Praktikum Sprachimplementierung mit Werkzeugen

Uwe Kastens

WS 1999/2000

Ziele:

Die Teilnehmer sollen lernen

- **anwendungsspezifische Sprachen** zu entwerfen**,**
- **Implementierungsaufgaben** dafür zu erkennen, einzuordnen und zu beschreiben,
- **Spezifikationstechniken** für die zentralen Werkzeuge von Eli anzuwenden,
- das **Eli-System wirksam einzusetzen** und zu bedienen.

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

Beispiele für Projektthemen

- Generator für Datenstrukturen
- Literatur-Report-Generator
- Steuersprache für Meßrechner
- Makrosprache zum Test von Instrumenten
- Programmstrukturen nach HTML übersetzt
- Spracherweiterungen (z. B. Occam)
- Notenschrift-Generator
- Interpretierer für Entscheidungstabellen
- Tischrechner
- Lindenmeyer-Grammatiken zur Bilderzeugung

Sprachen in der Software-Entwicklung

Spezifikations- und Modellierungssprachen:

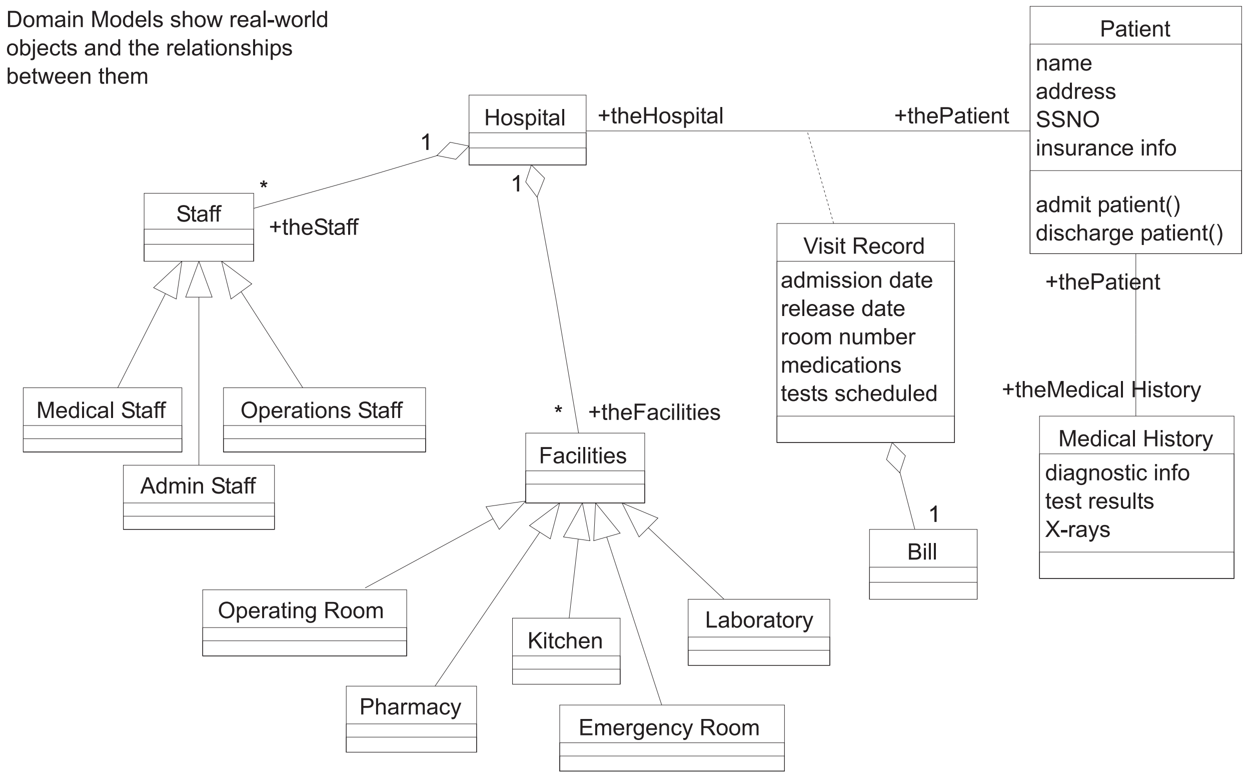
SDL Specification and Description Language (CCITT):

```
block Dialogue;
  signal
    Money, Release, Change, Accept, Avail, Unavail, Price,
    Showtxt, Choice, Done, Flushed, Close, Filled;
  process Coins referenced;
  process Control referenced;
  process Viewpoint referenced;
  signalroute Plop
    from env to Coins
      with Coin_10, Coin_50, Coin_100, Coin_x;
  signalroute Pong
    from Coins to env
      with Coin_10, Coin_50, Coin_100, Coin_x;
  signalroute Cash
    from Coins to Control
      with Money, Avail, Unavail, Flushed, Filled;
    from Control to Coins
      with Accept, Release, Change, Close;

  ...

  connect Pay and Plop;
  connect Flush and Pong;
endblock Dialogue;
```

UML Unified Modeling Language:



Sprachen in der Software-Entwicklung

Domain-Specific Language (DSL):
Sprache zugeschnitten auf ein spezielles Anwendungsgebiet

Application Generators:
Implementierung einer DSL durch einen Programmgenerator

Beispiele:

- **Simulation mechatronischer Regelsysteme**
- **Robotersteuerung**
- **Meßdatenerfassung**
- **Steuerung von Prüfständen für Kfz-Instrumente**
- **Literatur-Report-Generator:**

```
string name =   InString "Which author?";
int since =     InInt "Since which year?";
int cnt = 0;

"\nPapers of ", name, " since ", since, ":\n";
[ SELECT name <= Author && since <= Year;
  cnt = cnt + 1;
  Year, "\t", Title, "\n";
]
"\n", name, " published ", cnt, "papers.\n";
```

aus U. Kastens: Construction of Application Generators Using Eli, Workshop on Compiler Techniques for Application Domain Languages ..., Linköping, April 1996

zum Thema:

C.W. Krueger: Software Reuse, ACM Computing Surveys 24, June 1992

Conference on DSL (USENIX), Santa Babara, Oct. 1997

ACM SIGPLAN Workshop on DSL (POPL), Paris, Jan 1997

Sprachen in der Software-Entwicklung

Programmiersprachen als Quellsprachen:

- **Programmanalyse**
  Aufrufgraphen, Ablaufgraphen, Variablenabhängigkeiten
  z. B. für Year-2000-Problem
- **Strukturerkennung**
  z. B. für Reengineering

Programmiersprachen als Zielsprachen generiert aus

- **Spezifikationen (SDL, OMT, UML)**
- **graphischen Strukturmodellen**
- **DSL, Application Generator**

=> Übersetzeraufgabe: Source-to-Source-Übersetzung

Praktikumsplan

Termin	Vorlesung	Tutorium	Vor- u. Nacharbeit
1	Eli: Aufgabenzerlegung		Novice-Guide
2		Novice-Guide durcharbeiten	
3	konkrete u. abstrakte Syntax		
	Berechnungen im Baum		Computations in Trees
4		Programmanalyse I	
5	Modulbenutzung, Namensanalyse,		
	Eigenschaften von Objekten		
6		Programmanalyse II	
			Core-Spezifikation
7	Minisprache Core vorstellen		
		Minisprache Core erweitern	
8	Zieltext erzeugen		
		Core nach C übersetzen	
9	Projekt: Struktur-Generator vorstellen		
		Projekt: Struktur-Generator bearbeiten	
10 - 14	Eigene Projekte definieren		
		und bearbeiten	
15	Vorstellung der Projektergebnisse		

Compilation Tasks for Source-to-Source Translation

text in a domain specific language

Structuring	**Lexical analysis**	**Scanning** **Conversion**
	Syntactic analysis	**Parsing** **Tree construction**
Translation	**Semantic analysis**	**Name analysis** **Type analysis**
	Transformation	**Data mapping** **Action mapping**
Encoding	Code generation	Execution-order Register allocation Instruction selection
	Assembly	Internal addressing External addressing Instruction encoding

Waite, Carter: An Introduction to Compiler Construction, Harper Collins, 1992

application program (HLL)

Eli provides solution methods

generating tools

reusable precoined solutions for common tasks

extensible by domain specific routines

Lexical analysis	input processing token sets scanner generator encoding & conversion
Syntactic analysis	parser generator tree construction generator
Semantic analysis attribute evaluator generator	name analysis modules type analysis modules definition table generator
Transformation	pattern based text generator

Integrated tool set Eli

Eli knows how to

- drive the generators
- obtain the precoined solutions
- compose the translator
- relate the user specifications

User specifications for source-to-source processors:

	token descriptions select non-literal tokens
	context-free grammar
associate roles to tree grammar compute in tree contexts	use name analysis module use type analysis module specify object properties
compose target structure	specify target text patterns

Examples for Specification Fragments

- context-free grammar

```
OutputItems:     OutputItem // ','.
```

- select non-literal token specification

```
Ident:           C_IDENTIFIER
```

- select name analysis module

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

- associate roles to tree grammar symbols

```
SYMBOL DefIdent INHERITS IdDefScope END;
SYMBOL UseIdent INHERITS IdUseEnv END;
```

- specify object properties

```
Kind: DefTableKey;
```

- compute in tree context

```
RULE: Statement ::= UseIdent '=' Expr ';'
COMPUTE
IF (NE (GetKind (UseIdent.Key, VariableKind),
        VariableKind),
  message (ERROR, "variable required",
           0, COORDREF));
END;
```

- specify target text patterns

```
InpCall: "(" $1 string "(" $2 string "))"
```

- compose target structure

```
RULE: Expr ::= 'InString' StringLit COMPUTE
  Expr.CCode =
    PTGInpCall ("InString",
                StringTable(StringLit));
END;
```

FunnelWeb: Spezifikationsfragmente zusammenfassen und dokumentieren

@=~

Umschaltzeichen von @ auf Tilde umgestellt

~p maximum_input_line_length = infinity

beliebig lange Zeilen

Die Ausgabedatei ArrayType.con erzeugen:

~O~<ArrayType.con~>~{

Type: 'array' Index 'of' Type.

~}

Noch eine Ausgabedatei:

~O~<ArrayAccess.con~>~{

Variable: Variable '[' Expr ']'.

~}

Eli fügt alle .con Dateien zu einer zusammen.

Dies kann man auch mit fw-Makros tun:

Produktionen für Array-Typen:

~$~<Array type~>==~{

Type: 'array' Index 'of' Type.

~}

Produktionen für Array-Zugriffe:

~$~<Array access~>==~{

Variable: Variable '[' Expr ']'.

~}

Die zusammengesetzte Ausgabedatei:

~O~<Array.con~>~{

~<Array type~>

~<Array access~>

~}

Der Eli-Ausdruck **X.fw : fwGen** liefert eine Directory mit den Ausgabedateien.

Konkrete Syntax

Eine **kontext-freie Grammatik** definiert die **Struktur der Programme** der Sprache. Hieraus wird der Parser generiert. Sie muß deshalb eindeutig sein (und die LALR(1)-Bedingung erfüllen).

Notation an Beispielen:

```
Prog     ::= Block .
Block    ::= '{' Decl* Stmt+ '}' .
Stmt     ::= Block / Assign ';' / Expr ';' .
Decl     ::= 'int' VarDecls ';' .
VarDecls::= VarDecl // ',' .
VarDecl ::= Ident .
```

Benannte Nichtterminale:	`Prog Block Decl`
Benannte Terminale:	`Ident`
Literale Terminale:	`'{'  '}'  ';'`

EBNF-Konstrukte:

Folge:	`Decl*`
nicht-leere Folge:	`Stmt+`
nicht-leere Folge mit Trenner:	`VarDecl // ','`
Alternativen:	`Block / Assign ';'`
optional:	`[ Parameter ]`
Klammerung:	`( ':=' / ':-' )`

Hinweise zum Entwurf:

- Syntax einer vorgegebenen Sprache nicht unnötig ändern.
- Syntaktische Struktur soll der semantischen entsprechen.
- EBNF nur, wo keine Berechnungen im Baum erwartet werden.
- Namensanalyse erfordert syntaktische Unterscheidung von Bezeichnern: definierendes und angewandtes Auftreten.

Abstrakte Syntax

Eine **kontext-freie Grammatik** definiert **Strukturbäume** als Programmrepräsentation (**Baum-Grammatik**). Der Strukturbaum wird vom Parser aufgebaut.

Analyse und Transformation werden als **Berechnungen in** bestimmten **Strukturbaum-Kontexten** spezifiziert.

Konkrete Produktion:

```
Decl    ::= 'int' VarDecls ';' .
```

abstrakte Produktion, Baum-Kontext:

```
RULE: Decl ::= 'int' VarDecls ';' COMPUTE
        Berechnungen für diesen Kontext
END;
```

Die Knoten des Strukturbaumes repräsentieren Nichtterminale der Baum-Grammatik. Terminale stehen nicht im Baum. Literale Terminale dienen nur zur Identifikation der Produktion.

EBNF-Konstrukte gibt es in der Baum-Grammatik nur sehr eingeschränkt (`LISTOF`).

Es brauchen nur die Baum-Kontexte angegeben zu werden, zu denen es tatsächlich Berechnungen gibt. Eli-Werkzeuge vervollständigen die Baum-Grammatik automatisch.

Ebenso ergänzen sie Produktionen zur konkreten Syntax, die nur in der Baum-Grammatik stehen.

Der Aufbau des Strukturbaumes durch den Parser wird von Eli automatisch generiert.

Überflüssige Kettenproduktionen

Konkrete Produktionen mit gleicher Struktur und Bedeutung zu einem Baum-Kontext zusammenfassen.
Kettenproduktionen zwischen äquivalenten Symbolen entfallen.

Konkrete Ausdrucksgrammatik:

```
Expr    ::= Expr AddOpr Fact / Fact .
Fact    ::= Fact MulOpr Operand / Operand .
Operand ::= '(' Expr ')' .
Operand ::= Number .
AddOpr  ::= '+' / '-' .
MulOpr  ::= '*' / '/' .
```

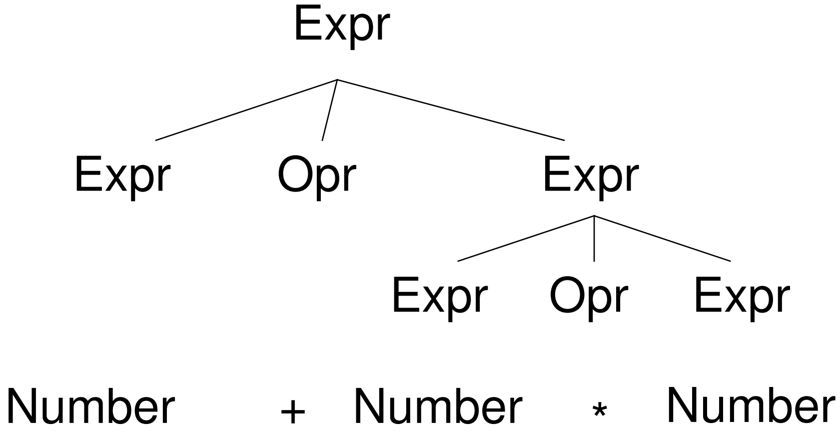
Äquivalente Symbole:

```
Expr ::= Fact Operand .
Opr  ::= AddOpr MulOpr .
```

Baum-Grammatik dazu:

```
RULE: Expr ::= Expr Opr Expr  END;
RULE: Expr ::= Number         END;
RULE: Opr  ::= '+'            END;
RULE: Opr  ::= '*'            END;
...
```

Strukturbaum dazu:



Ein Symbolknoten gehört zu zwei benachbarten Kontexten: oberer und unterer Kontext.

Abhängige Berechnungen

Der Wert eines Ausdrucks wird berechnet und ausgegeben:

```
ATTR value: int;

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

TERM Number: int;

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;

SYMBOL Opr: left, right: int;

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

RULE: Opr ::=  '*'  COMPUTE
  Opr.value = MUL (Opr.left, Opr.right);
END;
```

Vor- und Nachbedingungen von Berechnungen

Ausgabe von Ausdrücken in Postfix-Form:

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

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

Die Attribute `print` und `printed` haben keinen Wert `(Typ VOID)`.

Sie beschreiben Zustände als Vor- und Nachbedingungen für Berechnungen:

`Expr.print:` Postfix-Ausgabe bis vor diesem Knoten ist erledigt.

`Expr.printed:` Postfix-Ausgabe bis einschließlich dieses Knotens ist erledigt.

Abhängigkeitsmuster: CHAIN

CHAIN spezifiziert **left-right-depth-first** Abhängigkeit.
Berechnungen werden in die CHAIN „eingehängt“.

Ausgabe von Ausdrücken in Postfix-Form:

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

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

Wertberechnung entlang einer CHAIN, z. B. Variablenadressen:

```
CHAIN RelAdr: int;

RULE: Block ::= '{' Sequence '}' COMPUTE
  CHAINSTART HEAD.RelAdr = 0;
END;

RULE: Definition ::= 'define' Ident COMPUTE
  Definition.RelAdr =
    ADD (Definition.RelAdr, VariableSize);
END;
```

Die Kontexte, in denen die CHAIN nicht benutzt wird, brauchen nicht erwähnt zu werden!

Abhängigkeitsmuster: INCLUDING

Ein Attribut wird tiefer im Unterbaum benutzt.

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

Berechnung der Schachtelungstiefe von Blöcken:

```
ATTR depth: int;

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

Die Kontexte zwischen der Definition und der Benutzung der `INCLUDING`-Attribute brauchen nicht erwähnt zu werden!

Abhängigkeitsmuster: CONSTITUENTS für Zustände

Zusammenfassen von Attributen aus Unterbaum

`CONSTITUENTS Definition.DefDone` bezieht sich auf die `DefDone`-Attribute aller `Definition`-Knoten im Unterbaum.

Ausgabe aller Definitionen vor allen Anwendungen:

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

Die Kontexte zwischen der Definition und der Benutzung der `CONSTITUENTS`-Attribute brauchen nicht erwähnt zu werden!

Abhängigkeitsmuster: CONSTITUENTS für Werte

Zusammenfassung von Werten erfordert

```
CONSTITUENTS ... WITH (Type, BinFct, UnFct, ZeroFct)
```

`Type`	Typ des Ergebnis und der propagierten Werte
`BinFct`	verknüpft zwei Werte; sollte assoziativ sein
`UnFct`	erzeugt einen Wert aus dem Attributwert
`ZeroFct`	erzeugt einen "neutralen" Wert

Ausgabe der Anzahl der Anwendungen:

```
ATTR Count: int;

RULE: Block ::= '{' Sequence '}' COMPUTE
  printf  ("%d uses occurred\n",
           CONSTITUENTS Usage.Count
           WITH (int, ADD, IDENTICAL, ZERO));
END;

RULE: Usage ::= 'use' Ident COMPUTE
  Usage.Count = 1;
END;
```

Die Werte werden gemäß der Baumstruktur zusammengefaßt.

Berechnungen zu Symbolen

Berechnungen können Symbolen zugeordnet werden, wenn sie für jeden RULE-Kontext gleich sind, z. B.

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

Symbolberechnungen können auch `INCLUDING-`, `CONSTITUENTS-`, oder `CHAIN-`Konstrukte enthalten:

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

SYMBOL Usage COMPUTE
  SYNT.Count = 1;
END;
```

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

Berechnungen im `RULE`-Kontext überschreiben Berechnungen im SYMBOL-Kontext für dasselbe Attribut.

Verwendung für Rekursionsanfänge oder für Ausnahmen von der Symbolberechnung:

```
SYMBOL Block COMPUTE
  INH.depth = ADD (INCLUDING Block.depth, 1);
END;

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

Berechnungsmodule mit CLASS-Symbolen

Wiederverwendung von Berechnungsmustern

Berechnungen werden neuen CLASS-Symbolen zugeordnet und durch **INHERITS** an Baum-Symbole gebunden.

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

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

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

```
CHAIN Occurrence: int;
ATTR OccNo, TotalOccs: int;

CLASS SYMBOL OccRoot COMPUTE
  CHAINSTART HEAD.Occurrence = 0;
  THIS.TotalOccs = TAIL.Occurrence;
END;

CLASS SYMBOL OccElem COMPUTE
  SYNT.OccNo = THIS.Occurrence;
  THIS.Occurrence = ADD (SYNT.OccNo, 1);
END;
```

Benutzung:

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

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

Auf diesem Prinzip beruht die Benutzung Bibliotheksmodulen, z. B. zur Namensanalyse.

Bibliotheksmodule allgemein

Bibliotheksmodule machen vorgefertigte Lösungen für häufig auftretende Aufgaben verfügbar.

1. Modul in der Dokumentation identifizieren, Beschreibung lesen

2. Modul instantiieren:

```
$/Prop/Unique.gnrc : inst
```

3. Symbolrollen Grammatiksymbolen zuordnen und Attribute benutzen:

```
SYMBOL Program  INHERITS RangeUnique END;
SYMBOL DefIdent INHERITS Unique COMPUTE
  IF (NOT (THIS.Unique),
  message (ERROR,
           CatStrInd
           ("identifier is multiply defined: ",
             THIS.Sym),
           0, COORDREF));
END;
```

4. Wenn nötig Modul-Implementierung beschaffen und lesen:

```
-> $elipkg/Prop/Unique.gnrc : inst : viewlist
```

ADT Listen

Generischer Modul zur Listenimplementierung

```
$/Adt/List.gnrc+instance=DefTableKey +referto=deftbl :inst
```

Hier: Elementtyp `DefTableKey` definiert in Datei `deftbl.h`

Es werden der Typ `DefTableKeyList` und Funktionen wie

`ConsDefTableKeyList`, `HeadDefTableKeyList`,
`TailDefTableKeyList`, ...und viele andere

als C-Modul verfügbar.

Lido-Unterstützung zur Listenbenutzung:

```
$/Adt/LidoList.gnrc+instance=DefTableKey+referto=deftbl :inst
```

Listen in Baumkontexten aufbauen:

```
RULE: FunctionHead ::= '(' Parameters ')' TypeDenoter
COMPUTE
   ...ResetParamTypes (FunctionHead.Type,
                       Parameters.DefTableKeyList) ...
END;
SYMBOL Parameters INHERITS DefTableKeyListRoot END;
SYMBOL Parameter  INHERITS DefTableKeyListElem END;
RULE: Parameter ::= TypeDenoter DefIdent COMPUTE
  Parameter.DefTableKeyElem = TypeDenoter.Type;
END;
```

Listen in Baumkontexten abbauen entsprechend.

Weitere ADT-Module: BitSet, Stack

Namensanalyse (Bezeichneridentifikation)

Bezeichner benennen (statische) Programmobjekte.

Definition für Bezeichner b führt neues Programmobjekt ein und bindet es an den Bezeichner.

Die Bindung gilt in einem bestimmten Bereich des Programms: **Gültigkeitsbereich** der Definition.

Aufgabe: jedem Auftreten eines Bezeichners den Schlüssel des Programmobjektes zuordnen. *(konsistente Umbenennung)*

Grundlage: Gültigkeitsregeln der Sprache *(scope rules)*

Verdeckungsregeln für Sprachen mit geschachtelten Strukturen:

- **Algol-Regel:** Die Definition eines Bezeichners b gilt im **ganzen** kleinsten sie umfassenden Abschnitt, aber nicht in darin enthaltenen Abschnitten mit einer Definition von b.

- **C-Regel:** Die Definition eines Bezeichners b gilt im kleinsten sie umfassenden Abschnitt **von der Definitionsstelle an**, aber nicht in darin enthaltenen Abschnitten mit einer Definition von b von der Definitionsstelle an**.**

Abschnitt je nach Sprache: Block. Prozedur, Modul, ...

Implementierung: Operationen des **Umgebungsmoduls** in Baumkontexten aufrufen.

Umgebungsmodul

implementiert den abstrakten Datentyp **Environment**:

hierarchisch geschachtelte Mengen von Bindungen
(Bezeichner, Schlüssel)

Funktionen:

NewEnv () erzeugt ein neues Environment e als Wurzel.

NewScope (e1) erzeugt neues Environment e2.
Alle Bindungen aus e1 sind auch in e2, falls sie nicht durch Bindungen in e2 verdeckt werden.

DefineIdn (e, b) fügt Bindung (b, k) in e ein, falls sie dort noch nicht exisitiert;
k ist dann ein neuer Objektschlüssel.
Liefert in jedem Fall den an b gebundenen Schlüssel k.

KeyInEnv (e, b) liefert den Schlüssel k eines Paares (b, k) aus e (einschließlich aller e_i, aus denen e erzeugt wurde).
Liefert NoKey, falls kein solches Paar existiert.

KeyInScope (e, b) liefert den Schlüssel eines Paares (b, k), falls unmittelbar in e enthalten, NoKey sonst.

Umgebungsoperationen im Baum

Operationen in Baumkontexten und Reihenfolge ihrer Ausführung modelliert Gültigkeitsregeln.

Wurzelkontext `Root`:

```
Root.Env = NewEnv ();
```

Abschnitte `Range`, die Definitionen enthalten können:

```
Range.Env =
  NewScope(INCLUDING (Range.Env, Root.Env));
           nächst umgebende Range oder Root
```

definierendes Auftreten eines Bezeichners `IdDefScope`:

```
IdDefScope.Key =
  DefineIdn(INCLUDING Range.Env,
            IdDefScope.Symb);
```

angewandtes Auftreten eines Bezeichners `IdUseEnv`:

```
IdUseEnv.Key =
  KeyInEnv (INCLUDING Range.Env,
            IdUseEnv.Symb);
```

Vorbedingungen je nach Gültigkeitsregeln:
Algol-Regel: Vorbedingung für `KeyInEnv`:
alle `DefineIdn` aller umgebenden Ranges sind ausgeführt.

C-Regel: `KeyInEnv` und `DefineIdn` werden in der Reihenfolge des Programmtextes ausgeführt (links-abwärts).

weitere Prüfungen an dem Ergebnis der Bez.identifikation:

- keine Anwendung ohne Definition
- keine 2 Definitionen für einen Bezeichner in einem Range
- keine Anwendung vor der Definition (Pascal)
- Vorwärtsdefinitionen
- ...

Module 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 END;
SYMBOL TypeUseIdent INHERITS IdUseEnv END;
```

Benutzbare Attribute:

```
DefIdent.Key, UseIdent.Key
Program.Env, Block.Env
```

Instantiierung für Algol-Scope-Rules:

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

Instantiierung für C-Scope-Rules:

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

Instantiierung für neuen Namensraum

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

Symbolrollen: **`LabelRootScope, LabelRangeScope`**, ...

Instantiierung für zusätzliches Key-Attribut:

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

Attributnamen: `DefIdent.LabKey, UseIdent.LabKey`

PDL: Generator für Definitionsmodule

zentrale Datenstruktur ordnet **Programmobjekten** z*. B. Typen, Variablen, ...* **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: abhängige Operationen im Baum

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

Generierung des Definitionsmoduls:
Aus Spezifikationen

Eigenschaftsname: Eigenschaftstyp;

werden Funktionen `Reset, Set, Get` generiert.

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 vo 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 diese 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.

B: Eigenschaften von Programmobjekten

Vorbereitung: Programmobjekte werden meist durch Bezeichner im Programm benannt. In diesem Fall verwendet man einen Modul zur **Namensanalyse**, um Programmobjekte (repräsentiert durch `DefTableKeys`) and die Auftreten von Bezeichnern zu binden. Damit hat jeder Symbolknoten der Bezeichner ein `Key`-Attribut, dessen Wert das Programmobjekt identifiziert.

Berechnungen, die die Eigenschaft des Programmobjektes **setzen oder lesen**, sollten vorzugsweise im unteren oder oberen **Kontext des Symbols** für den Bezeichner plaziert werden, statt die `Key`-Werte im Baum weiterzureichen. So wird die Zugehörigkeit der Berechnung zum Programmobjekt deutlicher. Wenn möglich sollten **SYMBOL-Berechnungen** verwendet werden (siehe A).

Im allgemeinen benötigt man folgende Entwurfsschritte:

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).

Minisprache Core

Kleine imperative Programmiersprache, statisch typisiert.
Spezifikation der Analyse liegt vollständig vor.

Ziele:

- **Spezifikation der Typanalyse** kennenlernen
  -> Spezifikation verstehen, Sprache erweitern
- **Technik der Source-To-Source-Übersetzung** erlernen
  -> Abschnitt 7, Spezifikation vervollständigen

Core Beispielprogramm:

```
program
let
      a:    int;
      x, y: real;
      i = 10;
in
      input a;
      x := 3.5;
      y := a * i + x;
      output y;
end
```

Sprachkonstrukte:

- Block
- Variablen- und Konstantendeklarationen
- Zuweisungen
- E/A Anweisungen
- Ausdrücke

Erweiterungen

- Ablaufstrukturen
- geschachtelte Blöcke
- weitere Operatoren

Core: Struktur, Namensanalyse

Konkrete Syntax:

```
Program:         Source.
Source:          'program' Block.
Block:           'let' Declarations 'in'
                      Statements 'end'.
Declarations:    Declaration*.
Statements:      Statement*.

Declaration:     DefIdents ':' TypeDenoter ';'.
DefIdents:       DefIdent // ','.
DefIdent:        Ident.
TypeDenoter:     UseTypeIdent.
UseTypeIdent:    Ident.

Statement:       UseIdent ':=' Expression ';'.
Statement:       'input' UseIdent ';'.
Statement:       'output' Expression ';'.
...
Operand:         UseIdent.
Operand:         IntLiteral.
Operand:         FloatLiteral.
UseIdent:        Ident.
```

Ausdruckssyntax folgt später.
Symbolnotation wie in Pascal.

Namensanalyse (siehe Folie 5.4):

- prüfen: alle Bezeichner definiert, alle Definitionen eindeutig
- Bezeichner `int, real` mit Modul `PreDefine` vordefinieren:

```
PreDefKey ("int", intKey)
PreDefKey ("real", realKey)
```

Aufgaben und Begriffe der Typanalyse

Typisierte Programmobjekte (z. B. Variable)

- haben **Typ als Eigenschaft**;
- Typ wird durch Definition des Objektes bestimmt,
- Typ wird bei Benutzungen verwendet (Prüfung, Analyse)

Typisierte Programmkonstrukte (z. B. Ausdrücke)

- haben **Typ als Eigenschaft (Attribut);**
- sprachspezifische Regeln zur Bestimmung und Prüfung der Typen

Typen sind Programmobjekte

- Sie haben charakterisierende Eigenschaften; diese können wieder Typen enthalten.
- **vordefinierte Typen** (z. B. `int, real`)
- Typen, die durch **Typangaben** im Programm eingeführt werden (z. B. `record ... end`)
- **Typbezeichner** benennen einen Typ;
  Bindung Name - Typ durch **Typdefinition;**
  mehrere Namen können an denselben Typ gebunden sein

Relationen über Typen

- zwei Typen sind **gleich**,
  wenn sie dasselbe Typobjekt sind oder benennen, oder
  wenn sie sprachspezifische Regeln erfüllen
- ein Typ `tn` ist **verträglich** mit einem Typ `tw`,
  wenn Werte von `tn` als Werte von `tw` aufgefasst oder
  in solche umgewandelt werden können (**Coercion**).
- evtl. weitere sprachspezifische Relationen über Typen

Typrepräsentation des Moduls `Typing`

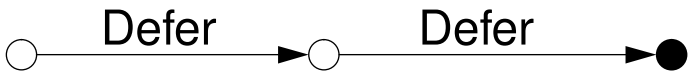
Typen werden durch **`DefTableKey`**s repräsentiert
`NoKey`: fehlerhafter, unbekannter Typ.

Relation `Defer` bindet Typ-Keys an die Keys von Typnamen:


```
type  t   =   record ... end
```

Indirekte Bindungen werden erst bei Anwendungen des Typs aufglöst; vermeidet Analysezyklen:

```
type  t    =   s;   s   =   record ... end
```



Die Funktion **`TransDefer`** angewandt auf einen Typnamens-Key oder Typ-Key liefert den Typ-Key am Ende der Defer-Kette.
Voraussetzung: alle Defer-Relationen existieren.

Defer-Ketten werden vom Modul **nicht zyklisch** erzeugt.

Eigenschaften von Typen werden immer den **Typ-Keys** (am Ende der Defer-Kette) zugeordnet.

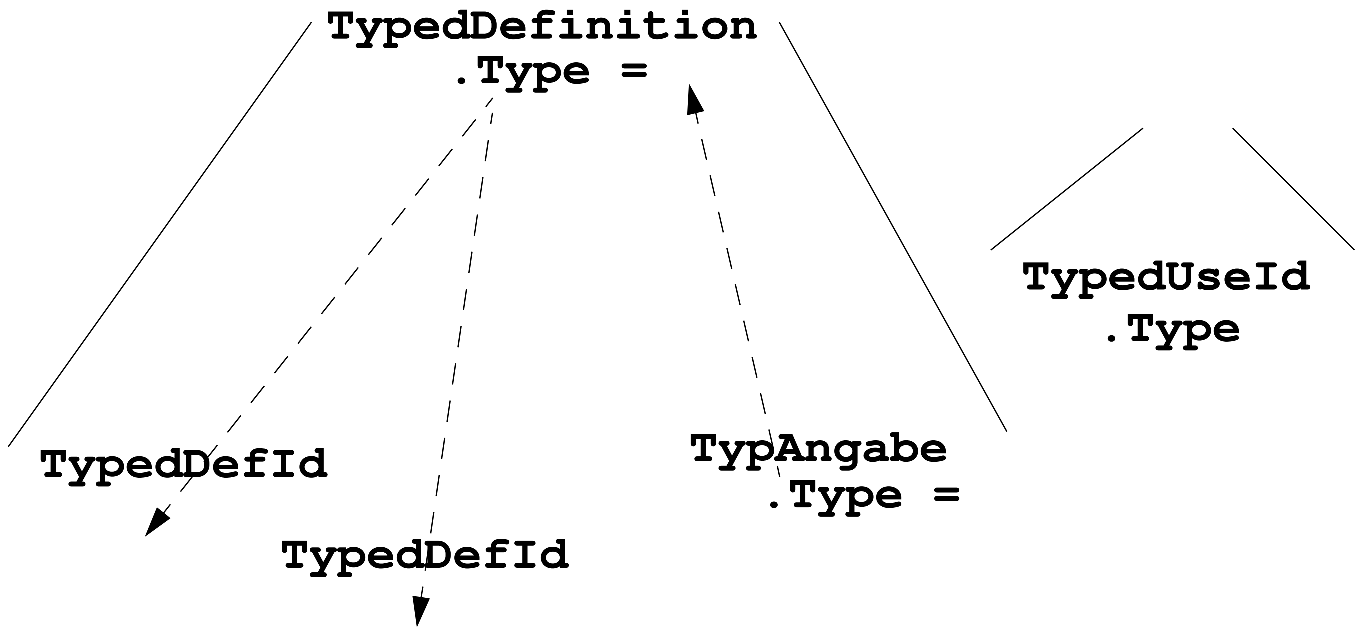
Typ-Keys werden durch die Eigenschaft **`IsType`** von anderen Keys unterschieden.

Typisierte Objekte

Programmobjekte (Variable, Parameter, usw.) haben einen Typ als Eigenschaft **`TypeOf`**.

Symbolrollen des Moduls Typing

- setzen die `TypeOf`-Eigenschaft (**`TypedDefId`**)
- im Kontext von Definitionen (**`TypedDefinition`**) und
- machen sie bei Anwendungen (**`TypedUseId`**) als Attribut **`Type`** verfügbar (Wert: Typ-Key am Ende der Defer-Kette)
- kennzeichnet Wurzel der Grammatik (**`RootType`**)



Zusammenwirken der Rollen garantiert die Reihenfolge für das Setzen und Lesen von `TypeOf` und Benutzen der Defer-Kette.

Beliebige **weitere Eigenschaften** können zusammen mit dem Typ gesetzt werden. Sie sind zusammen mit dem Typ verfügbar

```
SYMBOL DefIdent INHERITS TypedDefId COMPUTE
  SYNT.GotType = ResetKind (THIS.Key, ...);
END;
SYMBOL UseIdent INHERITS TypedUseId COMPUTE
  SYNT.Kind = GetKind (THIS.Key, ...)
    <- THIS.Type;
END;
```

Typnotationen

Typangaben z. B. Deklarationen, Signaturen, usw. sind **angewandte Typbezeichner** (Rolle `TypeDefUseId`) oder **Konstrukte, die einen Typ beschreiben** (z. B. Record, Array) (Rolle `TypeDenotation`)

Die Rolle `TypeDenotation` bildet ein **neues Typobjekt** (Type-Key). Beliebige **Eigenschaften des Typs** können in diesem Kontext gesetzt werden. Das Attribut `GotType` garantiert ihre Verfügbarkeit an Anwendungsstellen.

```
SYMBOL ArrayType INHERITS TypeDenotation END;
RULE: ArrayType ::= TypAngabe '[' Number ']'
COMPUTE
  ArrayType.GotType =
    ORDER (ResetElemType (ArrayType.Type,
                          TypAngabe.Type),
           ResetElemNo (ArrayType.Type, Number));
END;
```

Bei **Typbezeichnern als Typangabe (`TypeDefUseId`)** wird der Namens-Key als Typ verwendet; die Defer-Kette wird erst bei Anwendungen aufgelöst.

Bei **Anwendungen von Typeigenschaften**, die wieder Typen sind, muß `TransDefer` angewandt werden;
Vorbedingung: `<- INCLUDING RootType.GotType`
ist hier nicht nötig, da indirekt durch `Variable[2].Type`.

```
RULE: Variable ::= Variable '[' Expr ']' COMPUTE
  Variable[1].Type =
     TransDefer (GetElemType (Variable[2].Type,
                              NoKey));
END;
```

Überladene Operatoren

Ein Operatorsymbol (**Quelloperator**) kann verschiedene Bedeutungen (**Zieloperator**) haben, sie werden unterschieden durch die Typen der Operanden.
Werkzeug **OIL** und Modul **`Operator`** unterstützen die Analyse.

Ausdrucksgrammatik, z. B.

```
Expression: Expression AddOpr Factor / Factor.
Factor:     Factor MulOpr Operand / Operand.
Operand:    MonOpr Operand.
```

Beschreibung der **Quelloperatoren**
(Zeichen, konkr. Symbol, abstr. Symbol, eindeutiger Name)

```
SrcOpr ('+', AddOpr, BinOpr, AddKey)
SrcOpr ('*', MulOpr, BinOpr, MulKey)
SrcOpr ('-', MonOpr, UnOpr,  NegKey)
```

Beschreibung der **Zieloperatoren**
(Quellopr., eindeutiger Name, Signatur, Eigenschaftsinit.)

```
TgtOpr (AddKey, iAddKey, (intType,intType):intType,    TtStr={"+"})
TgtOpr (AddKey, rAddKey, (realType,realType):realType, TgtStr={"||"})
TgtOpr (MulKey, iMulKey, (intType,intType):intType,    TgtStr={"*"})
TgtOpr (MulKey, bAndKey, (realType,realType):realType, TgtStr={"&&"})
TgtOpr (NegKey, iNegKey, (intType):intType,            TgtStr={"-"})
TgtOpr (NegKey, bNotKey, (realType):realType,          TgtStr={"!"})
```

Implizite Typanpassung für OIL spezifizieren:

```
  COERCION cFloat (intType):realType;
```

Attributierung benutzt **Modulrollen `BinTgtOpr`, `ChkOpr`**:

```
SYMBOL Expression: Type, ReqType: DefTableKey;
SYMBOL BinOpr INHERITS BinTgtOpr, ChkOpr END;
RULE: Expression ::= Expression BinOpr Expression COMPUTE
  BinOpr.LType = Expression[2].Type;
  BinOpr.RType = Expression[3].Type;
  Expression[1].Type = BinOpr.ResType;
  Expression[2].ReqType = BinOpr.LTType;
  Expression[3].ReqType = BinOpr.RTType;
END;
```

Zieltexte erzeugen

Werkzeug PTG: Pattern-based Text Generator

Erzeugung strukturierter Texte in beliebigen Sprachen.
Anwendung als Berechnungen im Strukturbaum, und auch in beliebigen C-Programmen.

Prinzip:

1. Ausgabe-Pattern mit Einfügestellen spezifizieren:

```
ProgramFrame: $
              "void main () {\n"
              $
              "}\n"

Exit:         "exit (" $ int ");\n"

IOInclude:    "#include <stdio.h>"
```

2. PTG generiert Funktionen dazu. Aufrufe liefern Zielstruktur, bottom-up zusammensetzen:

```
PTGNode a, b, c;
a = PTGIOInclude ();
b = PTGExit (5);
c = PTGProgramFrame (a, b);
```

entsprechend mit Attributen im Strukturbaum

3. Ausgabe der Zielstruktur:

```
PTGOut (c);
PTGOutFile ("Output.c", c);
```

PTG-Muster: Wichtige Techniken

Indizierte Einfügestellen

Anwendung: **gleicher Text wird mehrfach eingefuegt**:

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

Aufruf `PTGModule (id, body);`

Anwendung: **Muster ändern - Aufruf beibehalten**:

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

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

immer Aufruf `PTGDecl (tp, ids);`

Typisierte Einfügestellen, Muster für Blatt-Texte:

```
CarId: $ string "-" $ string "-" $ int
```

Aufruf `PTGCarId ("PB", "AB", 127);`

String aus Csm-String-Speicher einsetzen:

```
CLASS SYMBOL IdOcc COMPUTE
  SYNT.Ptg = PTGId (TERM);
END;
```

Optionale Pattern-Elemente

Anwendung: Listen mit Trenner:

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

Aufruf `PTGCommaSeq (a, b);`

setzt Trenner nur ein, wenn weder `a` noch `b` gleich `PTGNULL` ist

Einige nützliche Pattern sind verfügbar in

`$/Output/PtgCommon.fw`

Zieltext im Strukturbaum zusammensetzen

Benachbarte Kontexte:

```
ATTR Ptg: PTGNode;

RULE: Statement ::= UseIdent ':=' Expression
COMPUTE
  Statement.Ptg =
    PTGAssign (UseIdent.Ptg, Expression.Ptg);
END;
```

Zusammenfassen bei ähnlichen Quell- und Zielstrukturen:

```
SYMBOL Statements COMPUTE
  SYNT.Ptg =
    CONSTITUENTS Statement.Ptg
    WITH (PTGNode, PTGSeq, IDENTICAL, PTGNull);
END;
```

Unregelmäßiges Zusammenfassen in link-rechts Ordnung:

```
CHAIN ConstPtg: PTGNode;
SYMBOL Block COMPUTE
  CHAINSTART HEAD.ConstPtg = PTGNULL;
END;
RULE: Declaration::= DefIdent '=' IntLiteral ';'
COMPUTE
  Declaration.ConstPtg =
    PTGSeq (Declaration.ConstPtg,
            PTGConstDecl
              (DefIdent.Ptg, IntLiteral.Ptg));
END;
```

Konsistente Umbenennung von Bezeichnern

Zu jedem bezeichneten Quellobjekt soll ein **eindeutiger Zielbezeichner** erzeugt werden, der gut **lesbar ist.**

Die **Bindung** der Zielbezeichner soll **unabhängig von den Regeln der Zielsprache** sein.

PTG-Muster hängt Nummer an den Quellbezeichner an:

```
  TransId:  $ string /*src name*/ "_" $ int
```

PDL-Eigenschaft: Übersetzung des Programmobjektes:

```
  TransId: PTGNode; "ptg_gen.h"
```

Berechnung zu allen Bezeichnerknoten **im Baum**:

```
CLASS SYMBOL IdOcc COMPUTE
  SYNT.Ptg = SetOrGetTransId (THIS.Key, TERM);
END;
```

C-Modul implementiert die Funktion `SetOrGetTransId`:

```
static int count = 1;
static PTGNode errid = PTGTransId ("_ERR", 0);

PTGNode SetOrGetTransId (DefTableKey k, int sym)
{ PTGNode result = GetTransId (k, PTGNULL);
  if (k == NoKey) return errid;
  if (result == PTGNULL)
  { result =
       PTGTransId (StringTable (sym), count++);
     ResetTransId (k, result);
  }
  return result;
}
```

Projektaufgabe

Generator für Implementierungen von Verbundtypen:

Verbundtypen sollen durch eine einfache Sprache beschrieben werden, z. B.

```
import "String.h"
Adresse
     (name:  String;
      plz:   int;
      Stadt: String;
     )
```

Daraus soll ein **ANSI-C-Modul** mit Schnittstellen- und Implementierungsdatei erzeugt werden.
Er definiert

- den **Verbundtyp und seinen Pointertyp**,
- eine **Erzeugerfunktion** `newAdresse`, die ein Verbundobjekt erzeugt und den Pointer darauf liefert,
- für jede Komponente ein **Zugriffsmakro**, z. B.
  `#define nameOfAdresse(x) ((x)->name_fld)`
- **`#include`-Anweisungen** für die benutzten Typen

Vorgehen:

- **Zieltext für das Beispiel** aufschreiben,
- **Konkrete Syntax** der Sprache entwerfen,
- Aufgaben der **Namensanalyse** aufschreiben, spezifizieren,
- notwendige **Eigenschaften von Objekten** feststellen, spezifizieren,
- **Zieltext-Erzeugung** spezifizieren,
- Generator und seine Produkte **validieren**.

Zieltextbeispiel zur Projektaufgabe

Eingabeprogramm `adr`:

```
import "String.h"
Adresse
     (name:  String;
      plz:   int;
      Stadt: String;
     )
```

Ausgabedatei `adr.h`:

```
#include "String.h"

typedef struct __Adresse *Adresse;
typedef struct __Adresse {
  String name_fld;
  int plz_fld;
  String stadt_fld;
} _Adresse;

#define nameOfAdresse(x) ((x)->name_fld)
#define plzOfAdresse(x) ((x)->plz_fld)
#define stadtOfAdresse(x) ((x)->stadt_fld)

extern Adresse newAdresse ();
```

Ausgabedatei `adr.c`:

```
#include "adr.h"
#include <stdlib.h>

Adresse newAdresse ()
{ return (Adresse)malloc(sizeof(_Adresse));
}
```