Tasks related to generating output

Uwe Kastens

University of Paderborn
D-33098 Paderborn
FRG

$Revision: 1.7 $

Table of Contents

This library contains the following modules:

`LeafPtg` PTG Output for Leaf Nodes

`PtgCommon`
Commonly used Output patterns for PTG

`Indent` Indentation for PTG Output

`OutStr` Output String Conversion

`PrettyPrint`
PrettyPrinting for PTG Output

`BlockPrint`
Typesetting for Block-Structured Output

`StringOut`
Process PTG-Output into string buffers

`Separator`
Introducing Separators in PTG Output

1 PTG Output for Leaf Nodes

The module `LeafPtg` provides some standard translations of terminal input strings into output strings. They may be attached to nonterminal symbols having an `int` valued attribute `Sym` which represents a string, an identifier or a number. That symbol usually derives to a terminal symbol which is used to compute the value of the `Sym` attribute.

The module does not have generic parameters. It is used by writing

```
$/Output/LeafPtg.fw
```

in a `.specs` file.

The computations of this module yield an attribute `Ptg` of type `PTGNode` representing the desired output string. It may be used to compose more complex output structures.

The required computation of the `Sym` attribute can be specified by (if not already done for other purposes):

```
ATTR Sym: int;
SYMBOL T COMPUTE SYNT.Sym = TERM; END;
```

There are the following symbol roles for different output representations:

```
SYMBOL T INHERITS  IdPtg  END;
```

`T.Sym` must refer to an identifier or a string then `T.Ptg` represents the same identifier or character sequence of the string in the output.

```
SYMBOL T INHERITS  CStringPtg  END;
```

`T.Sym` must refer to a string, then `T.Ptg` represents its value as a C string literal.

```
SYMBOL T INHERITS  PStringPtg  END;
```

`T.Sym` must refer to a string, then `T.Ptg` represents its value as a Pascal string literal.

```
SYMBOL T INHERITS  NumPtg  END;
```

`T.Sym` must represent an integral number, then `T.Ptg` represents that number.

The module uses some PTG patterns from the module `PtgCommon`.

Please note that this module may become outdated in future Eli-Versions. In new specifications use the module `PtgCommon` directly to create PTG representations for terminal symbols. The module `PtgCommon` is described in the next section.

2 Commonly used Output patterns for PTG

The module `PtgCommon` provides definitions for frequently used PTG patterns. These patterns fall into two categories: The first one supports different types of output leaves; the second contains frequently used patterns for building sequences.

The module does not have generic parameters. It is used by including

```
$/Output/PtgCommon.fw
```

in a `.specs` file.

The module introduces PTG specifications for patterns named `Id, AsIs, CString, PString, CChar, Numb, Seq, CommaSeq`. When using `PtgCommon.fw` these names must not be specified in any other PTG specification.

The functions provided by this module may be used in '`.lido`' specifications or in '`.c`' files. To introduce prototypes for the defined functions, use the header file `PtgCommon.h`.

Frequently Used Patterns

The module `PtgCommon` provides useful and commonly used PTG patterns, especially for the output of non-literal terminal symbols. They are documented both by showing their PTG pattern definitions and the signature of the resulting pattern functions:

`Pattern: Id: [PtgOutId $ int]`
`Resulting Function: PTGNode PTGId(int id)`

The argument `id` must refer to an identifier or a string stored in the character storage module of Eli, see Section 'Character Storage Module' in Library Reference Manual. The PTG pattern produces the same identifier or character string in the output.

`Pattern: AsIs: $ string`
`Resulting Function: PTGNode PTGAsIs(char *string)`

The PTG pattern produces the specified argument `string` in the output. The character string is not copied, only the pointer is.

Pattern: `Numb: $ int`
Resulting Function: `PTGNode PTGNumb(int numb)`

The PTG pattern produces the given integral number.

Pattern: `CString: [CPtgOutstr $ string]`
Resulting Function: `PTGNode PTGCString(char *string)`
and `PTGNode PTGCStringId(int id)`

The argument is a string. The PTG pattern function produces the same character string quoted according to the rules of the C language. `PTGCStringId` is macro based on `PTGCString`. It takes an index into the character storage module of Eli, see Section 'Character Storage Module' in Library Reference Manual. It produces the string stored there quoted according to the rules of the C language..

Pattern: `PString: [PPtgOutstr $ string]`
Resulting Function: `PTGNode PTGPString(char *string)`
and `PTGNode PTGPStringId(int id)`

The argument is a string. The PTG pattern function produces the same character string quoted according to the rules of the Pascal language. `PTGPStringId` is macro based on `PTGPString`. It takes an index into the character storage module of Eli, see Section 'Character Storage Module' in Library Reference Manual. It produces the string stored there quoted according to the rules of the Pascal language..

Pattern: `CChar: [CPtgOutchar $ int]`
Resulting Function: `PTGNode PTGCChar(int c)`

The PTG pattern produces the specified value as C character literal.

Pattern: `Seq: $ $`
Resulting Function: `PTGNode PTGPSeq(PTGNode, PTGNode)`

Takes two arguments and yields a new node that prints the concatenation of the given patterns.

Pattern: `CommaSeq: $ { ", " } $`
Resulting Function: `PTGNode PTGPCommaSeq(PTGNode, PTGNode)`

Takes two arguments and yields their concatenation separated by a comma. By enclosing the separator with braces, it is assured that a comma will be printed, only if none of the arguments refers to the predefined value `PTGNULL` that yields an empty output. This makes the pattern well suited to be used in conjunction with the `CONSTITUENTS` construct. See Section 'Optional Parts in Patterns' in Pattern-based text generator, for details.

Pattern: `Eol: $ "\n"`
Resulting Function: `PTGNode PTGEol(PTGNode)`

This pattern attaches a newline at the end of the given text. Note that the PTG output functions do not automatically put a newline at the end of the output.

Useful Embedded Functions

The functions embedded in the PTG patterns defined in `PTGCommon.fw` (See ⟨undefined⟩ [Frequently Used Patterns], page ⟨undefined⟩) might also be useful in user defined patterns. These functions are:

`void PtgOutId (PTG_OUTPUT_FILE fs, int c);`
: takes an index into the character storage module of Eli, see Section 'Character Storage Module' in Library Reference Manual. It outputs the string stored there.

`void (PTG_OUTPUT_FILE fs, char *s);`
: takes a string argument and outputs the same string quoted according to the rules of the C language.

`void (PTG_OUTPUT_FILE fs, int c);`
: takes an integer character code and outputs the character.

`void (PTG_OUTPUT_FILE fs, int c);`
: takes a string argument and outputs the same string quoted according to the rules of the Pascal language.

Examples

The first example will use a PTG pattern that prints an identifier or a floating point number. This is done by defining the symbol role `PtgLeaf` that computes a `ptg` attribute. It generates the source text of the identifier when processed through a ptg processing function. This role can be inherited by a tree symbol that appears directly above a terminal, that was processed through the `mkidn` gla processor.

```
CLASS SYMBOL PtgLeaf: ptg: PTGNode;

CLASS SYMBOL PtgLeaf COMPUTE
   THIS.ptg = PTGId(TERM);
END;
```

In this example, the class symbol `PtgLeaf` can be used later to denote all the different grammar symbols that compute ptg leaf patterns. This is done for example in the second example, that computes a PTG pattern that prints all occurrences of `PtgLeaf` in a comma separated list.

```
CLASS SYMBOL LeafCommaList: ptg: PTGNode;
CLASS SYMBOL LeafCommaList COMPUTE
```

```
    THIS.ptg =
      CONSTITUENTS PtgLeaf.ptg
      WITH (PTGNode, PTGCommaSeq, IDENTICAL, PTGNull);
END;
```

Please refer to Section 'Symbol Specifications' in Lido Reference Manual, for an explanation of symbol computations, see Section 'CONSTITUENT(S)' in Lido Reference Manual, for an explanation of the `CONSTITUENT(S)`-construct and read Section 'Predefined Entities' in Lido Reference Manual, for an explanation of the predefined `IDENTICAL`-function.

Special Situation when Using C String Literals

A special situation occurs, if C string literals are input tokens and are to be reproduced identically. Two different token processors can be used to store the string: If the `c_mkstr` processor is specified in a '`.gla`' file,

```
CStringLit: $\" (auxCString) [c_mkstr]
```

the string is interpreted and the result is stored. Such a string can be processed with the pattern functions `PTGCStringId()` and `PTGPStringId()` to yield C or Pascal string literals on output. However, as strings are null terminated in Eli, the first occurrence of an embedded zero-character terminates the string, and the result is truncated. A solution for this would be to not interpret the control sequences and to store the string verbatim as it is provided on input. That is achieved by the token processor `mkstr`:

```
CStringLit: $\" (auxCString) [mkstr]
```

As control sequences are not interpreted by `mkstr`, `PTGPString` and `PTGCString` can not be used to produce an identical output string. Instead, the pattern function `PTGAsIs` is to be used. Since the latter token processor can handle embedded zero characters, it is used in the canned description `C_STRING_LIT` for C string literals. See Section 'Canned Descriptions' in Lexical Analysis.

3 Indentation

The module `Indent` supplies a C module that implements some functions helpful for indenting text produced by PTG functions. The function names `IndentIncr`, `IndentDecr`, `IndentNewLine` can be inserted into the user's PTG specification, like

```
Block: "{" [IndentIncr] $ [IndentDecr] [IndentNewLine] "}"
Stmt:  [IndentNewLine] $ ";"
```

Those PTG specifications should not contain strings with the new line character, but should have the `[IndentNewLine]` call instead.

Use the function `IndentNewLine` to put a linefeed into the output and indent the next line. `IndentIncr` increments and `IndentDecr` decrements the indentation level. The width of a single indentation step may be set to `n` spaces by the call `IndentSetStep (n)` executed prior to initiating output (e.g. by PTGOut).

The module does not have generic parameters. It is used by writing

```
$/Output/Indent.fw
```

in a `.specs` file.

4 Output String Conversion

This module provides a set of functions that transform character values and character string values into C or Pascal literals.

The module does not have generic parameters. It is used by writing

```
$/Output/OutStr.fw
```

in a `.reqmod` (or a `.specs`) file.

The module exports the following functions:

`void C_outstr (FILE *fs, char *s)`
: Translates `s` into a C string literal and outputs it on file `fs`.

`void C_outchar (FILE *fs, char *s)`
: Translates `s` into a C character literal and outputs it on file `fs`.

`void P_outstr(FILE *fs, char *s)`
: Translates `s` into a Pascal string literal and outputs it on file `fs`.

`void outstr (FILE *fs, char *s)`
: Outputs `s` without translation on file `fs`.

5 Pretty Printing

The module 'PrettyPrint' supplies C functions that can be inserted in PTG patterns to handle line breaks properly. The functions try to break the current line at the last possible position that precedes the maximum line width. Furthermore, regions of text can be indented.

Functions exist to mark line breaks and the begin and end of an indentation region. These functions can be included into PTG pattern definitions.

`PP_BreakLine`
: Specifies, that a line break can be inserted at this point. A line will only be broken at these points.

`PP_Indent`
`PP_Exdent`
: `PP_Indent` marks the beginning of an indented region. Line feeds following this function call will not only begin a new line but also indent the next line by an indentation step, the width of indentation can be adjusted with a function discussed later. Indented regions can be nested and are terminated by a call to the `PP_Exdent`-pattern function.

`PP_Newline`
: Forces a line feed thereby inserting the newline sequence. The next line will be indented properly. The newline character '`"\n"`' in a PTG pattern is a shortcut for a call to this output function.

Examples

The following PTG patterns can be used to yield different styles of indenting for blocks. Here `{` and `}` are the symbols that denote the beginning and end of a block, `$` is the insertion point for the indented block. The first example sets those symbols in a new line at the indentation level of the outer block:

```
Block:  "\n{" [PP_Indent]
            "\n" $ [PP_Exdent]
        "\n}"
Stmt:   [PP_BreakLine] $
```

The next example also specifies an indented region. Here, the opening brace is set as last token outside the block, separated with whitespace instead of a newline:

```
Block:  " " [PP_BreakLine] "{" [PP_Indent]
           "\n" $ [PP_Exdent]
        "\n}"
```

The third example uses the indentation style commonly known as the GNU indentation style. Here, the braces are set on a new line, indented two positions. The indented region then follows indented four positions. To use this, set the indentation width to two by one of the function calls discussed later. Then use the following pattern:

```
Block:  [PP_Indent] "\n{" [PP_Indent]
            "\n" $ [PP_Exdent]
        "\n}" [PP_Exdent]
```

Additional functions

Additional functions exist to influence the behavior of the module.

`PP_SetLineWidth(int width)`
: Sets the linewidth to the specified value. The default is 80.

`PP_SetSoftBreakShortcut(char)`
: Assigns a character that should behave like a call to `PP_LineBreak`. A good choice for this would be the tab character. The default is set to the null character what disables substitution.

`PP_SetIndentationWidth(int width)`
: Sets the amount to indent in one indentation level. Indentation is normally done by spaces. If a negative value is used, a tab character will be used for one indentation step (counting as 8 character positions).

`PP_SetEndline(char *endline)`
: Assigns the given string to be used as end-of-line sequence. Default for this is '`"\n"`'. Another good choice would be '`"\r\n"`'.

All these functions need to be called prior to the start of the output with one of the following functions. They replace the PTG generated ones, if PrettyPrinting should be used.

`PP_OutFPtr(FILE *f, PTGNode n)`
: Outputs the given '`PTGNode`' to the given file that must have been opened for output.

`PP_OutFile(char *filename, PTGNode n)`
: Outputs the given '`PTGNode`' to the named file.

`PP_Out(PTGNode n)`
: Outputs the given '`PTGNode`' to the standard output.

Usage of Module

To use the pretty printing module, simply include it's name in one of the `.specs` files:

```
$/Output/PrettyPrint.fw
```

Restrictions

In two cases it is possible that an output line exceeds the given maximal length:

- A sequence of characters longer than the specified linewidth is output without intermediate call to `PP_LineBreak`.
- A PTG Pattern contains tab characters that will be counted to have a width of 1 which of course is not always true.

Additional information about this module and it's implementation can be obtained by the derivation

```
$elipkg/Output/PrettyPrint.fw :fwTexinfo :display
```

6 Typesetting for Block Structured Output

The module 'BlockPrint' supplies C functions that can be inserted in PTG patterns for block formatting. It is the aim of this module to print all the text between the block marks into one line. If that does not succeed, all embedded line breaks of the block are converted into newlines. Additionally, blocks can be nested and blocking can be combined with indentation.

There are functions to mark line breaks and the beginning and end of a block. These functions can be included into PTG Pattern definitions.

`BP_BreakLine`
: Specifies, that a LineBreak can be inserted at this point. A line will only be broken at these points.

`BP_BeginBlock`
`BP_EndBlock`
: Marks the beginning and end of a Block. If the text until the call to `BP_BlockEnd` has room in the current line, all line breaks will be discarded. Else, all embedded line breaks will be converted into newlines.

`BP_BeginBlockI`
`BP_EndBlockI`
: Same as the above. The block created by this pair of functions will additionally be indented by one indentation step.

`BP_Newline`
: Forces a linefeed thereby inserting the newline sequence. Note that with the presence of this pattern function, the enclosing block is automatically tagged as 'too long' and all the remaining Line breaks in the current block are also converted to newlines. The next line will be indented properly. The newline character '`"\n"`' in a PTG pattern is a shortcut for a call to this output function.

Examples

The following PTG patterns can be used to print nested C scopes with intermediate function calls. The statements in one block will be indented properly and always be separated by newlines. The arguments of a function call will be set into one line, if there is enough room. If not, newlines will be inserted between the arguments.

```
FCall:          "\n" $ string "(" [BP_BeginBlockI] $ [BP_EndBlockI] ");"
Arg:            $ { "," [BP_BreakLine] } $
Block:          "\n{" [BP_BeginBlockI]
                    $ [BP_EndBlockI]
                "\n}"
```

Additional functions

`BP_SetLineWidth(int width)`
: Sets the linewidth to the specified value. The default is 80.

`BP_SetSoftBreakShortcut(char)`
: Assings a character that should behave like a call to `BP_LineBreak`. A good choice for this would be the tab character. The default is set to the null character what disables substitution.

`BP_SetIndentationWidth(int width)`
: Sets the amount to indent in one indentation level. Indentation is normally done by spaces. If a negative value is used, a tab character will be used for one indentation step (counting as 8 character positions).

`BP_SetEndline(char *endline)`
: Assigns the given string to be used as end-of-line sequence. Default for this is `"\n"`. Another good choice would be `"\r\n"`.

All these functions need to be called prior to start output with one of the following functions. They replace the PTG generated ones, if block printing is used.

`BP_OutFPtr(FILE *f, PTGNode n)`
: Output the given '`PTGNode`' to the given file that must have been opened for output.

`BP_OutFile(char *filename, PTGNode n)`
: Output the given PTGNode to the named file.

`BP_Out(PTGNode n)`
: Output the given PTGNode to the standard output.

Usage of Module

To use the block printing module, simply include it's name in one of the `.specs` files:

```
$/Output/BlockPrint.fw
```

Restrictions

In two cases it is possible that an output line exceeds the given maximal length:

A sequence of characters longer than the specified linewidth is output without intermediate call to `BP_BreakLine`.

A PTG Pattern contains tab characters that will be counted to have a width of 1 which of course is not always true.

Additional information about this module and it's implementation can be obtained by the derivation

```
$elipkg/Output/BlockPrint.fw :fwTexinfo :display
```

7 Processing Ptg-Output into String Buffers

The module `StringOut` provides a possibility of processing the output associated to a PTG node structure recursively into a string buffer. The buffer is maintained by calls to `Obstack`-module-functions. The module `PtgOutput` is used to coordinate overrides of the PTG output functions.

This module supplies two C functions:

`char *PTG_StringOut(PTGNode root);`
: Takes the root to a PTG node structure as argument. Invokes the PTG printing functions and processes the output into an automatically allocated and growing string buffer. Upon termination, a pointer to to the start of this buffer is returned.

`void FreeStringOut();`
: Invocations of the `PTG_StringOut()` function can consume quite a lot of memory. It is possible, that at some time, the string buffers created by this functions are no longer needed. The memory consumed by this buffers can be returned to the system by an invocation of the `FreeStringOut()` function. Please note, that this function frees the space used by all invocations of `PTG_StringOut()` together.

Additional information about this module and it's implementation can be obtained by the derivation

```
$elipkg/Output/StringOut.fw :fwTexinfo :display
```

Usage of Module

To include this module into your specification, simply add the following line to one of your '`.specs`'-files:

```
$/Output/StringOut.fw
```

Restrictions

This module can be included to a specification together with other applications of the `PtgOutput` module functions, e.g. `PrettyPrint` and `BlockPrint`. By doing so, it is possible to pretty print a

PTG node structure into a file or to process it into a string buffer. A combination, for example to pretty print a PTG node structure into character buffer, is not possible.

8 Introducing Separators in PTG Output

The 'Separator' module supplies functions to insert separator characters into the generated output in a context dependent fashion.

It provides the function Separator which is meant to be embedded in PTG patterns, e.g.

```
loop: "while" [Separator] $1 [Separator] $2 [Separator]
```

The insertions of Separator mark the positions in the generated output texts, where separator characters might be placed.

The decision whether a separator is needed must be made by the user-supplied function Sep_Print:

```
        Sep_Print(PtgFilePtr file, char *last, char *next)
        /* On entry-
         *   file points to the output file
         *   last points to the last string printed
         *   next points to the string about to be printed
         * On exit-
         *   An appropriate separator has been added to the output file
    ***/
```

Based on the textual context Sep_Print should decide whether a separator character is required and, if so, must insert an appropriate separator into the output stream.

The 'Separator' module provides the following output functions which must be used instead of the corresponding PTG 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 'Separator' module is used in conjunction with the Unparser Generator 'Idem' (see Section 'Abstract Syntax Tree Unparsing' in Abstract Syntax Tree Unparsing) to simplify pretty-printing of the output. Idem inserts calls to Separator after every literal and terminal symbol in the templates corresponding to the grammar rules.

Usage

To include this module into your specification simply add the following line to one of your '.specs'-files:

```
$/Output/Separator.fw
```

An example of a Sep_Print function that works well with a C-like language is provided as 'C_Separator.fw': 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.

'C_Separator.fw' can be included in your specifications by putting

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

into a '.specs'-file.

'C_Separator.fw' is also useful as an example how to develop your own Sep_Print functions if none of the available modules is satisfactory. The simplest approach is to modify 'C_Separator.fw'. 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 'Sep_Print' function, as described above.

Restrictions

Since the 'Separator' module uses its own PTG output functions

```
Sep_Out
Sep_OutFile
Sep_OutFPtr
```

as explained above, it cannot be combined with specifications that influence PTG output by redefining the PTG output macros (see Section 'Influencing PTG Output' in PTG: Pattern-based Text Generation).

The memory for storing the last string printed for the 'Sep_Print' function is restricted to 1024 characters. If the last string printed exceeds 1024 characters, only its last 1024 characters are stored and passed to 'Sep_Print'.

Index