

6. Structured Output

Generator outputs structured text:

- programm in a suitable programming language
- data in suitable form (e.g. XML) to be processed by specific tools
- text in suitable form (e.g. HTML) to be presented by a text processor

Transformation phase of the generator defines the structure of the texts:

- parameterized text patterns
- instances of text patterns hierarchally nested

a text pattern with 2 parameters:

```
#define  Kind 
```

2 instances:

```
#define intKind 1
```

```
#define PairPtrKind 2
```

```
#ifndef WRAPPER_H
#define WRAPPER_H

#include "Pair.h"

#define noKind 0

#define intKind 1
#define PairPtrKind 2
#define floatKind 3

class intWrapper;
class PairPtrWrapper;
class floatWrapper;

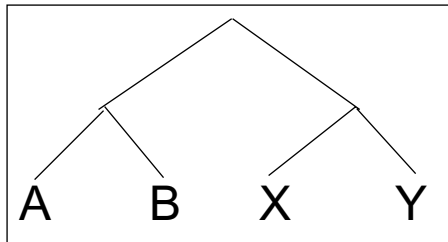
class Object {
public:
    class WrapperExcept {};
    int getKind () { return kind; }

    int getIntValue ();
    PairPtr getPairPtrValue ();
    float getFloatValue ();
protected:
    int kind;
};
```

„Structure Clash“ on Text Output

abstract program tree

drives creation of the target text
by a tree walk



target text

is composed of fragments

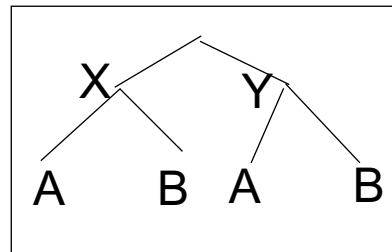
X A B Y A B

tree walk **order does not fit** to
sequence of target text fragments

**solution: text is composed into a buffer,
and sequentially written from there**

here:

the buffer is a tree or DAG representing
pattern applications



PTG: Pattern-Based Text Generator

Generates **constructor functions** from
specifications of text patterns

- A. PTG provides a Specification language for text patterns**
each is a sequence of text fragments and
insertion points

```
#define int Kind 1
```

- B. PTG generates constructor functions**
that build a data structure of pattern applications
- one function per pattern
one parameter per insertion point
- The functions are called on the tree walk.

- C. PTG generates output functions**
they walk recursively through the
data structure to output the target text

PTG's Specification Language: Introductory Example

Pattern: named sequence of C string literals and **insertion points**

KindDef:

```
"#define " $ string "Kind \t" $ int "\n"
```

WrapperHdr:

```
"#ifndef WRAPPER_H\n"
"#define WRAPPER_H\n\n"
$1 /* Includes */

"\n#define noKind          0\n"
$2 /* KindDefs */
"\n"

$3 /* ClassFwds */
"\n"

"class Object {\n"
"public:\n"
"  class WrapperExcept {};\n"
"  int getKind () { return kind; }\n"
$4 /* ObjectGets */
"protected:\n"
"  int kind;\n"
"};\n\n"
```

```
#define int Kind 1
```

```
#ifndef WRAPPER_H
#define WRAPPER_H

#include "Pair.h"

#define noKind          0

#define intKind 1
#define PairPtrKind 2
#define floatKind 3

class intWrapper;
class PairPtrWrapper;
class floatWrapper;

class Object {
public:
  class WrapperExcept {};
  int getKind () { return kind; }

  int getIntValue ();
  PairPtr getPairPtrValue ();
  float getfloatValue ();
protected:
  int kind;
};
```

Constructor Functions

A **constructor function** for each pattern.

A parameter for each insertion point:

```
PTGNode PTGKindDef (char *a, int b) {...}
```

```
PTGNode PTGWrapperHdr (PTGNode a, PTGNode b, PTGNode c, PTGNode d)
    {...}
```

Call of a constructor function

- creates an instance of the pattern with the supplied arguments and
- yields a reference to that instance

```
ik = PTGKindDef ("int", 1);
```

```
hdr = PTGWrapperHdr (ik, xx, yy, zz);
```

The arguments of calls are such references (type `PTGNode`) or they are values of the type specified in the pattern (e. g. string or int)

Such calls are used to **build the data structure bottom-up**.
It is acyclic, a DAG.

Output Functions

Predefined output functions:

- Call:

```
PTGOutFile ("example.h", hdr);
```

initiates a recursive walk through the data structure starting from the given node (2nd argument)

- All text fragments of all pattern instances are output in the specified order.
- Shared substructures are walked through and are output on each visit from above.
- User defined functions may be called during the walk, in order to cause side-effects (e.g. set and unset indentation).

Important Techniques for Pattern Specification

Elements of pattern specifications:

- string literals in C notation `"Value ();\n"`
- value typed insertion points `$string $int`
- untyped insertion points (`PTGNode`) `$ $1`
- comments in C notation `$ /* Includes */`
e.g. to explain the purpose of insertion points

All characters that **separate tokens** in the output and that **format the output** have to be **explicitly specified** using string literals `" " ";\n" "\tpublic:"`

Identifiers can be augmented by prefixes or suffixes:

```
KindDef: "#define "$ string "Kind \t" $ int "\n"
```

may yield

```
#define PairPtrKind 2
```

There are advanced techniques to create „pretty printed“ output (see PTG documentation).

Important Techniques: Indexed Insertion Points

Indexed insertion points: **\$1** **\$2** ...

1. Application: **one argument is to be inserted at several positions:**

```
ObjectGet: " " $1 string " get" $1 string "Value ();\n"
```

```
call: PTGObjectGet ("PairPtr") result: PairPtr getPairPtrValue ();
```

2. Application: **modify pattern - use calls unchanged:**

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

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

```
unchanged call: PTGDecl (tp, ids)
```

Rules:

- If n is the greatest index of an insertion point the constructor function has n parameters.
- If an index does not occur, its parameter exists, but it is not used.
- The order of the parameters is determined by the indexes.
- Do not have both indexed and non-indexed insertion points in a pattern.

Important Techniques: Typed Insertion Points

Untyped insertion points: \$ \$1

Instances of patterns are inserted, i.e. the results of calls of constructor functions

Parameter type: `PTGNode`

Typed insertion points: \$ string \$1 int

Values of the given type are passed as arguments and output at the required position

Parameter type as stated, e.g. `char*`, `int`, or other basic types of C

```
KindDef: "#define " $ string "Kind \t" $ int "\n"
```

```
call:      PTGKindDef ("PairPtr", 2)
```

Example for an application: generate identifiers

```
KindId:      $ string "Kind"          PTGKindId("Flow")
```

```
CountedId:   "_" $ string "_" $ int   PTGCountedId("Flow", i++)
```

Example for an application: conversion into a pattern instance

```
AsIs:      $ string   PTGAsIs("Hello")
```

```
Numb:     $ int       PTGNumb(42)
```

Rule:

- **Same index** of two insertion points **implies the same types.**

Important Techniques: Sequences of Text Elements

Pairwise concatenation:

```
Seq: $ $          PTGSeq(PTGFoo(...),PTGBar(...))
                res = PTGSeq(res, PTGFoo(...));
```

The application of an empty pattern yields PTGNULL

```
PTGNode res = PTGNULL;
```

Sequence with optional separator:

```
CommaSeq: $ {" , " } $          res = PTGCommaSeq (res, x);
```

Elements that are marked optional by `{ }` are not output,
if at least one insertion has the value `PTGNULL`

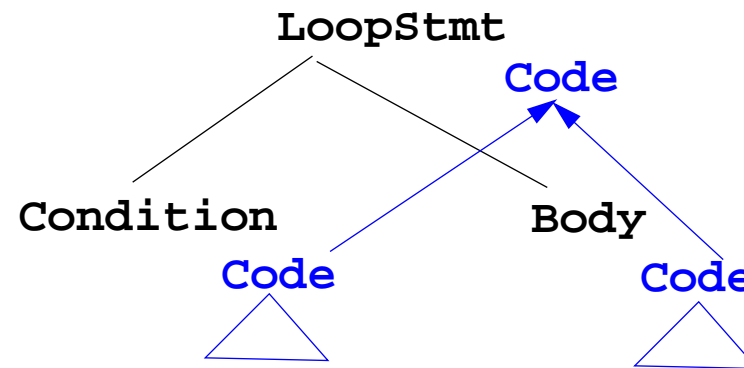
Optional parentheses:

```
Paren: {" (" } $ {" ) " }      no ( ) around empty text
```

The Eli specification `$/Output/PtgCommon.fw` makes some of these useful pattern definitions available: `Seq`, `CommaSeq`, `AsIs`, `Numb`

Compose Target Text in Adjacent Contexts

Attributes in adjacent tree contexts



ATTR Code: PTGNode;

RULE: LoopStmt ::= Condition Body COMPUTE

**LoopStmt.Code =
PTGWhile (Condition.Code, Body.Code);**

Application of the
while pattern

END;

