Ein Evaluatorgenerator für zwei heuristische Teilklassen Sequentiell Orientierbarer Erweiterter Affixgrammatiken

Diplomarbeit
März 1998

Denis Kuniß

Technische Universität Berlin
Fachbereich Informatik
Institut für Angewandte Informatik
FG Programmiersprachen und Compiler

Inhaltsverzeichnis

1 Vorwort

Die vorliegende Diplomarbeit ist der vorläufige Abschluß einer mehrjährigen intensiven Beschäftigung mit Theorie und Praxis der Compilergenerierung. Die Auseinandersetzung mit dieser Thematik bildete den Schwerpunkt meines Studiums und wurden durch die FG Programmiersprachen und Compiler des Fachbereiches Informatik der TU Berlin unter Leitung von Prof. Dr. Bleicke Eggers anregend begleitet. In dieser Hinsicht möchte ich insbesondere Sönke Kannapinn und Mario Kröplin danken, die meine Faszination für dieses Teilgebiet der Informatik durch ausgezeichnete Vorlesungen und didaktisch hervorragende Seminare und Tutorien geweckt haben. Ich verdanke ihnen einen Großteil meines ingenieur-technischen Wissens.

Prof. Eggers möchte ich für sein Bestreben danken, in seinen Vorlesungen eine ganzheitliche Sicht der Dinge, die sich nicht nur auf das Gebiet der Informatik beschränkt, vermittelt und Denkanstöße über die Verantwortung des Ingenieurs bzw. Wissenschaftlers für die Zukunft gegeben zu haben. Seinen Vorlesungen zur gesellschaftlichen Relevanz der Informatik verdanke ich, daß ich meine Neigung zur Philosophie wiederentdeckt habe und für gesellschaftskritische Betrachtungen sensibilisiert wurde.

Dieses Dokument richtet sich an Leser, die mit der Problematik der Compilergenerierung und der formalen Sprachen im allgemeinen und der geordneten Attributgrammatiken [Kastens] und Erweiterten Affixgrammatiken [Watt] im speziellen vertraut sind. Weiterhin wird eine vorhergehende Lektüre der Diplomarbeit von Demuth und Weber [DeWe] oder des Forschungsberichtes [DeWeKrKa] empfohlen.

Diese Arbeit widme ich meiner Frau Julia.

Berlin, den 26. März 1998.

Denis Kuniß

2 Einleitung

Compiler realisieren im Grunde komplexe partielle Funktionen: sie bilden Elemente einer wohldefinierten Teilmenge aller bildbaren Zeichenketten in andere Zeichenketten ab; Definitions- und Wertebereich eines als Funktion aufgefaßten Compilers sind folglich formale Sprachen. Die Implementierung eines Compilers ergibt sich also im Prinzip aus der Gegenüberstellung von Sätzen der Quellsprache und abzubildenden Sätzen der Zielsprache. Die Anzahl der Sätze in einer formalen Sprache ist jedoch potentiell unendlich, ein Compiler dieser Form wäre also nicht aufschreibbar. Nun lassen sich die Sätze vieler Sprachen so strukturieren, daß sich daraus eine endliche Struktur aller Sätze einer Sprache ableiten läßt. Diese Strukturen, Grammatiken genannt, könnten nun, da sie endlich sind, konstruktiv für die Gegenüberstellung in einem Compiler genutzt werden.

Leider ist diese Gegenüberstellung in „handgeschriebenen“ Compilern nicht nachvollziehbar. Damit ist aber die eigentliche Funktion eines Compilers nicht transparent. Auch das beste und genaueste Handbuch eines Compilers beschreibt seine Funktion nicht vollständig. Die einzige vollständige Beschreibung der Funktionalität eines Compilers ist seine Codierung. Wem aber will man schon zumuten, zur Verifikation den Quellcode eines Compilers zu lesen, zumal die eigentliche Übersetzungsstruktur von Programmierstilen, Unzulänglichkeiten der verwendeten Programmiersprache oder Optimierungen überdeckt wird, wenn der Quellcode überhaupt zur Verfügung steht. Wünschenswert wäre ein Formalismus, in dem Strukturen der Quell- und Zielsprache in einer einfachen, überschaubaren Form gegenübergestellt werden und aus dem automatisch ein Compiler erzeugt werden kann.

Die von Knuth eingeführten *Attributgrammatiken* (AGen) stellen einen Formalismus zur Verfügung, der nicht nur zur Beschreibung von Programmiersprachen verwendet werden, sondern auch als Grundlage für die automatische Erzeugung von Compilern dienen kann. Dieser Formalismus erfordert als offenes Kalkül jedoch die Benutzung einer weiteren Spezifikations- oder Programmiersprache zur Beschreibung der Kontextbedingungen und der Semantik einer Sprache. Dies erschwert das Verstehen des Compilers und verdeckt wie im „handgeschriebenen“ Compiler seine eigentliche Funktionalität.

Die von Watt vorgestellten *Erweiterten Affixgrammatiken* (EAGen) stellen ein geschlossenes Kalkül dar und gleichen diesen Nachteil aus [Watt]. Syntax und Semantik einer Programmiersprache werden in einem Formalismus beschrieben, der auf alle nicht relevanten Informationen - die Optimierung, verwendete Implementierungsprache und anderes betreffend - verzichtet. Die Gegenüberstellung der Quell- und Zielsprache ist explizit und beschreibt die durch den Compiler zu realisierende Transformationsfunktion eindeutig und in einfacher Form. 1984 wurde von Schröer am Fachbereich Informatik der TU Berlin ein Compilergenerator namens *Eta* auf Basis der Erweiterten Affixgrammatiken implementiert [Schröer], der die Vorteile dieses Konzeptes zur automatischen Generierung von Übersetzern bewiesen hat und seitdem mit vielen Erweiterungen auch im Rahmen von Lehrveranstaltungen eingesetzt wurde. Eine dieser Erweiterungen überträgt das Prinzip der *geordneten* Attributgrammatiken (OAGen, [Kastens]) in die Welt der Erweiterten Affixgrammatiken und wurde von Kutza 1989 in den Compilergenerator Eta integriert [Kutza]. Die OAGen sind eine polynomiell zu handhabende Teilklasse der von Engelfriet eingeführten *multi-visit* AGen [Engelfriet]. Für multi-visit AGen kann ein laufzeit- und größeneffizienter Evaluator angegeben werden, leider ist die Entscheidung, ob ein multi-visit AG vorliegt, NP-vollständig.

OAGen werden über die polynomielle Berechenbarkeit einer totalen Ordnung über dem Attributen jedes Grammtiksymbols X, so daß in jedem möglichen attributierten Ableitungsbaum die Attributinstanzen einer Instanz von X immer gemäß dieser Ordnung ausgewertet werden können, definiert. Es treten jedoch immer wieder praxisrelevante AGen auf, für die sich eine solche totale Ordnung der Attribute nicht berechnen läßt. Kastens gibt verschiedene Möglichkeiten an, um die Attribute einer AG doch noch anordnen zu können [KaHuZi], die jedoch viele problematische AGen unbeachtet lassen. Kröplin und Kannapinn haben in ihrer Arbeit eine Verallgemeinerung einer dieser Ideen entwickelt [KröpKann]. Die von ihnen angegebenen *sequentiell orientierbaren* AGen (SOAGen) sind eine Teilklasse der mult-visit AGen, und die Entscheidung, ob eine AG eine SOAG ist, ist ebenfalls NP-vollständig. Die Autoren stellen ein Verfahren vor, das den für die NP-Vollständigkeit „verantwortlichen“ Nichtdeterminismus durch einen ad-hoc-Determinismus ersetzt, wodurch für alle OAGen und viele Nicht-OAGen eine Attributauswertungsreihenfolge berechnet werden kann.

Die Aufgabe der vorliegenden Diplomarbeit bestand darin, das Prinzip der SOAGen auf die Erweiterten Affixgrammatiken zu übertragen und einen Evaluatorgenerator für das skizzierte Verfahren zu implementieren. Ausgangspunkt ist dabei der Compilergenerator *Epsilon*, der an der TU Berlin unter Anleitung von Kröplin und Kannapinn im Rahmen der Diplomarbeit von Demuth und Weber entstanden ist

[DeWe]. Er ersetzt das in seiner Konzeption „in die Jahre gekommene“ und mit den extrem gering gewordenen Personalressourcen nicht weiter pfleg- und beherrschbare Vorgängersystem *Eta* und stellt insbesondere einen Parsergenerator mit Scanner zur Verfügung.

Die Arbeit beschreibt die Implementierung des Prinzips der SOAGen im Konzept der Erweiterten Affixgrammatiken und lehnt sich eng an sie an. Um dieses Anliegen zu unterstreichen und die Komplexität der Implementierung zu verdeutlichen, werden die Quelltexte kommentiert und im Original mit abgedruckt. Damit entfällt die Beschreibung abstrakter Algorithmen. Diese Arbeit kann die unterliegende Theorie nur ansatzweise im Sinne der Implementierung beschreiben, für eine umfassende Behandlung muß auf entsprechende Quellen verwiesen werden [KröpKann].

Die nachfolgende Dokumentation beginnt mit einem allgemeinen Überblick der Implementierung und verwendeter Hilfsmoduln. Kapitel 3 führt in die Terminologie der EAGen ein und dokumentiert die zentralen Datenstrukturen. Die Kapitel 4, 5 und 6 stellen die Moduln vor, welche die Voraussetzung zur Generierung eines Evaluators schaffen; dies umfaßt die Berechnung von Auswertungsreihenfolgen, Besuchsreihenfolgen und Optimierungsinformationen. Kapitel 7 nimmt eine Aufwandsanalyse der vorgestellten Algorithmen vor, und im letzten Kapitel wird die eigentliche Codegenerierung behandelt.

Der Anhang enthält ein reduziertes Beispiel mit dem durch den Evaluatorgenerator erzeugten Modul, sowie eine Analyse zweier Nicht-OEAG, die vom Generator als SOEAG erkannt werden.

3 Allgemeines zur Implementierung

In Anlehnung an den vorhandenen Compilergenerator Epsilon erfolgte die Programmierung mit der Programmiersprache Oberon [ReiWi] im gleichnamigen Betriebssystem. Um über eine einheitliche Arbeits- und Testumgebung zu verfügen, wurde als Zielsprache der Generierung ebenfalls Oberon gewählt.

Im Compilergenerator Epsilon sind die zwei Basismoduln `IO` und `eSets` enthalten, die auch in dieser Systemerweiterung wieder benutzt werden. Eine Beschreibung dieser beiden Moduln findet man in [DeWe]. Die dort vorgeschlagenen Programmiertechniken der Zusammenfassung von Programmobjekten eines Typs in einem Feld wird weiterhin verwendet. Datentyperweiterungen erfolgen durch Anlegen eines Parallelfeldes. Zu jedem Feld *F* gibt es ein Konstante `first`*F*, die den ersten verwendbaren Index des Feldes symbolisiert, und eine Variable `Next`*F*, die auf den ersten leeren Feldeintrag verweist. Expandiert werden die Felder durch modul-lokale Prozeduren mit dem Namen `Expand`. Undefinierte Verweise auf diese Felder werden durch die Konstante `nil` dargestellt.

3.1 Listen und Stacks

Das Modul `eAList` implementiert Listen als dynamisch erweiterbare Felder und realisiert folgende Schnittstelle:

```
DEFINITION eALists;

    CONST
      firstIndex = 0;

    TYPE
      AList = POINTER TO AListDesc;
      AListDesc = RECORD
        Last: INTEGER;
        Elem: OpenList;
      END;

    PROCEDURE Append (VAR List: AList; Value: INTEGER);
    PROCEDURE Delete (VAR List: AList; Index: INTEGER);
    PROCEDURE IndexOf (VAR List: AList; Value: INTEGER): INTEGER;
    PROCEDURE New (VAR List: AList; Len: INTEGER);
    PROCEDURE Reset (VAR List: AList);

END eALists.
```

Die Prozeduren `New` und `Reset` erzeugen eine neue Liste bzw. leeren diese. Die Prozedur `Append` fügt an das Ende der Liste ein Element an. Die Prozedur `Delete` löscht eine Element aus der Liste, indem die Position des zu löschenden Elementes mit dem letzten Element der Liste überschrieben und die Liste um ein Element verkürzt wird. Damit ist die Aktion von konstantem Aufwand, verändert jedoch die Reihenfolge innerhalb der Liste. Die Funktion `IndexOf` liefert den Listenindex eines Elementes zurück. Das Element wird durch lineare Suche ermittelt.

Weiterhin wurde ein Modul `eStacks` zur Beschreibung von Kellerspeichern implementiert. Dieser Modul basiert auf dem vorangehenden Modul und realisiert die bekannten Kellerspeicherprozeduren:

```
DEFINITION eStacks;

    IMPORT eALists;

    TYPE
      Stack = POINTER TO RECORD (eALists.AListDesc) END;

    PROCEDURE IsEmpty (S: Stack): BOOLEAN;
    PROCEDURE New (VAR S: Stack; Len: INTEGER);
    PROCEDURE Pop (VAR S: Stack; VAR Val: INTEGER);
    PROCEDURE Push (VAR S: Stack; Val: INTEGER);
    PROCEDURE Reset (VAR S: Stack);
    PROCEDURE Top (VAR S: Stack; VAR Val: INTEGER);

END eStacks.
```

3.2 Unterschiedliche Implementierungen von Mengen

Der Basismodul `eSets` erwies sich in aufwandskritischen Abschnitten der Implementierung als ungeeignet. Insbesondere lies sich auf die in einer Menge enthaltenen Elemente nicht effizient als Liste zugreifen, dafür hätte der die Menge darstellende Bit-Vektor, selbst bei dünn besetzten Mengen, immer vollständig durchlaufen werden müssen. Es ist naheliegend, die Datenstruktur des Basismoduls um eine Liste der in der Menge enthaltenen Elemente zu erweitern. Dies ermöglicht einen effizienten Zugriff und erhöht den Speicherbedarf bei dünn besetzten Mengen nur unwesentlich. Dieser Ansatz wurde im Modul `eBSets` mit der folgender Schnittstelle realisiert:

```
DEFINITION eBSets;

    IMPORT eALists;

    CONST
      firstIndex = 0;

    TYPE
      BSet = POINTER TO BSetDesc;
      BSetDesc = RECORD
        Max: INTEGER;
        List: eALists.AList;
      END;

    PROCEDURE Delete (VAR S: BSet; Elem: INTEGER);
    PROCEDURE In (S: BSet; Elem: INTEGER): BOOLEAN;
    PROCEDURE Insert (VAR S: BSet; Elem: INTEGER);
    PROCEDURE New (VAR S: BSet; MaxElem: INTEGER);
    PROCEDURE Reset (VAR S: BSet);

END eBSets.
```

Die Datenstruktur `BSetDesc` enthält einen in der Schnittstelle nicht sichtbaren Bit-Vektor, der wie folgt deklariert ist:

```
BitVektor: eSets.OpenSet;
```

Die Prozedur `New` erzeugt eine Menge konstanter Größe. `Reset` löscht den Inhalt einer Menge, die allozierte Datenstruktur bleibt erhalten und kann wiederverwendet werden. Die Funktion `In` prüft, ob ein Element in der Menge enthalten ist. Auf die Liste der in der Menge S enthaltenen Elemente kann effektiv durch Konstrukte der Art `S.List[eBSets.firstIndex]` bis `S.List[S.List.Last]` zugegriffen werden. Die Prozeduren `Insert` und `Delete` fügen ein Element in die Menge ein bzw. löschen es aus der Menge. Das Löschen eines Elementes ist in dieser Implementierung von linearem Aufwand, da in der Liste der enthaltenen Elemente danach gesucht werden muß.

Der lineare Aufwand für das Löschen eines Elementes kann den Aufwand eines Algorithmus unnötigerweise um eine Potenz erhöhen. Deshalb wurden im Modul `eASets` Mengen implementiert, die das Löschen eines

Elementes in konstanter Zeit erlauben und für die die Liste der in einer Menge enthaltenen Elemente effizient abrufbar ist. Die Schnittstelle ist dem vorangegangenen Modul ähnlich:

```
DEFINITION eASets;

    IMPORT eALists;

    CONST
      firstIndex = 0;

    TYPE
      ASet = POINTER TO ASetDesc;
      ASetDesc = RECORD
        Max: INTEGER;
        List: eALists.AList;
      END;

    PROCEDURE Delete (VAR S: ASet; Elem: INTEGER);
    PROCEDURE In (S: ASet; Elem: INTEGER): BOOLEAN;
    PROCEDURE Insert (VAR S: ASet; Elem: INTEGER);
    PROCEDURE IsEmpty (VAR S: ASet): BOOLEAN;
    PROCEDURE New (VAR S: ASet; MaxElem: INTEGER);
    PROCEDURE Reset (VAR S: ASet);
    PROCEDURE Test;

END eASets.
```

Hinzugekommen ist lediglich die Funktion `IsEmpty`, die anzeigt, ob eine Menge leer ist. Im Gegensatz zu `eBSets` wurde der Bit-Vektor ganz weggelassen. Die Elemente werden in der Reihenfolge ihres Einfügens in die Menge in die Liste `ASet.List` eingetragen, wobei `ASet.List.Last` auf das letzte eingetragenen Element verweist. Die Liste ist als Feld realisiert und ergibt sich aus den Feldeinträgen von `ASets.List[eASets.firstIndex]` bis `ASets.List[ASet.List.Last]`. Ist der Wert `i` eines Elementes, das in die Menge eingefügt werden soll, größer als `ASet.List.Last`, so wird `i` an das Ende der Liste gehängt und in der Feldposition `ASet.List[i]` ein Zeiger auf das Ende der Liste eingetragen. Ist der Wert `i` des Elementes kleiner-gleich `ASet.List.Last`, so wird es an der Position `ASet.List[i]` eingetragen. Das Element, das vorher an seiner Position gestanden hat, wird an das Ende der Liste verschoben und sein Zeiger entsprechend umgelegt. Durch diese Vorgehensweise wird das Feld in zwei Hälften geteilt. Im ersten Teil sind die in der Menge enthaltenen Elemente aufgelistet. Der zweite Teil enthält für alle Mengenelemente, deren Wert größer als `ASet.List.Last` ist, einen Wert größer Null auf ihrer Feldposition und sonst den Wert `noelem`. Bei Änderungen des Inhaltes der Menge `S` durch die Schnittstellenprozeduren werden im Feld `ASet.List` folgende Invarianten erhalten.
Für alle Feldindizes `i`, die kleiner-gleich dem Datenelement `ASet.List.Last` sind, gilt:

$$\texttt{ASet.List[i]=i} \Leftrightarrow \texttt{i}\in\texttt{S} \wedge \texttt{ASet.List[i]}\neq\texttt{i} \Rightarrow \texttt{ASet.List[i]}\in\texttt{S}$$

und für alle Feldindizes `i`, die größer als das Datenelement `ASet.List.Last` sind, gilt:

$$\texttt{i}\in\texttt{S} \Leftrightarrow \texttt{ASet.List[i]}\neq\texttt{noelem} \wedge \texttt{i}\notin\texttt{S} \Leftrightarrow \texttt{ASet.List[i]=noelem}$$

Aus diesen Invarianten kann sehr einfach abgeleitet werden, wann ein Element in einer Menge enthalten ist. Dies macht den Bit-Vektor überflüssig. Die Prozeduren `Insert` und `Delete` können mit konstantem Aufwand realisiert werden.

Zusammenfassend kann festgestellt werden, daß der Modul `eASets` besonders gut zur Implementierung von dicht besetzten Mengen geeignet ist, während der Modul `eBSets` bei dünn besetzten Mengen, wenn kein zeitkritisches Löschen von Elementen erforderlich ist, verwendet werden sollte. Beide Moduln können nur konstante Mengen abbilden, dynamische Erweiterungen während der Laufzeit sind nicht möglich.

Beide Mengenimplementierungen benutzen das Modul `eAList`. Jedoch handelt es sich um eine vollständige Kapselung - man sollte auf die Listen-Datenstrukturen der Mengen keinesfalls die Schnittstellenprozeduren des Moduls `eAList` anwenden, da dadurch die Datenstruktur inkonsistent werden kann. Die Option der dynamischen Erweiterbarkeit von Listen wird in den Implementierungen der Mengen nicht benutzt.

Für alle generierten Compiler wurde zusätzlich der unabhängige Modul `eLIStacks` implementiert, der Datenelemente vom Typ `LONGINT` verwaltet.

3.3 Systemstruktur

Der SOAG-Evaluatorgenerator erweitert den ursprünglichen Compilergenerator Epsilon um die in Abbildung 3-1 dargestellten Moduln. Die Pfeile geben wesentliche Importbeziehungen zwischen den Moduln an. Die Moduln `eALists`, `eStacks`, `eASets` und `eBSets` werden mehrfach verwendet und implementieren Listen, Mengen und Kellerspeicher, die in beliebiger Anzahl instanziiert werden können. Die Moduln mit dem Präfix „SOAG“ beschreiben den eigentlichen SOAG-Evaluatorgenerator. Der Modul `SOAG` enthält die zentralen Datenstrukturen des Generators. Im Modul `SOAGPartition` wird die Affixpartition der analysierten Grammatik berechnet und damit entschieden, ob die Generierung eines Evaluators möglich ist. In

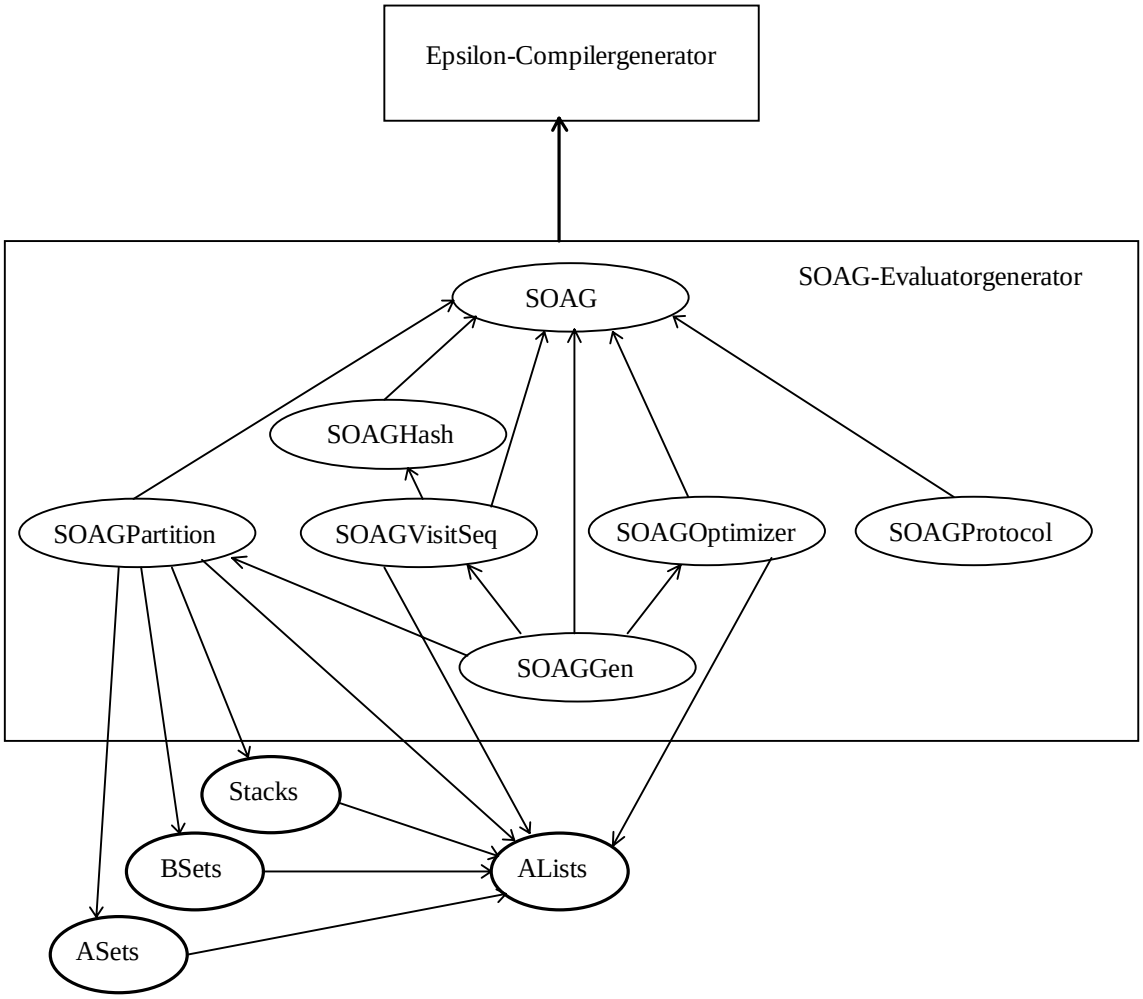


Abbildung 3-1: Systemstruktur

`SOAGVisitSeq` werden mit Hilfe des Moduls `SOAGHash`, das eine Hash-Tabelle implementiert, die Visit-Sequenzen für den Evaluator berechnet. Optional werden durch den Modul `SOAGOptimizer` Informationen bereitgestellt, die eine Optimierung des generierten Evaluators durch Speicherung von Affixvariablen in Kellerspeichern und globalen Variablen erlauben. Das Modul `SOAGGen` generiert aus den Visit-Sequenzen den Compiler in der Sprache Oberon-2. Das Modul `SOAGProtocol` dient vor allem zum Protokollieren von zentralen Datenstrukturinhalten während der Entwicklung.

Wie im ursprünglichen Compilergenerator Epsilon wird allen Moduln ein „e“ vorangestellt, um Namenskonflikte mit dem Oberon-System zu vermeiden.

3.4 Implementierungen

3.4.1 eALists.Mod

```
MODULE eALists; (* 1.11 01.1.98 Denis Kuniss *)

CONST firstIndex* = 0;

TYPE
```

```
  OpenList = POINTER TO ARRAY OF INTEGER;
  AListDesc* = RECORD
    Last*: INTEGER;   (* zeigt auf das letzte gueltige Element,
                         <0, wenn Liste leer *)
    Elem*: OpenList;  (* enthaelt die Liste *)
  END;
  AList* = POINTER TO AListDesc;

PROCEDURE Expand( VAR List: AList );
VAR List1: OpenList;
  i: INTEGER;
BEGIN
  IF LEN( List.Elem^ ) < MAX( INTEGER ) DIV 2 THEN
    NEW( List1, 2 * LEN( List.Elem^ ) + 1 )
  ELSE HALT(99) END;
  FOR i := firstIndex TO List.Last DO List1[i] := List.Elem[i] END;
  List.Elem := List1;
END Expand;



PROCEDURE New*( VAR List: AList; Len: INTEGER );
(* IN:  Referenzvariable der Liste, anfaengliche Laenge der Liste
  OUT:  Referenzvariable der Liste
  SEM:  Anlegen einer neune Liste (Speicherplatzreservierung)
  SEF:  - *)
BEGIN
  NEW( List );
  List.Last := -1;
  NEW( List.Elem, Len );
END New;

PROCEDURE Reset*( VAR List: AList );
(* IN:  Referenzvariable der Liste
  OUT:  Referenzvariable der Liste
  SEM:  Loeschen des Listeninhalts
  SEF:  - *)
BEGIN
  List.Last := -1
END Reset;

PROCEDURE Delete*( VAR List: AList; Index: INTEGER );
(* IN:  Referenzvariable der Liste, Index des zu loeschenden Elements
  OUT:  Referenzvariable der Liste
  SEM:  Loeschen eines Elements
  SEF:  Auf die Reihenfolge innerhalb der Liste*)
BEGIN
  IF Index >= firstIndex THEN
    List.Elem[Index] := List.Elem[List.Last];
    DEC( List.Last )
  END
END Delete;

PROCEDURE Append*( VAR List: AList; Value: INTEGER );
(* IN:  Referenzvariable der Liste, Wert des Elements
  OUT:  Referenzvariable der Liste
  SEM:  Anhaengen des Elements am Ende der Liste
  SEF:  - *)
BEGIN
  IF (List.Last + 1) >= LEN( List.Elem^ ) THEN Expand( List ) END;
  INC( List.Last );
  List.Elem[List.Last] := Value;
END Append;

PROCEDURE IndexOf*( VAR List: AList; Value: INTEGER ):INTEGER;
(* IN:  Referenzvariable der Liste, gesuchter Wert
  OUT:  Referenzvariable der Liste, Index des ges. Wertes
  SEM:  Liefert den Index des gesuchten Wertes; Nach einer
        Delete-Aktion ist dieser Wert inkonsistent!
  SEF:  - *)
VAR i: INTEGER;
BEGIN
  i := firstIndex;
  WHILE ((List.Elem[i] # Value) & (i <= List.Last)) DO INC( i ) END;
  IF i <= List.Last THEN RETURN i ELSE RETURN -1 END
END IndexOf;


END eALists.
```

3.4.2 eAStacks.Mod

```
MODULE eStacks;  (* Denis Kuniss 1.02 05.03.98 *)

IMPORT ALists := eALists, SYSTEM, IO := eIO;

TYPE Stack* =  POINTER TO RECORD (ALists.AListDesc) END;

PROCEDURE New*( VAR S: Stack; Len: INTEGER );
BEGIN
  ALists.New( SYSTEM.VAL( ALists.AList, S), Len )
END New;

PROCEDURE Reset*( VAR S: Stack );
BEGIN
  ALists.Reset( SYSTEM.VAL( ALists.AList, S ))
END Reset;

PROCEDURE Push*( VAR S: Stack; Val: INTEGER );
BEGIN
  ALists.Append( SYSTEM.VAL( ALists.AList, S ), Val )
END Push;

PROCEDURE Pop*( VAR S: Stack; VAR Val: INTEGER );
BEGIN
  Val := S.Elem[S.Last];
  ALists.Delete( SYSTEM.VAL( ALists.AList, S ), S.Last );
END Pop;

PROCEDURE Top*( VAR S: Stack): INTEGER;
BEGIN
  RETURN S.Elem[S.Last];
END Top;

PROCEDURE IsEmpty*( S: Stack ): BOOLEAN;
BEGIN
  RETURN S.Last < ALists.firstIndex
END IsEmpty;

END eStacks.
```

3.4.3 eBSets.Mod

```
MODULE eBSets; (* 2.0 29.3.97 Denis Kuniss *)

(* Spezifikation:
    Dieses Modul stellt einen Mengentyp zur Verfuegung, der effektiv als Liste implementiert
ist,
    und trotzdem einen Test, ob ein Element in der Menge enthalten ist, mit konstantem
    Aufwand erlaubt.
    Es duerfen ganzzahlige Elemente von 0 bis MaxElement eingefuegt werden.
    Auf die Elemente der Menge kann in Form von Set.List[firstIndex]  bis  Set.List[S.Last]
    zugegriffen werden
  Einschraenkungen/Nachteile:
    - Groesse der Menge ist fest nach der Initialisierung
    - Loeschen eines Elements ist von linearem Aufwand
  Vorteile/Anzeigen:
    - es wird fuer die Abspeicherung der Elemente nur soviel Platz benoetigt, wie Elemente
      in der Menge enthalten sind.
    Es ist angezeigt, die DS zu verwenden, wenn sehr duenn besetzte Mengen verwendet
    werden (Einsparung des Bitvektors moeglich - siehe eASets.Mod)
*)

IMPORT IO := eIO, Sets := eSets, ALists := eALists;

CONST
  firstIndex* =  ALists.firstIndex;

TYPE
  BSetDesc* = RECORD
    Max*: INTEGER;
    BitVektor: Sets.OpenSet;
    List*: ALists.AList
  END;
  BSet* = POINTER TO BSetDesc;


PROCEDURE New*( VAR S: BSet; MaxElem: INTEGER );
(*IN: Referenzvariable der Liste, anfaengliche Laenge der Liste
 OUT: Referenzvariable der Liste
 SEM: Anlegen einer neuen Liste (Speicherplatzreservierung)
  SEF: - *)
```

```
BEGIN
  NEW( S );
  ALists.New( S.List, 16 );
  Sets.New( S.BitVektor, MaxElem );
  S.Max := MaxElem
END New;

PROCEDURE Reset*( VAR S: BSet );
(*IN: Referenzvariable der Liste
 OUT: Referenzvariable der Liste
 SEM: Loeschen des Listeninhalts
  SEF: - *)
VAR i: INTEGER;
BEGIN
  ALists.Reset( S.List );
  Sets.Empty( S.BitVektor )
END Reset;


PROCEDURE Insert*( VAR S: BSet; Elem: INTEGER );
(*IN: Referenzvariable der Menge, einzufuegendes Element
 OUT: Referenzvariable der Menge
 SEM: fuegt Element in die Menge ein
  SEF: - *)
BEGIN
  IF Elem <= S.Max THEN
    IF ~Sets.In( S.BitVektor, Elem ) THEN
      Sets.Incl( S.BitVektor, Elem );
      ALists.Append( S.List, Elem )
    END
  ELSE
    IO.WriteString( IO.Msg, 'ERROR in eBSet.Insert: element is greater than max element: ' );
    IO.WriteInt( IO.Msg, Elem ); IO.WriteLn( IO.Msg ); IO.Update( IO.Msg );
    HALT(99)
  END
END Insert;

PROCEDURE Delete*( VAR S: BSet; Elem: INTEGER );
(*IN: Referenzvariable der Menge, zu loeschendes Element
 OUT: Referenzvariable der Menge
 SEM: loescht Element aus der Menge
  SEF: - *)
VAR i: INTEGER;
BEGIN
  IF Elem <= S.Max THEN
    IF Sets.In( S.BitVektor, Elem ) THEN
      Sets.Excl( S.BitVektor, Elem );
      i := ALists.IndexOf( S.List, Elem );
      ALists.Delete( S.List, i )
    END
  ELSE
    IO.WriteString( IO.Msg, 'ERROR in eBSet.Delete: element is greater than max element: ' );
    IO.WriteInt( IO.Msg, Elem ); IO.WriteLn( IO.Msg ); IO.Update( IO.Msg );
    HALT(99)
  END
END Delete;


PROCEDURE In*( S: BSet; Elem: INTEGER): BOOLEAN;
(*IN: Refrenzvariable der Menge, Element
 OUT: Referenzvariable der Menge
 SEM: Testet, ob Element in der Menge ist
  SEF: - *)
BEGIN
  RETURN Sets.In( S.BitVektor, Elem )
END In;

END eBSets.
```

3.4.4 eASets.Mod

```
MODULE eASets; (* 2.1 14.04.97 Denis Kuniss *)

(* Spezifikation:
    Dieses Modul stellt einen Mengentyp zur Verfuegung, der effektiv als Liste implementiert
ist,
    und trotzdem einen Test, ob ein Element in der Menge enthalten ist, mit konstantem
    Aufwand erlaubt.
    Es duerfen ganzzahlige Elemente von 0 bis MaxElement eingefuegt werden.
    Auf die Elemente der Menge kann in Form von Set.List[firstIndex]  bis
Set.List[S.List.Last]
```

```
    zugegriffen werden
  Einschraenkungen/Nachteile:
    - Groesse der Menge ist fest nach der Initialisierung
    - Groesse des alloziierten Speicherbereichs ist konstant: MaxElement*SIZE(INTEGER)
  Vorteile/Anzeigen:
    Es ist angezeigt, die DS zu verwenden, wenn sehr dicht besetzte Mengen verwendet
    werden (Einsparung des Bitvektors - siehe auch eBSet.Mod)
*)

IMPORT IO := eIO, ALists := eALists;

CONST
  firstIndex* =  ALists.firstIndex;
  noelem =  -1;

TYPE
  ASetDesc* = RECORD
    Max*: INTEGER;
    List*: ALists.AList
  END;
  ASet* = POINTER TO ASetDesc;

VAR S: ASet; i: INTEGER;

PROCEDURE New*( VAR S: ASet; MaxElem: INTEGER );
(*  IN: Referenzvariable der Liste, anfaengliche Laenge der Liste
  OUT: Referenzvariable der Liste
  SEM: Anlegen einer neuen Liste (Speicherplatzreservierung)
  SEF:  - *)
BEGIN
  NEW( S );
  ALists.New( S.List, MaxElem + 1 );
  S.Max := MaxElem;
  FOR i := firstIndex TO MaxElem DO S.List.Elem[i] := noelem END
END New;

PROCEDURE Reset*( VAR S: ASet );
(*IN:  Referenzvariable der Liste
 OUT:  Referenzvariable der Liste
 SEM:  Loeschen des Listeninhalts
  SEF:  - *)
VAR i: INTEGER;
BEGIN
  ALists.Reset ( S.List );
  FOR i := firstIndex TO S.Max DO S.List.Elem[i] := noelem END
END Reset;

PROCEDURE IsEmpty*( VAR S: ASet ): BOOLEAN;
(*IN:  Referenzvariable der Menge
 OUT:  Referenzvariable der Menge
 SEM:  Test, ob die Menge leer ist.
  SEF:  - *)
BEGIN
  RETURN (S.List.Last < firstIndex)
END IsEmpty;


PROCEDURE Insert*( VAR S: ASet; Elem: INTEGER );
(*IN: Referenzvariable der Menge, einzufuegendes Element
 OUT: Referenzvariable der Menge
 SEM: fuegt Element in die Menge ein.
  SEF: - *)
BEGIN
  IF Elem <= S.Max THEN
    IF Elem <= S.List.Last THEN
      IF S.List.Elem[Elem] # Elem THEN
        INC( S.List.Last );
        IF S.List.Elem[S.List.Last] > noelem THEN
          S.List.Elem[S.List.Elem[S.List.Last]] := S.List.Elem[Elem];
          S.List.Elem[S.List.Elem[Elem]] := S.List.Elem[S.List.Last];
          S.List.Elem[S.List.Last] := S.List.Last
        ELSE (* S.List.Elem[S.List.Last] <= noelem *)
          S.List.Elem[S.List.Last] := S.List.Elem[Elem];
          S.List.Elem[S.List.Elem[Elem]] := S.List.Last
        END;
        S.List.Elem[Elem] := Elem;
      END
    ELSE (* Elem > S.List.Last *)
      IF  S.List.Elem[Elem] <= noelem THEN
        INC( S.List.Last );
        IF S.List.Elem[S.List.Last] > noelem THEN
          S.List.Elem[S.List.Elem[S.List.Last]] := Elem;
```

```
          S.List.Elem[Elem] := S.List.Elem[S.List.Last];
          S.List.Elem[S.List.Last] := S.List.Last
        ELSE (* S.List.Elem[S.List.Last] <= noelem *)
          S.List.Elem[S.List.Last] := Elem;
          S.List.Elem[Elem] := S.List.Last
        END
      END
    END
  ELSE
    IO.WriteString( IO.Msg, 'ERROR in eASet.Insert: element is greater than max element: ' );
    IO.WriteInt( IO.Msg, Elem ); IO.WriteLn( IO.Msg ); IO.Update( IO.Msg );
    HALT(99)
  END
END Insert;

PROCEDURE Delete*( VAR S: ASet; Elem: INTEGER );
(*IN: Referenzvariable der Menge, zu loeschendes Element
 OUT: Referenzvariable der Menge
 SEM: loescht Element aus der Menge.
  SEF: - *)
VAR Last: INTEGER;
BEGIN
  IF Elem <= S.Max THEN
    IF Elem <= S.List.Last THEN
      IF S.List.Elem[Elem] = Elem THEN
        IF Elem = S.List.Last THEN
          S.List.Elem[S.List.Last] := noelem
        ELSE (* Elem < S.List.Last *)
          S.List.Elem[Elem] := S.List.Elem[S.List.Last];
          S.List.Elem[S.List.Last] := Elem;
        END;
        DEC( S.List.Last )
      END
    ELSE (* Elem > S.List.Last *)
      IF S.List.Elem[Elem] > noelem THEN
        IF S.List.Elem[Elem] = S.List.Last THEN
          S.List.Elem[S.List.Last] := noelem
        ELSE (* S.List.Elem[Elem] < S.List.Last *)
          S.List.Elem[S.List.Elem[Elem]] := S.List.Last;
          S.List.Elem[S.List.Last] := S.List.Elem[Elem];
        END;
        S.List.Elem[Elem] := noelem;
        DEC( S.List.Last )
      END
    END
  ELSE
    IO.WriteString( IO.Msg, 'ERROR in eASet.Delete: element is greater than max element: ' );
    IO.WriteInt( IO.Msg, Elem ); IO.WriteLn( IO.Msg ); IO.Update( IO.Msg );
    HALT(99)
  END
END Delete;


PROCEDURE In*( S: ASet; Elem: INTEGER): BOOLEAN;
(*IN: Referenzvariable der Menge, Element
 OUT: Referenzvariable der Menge
 SEM: Testet, ob Element in der Menge ist.
  SEF: - *)
BEGIN
  IF Elem > S.List.Last THEN RETURN (S.List.Elem[Elem] > noelem)
  ELSE RETURN (S.List.Elem[Elem] = Elem)
  END
END In;

END eASets.
```

3.4.5 eLIStacks.Mod

```
MODULE eLIStacks;  (* Denis Kuniss 1.00 09.03.98 *)

CONST
  emptyStack = -1;
  firstStackElem* = 0;


TYPE
  DataType = LONGINT;

  StackList = POINTER TO ARRAY OF DataType;

  Stack* =  POINTER TO RECORD
```

```
    Top: INTEGER;
    Elem: StackList
  END;


PROCEDURE Expand( VAR S: Stack );
VAR List1: StackList;
  i: INTEGER;
BEGIN
  IF LEN( S.Elem^ ) < MAX( INTEGER ) DIV 2 THEN
    NEW( List1, 2 * LEN( S.Elem^ ) + 1 )
  ELSE HALT(99) END;
  FOR i := firstStackElem TO S.Top DO List1[i] := S.Elem[i] END;
  S.Elem := List1;
END Expand;


PROCEDURE New*( VAR S: Stack; Len: INTEGER);
BEGIN
  NEW(S);
  NEW(S.Elem, Len);
  S.Top := emptyStack
END New;

PROCEDURE Reset*( VAR S: Stack );
BEGIN
  S.Top := emptyStack
END Reset;

PROCEDURE Push*( VAR S: Stack; Val: DataType );
BEGIN
  IF S.Top >= LEN(S.Elem^)-2 THEN Expand(S) END;
  INC(S.Top);
  S.Elem[S.Top] := Val
END Push;

PROCEDURE Pop*( VAR S: Stack );
BEGIN
  DEC(S.Top)
END Pop;

PROCEDURE Top*( VAR S: Stack): DataType;
BEGIN
  RETURN S.Elem[S.Top];
END Top;

PROCEDURE TopPop*( VAR S: Stack): DataType;
VAR R: DataType;
BEGIN
  R := S.Elem[S.Top];
  DEC(S.Top);
  RETURN R
END TopPop;

PROCEDURE IsEmpty*( S: Stack ): BOOLEAN;
BEGIN
  RETURN S.Top <= emptyStack
END IsEmpty;

END eLIStacks.
```

4 Begriffsdefinitionen und Internalisierung

Dieses Kapitel enthält eine kurze formale Definition der EAGen, um die in der weiteren Arbeit verwendete Terminologie festzuhalten. Sie ist eng an die Definitionen von Kutza angelehnt [Kutza]. Da jedoch EAGen im Compilergenerator Epsilon nicht mehr in Normalform vorliegen, wurden einige Erweiterungen und Neudefinitionen notwendig. Es werden die Begriffe *Affixparameter* und *definierendes Affix* neu eingeführt und die Datenstruktur zur internen Repräsentation der EAGen beschrieben.

4.1 Terminologie und Definition der EAGen

Eine Erweiterte Affixgrammatik ist ein 8-Tupel

$EAG = (MN, MT, MR, HN, HT, SPEC, HR, S)$

dessen einzelnen Komponenten folgendermaßen definiert sind:

- *MN* ist eine endliche Menge von *Meta-Nichtterminalen*. Ist $M \in MN$, dann sind M, $M1$, $M2$, ... sowie $\#M$, $\#M1$, $\#M2$, ... *Affixe* zu M.
  Für jedes Affix A zu M ist $dom(A) := M$ der Wertebereich des Affixes. In verschiedenen anderen Quellen wird der zum Begriff des *Affixes* synonyme Begriff der *Variablen* verwendet.
- *MT* ist die endliche Menge der *Meta-Terminale* mit $MN \cap MT = \emptyset$.
- *MR* stellt die endliche Menge der *Meta-Regeln* der Form $M_0 = M_1 .. M_n$ mit $n \geq 0$ und $M_0 \in MN$ und $M_i \in (MN \cup MT)$ dar.

Die kontextfreie Grammatik $MG_M := (MN, MT, MR, M)$ wird als die durch das Meta-Nichtterminal M aufgespannte *Meta-Grammatik* bezeichnet. *Affixformen* zu einem Meta-Nichtterminal M sind Satzformen von MG_M, in denen alle vorkommenden Meta-Nichtterminale durch entsprechende Affixe ersetzt wurden.

- *HN* ist eine endliche Menge von *Hyper-Nichtterminalen*.
- *HT* ist eine endliche Menge von *Hyper-Terminalen*, mit $HN \cap HN = \emptyset$.
- *SPEC* ist eine endliche Menge von *Spezifikationen* der Form $H(dir(a_1)\ dom(a_1), ..., dir(a_{\#a(H)})\ dom(a_{\#a(H)}))$, darin sind:
  - $\#a(H)$ die *Stelligkeit* von H
  - die Tupel a_i mit $0 < i \leq \#a(H)$ *Affixpositionen* von H, die auch in der Form a_i^H benannt werden können. Die Menge $A(H) = \{ a_i^H : 0 < i \leq \#a(H)\}$ ist die Menge aller Affixpositionen des Hyper-Nichtterminals H.
  - $dir(a_i) \in \{\uparrow,\downarrow\}$ die Richtung der Affixposition a_i. Affixpositionen mit der Richtung $\downarrow$ werden *inherited* (dt.: *ererbt*), die mit der Richtung $\uparrow$ *synthesized* (dt.: *abgeleitet*) genannt. $I(H)$ und $S(H)$ bezeichnen die Mengen der ererbten bzw. abgeleiteten Affixpositionen des Hyper-Nichtterminals H.
  - $dom(a_i) \in MN$ der Wertebereich einer Affixposition

Die Spezifikation einer *EAG* ist im Kalkül des Compilergenerators Epsilon nicht explizit vorhanden, sondern zur Vereinfachung in die syntaktische Struktur der Hyper-Regeln integriert. Formal ist jedoch eine Trennung unumgänglich und auch leichter zu handhaben.

Setzt man in die Affixpositionen eines Hyper-Nichtterminals Affixformen des zugehörigen Wertebereichs ein, so ergibt sich ein *Symbolvorkommen*, das formal folgendermaßen definiert ist:
Ist $H(dir(a_1)\ M_1, .., dir(a_{\#a(H)})\ M_{\#a(H)}$ eine Spezifikation und sind $f_1, .., f_{\#a(H)}$ Affixformen zu $M_1, .., M_{\#a(H)}$, dann ist H, parametrisiert mit Affixformen $H(f_1, .., f_{\#a(H)})$, ein *Symbolvorkommen*. In anderen Quellen wird dafür auch der Begriff des *Hypernotions* verwendet.

- *HR* ist eine endliche Menge von Hyper-Regeln. Eine Hyper-Regel r besteht aus einer linken und einer rechten Regelseite und hat die Form $X_0 : X_1 .. X_n$ mit $n \geq 0$, wobei X_0 ein Symbolvorkommen und die X_i Symbolvorkommen oder Hyper-Terminale sind. Der Doppelpunkt trennt die linke Regelseite von der rechten. Damit die Symbolvorkommen außerhalb des Regelkontextes eindeutig unterschieden werden können, werden sie zusätzlich mit der Regel parametrisiert: X_i^r.

Im Rahmen der Generierung eines Evaluators wird von den Hyper-Terminalen einer Hyper-Regel abstrahiert. Der Ausdruck $\#S(r)$ definiert die Anzahl der Symbolvorkommen in der Regel r.

Im Kontext einer Regel werden die Parameter a_i eines Symbolvorkommens $X(a_1,..,a_{\#a(H)})$ *Affixparameter* genannt. Um Affixparameter auch ohne den Kontext der Regel eindeutig unterscheiden zu können, werden sie mit der Regel r und dem Index des Symbolvorkommens X_i parametrisiert: $a_k^{(r,i)}$. Der Positionsindex k bezieht sich entweder auf die Position des Affixparameters in der Liste der Affixparameter des Symbolvorkommens oder, wenn der Index des Symbolvorkommens weggelassen wird, auf die Position in der Liste aller Affixparameter einer Regel r: $a_k^{(r)}$. Der Ausdruck $\#a(r)$ beziffert die Anzahl aller in der Regel r verwendeten Affixparameter. Der Begriff des Affixparameters ist in dieser Arbeit neu definiert worden. In [ZiVoKüNa] wird zwischen den Affixpositionen der Spezifikation und der Parametrisierung der Symbolvorkommen begrifflich nicht unterschieden; Affixparameter werden dort als *Affixpositionen einer Regel* definiert. Dies würde insbesondere bei der Beschreibung der Implementierung zu Verwirrung führen, da Affixparameter, wie in der nachfolgenden Definition zu sehen ist, zum Teil andere Eigenschaften besitzen, die auf Affixpositionen in keiner Weise übertragbar sind. Kutza definiert den synonymen Begriff des *Affixvorkommens* [Kutza], den ich jedoch nicht für adäquat halte, der jedoch aus historischen Gründen Eingang in die Implementierung gefunden hat. Da jedoch eine enge Beziehung zwischen Affixparametern und Affixpositionen besteht, wird folgende begriffliche Relation definiert: Ein Affixparameter $a_k^{(r,i)}$ einer Regel r *korrespondiert* zu einer Affixposition a_j^X, wenn $X_i^r=X$ und $k=j$ gilt. In diesem Fall ist X_i^r ein Symbolvorkommen zum Symbol X in der Regel r, und der Affixparameter $a_k^{(r,i)}$ steht auf der j-ten Parameterposition des Symbolvorkommens X_i^r.

Die Menge aller Affixparameter einer Regel r ist definiert durch $AP(r) := \{ a_k^{(r,i)} : 0<i\leq\#S(r) \text{ und } 0<k\leq\#a(X_i)\}$. Ein Affixparameter $a_k^{(r,i)}$ heißt *definierend*, wenn mit $X=X_i$ gilt: (i=0 und $dir(a_k^X)= \downarrow$) oder (i>0 und $dir(a_k^X)= \uparrow$), und *applizierend*, wenn mit $X=X_i$ gilt: (i=0 und $dir(a_k^X)= \uparrow$) oder (i>0 und $dir(a_k^X)= \downarrow$).
$AP_D(r)$ und $AP_A(r)$ bezeichnen die Mengen der definierenden bzw. applizierenden Affixparameter. Der Inhalt jedes applizierenden Affixparameters $a^{(r,i)}$ ergibt sich aus seiner Affixform. Die Affixform wiederum besteht aus Affixen, die sich aus definierenden Affixparametern der Regel r ergeben. Somit wird $a^{(r,i)}$ in Abhängigkeit einer Menge $D(a^{(r,i)})$ von Affixparamatern der gleichen Regel definiert. Durch die Wohlgeformtheitsbedingungen wird sichergestellt, daß die *EAG* in *Bochmann-Normalform* ist, also keine applizierenden Affixparameter in $D(a^{(r,i)})$ enthalten sind. D wird in der üblichen Weise als Relation auf Affixparametern interpretiert, d.h

$$(a^{(r,i)}, b^{(r,j)}) \in D \Leftrightarrow b^{(r,j)} \in D(a^{(r,i)}),$$

wobei die Abhängigkeiten in Richtung des Datenflusses beschrieben werden durch

$$D^{-1} = \{ (b^{(r,j)}, a^{(r,i)}): (a^{(r,i)}, b^{(r,j)}) \in D\}.$$

Ein Affix ist ein *definierendes Affix*, wenn es in einer Hyper-Regel textuell vor allen anderen gleichnamigen Affixen in einer Affixform eines definierenden Affixparameters steht. Eine Hyper-Regel ist *linksdefinierend*, wenn für jedes Affix V in applizierenden Affixparametern und für jedes negierte Affix $\#V$ in definierenden Affixparametern ein definierendes Affix V existiert.

- S ist ein ausgezeichnetes Hyper-Nichtterminal, das Startsymbol mit der Spezifikation $S(\uparrow M)$, wobei $M \in MN$.

*EAG*en erlauben die Formulierung sogenannter Prädikate. Prädikate werden durch Hyper-Nichtterminale spezifiziert, die sich nach leer ableiten lassen oder scheitern. Es ist sinnvoll, die *EAG* in einen generativen und prädikativen Teil zu zerlegen, da Prädikate nichts zur kontextfreien Struktur der Quellsprache beitragen.

Die Menge der *Grundnichtterminale GN* einer *EAG* ist wie folgt induktiv definiert:
- S ist ein Grundnichtterminal;
- enthält eine Hyper-Regel auf der rechten Regelseite ein Hyper-Terminal, so ist das Hyper-Nichtterminal auf der linken Regelseite ein Grundnichtterminal;
- enthält eine Hyper-Regel auf der rechten Regelseite ein Grundnichtterminal, so ist das Hyper-Nichtterminal auf der linken Regelseite ein Grundnichtterminal.

Die Menge *PN* der *Prädikatnichtterminale* enthält alle Hyper-Nichtterminale, die keine Grundnichtterminale sind.

Eine Hyper-Regel ist eine *Prädikatregel*, wenn auf ihrer linken Seite ein Symbolvorkommen eines Prädikatnichtterminals steht; alle anderen Regeln sind *Evaluatorregeln*. Die *Grundgrammatik* (auch *Parsergrammatik* genannt) einer *EAG* ist eine kontextfreie Grammatik, die aus den Grundnichtterminalen, den Hyper-Terminalen und den Evaluatorregeln besteht, in denen die Prädikatnichtterminale und alle Parametrisierungen eliminiert wurden. Als Startsymbol verbleibt das Startsymbol der *EAG*.

Ein aus der Grundgrammatik generierter Parser erzeugt Ableitungsbäume der Parsergrammatik, die jedoch keine Hyper-Terminale mehr enthalten. Ein Ableitungsbaum t ist ein geordneter Baum. Jeder Knoten von t ist

mit einem Grundnichtterminal markiert. Für jeden Knoten *k* des Ableitungsbaumes *t* existiert eine Regel $r=X_0^r: X^r_1,..,X^r_{\#S(r)}$, so daß *k* Instanz des Symbolvorkommens X_0^r ist und seine Söhne Instanzen der Symbolvorkommen $X^r_1,..,X^r_{\#S(r)}$ sind; *k* wird zusätzlich mit *r* markiert. Weiterhin sind jedem Knoten die Instanzen der in der Hyper-Regel vorkommenden Affixe - *Affixvariablen* genannt - zugeordnet, die den Wert des definierenden Affixes aufnehmen und die Übersetzung der Evaluation enthalten. Die Wurzel eines jeden durch den Parser erzeugten Ableitungsbaumes ist Instanz des Startsymbols *S* der *EAG*.

4.2 Wohlgeformtheitsbedingungen für EAGen

Eine EAG ist wohlgeformt, wenn

1. ihre Grundgrammatik eindeutig ist und jede Grundregel aus genau einer Hyper-Regel entsteht,
2. jedes Meta-Nichtterminale $M \in MR$ *strikt synthetisiert* ist oder die von *M* aufgespannten Meta-Grammatik MG_M eindeutig ist,
3. die EAG zyklenfrei ist und
4. alle Prädikatregeln links-definierend und die erreichbaren Prädikate eindeutig und konvergent sind.

Die erste Bedingung ermöglicht die strikte Trennung der kontextfreien Analyse von der Analyse der Kontextabhängigkeiten und damit eine unabhängige Generierung von Parsern und Evaluatoren. Sie ist nicht automatisch überprüfbar, da das Problem der Eindeutigkeit einer beliebigen kontextfreien Sprache nicht entscheidbar ist.

Ein Meta-Nichtterminal $M \in MR$ ist *strikt synthetisierend*, wenn für alle definierenden Affixparameter, die ein Affix *A* zu *M* enthalten, *A* der einzige Bestandteil der Affixform und ein definierendes Affix ist, und kein definierender Affixparameter das negierte Affix *#A* enthält. Die zweite Bedingung erhält die Eindeutigkeit aller Affixform-Analysen und verhindert bei mehrdeutigen Meta-Grammatiken Vergleiche für mehrdeutige Ableitungsbäume. Der Generator kann die Einhaltung dieser Bedingung nicht überprüfen, da das Problem wie bei der ersten Bedingung unentscheidbar ist.

Die Prüfung der Zyklenfreiheit einer EAG ist integraler Bestandteil eines jeden Evaluator-Generierungsverfahrens und wird im folgenden Kapitel ausführlich beschrieben.

Die vierte Bedingung sorgt für die Verfügbarkeit aller Übergabeparameter eines Prädikataufrufes und stellt die Termination sowie die einheitliche Berechnung der Rückgabeparameter aller Prädikataufrufe sicher. Ohne diese Bedingung wäre eine eindeutige Generierung von Prädikatprozeduren nicht möglich.

Die Spezifika des hier vorzustellenden Generierungsverfahrens erfordern noch eine weitere Bedingung, die jedoch für die meisten Evaluationsverfahren mit mehrfachem Besuch der Baumknoten gültig ist, und deshalb bereits jetzt genannt werden soll. Es handelt sich um die Forderung nach der Links-Definiertheit aller Hyper-Regeln, wie sie auch schon für Prädikatregeln gefordert wurde. Sie stellt sicher, daß jedes Affix in einem applizierenden Affixparameter, mindestens einmal in einem definierenden Affixparameter derselben Regel vorkommt, damit dessen Wert zum Zeitpunkt der Synthese bekannt ist. Diese Bedingung kann der Generator automatisch prüfen.

4.3 Interne Darstellung der EAGen

Da die primäre Datenstruktur des Compilergenerators Epsilon den EBNF-Regeln der Spezifikationssprache angepaßt ist [DeWe], für die Berechnung eines Evaluators für SOAGen die Regeln jedoch in BNF-Form vorliegen müssen, wurde eine entsprechende Datenstruktur und ein Algorithmus, der diese Transformation vornimmt, entwickelt. Beide werden im folgenden vorgestellt.

```
TYPE
    OpenTDP = POINTER TO ARRAY OF Sets.OpenSet;

    RuleDesc = RECORD
       SymOcc,
       AffOcc: EAG.ScopeDesc;
       TDP: OpenTDP;
       VS: EAG.ScopeDesc
    END;
    RuleBase = POINTER TO RuleDesc;

    EmptyRule = POINTER TO RECORD (RuleDesc)
       Rule: EAG.Rule
    END;

    OrdRule = POINTER TO RECORD (RuleDesc)
       Alt: EAG.Alt;
    END;

    OpenRule = POINTER TO ARRAY OF RuleBase;

VAR
    Rule: OpenRule;
```

Zur Berechnung der SOAG-Eigenschaft einer Spezifikation muß jede Alternative der Ausgangsdatenstruktur in eine eigenständige Regel umgewandelt werden. Die optionale Alternative und die Wiederholung müssen in ihrer BNF-adäquaten Form abgespeichert werden. Alle diese Regeln werden im Feld `Rule` aufgenommen. `SymOcc` verweist auf einen Bereich im Feld `SymOcc`, der alle Symbolvorkommen einer Regel enthält. `AffOcc` verweist auf einen Bereich im Feld `AffOcc`, der alle zur Regel gehörenden Affixparameter enthält.

Die Verbindung zur Ausgangsdatenstruktur des EAG-Moduls wird durch das RECORD-Feld `Rule` in der Typerweiterung `EmptyRule` und durch das RECORD-Feld `Alt` in der Typerweiterung `OrdRule` (*ordinary rule*) hergestellt. Diese sind notwendig, um den Zugriff auf die Parameter und die Textpositionen der Regeln zu ermöglichen. Die transformierten Regeln spiegeln sich direkt in der zu beschreibenden Datenstruktur wider. Zur bildlichen Veranschaulichung der Datenstruktur benutze ich die in Tabelle 4-1 aufgeführte

Ausgangsregel:	*transformierte Regeln:*
N = ‘i’ N | .	N = ‘i’ N | .
S <+ N_1: N>:	S<+N_1: N>: _A<N_2> _B<N_5>
<N_2>{<+ ‘i’ N_3: N> ‘a’ <N_4>} <+ λ_1 : N>	(r1)_A_1<+ ‘i’ N_3: N>: ‘a’ _A_2<N_4>. (r2)_A_3<+ λ_1: N>: .
<N_5>{<- ‘i’ N_6: N> ‘b’ <N_7>} <- λ_2: N>.	_B<- ‘i’ N_6: N>: ‘b’ _B<N_7>. _B<- λ_2: N>:.

Tabelle 4-1: Transformation der EBNF-Regeln

Beispielspezifikation. Das eindeutige Indizieren der Affixe in Tabelle 4-1 dient lediglich zur Verfolgung ihrer Positionierung nach der Transformation. λ symbolisiert das leere Wort. (r1) und (r2) bezeichnen zwei Regeln eindeutig, und einige Indizes an Hyper-Nichtterminalen dienen zur eindeutigen Unterscheidung verschiedener Vorkommen des Hyper-Nichtterminals _A.

```
TYPE
    SymOccDesc = RECORD
       SymInd,
       RuleInd: INTEGER;
       Nont: EAG.Nont;
       AffOcc: EAG.ScopeDesc;
       Next: INTEGER;
    END;

    OpenSymOcc = POINTER TO ARRAY OF SymOccDesc;

VAR
    SymOcc: OpenSymOcc;
```

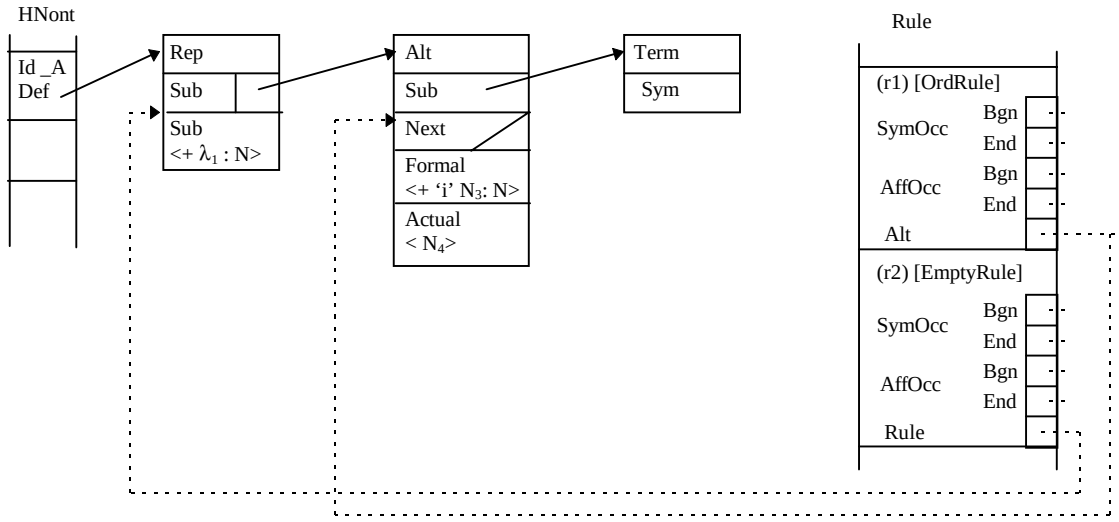


Abbildung 4-1: Veranschaulichung der Verbindung zwischen SOAG- und EAG-Datenstruktur

Das Feld `SymOcc` enthält alle Symbolvorkommen, die in den transformierten Regeln der Spezifikation vorkommen. Bei `SymInd` handelt es sich um einen Index in das Feld `Sym`. `Nont` verweist auf das Hyper-Nichtterminal der Ausgangsdatenstruktur des Moduls EAG. Wie schon in der vorangegangenen Typdeklaration verweist das RECORD-Feld `AffOcc` auf die zum Hyper-Nichtterminal gehörenden Affixparameter. Das RECORD-Feld `Next` zeigt auf das nächste Vorkommen des gleichen Hyper-Nichtterminals. Über `Next` wird eine Liste aller Vorkommen eines Hyper-Nichtterminals in allen Regeln gebildet.

```
TYPE
    AffOccDesc = RECORD
       ParamBufInd,
       SymOccInd: INTEGER;
       AffOccNum: RECORD
          InRule,
          InSym: INTEGER;
       END
    END;

    OpenAffOcc = POINTER TO ARRAY OF AffOccDesc;

VAR
    AffOcc: OpenAffOcc;
```

Die Affixparameter eines Hyper-Nichtterminal werden im Feld `AffOcc` abgespeichert. Jeder Feldeintrag enthält durch den Index `ParamBufInd` einen Verweis auf die Affixform, der auf das Feld `ParamBuf` des Moduls `EAG` verweist. `SymOccInd` verweist auf das Symbolvorkommen im Feld `SymOcc`. Weiterhin werden noch Koordinaten des Affixparameters bezüglich seiner Regel, in der er enthalten ist, und bezüglich des Hyper-Nichtterminals, das er parametrisiert, deklariert.

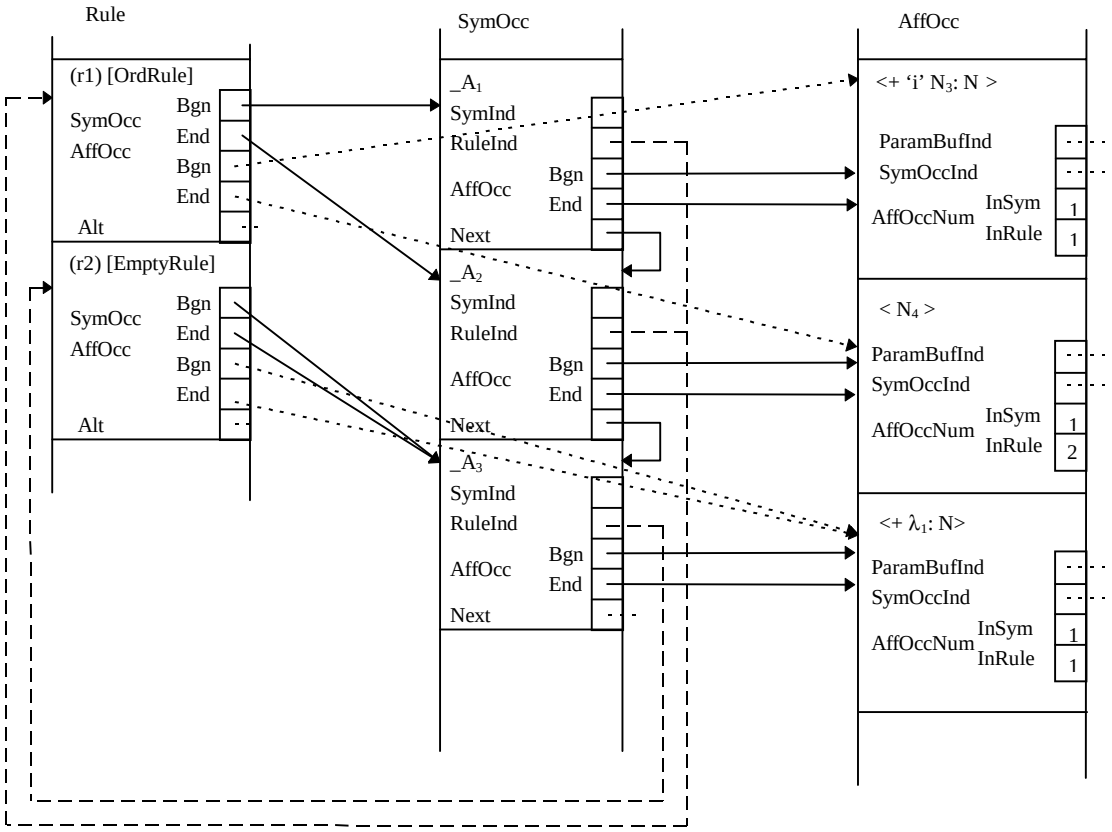


Abbildung 4-2: Veranschaulichung der Datenstruktur des SOAG-Evaluator-Generators

```
TYPE
    SymDesc = RECORD
      FirstOcc,
      MaxPart: INTEGER;
      AffPos: EAG.ScopeDesc;
    END;

    OpenSym = POINTER TO ARRAY OF SymDesc;
    OpenPart = POINTER TO ARRAY OF INTEGER;
    OpenDefAffOcc = POINTER TO ARRAY OF INTEGER;
    OpenAffixApplCnt = POINTER TO ARRAY OF INTEGER;

VAR
    Sym: OpenSym;
    PartNum: OpenPartNum;
    DefAffOcc: OpenDefAffOcc;
    AffixApplCnt: OpenAffixApplCnt;
```

Das Feld `Sym` existiert parallel zum Feld `HNont` und enthält in `FirstOcc` einen Anker, der auf eine Liste aller Vorkommen eines Hyper-Nichtterminals in der oben beschriebenen Struktur verweist. `AffPos` verweist auf einen Bereich im Feld `PartNum`, der alle Partitionsnummern der zum Hyper-Nichtterminal gehörenden Affixpositionen aufnehmen wird. `MaxPart` nimmt die maximale Partitionsnummer des Symbols auf. Beide Elemente werden erst im Modul `SOAGPartition` berechnet.

Das Feld `DefAffOcc` liegt parallel zu `EAG.Var` und nimmt für jede Affixvariable den Index des Affixparameters auf, der das definierende Affix der Affixvariable enthält.

Das Feld `AffixApplCnt` enthält für jede Affixvariable die Anzahl ihrer Anwendungen in Synthesen und Vergleichen. Es wird im Modul `SOAGGen` zur Berechnung der Lebensdauer von Affixvariablen verwendet. Sein Inhalt wird im Modul `SOAGPartition` berechnet.

```
    VAR
        NextSym,
        NextPartNum,
        NextRule,
        NextSymOcc,
        NextAffOcc,
        NextVS,
        NextDefAffOcc,
        NextAffixApplCnt: INTEGER;
```

Alle Variablen der Form `Next`*`Feldname`* verweisen auf den nächste freien Feldeintrag des jeweiligen Feldes. (Dieses Implementierungsprinzip wurde auch schon im Modul EAG verwendet.)

4.4 Implementierungen

4.4.1 eSOAG.Mod

```
MODULE eSOAG; (* dk 1.56 14.03.98 *)

IMPORT EAG := eEAG, IO := eIO, Sets := eSets, Predicates := ePredicates;

CONST
  firstSym* = EAG.firstHNont;
  firstRule* = 0;
  firstSymOcc* = 0;
  firstAffOcc* = 0;
  firstPartNum* = 0;
  firstAffOccNum* = 0;
  firstVS* = 1;
  firstDefAffOcc* = EAG.firstVar;
  firstStorageName* = 0;
  firstAffixApplCnt* = EAG.firstVar;
  nil* = -1;

TYPE

  SymDesc* = RECORD
    FirstOcc*,
    MaxPart*: INTEGER;
    AffPos*: EAG.ScopeDesc
  END;

  OpenTDP = POINTER TO ARRAY OF Sets.OpenSet;

  RuleDesc* = RECORD
    SymOcc*,
    AffOcc*: EAG.ScopeDesc;
    TDP*, DP*: OpenTDP;
    VS*: EAG.ScopeDesc
  END;
  RuleBase* = POINTER TO RuleDesc;

  EmptyRule* = POINTER TO RECORD (RuleDesc)
    Rule*: EAG.Rule;
  END;

  OrdRule* = POINTER TO RECORD (RuleDesc)
    Alt*: EAG.Alt;
  END;


  SymOccDesc* = RECORD
    SymInd*,
    RuleInd*: INTEGER;
    Nont*: EAG.Nont;
    AffOcc*: EAG.ScopeDesc;
    Next*: INTEGER;
  END;

  AffOccDesc* = RECORD
    ParamBufInd*,
    SymOccInd*: INTEGER;
    AffOccNum*: RECORD
      InRule*,
      InSym*: INTEGER;
    END
  END;
```

```
(* Datenstruktur der Visit-Sequenzen *)

TYPE
  InstructionDesc* = RECORD END;
  Instruction* = POINTER TO InstructionDesc;

  Visit* = POINTER TO RECORD ( InstructionDesc )
    SymOcc*,
    VisitNo*: INTEGER
  END;

  Leave* = POINTER TO RECORD ( InstructionDesc )
    VisitNo*: INTEGER
  END;

  Call* = POINTER TO RECORD ( InstructionDesc )
    SymOcc*: INTEGER
  END;


  OpenInteger* = POINTER TO ARRAY OF INTEGER;
  OpenSym* = POINTER TO ARRAY OF SymDesc;
  OpenPartNum* = OpenInteger;
  OpenRule* = POINTER TO ARRAY OF RuleBase;
  OpenSymOcc* = POINTER TO ARRAY OF SymOccDesc;
  OpenAffOcc* = POINTER TO ARRAY OF AffOccDesc;
  OpenVS* = POINTER TO ARRAY OF Instruction;
  OpenDefAffOcc* = OpenInteger;
  OpenStorageName* = OpenInteger;
  OpenAffixApplCnt* = OpenInteger;

VAR
  Sym*: OpenSym;          (* parallel zu EAG.Nont *)
  PartNum*: OpenPartNum;
  Rule*: OpenRule;
  SymOcc*: OpenSymOcc;
  AffOcc*: OpenAffOcc;
  VS*: OpenVS;
  DefAffOcc*: OpenDefAffOcc;   (* parallel zu EAG.Var *)
  StorageName*: OpenStorageName;  (* parallel zu PartNum (fuer jede Affixposition) *)
  AffixApplCnt*: OpenAffixApplCnt;  (* parallel zu EAG.Var *)

  NextSym*,
  NextPartNum*,
  NextRule*,
  NextSymOcc*,
  NextAffOcc*,
  NextVS*,
  NextDefAffOcc*,
  NextStorageName*,
  NextAffixApplCnt: INTEGER;

  MaxAffNumInRule*,
  MaxAffNumInSym*,
  MaxPart*: INTEGER;


CONST
  abnormalyError* = 1;
  cyclicTDP* = 2;
  notLeftDefined* = 3;
  notEnoughMemory* = 99;

PROCEDURE Error*( ErrorType: INTEGER; Proc: ARRAY OF CHAR );
BEGIN
  IO.WriteString( IO.Msg, 'ERROR: ');
  CASE ErrorType OF
    abnormalyError: IO.WriteText( IO.Msg, 'abnormaly error ' );
    | notEnoughMemory: IO.WriteText( IO.Msg, 'memory allocation failed ' );
    | cyclicTDP: IO.WriteText( IO.Msg, 'TDP is cyclic...aborted\n' );
    | notLeftDefined: IO.WriteText(IO.Msg, 'Grammar is not left defined\n');
  END;
  IF (ErrorType=abnormalyError) OR (ErrorType=notEnoughMemory) THEN
    IO.WriteString( IO.Msg, 'in procedure ' );
    IO.WriteString( IO.Msg, Proc ); IO.WriteLn( IO.Msg )
  END;
  IO.Update( IO.Msg );
  HALT(98)
END Error;
```

```
PROCEDURE Expand*;
VAR
  Rule1: OpenRule;
  SymOcc1: OpenSymOcc;
  AffOcc1: OpenAffOcc;
  VS1: OpenVS;
  i: LONGINT;

    PROCEDURE NewLen(ArrayLen: LONGINT): LONGINT;
    BEGIN
      IF ArrayLen < MAX(INTEGER) DIV 2 THEN RETURN 2 * ArrayLen + 1 ELSE HALT(99) END
    END NewLen;

BEGIN
  IF NextAffOcc >= LEN(AffOcc^) THEN NEW( AffOcc1, NewLen(LEN(AffOcc^)));
    FOR i := firstAffOcc TO LEN(AffOcc^)-1 DO AffOcc1[i] := AffOcc[i] END;
    AffOcc := AffOcc1
  END;
  IF NextSymOcc >= LEN(SymOcc^) THEN NEW( SymOcc1, NewLen(LEN(SymOcc^)));
    FOR i := firstSymOcc TO LEN(SymOcc^)-1 DO SymOcc1[i] := SymOcc[i] END;
    SymOcc := SymOcc1
  END;
  IF NextRule >= LEN(Rule^) THEN NEW( Rule1, NewLen(LEN(Rule^)));
    FOR i := firstRule TO LEN(Rule^)-1 DO Rule1[i] := Rule[i] END; Rule := Rule1
  END;
  IF NextVS >= LEN(VS^) THEN NEW( VS1, NewLen(LEN(VS^)));
    FOR i := firstVS TO LEN(VS^)-1 DO VS1[i] := VS[i] END;
    VS := VS1
  END;

END Expand;


PROCEDURE AppAffOcc( Params: INTEGER );
BEGIN
  IF Params # EAG.empty THEN
    WHILE EAG.ParamBuf[Params].Affixform # EAG.nil  DO
      AffOcc[NextAffOcc].ParamBufInd := Params;
      AffOcc[NextAffOcc].SymOccInd := NextSymOcc;
      AffOcc[NextAffOcc].AffOccNum.InRule := NextAffOcc - Rule[NextRule].AffOcc.Beg;
      AffOcc[NextAffOcc].AffOccNum.InSym := NextAffOcc -
                                  SymOcc[NextSymOcc].AffOcc.Beg;
      INC( NextAffOcc ); IF NextAffOcc >= LEN( AffOcc^ ) THEN Expand END;
      INC( Params );
    END
  END
END AppAffOcc;


PROCEDURE AppSymOccs( Factor: EAG.Factor );
BEGIN
  WHILE Factor # NIL DO
    IF Factor IS EAG.Nont THEN
      SymOcc[NextSymOcc].SymInd := Factor(EAG.Nont).Sym;
      SymOcc[NextSymOcc].RuleInd := NextRule;
      SymOcc[NextSymOcc].Nont := Factor(EAG.Nont);
      SymOcc[NextSymOcc].AffOcc.Beg := NextAffOcc;
      AppAffOcc( Factor(EAG.Nont).Actual.Params );
      SymOcc[NextSymOcc].AffOcc.End := NextAffOcc - 1;
      SymOcc[NextSymOcc].Next := Sym[Factor(EAG.Nont).Sym].FirstOcc;
      Sym[Factor(EAG.Nont).Sym].FirstOcc := NextSymOcc;
      INC( NextSymOcc ); IF NextSymOcc >= LEN( SymOcc^ ) THEN Expand END;
    END;
    Factor := Factor.Next;
  END
END AppSymOccs;


PROCEDURE AppLeftSymOcc( leftSym, Params: INTEGER );
BEGIN
  SymOcc[NextSymOcc].SymInd := leftSym;
  SymOcc[NextSymOcc].RuleInd := NextRule;
  SymOcc[NextSymOcc].Nont := NIL;
  SymOcc[NextSymOcc].AffOcc.Beg := NextAffOcc;
  AppAffOcc( Params );
  SymOcc[NextSymOcc].AffOcc.End := NextAffOcc - 1;
  SymOcc[NextSymOcc].Next := Sym[leftSym].FirstOcc;
  Sym[leftSym].FirstOcc := NextSymOcc;
  INC( NextSymOcc ); IF NextSymOcc >= LEN( SymOcc^) THEN Expand END;
END  AppLeftSymOcc;
```

```
PROCEDURE AppEmptyRule( leftSym: INTEGER; EAGRule: EAG.Rule);
VAR A: EmptyRule;
BEGIN
  NEW( A ); Rule[NextRule] := A;
  A(EmptyRule).Rule := EAGRule;
  A.SymOcc.Beg := NextSymOcc;
  A.AffOcc.Beg := NextAffOcc;
  IF EAGRule IS EAG.Opt THEN
    AppLeftSymOcc( leftSym, EAGRule(EAG.Opt).Formal.Params );
  ELSIF EAGRule IS EAG.Rep THEN
    AppLeftSymOcc( leftSym, EAGRule(EAG.Rep).Formal.Params )
  END;
  A.SymOcc.End := NextSymOcc - 1;
  A.AffOcc.End := NextAffOcc - 1;
  INC( NextRule ); IF NextRule >= LEN( Rule^ ) THEN Expand END;
END AppEmptyRule;

PROCEDURE AppRule( EAGAlt: EAG.Alt );
VAR A: OrdRule;
BEGIN
  NEW( A ); Rule[NextRule] := A;
  A.Alt := EAGAlt;
  A.SymOcc.Beg := NextSymOcc;
  A.AffOcc.Beg := NextAffOcc;
  AppLeftSymOcc( EAGAlt.Up, EAGAlt.Formal.Params );
  AppSymOccs( EAGAlt.Sub );
  A.SymOcc.End := NextSymOcc - 1;
  A.AffOcc.End := NextAffOcc - 1;
  INC( NextRule ); IF NextRule >= LEN( Rule^ ) THEN Expand END;
END AppRule;

PROCEDURE AppRepRule( EAGAlt: EAG.Alt );
VAR A: OrdRule;
BEGIN
  NEW( A ); Rule[NextRule] := A;
  A.Alt := EAGAlt;
  A.SymOcc.Beg := NextSymOcc;
  A.AffOcc.Beg := NextAffOcc;
  AppLeftSymOcc( EAGAlt.Up, EAGAlt.Formal.Params );
  AppSymOccs( EAGAlt.Sub );
  AppLeftSymOcc( EAGAlt.Up, EAGAlt.Actual.Params );
  A.SymOcc.End := NextSymOcc - 1;
  A.AffOcc.End := NextAffOcc - 1;
  INC( NextRule ); IF NextRule >= LEN( Rule^ ) THEN Expand END;
END AppRepRule;

PROCEDURE AppVS*( VAR I: Instruction );
(*IN: Instruktion
 OUT: -
 SEM: fuegt eine Instruktion in die Datenstruktur VS ein *)
BEGIN
  VS[NextVS] := I;
  INC( NextVS ); IF NextVS >= LEN( VS^ ) THEN Expand END;
END AppVS;


PROCEDURE IsInherited*( S, AffOccNum: INTEGER ): BOOLEAN;
(*IN: Symbol, Nummer des Affixposition
 OUT: boolscher Wert
 SEM: Test, ob Affixposition inherited ist *)
BEGIN
  RETURN (EAG.DomBuf[EAG.HNont[S].Sig + AffOccNum] < 0)
END IsInherited;

PROCEDURE IsSynthesized*( S, AffOccNum: INTEGER ): BOOLEAN;
(*IN: Symbol, Nummer des Affixposition
 OUT: boolscher Wert
 SEM: Test, ob Affixposition synthesized ist *)
BEGIN
  RETURN (EAG.DomBuf[EAG.HNont[S].Sig + AffOccNum] > 0)
END IsSynthesized;

PROCEDURE IsOrientable*( S, AffOccNum1, AffOccNum2: INTEGER ): BOOLEAN;
(*IN: Symbol, Nummern zweier Affixpositionen zum Symbol
 OUT: boolscher Wert
 SEM: Test, ob die beiden Affixpositionen orientierbar sind *)
BEGIN
  RETURN
    (IsInherited( S, AffOccNum1) & IsSynthesized( S, AffOccNum2 )) OR
    (IsInherited( S, AffOccNum2) & IsSynthesized( S, AffOccNum1 ))
END IsOrientable;
```

```
PROCEDURE IsEvaluatorRule*( R: INTEGER): BOOLEAN;
(*IN: Regel
 OUT: boolscher Wert
 SEM: Test, ob eine Evaluatorregel vorliegt
 PRECOND: Predicates.Check muss vorher ausgewertet sein *)
BEGIN
  RETURN ~Sets.In( EAG.Pred, SymOcc[Rule[R].SymOcc.Beg].SymInd )
END IsEvaluatorRule;

PROCEDURE IsPredNont*( SO: INTEGER): BOOLEAN;
(*IN: Szmbolvorkommen
 OUT: boolscher Wert
 SEM: Test, ob ein Praedikat vorliegt
 PRECOND: Predicates.Check muss vorher ausgewertet sein *)
BEGIN
  RETURN Sets.In( EAG.Pred, SymOcc[SO].SymInd )
END IsPredNont;

PROCEDURE isEqual*( I1, I2: Instruction ): BOOLEAN;
(*IN: zwei Instruktionen aus der Visit-Sequenz
  OUT: boolscher Wert
  SEM: Test, ob zwei Instruktionnen gleich sind; etwas optimiert  fuer den Fall, dass
       einer oder beide Parameter nil entsprechen *)
BEGIN
  IF ((I1 = NIL) & (I2 = NIL )) THEN RETURN TRUE
  ELSIF (I1 = NIL) OR (I2 = NIL ) THEN RETURN FALSE
  ELSIF ((I1 IS Visit) & (I2 IS Visit)) THEN
    RETURN (I1(Visit).SymOcc = I2(Visit).SymOcc) &
          (I1(Visit).VisitNo = I2(Visit).VisitNo)
  ELSIF ((I1 IS Leave) & (I2 IS Leave)) THEN
    RETURN (I1(Leave).VisitNo = I2(Leave).VisitNo)
  ELSIF ((I1 IS Call) & (I2 IS Call)) THEN
    RETURN (I1(Call).SymOcc = I2(Call).SymOcc)
  ELSE RETURN FALSE END
END isEqual;


PROCEDURE Init*;
(* SEM: Initialisierung der SOAG-Datenstruktur; Transformation der EAG-Datenstruktur *)
VAR A: EAG.Alt;
  i, a, Max: INTEGER;
BEGIN
  NEW( Sym, EAG.NextHNont );
  NEW( Rule, 128 );
  NEW( SymOcc, 256 );
  NEW( AffOcc, 512 );
  NEW( VS, 512 );
  NEW(DefAffOcc, EAG.NextVar );
  NEW(AffixApplCnt, EAG.NextVar );
  StorageName := NIL;
  NextSym := EAG.NextHNont;
  NextRule := firstRule;
  NextSymOcc := firstSymOcc;
  NextAffOcc := firstAffOcc;
  NextVS := firstVS;
  NextDefAffOcc := EAG.NextVar;
  NextStorageName := nil;
  NextAffixApplCnt := EAG.NextVar;

  Predicates.Check;
  FOR i := EAG.firstHNont TO EAG.NextHNont-1 DO
    Sym[i].FirstOcc := nil
  END;
  FOR i := EAG.firstHNont TO EAG.NextHNont-1 DO
    IF Sets.In( EAG.All, i ) THEN
      IF EAG.HNont[i].Def IS EAG.Rep THEN
        A := EAG.HNont[i].Def.Sub;
        WHILE A # NIL DO AppRepRule( A ); A := A.Next END
      ELSE
        A := EAG.HNont[i].Def.Sub;
        WHILE A # NIL DO AppRule( A ); A := A.Next END
      END;
      IF (EAG.HNont[i].Def IS EAG.Rep) OR
         (EAG.HNont[i].Def IS EAG.Opt)
      THEN AppEmptyRule( i, EAG.HNont[i].Def ) END
    END
  END;
  MaxAffNumInRule := 0;
  FOR i := firstRule TO NextRule-1 DO
    Max := Rule[i].AffOcc.End - Rule[i].AffOcc.Beg;
    IF Max > MaxAffNumInRule THEN MaxAffNumInRule := Max END;
```

```
    IF IsEvaluatorRule( i ) & (Max >= 0) THEN
      NEW( Rule[i].TDP, Max + 1 );
      NEW( Rule[i].DP, Max + 1 );
      FOR a := firstAffOccNum TO Max DO
        Sets.New( Rule[i].TDP[a], Max + 1 );
        Sets.New( Rule[i].DP[a], Max + 1 )
      END
    END
  END;
  MaxAffNumInSym := 0; NextPartNum := firstPartNum;
  FOR i := EAG.firstHNont TO EAG.NextHNont-1 DO
    IF Sets.In( EAG.All, i ) THEN
      Max := SymOcc[Sym[i].FirstOcc].AffOcc.End -
             SymOcc[Sym[i].FirstOcc].AffOcc.Beg;
      Sym[i].AffPos.Beg := NextPartNum;
      NextPartNum := NextPartNum + Max;
      Sym[i].AffPos.End := NextPartNum; INC(NextPartNum);
      IF Max > MaxAffNumInSym THEN MaxAffNumInSym := Max END;
      Sym[i].MaxPart := 0
    END
  END;
  NEW(PartNum, NextPartNum);
  MaxPart := 0;

  FOR i := EAG.firstVar TO EAG.NextVar-1 DO DefAffOcc[i] := -1; AffixApplCnt[i] := 0 END

END Init;

BEGIN
  IO.WriteText(IO.Msg, "SOAG-Evaluatorgenerator 1.06 Denis Kuniss 14.03.98\n");
IO.Update(IO.Msg)
END eSOAG.
```

5 Ermittlung von Affixpartitionen

Die in diesem Kapitel vorgestellte Theorie zur Berechnung von Affixpartitionen stützt sich im wesentlichen auf die Arbeit von Kröplin und Kannapin [KröpKann]. Es wurden lediglich einige Anpassungen an die verwendete Terminologie der EAGen vorgenommen.

5.1 Multi-visit EAGen

In diesem Abschnitt werden die multi-visit EAGen mit Hilfe der von Kastens [Kastens] angedeuteten Orientierungen definiert. Zur späteren Modifikation wird das OEAG-Verfahren rekonstruiert.

Ein Besuch eines Baumes beginnt und endet bei der Wurzel *r* und besteht dazwischen aus einer beliebigen Folge von Besuchen der Unterbäume, deren Wurzeln Söhne von *r* sind. Eine Familie geordneter Partitionen $(A_1(X),...,A_{n(X)}(X))$ für $X \in HN$ mit $n(X) \geq 0$ ist genau dann *visit-korrekt*, wenn für jeden Ableitungsbaum *t* die Werte aller Affixparameterinstanzen in $n(S)$ Besuchen von *t* berechnet werden können, wobei beim *i*-ten Besuch jedes Unterbaums, dessen Wurzel mit *X* markiert ist, genau die zugehörigen Instanzen der Affixparameter $A_i(X)$ ausgewertet werden. Damit ist eine EAG, für die es eine solche visit-korrekte Familie von Partitionen gibt, eine *(„simple“) multi-visit EAG*.

Um zumindest über ein konstruktives Kriterium zur Entscheidung, ob eine gegebene Familie von Partitionen visit-korrekt ist, zu verfügen, wird im folgenden eine äquivalente Charakterisierung der multi-visit EAGen angegeben. Statt dabei Auswertungsreihenfolgen durch totale Ordnungen auf $A(X)$ anzugeben, wird das angemessenere Konzept der Orientierungen verwendet, wobei eine Anordnung nur zwischen jeder inherited- und jeder synthesized-Affixposition bestimmt wird. Eine *Orientierung* des symmetrischen Produkts $A*B = (A \times B) \cup (B \times A)$ für Mengen *A* und *B* ist eine Relation $R \subseteq A*B$, wobei für jedes $a \in A$ und $b \in B$ entweder $(a,b) \in R$ oder $(b,a) \in R$ gilt. Im folgenden wird nun eine Bijektion zwischen den *kanonischen Partitionen* $(A_1(X),...,A_{n(X)}(X))$ von $A(X)$, bei denen jedes $A_i(X)$ für $1 \leq i < n(X)$ eine inherited-Affixposition und für $1 < i \leq n(X)$ eine synthesized-Affixposition enthält, und den entsprechenden azyklischen Orientierungen von $I(X) * S(X)$ erklärt.

Definition 5-1(*azyklische Orientierung*):
Sei $(A_1(X),...,A_{n(X)}(X))$ eine geordnete Partition von $A(X)$ für $X \in HN$. Dann ist

$$DS(X) = \{(a,b) \in I(X) \times S(X): a \in A_i(X) \text{ und } b \in A_j(X) \text{ für } i \leq j\} \cup \{(b,a) \in S(X) \times I(X): a \in A_i(X) \text{ und } b \in A_j(X) \text{ für } i > j\}$$

die zugehörige (azyklische) Orientierung von $I(X) * S(X)$.

Definition 5-2(*Konstruktion einer Partition*):
Sei $DS(X)$ eine azyklische Orientierung von $I(X) * S(X)$ für $X \in HN$. Dazu wird induktiv die Menge

$$\begin{aligned} B_0(X) &= \emptyset, \\ B_1(X) &= \{ a \in S(X): \text{es gibt kein } (a,b) \in DS(X)\}, \\ B_{i+1}(X) &= \{ a: \text{für alle } (a,b) \in DS(X) \text{ gilt } b \in B_i(X) \} \end{aligned}$$

definiert. Dann gibt es eine kleinste Zahl $n(X)$, für die $B_{2n(X)}(X) = A(X)$ gilt, und mit $A_i(X) = B_{2(n(X)-i+1)}(X) / B_{2(n(X)-i)}(X)$ ist $(A_1(X),...,A_{n(X)}(X))$ die zugehörige geordnete Partition von $A(X)$.

Für die Konstruktion der zugehörigen Partition von $A(X)$ gilt stets

$$\begin{aligned} A_i(X) \cap I(X) &= B_{2(n(X)-i+1)}(X) \setminus B_{2(n(X)-i)+1}(X) \\ A_i(X) \cap I(X) &= B_{2(n(X)-i)+1}(X) \setminus B_{2(n(X)-i)}(X), \end{aligned}$$

sogar wenn $DS(X)$ wie unten nur die transitive Hülle einer Teilmenge einer azyklischen Orientierung $I(X) * S(X)$ ist. In diesem Fall wird $(A_1(X),...,A_{n(X)}(X))$ als die zu $DS(X)$ gehörende *Kastens-Partition* bezeichnet, die dadurch charakterisiert ist, daß in einer möglichst kurzen Partition jede Affixposition so spät wie möglich („lazy“) angeordnet wird. Die zugehörige Orientierung wird dann *Kastens-Vervollständigung* genannt.

Es ist leicht einzusehen, daß eine EAG genau dann eine multi-visit EAG ist, wenn es für jedes $X \in HN$ eine Orientierung $DS(X)$ von $I(X) * S(X)$ gibt, so daß die erweiterten Abhängigkeiten

$$D^{-1} \cup \{(a^{(r,i)}, b^{(r,i)}): (a,b) \in DS(X^r_i)\}$$

azyklisch sind. Damit liegt das Entscheidungsproblem, ob eine gegebene EAG eine multi-visit EAG ist, in NP, und darüber hinaus haben Engelfriet und Filé für Attributgrammatiken gezeigt [EngFil], daß dieses Problem NP-vollständig ist.

Aus den Abhängigkeiten D^{-1} kann jedoch effizient eine notwendige Bedingung abgelesen werden, die jede visit-korrekte Familie von Partitionen erfüllen muß. Dazu werden alle direkten und sich ergebenden indirekten Abhängigkeiten zwischen Affixparametern eines Symbolvorkommens auf alle entsprechenden Affixparameter des gleichen Symbols übertragen.

Definition 5-3 (*induzierte Abhängigkeiten*):
Sei *DP* eine Relation auf Affixparametern. Dann werden die induzierten Abhängigkeiten als kleinste Relation definiert, die

$$ind(DP) = DP \cup \{(a^{(q,j)}, b^{(q,j)}): (a^{(r,i)}, b^{(r,i)}) \in ind(DP)^+ \text{ für } X^r{}_i = X^q{}_j \}$$

erfüllt, wobei R^+ die transitive Hülle einer Relation *R* bezeichnet.

Die induzierten Abhängigkeiten $IDP = ind(D^{-1})$ spiegeln nicht nur transitive Abhängigkeiten zu den Ableitungsbäumen wider, sondern auch solche Anordnungen, die sich zwingend aus der Abstraktion vom Kontext der Symbole ergeben. Die Projektion von *IDP* auf $A(X)$ wird mit $IDS(X)$ bezeichnet, d.h.

$$IDS(X) = \{(a,b) :(a^{(r,i)}, b^{(r,i)}) \in IDP \text{ für } X^r{}_i = \mathrm{X}\},$$

und für jede visit-korrekte Familie von Partitionen mit den zugehörigen Orientierungen $DS(X)$ gilt damit

$$IDS(X) \cap (I(X) * S(X)) \subseteq DS(X).$$

Weiterhin ist für jede multi-visit EAG *IDP* azyklisch. Wenn dies der Fall ist, so ist $IDS(X)$ für jedes $X \in HN$ die transitive Hülle einer Teilmenge einer azyklischen Orientierung von $I(X) * S(X)$.

Aus der angegebenen notwendigen Bedingung hat Kastens in [Kastens] eine Unterklassse der multi-visit AGen bestimmt, für die eine visit-korrekte Familie von Partitionen effizient berechnet werden kann, und die auch sehr einfach, wie von Kutza gezeigt wurde [Kutza], auf EAGen übertragbar ist. Eine EAG ist nämlich genau dann eine OEAG, wenn *IDP* azyklisch ist und die zu $IDS(X)$ gehörende Familie von Kastens-Partitionen visit-korrekt ist.

5.2 Sequentiell orientierbare EAGen

Im folgenden soll nun das OEAG-Verfahren modifiziert werden, um systematisch größere Unterklassen der multi-visit EAGen zu bestimmen.

Die Schwäche des OEAG-Verfahrens liegt darin begründet, daß die Partitionen aus $IDS(X)$ für $X \in HN$ unabhängig voneinander bestimmt werden. Zur Ausbesserung skizzierte Kastens in [KaHuZi] daher ein „vorsichtigeres“ Verfahren, bei dem die Familie von Partitionen symbolweise gebildet wird und dabei neue induzierte Abhängigkeiten berücksichtigt werden, die sich aus bereits bestimmten Partitionen ergeben. Diejenigen multi-visit AGen, für die dieses sequentielle Verfahren erfolgreich ist, werden als „automatically arranged orderly“ bezeichnet. Dieser Ansatz von Kastens trifft vor allem eine Aussage über die Reihenfolge der Symbole bei der Ermittlung einer Partition. Dieser Einfluß der Reihenfolge wird im folgenden in den Vordergrund der weiteren Untersuchungen gestellt.

Das in Abbildung 5-1 dargestellte Beispiel einer EAG vereinigt in zwei Regeln die Abhängigkeiten eines „left-to-right threading“ (a) und eines „right-to-left threading“ (b). Nach Reps und Teitelbaum [RepTei] illustrieren multi-visit AGen, die diesem Typ von EAGen entsprechen, eine praktisch relevante Konstellation, bei der das OAG-Verfahren und somit auch das OEAG-Verfahren scheitern. Da es in diesem Fall nur ein Symbol gibt, kann auch das sequentielle Verfahren nicht erfolgreich sein. An dieser Stelle zeigt sich die Überlegenheit des Konzepts der Orientierungen. Jede geordnete Partition der Menge $\{(X, \{a\}^*\{b\}): a \in I(X)$ und $b \in S(X)$ für $X \in HN\}$ ist eine *Orientierungsreihenfolge*. Eine Orientierungsreihenfolge $(C_1,...,C_m)$ ist *trivial*, wenn $m = 1$ gilt, und *elementar*, wenn jede Menge C_j nur genau ein Paar enthält.

Definition 5-4 (*unmittelbar erfolgreiche* Orientierungsreihenfolge):
Sei $(C_1,...,C_m)$ eine Orientierungsreihenfolge, dann wird induktiv folgendermaßen definiert: initial gilt $IDP_0 = IDP$. Für azyklisches IDP_{j-1} mit den zu den Kastens-Partitionen von $\{(a,b) :(a^{(r,i)}, b^{(r,i)}) \in IDP_{j-1}$ für $X^r_i = X\}$ gehörenden Orientierungen $DS_j(X)$ von $I(X) * S(X)$ gilt

$$IDP_j = ind(IDP_{j-1} \cup \{(a^{(r,i)}, b^{(r,i)}): (a,b) \in DS_j(X) \text{ für } (X, \{a\}^*\{b\}) \in C_j\}).$$

Damit ist die Orientierungsreihenfolge *unmittelbar erfolgreich*, wenn jedes IDP_j für $0 \le j \le m$ azyklisch ist. In diesem Fall sind die Projektionen von IDP_m auf $A(X)$, d.h.

$$DS(X) = \{(a,b) :(a^{(r,i)}, b^{(r,i)}) \in IDP_m \text{ für } X^r_i = X\},$$

azyklische Orientierungen von $I(X) * S(X)$, und die Familie der zugehörigen Partitionen ist visit-korrekt.

Offensichtlich ist eine EAG genau dann eine OEAG, wenn die triviale Orientierungsreihenfolge unmittelbar erfolgreich ist. Weiterhin ist dann jede Orientierungsreihenfolge unmittelbar erfolgreich, und es ist leicht einzusehen, daß für jedes $DS_j(X)$ die Kastens-Partition mit der Kastens-Partition von $IDS(X)$ übereinstimmt. Also wird für jede Orientierungsreihenfolge dieselbe visit-korrekte Familie von Partitionen wie beim OEAG-Verfahren bestimmt. Allgemein gilt, daß für jede unmittelbar erfolgreiche Orientierungsreihenfolge auch jede *feinere Orientierungsreihenfolge*, in der jedes C_j durch eine beliebige geordnete Partition ersetzt ist, unmittelbar erfolgreich ist.

Das von Kastens skizzierte „vorsichtigere“ Verfahren läßt sich nun dadurch charakterisieren, daß für eine festgelegte Reihenfolge $(X_1,...,X_m)$ der Symbole die Orientierungsreihenfolge $(C_1,...,C_m)$ mit $C_j = \{(X_j, \{a\}^*\{b\}): a \in I(X_j)$ und $b \in S(X_j)\}$ unmittelbar erfolgreich ist. Dabei handelt es sich um eine *symbolweise Orientierungsreihenfolge*, die eindeutig durch die Reihenfolge der Symbole bestimmt ist.

Definition 5-5 (*erfolgreiche* Orientierungsreihenfolge):
Sei $(C_1,...,C_m)$ nun eine elementare Orientierungsreihenfolge und sei IDP_j und $DS_j(X)$ wie bisher definiert. Ist IDP_j jedoch zyklisch, so wird statt dessen mit umgekehrten Abhängigkeiten definiert:

$$IDP_j = ind(IDP_{j-1} \cup \{(b^{(r,i)}, a^{(r,i)}): (a,b) \in DS_j(X_j) \text{ für } (X^r_i, \{a\}^*\{b\}) = C_j\}).$$

Die Orientierungsreihenfolge ist dann *erfolgreich*, wenn wieder jedes IDP_j für $0 \le j \le m$ azyklisch ist.

Offensichtlich ist jede elementare Orientierungsreihenfolge, die unmittelbar erfolgreich ist, auch erfolgreich, und für jede OEAG ist also jede elementare Orientierungsreihenfolge erfolgreich. Da dabei die erste Alternative stets azyklisch ist, ergibt sich auch hier dieselbe visit-korrekte Familie von Partitionen wie beim OEAG-

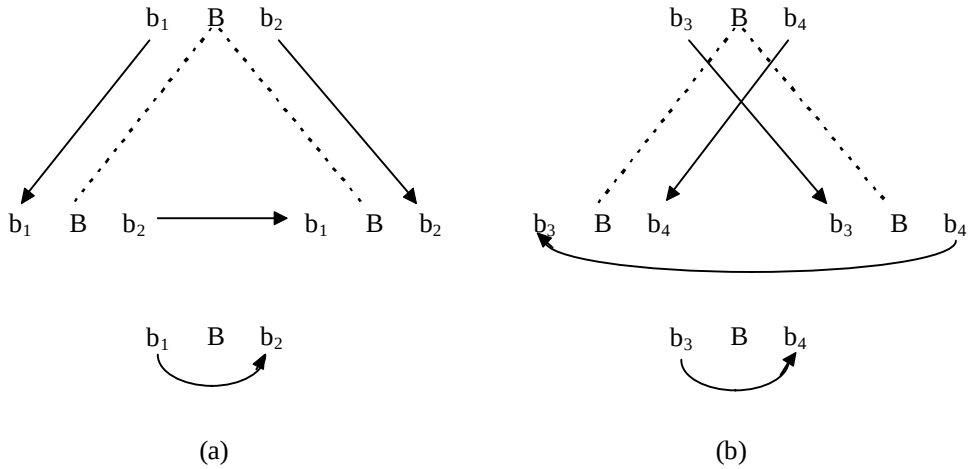


Abbildung 5-1: Ein "left-to-right threading" (a) und ein "right-to-left threading" (b)

Verfahren. Andererseits ist leicht einzusehen, daß es EAGen mit erfolgreichen Orientierungsreihenfolgen gibt, für die keine Orientierungsreihenfolge schon unmittelbar erfolgreich ist.

Eine EAG ist genau dann eine *sequentiell orientierbare* EAG (SOEAG), wenn es eine erfolgreiche Orientierungsreihenfolge gibt. Damit ist natürlich jede OEAG eine SOEAG, und jede SOEAG ist eine multi-visit EAG.

5.3 Zwei heuristische Teilklassen der SOEAGen

Da das Entscheidungsproblem, ob eine EAG eine SOEAG ist, NP-vollständig ist, erscheint die Implementierung eines SOEAG-Evaluator-Generators nicht ratsam. Statt dessen werden zwei sinnvolle heuristische Teilklassen der SOEAGen vorgestellt, die konstruktiv definiert werden, d.h. es wird ein Algorithmus zur Berechnung der Orientierungsreihenfolge angegeben und damit die Teilklasse definiert.

Ausgehend vom OEAG-Verfahren lassen Erwägungen leichter und effizienter Implementierbarkeit kaum eine andere vernünftige Entscheidung zu, als bezüglich der verfolgten Orientierungsstrategie grundsätzlich symbolweise, und zwar in einer beliebigen Ad-hoc-Reihenfolge $(X_1,...,X_k)$ der Symbole $X \in HN$, vorzugehen. Dazu sind folgende zwei Verfeinerungsgrade wachsender Mächtigkeit denkbar:

1. Die Orientierungsreihenfolge $(C_1,...,C_m)$ mit $C_j = \{(X, \{a\}^*\{b\}): a \in I(X)$ und $b \in S(X)$ für $X \in HN\}$ wird zu einer elementaren verfeinert, d.h. für jedes Symbol X_i wird eine Partition einelementiger Mengen gebildet. Jede dieser Mengen wird aufsteigend einem C_j zugeordnet. Der Erfolg jeder Orientierung wird sofort überprüft. Im Falle von Mißerfolgen wird jedoch aus Effizienzgründen von einem Backtracking abgesehen. Es wird nur der unmittelbare Erfolg *einer* Ad-hoc-Reihenfolge pro Symbol anstatt aller ihrer Permutationen überprüft. Diese Vorgehensweise wird das (Ad-hoc-)ESO-Verfahren (ad hoc elementar symbolsequentiell orientierend) genannt und approximiert die erste vorzustellende heuristische Grammatik-Teilklasse.
2. Wenig komplizierter zu implementieren ist dieselbe Strategie bei Überprüfung auf eine erfolgreiche statt unmittelbar erfolgreiche Orientierungsreihenfolge. Dies erfordert gegebenenfalls die Rücknahme vorgenommener Erweiterungen im Abhängigkeitsgraphen der Affixparameter einer Regel, was im schlechtesten Fall zu einem erhöhten Aufwand führt. Diese Vorgehensweise wird (Ad-hoc-)ESOB-Verfahren (ad hoc elementar symbolsequentiell orientierend mit lokalem Backtracking) genannt und beschreibt konstruktiv die zweite, mächtigere Grammatik-Teilklasse.

Es sei noch einmal betont, daß, gegeben eine multi-visit EAG, die nicht geordnet ist, die erfolgreiche Generierung eines Evaluators von den ad hoc gewählten Reihenfolgen *und* der Feinheit des Verfahrens abhängt: Scheitert ein Verfahren, so kann ein bei gleicher Reihenfolge feiner vorgehendes Verfahren, aber auch eine andere Reihenfolge zur erfolgreichen Erzeugung eines Evaluators führen. In jedem Fall sind alle Verfahren auf OEAGen erfolgreich.

Es ist leicht einzusehen, daß eine Implementierung zur Approximation der zweiten mächtigeren Grammatik-Teilklasse die Grammatiken der ersten Teilklasse mit berechnen würde. In der überarbeiteten Fassung von [KröpKann] werden die beiden Grammatik-Teilklassen, wie auch die beiden sie konstruktiv beschreibenden Verfahren ESO und ESOB, nicht mehr explizit erwähnt. Statt dessen wird nur noch von einem die SOAGen bzw. SOEAGen *approximierenden* Verfahren gesprochen. Ich schließe mich dieser Sichtweise an und werde von nun an nur noch von einem Verfahren sprechen, ohne auf die zwei angesprochenen Grammatik-Teilklassen zu referenzieren.

5.4 Dynamische transitive Hülle

Die durch die Affixformen induzierten Abhängigkeiten zwischen den Affixparametern einer Regel werden in der Datenstruktur *TDP* vorgehalten. Diese Datenstruktur wird auch zur Speicherung der transitiv abgeschlossenen Zwischenergebnisse IDP_j^+ benutzt, wobei beim j-ten Orientierungsschritt jeweils IDP_j^+ aus IDP_{j-1}^+ *in situ* berechnet wird. Die Datenstruktur *TDP* ist im vorliegenden Evaluatorgenerator im Modul `SOAG` als Datenelement der Datenstruktur `RuleDesc` im Feld `Rule` realisiert.

Für den Aufwand des approximierenden Verfahrens zur Erzeugung von SOEAG-Evaluatoren prägend ist der verwendete Algorithmus zum transitiven Erweitern der Relation *TDP*, wobei stets nur Abhängigkeiten zwischen Affixparametern derselben Regel auftreten. Im folgenden wird in formeller Notation ein effizienter Algorithmus von Ibaraki und Katoh zur inkrementellen transitiven Hüllenbildung einer monoton wachsenden Relation vorgestellt.

Definition 5-6 (*inkrementelle transitive Hülle*):
Sei S eine endliche Menge und $R \cup \{(u,v)\} \subseteq S^2$. Dann sind die Operatoren $^+$ und * gegeben durch

$$\begin{aligned} \varnothing^+ &= \varnothing, \\ (R \cup \{(u,v)\})^+ &= R^+ \cup \{x : x\ R^*\ u\} \times \{y : v\ R^*\ y\} \text{ und} \\ R^* &= R^+ \cup \{(x,x) : x \in S\}. \end{aligned}$$

Es ist sofort einzusehen, daß $R \cup \{(u,v)\}^+$ aus R^+ mit dem Aufwand $O(|S|^2)$ bestimmt werden kann. Es werden nun, bezogen auf R und (u,v), besondere Vorgänger- und Nachfolgermengen

$$\begin{aligned} P_R(u,v) &= \{x : x\ R^*\ u, \text{ aber nicht } x\ R^+\ v\} \text{ und} \\ S_R(u,v) &= \{x : v\ R^+\ x, \text{ aber nicht } u\ R^+\ x\} \end{aligned}$$

eingeführt. Damit ist zunächst verschärfend

$$(R \cup \{(u,v)\})^+ = R^+ \cup P_R(u,v) \times \{y : v\ R^*\ y\}, \qquad (1)$$

und Ibaraki und Katoh zeigen in [IbaKat], daß derart das transitive Erweitern der Ausgangsrelation R^+ durch sukzessives Hinzufügen von q (neuen) Beziehungen $(u,v) \in S^2$ kumulativ nur einen Aufwand von $O(|S|^3)$ erfordert. Dies kann sogar zu $O(q \cdot |S|)$ verbessert werden, wie unschwer zu zeigen ist. Die Schlüsselidee ist, daß jede neu hinzukommende Beziehung nur $|S|$- statt $|S|^2$-mal aufgenommen wird. Mithin kann die Relation R^+, beginnend bei $\varnothing$, inkrementell in $O(|S| \cdot |R^+|)$ aufgebaut werden.

Ebenso ist einfach zu zeigen, daß sich der asymmetrische Algorithmus (1) von Ibaraki und Katoh weiter zum symmetrischen Algorithmus

$$(R \cup \{(u,v)\})^+ = R^+ \cup P_R(u,v) \times S_R(u,v).$$

verschärfen läßt. Jedoch bleibt diese Verbesserung im kumulativen Aufwand ohne Konsequenz.

Bei der Transformation des induktiven Berechnungsschemas in ein Programm, wobei die in Rede stehenden Relation R^+ durch eine Adjazenzmatrix repräsentiert wird, ist darauf zu achten, das Bestimmen der Nachfolgermenge $S_R(u,v)$, was den Aufwand $\theta(|S|)$ erfordert, herauszuziehen, um das Aufzählen von $P_R(u,v) \times S_R(u,v)$ nicht mit Aufwand $\theta(|S|^2)$ zu formulieren.

Die transitive Erweiterung der Relation *TDP* ist im Modul `SOAGPartition` in der Prozedur `AddTDPTrans` implementiert. In der ersten Schleife der Prozedur wird in der Liste `NUV` die Nachfolgermenge $S_R(u,v)$ bestimmt. Die Nachfolgermenge kann als Liste implementiert werden, da jeder Knoten höchstens einmal hinzugefügt wird. In der zweiten Schleife erfolgt die Aufzählung der Relation $P_R(u,v) \times S_R(u,v)$, die den transitiven Abschluß in *TDP* bildet. Beide Schleifen sind, wie gefordert, nicht ineinander verschränkt, um den oben beschriebenen Aufwand einzuhalten.

5.5 Dynamisches topologisches Sortieren

In diesem Abschnitt wird ein Algorithmus präsentiert, der in einem einzigen topologischen Sortiervorgang pro Symbol eine erfolgreiche Orientierungsreihenfolge anzugeben versucht.

Betrachtet man $TDP = IDP_{k\text{-}1}{}^+$, dann sind für $X \in HN$ die Projektionen

$$DS(X) = \{ (a,b) \in I(X) * S(X) : (a^{(r,i)}\},b^{(r,i)}) \in TDP \text{ für } X^r{}_i\text{=X}\}$$

jeweils Teilmenge einer (azyklischen) Orientierung von $I(X) * S(X)$, welche es zu einer vollständigen (azyklischen) Orientierung zu erweitern gilt. (Diese Formulierung von $DS(X)$, die im Gegensatz zur Definition aus Abschnitt 5.1 einige zusätzliche transitive Abhängigkeiten enthält, erweist sich für den zu entwickelnden Algorithmus als günstiger.) Grundsätzlich werden diese Erweiterungen symbolweise vorgenommen, weswegen im folgenden die Betrachtung eines festen $X \in HN$ genügt. Wie bereits angedeutet, wird so vorgegangen, daß im Falle einer vorliegenden OEAG Orientierungen resultieren, die Kastens-Vervollständigungen von $IDS(X)$ für $X \in HN$ sind. Naheliegenderweise ist deswegen Ausgangspunkt die mit topologischem Sortieren verwandte Technik für die Bestimmung der Kastens-Vervollständigungen der obigen Projektion $DS(X)$. Wir führen für das in Rede stehende $X \in HN$ zunächst folgende symmetrische Relation $unor \subseteq I(X) * S(X)$ ein, welche den noch zu orientierenden Anteil der Affixpositionspaare aus $I(X) * S(X)$ angibt:

$$a\ unor\ b \quad \Leftrightarrow \quad DS(X) \cup (\{a\} * \{b\}) = \varnothing.$$

Das Festlegen einer Auswertungsreihenfolge zwischen zwei Affixpositionen $a,b \in A(X)$ mit *a unor b*, und dies mit der Bevorzugung, *a nach b* zu berechnen, wird abstrakt als Prozeduraufruf *orient*(*a*,*b*,*X*, *new*) mit dem

Ausgangsparameter *new* formuliert. In der folgenden Skizze der Wirkung eines Aufrufs *orient*(*a*,*b*,*X*, *new*) ist die Rücknahme der Abhängigkeit (*b*, *a*) zu Gunsten von (*a*, *b*) zu erkennen, falls durch (*b*, *a*) ein Zyklus entsteht.

TDP' := *TDP*;
TDP := *ind*(*TDP*' ∪ {($b^{(r,i)}$,$a^{(r,i)}$): $X^r{}_i$=*X* })$^+$;
IF *TDP* zyklisch THEN
  TDP := *ind*(*TDP*' ∪ {($a^{(r,i)}$,$b^{(r,i)}$): $X^r{}_i$=*X* })$^+$;
  IF *TDP* zyklisch THEN HALT END
END;
new := {(*a*,*b*) ∈ *I*(*X*) * *S*(*X*): ($a^{(r,i)}$,$b^{(r,i)}$) ∈ *TDP* \ *TDP*' für ein $X^r{}_i$=*X* }

Nach Termination gilt offensichtlich

new ∪ *DS*(*X*) = { (*a*,*b*) ∈ *I*(*X*) * *S*(*X*): ($a^{(r,i)}$,$b^{(r,i)}$) ∈ *TDP* für ein $X^r{}_i$=*X* }.

Es ist klar, daß schließlich *new* nicht nur entweder (*b*,*a*) oder (*a*,*b*) enthält, sondern möglicherweise auch noch andere neue, auf *A*(*X*) projizierte, induzierte Auswertungsreihenfolgen. Um *TDP* nachgeführt zu werden, ist die Relation *DS*(*X*) entsprechend um diese neuen Abhängigkeiten *new* zu erweitern, was wiederum Rückwirkungen hat auf den topologischen Sortiervorgang von *DS*(*X*). Der angestrebte Algorithmus läßt sich somit charakterisieren als ein dynamisches topologisches Sortieren der Projektion *DS*(*X*), verschränkt mit der Berechnung von *TDP* und der Überprüfung von Zyklen.

Zur Herleitung des Algorithmus wird im folgenden ein wesentliches Zwischenergebnis beschrieben:
Es sei j ≥ 0 derart, daß die Berechnung der Mengen ∅ = $B_0(X) \subseteq B_1(X) \subseteq ... \subseteq B_{2n(X)}(X)$=*A*(*X*) bereits bis einschließlich $B_j(X)$ abgeschlossen ist, da diese Mengen zunächst die Eigenschaften

$B_0(X)$ = ∅,
$B_1(X)$ = { *a* ∈ *S*(*X*): es gibt kein (*a*,*b*) ∈ *DS*(*X*)}

sowie für *i* < *j*

$B_{i+1}(X)$ = { *a*: für alle (*a*,*b*) ∈ *DS*(*X*) gilt *b* ∈ $B_i(X)$ }

besitzen und da zudem wegen der Gültigkeit von

{*b*: *a unor b*} = ∅ für alle *a* ∈ $B_j(X)$ (2)

azyklische Erweiterungen von *DS*(*X*) auf die Gültigkeit dieser Invarianten ohne Einfluß bleiben. Die Mengenvariable

cur = { *a* ∈ *A*(*X*) \ $B_i(X)$: für alle (*a*,*b*) ∈ *DS*(*X*) gilt *b* ∈ $B_j(X)$ } (3)

enthält Kandidaten für in $B_{j+1}(X)$ gegenüber $B_j(X)$ neue Affixpositionen. (Die Berechnung der Mengen $B_i(X)$ und auch von *n*(*X*) geschieht nur implizit.) Offensichtlich ist nun die Gültigkeit von

{*b*: *a unor b*} = ∅ für alle *a* ∈ *cur*

und mithin

{*b*: *a unor b*} = ∅ für alle *a* ∈ $B_{j+1}(X)$ (4)

anzustreben, indem für jede Affixposition *a* ∈ *cur*, zu der noch Auswertungsreihenfolgen bezüglich Affixpositionen *b* unbekannt sind, diese jetzt festgelegt werden, wobei jeweils die Reihenfolge, *a nach b* auszuwerten, bevorzugt wird.

Betrachtet man daher an dieser Stelle (*a*,*b*) mit *a* ∈ *cur* und *a unor b* und erweitert *TDP* vermöge eines Aufrufs *orient*(*a*,*b*,*X*, *new*), dann ist zu untersuchen, wie die angegebenen Invarianten zu erhalten sind, wenn nun *DS*(*X*) sukzessive um die neuen Beziehungen (*c*,*d*) ∈ *new* erweitert wird. Einher mit der Anweisung

DS(*X*) := *DS*(*X*) ∪ {(*c*,*d*)}

geht zunächst die Korrektur der Relation *unor* gemäß Definition durch

unor := *unor* \ ({*c*}*{*d*}).

Nun ist wegen (2) klar, daß *DS*(*X*), eingeschränkt auf Attribute aus $B_j(X)$ ∪ *cur*, bereits eine Orientierung ist, und (*c*,*d*) erweitert *DS*(*X*) zudem azyklisch. Daher folgt sofort, daß *c*,*d* ∉ $B_j(X)$ und |{*c*,*d*} ∪ *cur* | ≤ 1 gilt. Die nur im Falle *c* ∈ *cur* verletzte Invariante (3) erfordert also, *c* aus *cur* zu entfernen.

Beim Erweitern von $DS(X)$ um alle neuen Abhängigkeiten $(c,d) \in new$ wird entweder (b,a) oder (a,b) in $DS(X)$ aufgenommen, womit auf jeden Fall anschließend nicht mehr *a unor b* gilt; wurde (a,b) aufgenommen, so ist dann sogar $a \notin B_{j+1}(X)$. Nach Behandlung aller *a*, *b* mit *a unor b* ist also $cur = B_{j+1}(X) \setminus B_j(X)$, und es gilt tatsächlich (4).

Nun wird *j* (implizit) erhöht und es ist wieder (3) sicherzustellen. Dieses wird durch eine Anweisung

cur := *leaves*

erreicht, wobei die neuen Größen folgendermaßen definiert sind:

$$leaves = \{ a \in A(X) \setminus (B_j(X) \cup cur) : deg(a)=0 \},$$
$$deg(a) = |DS(X)(a) \setminus B_j(X)| \quad \text{für alle } a \in A(X).$$

Wie *deg* und *leaves* effizient mitgeführt werden können, wird im Abschnitt 5.6 „Implementierungsdetails“ näher beschrieben.

5.6 Implementierungsdetails

Zu Beginn einer jeden Partitionsbestimmung müssen alle direkten Abhängigkeiten zwischen den Affixparametern einer Regel *r* im Abhängigkeitsgraphen `SOAG.Rule[r].TDP` eingetragen werden. Dies geschieht für alle Regeln durch die Prozedur `ComputeDP`. Wird ein Affix in der Affixform eines applizierenden Affixparameters verwendet, so hängt dieser direkt vom definierenden Affixparameter, der das definierende Affix gleichen Namens enthält, ab. Vor der Bestimmung der Abhängigkeiten in einem Regelkontext, muß jedem in der Regel vorkommenden Affix der Affixparameter, in dessen Affixform er vorkommt, zugeordnet werden. Diese Zuordnung wird durch die Prozedur `SetAffOccforVars` im modul-globalen Feld `VarBuf` vorgenommen, das für jedes Affix einer Regel folgende Datenstruktur enthält

```
TYPE
    VarBufDesc = RECORD
      AffOcc, Sym,
      Num, VarInd: INTEGER
    END;
    OpenVarBuf = POINTER TO ARRAY OF VarBufDesc;

VAR
    VarBuf: OpenVarBuf;
    NextVarBuf: INTEGER;
```

Das Feld `VarBuf` wird für jede Regel geleert und wiederbenutzt, da die darin enthaltenen Informationen später nicht mehr benötigt werden.

Zur Explikation der Abhängigkeiten eines applizierenden Affixparameters muß für jedes darin enthaltene Affix der Affixparameter seines definierenden Affixes bekannt sein. Die Prozedur `ComputeDefAffOcc` berechnet für jede Affixvariable der aktuelle Regel im Feld `SOAG.DefAffOcc`, das parallel zu `EAG.Var` liegt, den Affixparameter, in dem sich das definierende Affix der Affixvariable befindet. Dies erfolgt durch lineare Suche im Feld `VarBuf`. Wird kein definierendes Affix gefunden, so liegt eine Verletzung der Bedingung der Links-Definiertheit vor; eine Fehlermeldung und die Position des undefinierten Affixes wird ausgegeben und das Programm abgebrochen.

Die Prozedur `ComputeAffixApplCnt` berechnet für alle Affixvariablen die Anzahl ihrer Applikationen in Synthesen und Vergleichen. Dazu wird das Feld `VarBuf` linear durchsucht und für nichtdefinierende Affixe und Affixe in applizierenden Affixparametern der Wert im Feld `AffixApplCnt` entsprechend inkrementiert. Zusätzlich werden für alle in einer Regel *r* durchzuführenden Vergleiche Kanten in die Datenstruktur `SOAG.Rule[r].DP` eingetragen. Diese Abhängigkeiten sind nicht wirklich vorhanden, sondern verweisen von Affixparametern mit definierenden Affixen ausgehend auf Affixparameter, die gleichnamige Affixe bzw. durch „#“ negierte gleichnamige Affixe enthalten. Diese zusätzlichen Informationen werden später bei der Optimierung der Speicherung von Affixvariablen benötigt. Vor Anwendung der Prozedur `ComputeAffixApplCnt` muß die Datenstruktur `SOAG.DefAffOcc` für die aktuelle Regel vollständig berechnet sein.

Nach Berechnung des Feldes `SOAG.DefAffOcc` werden alle direkten Abhängigkeit mittels der Prozedur `AddTDPTrans` in den Abhängigkeitsgraphen eingetragen und die transitive Vervollständigung gebildet. Alle durch `AddTDPTrans` ermittelten Abhängigkeiten werden zusätzlich durch Eintrag in den Kellerspeicher

`MarkedEdges` markiert. Parallel zur Berechnung von *TDP* werden in `ComputeDP` alle direkten Abhängigkeiten einer Regel *r* in die Datenstruktur `SOAG.Rule[`*r*`].DP` eingetragen.

In der Prozedur `ComputeInducedTDP` werden nacheinander alle markierten Kanten aus dem Kellerspeicher `MarkedEdges` ausgetragen. Für jede ausgetragene Kante werden alle transitiven Abhängigkeiten berechnet und im Abhängigkeitsgraphen nachgetragen, falls sie noch nicht enthalten waren. Alle neu eingetragenen Kanten erhalten wiederum durch Aufnahme in den Kellerspeicher `MarkedEdges` eine Markierung. Diese Vorgehensweise wird so lange wiederholt, bis der Kellerspeicher leer ist. Die Termination der Schleife wird durch die Endlichkeit der transitiven Hülle eines Abhängigkeitsgraphen gewährleistet.

Die Prozedur `Orient` realisiert die im Abschnitt 5.5 „Dynamisches topologisches Sortieren“ vorgestellte abstrakte Prozedur *orient*. Ergibt sich bei der Orientierung zweier Affixpositionen in *TDP* ein Zyklus, so müssen vor dem Einfügen der umgekehrten Abhängigkeit alle vorher getätigten Erweiterungen des *TDP* zurückgenommen werden. Dazu muß die modul-globale Variable `Phase` auf `dynTopSort` gesetzt sein. Ist dies der Fall, so werden während der transitiven Vervollständigung in `AddTDPTrans` durch die Prozedur `AddTDPChange` im Feld `ChangeBuf` mit der Struktur

```
TYPE
   ChangeBufDesc = RECORD
     RuleInd,
     AffOccInd1,
     AffOccInd2: INTEGER;
   END;
   OpenChangeBuf: POINTER TO ARRAY OF ChangeBufDesc;

VAR
   ChangeBuf: OpenChangeBuf;
   NextChangeBuf: INTEGER;
```

alle Änderungen des *TDP* aufgezeichnet. Diese Erweiterungen können dann durch die Prozedur `ResetTDPChanges` rückgängig gemacht werden. Die von *orient* zurückgelieferte Menge *new* wird in der Variable `New` als Menge vom Typ `BSets` implementiert. Jeder Eintrag in dieser Menge ergibt ein Paar Affixpositionen:

```
erstes Element  = Mengeneintrag DIV Separator
zweites Element = Mengeneintrag MOD Separator
```

Der `Separator` wird in der Prozedur `DynTopSort` berechnet.

Die Prozedur `DynTopSortSym` implementiert die im Abschnitt 5.5 „Dynamisches topologisches Sortieren“ motivierte Vorgehensweise zur topologischen Sortierung der Affixpositionsabhängigkeiten eines Symbols unter Heranführung an eine erfolgreiche bzw. unmittelbar erfolgreiche Orientierung. Der Algorithmus beginnt mit der Initialisierung der Datenstruktur `DS`.

```
TYPE
   DS: POINTER TO ARRAY OF ARRAY OF INTEGER;
```

Sie realisiert für ein Symbol *X* die Menge der azyklischen Orientierungen *DS*(*X*). Da außerdem *DS*(*X*) ∩ *unor* gilt, wird in der Datenstruktur `DS` zusätzlich die Relation *unor* integriert. Da die Orientierungen für alle Symbole unabhängig und nacheinander berechnet werden, kann die Datenstruktur `DS` für jedes Symbol wiederverwendet werden. Für einen Eintrag im Feld `DS` gilt somit

`DS[a][b] = element` ⇔ $(a, b) \in DS(X)$
`DS[a][b] = nil` ⇒ $(a, b) \notin DS(X)$
`DS[a][b] = unor` ⇔ a *unor* b

Während der Berechnung der Relation *unor* wird gleichzeitig das Feld `Deg` gefüllt, das für jede Affixposition des aktuellen Symbols die Anzahl der ausgehenden Kanten enthält.

Anschließend werden die beiden Mengen `Cur` und `Leave` initialisiert, die *cur* und *leave* aus Abschnitt 5.5 „Dynamisches topologisches Sortieren“ entsprechen. Beide Mengen sind vom Typ `ASets`, um Elemente mit konstanten Aufwand löschen zu können.

Der eigentliche topologische Sortiervorgang wird durch eine REPEAT-Schleife kontrolliert, die solange ausgeführt wird, bis alle Affixpositionen des aktuellen Symbols einer Partitionsmenge zugeordnet sind. Diese Bedingung tritt ein, wenn die Menge `Cur` leer ist, was der Abbruchbedingung der Schleife entspricht. Durch

die Verwendung einer REPEAT-Schleife wird die Schleife mindestens einmal durchlaufen, auch wenn `Cur` nach der Initialisierung leer ist. Nach der Initialisierung kann `Cur` genau dann leer sein, wenn es in der Partition $A_{n(X)}(X)$ eines Symbols X keine synthesized- sondern nur inherited-Affixpositionen gibt, also $B_1(X)$ leer ist.

Die in der Implementierung berechneten Partitionsmengen entsprechen aus implementierungstechnischen Gründen nicht ganz den theoretisch hergeleiteten. Für die in der Implementierung berechneten Partitionsmengen $A^I_j(X)$ gilt:

$$A^I_j(X) = B_j(X) \setminus B_{j-1}(X),$$

und die hergeleiteten Partitionsmengen ergeben sich dann aus

$$\begin{aligned} A_i(X) &= A^I_{2(n(X)-i+1)}(X) \cup A^I_{2(n(X)-i)+1}(X) \text{ mit} \\ A^I_{2(n(X)-i+1)}(X) &= A_i(X) \cap I(X) \text{ und} \\ A^I_{2(n(X)-i)+1}(X) &= A_i(X) \cap S(X). \end{aligned}$$

Die berechneten Partitionsmengen enthalten also abwechselnd nur inherited- oder synthesized-Affixpositionen. Dies schränkt den implementierten Algorithmus jedoch nicht weiter ein, da die gesuchten Partitionsmengen, wie oben angeben, sehr leicht ableitbar sind. Im folgenden sind als Partitionsmengen stets die implementierungstechnischen Partitionsmengen gemeint.

In jedem Schleifendurchlauf wird in `Cur` implizit genau eine Partitionsmenge der gesuchten Affixpartition berechnet. Alle Elemente der Menge `Cur` sind potentielle Kandidaten für diese Menge. Da die Affixpositionen in der Menge `Cur` in einer beliebigen, aber festen Reihenfolge untersucht werden sollen, werden sie vor der zweiten Schleife, die den Orientierungsaufruf `Orient` enthält, in die Liste `LastCur` übertragen. Die Reihenfolge der Elemente ist damit festgelegt. Dies ist erforderlich, da sich aufgrund der Implementierung des Mengentyps `ASets` durch Löschen von Elementen Änderungen in der Reihenfolge ergeben können.

Nach der Orientierung aller Affixpositionen in `Cur` respektive der zu ihnen in *unor*-Relation stehenden Affixpositionen wird die Partitionsnummer `Part` um eins erhöht. Alle in der Menge `Cur` verbliebenen Affixpositionen gehören jetzt einer Partitionsmenge an. Jeder dieser Affixpositionen $n \in$ `Cur` wird in der Datenstruktur `SOAG.PartNum[`n`]` die aktuelle Partitionsnummer zugeordnet. Alle eingehenden Kanten, die auf diese Affixpositionen verweisen, werden im Feld `Deg` durch Dekrementieren gelöscht. Gibt es für eine der Affixpositionen, die ursprünglich auf Affixpositionen aus `Cur` verwies, keine ausgehenden Kanten mehr, so wird diese in die Menge `Leave` aufgenommen.

Am Ende der REPEAT-Schleife ergibt sich `Cur` aus `Leave`, `Leave` wird geleert und der Sortiervorgang mit der Berechnung der nächsten Partitionsmenge fortgesetzt, bis `Cur` leer ist.

Die Zugehörigkeit einer Affixposition zu einer Partitionsmenge wird durch die Partitionsnummer im Feld `SOAG.PartNum` modelliert. Nach Abschluß des topologischen Sortiervorgangs besitzen alle Affixpositionen je einer Partitionsmenge dieselbe Partitionsnummer.

5.7 Implementierungen

5.7.1 eSOAGPartition.Mod

```
MODULE eSOAGPartition; (* dk 2.63 14.03.98 *)

IMPORT EAG := eEAG, SOAG := eSOAG, Sets := eSets, ALists := eALists, IO := eIO,
      Protocol := eSOAGProtocol, ASets := eASets, BSets := eBSets;


CONST
  unor = -1;
  nil = 0;
  element = 1;

  (* phases *)
  computeDPandIDP* = 1;
  dynTopSort* = 2;

  firstVarBuf = 0;
  firstChangeBuf = 0;

TYPE
  VarBufDesc = RECORD
```

```
    AffOcc, Sym,
    Num, VarInd: INTEGER
  END;
  OpenVarBuf = POINTER TO ARRAY OF VarBufDesc;

  ChangeBufDesc = RECORD
    RuleInd,
    AffOccInd1,
    AffOccInd2: INTEGER;
  END;
  OpenChangeBuf = POINTER TO ARRAY OF ChangeBufDesc;

VAR
  VarBuf: OpenVarBuf;
  NextVarBuf: INTEGER;
  ChangeBuf: OpenChangeBuf;
  NextChangeBuf: INTEGER;
  DS: POINTER TO ARRAY OF ARRAY OF INTEGER;
  Deg: POINTER TO ARRAY OF INTEGER;
  Phase: INTEGER;
  CyclicTDP: BOOLEAN;
  OEAG: BOOLEAN;

VAR
  NUV, MarkedEdges, LastCur: ALists.AList;
  Cur, Leave: ASets.ASet;
  New: BSets.BSet;
  Seperator: INTEGER;


PROCEDURE Expand;
VAR
  VarBuf1: OpenVarBuf;
  ChangeBuf1: OpenChangeBuf;
  i: LONGINT;

    PROCEDURE NewLen(ArrayLen: LONGINT): LONGINT;
    BEGIN
      IF ArrayLen < MAX(INTEGER) DIV 2 THEN RETURN 2 * ArrayLen + 1 ELSE HALT(99) END
    END NewLen;

BEGIN
  IF NextVarBuf >= LEN(VarBuf^) THEN NEW( VarBuf1, NewLen(LEN(VarBuf^)));
    FOR i := firstVarBuf TO LEN(VarBuf^)-1 DO VarBuf1[i] := VarBuf[i] END;
    VarBuf := VarBuf1
  END;
  IF NextChangeBuf >= LEN(ChangeBuf^) THEN NEW( ChangeBuf1, NewLen(LEN(ChangeBuf^)));
    FOR i := firstChangeBuf TO LEN(ChangeBuf^)-1 DO ChangeBuf1[i] := ChangeBuf[i] END;
    ChangeBuf := ChangeBuf1
  END;
END Expand;


PROCEDURE Push( VAR Stack: ALists.AList; S, A1, A2: INTEGER );
BEGIN
  ALists.Append( Stack, S );
  ALists.Append( Stack, A1 );
  ALists.Append( Stack, A2 );
END Push;

PROCEDURE EdgeInTDP( R, A1, A2: INTEGER ): BOOLEAN;
(*IN: Regel, zwei Affixparameter
 OUT: boolscher Wert
 SEM: Test, ob eine Abhaengigkeit zwischen den beiden Affixparametern besteht *)
BEGIN
  RETURN Sets.In( SOAG.Rule[R].TDP[SOAG.AffOcc[A1].AffOccNum.InRule],
                  SOAG.AffOcc[A2].AffOccNum.InRule )
END EdgeInTDP;

PROCEDURE AddTDPChange( R, AO1, AO2: INTEGER );
(*IN: Regel, 2 Affixparameter
 OUT: -
 SEM: Fuegt ein Tripel in das Feld ChangeBuf ein - Speicherung einer in TDP eingefuegten
    Abhaengigkeit. *)
BEGIN
  ChangeBuf[NextChangeBuf].RuleInd := R;
  ChangeBuf[NextChangeBuf].AffOccInd1 := AO1;
  ChangeBuf[NextChangeBuf].AffOccInd2 := AO2;
  INC( NextChangeBuf ); IF NextChangeBuf >= LEN( ChangeBuf^ ) THEN Expand END;
END AddTDPChange;

PROCEDURE ResetTDPChanges( End: INTEGER );
```

```
(*IN: Index in ChangeBuf
 OUT: -
 SEM: Ruecksetzen,der in ChangeBuf gespeicherten Abhaengigkeiten in den TDP's *)
VAR i: INTEGER;
BEGIN
  FOR i := firstChangeBuf TO End DO
    Sets.Excl( SOAG.Rule[ChangeBuf[i].RuleInd].TDP[
            SOAG.AffOcc[ChangeBuf[i].AffOccInd1].AffOccNum.InRule],
            SOAG.AffOcc[ChangeBuf[i].AffOccInd2].AffOccNum.InRule );
  END
END ResetTDPChanges;

PROCEDURE AddTDPTrans( R, AO1, AO2: INTEGER );
(*IN:  Regel; zwei Affixparameter
 OUT:  -
 SEM:  fuegt die Kante (AO1,AO2) in den TDP ein und bildet den transitiven
      Abschlußss TDP+; die eingefuegte Abhaengigkeit lautet: AO2 haengt ab von AO1,
      AO1->AO2 im Sinne des Datenflusses; markiert alle neu eingefuegten Kanten,
      indem sie auf einen Stack gelegt werden
  SEF:  NUV: AList ist global
      MarkedEdges falls Phase = computeDPandIDP
      ChangeBuf, CyclicTDP falls Phase = dynTopSort *)
VAR
  AO3, AO4, AN1, AN2, AN3, AN4, i: INTEGER;
BEGIN
  ALists.Reset( NUV );
  IF ~ EdgeInTDP( R, AO1, AO2 ) THEN
    AN1 := SOAG.AffOcc[AO1].AffOccNum.InRule;
    AN2 := SOAG.AffOcc[AO2].AffOccNum.InRule;
    FOR AO4 := SOAG.Rule[R].AffOcc.Beg TO SOAG.Rule[R].AffOcc.End DO
      AN4 := SOAG.AffOcc[AO4].AffOccNum.InRule;
      IF (( AN4 = AN2 ) OR
        Sets.In( SOAG.Rule[R].TDP[AN2], AN4 )) &
        ~ Sets.In( SOAG.Rule[R].TDP[AN1], AN4 )
      THEN
        ALists.Append( NUV, AO4 )
      END
    END;
    FOR AO3 := SOAG.Rule[R].AffOcc.Beg TO SOAG.Rule[R].AffOcc.End DO
      AN3 := SOAG.AffOcc[AO3].AffOccNum.InRule;
      IF (( AN3 = AN1 ) OR
        Sets.In( SOAG.Rule[R].TDP[AN3], AN1 )) &
        ~ Sets.In( SOAG.Rule[R].TDP[AN3], AN2 )
      THEN
        FOR i := ALists.firstIndex TO NUV.Last  DO
          AO4 := NUV.Elem[i];
          IF ~ EdgeInTDP(R, AO3, AO4 ) THEN
            Sets.Incl( SOAG.Rule[R].TDP[AN3], SOAG.AffOcc[AO4].AffOccNum.InRule );
            IF (SOAG.AffOcc[AO3].SymOccInd = SOAG.AffOcc[AO4].SymOccInd) &
               ~SOAG.IsPredNont( SOAG.AffOcc[AO3].SymOccInd ) THEN
              Push( MarkedEdges,
                  SOAG.SymOcc[SOAG.AffOcc[AO3].SymOccInd].SymInd,
                  SOAG.AffOcc[AO3].AffOccNum.InSym,
                  SOAG.AffOcc[AO4].AffOccNum.InSym )
            END;
            IF Phase = dynTopSort THEN
              AddTDPChange( R, AO3, AO4 )
            END
          END
        END
      END;
      IF Sets.In( SOAG.Rule[R].TDP[AN3], AN3 ) THEN
        IF Phase = computeDPandIDP THEN
          IO.WriteText( Protocol.Out, '...a cyclic affix dependence was found!\n' );
          Protocol.WriteRule( R ); IO.WriteLn( Protocol.Out );
          Protocol.WriteAffOcc( AO3 ); IO.WriteLn( Protocol.Out );
          IO.WriteString( Protocol.Out, 'Preorder  of ' );
          Protocol.WriteAffOcc( AO1 ); IO.WriteLn( Protocol.Out );
          IO.WriteString( Protocol.Out, 'Postorder of ' );
          Protocol.WriteAffOcc( AO2 ); IO.WriteLn( Protocol.Out );
          IO.WriteLn( Protocol.Out );
          IO.Update( Protocol.Out );
          SOAG.Error( SOAG.cyclicTDP, 'eSOAGVSGen.AddTDPTrans' );
        ELSIF Phase = dynTopSort THEN CyclicTDP := TRUE END
      END
    END
  END
END AddTDPTrans;


PROCEDURE SetAffOccforVars( AO, Affixform: INTEGER; isDef: BOOLEAN );
(*  IN:  Affixparameter, Affixform, Affixparameter definierend oder applizierend ?
```

```
  OUT:  -
  SEM:  ordnet im Feld VarBuf[] jeder Variablen, die im Baum der Affixform gefunden
        wird, den zugehoerigen Affixparameter, sowie das Variablensymbol zu
  SEF:  VarBuf[], NextVarBuf *)
VAR MA: INTEGER;
BEGIN
  IF Affixform < 0 THEN  (* Variable *)
    IF NextVarBuf+1 >= LEN( VarBuf^ ) THEN Expand END;
    VarBuf[NextVarBuf].AffOcc := AO;
    IF isDef THEN  VarBuf[NextVarBuf].Sym := -EAG.Var[-Affixform].Sym
    ELSE VarBuf[NextVarBuf].Sym := EAG.Var[-Affixform].Sym END;
    VarBuf[NextVarBuf].Num := EAG.Var[-Affixform].Num;
    VarBuf[NextVarBuf].VarInd := -Affixform;
    INC( NextVarBuf );
  ELSIF EAG.MAlt[EAG.NodeBuf[Affixform]].Arity > 0 THEN (* nichtleere Regel *)
    FOR MA := 1 TO EAG.MAlt[EAG.NodeBuf[Affixform]].Arity DO
      SetAffOccforVars( AO, EAG.NodeBuf[Affixform+MA], isDef )
    END
  END
END SetAffOccforVars;

PROCEDURE ComputeDefAffOcc(R: INTEGER);
(*IN:  Regel
 OUT:  -
 SEM:  berechnet fuer alle Affixvariablen einer Regel den Affixparameter des zugehoerigen
       definierenden Affixes und speichert  diesen in DefAffOcc[]
 SEF:  Zugriffe auf VarBuf[] *)
VAR Scope: EAG.ScopeDesc; EAGRule: EAG.Rule;
  V, i: INTEGER; Found: BOOLEAN;
BEGIN
  IF SOAG.Rule[R] IS SOAG.OrdRule THEN
      Scope := SOAG.Rule[R](SOAG.OrdRule).Alt.Scope
  ELSE EAGRule := SOAG.Rule[R](SOAG.EmptyRule).Rule;
    IF EAGRule IS EAG.Opt THEN Scope := EAGRule(EAG.Opt).Scope
    ELSIF EAGRule IS EAG.Rep THEN Scope := EAGRule(EAG.Rep).Scope
    END
  END;
  FOR V := Scope.Beg TO Scope.End-1 DO
    i := firstVarBuf-1;
    REPEAT
      INC(i);
      Found := (( EAG.Var[V].Sym = -VarBuf[i].Sym ) &  (* nur definierende Affixparam. *)
             ( EAG.Var[V].Num = VarBuf[i].Num))
    UNTIL Found OR (i >= NextVarBuf-1);
    IF Found THEN
      SOAG.DefAffOcc[V] := VarBuf[i].AffOcc;
    ELSE
      IO.WritePos(IO.Msg, EAG.Var[V].Pos);
      SOAG.Error( SOAG.notLeftDefined, 'eSOAGPartition.ComputeDefAffOcc' )
    END
  END
END ComputeDefAffOcc;


PROCEDURE ComputeAffixApplCnt(R: INTEGER);
(*IN:  Regel
 OUT:  -
 SEM:  Berechnet in AffixApplCnt die Anzahl der Applikationen eines Affixes, ausserdem wird
       fuer jeden Vergleich eine Abhaengigkeit in den DP aufgenommen
 PRE:  DefAffOcc[] muss berechnet sein *)
VAR Scope: EAG.ScopeDesc; EAGRule: EAG.Rule;
  A, AN, DAN, i: INTEGER;
BEGIN
  IF SOAG.Rule[R] IS SOAG.OrdRule THEN
      Scope := SOAG.Rule[R](SOAG.OrdRule).Alt.Scope
  ELSE EAGRule := SOAG.Rule[R](SOAG.EmptyRule).Rule;
    IF EAGRule IS EAG.Opt THEN Scope := EAGRule(EAG.Opt).Scope
    ELSIF EAGRule IS EAG.Rep THEN Scope := EAGRule(EAG.Rep).Scope
    END
  END;
  FOR A := Scope.Beg TO Scope.End-1 DO
    i := firstVarBuf;
    WHILE i < NextVarBuf DO
      IF ( EAG.Var[A].Sym = -VarBuf[i].Sym ) &  (* nur definierende Affixparam. *)
        ((( EAG.Var[A].Num = VarBuf[i].Num) &  (* Vergleiche *)
        ( SOAG.DefAffOcc[A] # VarBuf[i].AffOcc))  OR
        (( EAG.Var[A].Num = -VarBuf[i].Num) &  (* neg. Vergleiche *)
         ( SOAG.DefAffOcc[VarBuf[i].VarInd] = VarBuf[i].AffOcc)))
      THEN
        AN := VarBuf[i].AffOcc - SOAG.Rule[R].AffOcc.Beg;
        DAN := SOAG.DefAffOcc[A] - SOAG.Rule[R].AffOcc.Beg;
        Sets.Incl(SOAG.Rule[R].DP[DAN], AN );
```

```
        INC(SOAG.AffixApplCnt[A])
      ELSIF ( EAG.Var[A].Sym = VarBuf[i].Sym ) &  (* appl.  Affixparam. *)
          ( EAG.Var[A].Num = VarBuf[i].Num ) THEN (* Applikation *)
        INC(SOAG.AffixApplCnt[A])
      END;
      INC(i)
    END
  END
END ComputeAffixApplCnt;


PROCEDURE ComputeDP*;
(* SEM:  Initialisierung des Abhaengigkeitsgraphen fuer jeden Affixparamter aller
        Regeln aus der Spezifikationsdatenstruktur
  SEF:  auf alle globalen DSen *)
VAR
  R, AO, SO, i, j, PBI, FirstSOVar: INTEGER;
  Alt: EAG.Alt;
BEGIN
  Phase := computeDPandIDP;
  ALists.New( MarkedEdges, 256 );
  ALists.New( NUV, 56 );
  FOR R := SOAG.firstRule TO SOAG.NextRule-1 DO
    IF SOAG.IsEvaluatorRule( R ) THEN
      FOR SO := SOAG.Rule[R].SymOcc.Beg TO SOAG.Rule[R].SymOcc.End DO
        FirstSOVar := NextVarBuf;
        FOR AO := SOAG.SymOcc[SO].AffOcc.Beg TO SOAG.SymOcc[SO].AffOcc.End DO
          PBI := SOAG.AffOcc[AO].ParamBufInd;
          SetAffOccforVars( AO, EAG.ParamBuf[PBI].Affixform, EAG.ParamBuf[PBI].isDef )
        END;
        IF SOAG.IsPredNont( SO ) THEN
          FOR i := FirstSOVar TO NextVarBuf-1 DO
            FOR j := FirstSOVar TO NextVarBuf-1 DO
              IF (VarBuf[j].Sym < 0) &      (* definierend *)
                 (VarBuf[i].Sym > 0) THEN   (* applizierend *)
                AddTDPTrans( R, VarBuf[i].AffOcc, VarBuf[j].AffOcc );
                Sets.Incl(SOAG.Rule[R].DP[
                        SOAG.AffOcc[VarBuf[i].AffOcc].AffOccNum.InRule],
                        SOAG.AffOcc[VarBuf[j].AffOcc].AffOccNum.InRule );
              END
            END
          END
        END
      END;

      ComputeDefAffOcc(R);
      ComputeAffixApplCnt(R);

      IF SOAG.Rule[R].AffOcc.End >= SOAG.Rule[R].AffOcc.Beg THEN (* Parameter vorhanden *)
        FOR i := firstVarBuf TO NextVarBuf-1 DO
          IF VarBuf[i].Sym > 0  THEN (*applizierendes Vorkommen *)
            AddTDPTrans( R, SOAG.DefAffOcc[VarBuf[i].VarInd], VarBuf[i].AffOcc );
            Sets.Incl(SOAG.Rule[R].DP[
                SOAG.AffOcc[SOAG.DefAffOcc[VarBuf[i].VarInd]].AffOccNum.InRule],
                SOAG.AffOcc[VarBuf[i].AffOcc].AffOccNum.InRule );
          END
        END;
        NextVarBuf := firstVarBuf  (* Leeren von VarBuf[] *)
      END
    END
  END
END ComputeDP;

PROCEDURE ComputeInducedTDP;
(* SEM:  bildet in TDP alle induzierten Abhaengigkeiten solange MarkedEdges nicht leer ist
        und die Ausgabeinvariante TDP = ind(TDP) gilt.
  SEF:  MarkedEdges, TDP *)
VAR
  SO, AN1, AN2, AO1, AO2: INTEGER;
BEGIN
  WHILE MarkedEdges.Last >= ALists.firstIndex DO
    AN2 := MarkedEdges.Elem[MarkedEdges.Last]; ALists.Delete( MarkedEdges, MarkedEdges.Last );
    AN1 := MarkedEdges.Elem[MarkedEdges.Last]; ALists.Delete( MarkedEdges, MarkedEdges.Last );
    SO := MarkedEdges.Elem[MarkedEdges.Last]; ALists.Delete( MarkedEdges, MarkedEdges.Last );
    SO := SOAG.Sym[SO].FirstOcc;
    WHILE  SO # SOAG.nil DO
      AO1 := SOAG.SymOcc[SO].AffOcc.Beg + AN1;
      AO2 := SOAG.SymOcc[SO].AffOcc.Beg + AN2;
      AddTDPTrans( SOAG.SymOcc[SO].RuleInd, AO1, AO2 );
      SO := SOAG.SymOcc[SO].Next
    END
  END
```

```
END ComputeInducedTDP;

PROCEDURE ComputeIDPTrans*;
(* SEM:  bildet in TDP alle induzierten Abhaengigkeiten solange MarkedEdges nicht leer ist
        und die Ausgabeinvariante TDP = ind(TDP) gilt, damit ist dann TDP = IDP+
  SEF:  - *)
BEGIN
  ComputeInducedTDP
END ComputeIDPTrans;

PROCEDURE Orient( a, b, X: INTEGER; VAR New: BSets.BSet );
(*IN:  Affixpositionsnummern a und b, Symbol
OUT:  Liste von Paaren von Affixpositionsnummern des Symbols X
SEM:  findet fuer zwei Affixpositionen eines Symbols eine Orientierung, fuegt diese in alle
      Regelabhaengigkeitsgraphen ein und liefert die Liste aller bei der transitiven
      Vervollstaendigung neu entstandenen Abhaengigkeiten zurueck
SEF:   auf ChangeBuf[] *)

VAR SO, i, a1, b1, AO1, AO2: INTEGER;
BEGIN
  BSets.Reset( New ); CyclicTDP := FALSE; NextChangeBuf := firstChangeBuf;
  SO := SOAG.Sym[X].FirstOcc;
  AddTDPTrans( SOAG.SymOcc[SO].RuleInd,
            SOAG.SymOcc[SO].AffOcc.Beg + b,
            SOAG.SymOcc[SO].AffOcc.Beg + a);
  ComputeInducedTDP;
  IF CyclicTDP THEN CyclicTDP := FALSE; OEAG := FALSE;
    ResetTDPChanges( NextChangeBuf-1 );
    NextChangeBuf := firstChangeBuf; (* reset ChangeBuf *)
    AddTDPTrans( SOAG.SymOcc[SO].RuleInd,
              SOAG.SymOcc[SO].AffOcc.Beg + a,
              SOAG.SymOcc[SO].AffOcc.Beg + b);
    ComputeInducedTDP
  END;
  IF CyclicTDP THEN
    IO.WriteText(IO.Msg, "\tGarmmar is not SOAG\n");
    SOAG.Error( SOAG.cyclicTDP, 'eSOAGVSGen.Orient' )
  END;
  FOR i := firstChangeBuf TO NextChangeBuf - 1 DO
    AO1 := ChangeBuf[i].AffOccInd1;
    AO2 := ChangeBuf[i].AffOccInd2;
    IF (SOAG.AffOcc[AO1].SymOccInd = SOAG.AffOcc[AO2].SymOccInd) &
      (SOAG.SymOcc[SOAG.AffOcc[AO1].SymOccInd].SymInd = X)
    THEN
      a1 := SOAG.AffOcc[AO1].AffOccNum.InSym; (* a *)
      b1 := SOAG.AffOcc[AO2].AffOccNum.InSym; (* b *)
      IF SOAG.IsOrientable( X, a1, b1 ) THEN
        BSets.Insert( New, a1 * Seperator + b1);
      END
    END
  END
END Orient;

PROCEDURE WriteDS( XmaxAff: INTEGER);
VAR i,j: INTEGER;
BEGIN
  FOR i := 0 TO XmaxAff DO
    IO.WriteText(IO.Msg, 'Zeile '); IO.WriteInt(IO.Msg, i);
    FOR j := 0 TO XmaxAff DO
      IO.WriteInt( IO.Msg, DS[i][j] ); IO.WriteString( IO.Msg, ' ' )
    END; IO.WriteLn( IO.Msg )
  END; IO.Update( IO.Msg );
END WriteDS;

PROCEDURE DynTopSortSym( X: INTEGER );
(*IN: Symbol
 OUT: -
 SEM: dynamisches topologisches Sortieren der Affxipositionsabhaengigkeiten unter
    Heranfuehrung an eine erfolgreiche bzw. unmittelbar erfolgreiche Orientierung
  SEF: DS[][] *)
VAR
  XmaxAff,AO1, AO2, SO,
  Part, i, a1, a, b, c, d: INTEGER;
  tmp: ASets.ASet;
BEGIN
  Part := 0;
  XmaxAff := SOAG.SymOcc[SOAG.Sym[X].FirstOcc].AffOcc.End -
             SOAG.SymOcc[SOAG.Sym[X].FirstOcc].AffOcc.Beg;
  FOR a := 0 TO XmaxAff DO
    FOR b := 0 TO XmaxAff DO
      DS[a][b] := nil
    END
```

```
  END;
  (* Initialisierung DS(Sym) *)
  SO := SOAG.Sym[X].FirstOcc;
  WHILE SO # SOAG.nil DO
    FOR AO1 := SOAG.SymOcc[SO].AffOcc.Beg TO SOAG.SymOcc[SO].AffOcc.End DO
      FOR AO2 := SOAG.SymOcc[SO].AffOcc.Beg TO SOAG.SymOcc[SO].AffOcc.End DO
        IF EdgeInTDP( SOAG.SymOcc[SO].RuleInd, AO1, AO2 ) THEN
          a := SOAG.AffOcc[AO1].AffOccNum.InSym;
          b := SOAG.AffOcc[AO2].AffOccNum.InSym;
          IF SOAG.IsOrientable( X, a, b ) THEN DS[a][b] := element END
        END
      END
    END;
    SO := SOAG.SymOcc[SO].Next
  END;

  (* Initialisierung von Deg() und unor in DS *)
  FOR a := 0 TO XmaxAff DO
    Deg[a] := 0;
    FOR b := 0 TO XmaxAff DO
      IF DS[a][b] = element THEN
        INC( Deg[a] )
      ELSIF (DS[b][a] = nil) & SOAG.IsOrientable( X, a, b ) THEN
        DS[a][b] := unor; DS[b][a] := unor
      END
    END
  END;

  (* Initialisieunrg von Cur,Leave *)
  ASets.Reset( Cur );
  ASets.Reset( Leave );
  FOR a :=  0 TO XmaxAff DO
    IF Deg[a] = 0 THEN
      IF SOAG.IsSynthesized( X, a ) THEN
        ASets.Insert( Cur, a )
      ELSIF SOAG.IsInherited( X, a ) THEN
        ASets.Insert( Leave, a )
      END
    END
  END;

  REPEAT

    (* Auslagerung der Elem.-liste, da das Loeschen aus Cur Listenstruktur veraendert *)
    ALists.Reset( LastCur );
    FOR a := ASets.firstIndex TO Cur.List.Last DO ALists.Append( LastCur, Cur.List.Elem[a] )
END;

    FOR b := 0 TO XmaxAff DO
      FOR a1 := ALists.firstIndex TO LastCur.Last DO
        a := LastCur.Elem[a1];
        IF ASets.In(Cur, a) & (DS[a][b] = unor) THEN

          Orient( a, b, X, New );

          FOR i := BSets.firstIndex TO New.List.Last DO
            c := New.List.Elem[i] DIV Seperator;
            d := New.List.Elem[i] MOD Seperator;
            DS[c][d] := element; INC( Deg[c] );
            IF DS[d][c] = unor THEN DS[d][c] := nil END;

            IF ASets.In( Cur, c ) THEN  ASets.Delete( Cur, c )
            ELSIF Deg[c] = 1 THEN ASets.Delete( Leave, c ) END
          END
        END
      END
    END;

    INC( Part );

    FOR a := ASets.firstIndex TO Cur.List.Last DO
      SOAG.PartNum[SOAG.Sym[X].AffPos.Beg+Cur.List.Elem[a]] := Part;
      FOR b := 0 TO XmaxAff DO
        IF DS[b][Cur.List.Elem[a]] = element THEN
          DEC( Deg[b] );
          IF Deg[b] = 0 THEN ASets.Insert( Leave, b ) END
        END
      END
    END;

    tmp := Cur;
    Cur := Leave;
```

```
    Leave := tmp;
    ASets.Reset( Leave )

  UNTIL  ASets.IsEmpty( Cur );

  IF SOAG.Sym[X].MaxPart < Part THEN SOAG.Sym[X].MaxPart := Part END;
  IF SOAG.MaxPart < Part THEN SOAG.MaxPart := Part END;
END DynTopSortSym;

PROCEDURE DynTopSort*;
(* SEM: dynamisches topologisches Sortieren aller Symbole der Grammatik *)
VAR S: INTEGER;
BEGIN
  ASets.New( Cur, SOAG.MaxAffNumInSym + 1 );
  ASets.New( Leave, SOAG.MaxAffNumInSym + 1 );
  NEW( Deg, SOAG.MaxAffNumInSym + 1 );
  NEW( DS, SOAG.MaxAffNumInSym + 1, SOAG.MaxAffNumInSym + 1 );
  BSets.New( New, (SOAG.MaxAffNumInSym + 1) * (SOAG.MaxAffNumInSym + 1) );
  Seperator := SOAG.MaxAffNumInSym + 1;
  ALists.New( LastCur, 16 );

  FOR S := EAG.firstHNont TO EAG.NextHNont-1 DO
    IF ~Sets.In( EAG.Pred, S ) & Sets.In( EAG.All, S )THEN
      DynTopSortSym( S );
    END
  END;
END DynTopSort;


PROCEDURE Compute*;
(* SEM: Treiber *)
VAR i: INTEGER;
BEGIN
  SOAG.Init;
  NEW( VarBuf, 50 );
  NEW( ChangeBuf, 64  );
  NextVarBuf := firstVarBuf;
  NextChangeBuf := firstChangeBuf;
  OEAG := TRUE;
  Phase := computeDPandIDP;
  ComputeDP;
  ComputeIDPTrans;
  Phase := dynTopSort;
  DynTopSort;
  IF OEAG THEN IO.WriteText(IO.Msg, "\n\tGrammar is SOEAG\n")
  ELSE IO.WriteText(IO.Msg, "\n\tGrammar is SOEAG (backtracked)\n") END;
END Compute;


END eSOAGPartition.
```

6 Berechnung der Visit-Sequenzen

In diesem Kapitel wird beschrieben, wie zu einer SOAG die Visit-Sequenzen für ihre Evaluatorregeln konstruiert werden.

6.1 Visit-Sequenzen

Für jeden Knoten *k* eines Ableitungsbaumes, an dem die Regel *r* angewendet wurde, stellt die Visit-Sequenz VS_r eine lokale Traversierungvorschrift dar. Sie beschreibt, in welcher Reihenfolge die Söhne des Knotens *k* besucht werden müssen und wann zum Vaterknoten aufgestiegen werden muß. Implizit ist in den Visit-Sequenzen die Reihenfolge der Auswertung der Affixparameter enthalten, denn vor jedem Besuch eines Knotens müssen die während des Besuchs benötigten Instanzen von applizierenden Affixparametern synthetisiert werden. Nach dem Besuch müssen neu berechnete Instanzen von definierenden Affixparametern analysiert werden. Alle Instanzen definierender Affixparameter eines Knotens *k* werden im oberen Kontext von *k* berechnet, die seiner Söhne im unteren Kontext des jeweiligen Sohnes. Aus Sicht des Knotens *k* muß zur Berechnung dieser Instanzen ein Aufstieg zum Vaterknoten bzw. ein Besuch des Sohnes erfolgen. Die Instanzen definierender Affixparameter von Prädikaten werden durch den Aufruf des Prädikates berechnet. Deshalb ergeben sich als Inhalt der Visit-Sequenzen drei Typen von *Instruktionen*:

1. $VISIT(X_i^r, n)$ zeigt den *n*-ten Besuch des Symbolvorkommens X_i in der Regel *r* an,
2. $LEAVE(n)$ zeigt den *n*-ten Aufstieg zum Vaterknoten an,
3. $CALL(X)$ zeigt den Aufruf eines Prädikates *X* an.

Zur Abbildung von Affixparametern auf Instruktionen wird die Funktion *MAP_VS* definiert.

Definition 6-1 (*MAP_VS*):

$$MAP\_VS(a^{(r,i)}) := \begin{cases} VISIT(X_i^r, n) & \text{wenn } a^{(r,i)} \in AP_D(r),\ i>0 \text{ und } a^{(r,i)} \in A_n(X) \text{ mit } X=X_i^r \text{ und } X \text{ ist ein Grundnichtterminal; oder} \\ LEAVE(n) & \text{wenn } a^{(r,i)} \in AP_D(r),\ i=0 \text{ und } a^{(r,i)} \in A_n(X) \text{ mit } X=X_i^r \text{ und } X \text{ ist ein Grundnichtterminal; oder} \\ CALL(X) & \text{wenn } a^{(r,i)} \in AP_D(r),\ i>0 \text{ und mit } X=X_i^r \text{ ist } X \text{ ein Prädikatnichtterminal; oder} \\ NOP & \text{sonst (keine Instruktion).} \end{cases}$$

Die gefundene Partition $(A_1(X),..,A_{n(X)}(X))$ für die Affixpositionen eines Symbols *X* zeigt an, in welcher Reihenfolge die Affixpositionen berechnet werden müssen. Enthält eine Partitionsmenge $A_i(X)$ mehrere Affixpositionen, so ist deren Berechnungsreihenfolge beliebig wählbar. In den Abhängigkeitsgraphen der Regeln mit Symbolvorkommen von *X* sind zusätzliche Abhängigkeiten zwischen den Affixparametern des Symbolvorkommens *X* derart eingetragen, daß die Berechnungsreihenfolge dadurch manifestiert wird. Die Anwendung der Funktion *MAP_VS* auf die topologisch sortierten Affixparameter einer Regel liefert eine Liste von Instruktionen. Nach dem Anfügen einiger abschließender Instruktionen und der Eliminierung doppelter Instruktionen aus dieser Liste ergibt sich daraus die Visit-Sequenz einer Regel.

Die Visit-Sequenz einer Evaluatorregel *r* wird folgendermaßen konstruiert:
Sei $a_1^{(r)}...a_k^{(r)}$ eine topologische Sortierung des Graphen $(AP(r),IDP_m(r))$ mit $k=|AP(r)|$ zu einer Orientierungsreihenfolge $(C_1,...,C_m)$, dann entsteht die Visit-Sequenz VS_r aus der Folge

$$MAP\_VS(a_1^{(r)})...\ MAP\_VS(a_k^{(r)})\ INSTR(X_i^r)...INSTR(X^r_{\#S(r)})\ INSTR(X_0^r),$$

wobei doppelte Instruktionen gelöscht werden.

Definition 6-2 (*INSTR*):

$$INSTR(X_i^r) := \begin{cases} VISIT(X_i^r, n(X)), & \text{wenn } i>0 \text{ und } X_i^r=X \text{ ist ein Grundnichtterminal; oder} \\ LEAVE(n(X)), & \text{wenn } i=0 \text{ und } X_i^r=X \text{ ist ein Grundnichtterminal; oder} \\ CALL(X), & \text{wenn } i>0 \text{ und } X_i^r=X \text{ ist ein Prädikatnichtterminal; oder} \\ NOP, & \text{sonst.} \end{cases}$$

Die abschließenden Instruktionen stellen sicher, daß die Traversierung des Ableitungsbaumes vollständig ist und alle Affixparameter berechnet werden.

Jede Visit-Sequenz VS_r läßt sich in $n(X)$ Teile VS_r^1, ..., $VS_r^{n(X)}$ zerlegen, wobei $X=X_0^r$ gilt. Jedes VS_r^i endet jeweils mit $LEAVE(i)$ und wird *Plan* genannt.

6.2 Implementierungsdetails

Die Prozedur `ComputeVisitNo` berechnet für jede Affixposition aus ihrer Zugehörigkeit zu einer Partitionsmenge, deren Index in der Datenstruktur `SOAG.PartNum` gespeichert ist, ihre Visit-Nummer. Da die Partitionsmengen in umgekehrter Reihenfolge vorliegen, ergibt sich für die Berechnung der Visit-Nummer folgende Formel

```
(MaxPart + 1) DIV 2 - (PartNum + 1) DIV 2 + 1
```

`MaxPart` ist der maximale Index einer Partitionsmenge eines Symbols. Die berechnete Visit-Nummer wird wieder in der Datenstruktur `SOAG.PartNum` abgelegt, da deren Inhalt nicht mehr benötigt wird. Da Name und Inhalt der Datenstruktur nach der Berechnung der Visit-Nummern nicht mehr übereinstimmen, wurden die Funktionen `GetVisitNo` und `GetMaxVisitNo` als Schnittstellenfunktionen für den Zugriff auf `SOAG.PartNum` definiert. Die erste liefert die Visit-Nummer eines Affixparameters, die zweite die maximale Visit-Nummer eines Symbolvorkommens zurück. Beide Funktionen werden exportiert.

Die Funktionen `MapVS` und `CompleteTraversal` implementieren die im Abschnitt 6.1 „Visit-Sequenzen“ beschriebenen Funktionen *MAP_VS* bzw. *INSTR*. Beide Funktionen liefern Instruktionen vom Typ `Instruction` zurück, die im Modul `SOAG` definiert sind.

Die Funktion `TopSort` implementiert die topologische Sortierung aller Affixparameter einer Regel. Es wird der iterative Algorithmus von K.Mehlhorn benutzt, wie er in [Mehlhorn] beschrieben ist. Die beiden globalen Variablen

```
VAR
    InDeg: SOAG.OpenInteger;
    ZeroInDeg: Stacks.Stack;
```

dienen zur Speicherung von Zwischenergebnissen sowie zur Steuerung des topologischen Sortiervorgangs. `InDeg` enthält für alle Affixparameter der aktuellen Regel die Anzahl der eingehenden Kanten. `ZeroInDeg` enthält alle Affixparameter, die keine eingehenden Kanten besitzen. Nach der Initialisierung enthält `ZeroInDeg` alle Affixparameter, die als erstes ausgewertet werden müssen. Diese werde nacheinander aus der Menge `ZeroInDeg` entfernt und in die Visit-Sequenz unter Anwendung der Funktion `MapVS` übertragen. Für alle übertragenen Affixparameter werden die von diesen ausgehenden und in andere Affixparameter eingehenden Kanten anschaulich gelöscht, indem das Feld `InDeg` des entsprechenden Affixparameters dekrementiert wird. Hat ein solcher Affixparameter ebenfalls keine eingehenden Kanten mehr, ist also das Feld `InDeg` gleich Null, so wird dieser in die Menge `ZeroInDeg`, die als Stack implementiert ist, übertragen. Dies wird solange fortgesetzt, bis `ZeroInDeg` leer ist. Die berechneten Visit-Sequenzen werden für jede Regel nacheinander im Feld `VS` des Moduls `SOAG` eingetragen. Anfang und Ende der Visit-Sequenz einer Regel `R` sind in der Unterstruktur `VS` vom Typ `EAG.ScopeDesc` des Feldelementes `Rule[R]` im gleichen Modul enthalten.

Um eine Besuchsinstruktion wiederzufinden, wurden die Funktionen `GetVisit` und `GetNextVisit` definiert. Sie geben jeweils die Nummer der übergebenen Besuchsinstruktion aus der Liste `SOAG.VS` einer Regel zurück und werden auch exportiert.

6.3 Hash-Tabelle zur Eliminierung doppelter Instruktionen

Zur Eliminierung doppelter Instruktionen in der Visit-Sequenz einer Evaluatorregel werden alle Instruktionen zusätzlich in eine Hash-Tabelle eingetragen. Vor dem Eintragen einer neuen Instruktion in die Visit-Sequenz wird geprüft, ob die Instruktion schon in der Hash-Tabelle enthalten ist. Wenn dies der Fall ist, wird die aktuelle Instruktion verworfen, doppelte Instruktionsvorkommen werden damit vermieden.

Zur Ermittlung der Zugriffsadressen in der Hash-Tabelle wurde das Verfahren der doppelten Streuung (Doppel-Hashing) implementiert. Dies umfaßt die Benutzung zweier Hash-Funktionen *f* und *g* und folgender Vorschrift für die Berechnung einer Ausweichadresse bei Kollision für ein Element *i* und eine Tabelle der Länge *m*:

$$a_j(i) = (f(i) - (j\text{-}1)*g(i)) \bmod m$$

Die Länge der Tabelle wird mit $m=2^k$ und $k=1+\lceil log_2 max\{|AP(r)|: r \text{ ist Evaluatorregel}\}\rceil$ immer auf ein vielfaches von 2 festgelegt und die Funktion *g* mit

$$g = (f(i) \; div \; 2) * 2 + 1$$

bildet immer auf ungerade Zahlen ab. Dies stellt bei beliebig gewähltem *f* sicher, daß die Liste der Ausweichadressen auch wirklich eine Permutation ist, also die Ausweichadressen sich nicht wiederholen.

Bei einer Tabellenauslastung von weniger als 50%, was bei oben genannter Tabellengröße stets der Fall ist, sind die durchschnittlich zu erwartenden Kollisionstiefen bei gleich verteiltem *f*(i) laut [COMA] S.90 kleiner als 1,5. Die implementierte Funktion *f* erzeugt keine absolut gleich verteilten Werte, ist aber für den erforderlichen Zweck ausreichend, so daß die durchschnittlich zu erwartenden Kollisionstiefen kaum größer als 1,5 sein dürften.

6.4 Implementierungen

6.4.1 eSOAGVisitSeq.Mod

```
MODULE eSOAGVisitSeq; (* dk 1.54 05.03.98 *)

IMPORT EAG := eEAG, SOAG := eSOAG, HashTab := eSOAGHash, Sets := eSets, Stacks := eStacks,
      IO := eIO, Protocol := eSOAGProtocol;

CONST
  noVisit* = -1;

VAR
  InDeg: SOAG.OpenInteger;
  ZeroInDeg: Stacks.Stack;


PROCEDURE ComputeVisitNo;
(*IN: -
 OUT: -
 SEM: Berechnung der maximalen Visits in Sym[X].MaxPart und der Visit-Nummer der
    Affixpositionen in PartNum[AffPos]. *)
VAR S, AP, MaxPart, PartNum: INTEGER;
BEGIN
  FOR S := SOAG.firstSym TO SOAG.NextSym-1 DO
    MaxPart := (SOAG.Sym[S].MaxPart+1) DIV 2;
    SOAG.Sym[S].MaxPart := MaxPart;
    FOR AP := SOAG.Sym[S].AffPos.Beg TO SOAG.Sym[S].AffPos.End DO
      PartNum := (SOAG.PartNum[AP]+1) DIV 2;
      SOAG.PartNum[AP] := MaxPart - PartNum + 1
    END
  END
END ComputeVisitNo;

PROCEDURE GetVisitNo*(AP: INTEGER): INTEGER;
(*IN: Affixparameter
 OUT: Visit-Nummer des Affixparameters
 SEM: Auslesen der Visit-Nummer des Affixparameters, Schnittstellenprozedur *)
VAR S: INTEGER;
BEGIN
  S := SOAG.SymOcc[SOAG.AffOcc[AP].SymOccInd].SymInd;
  RETURN SOAG.PartNum[SOAG.Sym[S].AffPos.Beg+SOAG.AffOcc[AP].AffOccNum.InSym]
END GetVisitNo;

PROCEDURE GetMaxVisitNo*(SO: INTEGER): INTEGER;
(*IN: Symbolvorkommen, die keine Praedikate sind
 OUT: die maximale Visit-Nummer dieses Symbols
 SEM: Auslesen der maximalen Visit-Nummer des Symbolvorkommens, Schnittstellenprozedur *)
BEGIN
  RETURN SOAG.Sym[SOAG.SymOcc[SO].SymInd].MaxPart
END GetMaxVisitNo;

PROCEDURE GetNextVisit*(V, R, SO, VN: INTEGER): INTEGER;
(*IN: aktueller Visit, Regel, aktuller Visit, Symbolvorkommen, Visitnummer
 OUT: Nummer des Eintrages in der Visitsequenz der Regel
 SEM: Durchsucht die Visitsequenz nach dem Visit des Symbolvorkommens mit entsprechender
    Besuchsnummer. *)
BEGIN
  WHILE V <= SOAG.Rule[R].VS.End DO
    IF SOAG.VS[V] IS SOAG.Visit THEN
      IF (SOAG.VS[V](SOAG.Visit).SymOcc = SO) &
        (SOAG.VS[V](SOAG.Visit).VisitNo = VN)  THEN RETURN V END
    ELSIF SOAG.VS[V] IS SOAG.Call THEN
      IF SOAG.VS[V](SOAG.Call).SymOcc = SO  THEN RETURN V END
    ELSIF SOAG.VS[V] IS SOAG.Leave THEN
      IF (SOAG.VS[V](SOAG.Leave).VisitNo = VN) &
        (SOAG.Rule[R].SymOcc.Beg = SO)    THEN RETURN V END
    END;
```

```
    INC(V)
  END;
  RETURN noVisit
END GetNextVisit;

PROCEDURE GetVisit*(R, SO, VN: INTEGER): INTEGER;
(*IN: Regel, Symbolvorkommen, Visitnummer
 OUT: Nummer des Eintrages in der Visitsequenz der Regel
 SEM: Durchsucht die Visitsequenz nach dem Visit des Symbolvorkommens mit entsprechender
    Besuchsnummer. *)
BEGIN
  RETURN GetNextVisit(SOAG.Rule[R].VS.Beg, R, SO, VN)
END GetVisit;


PROCEDURE MapVS( AO: INTEGER ): SOAG.Instruction;
(*IN: Affixparameter
 OUT: Instruktion oder NIL fuer NOP
 SEM: Erzeugung einer Instruktion in Abhaeengigkeit vom uebergebenen Affixparameter *)
VAR
  Visit: SOAG.Visit; Leave: SOAG.Leave; Call: SOAG.Call;
  SO, R: INTEGER;
BEGIN
  IF EAG.ParamBuf[SOAG.AffOcc[AO].ParamBufInd].isDef THEN
    SO := SOAG.AffOcc[AO].SymOccInd;
    IF SOAG.IsPredNont( SO ) THEN
      NEW( Call );
      Call.SymOcc := SO;
      RETURN Call
    ELSE
      R := SOAG.SymOcc[SO].RuleInd;
      IF  SOAG.Rule[R].SymOcc.Beg = SO THEN (* linke Regelseite *)
        IF (GetVisitNo(AO)-1) > 0 THEN
          NEW( Leave );
          Leave.VisitNo := GetVisitNo(AO)-1;
          RETURN Leave
        ELSE  RETURN NIL END
      ELSE             (* rechte Regelseite *)
        NEW( Visit );
        Visit.SymOcc := SO;
        Visit.VisitNo := GetVisitNo(AO);
        RETURN Visit
      END
    END
  ELSE (* NOP *)
    RETURN NIL
  END
END MapVS;

PROCEDURE CompleteTraversal( SO: INTEGER ): SOAG.Instruction;
(*IN: Symbolvorkommen
 OUT: Instruktion
 SEM: Berechnung der abschliessenden Instruktionen fuer ein Symbolvorkommen *)
VAR
  Visit: SOAG.Visit; Leave: SOAG.Leave; Call: SOAG.Call;
  R, MaxVisitNo: INTEGER;
BEGIN
  IF SOAG.IsPredNont( SO ) THEN
      NEW( Call );
      Call.SymOcc := SO;
      RETURN Call
  ELSE
    R := SOAG.SymOcc[SO].RuleInd;
    IF  SOAG.Rule[R].SymOcc.Beg = SO THEN (* linke Regelseite *)
      NEW( Leave );
      Leave.VisitNo := GetMaxVisitNo(SO);
      RETURN Leave
    ELSE             (* rechte Regelseite *)
      NEW( Visit );
      Visit.SymOcc := SO;
      Visit.VisitNo := GetMaxVisitNo(SO);
      RETURN Visit
    END
  END
END CompleteTraversal;


PROCEDURE TopSort( R: INTEGER );
(*IN: Regel
 OUT: -
 SEM: topologische Sortierung der Affixparameter anhand ihrer Abhaengigkeiten *)
VAR
```

```
  AO, AN, BO, BN: INTEGER;
  Instr: SOAG.Instruction;
BEGIN
  Stacks.Reset( ZeroInDeg );
  FOR BO:= SOAG.Rule[R].AffOcc.End TO SOAG.Rule[R].AffOcc.Beg BY -1 DO
    BN := SOAG.AffOcc[BO].AffOccNum.InRule;
    InDeg[BN] := 0;
    FOR AO:= SOAG.Rule[R].AffOcc.Beg TO SOAG.Rule[R].AffOcc.End DO
      AN := SOAG.AffOcc[AO].AffOccNum.InRule;
      IF Sets.In( SOAG.Rule[R].TDP[AN], BN ) THEN
        INC( InDeg[BN] )
      END
    END;
    IF InDeg[BN] = 0 THEN Stacks.Push( ZeroInDeg, BO ) END
  END;
  WHILE ~Stacks.IsEmpty( ZeroInDeg ) DO
    Stacks.Pop( ZeroInDeg, AO );
    Instr := MapVS( AO );
    AN := SOAG.AffOcc[AO].AffOccNum.InRule;
    IF ~HashTab.IsIn( Instr ) THEN
      HashTab.Enter( Instr );
      SOAG.AppVS( Instr );
    END;
    FOR BO:= SOAG.Rule[R].AffOcc.End TO SOAG.Rule[R].AffOcc.Beg BY -1 DO
      BN := SOAG.AffOcc[BO].AffOccNum.InRule;
      IF Sets.In( SOAG.Rule[R].TDP[AN], BN ) THEN
        DEC( InDeg[BN]);
        IF InDeg[BN] = 0 THEN Stacks.Push( ZeroInDeg, BO ) END;
      END
    END
  END
END TopSort;


PROCEDURE Generate*;
(* SEM: Generierung der Visit-Sequenzen *)
VAR R, SO, MaxTry, S, V: INTEGER;
  Instr: SOAG.Instruction;
BEGIN
  MaxTry := 0;
  ComputeVisitNo;
  HashTab.Init(SOAG.MaxAffNumInRule);
  Stacks.New(ZeroInDeg, 32);
  NEW(InDeg, SOAG.MaxAffNumInRule + 1);

  FOR R := SOAG.firstRule TO SOAG.NextRule - 1 DO
    SOAG.Rule[R].VS.Beg := SOAG.NextVS;
    IF SOAG.IsEvaluatorRule(R) THEN
      HashTab.Reset;
      TopSort( R );
      IF MaxTry < HashTab.MaxTry THEN MaxTry := HashTab.MaxTry END;
      FOR SO := SOAG.Rule[R].SymOcc.Beg+1 TO SOAG.Rule[R].SymOcc.End DO
        Instr := CompleteTraversal( SO );
        IF ~HashTab.IsIn( Instr ) THEN
          SOAG.AppVS( Instr );
        END
      END;
      Instr := CompleteTraversal( SOAG.Rule[R].SymOcc.Beg );
      IF ~HashTab.IsIn( Instr ) THEN SOAG.AppVS( Instr ) END
    END;
    SOAG.Rule[R].VS.End := SOAG.NextVS-1;
  END
END Generate;

END eSOAGVisitSeq.
```

6.4.2 eSOAGHash.Mod

```
MODULE eSOAGHash;  (* dk 1.1 12.11.97 *)
(* Implementierung einer Hash-Tabelle (nur eine Instanz!) nach dem Doppel-Hash-Verfahren
   mit |H| = POWER(2,k) f: Instruction -> H und g: Instruction -> N
   und g(i) =  "Abbildung in die ungeraden Zahlen" *)

IMPORT SOAG := eSOAG, Math, IO := eIO;

CONST empty = 0;

TYPE
  HashEntry = POINTER TO RECORD
```

```
    Instr: SOAG.Instruction
  END;

  OpenHashTab = POINTER TO ARRAY OF HashEntry;

VAR
  HashTab: OpenHashTab;
  MaxHashTabIndex: INTEGER;
  MaxTry*: INTEGER;
  V4711, V711: INTEGER;

PROCEDURE Wr( S: ARRAY OF CHAR; I: INTEGER );
BEGIN
  IO.WriteString( IO.Msg, S ); IO.WriteInt( IO.Msg, I ); IO.WriteLn( IO.Msg ); IO.Update(
IO.Msg)
END Wr;


PROCEDURE Reset*;
(* SEM: leert die Hash-Tabelle *)
VAR i: INTEGER;
BEGIN
  MaxTry := 0;
  FOR i:= 0 TO MaxHashTabIndex-1 DO HashTab[i] := NIL END
END Reset;

PROCEDURE Init*( MaxAffInRule: INTEGER );
(*IN: maximale Anzahl an Affixparametern in einer Regel
 OUT: -
 SEM: reserviert Speicher fuer die Hash-Tabelle und setzt die max. Hash-Adresse*)
VAR Exp, i: INTEGER;
BEGIN
  Exp := SHORT(ENTIER( Math.ln( MaxAffInRule )/ Math.ln(2))+1);
  MaxHashTabIndex := 2;
  FOR i := 2 TO Exp DO MaxHashTabIndex := MaxHashTabIndex * 2 END;
  NEW( HashTab, MaxHashTabIndex );
  Reset
END Init;


PROCEDURE HashIndex( VAR I: SOAG.Instruction ): INTEGER;
(*IN: Instruktion aus der Visit-Sequenz, kein NIL !
  OUT: Index in der Hash-Tabelle
  SEM: Ermittlung Indexes in der Hash-Tabelle *)
VAR
  Index, Index0, Try: INTEGER;
  notfound: BOOLEAN;

  PROCEDURE HashFun( VAR I: SOAG.Instruction ): INTEGER;
  (* Fehler im Compiler: kann keine Integerkonstanten > 128 in Multiplikationen verarbeiten *)
  (*IN: Instruktion
   OUT: -
   SEM: Realisierung der Hash-Funktion *)
  VAR Index: INTEGER;
  BEGIN
    WITH
       I: SOAG.Visit DO
      Index := 100 + V4711 * I.SymOcc +  V711 * I.VisitNo
    | I: SOAG.Leave DO
      Index := 200 + V4711 * I.VisitNo
    | I: SOAG.Call DO
      Index := 300 + V4711 * I.SymOcc;
    ELSE (* I: NOP*)
      Index := 0;
    END;
    RETURN Index MOD MaxHashTabIndex;
  END HashFun;

BEGIN
  Try := 0;
  Index0 := HashFun( I );
  Index := Index0;
  IF HashTab[Index] = NIL THEN notfound := FALSE
  ELSE notfound := ~SOAG.isEqual( I, HashTab[Index].Instr ) END;
  WHILE notfound DO
    INC( Try );
    Index := (Index0 - Try * ( (Index0 DIV 2) * 2 + 1)) MOD MaxHashTabIndex;
    IF HashTab[Index] = NIL THEN notfound := FALSE
    ELSE notfound := ~SOAG.isEqual( I, HashTab[Index].Instr ) END;
  END;
  IF MaxTry < Try THEN MaxTry := Try END;
  RETURN Index
```

```
END HashIndex;


PROCEDURE IsIn*( I: SOAG.Instruction ): BOOLEAN;
(*IN: Instruktion aus der Visit-Sequenz
 OUT: boolscher Wert
 SEM: Test, ob die Instruktion schon in der Hash-Tabelle enthalten ist *)

VAR Index: INTEGER;
BEGIN
  IF I = NIL THEN RETURN TRUE
  ELSE
    Index := HashIndex( I );
    RETURN ~(HashTab[Index] = NIL)
  END
END IsIn;


PROCEDURE Enter*( I: SOAG.Instruction );
(*IN: Instruktion der Visit-Sequenz
  OUT: -
  SEM: fuegt die Instruktion in die Hash-Tabelle ein
*)
VAR Index: INTEGER;
Entry: HashEntry;
BEGIN
  IF I # NIL THEN
    Index := HashIndex( I );
    NEW( Entry );
    Entry.Instr := I;
    HashTab[Index] := Entry
  END
END Enter;

BEGIN
  V4711 := 4711;
  V711 := 711;
END eSOAGHash.
```

7 Optimierung der Affixvariablenspeicherung

Während der Evaluation eines dekorierten Ableitungsbaumes einer EAG werden die Instanzen aller Affixpositionen in Form ihrer Affixvariablen nacheinander berechnet und müssen im Speicher vorliegen, da sich die Synthesen anderer Affixpositionen auf sie beziehen können. Die Speicherung der Affixpositionsinstanzen beansprucht sehr viel Platz, und man ist deshalb darum bemüht, dafür eine Optimierung zu finden.

Im folgenden wird der Ansatz von Engelfriet und de Jong vorgestellt. In [EngJong] präsentieren sie einen polynomiellen Algorithmus, der für multi-visit AGen entscheidet, ob ein Attribut als Kellerspeicher implementiert werden kann. Die Übertragung dieses Ansatzes auf EAGen macht, wie schon beim SOAG-Verfahren, keine Probleme. Zusätzlich kann eine Variante des Algorithmus feststellen, ob in einem Kellerspeicher einer Affixposition nur ein Wert stehen kann und somit die Verwendung einer globalen Variable angezeigt ist.

Die in [EngJong] vorgestellten Optimierungsbedingungen beziehen sich auf Affixpositionen von Symbolen. Das im folgenden Kapitel noch vorzustellende Datenstrukturkonzept der generierten Compiler benutzt für Affixpositionen jedoch ohnehin globale Variablen (bzw. ein globales Feld von Variablen). Zentraler Gegenstand der semantischen Berechnungen sind dagegen die Affixvariablen jeder Regel. Wie jedoch leicht einzusehen ist, hat jede Affixvariable dieselben Optimierungseigenschaften wie die Affixposition, zu der der Affixparameter, in dem das Affix definiert wird, korrespondiert. Aus diesem Grund können die für Affixpositionen berechneten Eigenschaften zur Optimierung der Affixvariablenspeicherung eingesetzt werden.

Eine weitere Verfeinerung des hier vorgestellten Ansatzes bezüglich der Affixvariablen jeder Regel ist denkbar. Dabei könnte man davon ausgehen, daß die Affixvariablen einer Regel Optimierungseigenschaften besitzen, die von Affixvariablen anderer Regeln unabhängig sind, wohingegen im vorzustellenden Ansatz eine ungünstige Symbolkonstellation in einer Regel die Optimierungseigenschaften der Affixvariablen aller anderen Regeln mit gleichem Symbolvorkommen beeinflußt. Diese Verfeinerungsmöglichkeit wird im folgenden nicht weiter untersucht.

7.1 Optimierung durch Kellerspeicher

Eine Möglichkeit der Optimierung besteht darin, für Affixpositionen, die bestimmten Bedingungen genügen, Kellerspeicher zu benutzen, für jede Affixposition einen separaten Kellerspeicher. Eine Instanz einer solchen Affixposition würde nach ihrer Berechnung auf den Kellerspeicher gelegt werden und nach Berechnung der letzten von ihr abhängenden Instanz von dort wieder entfernt werden. Damit wird der Wert der Instanz nur für den kürzest-möglichen Zeitraum gespeichert.

Zunächst wird jeder Plan VS_r^i der Visit-Sequenz einer Regel r

$$VS_r^i = \text{VISIT}(X_1, j_1) \,..\, \text{VISIT}(X_k, j_k)\ \text{LEAVE}(i)$$

um die Mengen der vor und nach jedem Besuch (von CALL´s wird hier abstrahiert) zu berechnenden inherited-Affixpositionen $AI_l(X) = \{a_k^{(r,j)}: a_k^X \in A_l(X) \cap I(X) \text{ mit } X = X_j^r\}$ und synthesized-Affixpositionen $AS_l(X) = \{a_k^{(r,j)}: a_k^X \in A_l(X) \cap S(X) \text{ mit } X = X_j^r\}$erweitert:

$$EVS_r^i = AI_i(Y_0^r)\ AI_{j1}(X_1)\ \text{VISIT}(X_1, j1)\ AS_{j1}(X_1)\ ..\ AI_{jk}(X_k)\ \text{VISIT}(X_k, jk)\ AS_{jk}(X_k)\ AS_i(Y_0^r)\ \text{LEAVE}(i)$$

Die Pläne EVS_r^i werden zu einer *erweiterten Visit-Sequenz*

$$EVS_r = EVS_r^0\ EVS_r^{n(X)} \text{ mit } X = X_0^r$$

zusammengefaßt. Man beachte, daß jeder Affixparameter $a_k^{(r,j)}$ der Regel r Element genau einer Menge in EVS_r ist, die durch $set_r(a_k^{(r,j)})$ eindeutig beschrieben sei. Für zwei Elemente u und v der erweiterten Visit-Sequenz EVS_r wird nun eine Relation „<“ definiert, so daß $u<v$ gilt, wenn u vor v in der Sequenz EVS_r steht. Analog wird die Relation „≤“ definiert.

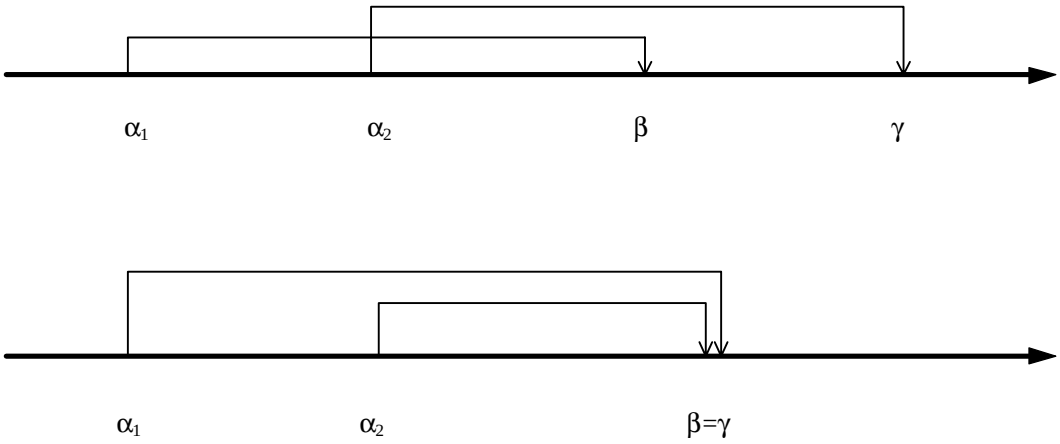


Abbildung 7-1: Ungeeignete Berechnungsreihenfolgen für Kellerspeicherung

Prinzipiell gibt es zwei Fälle von Auswertungsreihenfolge, die die Implementierung einer Affixpositionsinstanz als Kellerspeicher ausschließen. Seien α_1 und α_2 zwei Instanzen der Affixposition a^X und β, γ die Instanzen zweier anderer Affixpositionen. Wird nun β aus α_1 und γ aus α_2 berechnet, dann darf β nicht vor γ berechnet werden, wenn a^X als Kellerspeicher implementiert werden soll (siehe Abbildung 7-1, erste Zeichnung; die dicke Linie symbolisiert die zeitliche Reihenfolge der Berechnungen, die dünnen Pfeile zeigen die Datenflußabhängigkeiten an). Denn wenn die Werte von α_1 und α_2 auf dem Kellerspeicher liegen, so daß α_2 sich zuoberst befindet, dann kann bei der Berechnung von β nicht auf den Wert von α_1 zugegriffen werden. Ähnliches gilt für den Fall (siehe Abbildung 7-1, zweite Zeichnung), daß β und γ identisch sind. Zur Berechnung von $\beta = \gamma$ müßten sowohl der oberste, wie auch der darunterliegende Wert des Kellerspeichers benutzt werden. Auf einem Kellerspeicher ist jedoch stets nur der oberste Wert zugänglich, und es ist nicht eindeutig in welcher Reihenfolge die beiden Werte in der Berechnung benötigt werden. Alle anderen Reihenfolgen stören die Benutzung eines Kellerspeichers nicht. Besonders interessant sind in diesem Zusammenhang die „eingebetteten“ Berechnungsreihenfolgen, wie sie in Abbildung 7-2 dargestellt sind.

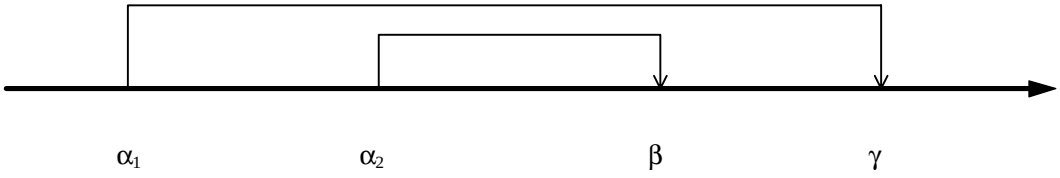


Abbildung 7-2: "Eingebettete" Berechnungsreihenfolgen

Um die oben genannte Problematik in einem Theorem ausdrücken zu können, das eineindeutig feststellt, ob eine Affixposition a^X als Kellerspeicher implementierbar ist, wird die *Menge der Besuchs-Abhängigkeiten* $VDS(a^X)$ benötigt, die folgendermaßen definiert ist:

Definition 7-1(VDS(a^X)):
VDS(a_m^X) enthält alle Tupel $(Y,(i,j))$ mit $Y \in HN$, $i,j \in [1,n(Y)]$ und $i<j$, für die ein dekorierter Ableitungsbaum t und ein Unterbaum t_0, dessen Wurzel mit Y markiert ist, existieren, so daß für eine Abhängigkeit $(a_l^{(r,k)}, b^{(r)}) \in DP(r)$ mit $m=l$ und $X=X_k^r$ der Affixparameter $a_l^{(r,k)}$ während des i-ten Besuchs des mit Y markierten Knotens und $b^{(r)}$ während des j-ten Besuchs berechnet werden.

Für die Untersuchung einer Affixposition sind also nur solche direkten Abhängigkeiten interessant, die von einem zur Affixposition korrespondierenden Affixparameter ausgehen. Die Berechnung der Menge *VDS* kann abstrakt wie folgt beschrieben werden:

1. Schritt: Für alle Hyper-Regeln $r=X_0: X_1...X_{\#S(r)}$ und alle direkten Abhängigkeiten $(a_l^{(r,k)}, b^{(r)}) \in DP(r)$, für die $a_l^{(r,k)}$ korrespondierender Affixparameter zur Affixposition a^X ist, wird das Tupel $(X_0, (i,j))$ in $VDS(a^X)$ eingefügt, wenn $a_l^{(r,k)}$ während des Planes VS_i^r und $b^{(r)}$ während des Planes VS_j^r berechnet wird und $i<j$ ist.
2. Schritt: Für alle Tupel $(Y,(i,j))$ der Menge $VDS(a^X)$, jede Hyper-Regel $r=X_0: X_1...X_{\#S(r)}$ und jedes $m \in [1,\#S(r)]$ mit $X_m=Y$ wird das Tupel $(X_0,(p,q))$ in die Menge $VDS(a^X)$ eingefügt, wenn die Besuchsinstruktionen $VISIT(X_m,i)$ im Plan VS_p^r und $VISIT(X_m,j)$ im Plan VS_q^r auftreten. Dies wird solange wiederholt, bis in $VDS(a^X)$ keine Tupel mehr neu hinzukommen.

Theorem 7-1:

Die Instanz einer Affixposition *a* kann im generierten SOAG-Evaluator genau dann nicht als Kellerspeicher implementiert werden, wenn eine Hyper-Regel $r=X_0: X_1...X_{\#S(r)}$ existiert, so daß (mindestens) eine der folgenden vier Bedingungen zutrifft:

1. Es existieren zwei direkte Abhängigkeit $(b,c) \in DP(r)$ und $(d,e) \in DP(r)$, so daß *b* und *d* korrespondierende Affixparameter zur Affixposition *a* sind und
   $set_r(b) < set_r(d) < set_r(c) < set_r(e)$ in EVS_r ist.
2. Es existiert eine direkte Abhängigkeit $(b,c) \in DP(r)$, bei der *b* ein korrespondierender Affixparameter zur Affixposition *a* ist und es ein Tupel $(X_m, (i,j))$ in $VDS(a)$ mit $m \in [1,\#S(r)]$ gibt, so daß
   $set_r(b) < VISIT(X_m,i) < set_r(c) < VISIT(X_m,j)$ in EVS_r gilt.
3. wie in 2., nur daß
   $VISIT(X_m,i) < set_r(b) < VISIT(X_m,j) < set_r(c)$ in EVS_r gilt.
4. Es existieren zwei Tupel $(X_k, (i,j))$ und $(X_m,(p,q))$ in $VDS(a)$ mit $k,m \in [1,\#S(r)]$ und $k \neq m$, so daß
   $VISIT(X_k,i) < VISIT(X_m,p) < VISIT(X_k,j) < VISIT(X_m,q)$ in EVS_r gilt.

Der Beweis für dieses Theorem ist in [EngJong] zu finden.

7.2 Optimierung durch globale Variablen

Wenn die Lebenszeiten der Instanzen von Affixpositionen, die als Kellerspeicher implementiert sind, einander nicht überlappen oder, anders ausgedrückt, wenn sich im Kellerspeicher immer nur höchstens ein Wert befindet, dann kann die Affixposition auch als globale Variable implementiert werden.

Um wiederum ein Theorem formulieren zu können, das eineindeutig aussagt, ob eine Affixposition als globale Variable implementierbar ist, wird die *Menge der Besuche* $VS(a^X)$ einer Affixposition a^X benötigt, die wie folgt definiert ist.

Definition 7-2($VS(a^X)$)**:**

Die Menge der Besuche $VS(a^X)$ einer Affixposition a^X enthält alle Paare (Y,i) mit $Y \in HN$ und $i \in [1,n(Y)]$, für die ein dekorierter Ableitungsbaum *t* und ein Knoten *y*, markiert mit *Y*, existieren, so daß mindestens eine Instanz von a^X während des *i*-ten Besuchs von *y* berechnet wird.

Theorem 7-2:

Die Instanz einer Affixposition *a* kann im generierten SOAG-Evaluator genau dann nicht als globale Variable implementiert werden, wenn es eine Hyper-Regel $r=X_0: X_1...X_{\#S(r)}$ gibt, so daß (mindestens) eine der folgenden vier Bedingungen zutrifft:

1. Es existiert eine direkte Abhängigkeit $(b,c) \in DP(r)$ und ein Affixparameter *d*, so daß *b* und *d* korrespondierende Affixparameter zur Affixposition *a* sind und
   $set_r(b) < set_r(d) < set_r(c)$ in EVS_r ist.
2. Es existiert eine direkte Abhängigkeit $(b,c) \in DP(r)$, bei der *b* ein korrespondierender Affixparameter zur Affixposition *a* ist und es ein Tupel (X_m, i) in $VS(a)$ mit $m \in [1,\#S(r)]$ gibt, so daß
   $set_r(b) < VISIT(X_m,i) < set_r(c)$ in EVS_r gilt.
3. Es existieren ein Tupel $(X_m, (i,j))$ in $VDS(a)$ mit $m \in [1,\#S(r)]$ und ein Affixparameter *d*, wobei *d* korrespondierender Affixparameter zur Affixposition *a* ist, so daß
   $VISIT(X_m,i) < set_r(b) < VISIT(X_m,j)$ in EVS_r gilt.
4. Es existieren ein Tupel $(X_k, (i,j))$ in $VDS(a)$ und ein Paar (X_m,q) in $VS(a)$ mit $k,m \in [1,\#S(r)]$ und $k \neq m$, so daß
   $VISIT(X_k,i) < VISIT(X_m,q) < VISIT(X_k,j)$ in EVS_r gilt.

Der Beweis für dieses Theorem ist in [EngJong] zu finden.

7.3 Implementierungsdetails

Die Algorithmen zur Berechnung der Optimierungsmöglichkeiten der Affixpositionsspeicherung sind im Modul `SOAGOptimizer` realisiert. Er enthält folgende modul-globale Datenstrukturen:

```
VAR
    PN: SOAG.OpenInteger;
    VDS, VS: ALists.AList;
    admissible, disjoint: BOOLEAN;
    GlobalVar, StackVar: INTEGER;
```

Das Feld PN erweitert die Liste der Visit-Sequenzen SOAG.VS. Durch die Initialisierungsprozedur Init wird jedem Eintrag in SOAG.VS die Nummer des Visit-Planes zugeordnet, in dem der Eintrag enthalten ist. Dies ist notwendig, um die Besuchsnummer nicht dynamisch berechnen zu müssen. Die eventuell im Eintrag enthaltene Besuchsnummer kann nicht verwendet werden, da sie sich auf den zu besuchenden Knoten bezieht.

Die in den oberen Kapiteln vorgestellten Mengen *VDS*(*a*) und *VS*(*a*) werden durch die beiden Listen VDS und VS implementiert. Um die Mengeneigenschaft sicherzustellen, prüfen die beiden Einfügeoperationen IncludeVDS und IncludeVS durch lineare Suche, daß keine Elemente doppelt in die Listen eingefügt werden. Die Berechnung der beiden Mengen erfolgt durch die Prozeduren InitVDSandVS und CompleteVDS, wobei die erste Prozedur den 1.Schritt des abstrakten Algorithmus aus dem Abschnitt 7.1 „Optimierung durch Kellerspeicher“ sowie die Berechnung der Menge *VS* implementiert. Die zweite Prozedur vervollständigt die Menge *VDS* gemäß dem 2.Schritt des abstrakten Algorithmus aus dem genannten Abschnitt. Die Funktion GetPlanNo liefert zur Berechnung der beiden Mengen die Nummer des Visit-Planes zurück, in dem der Wert des übergebenen Affixparameters ermittelt wird.

Die boolschen Variablen admissible und disjoint enthalten die Optimierungseigenschaften der aktuellen Affixposition. Gilt disjoint, so ist die Affixposition als globale Variable implementierbar. Ist admissible auf TRUE gesetzt, so ist die Speicherung in einem Kellerspeicher möglich. Sind beide Variablen nicht gesetzt, dann ist für die aktuelle Affixposition keine Optimierung möglich. Die beiden modul-globalen Variablen StackVar und GlobalVar geben jeweils die Nummer des nächsten zu vergebenden Kellerspeichers bzw. der nächsten zu vergebenden globalen Variable an.

Die Theoreme 7-1 und 7-2 erfordern die Realisierung der Relationen „<“ und „≤“ auf der Menge jeder erweiterten Visit-Sequenz, die jedoch in der Implementierung nicht konkret vorhanden sind. Da sich die beiden Relationen praktisch auf zu vergleichende Positionen innerhalb der erweiterten Visit-Sequenz beziehen, lassen sie sich für Besuchsinstruktionen auf einen Vergleich der Positionen innerhalb der „einfachen“ Visit-Sequenz abbilden. Um für die Funktion $set_r(a)$ die nachträgliche Realisierung der erweiterten Visit-Sequenz zu umgehen, wird die „einfache“ Visit-Sequenz symbolisch gestreckt, so daß vor und nach jeder Besuchsinstruktion eine zusätzliche Position frei wird. Die beiden Funktionen GetEVSPosforAffOcc und GetEVSPosforVisit realisieren diese symbolische Streckung. Die Funktion GetEVSPosforVisit bestimmt für die übergebene Besuchsinstruktion ihre Position in der „einfachen“ Visit-Sequenz, multipliziert sie mit drei und addiert zwei hinzu, um sie in der Mitte des erweiterten Bereiches zu plazieren. Sei *V* die Position der Besuchsinstruktion in der „einfachen“ Visit-Sequenz, dann ergibt sich die Position der Besuchsinstruktion in der erweiterten Visit-Sequenz aus

$$\text{Position in } EVS = V * 3 + 2.$$

Die Funktion GetEVSPosforAffOcc ermittelt für einen Affixparameter die Position der Besuchsinstruktion, vor bzw. nach deren Evaluation der Wert des Affixparameters berechnet wird, und positioniert die den Affixparameter enthaltene Menge vor bzw. nach der zugehörigen Besuchsinstruktion. In Abhängigkeit davon, ob der Affixparameter zu einer inherited- oder synthesized-Affixposition korrespondiert, wird die Position in der erweiterten Visit-Sequenz berechnet als

inherited: Position in $EVS = (V * 3 + 2) - 1 = V * 3 + 1$,
synthesized: Position in $EVS = (V * 3 + 2) + 1 = V * 3 + 3$.

Die Prozedur CheckStorageType nimmt die eigentliche Berechnung der Optimierungseigenschaften für eine übergebene Affixposition vor. Sie implementiert in ihren Unterprozeduren CheckT2P1andT1P1, CheckT1P2andP3, CheckT1P4, CheckT2P2, CheckT2P3 und CheckT2P4 die einzelnen in den Theoremen formulierten Bedingungen. Trifft eine der Bedingungen zu, so wird eine der globalen Variablen admissible und disjoint auf FALSE gesetzt. Die Optimierungseigenschaften werden für alle Affixposition in Evaluatorregeln und alle inherited-Affixposition in Prädikaten berechnet.

Die exportierte Prozedur Optimize nimmt die Optimierungsberechnung für alle notwendigen Affixpositionen vor. Die Optimierungseigenschaften einer Affixposition werden in der Datenstruktur SOAG.StorageName festgehalten. Ist in diesem Feld für eine Affixposition Null eingetragen, so kann keine Optimierung vorgenommen werden. Ein negativer Eintrag bedeutet, daß die Affixposition als globale Variable implementiert werden kann; der Absolutwert gibt die Nummer der Variablen an, aus der für den Evaluator ein eindeutiger Name generiert wird. Ein positiver Eintrag gibt die Nummer des Kellerspeichers an, der für die Affixposition verwendet werden soll.

7.4 Implementierung

7.4.1 eSOAGOptimizer.Mod

```
MODULE eSOAGOptimizer; (* dk 1.03 05.03.98 *)

IMPORT EAG := eEAG, SOAG := eSOAG, SOAGVisitSeq := eSOAGVisitSeq, ALists := eALists,
      Sets := eSets, Protocol := eSOAGProtocol, IO := eIO;

CONST
  firstGlobalVar* = 1;
  firstStackVar* = 1;

VAR
  GlobalVar*,
  StackVar*: INTEGER;

VAR
  PN: SOAG.OpenInteger; (* parallel zu SOAG.VS *)
  admissible, disjoint: BOOLEAN;
  VDS, VS: ALists.AList;


(* Hilfsprozeduren *)

PROCEDURE IncludeVDS(S, VN1, VN2: INTEGER);
(*IN: Tripel
 OUT: -
 SEM: bed. Einfuegen des Tripels in die modulglobale Liste VDS, die als Menge interpretiert
wird,
    deshalb wird das Tripel nur dann eingefuegt, wenn es nicht schon Bestandteil der Liste
ist. *)
VAR i: INTEGER; notisElement: BOOLEAN;
BEGIN
  i := ALists.firstIndex; notisElement := TRUE;
  WHILE (i < VDS.Last) & notisElement DO
    notisElement := (VDS.Elem[i] # S) OR (VDS.Elem[i+1] # VN1) OR (VDS.Elem[i+2] # VN2);
    INC(i,3)
  END;
  IF notisElement THEN
    ALists.Append(VDS, S); ALists.Append(VDS, VN1); ALists.Append(VDS, VN2)
  END
END IncludeVDS;

PROCEDURE WriteVDS;
VAR i: INTEGER;
BEGIN
  IO.WriteText(IO.Msg, "Inhalt VDS:\n");
  FOR i :=  ALists.firstIndex TO VDS.Last BY 3 DO
    EAG.WriteHNont(IO.Msg, VDS.Elem[i]);
    IO.WriteText(IO.Msg, ", "); IO.WriteInt(IO.Msg, VDS.Elem[i+1]);
    IO.WriteText(IO.Msg, ", "); IO.WriteInt(IO.Msg, VDS.Elem[i+2]);
    IO.WriteLn(IO.Msg); IO.Update(IO.Msg)
  END
END WriteVDS;


PROCEDURE IncludeVS(S, VN: INTEGER);
(*IN: Tupel
 OUT: -
 SEM: bed. Einfuegen des Tupels in die modulglobale Liste VS, die als Menge interpretiert
wird,
    deshalb wird das Tupel nur dann eingefuegt, wenn es nicht schon Bestandteil der Liste ist.
*)
VAR i: INTEGER; notisElement: BOOLEAN;
BEGIN
  i := ALists.firstIndex; notisElement := TRUE;
  WHILE (i < VS.Last) & notisElement DO
    notisElement := (VS.Elem[i] # S) OR (VS.Elem[i+1] # VN);
    INC(i,2)
  END;
  IF notisElement THEN
    ALists.Append(VS, S); ALists.Append(VS, VN);
  END
END IncludeVS;

PROCEDURE WriteVS;
VAR i: INTEGER;
BEGIN
  IO.WriteText(IO.Msg, "Inhalt VS:\n");
  FOR i :=  ALists.firstIndex TO VS.Last BY 2 DO
```

```
    EAG.WriteHNont(IO.Msg, VS.Elem[i]);
    IO.WriteText(IO.Msg, ", "); IO.WriteInt(IO.Msg, VS.Elem[i+1]);
    IO.WriteLn(IO.Msg); IO.Update(IO.Msg)
  END
END WriteVS;



PROCEDURE GetPlanNo(R,AP: INTEGER): INTEGER;
(*IN: Regel, Affixparameter
 OUT: Plannummer
 SEM: Ermittelt die Nummer des Visitplanes, waehrend dessen Auswertung der Affixparameter
    berechnet wird *)
VAR SO, VN: INTEGER;
BEGIN
  SO := SOAG.AffOcc[AP].SymOccInd;
  IF SO = SOAG.Rule[R].SymOcc.Beg THEN  (* Affpar. d. linke Regelseite *)
    RETURN SOAGVisitSeq.GetVisitNo(AP)
  ELSE  (* Affpar. d. rechte Regelseite *)
    VN := SOAGVisitSeq.GetVisitNo(AP);
    RETURN PN[SOAGVisitSeq.GetVisit(R,SO,VN)]
  END
END GetPlanNo;


PROCEDURE GetEVSPosforAffOcc(R,AP: INTEGER): INTEGER;
(*IN: Regel, Affixparameter
 OUT: Position in der virtuellen extended visit sequence (EVS)
 SEM: Ermittelt der Position des Affixparameter in der EVS (entspricht set(a) aus der Theorie)
*)
VAR SO, VN, V, S, AN: INTEGER;
BEGIN
  SO := SOAG.AffOcc[AP].SymOccInd;  S := SOAG.SymOcc[SO].SymInd;
  VN :=  SOAGVisitSeq.GetVisitNo(AP);  AN := SOAG.AffOcc[AP].AffOccNum.InSym;
  IF (SO = SOAG.Rule[R].SymOcc.Beg) & SOAG.IsInherited(S, AN) THEN
    IF VN = 1 THEN RETURN 0
    ELSE
      V := SOAGVisitSeq.GetVisit(R,SO,VN-1);
      RETURN V*3+1
    END
  END;
  V := SOAGVisitSeq.GetVisit(R, SO, VN);
  IF SOAG.IsInherited(S, AN) THEN RETURN V*3+1
  ELSE (* IsSynthesized(AO)*)    RETURN V*3+3 END
END GetEVSPosforAffOcc;


PROCEDURE GetEVSPosforVisit(R, SO, VN: INTEGER): INTEGER;
(*IN: Regel, Symbolvorkommen, Visit-Nummer
 OUT: Position in der virtuellen extended visit sequence (EVS)
 SEM: Ermittelt der Position des durch Symbolvorkommen und Visitnummer eindeutig
    gekennzeichneten Visits in der EVS (entspricht visit(j,m) aus der Theorie) *)
VAR V: INTEGER;
BEGIN
  V := SOAGVisitSeq.GetVisit(R, SO, VN);
  RETURN V*3+2
END GetEVSPosforVisit;

(* Berechnungen *)

PROCEDURE Init;
(* SEM: Initialisierung der Struktur PN - Berechnung der Plannummer jedes Visits *)
VAR R, V, PlanNo: INTEGER;
BEGIN
  NEW( PN, SOAG.NextVS );
  FOR R := SOAG.firstRule TO SOAG.NextRule-1 DO
    PlanNo := 1;
    FOR V := SOAG.Rule[R].VS.Beg TO SOAG.Rule[R].VS.End DO
      PN[V] := PlanNo;
      IF SOAG.VS[V] IS SOAG.Leave THEN INC(PlanNo) END
    END
  END;

  NEW( SOAG.StorageName, SOAG.NextPartNum );
  SOAG.NextStorageName := SOAG.NextPartNum;
  FOR V := SOAG.firstStorageName TO SOAG.NextStorageName-1 DO
    SOAG.StorageName[V] := 0
  END;

  ALists.New(VDS,16); ALists.New(VS,16);
END Init;
```

```
PROCEDURE InitVDSandVS(S, A: INTEGER);
(*IN: Symbol, Affixpositionsnummer
 OUT: -
 SEM: Initialisierung der Mengen VDS und VS fuer eine Affixposition (analog Step 1 Theorie)
*)
VAR SO, AP, PN,R, RS, AP1, PN1, AN, AN1: INTEGER;
BEGIN
  ALists.Reset(VDS); ALists.Reset(VS);
  SO := SOAG.Sym[S].FirstOcc;
  WHILE SO # SOAG.nil DO
    AP := SOAG.SymOcc[SO].AffOcc.Beg + A;
    R := SOAG.SymOcc[SO].RuleInd;
    IF SOAG.IsEvaluatorRule(R) THEN
      PN := GetPlanNo(R, AP);
      RS := SOAG.SymOcc[SOAG.Rule[R].SymOcc.Beg].SymInd; (* Symbol d. linken Regelseite *)
      IncludeVS(RS,PN);
      FOR AP1 := SOAG.Rule[R].AffOcc.Beg TO SOAG.Rule[R].AffOcc.End DO
        AN1 := SOAG.AffOcc[AP1].AffOccNum.InRule;
        AN := SOAG.AffOcc[AP].AffOccNum.InRule;
        IF Sets.In(SOAG.Rule[R].DP[AN], AN1) THEN
          PN1 := GetPlanNo(R, AP1);
          IF PN < PN1 THEN IncludeVDS(RS, PN, PN1) END
        END
      END
    END;
    SO := SOAG.SymOcc[SO].Next
  END
END InitVDSandVS;

PROCEDURE CompleteInitVDS;
(* SEM: Kompletiert die Initialisierung der Menge VDS (analog Step 2 der Theorie) *)
VAR i, R, S, SO, VN1, VN2, V1, V2, RS: INTEGER;
BEGIN
  i := ALists.firstIndex;
  WHILE i < VDS.Last DO
    S := VDS.Elem[i]; INC(i);
    VN1 :=  VDS.Elem[i]; INC(i);
    VN2 :=  VDS.Elem[i]; INC(i);
    SO := SOAG.Sym[S].FirstOcc;
    WHILE SO # SOAG.nil DO
      R := SOAG.SymOcc[SO].RuleInd;
      IF SOAG.IsEvaluatorRule(R) THEN
        IF SOAG.Rule[R].SymOcc.Beg # SO THEN (* SO nicht auf der linken S. der Regel *)
          V1 := SOAGVisitSeq.GetVisit(R, SO, VN1);
          V2 := SOAGVisitSeq.GetNextVisit(V1, R, SO, VN2);
          IF PN[V1] < PN[V2] THEN
            RS := SOAG.SymOcc[SOAG.Rule[SOAG.SymOcc[SO].RuleInd].SymOcc.Beg].SymInd;
            (* Symbol der linken Regelseite *)
            IncludeVDS(RS, PN[V1], PN[V2]);
          END
        END
      END;
      SO := SOAG.SymOcc[SO].Next
    END
  END
END CompleteInitVDS;

PROCEDURE CheckStorageType(S, A: INTEGER);
(*IN: Symbol, Affixpos.Nr.
 OUT: -
 SEM: Test, ob Affixposition als Stack oder als globale Variable abgespeichert werden kann -
    nach Theorem 1 und 3 der Theorie
*)
VAR R, SO, AP1, AN1, VN1, AP2, AN2, VN2, t, S1, SO1, V1, V2, PN1, PN2: INTEGER;

  PROCEDURE CheckT2P1andT1P1(S, A, R, PN1, PN2: INTEGER);
  (*IN: Symbol, Affixpos.Nr., aktuelle Regel, zwei Positionen der EVS*)
  VAR SO1, AP3, AN3, AP4, AN4, PN3, PN4: INTEGER;
  BEGIN
    SO1 := SOAG.Sym[S].FirstOcc;
    WHILE SO1 # SOAG.nil DO
      IF (R = SOAG.SymOcc[SO1].RuleInd) THEN
        AP3 := SOAG.SymOcc[SO1].AffOcc.Beg + A;
        AN3 := SOAG.AffOcc[AP3].AffOccNum.InRule;
        PN3 := GetEVSPosforAffOcc(R, AP3);

        IF (PN1 < PN3) & (PN3 < PN2) THEN disjoint := FALSE END; (* T2(1) *)

        FOR AP4 := SOAG.Rule[R].AffOcc.Beg TO SOAG.Rule[R].AffOcc.End DO
          AN4 := SOAG.AffOcc[AP4].AffOccNum.InRule;
          IF Sets.In(SOAG.Rule[R].DP[AN3], AN4) THEN
```

```
          PN4 := GetEVSPosforAffOcc(R, AP4);

          IF (PN1 < PN3) & (PN3 < PN2) & (PN2 <= PN4)
          THEN admissible := FALSE END;  (* T1(1) *)

        END (* IF *)
      END (* FOR *)
    END;  (* IF *)
    SO1 := SOAG.SymOcc[SO1].Next
  END; (* WHILE *)
END CheckT2P1andT1P1;

PROCEDURE CheckT2P2(R, PN1, PN2: INTEGER);
(*IN: aktuelle Regel, zwei Positionen in der EVS *)
VAR t, S1, SO1, PN, VN: INTEGER;
BEGIN
  FOR t := ALists.firstIndex TO VS.Last BY 2 DO
    S1 := VS.Elem[t]; VN := VS.Elem[t+1];
    SO1 := SOAG.Sym[S1].FirstOcc;
    WHILE SO1 # SOAG.nil DO
      IF (R = SOAG.SymOcc[SO1].RuleInd) & (SOAG.Rule[R].SymOcc.Beg # SO1) THEN
        PN := GetEVSPosforVisit(R, SO1, VN);

        IF (PN1 < PN) & (PN < PN2) THEN disjoint := FALSE END; (* T2(2) *)

      END; (* IF *)
      SO1 := SOAG.SymOcc[SO1].Next
    END (* WHILE *)
  END; (* FOR *)
END CheckT2P2;

PROCEDURE CheckT1P2andP3(R, PN1, PN2: INTEGER);
(*IN: aktuelle Regel, zwei Positionen in der EVS *)
VAR t, S1, SO1, VN3, VN4, PN3, PN4: INTEGER;
BEGIN
  FOR t := ALists.firstIndex TO VDS.Last BY 3 DO
    S1 := VDS.Elem[t]; VN3 := VDS.Elem[t+1]; VN4 := VDS.Elem[t+2];
    SO1 := SOAG.Sym[S1].FirstOcc;
    WHILE SO1 # SOAG.nil DO
      IF (R = SOAG.SymOcc[SO1].RuleInd) & (SOAG.Rule[R].SymOcc.Beg # SO1) THEN
        PN3 := GetEVSPosforVisit(R, SO1, VN3);
        PN4 := GetEVSPosforVisit(R, SO1, VN4);

        IF ((PN1 < PN3) & (PN3 < PN2) & (PN2 < PN4)) OR
          ((PN3 < PN1) & (PN1 < PN4) & (PN4 < PN2))  (* T1(2) u. (3) *)
        THEN admissible := FALSE END

      END;
      SO1 := SOAG.SymOcc[SO1].Next
    END (* WHILE *)
  END; (* FOR *)
END CheckT1P2andP3;

PROCEDURE CheckT2P3(S, A, R, PN1, PN2: INTEGER);
(*IN: Affixpos.Nr., aktuelle Regel, zwei Position in der EVS *)
VAR SO2, AP1, PN3: INTEGER;
BEGIN
  FOR SO2 := SOAG.Rule[R].SymOcc.Beg TO SOAG.Rule[R].SymOcc.End DO
    IF SOAG.SymOcc[SO2]. SymInd = S THEN
      AP1 := SOAG.SymOcc[SO2].AffOcc.Beg + A;
      PN3 := GetEVSPosforAffOcc(R, AP1);

      IF (PN1 < PN3) & (PN3 < PN2) THEN disjoint := FALSE END (* T2(3) *)

    END (* IF *)
  END (* FOR *)
END CheckT2P3;

PROCEDURE CheckT2P4(R, SO, PN1, PN2: INTEGER );
(*IN: aktuelle Regel, zwei Position in der EVS *)
VAR t, S2, SO2, VN3, PN3: INTEGER;
BEGIN
  FOR t := ALists.firstIndex TO VS.Last BY 2 DO
    S2 := VS.Elem[t]; VN3 := VS.Elem[t+1];
    SO2 := SOAG.Sym[S2].FirstOcc;
    WHILE SO2 # SOAG.nil DO
      IF (R = SOAG.SymOcc[SO2].RuleInd)  & (SOAG.Rule[R].SymOcc.Beg # SO2) &
        (SO # SO2) THEN
        PN3 := GetEVSPosforVisit(R, SO2, VN3);

        IF (PN1 < PN3) & (PN3 < PN2) THEN disjoint := FALSE END (* T2(4) *)
```

```
        END; (* IF *)
        SO2 := SOAG.SymOcc[SO2].Next
      END
    END; (* FOR *)
  END CheckT2P4;

  PROCEDURE CheckT1P4(R, SO, PN1, PN2: INTEGER);
  (*IN: aktuelle Regel, zwei Position in der EVS *)
  VAR t, S2, SO2, VN3, VN4, PN3, PN4: INTEGER;
  BEGIN
    FOR t := ALists.firstIndex TO VDS.Last BY 3 DO
      S2 := VDS.Elem[t]; VN3 := VDS.Elem[t+1]; VN4 := VDS.Elem[t+2];
      SO2 := SOAG.Sym[S2].FirstOcc;
      WHILE SO2 # SOAG.nil DO
        IF (R = SOAG.SymOcc[SO2].RuleInd) & (SOAG.Rule[R].SymOcc.Beg # SO2) &
          (SO # SO2) THEN
          PN3 := GetEVSPosforVisit(R, SO2, VN3);
          PN4 := GetEVSPosforVisit(R, SO2, VN4);

          IF (PN1 < PN3) & (PN3 < PN2) & (PN2 < PN4)  (* T1(4) *)
          THEN admissible := FALSE END

        END; (* IF *)
        SO2 := SOAG.SymOcc[SO2].Next
      END (* WHILE *)
    END; (* FOR *)
  END CheckT1P4;

BEGIN
  SO := SOAG.Sym[S].FirstOcc;
  WHILE SO # SOAG.nil DO
    AP1 := SOAG.SymOcc[SO].AffOcc.Beg + A;
    AN1 := SOAG.AffOcc[AP1].AffOccNum.InRule;
    R := SOAG.SymOcc[SO].RuleInd;
    IF SOAG.IsEvaluatorRule(R) THEN
      PN1 := GetEVSPosforAffOcc(R,AP1);
      FOR AP2 := SOAG.Rule[R].AffOcc.Beg TO SOAG.Rule[R].AffOcc.End DO
        AN2 := SOAG.AffOcc[AP2].AffOccNum.InRule;
        IF Sets.In(SOAG.Rule[R].DP[AN1], AN2) THEN
          PN2 := GetEVSPosforAffOcc(R,AP2);
          CheckT2P1andT1P1(S, A, R, PN1, PN2);
          CheckT2P2(R, PN1, PN2);
          CheckT1P2andP3(R, PN1, PN2)
        END (* IF Sets.In.. *)
      END (* FOR AP2.. *)
    END; (* IF *)
    SO := SOAG.SymOcc[SO].Next
  END; (* WHILE SO # .. *)

  FOR t := ALists.firstIndex TO VDS.Last BY 3 DO
    S1 := VDS.Elem[t]; VN1 := VDS.Elem[t+1]; VN2 := VDS.Elem[t+2];
    SO1 := SOAG.Sym[S1].FirstOcc;
    WHILE SO1 # SOAG.nil DO
      R := SOAG.SymOcc[SO1].RuleInd;
      IF SOAG.IsEvaluatorRule(R) THEN
        IF SOAG.Rule[R].SymOcc.Beg # SO1 THEN
          PN1 := GetEVSPosforVisit(R, SO1, VN1); PN2 := GetEVSPosforVisit(R, SO1, VN2);
          CheckT2P3(S, A, R, PN1, PN2);
          CheckT2P4(R, SO1, PN1, PN2);
          CheckT1P4(R, SO1, PN1, PN2)
        END
      END;
      SO1 := SOAG.SymOcc[SO1].Next
    END (* WHILE *)
  END; (* FOR t .. *)

END CheckStorageType;


PROCEDURE Optimize*;
VAR S, AP, A: INTEGER;
BEGIN
  Init;
  GlobalVar := firstGlobalVar-1; StackVar := firstStackVar-1;
  FOR S := SOAG.firstSym TO SOAG.NextSym-1 DO
    FOR AP := SOAG.Sym[S].AffPos.Beg TO SOAG.Sym[S].AffPos.End DO
      A := AP - SOAG.Sym[S].AffPos.Beg;
      IF ~Sets.In(EAG.Pred, S) OR SOAG.IsSynthesized(S, A) THEN
      disjoint := TRUE; admissible := TRUE;
      InitVDSandVS(S,A);
      CompleteInitVDS;
      CheckStorageType(S,A);
```

```
        IF disjoint THEN INC(GlobalVar); SOAG.StorageName[AP] := -GlobalVar
        ELSIF admissible THEN INC(StackVar); SOAG.StorageName[AP] := StackVar END
      END
    END
  END
END Optimize;

END eSOAGOptimizer.
```

8 Aufwandsanalyse

Zunächst werden, um den zu untersuchenden Gesamtaufwand des approximierenden SOAG-Verfahrens ausdrücken zu können, folgende Größen eingeführt:

$x = max\{|A(X)| : X \in HN\}$ max. Anzahl der Affixpositionen pro Symbol und
$p = max\{ \#S(r)+1 : r \in HR\}$ max. Anzahl an Symbolvorkommen pro Hyper-Regel

Der Initialisierungsschritt $TDP = (D^{-1})^+$ ist trivial, da aufgrund der Bochmann-Normalform D^{-1} bereits transitiv ist.

Die Implementierung des transitiv abgeschlossenen *ind*-Operators erfordert transitive Einfügungen in *TDP*. Hierbei ist zu beachten, daß im Falle von Backtracking (siehe Abschnitt 5.3 „Zwei heuristische Teilklassen der SOEAGen“) zusätzlicher Aufwand zu betrachten ist. Denn führt für eine festzulegende Anordnung zwischen zwei Affixpositionen *a* und *b* das Übernehmen der bevorzugten Abhängigkeit (*a*,*b*) auf entsprechende Affixparameter in *TDP* einschließlich Induzieren zu einem Zyklus, so müssen die neu hinzugekommenen Abhängigkeiten zunächst wieder aus *TDP* entfernt werden, bevor alternativ die umgekehrte Abhängigkeit (*b*,*a*) übernommen werden kann. Daher wird über die Erweiterungen von *TDP* mit entsprechendem Aufwand Buch geführt.

Für die Ermittlung einer Kastens-Vervollständigung $DS_j(X)$ wird ein modifiziertes topologisches Sortieren der Affixpositionen $A(X)$ gemäß der aus $TDP = IDP_{j-1}^+$ projizierten Abhängigkeiten durchgeführt.

Schließlich erfordert die Generierung von Visit-Sequenzen ein topologisches Sortieren der Affixparameter einer Regel gemäß der endgültigen Abhängigkeiten zu jeder Regel in $TDP = IDP_m^+$.

Zusammenfassend sind also hinsichtlich der Implementierung des Verfahrens zwei charakteristische Algorithmen für den Gesamtaufwand wesentlich:

- *topologisches Sortieren* von Affixpositionen bzw. Affixparametern und
- *transitives Einfügen* von Abhängigkeiten zwischen Affixparametern.

Ein topologisches Sortieren der Affixpositionen $A(X)$ für die Bestimmung einer Kastens-Vervollständigung $DS_j(X)$ für ein $X \in HN$ ist vom Aufwand $O(x^2)$; ein topologisches Sortieren der höchstens $p\ x$ zu einer Regel r gehörenden Affixparameter im Zuge der Erstellung einer Visit-Sequenz für r ist vom Aufwand $O(p^2\ x^2)$.

Wie in Abschnitt 5.4 „Dynamische transitive Hülle“ erläutert, erfordert das schrittweise transitive Erweitern einer beliebigen Relation R^+ über einer endlichen Trägermenge S um die Abhängigkeiten aus einer Menge $R' \subseteq S^2$ einen Aufwand von $O(|S|^3)$. Angenommen, die vom beschriebenen Verfahren für die vorliegende EAG ermittelte elementare Orientierungsreihenfolge $(C_1,...,C_m)$ ist *unmittelbar* erfolgreich, dann können nacheinander alle m Affixpositionsanordnungen auf die entsprechenden Affixparameter in *TDP* übertragen und induziert werden, ohne daß dabei ein Zyklus entsteht. Und kumulativ verursachen sämtliche dabei vorgenommenen transitiven Einfügungen in *TDP* dann den Aufwand $O(|HR|\ p^3\ x^3)$, denn es können in *TDP* stets nur Abhängigkeiten zwischen den höchstens $p\ x$ Affixparametern je einer Hyper-Regel entstehen.

Im Falle einer erfolgreichen anstelle einer *unmittelbar* erfolgreichen Orientierungsreihenfolge wird eine bestimmte Anzahl von Affixpositionsanordnungen nicht mit der bevorzugten, sondern erst mit der umgekehrten Affixparameterabhängigkeit zu einem azyklischen *TDP* erweitert. Sei diese Anzahl mit β angegeben, dann ist zusätzlich β-mal ein weiterer Aufwand von $O(|HR|\ p^3\ x^3)$ für transitive Einfügungen in *TDP* zu kalkulieren, die zu einem Zyklus führten. Diese Aufwandsangabe deckt auch die notwendigen Rücknahmen der getätigten *TDP*-Erweiterungen ab. Für den Aufwand des gesamten Verfahrens ergibt sich damit vorläufig

$$O(|HN|\ x^2\ x^2 + |HR|\ p^2\ x^2 + |HR|\ p^3\ x^3\ (1 + \beta)) \quad = \quad O(|HN|\ x^4 + |HR|\ p^3\ x^3\ (1 + \beta)).$$

Der Summand $|HN|\ x^4$ ist dabei Folge der Annahme, daß für die j-te $(1 \le j \le m \le |HN|\ x^2)$ festzulegende Anordnung zweier Affixpositionen aus $I(X) * S(X)$, ein vollständiges topologisches Sortieren von $A(X)$ gemäß der Projektion von IDP_{j-1}^+ auf $A(X)$ mit dem Ergebnis $DS_j(X)$ notwendig ist, obwohl die Gültigkeit von $IDP_{j-1}^+ \subseteq IDP_j^+$ gewährleistet ist und somit Information aus dem $(j-1)$-ten topologischen Sortiervorgang in den j-ten einfließen könnte. Der in Abschnitt 5.5 „Dynamisches topologisches Sortieren“ vorgestellte Algorithmus verarbeitet die Auswirkungen der Erweiterung von $TDP = IDP_{j-1}^+$ zu $TDP = IDP_j^+$ auf die Projektion von *TDP* auf $A(X)$ auf derartige Weise, daß beim topologischen Sortieren von $A(X)$ gemäß ebendieser Projektion alle

Affixpositionen zu einem Symbol *X* in einem einzigen *dynamischen* topologischen Sortiervorgang festgelegt werden können. Die Reihenfolge der Affixpositionsanordnungen ergibt sich aus diesem topologischen Sortiervorgang. Dabei führen natürlich nur solche Affixparameterabhängigkeiten zu echten Erweiterungen von *TDP*, die in der Projektion zuvor noch ohne Beziehung waren.

Somit ergibt sich für das approximative SOEAG-Verfahren ein Aufwand, der mit

$$O(|HN|\, x^2 + |HR|\, p^3\, x^3\, (1 + \beta))$$

abgeschätzt werden kann, wobei $\beta \leq m \leq |HN|\, x^2$ ist.
Für den Spezialfall $\beta = 0$, der gerade OEAGen umfaßt, ergibt sich der Aufwand

$$O(|HN|\, x^2 + |HR|\, p^3\, x^3).$$

Damit ist das approximative SOEAG-Verfahren nicht nur mächtiger, sondern zudem auch nicht aufwendiger als eine geschickte Implementierung des OEAG-Verfahrens.

Für den Aufwand des Optimierungsverfahrens bestimmend ist die Berechnung der Menge *VDS*. Der erste Schritt im abstrakten Algorithmus aus Abschnitt 7.1 „Optimierung durch Kellerspeicher“ S. 52 hat für die Berechnung der Eigenschaften einer Affixposition einen asymptotischen Aufwand von $O(|HR|\, p^2\, x)$, denn ein korrespondierender Affixparameter könnte in jedem Hyper-Nichtterminal einer Regel enthalten sein und eine Abhängigkeit zu allen anderen Affixparametern der Regel besitzen. Unter der Annahme, daß in jeder Partitionsmenge maximal zwei Affixpositionen enthalten sind, ergibt sich die kombinatorische Kardinalität der Menge *VDS* aus $|HN|*((x/2)^2+1)$, was asymptotisch $O(|HN|\, x^2)$ entspricht. Der zweite Schritt hat deshalb einen Aufwand von

$$O(|HN|\, x^2 * |HR|\, p) \quad = \quad O(|HR||HN|\, p\, x^2).$$

Die Prüfung der Optimierungseigenschaft beschränkt sich in Punkt 4 (dem aufwendigsten Punkt) des ersten Theorems aus Abschnitt 7.1 auf einen asymptotischen Aufwand von

$$O((|HN|\, x^2)^2 * |HR|) \quad = \quad O(|HR||HN|^2\, x^4),$$

denn die Menge *VDS* wird in zwei verschränkten Schleifen jeweils einmal durchlaufen. Da die Optimierung für alle Affixpositionen der Grammatik berechnet werden muß, ergibt sich für den gesamten Optimierungsalgorithmus ein asymptotischer Aufwand von

$$O(|HR||HN|^3\, x^5).$$

9 Generierung von Evaluationscode

9.1 Die generierten Evaluatoren

Um den Compilergenerator Epsilon möglichst einheitlich zu gestalten und schon vorhandene Implementierungen wiederverwenden zu können, wurden die Evaluatoren des SOAG-Evaluatorgenerators in Anlehnung an die in [DeWe] vorgestellten Konzepte entwickelt. Da die generierten Compiler möglichst umfassend und vollständig dargestellt werden sollen, wurden einige Passagen aus [DeWe] übernommen.

9.1.1 Datenstrukturen

Da der vom Parser erzeugte Ableitungsbaum nicht direkt um die notwendigen Instanzen der Affixparameter erweitert werden kann, wird in einem ersten, „syntaktisch“ genannten Paß eines Evaluators parallel zum kontextfreien Ableitungsbaum ein strukturgleicher zweiter Ableitungsbaum erzeugt, der die semantischen Informationen der Evaluation, also die berechneten Affixwerte, aufnimmt. Jeder Knoten dieses „semantischen“ oder *dekorierten* Ableitungsbaumes enthält folgende Datenstruktur:

```
TYPE
    IndexType = LONGINT;

    SemTreeEntry = POINTER TO RECORD
      Rule: LONGINT;
      Pos: IO.Position;
      Adr, VarInd: IndexType
    END;
```

Der Eintrag `Rule` markiert jeden Knoten mit der Regel, deren Struktur dem durch den Knoten und seine Söhne gebildeten Teilbaum entspricht. Der Eintrag `Pos` enthält die in Fehlerfällen auszugebende Position im Eingabetext. Um Speicherplatz zu sparen, ist jeder Knoten gleichzeitig Symbol einer rechten Regelseite als auch Symbol der linken Seite der Regel, mit der er markiert ist. Deshalb ist im Eintrag `Adr` ein Verweis auf das erste Symbol der zugehörigen rechten Regelseite abgespeichert. Die Symbole einer rechten Regelseite liegen zusammenhängend hintereinander im Feld `SemTree` vor, das den dekorierten Ableitungsbaum aufnimmt. Der kontextfreie Ableitungsbaum ist nach der vollständigen Konstruktion des dekorierten Ableitungsbaumes nicht mehr notwendig, sein Speicherplatz könnte freigegeben werden.

```
TYPE
    OpenSemTree = POINTER TO ARRAY OF SemTreeEntry;
VAR
    SemTree: OpenSemTree;
```

Die Abbildung 9-1 verdeutlicht an einem Beispiel den Aufbau eines dekorierten Ableitungsbaumes.

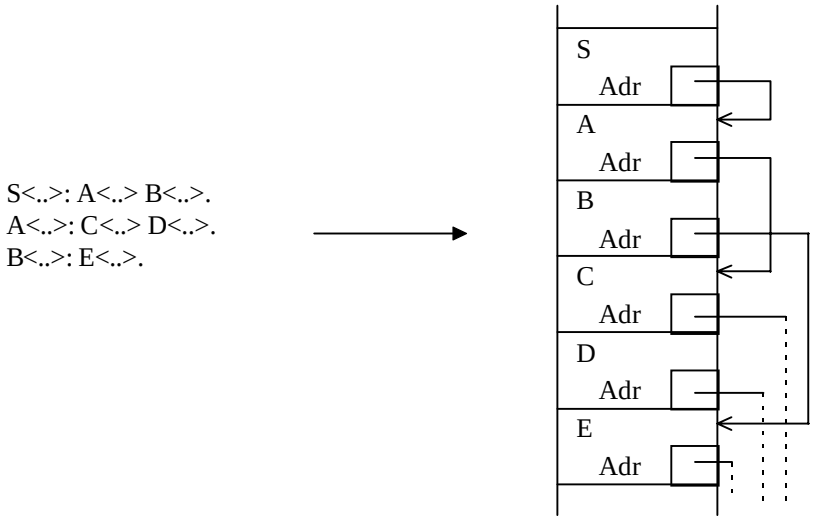


Abbildung 9-1: Aufbau des dekorierten Ableitungsbaumes

```
TYPE
    OpenVar = POINTER TO ARRAY OF HeapType;
    OpenAffPos = POINTER TO ARRAY OF HeapType;

VAR
    Var: OpenVar;
    AffPos: OpenAffPos;
    NextSemTree,
    NextVar,
    NextAffPos: IndexType;
```

Das Feld `Var` stellt die eigentliche Dekoration des Ableitungsbaumes dar. Sie enthält die Affixvariablen aller Regeln. Der Eintrag `VarInd` in der Datenstruktur `SemTreeEntry` verweist auf die erste Affixvariable der Regel, mit der der Knoten markiert ist. Die Affixvariablen einer Regel liegen im Feld `Var` in einem zusammenhängenden Block vor. Die Variablen `NextSemTree` und `NextVar` verweisen auf den ersten freien Eintrag des jeweiligen Feldes.

Als Affixparameterwerte werden die Sätze zum Wertebereichssymbol durch deren Ableitungsbäume repräsentiert. Dies erlaubt eine effiziente Durchführung von Analysen und Synthesen, Vergleiche können einfach strukturell realisiert werden. Die Eindeutigkeit der Repräsentationen ist durch die Wohlgeformtheitsbedingungen sichergestellt.

Das Feld `AffPos` enthält für jedes Symbol der Grammatik die zugehörigen Affixpositionen. Es handelt sich um ein global deklariertes Feld, welches als Schnittstelle zwischen den Regeln verwendet wird. Für die Affixparameter der Symbolvorkommen in den Regeln wird kein zusätzlicher Speicher reserviert, da die Parameterübergabe über die Feldeinträge des zugehörigen Symbols im Feld `AffPos` realisiert wird. Der Beginn der zu einem Symbol gehörenden Affixpositionen im Feld `AffPos` ist durch eine Konstante `S`*`n`* markiert, sie dient zur Vereinfachung der Indizierung.

```
VAR
    Heap: OpenHeap;
    NextHeap: HeapType;
```

Um die in [DeWe] gefundenen Problemlösungen wiederverwenden zu können, wird auch in den hier generierten Evaluatoren eine hochsprachliche Nachbildung des Speichers, der *Heap*, implementiert. In diesem Feld werden die Knoten der Ableitungsbäume in aufeinanderfolgenden Einträgen abgelegt. Sei `Heap[V]` ein Knotenbezeichner, dann verweist `Heap[V+i]` auf den i-ten Unterbaum (ein Beispiel ist in der Abbildung 9-2 illustriert). Der Knotenbezeichner ist konzeptionell ein Paar aus Alternativnummer und Stelligkeit der Regel, mit der der Knoten markiert ist. Die Stelligkeit entspricht der Anzahl der Unterbäume. Um Speicherplatz zu sparen, werden diese Paare durch jeweils eine Zahl codiert. Die Konstante `arityConst` wird wie folgt für die Rekonstruktion der Komponenten verwendet:

```
     Stelligkeit = Knotenbezeichner DIV arityConst,
Alternativnummer = Knotenbezeichner MOD arityConst.
```

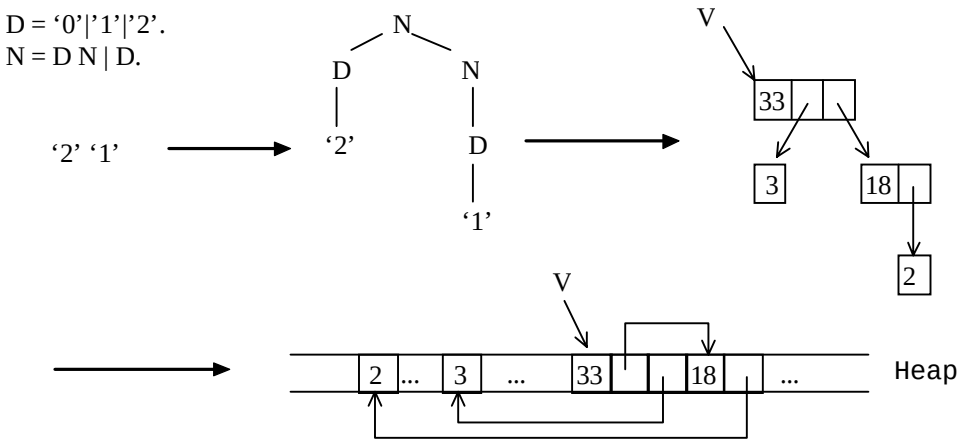


Abbildung 9-2: Repräsentation des Satzes '2' '1' zum Wertebereichssymbol N

9.1.2 Codeschemata

Für jede Evaluatorregel der Grammatik enthält der generierte Compiler eine *Evaluatorprozedur*, die durch die Nummer der Regel eindeutig identifiziert ist. Die Prozedur hat zwei Parameter: der erste verweist auf einen Knoten im dekorierten Ableitungsbaum, der zweite gibt die Nummer des aktuellen Besuchs an. Anschließend folgen die Deklaration der lokalen Variablen. Die Variablen `TreeAdr`, `VI` und `S` werden in jeder Prozedur deklariert und dienen zur verkürzten Indizierung globaler Felder bzw. zum Aufbau des dekorierten Ableitungsbaumes. Variablen der Form `V`*n* stellen temporäre Variablen für die Analyse dar.

```
PROCEDURE VisitRuleN(Symbol: LONGINT; VisitNo: INTEGER);

VAR TreeAdr, VI: IndexType; S: SemTreeEntry;
VAR V1,..,Vn: HeapType; (* temporäre Variablen der Analyse *)

BEGIN
    IF VisitNo = syntacticPart THEN
      Aufbau des dekorierten Ableitungsbaumes
    ELSE
      (* Berechung der Variablen zur verkürzten Indizierung *)
      TreeAdr := SemTree[Symbol].Adr;
      VI := SemTree[Symbol].VarInd;

      (* Umsetzung der Visit-Sequenz der Regel N *)
      CASE VisitNo OF
      1:
        Umsetzung des ersten Planes VS^1_N der Visit-Sequenz der Regel N
        ...
        Visit(TreeAdr+i, m);
        ...
        CheckP(...);
        ...
      | 2:
        Umsetzung des 2-ten Planes VS^2_N der Visit-Sequenz der Regel N
      | n:
        Umsetzung des n-ten Planes VS^n_N der Visit-Sequenz der Regel N
      END
    END
END VisitRuleN;
```

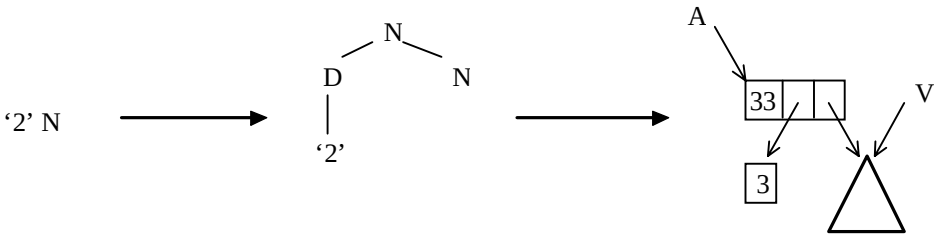
Der Rumpf der Prozedur ist in zwei Teile unterteilt. Im ersten wird die Datenstruktur des dekorierten Ableitungsbaumes aufgebaut. Der zweite Teil enthält eine Ablaufsteuerung, die der Visit-Sequenz der Evaluatorregel entspricht. Eine CASE-Anweisung unterscheidet die verschiedenen möglichen Besuche des Knotens voneinander. Jeder Fall der CASE-Anweisung nimmt einen Plan der Visit-Sequenz einer Regel auf. Der Besuch eines Sohnes wird durch den Aufruf der Prozedur `Visit` ausgelöst. Die Ausführung eines Prädikates wird durch Prozeduren der Form `Check`*P* veranlaßt. Bei jedem Besuch müssen zu Anfang die Variablen zur verkürzten Indizierung globaler Felder berechnet werden.

```
PROCEDURE Visit(Symbol: LONGINT; VisitNo: INTEGER);
BEGIN
    CASE SemTree[Symbol].Rule OF
        0: VisitRule0(Symbol, VisitNo);
      | 1: VisitRule1(Symbol, VisitNo);
      | 2: VisitRule2(Symbol, VisitNo);
      ...
      | k: VisitRulek(Symbol, VisitNo);
    END
END Visit;
```

Die Prozedur `Visit` ist notwendig, da anhand der Regel, mit der ein Knoten markiert ist, entschieden wird, welche Evaluatorprozedur aufgerufen werden muß. Dies ist erst zur Laufzeit möglich.

9.1.3 Synthesen

Bei einer Synthese wird für die Instanz eines Affixparameters ihr Wert berechnet. Dies erfolgt mit Hilfe der Affixe der aktuellen Regel, die entsprechend der Affixform des Affixparameters angeordnet werden. Die in der Affixform vorkommenden Affixe sind Platzhalter für synthetisierte Unterbäume. Die SOAG-Eigenschaft der Grammatik stellt sicher, daß diese bereits vollständig berechnet sind. Durch Anlegen von neuen Knoten wird aus diesen Unterbäumen schrittweise ein neuer Baum konstruiert (ein Beispiel dafür ist in Abbildung 9-3 zu sehen).

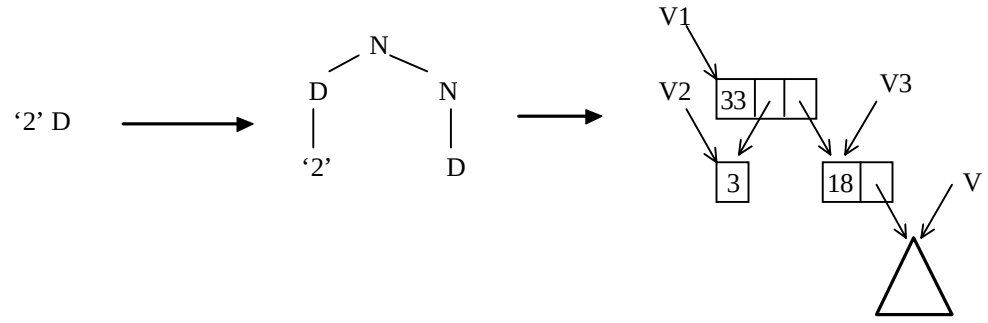



```
A := NextHeap;
Heap[NextHeap] := 33;
Heap[NextHeap + 1] := NextHeap + 3;
Heap[NextHeap + 3] := 3; Heap[NextHeap + 2] := V;
INC(NextHeap, 4);
```

Abbildung 9-3: Synthese des Satzes '2' N zum Wertebereichssymbol N

9.1.4 Analysen

Bei einer Analyse wird der Wert einer Affixparameterinstanz mit der durch die Affixform des Affixparameters vorgegebenen Struktur verglichen. Da dieser Wert als Ableitungsbaum vorliegt, werden die Knotenbezeichner während einer preorder-Traversierung zur Laufzeit des Compilers überprüft. Entspricht die Baumstruktur der Affixform, so werden für die darin vorkommenden Affixe die zugehörigen Unterbäume bestimmt. Um beim Zugriff auf die Unterbäume geschachtelte Indizierungen zu vermeiden, werden temporäre Variablen zur Elimination gemeinsamer Teilausdrücke verwendet (ein Beispiel ist in Abbildung 9-4 zu sehen). Dies optimiert zusätzlich die Laufzeit des Compilers.




```
V1 := A;
IF Heap[V1] # 33 THEN Error END; V2 := Heap[V1 + 1];
IF Heap[V2] # 3 THEN Error END; V3 := Heap[V1 + 2];
IF Heap[V3] # 18 THEN Error END;
V := Heap[V3 + 1];
```

Abbildung 9-4: Analyse des Satzes '2' D zum Wertebereichsymbol N

9.1.5 Vergleiche

Kommt ein Affix *M* mehrfach in Affixformen definierender Affixparameter einer Evaluatorregel vor, so müssen die Werte der Affixinstanzen aufgrund der Forderung nach konsistenter Ersetzung gleich sein. Dagegen müssen die Werte zweier Affixinstanzen *M* und *#M* verschieden sein. Das Vorkommen eines Affixes wird

immer mit dem definierenden Vorkommen des Affixes verglichen, dessen Inhalt in der Affixvariable festgehalten wurde.
Die Überprüfung wird einfach durch den Vergleich der Ableitungsbäume der Affixinstanz und der Affixvariable realisiert. Dazu wird die boolsche Funktion `Equal` aufgerufen, die eine rekursive Traversierung beider Ableitungsbäume vornimmt und bei ungleichen Verweisen abbricht. Um zu überprüfen, ob zwei Bäume verschieden sind, wird das Resultat negiert. Ein Aufruf dieser Prozedur erfolgt während der Analyse, direkt nach Bestimmung des zugehörigen Unterbaums.

9.1.6 Prädikate

Prädikate können nur zum leeren Wort abgeleitet werden, sie tragen nichts zur kontextfreien Definitionen der Quellsprache bei. Sie dienen in der Spezifikation eines Compilers eher dazu, „Berechnungen“ zu isolieren. Aufgrund der absichtlich für Prädikate zugelassenen Mehrdeutigkeit müssen sie besonders behandelt werden. Zur Generierung der Prädikate wird die Prozedur `GenPredProcs` des Moduls `SLEAG` benutzt.

Bei der Evaluation eines Prädikates muß anhand der Parametrisierung eine Ableitung des leeren Wortes im allgemeinen durch Backtracking gefunden werden. Das Scheitern einer Analyse bzw. eines Vergleichs zeigt hier keinen Fehler sondern eine Sackgasse an, die Auswahl der passenden Alternative ist hier wesentlich. Dazu werden alle Alternativen der Reihe nach ausprobiert:

```
PROCEDURE P (formale Parameter): BOOLEAN;
...
BEGIN
    failed := TRUE;
    Code zu Überprüfung der ersten Alternative
    IF failed THEN
       Code zu Überprüfung der zweiten Alternative
       ...
    END;
    RETURN ~ failed
END P;
```

Für eine Alternative steuern die Analysen und Vergleiche sowie die Prozeduraufrufe der zugehörigen Prädikate den weiteren Kontrollfluß. Dies wird durch geschachtelte IF-Anweisungen angemessen formuliert. Dabei müssen die im folgenden Schema nicht ausformulierten Analysen, insbesondere die Überprüfung des einzelnen Knotenbezeichners, ebenfalls in geschachtelte IF-Anweisungen umgesetzt werden.

```
(* Beginn einer Alternative *)
IF Analyse der Eingabeparameter der linken Seite (evtl. Vergleiche)
   erfolgreich THEN
    ...
    Synthese der Eingabeparameter zu P'
    IF P'(aktuelle Parameter) THEN
       IF Analyse der Ausgabeparameter zu P' (evtl. Vergleiche)
          erfolgreich THEN
         ...
         Synthese der Ausgabeparameter der linken Seite
         (evtl. Transfer)
         failed := FALSE;
         ...
       END
    END;
    ...
END;
(* Ende der Alternative *)
```

Erst der Aufruf einer Prädikatprozedur aus einer Evaluatorprozedur führt beim Scheitern zu einer Fehlermeldung.

```
Synthese der Eingabeparameter zu P
IF ~ P (aktuelle Parameter) THEN Fehlerbehandlung END;
Analyse der Ausgabeparameter zu P (evtl. Vergleiche)
```

Bevor im generierten Compiler eine Prädikatprozedur aufgerufen wird, erfolgt die Synthese der Eingabeparameter. Nach Ausführung werden die Ausgabeparameter analysiert und eventuell Vergleiche

durchgeführt. Als Parameter von Prädikaten werden die zum Symbol des Prädikates gehörenden Affixpositionen im Feld `AffPos` verwendet. Ausgenommen davon sind Parameter, die direkt Affixvariablen ergeben.

9.1.7 Fehlerbehandlung

Analysen, Vergleiche und Prädikate können scheitern. Dies zeigt Kontextfehler an und verhindert eine korrekte Übersetzung. Ein Abbruch des generierten Compilers ist jedoch nicht akzeptabel, statt dessen soll der Programmablauf fortgesetzt werden, um weitere Kontextfehler finden und melden zu können. Dabei sollen Folgefehler unterdrückt werden.

Anders als bei scheiternden Vergleichen kann bei scheiternden Analysen der Programmablauf nicht ohne weiteres fortgeführt werden, da im allgemeinen bei der Überprüfung der Unterbäume den in der zugehörigen Affixform vorkommenden Affixen Werte zugewiesen werden. Diese Affixe müssen wie auch die Ausgabeparameter scheiternder Prädikate auf definierte Werte gesetzt werden.

Es bietet sich an, einen speziellen Fehlerknoten anzulegen, der die maximale Stelligkeit aller in der Meta-Grammatik vorkommenden Regeln aufweist. Jeder seiner Unterbäume verweist wiederum auf den Fehlerknoten. Ein Verweis auf diesen Knoten wird im Falle von Kontextfehlern als Fehlerwert verwendet. Folgefehler können dann anhand dieses Werte erkannt und unterdrückt werden. In der Implementierung bilden die ersten Einträge des Heaps den Fehlerknoten. Der Fehlerwert wird durch die Konstante `errVal` bezeichnet und verweist auf diesen Knoten.

Scheitert in einer Analyse die Überprüfung eines Knotenbezeichners, wird der Verweis auf den Knoten durch den Fehlerwert ersetzt. Bei Fortsetzung der Analyse ergibt sich dann aufgrund der Struktur des Fehlerknotens, der mit maximaler Stelligkeit konstruiert wurde, für alle Unterbäume ebenfalls der Fehlerwert. Damit werden auch alle temporären Variablen und Affixe auf diesen Wert gesetzt. Die Meldung von Folgefehlern wird unterdrückt, wenn bereits der Fehlerwert untersucht wird:

```
IF V # errVal THEN Fehlermeldung; V := errVal END;
```

Wenn ein Prädikat scheitert werden alle seine Ausgabeparameter auf den Fehlerwert gesetzt. Fehlermeldungen werden in diesem Fall nur dann ausgegeben, wenn keines der Eingabeparameter den Fehlerwert aufweist. Scheitert der Vergleich zweier Bäume, wird die Fehlermeldung unterdrückt, falls für mindestens einen der Fehlerwert vorliegt:

```
IF (V1 # errVal) & (V2 # errVal) THEN Fehlermeldung END;
```

Um die Größe des generierten Codes zu verringern, werden die Fehlerbehandlungen in den Prozeduren `AnalyseError`, `Eq`, `UnEq` sowie `Check`*P* für ein Prädikat *P* zusammengefaßt.

Die Texte der Fehlermeldungen werden vom Generator automatisch aus den in der Spezifikation vorkommenden Bezeichnern erstellt. Zusammen mit der Fehlermeldung wird eine Position im Eingabetext ausgegeben. Analyse-Fehlermeldungen zum Hyper-Nichtterminal *N* haben die Form „analysis in *N* failed“. Für scheiternde Vergleiche wird der Name des Affixes *M* mit dem des zugehörigen Hyper-Nichtterminals zur Fehlermeldung „*M* failed in *N*“ verbunden. Scheitert ein Prädikat *P*, wird die Fehlermeldung „predicate *P* failed“ ausgegeben.

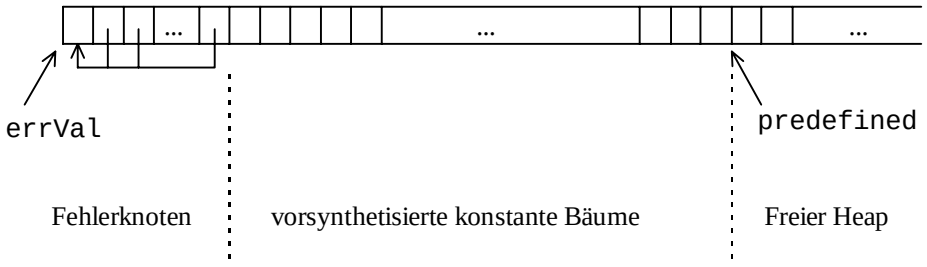


Abbildung 9-5: Erläuterung der Speicherstruktur in den generierten Evaluatoren

9.1.8 Optimierung terminaler Affixformen

Terminale Affixformen sind konstant in dem Sinne, daß sie im Evaluator durch immer gleiche Bäume dargestellt werden. Eine Synthese muß prinzipiell nur einmal erfolgen; später genügt es, einen Verweis auf den Baum zu liefern. Ebenso muß für jede abschließende Regel (ohne Nichtterminale) ein Blatt nur einmal im Heap

vorliegen. Durch eine solche *Konstantenfaltung* kann der Speicherbedarf gesenkt werden, und natürlich entsteht ein kürzeres und schnelleres Programm. In der Implementierung werden initial alle Blätter und die Bäume terminaler Affixformen in einem reservierten Teil des Heaps vorsynthetisiert. Die Verweise sind dadurch bekannt und können in Synthesen benutzt werden. Das Ende dieses reservierten Bereichs wird, wie in Abbildung 9-5 dargestellt, von der Konstanten `predefined` markiert. Das Beispiel einer Synthese aus Abbildung 9-3 verändert sich dadurch zu:

```
(* Synthese des Satzes '2'N zum Wertebereichssymbol N *)
A := NextHeap;
Heap[NextHeap] := 33;
Heap[NextHeap + 1] := 4;
Heap[NextHeap + 2] := V;
INC(NextHeap, 3);
```

In Analysen können terminale Affixformen im allgemeinen nicht genutzt werden, da die zu analysierenden Bäume auf vielfältige Art und Weise entstanden sein können. Da Blätter jedoch genau einmal im Heap vorliegen, kann die Überprüfung des Knotenbezeichners durch eine Überprüfung des Verweises abgekürzt werden. Das Beispiel aus Abbildung 9-4 verändert sich dadurch zu:

```
(* Analyse des Satzes '2'D zum Wertebereichssymbol N *)
V1 := A;
IF Heap[V1] # 33 THEN Fehlerbehandlung END; V2 := Heap[V1 + 1];
IF V2 # 4 (* Verweis auf ein Blatt *) THEN Fehlerbehandlung END;
V3 := Heap[V1 + 2];
IF Heap[V3] # 18 THEN Fehlerbehandlung END;
V := Heap[V3 + 1];
```

9.1.9 Freispeicherverwaltung

Da Speicherplatz für Knoten zwar bei Synthesen explizit angefordert, aber nie explizit freigegeben wird und da potentiell ein großer Teil des Heaps zur Darstellung von Zwischenergebnissen mit temporärer Bedeutung verwendet wird, soll der Evaluator wahlweise um eine Freispeicherverwaltung ergänzt werden können. Dabei kommt im generierten Evaluator ein Referenzzähler-Verfahren zur Anwendung, bei dem die in temporären Variablen vorliegenden Verweise auf die Wurzeln von Bäumen zur Laufzeit mitprotokolliert werden, was eine sofortige Wiederbenutzung nicht mehr referenzierter Knoten ermöglicht.

Die Referenz auf einen Baum wird durch die Erhöhung des Zählers des Wurzelknotens ausgedrückt. Durch Kollabierung entstehen mehrfache Referenzen auf Teilbäume. Verliert die letzte Referenz ihre Gültigkeit, erfolgt eine Freigabe des Wurzelknotens sowie aller Knoten seiner Unterbäume, auf die keine weitere Referenz existiert, indem sie in Freispeicherlisten eingetragen werden. Angeforderter Heap-Speicher wird dann vorrangig diesen Listen entnommen.

Neue Referenzen entstehen bei der Auswertung der Instanzen applizierender Affixparameter entsprechend ihrer Affixform. Sie ergeben sich aus den Verweisen auf die durch Affixe in der Affixform beschriebenen Unterbäume bei Synthese der Ableitungsbäume und aus der Zuweisung der synthetisierten Bäume an die Affixparameter. Erst mit dem letzten Besuch einer Regel verliert die Referenz auf einen Baum, der den Parameterwert einer definierenden Position darstellt, ihre Gültigkeit und wird freigegeben.

Für den Fehlerknoten und für vorsynthetisierte Bäume terminaler Affixformen funktioniert das Verfahren entsprechend. Beim Setzen des Fehlerwertes verliert der fehlerhafte (Unter-)Baum seine Referenz, der Fehlerknoten erhält dafür eine weitere, um eine Freigabe ohne Sonderbehandlung zu verhindern. Durch eine Synthese entsteht eine zusätzliche Referenz auf einen vorsynthetisierten Baum.

In der Implementierung wird der Knotenbezeichner konzeptionell um den Referenzzähler zu einem Tripel erweitert. Die zusätzliche Konstante `refConst` wird wie folgt für die Rekonstruktion der Komponenten verwendet:

```
    Referenzzähler = Knotenbezeichner DIV refConst,
       Stelligkeit = (Knotenbezeichner MOD refConst) DIV arityConst,
Alternativennummer = Knotenbezeichner MOD arityConst.
```

Im Referenzzähler wird lediglich die Anzahl der zusätzlichen Verweise auf einen Knoten gezählt.

Besonderheiten ergeben sich für die Parameter eines Prädikataufrufs. Ursprünglich sind die Prädikatprozeduren für Ein-Pass-Compiler konzipiert worden, deren Referenzzählung auch Prozedurparameter als Referenzen

einbezieht und somit auch die Freigabe von Parameterinhalten umfaßt. Handelt es sich beim Prozedurparameter um eine Affixvariable, so verliert der Ableitungsbaum, auf den sie verweist, durch den Prädikataufruf eine Referenzierung. Um die Referenzzählung innerhalb des Prädikates auszugleichen, werden alle Affixvariablen, die im Prädikataufruf als Eingabeparameter verwendet werden, um eine Referenzierung erhöht. Dadurch wird sichergestellt, daß der Inhalt der Affixvariablen nach dem Prädikataufruf noch nicht freigegeben ist. Nach einem Prädikataufruf sind die Referenzzähler aller Ableitungsbäume, auf die die Rückgabevariablen verweisen (Affixvariablen ausgenommen), um eine Referenz zu hoch. Deshalb müssen diese Variablen nach erfolgter Analyse explizit freigegeben werden.

Für jede Knotenstelligkeit wird eine eigene Freispeicherliste angelegt. Freigegebene Knoten werden gemäß ihrer Stelligkeit in diesen Listen verwaltet. Die Einträge des Feldes `FreeList` verweisen auf den jeweils ersten Knoten einer solchen Liste. Die Verkettung ist in dem jeweils ersten Eintrag der freigegebenen Knoten realisiert.

In Synthesen erfolgt nun die Anforderung von Speicherplatz knotenweise durch die Prozedur `GetHeap`. Das Löschen einer Referenz sowie eine eventuelle Freigabe des Speicherplatzes erfolgt durch die Prozedur `FreeHeap`.

Durch Anwendung des Referenzzähler-Verfahrens und unter der Voraussetzung, daß keine Optimierung terminaler Affixformen erfolgt und die Synthese nicht in einer Prädikatprozedur vorkommt, verändert sich das Beispiel der Synthese sich aus Abbildung 9-3 zu:

```
(* Synthese des Satzes '2'N zum Wertebereichssymbol N *)
GetHeap(2, V1); A := V1; Heap[V1] := 33;
GetHeap(0, V2);
Heap[V1 + 1] := V2; INC(Heap[V2], refConst);
Heap[V2] := 3;
Heap[V1 + 2] := V; INC(Heap[V, refConst);
```

Komplementäre Referenzzähler-Operationen werden aufgelöst und damit der Code weiterhin optimiert.

Das Beispiel aus Abbildung 9-4 verändert sich unter der Voraussetzung, daß keine Optimierung terminaler Affixformen erfolgt und die Analyse in einer Evaluatorprozedur vorkommt, zu:

```
(* Analyse des Satzes '2'D zum Wertebereichssymbol N *)
V1 := A;
IF Heap[V1] MOD refConst # 33 THEN Fehlerbehandlung END;
V2 := Heap[V1 + 1];
IF Heap[V2] MOD refConst # 3 THEN Fehlerbehandlung END;
V3 := Heap[V1 + 2];
IF Heap[V3] MOD refConst # 18 THEN Fehlerbehandlung END;
V := Heap[V3 + 1];
```

9.1.10 Optimierung der Affixvariablenspeicherung

Die Optimierung der Affixvariablenspeicherung ist in zwei Formen möglich. Entweder wird die Affixvariable als globale Variable oder als Kellerspeicher verwendet. Im Falle des Kellerspeichers werden die drei bekannten Prozeduren `Push` zur Wertespeicherung, `Top` zur Rückgabe und `Pop` zur Freigabe des obersten Kellerelementes benutzt. Nur Affixvariablen mit besonderen Eigenschaften können in ihrer Speicherung optimiert werden. Diese Eigenschaften werden durch den Modul `SOAGOptimizer` berechnet und sind im Kapitel 7 „Optimierung der Affixvariablenspeicherung“ näher erläutert.

Unter der Annahme, die Affixvariable *V* zum Affix *N* besäße die geforderten Eigenschaften, so daß ihre Speicherung durch den Kellerspeicher `Stack1` optimiert werden kann, verändert sich das Codefragment des Beispiels einer Synthese aus Abbildung 9-3 unter der Voraussetzung, daß keine Optimierung terminaler Affixformen und keine Referenzzählung stattfindet, zu:

```
(* Synthese des Satzes '2'N zum Wertebereichssymbol N *)
A := NextHeap;
Heap[NextHeap] := 33;
Heap[NextHeap + 1] := NextHeap + 3;
Heap[NextHeap + 3] := 3;
Heap[NextHeap + 2] := Stacks.Top(Stack1);
INC(NextHeap, 4);
```

Das Beispiel einer Analyse aus Abbildung 9-4 adaptiert unter den gleichen Voraussetzungen zu:

```
(* Analyse des Satzes '2'D zum Wertebereichssymbol N *)
V1 := A;
IF Heap[V1] # 33 THEN Error END; V2 := Heap[V1 + 1];
IF Heap[V2] # 3 THEN Error END; V3 := Heap[V1 + 2];
IF Heap[V3] # 18 THEN Error END;
Stacks.Push(Stack1, Heap[V3 + 1]);
```

Die Operation `Pop` wird immer dann ausgeführt, wenn die Lebensdauer einer Affixvariable abgelaufen ist. D.h., wenn es keinen Affixparameter mehr gibt, bei dessen Synthese oder Vergleich die Affixvariable in der verbleibenden Visit-Sequenz der Regel angewendet wird.

Wird der Wert einer Affixvariable, die als Kellerspeicher implementiert ist, durch den Aufruf einer Prädikatprozedur definiert, so muß dieser Aufruf zunächst mit der zum Prädikatsymbol zugehörigen globalen Variable aus dem Feld `AffPos` parametrisiert werden. Nach dem Prädikataufruf wird der Inhalt der benutzten Schnittstellenvariable durch eine `Push`-Operationen auf den Kellerspeicher gelegt.

9.1.11 Speicherung von Positionsangaben

Der vom Parser aufgebaute kontextfreie Ableitungsbaum ist durch das parallel liegende Feld `PosTree` um Positionsangaben angereichert. Während des Aufbaus des dekorierten Ableitungsbaumes im Evaluator werden die notwendigen Positionsangaben in das Datenelement `Pos` der Baum-Datenstruktur übertragen. Auf das Feld `PosTree` wird danach nicht mehr zugegriffen, es wird nicht mehr benötigt.

9.1.12 Ausgabe der Übersetzung

Im generierten Compiler stellen die vom Evaluator berechneten Werte der Ausgabeparameter des Startsymbols die Übersetzung dar. Der Modul `EmitGen` generiert Code, der diese Übersetzung ausgibt. Intern wird die Übersetzung vom Evaluator durch einen Ableitungsbaum zur Meta-Grammatik dargestellt. Die Ausgabe erfolgt daher durch eine Traversierung dieses Ableitungsbaums.

Dabei ist zu beachten, daß die Ausgabe von Alternativen eines Nichtterminals, das mit einer Token-Markierung versehen ist (einem sogenannten *Token*) so wie deren Unterbäume bündig erfolgen soll, d.h. ohne die Ausgabe eines Trennzeichens zwischen den einzelnen Terminalen; im anderen Fall jedoch soll nach jedem Terminal ein Trennzeichen ausgegeben werden. Dadurch ergibt sich die Situation, daß ein Nichtterminal ein *Subtoken* sein kann, d.h. sowohl in einem Unterbaum eines Tokens als auch in einem nicht von einem Token ausgehenden Unterbaum vorkommen kann. Bei der Ausgabe von Alternativen eines Subtokens müssen also abhängig vom Kontext im Ableitungsbaum Trennzeichen ausgegeben bzw. nicht ausgegeben werden.

Die Meta-Nichtterminale lassen sich jetzt nach dieser Ausgabeeigenschaft in zwei Mengen einteilen. In der Menge `Type3` sollen sich die Token sowie die von Token erreichbaren Nichtterminale befinden. In der Menge `Type2` sollen sich die vom Startsymbol der Meta-Grammatik ausgehend erreichbaren Nichtterminale befinden, jedoch ohne die Token und ohne die nur über Token erreichbaren Nichtterminale. Die Subtoken bilden also genau die Schnittmenge dieser beiden Mengen.

Die Ausgabe erfolgt in der Implementierung durch Prozeduren, in denen eine Fallunterscheidung über Alternativen eines Nichtterminals stattfindet. Für die Ausgabe mit und ohne Trennzeichen werden getrennte Prozeduren generiert. Für Subtoken werden insbesondere zwei Prozeduren generiert. In den Prozeduren wird zusätzlich die Größe des ausgegebenen Baums berechnet, d.h. die Anzahl der benötigten Heapeinträge bei einer nichtkollabierten Speicherweise. Die generierten Prozeduren lassen sich durch folgendes Codeschema darstellen:

```
(* Meta-Nichtterminal M *)
PROCEDURE EmitMType2/3 (Ptr: HeapType);
BEGIN
    CASE Heap[Ptr] MOD arityConst OF
      | 1: Code für die erste Alternative
         .
         .
      | n: Code für die n-te Alternative
    END
END EmitMType2/3;

(* Ausgabe von Meta-Terminal T *)
IO.WriteText(Out, T); (evtl. IO.Write(Trennzeichen); )

(* Prozeduraufruf für Meta-Nichtterminal N *)
EmitNType2/3(Heap[Ptr + i]);
```

Die Generierung der Ausgabeprozeduren erfolgt durch die Prozedur `GenEmitProc`. In dieser erfolgt die Berechnung der Mengen `Type2` und `Type3` durch Traversierung der Meta-Grammatik.

9.2 Der Generator

9.2.1 Vergabe von Variablennamen

Lokale Variablen werden in den Evaluatorprozeduren des generierten Compilers für die Analyse definierender Affixparameter verwendet, sie nehmen die Verweise der zu analysierenden Unterbäume auf. Dabei werden die Variableninhalte nur während der Analyse benötigt und haben danach keine Bedeutung mehr. Da die einzelnen Analysen unabhängig voneinander ablaufen, können die gleichen Variablen in allen Analysen einer Evaluatorprozedur benutzt werden. Die Anzahl der Variablen orientiert sich dabei an der umfangreichsten Analyse der Evaluatorprozedur. Weitere lokale Variablen werden für applizierende Affixparameter im Falle der Benutzung des Referenzzähler-Verfahrens für die Speicheralloziierung benötigt.

```
VAR
    LocalVars: SOAG.OpenInteger;
    NodeName: SOAG.OpenInteger;
```

Die lokalen Variablen einer Analyse sind aufsteigend numeriert. Der Name einer Variablen ergibt sich aus ihrer Nummer ergänzt um das Präfix `V`. Zur Deklaration der Variablen muß lediglich deren maximale Anzahl in der umfangreichsten Analyse bekannt sein, diese wird für jede Regel im Feld `LocalVars` abgespeichert. Für jeden Affixbaumknoten einer Analyse wird die Nummer der lokalen Variable, die den Verweis des Knotens aufnimmt, im Feld `NodeName` festgehalten, das parallel zu `EAG.Node` liegt. Die Berechnung der Variablennamen in den oben genannten Datenstrukturen erfolgt durch eine die Evaluation vorwegnehmende Traversierung aller Affixformen in der Prozedur `ComputeNodeNames`.

9.2.2 Generierung der Evaluatorprozeduren

Die Prozedur `GenVisitRule` generiert für jede Regel eine Evaluatorprozedur analog den in Abschnitt 9.1.2 angegebenen Codeschemata. Nach der Ausgabe des Prozedurkopfes folgt die Deklaration der lokalen Variablen durch die Prozedur `GenVarDecls`. Wie schon erwähnt teilt sich der Rumpf der Evaluatorprozedur in zwei Teile. Im ersten wird der Aufbau des dekorierten Ableitungsbaumes codiert. Es folgt die Generierung der Speicheralloziierung für Affixvariablen. Anschließend wird das Anlegen des Knotens und die Berechnung der Regelnummer, mit der er markiert werden soll, ausgegeben. Am Schluß diesen Teils wird für alle Grundnichterminale der aktuellen Regel ein Besuch generiert, damit der Aufbau des dekorierten Ableitungsbaumes rekursiv fortgesetzt werden kann.

Im zweiten Teil wird die Visit-Sequenz einer Regel in entsprechende Oberon-Anweisungen umgesetzt. Falls der Eintrag den Besuch eines Sohnes erfordert, so wird dieser durch die Prozedur `GenVisitCall` generiert. Besteht der Eintrag aus einem Prädikataufruf, so erzeugt diesen die Prozedur `GenPredCall`. Vor dem Prädikataufruf wird durch die Prozedur `GenPredPos` eine Positionszuweisung für die Fehlerbehandlung im Falle des Scheiterns des Prädikates codiert. Dafür wird die Position des vorhergehenden Grundnichtterminals der aktuellen Regel verwendet.

Für alle eben genannten Einträge muß vor dem Besuch bzw. Aufruf die Synthese der applizierenden Affixparameter und danach die Analyse der definierenden Affixparameter des aktuellen Besuches durchgeführt werden. Die Erzeugung dieser Codebestandteile übernehmen die Prozeduren `GenSynPred` bzw. `GenAnalPred`. Damit nur die notwendigen Affixparameter untersucht werden, wird den beiden Prozeduren die entsprechende Besuchsnummer übergeben. Ein Prädikataufruf ist an keine Besuchsnummer gebunden, deshalb werden die Prozeduren in diesem Fall mit -1 parametrisiert.

Jeder LEAVE-Eintrag in der Visit-Sequenz einer Regel zeigt das Ende eines Plans an. In diesem Fall werden die den Besuch abschließenden Synthesen generiert, und nach der Ausgabe der nächsten Besuchsnummer erfolgt die Generierung der Analysen, mit denen der neue Besuch beginnt.

Die Prozedur `GenVisit` erzeugt die im Abschnitt 9.1.2 erläuterte `Visit`-Prozedur.

9.2.3 Berechnung der anonymen Prädikate

Die Analysen und Synthesen der Affixparameterinstanzen eines generierten Compilers sind unmittelbare Bestandteile (Inline-Code) jeder implementierten Visit-Sequenz. Sie werden nicht über den Umweg eines Prozeduraufrufes ausgeführt, aus diesem Grund werden sie auch *anonyme* Prädikate genannt.

Die Prozedur `GenSynPred` erzeugt für alle applizierenden Affixparameter in Abhängigkeit der übergebenen Besuchsnummer den Code der Synthese des Instanzwertes. In der Synthese wird aus den Affixvariablenwerten der in der Affixform vorkommenden Affixe ein Ableitungsbaum konstruiert, dessen Struktur der Affixform des Affixparameters entspricht. Da die Affixformen zur Generierungszeit ebenfalls als Ableitungsbäume vorliegen, erfolgt die Erzeugung des Synthesecodes in der Prozedur `GenSynTraverse` durch eine rekursive Traversierung dieses Ableitungsbaumes, während derer die entsprechenden Syntheseaktionen ausgegeben werden. Als Referenzparameter wird in dieser Prozedur der nächste freie Eintrag im Heap als Offset mitgeführt, so daß nach vollständiger Traversierung eines Unterbaumes der Code für das Anlegen des nächsten Unterbaums generiert werden kann. Im Falle der Verwendung des Referenzzähler-Verfahrens ist dieser Referenzparameter nicht notwendig, da neuer Speicher im generierten Compiler über die Prozedur `GetHeap` angefordert wird. Deshalb erfolgt in diesem Fall die Traversierung durch die Prozedur `GenSynTraverseRefCnt`.

Der Zugriff auf eine Affixvariable wird durch die Prozedur `GenAffix` generiert. Sie unterscheidet automatisch, ob eine Affixvariable als globale Variable, Kellerspeicher oder Bestandteil des Feldes `Var` ausgegeben werden soll. Analog verhält sich die Prozedur `GenAffixAssign`, die den linken Teil einer Zuweisung an eine Affixvariable generiert. Sie wird stets in Kombination mit der Prozedur `GenClose` aufgerufen. Wird eine Affixvariable als Kellerspeicher implementiert, so generiert `GenClose` die schließende Klammer der `Push`-Operation. Die Information, ob eine schließende Klammer generiert werden soll, wird durch die globale boolsche Variable `Close` übermittelt. Die Anweisungen im Generator

```
GenAffixAssign(V); GenAffPos(S, AN); GenClose;
```

ergeben im generierten Compiler die Codesequenz

```
Stacks.Push(Stack1, AffPos[Sn+AN]),
```

falls `V` als Kellerspeicher `Stack1` implementiert wird, und

```
Var[V] := AffPos[Sn+AN]
```

falls `V` nicht optimierbar ist.

Die Prozedur `GenAnalPred` generiert für alle definierenden Affixparameter in Abhängigkeit der parametrisierten Besuchsnummer den Code für die Analyse aller in einem Besuch auszuwertenden Affixparameter. Die Codierung erfolgt wiederum während einer Traversierung des Ableitungsbaumes der zu einem Affixparameter gehörenden Affixform in der Prozedur `GenAnalTraverse`.

Die durch die Forderung nach konsistenter Ersetzung notwendigen Vergleiche werden durch die beiden Generierungsprozeduren `GenEqualPred` und `GenUnequalPred` direkt in den Analysecode integriert.

Ist die Lebenszeit einer Affixvariable abgelaufen, so wird durch die Prozedur `GenFreeAffix` ihre Freigabe generiert. Ist sie als Kellerspeicher optimiert, so generiert die Prozedur `GenPopAffix` die Freigabe des obersten Kellerelementes.

9.2.4 Datenstrukturen für die Generierung

Für die Generierung eines Compilers werden des weiteren folgende Datenstrukturen benötigt:

```
VAR
    AffixOffset: SOAG.OpenInteger;
    AffixAppls: SOAG.OpenInteger;
    AffixVarCount: SOAG.OpenInteger;
```

Das Feld `AffixAppls` dient während der Generierung zur Berechnung der Lebensdauer von Affixvariablen. Dazu wird es mit den Werten des parallel-liegenden Feldes `SOAG.AffixApplCnt` initialisiert, das die Anzahl der Applikationen des zur Affixvariable gehörenden Affixes enthält. Während der Generierung der Evaluatorprozeduren wird der Wert in `AffixAppls[`*A*`]` für jede Applikation der Affixvariable *A* um eins dekrementiert. Ergibt sich nach einer Applikation einer Affixvariable der Wert Null, so ist die Lebenszeit der Affixvariable abgelaufen und sie kann freigegeben werden. Ist die Affixvariable als Kellerspeicher implementiert, so kann der oberste Wert des Kellerspeichers entfernt werden.

`AffixOffset` ist eine Datenstrukturerweiterung des Feldes `EAG.Var` und nimmt für jedes Affix die Offset-Nummer der entsprechenden Variable im Feld `Var` des generierten Compilers auf. Ist eine Affixvariable optimiert, so enthält ihr Feldeintrag den Wert `optimizedStorage`. Wird der Wert einer Affixvariable in keiner Synthese und in keinem Vergleich verwendet, hat sie also keine Applikationen, so muß ihr Wert in keiner Variable vorgehalten werden. Der Eintrag in `AffixOffset` hat in diesem Fall den Wert `notApplied`. Wertzuweisungen an diese Affixvariable werden in der Generierung unterdrückt. Die Prozedur `ComputeAffixOffset` berechnet für eine Regel den Inhalt des Feldes `AffixOffset`.

Das Feld `AffixVarCount` nimmt zu jeder Evaluatorregel die Anzahl der zu generierenden Affixvariablen im Feld `Var` des generierten Compilers auf.

```
VAR
    SubTreeOffset: SOAG.OpenInteger;
```

Da im dekorierten Ableitungsbaum im Gegensatz zu den Evaluatorregeln keine Prädikate vorkommen, müssen die Offsets der Grundnichtterminale im Baum im Feld `SubTreeOffset` berechnet werden. Diese Feld liegt parallel zu `SOAG.SymOcc`.

```
VAR
    FirstRule: SOAG.OpenInteger;
```

Der vom Parser gelieferte kontextfreie Ableitungsbaum enthält in jedem seiner Knoten die Nummer der auf ihn angewendeten Regel. Dabei werden die jeweils zu einem Nichtterminal gehörenden Regeln stets bei eins beginnend durchnumeriert, sie bilden damit eine Art Offset zum Nichtterminal. Im dekorierten Ableitungsbaum werden die Knoten jedoch mit den Regelnummern der SOAG-Datenstruktur markiert, die durchgehend numeriert sind. Da Regeln zu einem Nichtterminal zusammenhängend und in derselben Reihenfolge wie in der EAG-Datenstruktur vorliegen, kann aus dem Regel-Offset eines Nichtterminals im kontextfreien Ableitungsbaum und der Nummer der erste Regel des Nichtterminals in der SOAG-Datenstruktur die zugehörige Regelnummer berechnet werden. Dazu wird die Nummer der ersten Regel jedes Symbols im Feld `FirstRule` berechnet, das parallel zu `SOAG.Sym` liegt. Dieses Feld dient zur Berechnung der Regelnummer, mit der jeder Knoten des dekorierten Ableitungsbaumes markiert ist.

```
VAR
    UseConst: BOOLEAN;
    UseRefCnt: BOOLEAN;
    Optimize: BOOLEAN;
```

Die Schalter `UseConst`, `UseRefCnt` und `Optimize` steuern die Generierung eines Compilers unter Verwendung der Konstantenfaltung, des Referenzzähler-Verfahrens bzw. der Optimierung der Affixvariablenspeicherung. Der Inhalt der oben beschriebenen Datenstrukturen wird in der Prozedur `Init` berechnet.

Die Prozedur `GenerateModule` nimmt die eigentliche textuelle Generierung der Compiler-Quelldatei vor. Dabei wird die konstante Datei `eSOAG.Fix` eingelesen und sukzessive erweitert. Sie enthält an den zu erweiternden Positionen eine Markierung, bis zu der die Operation `InclFix` den Inhalt der Datei jeweils einliest. Danach werden die erforderlichen Erweiterungen eingetragen. Die Prozedur `Generate` steuert den gesamten Generierungsprozeß und wird auch exportiert.

9.3 Implementierungen

9.3.1 eSOAGGen.Mod

```
MODULE eSOAGGen; (* dk 3.03 14.03.98 *)

IMPORT EAG := eEAG, SOAG := eSOAG, IO := eIO, Sets := eSets,
      SOAGPartition := eSOAGPartition, SOAGVisitSeq := eSOAGVisitSeq,
      SLEAGGen := eSLEAGGen, EmitGen := eEmitGen, SOAGOptimizer := eSOAGOptimizer,
      Protocol := eSOAGProtocol;

CONST
  cTab = 1; (* Tabulatorabstand *)
  firstAffixOffset = 0;
  (* Konstanten fuer AffixOffset[] *)
  optimizedStorage = -1;
  notApplied = -2;

VAR
  UseConst: BOOLEAN; (* Schalter fuer Konstantenfaltung *)
  UseRefCnt: BOOLEAN; (* Schalter fuer Referenzzaehler-Verfahren *)
  Optimize: BOOLEAN; (* Schalter fuer Optimierung der Affixspeicherung *)
  LocalVars: SOAG.OpenInteger; (* parallel zu SOAG.Rule[] *)
  NodeName: SOAG.OpenInteger; (* parallel zu EAG.NodeBuf[] *)
  AffixOffset: SOAG.OpenInteger; (* parallel zu EAG.Var[] *)
  AffixVarCount: SOAG.OpenInteger; (* parallel zu SOAG.Rule[] *)
  SubTreeOffset: SOAG.OpenInteger; (* parallel zu SOAG.SymOcc[] *)
  FirstRule: SOAG.OpenInteger; (* parallel zu SOAG.Sym[] *)
  AffixAppls: SOAG.OpenInteger; (* parallel zu EAG.Var *)

  Out*: IO.TextOut; (* Deskriptor der Ausgabedatei *)
  Indent: INTEGER;  (* Einrueckungsgroesse fuer generierte Codezeilen *)
  Error, ShowMod, Close: BOOLEAN;

(*--------------- Berechnungen ---------------------------------------*)

PROCEDURE ComputeNodeNames(R:INTEGER);
(*IN: Regel
 OUT: -
 SEM: Berechnet im Feld NodeNames fuer alle Affixbaumanalysen der Regel die Namen
    der temp. Variablen fuer die Baumknoten;
    die maximale Variablenummer der Regel wird in LocalVars[] abgelegt *)
VAR
  Var, ProcVar, AP, AN, Node, SO, V, PBI, AP1, PBI1, V1: INTEGER;
  sameAffix: BOOLEAN;

  PROCEDURE Traverse(Node: INTEGER; VAR Var: INTEGER);
  (* IN: Knoten in NodeBuf[], Variablenname
   OUT: -
   SEM: Berechnet fuer jeden Knoten des Teilbaums NodeBuf[Node] die temp. Variable
      fuer die Baumanalyse oder -synthese*)
  VAR Node1, n, Arity: INTEGER;
  BEGIN
    Arity := EAG.MAlt[EAG.NodeBuf[Node]].Arity;
    INC(Var); NodeName[Node] := Var;
    FOR n := 1 TO Arity DO
      Node1 := EAG.NodeBuf[Node + n];
      IF Node1 > 0 THEN (* Knoten *)
        IF UseConst & (EAG.MAlt[EAG.NodeBuf[Node1]].Arity = 0) THEN
          INC(Var); NodeName[Node1] := Var
        ELSE Traverse(Node1, Var) END
      END
    END
  END Traverse;

BEGIN (* ComputeNodeNames *)
  (* moegliche temp. Variablen fuer definierenden Affixparameter
    zusaetzlich werden im Falle der Benutzung des Referenzzaehler-Verfahrens fuer alle
    appl. Affixparameter die temp. Variablen fuer die Speicherallozііerung berechnet *)
  LocalVars[R] := 0;
  FOR AP := SOAG.Rule[R].AffOcc.Beg TO SOAG.Rule[R].AffOcc.End DO
    PBI := SOAG.AffOcc[AP].ParamBufInd;
    IF EAG.ParamBuf[PBI].isDef OR UseRefCnt THEN Var := 0;
      Node := EAG.ParamBuf[PBI].Affixform;
      IF Node > 0 THEN (* Knoten *)
        IF UseConst & (EAG.MAlt[EAG.NodeBuf[Node]].Arity = 0) THEN
          INC(Var); NodeName[Node] := Var
        ELSE Traverse(Node,Var) END
      END;
      IF Var > LocalVars[R] THEN LocalVars[R] := Var END
    END
```

```
  END;

  (* temp. Variablen fuer die applizierenden Affixparameter von Praedikateaufrufen *)
  FOR SO := SOAG.Rule[R].SymOcc.Beg TO SOAG.Rule[R].SymOcc.End DO
    IF SOAG.IsPredNont(SO) THEN Var := 0; ProcVar := 0;
      FOR AP := SOAG.SymOcc[SO].AffOcc.Beg TO SOAG.SymOcc[SO].AffOcc.End DO
        PBI := SOAG.AffOcc[AP].ParamBufInd;
        Node := EAG.ParamBuf[PBI].Affixform;
        IF ~EAG.ParamBuf[PBI].isDef THEN
          IF (Node > 0) THEN INC(Var); NodeName[Node] := Var END
        END
      END;
      IF Var > LocalVars[R] THEN LocalVars[R] := Var END;
    END
  END
END ComputeNodeNames;

PROCEDURE GetCorrespondedAffPos(AP: INTEGER): INTEGER;
(*IN: Affixparameter
 OUT: Affixposition
 SEM: gibt die Affixposition zurueck, zu der der Affixparameter korrespondiert *)
VAR
  SO, AN: INTEGER;
BEGIN
  SO := SOAG.AffOcc[AP].SymOccInd;
  AN := AP - SOAG.SymOcc[SO].AffOcc.Beg;
  RETURN SOAG.Sym[SOAG.SymOcc[SO].SymInd].AffPos.Beg + AN
END GetCorrespondedAffPos;


PROCEDURE WrAffixAppls(R: INTEGER);
VAR
  Scope: EAG.ScopeDesc; EAGRule: EAG.Rule;
  V : INTEGER;
BEGIN
  IF SOAG.Rule[R] IS SOAG.OrdRule THEN
      Scope := SOAG.Rule[R](SOAG.OrdRule).Alt.Scope
  ELSE EAGRule := SOAG.Rule[R](SOAG.EmptyRule).Rule;
    IF EAGRule IS EAG.Opt THEN Scope := EAGRule(EAG.Opt).Scope
    ELSIF EAGRule IS EAG.Rep THEN Scope := EAGRule(EAG.Rep).Scope
    END
  END;
  FOR V := Scope.Beg TO Scope.End-1 DO
    EAG.WriteVar(IO.Msg, V); IO.WriteText(IO.Msg, ": "); IO.WriteInt(IO.Msg, AffixAppls[V]);
    IO.WriteLn(IO.Msg); IO.Update(IO.Msg)
  END
END WrAffixAppls;


PROCEDURE ComputeAffixOffset(R: INTEGER);
(*IN: Regel
 OUT: -
 SEM: berechnet im Feld AffixOffset[], das parallel zu EAG.Var liegt, den Offset der
    Affixvariablen im Feld Var[] des generierten Compilers;  alle nicht-applizierten
    Affixvariablen (AffixAppls[]=0) werden ausgelassen
 PRE: AffixAppls[] muss berechnet sein *)
VAR
  Scope: EAG.ScopeDesc; EAGRule: EAG.Rule;
  A, AP, Offset: INTEGER;
BEGIN
  IF SOAG.Rule[R] IS SOAG.OrdRule THEN
      Scope := SOAG.Rule[R](SOAG.OrdRule).Alt.Scope
  ELSE EAGRule := SOAG.Rule[R](SOAG.EmptyRule).Rule;
    IF EAGRule IS EAG.Opt THEN Scope := EAGRule(EAG.Opt).Scope
    ELSIF EAGRule IS EAG.Rep THEN Scope := EAGRule(EAG.Rep).Scope
    END
  END;
  Offset := firstAffixOffset;
  FOR A := Scope.Beg TO Scope.End-1 DO
    IF AffixAppls[A] > 0 THEN
      IF Optimize THEN
        AP := GetCorrespondedAffPos(SOAG.DefAffOcc[A]);
        IF SOAG.StorageName[AP] = 0 THEN
          AffixOffset[A] := Offset; INC(Offset)
        ELSE AffixOffset[A] := optimizedStorage END
      ELSE (* keine Optimierung *)
        AffixOffset[A] := Offset; INC(Offset)
      END
    ELSE
      AffixOffset[A] := notApplied
    END
  END;
```

```
  AffixVarCount[R] := Offset - firstAffixOffset
END ComputeAffixOffset;


PROCEDURE GetAffixCount(R: INTEGER): INTEGER;
(* SEM: liefert die echte Anzahl an Affixvariablen in der Regel; nur zur Information ueber die
      Optimierungleistung *)
VAR
  Scope: EAG.ScopeDesc; EAGRule: EAG.Rule;
BEGIN
  IF SOAG.Rule[R] IS SOAG.OrdRule THEN
      Scope := SOAG.Rule[R](SOAG.OrdRule).Alt.Scope
  ELSE EAGRule := SOAG.Rule[R](SOAG.EmptyRule).Rule;
    IF EAGRule IS EAG.Opt THEN Scope := EAGRule(EAG.Opt).Scope
    ELSIF EAGRule IS EAG.Rep THEN Scope := EAGRule(EAG.Rep).Scope
    END
  END;
  RETURN Scope.End - Scope.Beg
END GetAffixCount;


PROCEDURE HyperArity*() : INTEGER;
(*IN: -
 OUT: Hyper-Arity-Konstante
 SEM: Liefert die Arity-Konstante fuer den Ableitungsbaum, der durch den Parser erzeugt wird;
    muesste eigentlich von SLEAG geliefert werden (in SSweep wurde es auch intern definiert,
    deshalb wird es hier fuer spaetere Module exportiert) *)
VAR N, i, Max : INTEGER; A : EAG.Alt; Nonts : Sets.OpenSet;
BEGIN
  Sets.New(Nonts, EAG.NextHNont);
  Sets.Difference(Nonts, EAG.All, EAG.Pred);
  Max := 0;
  FOR N := EAG.firstHNont TO EAG.NextHNont - 1 DO
    IF Sets.In(Nonts, N) THEN
      A := EAG.HNont[N].Def.Sub; i := 0;
      REPEAT INC(i); A := A.Next UNTIL A = NIL;
      IF (EAG.HNont[N].Def IS EAG.Opt) OR (EAG.HNont[N].Def IS EAG.Rep) THEN INC(i) END;
      IF i > Max THEN Max := i END;
    END
  END;
  i := 1; WHILE i <= Max DO i := i * 2 END;
  RETURN i
END HyperArity;

PROCEDURE Init;
(* SEM: Initialisierung der Datenstrukturen des Moduls *)
VAR i, R, SO, S, Offset, len, POI: INTEGER;
BEGIN
  NEW(LocalVars, SOAG.NextRule);
  NEW(AffixVarCount, SOAG.NextRule);
  NEW(AffixOffset, EAG.NextVar);
  NEW(NodeName, EAG.NextNode);
  NEW(SubTreeOffset, SOAG.NextSymOcc);
  NEW(FirstRule, SOAG.NextSym);
  NEW(AffixAppls, EAG.NextVar);
  FOR i := SOAG.firstRule TO SOAG.NextRule-1 DO LocalVars[i] := 0; AffixVarCount[i] := -1 END;
  FOR i := EAG.firstNode TO EAG.NextNode-1 DO NodeName[i] := -1 END;
  FOR i := EAG.firstVar TO EAG.NextVar-1 DO
    EAG.Var[i].Def  := FALSE; AffixAppls[i] := SOAG.AffixApplCnt[i]
  END;

  (* SubTreeOffset[] nimmt zu jedem Symbolvorkommen den Offset des Unterbaumes in einem
    Knoten des Ableitungsbaumes auf; die Groesse von PredOcc[] wird bestimmt *)
  FOR R := SOAG.firstRule TO SOAG.NextRule-1 DO Offset := 0;
    FOR SO := SOAG.Rule[R].SymOcc.Beg+1 TO SOAG.Rule[R].SymOcc.End DO
      IF ~SOAG.IsPredNont(SO) THEN INC(Offset); SubTreeOffset[SO] := Offset END
    END
  END;

  (* FirstRule[] enthaelt fuer jedes Symbol den Index der ersten Regel, in der das Symbol
    auf der linken Seite vorkommt;  Berechnung von AffPosOffset[] fuer jedes Symbol *)
  FOR S := SOAG.firstSym TO SOAG.NextSym-1 DO
    SO := SOAG.Sym[S].FirstOcc;
    IF SO # SOAG.nil THEN
      R := SOAG.SymOcc[SO].RuleInd;
      WHILE SO # SOAG.Rule[R].SymOcc.Beg DO
        SO := SOAG.SymOcc[SO].Next;
        R := SOAG.SymOcc[SO].RuleInd
      END;
      SO := SOAG.SymOcc[SO].Next;
      WHILE (R >= SOAG.firstRule) & (S = SOAG.SymOcc[SOAG.Rule[R].SymOcc.Beg].SymInd) DO
        FirstRule[S] := R;
```

```
        DEC(R)
      END
    END
  END;

END Init;

(*--------------- Generierung -------------------------------------*)

PROCEDURE Ind;
VAR i: INTEGER;
BEGIN
  FOR i := 1 TO Indent DO
    IO.WriteText(Out, "\t");
  END
END Ind;

PROCEDURE WrS(String: ARRAY OF CHAR);
BEGIN
  IO.WriteText(Out, String);
END WrS;

PROCEDURE WrI(Int: INTEGER);
BEGIN
  IO.WriteInt(Out, Int);
END WrI;

PROCEDURE WrSI(String: ARRAY OF CHAR; Int: INTEGER);
BEGIN
  IO.WriteText(Out, String);
  IO.WriteInt(Out, Int);
END WrSI;

PROCEDURE WrIS(Int: INTEGER; String: ARRAY OF CHAR);
BEGIN
  IO.WriteInt(Out, Int);
  IO.WriteText(Out, String);
END WrIS;

PROCEDURE WrSIS(String1: ARRAY OF CHAR;
              Int: INTEGER; String2: ARRAY OF CHAR);
BEGIN
  IO.WriteText(Out, String1);
  IO.WriteInt(Out, Int);
  IO.WriteText(Out, String2);
END WrSIS;

PROCEDURE GenHeapInc(n: INTEGER);
BEGIN
  IF n # 0 THEN
    IF n = 1 THEN
      WrS("INC(NextHeap); \n")
    ELSE
      WrSIS("INC(NextHeap, ", n, "); \n")
    END
  END
END GenHeapInc;


PROCEDURE GenVar(Var: INTEGER);
BEGIN
  WrSI("V", Var)
END GenVar;


PROCEDURE GenHeap(Var, Offset: INTEGER);
BEGIN
  WrS("Heap[");
  IF Var > 0 THEN GenVar(Var) ELSE WrS("NextHeap") END;
  IF Offset > 0 THEN WrSI(" + ", Offset)
  ELSIF Offset < 0 THEN WrSI(" - ", - Offset)
  END;
  WrS("]")
END GenHeap;


PROCEDURE GenOverflowGuard(n: INTEGER);
BEGIN
  IF n > 0 THEN
    WrSIS("IF NextHeap >= LEN(Heap^) - ", n, " THEN EvalExpand END; \n")
  END
END GenOverflowGuard;
```

```
PROCEDURE GenAffPos(S, AN: INTEGER);
(*IN: Symbol, Nummer eines Affixparameter relativ zum Symbolvorkommen
 OUT: -
 SEM: Generierung eines Zugriffs auf die Instanz einer Affixposition *)
VAR
  SN, AP: INTEGER;
BEGIN
  WrSIS("AffPos[S", S, "+"); WrIS(AN, "] ")
END GenAffPos;


PROCEDURE GenAffix(V: INTEGER);
(*IN: Affixnummer
 OUT: -
 SEM: Generiert einen Zugriff auf den Inhalt eines Affixes *)
VAR AP: INTEGER;
BEGIN
  ASSERT( AffixOffset[V] # notApplied);
  IF AffixOffset[V] = optimizedStorage THEN
    AP := GetCorrespondedAffPos(SOAG.DefAffOcc[V]);
    IF SOAG.StorageName[AP] > 0 THEN
      WrSIS("Stacks.Top(Stack", SOAG.StorageName[AP], ") ")
    ELSE
      WrSI("GV", -SOAG.StorageName[AP])
    END
  ELSE
    WrSIS("Var[VI+", AffixOffset[V], "]")
  END
END GenAffix;


PROCEDURE GenAffixAssign(V: INTEGER);
(*IN: Affix
 OUT: -
 SEM: Generierung einer Zuweisung zu einer Instanz einer Affixvariable;
    nur in Kombination mit der Prozedur GenClose zu verwenden *)
VAR
  AP: INTEGER;
BEGIN
  ASSERT( AffixOffset[V] # notApplied);
  IF AffixOffset[V] = optimizedStorage THEN
    AP := GetCorrespondedAffPos(SOAG.DefAffOcc[V]);
    IF SOAG.StorageName[AP] > 0 THEN
      WrSIS("Stacks.Push(Stack", SOAG.StorageName[AP], ", "); Close := TRUE
    ELSE
      WrSIS("GV", -SOAG.StorageName[AP], " := "); Close := FALSE
    END
  ELSE
    WrSIS("Var[VI+", AffixOffset[V], "] := "); Close := FALSE
  END
END GenAffixAssign;


PROCEDURE GenClose;
BEGIN
  IF Close THEN WrS("); ") ELSE WrS("; ") END
END GenClose;


PROCEDURE GenIncRefCnt(Var: INTEGER);
(*IN: Affixvariable oder (< 0) lokale Variable
 OUT: -
 SEM: Generiert eine Erhoehung des Referenzzaehlers des Knotens auf den das Affixes bzw.
    der Index verweist, im Falle eines Stacks wird globale Var. RefIncVar verwendet *)
VAR AP: INTEGER;
BEGIN
  WrS("INC(Heap[");
  IF Var < 0 THEN GenVar(- Var) ELSE GenAffix(Var) END;
  WrS("],  refConst);\n")
END GenIncRefCnt;


PROCEDURE GenFreeAffix(V: INTEGER);
(*IN: Affixvariable
 OUT: -
 SEM: generiert die Freigabe des alloziierten Speichers , wenn
    die Affixvariable das letzte mal appliziert wurde (AffixAppls = 0) *)
VAR AP: INTEGER;
BEGIN
  IF AffixAppls[V] = 0 THEN Ind; WrS("FreeHeap("); GenAffix(V); WrS(");\n") END
```

```
END GenFreeAffix;

PROCEDURE GenPopAffix(V: INTEGER);
(*IN: Affixvariable
 OUT: -
 SEM: generiert die Kellerspeicherfreigabe, wenn
    die Affixvariable das letzte mal appliziert wurde (AffixAppls = 0) *)
VAR AP: INTEGER;
BEGIN
  IF AffixAppls[V] = 0 THEN
    IF AffixOffset[V] = optimizedStorage THEN
      AP := GetCorrespondedAffPos(SOAG.DefAffOcc[V]);
      IF SOAG.StorageName[AP] > 0 THEN
        Ind; WrSIS("Stacks.Pop(Stack", SOAG.StorageName[AP], ");\n")
      ELSE
        Ind; WrSIS("GV", -SOAG.StorageName[AP], " := -1;\n");
      END
    END
  END
END GenPopAffix;

PROCEDURE GenSynPred(SymOccInd, VisitNo: INTEGER);
(*IN: Symbolvorkommen
 OUT: -
 SEM: Generierung der Syntheseaktionen eines Besuchs fuer die besuchsrelevanten Affixparameter
    eines Symbolvorkommens *)
VAR
  Node, S, Offset, AP, AN, V, SN, P: INTEGER; IsPred: BOOLEAN;

  PROCEDURE GenSynTraverse(Node: INTEGER; VAR Offset: INTEGER);
  (*IN: Knoten des Affixbaumes, Offset des naechsten freien Heap-Elementes
   OUT: -
   SEM: Traversierung eines Affixbaumes und Ausgabe der Syntheseaktionen fuer den
      zu generierenden Compiler *)
  VAR
    Offset1, Node1, n, V, Alt: INTEGER;
  BEGIN
    Alt := EAG.NodeBuf[Node];
    Ind; GenHeap(-1, Offset); WrSIS(" := ", SLEAGGen.NodeIdent[Alt], ";\n");
    Offset1 := Offset; INC(Offset, 1+ EAG.MAlt[Alt].Arity);
    FOR n := 1 TO EAG.MAlt[Alt].Arity DO
      Node1 := EAG.NodeBuf[Node + n];
      IF Node1 < 0 THEN V := - Node1;
        IF ~EAG.Var[V].Def THEN
          SOAG.Error( SOAG.abnormalyError, 'eSOAGGen.GenSynTraverse: Affix nicht definiert.' )
        ELSE
          Ind; GenHeap(-1, Offset1+n); WrS(" := "); GenAffix(V); WrS(";\n");
          DEC(AffixAppls[V]);
          IF UseRefCnt THEN GenFreeAffix(V) END;
          IF Optimize THEN GenPopAffix(V) END
        END
      ELSE Ind; GenHeap(-1, Offset1+n); WrS(" := ");
        IF UseConst & (EAG.MAlt[EAG.NodeBuf[Node1]].Arity = 0) THEN
          WrIS(SLEAGGen.Leaf[EAG.NodeBuf[Node1]], ";\n")
        ELSE
          WrSIS("NextHeap + ", Offset, ";\n");
          GenSynTraverse(Node1, Offset)
        END
      END
    END
  END GenSynTraverse;

  PROCEDURE GenSynTraverseRefCnt(Node: INTEGER);
  (*IN: Knoten des Affixbaumes
   OUT: -
   SEM: Traversierung eines Affixbaumes und Ausgabe der Syntheseaktionen mit
       Referenzzaehler-Verfahren fuer den zu generierenden Compiler *)
  VAR
    Node1, n, V, Alt: INTEGER;
  BEGIN
    Alt := EAG.NodeBuf[Node];
    Ind; GenHeap(NodeName[Node],0); WrSIS(" := ", SLEAGGen.NodeIdent[Alt], ";\n");
    FOR n := 1 TO EAG.MAlt[Alt].Arity DO
      Node1 := EAG.NodeBuf[Node + n];
      IF Node1 < 0 THEN V := - Node1;
        IF ~EAG.Var[V].Def THEN
          SOAG.Error( SOAG.abnormalyError, 'eSOAGGen.GenSynTraverse: Affix nicht definiert.' )
        ELSE
          Ind; GenHeap(NodeName[Node],n); WrS(" := "); GenAffix(V); WrS("; ");
          DEC(AffixAppls[V]);
          IF AffixAppls[V] > 0 THEN GenIncRefCnt(V);
          ELSE WrS("(* komplementaere Referenzzaehlerbehandlung *)\n") END;
```

```
          IF Optimize THEN GenPopAffix(V) END
        END
      ELSE Ind;
        IF UseConst & (EAG.MAlt[EAG.NodeBuf[Node1]].Arity = 0) THEN
          GenHeap(NodeName[Node],n);
          WrSIS(" := ", SLEAGGen.Leaf[EAG.NodeBuf[Node1]], "; ");
          WrSIS("INC(Heap[", SLEAGGen.Leaf[EAG.NodeBuf[Node1]], "], refConst);\n")
        ELSE
          WrSIS("GetHeap(", EAG.MAlt[EAG.NodeBuf[Node1]].Arity, ", ");
          GenVar(NodeName[Node1]); WrS(");\n");
          GenSynTraverseRefCnt(Node1); Ind;
          GenHeap(NodeName[Node],n); WrS(" := "); GenVar(NodeName[Node1]);  WrS(";\n");
        END
      END
    END
  END GenSynTraverseRefCnt;

BEGIN
  S := SOAG.SymOcc[SymOccInd].SymInd;
  IsPred := (VisitNo=-1);
  FOR AP := SOAG.SymOcc[SymOccInd].AffOcc.Beg TO SOAG.SymOcc[SymOccInd].AffOcc.End DO
    AN := AP - SOAG.SymOcc[SymOccInd].AffOcc.Beg;
    P := SOAG.AffOcc[AP].ParamBufInd;
    IF ~EAG.ParamBuf[P].isDef & ((SOAGVisitSeq.GetVisitNo(AP) = VisitNo) OR IsPred)
    THEN (* Traversierung des Affixbaumes *)
      Node := EAG.ParamBuf[P].Affixform;
      SN := SymOccInd-SOAG.Rule[SOAG.SymOcc[SymOccInd].RuleInd].SymOcc.Beg;
      IF Node < 0 THEN (* Affix *) V := -Node;
        IF ~EAG.Var[V].Def THEN
          SOAG.Error( SOAG.abnormalyError, 'eSOAGGen.GenSynTraverse: Affix nicht definiert.' )
        ELSIF ~IsPred THEN
          Ind; GenAffPos(S, AN); WrS(":= "); GenAffix(V); WrS("; "); DEC(AffixAppls[V]);
          IF UseRefCnt & (AffixAppls[V] > 0) THEN GenIncRefCnt(V)
          ELSE WrS("(* komplementaere Referenzzaehlerbehandlung *)\n")END;
          IF Optimize THEN GenPopAffix(V) END;
          WrS("\n");
        END
      ELSE Ind;
        IF UseConst & (SLEAGGen.AffixPlace[P] >= 0) THEN
          GenAffPos(S,AN); WrSI(" := ", SLEAGGen.AffixPlace[P]);
          IF UseRefCnt THEN WrSIS("; INC(Heap[", SLEAGGen.AffixPlace[P], "], refConst)") END;
          WrS(";\n");
        ELSIF UseRefCnt THEN
          WrSIS("GetHeap(", EAG.MAlt[EAG.NodeBuf[Node]].Arity, ", ");
          GenVar(NodeName[Node]); WrS(");\n");
          GenSynTraverseRefCnt(Node);
          Ind; GenAffPos(S, AN); WrS( ":= "); GenVar(NodeName[Node]); WrS(";\n")
        ELSE
          GenOverflowGuard(SLEAGGen.AffixSpace[P]);
          Ind; GenAffPos(S, AN); WrS(":= NextHeap;\n");
          Offset := 0; GenSynTraverse(Node, Offset); Ind; GenHeapInc(Offset)
        END
      END
    END
  END
END GenSynPred;


PROCEDURE GenAnalPred(SymOccInd, VisitNo: INTEGER);
(*IN: Symbolvorkommen, Visit-Nummer
 OUT: -
 SEM: Generierung der Analyseaktionen eines Besuchs fuer die besuchsrelevanten Affixparameter
    eines Symbolvorkommens *)
VAR
  S, AP, AN, Node, V, SN: INTEGER; IsPred, PosNeeded: BOOLEAN;

  PROCEDURE GenEqualErrMsg(Var: INTEGER);
  BEGIN
    WrS("\'"); EAG.WriteVar(Out, Var); WrS("' failed in '");
    EAG.WriteNamedHNont(Out, SOAG.SymOcc[SymOccInd].SymInd); WrS("'\'")
  END GenEqualErrMsg;

  PROCEDURE GenAnalErrMsg;
  BEGIN
    WrS('"'); EAG.WriteNamedHNont(Out, SOAG.SymOcc[SymOccInd].SymInd); WrS('"');
  END GenAnalErrMsg;

  PROCEDURE GenEqualPred(V, Node, n: INTEGER);
  (*IN: Index auf EAG.Var[] des def. Affixes, Index auf NodeName[] und Nr. des Sohnes im Heap
   OUT: -
   SEM: Generiert einen Vergleich zwischen einer Variable eines def. Affixes und einem
      Baumeintrag *)
```

```
  BEGIN
    WrS("Eq("); GenHeap(NodeName[Node], n); WrS(", "); GenAffix(V); WrS(", ");
    GenEqualErrMsg(V); WrS(");\n");
  END GenEqualPred;

  PROCEDURE GenUnequalPred(V1, V2: INTEGER);
  (*IN: zwei Indexe auf EAG.Var[]
   OUT: -
   SEM: Generiert einen Vergleich zwischen zwei Variablen der Felder Var[] (gen. Compiler) *)
  BEGIN
    WrS("UnEq("); GenAffix(V1); WrS(", "); GenAffix(V2); WrS(", ");
    IF EAG.Var[V1].Num < 0 THEN GenEqualErrMsg(V1) ELSE GenEqualErrMsg(V2) END;
    WrS(");\n");
  END GenUnequalPred;

  PROCEDURE GenPos(VAR PosNeeded: BOOLEAN);
  (* SEM: Generierung einer Positionszuweisung, wenn notwendig *)
  BEGIN
    IF PosNeeded THEN
      Ind; WrSIS("Pos := SemTree[TreeAdr+", SubTreeOffset[SymOccInd], "].Pos;\n");
      PosNeeded := FALSE
    END
  END GenPos;


  PROCEDURE GenAnalTraverse(Node: INTEGER);
  (*IN: Knoten des Affixbaumes
   OUT: -
   SEM: Traversierung eines Affixbaumes und Ausgabe der Analyseaktionen
       fuer den zu generierenden Compiler *)
  VAR
    Node1, n, V, Alt: INTEGER;
  BEGIN
    Ind; WrS("IF "); Alt :=  EAG.NodeBuf[Node];
    IF UseConst & (EAG.MAlt[Alt].Arity = 0) THEN
      GenVar(NodeName[Node]); WrSI(" # ", SLEAGGen.Leaf[Alt])
    ELSE
      GenHeap(NodeName[Node],0); IF UseRefCnt THEN WrS(" MOD refConst") END;
      WrSI(" # ", SLEAGGen.NodeIdent[Alt])
    END;
    WrS(" THEN AnalyseError("); GenVar(NodeName[Node]); WrS(", ");
    GenAnalErrMsg; WrS(") END; \n");
    FOR  n := 1 TO EAG.MAlt[Alt].Arity DO
      Node1 := EAG.NodeBuf[Node + n];
      IF Node1 < 0 THEN (* Variable *)
        V := - Node1;
        IF EAG.Var[V].Def THEN (* Affix schon berechnet *)
          Ind; GenEqualPred(V, Node, n ); DEC(AffixAppls[V]);
          IF UseRefCnt THEN GenFreeAffix(V) END;
          IF Optimize THEN GenPopAffix(V) END
        ELSE EAG.Var[V].Def := TRUE;
          IF AffixOffset[V] # notApplied THEN
            Ind; GenAffixAssign(V); GenHeap(NodeName[Node],n); GenClose;
            IF EAG.Var[EAG.Var[V].Neg].Def THEN
              WrS("\n"); Ind; GenUnequalPred(EAG.Var[V].Neg, V);
              DEC(AffixAppls[EAG.Var[V].Neg]); DEC(AffixAppls[V]);
              IF UseRefCnt & (AffixAppls[V] > 0) THEN GenIncRefCnt(V) END;
              IF UseRefCnt THEN GenFreeAffix(EAG.Var[V].Neg) END;
              IF Optimize THEN GenPopAffix(EAG.Var[V].Neg); GenPopAffix(V) END
            ELSIF UseRefCnt THEN GenIncRefCnt(V) END
          END
        END
      ELSE (* Tree *) Ind;
        GenVar(NodeName[Node1]); WrS(" := ");
        GenHeap(NodeName[Node], n); WrS(";\n");
        GenAnalTraverse(Node1)
      END
    END
  END GenAnalTraverse;


BEGIN
  S := SOAG.SymOcc[SymOccInd].SymInd;
  IsPred := (VisitNo=-1); PosNeeded := ~IsPred;
  FOR AP := SOAG.SymOcc[SymOccInd].AffOcc.Beg TO SOAG.SymOcc[SymOccInd].AffOcc.End DO
    AN := AP - SOAG.SymOcc[SymOccInd].AffOcc.Beg;
    IF EAG.ParamBuf[SOAG.AffOcc[AP].ParamBufInd].isDef &
      ((SOAGVisitSeq.GetVisitNo(AP) = VisitNo) OR IsPred) THEN (* Traversierung des
Affixbaumes *)
      Node := EAG.ParamBuf[SOAG.AffOcc[AP].ParamBufInd].Affixform;
      SN := SymOccInd-SOAG.Rule[SOAG.SymOcc[SymOccInd].RuleInd].SymOcc.Beg;
      IF Node < 0 THEN (* Affix *) V := -Node;
```

```
        IF EAG.Var[V].Def THEN (* Affix schon berechnet *)
          GenPos(PosNeeded); Ind;
          WrS("Eq("); GenAffPos(S, AN); WrS(", "); GenAffix(V); WrS(", ");
          GenEqualErrMsg(V); WrS(");\n"); DEC(AffixAppls[V]);
          IF UseRefCnt THEN GenFreeAffix(V) END;
          IF Optimize THEN GenPopAffix(V) END;
          IF UseRefCnt THEN Ind;
            WrS("FreeHeap("); GenAffPos(S, AN); WrS(");\n")
          END
        ELSE (* Affix das erste Mal berechnet *)
          EAG.Var[V].Def := TRUE;
          IF ~IsPred THEN
            IF AffixOffset[V] # notApplied THEN
              Ind; GenAffixAssign(V); GenAffPos(S, AN); GenClose;
              IF UseRefCnt THEN WrS("(* komplementaere Referenzzaehlerbehandlung *)") END;
              WrS("\n")
            END
          END;
          IF EAG.Var[EAG.Var[V].Neg].Def THEN
            GenPos(PosNeeded);
            Ind; WrS("UnEq("); GenAffix(EAG.Var[V].Neg); WrS(", ");
            GenAffix(V); WrS(", "); GenEqualErrMsg(V); WrS(");\n");
            DEC(AffixAppls[EAG.Var[V].Neg]); DEC(AffixAppls[V]);
            IF UseRefCnt THEN
              GenFreeAffix(EAG.Var[V].Neg); GenFreeAffix(V)
            END;
            IF Optimize THEN
              GenPopAffix(EAG.Var[V].Neg); GenPopAffix(V)
            END
          END
        END
      ELSE GenPos(PosNeeded); Ind;
        GenVar(NodeName[Node]); WrS(" := "); GenAffPos(S, AN); WrS(";\n");
        GenAnalTraverse(Node);
        IF UseRefCnt THEN Ind;
          WrS("FreeHeap("); GenAffPos(S, AN); WrS(");\n")
        END
      END
    END
  END
END GenAnalPred;


PROCEDURE GenVisitCall(SO, VisitNo: INTEGER);
(*IN: Symbolvorkommen, Visit-Nummer
 OUT: -
 SEM: Generierung eines Aufrufes der Prozedur 'Visit' fuer den zu generierenden Compiler *)
BEGIN
  Ind; WrSIS("Visit(TreeAdr+", SubTreeOffset[SO], ", "); WrIS(VisitNo,");\n");
END GenVisitCall;


PROCEDURE GenLeave(SO, VisitNo: INTEGER);
(* SEM: generriert nur Kommentar *)
BEGIN
  Ind; WrSIS("(* Leave; VisitNo: ",VisitNo, " *)\n");
END GenLeave;


PROCEDURE GenPredCall(SO: INTEGER);
(*IN: Symbolvorkommen eines Praedikates
 OUT: -
 SEM: Generierung des Aufrufes einer Praedikatprozedur *)
VAR S, AP, AN, AP1, Node, V: INTEGER;
BEGIN
  IF UseRefCnt THEN
    FOR AP := SOAG.SymOcc[SO].AffOcc.Beg TO SOAG.SymOcc[SO].AffOcc.End DO
      IF ~EAG.ParamBuf[SOAG.AffOcc[AP].ParamBufInd].isDef THEN
        AN := AP - SOAG.SymOcc[SO].AffOcc.Beg;
        V := -EAG.ParamBuf[SOAG.AffOcc[AP].ParamBufInd].Affixform;
        IF V > 0 THEN
          Ind; GenIncRefCnt(V)
        END
      END
    END
  END;
  S := SOAG.SymOcc[SO].SymInd;
  Ind; WrSIS("Check", S, "('"); EAG.WriteNamedHNont(Out, S); WrS("', ");
  FOR AP := SOAG.SymOcc[SO].AffOcc.Beg TO SOAG.SymOcc[SO].AffOcc.End DO
    Node := EAG.ParamBuf[SOAG.AffOcc[AP].ParamBufInd].Affixform;
    AN := AP - SOAG.SymOcc[SO].AffOcc.Beg; V := -Node;
    IF EAG.ParamBuf[SOAG.AffOcc[AP].ParamBufInd].isDef THEN
```

```
      IF (V > 0) & (SOAG.DefAffOcc[V] = AP) THEN
        IF Optimize & (AffixOffset[V] = optimizedStorage) THEN
          AP1 := GetCorrespondedAffPos(SOAG.DefAffOcc[V]);
          IF SOAG.StorageName[AP1] > 0 THEN (* Stack *)
            GenAffPos(S, AN)
          ELSE
            WrSI("GV", -SOAG.StorageName[AP1])
          END
        ELSIF AffixOffset[V] = notApplied THEN GenAffPos(S, AN)
        ELSE GenAffix(V) END
      ELSE GenAffPos(S, AN) END
    ELSE (* appl. Affixpos *)
      IF Node > 0 THEN GenAffPos(S, AN) ELSE GenAffix(V) END
    END;
    IF AP # SOAG.SymOcc[SO].AffOcc.End THEN WrS(", ") ELSE WrS(");\n") END
  END;
  FOR AP := SOAG.SymOcc[SO].AffOcc.Beg TO SOAG.SymOcc[SO].AffOcc.End DO
    AN := AP - SOAG.SymOcc[SO].AffOcc.Beg;
    V := -EAG.ParamBuf[SOAG.AffOcc[AP].ParamBufInd].Affixform;
    IF V > 0 THEN
      IF EAG.ParamBuf[SOAG.AffOcc[AP].ParamBufInd].isDef THEN
        IF AffixOffset[V] = optimizedStorage THEN
          AP1  := GetCorrespondedAffPos(SOAG.DefAffOcc[V]);
          IF SOAG.StorageName[AP1] > 0 THEN (* Stack *)
            Ind; WrSIS("Stacks.Push(Stack", SOAG.StorageName[AP1], ", ");
            GenAffPos(S, AN); WrS(");\n")
          END
        ELSIF AffixOffset[V] = notApplied THEN
          Ind; WrS("FreeHeap("); GenAffPos(S, AN); WrS("); (* Dummy-Variable *)\n")
        END
      ELSE (* applizierend *)
        DEC(AffixAppls[V]);
        IF UseRefCnt THEN GenFreeAffix(V) END;
        IF Optimize THEN GenPopAffix(V) END
      END
    END
  END
END GenPredCall;

PROCEDURE GenVarDecls(R: INTEGER);
(*IN: Regel
 OUT: -
 SEM: Generierung der Variablendeklarationen einer Regel *)
VAR SO, i: INTEGER;
BEGIN
  WrS("VAR TreeAdr, VI: IndexType; S: SemTreeEntry;\n");
  IF LocalVars[R] > 0 THEN WrS("VAR "); GenVar(1);
    FOR i := 2 TO LocalVars[R] DO WrS(", "); GenVar(i) END; WrS(": HeapType;\n")
  END;
END GenVarDecls;

PROCEDURE GenPredPos(R, i: INTEGER; VAR PosNeeded: BOOLEAN);
(*IN: Regel, Nummer des Visit-Sequenz-Eintrages, Notwendigkeit der Positionszuweisung
 OUT: -
 SEM: Generierung der Positionszuweisung vor Praedikatprozeduraufrufen; zugewiesen wird die
    Position des vorhergehenden Visits *)
VAR
  k: INTEGER;
BEGIN
  IF PosNeeded THEN
    DEC(i);
    WHILE (~(SOAG.VS[i] IS SOAG.Visit) & ~(SOAG.VS[i] IS SOAG.Leave))
        & (i > SOAG.Rule[R].VS.Beg)
    DO DEC(i) END;
    IF SOAG.VS[i] IS SOAG.Visit THEN
      k := SubTreeOffset[SOAG.VS[i](SOAG.Visit).SymOcc]
    ELSE (* SOAG.VS[i] IS SOAG.Leave OR  i = SOAG.Rule[R].VS.Beg *)
      k := SOAG.Rule[R].SymOcc.Beg
    END;
    Ind; WrSIS("Pos := SemTree[TreeAdr+",k ,"].Pos;\n");
    PosNeeded := FALSE
  END
END GenPredPos;


PROCEDURE GenVisitRule(R: INTEGER);
(*IN: Regelnummer
 OUT: -
 SEM: Generiert Code fuer die Visit-Sequenzen einer Regel *)
VAR
  SO, VN, VisitNo, i, S, NontCnt: INTEGER;
  onlyoneVisit, first, PosNeeded: BOOLEAN;
```

```
BEGIN
  Indent := 0;
  WrSIS("PROCEDURE VisitRule", R, "(Symbol: LONGINT; VisitNo: INTEGER);\n");
  WrS( "(*\n"); Protocol.Out := Out;
  WrS( "   "); Protocol.WriteRule(R);
  WrS( "*)\n"); Protocol.Out := IO.Msg;
  GenVarDecls(R);
  WrS("BEGIN\n"); INC(Indent, cTab);

  NontCnt := 1;
  FOR SO := SOAG.Rule[R].SymOcc.Beg+1 TO SOAG.Rule[R].SymOcc.End DO
    IF ~SOAG.IsPredNont(SO) THEN INC(NontCnt) END
  END;
  SO := SOAG.Rule[R].SymOcc.Beg;
  Ind; WrS   ("IF VisitNo = syntacticPart THEN\n"); INC(Indent, cTab);
  Ind; WrSIS("IF NextSemTree >= LEN(SemTree^)-", NontCnt, " THEN ExpandSemTree END;\n");
  Ind; WrS   ("TreeAdr := SemTree[Symbol].Adr;\n");
  Ind; WrS   ("SemTree[Symbol].Adr := NextSemTree;\n");
  Ind; WrS   ("SemTree[Symbol].Pos := PosTree[TreeAdr];\n");
  Ind; WrSIS("INC(AffixVarCount, ", GetAffixCount(R), ");\n");
  IF AffixVarCount[R] > 0 THEN
    Ind; WrSIS("IF NextVar >= LEN(Var^)-", AffixVarCount[R], " THEN ExpandVar END;\n");
    Ind; WrSIS("SemTree[Symbol].VarInd := NextVar; INC(NextVar, ", AffixVarCount[R],");\n")
  ELSE Ind; WrS("SemTree[Symbol].VarInd := nil;\n") END;
  Ind; WrS   ("SemTree[NextSemTree] := SemTree[Symbol];\n");
  Ind; WrS   ("INC(NextSemTree);\n");
  FOR SO := SOAG.Rule[R].SymOcc.Beg+1 TO SOAG.Rule[R].SymOcc.End DO
    IF ~SOAG.IsPredNont(SO) THEN
      Ind; WrS   ("NEW(S);\n");
      Ind; WrSIS("S.Adr := Tree[TreeAdr+", SubTreeOffset[SO], "];\n");
      Ind; WrSIS("S.Rule := ", FirstRule[SOAG.SymOcc[SO].SymInd]-1, " + Tree[S.Adr] MOD
hyperArityConst;\n");
      Ind; WrS   ("SemTree[NextSemTree] := S; INC(NextSemTree);\n")
    END
  END;
  first := TRUE;
  FOR SO := SOAG.Rule[R].SymOcc.Beg+1 TO SOAG.Rule[R].SymOcc.End DO
    IF ~SOAG.IsPredNont(SO) THEN
      IF first THEN Ind;WrS("TreeAdr := SemTree[Symbol].Adr;\n"); first := FALSE END;
      Ind; WrSIS("Visit(TreeAdr+", SubTreeOffset[SO], ", syntacticPart);\n")
    END
  END;

  DEC(Indent, cTab);
  Ind; WrS("ELSE\n"); INC(Indent, cTab);

  (* lokale Variablen *)
  Ind; WrS("TreeAdr := SemTree[Symbol].Adr;\n");
  IF AffixVarCount[R] > 0 THEN Ind; WrS("VI := SemTree[Symbol].VarInd;\n\n") END;

  IF SOAGVisitSeq.GetMaxVisitNo(SOAG.Rule[R].SymOcc.Beg) = 1 THEN
    onlyoneVisit := TRUE
  ELSE
    onlyoneVisit := FALSE;
    Ind; WrS( "CASE VisitNo OF\n"); INC(Indent, cTab);
    Ind; WrS( "1:\n"); INC(Indent, cTab)
  END;

  VisitNo := 1; PosNeeded := TRUE;
  Ind; WrS("(* Visit-beginnende Analyse *)\n");
  GenAnalPred(SOAG.Rule[R].SymOcc.Beg, VisitNo);
  FOR i := SOAG.Rule[R].VS.Beg TO SOAG.Rule[R].VS.End DO
    IF SOAG.VS[i] IS SOAG.Visit THEN
      SO := SOAG.VS[i](SOAG.Visit).SymOcc;
      S := SOAG.SymOcc[SO].SymInd;
      VN := SOAG.VS[i](SOAG.Visit).VisitNo;
      Ind; WrS("(* Synthese *)\n");
      GenSynPred(SO, VN);
      GenVisitCall(SO, VN);
      Ind; WrS("(* Analyse *)\n");
      GenAnalPred(SO, VN); WrS("\n");
      PosNeeded := TRUE
    ELSIF SOAG.VS[i] IS SOAG.Call THEN
      SO := SOAG.VS[i](SOAG.Call).SymOcc;
      Ind; WrS("(* Synthese *)\n");
      GenSynPred(SO, -1); (* VisitNo undefiniert! *)
      GenPredPos(R, i, PosNeeded);
      GenPredCall(SO);
      Ind; WrS("(* Analyse *)\n");
      GenAnalPred(SO, -1); WrS("\n")
    ELSIF SOAG.VS[i] IS SOAG.Leave THEN
      SO := SOAG.Rule[R].SymOcc.Beg;
```

```
      VN := SOAG.VS[i](SOAG.Leave).VisitNo;
      ASSERT(VN = VisitNo); (* Konzeptioneller Fehler, falls Assert scheitert! *)
      Ind; WrS("(* Visit-abschliessende Synthese *)\n");
      GenSynPred(SO, VisitNo);
      GenLeave(SO, VisitNo);
      IF VisitNo < SOAGVisitSeq.GetMaxVisitNo(SO) THEN
        DEC(Indent, cTab);
        INC(VisitNo); PosNeeded := TRUE;
        Ind; WrSIS("| ",VisitNo, ":\n"); INC(Indent, cTab);
        Ind; WrS("(* Visit-beginnende Analyse *)\n");
        GenAnalPred(SO, VisitNo);
      ELSE
        DEC(Indent, cTab)
      END;
    END
  END;
  IF ~ onlyoneVisit THEN DEC(Indent, cTab); Ind; WrS("END\n"); DEC(Indent, cTab) END;
  Ind; WrS("END\n");
  WrSIS("END VisitRule", R, ";\n\n");
END GenVisitRule;


PROCEDURE GenVisit;
(* SEM: Generierung der Prozedur 'Visit', die die Besuche auf die entsprechenden Regeln
      verteilt *)
VAR R: INTEGER;
BEGIN
  Indent := 0;
  WrS("PROCEDURE Visit(Symbol: LONGINT; VisitNo: INTEGER);\n");
  WrS("BEGIN\n"); INC(Indent, cTab);
  Ind; WrS("CASE SemTree[Symbol].Rule OF\n"); INC(Indent, cTab);
  FOR R := SOAG.firstRule TO SOAG.NextRule-1 DO
    IF SOAG.IsEvaluatorRule(R) THEN Ind;
      IF R # SOAG.firstRule THEN WrSIS("| ", R, ": ") ELSE WrSIS("  ", R, ": ") END;
      WrSIS("VisitRule", R, "(Symbol, VisitNo);\n")
    END
  END; DEC(Indent, cTab);
  Ind; WrS("END\n");
  WrS("END Visit;\n\n");
END GenVisit;

PROCEDURE GenConstDeclarations;
(* SEM: Generierung der Konstanten fuer den Zugriff auf AffPos[] im generierten Compiler *)
VAR S: INTEGER;
BEGIN
  FOR S := SOAG.firstSym TO SOAG.NextSym-1 DO
    WrSIS("\tS", S, " = "); WrIS(SOAG.Sym[S].AffPos.Beg, "; (* ");
    EAG.WriteHNont(Out, S); WrS(" *)\n")
  END
END GenConstDeclarations;

PROCEDURE GenStackDeclarations;
(* SEM: Generierung der Deklarationen der globalen Variablen und Stacks *)
VAR V: INTEGER;
BEGIN
  IF (SOAGOptimizer.GlobalVar > 0) OR (SOAGOptimizer.StackVar > 0) THEN
    WrS("\nVAR\n");
    FOR V := SOAGOptimizer.firstGlobalVar TO SOAGOptimizer.GlobalVar DO
      WrSIS("\tGV", V, ": HeapType;\n")
    END;
    FOR V := SOAGOptimizer.firstStackVar TO SOAGOptimizer.StackVar DO
      WrSIS("\tStack", V, ": Stacks.Stack;\n")
    END;
    WrS("\n")
  END;
END  GenStackDeclarations;

PROCEDURE GenStackInit;
(* SEM: Generierung der Initialisierungen der Stacks *)
VAR S: INTEGER;
BEGIN
  IF SOAGOptimizer.StackVar > 0 THEN
    FOR S := SOAGOptimizer.firstStackVar TO SOAGOptimizer.StackVar DO
      WrSIS("\tStacks.New(Stack", S, ", 8);\n")
    END
  END
END GenStackInit;


PROCEDURE GenerateModule;
(* SEM: Generierung des Compiler-Moduls *)
VAR R: INTEGER;
```

```
  Name : ARRAY EAG.BaseNameLen + 10 OF CHAR;
  Fix : IO.TextIn;
  StartRule: INTEGER;

  PROCEDURE InclFix(Term : CHAR);
    VAR c : CHAR;
  BEGIN
    IO.Read(Fix, c);
    WHILE c # Term DO
      IF c = 0X THEN
        IO.WriteText(IO.Msg, "\n  error: unexpected end of eSOAG.Fix\n");
        IO.Update(IO.Msg); HALT(99)
      END;
      IO.Write(Out, c); IO.Read(Fix, c)
    END
  END InclFix;

  PROCEDURE Append(VAR Dest : ARRAY OF CHAR; Src, Suf : ARRAY OF CHAR);
    VAR i, j : INTEGER;
  BEGIN
    i := 0; j := 0;
    WHILE (Src[i] # 0X) & (i < LEN(Dest) - 1) DO Dest[i] := Src[i]; INC(i) END;
    WHILE (Suf[j] # 0X) & (i < LEN(Dest) - 1) DO Dest[i] := Suf[j]; INC(i); INC(j) END;
    Dest[i] := 0X
  END Append;

BEGIN
  IO.OpenIn(Fix, "eSOAG.Fix", Error);
  IF Error THEN
    IO.WriteText(IO.Msg, "\n  error: could not open eSOAG.Fix\n"); IO.Update(IO.Msg); HALT(99)
  END;
  Append(Name, EAG.BaseName, "Eval");
  IO.CreateModOut(Out, Name);
  SLEAGGen.InitGen(Out, SLEAGGen.sSweepPass);
  InclFix("$"); WrS(Name);          (* module name *)
  InclFix("$"); WrI(HyperArity());     (* hyperArityConst *)
  InclFix("$"); GenConstDeclarations;
  InclFix("$"); IF Optimize THEN GenStackDeclarations END; SLEAGGen.GenDeclarations;
  InclFix("$"); SLEAGGen.GenPredProcs;
  FOR R := SOAG.firstRule TO SOAG.NextRule-1 DO
    IF SOAG.IsEvaluatorRule(R) THEN
      ComputeNodeNames(R);
      ComputeAffixOffset(R);
      GenVisitRule(R)
    END
  END;
  GenVisit;
  EmitGen.GenEmitProc(Out);
  InclFix("$"); WrI(SOAG.NextPartNum);
  InclFix("$"); IF Optimize THEN GenStackInit END;
  StartRule := FirstRule[SOAG.SymOcc[SOAG.Sym[EAG.StartSym].FirstOcc].RuleInd];
  InclFix("$"); IF StartRule-1 # 0 THEN WrIS(StartRule-1, " + ") END;
  InclFix("$"); WrSI("S", EAG.StartSym);
  InclFix("$"); EmitGen.GenEmitCall(Out);
  InclFix("$"); EmitGen.GenShowHeap(Out);
  InclFix("$"); IF Optimize THEN WrI(SOAGOptimizer.StackVar) ELSE WrI(0) END;
  InclFix("$"); IF Optimize THEN WrI(SOAGOptimizer.GlobalVar) ELSE WrI(0) END;
  InclFix("$"); WrS(EAG.BaseName); WrS("Eval");
  InclFix("$");                          (* rest of file *)
  IO.Update(Out);
  IF ShowMod THEN IO.Show(Out)
  ELSE IO.Compile(Out, Error);
    IF Error THEN IO.Show(Out) END
  END;
  SLEAGGen.FinitGen; IO.CloseIn(Fix); IO.CloseOut(Out)
END GenerateModule;

PROCEDURE Generate*;
(* SEM: Steuerung der Generierung *)
BEGIN
  UseConst := ~ IO.IsOption("c");
  UseRefCnt := ~ IO.IsOption("r");
  ShowMod := IO.IsOption("m");
  Optimize := ~IO.IsOption("o");

  SOAGPartition.Compute;
  SOAGVisitSeq.Generate;
  IF Optimize THEN SOAGOptimizer.Optimize END;

  IO.WriteText(IO.Msg, "SOAG writing "); IO.WriteString(IO.Msg, EAG.BaseName);
  IO.WriteText(IO.Msg, "   ");
  IF Optimize THEN IO.WriteText(IO.Msg, "+o ") ELSE IO.WriteText(IO.Msg, "-o ") END;
```

```
  IO.Update(IO.Msg);
  IF EAG.Performed({EAG.analysed, EAG.predicates}) THEN
    Init;
    GenerateModule;
    IF ~ Error THEN INCL(EAG.History, EAG.isSSweep); INCL(EAG.History, EAG.hasEvaluator) END;
  END
END Generate;

END eSOAGGen.
```

9.3.2 eSOAG.Fix

```
MODULE $;      (* eSOAG.Fix, dk 09.03.97  -  Ver 1.2 *)

IMPORT IO := eIO, Stacks := eLIStacks;

CONST
   nil = -1;
   initialStorageSize = 128;
   syntacticPart = 0;
   hyperArityConst = $;

CONST
$

TYPE
   TreeType* = LONGINT;
   OpenTree* = POINTER TO ARRAY OF TreeType;
   OpenPos* = POINTER TO ARRAY OF IO.Position;
   HeapType* = LONGINT;
   IndexType = LONGINT;

VAR
   Tree : OpenTree;
   PosTree : OpenPos;
   ErrorCounter : LONGINT;
   AffixVarCount : INTEGER;
   Pos : IO.Position;
   Out : IO.TextOut;
   RefIncVar: HeapType;

TYPE
   SemTreeEntry = POINTER TO  RECORD
      Rule: LONGINT;
      Pos: IO.Position;
      Adr, VarInd: IndexType
   END;

   OpenSemTree = POINTER TO ARRAY OF SemTreeEntry;
   OpenVar = POINTER TO ARRAY OF HeapType;
   OpenAffPos = POINTER TO ARRAY OF HeapType;

VAR
   SemTree: OpenSemTree;
   Var: OpenVar;
   AffPos: OpenAffPos;
   NextSemTree,
   NextVar: IndexType;


(* insert evaluator global things *)

$

PROCEDURE ExpandSemTree;
VAR SemTree1: OpenSemTree; i: IndexType;
BEGIN
   NEW(SemTree1, 2*LEN(SemTree^));
   FOR i := 0 TO LEN(SemTree^) - 1 DO SemTree1[i] := SemTree[i] END;
   SemTree := SemTree1
END ExpandSemTree;


PROCEDURE ExpandVar;
VAR Var1: OpenVar; i: IndexType;
BEGIN
   NEW(Var1, 2*LEN(Var^));
   FOR i := 0 TO LEN(Var^) - 1 DO Var1[i] := Var[i] END;
```

```
   Var := Var1
END ExpandVar;

PROCEDURE ^ Visit(Symbol: LONGINT; VisitNo: INTEGER);

(* Predicates *)

$

PROCEDURE Init;
BEGIN
   NEW(SemTree, initialStorageSize);
   NEW(AffPos, $);
   NEW(Var, 8*initialStorageSize);
   NextSemTree := 0;
   NextVar := 0;
   AffixVarCount := 0;
$END Init;

PROCEDURE TraverseSyntaxTree*(Tree1 : OpenTree; PosTree1 : OpenPos;
                 ErrCounter : LONGINT; Adr : TreeType; HyperArity : INTEGER);
   VAR StartSymbol: IndexType; V1: HeapType;
BEGIN
   IF HyperArity # hyperArityConst THEN
      IO.WriteText(IO.Msg, "\n   internal error: 'arityConst' is wrong\n");
      IO.Update(IO.Msg); HALT(99)
   END;
   Tree := Tree1;
   PosTree := PosTree1;
   ErrorCounter := ErrCounter;
   Init;
   StartSymbol := NextSemTree;
   NEW(SemTree[StartSymbol]);
   SemTree[StartSymbol].Adr := Adr;
   SemTree[StartSymbol].Rule := $Tree[Adr] MOD hyperArityConst;
   INC(NextSemTree);
   Visit(StartSymbol, syntacticPart);
   Visit(StartSymbol, 1); V1 := AffPos[$];
   IF ErrorCounter > 0 THEN
      IO.WriteText(IO.Msg, "  "); IO.WriteInt(IO.Msg, ErrorCounter);
      IO.WriteText(IO.Msg, " errors detected\n"); IO.Update(IO.Msg)
   ELSE
$  END;
$  IF IO.IsOption("i") THEN
         IO.WriteText(IO.Msg, "\tsemantic tree of "); IO.WriteInt(IO.Msg, AffixVarCount);
         IO.WriteText(IO.Msg, " affixes uses "); IO.WriteInt(IO.Msg, NextVar);
         IO.WriteText(IO.Msg, " affix variables, with\n\t\t");
         IO.WriteInt(IO.Msg, $); IO.WriteText(IO.Msg, " stacks and\n\t\t");
         IO.WriteInt(IO.Msg, $); IO.WriteText(IO.Msg, " global variables\n")
   END;
   Tree := NIL; PosTree := NIL; SemTree := NIL; Var := NIL; AffPos := NIL
END TraverseSyntaxTree;

END $.   (* insert module name *)
```

Anhang A: Beispiele von Nicht-OEAGen

Ausgehend von dem in [KröpKann] aufgezeigten Fragment einer Nicht-OEAG wurde eine SOEAG konstruiert, die keine OEAG ist. Das dort angegeben Schema einer Grammatik ergab die folgende Spezifikation.

Grammatik G_1:

```
N = "0" | N N | .

S<+ "0": N>:
    B<"0","0","0",N1,N2,N3>.

B<- N5: N, - N6: N, - N7: N, + N7:N, + N6: N, + N5: N>:
    B<"0", N1 N4, "0", N1, N2, N3>
    C<N3, N4>.

C<- N1: N, + "0": N>: 'a'.
```

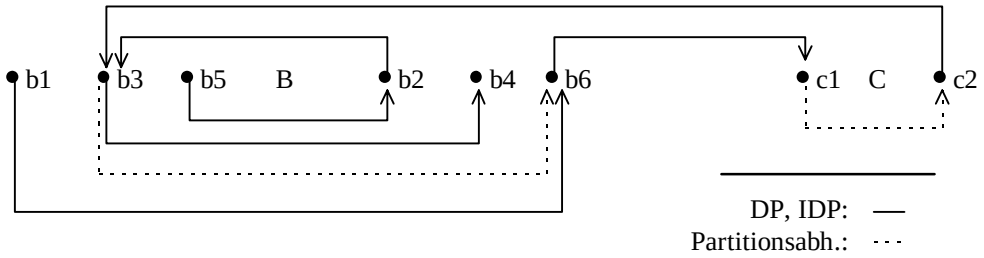


Abbildung A-1: Abhängigkeitsteilgraph einer Nicht-OEAG

Abbildung A-1 stellt einen Teil des Abhängigkeitsgraphen der Hyper-Regel für das Symbol *B* dieser Grammatik dar. Affixpositionen zu einem Symbol werden klein geschrieben, ungerade Indizes symbolisieren inherited-Affixpositionen, gerade Indizes synthesized-Affixpositionen.

Ein Evaluatorgenerator, der die Affixpositionen in einer Affixpartition so spät wie möglich („lazy“) anordnet, würde für die Symbole *B* und *C* laut [KröpKann] die folgenden beiden Partitionen ermitteln:

$A(B) = (\{b1,b2\}, \{b3, b4, b5, b6\})$,
$A(C) = (\{c1, c2\})$.

Werden die Affixpositionen so früh wie möglich („greedy“) angeordnet, so ergeben sich die Partitionen

$A(B) = (\{b2, b5\}, \{b1, b3, b4, b6\})$ und
$A(C) = (\{c1, c2\})$.

Um die durch die Partitionsmengen gegebene Reihenfolge der Berechnung der Affixpositionen im Regel-Abhängigkeitsgraph zu manifestieren, werden zusätzliche Abhängigkeiten, sogenannte *Partitionsabhängigkeiten*, eingefügt. Für die Grammatik G_1 ergeben sich für eine „lazy“- wie für eine „greedy“-Partition die Abhängigkeiten $b3 \rightarrow b6$ und $c1 \rightarrow c2$, die durch

$$b3 \rightarrow b6 \rightarrow c1 \rightarrow c2 \rightarrow b3$$

einen Zyklus bilden, wie im Graphen in Abbildung A-1 zu sehen ist. Der in dieser Arbeit vorgestellte SOEAG-Evaluator berechnet die Affixpartitionen nicht mehr unabhängig voneinander und die Affixpositionen werden so früh wie möglich angeordnet, so daß der Evaluator für G_1 die Partitionen

$A(B) = (\{b2,b5\}, \{b1, b3, b4, b6\})$ und
$A(C) = (\{c2\}, \{c1\})$

berechnet. Anstelle der ursprünglichen Abhängigkeit zwischen $c1$ und $c2$ wird $c2 \rightarrow c1$ in den Abhängigkeitsgraphen eingefügt, und es entsteht kein Zyklus mehr. Dies ergibt sich vor allem dadurch, daß die Affixpartition des Symbols *B* vor der von *C* berechnet wird und alle daraus resultierenden Abhängigkeiten, im Gegensatz zum ursprünglichen OEAG-Verfahren, sofort in den Regel-Abhängigkeitsgraphen eingetragen

werden und damit Einfluß auf die nachfolgende Berechnung der Affixpartition des Symbols *C* haben. Die Grammatik G_1 ist also keine OEAG, sondern eine SOEAG.

Grammatik G_2:

```
N = "0" .

S<+"0": N>: 'a'.

B<- N1: N, - N2: N, + N2: N, + N1: N>:
    B<"0", N6, N3, N4>
    C<N4,N5>C<N5,N6>.

C<- N1: N, + "0": N>: 'a'.
```

Die Grammatik G_2 ist durch eine Modifikation der Grammatik G_1 entstanden und ebenfalls keine OEAG. Das vorgestellte SOEAG-Verfahren berechnet zunächst wieder die Affixpartition des Symbols *B* und danach die Affixpartition des Symbols *C*:

$A(B) = (\{b1, b3, b2, b4\})$
$A(C) = (\{c1, c2\}).$

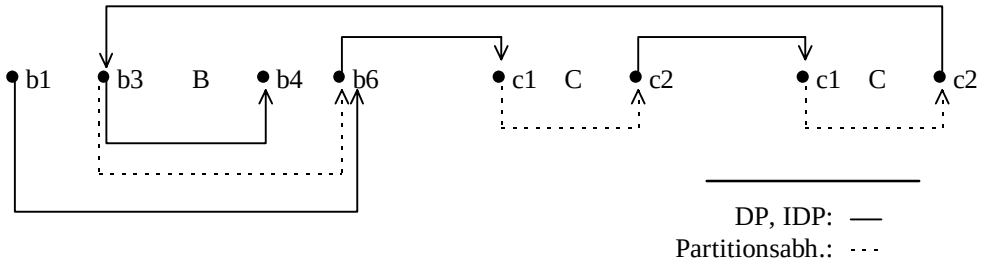


Abbildung A-2: Abhängigkeitsteilgraph einer Nicht-OEAG

Wie inAbbildung A-2, die wiederum nur den relevanten Teil des Abhängigkeitsgraphen für die Hyper-Regel des Symbols *B* darstellt, zu sehen ist, ergibt sich nach Einfügen der Abhängigkeit $c1 \rightarrow c2$ der Zyklus

$$b3 \rightarrow b4 \rightarrow c1 \rightarrow c2 \rightarrow c1 \rightarrow c2 \rightarrow b3.$$

In dieser Situation wird die Abhängigkeit $c1 \rightarrow c2$ wieder aus dem Graphen entfernt und die umgekehrte Abhängigkeit $c2 \rightarrow c1$ eingefügt. Ein Zyklus wird so vermieden und ein Evaluator kann generiert werden. Die neue Affixpartition für das Symbol *C* lautet:

$A(C) = (\{c2\}, \{c1\}).$

Damit ist die angegeben Grammatik keine OEAG, aber eine SOEAG, die mit lokalem Backtracking ermittelt werden kann.

Weitere Spezifikation, die typische Beispiele von „Nicht“-OAGen aus der Literatur ([RepTei] S.275, [GySiMa]) enthalten, sind in den Dateien `NotOEAG`*n*`.Eps` zu finden. Für alle in der Literatur gefundenen Grammatikbeispiele können SOEAG ohne Backtracking berechnet werden.

Anhang B: Benutzung des SOEAG-Evaluator-Generators

Die Generierung eines Compilers erfolgt in vier Schritten, die durch Aufruf entsprechender Kommandos initiiert werden.

1. Internalisierung der Spezifikation (Kommando: `eAnalyser.Analyse`)
2. Generierung des Scanners (Kommando: `eScanGen.Generate`)
3. Generierung des SOEAG-Evaluators (Kommando: `eSOAGGen.Generate`)
4. Generierung des Parsers (Kommando: `eELL1Gen.GenerateParser`)

Die Generierung des Parsers muß im Epsilon-Compilergenerator unbedingt als letztes erfolgen, da der Parsergenerator die internalisierte Datenstruktur irreversibel verändert.

Eine entsprechende Oberon-Tool-Datei mit den zur Generation notwendigen Kommandos, sowie deren Beschreibung, ist unter dem Namen `eSOAG.Tool` zu finden.

Anhang C: Ein einfaches Beispiel

Die folgende Beispielspezifikation liest ein Liste von *a´*s und eine gleich-lange Liste von *b´*s ein. Als „Übersetzung“ wird die Anzahl von *b´*s als Zahl ausgegeben.

```
N = "i" N | .

Z* = Z "0"|Z "1"|Z "2"|Z "3"|Z "4"|Z "5"|Z "6"|Z "7"|Z "8"|Z "9"| .

S:<+Z: Z>
    A<N> B<N,Z>.

A: <+"i" N: N> 'a' A<N> | <+ :N>.

B: <-"i" N: N, + Z1: Z> 'b' B<N,Z> INC<Z,Z1> | <- : N, + "0": Z>.

INC:
- Z "0":Z, + Z "1":Z>|
<- Z "1":Z, + Z "2":Z>|
<- Z "2":Z, + Z "3":Z>|
<- Z "3":Z, + Z "4":Z>|
<- Z "4":Z, + Z "5":Z>|
<- Z "5":Z, + Z "6":Z>|
<- Z "6":Z, + Z "7":Z>|
<- Z "7":Z, + Z "8":Z>|
<- Z "8":Z, + Z "9":Z>|
<- Z "9":Z, + Z1 "0":Z> INC<Z, Z1>|
<- :Z, + "1": Z>.
```

Der Evaluatorgenerator erzeugt aus der vorstehenden Spezifikation unter Benutzung der Konstantenfaltung und des Referenzzähler-Verfahrens, sowie mit Optimierung der Affixvariablenspeicherung folgendes Oberon-Programm:

```
MODULE SEval;    (* eSOAG.Fix, dk 09.03.97  -  Ver 1.2 *)

IMPORT IO := eIO, Stacks := eLIStacks;

CONST
  nil = -1;
  initialStorageSize = 128;
  syntacticPart = 0;
  hyperArityConst = 4;

CONST
  S0 = 0; (* S *)
  S1 = 1; (* A *)
  S2 = 2; (* B *)
  S3 = 4; (* INC *)


TYPE
  TreeType* = LONGINT;
  OpenTree* = POINTER TO ARRAY OF TreeType;
  OpenPos* = POINTER TO ARRAY OF IO.Position;
  HeapType* = LONGINT;
  IndexType = LONGINT;

VAR
  Tree : OpenTree;
  PosTree : OpenPos;
  ErrorCounter : LONGINT;
  AffixVarCount : INTEGER;
  Pos : IO.Position;
  Out : IO.TextOut;
  RefIncVar: HeapType;

TYPE
  SemTreeEntry = POINTER TO  RECORD
    Rule: LONGINT;
    Pos: IO.Position;
    Adr, VarInd: IndexType
```

```
  END;

  OpenSemTree = POINTER TO ARRAY OF SemTreeEntry;
  OpenVar = POINTER TO ARRAY OF HeapType;
  OpenAffPos = POINTER TO ARRAY OF HeapType;

VAR
  SemTree: OpenSemTree;
  Var: OpenVar;
  AffPos: OpenAffPos;
  NextSemTree,
  NextVar: IndexType;


(* insert evaluator global things *)


VAR
  GV1: HeapType;
  GV2: HeapType;
  GV3: HeapType;
  GV4: HeapType;
  GV5: HeapType;

(* ----------- eSLEAGGen.Fix Version 1.03 --  dk 20.11.97 --------- *)
  CONST
    errVal = 0;
    predefined = 7;
    arityConst = 16;
    undef = - 1;
    initialHeapSize = 8192;
  TYPE

    OpenHeap = POINTER TO ARRAY OF HeapType;
  VAR
    Heap: OpenHeap; NextHeap: HeapType;
    OutputSize: LONGINT;

  CONST
    maxArity = 2;
    refConst = 32;
  VAR
    FreeList: ARRAY maxArity OF HeapType;

  PROCEDURE EvalExpand;
    VAR Heap1: OpenHeap; i: LONGINT;
  BEGIN
    NEW(Heap1, 2 * LEN(Heap^));
    FOR i := 0 TO LEN(Heap^) - 1 DO Heap1[i] := Heap[i] END;
    Heap := Heap1
  END EvalExpand;

  PROCEDURE Reset*;
  BEGIN
    Heap := NIL
  END Reset;

  PROCEDURE GetHeap(Arity: HeapType; VAR Node: HeapType);
  BEGIN
    IF FreeList[Arity] = 0 THEN
      Node := NextHeap; IF NextHeap >= LEN(Heap^) - Arity - 1 THEN EvalExpand END;
      Heap[NextHeap] := 0; INC(NextHeap, Arity + 1)
    ELSE
      Node := FreeList[Arity]; FreeList[Arity] := Heap[FreeList[Arity]]; Heap[Node] := 0
    END;
    ASSERT(Heap[Node] DIV refConst = 0, 95)
  END GetHeap;

  PROCEDURE FreeHeap(Node: HeapType);
    VAR RArity: LONGINT; i: HeapType;
  BEGIN
    ASSERT(Node >= 0, 97);
    IF Heap[Node] DIV refConst <= 0 THEN
      RArity := (Heap[Node] MOD refConst) DIV arityConst;
      FOR i := Node + 1 TO Node + RArity DO FreeHeap(Heap[i]) END;
      ASSERT(Heap[Node] DIV refConst = 0, 96); ASSERT(Node > 0, 95);
      Heap[Node] := FreeList[RArity]; FreeList[RArity] := Node
    ELSE
      DEC(Heap[Node], refConst)
    END
  END FreeHeap;
```

```
  PROCEDURE CountHeap(): LONGINT;
    VAR i, HeapCells: LONGINT; Node: HeapType;
  BEGIN
    HeapCells := NextHeap;
    FOR i := 0 TO maxArity - 1 DO
      Node := FreeList[i];
      WHILE Node # 0 DO DEC(HeapCells, i + 1); Node := Heap[Node] END
    END;
    RETURN HeapCells
  END CountHeap;

  PROCEDURE SetErr;
  BEGIN
    INC(ErrorCounter); IO.WriteText(IO.Msg, "  "); IO.WritePos(IO.Msg, Pos);
IO.WriteText(IO.Msg, "  ");
  END SetErr;

  PROCEDURE Error(Msg: ARRAY OF CHAR);
  BEGIN
    SetErr; IO.WriteText(IO.Msg, Msg); IO.WriteLn(IO.Msg); IO.Update(IO.Msg)
  END Error;

  PROCEDURE PredError(Msg: ARRAY OF CHAR);
  BEGIN
    SetErr; IO.WriteText(IO.Msg, "predicate "); IO.WriteText(IO.Msg, Msg);
IO.WriteText(IO.Msg, " failed");
    IO.WriteLn(IO.Msg); IO.Update(IO.Msg)
  END PredError;

  PROCEDURE AnalyseError(VAR V: HeapType; Msg: ARRAY OF CHAR);
  BEGIN
    IF V # errVal THEN
      SetErr;
      IO.WriteText(IO.Msg, "analysis in '"); IO.WriteText(IO.Msg, Msg); IO.WriteText(IO.Msg,
"' failed");
      IO.WriteLn(IO.Msg); IO.Update(IO.Msg);
      INC(Heap[errVal], refConst); FreeHeap(V);
      V := errVal;
    END
  END AnalyseError;

  PROCEDURE Equal(Ptr1, Ptr2: HeapType): BOOLEAN;
    VAR i: LONGINT;
  BEGIN
    IF Ptr1 = Ptr2 THEN RETURN TRUE
    ELSIF Heap[Ptr1] MOD refConst = Heap[Ptr2] MOD refConst THEN
      FOR i := 1 TO (Heap[Ptr1] MOD refConst) DIV arityConst DO
        IF ~ Equal(Heap[Ptr1 + i], Heap[Ptr2 + i]) THEN RETURN FALSE END
      END;
      RETURN TRUE
    END;
    RETURN FALSE
  END Equal;

  PROCEDURE Eq(Ptr1, Ptr2: HeapType; ErrMsg: ARRAY OF CHAR);
  BEGIN
    IF ~ Equal(Ptr1, Ptr2) THEN IF (Ptr1 # errVal) & (Ptr2 # errVal) THEN Error(ErrMsg) END
END
  END Eq;

  PROCEDURE UnEq(Ptr1, Ptr2: HeapType; ErrMsg: ARRAY OF CHAR);
  BEGIN
    IF Equal(Ptr1, Ptr2) THEN IF (Ptr1 # errVal) & (Ptr2 # errVal) THEN Error(ErrMsg) END END
  END UnEq;

  PROCEDURE EvalInitSucceeds*(): BOOLEAN;
    CONST
      magic = 1818326597;
      name = "SEval.EvalTab";
      tabTimeStamp = 29879994;
    VAR
      Tab : IO.File; OpenError : BOOLEAN;
      i : INTEGER; l : LONGINT;

    PROCEDURE LoadError(Msg : ARRAY OF CHAR);
    BEGIN
      IO.WriteText(IO.Msg, "  loading the evaluator table "); IO.WriteString(IO.Msg, name);
      IO.WriteText(IO.Msg, " failed\n\t");
      IO.WriteText(IO.Msg, Msg); IO.WriteLn(IO.Msg); IO.Update(IO.Msg)
    END LoadError;

  BEGIN   (* EvalInitSucceeds*(): BOOLEAN; *)
```

```
    IO.OpenFile(Tab, name, OpenError);
    IF OpenError THEN LoadError("it could not be opened"); RETURN FALSE END;
    IO.GetLInt(Tab, l);
    IF l # magic THEN LoadError("not an evaluator table"); RETURN FALSE END;
    IO.GetLInt(Tab, l);
    IF l # tabTimeStamp THEN LoadError("wrong time stamp"); RETURN FALSE END;
    IO.GetLInt(Tab, l);
    IF l # predefined THEN LoadError("wrong heap size"); RETURN FALSE END;
    IF Heap = NIL THEN NEW(Heap, initialHeapSize) END;
    WHILE predefined >= LEN(Heap^) DO EvalExpand END;
    FOR i := 0 TO predefined DO IO.GetLInt(Tab, l); Heap[i] := l END;
    IO.GetLInt(Tab, l);
    IF l # tabTimeStamp THEN LoadError("file corrupt"); RETURN FALSE END;
    IO.CloseFile(Tab);
    FOR i := 0 TO maxArity - 1 DO FreeList[i] := 0 END;
    NextHeap := predefined + 1; OutputSize := 0;
    RETURN TRUE
  END EvalInitSucceeds;

(* -------------- Ende eSLEAGGen.Fix --------------- *)



PROCEDURE ExpandSemTree;
VAR SemTree1: OpenSemTree; i: IndexType;
BEGIN
  NEW(SemTree1, 2*LEN(SemTree^));
  FOR i := 0 TO LEN(SemTree^) - 1 DO SemTree1[i] := SemTree[i] END;
  SemTree := SemTree1
END ExpandSemTree;


PROCEDURE ExpandVar;
VAR Var1: OpenVar; i: IndexType;
BEGIN
  NEW(Var1, 2*LEN(Var^));
  FOR i := 0 TO LEN(Var^) - 1 DO Var1[i] := Var[i] END;
  Var := Var1
END ExpandVar;

PROCEDURE ^ Visit(Symbol: LONGINT; VisitNo: INTEGER);

(* Predicates *)

PROCEDURE ^ Pred3(V1: HeapType; VAR V2: HeapType): BOOLEAN; (* INC *)

PROCEDURE Check3(ErrMsg: ARRAY OF CHAR; V1: HeapType; VAR V2: HeapType);
BEGIN
  IF ~ Pred3(V1, V2) THEN IF (V1 # errVal) THEN PredError(ErrMsg) END END
END Check3;

PROCEDURE Pred3(V1: HeapType; VAR V2: HeapType): BOOLEAN; (* INC *)
  VAR V3, V4: HeapType;
  VAR Failed: BOOLEAN;
BEGIN
  Failed := TRUE;
IF Heap[V1] MOD refConst = 17 THEN
V3 := Heap[V1 + 1]; GetHeap(1, V2); Heap[V2] := 18; Heap[V2 + 1] := V3; INC(Heap[V3],
refConst);

Failed := FALSE;
END ;
IF Failed THEN (* 2. Alternative *)
IF Heap[V1] MOD refConst = 18 THEN
V3 := Heap[V1 + 1]; GetHeap(1, V2); Heap[V2] := 19; Heap[V2 + 1] := V3; INC(Heap[V3],
refConst);

Failed := FALSE;
END ;
IF Failed THEN (* 3. Alternative *)
IF Heap[V1] MOD refConst = 19 THEN
V3 := Heap[V1 + 1]; GetHeap(1, V2); Heap[V2] := 20; Heap[V2 + 1] := V3; INC(Heap[V3],
refConst);

Failed := FALSE;
END ;
IF Failed THEN (* 4. Alternative *)
IF Heap[V1] MOD refConst = 20 THEN
V3 := Heap[V1 + 1]; GetHeap(1, V2); Heap[V2] := 21; Heap[V2 + 1] := V3; INC(Heap[V3],
refConst);

Failed := FALSE;
```

```
END ;
IF Failed THEN (* 5. Alternative *)
IF Heap[V1] MOD refConst = 21 THEN
V3 := Heap[V1 + 1]; GetHeap(1, V2); Heap[V2] := 22; Heap[V2 + 1] := V3; INC(Heap[V3],
refConst);

Failed := FALSE;
END ;
IF Failed THEN (* 6. Alternative *)
IF Heap[V1] MOD refConst = 22 THEN
V3 := Heap[V1 + 1]; GetHeap(1, V2); Heap[V2] := 23; Heap[V2 + 1] := V3; INC(Heap[V3],
refConst);

Failed := FALSE;
END ;
IF Failed THEN (* 7. Alternative *)
IF Heap[V1] MOD refConst = 23 THEN
V3 := Heap[V1 + 1]; GetHeap(1, V2); Heap[V2] := 24; Heap[V2 + 1] := V3; INC(Heap[V3],
refConst);

Failed := FALSE;
END ;
IF Failed THEN (* 8. Alternative *)
IF Heap[V1] MOD refConst = 24 THEN
V3 := Heap[V1 + 1]; GetHeap(1, V2); Heap[V2] := 25; Heap[V2 + 1] := V3; INC(Heap[V3],
refConst);

Failed := FALSE;
END ;
IF Failed THEN (* 9. Alternative *)
IF Heap[V1] MOD refConst = 25 THEN
V3 := Heap[V1 + 1]; GetHeap(1, V2); Heap[V2] := 26; Heap[V2 + 1] := V3; INC(Heap[V3],
refConst);

Failed := FALSE;
END ;
IF Failed THEN (* 10. Alternative *)
IF Heap[V1] MOD refConst = 26 THEN
V3 := Heap[V1 + 1]; INC(Heap[V3],  refConst);
  IF Pred3(V3, V4) THEN (* INC *)
GetHeap(1, V2); Heap[V2] := 17; Heap[V2 + 1] := V4; INC(Heap[V4],  refConst);

Failed := FALSE;
 END; (* INC *)
FreeHeap(V4);
END ;
IF Failed THEN (* 11. Alternative *)
IF V1 = 3 THEN
V2 := 6;
INC(Heap[6],  refConst);
Failed := FALSE;
END ;
END END END END END END END END END ;
FreeHeap(V1);
IF Failed THEN V2 := errVal; INC(Heap[errVal], refConst);  END;
RETURN ~ Failed
END Pred3;

PROCEDURE VisitRule0(Symbol: LONGINT; VisitNo: INTEGER);
(*
   Rule 0 : S< +Z(1)  > : A< -N(1)  >B< +N(1) , -Z(1)  >.
*)
VAR TreeAdr, VI: IndexType; S: SemTreeEntry;
BEGIN
  IF VisitNo = syntacticPart THEN
    IF NextSemTree >= LEN(SemTree^)-3 THEN ExpandSemTree END;
    TreeAdr := SemTree[Symbol].Adr;
    SemTree[Symbol].Adr := NextSemTree;
    SemTree[Symbol].Pos := PosTree[TreeAdr];
    INC(AffixVarCount, 2);
    SemTree[Symbol].VarInd := nil;
    SemTree[NextSemTree] := SemTree[Symbol];
    INC(NextSemTree);
    NEW(S);
    S.Adr := Tree[TreeAdr+1];
    S.Rule := 0 + Tree[S.Adr] MOD hyperArityConst;
    SemTree[NextSemTree] := S; INC(NextSemTree);
    NEW(S);
    S.Adr := Tree[TreeAdr+2];
    S.Rule := 2 + Tree[S.Adr] MOD hyperArityConst;
    SemTree[NextSemTree] := S; INC(NextSemTree);
    TreeAdr := SemTree[Symbol].Adr;
```

```
    Visit(TreeAdr+1, syntacticPart);
    Visit(TreeAdr+2, syntacticPart);
  ELSE
    TreeAdr := SemTree[Symbol].Adr;
    (* Visit-beginnende Analyse *)
    (* Synthese *)
    Visit(TreeAdr+1, 1);
    (* Analyse *)
    GV2 := AffPos[S1+0] ; (* komplementaere Referenzzaehlerbehandlung *)

    (* Synthese *)
    AffPos[S2+0] := GV2; (* komplementaere Referenzzaehlerbehandlung *)
    GV2 := -1;

    Visit(TreeAdr+2, 1);
    (* Analyse *)
    GV4 := AffPos[S2+1] ; (* komplementaere Referenzzaehlerbehandlung *)

    (* Visit-abschliessende Synthese *)
    AffPos[S0+0] := GV4; (* komplementaere Referenzzaehlerbehandlung *)
    GV4 := -1;

    (* Leave; VisitNo: 1 *)
  END
END VisitRule0;

PROCEDURE VisitRule1(Symbol: LONGINT; VisitNo: INTEGER);
(*
   Rule 1 : A< +"i" N(1)  > : A< -N(1)  >.
*)
VAR TreeAdr, VI: IndexType; S: SemTreeEntry;
VAR V1: HeapType;
BEGIN
  IF VisitNo = syntacticPart THEN
    IF NextSemTree >= LEN(SemTree^)-2 THEN ExpandSemTree END;
    TreeAdr := SemTree[Symbol].Adr;
    SemTree[Symbol].Adr := NextSemTree;
    SemTree[Symbol].Pos := PosTree[TreeAdr];
    INC(AffixVarCount, 1);
    SemTree[Symbol].VarInd := nil;
    SemTree[NextSemTree] := SemTree[Symbol];
    INC(NextSemTree);
    NEW(S);
    S.Adr := Tree[TreeAdr+1];
    S.Rule := 0 + Tree[S.Adr] MOD hyperArityConst;
    SemTree[NextSemTree] := S; INC(NextSemTree);
    TreeAdr := SemTree[Symbol].Adr;
    Visit(TreeAdr+1, syntacticPart);
  ELSE
    TreeAdr := SemTree[Symbol].Adr;
    (* Visit-beginnende Analyse *)
    (* Synthese *)
    Visit(TreeAdr+1, 1);
    (* Analyse *)
    GV2 := AffPos[S1+0] ; (* komplementaere Referenzzaehlerbehandlung *)

    (* Visit-abschliessende Synthese *)
    GetHeap(1, V1);
    Heap[V1] := 17;
    Heap[V1 + 1] := GV2; (* komplementaere Referenzzaehlerbehandlung *)
    GV2 := -1;
    AffPos[S1+0] := V1;
    (* Leave; VisitNo: 1 *)
  END
END VisitRule1;

PROCEDURE VisitRule2(Symbol: LONGINT; VisitNo: INTEGER);
(*
   Rule 2 : A< + > : .
*)
VAR TreeAdr, VI: IndexType; S: SemTreeEntry;
BEGIN
  IF VisitNo = syntacticPart THEN
    IF NextSemTree >= LEN(SemTree^)-1 THEN ExpandSemTree END;
    TreeAdr := SemTree[Symbol].Adr;
    SemTree[Symbol].Adr := NextSemTree;
    SemTree[Symbol].Pos := PosTree[TreeAdr];
    INC(AffixVarCount, 0);
    SemTree[Symbol].VarInd := nil;
    SemTree[NextSemTree] := SemTree[Symbol];
    INC(NextSemTree);
  ELSE
```

```
    TreeAdr := SemTree[Symbol].Adr;
    (* Visit-beginnende Analyse *)
    (* Visit-abschliessende Synthese *)
    AffPos[S1+0]  := 2; INC(Heap[2], refConst);
    (* Leave; VisitNo: 1 *)
  END
END VisitRule2;

PROCEDURE VisitRule3(Symbol: LONGINT; VisitNo: INTEGER);
(*
   Rule 3 : B< -"i" N(1) , +Z1(3)  > : B< +N(1) , -Z(1)  >INC< +Z(1) , -Z1(3)  >.
*)
VAR TreeAdr, VI: IndexType; S: SemTreeEntry;
VAR V1: HeapType;
BEGIN
  IF VisitNo = syntacticPart THEN
    IF NextSemTree >= LEN(SemTree^)-2 THEN ExpandSemTree END;
    TreeAdr := SemTree[Symbol].Adr;
    SemTree[Symbol].Adr := NextSemTree;
    SemTree[Symbol].Pos := PosTree[TreeAdr];
    INC(AffixVarCount, 3);
    SemTree[Symbol].VarInd := nil;
    SemTree[NextSemTree] := SemTree[Symbol];
    INC(NextSemTree);
    NEW(S);
    S.Adr := Tree[TreeAdr+1];
    S.Rule := 2 + Tree[S.Adr] MOD hyperArityConst;
    SemTree[NextSemTree] := S; INC(NextSemTree);
    TreeAdr := SemTree[Symbol].Adr;
    Visit(TreeAdr+1, syntacticPart);
  ELSE
    TreeAdr := SemTree[Symbol].Adr;
    (* Visit-beginnende Analyse *)
    Pos := SemTree[TreeAdr+0].Pos;
    V1 := AffPos[S2+0] ;
    IF Heap[V1] MOD refConst # 17 THEN AnalyseError(V1, "B") END;
    GV3 := Heap[V1 + 1]; INC(Heap[GV3],  refConst);
    FreeHeap(AffPos[S2+0] );
    (* Synthese *)
    AffPos[S2+0] := GV3; (* komplementaere Referenzzaehlerbehandlung *)
    GV3 := -1;

    Visit(TreeAdr+1, 1);
    (* Analyse *)
    GV4 := AffPos[S2+1] ; (* komplementaere Referenzzaehlerbehandlung *)

    (* Synthese *)
    Pos := SemTree[TreeAdr+1].Pos;
    INC(Heap[GV4],  refConst);
    Check3('INC', GV4, GV5);
    FreeHeap(GV4);
    GV4 := -1;
    (* Analyse *)

    (* Visit-abschliessende Synthese *)
    AffPos[S2+1] := GV5; (* komplementaere Referenzzaehlerbehandlung *)
    GV5 := -1;

    (* Leave; VisitNo: 1 *)
  END
END VisitRule3;

PROCEDURE VisitRule4(Symbol: LONGINT; VisitNo: INTEGER);
(*
   Rule 4 : B< -, +"0"  > : .
*)
VAR TreeAdr, VI: IndexType; S: SemTreeEntry;
VAR V1: HeapType;
BEGIN
  IF VisitNo = syntacticPart THEN
    IF NextSemTree >= LEN(SemTree^)-1 THEN ExpandSemTree END;
    TreeAdr := SemTree[Symbol].Adr;
    SemTree[Symbol].Adr := NextSemTree;
    SemTree[Symbol].Pos := PosTree[TreeAdr];
    INC(AffixVarCount, 0);
    SemTree[Symbol].VarInd := nil;
    SemTree[NextSemTree] := SemTree[Symbol];
    INC(NextSemTree);
  ELSE
    TreeAdr := SemTree[Symbol].Adr;
    (* Visit-beginnende Analyse *)
    Pos := SemTree[TreeAdr+0].Pos;
```

```
    V1 := AffPos[S2+0] ;
    IF V1 # 2 THEN AnalyseError(V1, "B") END;
    FreeHeap(AffPos[S2+0] );
    (* Visit-abschliessende Synthese *)
    AffPos[S2+1]  := 4; INC(Heap[4], refConst);
    (* Leave; VisitNo: 1 *)
  END
END VisitRule4;

PROCEDURE Visit(Symbol: LONGINT; VisitNo: INTEGER);
BEGIN
  CASE SemTree[Symbol].Rule OF
      0: VisitRule0(Symbol, VisitNo);
    | 1: VisitRule1(Symbol, VisitNo);
    | 2: VisitRule2(Symbol, VisitNo);
    | 3: VisitRule3(Symbol, VisitNo);
    | 4: VisitRule4(Symbol, VisitNo);
  END
END Visit;

PROCEDURE ^ Emit2Type3(Ptr: HeapType);

PROCEDURE Emit2Type3(Ptr: HeapType);
BEGIN
  INC(OutputSize, ((Heap[Ptr] MOD refConst) DIV arityConst) + 1);
  CASE Heap[Ptr] MOD arityConst OF
  | 1: Emit2Type3(Heap[Ptr + 1]); IO.WriteText(Out, "0");
  | 2: Emit2Type3(Heap[Ptr + 1]); IO.WriteText(Out, "1");
  | 3: Emit2Type3(Heap[Ptr + 1]); IO.WriteText(Out, "2");
  | 4: Emit2Type3(Heap[Ptr + 1]); IO.WriteText(Out, "3");
  | 5: Emit2Type3(Heap[Ptr + 1]); IO.WriteText(Out, "4");
  | 6: Emit2Type3(Heap[Ptr + 1]); IO.WriteText(Out, "5");
  | 7: Emit2Type3(Heap[Ptr + 1]); IO.WriteText(Out, "6");
  | 8: Emit2Type3(Heap[Ptr + 1]); IO.WriteText(Out, "7");
  | 9: Emit2Type3(Heap[Ptr + 1]); IO.WriteText(Out, "8");
  | 10: Emit2Type3(Heap[Ptr + 1]); IO.WriteText(Out, "9");
  | 11:
  ELSE IO.WriteInt(Out, Heap[Ptr])
  END
END Emit2Type3;



PROCEDURE Init;
BEGIN
  NEW(SemTree, initialStorageSize);
  NEW(AffPos, 6);
  NEW(Var, 8*initialStorageSize);
  NextSemTree := 0;
  NextVar := 0;
  AffixVarCount := 0;
END Init;

PROCEDURE TraverseSyntaxTree*(Tree1 : OpenTree; PosTree1 : OpenPos;
            ErrCounter : LONGINT; Adr : TreeType; HyperArity : INTEGER);
  VAR StartSymbol: IndexType; V1: HeapType;
BEGIN
  IF HyperArity # hyperArityConst THEN
    IO.WriteText(IO.Msg, "\n   internal error: 'arityConst' is wrong\n");
    IO.Update(IO.Msg); HALT(99)
  END;
  Tree := Tree1;
  PosTree := PosTree1;
  ErrorCounter := ErrCounter;
  Init;
  StartSymbol := NextSemTree;
  NEW(SemTree[StartSymbol]);
  SemTree[StartSymbol].Adr := Adr;
  SemTree[StartSymbol].Rule := -1 + Tree[Adr] MOD hyperArityConst;
  INC(NextSemTree);
  Visit(StartSymbol, syntacticPart);
  Visit(StartSymbol, 1); V1 := AffPos[S0];
  IF ErrorCounter > 0 THEN
    IO.WriteText(IO.Msg, "  "); IO.WriteInt(IO.Msg, ErrorCounter);
    IO.WriteText(IO.Msg, " errors detected\n"); IO.Update(IO.Msg)
  ELSE
      IF IO.IsOption("w") THEN IO.CreateOut(Out, "S.Out") ELSE Out := IO.Msg END;
      Emit2Type3(V1); IO.WriteLn(Out); IO.Show(Out);
  END;
    IF IO.IsOption("i") THEN
      IO.WriteText(IO.Msg, "\ttree of "); IO.WriteInt(IO.Msg, OutputSize);
      IO.WriteText(IO.Msg, " uses "); IO.WriteInt(IO.Msg, CountHeap()); IO.WriteText(IO.Msg, "
```

```
of ");
      IO.WriteInt(IO.Msg, NextHeap);  IO.WriteText(IO.Msg, " allocated, with ");
IO.WriteInt(IO.Msg, predefined + 1);
      IO.WriteText(IO.Msg, " predefined\n"); IO.Update(IO.Msg);
    END;
  IF IO.IsOption("i") THEN
      IO.WriteText(IO.Msg, "\tsemantic tree of "); IO.WriteInt(IO.Msg, AffixVarCount);
      IO.WriteText(IO.Msg, " affixes uses "); IO.WriteInt(IO.Msg, NextVar);
      IO.WriteText(IO.Msg, " affix variables, with\n\t\t");
      IO.WriteInt(IO.Msg, 0); IO.WriteText(IO.Msg, " stacks and\n\t\t");
      IO.WriteInt(IO.Msg, 5); IO.WriteText(IO.Msg, " global variables\n")
  END;
  Tree := NIL; PosTree := NIL; SemTree := NIL; Var := NIL; AffPos := NIL
END TraverseSyntaxTree;

END SEval.  (* insert module name *)
```

Anhang D: Literaturverzeichnis

[COMA] Wagner, Ripphausen-Lipa, Scheffler
Computerorientierte Mathematik II
Skript zur LV, SS 1991

[DeWe] Jochen Demuth, Stephan Weber:
Eine konzeptionelle Revision des Eta-Compilergenerators und ihre Implementierung
Diplomarbeit TU Berlin, Fachbereich Informatik, Institut für Angewandte Informatik, Dezember 1996

[DeWeKaKr] Jochen Demuth, Stephan Weber, Sönke Kannapinn, Mario Kröplin
Echte Compilergenerierung
Effiziente Implementierung einer abgeschlossenen Theorie
aus der Reihe Forschungsberichte des FB Informatik, Bericht 1997/6

[Engelfriet] J. Engelfriet:
Attribute grammars: Attribute evaluation methods
In B.Lorho, editor, Methods and Tools for Compiler Construction, pages 103-138
Cambridge University Press 1984

[EngFil] J. Engelfriet, G.Filé
Simple multi-visit attribute grammars
Journal of Computer and System Sciences, 24(3):283-314, June 1982

[EngJong] Joost Engelfriet, Willem de Jong:
Attribute Storage Optimization by Stacks
Acta Informatica 27, 568-581, 1990

[GySiMa] Gyimoth, Simon, Makey
An implementation of the HLP
in Acta Informatica 1983, 06/83

[IbaKat] T.Ibaraki, N.Katoh:
On-line computation of transitive closures of graphs.
Information Processing Letters, 16:95-97, 1983

[Kastens] U.Kastens:
Ordered Attribute Grammars.
Acta Informatica, 13(3): 229-256, 1980

[KaHuZi] U.Kastens, B.Hutten, E.Zimmermann:
GAG: A Practical Compiler Generator
Volume 141 of Lecture Notes in Computer Science
Springer Verlag 1982

[KröpKann] Mario Kröplin, Sönke Kannapinn:
Sequentiell orientierbare Attributgrammatiken
Vorabdruck, TU Berlin, Fachbereich Informatik, Institut für Angewandte Informatik, 25. Januar 1995

[Kutza] Karsten Kutza:
Evaluation geordneter EAGen im eta-Compiler-Generator
Bericht 1989/2 TU Berlin, Fachbereich Informatik, 1989

[Mehlhorn] Kurt Mehlhorn
Graph Algorithms and NP-Completeness
ETACS Series in Computer Science, Springer Verlag

[ReiWi] Martin Reiser, Niklaus Wirth:
Programmieren in Oberon: Das neue Pascal
Bonn-Paris, Addison-Wesley 1994

[RepTei] T.W.Reps, T.Teitelbaum:
The Synthesizer Generator: A System for Constructing Language-Bases Editors
Texts and Monographs in Computer Science, Springer-Verlag, 1988

[Schröer] F.W. Schröer:
Eta: Ein Compiler-Generator auf Basis zweistufiger Grammatiken
Bericht 84/2, TU Berlin, Fachbereich Informatik, März 1984

[Watt] D.A. Watt:
Analysis Oriented Two Level Grammars
Ph. D. thesis, Galsgow 1974

[ZiVoKüNa] B.Zimmermann, K.Voßloh, D.Kürbis, N.Nayeri:
Compiler-Generierung II: Spezifikationskalküle und Implementierungskonzepte
Skript einer Lehrveranstaltung an der TU Berlin WS94/95