Spezifikation und Implementierung der Transformation attributierter Bäume

Bertram Vielsack

Diplomarbeit

Universität Karlsruhe Fakultät für Informatik

Juni 1989

Betreuer:

Prof. Dr. G. Goos
Prof. Dr. S. Jähnichen
Dr. J. Grosch

Hiermit erkläre ich, daß ich die vorliegende Arbeit selbständig erstellt habe und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Karlsruhe, im Juni 1989

(Bertram Vielsack)

1. Einleitung

Bei der Erstellung von Übersetzern spielen Zwischensprachen, die durch attributierte Bäume dargestellt werden, eine wichtige Rolle. Die Zwischensprachen gliedern den komplexen Übersetzungsvorgang in mehrere Transformationen und damit den Entwurf und die Entwicklung eines Übersetzers in mehrere unabhängige Teilaufgaben.

Wesentliche Aufgaben beim Bau eines Übersetzers bestehen also darin, attributierte Bäume zu transformieren. Das Ergebnis einer Transformation kann wiederum ein attributierter Baum sein. Andere Strukturen (Graph, Liste, sequentielle Ausgabe etc.) sind ebenso möglich. Der einheitlichen Darstellung der Eingabe steht also eine Vielfalt an Möglichkeiten zur Darstellung der Ausgabe gegenüber.

Ein Ziel der vorliegenden Arbeit ist die Entwicklung einer Spezifikationssprache (ESTRAL[1]) zur Beschreibung von Transformationen. Diese Sprache muß einerseits an die einheitliche Struktur der Eingabe angepaßt sein. Andererseits müssen universelle Mittel zur Beschreibung der Ausgabe bereit gestellt werden.

Zur automatischen Umsetzung einer Spezifikation in eine Implementierung wird ein Generator (ESTRA[2]) entwickelt.

Spezifikationssprache und Generator zusammen ermöglichen es, bei der Realisierung von Transformationen attributierter Bäume von der Implementierung auf eine problemorientierte Spezifikation überzugehen.

---

[1] ESTRAL - **E**fficient **S**tructure tree **TRA**nsformation **L**anguage

[2] ESTRA - **E**fficient **S**tructure tree **TRA**nsformation

2. Möglichkeiten zur Beschreibung von Transformationen

Eine Transformation im Sinne dieser Arbeit ist jede Abbildung eines attributierten Baumes. Ob es sich beim Ergebnis einer solchen Abbildung wiederum um einen attributierten Baum handelt oder nicht spielt hierbei keine Rolle. Entscheidend ist einzig die Struktur der Eingabe.

Es werden drei unterschiedliche Möglichkeiten zur Beschreibung von Transformationen vorgestellt.

2.1. Attributierte Grammatiken

Attributierte Grammatiken (AGs) [Knu68] werden im Übersetzerbau vorwiegend eingesetzt, um die Semantische Analyse zu beschreiben. Sie werden aber auch zu Beschreibung der Codeerzeugung und -optimierung benutzt. AGs können auch benutzt werden, um Transformationen zu beschreiben.

Normalerweise wird dabei so vorgegangen, daß das Ergebnis der Transformation durch den Wert eines Attributs an der Wurzel des Baumes dargestellt wird. Die Konsequenz dieser Vorgehensweise ist, daß das komplette Ergebnis der Transformation als Wert gespeichert werden muß. Falls eine Ausgabe dieses Wertes erfolgen soll, muß dies außerhalb des AG-Kalküls beschrieben werden. Die Verwendung von Seiteneffekten zum sequentiellen Aufbau der Ausgabe ist mit diesem Ansatz nicht ohne weiteres möglich, da die Reihenfolge, in der die einzelnen Attribute berechnet werden, nicht unmittelbar gesteuert werden kann.

Falls der Wunsch besteht, Seiteneffekte einzusetzen (z.B. zur Dateiausgabe), muß die gewünschte Auswertungsreihenfolge durch Verwendung zusätzlicher Attribute mit entsprechenden Abhängigkeiten erzwungen werden. Diese komplizierte und aufwendige Art der Steuerung der Reihenfolge, die zudem recht unnatürlich ist, könnte vermieden werden, wenn der sequentielle Ablauf durch imperative Programmierung unmittelbar beschrieben werden könnte.

Transformationen realisieren in der Regel Abbildungen, die mehrere Lösungen zulassen und einen sequentiellen Aufbau einer solchen Lösung nahelegen. Transformationen sind also von Natur aus mehrdeutig und sequentiell. Es ist deshalb sinnvoll, zur Beschreibung einer Transformation eine mehrdeutige Spezifikation zu verwenden. Die Attributberechnung muß allerdings auf alle Fälle eindeutig sein. Mehrdeutigkeit in der Logik der Transformation muß deshalb in der AG zur Beschreibung der Transformation durch zusätzliche Attribute und Berechnungen eindeutig gemacht werden.

Auf Grund der mehrdeutigen und sequentiellen Natur von Transformationen bieten AGs in ihrer Reinform keine ausreichenden Mittel, um Transformationen sinnvoll zu beschreiben.

2.2. Modifikation der Eingabe

Eine weitere Möglichkeit, Transformationen durchzuführen besteht darin, die Ausgabe durch Modifikation der Eingabe zu erzeugen, wie es beispielsweise in [MWW84] beschrieben wird. Der Eingabebaum wird hier in einem iterativen Prozeß so lange umgebaut, bis das Ergebnis der gewünschten Ausgabe entspricht. Die Transformation kann beschrieben werden, indem die einzelnen Schritte diese Prozeßes durch Vorschriften festgelegt werden. Die Eleganz dieser Methode besteht darin, daß eine Transformation durch einzelne aufeinander aufbauende Schritte einfach und anschaulich beschrieben werden kann.

Die Reihenfolge und der Ort, auf den die einzelnen Vorschriften angewandt werden, kann die Anwendung weiterer Vorschriften und letztlich das gesamte Ergebnis entscheidend beeinflussen. Zur Beschreibung einer Transformation muß deshalb neben den Vorschriften eine Strategie festgelegt werden, die Reihenfolge und Ort der Regelanwendungen bestimmt.

Es ist möglich und i.a. sogar erwünscht, daß durch eine Modifikation eine weitere Modifikation erst möglich wird. Das birgt jedoch die Gefahr in sich, daß der iterative Prozeß nie endet, da

immer weitere Modifikationen möglich werden. Neben der Frage nach der Terminierung ergeben sich aus diesem Umstand Probleme bei der Abschätzung des Aufwands, der zur Durchführung der Transformation erforderlich ist.

Bei einer Modifikation kann nicht ausgeschlossen werden, daß die Werte der Attribute des Baumes verloren gehen oder zumindest inkonsistent werden. Falls die einzelnen Vorschriften an Bedingungen über die Attribute geknüpft werden sollen, ist es deshalb notwendig, unmittelbar nach jeder Modifikation eine Reattributierung durchzuführen. Wenn man auf eine Reattributierung verzichten will, verbleibt die Möglichkeit, die Modifikation der Eingabe einzusetzen, um nicht attributierte Bäume zu transformieren.

2.3. Funktionale Abbildung

Das Wesen einer funktionalen Abbildung eines attributierten Baumes besteht darin, daß der Ausgabewert neu berechnet wird, der alte Eingabebaum wird nur gelesen und bleibt somit unverändert. Das Problem der Reattributierung, wie es bei der Modifikation der Eingabe besteht, kann folglich nicht entstehen.

Durch eine imperative Beschreibung der Abbildung ist es unmittelbar möglich, die Reihenfolge der Abarbeitung zu steuern, so daß eine sequentielle Ausgabe durch den Einsatz von Seiteneffekten ebenso möglich ist wie der Aufbau eines neuen Baumes.

Eine mehrdeutige Beschreibung von Transformationen ist durch den Einsatz von Mustern und Kosten auf einfache und elegante Weise möglich.

Das sequentielle und mehrdeutige Wesen von Transformationen kann mit dieser Methode unmittelbar beschrieben werden. Probleme, die durch den Einsatz von AGs oder die Modifikation der Eingabe entstehen, werden somit vermieden.

3. Anforderungen an eine funktionale Beschreibung

Im folgenden wird an Beispielen gezeigt, welche Anforderungen eine funktionale Beschreibung von Transformationen erfüllen muß, und mit welchen Mitteln diese erfüllt werden können.

Die einführenden Beispiele aus dem Bereich der Codeerzeugung wurden ausgewählt, da sie sich eignen viele Probleme, die bei der Beschreibung von Transformationen entstehen, darzustellen. Gleichwohl sind dies nicht die typischen Anwendungsbeispiele, da für die Beschreibung der Codeerzeugung spezielle Sprachen und Werkzeuge [Emm88] existieren.

Die in den Beispielen verwendete Notation gibt einen Vorgeschmack auf die Spezifikationssprache ESTRAL, die in Kapitel 5 ausführlich beschrieben wird.

3.1. Abbildung der einzelnen Knoten

Eine einfache Methode, Strukturbäume zu transformieren, besteht darin, alle Knoten eines Baumes zu besuchen und dabei das Ergebnis der Transformation sukzessive durch Abbildung der einzelnen Knoten zu erzeugen. Würde man eine feste Reihenfolge (z.B. Postfix) zum Besuch der einzelnen Knoten festlegen, so würde es genügen, die Abbildung für jeden Knoten festzulegen. Diese einfache Methode wäre bereits ausreichend, um Code zur Berechnung von arithmetischen Ausdrücken auf einer Kellermaschine zu erzeugen (Abb. 3.1).

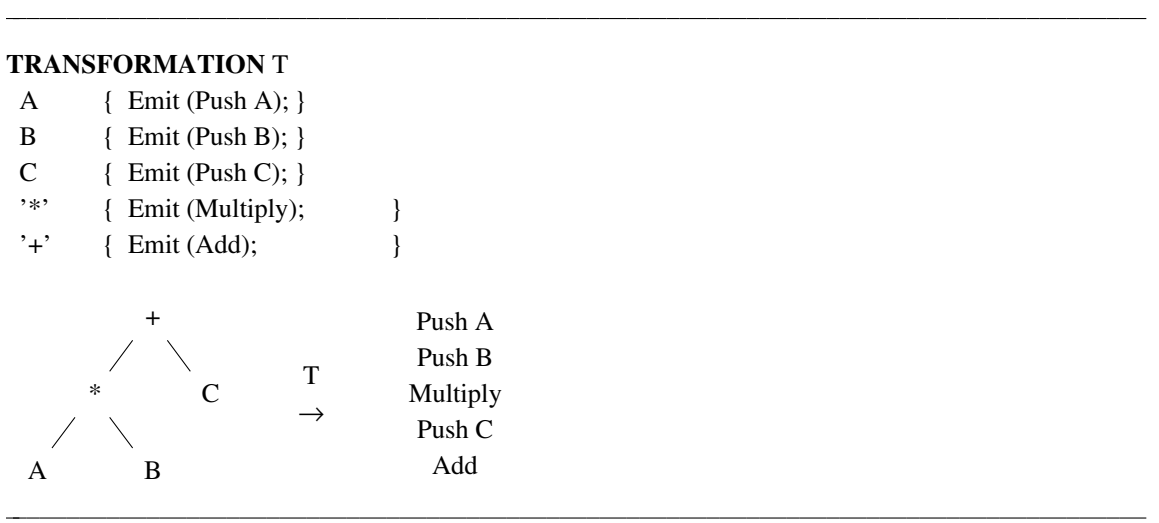

Abb. 3.1: Transformation eines Strukturbaumes

Doch ist diese Methode nur begrenzt einsetzbar. Betrachten wir hierzu folgendes Beispiel (Abb. 3.2).

Abb. 3.2: Strukturbaum einer Zuweisung

Hier wäre es offensichtlich falsch, für den ersten Operanden der Zuweisung ein 'Push A' zu

generieren. Hingegen ist dies für den ersten Operanden der Addition weiterhin erforderlich. Außerdem zeigt das Beispiel, daß eine feste Ablaufstrategie (wie Postfix) zum Besuch der einzelnen Knoten nicht immer brauchbar ist, denn bevor die Zuweisung erfolgen kann (Abb. 3.2), muß die Summe berechnet werden. D.h. die Addition mit ihren Operanden A und B ist vor dem linken Operand der Zuweisung zu besuchen.

3.2. Steuerung der Reihenfolge

Derartige Probleme können gelöst werden, wenn man die Transformation in *Funktionen* gliedert und sowohl die Reihenfolge als auch die Art der durchzuführenden Funktionen vom Kontext abhängig macht.

TRANSFORMATION CODE

FUNCTION PUSH

```
':='    {  F1 (':='.ro); F2 (':='.lo);                }
A       {  Emit (Push A);                             }
B       {  Emit (Push B);                             }
'+'     {  F1 ('+'.lo); F1 ('+'.ro); Emit (Add);      }
```

FUNCTION STORE

```
A       {  Emit (Store A);                            }
B       {  Emit (Store B);                            }
```

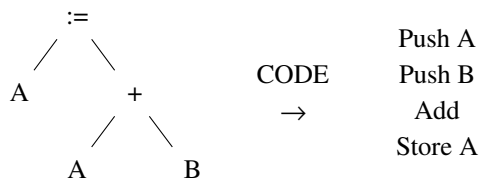

Abb. 3.3: Gliederung einer Transformation in mehrere Funktionen

In den Vorschriften (Abb. 3.3) erfolgt ein expliziter Aufruf von zum Teil verschiedenen Funktionen für die einzelnen Operanden (.lo bezeichnet hierbei den linken, .ro den rechten Operanden des angegebenen Knotens). Die Abarbeitungsreihenfolge ist nun explizit spezifiziert.

3.3. Zugriff auf die Attribute

Die Vorschriften zur Abbildung der Knoten A und B in Abb. 3.3 sind offensichtlich ähnlich. Dies liegt daran, daß es sich jeweils um einen Knoten handelt, der eine Variable darstellt. In solchen Fällen sollte es möglich sein, die Abbildung durch eine einzige Vorschrift zu beschreiben. Um das zu erreichen, fassen wir die Namen der Variablen als *Attribute* auf. Ein nunmehr attributierter Strukturbaum erhält damit etwa folgende Gestalt (Abb. 3.4).

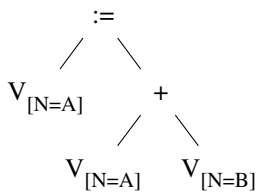

Abb. 3.4: Attributierter Strukturbaum einer Zuweisung

An Stelle der Knoten A und B erscheinen nur noch Knoten vom Typ V (Variable), die ein Attribut N (Name der Variablen) besitzen, das hier die Werte A und B annimmt. Die Beschreibung der Transformation nimmt dann folgende Form an (Abb. 3.5).

```
TRANSFORMATION CODE

FUNCTION PUSH
 ':='    {  F1 (':='.ro); F2 (':='.lo);                }
 V       {  Emit (Push V.N);                           }
 '+'     {  F1 ('+'.lo); F1 ('+'.ro); Emit (Add);      }

FUNCTION STORE
 V       {  Emit (Store V.N);                          }
```

Abb. 3.5: Beschreibung der Transformation eines attributierten Strukturbaumes

3.4. Berechnung eines synthetisierten Attributs

Bei der bisher betrachteten Kellermaschine werden Ergebnisse eines Teilausdrucks auf dem Keller abgelegt. Hat man es dagegen mit einer Registermaschine zu tun, wird man Zwischenergebnisse in einem Register ablegen. Um nun Code für eine Operation zu erzeugen, ist es erforderlich, zu wissen in welchen Registern die Zwischenergebnisse stehen.

Diese Information kann zur Verfügung gestellt werden, wenn gleichzeitig mit der Transformation eine *S-Attributierung* (Attributierung, bei der nur synthetisierte Attribute berechnet werden) durchgeführt wird. Da die Attribute bei der Transformation unmittelbar verarbeitet werden, ist es nicht notwendig, sie explizit im Baum zu speichern, es genügt, wenn die Attribute unmittelbar nach der Transformation eines Teilbaums zur Verfügung stehen. Dies wird erreicht, indem die Attribute als Ergebnisse der einzelnen Funktionen dargestellt werden.

In Abb. 3.6 wird dieser Mechanismus benutzt um festzuhalten, in welchem Register das Zwischenergebnis steht. Die Funktion Reg, die die Transformation durchführt, liefert dieses Register als Ergebnis.

```
TRANSFORMATION T

FUNCTION Reg: register
 '+'     {
         i := Reg ('+'.lo);
         j := Reg ('+'.ro);
         Emit (ADD Rj, Ri);
         FreeReg (j);              (* Register j ist nun wieder frei *)
         RETURN i;
         }

 '*'     {
         i := Reg ('*'.lo);
         j := Reg ('*'.ro);
         Emit (MULS Rj, Ri);
         FreeReg (j);              (* Register j ist nun wieder frei *)
         RETURN i;
         }

 V       {
         i := GetReg ();           (* Beschaffe Register für Zwischenergebnis *)
         Emit (MOVE V.N@, Ri);
         RETURN i;
         }
```

Abb. 3.6: S-Attributierung

Die Registervergabe erfolgt 'on the fly'. Um das Beispiel einfach zu halten, wurde davon ausgegangen, daß immer genügend Register vorhanden sind, was z.B. dann der Fall ist, wenn bei der Erzeugung von Zwischencode Pseudoregister an Stelle der realen Register verwendet werden.

Das Beispiel in Abb. 3.6 zeigt, daß es nun prinzipiell möglich ist, eine Transformation zu beschreiben, die Code für eine Registermaschine liefert. Allerdings ist der generierte Code schlecht im Vergleich zu der in Abb. 3.7 angegebenen Transformation.

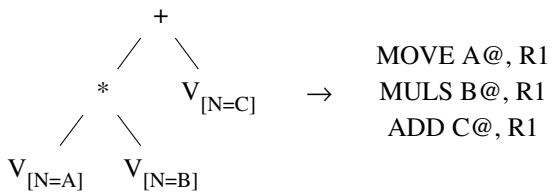

Abb. 3.7: optimierte Transformation eines arithmetischen Ausdrucks

3.5. Muster

Offensichtlich ist es nicht erforderlich, die zweiten Operanden der Addition und Multiplikation in ein Hilfsregister zu laden, wenn es sich dabei wie hier um eine (direkt zugreifbare) Variable handelt. Da wir aber bislang immer nur einzelne Knoten abgebildet haben, konnte bei der Transformation der Addition (bzw. Multiplikation) nicht erkannt werden, ob es sich beim rechten Operanden um eine Variable handelt. Im folgenden werden wir deshalb an Stelle von einzelnen Knoten (kleine) Ausschnitte möglicher Strukturbäume, sogenannte *Muster*, verwenden, um eine Transformation festzulegen.

Diese Muster müssen nun auf den Teil des Strukturbaums passen, der mit dem aktuellen (d.h. dem zu transformierenden) Knoten beginnt. Die Muster werden durch ihre geklammerte Prefixform dargestellt. Um Operanden zu beschreiben, die nicht eindeutig festgelegt sind, können Nichtterminale verwendet werden. So wird z.B. in Abb. 3.8 das Nichtterminal expr verwendet, wenn beliebige Ausdrücke erlaubt sind. Um in den Anweisungen leichter auf den Baum zugreifen zu können, können die Namen der Knoten und Nichtterminale, die im Muster stehen, verwendet werden. Falls es hierbei zu Mehrdeutigkeit kommt, kann diese durch freigewählte Bezeichner, die im Muster mit angegeben werden, aufgelöst werden (siehe e1 und e2 in Abb. 3.8).

Mit Hilfe von Mustern sollte es nun möglich sein, eine Transformation zu beschreiben, die in der Lage ist, den optimalen Code von Abb. 3.7 zu erzeugen. Dazu erweitern wir die Beschreibung von Abb. 3.6 um zwei Vorschriften, die die Spezialfälle behandeln, daß es sich beim rechten Operanden einer Addition bzw. Multiplikation um eine (direkt zugreifbare) Variable handelt.

3.6. Mehrdeutigkeit und Kosten

Offensichtlich muß die dadurch entstandene Beschreibung mehrdeutig sein, da ja die alte Möglichkeit der nicht optimalen Codeerzeugung weiterhin besteht. Damit die optimale Lösung ausgewählt werden kann, werden Kosten für die Vorschriften eingeführt. Mit Hilfe dieser Kosten lassen sich dann die Kosten der Transformation als Summe der Kosten aller angewandten Vorschriften berechnen. Im Falle einer Mehrdeutigkeit wird nun die optimale Lösung, d.h. die Lösung mit den geringsten Kosten ausgewählt.

Wenn die Länge des erzeugten Codes zur Festlegung der Kosten herangezogen wird, ergibt sich für unser Beispiel die Beschreibung von Abb. 3.8.

```
TRANSFORMATION T

FUNCTION Reg: Register
 '+' (e1: expr, e2: expr)    COST 2
                             {
                             i := Reg (e1); j := Reg (e2);
                             Emit (ADD Rj, Ri);
                             FreeReg (j); RETURN i;
                             }

 '*' (e1: expr, e2: expr)    COST 2
                             {
                             i := Reg (e1); j := Reg (e2);
                             Emit (MULS Rj, Ri);
                             FreeReg (j); RETURN i;
                             }

 V ()                        COST 4
                             {
                             i := GetReg ();
                             Emit (MOVE V.N@, Ri);
                             RETURN i;
                             }

 '+' (expr, V ())            COST 4
                             {       (* rechter Operand ist eine Variable *)
                             i := Reg (expr);
                             Emit (ADD V.N@, Ri)
                             RETURN i;
                             }

 '*' (expr, V ())            COST 4
                             {       (* rechter Operand ist eine Variable *)
                             i := Reg (expr);
                             Emit (MULS V.N@, Ri)
                             RETURN i;
                             }
```

Abb. 3.8: Mehrdeutigkeit und Kosten zur Optimierung einer Transformation

Betrachten wir nochmals die Lösungen von Abb. 3.6 und 3.7 und berechnen jeweils die Kosten. Im ersten Fall ergibt sich eine Summe von 16, im zweiten (optimierten) Fall sind es nur 12. Die Einführung von Kosten liefert also offensichtlich das gewünschte Ergebnis.

3.7. Bedingungen

Attribute des Strukturbaumes wurden bislang nur benutzt, um die bei der Codeerzeugung notwendigen konkreten Werte (z.B. Adressen) zur Verfügung zu stellen. Es kann aber auch vorkommen, daß eine Vorschrift nur dann verwendet werden darf, wenn Attributwerte eine spezielle *Bedingung* erfüllen.

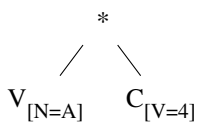

Abb. 3.9: Produkt einer Variablen mit einer Konstanten (Zweierpotenz)

Um beispielsweise das Produkt einer Variablen und der Konstanten 4 (Abb. 3.9) zu berechnen, genügt es, den Wert der Variablen um zwei Binärstellen nach links zu verschieben. Dieses Verfahren geht aber offensichtlich nicht für beliebige Konstanten, sondern nur für solche, die eine Zweierpotenz darstellen. Um dies durch eine Vorschrift beschreiben zu können, führen wir die Möglichkeit ein, eine Vorschrift mit einer Bedingung (condition) zu verknüpfen (Abb. 3.10).

```
'*' (expr, C ())        COST 2
                        CONDITION { IsPowerOf2 (C.V) }
                        {
                        i := Reg (expr);
                        d := Log2 (C.V);
                        Emit (ASL #d, Ri)
                        RETURN i;
                        }
```

Abb. 3.10: Vorschrift mit einer Bedingung

Durch die Kombination Mehrdeutigkeit, Bedingungen und Kosten ist es auf einfache Weise möglich, Transformatoren zu beschreiben, die in der Lage sind, eine optimierte Abbildung durchzuführen.

3.8. Mehrfachtransformation

Eine weiteres Mittel bei der Beschreibung von Transformationen, das in den bisherigen Beispielen noch nicht verwendet wurde, ist die *Mehrfachtransformation*. Eine Mehrfachtransformation liegt dann vor, wenn ein Teilbaum mehrmals auf die selbe (Abb. 3.13) oder auf unterschiedliche Weise transformiert wird.

```
TRANSFORMATION T

FUNCTION Copy: tree
 WHILE (expr, stmt)      {
                         e1 := Copy (expr);
                         s  := Copy (stmt);
                         e2 := Copy (expr);
                         e3 := MakeNOT (e2);
                         r  := MakeREPEAT (s, e3);
                         RETURN MakeIF (e1, r);
                         }
```

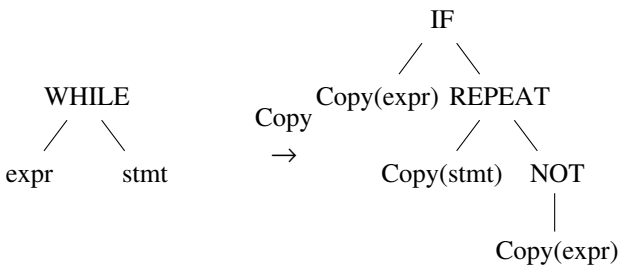

Abb. 3.11: Transformation einer While-Schleife

Eine While-Schleife kann durch eine Repeat-Schleife ersetzt werden, wenn letztere in eine bedingte Anweisung (IF) eingeschlossen wird. Der Ausdruck (expr) der Bedingung muß hierzu offensichtlich dupliziert werden. Abb. 3.11 zeigt, wie dies durch eine Mehrfachtransformation realisiert wird.

Eine andere Form der Mehrfachtransformation ist die mehrfache Transformation eines Strukturbaumes auf unterschiedliche Weise. Bei dieser Form der Mehrfachtransformation werden verschiedene Funktionen auf einen Teilbaum angewandt.

3.9. Synthetisierte und vererbte Attribute

Bei komplexen Transformation reicht eine S-Attributierung, wie sie 2.4. eingeführt, nicht aus.

Ein Beispiel für eine solche Transformation ist die Normierung des Strukturbaumes eines regulären Ausdrucks. Ein regulärer Ausdruck ist ein Ausdruck, der den folgenden Regeln gehorcht:

1. jeder Bezeichner (Identifier) ist ein regulärer Ausdruck
2. wenn R1 und R2 reguläre Ausdrücke sind, dann sind auch:
   R1 '|' R2 (Alternative)
   R1 R2 (Sequenz)
   (R1)
   R1 '*' (Wiederholung)
   Reguläre Ausdrücke.

Da es sich bei der Sequenz (das selbe gilt für die Alternative) um eine assoziative Operation handelt, gibt es unterschiedliche Möglichkeiten zur Darstellung des selben regulären Ausdrucks in Form eines Strukturbaumes. Durch die unnötige Verwendung von Klammern oder die automatische Erzeugung von regulären Ausdrücken ist es unter Umständen unvermeidlich, daß diese verschiedenen Strukturbäume tatsächlich aufgebaut werden.

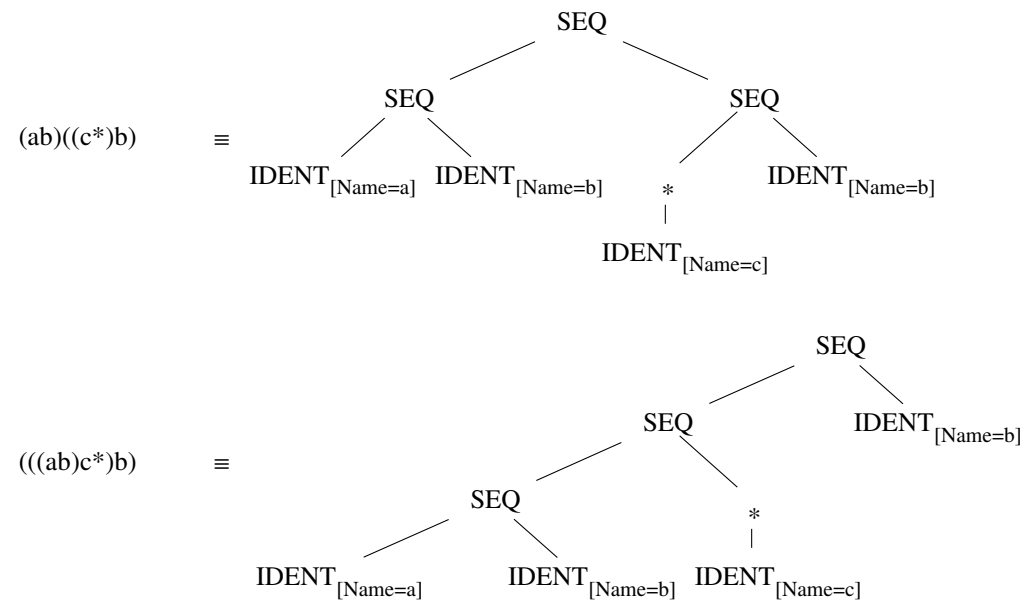

Abb. 3.12: verschiedene Darstellungen des selben regulären Ausdrucks

Um solche Strukturbäume (Abb. 3.12) zu normieren, d.h. für eine einheitliche Darstellung der regulären Ausdrücke zu sorgen, geht man zweckmäßigerweise wie folgt vor:

1. Nicht assoziative und unäre Operatoren werden eins zu eins abgebildet.
2. Beim Antreffen eines assoziativen Operators werden alle darunterliegenden Knoten mit demselben Operator zusammengefaßt. Die darunterliegenden Teilbäume werden der Reihe nach transformiert. Dabei wird ein normierter (zur Liste entarteter) Baum mit dem aktuellen Operator aufgebaut, dessen Blätter durch die Transformationsergebnisse der Teilbäume gebildet werden.

Um dies zu realisieren reicht eine S-Attributierung deshalb nicht aus, weil die Bäume zur Normierung nicht nur hochgereicht, d.h. *synthetisiert*, sondern auch nach unten gegeben d.h. *vererbt* werden müssen.

Um diese Vererbung beschreiben zu können, ordnen wir einer Funktion zusätzlich zum Ergebnisattribut noch weitere vererbte und synthetisierte Attribute zu. Diese Attribute müssen beim Aufruf einer Funktion neben dem zu transformierenden Strukturbaum, der immer das erste Argument bildet, übergeben werden.

```
TRANSFORMATION OrderTree

FUNCTION Order: tTree
  Alt (r1: RegExpr, r2: RegExpr)              { RETURN OrderAlt (r2, Order (r1)); }
  Seq (r1: RegExpr, r2: RegExpr)              { RETURN OrderSeq (r2, Order (r1)); }
  Star (RegExpr)                              { RETURN MakeSTAR (Order (RegExpr)); }
  Ident ()                                    { RETURN MakeIdent (Ident.Name); }

FUNCTION OrderSeq          List: tTree -> : tTree
  Seq (r1: RegExpr, r2: RegExpr)  COST 1      { RETURN OrderSeq (r2, OrderSeq (r1, List)); }
  Ident ()                        COST 1      { RETURN MakeSEQ (List, MakeIdent (Ident.Name)); }
  RegExpr                         COST 2      { RETURN MakeSEQ (List, Order (RegExpr)); }

FUNCTION OrderAlt          List: tTree -> : tTree
  Alt (r1: RegExpr, r2: RegExpr)  COST 1      { RETURN OrderAlt (r2, OrderAlt (r1, List)); }
  Ident ()                        COST 1      { RETURN MakeALT (List, MakeIdent (Ident.Name)); }
  RegExpr                         COST 2      { RETURN MakeALT (List, Order (RegExpr)); }
```

Abb. 3.13: Normierung eines regulären Ausdruckes durch Transformation

Die Funktion OrderSeq sammelt die gesamte Sequenz ein und baut mit Hilfe des vererbten Attributs List (vererbte Attribute stehen links von Pfeil ('->'), synthetisierte rechts) und dem Ergebnisattribut sukzessive den normierten Baum für die Sequenz auf. Die allgemeine Vorschrift (die Vorschrift mit dem Muster 'RegExpr') wird aufgrund der Kosten nur für die übrigen Operatoren (nicht für die Sequenz) gewählt. Sie stößt dann wiederum die Funktion Order für den aktuellen Knoten an. Die Funktion OrderAlt arbeitet entsprechend für Alternativen.

4. Begriffe

Bevor weiter auf die Spezifikation und die Implementierung von Transformationen eingegangen wird, müssen einige zentrale Begriffe geklärt werden.

4.1. Baumgrammatik

Eine Baumgrammatik ist eine spezielle kontextfreie Grammatik, die hier eingesetzt wird, um die Struktur der zu transformierenden Bäume zu beschreiben.

Eine Baumgrammatik ist ein Viertupel $G = (C, N, \Pi, Z)$ mit

1. C = Menge der Klassen
2. N = Menge der Knoten
3. Π = Menge der Produktionen $\subseteq C \times (C \cup N\, C^*)$
4. Z = Menge der Startsymbole $\subseteq C$

Die Klassen C einer Baumgrammatik entsprechen den Nichtterminalen, die Knoten den Terminalen einer gewöhnlichen kontextfreien Grammatik.

Bei den Produktionen Π einer Baumgrammatik werden zwei Typen unterschieden:

1. Kettenproduktion:
   $c_0 \rightarrow c_1$ mit $c_0, c_1 \in C$
2. Knotenproduktion:
   $c_0 \rightarrow n\ (c_1, ..., c_m)$ mit $n \in N, c_0, c_1, ..., c_m \in C$

Die *Kettenproduktion* ersetzt eine Klasse c_0 durch eine Klasse c_1; c_0 heißt Oberklasse von c_1.

Bei der *Knotenproduktion* werden ein Knoten n aufgebaut, und die Klassen $c_1, ..., c_m$ seiner Söhne festgelegt. Da die rechte Seite einer Knotenproduktion einen Knoten n beschreibt wird sie auch als Knotenbeschreibung (oder kurz Beschreibung) des Knotens n bezeichnet. Die Klammern in den Knotenproduktionen haben keine Bedeutung, sie erhöhen jedoch die Lesbarkeit und ermöglichen außerdem die Unterscheidung von Ketten- und Knotenproduktionen aufgrund der Darstellung.

Damit eine Grammatik für unsere Zwecke brauchbar ist, müssen weitere Forderungen erfüllt werden:

1. Zyklenfreiheit:
   $c_0 \rightarrow c_1 \rightarrow ... \rightarrow c_m$

   $c_0 \neq c_m$
   die Kettenproduktionen müssen azyklisch sein
2. Eindeutigkeit der Oberklasse:
   $c_1 \rightarrow c_0, c_2 \rightarrow c_0$

   $c_1 = c_2$
   jede Klasse besitzt höchstens eine Oberklasse:
3. Feste Wertigkeit:
   $c_0 \rightarrow n\ (c_1, ..., c_m), c_0 \rightarrow n\ (c_1, ..., c_{m'})$

   $m = m'$
   die Anzahl der Söhne ist für alle Knotenbeschreibungen eines Knotens identisch

4. Eindeutigkeit der Knotenbeschreibung innerhalb einer Klasse:
   $c_0 \rightarrow n\ (c_1, ..., c_m), c_0 \rightarrow n\ (c_1, ..., c_m)$

   $c_1 = c_1, ..., c_m = c_m,$
   ein Knoten darf in einer Klasse nur einmal beschrieben sein
5. Prinzip der Hauptbeschreibung:
   $\forall\ n \in N: \exists\ c_0 \rightarrow n\ (c_1, ..., c_m) \in \Pi:$
   $c_0 \rightarrow n\ (c_1, ..., c_m) \in \Pi$

   $c_0 \rightarrow ... \rightarrow c_0, c_1 \rightarrow ... \rightarrow c_1, ..., c_m \rightarrow ... \rightarrow c_m$
   für jeden Knoten existiert eine Beschreibung, aus der alle andern Beschreibungen des selben Knotens abgeleitet werden können.
6. Reduziertheit:
   Von der Baumgrammatik wird verlangt, daß sie reduziert ist, d.h. alle Klassen und Knoten müssen von den Startsymbolen aus erreichbar sein und alle Klassen müssen terminalisierbar sein.

Die Eigenschaften erleichtern zum einen die Darstellung und den Umgang mit der Baumgrammatik, andererseits sind sie notwendig damit die Bäume vernünftig implementiert werden können.

G = (C, N, Π, Z)

C = {expr, const, index}
N = {'+', Const, Ident}
Π = { expr → const,
expr → index,
expr → '+' (expr, expr),
const → Const (),
const → '+' (const, const),
index → Ident () }
Z = {expr}

Abb. 4.1: Baumgrammatik für arithmetische Ausdrücke

Abb. 4.1 zeigt eine Baumgrammatik für arithmetische Ausdrücke, die diese Forderungen erfüllt. Die Grammatik ist zyklenfrei (1). Die Klassen const und index besitzen beide die eindeutige Oberklasse expr (2). Beide Beschreibungen von '+' haben die Wertigkeit zwei (3) und liegen in unterschiedlichen Klassen (4). Die Produktion expr → '+' (expr, expr) legt die Hauptbeschreibung fest. Die zweite Knotenproduktion für '+' (const → '+' (const, const)) kann aus der ersteren abgeleitet werden (5). Die Grammatik ist reduziert (6).

4.2. Muster

Wenn man ausgehend von einer Klasse endlich viele Produktionen anwendet, entsteht ein Ausdruck, den wir als Muster bezeichnen. Muster werden verwendet, um die Menge aller aus ihnen ableitbaren Bäumen darzustellen.

Wenn ein Baum aus einem Muster abgeleitet werden kann, dann *paßt* das Muster auf diesen Baum.

Bei der Darstellung von Mustern ist es zulässig, Klassen wegzulassen, wenn diese aufgrund der Hauptbeschreibung ohnehin festgelegt sind. Diese Freiheit hat zur Folge, daß es verschiedene Muster gibt, die die selbe Menge von Bäumen darstellen.

Abb. 4.2 zeigt ein Beispiel. Die Doppelpunkte in den Mustern dienen als Platzhalter für die weggelassenen Klassen. Der Doppelpunkt stellt das sogenannte *allgemeine Muster*, das immer paßt, dar.

```
'+'   (:, :)          (normiert)
'+'   (expr, :)
'+'   (:, expr)
'+'   (expr, expr)
```

Abb. 4.2: verschiedene Muster zur Darstellung der selben Mengen von Bäumen

Da in der Grammatik von Abb. 4.1 *Ident* der einzige Knoten ist, der aus der Klasse *index* abgeleitet werden kann, stellen die beiden Muster von Abbildung 4.3 ebenfalls die selbe Menge von Bäumen dar.

```
index             (normiert)
Ident  ()
```

Abb. 4.3: verschiedene Muster zur Darstellung der selben Mengen von Bäumen

Der Grund für die Möglichkeit, ein Menge von Bäumen durch verschiedene Muster darzustellen, ist also zum einen die Möglichkeit, die Klasse des Sohnes nicht zu spezifizieren, zum anderen die Eigenschaft, daß ein Knoten mitunter bereits durch eine Klasse eindeutig festgelegt ist.

Um diese unterschiedliche Darstellungen zu vermeiden, wird der Begriff des *normierten Musters* eingeführt. Ein Muster heißt normiert, wenn es durch die zwei oben genannten Eigenschaften (Klasse der Söhne muß nicht spezifiziert werden, Knoten ist durch Klasse festgelegt) nicht vereinfacht werden kann.

Eine dritte Möglichkeit besteht, wenn die Oberklasse einer Klasse keine eigenen Knotenbeschreibungen enthält und für keine weitere Klasse Oberklasse ist. In diesem Fall kann aus der Oberklasse nichts abgeleitet werden, was nicht auch aus der Klasse selbst entstehen könnte. Somit sind diese beide Klassen bei der Darstellung eines Musters vollkommen austauschbar.

In der Praxis wird diese Möglichkeit in der Regel nicht auftreten, da man normalerweise versuchen wird, mit einer möglichst einfachen Grammatik auszukommen und deshalb das Auftreten äquivalenter Klassen vermieden wird.

4.3. Beziehungen zwischen Mustern

Um die Beziehungen zwischen zwei Mustern beschreiben zu können, werden die folgenden Begriffe und Notationen verwendet.

1. $p_1 = p_2$
   Zwei Muster sind *gleich*, falls gilt, daß entweder beide Muster passen oder keines der Muster paßt.
2. $p_1 < p_2$
   Ein Muster p_1 ist *kleiner* als p_2, wenn das Passen von p_1 aus dem Passen von p_2 folgt und die beiden Muster nicht gleich sind.
3. $p_1 > p_2$
   Ein Muster p_1 ist *größer* als p_2, wenn p_2 kleiner als p_1 ist.
4. $p_1 \parallel p_2$
   Ein Muster p_1 ist *inkonsistent* zu p_2, wenn es keinen Baum gibt, auf den sowohl p_1 als auch p_2 paßt.
5. $p_1 \sim p_2$
   Ein Muster p_1 ist *unabhängig* von p_2, wenn p_1 und p_2 unabhängig voneinander passen können.

Die Begriffe sind [HoO82] entnommen. Dort werden jedoch keine unterschiedlichen Klassen betrachtet, wie es hier geschieht. Statt dessen gibt es lediglich ein Symbol, das für beliebige Bäume steht.

Die Einführung von Klassen hat zur Folge, daß es aufwendiger wird zu berechnen, welche Beziehung zwischen zwei Mustern gilt.

Betrachten wir den Fall, daß es sich bei den beiden Mustern jeweils nur um eine Klasse handelt. Die Frage nach der Beziehung zwischen den beiden Mustern ist dann gleichbedeutend zu entscheiden, ob die Sprachen, die durch zwei unterschiedliche Baumgrammatiken definiert werden, identisch sind (gleich), ob eine Sprache eine Untermenge der anderen ist (kleiner/größer), ob der Schnitt der beiden Sprachen leer ist (inkonsistent) oder ob es sich um Quermengen handelt (unabhängig).

In der Praxis ist es jedoch möglich, mit einer Näherungslösung auszukommen. Diese Näherungslösung entsteht, indem in einzelnen Fällen in Kauf genommen wird, daß zwei Muster als unabhängig betrachtet werden, obwohl eine der anderen Beziehungen gilt. Dieser Fehler kann in Kauf genommen werden, da er nur in pathologischen Fällen eintritt und keine Folgefehler verursacht.

Im Algorithmus zur Berechnung der Beziehung zwischen zwei (normierten) Mustern (Abb. 4.5) werden folgende Hilfsfunktionen verwendet (Abb. 4.4).

```
Ident          (pat: tPattern): tIdent      (* liefert den Bezeichner der Wurzel von pat      *)
Type           (id: tIdent): tType          (* liefert den Typ (class oder node) von id       *)
Arity          (node: tIdent): INTEGER      (* liefert die Wertigkeit des Knotens             *)
Classes        (pat: tPattern): tSet        (* liefert die Klassen zu denen pat gehört        *)
Subclasses     (class: tIdent): tSet        (* liefert die Unterklassen von class             *)
ClassesOfNode  (node: tIdent): tSet         (* liefert alle Klassen in denen node beschrieben ist *)
NodesOfClass   (class: tIdent): tSet        (* liefert alle Knoten die aus class hervorgehen können *)
```

Abb. 4.4: Hilfsfunktionen zur Berechnung der Beziehung zwischen zwei Mustern

---

```
Relation (pat1, pat2: tPattern): tRelation;

if (pat1 = NoPat) & (pat2 = NoPat) then return equal      (* pat1 = pat2      *)
if (pat1 = NoPat) then return less                        (* pat1 < pat2      *)
if (pat2 = NoPat) then return greater                     (* pat1 > pat2      *)

id1 := Ident (pat1)
type1 := Type (id1)
id2 := Ident (pat2)
type2 := Type (id2)

if (type1 = class) & (type2 = class) then                 (* A *)
  if (id1 = id2) then return equal                        (* pat1 = pat2      *)
  elsif (id1 ∈ Classes (pat2)) then return less           (* pat1 < pat2      *)
  elsif (id2 ∈ Classes (pat1)) then return greater        (* pat1 > pat2      *)
  if NodesOfClass (id1) ∩ NodesOfClass (id2) = ∅ then
    return inconsistent                                   (* pat1 ~ pat2      *)
  else return independent                                 (* pat1 || pat2     *)

elsif (type1 = class) then                                (* B *)
  if (id1 ∈ Classes (pat2)) then return less              (* pat1 < pat2      *)
  else
    common := (Subclasses (id1)∪{id1}) ∩ ClassesOfNode (id2)
    if common = ∅ then return inconsistent                (* pat1 || pat2     *)
    else return independent                               (* pat1 ~ pat2      *)

elsif (type2 = class) then                                (* C *)
  if (id2 ∈ Classes (pat1)) then return greater           (* pat1 > pat2      *)
  else
    common := (Subclasses (id2)∪{id2}) ∩ ClassesOfNode (id1)
    if common = ∅ then return inconsistent                (* pat1 || pat2     *)
    else return independent                               (* pat1 ~ pat2      *)
```

```
else                                                    (* D *)
 if id1 = id2 then
  relation := equal;                                    (* up to now: pat1 = pat2    *)
  for pos := 1 to Arity (id1) do
   case Relation (Son (pat1, pos), Son (pat2, pos)) of
   | equal:
   | independent:
       relation := independent                          (* now: pat1 ~ pat2          *)
   | inconsistent:
      return inconsistent                               (* pat1 || pat2              *)
   | greater:
      if relation = equal
       then relation := greater                         (* now: pat1 > pat2          *)
      elsif relation = less
       then relation := independent                     (* now: pat1 ~ pat2          *)
   | less:
      if relation = equal
       then relation := less                            (* now: pat1 < pat2          *)
      elsif relation = greater
       then relation := independent                     (* now: pat1 ~ pat2          *)
  return relation
 else return inconsistent                               (* pat1 || pat2              *)
```

Abb. 4.5: Algorithmus zur Berechnung der Beziehung von zwei Mustern

Zu Beginn des Algorithmus Relation (Abb. 4.5) werden die einfachen Fälle behandelt, die entstehen, wenn eines der Muster oder beide Muster leer sind (NoPat). Falls hier keine Entscheidung fällt, ist sichergestellt, daß beide Muster eine Klasse oder einen Knoten als Wurzel enthalten.

Handelt es sich um zwei Klassen (A), so gilt Gleichheit, falls die Klassen identisch sind. Trifft dies nicht zu wird geprüft, ob ein Muster aus der Klasse des anderen abgeleitet werden kann und somit kleiner als dieses ist. Zuletzt wird noch geprüft ob es Knoten gibt, die aus beiden Klassen hervorgehen können, die beiden Klassen werden dann als unabhängig bezeichnet. Bei dieser Festlegung handelt es sich um einen Näherung, denn tatsächlich kann hier (in pathologischen Fällen) auch jede andere Beziehung gelten. Im übrigen sind die Klassen und damit die Muster mit Sicherheit inkonsistent.

Bei den beiden nächsten Fällen (B, C) liegen jeweils eine Klasse und ein Knoten als Wurzel vor. Kann das Muster mit dem Knoten als Wurzel aus der Klasse abgeleitet werden, steht die Beziehung fest. Im anderen Falle wird geprüft ob es überhaupt Muster gibt, die aus der Klasse abgeleitet werden können und den Knoten als Wurzel haben, denn dann werden die Muster als unabhängig betrachtet. Ansonsten sind die beiden Muster inkonsistent.

Falls beide Muster einen Knoten als Wurzel haben (D), gibt es zwei Möglichkeiten. Sind die beiden Knoten identisch, müssen die Söhne betrachtet werden. Hierzu wird ein Hilfsvariable (relation) auf gleich (equal) initialisiert. Diese Hilfsvariable stellt die Beziehung der bereits betrachteten Teile der beiden Muster dar. In Abhängigkeit von den Beziehungen der Söhne wird der Wert von relation gegebenenfalls angepaßt, bis spätestens nach Betrachten aller Söhne das Ergebnis feststeht. Sind die beiden Knoten hingegen verschieden, steht sofort Inkonsistenz fest.

5. Spezifikation von Transformationen mit ESTRAL

Im folgenden wird die Eingabesprache zur Spezifikation von Transformationen beschrieben.

5.1. Reservierte Wörter und Symbole

Die folgende Liste enthält die reservierten Wörter in ESTRAL:

BEGIN, CLOSE, CONDITION, COSTS, DECLARE, EXPORT,
FUNCTION, GLOBAL, GRAMMAR, TRANSFORMATION

Die Symbole

( ) { } , . / : ; = | –>

werden als Operatoren und Begrenzer verwendet.

5.2. Bezeichner

Bezeichner (identifiers) sind Folgen aus Buchstaben (letters), Ziffern (digits) und dem Unterstreichungsstrich (underscore). Ein Bezeichner muß immer mit einem Buchstaben beginnen. Wenn reservierte Wörter als Bezeichner verwendet werden, sind sie mit einem '\' als Fluchtsymbol zu maskieren. Dieses Fluchtsymbol, das auch vor jedem anderen Bezeichner stehen darf, ist jedoch nicht Bestandteil des Bezeichners, es dient lediglich zur Unterscheidung von Schlüsselwörtern und Bezeichnern.

Abb. 5.1 zeigt einen regulären Ausdruck, der den Aufbau eines Bezeichners beschreibt.

---

ident = ['\'] letter (letter | digit | '_')*

Abb. 5.1: regulärer Ausdruck für Bezeichner

Beispiele für korrekte Bezeichner sind:

Bezeichner, Doppel_Wort, \STOP, END, \BEGIN, A2, b4

Die folgende Zeichenfolgen sind hingegen keine gültigen Bezeichner:

BEGIN	*(Schlüsselwörter sind nicht erlaubt)*
\\END	*(nur ein '\' ist erlaubt)*
4A	*(Ziffer darf nicht am Anfang stehen)*
A-Z	*(Bindestrich ist nicht erlaubt)*

Bei der Verwendung von Bezeichnern ist außerdem zu beachten, daß sie vom Generator verwendet werden, um Bezeichner für das Zielprogramm zu bilden. Einschränkungen, die in der Zielsprache für Bezeichner gelten, sollten deshalb unbedingt auch eingehalten werden.

Beispiele für solche Einschränkungen sind das Verbot von '_' in Bezeichnern der Programmiersprache Modula-2 [Wir85] oder die Beschränkung der signifikanten Länge von Bezeichnern in C [KeR83].

5.3. Zeichenketten

Zeichenketten (strings) sind in Anführungszeichen ('"') oder Apostrophe ("'") eingeschlossene Zeichenfolgen. Die Zeichenfolge darf alle Zeichen außer dem Zeilenende und dem Begrenzer selbst enthalten.

5.4. Zahlen

Da nur positive ganze Zahlen benötigt werden, genügt es, Zahlen (numbers) als Folgen von Ziffern aufzufassen.

5.5. Kommentare

Kommentare können in den von Modula-2 und C gewohnten Formen geschrieben werden, so daß es möglich ist, Kommentare innerhalb und außerhalb von Quelltexten in der selben Weise zu schreiben.

5.6. Quelltext

Zur Darstellung von Bedingungen, Kosten, Vereinbarungen und Anweisungen wird Quelltext (source text) der Zielsprache verwendet. Der Quelltext wird in geschweifte Klammern eingeschlossen. Da dieser Text nicht immer direkt übernommen werden kann, sondern zum Teil umgesetzt werden muß, sind einige Regeln zu beachten, die es dem Generator erlauben, den Quelltext zu analysieren.

Kommentare, Zeichenketten und Bezeichner, die im Quelltext stehen, werden als Einheiten betrachtet, die nicht weiter untergliedert werden. Geschweifte Klammern dürfen innerhalb des Quelltextes nur paarweise auftreten, damit das Ende des Quelltextes erkennbar ist. Klammern innerhalb von Kommentaren und Zeichenketten werden hierbei natürlich nicht mitgezählt.

Obwohl diese Regeln so gewählt wurden, daß sie beim Schreiben von Quelltexten möglichst wenig einschränken, gibt es einige Fehlerquellen, auf die man achten sollte.

So darf folgendes korrekte C-Programmfragment keinesfalls in dieser Form geschrieben werden.

```
{ i = 3 * (*p + i); }
```

Die Folge wäre, daß ein (nicht geschlossener) Modula-2-Kommentar erkannt würde. Doch kann dieses Problem leicht beseitigt werden, indem man ein Leerzeichen zur Trennung der Klammer und des Adreßoperators benutzt.

```
{ i = 3 * ( *p + i) }
```

Nun kann nicht mehr fälschlich ein Kommentar erkannt werden.

Weitere Probleme können bei Zeichenketten in C entstehen, da sich die Festlegung der Schreibweise von Zeichenketten an den Konventionen von Modula-2 orientierte.

```
{ c = '\''; }
{ printf ("\""; }
```

Solche Eingaben führen zu Fehlermeldungen, da jeweils eine nicht geschlossene Zeichenkette erkannt wird.

5.7. Transformation

Die Beschreibung einer Transformation (Abb. 5.2) besteht aus einer *Baumgrammatik* (grammar), zur Beschreibung der Struktur der zu transformierenden Bäume und mehreren *Funktionen* (function), die die Abbildung beschreiben. Die *Integration* des generierten Programms in eine Umgebung wird in einem weiteren Abschnitt (integration) beschrieben.

transformation	=	**TRANSFORMATION**	
		ident	*Name der Transformation*
		integration	*Integration*
		grammar	*Baumgrammatik*
		function *	*Funktionen*

Abb. 5.2: Transformation

Um mehrere Transformationen unterscheiden zu können, wird jeder Transformation ein *Name* zugeordnet.

5.8. Baumgrammatik

Die Baumgrammatik wird benutzt, um die Struktur der zulässigen Bäume zu beschreiben und bildet die Grundlage für alle Konsistenzprüfungen (vgl. 4.1.) denen die Eingabe unterzogen wird.

grammar	=	**GRAMMAR**	
		ident	*Name der Grammatik*
		class *	*Beschreibung der Klassen*

Abb. 5.3: Baumgrammatik

Der *Name der Grammatik* wird in der Implementierung benutzt, um das Modul, das die Bäume realisiert, zu bezeichnen. Die eigentliche Grammatik wird mit Hilfe der *Klassen* beschrieben.

5.9. Klassen

Klassen werden durch einen Bezeichner, dem ein Gleichheitszeichen folgt, definiert. Zusammen mit den Klassen werden auch die Produktionen der Baumgrammatik festgelegt. Die Kettenproduktionen können aufgrund des Oberklassenprinzips (vgl. Kapitel 4) durch die optionale Angabe einer Oberklasse beschrieben werden. Die Knotenproduktionen werden beschrieben, indem alle Knotenbeschreibungen bei der zugehörigen Klasse aufgezählt werden.

class	=	[ident '–>']	*Bezeichner der Oberklasse*
		ident '='	*Bezeichner der Klasse*
		node *	*Knotenbeschreibungen*

Abb. 5.4: Klassen

5.10. Knoten

Die Knotenbeschreibung ordnet dem Knoten einen *Namen* in Form einer Zeichenkette oder eines Bezeichners zu und zählt seine Söhne auf.

---

node	=	'|' (string | ident)	*Name des Knotens*
		[':' ident]	*Bezeichner des Knotens (siehe Hauptbeschreibung)*
		'(' [son || ','] ')'	*Söhne*

Abb. 5.5: Knoten

Beispiele:

| '+'	: Plus	(e1: expr, e2: expr)
| ':='	: Asgn	(index, expr)
| While		(expr, then: stats, else: stats)
| Const		()

5.11. Söhne

Um den Sohn eines Knotens festzulegen, wird seine *Klasse* angegeben.

---

son	=	[ident ':']	*Name des Sohnes (nur in der Hauptbeschreibung)*
		ident	*Klasse des Sohnes*

Abb. 5.6: Söhne

Beispiele:

expr
e1: expr
const

Inhaltsverzeichnis

1. Einleitung 4

2. Möglichkeiten zur Beschreibung von Transformationen 5
2.1. Attributierte Grammatiken 5
2.2. Modifikation der Eingabe 5
2.3. Funktionale Abbildung 6

3. Anforderungen an eine funktionale Beschreibung 7
3.1. Abbildung der einzelnen Knoten 7
3.2. Steuerung der Reihenfolge 8
3.3. Zugriff auf die Attribute 8
3.4. Berechnung eines synthetisierten Attributs 9
3.5. Muster 11
3.6. Mehrdeutigkeit und Kosten 11
3.7. Bedingungen 12
3.8. Mehrfachtransformation 13
3.9. Synthetisierte und vererbte Attribute 14

4. Begriffe 17
4.1. Baumgrammatik 17
4.2. Muster 18
4.3. Beziehungen zwischen Mustern 19

5. Spezifikation von Transformationen mit ESTRAL 23
5.1. Reservierte Wörter und Symbole 23
5.2. Bezeichner 23
5.3. Zeichenketten 23
5.4. Zahlen 23
5.5. Kommentare 24
5.6. Quelltext 24
5.7. Transformation 24
5.8. Baumgrammatik 25
5.9. Klassen 25
5.10. Knoten 26
5.11. Söhne 26

5.12. Hauptbeschreibung

Aufgrund des in Kapitel 4.1. geforderten Prinzips der Hauptbeschreibung, muß für jeden Knoten eine Hauptbeschreibung existieren. Diese Hauptbeschreibung hat neben ihrer Funktion als Knotenproduktion der Baumgrammatik die Aufgabe, die Implementierung des Knotens zu beschreiben. In der Hauptbeschreibung eines Knotens werden deshalb der *Bezeichner des Knotens* (Abb. 5.5) und die *Namen der Klassen* (Abb. 5.6) festgelegt. Diese werden in der Implementierung verwendet, um auf den Knoten, dessen Attribute und Söhne zuzugreifen und die Art des Knotens festzustellen (vgl. Kapitel 9.1.). Falls keine explizite Festlegung erfolgt, wird der Bezeichner des Knotens mit seinem Namen und die Namen der Söhne mit den Klassen gleichgesetzt. Klassen gleichgesetzt.

5.13. Funktionen

Die Funktionen beschreiben die Abbildung, die durch die Transformation realisiert werden soll.

Die Beschreibung einer *Funktion* (function) (Abb. 5.7) umfaßt den *Namen* (ident) der Funktion, die Beschreibung der *vererbten* und *synthetisierten Attribute* (attributes), die Festlegung des *Ergebnistyps* und des *Definitionsbereichs*, sowie *Vorschriften* (directive) zur Durchführung der Transformation.

Function	=	**FUNCTION**	
		ident	*Name der Funktion*
		[attributes '–>' attributes]	*vererbte und synthetisierte Attribute*
		[':' type]	*Ergebnistyp*
		domain	*Definitionsbereich*
		directive *	*Vorschriften zur Beschreibung der Funktion*

Abb. 5.7: Funktionen

Die Transformation wird durchgeführt, indem die erste Funktion auf den zu transformierenden Baum angewandt wird. Der Definitionsbereich der Transformation kann deshalb mit dem Definitionsbereich der ersten Funktion gleichgesetzt werden.

Die Vollständigkeit wird in der in Kapitel 8 beschriebenen Weise überprüft.

5.14. Typen

Typen (Abb. 5.8) werden durch einen Bezeichner wie z.B.

INTEGER, REAL, BOOLEAN

beschrieben.

Um einen qualifizierten Import, wie er in Modula-2 möglich ist, zu beschreiben, kann ein weitere Bezeichner verwendet werden, um das Modul festzulegen. Damit sind auch Angaben der Form

SYSTEM.TSIZE, Sets.tSet, IO.WriteS

möglich.

type	=	[ident '.']	*Angabe des Moduls zur Qualifikation*
		ident	*Bezeichner des Typs*

Abb. 5.8: Typen

5.15. Attribute

Die Angabe der Attribute ist optional. Einzelne Attribute werden durch einen Bezeichner und einen Typ beschrieben. Um mehrere Attribute des selben Typs zu beschreiben, werden die Bezeichner, durch Komma getrennt, aufgezählt. Wenn Attribute mit unterschiedlichen Typen beschrieben werden, sind die einzelnen Beschreibungen durch Strichpunkte zu trennen.

attributes	=	[((ident ‖ ','	*Liste von Bezeichnern*
		':' type)	*Angabe des Typs*
		‖ ';')]	*mehrere solche Listen durch ';' getrennt*

Abb. 5.9: Attribute

Beispiel:

A: INTEGER; B,C: REAL

5.16. Definitionbereich

Der *Definitionsbereich* (domain) einer Funktion wird festgelegt, indem die *Klassen* (ident), für die die Funktion definiert ist, aufgezählt werden.

domain	=	'/' ident ‖ ',' '/'	*Liste von Klassen*

Abb. 5.10: Definitionsbereich

Beispiel:

/ stats, expr /

Der Definitionsbereich der ersten Funktion legt außerdem die Startsymbole fest und bildet somit die Basis für die Überprüfung der Reduziertheit der Grammatik (vgl. 4.1.).

5.17. Vorschriften

Die *Vorschriften* (directive), die die Abbildungen im einzelnen beschreiben, bestehen im einfachsten Falle aus einem *Muster* (pattern), das festlegt, auf welche (Teil-) Bäume die Vorschrift angewandt werden kann und *Anweisungen* (instructions), die festlegen, wie der betreffende (Teil-) Baum behandelt werden muß.

```
directive   =   pattern              Muster
                [ condition ]        Bedingung
                [ costs ]            Kosten
                [ declarations ]     lokale Vereinbarungen
                instructions         Anweisungen
```

Abb. 5.11: Vorschriften

5.18. Muster

Das Muster beschreibt einen Ausschnitt des Baumes, der in den Anweisungen bearbeitet wird. Mit Hilfe der Selektoren kann in den Vorschriften auf die entsprechenden Knoten des Strukturbaumes zugegriffen werden. Die Selektoren werden, falls sie nicht explizit angegeben sind, aus den Namen der Knoten bzw. Klassen abgeleitet. Falls es hierbei zu Mehrdeutigkeit kommt, liegt ein Fehler vor. Die automatische Erzeugung eines Selektors wird unterdrückt, wenn ein Doppelpunkt angegeben wird, dem kein Bezeichner vorangestellt ist. Wenn auf spezielle Knoten in den Anweisungen nicht zugegriffen werden muß, können durch die Unterdrückung Mehrdeutigkeiten vermieden werden.

```
pattern   =   [ [ ident ] ':' ]          Selektor
                (ident | string)         Name des Knotens
                '(' [ pattern || ',' ] ')'   Söhne
          |   [ ident ] ':'              Selektor
                [ ident ]                Bezeichner der Klasse
          |   ident                      Selektor und Bezeichner der Klasse
```

Abb. 5.12: Muster

Beispiele:

```
'+' (e1:, e2:)
':=' (index, expr)
If (expr, then:, else:)
:'+' (:'+' (e1: const, e2: const), e3:)
expr
const
```

5.19. Anweisungen

```
instructions   =   '{' source_text '}'
```

Abb. 5.13: Anweisungen

Die *Anweisungen* (instructions) sind der Kern jeder Vorschrift, sie werden vom Transformator ausgeführt, wenn die Vorschrift angewandt wird.

Zum Zugriff auf den durch das Muster beschriebenen Teil des Baumes können die Selektoren des Musters verwendet werden, wobei zwei Fälle zu unterscheiden sind. Folgt dem Selektor ein Punkt, wird ein Zugriff auf ein Attribut angenommen und der Selektor wird automatisch durch den Zugriff auf den entsprechenden Knoten im Baum ersetzt. Folgt kein Punkt, wird der Selektor durch einen Zeiger auf den Baum ersetzt.

Bei Funktionsaufrufen innerhalb der Vorschrift sollte das erste Argument, das den zu transformierenden Baum beschreibt, normalerweise eine Selektor sein, da nur dann vom Generator geprüft werden kann, ob das Argument im Definitionsbereich liegt. Da es jedoch in speziellen Fällen zweckmäßig ist, einen Baum zu transformieren, der durch ein vererbtes Attribut beschrieben wird, wird es auch akzeptiert, wenn der zu transformierende Baum auf andere Weise spezifiziert wird.

5.20. Bedingungen

Um die Anwendbarkeit einer Vorschrift einzuschränken, kann eine *Bedingung* (condition) an die Attribute gestellt werden. Eine Bedingung wird durch eine booleschen Ausdruck in Form von Quelltext beschrieben.

Selektoren können in den Bedingungen ebenfalls verwendet werden.

Der Aufruf von Funktionen und Prozeduren ist in Bedingungen prinzipiell möglich, die verwendeten Funktionen sollten jedoch keine Seiteneffekte haben, da die Reihenfolge, in der die Bedingungen getestet werden nicht festgelegt ist. Außerdem darf keine Funktion für die Wurzel des Musters aufgerufen werden, da die Vorschrift die hierzu benötigt wird, von eben dieser Bedingung abhängen könnte.

---

condition = **CONDITION**
'{' source_text '}' *Ausdruck zur Berechnung der Bedingung*

Abb. 5.14: Bedingungen

Beispiel:

```
'*' (expr, const)        CONDITION { IsPowerOf2 (const.value) }
                         {
                         ...
                         }
```

5.21. Kosten

Um die *Kosten*, die bei Anwendung eine Vorschrift entstehen, zu beschreiben, gibt es zwei Möglichkeiten. Die Kosten werden entweder durch eine Konstante (number) festgelegt oder es wird ein Ausdruck (source text) angegeben, die die Kosten für den gesamten durch das Muster abgedeckten Baum berechnen. Im ersten Fall werden die Kosten, die durch Funktionsaufrufe innerhalb der Vorschriften verursacht werden, automatisch mit einfacher Gewichtung berücksichtigt. Im zweiten Fall ist dies Aufgabe des Anwenders.

```
costs   =   COSTS
              (number                 Beschreibung der Kosten durch eine Konstante
              | '{' source_text '}')  Ausdruck zur Berechnung der Kosten
```

Abb. 5.15: Kosten

Um diese Kosten zu berücksichtigen, stehen dem Anwender Prozeduren zur Verfügung, die die Kosten für eine bestimmte Funktion liefern. Die Namen dieser Prozeduren setzen sich aus dem Wort 'Cost' und dem Name der betreffenden Funktion zusammen. Als einziges Argument wird der Selektor des betreffenden Teilbaums erwartet.

Beispiel:

```
'+' (e1: expr, e2:expr)     COSTS { 1 + 2 * CostF (e1) + CostF (e2) }
                                    {
                                    ... F (e1); ... F (e2); ...
                                    }
```

Falls die Kosten nicht explizit angegeben werden, wird *COSTS 1* angenommen.

5.22. Vereinbarungen

Die *Vereinbarungen* (declarations) sind lokal d.h. nur für die Anweisungen einer Vorschrift sichtbar. Sie können z.B. benutzt werden, um Variablen zu vereinbaren, die nur in den Anweisungen einer Vorschrift benötigt werden.

```
declarations   =   DECLARE '{' source_text '}'
```

Abb. 5.16: Vereinbarungen

Beispiel:

```
DECLARE     {
VAR I,J: INTEGER;
}
```

5.23. Integration

Die optionalen EXPORT-, LOCAL-, BEGIN- und CLOSE-Abschnitte werden verwendet, um das generierte Programm in die Umgebung zu integrieren. Der Quelltext im EXPORT-Abschnitt dient zur Vereinbarung globaler Daten, die auch außerhalb des generierten Programms sichtbar sein sollen. Der GLOBAL-Abschnitt enthält von außen nicht sichtbare globale Vereinbarungen. Die Anweisungen im BEGIN-Abschnitt werden vor der Transformation ausgeführt, die im CLOSE-Abschnitt danach. Der Zweck der beiden letzten Abschnitte besteht darin, globale Daten zu initialisieren, bevor eine Transformation durchgeführt wird, sowie Abschlußarbeiten (z.B. Speicherbereinigung oder Schließen von Dateien) durchzuführen, nachdem die Transformation abgeschlossen ist.

integration	=	[**EXPORT** '{' source_text '}']	*öffentliche Vereinbarungen*
		[**GLOBAL** '{' source_text '}']	*globale Vereinbarungen*
		[**BEGIN** '{' source_text '}']	*Anweisungen zur Initialisierung*
		[**CLOSE** '{' source_text '}']	*Anweisungen zum Abschluß*

Abb. 5.17: Integration

6. Implementierung von Transformationen durch ESTRA

Im folgenden werden die Struktur von mit ESTRA erzeugten Transformatoren und deren wesentliche Eigenschaften beschrieben. Es kann jedoch nicht Aufgabe diese Kapitels sein, sämtliche Details der erzeugten Implementierungen zu besprechen. Diese Details kann der interessierte Leser im Anhang B.2 finden. Das Beispiel im Anhang ist überdies geeignet, die folgenden Ausführungen zu ergänzen.

Mit ESTRA generierte Transformatoren führen die Transformation in zwei Schritten durch. Im ersten Schritt (Vorbereitung) wird festgelegt, welche Vorschriften zur Transformation des speziellen Baumes im einzelnen anzuwenden sind. Im zweiten Schritt (Durchführung) werden diese Vorschriften angewandt.

Zur Vorbereitung der Transformation werden in jedem Knoten des Baumes die Vorschriften mit den geringsten Kosten für die Durchführung einer Funktion festgehalten. Um diese Vorschriften zu ermitteln, ist es notwendig, für jede Vorschrift zu prüfen, ob das Muster auf den Knoten paßt und die Bedingung (sofern vorhanden) erfüllt ist. Dann muß für jede Funktion von den verbliebenen Vorschriften diejenige mit den geringsten Kosten im Knoten festgehalten werden.

Da die Kosten für die Anwendung einer Vorschrift im allgemeinen von den Kosten der Söhne (bzw. 'Nachkommen') des Knotens abhängig sind, muß diese Berechnung in einem Bottom-Up-Durchlauf durch den Baum erfolgen.

Die Durchführung der Transformation erfolgt durch Anwendung der ersten Funktion auf die Wurzel, d.h. die Vorschrift, die in der Wurzel für diese Funktion festgehalten wurde, wird ausgeführt. Funktionsaufrufe für Teilbäume, die bei der Abarbeitung einer Vorschrift anfallen, werden rekursiv abgearbeitet.

6.1. Vorbereitung der Transformation

Bevor die Transformation in der obengenannten Weise durchgeführt werden kann, muß sie durch die Festlegung der Lösung mit minimalen Kosten vorbereitet werden.

Die Vorbereitung entspricht einer S-Attributierung, bei der für jeden Knoten die folgenden Attribute berechnet werden:

1. die Menge der Klassen, zu denen der Knoten gehört
2. die Vorschriften, die auf den Knoten (genauer dessen Teilbaum) anwendbar sind
3. für jede Funktion die Vorschrift, die diese mit den geringsten Kosten realisiert und die hierbei entstehenden Kosten

Die rekursive Prozedur yyTraverse, die diese Attribute berechnet, hat folgenden Aufbau:

1. Beschaffung von Speicher für die Attribute
2. Prozeduraufrufe zur Berechnung der Attribute der Söhne des Knotens
3. Berechnung der Klassen des aktuellen Teilbaums
4. Berechnung der zulässigen Vorschriften
5. Berechnung der kostengünstigsten Vorschrift für die einzelnen Funktionen

6.2. Darstellung von Funktionen und Vorschriften

In Modula-2 [Wir85] werden sowohl die Funktionen als auch die Vorschriften (genauer die Vereinbarungen und Anweisungen der Vorschriften) als Prozeduren realisiert. Die Prozeduren, die den Funktionen entsprechen, wählen hierbei lediglich die Vorschrift aus und rufen die entsprechende Prozedur auf. Das bedeutet, daß bei der Durchführung einer Transformation für jede Anwendung einer Vorschrift zwei Prozeduraufrufe notwendig sind. Besser wäre es, wenn man mit je einem Aufruf auskäme.

Um dies zu erreichen gibt es zwei Möglichkeiten. Entweder man spart die Prozeduren der Funktionen, d.h. die Auswahl der Vorschrift wird an den Ort des Funktionsaufrufs verschoben, oder man spart die Prozeduren für die Vorschriften, d.h. alle Vorschriften einer Funktion werden in die Prozedur der Funktion eingebettet.

Die Information, welche Vorschriften im einzelnen anzuwenden sind, wird nicht unmittelbar in den Knoten gespeichert. Es ist vielmehr so, daß dort lediglich eine Adresse zu finden ist, die einen Verweis auf diese Information darstellt (vgl. Kapitel 9). Infolgedessen ist vor dem Zugriff auf diese Information eine Typwandlung, die die Adresse in den erforderlichen Zeigertyp wandelt, notwendig.

Um aber in Modula-2 eine Adresse in einen Zeiger umzuwandeln und auf die zugehörige Information über diesen Zeiger zuzugreifen, ist eine Zuweisung erforderlich. Dies bedeutet, daß beim Verschieben der Auswahl an den Ort des Funktionsaufrufs für jeden Aufruf eine zusätzliche Anweisung, eben diese Zuweisung, erforderlich ist. Da der Aufwand, diese Anweisung in die vom Anwender vorgegebenen Anweisungen einzufügen, recht groß wäre, wurde diese Möglichkeit verworfen.

Die zweite Möglichkeit, die darin besteht, alle Vorschriften einer Funktion in eine Prozedur zu packen, scheitert in Modula-2 daran, daß die lokalen Vereinbarungen, die zu den einzelnen Vorschriften gehören, in der Regel nicht in einer Prozedur untergebracht werden können, da es sonst zu Namenskonflikten zwischen diesen Vereinbarungen kommt.

In der Programmiersprache C [KeR83] sind hingegen beide Lösung realisierbar, da weder die Typwandlung (type cast) noch die Vereinbarung von lokalen Daten (innerhalb von Blöcken) Probleme darstellen. Da bei der zweiten Lösung insgesamt weniger Funktionen (die Unterprogramme in C werden als Funktionen bezeichnet) benötigt werden und bei dieser Lösung außerdem weniger Aufwand beim Umsetzen der Anweisungen der einzelnen Vorschriften erforderlich ist, ist diese bei der Verwendung von C vorzuziehen.

6.3. Darstellung der Attribute

Die Klassen des aktuellen Teilbaums werden als Menge dargestellt und in Modula-2 als ARRAY OF BITSET realisiert. Zur Darstellung der einzelnen Klassen werden diese durchnumeriert. Die zulässigen Vorschriften werden durch ein boolesches Feld realisiert. Die kostengünstigste Vorschrift jeder Funktion wird durch den Wert einer Prozedur-Variablen und die zugehörigen Kosten durch einen INTEGER-Wert dargestellt.

Mit Ausnahme der zulässigen Vorschriften werden alle Attribute im Knoten abgespeichert. Bei den zulässigen Vorschriften ist dies nicht notwendig, da diese nur solange benötigt werden, bis die kostengünstigsten Vorschriften feststehen. Lokale Variablen der Prozedur yyTraverse würden deshalb zu Realisierung vollkommen ausreichen.

Da aber außerdem der Zeitraum von Berechnung bis Auswertung der zulässigen Vorschriften für verschiedene Knoten völlig disjunkt ist, genügt sogar ein einziges globales Feld zur Darstellung.

6.4. Speicherverwaltung

Da Größe und Struktur der Daten, die zur Vorbereitung der Transformation benötigt werden, bei der Implementierung des Baumes i.a. nicht bekannt sind, wird im Baum lediglich Platz für eine Adresse reserviert. Bei der Vorbereitung der Transformation wird dann für jeden Knoten dynamischer Speicher beschafft, um die Attribute abzuspeichern. Der Zeiger auf den dynamischen Speicher wird im Knoten abgelegt, sodaß über die Knoten jederzeit auf die Attribute zugegriffen werden kann.

Da diese Attribute nach Durchführung der Transformation nicht mehr benötigt werden, sollte der belegte Speicher wieder freigegeben werden. Um dies effizient durchzuführen, enthält der

generierte Transformator eine eigene Speicherverwaltung, die nach dem Haldenprinzip arbeitet. Die Speicherverwaltung beschafft den Speicher in größeren Einheiten vom System und vergibt ihn im benötigten Umfang. Nach der Transformation wird der Speicher schließlich in den großen Einheiten, die bereits bei der Beschaffung linear verkettet wurden, wieder freigegeben.

Durch diese Methode wird zweierlei erreicht. Zum einen kann die Speicherfreigabe sehr schnell durchgeführt werden, da kein aufwendiger Durchlauf durch den Baum notwendig ist, wie er entstünde, wenn der vergebene Speicher nur über die Knoten zu erreichen wäre. Außerdem wird vermieden, daß der Speicher durch die Verwendung in kleine, kaum wiederverwendbare Stücke zerfällt.

6.5. Berechnung der Klassen

Grundlage zur Berechnung der Klassen des aktuellen Teilbaums sind die Knotenbeschreibungen. Da die Berechnung abhängig vom aktuellen Knoten durchgeführt wird, genügt es hierbei, die Knotenbeschreibungen zu betrachten, die zum aktuellen Knoten passen.

Falls alle Söhne eines Knotens zu den jeweiligen Klassen aus der Knotenbeschreibung passen, paßt auch der aktuelle Teilbaum zu dieser Knotenbeschreibung. Folglich paßt auch die Klasse auf der linken Seite der zugehörigen Knotenproduktion.

Da mit einer Klasse immer auch deren Oberklasse auf einen Teilbaum paßt, wird hier nicht nur diese Klasse, sondern auch deren transitive Hülle bezüglich der Oberklassenrelation erkannt.

Da die Hauptbeschreibung eines Knotens immer auf diesen Knoten passen muß, wird deren transitive Hülle immer in die Klassen des aktuellen Teilbaumes aufgenommen. Beim Erkennen einer Knotenbeschreibung können diese deshalb vernachlässigt werden, da sie ja ohnehin erkannt werden.

6.6. Berechnung der zulässigen Vorschriften

Um die zulässigen Vorschriften festzulegen, werden die Muster der Vorschriften, mit dem Teilbaum verglichen. Zu diesem Zweck werden alle Muster betrachtet, die mit dem aktuellen Knoten verträglich sind. Es werden als nur solche Muster untersucht, die mit dem aktuellen Knoten beginnen, oder einer Klasse darstellen, aus der der aktuelle Knoten ableitbar ist.

Um festzustellen, ob ein Muster mit dem aktuellen Teilbaum übereinstimmt, wird jeder Knoten und jede Klasse des Muster mit dem aktuellen Teilbaum verglichen. Falls es sich bei der Wurzel eines Musters um einen Knoten handelt, muß dieser nicht überprüft werden, da seine Übereinstimmung bereits durch die Auswahl der Muster gegeben ist. Ebenso ist es nicht erforderlich, Klassen im Muster zu überprüfen, die bei einer Normierung des Musters entfallen, da diese immer vorliegen, wenn der Baum der gegebenen Grammatik genügt.

Bei Übereinstimmung des Musters mit dem Teilbaum und Zutreffen der eventuell vorhandenen Bedingung wird die Vorschrift festgehalten.

6.7. Berechnung der kostengünstigsten Vorschriften

Zur Festlegung der kostengünstigsten Vorschrift werden die Kosten aller anwendbaren Vorschriften bestimmt. Ergeben sich dabei für eine Funktion geringere Kosten als sie zuvor vorlagen, werden die betreffende Funktion und diese Kosten festgehalten. Da die Kosten einer Vorschrift von den Kosten einer Funktion für den selben Knoten abhängen können, genügt es i.a. jedoch nicht, für jede Vorschrift einmal die Kosten zu berechnen. Die einfachste Vorgehensweise besteht darin, die Berechnung der Kosten für alle Vorschriften solange zu wiederholen, bis sich keine Verbesserungen der Kosten mehr ergeben. Da dieses Verfahren zu sehr vielen unnötigen Berechnungen führen muß, wird eine Optimierung durchgeführt, die in den meisten praktischen Fällen mit einem Durchlauf auskommt.

Offensichtlich ist eine Wiederholung der Berechnung, wenn überhaupt dann nur für solche Vorschriften erforderlich, deren Kosten von den Kosten einer anderen Funktion für den aktuellen Knoten abhängen. Die Vorschriften werden deshalb gemäß diesem Kriterium in zwei Gruppen gegliedert. Die eine Gruppe (und die umfaßt in der Praxis alle oder zumindest die meisten Vorschriften) muß nur einmal abgearbeitet werden. Die übrigen Vorschriften werden (sofern mindestens zwei verbleiben) wiederholt abgearbeitet, bis sich bei einem Durchlauf keine Verbesserung ergibt.

7. Bottom-Up Pattern-Matching mit einem Baumautomaten

Bei dem in Kapitel 6 vorgestellten Verfahren zum Festlegen der auf einen Teilbaum anwendbaren Vorschriften wird jedes Muster getrennt betrachtet. Das hat zur Folge, daß der Aufwand für diesen Schritt mit Zahl und Größe der zu untersuchenden Muster zunimmt. Beim sogenannten Bottom-Up Pattern-Matching mit einem Baumautomaten [HoO82] wird dies vermieden.

Bei diesem Verfahren werden zum Zeitpunkt der Generierung alle Mengen von Mustern bestimmt, die für das Pattern-Matching von Bedeutung sind. Diese Mengen bilden den Zustandsraum des Baumautomaten. Das Pattern-Matching erfolgt dann, indem für jeden Knoten des Baumes der Zustand berechnet wird, der die Menge von Mustern darstellt, welche auf den zugehörigen Teilbaum passen. Aus diesem Zustand läßt sich unmittelbar ablesen, welche Vorschriften aufgrund ihrer Muster in Frage kommen. Eventuell vorhandene Bedingungen müssen weiterhin gesondert überprüft werden.

7.1. Relevante Muster

Bevor der Zustandsraum bestimmt werden kann, müssen die relevanten Muster, d.h. die Muster, die für das Bottom-Up Pattern-Matching von Bedeutung sind, festgelegt werden. Grundlage für die Bestimmung der relevanten Muster sind die normierten Muster der gegebenen Vorschriften.

Für eine gegebene Menge P_N normierter Muster ist die Menge relevanter Muster die kleinste Menge P_R mit den Eigenschaften:

1. $P_N \subseteq P_R$
   Die vorgegeben normierten Muster sind relevant.
2. $n\,(p_1, ..., p_m) \in P_R$
   $p_1 \in P_R, ..., p_m \in P_R$
   Wenn ein Muster relevant ist, sind auch alle seine Teilmuster relevant.
3. $c \in P_R, c \rightarrow n\,(p_1, ..., p_m)$
   $\text{Normalize}\,(n\,(p_1, ..., p_m)) \in P_R$
   Ist ein einer Klasse entsprechendes Muster relevant, dann ist auch jedes normierte Muster einer aus dieser Klasse ableitbaren Knotenbeschreibung relevant.

Durch (1) wird sichergestellt, daß die vorgegebenen Muster, die beim Pattern-Matching erkannt werden sollen, in der Menge P_R enthalten sind. Mit (2) wird dafür gesorgt, daß auch die Teilmuster, die zum Erkennen eines Musters beitragen, in P_R enthalten sind. Da die Klassen zwar in den Mustern, nicht aber im realen Baum vorkommen, müssen die normierten Muster aller Knotenbeschreibungen, die einer in P_R enthaltenen Klasse entsprechen, ebenfalls in P_R liegen (3), damit die Klassen anhand dieser Muster erkannt werden können.

Abb. 7.1 zeigt die relevanten Muster, die entstehen, wenn wir die Mustern '+' (const, const) und '+' (expr, expr) zugrunde legen.

$P_N = \{ p_0, p_1 \}$
$P_R = \{ p_0, p_1, p_2, p_3 \}$

mit:

p_0 =	'+'	(const, const)	
p_1 =	'+'	(:, :)	*Normierung von '+' (expr, expr)*
p_2 =	const		*Teilmuster von p_0*
p_3 =	Const	()	*Beschreibung eines Knotens der Klasse const*

Abb. 7.1: relevante Muster

7.2. Zustände

Zur Beschreibung der Zustände Q des Baumautomaten könnte die Menge 2^{P_R} verwendet werden. Tatsächlich können jedoch nicht alle Teilmengen von P_R als Zustände vorkommen, da alle Zustände notwendigerweise die folgenden Bedingungen erfüllen müssen.

1. $p_1, p_2 \in q \; \neg\, p_1 \| p_2$
2. $p_1 \in P_R, p_2 \in q, p_1 < p_2$
   $p_1 \in q$

Die Bedingungen resultieren daraus, das jeder Zustand einen realen Baum beschreiben muß. Es ist deshalb nicht möglich, daß ein Zustand zwei sich widersprechende Muster enthält (1). Außerdem muß mit dem größeren Muster p_2 immer auch das kleinere Muster p_1 in q liegen.

In unserem Beispiel erfüllen nur sechs der 16 Mengen diese Eigenschaften (Abb. 7.2). Die übrigen scheiden aus (Abb. 7.3).

$Q = \{ q_0, q_1, q_2, q_3, q_4, q_5 \}$

mit:

q_0 =	$\{ p_0, p_1, p_2 \}$
q_1 =	$\{ p_1, p_2 \}$
q_2 =	$\{ p_1 \}$
q_3 =	$\{ p_2, p_3 \}$
q_4 =	$\{ p_2 \}$
q_5 =	$\{ \}$

Abb. 7.2: Zustände des Baumautomaten

Daß die angegebenen Bedingungen nicht hinreichend sind, zeigt sich an unserem Beispiel. Auf Grund der Grammatik (Abb. 7.1.) muß immer, wenn die Muster p_1 und p_2 passen, auch p_0 passen. Damit scheidet der Zustand q_1, der nur aus p_1 und p_2 besteht, aus. Der Zustand q_4 kann ebenfalls nie auftreten, da p_2 nur passen kann, wenn auch p_1 oder p_3 paßt.

$\{ p_0, p_1, p_2, p_3 \}$	nicht relevant da: $p_0 \parallel p_3$
$\{ p_0, p_1, p_3 \}$	nicht relevant da: $p_0 \parallel p_3$
$\{ p_0, p_1 \}$	nicht relevant da: $p_2 < p_0$
$\{ p_0, p_2, p_3 \}$	nicht relevant da: $p_0 \parallel p_3$
$\{ p_0, p_2 \}$	nicht relevant da: $p_1 < p_0$
$\{ p_0, p_3 \}$	nicht relevant da: $p_2 < p_0$
$\{ p_0 \}$	nicht relevant da: $p_1 < p_0$
$\{ p_1, p_2, p_3 \}$	nicht relevant da: $p_1 \parallel p_3$
$\{ p_1, p_3 \}$	nicht relevant da: $p_1 \parallel p_3$
$\{ p_3 \}$	nicht relevant da: $p_1 < p_3$

Abb. 7.3: nicht relevante Mengen von Mustern

Ein Ansatz zur Beseitigung dieses Mangel wird in Kapitel 7.7 gegeben.

In der Hauptbeschreibung eines Knotens ist für jeden Sohn eine Klasse festgelegt. Für jeden Sohn sind deshalb nur die Zustände relevant, die ausschließlich Muster enthalten, die aus dieser Klasse abgeleitet werden können.

Jeder Klasse c wird deshalb die Menge der für diese Klasse relevanten Muster P_c und der für diese Klasse relevante Zustände Q_c zugeordnet.

$$P_c := \{ p \in P_R \;\; p < c \;\; p = c \}$$
$$Q_c := \{ q \in Q \;\; q \subseteq P_c \}$$

Ebenso wie den Klassen kann man auch jedem Knoten n die Menge der für diesen Knoten relevanten Muster P_n und die für diesen Knoten relevanten Zustände Q_n zuordnen.

$$P_n := \{ p \in P_R \;\; p < n \;\; p = n \}$$
$$Q_n := \{ q \in Q \;\; q \subseteq P_n \}$$

7.3. Zustandsübergangsfunktionen

Um das Bottom-Up Pattern-Matching durchzuführen, wird für jeden Knoten n eine Zustandsübergangsfunktion f_n benötigt. Die Stelligkeit von f_n entspricht der Wertigkeit m des Knotens n.

$$f_n: Q_{c_1} \times ... \times Q_{c_m} \to Q_n$$

Die Funktion f_n bestimmt aufgrund der Zustände q_1, ..., q_m der Söhne eines Knotens n den Zustand für diesen Knoten.

Um für gegebene Zustände q_1, ..., q_m, $q = f_n (q_1, ..., q_m)$ festzulegen, wird so vorgegangen, daß für alle Muster $p \in P_n$ geprüft wird, ob sie in q enthalten sein müssen.

1. $n (p_1, ..., p_m) \in P_n, p_1 \in q_1, ..., p_m \in q_m$
   $n (p_1, ..., p_m) \in q$
2. $c \to n (c_1, ..., c_m)$, Normalize $(n (c_1, ..., c_m)) \in q \;\; c \in q$
3. $c \to n (c_1, ..., c_m)$, Normalize $(n (c_1, ..., c_m)) = c \; c \in q$

Ein Muster das mit einem Knoten beginnt liegt in q, wenn alle Teilmuster p_i bereits im entsprechenden Zustand q_i liegen (1). Das Muster einer Klasse liegt in q falls, eine Knotenbeschreibung dieser Klasse in normierter Form bereits in q enthalten ist (2), oder diese normierte Form mit der Klasse übereinstimmt (3).

Für unser Beispiel ergeben sich damit folgenden Übergangsfunktionen.

$f_{,+,}(q_0, q_0) = q_0$ $f_{,+,}(q_0, q_1) = q_0$ $f_{,+,}(q_0, q_2) = q_2$
$f_{,+,}(q_0, q_3) = q_0$ $f_{,+,}(q_0, q_4) = q_0$ $f_{,+,}(q_0, q_5) = q_2$
$f_{,+,}(q_1, q_0) = q_0$ $f_{,+,}(q_1, q_1) = q_0$ $f_{,+,}(q_1, q_2) = q_2$
$f_{,+,}(q_1, q_3) = q_0$ $f_{,+,}(q_1, q_4) = q_0$ $f_{,+,}(q_1, q_5) = q_2$
$f_{,+,}(q_2, q_0) = q_2$ $f_{,+,}(q_2, q_1) = q_2$ $f_{,+,}(q_2, q_2) = q_2$
$f_{,+,}(q_2, q_3) = q_2$ $f_{,+,}(q_2, q_4) = q_2$ $f_{,+,}(q_2, q_5) = q_2$
$f_{,+,}(q_3, q_0) = q_0$ $f_{,+,}(q_3, q_1) = q_0$ $f_{,+,}(q_3, q_2) = q_2$
$f_{,+,}(q_3, q_3) = q_0$ $f_{,+,}(q_3, q_4) = q_0$ $f_{,+,}(q_3, q_5) = q_2$
$f_{,+,}(q_4, q_0) = q_0$ $f_{,+,}(q_4, q_1) = q_0$ $f_{,+,}(q_4, q_2) = q_2$
$f_{,+,}(q_4, q_3) = q_0$ $f_{,+,}(q_4, q_4) = q_0$ $f_{,+,}(q_4, q_5) = q_2$
$f_{,+,}(q_5, q_0) = q_2$ $f_{,+,}(q_5, q_1) = q_2$ $f_{,+,}(q_5, q_2) = q_2$
$f_{,+,}(q_5, q_3) = q_2$ $f_{,+,}(q_5, q_4) = q_2$ $f_{,+,}(q_5, q_5) = q_2$

$f_{Const}() = q_3$ $f_{Ident}() = q_5$

Abb. 7.4: Zustandsübergangsfunktionen

Das Pattern-Matching läuft nun so ab, daß in einem Bottom-Up-Durchlauf durch den Baum für jeden Knoten ein Zustand $q \in Q$ berechnet wird. Diese Berechnung erfolgt, indem die zum jeweiligen Knoten n gehörige Funktion f_n auf die bereits berechneten Zustände q_1, ..., q_m der Söhne angewandt wird.

7.4. Felder zur Darstellung der Zustandsübergangsfunktionen

Um die Zustandsübergangsfunktionen darzustellen, könnte man für jeden Knoten ein Feld vorsehen. Die Felder hätten jeweils soviele Dimensionen, wie der zugehörige Knoten Söhne hat. Die Einträge der Felder können zum Zeitpunkt der Generierung bestimmt werden. Zur Laufzeit wäre lediglich ein Zugriff auf das Feld erforderlich, um den Zustand für den aktuellen Knoten zu bestimmen.

'+' (i, j)

i \ j	q_0	q_1	q_2	q_3	q_4	q_5
q_0	q_0	q_0	q_2	q_0	q_0	q_2
q_1	q_0	q_0	q_2	q_0	q_0	q_2
q_2	q_2	q_2	q_2	q_2	q_2	q_2
q_3	q_0	q_0	q_2	q_0	q_0	q_2
q_4	q_0	q_0	q_2	q_0	q_0	q_2
q_5	q_2	q_2	q_2	q_2	q_2	q_2

Const ()

q_3

Ident ()

q_5

Abb. 7.5: Felder zur Darstellung der Zustandsübergangsfunktionen

Abb. 7.5 zeigt die Felder zur Darstellung der Zustandsübergangsfunktionen, die sich für unser Beispiel ergeben.

Da für Söhne aus der Klasse c nur Zustände aus Q_c vorkommen können, würde es ausreichen die Felder für diesen Teil zu realisieren. Da aber Muster und damit auch Zustände durchaus für verschiedene Klassen relevant sein können, kann die Codierung dieser Zustände nicht immer dicht liegen. Bei realistischen Beispielen wären die Felder deshalb sehr groß jedoch nicht vollständig besetzt.

Um den enormen Speicherbedarf für die mehrdimensionalen Felder zu vermeiden, liegt es nahe, eine Komprimierung durchzuführen.

7.5. Automaten zur Darstellung der Übergangsfunktionen

Die Komprimierung erfolgt in Anlehnung an das in [BMW87] vorgestellte Verfahren. Im ersten Schritt werden an Stelle der Felder Automaten aufgebaut. Diese Automaten arbeiten horizontal, die Söhne eines Knotens werden hierbei von links nach rechts abgearbeitet, wobei jeweils ein durch den Zustand des Sohnes gesteuerter Übergang im Automat stattfindet. Anstelle des Zugriffs auf ein n-dimensionales Feld erfolgen also n Übergänge in einem Automaten.

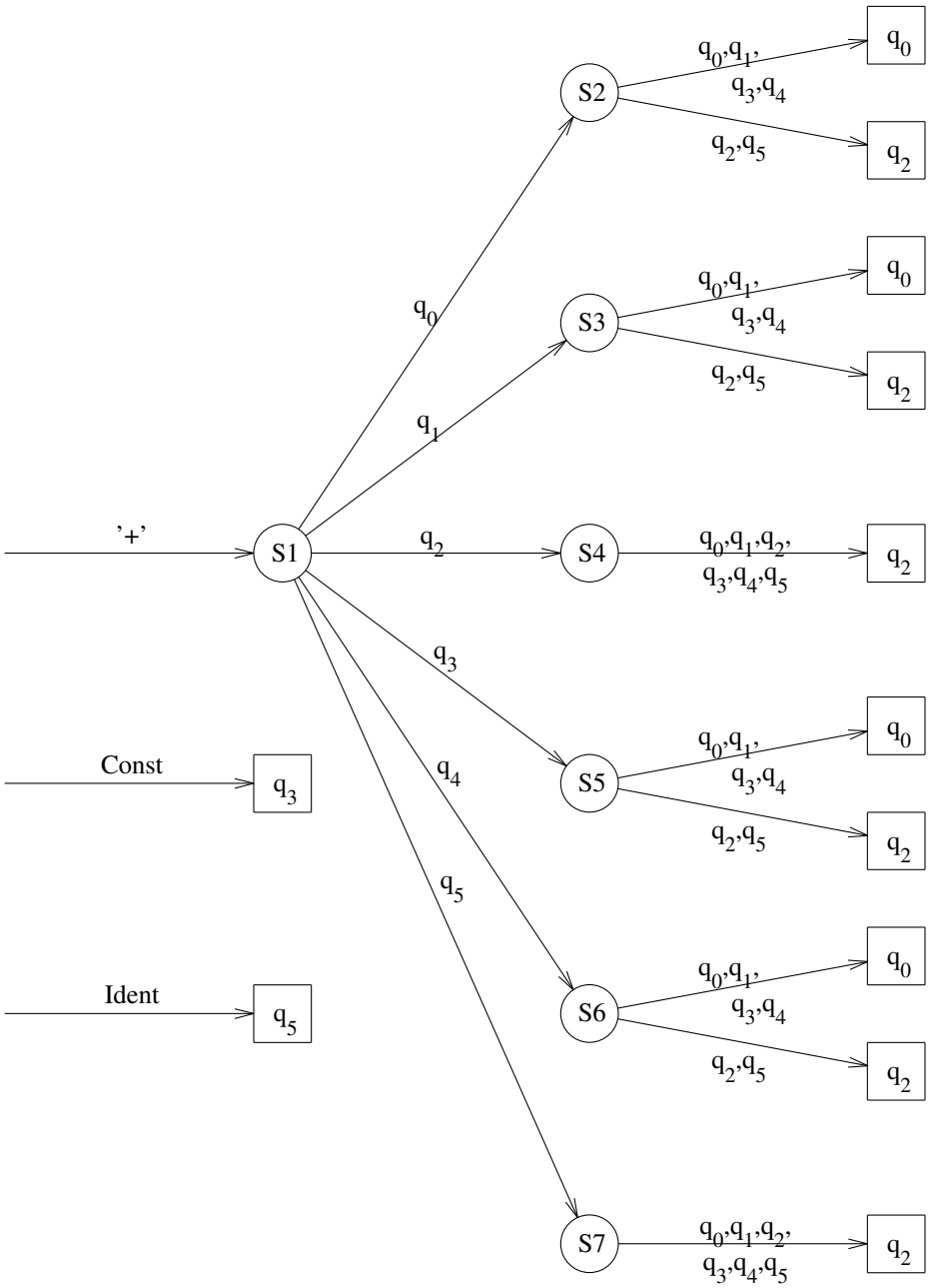

Abb. 7.6: Automaten zur Darstellung der Zustandsübergangsfunktionen

In Abb. 7.6 sind die Automaten dargestellt, die den Feldern von Abb. 7.5 entsprechen. Die durch Kreise dargestellten Zustände werden bei der Abarbeitung der Söhne eines Knotens durchlaufen. Die durch Quadrate dargestellten Endzustände, stellen das Ergebnis der Zustandsübergangsfunktion dar.

Es ist unschwer zu erkennen, daß die Automaten zum Teil vereinfacht werden kann. Offensichtlich sind die Zustände S2, S3, S5 und S6 äquivalent. Das selbe gilt für S4 und S7. Faßt man diese Zustände zusammen so erhält man einen reduzierten Automaten (Abb. 7.7).

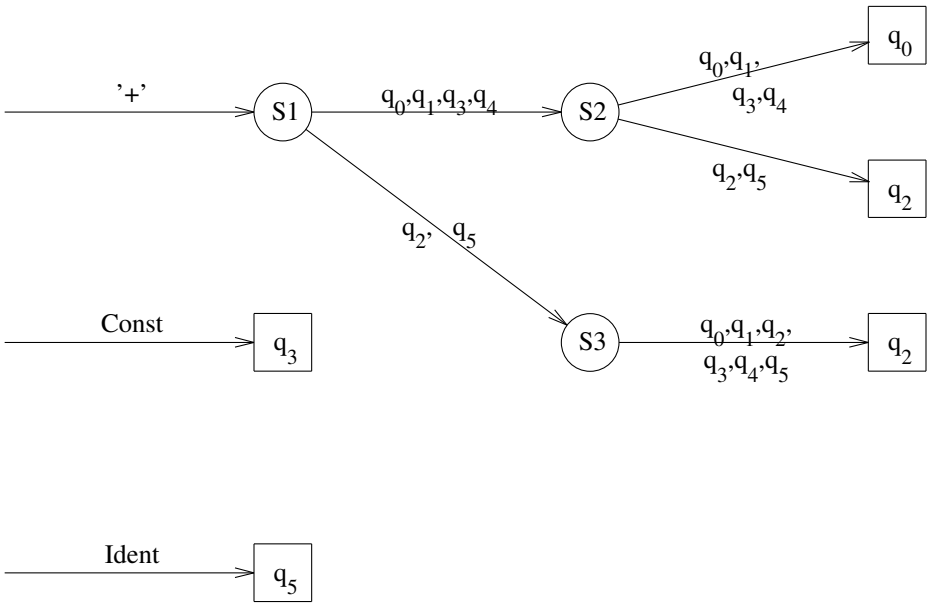

Abb. 7.7: reduzierte Automaten zur Berechnung der Zustände

Um diese Komprimierung zu erreichen, wird so vorgegangen, daß man verträgliche Zustände zusammenfaßt, wobei Zustände dann verträglich sind, wenn sie bei der gleichen Eingabe zum selben Zustand führen. Da sich die Zusammenfassung von Zuständen im allgemeinen positiv auf die Verträglichkeit der Vorgänger auswirkt, sollten dabei alle Nachfolger eines Zustandes auf mögliche Verträglichkeiten untersucht und mögliche Zusammenfassungen durchgeführt sein, bevor der Zustand selbst betrachtet wird.

7.6. Realisierung der Automaten

Um den Automaten zu realisieren, werden die zu einem Zustand gehörigen Übergänge als eindimensionale Felder dargestellt (Abb. 7.8).

	q_0	q_1	q_2	q_3	q_4	q_5
S1	S2	S2	S3	S2	S2	S3

	q_0	q_1	q_2	q_3	q_4	q_5
S2	q_0	q_0	q_2	q_0	q_0	q_2

	q_0	q_1	q_2	q_3	q_4	q_5
S3	q_2	q_2	q_2	q_2	q_2	q_2

Abb. 7.8: Felder zur Berechnung der Zustandsübergänge

Um den Speicherbedarf für diese Felder zu reduzieren, werden alle in ein gemeinsames Feld eingebettet. Bei diesem unter dem Begriff Kammvektortechnik bekannten Verfahren werden soweit möglich identische Einträge verschiedener Felder überlagert, und Lücken in den Feldern durch die Einträge andere Felder gefüllt.

S2	S2	S3	S2	S2	S3											
						q_0	q_0	q_2	q_0	q_0	q_2					
											q_2	q_2	q_2	q_2	q_2	q_2

S2	S2	S3	S2	S2	S3	q_0	q_0	q_2	q_0	q_0	q_2	q_2	q_2	q_2	q_2	q_2

Abb. 7.8: Einbettung des Automaten in ein eindimensionales Feld

In unserem Beispiel (Abb. 7.9) ist die erzielte Einsparung sehr gering, da es keine Lücken gibt. Dies liegt daran, daß die zugrunde gelegte Grammatik extrem einfach ist. Die Klasse (expr), die bei der Festlegung der zulässigen Söhne von '+' verwendet wurde, führt zu keinerlei Einschränkungen, und somit sind alle Zustände für die Söhne möglich ($Q_{expr} = Q$).

In realistischen Beispielen wird durch diese Maßnahme jedoch eine drastische Einsparung erzielt, da dort durch Verwendung vieler sich gegenseitig ausschließender Klassen in den einzelnen Felder viele Lücken entstehen, die bei der Einbettung in ein gemeinsames Feld zum Teil geschlossen werden.

7.7. Vermeidung unnötiger Zustände

In 9.2. wurde festgestellt, daß die Bedingungen zur Festlegung der Zustände des Baumautomaten nicht hinreichend sind, um sicher zu stellen, daß nur solche Zustände betrachtet werden, die wirklich eintreten können. Es gibt jedoch eine Möglichkeit, diesen Mangel im nachhinein zu beheben.

Offensichtlich ist es so, daß die möglichen Zustände des Baumautomaten genau den erreichbaren Endzuständen der Automaten zur Darstellung der Übergangsfunktionen entsprechen. Um sie zu erkennen, genügt es also, die erreichbaren Zustände dieser Automaten zu berechnen.

7.8. Implementierung des Bottom-Up Pattern-Matching

Neben dem eindimensionalen Feld (yyComb), das die horizontalen Automaten beschreibt, wird für die Realisierung des Pattern-Matching noch ein Feld benötigt, das jedem Zustand des Baumautomaten die Menge der Vorschriften, deren Muster paßt, zuordnet (yySets).

Das Pattern-Matching läuft dann für jeden Knoten c folgendermaßen ab:

1. state := $const_c$
2. $\forall$ Söhne s_i von c
   state := yyComb [state + yyTraverse (s_i)]
3. match := ADR (yySets [state])
4. RETURN state

Zu Beginn (1) wird eine lokale Variable *state* mit einem knotenspezifischen Wert initialisiert. Die Variable state beschreibt sowohl die Zustände des horizontalen Automaten als auch die des

Baumautomaten. Im zweiten Schritt (2) werden die Söhne rekursiv abgearbeitet. Gleichzeitig werden, gesteuert durch die Zustände der Söhne, die von yyTraverse als Ergebnis geliefert werden, die Übergänge im horizontalen Automaten durchgeführt. Schließlich wird das Ergebnis für den eigenen Teilbaum, das durch die Variable state beschrieben wird, benutzt, um die Vorschriften zu bestimmen, deren Muster auf den aktuellen Teilbaum passen (3) und anschließend zurückgeliefert (4).

Wenn das Pattern-Matching in die Prozedur yyTraverse, die in Kapitel 6 beschrieben ist, integriert wird, ergeben sich folgende Änderungen.

1. Die Abarbeitung der Söhne wird mit der Abarbeitung des horizontalen Automaten kombiniert (Punkte (1) und (2) von oben)
2. Die Berechnung der Klassen entfällt, da diese beim Bottom-Up Pattern-Matching nicht benötigt werden.
3. Um die zulässigen Vorschriften zu bestimmen, wird neben den Bedingungen lediglich die über die Variable match zugängliche Menge benutzt (Punkt (3) von oben).
4. Die Prozedur yyTraverse liefert nun einen Zustand als Ergebnis zurück (Punkt (4) von oben).

Da Modula-2 keine initialisierte Variablen kennt, müssen außerdem die Felder yySets und yyComb eingelesen werden, bevor die Transformation vorbereitet werden kann.

Die hier beschriebenen Unterschiede können im Anhang B.2, der beide Lösungen im Vergleich zeigt, im Detail studiert werden.

8. Vollständigkeit

Eine wesentliche Forderung, die an eine Spezifikation gestellt wird, ist die der Vollständigkeit. Eine Spezifikation heißt vollständig, wenn sie geeignet ist, für jede zulässige Eingabe die Transformation zu beschreiben.

Die allgemeine Frage, ob eine Spezifikation vollständig ist, ist nicht entscheidbar. Im folgenden wird deshalb eine schärfere Eigenschaft betrachtet, die die Vollständigkeit impliziert. Obwohl auch diese Eigenschaft nicht entscheidbar ist, hilft sie uns einen Schritt weiter, denn einzelnen Teile dieser Eigenschaft können entschieden werden und eignen sich, bestimmte Lücken in der Vollständigkeit aufzuspüren.

Um eine Transformation durchzuführen, wird die Aufgabe gestellt, die Wurzel mit der ersten Funktion der Spezifikation zu bearbeiten. Um eine solche Aufgabe zu lösen, muß eine passende Vorschrift gefunden werden. Funktionsaufrufe in den Anweisungen dieser Vorschrift bilden neue Aufgaben, die ebenfalls bearbeitet werden müssen.

Falls man sicherstellen kann, daß zur Durchführung der Transformation nur zulässige Aufgaben gestellt werden (Einhaltung des Definitionsbereichs), daß für jede zulässige Aufgabe eine passende Vorschrift existiert (Überdeckung des Definitionsbereichs) und daß alle zulässigen Aufgaben abgearbeitet werden können (Terminierung), dann ist die Spezifikation sicher vollständig.

8.1. Einhaltung des Definitionsbereichs

Um die Einhaltung des Definitionsbereichs sicherzustellen, muß geprüft werden, ob die Teilbäume, auf die eine Funktion angewandt wird, in den durch den Definitionsbereich festgelegten Klassen liegen.

Wenn beim Funktionsaufruf ein Selektor verwendet wird, wird diese Prüfung vorgenommen, indem das Teilmuster, das durch den Selektor bestimmt ist, analysiert wird. Die Einhaltung des Definitionsbereichs ist sichergestellt, falls dieses Teilmuster aus einer Klasse des Definitionsbereichs abgeleitet werden kann.

Wird der zu transformierende Baum beim Funktionsaufruf auf eine andere Weise festgelegt (z.B. durch eine Variable), kann die Einhaltung des Definitionsbereichs nicht automatisch überprüft werden, da dann zur Generierungszeit keine Möglichkeit besteht, die Klassenzugehörigkeit dieses Baumes zu bestimmen. Der Anwender von ESTRAL wird in solchen Fällen durch eine Warnung auf die Gefahr hingewiesen.

8.2. Überdeckung des Definitionsbereichs

Der wesentliche Bestandteil der Vollständigkeitsprüfung ist die Prüfung, ob der Definitionsbereich durch die Vorschriften abgedeckt wird.

Dieser Test wird durchgeführt, indem jede Klasse (c) des Definitionsbereichs (Domain) als (zunächst einelementige) Menge von synthetisierten Mustern (syn) aufgefaßt wird und alle realen Muster (real) ausgefiltert werden, die durch eine Vorschrift abgedeckt werden. Die (in syn) verbliebenen Muster können mit den Vorschriften nicht bearbeitet werden, sie dokumentieren die Unvollständigkeit.

Um die Bedingungen der Vorschriften zu berücksichtigen, werden zwei Durchläufe unterschieden. Im ersten Durchlauf wird versucht, mit den Vorschriften auszukommen, die nicht mit einer Bedingung verknüpft sind. Falls dies nicht gelingt, werden die bedingten Vorschriften ebenfalls zum Test herangezogen. Schließlich wird für jedes Muster, das zum Schluß übrig geblieben ist, ein Fehler (Error) gemeldet. Muster die erst beim zweiten Durchlauf ausgefiltert werden, führen zu einer Warnung (Warning).

```
Cover

for all functions f do
 for all c ∈ Domain (f) do
  cp := MakePattern (c)
  syn := EmptyList
  Append (syn, cp)
  real := GetUnCondPatterns (f)
  match := EmptyList
  Filter (syn, real, match)
  real := GetCondPatterns (f)
  match := EmptyList
  Filter (syn, real, match)
  while ¬ IsEmpty (match) do
   p := TakeHead (match)
   Warning ('no unconditional pattern matching', p)
  while ¬ IsEmpty (syn) do
   p := TakeHead (syn)
   Error ('no pattern at all matching', p)
```

Abb. 8.7: Überdeckung des Definitionsbereich

Zum Filtern (Abb. 8.8) wird die Funktion Relation von Abb. 7.6 herangezogen. Wenn ein synthetisiertes Muster (sp) durch ein reales Muster (rp) vollständig überdeckt wird, muß sp nicht weiter betrachtet werden. Ist rp nicht geeignet, um sp vollständig zu überdecken, muß so vorgegangen werden, daß sp durch Anwenden der Produktionen der Grammatik durch weitere synthetisierte Muster ersetzt wird (Extend), die das ursprüngliche vollständig beschreiben.

Um diese Erweiterung des synthetisierten Musters (sp) zu unterstützen, wird eine Funktion RelationP benutzt. RelationP ist eine Erweiterung von Relation, die für die Fälle unabhängig (independent) und größer (greater) eine Position zurückliefert, an der eine sinnvolle Erweiterung ansetzen kann. Mit Hilfe dieser Position und anhand der Grammatik wird dann von Extend die Erweiterung, d.h. die Synthese von weiteren Mustern durchgeführt. Die entstandenen Muster werden in die Liste syn eingefügt, um im folgenden bearbeitet zu werden.

Schließlich wird es möglich sein, einige der generierten Muster auszufiltern (match), andere Muster werden übrig bleiben (rest). Der Rest bildet den Ausgangspunkt für den nächsten Durchlauf. Nach Betrachten aller realen Muster enthält syn alle synthetisierten Muster, die nicht durch die realen Muster überdeckt werden konnten.

```
Filter (VAR syn, real, match: tPatternList)

while ¬ Empty (syn) & ¬ Empty (real) do
 rest := EmptyList
 rp := TakeHead (real)
 while ¬ Empty (syn) do
  sp := TakeHead (syn)
  case RelationP (sp, rp, pos) of
  | equal, less:
     Append (match, sp)
  | inconsistent:
     Append (rest, sp)
  | independent, greater:
     if pos = undefined then
      Append (rest, sp)
     else
      Extend (sp, pos, syn)
 syn := rest
```

Abb. 8.8: Filtern von Mustern

Abb. 8.9 zeigt zwei unabhängige Muster, die der Grammatik von Abb. 4.1 entsprechen, für die RelationP jedoch keine sinnvolle Position zurückliefern kann (pos = undefined).

```
sp = '+' (:,:)
rp = const
```

Abb. 8.9: Muster ohne sinnvolle Erweiterung

Für die beiden Muster sp und rp existiert keine sinnvolle Erweiterung. Zwar würde die Belegung der Blätter von sp mit allen Möglichkeiten, die durch die Grammatik gegeben sind, dazu führen, daß ein Teil der dadurch entstandenen Muster ausgefiltert werden kann, doch würden dabei immer neue von rp unabhängige Muster entstehen, die weiterbearbeitet werden müßten. Um die Terminierung des Algorithmus an dieser Stelle sicherzustellen, wird in solchen Fällen auf eine Erweiterung verzichtet. In pathologischen Fällen kann das zwar dazu führen, daß eine Eingabe irrtümlich als unvollständig erkannt wird, doch kommt dies in der Praxis kaum vor.

8.3. Terminierung

Die Terminierung kann meist dadurch sichergestellt werden, daß sich die Funktionsaufrufe auf tieferliegende Teilbäume beziehen. Da der Eingabebaum endlich ist, werden schließlich die Blätter erreicht und folglich keine weiteren Funktionen aufgerufen.

Falls sich Funktionsaufrufe hingegen auf den aktuellen Teilbaum insgesamt beziehen, wird die Terminierung in Frage gestellt. Denn durch einen solchen Funktionsaufruf wird die Terminierung einer Vorschrift unmittelbar von der Terminierung der aufgerufenen Funktion abhängig. Damit entsteht eine Abhängigkeiten zwischen Funktionen (aufrufende und aufgerufene Funktion). Die Abhängigkeit beschränkt sich allerdings auf das Muster der

betreffenden Vorschrift. Außerdem ist die Abhängigkeit nur dann unvermeidbar, wenn es keine andere Vorschrift gibt, die alternativ angewandt werden könnte und keine Abhängigkeit hervorruft. Um unter solchen Bedingungen die Terminierung nachzuweisen, müßte gezeigt werden, daß die Entstehung von zyklischen Abhängigkeiten vermieden werden kann.

Falls diese Frage überhaupt entscheidbar ist, ist die hierzu notwendige Untersuchung auf alle Fälle äußerst aufwendig. Andererseits sind die Abhängigkeiten zwischen Funktionen in der Praxis recht gering und somit auch von Hand zu überschauen. Von ESTRA wird deshalb zum Zeitpunkt der Generierung kein Test auf Terminierung durchgeführt.

9. Schnittstellen des Transformators

9.1. Struktur der Eingabebäume

Bei der Entwicklung von ESTRA wurde eine Darstellung der attributierten Bäume vorausgesetzt, wie sie von ast [Gro89] zur Verfügung gestellt wird.

```
CONST

  Plus     = 1;
  Const    = 2;
  Ident    = 3;

  expr     = 4;
  const    = 5;
  index    = 6;

TYPE

  tTree    = POINTER TO tNode;

  yexpr    = RECORD pos: tPosition END;
  yconst   = RECORD pos: tPosition END;
  yindex   = RECORD pos: tPosition END;

  yPlus    = RECORD pos: tPosition; lo: tTree; ro: tTree; END;
  yConst   = RECORD pos: tPosition; value: INTEGER; END;
  yIdent   = RECORD pos: tPosition; ident: tIdent; END;

  tNode    = RECORD
    yyEstraInfo:  ADDRESS;
    CASE Kind:    INTEGER OF
    | expr:       expr:     yexpr;
    | const:      const:    yconst;
    | index:      index:    yindex;
    | Plus:       Plus:     yPlus;
    | Const:      Const:    yConst;
    | Ident:      Ident:    yIdent;
    END;
  END;
```

Abb. 9.1: Darstellung der attributierten Bäume in Modula-2

Die attributierten Bäume werden als Zeiger auf variante Strukturen realisiert (tTree). Die Struktur (tNode) enthält für jede Knotenart und jede Klasse eine Variante. Die gültige Variante wird durch das Tag-Feld (Kind) angezeigt.

Würde immer über die Variante, die durch das Tag-Feld ausgezeichnet ist, auf den Knoten zugegriffen, wären die Varianten, die den Klassen entsprechen, überflüssig. Tatsächlich ist es aber so, daß diese Varianten benutzt werden, um auf Attribute von Söhnen zuzugreifen, deren Knotenart nicht feststeht. In unserem Beispiel kann auf diese Weise in jedem Knoten auf das Attribut *Pos* zugegriffen werden. Damit dies wirklich funktioniert, genügt es nicht, daß dieses Attribut in jeder Variante vorkommt, es muß auch immer an der selben Stelle stehen. Dies kann

im allgemeinen sichergestellt werden, wenn die Variante eines Knotens eine Erweiterungen der Varianten der Klassen darstellen, aus denen er hervorgehen kann.

Das Feld yyEstraInfo dient zur Aufnahme der Attribute, die zur Festlegung der Transformation benötigt werden.

9.2. Schnittstelle des generierten Moduls

Von ESTRA wird ein Modul generiert, das den Namen der Transformation (z.B. Trans) erhält.

```
DEFINITION MODULE Trans;

IMPORT      Tree;

TYPE        tTree = Tree.tTree

(* for bottom-up pattern-matching only *)
VAR         TransTabName: ARRAY [0..127] OF CHAR;

PROCEDURE BeginTrans;
PROCEDURE DoTrans (yyt: Tree.tTree);
PROCEDURE CloseTrans;

END Trans.
```

Abb. 9.2: Schnittstelle des generierten Transformators

Die parameterlosen Prozeduren BeginTrans und CloseTrans dienen zur Initialisierung und zur Durchführung von Abschlußarbeiten. DoTrans führt die Transformation des Baumes yyt durch. Die Schnittstelle von DoTrans entspricht der Schnittstelle der ersten Funktion, die in Trans spezifiziert wurde. D.h. falls diese Funktion vererbte oder synthetisierte Attribute hat, wird die Schnittstelle von DoTrans entsprechend erweitert.

Der Typ tTree, der zur Darstellung der Bäume verwendet wird, muß im EXPORT-Abschnitt bekannt gemacht werden, damit er für die Prozedur DoTrans zur Verfügung steht.

Die Variable TransTabName wird bei Verwendung des Bottom-Up Pattern-Matching Algorithmus benutzt, um den Namen der Tabellendatei (hier "Trans.tab") aufzunehmen. Damit besteht die Möglichkeit, den Namen vor dem Aufruf von BeginTrans zu umzusetzen, falls ein anderer Dateiname benutzt werden soll.

10. Vergleich mit anderen Werkzeugen

10.1. Twig

Twig [AGT87] ist ein Werkzeug zur Erstellung von Codegeneratoren. Die Beschreibung der Codegeneratoren erfolgt durch Regeln, die aus einer Umschreibregel, Kosten und Aktionen, bestehen.

Eine Umschreibregel besteht aus einem Muster und einem Nichtterminal, dem sogenannten *replacement node*. Das Muster legt fest, wo die Regel benutzt werden kann. Nachdem dann die zugehörige Aktion ausgeführt wurde, wird der Teilbaum durch den replacement node ersetzt.

Bei der Ausführung von Aktionen werden bei Twig verschiedene Strategien unterschieden. Neben dem *topdown-mode*, dessen Vorgehensweise der von ESTRA entspricht, kennt Twig noch den *rewrite-mode* und den *deferred-mode*. Der *deferred-mode* entspricht der Postfixstrategie und stellt somit lediglich eine Abkürzung dar, da die Transformation nicht explizit für die einzelnen Söhne aufgerufen werden muß.

Diese Abkürzung ist deshalb möglich, weil Twig keine explizite Auswahl der Funktionen kennt, mit der die einzelnen Blätter des Musters zu bearbeiten sind. Die Bearbeitung der Blätter wird hier durch die Nichtterminale im Muster bestimmt. Diese Nichtterminale müssen mit den *replacement nodes* der zur Transformation der Blätter angewandten Regeln übereinstimmen. Eine mehrfache Abbildung eines Teilbaums mit unterschiedlichen Funktionen, wie ESTRA sie zuläßt, ist folglich mit Twig nicht möglich.

Beim *rewrite-mode*, der von ESTRA nicht unterstützt wird, wird der aktuelle Teilbaum durch die Aktion ersetzt und anschließend neu bearbeitet. Diese Methode wird bei Twig angeboten, obwohl die Gefahr besteht, daß es zu einem endlosen Umschreiben kommt. Maßnahmen zur automatischen Vermeidung eines solchen endlosen Umschreibens werden von Twig nicht getroffen (vgl. Kapitel 5).

Der Zugriff auf den Baum erfolgt bei Twig über einen abstrakten Datentyp (ADT), wodurch eine weitgehende Unabhängigkeit von der Darstellung des Baumes erreicht wird.

In ESTRAL wurde auf die Verwendung eines ADTs hingegen verzichtet, um den Verlust an Effizienz, der durch die in MODULA-2 notwendige Realisierung der Operationen als Unterprogramme entsteht, zu vermeiden. Auf der anderen Seite ist dadurch die Flexibilität in der Darstellung der Bäume genommen. Dies ist jedoch nicht problematisch, da davon ausgegangen wird, daß der Anwender von ESTRA bei der Darstellung der Bäume das Werkzeug Ast [Gro89] zu Hilfe nimmt. Denn das von Ast verwendete Format wurde bei der Entwicklung von ESTRA zugrundegelegt und wird somit voll unterstützt.

Zur Darstellung der zur Transformation nötigen Attribute wird bei Twig kein Platz in den Knoten des Baumes reserviert. Stattdessen wird eine neuer Baum mit der selben Struktur aufgebaut, der diese Attribute sowie Verweise auf die Knoten des ursprünglichen Baumes aufnimmt.

10.2. BEG

BEG (Back End Generator) [Emm88] ist ein Werkzeug zur automatischen Erzeugung effizienter Codegeneratoren. Die Beschreibung erfolgt durch Baumersetzungsregeln.

Eine Baumersetzungsregel beschreibt die Reduktion eines Teilbaumes auf ein Nichtterminal. Die Transformation des gesamten Baumes wird durch die Reduktion des Baumes auf das Startsymbol bewirkt. Die Nichtterminale spielen somit die selbe Rolle wie bereits bei Twig, sie stellen sicher, daß die Regeln zusammenpassen.

Die einzelnen Regeln können hier ebenfalls mit Kosten und Bedingungen versehen werden.

Daß es sich bei BEG um ein spezielles Werkzeug zur Beschreibung von Codegeneratoren handelt, zeigt die Tatsache, daß BEG spezielle Mittel zur Beschreibung der Registervergabe zur Verfügung stellt.

Außerdem beschränkt sich BEG auf die für die Codeerzeugung ausreichende Postfixstrategie. Dies hat den Vorteil, daß in den Regeln kein expliziter Aufruf zur Transformation der Söhne erforderlich ist. Andererseits ist BEG damit für allgemeine Transformationen unbrauchbar.

10.3. OPTRAN

Die Spezifikationssprache OPTRAN [MWW84] wurde entwickelt, um die Transformation attributierter Bäume zu beschreiben. Da hier verlangt wird, daß das Ergebnis der Transformation wieder ein attributierter Baum ist, können die Strukturen der Ein- und Ausgabe durch attributierte Grammatiken beschrieben werden.

Zur Beschreibung der Transformation werden Regeln benutzt, die die Abbildung eines Eingabe-Musters in ein Ausgabe-Muster festlegen, wobei die Regeln so geartet sein müssen, daß sie entweder innerhalb einer Struktur (Ein- oder Ausgabe) bleiben, oder den Übergang von der Eingabe- zur Ausgabe-Struktur beschreiben.

Wie bereits in Kapitel 2 erwähnt wurde, spielt die Reihenfolge, in der die Regeln angewandt werden, bei dieser Methode eine entscheidende Rolle. In OPTRAN wird deshalb vom Benutzer verlangt, eine Strategie zu bestimmen, die diese festlegt.

Im Unterschied zu ESTRA legt OPTRAN besonderen Wert auf die Attributierung. Dabei wird zwischen einer Erstattributierung und einer Reattributierung unterschieden. Die Erstattributierung dient dazu, vor der Transformation die Attribute gemäß der Eingabe-Grammatik zu berechnen. Nach der Transformation muß der Baum gemäß der Ausgabe-Grammatik attributiert sein. Um dies zu erreichen, wird nach der Anwendung von Regeln eine Reattributierung durchgeführt, die sicherstellt, daß Attribute, die sich aufgrund der Regelanwendung geändert haben, neu berechnet werden.

11. Zusammenfassung

Mit ESTRAL wurde eine universelle Spezifikationssprache entwickelt, die geeignet ist, beliebige Transformationen attributierter Bäume zu beschreiben.

Durch die Verwendung von Mustern, Bedingungen und Kosten können Transformationen auf natürliche Weise, d.h. mehrdeutig beschrieben werden. Der sequentiellen Natur der Transformation wird durch die imperative Beschreibung der einzelnen Vorschriften Rechnung getragen.

Um die Beschreibung einer Transformation automatisch in eine Implementierung umzusetzen, wurde der Generator ESTRA entwickelt. Zur Umsetzung werden von ESTRA zwei Versionen angeboten, die sich in Bezug auf das Pattern-Matching unterscheiden.

Zum einen wird ein naives Pattern-Matching angeboten, bei dem jedes Muster getrennt betrachtet wird, zum anderen kann auch ein Bottom-Up Pattern-Matching, das alle Muster im Zusammenhang sieht, benutzt werden. Das Bottom-Up Pattern-Matching zahlt sich insbesondere dann aus, wenn bei der Beschreibung der Transformationen viele tiefe Muster verwendet werden. Falls es hingegen nur sehr einfache Muster gibt, ist das naive Pattern-Matching überlegen.

Erste eigene Einsätze von ESTRA haben gezeigt, daß die in ESTRAL erstellten Spezifikationen in Bezug auf die Wartbarkeit deutliche Vorteile gegenüber einer direkten Implementierung aufweisen. Neben dem höheren Abstraktionsniveau zahlt sich vor allem die Möglichkeit aus, Spezifikation zu verbessern, in dem man Vorschriften hinzufügt, um Spezialfälle zu optimieren.

Zielsprache des Generators ist MODULA-2. Erweiterungen des Generators zur Unterstützung anderer Programmiersprachen (insbesondere C) wurden aber bereits beim Entwurf und der Implementierung des Generators vorbereitet.

Anhang A: Syntax von ESTRAL

transformation	=	**TRANSFORMATION**	
		ident	*Name der Transformation*
		[**EXPORT** '{' source_text '}']	*öffentliche Vereinbarungen*
		[**GLOBAL** '{' source_text '}']	*globale Vereinbarungen*
		[**BEGIN** '{' source_text '}']	*Anweisungen zur Initialisierung*
		[**CLOSE** '{' source_text '}']	*Anweisungen zum Abschluß*
		grammar	*Baumgrammatik*
		function *	*Funktionen*
grammar	=	**GRAMMAR**	
		ident	*Name der Grammatik*
		production *	*Klassen*
production	=	class	*Klasse*
		node *	*Knotenbeschreibungen*
class	=	[ident '–>']	*Bezeichner der Oberklasse*
		ident '='	*Bezeichner der Klasse*
node	=	'|' (string | ident)	*Name des Knotens*
		[':'ident]	*Bezeichner des Knotens*
		'(' [son || ','] ')'	*Söhne*
son	=	[ident ':']	*Name des Sohnes*
		ident	*Bezeichner der Klasse des Sohnes*
function	=	**FUNCTION**	
		ident	*Name der Funktion*
		[attributes	*vererbte Attribute*
		'–>'	
		attributes]	*synthetisierte Attribute*
		[':' type]	*Ergebnistyp*
		domain	*Definitionsbereich*
		directive *	*Vorschriften zur Beschreibung der Funktion*
attributes	=	[(ident || ','	*Liste von Bezeichnern*
		':' type)	*Angabe des Typs*
		|| ';']	*mehrere solche Listen durch ';' getrennt*
type	=	[ident '.']	*Angabe des Moduls zur Qualifikation*
		ident	*Bezeichner des Typs*
domain	=	'/' ident || ',' '/'	*Liste von Klassenbezeichnern*
directive	=	pattern	*Muster*
		[**CONDITION** '{' source_text '}']	*Bedingung*
		[**COSTS** (number | '{' source_text '}')]	*Kosten*
		[**DECLARE** '{' source_text '}']	*lokale Vereinbarungen*
		'{' source_text '}'	*Anweisungen zur Umsetzung*
pattern	=	[ident] ':'	*Selektor*
		[ident]	*Bezeichner der Klasse*
	|	ident	*Selektor und Bezeichner der Klasse*
	|	[[ident] ':']	*Selektor*
		(ident | string)	*Name des Knotens*
		'(' [pattern || ','] ')'	*Muster der Söhne*

Anhang B: Beispiel für die Anwendung von ESTRA

Anhang B1: Spezifikation in ESTRAL

```
TRANSFORMATION Trans

/*
 *      Quelltexterzeugung und Konstantenfaltung:
 *      - es wird MODULA-2 Quelltext ausgegeben
 *      - arithmetische und boolesche Konstanten werden gefaltet
 */

EXPORT  {
FROM    Tree        IMPORT  tTree;
}

GLOBAL  {
FROM    StdIO       IMPORT  BeginIO, CloseIO, WriteS, WriteI, WriteNl, WriteIdent;
FROM    Tree        IMPORT  tTree;
}

BEGIN   {
BeginIO;
}

CLOSE   {
CloseIO;
}

GRAMMAR             Tree

/* statements */

stats       =
| Stats               (stat, stats)
| Stats0     ()

stat        =
| If                  (bExpr, then: stats, else: stats)
| ':=':     Assign    (index, aExpr)

/* arithmetic expressions */

aExpr       =
| '+':      Plus      (lo: aExpr, ro: aExpr)

aExpr       ->
index       =
| aIdent     ()

aExpr       ->
aConst      =
| Const               ()
| '+'                 (aConst, aConst)

/* boolean expressions */

bExpr       =
| '<': Less (lo: aExpr, ro: aExpr)
| bIdent     ()

bExpr       ->
bConst      =
| True                ()
| False               ()
| '<'                 (aConst, aConst)
```

```
FUNCTION        Code                                /stats, stat, aExpr, bExpr/
/*      Ausgabe von MODULA-2 Quelltext              */
/* statements */
Stats   (stat, stats)                               COSTS   1
{       Code    (stat);
        WriteS  (';'); WriteNl;
        Code    (stats);                            }

Stats0  ()                                          COSTS   0
{}

If      (bConst, then: stats, else: stats)          CONDITION { BFold (bConst) = TRUE }
                                                            COSTS   0
{       Code    (then);                             }

If      (bConst, then: stats, else: stats)          CONDITION { BFold (bConst) = FALSE }
                                                            COSTS   0
{       Code    (else);                             }

If      (bExpr, then: stats, else: Stats0 ())       COSTS   3
{       WriteS  ('IF ');
        Code    (bExpr);
        WriteS  (' THEN'); WriteNl;
        Code    (then);
        WriteS  (' END;');                          }

If      (bExpr, then: Stats0 (), else: stats )      COSTS   3
{       WriteS  ('IF NOT (');
        Code    (bExpr);
        WriteS  (') THEN'); WriteNl;
        Code    (else);
        WriteS  (' END;');                          }

If      (bExpr, then: stats, else: stats)           COSTS   4
{       WriteS  ('IF ');
        Code    (bExpr);
        WriteS  (' THEN'); WriteNl;
        Code    (then);
        WriteS  (' ELSE'); WriteNl;
        Code    (else);
        WriteS  (' END;');                          }

':='    (index, aExpr)                              COSTS   1
{       Code    (index);
        WriteS  (' := ');
        Code    (aExpr);                            }

/* arithmetic expressions */
'+'     (lo: aExpr, ro: aExpr)                      COSTS   1
{       WriteS  ('( ');
        Code    (lo);
        WriteS  (' + ');
        Code    (ro);
        WriteS  (' )');                             }
```

```
aConst                                                  COSTS   1
{       WriteI      (AFold (aConst));                   }

aIdent  ()                                              COSTS   1
{       WriteIdent  (aIdent.ident);                     }

'<'     (lo:aExpr, ro:aExpr)                            COSTS   1
{       WriteS      ('( ');
        Code        (lo);
        WriteS      (' < ');
        Code        (ro);
        WriteS      (')');                              }

bIdent  ()                                              COSTS   1
{       WriteIdent  (bIdent.ident);                     }

bConst                                                  COSTS   1
{       IF BFold (bConst) THEN
          WriteS    ('TRUE');
        ELSE
          WriteS    ('FALSE');
        END;                                            }

FUNCTION        AFold       : INTEGER                   /aConst/
/*      Faltung arithmetischer Konstanten               */

Const   ()                                              COSTS   0
{       RETURN Const.value;                             }

'+'     (lo:aConst, ro:aConst)                          COSTS   0
{       RETURN AFold (lo) + AFold (ro);   }

FUNCTION        BFold       : BOOLEAN                   /bConst/
/*      Faltung boolescher Konstanten                   */

True    ()                                              COSTS   0
{       RETURN TRUE;                                    }

False   ()                                              COSTS   0
{       RETURN FALSE;                                   }

'<'     (lo: aConst, ro: aConst)                        COSTS   0
{       RETURN AFold (lo) < AFold (ro);   }
```

Anhang B2: Generierter Code

Im folgenden ist der von ESTRA generierte Code dargestellt.

Die Unterschiede, der beiden Versionen, die durch den optionalen Einsatz des Bottom-Up Pattern-Matching entstehen, werden so dargestellt, daß der Leser diese unmittelbar vergleichen kann.

```
(* ------ simple pattern-matching ------
  (A)
  ------------------------------------------- *)
  (B)
(* --- bottom-up pattern-matching --- *)
Beim Einsatz des Bottom-Up Pattern-Matching wird (B) generiert,
ansonsten wird (A) erzeugt.
```

```
DEFINITION MODULE Trans;

IMPORT Tree;
   (* line 3 Trans.estra *)

FROM      Tree          IMPORT    tTree;

(* ------ simple pattern-matching ------
  ------------------------------------------- *)
VAR TransTabName: ARRAY [0..127] OF CHAR;
(* --- bottom-up pattern-matching --- *)

PROCEDURE BeginTrans;
PROCEDURE DoTrans (yyt: Tree.tTree);
PROCEDURE CloseTrans;
END Trans.
```

```
IMPLEMENTATION MODULE Trans;

(* ------ simple pattern-matching ------
IMPORT SYSTEM, IO, Memory, Tree;
  ------------------------------------------- *)
IMPORT SYSTEM, IO, Memory, SystemIO, Tree;
(* --- bottom-up pattern-matching --- *)

   (* line 7 Trans.estra *)

FROM      StdIO      IMPORT    BeginIO, CloseIO, WriteS, WriteI, WriteNl, WriteIdent;
FROM      Tree       IMPORT    tTree;

CONST
  yyInfinite = 2147483643;

  yyBitsPerBitset = 32;
(* ------ simple pattern-matching ------
  yyCstats = 0;
  yyCstat = 1;
  yyCbExpr = 5;
  yyCindex = 3;
  yyCaExpr = 2;
  yyCaConst = 4;
  yyCbConst = 6;
  yyMaxClass = 6;
   ------------------------------------------- *)
  ySetSize = 22;
  yyMaxIndex = 26;
  yyCombSize = 117;
  yyStartState = 0;
(* --- bottom-up pattern-matching --- *)

  yyPoolSize = 10240;

TYPE
  yytBlockPtr = POINTER TO yytBlock;
  yytBlock =
  RECORD
    Successor: yytBlockPtr;
    Block: ARRAY [1..yyPoolSize] OF CHAR;
  END;

(* ------ simple pattern-matching ------
   ------------------------------------------- *)
  yyStateType = INTEGER;
  yySetType = ARRAY [0..ySetSize DIV yyBitsPerBitset] OF BITSET;
  yySetsType = ARRAY [0..yyMaxIndex] OF yySetType;
  yyCombType = ARRAY [0..yyCombSize] OF yyStateType;

(* --- bottom-up pattern-matching --- *)
  yyPCode = PROCEDURE (Tree.tTree);
  yyPAFold = PROCEDURE (Tree.tTree): INTEGER;
  yyPBFold = PROCEDURE (Tree.tTree): BOOLEAN;

  yyInfoPtr  = POINTER TO yyInfoType;
  yyInfoType =
    RECORD
(* ------ simple pattern-matching ------
      yyClasses: ARRAY [0..yyMaxClass DIV yyBitsPerBitset] OF BITSET;
   ------------------------------------------- *)
(* --- bottom-up pattern-matching --- *)
      Code: RECORD Cost: INTEGER; Proc: yyPCode; END;
      AFold: RECORD Cost: INTEGER; Proc: yyPAFold; END;
      BFold: RECORD Cost: INTEGER; Proc: yyPBFold; END;
```

```
   END;

VAR
  yySets: ySetsType;
  yyComb: yyCombType;
  yyInfo: yyInfoType;
  yyMatch: ARRAY [0..22] OF BOOLEAN;
  yyBlockList: yytBlockPtr;
  yyPoolFreePtr, yyPoolEndPtr: SYSTEM.ADDRESS;

(* ------ simple pattern-matching ------
PROCEDURE yyClass (yyt: Tree.tTree;Bit, Set: INTEGER): BOOLEAN;
VAR info: yyInfoPtr;
BEGIN
  info := yyt^.yyEstraInfo;
  RETURN Bit IN info^.yyClasses [Set];
END yyClass;

   ------------------------------------------- *)
(* --- bottom-up pattern-matching --- *)
PROCEDURE yyAlloc (): SYSTEM.ADDRESS;
VAR BlockPtr: yytBlockPtr;
BEGIN
  IF LONGINT (yyPoolEndPtr - yyPoolFreePtr) < SYSTEM.TSIZE (yyInfoType) THEN
    BlockPtr  := yyBlockList;
    yyBlockList  := Memory.Alloc (SYSTEM.TSIZE (yytBlock));
    yyBlockList^.Successor := BlockPtr;
    yyPoolFreePtr := SYSTEM.ADR (yyBlockList^.Block);
    yyPoolEndPtr  := yyPoolFreePtr + yyPoolSize;
  END;
  INC (yyPoolFreePtr, SYSTEM.ADDRESS (SYSTEM.TSIZE (yyInfoType)));
  RETURN yyPoolFreePtr - SYSTEM.ADDRESS (SYSTEM.TSIZE (yyInfoType));
END yyAlloc;

PROCEDURE yyReleaseHeap;
VAR BlockPtr: yytBlockPtr;
BEGIN
  WHILE yyBlockList # NIL DO
    BlockPtr:= yyBlockList;
    yyBlockList:= yyBlockList^.Successor;
    Memory.Free (SYSTEM.TSIZE (yytBlock), BlockPtr);
  END;
END yyReleaseHeap;

PROCEDURE Code (yyt: Tree.tTree);
VAR InfoPtr: yyInfoPtr;
BEGIN
  InfoPtr := yyInfoPtr (yyt^.yyEstraInfo);
  InfoPtr^.Code.Proc (yyt);
END Code;

PROCEDURE AFold (yyt: Tree.tTree): INTEGER;
VAR InfoPtr: yyInfoPtr;
BEGIN
  InfoPtr := yyInfoPtr (yyt^.yyEstraInfo);
  RETURN InfoPtr^.AFold.Proc (yyt);
END AFold;

PROCEDURE BFold (yyt: Tree.tTree): BOOLEAN;
VAR InfoPtr: yyInfoPtr;
BEGIN
  InfoPtr := yyInfoPtr (yyt^.yyEstraInfo);
  RETURN InfoPtr^.BFold.Proc (yyt);
END BFold;
```

```
PROCEDURE yyECode (yyt: Tree.tTree);
BEGIN
  IO.WriteS (IO.StdError, 'Function Code is not defined for this tree');
  IO.WriteNl (IO.StdError); IO.CloseIO; HALT;
END yyECode;

PROCEDURE yyEAFold (yyt: Tree.tTree): INTEGER;
BEGIN
  IO.WriteS (IO.StdError, 'Function AFold is not defined for this tree');
  IO.WriteNl (IO.StdError); IO.CloseIO; HALT;
END yyEAFold;

PROCEDURE yyEBFold (yyt: Tree.tTree): BOOLEAN;
BEGIN
  IO.WriteS (IO.StdError, 'Function BFold is not defined for this tree');
  IO.WriteNl (IO.StdError); IO.CloseIO; HALT;
END yyEBFold;

PROCEDURE yyF1Code (yyt: Tree.tTree);

BEGIN   (* line 72 Trans.estra *)
          Code      (yyt^.Stats.stat);
          WriteS    (';'); WriteNl;
          Code      (yyt^.Stats.stats);
END yyF1Code;

PROCEDURE yyF2Code (yyt: Tree.tTree);

BEGIN
END yyF2Code;

PROCEDURE yyF3Code (yyt: Tree.tTree);

BEGIN   (* line 90 Trans.estra *)
          Code      (yyt^.If.then);
END yyF3Code;

PROCEDURE yyF4Code (yyt: Tree.tTree);

BEGIN   (* line 99 Trans.estra *)
          Code      (yyt^.If.else);
END yyF4Code;

PROCEDURE yyF5Code (yyt: Tree.tTree);

BEGIN
END yyF5Code;

PROCEDURE yyF6Code (yyt: Tree.tTree);

BEGIN   (* line 113 Trans.estra *)
          WriteS    ('IF ');
          Code      (yyt^.If.bExpr);
          WriteS    (' THEN'); WriteNl;
          Code      (yyt^.If.then);
          WriteS    (' END;');
END yyF6Code;

PROCEDURE yyF7Code (yyt: Tree.tTree);

BEGIN   (* line 124 Trans.estra *)
          WriteS    ('IF NOT (');
          Code      (yyt^.If.bExpr);
          WriteS    (') THEN'); WriteNl;
          Code      (yyt^.If.else);
          WriteS    (' END;');
END yyF7Code;

PROCEDURE yyF8Code (yyt: Tree.tTree);
```

```
BEGIN   (* line 135 Trans.estra *)
            WriteS      ('IF ');
            Code        (yyt^.If.bExpr);
            WriteS      (' THEN'); WriteNl;
            Code        (yyt^.If.then);
            WriteS      (' ELSE'); WriteNl;
            Code        (yyt^.If.else);
            WriteS      (' END;');
END yyF8Code;

PROCEDURE yyF9Code (yyt: Tree.tTree);

BEGIN   (* line 148 Trans.estra *)
            Code        (yyt^.Assign.index);
            WriteS      (' := ');
            Code        (yyt^.Assign.aExpr);
END yyF9Code;

PROCEDURE yyF10Code (yyt: Tree.tTree);

BEGIN   (* line 157 Trans.estra *)
            WriteS      ('( ');
            Code        (yyt^.Plus.lo);
            WriteS      (' + ');
            Code        (yyt^.Plus.ro);
            WriteS      (' )');
END yyF10Code;

PROCEDURE yyF11Code (yyt: Tree.tTree);

BEGIN   (* line 168 Trans.estra *)
            WriteI      (yyt^.Const.value);
END yyF11Code;

PROCEDURE yyF12Code (yyt: Tree.tTree);

BEGIN   (* line 175 Trans.estra *)
            WriteS      ('( ');
            Code        (yyt^.Plus.lo);
            WriteS      (' + ');
            Code        (yyt^.Plus.ro);
            WriteS      (' )');
END yyF12Code;

PROCEDURE yyF13Code (yyt: Tree.tTree);

BEGIN   (* line 186 Trans.estra *)
            WriteIdent  (yyt^.aIdent.ident);
END yyF13Code;

PROCEDURE yyF14Code (yyt: Tree.tTree);

BEGIN   (* line 193 Trans.estra *)
            WriteS      ('TRUE');
END yyF14Code;

PROCEDURE yyF15Code (yyt: Tree.tTree);

BEGIN   (* line 200 Trans.estra *)
            WriteS      ('FALSE');
END yyF15Code;

PROCEDURE yyF16Code (yyt: Tree.tTree);

BEGIN   (* line 207 Trans.estra *)
            WriteS      ('( ');
            Code        (yyt^.Less.lo);
            WriteS      (' < ');
            Code        (yyt^.Less.ro);
```

```
            WriteS      (')');
END yyF16Code;

PROCEDURE yyF17Code (yyt: Tree.tTree);

BEGIN   (* line 218 Trans.estra *)
            WriteIdent  (yyt^.bIdent.ident);
END yyF17Code;

PROCEDURE yyF18AFold (yyt: Tree.tTree): INTEGER;

BEGIN   (* line 229 Trans.estra *)
            RETURN yyt^.Const.value;
END yyF18AFold;

PROCEDURE yyF19AFold (yyt: Tree.tTree): INTEGER;

BEGIN   (* line 236 Trans.estra *)
            RETURN AFold (yyt^.Plus.lo) + AFold (yyt^.Plus.ro);
END yyF19AFold;

PROCEDURE yyF20BFold (yyt: Tree.tTree): BOOLEAN;

BEGIN   (* line 247 Trans.estra *)
            RETURN TRUE;
END yyF20BFold;

PROCEDURE yyF21BFold (yyt: Tree.tTree): BOOLEAN;

BEGIN   (* line 254 Trans.estra *)
            RETURN FALSE;
END yyF21BFold;

PROCEDURE yyF22BFold (yyt: Tree.tTree): BOOLEAN;

BEGIN   (* line 261 Trans.estra *)
            RETURN AFold (yyt^.Less.lo) < AFold (yyt^.Less.ro);
END yyF22BFold;

PROCEDURE CostCode (yyt: Tree.tTree): INTEGER;
VAR
  InfoPtr: yyInfoPtr;
BEGIN
  InfoPtr := yyt^.yyEstraInfo;
  RETURN InfoPtr^.Code.Cost;
END CostCode;

PROCEDURE CostAFold (yyt: Tree.tTree): INTEGER;
VAR
  InfoPtr: yyInfoPtr;
BEGIN
  InfoPtr := yyt^.yyEstraInfo;
  RETURN InfoPtr^.AFold.Cost;
END CostAFold;

PROCEDURE CostBFold (yyt: Tree.tTree): INTEGER;
VAR
  InfoPtr: yyInfoPtr;
BEGIN
  InfoPtr := yyt^.yyEstraInfo;
  RETURN InfoPtr^.BFold.Cost;
END CostBFold;

(* ------ simple pattern-matching ------
PROCEDURE yyTraverse (yyt: Tree.tTree);
   ------------------------------------------- *)
PROCEDURE yyTraverse (yyt: Tree.tTree): yyStateType;
(* --- bottom-up pattern-matching --- *)
VAR
```

```
  state: yyStateType;
  match: POINTER TO ySetType;
  cost: INTEGER;
  info: yyInfoPtr;
  success: BOOLEAN;
BEGIN
  info := yyAlloc ();
  infoˆ := yyInfo;
  yytˆ.yyEstraInfo := info;

  CASE yytˆ.Kind OF
  | Tree.Stats:
(* ------ simple pattern-matching ------
     yyTraverse (yytˆ.Stats.stat);
     yyTraverse (yytˆ.Stats.stats);
     INCL (infoˆ.yyClasses [yyCstats DIV yyBitsPerBitset], yyCstats MOD yyBitsPerBitset);
   ---------------------------------------------- *)
     state := 0;
     state := yyComb [state + yyTraverse (yytˆ.Stats.stat)];
     state := yyComb [state + yyTraverse (yytˆ.Stats.stats)];
(* --- bottom-up pattern-matching --- *)

     cost := 1
     + CostCode (yytˆ.Stats.stat)
     + CostCode (yytˆ.Stats.stats);
     IF cost < infoˆ.Code.Cost THEN
       infoˆ.Code.Cost := cost;
       infoˆ.Code.Proc := yyF1Code;
     END;
  | Tree.Stats0:
(* ------ simple pattern-matching ------
     INCL (infoˆ.yyClasses [yyCstats DIV yyBitsPerBitset], yyCstats MOD yyBitsPerBitset);
   ---------------------------------------------- *)
     state := 1;
(* --- bottom-up pattern-matching --- *)

     cost := 0;
     IF cost < infoˆ.Code.Cost THEN
       infoˆ.Code.Cost := cost;
       infoˆ.Code.Proc := yyF2Code;
     END;
  | Tree.If:
(* ------ simple pattern-matching ------
     yyTraverse (yytˆ.If.bExpr);
     yyTraverse (yytˆ.If.then);
     yyTraverse (yytˆ.If.else);
     INCL (infoˆ.yyClasses [yyCstat DIV yyBitsPerBitset], yyCstat MOD yyBitsPerBitset);
     yyMatch [3] := yyClass (yytˆ.If.bExpr, yyCbConst MOD yyBitsPerBitset, yyCbConst DIV yyBitsPerBitset)
     & (         (* line 86 Trans.estra *)
                 BFold (yytˆ.If.bExpr) = TRUE          );
     yyMatch [4] := yyClass (yytˆ.If.bExpr, yyCbConst MOD yyBitsPerBitset, yyCbConst DIV yyBitsPerBitset)
     & (         (* line 95 Trans.estra *)
                 BFold (yytˆ.If.bExpr) = FALSE         );
     yyMatch [5] := (yytˆ.If.thenˆ.Kind = Tree.Stats0)
      & (yytˆ.If.elseˆ.Kind = Tree.Stats0);
     yyMatch [6] := (yytˆ.If.elseˆ.Kind = Tree.Stats0);
     yyMatch [7] := (yytˆ.If.thenˆ.Kind = Tree.Stats0);
   ---------------------------------------------- *)
```

```
    state := 1;
    state := yyComb [state + yyTraverse (yyt^.If.bExpr)];
    state := yyComb [state + yyTraverse (yyt^.If.then)];
    state := yyComb [state + yyTraverse (yyt^.If.else)];
    match := SYSTEM.ADR (yySets [state]);

    yyMatch [3] := (3 IN match^[0]) & (         (* line 86 Trans.estra *)
              BFold (yyt^.If.bExpr) = TRUE        );
    yyMatch [4] := (4 IN match^[0]) & (         (* line 95 Trans.estra *)
              BFold (yyt^.If.bExpr) = FALSE       );
    yyMatch [5] := (5 IN match^[0]);
    yyMatch [6] := (6 IN match^[0]);
    yyMatch [7] := (7 IN match^[0]);
(* --- bottom-up pattern-matching --- *)

    IF yyMatch [3] THEN
      cost := 0
      + CostBFold (yyt^.If.bExpr)
      + CostCode (yyt^.If.then);
      IF cost < info^.Code.Cost THEN
        info^.Code.Cost := cost;
        info^.Code.Proc := yyF3Code;
      END;
    END;

    IF yyMatch [4] THEN
      cost := 0
      + CostBFold (yyt^.If.bExpr)
      + CostCode (yyt^.If.else);
      IF cost < info^.Code.Cost THEN
        info^.Code.Cost := cost;
        info^.Code.Proc := yyF4Code;
      END;
    END;

    IF yyMatch [5] THEN
      cost := 0;
      IF cost < info^.Code.Cost THEN
        info^.Code.Cost := cost;
        info^.Code.Proc := yyF5Code;
      END;
    END;

    IF yyMatch [6] THEN
      cost := 3
      + CostCode (yyt^.If.bExpr)
      + CostCode (yyt^.If.then);
      IF cost < info^.Code.Cost THEN
        info^.Code.Cost := cost;
        info^.Code.Proc := yyF6Code;
      END;
    END;

    IF yyMatch [7] THEN
      cost := 4
      + CostCode (yyt^.If.bExpr)
      + CostCode (yyt^.If.else);
      IF cost < info^.Code.Cost THEN
        info^.Code.Cost := cost;
        info^.Code.Proc := yyF7Code;
      END;
    END;

    cost := 4
```

```
    + CostCode (yyt^.If.bExpr)
    + CostCode (yyt^.If.then)
    + CostCode (yyt^.If.else);
    IF cost < info^.Code.Cost THEN
      info^.Code.Cost := cost;
      info^.Code.Proc := yyF8Code;
    END;

  | Tree.Assign:
(* ------ simple pattern-matching ------
    yyTraverse (yyt^.Assign.index);
    yyTraverse (yyt^.Assign.aExpr);
    INCL (info^.yyClasses [yyCstat DIV yyBitsPerBitset], yyCstat MOD yyBitsPerBitset);
   ---------------------------------------------- *)
    state := 19;
    state := yyComb [state + yyTraverse (yyt^.Assign.index)];
    state := yyComb [state + yyTraverse (yyt^.Assign.aExpr)];
(* --- bottom-up pattern-matching --- *)

    cost := 1
    + CostCode (yyt^.Assign.index)
    + CostCode (yyt^.Assign.aExpr);
    IF cost < info^.Code.Cost THEN
      info^.Code.Cost := cost;
      info^.Code.Proc := yyF9Code;
    END;

  | Tree.Plus:
(* ------ simple pattern-matching ------
    yyTraverse (yyt^.Plus.lo);
    yyTraverse (yyt^.Plus.ro);
    INCL (info^.yyClasses [yyCaExpr DIV yyBitsPerBitset], yyCaExpr MOD yyBitsPerBitset);
    IF yyClass (yyt^.Plus.lo, yyCaConst MOD yyBitsPerBitset, yyCaConst DIV yyBitsPerBitset)
     & yyClass (yyt^.Plus.ro, yyCaConst MOD yyBitsPerBitset, yyCaConst DIV yyBitsPerBitset) THEN
      INCL (info^.yyClasses [yyCaConst DIV yyBitsPerBitset], yyCaConst MOD yyBitsPerBitset);
    END;

    yyMatch [12] := yyClass (yyt^.Plus.lo, yyCaConst MOD yyBitsPerBitset, yyCaConst DIV yyBitsPerBitset)
     & yyClass (yyt^.Plus.ro, yyCaConst MOD yyBitsPerBitset, yyCaConst DIV yyBitsPerBitset);
    yyMatch [19] := yyClass (yyt^.Plus.lo, yyCaConst MOD yyBitsPerBitset, yyCaConst DIV yyBitsPerBitset)
     & yyClass (yyt^.Plus.ro, yyCaConst MOD yyBitsPerBitset, yyCaConst DIV yyBitsPerBitset);
   ---------------------------------------------- *)
    state := 50;
    state := yyComb [state + yyTraverse (yyt^.Plus.lo)];
    state := yyComb [state + yyTraverse (yyt^.Plus.ro)];
    match := SYSTEM.ADR (yySets [state]);

    yyMatch [12] := (12 IN match^[0]);
    yyMatch [19] := (19 IN match^[0]);
(* --- bottom-up pattern-matching --- *)

    cost := 1
    + CostCode (yyt^.Plus.lo)
    + CostCode (yyt^.Plus.ro);
    IF cost < info^.Code.Cost THEN
      info^.Code.Cost := cost;
      info^.Code.Proc := yyF10Code;
    END;

    IF yyMatch [12] THEN
      cost := 1
      + CostCode (yyt^.Plus.lo)
      + CostCode (yyt^.Plus.ro);
      IF cost < info^.Code.Cost THEN
```

```
        info^.Code.Cost := cost;
        info^.Code.Proc := yyF12Code;
      END;
    END;

    IF yyMatch [19] THEN
      cost := 1
      + CostAFold (yyt^.Plus.lo)
      + CostAFold (yyt^.Plus.ro);
      IF cost < info^.AFold.Cost THEN
        info^.AFold.Cost := cost;
        info^.AFold.Proc := yyF19AFold;
      END;
    END;

  | Tree.aIdent:
(* ------ simple pattern-matching ------
    INCL (info^.yyClasses [yyCindex DIV yyBitsPerBitset], yyCindex MOD yyBitsPerBitset);
    INCL (info^.yyClasses [yyCaExpr DIV yyBitsPerBitset], yyCaExpr MOD yyBitsPerBitset);
  ---------------------------------------------- *)
    state := 17;
(* --- bottom-up pattern-matching --- *)

    cost := 1;
    IF cost < info^.Code.Cost THEN
      info^.Code.Cost := cost;
      info^.Code.Proc := yyF13Code;
    END;

  | Tree.Const:
(* ------ simple pattern-matching ------
    INCL (info^.yyClasses [yyCaConst DIV yyBitsPerBitset], yyCaConst MOD yyBitsPerBitset);
    INCL (info^.yyClasses [yyCaExpr DIV yyBitsPerBitset], yyCaExpr MOD yyBitsPerBitset);
  ---------------------------------------------- *)
    state := 16;
(* --- bottom-up pattern-matching --- *)

    cost := 1;
    IF cost < info^.Code.Cost THEN
      info^.Code.Cost := cost;
      info^.Code.Proc := yyF11Code;
    END;

    cost := 1;
    IF cost < info^.AFold.Cost THEN
      info^.AFold.Cost := cost;
      info^.AFold.Proc := yyF18AFold;
    END;

  | Tree.Less:
(* ------ simple pattern-matching ------
    yyTraverse (yyt^.Less.lo);
    yyTraverse (yyt^.Less.ro);
    INCL (info^.yyClasses [yyCbExpr DIV yyBitsPerBitset], yyCbExpr MOD yyBitsPerBitset);
    IF yyClass (yyt^.Less.lo, yyCbConst MOD yyBitsPerBitset, yyCbConst DIV yyBitsPerBitset)
     & yyClass (yyt^.Less.ro, yyCbConst MOD yyBitsPerBitset, yyCbConst DIV yyBitsPerBitset) THEN
      INCL (info^.yyClasses [yyCbConst DIV yyBitsPerBitset], yyCbConst MOD yyBitsPerBitset);
    END;

    yyMatch [22] := yyClass (yyt^.Less.lo, yyCaConst MOD yyBitsPerBitset, yyCaConst DIV yyBitsPerBitset)
     & yyClass (yyt^.Less.ro, yyCaConst MOD yyBitsPerBitset, yyCaConst DIV yyBitsPerBitset);
  ---------------------------------------------- *)
    state := 73;
    state := yyComb [state + yyTraverse (yyt^.Less.lo)];
```

```
    state := yyComb [state + yyTraverse (yyt^.Less.ro)];
    match := SYSTEM.ADR (yySets [state]);

    yyMatch [22] := (22 IN match^[0]);
(* --- bottom-up pattern-matching --- *)

    cost := 1
    + CostCode (yyt^.Less.lo)
    + CostCode (yyt^.Less.ro);
    IF cost < info^.Code.Cost THEN
      info^.Code.Cost := cost;
      info^.Code.Proc := yyF16Code;
    END;

    IF yyMatch [22] THEN
      cost := 0
      + CostAFold (yyt^.Less.lo)
      + CostAFold (yyt^.Less.ro);
      IF cost < info^.BFold.Cost THEN
        info^.BFold.Cost := cost;
        info^.BFold.Proc := yyF22BFold;
      END;
    END;

  | Tree.bIdent:
(* ------ simple pattern-matching ------
    INCL (info^.yyClasses [yyCbExpr DIV yyBitsPerBitset], yyCbExpr MOD yyBitsPerBitset);
  ---------------------------------------------- *)
    state := 23;
(* --- bottom-up pattern-matching --- *)

    cost := 1;
    IF cost < info^.Code.Cost THEN
      info^.Code.Cost := cost;
      info^.Code.Proc := yyF17Code;
    END;

  | Tree.True:
(* ------ simple pattern-matching ------
    INCL (info^.yyClasses [yyCbConst DIV yyBitsPerBitset], yyCbConst MOD yyBitsPerBitset);
    INCL (info^.yyClasses [yyCbExpr DIV yyBitsPerBitset], yyCbExpr MOD yyBitsPerBitset);
  ---------------------------------------------- *)
    state := 18;
(* --- bottom-up pattern-matching --- *)

    cost := 1;
    IF cost < info^.Code.Cost THEN
      info^.Code.Cost := cost;
      info^.Code.Proc := yyF14Code;
    END;

    cost := 0;
    IF cost < info^.BFold.Cost THEN
      info^.BFold.Cost := cost;
      info^.BFold.Proc := yyF20BFold;
    END;

  | Tree.False:
(* ------ simple pattern-matching ------
    INCL (info^.yyClasses [yyCbConst DIV yyBitsPerBitset], yyCbConst MOD yyBitsPerBitset);
    INCL (info^.yyClasses [yyCbExpr DIV yyBitsPerBitset], yyCbExpr MOD yyBitsPerBitset);
  ---------------------------------------------- *)
    state := 19;
(* --- bottom-up pattern-matching --- *)
```

```
    cost := 1;
    IF cost < info^.Code.Cost THEN
      info^.Code.Cost := cost;
      info^.Code.Proc := yyF15Code;
    END;

    cost := 0;
    IF cost < info^.BFold.Cost THEN
      info^.BFold.Cost := cost;
      info^.BFold.Proc := yyF21BFold;
    END;

  END;
(* ------ simple pattern-matching ------
   ---------------------------------------------- *)
  RETURN state;
(* --- bottom-up pattern-matching --- *)
END yyTraverse;

(* ------ simple pattern-matching ------
PROCEDURE BeginTrans;
BEGIN
   ---------------------------------------------- *)
PROCEDURE yyErrorCheck (i: INTEGER; s1, s2: ARRAY OF CHAR);
BEGIN
  IF i < 0 THEN
    IO.WriteS (IO.StdError, s1);
    IO.WriteS (IO.StdError, s2);
    IO.WriteNl (IO.StdError); IO.CloseIO; HALT;
  END;
END yyErrorCheck;

PROCEDURE BeginTrans;
VAR yyf: SystemIO.tFile; yyi: INTEGER;
BEGIN
  yyf := SystemIO.OpenInput (TransTabName);
  yyErrorCheck (yyf, 'cannot open ', TransTabName);
  yyi := SystemIO.Read (yyf, SYSTEM.ADR (yySets), SYSTEM.TSIZE (yySetsType));
  yyErrorCheck (yyi, 'cannot read ', TransTabName);
  yyi := SystemIO.Read (yyf, SYSTEM.ADR (yyComb), SYSTEM.TSIZE (yyCombType));
  yyErrorCheck (yyi, 'cannot read ', TransTabName);
  SystemIO.Close (yyf);
(* --- bottom-up pattern-matching --- *)
    (* line 12 Trans.estra *)

BeginIO;

END BeginTrans;

PROCEDURE DoTrans (yyt: Tree.tTree);
VAR state: yyStateType;
BEGIN
(* ------ simple pattern-matching ------
  yyTraverse (yyt);
   ---------------------------------------------- *)
  state := yyTraverse (yyt);
(* --- bottom-up pattern-matching --- *)
  Code (yyt);
  yyReleaseHeap;
END DoTrans;

PROCEDURE CloseTrans;
BEGIN
    (* line 16 Trans.estra *)
```

```
CloseIO;

END CloseTrans;

BEGIN
  TransTabName := 'Trans.tab';
  WITH yyInfo DO
(* ------ simple pattern-matching ------
    yyClasses [0] := {};
   ------------------------------------------- *)
(* --- bottom-up pattern-matching --- *)
    Code.Cost := yyInfinite;
    Code.Proc := yyECode;
    AFold.Cost := yyInfinite;
    AFold.Proc := yyEAFold;
    BFold.Cost := yyInfinite;
    BFold.Proc := yyEBFold;
  END;
  yyBlockList:= NIL;
  yyPoolFreePtr:= NIL;
  yyPoolEndPtr:= NIL;
END Trans.
```

Anhang B3: Bottom-Up Pattern-Matching Automat

Relevante Muster:

p_0 = Stats (:, :)
p_0 = Stats (:, :)
p_1 = Stats0 ()
p_2 = If (bConst, :, :)
p_3 = If (:, :, Stats0 ())
p_4 = If (:, Stats0 (), :)
p_5 = If (:, :, :)
p_6 = ':=' (:, :)
p_7 = '+' (:, :)
p_8 = aConst
p_9 = index
p_{10} = '<' (:, :)
p_{11} = bIdent ()
p_{12} = bConst
p_{13} = Const ()
p_{14} = '+' (aConst, aConst)
p_{15} = True ()
p_{16} = False ()
p_{17} = '<' (aConst, aConst)
p_{18} = :

Zustände:

q_0 = { p_0, p_{18} }
q_1 = { p_1, p_{18} }
q_2 = { p_2, p_3, p_4, p_5, p_{18} }
q_3 = { p_2, p_3, p_5, p_{18} }
q_4 = { p_2, p_4, p_5, p_{18} }
q_5 = { p_2, p_5, p_{18} }
q_6 = { p_3, p_4, p_5, p_{18} }
q_7 = { p_3, p_5, p_{18} }
q_8 = { p_4, p_5, p_{18} }
q_9 = { p_5, p_{18} }
q_{10} = { p_6, p_{18} }
q_{11} = { p_7, p_8, p_{14}, p_{18} }
q_{12} = { p_7, p_8, p_{18} }
q_{13} = { p_7, p_{18} }
q_{14} = { p_8, p_{13}, p_{18} }
q_{15} = { p_8, p_{18} }
q_{16} = { p_9, p_{18} }
q_{17} = { p_{10}, p_{12}, p_{17}, p_{18} }
q_{18} = { p_{10}, p_{12}, p_{18} }
q_{19} = { p_{10}, p_{18} }
q_{20} = { p_{11}, p_{18} }
q_{21} = { p_{12}, p_{15}, p_{18} }
q_{22} = { p_{12}, p_{16}, p_{18} }
q_{23} = { p_{12}, p_{18} }
q_{24} = { p_{18} }

Zustandsübergangsfunktionen:

Stats	$[q_2 - q_{10}, q_{24}]$	$[q_0, q_1, q_{24}]$		=	q_0
Stats0				=	q_1
If	$[q_{17}, q_{18}, q_{21}, q_{22}, q_{23}]$	$[q_0, q_{24}]$	$[q_0, q_{24}]$	=	q_5
If	$[q_{17}, q_{18}, q_{21}, q_{22}, q_{23}]$	$[q_0, q_{24}]$	$[q_1]$	=	q_3
If	$[q_{17}, q_{18}, q_{21}, q_{22}, q_{23}]$	$[q_1]$	$[q_0, q_{24}]$	=	q_4
If	$[q_{17}, q_{18}, q_{21}, q_{22}, q_{23}]$	$[q_1]$	$[q_1]$	=	q_2
If	$[q_{19}, q_{20}, q_{24}]$	$[q_0, q_{24}]$	$[q_0, q_{24}]$	=	q_9
If	$[q_{19}, q_{20}, q_{24}]$	$[q_0, q_{24}]$	$[q_1]$	=	q_7
If	$[q_{19}, q_{20}, q_{24}]$	$[q_1]$	$[q_0, q_{24}]$	=	q_8
If	$[q_{19}, q_{20}, q_{24}]$	$[q_1]$	$[q_1]$	=	q_6
':='	$[q_{16}, q_{24}]$	$[q_{11} - q_{16}, q_{24}]$		=	q_{10}
'+'	$[q_{11}, q_{12}, q_{14}, q_{15}]$	$[q_{11}, q_{12}, q_{14}, q_{15}]$		=	q_{11}
'+'	$[q_{11}, q_{12}, q_{14}, q_{15}]$	$[q_{13}, q_{16}, q_{24}]$		=	q_{13}
'+'	$[q_{13}, q_{16}, q_{24}]$	$[q_{11} - q_{16}, q_{24}]$		=	q_{13}
aIdent				=	q_{16}
Const				=	q_{14}
'<'	$[q_{11}, q_{12}, q_{14}, q_{15}]$	$[q_{11}, q_{12}, q_{14}, q_{15}]$		=	q_{17}
'<'	$[q_{11}, q_{12}, q_{14}, q_{15}]$	$[q_{13}, q_{16}, q_{24}]$		=	q_{19}
'<'	$[q_{13}, q_{16}, q_{24}]$	$[q_{11} - q_{16}, q_{24}]$		=	q_{19}
bIdent				=	q_{20}
True				=	q_{21}
False				=	q_{22}

Literaturhinweise

[AGT87] Alfred V. Aho, Mahadevan Ganapathi and Steven W. K. Tjiang, *Code Generation Using Tree Matching and Dynamic Programming*, AT&T Bell Laboratories, 1987.

[BMW87] Jürgen Börstler, Ulrich Möncke and Reinhard Wilhelm, Table Compression for Tree Automata, *Aachener Informatik-Berichte 12*, (1987), .

[Emm88] Helmut Emmelmann, *Automatische Erzeugung effizienter Codegeneratoren*, Diplomarbeit, GMD Forschungsstelle an der Universität Karlsruhe, Sep. 1988.

[Gro89] Josef Grosch, *Ast - A generator for Abstract Syntax Trees*, GMD Forschungsstelle an der Universität Karlsruhe, Feb. 1989.

[HoO82] Christoph M. Hoffmann and Michael J. O'Donnell, Pattern Matching in Trees, *J. ACM 29*, 1 (Jan. 1982), 68-95.

[KeR83] Brian W. Kernighan and Dennis M. Ritchie, *Programmieren in C*, Hanser München, 1983.

[Knu68] D. E. Knuth, Semantics of Context-Free Languages, *Mathematical Systems Theory 2*, 2 (June 1968), 127-146.

[MWW84] Ulrich Mönke, Beatrix Weisgerber and Reinhard Wilhelm, How to Implement a System for Manipulation of Attributed Trees, in *Informatik Fachberichte*, vol. 77, U. Ammann (ed.), Springer Verlag, Mar. 1984.

[Wir85] Niklaus Wirth, *Programmieren in Modula-2*, Springer Verlag, 1985.

5.12. Hauptbeschreibung 28
5.13. Funktionen 28
5.14. Typen 28
5.15. Attribute 29
5.16. Definitionbereich 29
5.17. Vorschriften 29
5.18. Muster 30
5.19. Anweisungen 30
5.20. Bedingungen 31
5.21. Kosten 31
5.22. Vereinbarungen 32
5.23. Integration 32

6. Implementierung von Transformationen durch ESTRA 34
6.1. Vorbereitung der Transformation 34
6.2. Darstellung von Funktionen und Vorschriften 34
6.3. Darstellung der Attribute 35
6.4. Speicherverwaltung 35
6.5. Berechnung der Klassen 36
6.6. Berechnung der zulässigen Vorschriften 36
6.7. Berechnung der kostengünstigsten Vorschriften 36

7. Bottom-Up Pattern-Matching mit einem Baumautomaten 38
7.1. Relevante Muster 38
7.2. Zustände 39
7.3. Zustandsübergangsfunktionen 40
7.4. Felder zur Darstellung der Zustandsübergangsfunktionen 41
7.5. Automaten zur Darstellung der Übergangsfunktionen 42
7.6. Realisierung der Automaten 44
7.7. Vermeidung unnötiger Zustände 45
7.8. Implementierung des Bottom-Up Pattern-Matching 45

8. Vollständigkeit 47
8.1. Einhaltung des Definitionsbereichs 47
8.2. Überdeckung des Definitionsbereichs 47
8.3. Terminierung 49

9. Schnittstellen des Transformators 51
9.1. Struktur der Eingabebäume 51
9.2. Schnittstelle des generierten Moduls 52

10. Vergleich mit anderen Werkzeugen 53
10.1. Twig 53
10.2. BEG 53

10.3. OPTRAN 54

11. Zusammenfassung 55

Anhang A: Syntax von ESTRAL 56

Anhang B: Beispiel für die Anwendung von ESTRA 57
Anhang B1: Spezifikation in ESTRAL 57
Anhang B2: Generierter Code 60
Anhang B3: Bottom-Up Pattern-Matching Automat 73

Literaturhinweise 76