Generierung von Software aus Spezifikationen

Prof. Dr. Uwe Kastens

WS 2002/2003

Ziele

Die Teilnehmer sollen lernen

- **Generatoren** für spezielle Software-Aufgaben **anzuwenden**,
- anwendungsspezifische **Sprachen zu entwerfen**,
- Programm**generatoren dafür herzustellen**,
- das **Werkzeugsystem Eli** für solche Aufgaben einzusetzen.

Sie sollen ein **selbstdefiniertes Anwendungsprojekt** durchführen können.

Inhalt (aktualisiert)

Phase 1: Vorlesung, praktisches Tutorium und selbständige Vor- und Nacharbeit eng verzahnt

Phase 2: selbständig Projekte bearbeiten.
In den Vorlesungsstunden beraten wir zu Problemen und tauschen Erfahrungen aus

Referenzen

- U. Kastens: Generierung von Software aus Spezifikationen
  Elektronisches Skript, WS 2002/2003
  http://www.upb.de/cs/ag-kastens/gss
- Eli Online Documentation
  http://www.uni-paderborn.de/project-hp/eli.html
- U. Kastens: Praktikum Sprachimplementierung mit Werkzeugen
  Elektronisches Skript, WS 1999/2000
  http://www.upb.de/cs/ag-kastens/eliprakt/
- Artikel zum Einsatz von Eli
  http://www.uni-paderborn.de/fachbereich/AG/agkastens/eli/bib/eli_bibD.html
  http://www.uni-paderborn.de/fachbereich/AG/agkastens/literatur/literatur.eli.html
  http://www.uni-paderborn.de/fachbereich/AG/agkastens/forschung/visuell.html

Artikel zu Wiederverwendung:

- Ch. W. Kruger: Software Reuse, ACM Computing Surveys, 24(2), 1992
- R. Prieto-Diaz: Status Report: Software reusability, IEEE Software, 10(3), 1993

1.1 Wiederverwendung (Reuse)
Was wird wiederverwendet?

Produkt	**Wiederverwendung von ...**
Bibliothek mit Funktionen	Implementierung
Modul, Komponente	Code
generischer Modul	geplante Varianten von Code
Software Architektur	Entwurf
Framework (Programmgerüst)	Entwurf und Code
Entwurfsmuster	Strategien zur Konstruktion
Generator	Wissen, wie man Lösungen aus Beschreibungen konstruiert
Konstruktionsprozess	Wissen, wie man Werkzeuge anwendet und kombiniert

Ch. W. Kruger: Software Reuse, ACM Computing Surveys, 24(2), 1992

R. Prieto-Diaz: Status Report: Software reusability, IEEE Software, 10(3), 1993

Wo wird Wiederverwendung praktiziert?

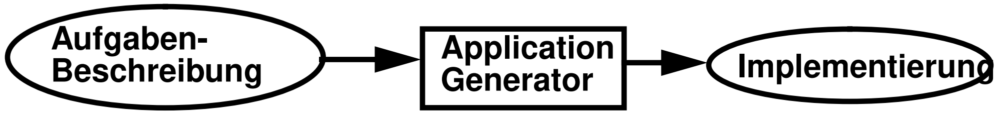
Anwendungsgebiet bestimmt das Potenzial der Wiederverwendung

breites, allgemeines Gebiet — **enge** Wiederverwendung (horizontal)

enges Gebiet,
gut verstandene Begriffe — **mächtige** Wiederverwendung (vertikal)
Generatoren, Frameworks

Anwendungsspezifische Sprachen
(Domain specific languages, DSL)

Anwendungsgebiete außerhalb der Informatik
- Robotorsteuerung
- Börsenhandel
- mechanische Steuerungssysteme
- Musiknoten



Gebiete der Software -Technik
- Datenbank-Reports
- Bedienungsschnittstellen
- Repräsentation von Datenstrukturen (XML)

Sprachimplementierung als Anwendungsgebiet
- Scanner spezifiziert durch reguläre Ausdrücke
- Parser spezifiziert durch kontextfreie Grammatiken
- Sprachimplementierung spezifiziert für *Eli*

Wie wird Wiederverwendung organisiert?

Wie	Produkte	Konsequenzen
ad hoc	• Code kopieren und modifizieren • inkrementelles Adaptieren durch oo Unterklassen	• keine Kosten a priori • sehr problematisch für die Wartung
geplant	• oo Bibliotheken, Frameworks • Spezialisierung von Klassen	• hohe Kosten a priori • wirksame Wiederverwendung
automatisch	• Generatoren, Entwicklungsumgebungen	• hohe Kosten a priori • wirksame Wiederverwendung • große kognitive Distanz

Wer bereitet Wiederverwendung vor, wer nutzt sie?

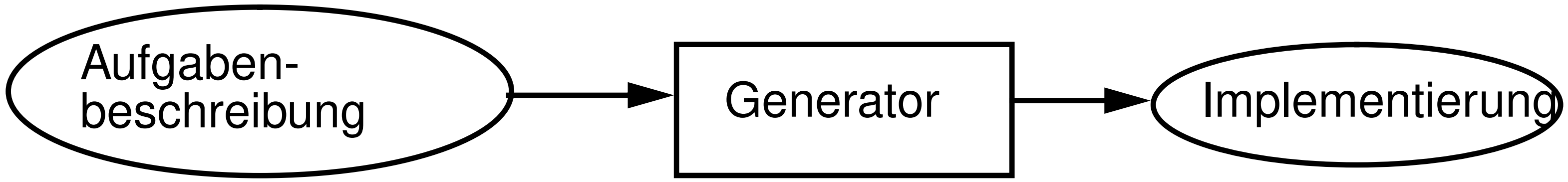
Bereitstellung und Wiederverwendung auf demselben Erfahrungsstand:

- dieselben Personen, Gruppen, Berufe
- Anbieter nimmt an, dass der Benutzer die Produkte genauso versteht
- Beispiele: Wiederverwendung von Code, Entwurfsmuster

Experte bietet Produkte an zum wiederverwenden durch Amateure:

- Wiederverwendung überspannt eine große **kognitive Distanz**
- **Expertenwissen** wird Nicht-Experten verfügbar gemacht
- erfordert, dass das Anwendungsgebiet **vollständig verstanden ist**
- erfordert Begriffe und **Konzepte auf hohem Niveau**
- Beispiele: Generatoren, Frameworks, Entwicklungsumgebungen

1.2. Das Generator-Prinzip



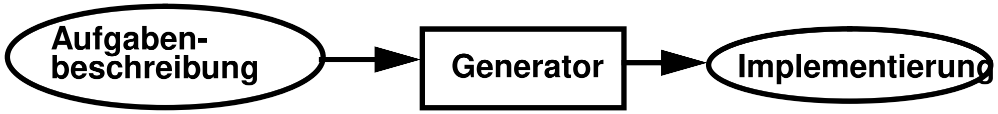
Application generator: die wirksamste Wiederverwendungs-Methode
[Ch. W. Kruger: Software Reuse]

spezielles Anwendungsgebiet eng	vollständig verstanden Implementierungen automatisch erzeugt
Abstraktionen auf hohem Niveau	abgebildet auf ausführbare software
Benutzer versteht **Abstraktionen** des Anwendungsgebietes	**Experte** versteht **Implementierungsmethoden**

große kognitive Distanz
Generator macht Expertenwissen verfügbar

Beispiele: Datenbank-Report-Generator
Bedienungsschnittstellen-Generator
Parser-Generator

Spezifikationssprachen für Generatoren



Anwendungsspezifische Sprachen
(Domain specific languages, DSL)

Anwendungsgebiete außerhalb der Informatik
- Robotorsteuerung
- Börsenhandel
- mechanische Steuerungssysteme
- Musiknoten

Gebiete der Software -Technik
- Datenbank-Reports
- Bedienungsschnittstellen
- Repräsentation von Datenstrukturen (XML)

Sprachimplementierung als Anwendungsgebiet
- Scanner spezifiziert durch reguläre Ausdrücke
- Parser spezifiziert durch kontextfreie Grammatiken
- Sprachimplementierung spezifiziert für *Eli*

Generator: **übersetzt Spezifikationssprache**
in ein ausführbares **Programm und/oder Daten**
benutzt anwendungsspezifische
Methods und Techniken

Beispiel: Wrapper-Klassen generieren

```
int;
PairPtr; "Pair.h"
float;
```

Spezifikation

generierter
C++ Modul

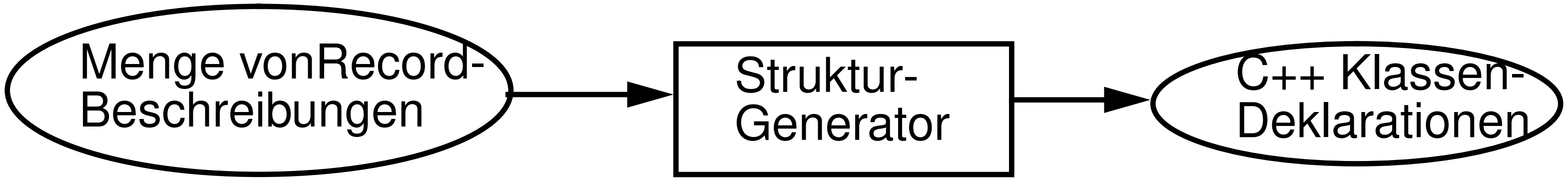
```
#ifndef WRAPPER_H
#define WRAPPER_H

#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;
};
```

```
class intWrapper : public Object {
private:
   int v;
public:
   intWrapper (int value)
       { kind = intKind; v = value; }
   int getValue () { return v; }
};

class PairPtrWrapper : public Object {
private:
   PairPtr v;
public:
   PairPtrWrapper (PairPtr value)
       { kind = PairPtrKind; v = value; }
   PairPtr getValue () { return v; }
};

class floatWrapper : public Object {
private:
   float v;
public:
   floatWrapper (float value)
       { kind = floatKind; v = value; }
   float getValue () { return v; }
};

#endif
```

```
#include „example.h"

int Object::getintValue () {
   if (kind == intKind)
       return ((intWrapper*)this)->getValue();
   else
       throw WrapperExcept();
}

PairPtr Object::getPairPtrValue () {
   if (kind == PairPtrKind)
     return ((PairPtrWrapper*)this)->getValue();
   else
       throw WrapperExcept();
}

float Object::getfloatalue () {
   if (kind == floatKind)
       return ((floatWrapper*)this)->getValue();
   else
       throw WrapperExcept();
}
```

Beispiel: Struktur-Generator

Generator um Record-Strukturen zu implementieren,
zu benutzen als **Hilfsmittel bei der Software-Konstruktion**




```
Customer ( addr:      Address;
           account:   int; )

Address (  name:  String;
           zip:   int;
           city:  String; )

import String from "util.h"
```

```
#include "util.h"

typedef class Customer_Cl *Customer;
typedef class Address_Cl *Address;

class Customer_Cl {
   private:
      Address addr_fld;
      int account_fld;
   public:
      Customer_Cl
         (Address addr, int account)
      {  addr_fld=addr;
         account_fld=account;  }
...
};
```

Aufgabenzerlegung für Implementierung von Spezifikationssprachen

Structuring	**Lexical analysis**	**Scanning** **Conversion**
	Syntactic analysis	**Parsing** **Tree construction**
Translation	**Semantic analysis**	**Name analysis** **Property analysis**
	Transformation	**Data mapping** **Action mapping**

[W. M. Waite, L. R. Carter: Compiler Construction, Harper Collins College Publisher, 1993]

Entspricht der Aufgabenzerlegung für

Frontend von Programmiersprach-**Übersetzer** (ohne Code-Erzeugung)

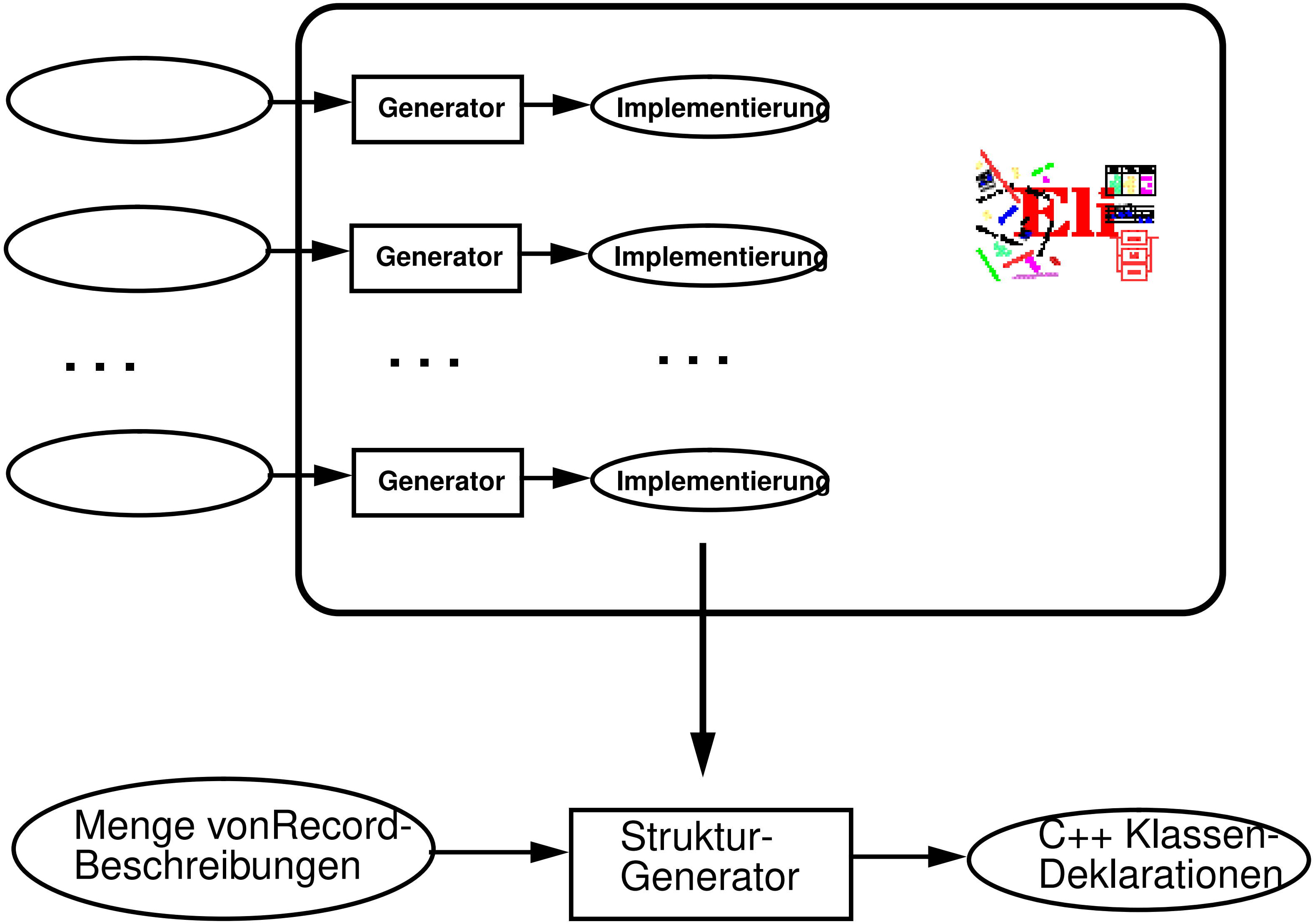
Source-to-Source Übersetzung

Aufgabenzerlegung für den Struktur-Generator

Structuring	Lexical analysis	Symbole der Berschreibung erkennen Namen speichern und codieren
	Syntactic analysis	Struktur der Beschreibung erkennen Struktur als Baum repräsentieren
Translation	Semantic analysis	Namen an Strukturen und Felder binden Eigenschaften speichern und prüfen
	Transformation	Klassen-Deklarationen generieren mit Konstructoren und Zugriffsmethoden

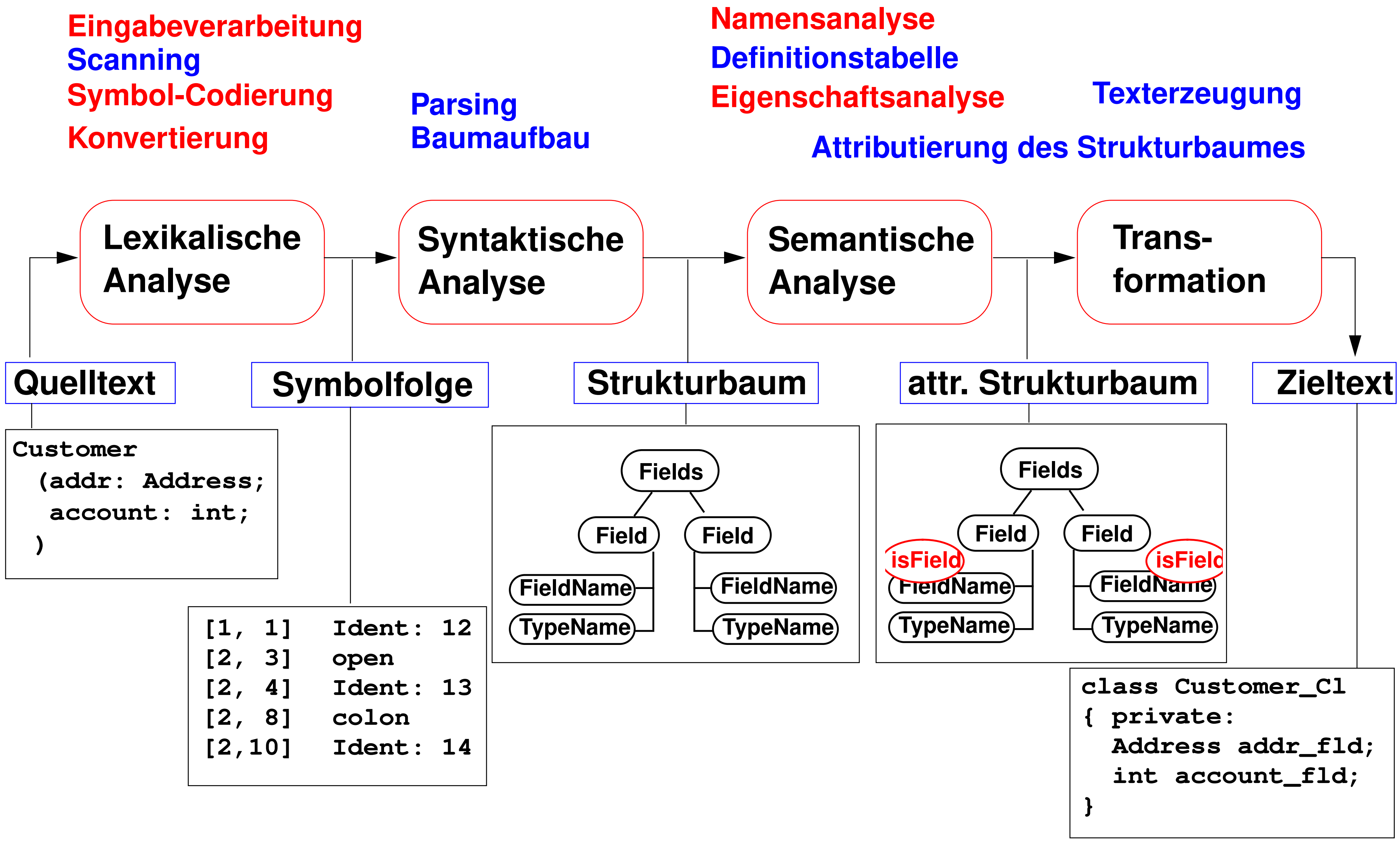
```
Customer ( addr:      Address;
           account:   int; )

Address (  name:  String;
           zip:   int;
           city:  String; )

import String from "util.h"
```

Eli generiert Struktur-Generator

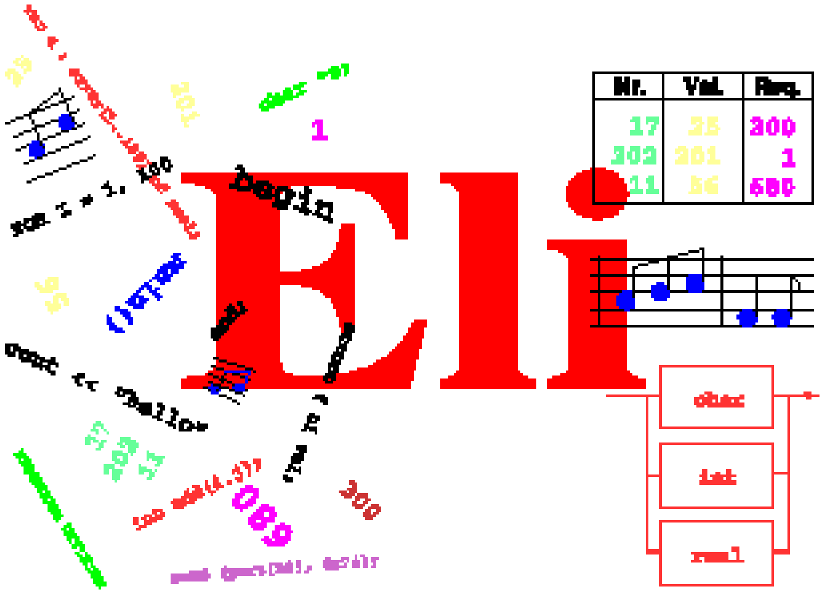


Generator-Architektur gemäß Aufgabenzerlegung

Spezialisierte Werkzeuge erledigen spezielle Aufgaben für die Herstellung des Produktes:



Das Eli System



- **Werkzeugsystem zur Sprachimplementierung**
- Einsetzbar für jede Art von Sprachen:
  Anwendungs-Spezifikationssprachen,
  Programmiersprachen
- **state-of-the-art Compiler Technik**
- Basiert auf der (vollständigen)
  Aufgabenzerlegung (siehe GSS-1.12)
- **Automatischer Herstellungsprozess**

- In vielen **praktischen Projekten** weltweit benutzt
- Entwickelt, erweitert und gewartet seit 1989 von
  William M. Waite (University of Colorado at Boulder),
  Uwe Kastens (University of Paderborn), and
  Antony M. Sloane (Macquarie University, Sydney)
- **Frei verfügbar** via Internet:
  http://www.uni-paderborn.de/project-hp/eli.html

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
```

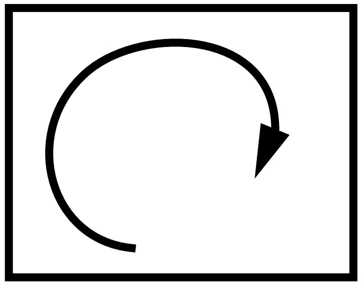
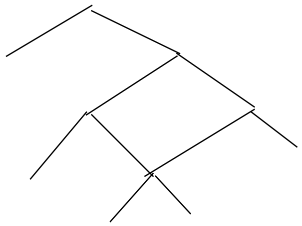
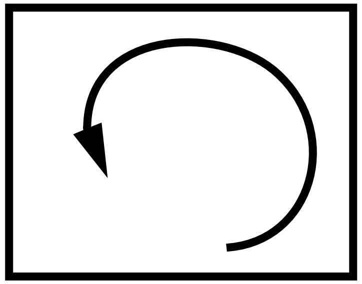
```
#ifndef WRAPPER_H
#define WRAPPER_H

#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**

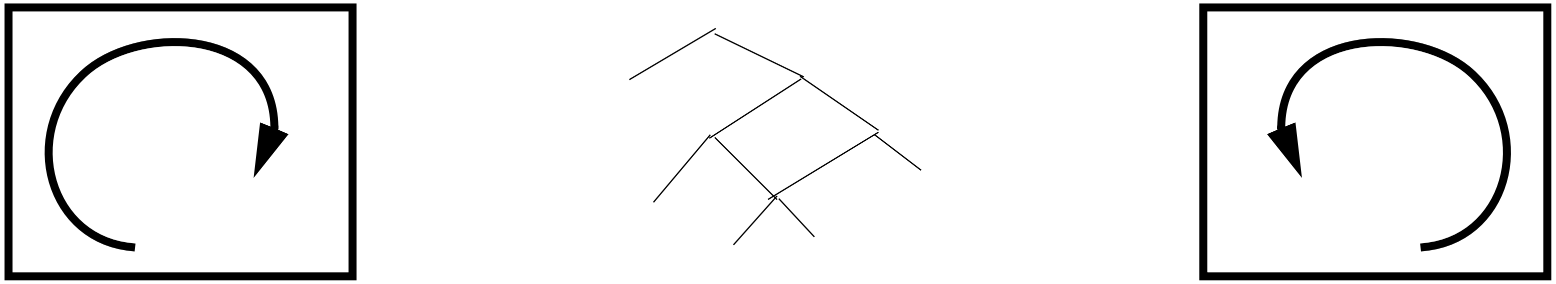
Puffer löst Structure Clash auf

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

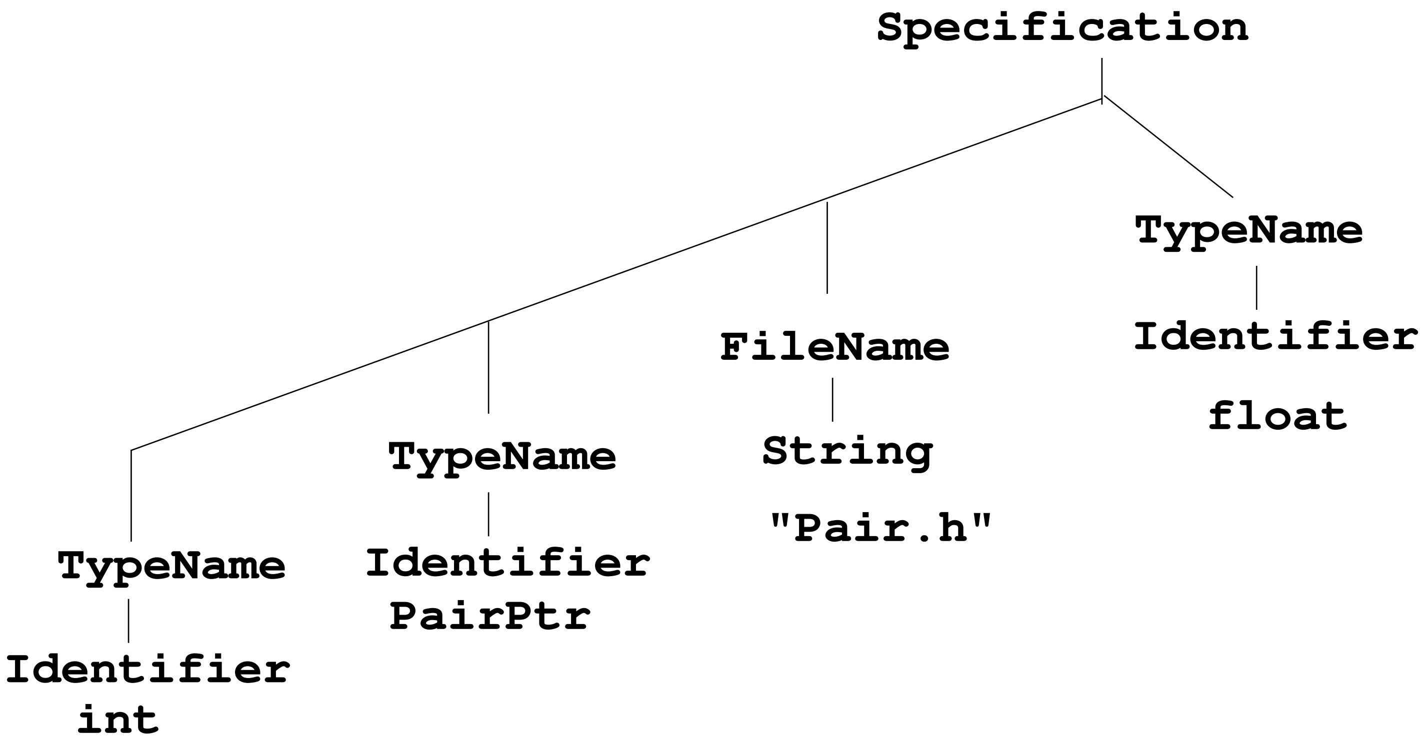
Mit der Eli-Spezifikation **`$/Output/PtgCommon.fw`** bekommt man einige dieser nützlichen Pattern-Definitionen: **`Seq, CommaSeq, AsIs, Numb`**

Zieltext aus Strukturbaum erzeugen

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

Wrapper-Generator:

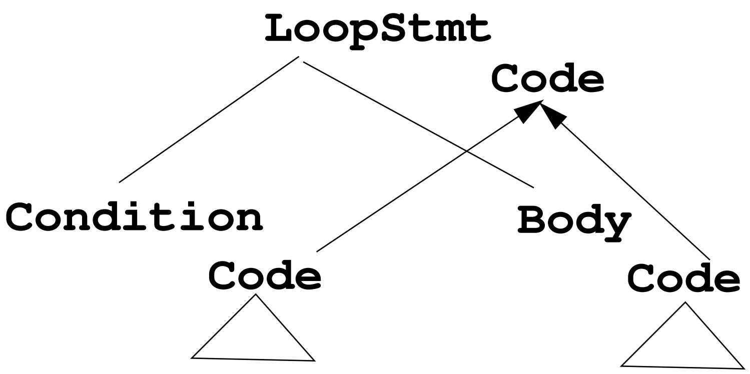
`Specification` ist eine Folge von **`TypeName`**- und **`FileName`**-Knoten



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

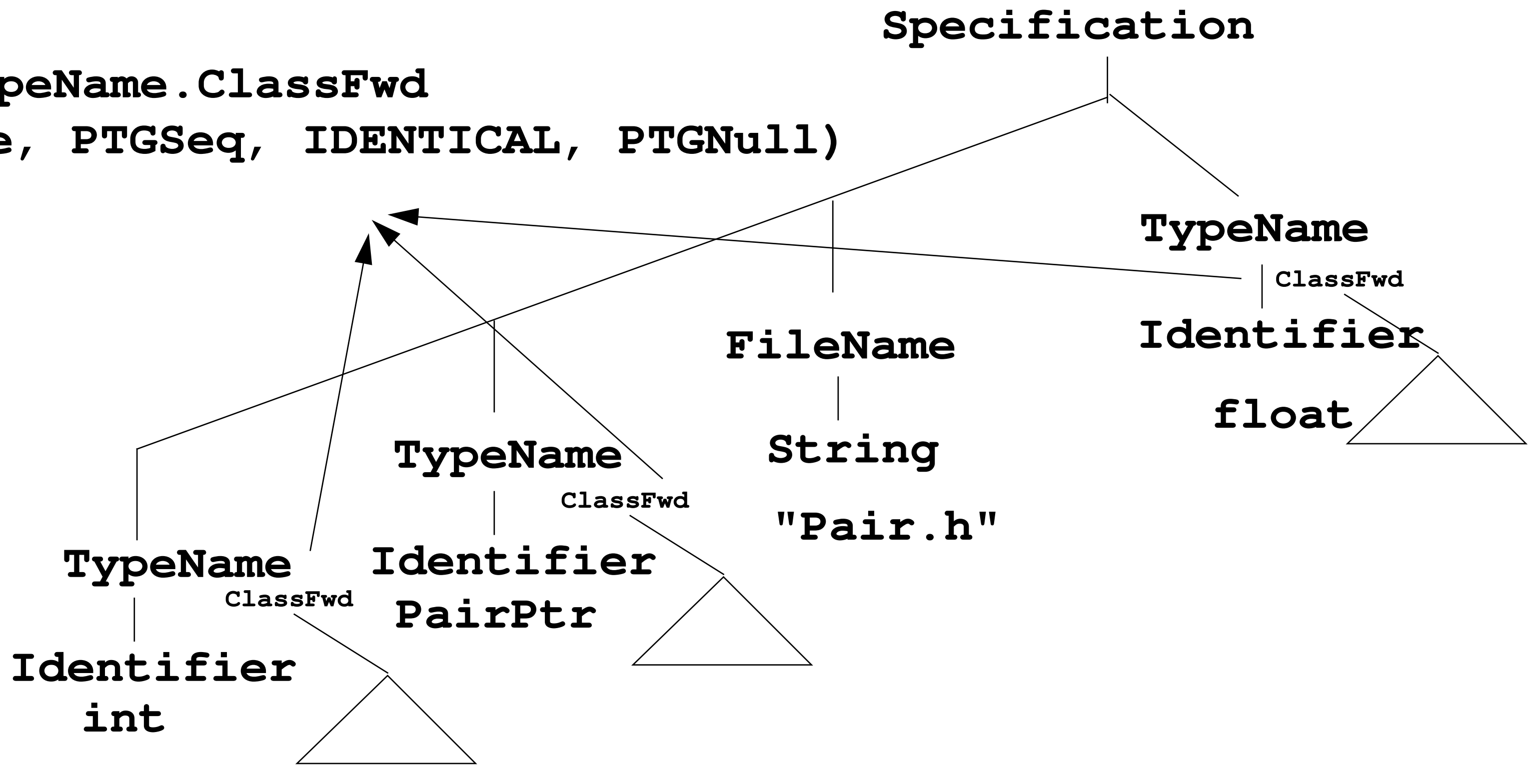
```
RULE: LoopStmt ::= Condition Body COMPUTE
```

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 Verknüpfungsfunktion	1-stellige Funktion, auf jedes Attribut angewandt	0-stellige Funktion für optionale Teilbäume

3. Bäume aufbauen - Überblick

Notation und Struktur der Eingabe prüfen und als Baum repräsentieren

Aufgaben

Eingabeverarbeitung
Scanning
Symbol-Codierung
Konvertierung

Parsing
Baumaufbau

Module

Lexikalische Analyse

Syntaktische Analyse

Schnittstellen: Repräsentation der Eingabe

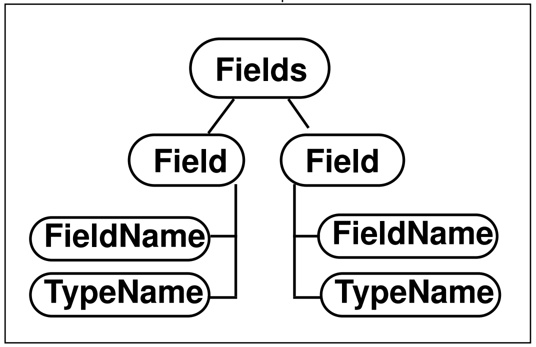
Quelltext

```
Customer
  (addr: Address;
   account: int;
  )
```

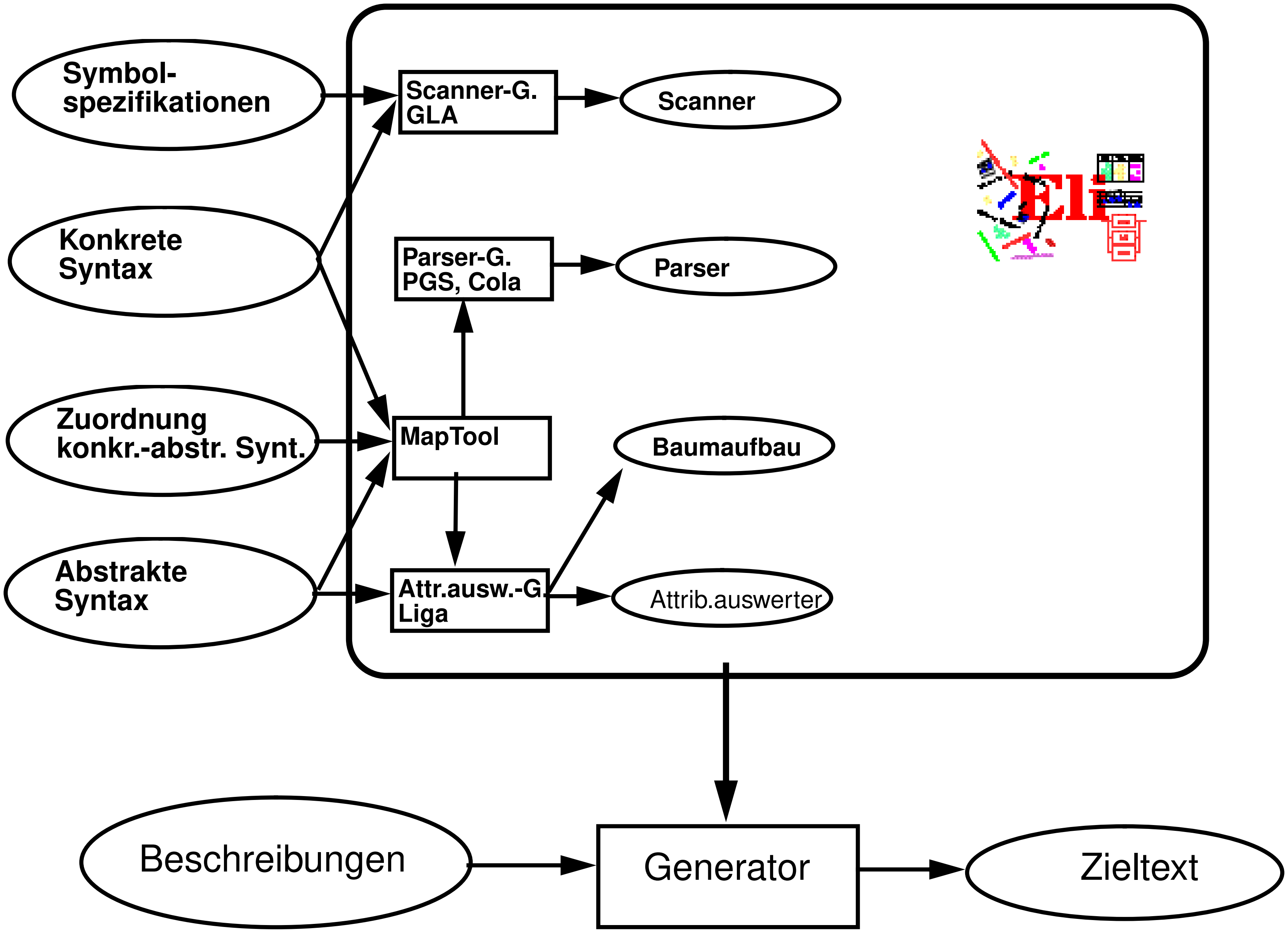
Symbolfolge

```
[1, 1]   Ident: 12
[2, 3]   open
[2, 4]   Ident: 13
[2, 8]   colon
[2,10]   Ident: 14
```

Strukturbaum



Eli: Spezifikation und Generierung des Baumaufbaus



Spezifikationen am Beispiel des Wrapper-Generators

Symbol-spezifikationen

Notation der Grundsymbole
.gla

```
Identifier:   C_IDENTIFIER
String:       C_STRING_LIT
              C_COMMENT
```

Konkrete Syntax

Struktur der Eingabe
.con

```
Specification:  Sequence.
Sequence:       Sequence Element / .
Element:        TypeName ';'.
Element:        FileName.
TypeName:       Identifier.
FileName:       String.
```

Zuordnung konkr.-abstr. Synt.

.map

leer -
konkrete und abstrakte Syntax sind gleich

Abstrakte Syntax

Struktur der Bäume
.lido

```
RULE: Specification LISTOF Element
COMPUTE ...

SYMBOL FileName COMPUTE ...
SYMBOL TypeName COMPUTE ...
```

nur die Symbole und Produktionen, die Berechnungen haben

Strukturierung am Beispiel Terminliste

Neues Beispiel für die Spezifikationen der Strukturierung bis zum Baumaufbau:

Eingabesprache: Listen von Terminangaben:

```
1.11.       20:00     "Theaterbesuch"
Do          14:15     "GSS Vorlesung"
werktags    12:05     "Essen im Palmengarten"
Mo, Do      8:00      "Dekanat"
31.12.      23:59     "Jahresende"
12/31       23:59     "End of year"
```

Konkrete Syntax

spezifiziert die **Struktur der Eingabe** durch eine kontext-freie Grammatik:

```
Termine:       Eintrag+ .
Eintrag:       Tag Zeit Beschreibung .
Tag:           Datum / Muster .
Datum:         TagNum '.' MonNum '.' /
               MonNum '/' TagNum .
TagNum:        Integer .
MonNum:        Integer .
Muster:        Muster ',' WochenTag /
               WochenTag /
               TagesGruppe .
WochenTag:     TagesName .
TagesGruppe:   'werktags' .
Zeit:          Zeitpunkt .
Zeitpunkt:     Uhrzeit .
```

Notation:

- Folge von Produktionen
- literale Terminale in '
- EBNF-Konstrukte:
  / Alternative
  ( ) Klammern
  [ ] Option
  +, * Wiederholung
  // Wiederholung mit Trenner
  Bedeutung siehe GdP

Beispiel:

```
Do          14:15     "GSS Vorlesung"
werktags    12:05     "Essen im Palmengarten"
Mo, Do      8:00      "Dekanat"
31.12.      23:59     "Jahresende"
12/31       23:59     "End of year"
```

Abstrakte Syntax

spezifiziert die **Strukturbäume** durch eine kontext-freie Grammatik:

```
RULE pTermine:      Termine   LISTOF Eintrag               END;
RULE pZeitEintrag:  Eintrag   ::= Tag Zeit Beschreibung END;

RULE pDatum:        Tag       ::= Datum                    END;
RULE pMuster:       Tag       ::= Muster                   END;

RULE pDatumZahlen:  Datum     ::= TagNum MonNum            END;
RULE pTag:          TagNum    ::= Integer                  END;
RULE pMonat:        MonNum    ::= Integer                  END;

RULE pMusterListe:  Muster    LISTOF Element               END;
RULE pTagesName:    Element   ::= TagesName                END;
RULE pWerktage:     Element   ::= 'werktags'               END;

RULE pZeitpunkt:    Zeit      ::= Zeitpunkt                END;
RULE pUhrzeit:      Zeitpunkt ::= Uhrzeit                  END;
```

Notation:

- Sprache Lido für Berechnungen in Strukturbäumen
- optional benannte Produktionen,
- kein EBNF außer LISTOF (evtl. leere Folge)

Beispiel eines Strukturbaumes

- Produktionsnamen als Knotentypen
- Werte der Terminale an den Blättern

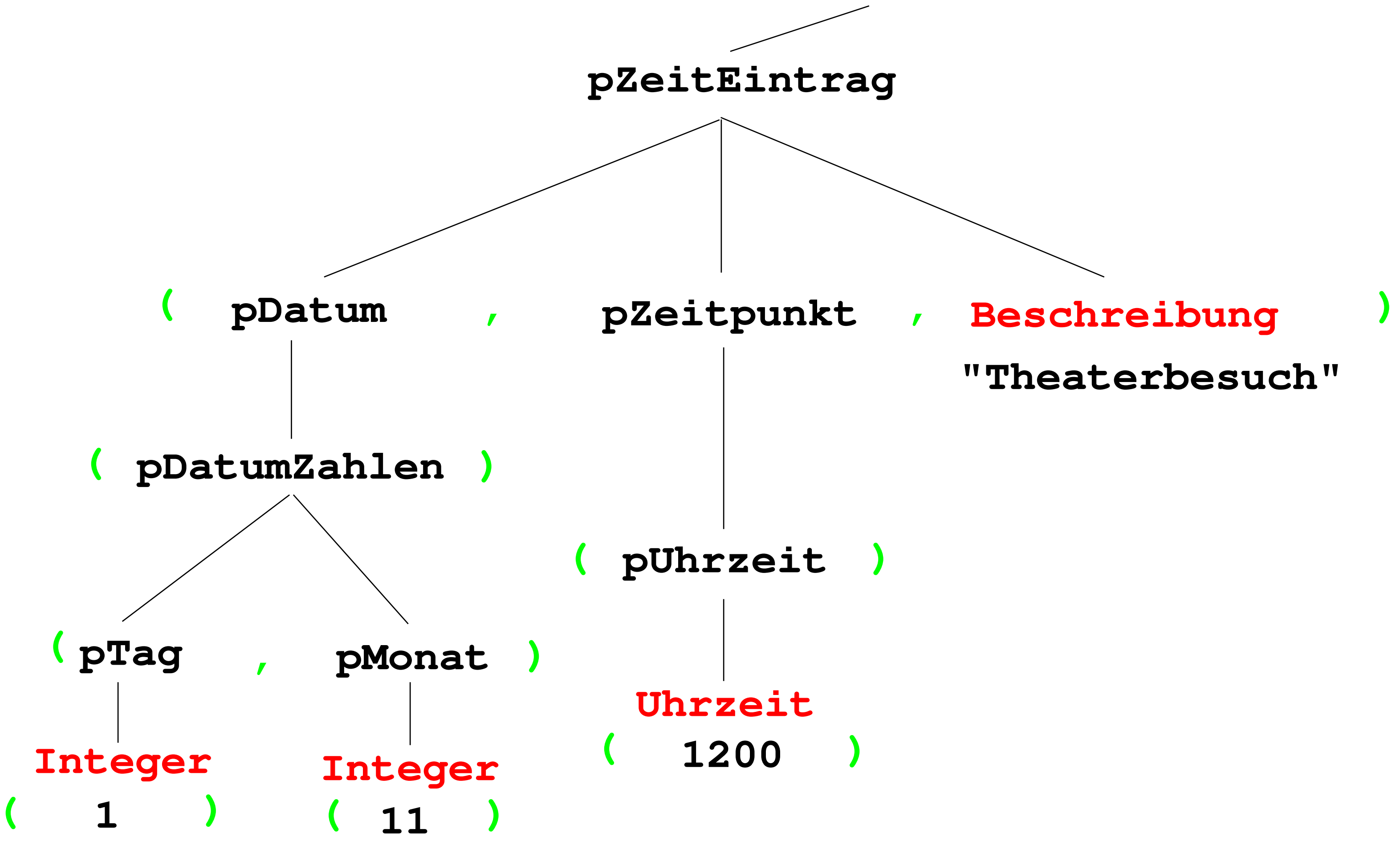
Ausgabe automatisch erzeugt durch Eli‘s Unparser-Generator

```
pZeitEintrag(
   pDatum(pDatumZahlen(pTag(1),pMonat(11))),
   pZeitpunkt(pUhrzeit(1200)),"Theaterbesuch"),
pZeitEintrag(
   pMuster(pTagesName(4)),
   pZeitpunkt(pUhrzeit(855)),"GSS Vorlesung"),
pZeitEintrag(
   pMuster(pWerktage()),
   pZeitpunkt(pUhrzeit(725)),"Essen im Palmengarten"),
pZeitEintrag(
   pMuster(pTagesName(1),pTagesName(4)),
   pZeitpunkt(pUhrzeit(480)),"Dekanat"),
pZeitEintrag(
   pDatum(pDatumZahlen(pTag(31),pMonat(12))),
   pZeitpunkt(pUhrzeit(1439)),"Jahresende"),
pZeitEintrag(
   pDatum(pDatumZahlen(pTag(31),pMonat(12))),
   pZeitpunkt(pUhrzeit(1439)),"End of year")
```

Strukturbaumausschnitt graphisch

- Produktionsnamen als Knotentypen
- Werte der Terminale an den Blättern

Ausgabe automatisch erzeugt durch Eli‘s Unparser-Generator,
Baumstruktur durch Klammerung



Symbolabbildung: konkrete - abstrakte Syntax

konkrete Syntax:

```
Muster:       Muster ',' WochenTag /
              WochenTag /
              TagesGruppe .
WochenTag:    TagesName .
TagesGruppe: 'werktags' .
```

Vereinfachung der abstrakten Syntax

Menge von Nichtterminalen der konkreten Syntax ->

ein Nichtterminal der abstrakten Syntax

Abbildung:

```
MAPSYM
Element ::= TagesGruppe WochenTag.
```

`Element` steht in der abstrakten Syntax für die angegebenen Nichtterminale

abstrakte Syntax:

```
RULE pMusterListe:  Muster    LISTOF Element          END;
RULE pTagesName:    Element   ::= TagesName            END;
RULE pWerktage:     Element   ::= 'werktags'           END;
```

Regelabbildung

konkrete Syntax:

```
Datum:          TagNum '.' MonNum '.' /
                MonNum '/' TagNum .
```

Abbildung:

```
MAPRULE
Datum: TagNum '.' MonNum '.' < $1 $2 >.
Datum: MonNum '/' TagNum     < $2 $1 >.
```

unterschiedliche Produktionen der konkreten Syntax

werden in der abstrakten Syntax **vereinheitlicht**

abstrakte Syntax:

```
RULE pDatumZahlen:   Datum ::= TagNum MonNum END;
```

Literale und nicht-literale Terminale

Definition der Notation von

- literalen Terminalen (unbenannt): **in der konkreten Syntax**
- nicht-literalen Terminalen (benannt): in zusätzlicher **Spezifikation für den Scanner-Generator**

```
Termine:       Eintrag+ .
Eintrag:       Tag Zeit Beschreibung .
Tag:           Datum / Muster .
Datum:         TagNum '.' MonNum '.' /
               MonNum '/' TagNum .
TagNum:        Integer .
MonNum:        Integer .
Muster:        Muster ',' WochenTag /
               WochenTag /
               TagesGruppe .
WochenTag:     TagesName .
TagesGruppe:   'werktags' .
Zeit:          Zeitpunkt .
Zeitpunkt:     Uhrzeit .
```

Spezifikation nicht-literaler Terminale

Der Generator GLA generiert einen Scanner aus

- Notation der literalen Terminale
  von Eli aus der konkreten Syntax extrahiert
- Spezifikation der nicht-literalen Terminale
  in Dateien vom Typ **`.gla`**

Form der Spezifikationen:

```
Name:          $  regulärer Ausdruck                    [CodierFunktion]

TagesName:     $  Mo|Di|Mi|Do|Fr|Sa|So                  [mktag]

Uhrzeit:       $(([0-9]|1[0-9]|2[0-3]):[0-5][0-9])      [mkuhrzeit]
```

Vorgefertigte Spezifikationen:

```
Beschreibung:   C_STRING_LIT
Integer:        PASCAL_INTEGER
```

Scanner-Spezifikation: Reguläre Ausdrücke

<table>
<tr><th>Notation</th><th>Zeichenfolgen, die akzeptiert werden</th></tr>
<tr><td><code>c</code></td><td>das Zeichen c; außer Zeichen mit spezieller Bedeutung, siehe c</td></tr>
<tr><td><code>\c</code></td><td>Zwischenraum, Tab, Zeilenwechsel, <code>\".[]^()|?+*{}/$<</code></td></tr>
<tr><td><code>"s"</code></td><td>die Zeichenfolge s</td></tr>
<tr><td><code>.</code></td><td>jedes einzelne Zeichen außer Zeilenwechsel</td></tr>
<tr><td><code>[xyz]</code></td><td>genau ein Zeichen aus der Menge {x, y, z}</td></tr>
<tr><td><code>[^xyz]</code></td><td>genau ein Zeichen, das nicht aus der Menge {x, y, z} ist</td></tr>
<tr><td><code>[c-d]</code></td><td>genau ein Zeichen, dessen ASCII-Code zwischen c und d einschließlich liegt</td></tr>
<tr><td><code>(e)</code></td><td>durch e spezifizierte Zeichenfolge</td></tr>
<tr><td><code>ef</code></td><td>durch e spezifizierte Zeichenfolge gefolgt von einer durch f spezifizierten</td></tr>
<tr><td><code>e | f</code></td><td>durch e oder durch f spezifizierte Zeichenfolge</td></tr>
<tr><td><code>e?</code></td><td>leere oder durch e spezifizierte Zeichenfolge</td></tr>
<tr><td><code>e+</code></td><td>ein oder mehrere Auftreten von Zeichenfolgen, die durch e spezifiziert sind</td></tr>
<tr><td><code>e*</code></td><td>leere oder durch e+ spezifizierte Zeichenfolge</td></tr>
<tr><td><code>e {m,n}</code></td><td>mindestens m und höchstens n Zeichenfolgen, die durch e spezifiziert sind</td></tr>
</table>

e und f sind reguläre Ausdrücke, wie sie hier definiert sind.

Jeder reguläre Ausdruck **akzeptiert die längste Zeichenfolge**, die seiner Definition genügt.

Auflösung von **Mehrdeutigkeiten**:
1. die **längere akzeptierte Zeichenfolge**
2. bei gleicher Länge: die **früher angegebene Regel**

Scanner-Spezifikation: Programmierte Scanner

Das Akzeptieren bestimmter Zeichenfolgen kann durch eine Funktion implementiert werden. Damit kann man Zeichenfolgen operational spezifizieren, die durch reguläre Ausdrücke nicht oder schwierig beschreibbar sind.

Der Anfang der Zeichenfolge wird durch einen regulären Ausdruck spezifiziert, es folgt der geklammerte Name der Funktion. z. B. für Zeilenkommentare wie in Ada

```
$-- (auxEOL)
```

Parameter der Funktion: ein Pointer auf das erste Zeichen der bisher akzeptiertenFolge, und ihre Länge. Ergebnis: Pointer auf das erste Zeichen nach der gesamten Folge:

```
char *Name(char *start, int length)
```

Einige verfügbare programmierte Scanner und was sie akzeptieren:

auxEOL	alle Zeichen bis zum nächsten Zeilenwechsel einschließlich
auxCString	ein C-String-Literal nach dem einleitenden "
auxM3Comment	Modula-3-Kommentar nach dem einleitenden (*, mit (* und *) geklammert, auch geschachtelt
Ctext	C-Verbundanweisungen nach dem einleitenden { mit { und } geklammert, auch geschachtelt

Scanner-Spezifikation: Codier-Funktionen

Die **akzeptierte Zeichenfolge** wird an eine Codier-Funktion übergeben (`start`, `length`)

Sie liefert die Codierung des akzeptierten Symbols (`intrinsic`)
d. h. eine ganze **Zahl, die seine Identität oder sein Wert repräsentiert.**

Dazu wird ggf. die Zeichenfolge **gespeichert und/oder konvertiert**.

Alle Codier-Funktionen haben dieselbe **Signatur**:

```
void Name (char *start, int length, int *class, int *intrinsic)
```

Die **Symbolklasse** (Terminalsymbol) kann ggf. geändert werden (`class`), z. B. um Wortsymbole von Bezeichnern zu unterscheiden.

Verfügbare Codierfunktionen:

`mkidn`	Zeichenfolge in eine Hash-Tabelle eintragen und bijektiv codieren
`mkstr`	Zeichenfolge speichern; jede erhält einen neuen Code
`c_mkstr`	C-String-Literal in seinen Wert konvertieren, speichern und codieren
`mkint`	Ziffernfolge in eine ganze Zahl konvertieren und diesen Wert als Code liefern
`c_mkint`	Literal für ganze Zahlen in C konvertieren und diesen Wert als Code liefern

Scanner-Spezifikation: Vorgefertigte Spezifikationen

Vorgefertigte vollständige Spezifikationen (regulärer Ausdruck, ggf. programmierter Scanner und Codier-Funktion) können **unter ihrem Namen** abgerufen werden:

```
Identifier:  C_IDENTIFIER
```

Für viele Symbole aus Programmiersprachen sind vorgefertigte Spezifikationen verfügbar (vollständige Spezifikation in der Dokumentation):

C_IDENTIFIER, C_INTEGER, C_INT_DENOTATION, C_FLOAT,
C_STRING_LIT, C_CHAR_CONSTANT, C_COMMENT

PASCAL_IDENTIFIER, PASCAL_INTEGER, PASCAL_REAL,
PASCAL_STRING, PASCAL_COMMENT

MODULA2_INTEGER, MODULA2_CHARINT, MODULA2_LITERALDQ,
MODULA2_LITERALSQ, MODULA2_COMMENT

MODULA3_COMMENT, ADA_IDENTIFIER, ADA_COMMENT, AWK_COMMENT

SPACES, TAB, NEW_LINE
werden nur benutzt, wenn irgendein Symbol mit solch einem Zeichen beginnt, diese Zeichen aber trotzdem Symbole trennen sollen.

Die Codier-Funktionen können auch überschrieben werden.

Generierung der Baumausgabe

Strukturbaum mit Knotentypen und Werten an den terminalen Blättern ausgeben:

```
pZeitEintrag(
   pDatum(pDatumZahlen(pTag(1),pMonat(11))),
   pZeitpunkt(pUhrzeit(1200)),"Theaterbesuch"),
pZeitEintrag(
```

Ausgabe durch Aufrufe von Pattern-Konstruktor-Funktionen in Baumkontexten

Spezifikationen dafür kann der Unparser-Generator automatisch **erzeugen**:

Unparser generiert aus abstrakter Syntax:

```
Abstract.lido:tree
```

Ausgabe an der Grammatik-Wurzel:

```
SYMBOL ROOTCLASS COMPUTE
   BP_Out(THIS.IdemPtg);
END;
```

Ausgabe nicht-literaler Terminale:

```
TagesName:     $ int
Uhrzeit:       $ int
Integer:       $ int
```

Umbenennung einer PTG-Funktion:

```
#define PTGBeschreibung(x)PTGId(x)
```

Benutzung vordefinierter PTG-Pattern:

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

4. Berechnungen im Strukturbaum Übersicht

Berechnungen im Srukturbaum für beliebige Zwecke, z. B.

- **Eigenschaften** der Eingabestrukturen **berechnen, prüfen**, z. B. Typen, Wertebereiche
- **Zusammenhänge** bestimmen, prüfen; z. B. Definition - Anwendung
- **Datenstrukturen**, **Zieltext** konstruieren

Modell: Attributierte Grammatik, Spezifikationssprache **Lido**, Generator **Liga**:

Abstrakte Syntax wird erweitert:

Attribute zu Nichtterminalsymbolen:
Daten zu den Baumknoten, z. B. Expr.Value   Expr.Type   TypeName.ClassFwd

zum Speichern von **Werten an den Knoten**, Propagieren von Werten durch den Baum, Spezifizieren von Abhängigkeiten zwischen Berechnungen

Berechnungen zu Produktionen (RULE) oder Nichtterminalsymbolen (SYMBOL):
Berechnen Attributwerte aus anderen **Attributen im Kontext** (RULE oder SYMBOL), oder bewirken Effekte, z. B. Prüfungen, Meldungen, Ausgabe

Jede Berechnung wird genau einmal für jeden Knoten dieses Typs augeführt.
An jedem Knoten wird jedes seiner Attribute genau einmal berechnet.
Die **Reihenfolge der Berechnungen und den Baumdurchlauf** ermittelt der **Generator** aus den Abhängigkeiten zwischen den Attributen.

Abhängige Berechnungen

```
SYMBOL Expr, Opr: value: int SYNT;
SYMBOL Opr: left, right: int INH;
TERM Number: int;

RULE: Root ::= Expr COMPUTE
   printf ("value is %d\n", Expr.value);
END;

RULE: Expr ::= Number COMPUTE
   Expr.value = Number;
END;

RULE: Expr ::= Expr Opr Expr COMPUTE
   Expr[1].value = Opr.value;
   Opr.left = Expr[2].value;
   Opr.right = Expr[3].value;
END;

RULE: Opr ::= '+' COMPUTE
   Opr.value = ADD (Opr.left, Opr.right);
END;
RULE: Opr ::= '-' COMPUTE
   Opr.value = SUB (Opr.left, Opr.right);
END;
```

´typisierte Attribute zu Symbolen

Terminalsymbol mit int-Wert

SYNThesized Attribute werden im unteren Kontext berechnet, INHerited Attribute im oberen.

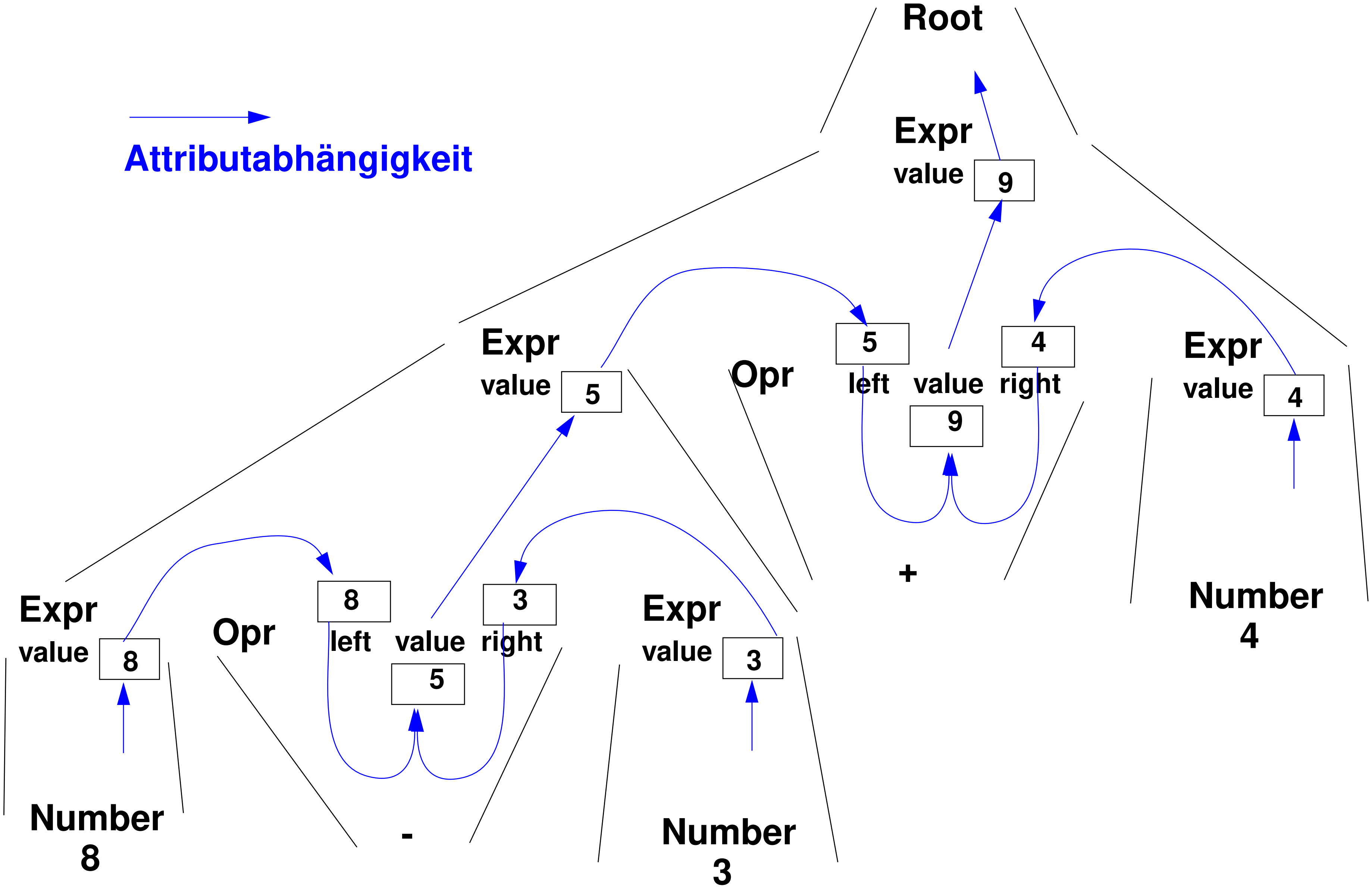
Angabe ist meist optional.

Reihenfolge der Berechnungen wird vom Generator aus den Abhängigkeiten bestimmt

Beispiel:

Berechnung und Ausgabe des Wertes eines Ausdruckes

Ein attributierter Strukturbaum



Vor- und Nachbedingungen von Berechnungen

```
RULE: Root ::= Expr COMPUTE
   Expr.print = "yes";
   printf ("n") <- Expr.printed;
END;

RULE: Expr ::= Number COMPUTE
   Expr.printed =
      printf ("%d ", Number) <-Expr.print;
END;

RULE: Expr ::= Expr Opr Expr COMPUTE
   Expr[2].print = Expr[1].print;
   Expr[3].print = Expr[2].printed;
   Opr.print = Expr[3].printed;
   Expr[1].printed = Opr.printed;
END;

RULE: Opr ::= '+' COMPUTE
   Opr.printed =
      printf ("+ ") <- Opr.print;
END;
```

Attribute **`print`** und **`printed`** haben **keine Werte** (Typ **`VOID`**)

Sie beschreiben die Zustände als **Vor- und Nachbedingungen** der Berechnungen:

`Expr.print:`

Postfix-Ausgabe bis vor diesem Knoten ist erledigt

`Expr.printed:`

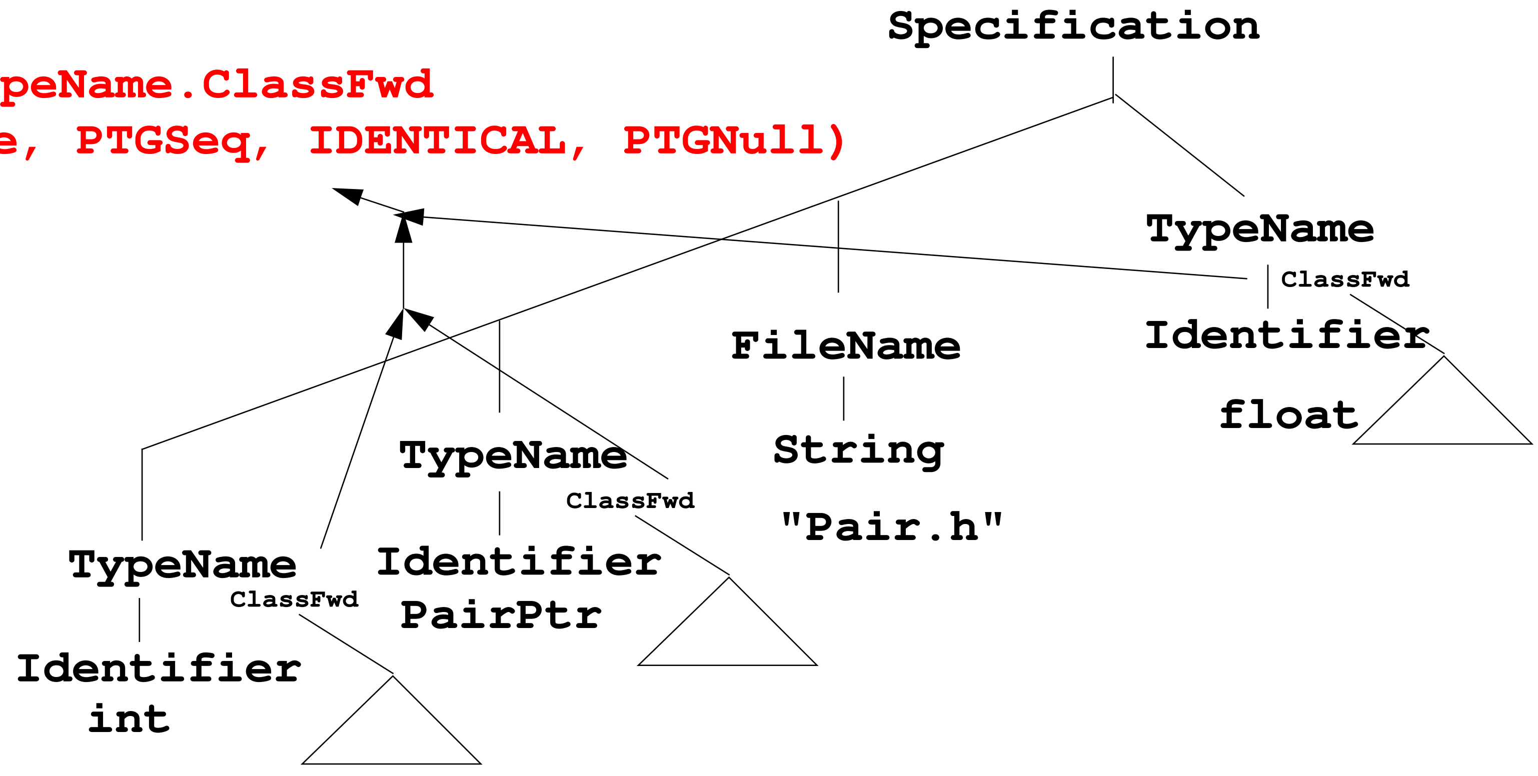
Postfix-Ausgabe bis einschließlich dieses Knotens ist erledigt

Beispiel:

Ausgabe des Ausdrucks in Postfix-Form

Schema: Attributwerte aus Unterbaum zusammenfassen

```
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

Schema: Attribut eines entfernten Oberknotens benutzen

```
SYMBOL Block: depth: int;

RULE: Root ::= Block COMPUTE
   Block.depth = 0;
END;

RULE: Block ::= '(' Sequence ')' END;
RULE: Sequence LISTOF
         Definition / Statement END;
...

RULE: Statement ::= Block COMPUTE
   Block.depth =
      ADD (INCLUDING Block.depth, 1);
END;

TERM Ident: int;

RULE: Definition ::= 'define' Ident
COMPUTE
   printf("%s defined on depth %d\n",
      StringTable (Ident),
      INCLUDING Block.depth);
END;
```

Beispiel:

Berechnung der Tiefe geschachtelter Blöcke

`INCLUDING Block.depth` bezieht sich auf das **`depth`**-Attribut des nächsten Oberknotens **`Block`**

Das **`INCLUDING`**-Attribut wird **automatisch durch die Kontexte** zwischen seiner Definition im Oberknoten und seiner Benutzung im **`INCLUDING`**-Konstrukt **transportiert**.

Schema: Vorbedingungen aus Unterbaum zusammenfassen

```
SYMBOL Block: depth: int;

RULE: Root ::= Block END;

RULE: Block ::= '(' Sequence ')'
COMPUTE
   Block.DefDone =
      CONSTITUENTS Definition.DefDone;
END;

...

RULE: Definition ::= 'define' Ident
COMPUTE
   Definition.DefDone =
   printf("%s defined in line %d\n",
      StringTable (Ident), LINE);
END;

RULE: Statement ::= 'use' Ident
COMPUTE
   printf("%s used in line %d\n",
      StringTable (Ident), LINE)
      <- INCLUDING Block.DefDone;
END;
```

Beispiel:

Erst alle Definitionen dann alle Anwendungen ausgeben

Die Attribute `DefDone` haben **keine Werte**, sondern sie spezifizieren **nur Abhängigkeiten**

Das `CONSTITUENTS`-Konstrukt brucht **keine `WITH`-Klausel**, da **keine Werte transportiert** werden.

Typische Kombination von `CONSTITUENTS`-Konstrukt und `INCLUDING`-Konstrukt, um die Reihenfolge der Seiteneffekte zu spezifizieren.

Schema: Abhängigkeiten links-abwärts durch den Baum

```
CHAIN print: VOID;

RULE: Root ::= Expr COMPUTE
   CHAINSTART HEAD.print = "yes";
   printf ("n") <- TAIL.print;
END;

RULE: Expr ::= Number COMPUTE
   Expr.print =
      printf ("%d ", Number) <-Expr.print;
END;

RULE: Expr ::= Expr Opr Expr COMPUTE
   Opr.print = Expr[3].print;
   Expr[1].print = Opr.print;
END;

RULE: Opr ::= '+' COMPUTE
   Opr.print =
      printf ("+ ") <- Opr.print;
END;
```

CHAIN spezifiziert **left-right depth-first** Abhängigkeit.

CHAINSTART im **Wurzel-Kontext der CHAIN**

Berechnungen werden mit Vor- und Nachbedingungen **in die CHAIN „eingehängt“**

CHAIN-Reihenfolge kann **überschrieben** werden.

Weggelassene CHAIN-Berechnungen werden **automatisch ergänzt**

Beispiel:

Ausgabe des Ausdrucks in Postfix-Form (vgl. GSS-4.4)

Berechnungen zu Symbolen

Berechnungen können **Symbolen zugeordnet** werden, wenn sie **für jedes Auftreten** des Symbols in Produktionen **gleich** sind.

```
SYMBOL Expr COMPUTE
   printf ("expression value %d in line %d\n", THIS.value, LINE);
END;
```

Symbol-Berechnungen können `INCLUDING`-, `CONSTITUENTS`-, `CHAIN`-Konstrukte enthalten:

```
SYMBOL Block COMPUTE
   printf ("%d uses occurred\n",
      CONSTITUENTS Usage.Count WITH (int, ADD, IDENTICAL, ZERO);
END;
```

`SYNT.a` bzw. `INH.a` gibt an, dass die Berechnung zum **unteren bzw. oberen Kontext** des Symbols gehört:

```
SYMBOL Block COMPUTE
   INH.depth = ADD (INCLUDING Block.depth);
END;
```

Berechnungen im **RULE-Kontext überschreiben Berechnungen im SYMBOL-Kontext**, z. B. für Rekursionsanfänge, Ausnahmen, oder Defaults:

```
RULE: Root ::= Block COMPUTE
   Block.depth = 0;
END;
```

Wiederverwendung von Berechnungen

```
CLASS SYMBOL IdOcc: Sym: int;
CLASS SYMBOL IdOcc COMPUTE
   SYNT.Sym = TERM;
END;
```

Berechnungen werden neuen `CLASS`-Symbolen zugeordnet.

```
SYMBOL DefIdent INHERITS IdOcc END;
SYMBOL UseIdent INHERITS IdOcc END;
```

Durch `INHERITS` werden sie an Baum-Symbole gebunden

```
CLASS SYMBOL OccRoot COMPUTE
   CHAINSTART HEAD.Occurs = 0;
   SYNT.TotalOccs = TAIL.Occurs;
END;
CLASS SYMBOL OccElem COMPUTE
   SYNT.OccNo = THIS.Occurs;
   THIS.Occurs = ADD (SYNT.OccNo, 1);
END;
```

Zusammenwirkende `CLASS`-Symbole, z. B. Auftreten eines syntaktischen Konstruktes zählen

```
SYMBOL Block      INHERITS OccRoot END;
SYMBOL Definition INHERITS OccElem END;

SYMBOL Statement INHERITS OccRoot END;
SYMBOL Usage     INHERITS OccElem END;
```

Paarweise Wiederverwendung

Auf diesem Prinzip beruht die Benutzung von Bibliotheksmodulen, z. B. zur Namensanalyse (siehe Kap. 6)

Richtlinien zum Entwurf von Berechnungen im Baum

1. Zerlege die Aufgabe in **Teilaufgaben**, die klein genug sind, um durch wenige der u. g. Spezifikationsmuster gelöst zu werden.
   Entwickle für jede Teilaufgabe ein `.lido`-Fragment und erläutere es im umgebenden `.fw`-Text.

2. Arbeite den **zentralen Aspekt der Teilaufgabe** heraus und bilde ihn auf einen der folgenden Fälle ab:

   A. Der Aspekt wird in natürlicher Weise durch **Eigenschaften von einigen Programmkonstrukten** beschrieben,
   z. B. Typen von Ausdrücken, Schachtelungstiefe von Blöcken, Übersetzung der Anweisungen eines Blockes.

   B. Der Aspekt wird in natürlicher Weise durch **Eigenschaften von Programmobjekten** beschrieben, z. B. Relativadresse von Variablen, Benutzung von Variablen vor der Definition.

   Entwirf die Berechnung wie für A oder B beschrieben.

3. Im Schritt 2 können Anforderungen an **weitere Aspekte** der Teilaufgabe entstehen (benutzte Attribute, deren Berechnung noch nicht entworfen ist): Wiederhole Schritt 2 für diese.

A: Eigenschaften von Programmkonstrukten

Bestimme den **Typ der Werte**, die die Eigenschaft beschreiben. Führe **Attribute dieses Typs zu allen Symbolen** ein, die die **Programmkonstrukte** repräsentieren. Prüfe, welcher der folgenden Fälle für die Bestimmung der Eigenschaft am besten zutrifft.

A1: Jeder **untere Kontext** bestimmt die Eigenschaft in unterschiedlicher Weise: Entwirf **RULE-Berechnungen**.

A2: Wie A1; aber **oberer Kontext**.

A3: Die Eigenschaft kann **unabhängig von den RULE-Kontexten** berechnet werden, wobei nur Attribute des Symbols oder über INCLUDING, CONSTITUENT(S), CHAIN erreichbare Attribute verwendet werden: Entwirf eine **untere (SYNT) SYMBOL-Berechnung**.

A4: Wie A3. Aber es gibt **wenige Ausnahmen**, wo entweder untere oder obere (nicht beide!) RULE-Kontexte die Eigenschaft abweichend bestimmen: Entwirf eine obere (INH) oder untere (SYNT) **SYMBOL-Berechnung** und **überschreibe sie in solchen RULE-Kontexten**.

A5: Wie A4; aber für **rekursive Symbole**: Die Fundierung der Rekursion ist die Ausnahme aus A4, z. B. Schachtelungstiefe von Blöcken.

Wenn keiner der Fälle passt, muss die Modellierung der Eigenschaft überdacht werden. Sie könnte zu komplex sein und eine weitere Zerlegung erfordern.

5. Namen und Eigenschaften

Namen in der Beschreibung repräsentieren **Objekte** als Teil der Bedeutung des Textes

Auftreten von Namen sind an **Objekte** gebunden.

Die Bindung wird durch **Gültigkeitsregeln** der Beschreibungssprache geregelt. Z. B:

- Struktur-Namen werden in der ganzen Beschreibung identisch gebunden.
- Feld-Namen werden für jede Struktur separat gebunden.
- Typ-Namen sind angewandte Auftreten von Namen.
  Sie erfordern, dass es irgendwo ein definierendes Auftreten desselben Names gibt.

einige Auftreten von Namen: **einige Bindungen:** **einige Objekte:**

```
Customer ( addr:    Address;
           account: int;
)
Address  (  name:    String;
            zip:     int;
)
Article  (  name:    String;
            price:   int;
)
```

- eine Struktur (namens Address)
- ein Feld (Namens name)
- eine Struktur (namens Article)
- ein Feld (namens name)
- ...

Schlüssel und Eigenschaften

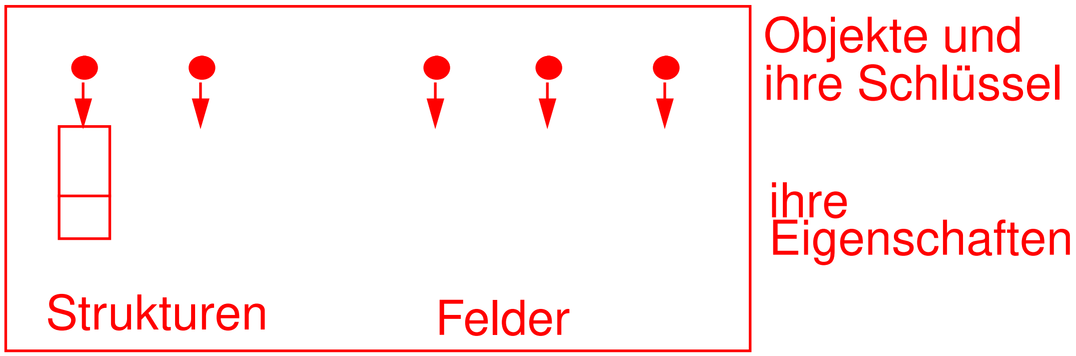
attributierter Strukturbaum

Umgebungen

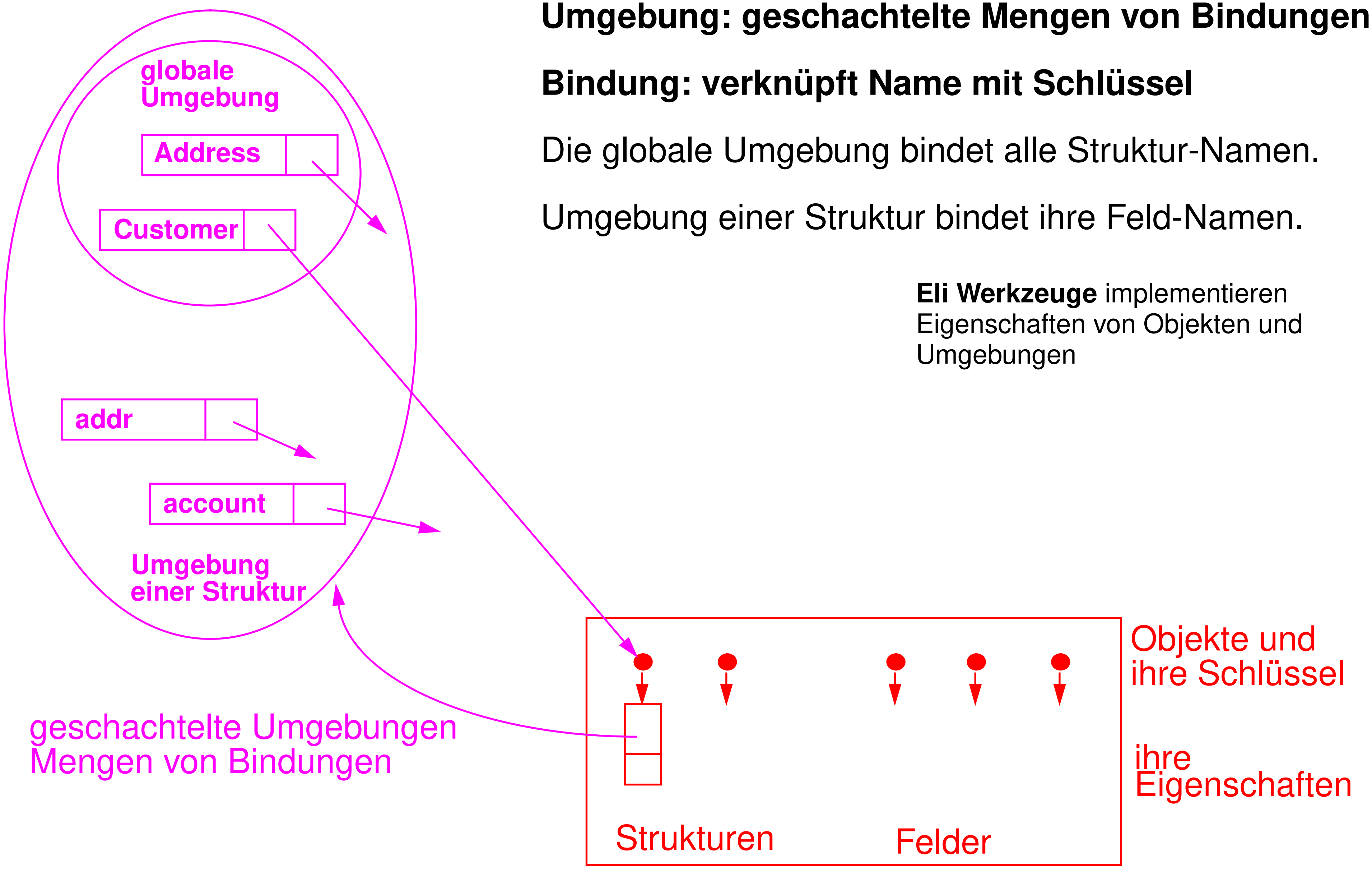
Eli Werkzeuge implementieren Eigenschaften von Objekten und Umgebungen

Olbjekte werden durch **Schlüssel** repräsentiert.
Eigenschaften werden ihnen zugeordnet.

Strukturen haben eine Eigenschaft „Umgebung“



Bindungen und Umgebungen



Umgebung: geschachtelte Mengen von Bindungen

Bindung: verknüpft Name mit Schlüssel

Die globale Umgebung bindet alle Struktur-Namen.

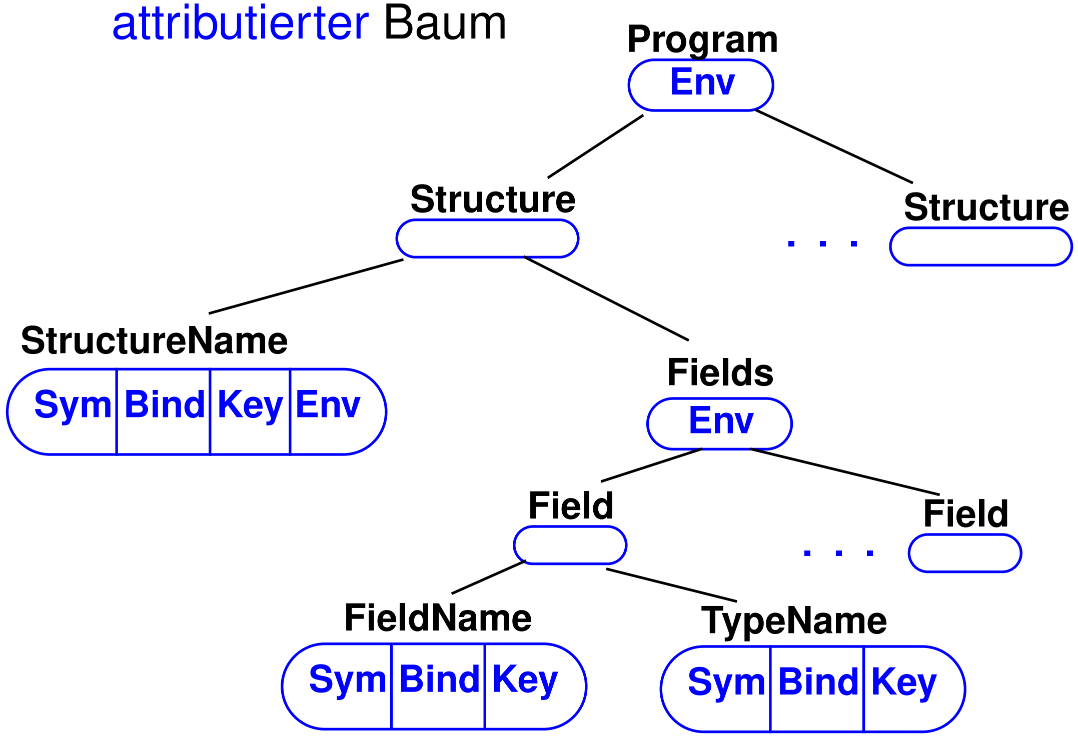
Umgebung einer Struktur bindet ihre Feld-Namen.

Eli Werkzeuge implementieren Eigenschaften von Objekten und Umgebungen

Attributierter Baum für Namensanalyse

Attribute der Baumknoten beschreiben Eigenschaften des repräsentierten Konstruktes

attributierter Baum



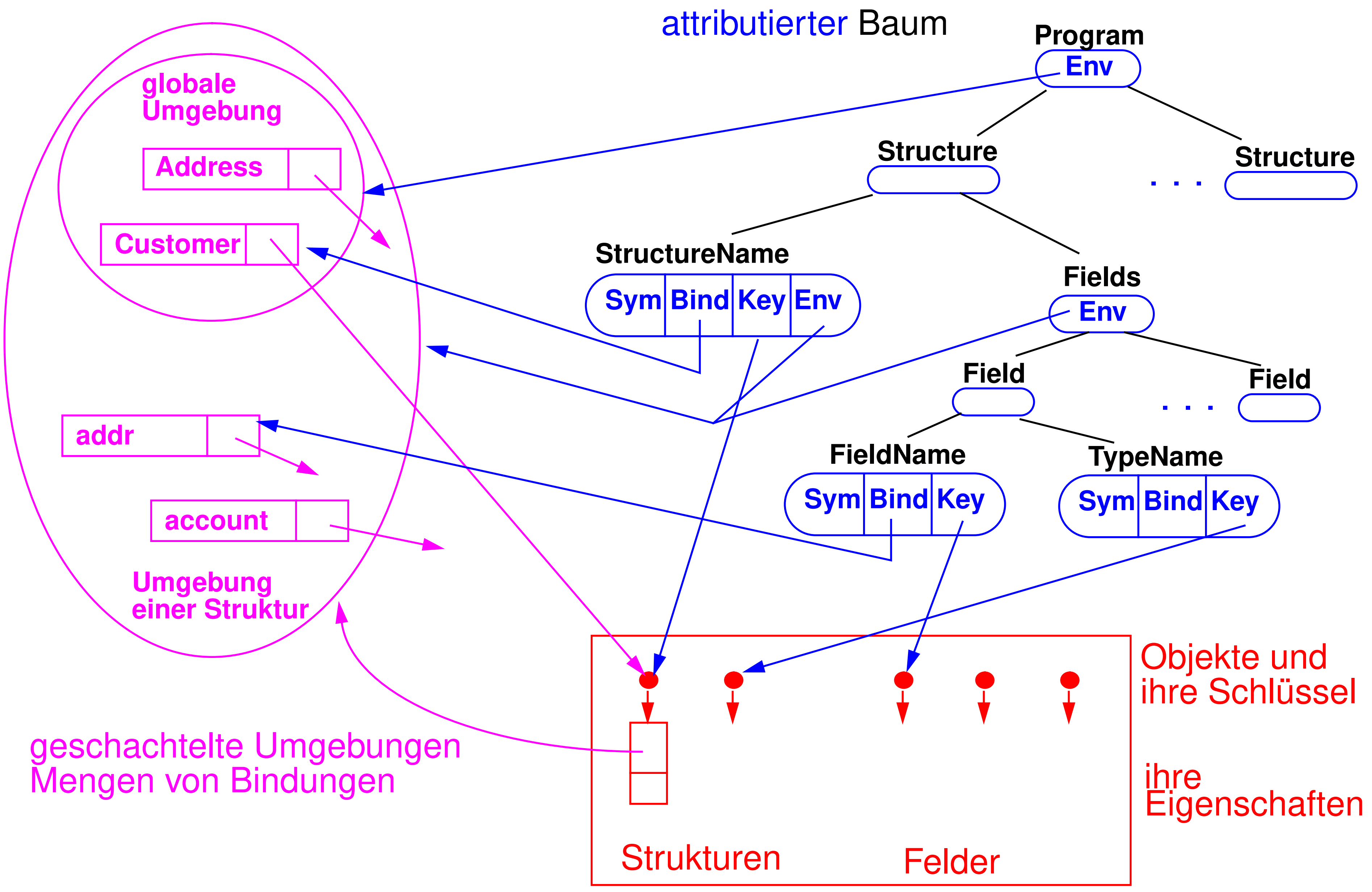
Program hat die globale Umgebung

StructureName und Fields haben die Umgebung der Struktur

Jeder Namensknoten hat Attribute für

- die Codierung des Symbols,
- die Bindung seines Namens, und
- sein Schlüssel

Attribute, Umgebungen und Schlüssel



Umgebungsmodul

Implementiert den abstrakten Datentyp **Environment** (Umgebung):
hierarchisch geschachtelte Mengen von **Bindungen (Name, Umgebung, Schlüssel)**

Funktionen:

NewEnv ()	bildet eine neue Umgebung e, als Wurzel einer Hierarchie; eingesetzt im **Wurzelkontext**
NewScope (e_1)	bildet eine neue Umgebung e_2 das in e_1 geschachtelt ist. Jede Bindung von e_1 ist auch eine Bindung von e_2, falls sie dort nicht verdeckt ist; eingesetzt im **Range-Kontext**
BindIdn (e, id)	bildet eine neue Bindung (id, e, k) falls e noch keine Bindung für id hat; dann ist k ein neuer Schlüssel für ein neues Objekt; das Ergebnis ist immer das Bindungstripel (id, e, k); eingesetzt für **definierende Auftreten**.
BindingInEnv (e, id)	liefert ein Bindungstripel (id, e_1, k) aus e oder einer umfassenden Umgebung von e; liefert NoBinding, falls es keine solche Bindung gibt; eingesetzt für **angewandte Auftreten**
BindingInScope (e, id)	liefert ein Bindungstripel (id, e, k) aus e, falls e es direkt enthält, sonst NoBinding; eingesetzt z. B. für **qualifizierte Namen**

PDL: Generator für Definitionsmodule

zentrale Datenstruktur ordnet **Objekten Eigenschaften** zu,
z. B. T*yp einer Variablen, Elementtyp eines Array-Typs.*

Objekte werden durch **Schlüssel** *(key)* identifiziert.

Operationen:

`NewKey ( )`	liefert neuen Schlüssel
`ResetP (k, v)`	setzt zum Schlüssel `k` die Eigenschaft `P` mit Wert `v`
`SetP (k, v, d)`	setzt (ersetzt) zum Schlüssel `k` die Eigenschaft `P` mit Wert `v` (bzw. `d`)
`GetP (k, d)`	liefert Wert der Eigenschaft `P` des Schlüssels `k`; liefert `d`, falls `P` zu `k` nicht gesetzt ist

Aufrufe: in abhängigen Operationen im Baum

Implementierung: z. B. Liste von Eigenschaften zu jedem Schlüssel

aus **Spezifikationen**

Eigenschaftsname: Eigenschaftstyp;

werden Funktionen **`ResetP, SetP, GetP`** für die Eigenschaft **`P`** generiert.

Beispiel: Namen und Eigenschaften für Struktur-Generator

Abstrakte Syntax

```
RULE: Descriptions  LISTOF Import | Structure             END;
RULE: Import ::= 'import' ImportNames 'from' FileName     END;
RULE: ImportNames   LISTOF ImportName                     END;
RULE: Structure ::= StructureName '(' Fields ')'          END;
RULE: Fields        LISTOF Field                          END;
RULE: Field ::=     FieldName ':' TypeName ';'            END;
RULE: StructureName ::= Ident                             END;
RULE: ImportName ::=    Ident                             END;
RULE: FieldName ::=     Ident                             END;
RULE: TypeName ::=      Ident                             END;
```

Verschiedene Nichtterminale für Bezeichner in unterschiedlichen Rollen, wenn unterschiedliche Berechnungen erwartet werden, z. B. für definierende und angewandte Auftreten.

Umgebungsattribute berechnen

2 Symbole spielen die **Rolle eine Range**: Unterbaum begrenzt Gültigkeit darin enthaltener Definitionen

```
CLASS SYMBOL Range: Env: Environment;
SYMBOL Descriptions INHERITS Range END;
SYMBOL Fields  INHERITS Range END;
```

Wurzel der Umgebungshierarchie

```
SYMBOL Descriptions COMPUTE
   SYNT.Env = NewEnv ();
END;
```

Jedes Structure-Objekt hat eine **Umgebung als Eigenschaft**.

Darin werden die Feldnamen gebunden.

Sie wird in die globale Umgebung eingebettet

```
RULE: Structure ::= StructureName '(' Fields ')'
COMPUTE
   Fields.Env = StructureName.Env;
END;

SYMBOL StructureName COMPUTE
   SYNT.GotEnvir =
      IF (EQ (GetEnvir (THIS.Key, NoEnv), NoEnv),
         ResetEnvir
            (THIS.Key,
             NewScope (INCLUDING Range.Env)));

   SYNT.Env =
      GetEnvir (THIS.Key, NoEnv) <- SYNT.GotEnvir;
END;
```

Definierende Auftreten von Bezeichnern

Alle `IdentOcc`-Knoten haben diese 3 Attribute

Diese Brechnungen sind gleich für alle `IdentOcc`-Knoten

```
CLASS SYMBOL IdentOcc:
   Sym: int,
   Bind: Binding,
   Key: DefTableKey;

CLASS SYMBOL IdentOcc COMPUTE
   SYNT.Sym = TERM;
   SYNT.Key =  KeyOf (SYNT.Bind);
END;
```

Alle definierenden Auftreten **binden** ihren Namen im Env-Attribut des **nächst-umfassenden Range**

```
CLASS SYMBOL DefineAnIdent INHERITS IdentOcc
COMPUTE
   SYNT.Bind =
      BindIdn (INCLUDING Range.Env, THIS.Sym);
END;

SYMBOL StructureName INHERITS DefineAnIdent END;
SYMBOL ImportName    INHERITS DefineAnIdent END;
SYMBOL FieldName     INHERITS DefineAnIdent END;
```

Angewandte Auftreten von Bezeichnern

Vorbedingung für das Suchen einer Bindung:

Alle Bindungen in den umfassenden Ranges sind hergestellt.

```
SYMBOL Descriptions COMPUTE
   SYNT.GotBindings =
      CONSTITUENTS DefineAnIdent.Bind SHIELD Range;
END;

SYMBOL Fields COMPUTE
   SYNT.GotBindings =
      CONSTITUENTS DefineAnIdent.Bind
      <- INCLUDING Range.GotBindings;
END;
```

Für den Bezeichner eine Bindung in den umfassenden Umgebungen suchen

Misserfolg melden

Nur ein angewandtes Auftreten

```
CLASS SYMBOL UseAnIdent INHERITS IdentOcc COMPUTE
   SYNT.Bind =
      BindingInEnv (INCLUDING Range.Env, THIS.Sym)
         <- INCLUDING Range.GotBindings;

   IF (EQ (SYNT.Bind, NoBinding),
      message (ERROR,
         CatStrInd ("Name is not defined: ",
                    THIS.Sym),
          0, COORDREF));
END;

SYMBOL TypeName   INHERITS UseAnIdent END;
```

Eigenschaften definieren, setzen und prüfen

Beispiel: Ein Feldname darf nicht an der Stelle eines Typnamens auftreten:

```
Customer (addr: Address; account: addr;)
```

Boolsche Eigenschaft isField definiert für das Werkzeug PDL:

```
isField: int;
```

Eigenschaft auf wahr setzen für FieldName:

```
SYMBOL FieldName COMPUTE
   SYNT.GotisField =
      ResetisField (THIS.Key, 1);
END;
```

Eigenschaft prüfen im TypeName Kontext:

```
SYMBOL TypeName COMPUTE
  IF (GetisField (THIS.Key, 0),
    message (ERROR,
      CatStrInd
        ("Field name not allowed",
         THIS.Sym),0, COORDREF))
   <- INCLUDING Program.GotisField;
END;
```

Vorbedingung garantiert, dass alle ResetisField aufgerufen sind, bevor ein GetisField aufgerufen wird:

```
SYMBOL Program COMPUTE
  SYNT.GotisField =
    CONSTITUENTS FieldName.GotisField;
END;
```

Eigenschaften von Objekten
- beschreiben die **Bedeutung**
- gegen **Einschränkungen** prüfen
- zur **Transformation** verwenden

Richtlinien zu: Eigenschaften von Objekten (B)

Vorbereitung:

- Objekte werden meist durch Bezeichner in der Beschreibung benannt; durch **`DefTableKeys`** repräsentiert
- Modul der **Namensanalyse** bindet Auftreten von Bezeichnern.
- Symbolknoten der Bezeichner haben ein **`Key`**-Attribut; es identifiziert das Objekt

Entwurfsschritte für Berechnungen von Eigenschaften:

1. Bestimme **Namen und Typ der Eigenschaft** und spezifiziere sie für PDL.
2. Bestimme die **Kontexte und die Operation für das Setzen** der Eigenschaft.
3. Bestimme die **Kontexte für das Lesen** der Eigenschaft.
4. Bestimme die **Abhängigkeiten zwischen (2) und (3)**.
   In einfachen Fällen ist dies: "alle Setzen vor jedem Lesen".
5. Spezifiziere (2), (3) und das Abhängigkeitsmuster (4).

Eigenschaften des Objektes **setzen oder lesen**, möglichst im
Kontext des Symbols für den Bezeichner platzieren
nicht die die **`Key`**-Werte im Baum weiterreichen.

Wenn möglich, **SYMBOL-Berechnungen** verwenden (siehe A).

Themen für Projekte entwickeln

- spezielles, enges Anwendungsgebiet
  (aus Programmentwicklung, Web-Entwurf, Informatik-fremdes Gebiet, Hobby)
- gleichartige, strukturierte Texte in verschiedenen Varianten benötigt
- systematisch erzeugbar aus einfachen Beschreibungen

Beispiele:

- Datenstruktur-Generator
- Roboter-Steuerung
- Molekül-Darstellung
- Spielplan-Gestaltung

Vorgehen:

1. Thema formulieren
2. Beispiele für Ausgabe entwickeln
3. Beispiele für Beschreibungen (Eingabe) entwickeln
4. Regeln für Erzeugung von Ausgabe aus Beschreibungen angeben

Prüfen: **Thema geeignet?**

Technik: Do it once

Aufgabe:

- Viele Auftreten eines Namens an dasselbe Objekt gebunden
- An genau einem Auftreten eine Berechnung ausgeführen (z. B. Zieltext für das Objekt)

Lösung:

Boolsches Attribut berechnen:
An genau einem Auftreten wahr
sonst falsch.
Eigenschaft benutzen; Entwurfsschritte:

1. Eigenschaft definieren: `Done: int;`
2. Setzen im Namenskontext, wenn noch nicht gesetzt
3. Lesen im Namenskontext
4. keine Abhängigkeiten
5. siehe rechts

```
CLASS SYMBOL DoItOnce:
                    DoIt: int;

CLASS SYMBOL DoItOnce
      INHERITS IdentOcc COMPUTE
   SYNT.DoIt =
      IF (GetDone (THIS.Key, 0),
         0,
         ORDER
          (ResetDone (THIS.Key, 1),
          1));
END;
```

Anwendung:

```
SYMBOL StructName INHERITS DoITOnce
COMPUTE
   SYNT.Text =
      IF (THIS.DoIt,
         PTGTransform (...),
         PTGNULL);
END;
```

6. Bibliothek mit Spezifikationsmodulen

Spezifikationsmodul

- löst eine häufig vorkommende Aufgabe,
  z. B. Bindung von Namen gemäß Algol-Verdeckungsregel
- bietet abstrakte Symbole (**`CLASS`**) mit vorgefertigten Berechnungen,
  Symbolrollen mit vorgefertigten Berechnungen, z. B. **`IdUseEnv`**
- enthält (als FunnelWeb-Datei) alle sonst notwendigen Spezifikationen,
  z. B. Implementierung von C-Funktionen
- ist in thematisch zusammenhängenden Bibliotheken organisiert,
  z. B. Namensanlalyse, Ausgabe mit PTG
- ist ausführlich dokumentiert

Anwender

- wählt Modul aus,
- instanziiert ihn,
- ordnet Symbolrollen einigen Symbolen seiner abstrakten Syntax zu,
- benutzt die berechneten Attribute in eigenen Berechnungen

Grundlegender Modul zur Namensanalyse

Symbolrollen:
Wurzel der Grammatik:

```
SYMBOL Program INHERITS RootScope END;
```

Abschnitte mit Definitionen:

```
SYMBOL Block INHERITS RangeScope END;
```

Definierter Bezeichner:

```
SYMBOL DefIdent INHERITS IdDefScope END;
```

Angewandte Bezeichner:

```
SYMBOL UseIdent
   INHERITS IdUseEnv,ChkIdUse END;
```

Benutzbare Attribute:

```
   DefIdent, UseIdent: Key, Bind
   Program, Block: Env
```

Instantiierung in .specs-Datei für Algol-Scope-Rules:

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

für C-Scope-Rules:

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

für neuen Namensraum

```
$/Name/AlgScope.gnrc
     +instance=Label
     :inst
```

Symbolrollen:

`LabelRootScope`,
`LabelRangeScope`, ...

Struktur-Generator mit Namensmodul spezifizieren

`Fields` spielt die **Rolle eine Range**: Unterbaum begrenzt Gültigkeit darin enthaltener Definitionen

```
SYMBOL Fields  INHERITS RangeScope END;
```

Wurzel der Umgebungshierarchie

```
SYMBOL Descriptions INHERITS RootScope END;
```

Alle definierenden Auftreten **binden** ihren Namen im Env-Attribut des **nächst-umfassenden Range**

```
CLASS SYMBOL DefineAnIdent
   INHERITS IdentOcc, IdDefScope END;
```

Für den Bezeichner eine Bindung in den umfassenden Umgebungen suchen

```
CLASS SYMBOL UseAnIdent
   INHERITS IdentOcc, IdUseEnv, ChkIdUse
END;
```

Misserfolg melden

Nur ein angewandtes Auftreten

vergl. Folien 5.9 -5.11

Spezifikations Bibliotheken in Eli

Inhalt der Eli Dokumentation
Specification Module Library:

- Introduction of a running example
- How to use Specification Modules

- Name analysis according to scope rules
- Association of properties to definitions
- Type analysis tasks
- Tasks related to input processing
- Tasks related to generating output
- Abstract data types to be used in specifications
- Solutions of common problems

- Migration of Old Library Module Usage

Namen und Typen

Name analysis according to scope rules

- Tree Grammar Preconditions
- Basic Scope Rules, 3 variants:
  Algol-like, C-like, Bottom-Up
- Predefined Identifiers
- Joined Ranges (3 variants)
- Scopes Being Properties of Objects
  (4 variants)
- Inheritance of Scopes (3 variants)
- Name Analysis Test
- Environment Module

Type analysis tasks

- Basic Type Analysis
  Example for Basic Type Analysis
- Properties of Types
  Array Types
  Pointer Types
  Function Types
  Types Having Scope Properties
- Operator Identification
- Deferred Property Association

Zuordnen von Eigenschaften

Association of properties to definitions

- Common Aspects of Property Modules
- Count Occurrences of Objects
- Set a Property at the First Object Occurrence
- Check for Unique Object Occurrences
- Determine First Object Occurrence
- Map Objects to Integers
- Associate Kinds to Objects
- Associate Sets of Kinds to Objects
- Reflexive Relations Between Objects
- Some Useful PDL Specifications

Eingabe und Ausgabe

Tasks related to input processing

- Insert a File into the Input Stream
- Accessing the Current Token
- Command Line Arguments for Included Files

Tasks related to generating output

- PTG Output for Leaf Nodes
- Commonly used Output patterns for PTG
- Indentation
- Output String Conversion
- Pretty Printing
- Typesetting for Block Structured Output
- Processing Ptg-Output into String Buffers
- Introduce Separators in PTG Output

Allgemein nützliche Module

Abstract data types to be used in specifications

- Lists in LIDO Specifications
- Linear Lists of Any Type
- Bit Sets of Arbitrary Length
- Bit Sets of Integer Size
- Stacks of Any Type
- Mapping Integral Values To Other Types
- Dynamic Storage Allocation

Solutions of common problems

- Error Reports
- String Concatenation
- Counting Symbol Occurrences
- Generating Optional Identifiers
- Sorting Elements of an Array

Projektbeschreibung (Beispiel)
1. Aufgabenbeschreibung des Structure-Generator

- Verbunde mit typisierten Komponenten als **C++-Klassen** implementieren
- Eingabe beschreibt mehrere **Verbunde mit ihren Komponenten**
- Ausgabe ist eine Datei mit einer C++-Klassendeklaration für jeden Verbund
- Datenattribute mit **Schreib- und Lese-Methoden** und initialisierender **Konstruktor** werden generiert

- Einsatz des Generators zur Unterstützung der **Software-Entwicklung**
- Generierte Klassen sollen **manuell weiterentwickelt** werden können (Pretty-Printing)
- **Generator soll erweiterbar** sein, z. B. Lesen und Schreiben von Objekten
- Beschreibungssprache soll erlauben, dass Verbunde aus mehreren Dateien **akkumuliert** werden, Duplikate sind erlaubt

2. Ausgabebeispiele

Import extern definierter Verbunde:

```
#include "util.h"
```

Vorwärtsreferenzen:

```
typedef class Customer_Cl *Customer;
typedef class Address_Cl *Address;
```

Klassendeklaration:

Verbundkomponenten:

initialisierender Konstruktor:

set- und get-Methoden für Komponenten:

```
class Customer_Cl {
private:
   Address addr_fld;
   int account_fld;
public:
   Customer_Cl (Address addr, int account)
      {addr_fld=addr; account_fld=account; }
   void set_addr (Address addr)
      {addr_fld=addr;}
   Address get_addr ()
      {return addr_fld;}
   void set_account (int account)
      {account_fld=account;}
   int get_account ()
      {return account_fld;}
};
```

weitere Klassendeklarationen

```
class Address_Cl {
...
```

3. Eingabebeispiele

geschlossene Form:

Folge von Verbundbeschreibungen
bestehend aus
Folge von Komponentenbeschreibungen

```
Customer(  addr:    Address;
           account: int;
        )
Address (  name:  String;
           zip:   int;
           city:  String;
        )
import String from "util.h"
```

Wiederholungen möglich

```
Address (  zip:   int;
           phone: int;
        )
```

offene Form:

Folge von Komponentenbeschreibungen

```
Customer.addr:  Address;
Address.name: String;
Address.zip: int;
import String from "util.h"
```

Wiederholungen möglich

```
Customer.account: int;
Address.zip: int;
```

4. Analyse und Transformation

Syntax:

Je nach Einsatzgebiet soll eine geeignete Notation in geschlossener oder offener Form gewählt werden.

Namen:

Strukturnamen und Komponentennamen dürfen mehrfach definierend auftreten.

Typnamen müssen definierte Strukturen, vordefiniert oder importiert sein.

Eigenschaften:

Die Typen mehrfach definierter Komponenten müssen gleich sein.

Die Komponenten mehrfach definierter Strukturen werden akkumuliert.

Transformation:

Zu mehrfachen Auftreten derselben Struktur oder Komponente wird nur jeweils eine Deklaration erzeugt.

Die Ausgabe soll mit Pretty-Printing Technik formatiert werden.

Arbeitsschritte zur Entwicklung eines Generators

1. Aufgaben definieren
   a. Beschreibung
   b. Ausgabebeispiele
   c. Eingabebeispiele
   d. Aufgaben der Analyse und Transformation

2. Strukturierungsphase
   a. konkrete Syntax
   b. Notation der Grundsymbole
   c. abstrakte Syntax
   d. umfassende Tests

3. Semantische Analyse
   a. fehlerhafte Eingabe durch Testfälle charakterisieren
   b. Namen binden
   c. Eigenschaften berechnen
   d. umfassende Tests

4. Transformation
   a. Ausgabemuster entwerfen
   b. Zieltext erzeugen
   c. umfassende Tests

5. Präsentation des Generators

Projektthemen 2003

	Thema	**Bearbeiter**
A	**Automaten**	Hagenkötter, Schwindt
B	**MusiXTeX**	Seifert, Schröder, Walther
C	**CoverGen**	Birkenheuer, Keller
D	**StreckGen**	Soltenborn, Thygs
E	**KlausurErgebnisse**	Böning, Hollmann
F	**Graphen in LaTeX**	Lindner, Viergutz
G	**Planeten**	Gebel
H	**PersWeb**	Junklewitz, Rödig