (see Section “Character String Storage” in *Library Reference Manual*) and it outputs the indexed string.

This default pattern for a non-literal terminal symbol assumes that the value of that symbol is, in fact, a string table index. If the internal representation of the symbol was created by either the token processor `mkidn` (see Section “Available Processors” in *Lexical Analysis*) or the token processor `mkstr`, this will be the case.

In the expression language specification, `mkidn` is used to establish the internal representation of an `Identifier`, and `mkstr` is used to establish the internal representation of an `Integer`. Suppose, however, that the internal representation of an `Integer` was created by the token processor `mkint`. In that case, the user would have to provide the following PTG pattern to override the normal pattern generation.

```
Idem_Integer: $ int
```

It is vital to ensure that the PTG pattern associated with a non-literal terminal symbol is compatible with the token processor creating the internal representation of that symbol.

The only differences between the infix and postfix representations of an expression tree are in the literal terminal symbols reconstructed by the textual unparser (parentheses appear in an infix representation but not in a postfix representation) and in the order in which values are combined (operators between operands in an infix representation but following them in a postfix representation). Thus we can override the PTG patterns generated from the expression language definition to produce a postfix unparser:

```
Idem_PlusExp: $1 $2  "+" [Separator]
Idem_StarExp: $1 $2  "*" [Separator]
Idem_Parens:  $1
Idem_CallExp: $1 $2 [Separator]
```

Earlier (see Chapter 1 [Using an Unparser], page 3), we used a LIDO computation to ensure that a textual unparser generated from the expression language definition separated the arguments of a call with commas. The same effect can be achieved by simply overriding the PTG pattern that defines the "combine" function of the computation inherited by `Arguments`:

```
Idem_2ArgList: $ { "," [Separator] } $
```

As usual, an invocation of `Separator` follows the terminal symbol `,`.

In some situations, it is necessary to omit one or more children of a node. This cannot be done simply by omitting indexed insertion points from the appropriate PTG pattern, because PTG determines the number of arguments to the generated function from the set of insertion points. An invocation of the generated function, with one argument per child, already appears in the computation for the node. Thus any change in the number of insertion points would result in a mismatch between the number of parameters to the function and the number of arguments to the call.

A child can be omitted from the unparsed tree by "wrapping" the corresponding indexed insertion point in the PTG pattern ('`i`' is the integer index):

```
[ IGNORE  $‘i’  ]
```

`IGNORE` is a macro defined in the generated FunnelWeb file. It does nothing, so the effect is that the indexed sub-tree does not appear in the unparsed output.

3.2 Changing `IdemPtg` computations

The unparser generator implements the computation of the `IdemPtg` attribute as a class symbol computation. This class symbol computation can be overridden either by a tree symbol computation or by a rule computation (see Section "Inheritance of Computations" in *LIDO - Reference Manual*).

When overriding the default computation for an `IdemPtg` value, it is often convenient to be able to write the new computation in terms of the overridden value. Thus the unparser generator actually produces two class symbol computations: The `IdemOrigPtg` attribute of the class symbol is first computed by applying the appropriate PTG function to the `IdemPtg` attributes of the children. Then the `IdemPtg` attribute of the class symbol is assigned the value of the `IdemOrigPtg` attribute of the class symbol.

To see how `IdemPtg` and `IdemOrigPtg` could be used when an unparser's behavior must be changed, suppose that the `Parens` rule were omitted from the definition of the expression language. In that case, the unparser has no information about parentheses present in the original input text. Thus a pretty-printer would fail to output parentheses that were necessary to override the normal operator precedence and association in certain expressions, changing the meaning of those expressions.

Here is a simple tree symbol computation to ensure that the unparsed form has the same meaning as the original tree. It overrides the class symbol computation for `IdemPtg` that was produced by the unparser generator by a tree symbol computation:

```
SYMBOL Expression COMPUTE
  SYNT.IdemPtg=PTGParen(THIS.IdemOrigPtg);
END;
```

`PTGParen` is defined by the pattern:

```
Paren: "(" $ ")"
```

This specification puts parentheses around *every* expression, which certainly preserves the meaning but may make the result hard to read. A more readable representation could be created by parenthesizing only those expressions containing operators:

```
RULE: Expression ::= Expression Operator Expression
COMPUTE
  Expression[1].IdemPtg=PTGParen(Expression[1].IdemOrigPtg);
END;

RULE: Expression ::= Operator Expression
COMPUTE
  Expression[1].IdemPtg=PTGParen(Expression[1].IdemOrigPtg);
END;
```

This illustrates the use of rule computations to override the generated class symbol computation.

A comma-separated argument list can be produced by overriding the computation of `IdemOrigPtg` (or `IdemPtg`, see Chapter 1 [Using an Unparser], page 3):

```
RULE: Arguments LISTOF Expression
COMPUTE
  Arguments.IdemOrigPtg=
    CONSTITUENTS Expression.IdemPtg SHIELD Expression
      WITH (PTGNode, PTGArgSep, IDENTICAL, PTGNull);
END;
```

4 Deriving an Unparser

Recall the example of the pretty-printer that was defined by the file following type-‘specs’ file (see Chapter 1 [Using an Unparser], page 3):

```
example.fw
example.fw :idem
Add.fw
```

The first line is the name of a file defining a processor that builds a tree from a sentence in the expression language. The second line is a request to derive a textual unparser from the definition of the expression language. Finally, the third line is the name of a file containing the computation that outputs the unparsed tree. These three lines constitute the complete definition of the pretty-printer, which could be derived from this type-‘specs’ file in the usual way.

Here we are concerned only with the problem of *deriving* an unparser, exemplified by the second line above. Such a derivation always yields a FunnelWeb file that defines the desired unparser. Since the derivation occurs as a component of a type-‘specs’ file, the derived unparser becomes a component of the processor defined by that type-‘specs’ file.

All of the information needed to construct the unparser must be derivable from its *basis* (file ‘example.fw’ in this case). Different derivations are applied to the basis to create different kinds of unparsers, to control the representation language of the unparsed text, and to obtain a definition of the output structure.

4.1 Establishing a basis

In the simplest case, the only information needed to derive an unparser is the tree grammar rules defining the set of trees to be unparsed.

Since the generated unparser will be a component of some processor, all of the rules defining trees to be unparsed must be valid rules of the tree grammar for that processor. The easiest way to satisfy this requirement is for the basis of the unparser derivation to define a complete tree grammar for the processor. This is the situation in our example; file ‘example.fw’ defines the complete tree grammar for the expression language and therefore for the pretty-printer. (See Section 4.3 [Deriving structural unparsers], page 22, for applications in which unparsers are derived from parts of the tree grammar for a processor.)

Suppose that we were to create a file ‘evaluate.fw’ containing computations that evaluate sentences in the expression language. A “desk calculator” could then be defined by a file ‘calculator.specs’ with the content:

```
example.fw
evaluate.fw
```

In this case, ‘calculator.specs’ still defines the complete tree grammar for the expression language. Thus the following type-‘specs’ file would define a processor that reads sentences in the expression language, evaluates them, and prints them in a standard format:

```
calculator.specs
calculator.specs :idem
Add.fw
```

The situation is more complex when some PTG patterns must be overridden to obtain the desired output. Overriding patterns must be specified as part of the basis from which the unparser is derived, and they will be incorporated into the generated unparser definition.

One way to include overriding PTG patterns in the basis of an unparser derivation is to make them a part of the overall processor specification. Thus, for example, they could be included in '`example.fw`' of the specifications above. Then either of the derivations shown (the one based on '`example.fw`' or the one based on '`calculator.specs`') would produce an unparser with the specified patterns overridden. It is important to note that the tree grammar and the PTG patterns are the *only* things defined by '`calculator.specs`' (or by '`example.fw`' in the earlier derivation) that are relevant to deriving an unparser. All other information is ignored. PTG patterns whose names do not match prefixed rule names from the tree grammar are also ignored.

It is often a violation of modularity to combine overriding patterns with the overall processor specification. For example, consider an unparser that outputs a postfix representation of a sentence in the expression language (see Section 3.1 [Overriding PTG patterns], page 15). The overriding patterns are specific to this particular processor, and have nothing to do with the definition of the expression language itself. Including them in '`example.fw`' would pollute the language specification, tying it to this application.

We can easily avoid this violation of modularity by adding a `patterns` parameter to '`example.fw`' to form the basis of the derivation. First, the overriding patterns are defined in a file named (say) '`Postfix.ptg`':

```
Idem_PlusExp: $1 $2  "+" [Separator]
Idem_StarExp: $1 $2  "*" [Separator]
Idem_Parens:  $1
Idem_CallExp: $1 $2      [Separator]
```

This file is then supplied as the value of the `patterns` parameter (see Section "Parameterization Expressions" in *Eli User Interface Reference Manual*):

```
example.fw +patterns=(Postfix.ptg)
```

The unparser derivation would then be:

```
example.fw +patterns=(Postfix.ptg) :idem
```

A complete processor accepting a sentence in the expression language and printing its postfix equivalent in standard form would then be defined by the following type-'`specs`' file:

```
example.fw
example.fw +patterns=(Postfix.ptg) :idem
Add.fw
```

A basis may include any number of file-valued `patterns` parameters. Only the PTG patterns defined by these files are relevant to the unparser generation; all other information is ignored. PTG patterns whose names do not match prefixed rule names from the tree grammar are also ignored.

Any unparser can be derived with a prefix other than `Idem` for the type-`PTGNode` attributes and PTG patterns that it creates. The desired prefix forms part of the basis from which the unparser is derived. This feature is useful if an application involves more than one unparser (see Section 4.5 [Deriving multiple unparsers], page 23).

The desired prefix is supplied as the value of the `prefix` parameter. For example, the basis for an expression language unparser computing the attributes `TargetPtg` and `TargetOrigPtg` instead of `IdemPtg` and `IdemOrigPtg` would be:

```
example.fw +prefix=Target
```

All PTG pattern names in an unparser derived from this basis would begin with `Target_` . Thus, if we wished to override the generated patterns in order to produce postfix, the overriding pattern names in '`Postfix.ptg`' would have to reflect the new prefix:

```
Target_PlusExp: $1 $2  "+" [Separator]
Target_StarExp: $1 $2  "*" [Separator]
Target_Parens:  $1
Target_CallExp: $1 $2      [Separator]
```

The basis of such an unparser consists of the the specification file for the tree grammar, modified by the two parameters (which may be given in any order see Section "Parameterization Expressions" in *Eli User Interface Reference Manual*):

```
example.fw +prefix=Target +patterns=(Postfix.ptg)
```

In the remainder of this document, '`Basis`' will be used to denote the basis of an unparser derivation. As we have seen in this section, '`Basis`' consists of a single file defining a tree grammar, possibly parameterized by a set of overriding PTG patterns and/or a prefix to replace the default `Idem`.

4.2 Deriving textual unparsers

A textual unparser is constructed by deriving the `:idem` product:

```
Basis :idem
```

The result of this derivation is a FunnelWeb file defining a textual unparser. That FunnelWeb file contains:

- Named LIDO rules for the tree grammar specified by '`Basis`'.
- PTG patterns either generated from the tree grammar (see Section 2.1 [Textual unparser], page 7) or supplied as part of '`Basis`' (see Section 4.1 [Establishing a basis], page 19).
- `CLASS SYMBOL` computations for `IdemPtg` and `IdemOrigPtg` (see Section 2.1 [Textual unparser], page 7). Those computations invoke PTG routines generated from the patterns. The prefix of the attribute and routine names is either `Idem` or the string specified via a `prefix` parameter in '`Basis`' (see Section 4.1 [Establishing a basis], page 19).
- A definition of the `IGNORE` macro (see Section 3.1 [Overriding PTG patterns], page 15).
- Invocations of the `PtgCommon` (see Section "Commonly used Output patterns for PTG" in *Tasks related to generating output*) and `Separator` (see Section "Introduce Separators in PTG Output" in *Tasks related to generating output*) library modules.

A PostScript version of the unparser definition can also be derived for documentation purposes:

```
Basis :idem :fwTex :ps
```

4.3 Deriving structural unparsers

A structural unparser is constructed by deriving the `:tree` product:

```
Basis :tree
```

The result of this derivation is a FunnelWeb file defining a structural unparser. That FunnelWeb file contains:

- Named LIDO rules for the tree grammar specified by '`Basis`'.
- PTG patterns either generated from the tree grammar (see Section 2.1 [Textual unparser], page 7) or supplied as part of '`Basis`' (see Section 4.1 [Establishing a basis], page 19).
- `CLASS SYMBOL` computations for `IdemPtg` and `IdemOrigPtg` (see Section 2.1 [Textual unparser], page 7). Those computations invoke PTG routines generated from the patterns. The prefix of the attribute and routine names is either `Idem` or the string specified via a `prefix` parameter in '`Basis`' (see Section 4.1 [Establishing a basis], page 19).
- A definition of the `IGNORE` macro (see Section 3.1 [Overriding PTG patterns], page 15).
- Invocations of the `PtgCommon` (see Section "Commonly used Output patterns for PTG" in *Tasks related to generating output*) and `BlockPrint` (see Section "Typesetting for Block Structured Output" in *Tasks related to generating output*) library modules.

A PostScript version of the unparser definition can also be derived for documentation purposes:

```
Basis :tree :fwTex :ps
```

A structural unparser produces a description of the tree in some language. Recall that a generic functional representation is used by default. Any other standard representation language can be specified by supplying an appropriate value of the `lang` parameter to the derivation. For example, the following derives a structural unparser producing a description of the tree in XML:

```
Basis +lang=XML :tree
```

See Section 2.2.3 [Languages describing tree structure], page 13, for a list of the standard representation languages.

When none of the standard representation languages is appropriate, you can specify your own unparser generator. This unparser generator can be invoked by supplying its executable file to the derivation as the value of the `lang` parameter.

The most common way to specify a new unparser generator is to modify an existing specification and then use Eli to produce an executable file from that modified specification. We have already given an example of this technique (see Section 2.2.3 [Languages describing tree structure], page 13). In that example, file '`M3.specs`' defined a generator producing an unparser that represents a tree by a Modula-3 program. The executable version of that generator could be obtained in the usual way by deriving the `exe` product from '`M3.specs`' (see Section "exe – Executable Version of the Processor" in *Products and Parameters Reference*). Thus the following derivation would create a Modula-3 unparser for the trees defined by '`Basis`':

```
Basis +lang=(M3.specs :exe) :tree
```

Here the executable file supplied as the value of the `lang` parameter is the one derived from the specification of the Modula-3 unparser generator.

4.4 Obtaining the structure definition

Structural unparser generators producing application-language code also deliver a definition of the data structure(s) described by that code. For example, an unparser generator producing tree descriptions in XML also delivers a "document type declaration" (DTD) file defining a grammar for those descriptions. Here's the DTD file for the expression language:

```
<!ENTITY % Axiom "(rule_000)">
<!ENTITY % Expression
  "(PlusExp | StarExp | Parens | IdnExp | IntExp | CallExp)">
<!ENTITY % Arguments "(ArgList)">
<!ELEMENT rule_000 (%Expression;)>
<!ELEMENT PlusExp (%Expression;, %Expression;)>
<!ELEMENT StarExp (%Expression;, %Expression;)>
<!ELEMENT Parens (%Expression;)>
<!ELEMENT IdnExp (#PCDATA)>
<!ELEMENT IntExp (#PCDATA)>
<!ELEMENT CallExp (#PCDATA, %Arguments;)>
<!ELEMENT ArgList ((%Expression;)*)>
```

This definition depends only on the tree grammar, not on any particular tree defined by that grammar. Thus it is built separately:

```
Basis +lang=XML :treeStruc
```

The `treeStruc` product is a set of files. Both the number of files in that set and their types depend on the particular structural unparser being generated. For example, the set is empty for the generic functional representation. The XML unparser generator produces a single DTD file, and the Java unparser generator produces one type-'`java`' file for every class in the representation.

You can list the files in the set with the following request:

```
Basis +lang=XML :treeStruc :ls >
```

To obtain copies of the definition files, make a copy of the set itself (see Section "Extracting and Editing Objects" in *Eli User Interface Reference Manual*):

```
Basis +lang=XML :treeStruc > Structure
```

(This request copies the generated files into a sub-directory named '`Structure`' of the current directory; the destination name '`Structure`' could be replaced by any directory name. The directory must exist before the request is made.)

4.5 Deriving multiple unparsers

Consider a translator that builds a target program tree corresponding to the source program presented to it. Perhaps we would like to make that translator output a listing of the source text formatted according to standard rules and also an XML file that defined the target program tree. This can be done by including two unparsers, one textual and the other structural.

To make the discussion concrete, let '`Source_i.specs`' define a processor that reads a sentence in language '`i`' and builds a corresponding decorated tree. '`Translator.fw`' specifies computations over such a source program tree that build a target program tree according to the structure defined by file '`Target_j.specs`'. File '`Translator.specs`', consisting of the following three lines, would then define a translator that would build a target program tree corresponding to a sentence in language ‘i’:

```
Source_i.specs
Target_j.specs
Translate.fw
```

If the root of the tree grammar defined in ‘Source_i.specs’ is Source, and the root of the tree grammar defined in ‘Target_j.specs’ is Target, then ‘Translate.fw’ might contain the following LIDO computation:

```
RULE Axiom: Root ::= Source $ Target
COMPUTE
  Target.GENTREE=Source.Code;
END;
```

This computation takes the target program tree that has been built as the value of attribute Source.Code, and makes it the second child of the root node (see Section “Computed Subtrees” in *LIDO - Reference Manual*).

Given ‘Translator.specs’, one way to define a processor producing a listing of the source text formatted according to standard rules and also an XML file defining the target program tree would be to write the following type-‘specs’ file:

```
Translator.specs
Translator.specs                          :idem
Translator.specs +prefix=Target +lang=XML :tree
Add.fw
```

The first line of this file defines the translator itself, and the second line defines a textual unparser computing IdemPtg attributes. A structural unparser computing TargetPtg attributes that hold XML representations of their nodes is defined by the third line. Two of the additional computations defined by the last line of this file might be:

```
SYMBOL Source COMPUTE Sep_Out   (      THIS.IdemPtg);   END;
SYMBOL Target COMPUTE BP_OutFile("xml",THIS.TargetPtg); END;
```

These computations will write the pretty-printed source program to the standard output, and the XML representation of the target program tree to file ‘xml’.

4.6 Deriving unparsers from sub-grammars

Suppose that the tree grammars defined by ‘Source_i.specs’ and ‘Target_j.specs’ in the example of the previous section are disjoint. In that case, the processor defined there will compute unnecessary IdemPtg attributes for target tree nodes and unnecessary TargetPtg attributes for source tree nodes. These unnecessary computations can be avoided by simply changing the type-‘specs’ file to derive each unparser from the tree grammar to which it applies:

```
Translator.specs
Source_i.specs                          :idem
Target_j.specs +prefix=Target +lang=XML :tree
Add.fw
```

Note that no other changes are needed in any of the files.

Each of the two tree grammars on which the unparsers are based defines a complete, rooted sub-tree of the complete tree. Moreover, because the tree grammar defined by '`Target_j.specs`' describes a tree created by attribution, each of its rules has been given an explicit name (see Section "Tree Construction Functions" in *LIDO - Reference Manual*).

The fact that no more than one of the tree grammars contains unnamed rules is crucial to the success of the complete processor derivation. Recall that an unparser definition contains the definition of the tree grammar on which it is based, and every rule in that tree grammar is named. If the names were not explicit in the unparser's basis, the names in the unparser definition will have been created as part of the unparser generation. The same name creation process is applied during every unparser generation, and therefore if two unparsers generated from disjoint tree grammars with unnamed rules are combined there will be name clashes.

Index