PTG: Pattern-based Text Generator

U. Kastens

University of Paderborn
D-33098 Paderborn
FRG

$Revision: 2.10 $

Table of Contents

The Pattern-Based Text Generator PTG supports translations into any kind of structured text. The structure of the target text is described by a set of patterns. PTG generates a set of functions for them. They are called to compose an instance of the target structure which is then output.

PTG is suitable to produce any kind of target language, e. g. programs of any programming language, special purpose languages like TeX or PostScript, or just structured data or tables in textual form.

PTG is typically applied for tasks of a language processor's translation phase. The calls of the patterns functions are then used in LIDO specifications to describe the translation of certain tree contexts. PTG may as well be used in C functions or in stand-alone C programs that translate some data structure into an output text.

Reading the documentation online one can use the documentation browser's `Run` command to obtain a copy of the complete specification for the example described in this manual.

1 Introduction to PTG

A PTG specification is a set of named patterns describing the structure and textual components of an output text. They are contained in files of type .ptg. PTG generates a C module ptg_gen.[ch] that has one function for each pattern specified. Calls of these functions apply the patterns in order to compose an instance of the target text, which can be output by a call of PTG's output functions. Those functions may be used in LIDO specifications or in C modules which import the interface file of the generated module ptg_gen.h.

Consider the following simple example: Assume we want to produce parenthesized representations of binary trees like S-expressions in LISP:

```
((1.nil).(2.(3.nil))
```

We specify three named patterns, one for the parenthesized structure, one for the literal nil, and one for numbers:

```
Pair:   "(" $ "." $ ")"
Nil:    "nil"
Numb:   $ int
```

For each of these patterns PTG generates a function which yields an internal representation of a pattern application. The following nested calls produce the above output text:

```
PTGOut (
  PTGPair (
     PTGPair(PTGNumb(1), PTGNil()),
             PTGPair(PTGNumb(2), PTGPair(PTGNumb(3), PTGNil()))
     ));
```

Of course one may store intermediate results of pattern applications and defer output until the target text is copletely composed:

```
n1 = PTGPair (PTGNumb (1), PTGNil ());
n2 = PTGPair(PTGNumb(2), PTGPair(PTGNumb(3), PTGNil()));
PTGOut (PTGPair (n1, n2));
```

The benefits of using PTG can best be described by a comparison with using C printf functions directly. For the above example a C program would contain statements like

```
printf ("( %s. %s)", a, b);
```

where `a` and `b` are pointers to the strings to be inserted.

Such a statement implements `what` text to be generated by the format string and the arguments, `when` it is to be output by the placement of the statement within the program, and `how` the text is produced by format strings and output functions.

PTG separates the issues `what` and `when` by the pattern specifications (`what`) and the function calls (`when`). The calls produce an internal structure with all information necessary to output the text, rather than immediately outputting it. PTG automatically and efficiently implements the `how`.

2 Pattern Specifications

A pattern is specified by named sequence of C string literals and `$` tokens that denote insertion points, e.g.

```
Pair: "(" $ "." $ ")" /* S-expression */
```

C style comments may be inserted anywhere in a PTG specification.

The pattern describes an output text that consists of the specified sequence of strings with the results of pattern applications being inserted at each insertion point.

A pattern is applied by calling a PTG generated function that has the name of the pattern preceeded by `PTG`, `PTGPair` in this case. The result of such a call yields a pointer of type `PTGNode` which represents that pattern application.

The pattern function takes as many arguments of type `PTGNode` as the pattern has insertion points. The arguments are obtained from other calls of pattern functions. Their order corresponds to that of the insertion points in the pattern. (Alternative forms of are described in See Section 2.1 [Indexed], page 6 and See Section 2.2 [Typed], page 7.)

The pattern function for the example above has the following signature:

```
PTGNode PTGPair (PTGNode a, PTGNode b)
```

Examples for applications of this pattern are:

```
x = PTGPair (PTGNil(), PTGNil());
y = PTGPair (x, x);
```

Restrictions

The names of the patterns must be unique over all `.ptg` specifications for a language processor, and they must not collide with identifiers predefined for PTG.

PTG does not insert any additional white space before or after elements of the pattern sequence. Token separation and new line characters (especially at the end of a file) have to be specified explicitly.

2.1 Indexed Insertion Points

The insertion points of a pattern may be identified by numbers, e.g. `$1`, `$2`. This facility allows to insert an argument of a pattern function call at several positions in the pattern, and it allows to modify patterns without the need to change their application calls.

In the following example the first argument (the module name) is inserted at two positions:

```
Module: "module " $1 "\nbegin" $2 "end " $1 ";\n"
```

The pattern function is called with two arguments.

The correspondence between insertion points and function parameters is specified by the numbers of the insertion points, i.e. the first argument is inserted at the insertion points `$1`.

This facility also makes the calls of pattern functions more independent of pattern modifications. For example, the following pattern describing declarations

```
Decl: $1 /* type */ " " $2 /* identifiers */ ";\n"
```

would be applied by a call `PTGDecl (tp, ids)`, with the first argument inserting the type and the second inserting the identifiers. Those calls are invariant against changing the pattern to Pascal-like declaration style:

```
Decl: $2 ":" $1 ";"
```

In the same way one variant of a pattern may omit an argument specified in an other variant.

In general a pattern may contain several occurrences of any of the insertion point markers `$`*i*. There is practically no upper bound for `$`*i*. The generated function has n parameters, where n is the maximal i occurring in a `$`*i* of the pattern. The i-th function argument is substituted at each occurrence of `$`*i* in the pattern.

If a pattern does not mention all `$`*i* between `$1` and the maximum `$`*n*, e.g. `$1` and `$3` but not `$2`, the function has n parameters, but some are not used.

Restrictions

Indexed and non-indexed insertion point markers must not be merged in a single pattern.

2.2 Typed Insertion Points

Data items can be inserted into the output text by specifying insertion points to have one of the types `int`, `string`, `long`, `short`, `char`, `float` or `double`, e.g.

```
Matrix:  "float " $1 "[" $2 int "][" $3 int "];\n"
```

The generated pattern function has the following signature:

```
PTGNode PTGMatrix (PTGNode a, int b, int c)
```

Function calls must supply arguments of corresponding types. They are output in a standard output representation.

If an indexed insertion point occurs multiply in a pattern its type must be the same for all occurrences.

Another typical application of typed insertion point is generating identifiers:

```
Ident: $ string $ int
```

This pattern may be used to compose identifiers from a string and a number, e.g. a call `PTGIdent ("abc", 5)` producing `abc5`. The string item is often taken from the input of the language processor, and the number is used to guarantee uniqueness of identifiers in the output. (This construct also substitutes the outdated leaf patterns, See Chapter 8 [Outdated], page 33.)

A typical example for composition of output text fragments from data of basic types is given by a pattern that produces German car identifications:

```
CarId: $ string $ string $int
```

which is applied for example by `PTGCarId ("PB-", "AB-", 127)`.

Restrictions

If indexed insertion points are used with a type specification, the type specification must be the same for all occurrences the insertion.

2.3 Function Call Insertion

There are situations where it is inconvenient or impossible to specify an output component by a pattern. In such cases calls of user defined functions can be specified instead of insertion points in a pattern.

Assume as an example that indentation shall be specified for the output of a block structured language:

```
Block:   [NewLine] "{" [Indent] $1   /* declarations */
                              $2   /* statements */
                        [Exdent]
         [NewLine] "}"
```

This pattern for producing a block has two ordinary insertion points, one for the declarations and one for the statements, and the pattern function is called as usual with two corresponding arguments. When the output text is produced the user defined functions `NewLine`, `Indent`, and `Exdent` are called in order to insert text at the specified pattern positions.

In this case the functions must have exactly one parameter that is a `PTG_OUTPUT_FILE`:

```
extern void NewLine (PTG_OUTPUT_FILE f);
extern void Indent (PTG_OUTPUT_FILE f);
extern void Exdent (PTG_OUTPUT_FILE f);
```

The type `PTG_OUTPUT_FILE` can be supported by the user. If it is not, a default is supported by the generated file `ptg_gen.h`. The function has to be implemented such that a call outputs the desired text to the file pointed to by `f` by using some provided output macros, See Chapter 7 [Macros], page 29.

Note: These function are called when the output text is produced. They are not yet called when the patterns are applied. PTG guarantees that those calls occurr in left-to-right order of the produced output text. Hence, the above triple of functions may use global variables to keep track of the indentation level.

Functions, that support indentation ready to use in a PTG specification can be found in the module library, See Section 'Indentation ' in MODLIB - Specification Module Library.

Such function calls may also take arguments which are passed through from the call of the pattern function. They are specified by occurrences of insertion within the call specification:

```
Block:   [NewLine] "{" [Indent $3 int] $1   /* declarations */
                                       $2   /* statements */
                       [Exdent $3 int]
         [NewLine] "}"
```

In this case the indentation depth is determined individually for each application of the `Block` pattern which is called for example by `PTGBlock (d, s, 3)`. The last argument is passed through to the calls of `Indent` and `Exdent` which now must have the signatures

```
extern void Indent (PTG_OUTPUT_FILE f, int i);
extern void Exdent (PTG_OUTPUT_FILE f, int i);
```

Note: The arguments supplied with a pattern application are stored until the functions are called when the output is produced.

In general several arguments may be specified to be passed through to a function call. They may be typed by one of the types `int`, `string`, `long`, `short`, `char`, `float`, `double` or `pointer`. In case of type `pointer` the supplied argument of the pattern function call must have a pointer type that is defined for the corresponding parameter of the user function. If no type is specified an argument of type `PTGNode` is passed through.

Arguments specified of type `pointer` are typically used if the translation of certain data structures by user specified functions is to be inserted into pattern driven translations.

2.4 Optional Parts in Patterns

Parts of a pattern can be marked as optional by surrounding them with braces. Using this notation, the optional parts will only be printed in the output, if all non-optional insertions of the pattern produce output. This can be applied to simplify the construction of lists considerably.

```
CommaSeq:    $1 {", "} $2
```

A call of the pattern function `PTGCommaSeq(a,b)` produces the separator only, if neither `a` nor `b` are empty; otherwise `a` and `b` are just concatenated, leaving out the optional part. This facility is especially useful if such separated lists are composed by pattern function calls that occur in loops or in separated contexts. See Chapter 5 [Example], page 17, for a more sophisticated example.

Note: The result of a pattern call is the unique value `PTGNULL` if the empty output string is produced. (There is no way to further inspect the intermediate results of pattern applications.) Certain pattern constructs do not yield `PTGNULL` even if they may represent empty strings:

Typed insertions and function call insertions are considered not empty.

Empty strings and literals.

Another example for optional parts in patterns can be the following:

```
Paren:      {"("} $ {")"}
```

The pattern function `PTGParen(a)` will procude parenteses around `a` if `a` is not empty. Otherwise, `PTGParen(a)` will be empty.

Restrictions

An optional pattern is printed, if all non-optional insertions in the node are not `PTGNULL`. However, if there are no non-optional insertions, the braces are ignored and a warning is issued.

It is possible to include more than one pattern in braces. Multiple optional parts can be included in one Rule. However, the braces marking an optional pattern cannot be used recursively inside an optional pattern.

3 Output Functions

PTG separates the composition of a target text from outputting it: A target text is composed by calls of pattern functions. They yield results of type `PTGNode` representing the target text or fragments thereof.

There are three predefined PTG functions that can be applied to `PTGNode` values in order to output the text. They have the following signatures:

```
PTGNode PTGOut(PTGNode r)
PTGNode PTGOutFile(char *f, PTGNode r)
PTGNode PTGOutFPtr(FILE *f, PTGNode r)
```

A call `PTGOut (x)` writes the text represented by `x` to stdout.

The function `PTGOutFile` takes the name of the output file as an additional parameter. A call `PTGOutFile (n, x)` opens the file named `n` for writing, writes the text represented by `x` to it, and closes the file before returning.

The function `PTGOutFPtr` takes the file pointer to an already open file as additional parameter. A call `PTGOutFPtr (f, x)` writes the text represented by `x` to `f`, and leaves The function `PTGOutFPtr` takes open upon exit.

Each of the functions yields its `PTGNode` argument as result. Hence, nested calls of output functions may produce parts of the complete target text on separate files, e.g.

```
PTGOutFile ("f1",
   PTGModule (
      PTGOutFile ("f2", PTGInterface ( ...... )),
      PTGBody ( .... )))
```

In the above example a 'Module' is composed by two text fragments `Interface` and `Body`. The whole text is written to the file `f1`. The `Interface` fragment is also written to the file `f2`.

Of course nested calls like the above can be decomposed by storing intermediate `PTGNode` values. Nested calls of `PTGOutFile` to the same file would destroy the results of the former calls. Nested calls of `PTGOutFPtr` to the same file would append the results of each call to the end of the file.

Note: Be aware that the output functions do not add a newline character to the end of an output text. It has to be explicitly specified by a pattern. This behaviour has been changed compared to previous versions of PTG. Adding a newline character to a text is simply specified by a pattern like

```
NewLine:    $ "\n"
```

4 Some Useful Techniques

This chapter describes some techniques that solve common tasks in PTG applications:

4.1 Output of Data Items

Usually data items like numbers, identifiers, or strings which are computed by the translation process or taken from the input are to be inserted into the output text at certain positions. This is best achieved by using typed insertion points as described in See Section 2.2 [Typed], page 7.

It is often necessary to convert single data items into a `PTGNode` pointer to be used in pattern applications. This can be achieved by defining patterns for single data items (see See Section 2.2 [Typed], page 7):

```
Number:    $ int
String:    $ string
StringLit: "\"" $ string "\""
```

Such patterns are applied by calls like `PTGNumber(5)`, `PTGString("+")`, `PTGStringLit("Hello!")` producing the text items `5`, `+`, and `"Hello!"` respectively.

Often data items of the input text, e.g. identifiers or floating point numbers, are reproduced in the output. They may be taken from token strings stored in the `csm` module, e.g. `PTGString(StringTable(id))` where `id` is the token identification.

The `LeafPtg` module of the Module Library supports this kind of translations.

4.2 Generating Identifiers

Common techniques for producing identifiers are recommended especially for translation into programming languages. In the simplest case identifiers are just reproduced from the input as shown above. The following set of patterns allow to add prefixes or postfixes to the original identifiers (see See Section 2.2 [Typed], page 7):

```
PointerId: "PTR_" $ string
ValueId:   "VAL_" $ string
UniqueId:  $ string $ int
```

Patterns like the first two may be used to generate different output identifiers from one input identifier. The last pattern allows to attach a number to an identifier, e.g. to guarantee uniqueness in the output in cases where the source and the target language have different scope rules.

Note: PTG does not add any white space between pattern items. Hence, patterns where identifiers or numbers may be inserted have to ensure that the component tokens are separated, e.g.

```
Decl: $ /* Type */ " " $ /* Ident */
```

4.3 Output of Sequences

The construction of output text often requires to compose arbitrary long sequences of items, e.g tokens, statements, or procedures. Some simple techniques for those tasks are described here. The examples can easily be generalized for similar applications.

A generally applicable pattern for constructing sequences is

```
Seq: $ $
```

An application

```
PTGSeq (PTGString("."), PTGSeq (PTGString("."), PTGString(".")))
```

produces a sequence of 3 dots, assuming pattern `String` be suitably defined.

The following C loop computes a sequence of `n` dots:

```
PTGNode dots = PTGNULL; int i;
for (i=0; i<n; i++) dots = PTGSeq (dots, PTGString("."));
```

Sequences where the items are separated by a certain string, e.g. a comma or a space character can be specified by

```
CommaSeq: $ {", "} $
```

An application

```
PTGCommaSeq (PTGNumber (1), PTGNumber (2))
```

produces `1, 2`, assuming pattern `Number` be suitably defined.

The following C loop computes a comma separated sequence of the numbers 1 to `n`:

```
PTGNode numseq = PTGNULL; int i;
for (i=0; i<n; i++)
  numseq = PTGCommaSeq (numseq, PTGNumber (i));
```

5 A Complete Example

In this chapter we demonstrate the use of PTG for translating a simple assignment language into C code. This example shows PTG techniques in the context of a complete translator specification. Special emphasis is put on techniques for pattern applications in LIDO specifications.

Reading this manual online use the documentation browser's `Run` command to obtain a copy of the complete specification for your own experiments.

You will get a Funnelweb file `PtgEx.fw` containing the content of this chapter (see FunnelWeb). It can be used for example to derive the specified processor:

```
PtgEx.fw :exe >.
```

or to derive the set of files described below:

```
PtgEx.fw :fwGen >.
```

5.1 Source Languge Structure

Programs of this example language are sequences of assignments, input-, and output-statements, like

simple[1]==

```
input a;
output a;
x := a + 1;
y := x - 5;
output x;
output y + x;
```

This macro is attached to an output file.

The values of variables and expressions are integral numbers. There are only the binary operators `+` and `-`.

The above program shall be translated into the following C program:

simple.out[2]==

```
#include <stdio.h>

int a = 0, x = 0, y = 0;

int main (void) {

scanf ("%d", &(a));
printf ("%d\n",a);
x = a+1;
y = x-5;
printf ("%d\n",x);
printf ("%d\n",y+x);

exit (0);
}
```

This macro is attached to an output file.

The structure of source programs is specified by the following concrete grammar:

Program.con[3]==

```
Program:     Statement*.

Statement:   Variable ':=' Expression ';'.
Statement:   'input' Variable ';'.
Statement:   'output' Expression ';'.

Expression:  Expression Operator Operand / Operand.

Operator:    '+' / '-'.

Operand:     Variable.
Operand:     IntLit.
Variable:    Ident.
```

This macro is attached to an output file.

In the tree grammar `Expressions` and `Operands` are represented both by `Expression` nodes, as specified by the type `.sym` rule:

Expr.sym[4]==

```
Expression ::= Operand .
```

This macro is attached to an output file.

Identifier tokens, number literals, and comments are denoted as in Pascal, as stated by the following type `.gla` specification:

Mini.gla[5]==

```
Ident:   PASCAL_IDENTIFIER
IntLit:  PASCAL_INTEGER
         PASCAL_COMMENT
```

This macro is attached to an output file.

5.2 Program Frame

In this section the overall structure of the target program is specified, the name of the output file is determined, and its contents is produced by a PTG output function.

We first specify a pattern forthe overall structure of the target program:

Frame.ptg[6]==

```
Frame:
   "#include <stdio.h>\n\n"

   $1 /* declarations */

   "\nint main (void) {\n\n"
```

```
$2 /* statements */

"\nexit (0);\n}\n"
```

This macro is attached to an output file.

It has two insertion points, one for the variable declarations and one for the statement sequence. The text to be inserted is obtained from the attributes `Program.DeclPtg` and `Program.StmtPtg` of type `PTGNode` Their computation is shown in the following sections. Here they are used as arguments of the `Frame` pattern application:

TransProg.lido[7]==

```
ATTR DeclPtg, StmtPtg: PTGNode;

SYMBOL Program COMPUTE
  PTGOutFile (AppendToSrcFileName (".c"),
              PTGFrame (THIS.DeclPtg, THIS.StmtPtg));
END;
```

This macro is attached to an output file.

The above call of the output function `PTGOutFile` writes to a file which name is derived from the file name of the source program by appending `".c"` (see See Chapter 3 [Output], page 11). This is achieved by the macro `AppendToSrcFileName` we supply as a type `.head` specification without further explanation here:

FileName.head[8]==

```
#include "csm.h"
#include "source.h"
#define AppendToSrcFileName(post) \
       (obstack_strgrow(Csm_obstk, SRCFILE),\
       CsmStrPtr = (char*)obstack_strcpy(Csm_obstk, post))
```

This macro is attached to an output file.

5.3 Expressions

In this section we specify the translation of expressions. Target expressions are composed by applications of patterns that construct the text in a bottom-up way, i.e. from the leaves upto the complete expression.

In our simple example this translation is one-to-one as specified by the three patterns:

TransExpr.ptg[9]==

```
BinOperation: $ $ $
Number:       $ int
String:       $ string
```

This macro is attached to an output file.

The `BinOperation` pattern composes a left operand, an operator, and a right operand. The `Number` pattern just converts an integral number into text. The `String` pattern reproduces its argument. It is used here for output of operators and of identifiers.

These patterns are applied in computations of `Expression` contexts. Attributes `Ptg` of type `PTGNode` are used for the intermediate results:

TransExpr.lido[10]==

```
ATTR Ptg: PTGNode;

RULE: Expression ::= Expression Operator Expression COMPUTE
   Expression[1].Ptg =
      PTGBinOperation (
         Expression[2].Ptg, Operator.Ptg, Expression[3].Ptg);
END;

RULE: Operator ::= '+' COMPUTE
   Operator.Ptg = PTGString ("+");
END;

RULE: Operator ::= '-' COMPUTE
   Operator.Ptg = PTGString ("-");
END;
```

```
RULE: Expression ::= Variable COMPUTE
   Expression.Ptg = Variable.Ptg;
END;

RULE: Expression ::= IntLit COMPUTE
   Expression.Ptg = PTGNumber (IntLit);
END;

RULE: Variable ::= Ident COMPUTE
   Variable.Ptg = PTGString (StringTable (Ident));
END;

ATTR Sym: int;
```

This macro is attached to an output file.

The last two computations use the token attributes `Sym` to supply the number value to the `Number` pattern, and to access the identifier string from the `StringTable` to be reproduced by the `String` pattern.

5.4 Using LIDO CHAINs

In this sections the translation of statement sequences is shown. The LIDO `CHAIN` construct is used to compose a sequence of translated stements in left-to-right order.

Assignments, input, and output statements are translated by the following patterns:

TransStmt.ptg[11]==

```
AssignStmt:  $1 /* lhs */ " = " $2 /* rhs */ ";\n"
InputStmt:   "scanf (\"%d\", &(" $1 /* variable */ "));\n"
OutPutStmt:  "printf (\"%d\\n\","  $1 /* expression */ ");\n"

Seq:         $1 $2
```

This macro is attached to an output file.

The last pattern is used to combine two text components (statement sequences in this case) into one (see See Section 4.3 [Sequences], page 14).

A CHAIN named StmtChn is defined to compose PTGNode values in left-to-right order through the tree. The CHAIN starts in the root context with an empty text. The result is obtained at the end of the CHAIN by TAIL.StmtChn:

TransStChn.lido[12]==

```
CHAIN StmtChn: PTGNode;

SYMBOL Program COMPUTE
   CHAINSTART HEAD.StmtChn = PTGNULL;
   SYNT.StmtPtg = TAIL.StmtChn;
END;
```

This macro is attached to an output file.

In each of the three statement contexts the translation is produced by application of the corresponding pattern and added to the end of the CHAIN using the Seq pattern:

TransStmt.lido[13]==

```
RULE: Statement ::= Variable ':=' Expression ';' COMPUTE
   Statement.StmtChn = PTGSeq (Statement.StmtChn,
      PTGAssignStmt (Variable.Ptg, Expression.Ptg));
END;

RULE: Statement ::= 'input' Variable ';' COMPUTE
   Statement.StmtChn = PTGSeq (Statement.StmtChn,
      PTGInputStmt (Variable.Ptg));
END;

RULE: Statement ::= 'output' Expression ';' COMPUTE
   Statement.StmtChn = PTGSeq (Statement.StmtChn,
      PTGOutPutStmt (Expression.Ptg));
END;
```

This macro is attached to an output file.

5.5 Using LIDO CONSTITUENTS

In this section the construction of a declarator sequence is described using the LIDO `CONSTITUENTS` construct. It is also shown how a list wiht separators is produced, and how text is generated only once for each identifier that occurs in the program.

The source language does not have declarations; variables are introduced by just using them. Hence, we have to generate declarations in the target program, one for each variable that occurs in the source.

A variable may occur several times, but its declaration must be generated only once. For that purpose each variable is identified by a key which is associated to every occurrence of the variable.

This task is an instance of a name analysis task. We can use the `AlgScope` module of the module library to solve it:

ScopeLib.specs[14]==

```
$/Name/AlgScope.gnrc:inst
```

This macro is attached to an output file.

The computational roles `RootScope` and `IdDef` provided by that module are associated to the grammar symbols `Program` and `Variable`:

Scope.lido[15]==

```
SYMBOL Program  INHERITS RootScope END;
SYMBOL Variable INHERITS IdDef COMPUTE
   SYNT.Sym = TERM;
END;
```

This macro is attached to an output file.

The computations of that module yield an attribute `Variable.Key` which identifies the the variable denoted by each identifier occurrence.

We now associate a property `IsDeclared` to variables by a type `.pdl` specification:

Decl.pdl[16]==

```
IsDeclared: int;
```

This macro is attached to an output file.

It describes a state of that variable with respect to the translation process: A declaration is only produced if `IsDeclared` is not set, and then `IsDeclared` is set. The attribute `Variable.DeclPtg` takes the result, either the application of the `DeclVariable`: pattern or `PTGNULL`.

VarDecl.lido[17]==

```
RULE: Variable ::= Ident COMPUTE
   Variable.DeclPtg =
      IF (GetIsDeclared (Variable.Key, 0),
          PTGNULL,
          ORDER (ResetIsDeclared (Variable.Key, 1),
                 PTGDeclVariable (StringTable (Ident))));
END;
```

This macro is attached to an output file.

The pattern `DeclVariable` is used here to reproduce the variable name from the `StringTable` and to add the initialization to it: A single variable declarator is specified by the pattern

Decl.ptg[18]==

```
DeclVariable:  $ string " = 0"
Declaration:   "int " $ ";\n"
```

This macro is attached to an output file.

The second pattern constitutes a complete declaration where the delarator list is inserted.

The delarator list is collected in the Program context using a CONSTITUENTS construct. It combines the PTGNode values of all Variable.DeclPtg attributes of the tree using the three functions of the WITH clause: PTGNull is the nullary predefined function producing no text. IDENTICAL is a unary function predefined in LIDO; it is applied to each Variable.DeclPtg attributes reproducing its value. PtgComSeq is a binary macro specified below; it is combines two sublists into one.

ProgDecl.lido[19]==

```
SYMBOL Program COMPUTE
   SYNT.DeclPtg =
      PTGDeclaration (
         CONSTITUENTS Variable.DeclPtg
         WITH (PTGNode, PTGCommaSeq, IDENTICAL, PTGNull));
END;
```

This macro is attached to an output file.

In order to separate declarators by commas we specify the following pattern (see See Section 4.3 [Sequences], page 14):

Comma.ptg[20]==

```
CommaSeq:  $ {", "} $
```

This macro is attached to an output file.

This pattern can be applied directly in the CONSTITUENTS construct, since the Colon is only inserted, if both insertions contain declarators and do not represent the empty text (PTGNULL).

6 Predefined Entities

PTG generates a C module consisting of an interface file `ptg_gen.h` and an implementation file `ptg_gen.c`. The interface file exports definitions for the following identifiers:

`PTGNode` the pointer type for internal representations of pattern applications

`PTGNULL` a pointer of type `PTGNode` representing no text; Note: There are many ways to represent no text; comparing a `PTGNode` to `PTGNULL` is only a pointer comparison

`PTGNull()`

a macro with no parameter that yields `PTGNULL`, to be used where a function notation is needed, as in LIDO `CONSTITUENTS WITH` clauses

`void PTGFree (void)`

a call of this function deallocates all data generated by pattern applications; to be used for reduction of dynamic memory usage in cases where output is produced in several phases

The following functions can be used to process the contents of a `PTGNode` and it's insertions recursively into an output file. Theese functions are only available, under certain preconditions, See Chapter 7 [Macros], page 29.

`PTGNode PTGOut (PTGNode root)`

a function that outputs the text represented by the parameter `root` to standard output

`PTGNode PTGOutFile (char *filename, PTGNode root)`

a function that opens a file with the name given by the parameter `filename`, outputs the text represented by the parameter `root`, and closes the file

`PTGNode PTGOutFPtr (FILE *output, PTGNode root)`

a function that expects the parameter `output` to be a file which is open for writing, outputs the text represented by the parameter `root` to the file, and leaves the file open

`PTGNode PTGProcess (PTG_OUTPUT_FILE file, PTGNode root)`

a function that expects the parameter `output` to be of a type that is provided by the user. The text represented by the parameter `root` is processed with the output macros, described in Chapter 7 [Macros], page 29.

The user should ensure that these exported identifiers do not clash with other definitions in the application importing the interface file. In particular, the pattern names should be chosen such that prefixing them with `PTG` does not yield a predefined name, e.g. a pattern name `Free` is a bad choice.

The implementation file contains external references to any user defined function mentioned in the particular specification.

7 Influencing PTG Output

The usage of PTG Patterns functions is a very flexible way to construct the output of a program. However, some topics are not accessible easily with pattern construction functions:

Pretty Printing

A pattern construction function does not have access to the current column position. Therefore, it cannot know where to insert linebreaks to get the output formatted properly.

Output destination

It can be desirable to not always process the output of PTG patterns to a file directly. For instance can it be desirable to redirect the output into a string buffer or to postprocess the output.

To solve theese problems, PTG does not print it's output directy into a file. Instead, a set of macros is defined that can be adjusted to change the behaviour of PTG. If theese macros are not defined, PTG supports default definitions that process the contents of a `PTGNode` into a named file or a given file pointer. In the remains of this chapter, the different macros will be explained and small examples will be given.

To process it's output, PTG uses the following macros:

`PTG_OUTPUT_FILE`

This macro defines the type name of the `file`-parameter of the other macros, the output-functions defined in Chapter 3 [Output], page 11 and the function call insertions. If no definition is supplied, it's value is defined to be `FILE *`. It's value must be assignable through the C-operator `=`. If this default is changed, a definition for the next macro must also be provided.

`PTG_OUTPUT_STRING(file,param)`

This macro is called to process a string value into an output file. It is used by default for every printed object in PTG. So, if the default behaviour of PTG should be changed, for instance to support pretty printing, a redefinition of this macro suffices. If you change the file-type, by defining `PTG_OUTPUT_FILE` to something, you must also support a definition for this macro.

`PTG_OUTPUT_INT(file,param)`

`PTG_OUTPUT_SHORT(file,param)`

`PTG_OUTPUT_LONG(file,param)`

`PTG_OUTPUT_CHAR(file,param)`

`PTG_OUTPUT_FLOAT(file,param)`
`PTG_OUTPUT_DOUBLE(file,param)`

Theese macros are used to process the different typed insertions into an output file. By default, they are adjusted in a way that they convert their second argument into a string and call `PTG_OUTPUT_STRING` to process the output. So, when redefining PTG-Output, you need not to supply a definition for theese macros.

Of couse, there may be a more efficient posibility to output for instance a character that to convert it into a string and send it to the string handling function. If you want to supply such a function, redefine the default-behaviour of `PTG_OUTPUT_CHAR`. So, theese macros are used for efficiency only.

To override the default implementations for theese macros, implement a substitution function in a type `.c` file. Include the file `ptg_gen.h` to supply your new function with definitions for the other macros, especially `PTG_OUTPUT_FILE`. Write Preprocessor-directives that define the desired macro(s) to a type `.ptg.phi` file and include it in your specification. Eli will then concatenate all those definitions and provide it as a header-file `ptg.h` to your processor.

As an application of theese macros, the following section will provide a simple and easy way to implement pretty-printing in the generated PTG-output. The Module Library included in Eli provides some more applications of theese macros, at least, it will in the future.

7.1 Changing Default-Output for Limited Line Length

Sometimes, it is necessary to postprocess PTG-Output. For example can it be useful, if the generated output has lines, no longer than a certain constant, so that other tools can process it. Without postprocessing of PTG-Output, there is no way to determine the current position in a line for a fixed PTG-Node. So, it is very difficult to assure a constant maximal line length and at the same time produce readable output.

Because all the output-macros for typed insertions call the macro `PTG_OUTPUT_STRING`, it is relatively easy to implement a postprocessor, that computes the current position in a line. With some more effort, it would be possible do complete line breaking by buffering one line of output and looking for good break points after the linefeed is encountered.

The following code keeps track of the current column position and implements a function, that conditionally inserts a linebreak if the line is longer than 65 chars.

linepos.c[21]==

```
#include <string.h>
#include "ptg_gen.h"

static col = 0;

void InitCol(void)
{
   col = 0;
}

void OutputLine(FILE *f, char *s)
{
   int l;
   char *nl;

   if (!s) return;

   l = strlen(s);
   nl = strrchr(s, '\n');

   if (!nl)
      col += l;
   else
      col = (l - 1 - (nl - s));
   fputs(s, f);
}

void CondNl(FILE *f)
{
   if (col > 65)
      OutputLine(f, "\n");
}
```

This macro is attached to an output file.

The function `InitCol` serves as initialisation, if more than one output file should be generated. The function `CondNl` inserts a linebreak, if the current line is longer than 65 characters. The function `OutputLine` overrides the default-implementation of `PTG_OUTPUT_STRING`.

linepos.ptg.phi[22]==

```
#define PTG_OUTPUT_STRING(file,param) OutputLine(file,param)
```

This macro is attached to an output file.

This can be used for example in the following PTG specification:

linepos.ptg[23]==

```
ColonSeq:       $ ", " [CondNl] $
```

This macro is attached to an output file.

Now, in a large iteration of calls to `PTGColonSeq()`, a linebreak is inserted automatically, if a line exeeds 65 characters.

Using the online documentation, you can obtain a copy of the attached files.

8 Outdated Constructs

Previous versions of PTG did not allow for typed insertion points and for arguments of pattern function calls beeing passed through to user supplied functions. Instead they required to specify additional leaf patterns for such cases, e.g.

```
Numb:  int []
CarId: string string int []
Msg:  int int int [PrintMsg]
```

They are now specified by general pattern specifications:

```
Numb:  $ int
CarId: $ string $ string $ int
Msg:  [PrintMsg $ int $ int $ int]
```

The present version of PTG still supports those leaf patterns. But it is strongly recommended to rewrite such outdated specifications into the new format, since it is more general, more flexible, and it can be more efficiently applied. The leaf pattern facilility may be deimplemented in a future version of PTG.

9 Syntax of PTG Specifications

```
PTGSpec:         PatternSpec+

PatternSpec:     PatternName ':' (Item | Optional)*
                 | 'include' CString

PatternName:     Identifier

Item:            CString | Insertion | FunctionCall

Insertion:       '$' [ Number ] [ Type ]

FunctionCall:    '[' Identifier Arguments ']'

Arguments:       Insertion*

Type:            'int' | 'string' | 'pointer' | 'long' | 'short'
                 | 'char' | 'float' | 'double'

Optional:        '{' Item* '}'
```

`Identifier` and `CString` tokens are denoted as in C. `Number` tokens consist of decimal digits. Comments are written in C style. Line comments starting with `#` are also accepted.

Note: A `CString` token may only extend over several lines if all but the last line end with a backslash character. An error message like `illegal newline in string literal` indicates a violation of that rule.

The following additional alternative for `PatternSpec` defines leaf patterns. This construct is outdated but still accepted by PTG:

```
PatternSpec:     PatternName ':' Type+ '[' [ Identifier ] ']'
```

Since that construct may be deimplemented in future versions of PTG it is recommended to rewrite uses of these constructs by pattern specifications as described above, e.g.

```
MyLeaf:   int string []
YourLeaf: int string [YourFct]
```

should be rewritten into

```
MyLeaf:   [$ int $ string]
YourLeaf: [YourFct $ int $ string]
```

Index

S

T

U

W