2. Strukturierte Texte generieren

Generator erzeugt strukturierte Texte:

- Programme in einer passenden Programmiersprache
- Daten in passender Form zum Verarbeiten durch spezielle Programme
- Texte in passender Form zur Präsentation durch Textverarbeitung

Transformation des Generators definiert die Struktur der Texte:

- feste Textmuster, variabel instanziiert
- Instanzen von Textmustern
- hierarchisch ineinander eingesetzt

```
#define [    ] Kind [    ]

#define intKind 1

#define PairPtrKind 2
```

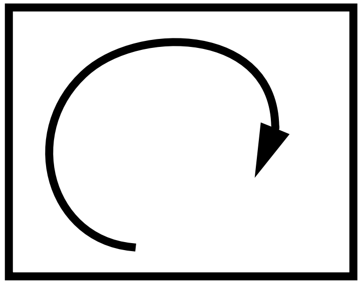
```
#ifndef WRAPPER_H
#define WRAPPER_H

#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;
};
```

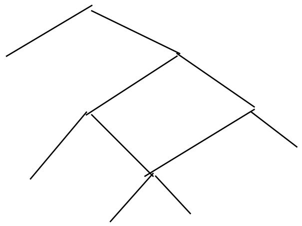
„Structure Clash“ bei der Texterzeugung

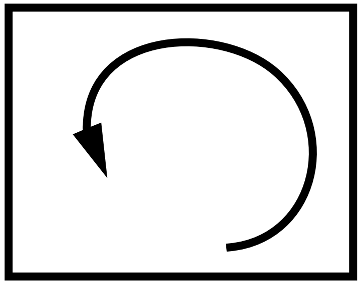
Zieltext erzeugen
aus der Eingabe des Generators

Zieltext ausgeben

- unterschiedliche **Reihenfolge**
- unterschiedliche **Strukturen**







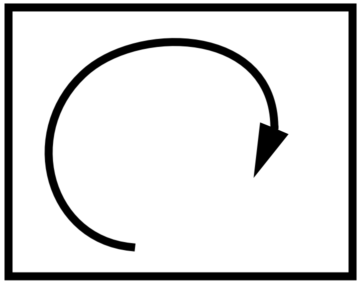
Puffer löst Structure Clash auf

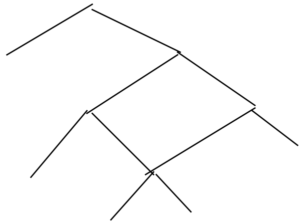
hier:

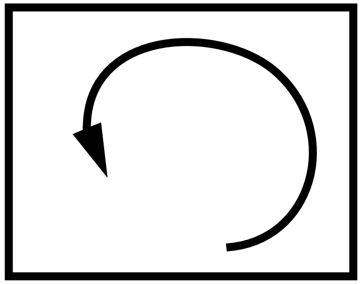
Datenstruktur (Baum, DAG):
Knoten repräsentiert Musteranwendung

PTG: Pattern-Based Text Generator

Was leistet der Generator?







A. Spezifikationssprache für Textmuster
Folge von Textfragmenten und Einfügestellen

B. generiert Konstruktormethoden
zum Aufbau der Datenstruktur
je eine pro Muster
ein Parameter pro Einfügestellen

C. Ausgabefunktionen
für Durchlauf der
Datenstruktur und
Ausgabe der Texte

Spezifikationssprache für Textmuster: einführendes Beispiel

Pattern: Benannte Folge von C-String-Literalen und Einfügestellen

```
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 intKind 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;
};
```

Konstruktorfunktionen und Ausgabe

Eine **Konstruktorfunktion** zu jedem Pattern.

Ein Parameter für jede Einfügestelle:

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

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

Aufruf einer Konstruktorfunktion

- erzeugt eine Pattern-Anwendung mit den gegebenen Parametern und
- liefert Referenz auf die erzeugte Pattern-Anwendung

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

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

Parameter sind solche Referenzen (Typ `PTGNode`) oder haben den im Pattern angegebenen Typ (z. B. string)

Datenstruktur wird durch solche Aufrufe **bottom-up aufgebaut**. Ist zyklenfrei; DAG.

Ausgabe

Vordefinierte Ausgabefunktionen

- Aufruf:

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

  stößt rekursiven Durchlauf der Datenstruktur vom angegebenen Knoten aus an
- alle Texte aller Musteranwendungen werden in der richtigen Reihenfolge ausgegeben
- gemeinsame Unterstrukturen werden mehrfach durchlaufen und ausgegeben
- Anwendungsfunktionen können wäherend des Durchlaufs aufgerufen werden

Wichtige Techniken zu Pattern-Spezifikationen

Elemente von Pattern-Spezifikationen:

- String-Literale in der Schreibweise von C `"Value ();\n"`
- Einfügestellen `$   $1   $ string`
- Kommentare in der Schreibweise von C `$ /* Includes */`
  z. B. Zweck von Einfügestellen erklären

Alle Zeichen zum **Trennen von Symbolen** und zum **Formatieren** des Textes müssen in den String-Literalen explizit angegeben werden `" "   ";\n"   "\tpublic:"`

Bezeichner können mit festen Präfixen oder Suffixen versehen werden:

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

erzeugt z. B.

```
#define PairPtrKind    2
```

Mit fortgeschrittenen Techniken kann „pretty printing“ erzielt werden (siehe PTG-Dokumentation).

Wichtige Techniken: Indizierte Einfügestellen

Indizierte Einfügestellen: `$1    $2 ...`

1. Anwendung: **gleicher Text soll an mehreren Stellen** eingefügt werden:

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

Aufruf `PTGObjectGet ("PairPtr")`

2. Anwendung: **Muster ändern - Aufrufe beibehalten**:

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

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

Aufruf unverändert `PTGDecl (tp, ids)`

Regeln:

- Konstruktorfunktion hat soviele Parameter wie der größte Index einer Einfügestelle angibt.
- Reihenfolge der Parameter ist durch die Indizes festgelegt.
- Werden Indizes im Pattern ausgelassen, werden trotzdem Parameter dafür angegeben.
- Indizierte und nicht-indizierte Einfügestellen nicht in einem Pattern mischen.

Wichtige Techniken: Typisierte Einfügestellen

Untypsierte Einfügestellen: `$` `$1`

eingefügt werden Pattern-Anwendungen, d. h. die Ergebnisse von Pattern-Aufrufen
Parametertyp: **`PTGNode`**

Typisierte Einfügestellen: `$ string` `$1 int`

eingefügt werden Werte des angegebenen Typs, als aktuelle Parameter übergeben,
Parametertyp wie angegeben, z. B. **`char*, int`**, oder andere Grundtypen aus C

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

Aufruf: **`PTGKindDef ("PairPtr", 2)`**

Anwendungsbeispiel: Bezeichner generieren

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

Anwendungsbeispiel: Konvertierungs-Pattern

```
AsIs:   $ string                          PTGAsIs("Hello")
Numb:   $ int                             PTGNumb(42)
```

Regel:

- Mehrfaches Auftreten einer Einfügestelle muss denselben Typ haben.

Wichtige Techniken: Folgen von Textelementen

Paarweise Konkatenation:

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

Ergebnis der Anwendung eines leeren Pattern: `PTGNULL`

```
                                      PTGNode res = PTGNULL;
```

Folge mit optionalem Trenner:

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

optionale Teile werden nur dann eingesetzt, wenn die Parameter zu jeder Einfügestelle verschieden sind von **`PTGNULL`**

Optionale Klammern:

`Paren:  {"("} $ {")"}` klammert leeren Text nicht

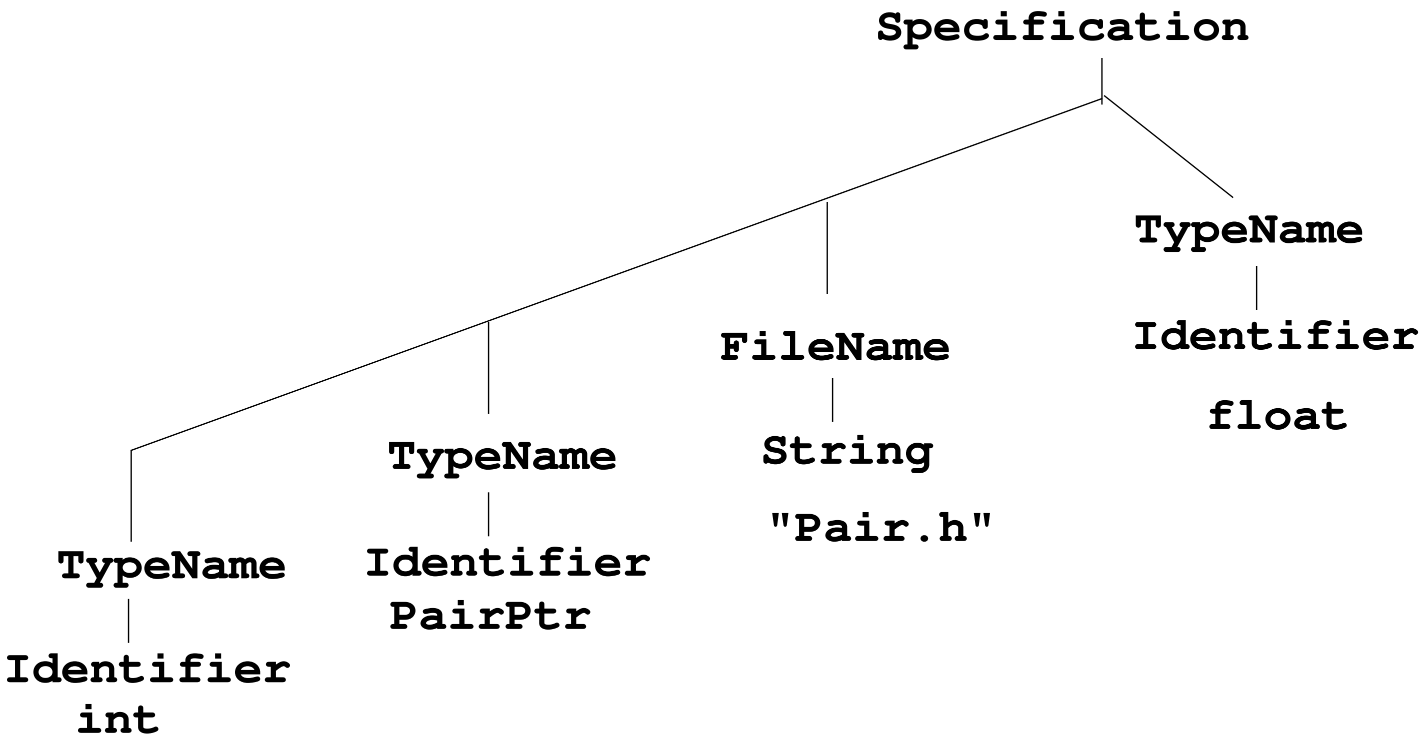
Mit der Eli-Spezifikation **`$/Output/PtgCommon.fw`** bekommt man einige dieser nützlichen Pattern-Definitionen: **`Seq, CommaSeq, AsIs, Numb`**

Zieltext aus Strukturbaum erzeugen

Vorgriff: Generator repräsentiert seine Eingabe intern durch einen **Strukturbaum**

Wrapper-Generator:

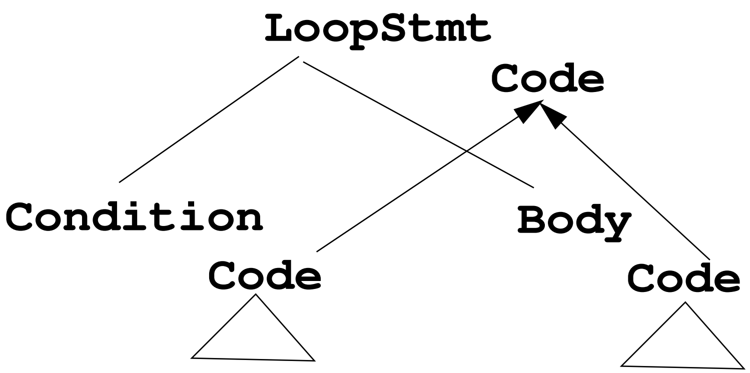
`Specification` ist eine Folge von **`TypeName`**- und **`FileName`**-Knoten



Knoten im Baum haben Attribute, z. B. **`TypeName.ClassFwd`**,
Attribute enthalten Referenzen auf erzeugte Pattern-Anwendungen

Zieltexte zusammensetzen: benachbarte Kontexte

Im Strukturbaum **direkt benachbarte Kontexte**
(werden im Wrapper-Generator nicht verwendet)



Im Kontext

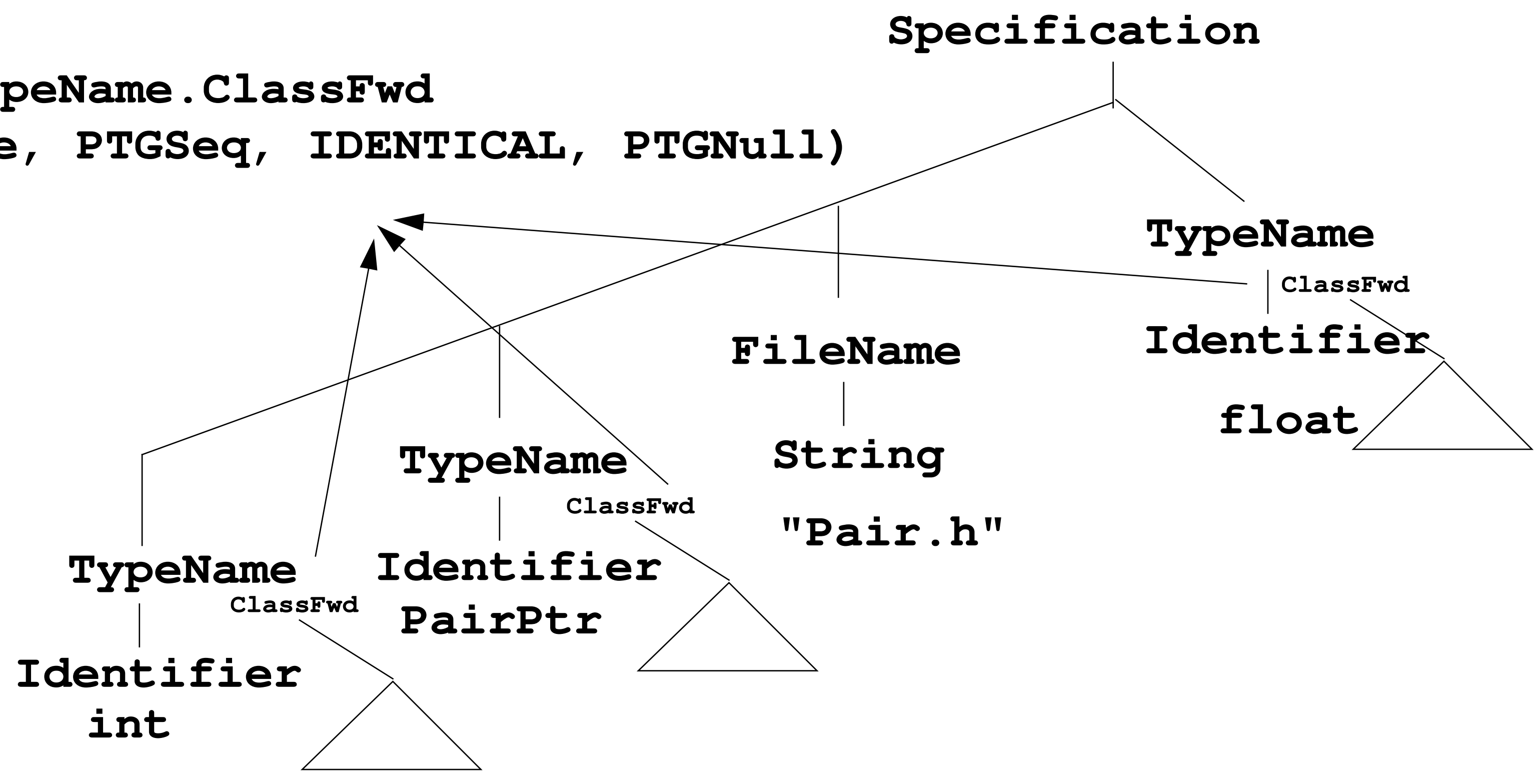
```
RULE: LoopStmt ::= Condition Body COMPUTE
```

eine Pattern-Anwendung als Attribut-Berechnung:

```
   LoopStmt.Code =
      PTGWhile (Condition.code, Body.Code);

END;
```

Zieltexte zusammensetzen: Attribute aus Unterbaum

```
CONSTITUENTS TypeName.ClassFwd
   WITH (PTGNode, PTGSeq, IDENTICAL, PTGNull)
```



CONSTITUENTS fasst Attribute aus Unterbaum zusammen, hier **TypeName.ClassFwd**

	WITH (PTGNode,	**PTGSeq,**	**IDENTICAL,**	**PTGNull)**
Bedeutung:	Typ	2-stellige Verknüp-fungs-funktion	1-stellige Funktion, auf jedes Attribut angewandt	0-stellige Funktion für optionale Teilbäume