Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH

**Technical Memo**
TM-91-10

# Tree Adjoining Grammars mit Unifikation

**Béla Buschauer, Peter Poller, Anne Schauder, Karin Harbusch**

**Oktober 1991**

**Deutsches Forschungszentrum für Künstliche Intelligenz GmbH**

Postfach 20 80
D-6750 Kaiserslautern, FRG
Tel.: (+49 631) 205-3211/13
Fax: (+49 631) 205-3210

Stuhlsatzenhausweg 3
D-6600 Saarbrücken 11, FRG
Tel.: (+49 681) 302-5252
Fax: (+49 681) 302-5341

# Deutsches Forschungszentrum für Künstliche Intelligenz

The German Research Center for Artificial Intelligence (Deutsches Forschungszentrum für Künstliche Intelligenz, DFKI) with sites in Kaiserslautern und Saarbrücken is a non-profit organization which was founded in 1988 by the shareholder companies ADV/Orga, AEG, IBM, Insiders, Fraunhofer Gesellschaft, GMD, Krupp-Atlas, Mannesmann-Kienzle, Philips, Siemens and Siemens-Nixdorf. Research projects conducted at the DFKI are funded by the German Ministry for Research and Technology, by the shareholder companies, or by other industrial contracts.

The DFKI conducts application-oriented basic research in the field of artificial intelligence and other related subfields of computer science. The overall goal is to construct *systems with technical knowledge and common sense* which - by using AI methods - implement a problem solution for a selected application area. Currently, there are the following research areas at the DFKI:

- Intelligent Engineering Systems
- Intelligent User Interfaces
- Intelligent Communication Networks
- Intelligent Cooperative Systems.

The DFKI strives at making its research results available to the scientific community. There exist many contacts to domestic and foreign research institutions, both in academy and industry. The DFKI hosts technology transfer workshops for shareholders and other interested groups in order to inform about the current state of research.

From its beginning, the DFKI has provided an attractive working environment for AI researchers from Germany and from all over the world. The goal is to have a staff of about 100 researchers at the end of the building-up phase.

Prof. Dr. Gerhard Barth
Director

# Tree Adjoining Grammars mit Unifikation

**Béla Buschauer, Peter Poller, Anne Schauder, Karin Harbusch**

DFKI-TM-91-10

Diese Arbeit wurde finanziell unterstützt durch das Bundesministerium für Forschung und Technologie (FKZ ITW-8901 8).

# Tree Adjoining Grammars mit Unifikation

Béla Buschauer, Peter Poller, Anne Schauder, Karin Harbusch

**Zusammenfassung**

Ausgehend von dem von Harbusch entwickelten Parsingalgorithmus (siehe [3]), der auf dem Grammatikformalismus *Tree Adjoining Grammar (TAG)* arbeitet, ergab sich, daß der gewählte Formalismus in Hinblick auf Kompaktheit und Handhabbarkeit beim Grammatikentwurf Verbesserungsmöglichkeiten offenließ. Wir haben die beiden Formalismen TAG und Unifikation zusammengefügt in dem Bestreben, die Vorteile von beiden zu vereinigen. Unser Ansatz wird kontrastiv zu dem bestehenden Ansatz von Vijay-Shanker (siehe [12]) vorgestellt.

Der erste Teil dieser Arbeit beinhaltet theoretische Betrachtungen. In der Einleitung erläutern wir die Ideen der Definitionen von TAG und Unifikation als Hilfsmittel zur Sprachanalyse und motivieren mit Hilfe von Beispielen die Verknüpfung der beiden Formalismen (Kapitel 1). Unsere Definition von TAGs mit Unifikation, Überlegungen und Beispiele, die uns zu bestimmten Einschränkungen der ursprünglichen Formalismen bzw. zu Entscheidungen über das Aussehen des neuen Formalismus veranlaßt haben, stellen wir im nächsten Abschnitt vor (Kapitel 2). Es folgt die Beschreibung unseres Parsingalgorithmus, der sich auf den Algorithmus für reine TAGs von Harbusch stützt, und an die Notwendigkeiten angepaßt werden mußte, die durch das Hinzufügen von Unifikation entstanden (Kapitel 3). Unsere Lösung wird als Alternative zur entsprechenden Definition von Vijay-Shanker geschildert und bewertet.

Im zweiten Teil (Kapitel 4) kann man die Einzelheiten der Implementierung nachlesen. Eine Bewertung unserer Arbeit treffen wir in Kapitel 5.

Die vorliegende Arbeit entstand im Rahmen eines Fortgeschrittenenpraktikums an der Universität des Saarlandes. Sie wurde im Rahmen des WIP-Projektes am Deutschen Forschungszentrum für künstliche Intelligenz überarbeitet.

# Inhalt


**1 Einleitung** **4**
- 1.1 Motivation am Beispiel . . . . . 4
- 1.2 Aufgabenstellung . . . . . 11
- 1.3 Lesehinweise . . . . . 11

**2 Definition des Formalismus** **13**
- 2.1 Definition der Formalismen Tree Adjoining Grammar und Unifikation . . . 13
  - 2.1.1 Formale Definition von Tree Adjoining Grammars . . . . . 13
  - 2.1.2 Formale Definition von Unifikation . . . . . 16
- 2.2 Beschreibung von TAGs mit Unifikation . . . . . 21
  - 2.2.1 Ansatz von Vijay-Shanker . . . . . 22
  - 2.2.2 Definition von TAGs mit Unifikation . . . . . 23

**3 Übertragung der Definition auf konkrete Parsingalgorithmen** **34**
- 3.1 Der Ansatz von Vijay-Shanker und Joshi . . . . . 35
  - 3.1.1 Parsing von TAGs in Reinform . . . . . 35
  - 3.1.2 Parsing von TAGs mit Unifikation . . . . . 37
- 3.2 Unser Ansatz . . . . . 38
- 3.3 Transformation in eine Normalform . . . . . 39
  - 3.3.1 Transformation in Zwei-Form . . . . . 42
  - 3.3.2 Entfernung aller $\epsilon$-Produktionen . . . . . 43
  - 3.3.3 Transformation der $\epsilon$-freien Zwei-Form in Normalform . . . . . 45
- 3.4 Parsing von TAGs ohne Unifikation . . . . . 53
- 3.5 Parsing von TAGs mit Unifikation . . . . . 59

**4 Implementierung** **62**
- 4.1 Ein- und Ausgabe . . . . . 62
  - 4.1.1 Ein- und Ausgabe und Aufbau der TAG-Grammatik . . . . . 62
  - 4.1.2 Lexikon . . . . . 67
- 4.2 Transformation in Harbusch-Normalform . . . . . 68
  - 4.2.1 Erzeugen von DAGs und DAG-Kanten . . . . . 68
  - 4.2.2 Berechnung der Vererbungsrichtung im auxiliaren Baum . . . . . 71
  - 4.2.3 Adjunktion . . . . . 81
  - 4.2.4 Transformation in 2-Form . . . . . 88

4.2.5 Epsilon-Freiheit . . . . . 96
4.2.6 Transformation der $\epsilon$-freien Zwei-Form in Normalform . . . . . 107
4.3 Beschreibung unseres Parsingalgorithmus . . . . . 119
4.3.1 Berechnung des kontextfreien Kerns der TAG . . . . . 120
4.3.2 CYK-Analyse des kontextfreien Kerns einer TAG . . . . . 123
4.3.3 Iteration zur Elimination innerster Bäume . . . . . 130
4.4 Interne Hilfsfunktionen . . . . . 133

**5 Bewertung** **135**
5.1 Vergleich mit dem Ansatz von Vijay . . . . . 135
5.2 Laufzeitanalyse . . . . . 136
5.2.1 Transformation in Normalform . . . . . 136
5.2.2 Parsing . . . . . 143
5.3 Erweiterungsmöglichkeiten . . . . . 147

# Kapitel 1

# Einleitung

## 1.1 Motivation am Beispiel

Ein wesentlicher Nachteil von Tree Adjoining Grammars ist die Notwendigkeit, für jede auftretende Alternative (etwa durch Kasus-, Numerus- oder Genus-Fälle) eine eigene Regel aufzustellen. Daraus ergab sich die Idee, TAGs mit einem weiteren Formalismus – dem Unifikationsformalismus – zu verknüpfen, der ein elegantes Beschreibungsmittel (Variablen) für Regeln dieser Art bereitstellt. Der Unifikationsformalismus erlaubt die Speicherung von Informationsmengen (Attributen und Werten) an Regelsymbolen einer kontextfreien Regel. Die Attribute können per "Unifikation" auf Übereinstimmung getestet werden. So kann man z.B. das Agreement zwischen Subjekt und Verb eines einfachen Satzes kontrollieren, ohne alle erlaubten Kombinationen aufzuzählen.

Auch Vijay-Shanker beschäftigte sich mit der Vereinigung der beiden Formalismen TAG und Unifikation. Der entscheidende Unterschied der beiden Ansätze liegt in der Wahl des Zeitpunktes, zu dem die Unifikation ausgeführt wird. Vijay-Shanker isoliert die Rekursionsmöglichkeiten der beiden Formalismen, wir dagegen integrieren sie. Auf Gemeinsamkeiten und Unterschiede zu seinem Ansatz gehen wir im folgenden noch näher ein.

Zunächst stellen wir die grundlegenden Eigenschaften der Formalismen TAG und Unifikation informal vor und motivieren ihre Verknüpfung. Charakteristische Merkmale der Tree Adjoining Grammar machen wir an einem Beispiel fest, um Unifikation als Ergänzung des Formalismus zu motivieren, bevor wir in Kapitel 2 zu den formalen Definitionen übergehen.

Der Formalismus der Tree Adjoining Grammars wurde 1975 von A. Joshi und M. Takahashi vorgestellt (siehe [5,6]). Die Strukturen einer TAG sind Bäume. Bäume haben eine Tiefe, in der komplexe strukturelle Zusammenhänge dargestellt werden können. Das wird besonders deutlich im Vergleich zu Regeln einer kontextfreien Grammatik (KFG), die aus einem Symbol (Nichtterminal) auf der linken Seite eine Folge von Symbolen auf der rechten Seite (Nichtterminale und Terminale) ableitet und im Vergleich zu einem TAG-Baum nur die Tiefe 1 hat (siehe Abbildung 1.1).

Der Name "Tree Adjoining Grammar" gibt Aufschluß über die Verknüpfungsoperation des Formalismus, die Adjunktion (ein Ineinander-Einhängen von Bäumen). Eine

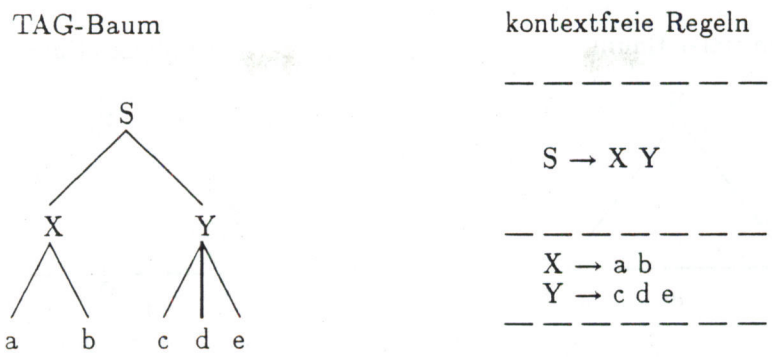



Abbildung 1.1: Vergleich von TAG-Baum und kontextfreier Regel

Menge von elementaren Bäumen bildet den Kern der Grammatik, mit ihnen darf operiert werden, d.h. sie dürfen ineinander eingesetzt werden. Man unterscheidet nach der Rolle, die Bäume bei der Adjunktionsoperation spielen, zwischen der Menge der initialen Bäume (I) und der Menge der auxiliaren Bäume (A) (siehe Abbildung 1.2).

Initiale Bäume können nicht in andere eingehängt werden. Sie bilden die Basis der Grammatikbeschreibung und können für sich stehen, da sie ein vollständiges terminales Blattwort besitzen. Die Blattfolgen initialer Bäume sind mit Terminalen (T) beschriftet und bilden Elemente der Zielsprache, alle inneren Knoten sind Nichtterminale (NT), in ihnen können andere Bäume eingesetzt werden. Zusätzlich wird von einem initialen Baum gefordert, daß er ein ausgezeichnetes Nichtterminal als Wurzel besitzt, das Startsymbol (S). Wenn wir im folgenden von Terminalen sprechen, meinen wir wie in der Computerlinguistik üblich die Präterminale, d.h. zusammenfassende Kategorienamen, die für eine größere Klasse von natürlichsprachlichen Wörtern gelten (Verben, Nomen, ...). Die Zuordnung von eigentlichem Terminal zum Präterminal erfolgt über ein Lexikon. Es ist klar, daß sich die Anzahl der Grammatikregeln drastisch reduziert, wenn nicht mehr jedes natürlichsprachliche Wort in einer Regel auftauchen muß. Den Lexikonzugriff setzen wir im folgenden immer voraus und argumentieren nur noch über die Präterminale.

Zur Beschreibung der Rekursion bei TAGs dienen die auxiliaren Bäume. Auxiliare Bäume können ineinander und in initiale Bäume adjungiert werden. Den auxiliaren Baum, der an einem Knoten X eines anderen Baumes eingehängt werden darf, identifiziert man an seiner Wurzel, die ebenfalls den Namen X tragen muß. Damit der unter X hängende Teilbaum bei der Adjunktion nicht vom Gesamtbaum getrennt wird, wird dieser an einen Knoten des eingefügten auxiliaren Baumes angehängt. Dieser Knoten muß, da er die Stelle des ursprünglichen Knotens X einnehmen soll, ebenfalls den Namen X tragen und darf keine Söhne haben. Er wird "Fußknoten" genannt.

Damit ist auch schon die Adjunktion erklärt: ein Knoten X in einem initialen oder durch Adjunktion bereits modifizierten Baum $\alpha$ wird durch einen auxiliaren Baum $\beta$ mit Wurzel und Fußknoten X ersetzt (siehe Abbildung 1.3).

Mit den Grammatikregeln und der Verknüpfungsoperation sind alle Beschreibungsmittel einer TAG definiert. Die Arbeitsweise illustrieren wir an einem Beispiel. In diesem Beispiel (siehe Abbildung 1.4) enthalten I und A jeweils nur einen Baum.

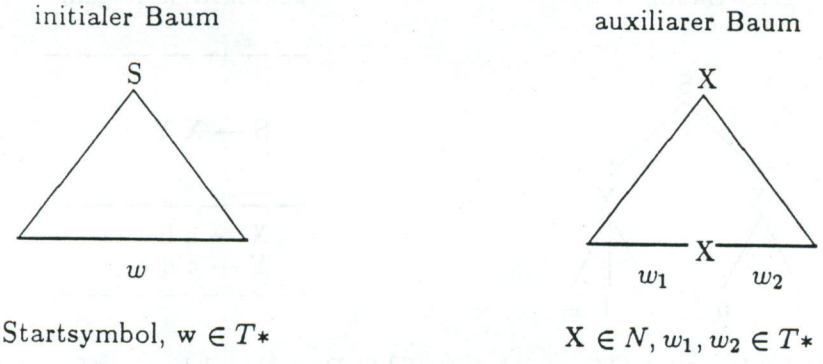



Abbildung 1.2: Initialer und auxiliarer Baum einer TAG

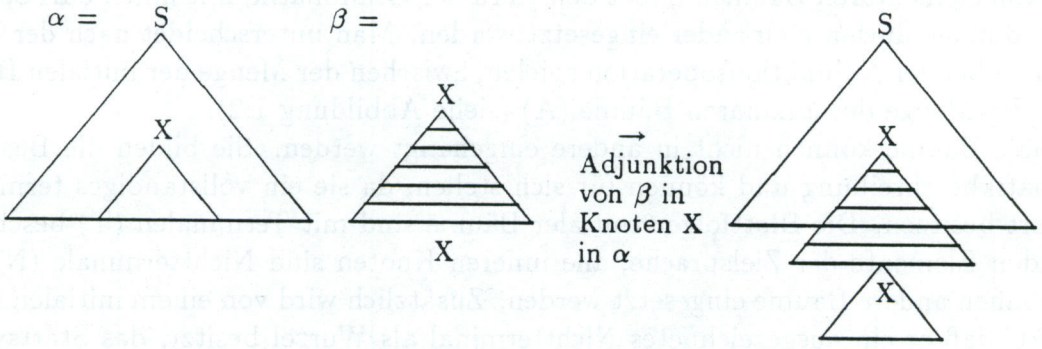



Abbildung 1.3: Adjunktion

Wurzel und Fußknoten des auxiliaren Baumes tragen als Namen die Kategorie NP (für Nominalphrase), der Baum ist also als Ergänzung der Nominalphrase in $\alpha$ gedacht, wie man nach der Adjunktion in Abbildung 1.5 sieht.

Wie man schon an diesem einfachen Beispiel sieht, sind TAGs mit ihrer Idee vom Einhängen von Bäumen in Bäume sehr gut dazu geeignet, natürlichsprachliche Strukturen zu erfassen. Die Erweiterbarkeit (wie z.B. Erweiterung der Nominalphrase um einen Relativsatz im obigen Beispiel) entspricht der Idee der Adjunktion. Rekursivität läßt sich durch wiederholtes Ineinandersetzen erreichen. Auch Sprachstrukturen wie Schachtelung und sogenannte "Cross-References" sind mit Hilfe von TAGs darstellbar. Schachtelung entspricht der Eigenschaft, daß man Unterbäume nicht vorne oder hinten anhängen muß , sondern durch Adjunktion an entsprechende Knoten "in die Mitte" und bis zu einem gewissen Grad ortsunabhängig einsetzen kann. Um geschachtelte oder gekreuzte Abhängigkeiten ("Cross-References") graphisch besser darzustellen, kann man TAGs mit Links benutzen. Hierbei können Knoten eine oder mehrere Verbindungen (Links) zu anderen Knoten im gleichen Baum haben, die auch bei der Adjunktion erhalten bleiben. Wie wir im Definitionsteil von TAGs an einem Beispiel sehen werden, ist die Mächtigkeit von TAGs schwach kontextsensitiv und bildet eine gute Näherung an die angenommene Mächtigkeit natürlicher Sprache (siehe [9]).

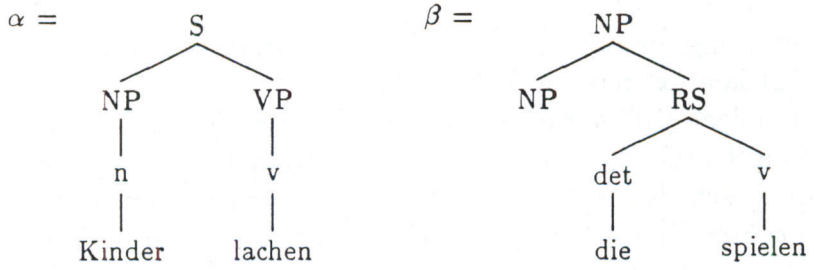



Abbildung 1.4: Beispiel einer TAG

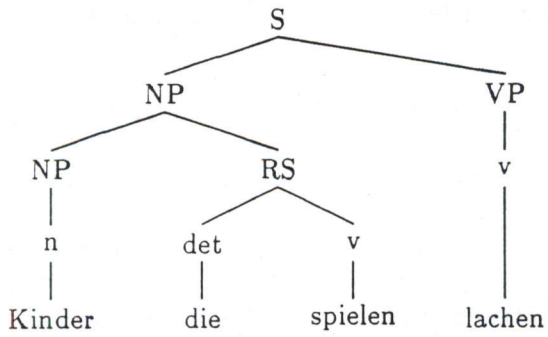



Abbildung 1.5: Beispiel einer Adjunktion

Was jedoch bei ihr wie bei vielen anderen Formalismen (angefangen bei kontextfreien Grammatiken) fehlt, sind Strukturen zur Erfassung und Verarbeitung komplexer Eigenschaften, die über die Kategorie hinausgehen. So muß man etwa für jeden Fall von unterschiedlichen Person- und Numerus-Attributen, den die Grammatik generieren soll, einen eigenen Baum aufstellen, in dem die Eigenschaften explizit in den Knotennamen festgemacht werden (siehe Abbildung 1.6).

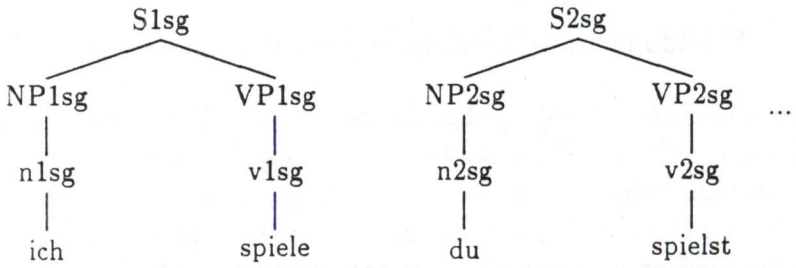



Abbildung 1.6: Explizite Repräsentation von Eigenschaften

Eleganter und kompakter wäre eine Lösung, die die TAG-Strukturen um eine Dimension erweitert, indem sie neben der Kategorie das Speichern von Eigenschaften an den Knoten und das variable Abtesten dieser Eigenschaften im Baum ermöglicht.

Im obigen Beispiel müßte man dann nur noch festlegen, was erlaubte Werte für *num* und *pers* sind und daß Numerus- und Person-Attribute im ganzen Baum übereinstimmen

sollen.

Eine Realisierung dieser Idee liefert der Unifikationsformalismus, der im folgenden in Verbindung mit kontextfreien Regeln erklärt wird.

Unifikation (siehe [10]) verarbeitet Mengen von Eigenschaften an Knoten im kontextfreien Ableitungsbaum, die sowohl mit Nichtterminalen als auch mit Terminalen bzw. Präterminalen beschriftet sein können. Diese Eigenschaftsmengen werden strukturiert dargestellt (*Feature-Value-Pfade*), hier als gerichtete azyklische Graphen (*DAG: directed acyclic graph*, siehe Abbildung 1.7). Sie besitzen eine Wurzel am jeweiligen Knoten (im Beispiel an NP) und reichen über sich verzweigende Attribut- (Feature-) Pfade bis zu den Werten (Values). Sobald sich die Attribute unterscheiden, verzweigt sich der Pfad. Gleiche Präfixe von Pfaden werden nur einmal aufgeführt (im Beispiel ist das Präfix "syntax" gemeinsam).

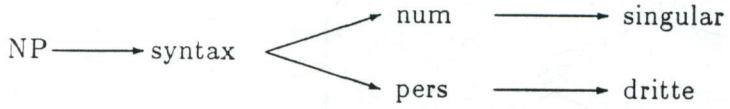



Abbildung 1.7: Ein DAG

Werden zwei Strukturen unifiziert, so bedeutet das ein Übereinanderlegen ihrer vergleichbaren Teile (über die Attributnamen). D.h. gleiche Pfadanfänge vom Ursprungsknoten aus werden verfolgt, ein Test auf Gleichheit in den auf beiden Seiten belegten Strukturteilen wird ausgeführt und eine Vereinigung der nicht vergleichbaren Teile findet statt. Sinngemäß stellt sich Unifikation wie in Abbildung 1.8 dar.

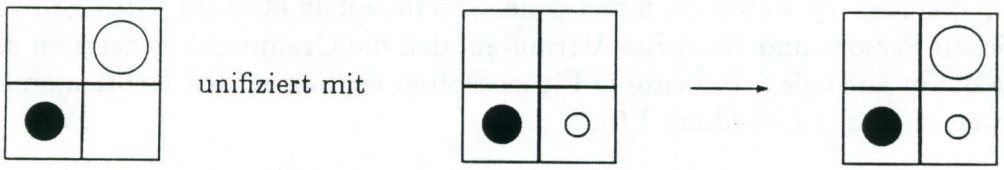



Abbildung 1.8: Charakterisierung der Unifikation

Stellt man sich beim Parsen natürlicher Sprache zwei Knoten mit umfangreichen Attributen vor, die nach Anwenden der kontextfreien Regel in einem neuen Nichtterminal zusammengefaßt werden sollen, so erscheint es wenig sinnvoll, alle Teile der Strukturen zu unifizieren und bei Erfolg weiterzuvererben. Um auf ausgewählte Teile der Attribute zugreifen zu können, werden sogenannte *Unifikationsregeln* erstellt, bei denen rechte und linke Seite unifiziert werden. Auf jeder Regelseite wird jeweils der DAG-Pfad bis zur gewünschten Unifikationsstelle angegeben. Der jeweilige Ursprungsknoten und damit die Wurzel des Pfades wird über Nummern identifiziert, die man wie in Abbildung 1.9 an die kontextfreien Regeln vergibt, d.h. die linke Seite der Regel erhält eine Null, alle Elemente der rechten Seite werden von links nach rechts aufsteigend numeriert, beginnend bei Eins.

Bei der Regelbildung beziehen wir uns auf den PATR-Formalismus (siehe [7]), bei dem sich eine Regel aus einer Konstituentenliste und einer Spezifikationsliste (Unifikati-

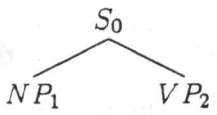


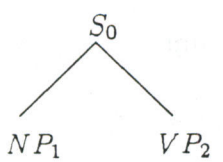

Abbildung 1.9: Nummernvergabe an Komponenten einer kontextfreien Regel

onsregel) zusammensetzt. In der Konstituentenliste werden unter Berücksichtigung der Reihenfolge die Konstituenten der kontextfreien Regel aufgeführt (linke Seite - rechte Seite), in der Spezifikationsliste die oben beschriebenen Unifikationsregeln, die nur zwei Formen annehmen dürfen: (Pfad Pfad) für zwei DAG-Pfade bis zur gewünschten Unifikationsstelle oder (Pfad Wert) für die Festlegung eines Wertes am Ende des angegebenen Pfades.

Ein Beispiel für eine Spezifikationsliste zur Konstituentenliste (S NP VP) ist:

((0 num)(2 num))
((0 pers)(2 pers))
((1 num)(2 num))
((1 pers)(2 pers))
((0 satztyp) aktiv)

In Abbildung 1.10 werden noch einmal obige Unifikationsregeln mit Erläuterungen dargestellt.




| | | |
|---|---|---|
| ((0 num) (2 num)) | } Vererbung | (1) |
| ((0 pers) (2 pers)) | | (2) |
| ((1 num) (2 num)) | } Test | (3) |
| ((1 pers) (2 pers)) | | (4) |
| ((0 satztyp) aktiv) | } Definition | (5) |

Abbildung 1.10: Unifikationsregeln mit Erläuterungen

Die Eleganz der Unifikation wird erkennbar, wenn man die verschiedenen Interpretationen betrachtet, wie sie für das Parsing an der rechten Seite aufgeführt sind.

Regel 1 und 2 stellen eine Vererbung von Attribut-Teilen aus VP an S dar, da S beim Bottom-up-Parsing noch keine Attribute trägt. Mit dem gleichen Argument kann man Regel 5 als Definition bezeichnen. Entsprechend testen Regel 3 und 4 auf Gleichheit, weil NP und VP mit Attributen belegt sind.

In Abbildung 1.11 sieht man Regelmenge und DAG am Beispiel einer fehlschlagenden Unifikation: Die Attribute "num" und "pers" von NP und VP sind wie in der Abbildung beschrieben definiert. Mit diesen Regeln simulieren wir das Lexikon, aus dem eigentlich die Werte bezogen werden sollen. Bei der Anwendung der Unifikationsregeln aus Abbildung 1.10 wird versucht, zwischen den bezeichneten Strukturpunkten Adressengleichheit herzustellen (die unifizierten Stellen werden im DAG durch eine Kante verbunden). Was in den beiden Strukturen "hinter" einer DAG-Kante steht, muß gleich oder in mindestens einer von beiden undefiniert sein, damit es nicht zum Fehlschlag der Unifikation kommt.

Ändert man im Beispiel von Abbildung 1.11 den Wert von (2 pers) zu 3, so ergibt

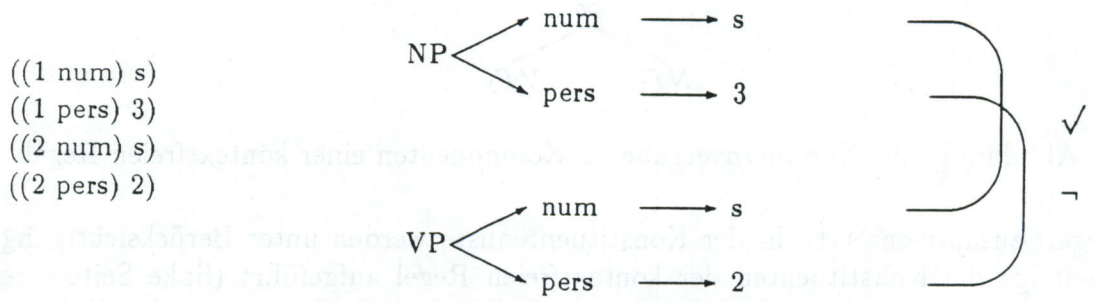



Abbildung 1.11: Beispiel für fehlschlagende Unifikation

sich ein Beispiel für eine erfolgreiche Unifikation (siehe Abbildung 1.12). Vererbte Werte werden in der neuen Struktur nicht neu aufgeführt, sondern sind über die DAG-Kante (die in beiden Richtungen lesbar ist) in anderen Strukturen nachzulesen. So kann man z.B. von S-num, von NP-num und von VP-num aus auf den gleichen Wert zugreifen und alle weiteren Manipulationen sind von allen drei Punkten aus lesbar.

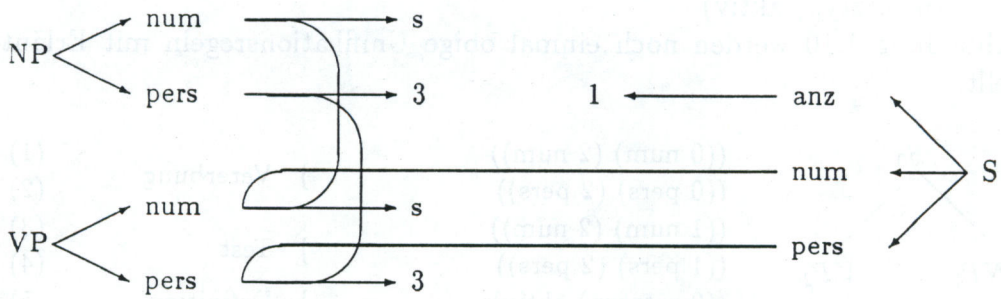



Abbildung 1.12: Beispiel für erfolgreiche Unifikation

Wie die Regelinterpretation zeigt, ist die Unifikation mit ihren gleichlautenden Regeln für verschiedene Vorgänge ein einfacher Formalismus. Darüber hinaus werden in den DAGs komplexe Eigenschaften kompakt erfaßt. Mit den DAG-Kanten können Zusammenhänge über Knoten hinweg hergestellt werden. Der Formalismus Unifikation ist Turing-mächtig.

Die erste Eigenschaft erscheint uns wünschenswert als Ergänzung der TAGs. Der Kern unserer Arbeit wird darin bestehen, das Zusammenbringen von TAG und Unifikation formal zu definieren, was im nächsten Abschnitt noch einmal motiviert wird.

Der Mächtigkeitsgewinn von TAGs mit Unifikation kann nur eingeschränkt als Vorteil betrachtet werden, da TAGs bereits über eine ausreichende Mächtigkeit verfügen. Deshalb haben wir uns vorgenommen, die Mächtigkeit der Unifikation zu verringern, so daß wir einen für unsere Definition passenden, eingeschränkten Unifikationsformalismus verwenden können. Eine weitere wichtige Anmerkung ist, daß die Monotonie der Unifikation wie in kontextfreien Grammatiken nicht erhalten werden konnte, da jetzt bereits im Baum vererbtes Wissen durch nachträgliches Einfügen eines auxiliaren Baumes zurückgenommen

(keine Vererbung des Attribute zwischen Wurzel und Fußknoten im Auxiliarbaum) oder (durch Verlängerung des Attribut-Pfades) verändert werden kann.

## 1.2 Aufgabenstellung

In diesem Bericht stellen wir unsere Definition von TAGs mit Unifikation und den dazugehörenden Parsingalgorithmus vor, sowie die Realisierung dieses Parsers in LISP.

Ziel unserer Arbeit war es, eine sinnvolle Verbindung zwischen Tree Adjoining Grammars und Unifikation als Beschreibungsmittel für Syntaxanalyse natürlicher Sprache zu finden. Dabei bestand die besondere Schwierigkeit darin, möglichst viele Vorteile beider Formalismen zu erhalten.

Unsere Lösung steht in Kontrast zum Ansatz von Vijay-Shanker: Er sieht zunächst die Ausführung aller Adjunktionen vor (wobei alle Spezifikationslisten konserviert werden) und daran anschließend die Ausführung aller Unifikationen. Bei dieser Zweiteilung setzt unsere Hauptkritik an, die zu unserem Ansatz der Integration führte. Wir nahmen dabei den Verlust anderer Eigenschaften (z.B. der Monotonie) in Kauf, was wir allerdings nicht generell als Nachteil ansehen. Näheres hierzu findet man in Kapitel 5.1 oder bei [12].

Auf der Implementationsseite mußten wir über die Definition hinaus auch den von Harbusch (siehe [3]) entwickelten Parsingalgorithmus für TAGs um Strukturen und Funktionen der Unifikation erweitern. Der Algorithmus setzt eine Grammatik in Normalform voraus, weshalb wir uns auch über eine Transformation in spezielle Normalform (in Äquivalenz zur Chomsky-Normalform), nun aber unter Berücksichtigung der Spezifikationslisten, Gedanken machen mußten.

Als Vorarbeit für die Implementierung des entwickelten Algorithmus haben wir Objekte und Operatoren der Unifikation aus D-PATR übernommen. Der von Karttunen (siehe [7]) entwickelte Formalismus wurde von Wolfgang Finkler und Günther Neumann nach Zeta-LISP übertragen und erweitert (siehe [1]) und von uns nach Common-LISP portiert. Die Definition der Grammatikstrukturen, die Umwandlung der Grammatik in unsere Normalform und der Parsingalgorithmus wurden modular entwickelt.

Bevor wir zur formalen Definition von TAGs, Unifikation und TAG mit Unifikation kommen, folgen noch einige Hinweise auf die Gliederung der Arbeit und die Zielsetzung der einzelnen Abschnitte.

## 1.3 Lesehinweise

An dieser Stelle geben wir zu jedem Abschnitt eine kurze Zusammenfassung, um dem Leser zu ermöglichen, die für ihn interessanten Fakten in diesem Bericht gezielt zu finden.

Die Arbeit gliedert sich in fünf Kapitel. Zur Motivation der Verknüpfung der Formalismen Tree Adjoining Grammar und Unifikation wurden in der Einleitung die Eignung von TAGs und Unifikation für die Strukturbeschreibung natürlicher Sprache, aber auch die Nachteile der beiden Formalismen in Reinform beschrieben.

In Kapitel 2 werden die intuitiven Beschreibungen von TAG und Unifikation formali-

siert (Abschnitt 2.1). Anschließend definieren wir in Abschnitt 2.2 TAGs mit Unifikation. Dabei geben wir zunächst eine kurze Erläuterung des Ansatzes von Vijay-Shanker, beschreiben dann ausführlich unsere Definition und ziehen schließlich einen Vergleich zwischen beiden Alternativen.

Im dritten Kapitel gehen wir auf den von uns definierten Parsingalgorithmus von TAG mit Unifikation ein, wobei wir genau wie in Kapitel 2 zunächst den Ansatz von Vijay-Shanker vorstellen. Unser Algorithmus basiert auf dem von Harbusch entwickelten $O(n^4 log\ n)$-Algorithmus für TAGs, der in Abschnitt 3.2 beschrieben und dann um den Unifikationsformalismus erweitert wird.

In Kapitel 4 folgen Einzelheiten der Implementation. Wir haben die Schritte der Normalformtransformation (die Voraussetzung für den Parsingalgorithmus ist) und den Parsingalgorithmus selbst modular entwickelt. Zu jedem Modul (das einer Datei in LISP-Code entspricht) beschreiben wir zunächst Erweiterungen der Definition von TAGs mit Unifikation, die sich aus den Notwendigkeiten der Programmerstellung ergaben. Im Abschnitt "Realisierung" schließt sich die Beschreibung der Datenstrukturen, eine Auflistung der wichtigsten Funktionen und die Erläuterung des Algorithmus an. Den Abschluß jeder Modul-Beschreibung bilden Anweisungen zur Handhabung, die ausreichen sollten, um das Programm ohne Kenntnis von Einzelheiten der Implementierung zu verwenden.

Das 5. Kapitel schließt mit einer Bewertung, einer Laufzeitanalyse und Erweiterungsmöglichkeiten unserer Arbeit als Ausblick den Bericht ab.

# Kapitel 2

# Definition des Formalismus

## 2.1 Definition der Formalismen Tree Adjoining Grammar und Unifikation

### 2.1.1 Formale Definition von Tree Adjoining Grammars

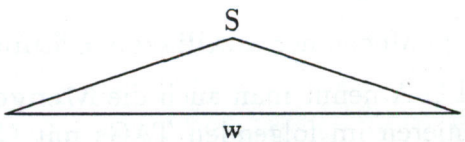
Die Regeln einer Tree Adjoining Grammar werden, wie bereits erwähnt, als Bäume dargestellt. Dabei kann man zwei verschiedene Baumtypen unterscheiden. Die Bäume jeden Typs werden in Mengen zusammengefaßt.

**Definition 1 :** Ein Baum $\alpha$ ist in **der Menge initialer Bäume I** genau dann, wenn folgendes gilt :

- Der Wurzelknoten des Baumes ist mit S, dem Startsymbol der Grammatik, beschriftet,
- jeder innere Knoten (d.h. Knoten mit ausgehenden Kanten) ist mit einem Nichtterminal beschriftet und
- jeder Blattknoten trägt ein Terminal oder $\epsilon$, das leere Wort, als Beschriftung.

Abbildung 2.1 illustriert diese Definition.

$\alpha \in$ I, der Menge initialer Bäume $\Longleftrightarrow \alpha$ :




Die Wurzel von $\alpha$ ist mit S, dem Startsymbol, beschriftet; das Blattwort w besteht aus Terminalen und alle inneren Knoten des Baumes sind mit Nichtterminalen beschriftet.

Abbildung 2.1: Definition eines initialen Baumes

**Definition 2 :** Ein Baum $\beta$ ist in **der Menge auxiliarer Bäume A** genau dann, wenn er die folgenden Bedingungen erfüllt:

- Der Wurzelknoten ist mit einem Nichtterminal beschriftet,
- jeder innere Knoten ist mit einem Nichtterminal beschriftet und
- jeder Blattknoten bis auf genau einen, den **Fußknoten**, trägt ein Terminal oder $\epsilon$ als Beschriftung.
- Die Beschriftung des Fußknotens ist X, das gleiche Nichtterminal wie die Wurzel des Baumes.
- Das Blattwort muß mindestens ein Terminal enthalten.

Man betrachte dazu auch Abbildung 2.2. Der Einfachheit halber kann man – falls keine


Abbildung 2.2: Definition eines auxiliaren Baumes

Gefahr der Verwechslung besteht – statt von Knoten $k_1$ mit Beschriftung X kurz von Knoten X sprechen.

**Definition 3 :** Eine **Tree Adjoining Grammar** (kurz : **TAG**) G ist ein 5-Tupel G = (N, T, S, I, A), wobei gilt :

- N ist die endliche Menge der **Nichtterminale**,
- T ist die endliche Menge der **Terminale**, wobei $N \cap T = \emptyset$ gilt,
- S ist das **Startsymbol** ($S \in N$),
- I ist die Menge der **initialen Bäume von G** und
- A ist die Menge der **auxiliaren Bäume von G.**

Die Menge $I \cup A$ nennt man auch die **Menge der elementaren Bäume.**

Wir definieren im folgenden TAGs mit Constraints. Diese Erweiterungsmöglichkeit spielt allerdings für die Definition von TAGs mit Unifikation keine Rolle, da wir dort keine Constraints zulassen. Wir nehmen an, daß sich die Constraints in Unifikationsregeln übersetzen lassen, und sie deshalb dem Benutzer als Beschreibungsmittel nicht zur Verfügung gestellt werden müssen. Unser Parser jedoch basiert auf einer Normalform für TAGs mit Unifikation und bei der dabei erforderlichen Transformation der Grammatik verwenden wir der Einfachheit halber Constraints.

**Definition 4 :** Eine **Tree Adjoining Grammar mit Constraints (TAGC)** ist eine TAG, wobei jedem Knoten n in jedem Baum t $\in$ (I $\cup$ A), eine Constraintmenge C(n) zugeordnet ist, die auf auxiliare Bäume verweist :

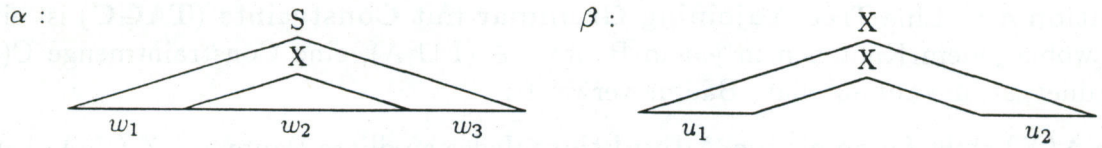
- **SA(X)** steht für **selektive Adjunktion**: Jeder auxiliare Baum aus X ($\subseteq$ A) kann in n adjungiert werden, muß aber nicht; falls die Adjunktion aller auxiliaren Bäume (deren Nichtterminal an Wurzel- und Fußknoten gleich dem Nichtterminal von n ist) erlaubt ist, darf das Constraint der Einfachheit halber weggelassen werden,
- **NA** (:= $\emptyset$) steht für **Null-Adjunktion**: Kein Baum darf im Knoten n adjungiert werden,
- **OA(X)** steht für **Obligatorische Adjunktion** Ein auxiliarer Baum aus X ($\subseteq$ A) muß adjungiert werden.

**Definition 5 :** Die **Adjunktion** als Operation zum Verknüpfen von gegebenen Bäumen $\alpha$ und $\beta$ ist wie folgt definiert :
Sei $\alpha$ ein durch eine endliche (ggf. leere) Folge von Adjunktionen aus einem initialen Baum entstandener Baum. Ferner enthalte $\alpha$ ein Nichtterminal X (X kann auch der Wurzelknoten S sein).
Der Baum $\beta$ sei ein auxiliarer Baum mit Wurzel- und Fußknoten X.
Bei der Adjunktion geschieht folgendes :
Der Knoten X in Baum $\alpha$ wird entfernt und die in diesen Knoten eingehende Kante geht nun in die Wurzel von Baum $\beta$ ein bzw. falls X die Wurzel von $\alpha$ ist, wird die Wurzel von Baum $\beta$ zur Wurzel des neuen Baumes.
Unter dem Fußknoten von $\beta$ hängt der Unterbaum, der bisher unter X in $\alpha$ hing.
Damit ist durch die **Adjunktion von Baum $\beta$ in Baum $\alpha$ im Knoten X** wieder ein Baum mit Wurzelknoten S und einem Blattwort aus Terminalen entstanden. Abbildung 2.3 verdeutlicht diese Definition.

**Definition 6 :** Eine (ggf. leere) Folge von Tripeln $(t_0, b_0, k_0)$, ... $(t_n, b_n, k_n)$ nennt man **Ableitung oder Ableitungsfolge zu w**, wobei gilt :

- $b_0$ ist ein initialer Baum $\alpha$ (*in* I),
- $k_i$ ist $\epsilon$, wenn n = i ist; sonst ist $k_i$ eine Knotennummer in $t_i$ (0 $\leq$ i $\leq$ n-1),
- $t_i$ ist der Baum $t_0$ (:= $b_0$), in dem alle Adjunktionen von Bäumen $b_j$ im Knoten $k_{j-1}$ (1 $\leq$ j $\leq$ i) stattgefunden haben,
- $b_i$ (1 $\leq$ i $\leq$ n) ist der Name eines auxiliaren Baumes, der in $t_{i-1}$ adjungiert wird und
- das Blattwort von $t_n$ ist w.

Unter dem **Blattwort** versteht man dabei die Beschriftungen der Blattknoten von links nach rechts als String gelesen. Im folgenden werden wir, wenn wir von einer Ableitung sprechen, in der Regel den Baum $t_i$ (1 $\leq$ i $\leq$ n) meinen.




S ist Startsymbol, X ∈ N, $w_1 \cdot w_2 \cdot w_3 \in T^*$, $\alpha$ kann ein initialer Baum oder ein durch Adjunktion modifizierter initialer Baum mit Wurzel S und dem Blattwort w = $w_1$ $w_2$ $w_3$ sein.

X ∈ N, $u_1 \cdot u_2 \in T^+$, $\beta$ ist ein auxiliarer Baum mit der nichtterminalen Wurzel X und dem Blattwort $u_1$ X $u_2$.

Das **Resultat der Adjunktion von $\beta$ am Knoten X in** $\alpha$ sieht wie folgt aus :

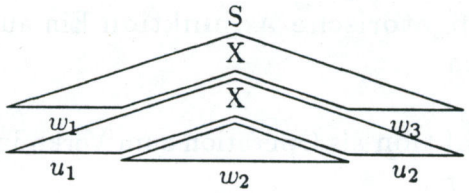



Abbildung 2.3: Definition der Adjunktion

**Definition 7 :** Die **Baummenge T(G) einer TAG G** besteht aus allen initialen Bäumen, sowie denen, die sich ausgehend von allen initialen Bäumen, mittels Adjunktion bilden lassen. Man kann die Baummenge auch über die Menge aller Ableitungen definieren. Die Baummenge besteht aus allen Bäumen, die entstehen, wenn man für alle Ableitungsfolgen die Adjunktionen expliziert.

**Definition 8 :** Die **von einer TAG G definierte Sprache L(G)** ist definiert als : L(G) := {w | w ∈ $T^*$ und w ist das Blattwort eines Baumes aus T(G)}.
Damit haben wir alle in diesem Zusammenhang benutzten Definitionen aufgelistet. Für weitergehende Informationen sei der Leser auf [3], [5] und [6] verwiesen.

### 2.1.2 Formale Definition von Unifikation

Der einfachste Formalismus auf Unifikationsbasis ist PATR, der von Shieber in [10] beschrieben wurde. PATR kodiert Informationen in Merkmalsstrukturen. Sie beschreiben Inhalte und Anforderungen an Terminale, z.B. die verlangte Kategorie, Numerus, Genus usw.

**Definition 9 :** **Merkmalsstrukturen** sind partielle Funktionen von Merkmalen (Attributen) auf Werte (Values). Die Werte können selber Merkmalsstrukturen sein (siehe Abbildung 2.4).

Eine andere äquivalente Darstellungsmöglichkeit zur Klammerform bietet der DAG (directed acyclic graph). Jede öffnende Klammer kann man als Wurzel eines Teilgraphen betrachten, von der aus Pfeile auf Werte oder neue "Unterwurzeln" weisen, wie in

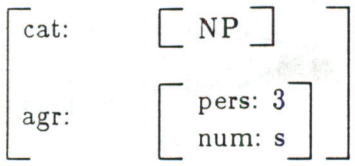



Abbildung 2.4: Merkmalsstrukturen in Klammerform

Abbildung 2.5 zu sehen.

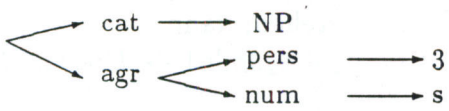



Abbildung 2.5: Merkmalsstruktur als DAG

Ein **Pfad** ist eine Folge von Attributen, mit der man einen bestimmten Teilbereich einer Merkmalsstruktur bezeichnet. Der Pfad entspricht einer Pfeilfolge im Graphen ((agr number) entspricht z.B. ⟶ agr ⟶ number).

**Definition 10 :** Besitzen zwei Attribute als Wert eine Beschreibung des gleichen Vorkommens (z.B. die Beschreibung des Subjekts eines Satzes unter dem Attribut *subj* und unter dem Attribut *pred* als "zugehöriges Subjekt"), so spricht man von **Identität** der Attribute. Man stellt sie in der DAG-Schreibweise als durchgezogene Linie, als **DAG-Kante** (Link) dar (siehe Abbildung 2.6). Der Wert und alle seine Modifikationen sind über beide Pfade (subj bzw. pred subj) zugänglich.

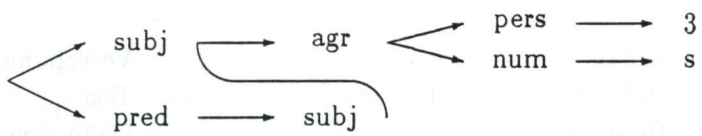



Abbildung 2.6: DAG mit DAG-Kanten

Die Idee von Unifikation als Verknüpfung zweier Merkmalsstrukturen ist folgende: Man kombiniert die Information zweier Merkmalsstrukturen, so daß man eine weitere erhält, die die Information beider beinhaltet. Das Ergebnis soll die kleinste, also allgemeinste Struktur sein, die nicht mehr Information als ihre Ausgangsstrukturen enthält. Ein Beispiel für eine so allgemeine Unifikation zeigt Abbildung 2.7.

**Definition 11 :** Das Ergebnis der **Unifikation** zweier DAGs $v_1$ und $v_2$ ist ein DAG v, wobei

(a) v = $v_1$, falls $v_1 = v_2$,

(b) v = $v_1$, falls $v_1$ nur aus einem Wert besteht und $v_2$ leer ist,

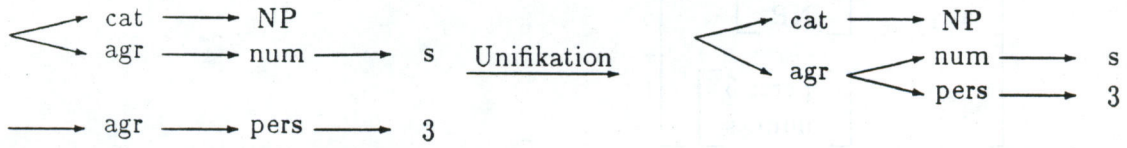



Abbildung 2.7: Idee der Unifikation

(c) v = $v_2$, falls $v_1$ leer ist und $v_2$ nur aus einem Wert besteht,

(d) falls weder $v_1$ noch $v_2$ nur aus einem Wert bestehen, dann:
∀ Attribute l, für die gilt: l → $n_1 \in v_1$, l → $n_2 \in v_2$, gilt l → $Unifikation(n_1, n_2) \in$ v und
∀ Attribute l, für die gilt: l → $n \in (n_1 \cup n_2) - (n_1 \cap n_2)$, gilt l → $n \in$ v.

(e) sonst schlägt die Unifikation fehl und der gesamte DAG wird gelöscht.

Bei der Analyse natürlicher Sprache ist es wenig zweckmäßig, unkontrolliert alle Information zu unifizieren, d.h. weiterzugeben und dabei auf Vereinbarkeit zu prüfen. Es genügt, bestimmte Übereinstimmungen der Ausgangsstrukturen zu verlangen und nur eine ausgewählte Menge an Information weiterzugeben.

**Definition 12 :** Um festzulegen, welche Teile der Strukturen unifiziert werden sollen, werden diese in **Spezifikationslisten** gleichgesetzt in der Form Pfad = Wert oder Pfad = Pfad. Ein Beispiel dafür sieht man in Abbildung 2.8.

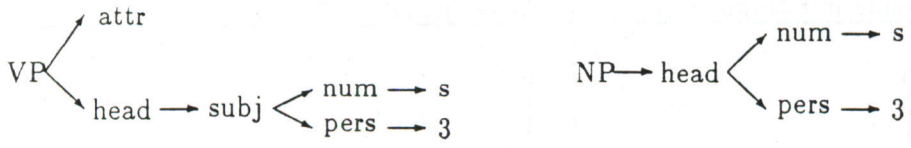
| KF-Regel | Spezifikationslisten | Bedeutung |
|---|---|---|
| $S_0 \rightarrow NP_1VP_2$ | (0 head) = (2 head) | Vererbung |
| | (0 head subj) = (1 head) | Test |
| | (0 attr) = aktiv | Definition |

Abbildung 2.8: Spezifikationslisten

In diesen Regeln wird festgelegt, daß der Satz S die head-Informationen (die die Haupteigenschaften des Satzes ausmachen) von der Verbalphrase VP erbt. Das zu VP gehörende Subjekt muß das gleiche Vorkommen bezeichnen, wie das in NP beschriebene. Unter dem Attribut "attr" wird der Satz als "aktiv" definiert.

Die notwendigen Tests werden bei DAGs durch Verfolgung der beschriebenen Pfade durchgeführt, die Definition durch Anhängen eines ausgezeichneten Pfeiles mit dem zu definierenden Wert und die Vererbung durch Anhängen des Unterbaums, jedoch nicht durch Erstellung einer Kopie sondern durch Bildung von DAG-Kanten. Eine solche DAG-Kante wird auch zwischen den getesteten DAG-Teilen gezogen, wodurch gleiche Teile nur einmal vorkommen, aber von allen beteiligten Strukturen angesprochen werden können (siehe Abbildung 2.9).

Vor der Unifikation nach Regeln aus Abbildung 2.8




Nach der Unifikation:

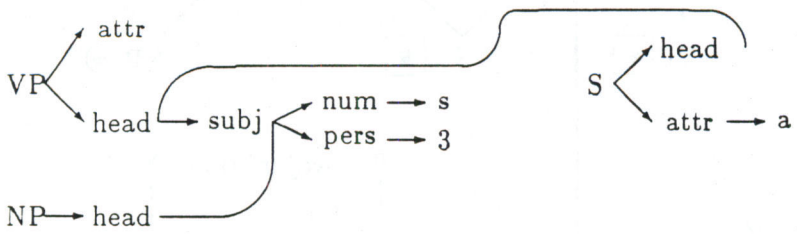



Abbildung 2.9: Unifikation nach Regeln

Bevor wir von der formalen Definition der beiden Formalismen TAG und Unifikation in Reinform zur Definition des neuen Formalismus TAG mit Unifikation übergehen, werden an dieser Stelle die wichtigsten Beobachtungen bei der Übertragung von Spezifikationslisten auf Bäume beschrieben. Damit sollen die grundlegenden Probleme bei der Verbindung der beiden Formalismen geklärt und unsere Definition motiviert werden.

Will man TAGs um Unifikation erweitern, so muß man zunächst an jedem Baumknoten die Möglichkeit dafür schaffen, daß dort Attribute abgespeichert und DAG-Kanten zu Merkmalsstrukturen anderer Knoten gezogen werden können.

In den einzelnen elementaren Bäumen läuft Unifikation wie in kontextfreien Grammatiken mit Unifikation ab, da man sich jeden elementaren Baum ebenenweise als Menge kontextfreier Regeln vorstellen kann. Dazu wird jeder innere Knoten als linke Seite und seine Söhne in ihrer Reihenfolge von links nach rechts als rechte Seite dieser Regel interpretiert (die dabei entstehende Grammatik heißt im folgenden auch kontextfreier Kern einer TAG). Bei der Adjunktion allerdings läßt sich das Verhalten nicht mehr mit dem Verhalten einer kontextfreien Regelinterpretation gleichsetzen. Beim Parsing mit kontextfreier Grammatik mit Unifikation faßt man Knotenfolgen, die man als rechte Regelseite erkennen kann, zu einem Knoten (der linken Regelseite) zusammen. Dieser Knoten und sein Attribut entstehen neu durch die Regelanwendung. Ähnlich verhält es sich beim Aufbau initialer Bäume aus dem Terminalwort. Bei der Adjunktion allerdings muß man mit zwei komplexen DAG-Strukturen umgehen, d.h. alle Knoten tragen DAGs, die jeweils untereinander verzeigert sein können. Es wird nicht mehr durch Anwendung einer Regel ein neuer DAG definiert, sondern ein Teilbaum wird in einen anderen eingesetzt, weshalb man sich Gedanken darüber machen muß , wie die DAGs der "aufeinandertreffenden" Knoten unifiziert werden sollen, so daß der Informationsfluß durch den eingefügten Teilbaum hindurch erhalten bleibt. Wir wollen diese Überlegungen noch einmal anhand eines

Beispiels (siehe Abbildung 2.10) verdeutlichen.

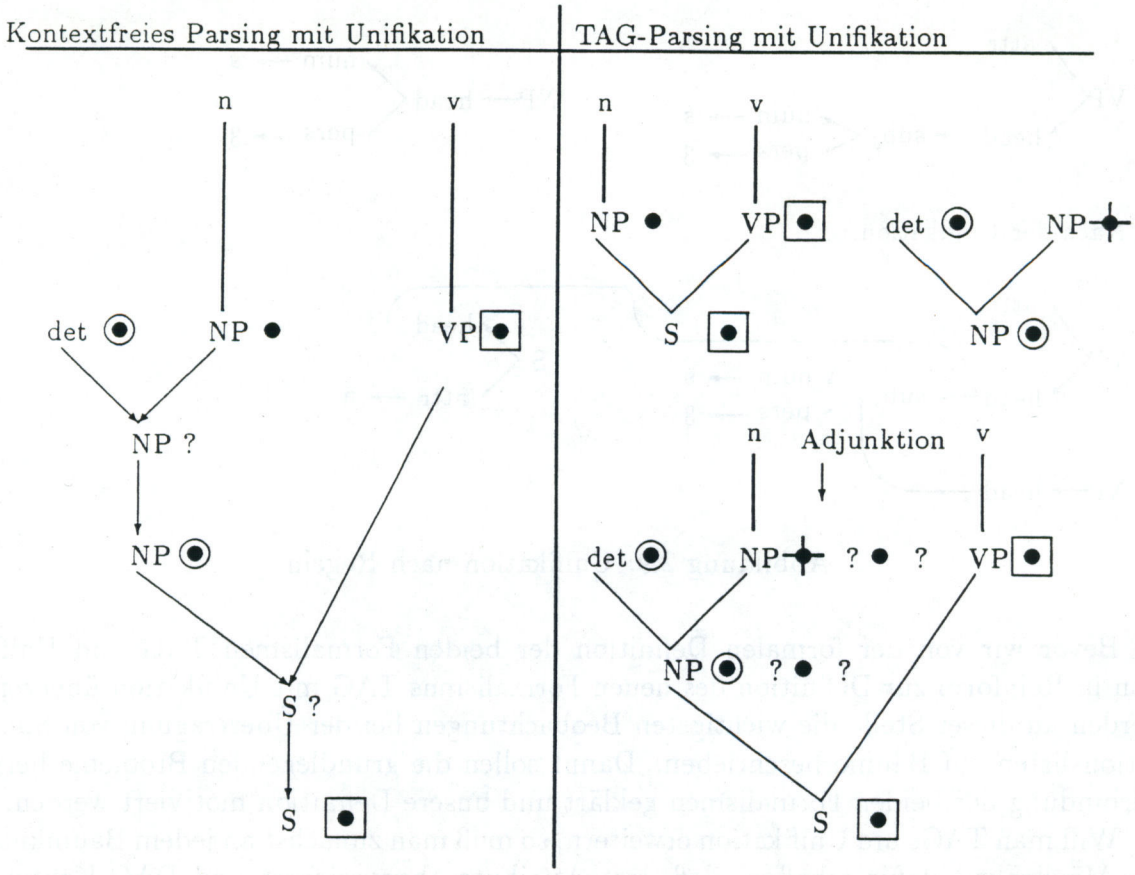



Abbildung 2.10: Probleme bei der Verknüpfung

Beim kontextfreien Parsing seien an *det*, *NP* und *VP* die einzelnen stilisierten Attribute berechnet worden. Wendet man die Regel $NP \rightarrow det\ NP$ an, so besitzt das NP auf der linken Seite noch keine Information und kann durch Anwenden der Spezifikationsregeln "aufgefüllt" werden. In unserem Beispiel heißt das, daß die Information von det vererbt werden. Das gleiche geschieht bei $S \rightarrow NP\ VP$, hier wird die Information von VP übernommen. Beim Aufbau der elementaren TAG-Bäume, die hier fertig als Ausgangsstrukturen dargestellt sind, werden ebenfalls Merkmale vererbt. Bei der Adjunktion kommt – anders als bei der Anwendung kontextfreier Regeln – kein neuer Knoten mit undefiniertem DAG hinzu, den man mit dem Ergebnis der Unifikation belegen kann.

Es ist nicht ohne weiteres entscheidbar, was mit den Attributen am Adjunktionsknoten und dem Unifikationsergebnis geschehen soll, da nicht nur kein neuer Knoten hinzukommt, sondern vielmehr durch die Ersetzung der Anhängeknoten (und damit sein DAG) wegfällt.

Daraus ergeben sich folgende Probleme für die Attribute und DAG-Kanten der betroffenen Knoten und besonders für die im Anhängeknoten definierten Werte:

1. Was geschieht mit den Attributen und DAG-Kanten? Betrachten wir dazu Abbildung 2.11.

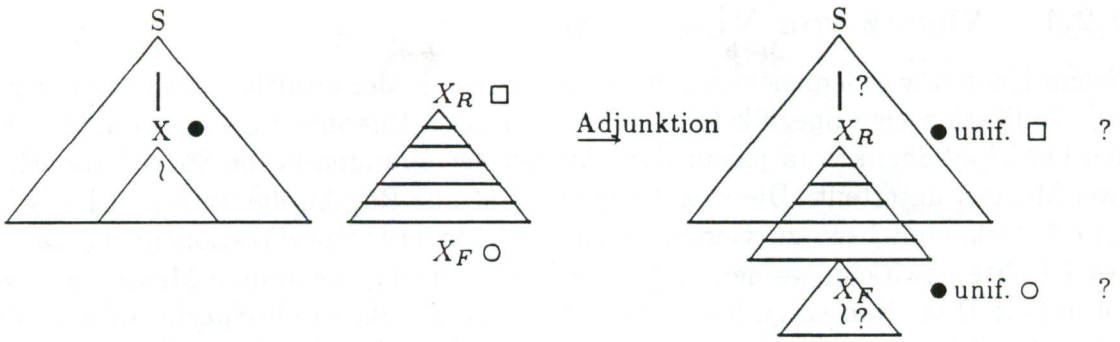



Abbildung 2.11: Was geschieht mit Attributen und DAG-Kanten?

Sollen die DAG-Kanten von X mit seinen Nachbarknoten auf $X_R$ und $X_F$ (Wurzel und Fußknoten) übertragen werden und wie? Sollen die Attribute insgesamt unifiziert werden?

2. Was geschieht mit den Werten? Betrachten wir dazu Abbildung 2.12.

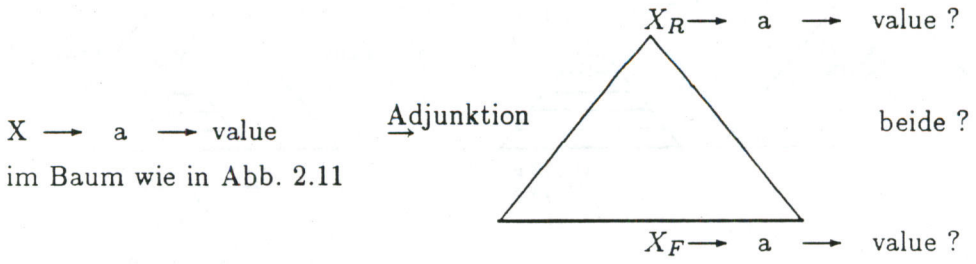



Abbildung 2.12: Was geschieht mit den Werten?

Erhalten beide Knoten den Wert oder keiner oder einer? Nach welchen Kriterien wird die Auswahl getroffen?

Damit ist klar, daß die natürliche Übertragung der Unifikation auf TAGs einige Probleme aufwirft. Diese Probleme werden in der folgenden Definition angesprochen und gelöst.

## 2.2 Beschreibung von TAGs mit Unifikation

Die Idee, TAGs und Unifikation zu einem Formalismus zusammenzufassen, hatte vor uns bereits Vijay-Shanker. Wir möchten daher, bevor wir unsere Definition von TAGs mit Unifikation vorstellen, zuerst noch zeigen, auf welche Weise Vijay-Shanker die beiden Formalismen verknüpft, um später einen Vergleich ziehen zu können.

### 2.2.1 Ansatz von Vijay-Shanker

Jedem Knoten in jedem elementaren Baum sei wie in der intuitiven Definition von TAGs mit Unifikation eine Spezifikationsliste zugeordnet. Unabhängig von Adjunktionen werden bei Vijay-Shanker an jedem Baumknoten der Grammatik die Spezifikationslisten in zwei Mengen unterteilt. Die eine Menge umfaßt alle Regeln, die im Baum DAG-Kanten zum Vaterknoten bzw. zu Geschwistern bewirken (alle Spezifikationen, die eine solche DAG-Kante bewirken, werden in ↑X zusammengefaßt), die andere Menge entsprechend solche, die DAG-Kanten zu Söhnen des Knotens bewirken (alle Spezifikationen, die eine solche DAG-Kante bewirken, werden in ↓X zusammengefaßt) . Die gleiche Unterteilung der Spezifikationslisten findet auch in unserer Definition statt. Daher werden die Bezeichnungen ↑X und ↓X formal in Abschnitt 2.2.2 eingeführt. Es ist klar, daß ↑X und ↓X zusammen mit allen Definitionen von Werten im Knoten X die gesamte Spezifikationsliste von X ergeben.

Im Beispiel aus Abbildung 2.13 sind zwei Bäume $\alpha$ und $\beta$ und das Resultat der Adjunktion von $\beta$ im Knoten $X$ des Baumes $\alpha$ gegeben.

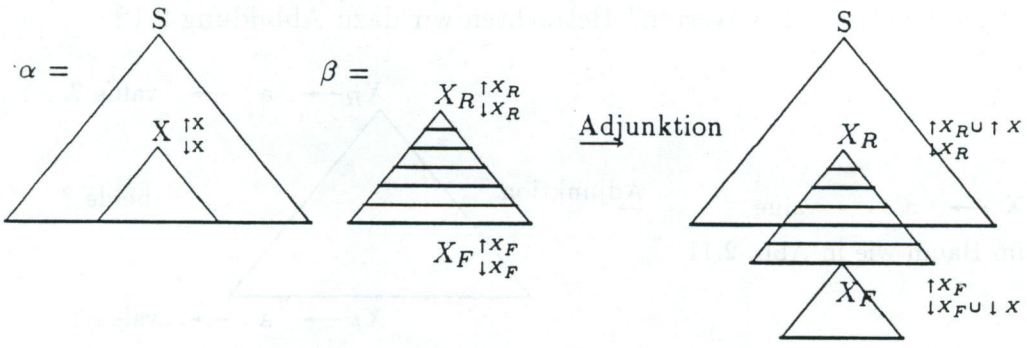



Abbildung 2.13: Ansatz von Vijay-Shanker

Zwischen den zu den Knoten eines elementaren Baumes gehörenden DAGs bestehen beim Ansatz von Vijay-Shanker keine DAG-Kanten. Genauso beschränkt sich Vijay-Shanker darauf, beim Adjungieren lediglich $\uparrow X_R$ mit $\uparrow X$ und $\downarrow X_R$ mit $\downarrow X$ zu unifizieren. Erst wenn alle Adjunktionen durchgeführt sind, werden DAG-Kanten zwischen DAGs einzelner Knoten gezogen, also der den gesamten Ableitungsbaum überspannende DAG aufgebaut.

Der Nachteil dieser Regelung ist deutlich: Unifikationsfehler treten bei der einzelnen Adjunktion nicht zutage, sondern erst, wenn die Folge der Adjunktionen abgeschlossen ist. Damit wächst der Umfang der Bäume, die als Anwärter auf ein korrektes Ergebnis in Frage kommen, stark an.

Wir haben uns deshalb dazu entschlossen, jede Adjunktion vollständig auszuführen, um Fehler sofort zu erkennen, wodurch die Zeit für jede einzelne Adjunktion natürlich zunimmt. Ein genauerer Vergleich und eine Bewertung der beiden Ansätze soll erst nach exakter Definition unserer Fassung von TAGs mit Unifikation erfolgen und ist in Kapitel 5

nachzulesen.

### 2.2.2 Definition von TAGs mit Unifikation

Eine Definition der TAG mit Unifikation muß notwendigerweise in zwei Schritten erfolgen. Zuerst definiert man die Struktur der um Spezifikationslisten erweiterten Bäume. Darauf aufbauend kann dann erst die Definition der Adjunktion mit Unifikation erfolgen. Dementsprechend folgt an dieser Stelle zunächst die Definition der Baumstrukturen.

#### Definition der Strukturen

**Definition 13 :** Sei G = (N, T, S, I, A) eine TAG gemäß Definition 3. Darüberhinaus gelte:

(a) kein Knoten aus I oder A besitzt ein Constraint,

(b) jeder Baum aus I und A besitzt eine eindeutige Nummer, die nach Definition 14 berechnet wird,

(c) die Wurzel jedes Baumes aus I und A besitzt eine eindeutige Nummer,

(d) jeder Knoten eines Baumes aus I und A besitzt eine eindeutige Nummer, die sich aus der Nummer gemäß Definition 14 berechnet und

(e) jeder Knoten eines Baumes aus I oder A besitzt eine Menge von Spezifikationslisten folgender Form:

((num path) value) oder

((num path) (num path))

wobei:

num die eindeutige Nummer des Knotens bzw. eines seiner Söhne ist.

path eine beliebig lange Folge von Elementen aus der Menge der atomaren Attributnamen und

value ein Atom ist.

**Definition 14 :** Die **Knotennummer kn zu jedem Knoten eines elementaren Baumes in einer TAG G** ist folgendermaßen eindeutig definiert :

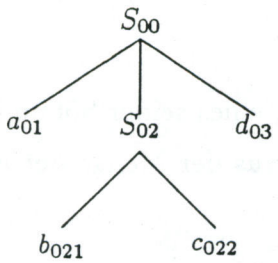
- Jeder elementare Baum erhält eine eindeutige Nummer (t), wobei die Nummerierung bei Null beginnt,
- die Wurzel eines Baumes mit der Baumnummer t erhält :
  kn := Baumnummer konkateniert mit Null (t0),

- der linke bzw. einzige Sohn der Wurzel erhält :
  kn := Baumnummer konkateniert mit Eins (t1),
- der rechte Sohn der Wurzel (falls existent) erhält :
  kn := Baumnummer konkateniert mit Zwei (t2),
- jeder linke bzw. einzige Sohn eines Knotens n (n nicht Wurzel) erhält :
  kn := Nummer des Vaterknotens konkateniert mit Eins (ti1, i $\epsilon$ {1,2}$^+$ Nummer des Vaters),
- jeder rechte Sohn eines Knotens n (n nicht Wurzel) erhält :
  kn := Nummer des Vaterknotens konkateniert mit Zwei (ti2, i $\epsilon$ {1,2}$^+$ Nummer des Vaters).

Nach dieser Notation sind die verwendeten Nummern nur dann eindeutig, wenn sie einstellig sind, d.h. Knoten einen einstelligen Outdegree haben, und die Grammatik eine einstellige Anzahl von Bäumen besitzt. In unserer Implementierung setzen wir jeweils zwischen zwei zu konkatenierende Nummern einen Punkt. Damit ist die Eindeutigkeit der Nummern ohne Einschränkung gewährleistet.

Abbildung 2.14 erläutert die Definitionen 13 und 14 an einem Beispiel. Der Baum ist der von uns entwickelten Grammatik für die Sprache L = $\{a^n b^n \ldots e^n, n \geq 1\}$ entnommen, die wir in diesem Artikel nicht vollständig vorstellen können, da sie zu umfangreich ist. Die Existenz dieser Grammatik ist allerdings ein Beweis dafür, daß TAGs, die um Unifikation erweitert werden, mächtiger sind als TAGs in Reinform. Auch die Bäume der noch folgenden Beispiele sind dieser Grammatik entnommen.

Initialer Baum mit Nummer 0




Spezifikationslisten

((00 abcd n) (02 abcd))
((00 e) (02 e))
((00 stop) (02 stop))
((00 t) (02 t))

((02 abcd) ok)
((02 e) ok)
((02 t) ok)

wobei abcd, e, stop, t und n
Attribute, ok Wert

Abbildung 2.14: Beispiel eines initialen Baumes mit Spezifikationslisten

Zur Vermeidung der "langen" Knotennummern in den Spezifikationslisten, wie sie auch in Abbildung 2.14 auftreten, ist es auch erlaubt, die Regeln an den Knoten selbst zu

schreiben und dabei die einfachere Knotennummerierung aus **SB-PATR** zu verwenden. Hier bezeichnet 0 den Knoten selbst, seine Söhne werden von 1 bis n durchnummeriert. Allerdings ist dann aber jede Regel an "ihren" Knoten gebunden, so daß TAG-Struktur und Spezifikationslisten eines Baumes nicht mehr separat betrachtet werden können.

Für $S_{00}$ in Abbildung 2.14 ergibt sich damit die vereinfachte Regelmenge der Abbildung 2.15.

```
S00: (    ((0 abcd n) (2 abcd))
          ((0 e) (2 e))
          ((0 stop) (2 stop))
          ((0 t) (2 t))          )
```

Abbildung 2.15: Vereinfachte Regelmenge für $S_{00}$

Da sich die Spezifikationslisten eines Knotens nur auf ihn selbst oder seine Söhne beziehen dürfen, kann durch sie ähnlich wie in **SB-PATR** (nur) eine knotenweise Vererbung bzw. Weitergabe von Informationen definiert werden. Die Regel ((0 *abcd n*) (2 *abcd*)) in Abbildung 2.15 z.B. besagt, daß die in $S_{002}$ unter dem Pfad *abcd* vorliegenden Information an $S_{00}$ vererbt und dort unter dem Pfad *abcd n* abgelegt werden sollen. Zur graphischen Darstellung dieser Information bietet sich wie in **SB-PATR** die DAG-Schreibweise für die einzelnen Spezifikationslisten an. Wird wie bei ((0 *abcd n*) (2 *abcd*)) in Abbildung 2.15 Information vererbt, so wird dies in den DAGs der beiden Knoten so repräsentiert, daß vom DAG des einen Knoten eine ungerichtete DAG-Kante zur entsprechenden Stelle im DAG des anderen gezogen wird (siehe Abbildung 2.16). Dabei ist es egal, in welchem DAG die Information steht, und welcher DAG auf sie per DAG-Kante zugreift, da DAG-Kanten einen Identifikationspunkt von Pfaden beschreiben und selbst keine Information enthalten. Für unser Beispiel aus Abbildung 2.14 ergibt sich der DAG aus Abbildung 2.16.

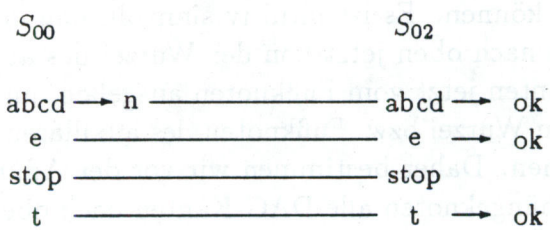



Abbildung 2.16: DAG des Beispielbaumes

Die an einem Knoten bez. eines Attributes vorliegende Information erhält man durch Verfolgen aller gerichteter Kanten (die für die Verbindungen zwischen Attributen stehen) und aller ungerichteter DAG-Kanten (die die durchgeführten Unifikationen darstellen) bis zum Endpunkt des Pfades. So liegt z.B. in Abbildung 2.16 in $S_{00}$ *abcd - n - ok* vor, wobei der Wert *ok* per DAG-Kante aus $S_{002}$ kommt. Dabei sollen Kanten, auch wenn sie aus graphischen Übersichtlichkeitsgründen einerseits links und andererseits rechts eines Knotens starten, den gleichen Ursprung haben.

In Abbildung 2.16 fällt außerdem auf, daß für *stop* noch kein Wert existiert. Dies widerspricht nicht der Definition für TAG-Bäume mit Unifikation, wohl aber den Anforderungen des Parsings, die für jedes Attribut am Ende des Parsingprozesses einen Wert verlangt. Somit wird in dem Baum aus Abbildung 2.14 die Adjunktion weiterer auxiliarer Bäume erforderlich, durch die der Wert geliefert werden muß.

Die ungerichteten DAG-Kanten in der DAG-Struktur aus Abbildung 2.16 stellen gerade die durchgeführten Unifikationen gemäß den Spezifikationslisten dar, denn ab dieser Stelle besteht die Information für beide Knoten aus dem beiden zugänglichen Unifikationsergebnis. Man kann also eine Regel wie ((0 *abcd n*) (2 *abcd*)) in Abbildung 2.15 auch folgendermaßen interpretieren: Unifiziere die an $S_{00}$ unter *abcd* - *n* und an $S_{002}$ unter *abcd* vorliegende Information zu einer einzigen und stelle sie bei erfolgreicher Unifikation beiden Knoten zur Verfügung. Theoretisch besteht hier natürlich die Möglichkeit eines "FAIL" der Unifikation, also der Unvereinbarkeit der Information. Es ist jedoch wenig sinnvoll, wenn bereits die elementaren Bäume einer Grammatik einen "FAIL" der Unifikation enthalten und damit nicht mehr korrekt sind. Viel nützlicher ist es, wenn bei der Adjunktion durch "FAIL" der Unifikation unerwünschte Fälle herausgefiltert werden können.

### Definition der Adjunktion mit Unifikation

Um zu einer sinnvollen Definition der Adjunktion mit Unifikation zu gelangen, ist es notwendig, sich die DAG-Situation des Knotens, in den adjungiert werden soll (auch *Anhängeknoten* genannt), zu veranschaulichen. Aus der Definition von TAGs mit Unifikation ergibt sich, daß über diesen Knoten Information nach oben oder unten vererbt werden kann, d.h. daß DAG-Kanten nach oben bzw. unten bestehen. Das Problem dabei ist, wie mit ihnen verfahren werden soll, wenn der Knoten bei der Adjunktion entfernt und durch einen auxiliaren Baum ersetzt wird. Es läßt sich offenbar nicht vermeiden, bereits bestehende DAG-Kanten aufzubrechen und neu zu setzen, um den auxiliaren Baum einhängen zu können. Es ist intuitiv sinnvoll, daß dabei ehemalige DAG-Kanten des Anhängeknotens nach oben jetzt von der Wurzel des auxiliaren Baumes und ebenso DAG-Kanten nach unten jetzt vom Fußknoten ausgehen, und dabei in Abhängigkeit von der DAG-Situation in Wurzel bzw. Fußknoten des auxiliaren Baumes vor der Adjunktion modifiziert sein können. Daher bestimmen wir vor der Adjunktion gemäß der folgenden Definition 15 im Anhängeknoten alle DAG-Kanten nach oben bzw. unten und außerdem auch alle dort definierten Werte, da zunächst unklar ist, an welcher Stelle sie nach der Adjunktion im eingehängten auxiliaren Baum stehen sollen.

**Definition 15:** Sei **X** ein Knoten in einem initialen oder einem durch Adjunktion bereits modifizierten initialen Baum $\alpha$. Für den Knoten X werden folgende Mengen definiert:

($\uparrow$**X**) := Menge aller Spezifikationen, in denen die Nummern (und damit die DAGs) des Vaters oder der Geschwister von X zu X in Bezug stehen.

($\downarrow$X) := Menge aller Spezifikationen von X, in denen num(X) und die Nummer eines seiner Söhne auftritt.

($\circ$X) := Alle Wert-Definitionen an X.

Nun können wir die Adjunktion mit Unifikation formal definieren.

**Definition 16 :** Sei X ein Anhängeknoten in einem initialen oder einem durch Adjunktion bereits modifizierten initialen Baum $\alpha$, $\beta$ ein auxiliarer Baum mit Wurzel $X_R$ und Fußknoten $X_F$, der im Anhängeknoten adjungiert werden soll. Die Adjunktion von $\beta$ in $\alpha$ im Knoten X mit Unifikation ist folgendermaßen definiert:

- Entferne X aus dem Baum $\alpha$.
- Entferne alle DAG-Kanten, die von X ausgehen, d.h. alle Spezifikationen aus ($\uparrow$X) und ($\downarrow$X).
- Entferne alle in X definierten Werte, d.h. alle Wertdefinitionen aus ($\circ$X).
- Setze für X den auxiliaren Baum $\alpha$ gemäß Definition 4 ein.
- Bestimme DAG($X_R$) durch Unifikation von ($\uparrow$X) und ($X_R$), wobei ($X_R$) aus allen Spezifikationen besteht, in denen num($X_R$) vorkommt.

  Dadurch entstehen alle DAG-Kanten, die der Vater oder die Geschwister von X zu X hatten, jetzt zu jeder von der Wurzel des auxiliaren Baumes $\alpha$ aus zugänglichen Information.
- Bestimme DAG($X_F$) durch Unifikation von ($\downarrow$X) und ($X_F$), wobei ($X_F$) aus allen Spezifikationen besteht, in denen num($X_F$) vorkommt.

  Dadurch entstehen alle DAG-Kanten, die X zu seinen Söhnen hatte, jetzt vom Fußknoten des auxiliaren Baumes $\alpha$ aus.
- Setze alle definierten Werte aus ($\circ$X) nach einer Vererbungsuntersuchung im auxiliaren Baum gemäß folgender Definition 17 hinter den längsten expliziten Pfad, unter dem der Wert benutzt werden kann.

Durch die Bestimmungsvorschrift für DAG($X_R$) erhält man gerade die alten Verbindungen (DAG-Kanten) von X zu seinem Vater, jetzt aber ausgehend von der an $X_R$ zugänglichen Information. Analog enthält DAG($X_F$) die alten Verbindungen (DAG-Kanten) von X nach unten, jetzt aber ausgehend von der an $X_F$ zugänglichen Information.

Hier tritt der Fall ein, daß bereits bestehende DAG-Kanten aufgebrochen und neu gesetzt werden, was bei der Unifikation in Reinform unmöglich ist. Da die neuen DAG-Kanten bis auf die Tatsache, daß sie jetzt auf die im eingehängten auxiliaren Baum vorliegende Information zugreifen, den alten entsprechen, kann man sich jedoch die Adjunktion als nachträgliches "Einschieben" eines Baumes und seiner DAGs vorstellen. Dabei ist

es gleichgültig, auf welche Weise im auxiliaren Baum Vererbungsschritte entlang dessen Knoten stattfinden. Für das Einhängen des Baumes sind lediglich die "Nahtstellen" von Interesse, da nur hier neue DAG-Kanten gesetzt werden müssen. Alle Vererbungen, die innerhalb des auxiliaren Baumes stattfinden, wurden in ihm selbst definiert und werden bei der Adjunktion nicht verändert. Die einzige Änderung, die über die Nahtstelle zwischen dem eingefügten auxiliaren Baum und dem ursprünglichen Baum hinausgeht, ist das Einfügen der Wertdefinitionen des Anhängeknotens. Dazu findet eine Vererbungsuntersuchung im auxiliaren Baum statt. Ziel dieser Vererbungsuntersuchung ist es, jeden hinter einem Pfad aus Attributen an X definierten Wert so in den eingehängten Baum einzusetzen, daß alle Pfade, unter denen der Wert benutzt werden kann, diesen auch erhalten. Dazu sucht man den passenden Pfad innerhalb des eingehängten auxiliaren Baumes und setzt den Wert dahinter. Die anderen Pfade greifen dann i.a. automatisch über ihre DAG-Kanten in diesen Pfad auf den Wert zu. Das muß nicht immer an der Wurzel des eingehängten auxiliaren Baumes sein.

Bevor wir zur eigentlichen Definition des Verfahrens kommen, möchten wir zunächst eine detaillierte Beschreibung geben.

Zur Untersuchung wird zwischen je zwei Knoten entlang des Weges von der Wurzel zum Fußknoten des auxiliaren Baumes die Vererbung bez. des gegebenen Anfangspfades betrachtet und durch einen entsprechenden Vererbungspfeil gemäß Abbildung 2.17 vermerkt. In ihr werden vier verschiedene Vererbungsschritte betrachtet, die sich aus der Relation der beiden DAGs bis zur Unifikationsstelle ergeben und durch ein einfaches Beispiel veranschaulicht sind. Besteht zwischen zwei DAGs bez. des untersuchten Anfangspfades keine DAG-Kante, so wird auch kein Vererbungspfeil notiert. Die Untersuchung wird in einem solchen Fall von der Wurzel aus nach unten zum Fußknoten und vom Fußknoten aus nach oben zur Wurzel nur bis zu einer solchen Stelle, an der keine Verbindung mehr besteht, durchgeführt.

Die Pfeilrichtung der Vererbungspfeile in Abbildung 2.17 markiert dabei die gewünschte Wertstelle für je zwei untersuchte Knoten, falls sie eindeutig festzulegen ist.

Um nun die Wertstelle innerhalb des ganzen Baumes herauszufinden, versucht man, die gesammelten Vererbungspfeile paarweise gemäß Abbildung 2.18 zu vereinigen, um am Schluß möglichst nur noch einen Pfeil (auch *Restpfeil* genannt) übrig zu behalten, durch den dann die Stelle des Wertes entsprechend seiner Richtung festgelegt ist.

Treffen $\uparrow$ und $\downarrow$ direkt aufeinander, so ist in dieser Situation kein Kürzen mehr möglich und man hat das Ende des zu untersuchenden Weges erreicht.

Mit diesen vorab erläuterten Schritten können wir die Vererbungsuntersuchung im auxiliaren Baum definieren.

**Definition 17 :** Unter der **Vererbungsuntersuchung** in einem auxiliaren Baum bez. eines wertdefinierenden DAG-Pfades versteht man die knotenweise Untersuchung der Vererbungsschritte bez. dieses Pfades entlang des Weges von der Wurzel zum Fußknoten des auxiliaren Baumes (gemäß Abbildung 2.17) und die Vereinigung der notierten Vererbungspfeile zu einem oder mehreren Restpfeilen (gemäß Abbildung 2.18), die dann angeben, an welchen Knoten des bei einer Adjunktion eingehängten auxiliaren Baumes ein für diesen

| Relation | Beispiel | Vererbungspfeil |
|---|---|---|
| X Präfix Y | X: a → b<br>Y: a → b → c | ↑ |
| Y Präfix X | X: a → b → c<br>Y: a → b | ↓ |
| Gleichheit | X: a → b<br>Y: a → b | ↕ |
| Ungleichheit | X: a → b<br>Y: a → d | ↕ |

wobei X, Y Knoten und a, b, c, d Attribute

Abbildung 2.17: Tabelle der Vererbungspfeile

$P \in \{\uparrow, \downarrow\}$
$E = \updownarrow$

PP → P
PE → P
EP → P
EE → E

Abbildung 2.18: Vereinigungen von Vererbungspfeilen

Pfad im Anhängeknoten definierter Wert (gemäß Abbildung 2.19) gesetzt wird.

Für jeden untersuchten Weg bleiben also maximal zwei Pfeile übrig. Sie legen für jeden Weg fest, an welche(n) Stelle(n) die Wertdefinition erfolgt. In Abbildung 2.19 sind die möglichen Fälle zusammengefaßt.

| Restpfeile | Knoten für Werte |
|---|---|
| x<br>↑<br>x | oberster Knoten des Pfades |
| x<br>↓<br>x | unterster Knoten des Pfades |
| x x<br>↓ ↑<br>x x | Anfangs- und Endknoten des Pfades |
| ↓<br>x<br>↑ | Knoten mit Wechsel der Vererbungsrichtung |

Abbildung 2.19: Wertstellen nach Vererbungsuntersuchung

Mit der Bestimmung der Knoten, an denen jetzt die Wertdefinition erfolgen kann ist die Vererbungsuntersuchung und damit auch die Adjunktion mit Unifikation vollständig beschrieben.

Betrachten wir abschließend ein Beispiel für eine Adjunktion. Dazu gehen wir von den Bäumen aus Abbildung 2.20 und Abbildung 2.21 aus. Auch sie stammen aus unserer Grammatik für $\{a^n b^n \ldots e^n, \text{ n} \geq 1 \}$. Die definierten Spezifikationen dienen dazu, die a's und b's nach oben sowie die c's und d's nach unten mitzuzählen, was durch die Anzahl der $n$ geschieht.

Es soll $(\beta)$ in $S_{02}$ adjungiert werden :

Nach Definition bestimmen wir zunächst die Mengen $(\uparrow S_{02})$, $(\downarrow S_{02})$ und $(\circ S_{02})$ :

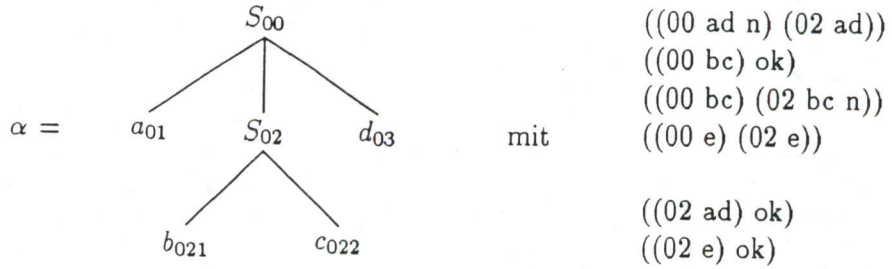
$(\uparrow S_{02})$ = { ((00 ad n)(02 ad)), ((00 bc)(02 bc n)), ((00 e)(02 e)) }

$(\downarrow S_{02})$ = { }

$(\circ S_{02})$ = { ((02 ad) 0), ((02 e) 0) }

Da in $(\circ S_{02})$ die Werte für Pfade aus je einem Attribut *ad* bzw. *e* stehen, müssen wir eine Vererbungsuntersuchung in $(\beta)$ bez. dieser beiden Attribute durchführen. Sie ist in Abbildung 2.22 veranschaulicht.

Die Berechnung auf dem Pfad von der Wurzel zum Fußknoten und umgekehrt ergibt, daß der Wert für das Attribut *ad* am Knoten $S_{1022}$ und der Wert für das Attribut *e* an den Knoten $S_{10}$ und $S_{1022}$ definiert werden muß. Jetzt können wir gemäß unserer Definition adjungieren. Die Knotennummern der Knoten des eingehängten auxiliaren Baumes




ergibt folgende DAG-Struktur:

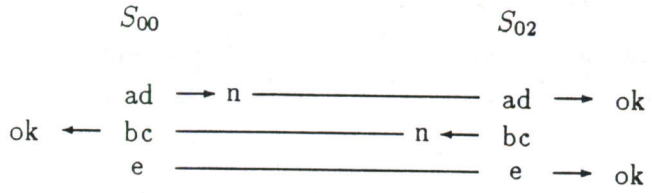
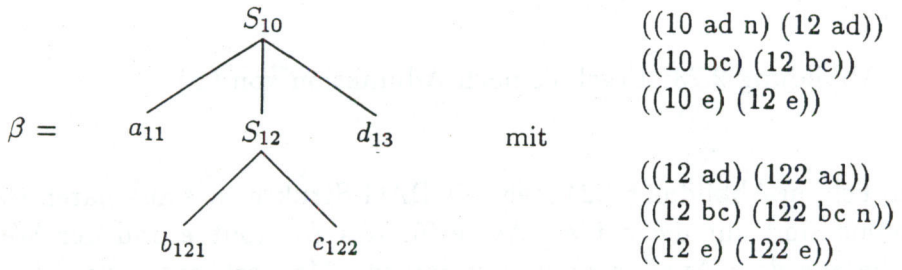



Abbildung 2.20: Initialer Baum mit Spezifikationslisten und DAG-Struktur




ergibt folgende DAG-Struktur:

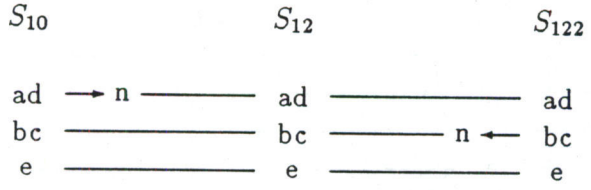



Abbildung 2.21: Auxiliarer Baum mit Spezifikationslisten und DAG-Struktur

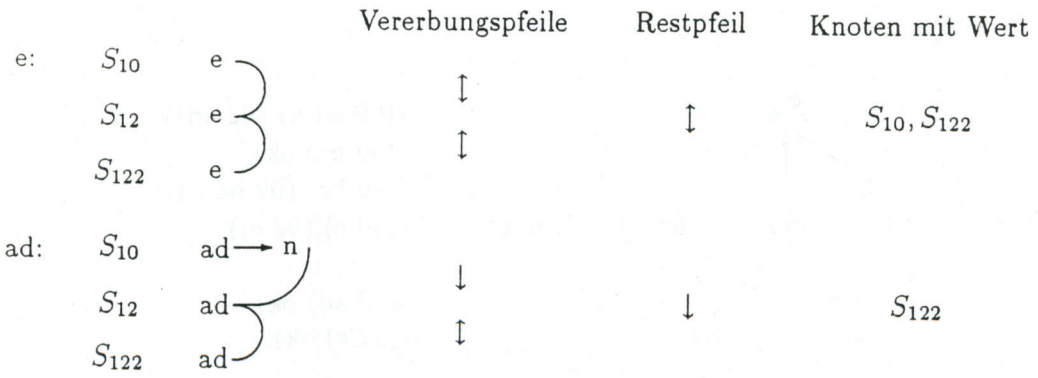



Abbildung 2.22: Vererbungsuntersuchung in $(\beta)$

erhalten als Präfix die in Klammern gesetzte Knotennummer des Anhängeknotens. Es ergibt sich der Baum aus Abbildung 2.23. Dieser Baum besitzt die DAG-Struktur in

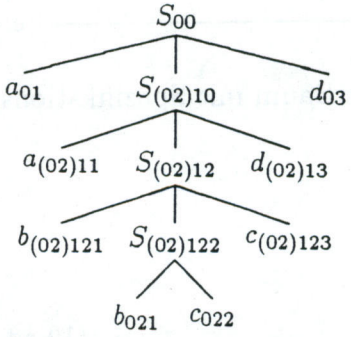



Abbildung 2.23: Ergebnis nach Adjunktion von $(\beta)$

Abbildung 2.24.

Im umrandeten Feld in Abbildung 2.24 steht die DAG-Struktur des auxiliaren Baumes $(\beta)$. Hinzugekommen sind nur die beiden Werte für das Attribut *e* und der Wert für das Attribut *ad*, die aus dem Anhängeknoten stammen. Man erkennt: Die Strukturen selbst werden nicht verändert, sondern lediglich der neuen Situation nach der Adjunktion

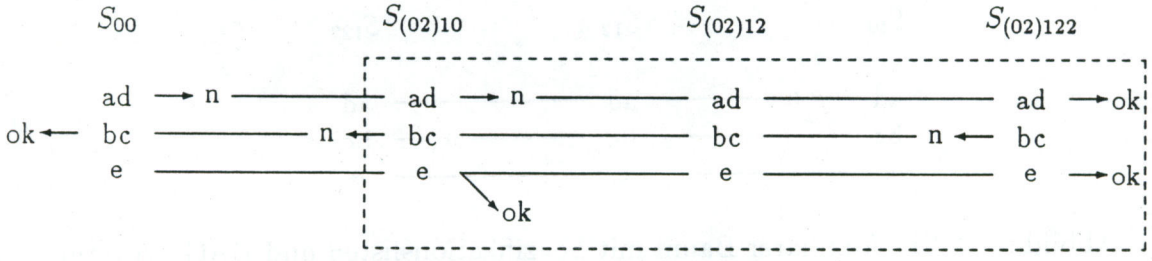



Abbildung 2.24: DAG-Struktur nach Adjunktion von $(\beta)$

angeglichen. Trotzdem kann sich dadurch die Information der Knoten verändern. $S_{00}$ z.B. hatte vor der Adjunktion den Pfad $ad \rightarrow n \rightarrow 0$ und jetzt $ad \rightarrow n \rightarrow n \rightarrow 0$. Das zweite $n$ stammt dabei aus dem eingehängten auxiliaren Baum. Die Adjunktion hat also in diesem Fall die Information an $S_{00}$ um das "Wissen" von $(\beta)$ erweitert, da es nach oben vererbt wird. Genau umgekehrt ist es für $S_{(02)122}$ . Als ehemaliger Knoten $S_{122}$ hatte er für das Attribut $bc$: $bc \rightarrow n$, allerdings noch keinen Wert. Nach der Adjunktion hat der Knoten $S_{(02)122}$ für $bc$: $bc \rightarrow n \rightarrow n \rightarrow 0$, wobei das zweite $n$ und der Wert aus dem initialen Baum stammen und nach unten vererbt wurden. Es ist also auch umgekehrt möglich, die im auxiliaren Baum anfangs vorliegende Information zu erweitern.

Unsere Definition gibt dem Grammatikschreiber die Möglichkeit, Informationsweitergabe in beliebiger Richtung und an beliebiger Stelle zu konstruieren. Diese Möglichkeit besteht allerdings auch im kontextfreien Fall bei Hinzunahme der Unifikation. Von daher stellt sich die Frage, worin der Vorteil von TAGs mit Unifikation besteht. Ein Argument ist der Gewinn an Mächtigkeit, der über TAGs in Reinform hinausgeht. Wie bereits erwähnt kann man die Bäume aus Abbildung 2.20 und Abbildung 2.21 zu einer Grammatik für die Sprache $\{a^n b^n \ldots e^n, \text{ n} \geq 1 \}$ erweitern. Dazu benötigt man einen weiteren auxiliaren Baum der das Terminal e als Blattwort hat und in seinem DAG den Pfad $e$ um das Attribut $n$ verlängert. Dieser Gewinn an Mächtigkeit kann aber nicht das entscheidende Argument für die Erweiterung der TAGs um Unifikation sein, denn im Mittelpunkt unseres Interesses steht die Aufgabe, einen Formalismus für die Analyse natürlicher Sprache zu entwickeln. Dabei haben TAGs entscheidende Vorteile gegenüber kontextfreien Grammatiken, da sprachliche Phänomene in ihnen wesentlich einfacher formuliert werden können, weil ganze Bäume als elementares Beschreibungsmittel dienen. Die Unifikation dient zur Beseitigung des gemeinsamen Nachteils von TAGs und kontextfreien Grammatiken, nämlich daß sich keine über die Kategorien hinausgehenden Informationen auf einfache Weise formulieren lassen. Unsere Definition erfüllt von daher den Anspruch, daß man alle Ausdrucksmöglichkeiten von Unifikation – von den kontextfreien Regeln auf elementare Bäume übertragen – besitzt und das Unifikationsergebnis analog zum kontextfreien Fall interpretiert werden kann.

Der nächste Schritt nach der Definition ist es nun, diese auf einen konkreten Parsingalgorithmus zu übertragen. Der einfachste Weg dazu ist, einen bereits bestehenden Parser für TAGs in Reinform möglichst genau zu übernehmen und Unifikation ohne weitgehende Erweiterungen in ihn zu integrieren. Auf diesem Weg werden wir von dem von Harbusch entwickelten Parser (siehe [3]) für TAGs in Reinform zu unserem Parser für TAGs mit Unifikation gelangen. Um dem Leser auch in diesem Punkt eine Vergleichsmöglichkeit zu bieten, werden wir zunächst noch auf den von Vijay-Shanker und Joshi entwickelten Parser für TAGs in Reinform (siehe [5]) eingehen und seine Erweiterungsmöglichkeiten für den Einbau von Unifikation aufzeigen.

# Kapitel 3

# Übertragung der Definition auf konkrete Parsingalgorithmen

Obwohl der Grammatikformalismus der TAGs noch relativ neu ist, sind bereits einige Parsingalgorithmen für TAGs in Reinform entwickelt und implementiert worden. Ein Hauptkriterium der Auswahl eines Parsingalgorithmus für unsere Definition von TAGs mit Unifikation war die Laufzeitkomplexität. Weiter sollte ein bereits existierender Unifikationsformalismus einfach in den Algorithmus integrierbar sein, ohne daß sich seine Laufzeiteigenschaften dadurch wesentlich ändern.

Wir haben den von Harbusch im Rahmen ihrer Dissertation entwickelten Parsingalgorithmus für TAGs (siehe [3]) (von jetzt ab Harbusch-Algorithmus) zur Grundlage unseres Fortgeschrittenenpraktikums gemacht. Bisher war in der Literatur der Algorithmus von Vijay-Shanker und Joshi (siehe [12]) als derjenige mit der besten Laufzeitkomplexität bekannt, nämlich $O(n^6)$. Die Vorteile des Harbusch-Algorithmus liegen in seiner deutlich besseren Laufzeitkomplexität von $O(n^4 \; log \; n)$ begründet. Bei beiden Ansätzen ist die Integration eines bestehenden Unifikationsformalismus in den Parsingalgorithmus für TAGs in Reinform in ähnlicher Weise realisierbar, wie wir später noch sehen werden. Die Grundstrukturen der beiden Algorithmen werden durch die Erweiterung um Unifikation nicht geändert, da nur lokal Funktionsaufrufe zur Durchführung und Verwaltung der Unifikation hinzugefügt werden.

Im nächsten Abschnitt werden noch einmal die Hauptmerkmale des Ansatzes von Vijay-Shanker und Joshi in knapper Form herausgearbeitet, um dem Leser die Vorzüge des Harbusch-Algorithmus, auf den in Abschnitt 3.4 eingegangen wird, deutlich zu machen. Die exakte Beschreibung des Parsingalgorithmus von Vijay-Shanker und Joshi findet sich in [12].

Anschließend wird in Abschnitt 3.2 unser Ansatz zum Parsing von TAGs mit Unifikation aufbauend auf dem Harbusch-Algorithmus vorgestellt. Wie in Abschnitt 3.1 werden wir versuchen, die grundlegenden Ideen des Algorithmus zu vermitteln. Danach wird auf Erweiterungen des Algorithmus, die sich durch unsere Definition von TAGs mit Unifikation ergeben haben, konkret eingegangen.

Die Implementierung unseres Ansatzes wird in einem eigenen Kapitel gesondert behandelt. Diese Zweiteilung in abstrakte Betrachtungsweise zum einen und Beschreibung der Implementierung zum andern ist von uns bewußt gewählt. Das abstrakte Niveau der Beschreibung vermittelt dem Leser sehr schnell ein intuitives Verständnis der sinnvollen Arbeitsschritte. Die Vorzüge des Harbusch-Algorithmus als Grundlage sind leicht zu erkennen, ohne daß man den Leser mit vielen Implementierungsdetails belasten muß. Der Leser wird dann in Kapitel 4 der Beschreibung der Implementierung leichter folgen können.

## 3.1 Der Ansatz von Vijay-Shanker und Joshi

Wir werden zuerst die Grundidee des Parsingalgorithmus von Vijay-Shanker und Joshi (siehe [12]) für TAGs in Reinform vorstellen. Auf Erweiterungen des Algorithmus in Bezug auf Unifikation wird anschließend eingegangen.

### 3.1.1 Parsing von TAGs in Reinform

Beide in diesem Kapitel vorgestellten Algorithmen bauen grundsätzlich auf dem Cocke-Younger-Kasami-Algorithmus (CYK-Algorithmus bzw. CYK-Analyse) (siehe [13]) zum Parsing von kontextfreien Grammatiken auf.

In der Einleitung haben wir die Definition von TAGs ausgehend von kontextfreien Grammatiken motiviert. Umgekehrt kann man alle Bäume einer TAG ebenenweise kontextfrei interpretieren und erhält den kontextfreien Kern der TAG (die zur TAG korrespondierende kontextfreie Grammatik). Dabei bedeutet ebenenweise kontextfrei interpretieren, daß in jedem elementaren Baum jeder innere Knoten als linke Seite einer kontextfreien Regel angesehen wird und seine Söhne in ihrer Reihenfolge von links nach rechts die rechte Seite dieser kontextfreien Regel bilden. Auf die genaue Beschreibung dieses Sachverhaltes gehen wir in der Beschreibung unseres Ansatzes in Abschnitt 3.2 genauer ein und illustrieren die Definition am Beispiel.

Der CYK-Algorithmus ist anwendbar auf kontextfreie Grammatiken in einer Normalform, die die Anzahl der Elemente auf der rechten Seite einer Regel auf höchstens zwei beschränkt bzw. weitere Anforderungen an die Elemente selbst stellt (Zweiform bzw. Chomsky-Normalform). Vijay-Shanker und Joshi gehen in ihrem Ansatz von einer kontextfreien Grammatik in Zweiform bzw. von einer korrespondierenden TAG in der entsprechenden Normalform aus. Die Übertragung erfolgt natürlich (auch im Hinblick auf die Transformation in die Normalform) durch die Darstellung jeder kontextfreien Regel als Baum, in dem die linke Seite den Vaterknoten bildet und die Elemente der rechten Seite in ihrer Reihenfolge die Söhne dieses Knotens. Damit ist eine TAG in Zweiform, wenn jeder Knoten in jedem elementaren Baum höchstens zwei Söhne hat.

Unser Ansatz fordert Chomsky-Normalform für die kontextfreie Grammatik und die TAG. Da die Transformation in Chomsky-Normalform nicht mehr analog zum kontextfreien Fall verläuft, gehen wir auf die einzelnen Transformationsschritte in einem eigenen Abschnitt (siehe Abschnitt 3.3.3) ein.

Der CYK-Algorithmus fällt in die Klasse der breadth-first-, bottom-up-Verfahren und baut zu einem Eingabesatz $w = w_1...w_n$ der Länge $n \geq 1$ eine Dreiecksmatrix auf. Ein Nichtterminal A im Element (i,j) (i=Spalte, j=Zeile) dieser sogenannten CYK-Matrix steht für die Ableitbarkeit des Teilwortes $w_i...w_{i+j-1}$, ausgehend von A. Entsprechend ist der Eingabesatz $w$ Element der durch die kontextfreie Grammatik beschriebenen Sprache, falls S im Element (1,n) der CYK-Matrix steht. Eine detaillierte Beschreibung des CYK-Algorithmus würde an dieser Stelle zu weit führen und ist z.B. in Hopcroft/Ullman (siehe [4]) nachzulesen. Der CYK-Algorithmus benötigt $O(n^3)$ Zeiteinheiten.

Man kann das Vorgehen des Algorithmus von Vijay-Shanker und Joshi als die "natürliche" Übertragung des CYK-Verfahrens auf TAGs ansehen.

Wenn ein Eingabesatz einen Ableitungsbaum zu dieser kontextfreien Grammatik besitzt, ist erst eine notwendige Bedingung für die Existenz einer Ableitungsfolge zu einer TAG erfüllt. Bei TAGs gelten zusätzlich Bedingungen für die Anwendbarkeit einer kontextfreien Regel r. Es müssen alle kontextfreien Regeln aus einem elementaren Baum, die unter jedem Symbol auf der rechten Seite der Regel r liegen, so angewandt werden, daß der jeweilige Teilbaum des elementaren Baumes aufgebaut wird (wenn man adjungierte auxiliare Bäume ignoriert).

Diese zusätzliche Anforderung kann man in das CKY-Verfahren integrieren. Man merkt sich zu jedem Nichtterminal in einer Zelle der Dreiecksmatrix, welchen Teilbaum eines elementaren Baumes es ableiten kann. So kann man aber nur einen Ableitungsbaum beschreiben (d.h. auch seine Wurzel trägt diese zusätzliche Information), wenn er nur aus einem initialen Baum besteht. Wir sagen nun, wie man vorgeht beim *Ignorieren eines adjungierten Baumes.* Ein solcher Baum wird in einem inneren Knoten eines initialen oder durch Adjunktionen bereits modifizierten initialen Baumes adjungiert. Das entspricht im CKY-Verfahren der Situation, daß eine kontextfreie Regel angewandt werden soll, bei der der Fußknoten Symbol auf der rechten Seite der Regel ist. Dabei muß überprüft werden, ob es für alle Symbole auf der rechten Seite dieser Regel entsprechende Zellen gibt. Also muß es auch eine Zelle geben, in der das Symbol des Fußknotens steht, genauer gesagt, der Teilbaum der Ableitung unter dem Fußknoten. Diese Zelle enthält wie oben beschrieben als zusätzliche Information, welchen elementaren Teilbaum sie ableiten kann.

Da eine zentrale Idee des CKY-Verfahrens ist, alle notwendige Information in den Zellen zu finden, die die Symbole auf der rechten Seite einer kontextfreien Regel charakterisieren (also auch die zusätzliche, daß der Teilbaum ableitbar ist TAG-gemäß), muß man diese zusätzliche Information konservieren bis zum Erreichen der Wurzel dieses auxiliaren Baumes. Nur so kann man sicherstellen, daß der gesamte elementare Baum, der unter dem Fußknoten teilweise beschrieben wurde, im Ableitungsbaum enthalten ist. Im Hinblick auf die beliebig häufige Unterbrechung durch weitere Adjunktionen, die die Konservierung des Zustandes des gerade bearbeiteten auxiliaren Baumes mit sich zieht, bietet sich das Prinzip des Kellerns an.

Erreicht man die Wurzel eines adjungierten Baumes, so interessiert nicht weiter, welcher Baum das war. Er darf für weitere Untersuchungen ignoriert werden. Relevant ist nur noch die gekellerte Information über noch nicht vollständige adjungierte Bäume und

Teile des initialen Baumes.

Zur Verwaltung dieser Information kann man sich eine 4-dimensionale Matrix vorstellen. Zwei Dimensionen werden von der Dreiecksmatrix im kontextfreien Fall übernommen. Es wird ebenfalls der linke und rechte Rand des überdeckten Teilbaumes abgespeichert. Die beiden anderen Dimensionen dienen dazu, den Teilbaum zu charakterisieren, der unter dem Fußknoten liegt. Dieser wird ebenfalls durch linken und rechten Rand beschrieben. Diese beiden neuen Dimensionen charakterisieren gerade die Wurzel des Teilbaumes, der unter dem Fußknoten liegt. Beim Erreichen der Wurzel des adjungierten Baumes kann man also mit dieser Information die oben als gekellert beschriebene, konservierte Beschreibung des Teilbaumes abrufen und sie in der Wurzel des adjungierten Baumes (der nun ignoriert wird) fortsetzen.

Man muß also als Initialisierung dieses Verfahrens alle Terminale $w_i$ des Eingabesatzes $w = w_1...w_n$ als Teilbäume von elementaren Bäumen beschreiben, in denen sie als Blatt vorkommen, d.h. der linke Rand ist vor $w_i$, der rechte Rand hinter $w_i$ und der linke Rand u und rechte Rand v des Teilbaumes unter dem Fußknoten wird durch Gleichsetzung u = v als der 'leere Baum' beschrieben.

Auch jeder Fußknoten stellt ein Blatt eines auxiliaren Baumes dar. Also muß auch seine Existenz in der Initialisierung beschrieben werden. Hier setzt man linker Rand = linker Rand des Teilbaumes und rechter Rand = rechter Rand des Teilbaumes. So hat man beschrieben, daß der Fußknoten keine Terminale ableitet und hat ausgesagt, welche Terminale des Teilbaums unter dem Fußknoten liegen (alle von linker Rand des Teilbaumes bis rechter Rand des Teilbaumes). Jeder Knoten in der Dreiecksmatrix kann Fußknoten sein (erst die Überprüfung, ob es eine kontextfreie Regel gibt, in der der Fußknoten Symbol auf der rechten Regelseite ist, und ob der Teilbaum unter dem Fußknoten das gleiche Nichtterminal besitzt, erlauben den Weiterbau in der Dreiecksmatrix mit dieser Knoteninformation). Also muß man alle Zellen der 4-dimensionalen Matrix mit linker Rand = linker Rand des Teilbaumes = i $\in$ {1, ..., n} und rechter Rand = rechter Rand des Teilbaumes = j ($\geq$ i), ..., n initialisieren mit der Menge aller Nichtterminale, die an einem Fußknoten stehen können.

Damit ist der Eingabesatz des CKY-Algorithmus nicht mehr w sondern alle initialisierten Zellen, d.h. die Länge der Eingabe erhöht sich auf $O(n^2)$. Damit ist sofort klar, daß das Verfahren $O((n^2)^3) = O(n^6)$ Schritte benötigt.

Dieser Algorithmus wird nun erweitert, so daß man mit ihm auch TAGs mit Unifikation verarbeiten kann.

### 3.1.2 Parsing von TAGs mit Unifikation

Wir erweitern den obene vorgestellten Parsingalgorithmus nun nach der Definition von TAGs mit Unifikation wie Vijay-Shanker sie vorgeschlagen hat (siehe Abschnitt 2.2.1).

Zum Entstehungszeitpunkt dieser Arbeit ist aus der Literatur (siehe [12]) lediglich bekannt gewesen, daß ein Parser für TAGs mit Unifikation aufbauend auf dem im vorangehenden Abschnitt vorgestellten Algorithmus zum Parsing von TAGs in Reinform bereits

implementiert ist. Allerdings sind an keiner Stelle weitergehende Hinweise zu finden, aus denen die konkrete Realisierung dieses Parsers hervorgeht. Insbesondere wäre es interessant zu wissen, welcher Unifikationsformalismus zugrundegelegt und wie dieser in den Parser integriert worden ist.

Wie bereits in der Einleitung zu diesem Kapitel erwähnt, weisen der Ansatz von Vijay-Shanker und Joshi zum Parsing von TAGs in Reinform und der Harbusch-Algorithmus Gemeinsamkeiten auf. So sind beide Algorithmen vom CYK-Algorithmus zur Analyse kontextfreier Grammatiken geprägt. Der CYK-algorithmus fällt in die Klasse der breadth-first-, bottom-up-Verfahren. Um die Voraussetzungen zur Anwendung eines auf einer kontextfreien Analyse aufbauenden Parsers zu schaffen, muß natürlich in beiden Fällen eine zur TAG korrespondierende kontextfreie Grammatik vorliegen.

Aufgrund der oben beschriebenen Situation lassen sich hier nur Vorschläge zur Integration eines Unifikationsformalismus in den bestehenden Parser von Vijay-Shanker und Joshi machen.

Eine mögliche Strategie eines Parsers für TAGs mit Unifikation kann es sein, bereits während der kontextfreien Analyse parallel zum Ableitungsbaum die zu diesem gehörende DAG-Struktur möglichst vollständig aufzubauen.

Die Erfahrungen mit unserem Ansatz und die Gemeinsamkeiten beider Basis-Algorithmen lassen den Schluß zu, daß sich unser Vorgehen bei gleicher Zielsetzung fast 1:1 auf den Parsingalgorithmus von Vijay-Shanker und Joshi übertragen läßt.

Daher sei der Leser für weitergehende Betrachtungen auf die Beschreibung unseres Ansatzes im nächsten Abschnitt (insbesondere in 3.5) verwiesen.

Die entgegengesetzte Strategie würde nur Verweise auf korrespondierende Spezifikationslisten zu allen Knoten im Ableitungsbaum abspeichern und in einem Schritt nach dem Erkennen aller Adjunktionen alle Unifikationen gemäß den Spezifikationslisten ausführen.

Welcher Ansatz gewählt wurde, ist uns unbekannt. Es bietet sich aber die integrierte Lösung an, wenn man über ein Unifikationsmodul wie PATR verfügt. Auf die einzelnen Argumente dafür gehen wir nach der Beschreibung unserer Verbindung der beiden Formalismen im folgenden Abschnitt näher ein.

## 3.2 Unser Ansatz

Der Harbusch-Algorithmus nimmt in unserem Ansatz zum Parsing von TAGs mit Unifikation eine zentrale Rolle ein. Der Harbusch-Algorithmus parst mit einer Laufzeit von $O(n^4\ log\ n)$ Zeiteinheiten (n = Länge der Eingabe) und stellt gegenüber dem bisher aus der Literatur bekannten Algorithmus von Vijay-Shanker und Joshi (siehe [12]) mit einer Laufzeit von $O(n^6)$ eine deutliche Verbesserung dar, da der Algorithmus von Vijay-Shanker und Joshi sowohl im besten als auch im schlechtesten Fall diese Laufzeit besitzt.

Hauptidee des Algorithmus ist, zuerst eine komplette kontextfreie Strukturanalyse des Eingabesatzes mit dem CYK-Verfahren durchzuführen. Von dieser zur TAG korrespondierenden kontextfreien Grammatik wird Chomsky-Normalform gefordert.

In Abschnitt 3.3.3 stellen wir den von Harbusch entwickelten Transformationsalgorithmus und seine Erweiterungen in Bezug auf Unifikation vor. Wir werden sehen, daß sich jede TAG in diese Normalform bringen läßt, so daß die kontextfreie Grammatik zur TAG in Chomsky-Normalform ist. Die Normalform für TAGs geht über die Zweiform hinaus, die Voraussetzung für den Algorithmus von Vijay-Shanker und Joshi ist.

Der Transformationsalgorithmus ist nicht so einfach wie der zur Zweiform, der die natürliche Übertragung der Transformation einer kontextfreien Grammatik in Chomsky-Normalform auf alle inneren Knoten von elementaren Bäumen darstellt.

Bei der Transformation in die hier verwendete Normalform kann man von einer TAG in Zweiform ausgehen, für die darüberhinaus die Eigenschaft der $\epsilon$-Freiheit und die Forderungen gelten, daß ein Knoten mit zwei Söhnen nur mit Nichtterminalen beschriftete Söhne hat und daß ein bei Knoten mit genau einem Sohn dieser mit einem Terminal beschriftet ist. Bei der Transformation in diese Normalform kann man nicht mehr der Transformation in Chomsky-Normalform folgen, da dabei Nichtterminalknoten weggelassen werden. Wenn in einem solchen Knoten aber eine Adjunktion erlaubt ist, so darf man ihn nicht einfach weglassen. Aufgrund dieser Problematik erfolgt die Transformation durch das Herauslösen von Teilbäumen, die nicht in Normalform sind und Entfernen solcher Konstrukte durch die explizite Ausführung von erlaubten Adjunktionen. Der Leser findet die genaue Beschreibung der Transformation in (siehe [3]). In Abschnitt 3.3 werden die einzelnen Schritte der Transformation erläutert und die Erweiterungen der Transformation im Hinblick auf die Integration der Unifikationsinformation beschrieben.

Eine TAG in Normalform ist dann die Voraussetzung für unseren Analyseansatz für TAGs mit Unifikation. In Abschnitt 3.4 wird zunächst der Kern des Harbusch-Algorithmus zum Parsing von TAGs in Reinform vorgestellt. Die Erweiterungen des Harbusch-Algorithmus, die zum Parsing von TAGs mit Unifikation nötig sind, werden in Abschnitt 3.5 behandelt.

## 3.3 Transformation in eine Normalform

Für unseren Parser ist es erforderlich, daß die zugrundeliegende Grammatik in eine bestimmte Normalform transformiert wird. Das bedeutet für TAGs, daß jeder Nichtterminalknoten innerhalb eines Baumes entweder genau zwei nichtterminale Söhne oder genau einen terminalen Sohn besitzt. Zur Transformation benutzen wir den im Anschluß an diese Einleitung folgenden Algorithmus, der der Erweiterung um Unifikation Rechnung trägt. Im Schritt der Elimination von Ketten (Baumstrukturen aus Knoten mit je einem Sohn) tritt z.B. als Problem auf, daß Adjunktionen vorweggenommen werden müssen, die in den Kettengliedern möglich sind. Die Problematik für die DAGs erläutern wir anhand von Abbildung 3.1. Der initiale Baum enthält die Kette $Y_1$, $Y_2$, $Y_3$, wobei jeder Knoten einen stilisierten DAG hat.

Um hier unsere Normalform zu erhalten, ist es erforderlich, die Kette $Y_1$ - $Y_2$ - $Y_3$ durch einen einzigen Knoten, z.B. $K$ zu ersetzen. Rein syntaktisch ist das kein Problem (siehe [3]). Was jedoch geschieht mit den DAGs von $Y_1$, $Y_2$, $Y_3$? Sie können nicht zu einem

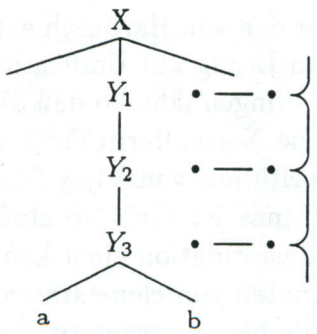



Abbildung 3.1: Beispiel einer Kette

einzigen DAG zusammengefaßt werden. Daher erlauben wir in einem solchen Fall, daß der neue Knoten $K$ alle drei DAGs von $Y_1$, $Y_2$ und $Y_3$ übernimmt und bezeichnen als DAG($K$) die Liste der drei DAGs. Einen sochen DAG bezeichnen wir salopp als "dicken DAG". Die Kette in Abbildung 3.1 verschwindet, und es entsteht die Situation aus Abbildung 3.2.

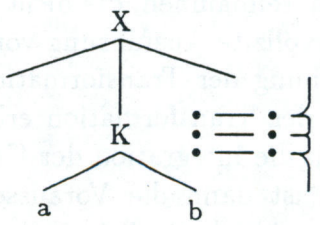



Abbildung 3.2: Dicker DAG der Beispielkette

Zusätzlich muß bei der Entfernung der Knoten $Y_1$, $Y_2$ und $Y_3$ beachtet werden, daß in ihnen Adjunktionen stattfinden konnten, die auch weiterhin möglich sein müssen. Daher wird bei einem solchen Eliminationsschritt eine Anzahl neuer auxiliarer Bäume (auch *K-Bäume* genannt) generiert, die alle möglichen Kombinationen von Adjunktionen in die Y-Kette enthalten und sicherstellen, daß sich der Sprachumfang der Grammatik nicht ändert. Innerhalb eines solchen K-Baumes kann es dann ebenfalls Knoten mit "dicken DAGs" geben. Sie tragen als Knotenlabel den Label des untersten Knotens der Kette, die dort ersetzt wurde. Unter diesen Umständen müssen wir natürlich noch definieren, wie eine Adjunktion abläuft, wenn der Anhängeknoten oder Wurzel bzw. Fußknoten des auxiliaren Baumes einen "dicken DAG" besitzen. In der formalen Definition, die wir aus Verständnisgründen erst später vorstellen, wird man sehen, daß man K als "dicken Knoten" (d.h. wie die Kette von Knoten, die jeweils einen DAG besitzen) interpretiert und wie gewohnt unifiziert.

Im folgenden gehen wir nun auf die einzelnen Schritte der Transformation ein. Zunächst definieren wir die hier verwendete Normalform.

**Definition 18 :** Eine TAG mit Unifikation G ist in unserer **Normalform**, genau dann, wenn folgendes gilt:

(1) $\epsilon$, das leere Wort, ist in L(G) genau dann, wenn $\emptyset$ — $\epsilon$ ein initialer Baum ist.

(2) Ansonsten ist $\epsilon$ nicht als Terminalknoten erlaubt, d.h. die Grammatik ist $\epsilon$-frei.

(3) Für jeden Nichtterminalknoten in jedem Baum gilt: Er besitzt entweder genau zwei nichtterminale Söhne oder genau einen terminalen Sohn.

Abbildung 3.3 veranschaulicht die beiden möglichen Situationen für innere Knoten, die nicht im $\epsilon$-Baum liegen.

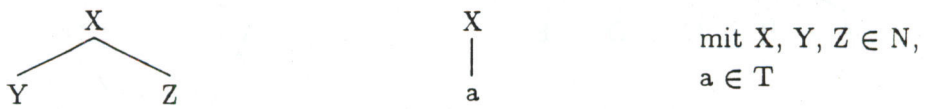



Abbildung 3.3: Ableitungen in Normalform

In diesem Kapitel wird ein algorithmisches Verfahren angegeben, wie TAGs mit Unifikation in unsere Normalform transformiert werden. Es beinhaltet drei wesentliche Schritte:

**Tranformation der Grammatik in Zwei-Form (Definition 19):** Eine Tree Adjoining Grammar mit Unifikation ist in **Zwei-Form** genau dann, wenn jeder Nichtterminalknoten höchstens zwei Söhne besitzt.

Zur Transformation kann das bekannte Verfahren für Chomsky-Normalform bei kontextfreien Grammatiken, die Einführung von Hilfsknoten, übernommen werden, allerdings muß dabei beachtet werden, daß Vererbungsschritte der Unifikation trotz neuer Knoten weiterhin analog stattfinden.

**Entfernung aller $\epsilon$-Produktionen :** Das wesentliche Problem hier ist, daß die Entfernung von $\epsilon$-Knoten zu nichtterminalen Blättern führt, die dann ebenfalls entfernt werden müssen, um wieder einen korrekten TAG-Baum zu erhalten. Allerdings können in ihnen Adjunktionen stattfinden. Da der Sprachumfang sich nicht ändern darf, müssen neue Bäume hinzukommen, die die weggefallenen Adjunktionsmöglichkeiten repräsentieren.

**Elimination von Ketten** Auch bei Ketten müssen Knoten, in denen adjungiert werden kann, entfernt werden, d.h. auch hier muß die Entfernung von Knoten durch neue Bäume zur Erhaltung des Sprachumfangs aufgefangen werden.

Damit ist eine Kurzbeschreibung der bei der Transformation in die hier verwendete Normalform auftretenden Probleme gegeben. Im folgenden betrachten wir die einzelnen Schritte genauer.

### 3.3.1 Transformation in Zwei-Form

Eine TAG in Reinform läßt sich durch das von den kontextfreien Grammatiken her bekannte Verfahren der Einführung von Hilfsknoten leicht in Zwei-Form transformieren. Als einzige Problematik bei der Erweiterung um Unifikation ist zu beachten, daß die Vererbungsschritte weiterhin analog ablaufen müssen.

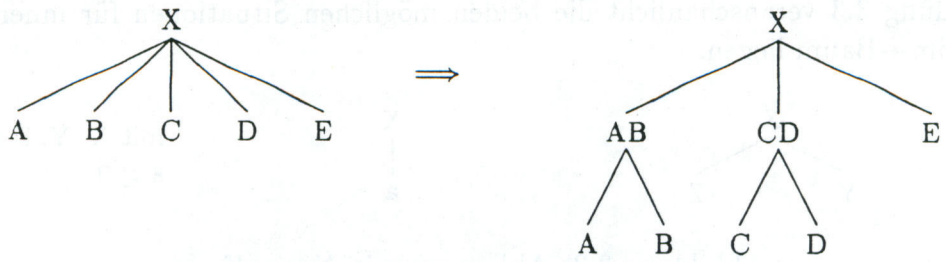



Abbildung 3.4: Einführung von Hilfsknoten

Abbildung 3.4 veranschaulicht die Einführung der Hilfsknoten an einem konkreten Fall. Je zwei Söhne eines Knotens mit mehr als zwei Söhnen erhalten (von links beginnend) einen Hilfsknoten mit neuem Knotenlabel als Vater. Durch den neuen Label ist sichergestellt, daß im Hilfsknoten keine Adjunktion stattfinden kann, da keine passenden auxiliaren Bäume für ihn vorliegen können. Dieses Verfahren ist allerdings nicht die klassische kontextfreie Transformation , bei der in jedem Schritt nur jeweils die beiden am weitesten rechts stehenden Nichtterminale einen Hilfsknoten als Vater erhalten. Unser Augenmerk lag dabei vor allem darauf, der besseren Lesbarkeit wegen die Tiefe des transformierten Baumes nicht mehr als notwendig zu erhöhen, was bei der klassischen kontextfreien Transformation passieren würde.

Nun muß man noch sicherstellen, daß die DAG-Kanten, die vorher zwischen Vater und Sohn bzw. Geschwistern bestanden, nun über die Hilfsknoten hinweg bestehen. Jeder Knoten kann drei Arten von Regeln besitzen:

a) **Vererbung zum Vater** z.B. ((0 x) (1 y))

b) **Vererbung zu einem Geschwister :** z.B. ((1 y) (3 z))

c) **Wertdefinition:** z.B. ((0 y) ok)

wobei x, y, z beliebig lange Folge von Attributen und ok ein Atom ist.

Der betrachtete Knoten und seine ehemaligen Söhne behalten weiterhin dasselbe Wissen. Die bei der Zweiform-Transformation eingeführten Hilfsknoten erfordern lediglich eine modifizierte Handhabung.

Definitionen von Werten in Fall c) haben keinerlei Auswirkungen auf die Regeln, sie bleiben unverändert stehen.

Regeln wie a), die eine Beziehung zwischen dem DAG des Knotens und dem eines Sohnes beschreiben, werden einfach auf die neuen Hilfsknoten übertragen. Lautet die Regel

etwa *((0 x) (3 y))* und befindet sich Sohn 3 nach der Transformation unter Hilfsknoten 2, so muß die Regel entsprechend in *((0 x) (2 y))* geändert werden. Darüberhinaus muß in allen Hilfsknoten dafür gesorgt werden, daß sie die entsprechenden DAG-Verbindungen zwischen Knoten und ehemaligem Sohn aufrechterhalten. Dies geschieht durch eine schlichte Vererbung *((0 Pfad) (1/2 Pfad))* (vgl. dazu auch Kapitel 4.2.4). Auf die gleiche Weise werden Vererbungsregeln zwischen Geschwistern angeglichen.

Problematisch ist alleine der Fall, wo auf den gesamten DAG eines Sohne (z.B. *((0 x) (1))*) Bezug genommen wird. Durch die Zusammenfassung mehrere Söhne unter denselben Hilfsknoten ist die Referenznummer nicht mehr eindeutig (wenn z.B. Sohn 1 und 2 nun beide über Hilfsknoten 1 angesprochen werden müssen). Einen Ausweg bildet ein künstliches Eindeutig-Machen durch die Verwendung der (eindeutigen) Knotennummer als zusätzlichen Pfadpräfix. Trägt etwa der ehemalige Sohn 1 nach der Transformation Knotennummer "x.y.z" und befindet er sich nun unter Hilfsknoten 1, so ändert sich die Regel des Vaters *((0 x) (1))* in *((0 x) (1 "x.y.z"))*, der Hilfsknoten erhält als Regel *((0 "x.y.z") (1))*, falls der Sohn hier erster Nachfolger ist, usw.

Wir können also ab jetzt für die weiteren Transformationsschritte von einer TAG-Grammatik mit Unifikation in Zwei-Form ausgehen.

### 3.3.2 Entfernung aller $\epsilon$-Produktionen

Den folgenden Algorithmus beschreiben wir auf der Basis von TAGs in Reinform. Um zu dem entsprechenden Algorithmus für TAGs mit Unifikation zu gelangen, müssen statt der Adjunktionen für TAGs in Reinform einfach Adjunktionen mit Unifikation für TAGs mit Unifikation ausgeführt werden.

Nach unserer Definition der Normalform darf, falls $\epsilon \in L(G)$, $S\emptyset \rightarrow \epsilon$ der einzige Baum sein, der $\epsilon$ ableitet, und in dem $\epsilon$ vorkommt. Gibt es in der gegebenen Grammatik einen oder mehrere initiale Bäume mit Blattwort $\epsilon$, so erzeugen wir zusätzlich den durch die Definition der Normalform geforderten $\epsilon$-Baum aus Abbildung 3.5.

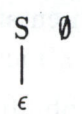



Abbildung 3.5: $\epsilon$-Baum

Statt der ursprünglichen Bäume mit Blattwort $\epsilon$ werden durch Adjunktion aller möglichen auxiliaren Bäume in diesen neue Bäume erzeugt und zur Grammatik hinzugenommen. So wird sichergestellt, daß das Blattwort $\neq \epsilon$ ist und die Struktur für weitere Adjunktionen erhalten bleibt. Nun gibt es außer dem $\epsilon$-Baum keinen Baum mehr mit Blattwort $\epsilon$, aber weiterhin kann es Blätter in elementaren Bäumen geben, die $\epsilon$ als Beschriftung tragen. Diese Blätter können nicht einfach weggelassen werden, da dann nichtterminale Blätter entstehen (siehe Abbildung 3.6). Sie müssen im folgenden noch korrigiert werden.

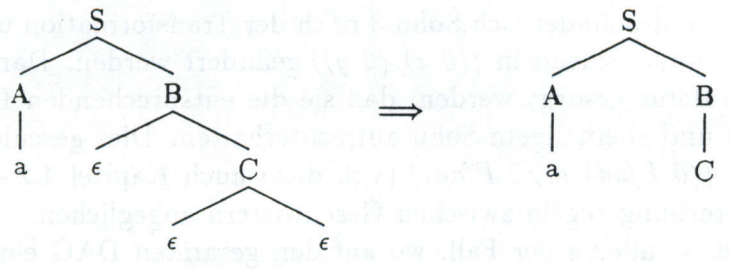



Abbildung 3.6: Entfernung der $\epsilon$-Knoten

Dabei geht man von jedem nichtterminalen Blatt aus so weit im Baum nach oben, bis ein Knoten erreicht ist, der entweder einen korrekten Fußknoten oder ein terminales Blatt ableitet. Den so durchschrittenen Teilbaum trennt man heraus und macht ihn zu einem auxiliaren Baum mit NA-Constraints in Wurzel und Fußknoten, um eine zyklische Adjunktion dieses Baumes zu verhindern. Abbildung 3.7 verdeutlicht diesen Vorgang.

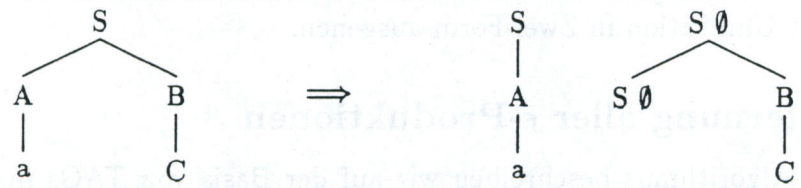



Abbildung 3.7: Auftrennen der falschen auxiliaren Bäume

Der ursprüngliche Baum ist nach der Modifikation ein korrekter elementarer Baum. Der erzeugte auxiliare Baum aber besitzt immer noch ein nichtterminales Blatt. Um dieses zu entfernen, tut man folgendes:

(1) Führe alle möglichen Adjunktionen in diesem nichtterminalen Blatt durch und entferne dieses anschließend. Dadurch entstehen genau so viele neue korrekte auxiliare Bäume, wie es verschiedene Adjunktionsmöglichkeiten im falschen Fußknoten gibt.

Diesen Vorgang veranschaulicht Abbildung 3.8.

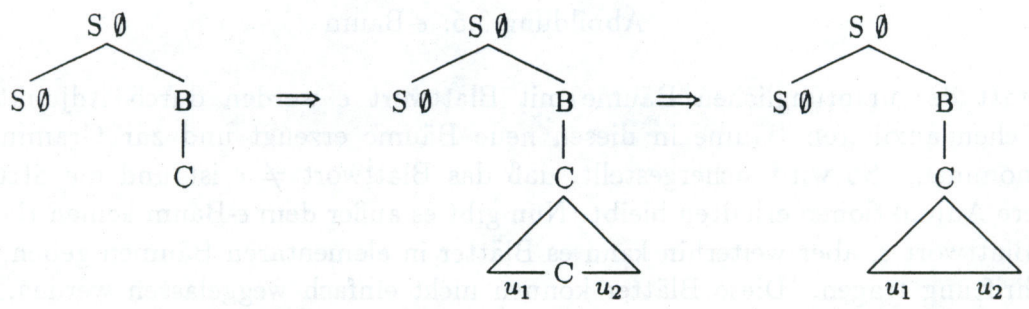



Abbildung 3.8: Adjunktion im falschen Fußknoten und Entfernung desselben

(2) Schneide im Ausgangsbaum mit dem nichtterminalen Blatt dieses ab. Wird ein Knoten erreicht, der weder Vater des korrekten Fußknotens ist, noch mindestens ein Terminal ableitet, gehe zu (1), sonst ist man fertig.

In unserem Beispiel aus Abbildung 3.7 bzw. Abbildung 3.8 muß nochmals adjungiert werden, da der Knoten mit Beschriftung B in (2) als erneuter 'falscher Fußknoten' nicht akzeptiert wird. Diese Iteration ist in Abbildung 3.9 dargestellt.

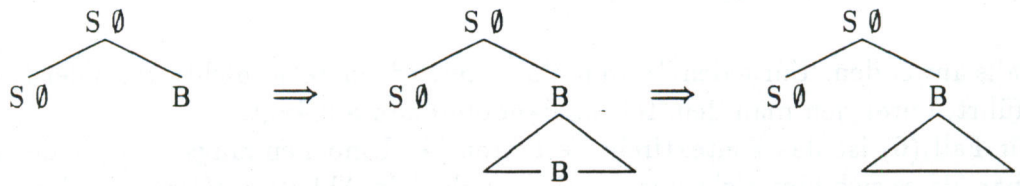



Abbildung 3.9: Adjunktion im falschen Fußknoten und Entfernung desselben

Zum Schluß muß eines noch beachtet werden: Die oben erzeugten auxiliaren Bäume tragen an Wurzel und Fußknoten natürlich den Label des Trennknotens, aus dem der Ausgangsbaum ausgeschnitten wurde. Allerdings dürfen sie nur in diesem Trennknoten adjungiert werden. Daher müssen alle anderen Knoten mit diesem Label als Constraint erhalten, daß nur die bisher adjungierbaren auxiliaren Bäume in ihnen adjungiert werden dürfen und nicht die eben neu erzeugten . Diese Constraints stellen dann die geforderte Bedingung sicher.

Wir können also ab jetzt von einer $\epsilon$-freien TAG-Grammatik mit Unifiaktion ausgehen, da für TAGs mit Unifikation lediglich Adjunktionen mit Unifikation, die syntaktisch gesehen das gleiche Ergebnis liefern, ausgeführt werden. Auch hierfür liefern wir keinen ausführlichen Korrektheitsbeweis. Ihn findet man in [3].

### 3.3.3 Transformation der $\epsilon$-freien Zwei-Form in Normalform

Im letzten Transformationsschritt muß die Grammatik aus der ($\epsilon$- freien) Zwei-Form in Normalform überführt werden. Aufwendig ist dabei vor allem die rein syntaktische Transformation der TAG (siehe [3]). Die notwendigen Veränderungen, die für die Spezifikationslisten erforderlich sind, lassen sich durch einfache Fallunterscheidung leicht bewältigen und erfordern natürlich keine Veränderungen des Verfahrens von Harbusch, da das Unifikationspaket eines Baumes ja getrennt von seiner syntaktischen Struktur betrachtet werden kann. Daher stellen wir an dieser Stelle zuerst die syntaktischen Transformationsschritte vor, zu denen abschließend immer die Erweiterung um Unifikation hinzugefügt wird.

Innerhalb eines TAG-Baumes in Zwei-Form gibt es sechs verschiedene mögliche Situationen für innere Knoten, die nicht im $\epsilon$-Baum liegen. Sie sind in Abbildung 3.10 dargestellt.

Hierzu läßt sich folgendes feststellen. Die Fälle (1) und (5) sind bereits in Normalform und brauchen daher nicht mehr betrachtet zu werden. Für die Fälle (2), (3) und (4) läßt sich das Verfahren zur Transformation kontextfreier Grammatiken in Normalform

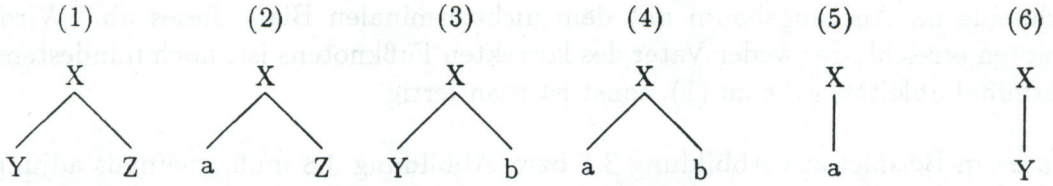



Abbildung 3.10: mögliche Vater-Sohn-Beziehungen für innere Knoten in elementaren Bäumen, die nicht der $\epsilon$-Baum sind

ebenfalls anwenden: Für jeden Terminalknoten wird ein neuer nichtterminaler Hilfsknoten eingeführt, unter den man den Terminalknoten dann anhängt.

Für Fall (6) ist das kontextfreie Verfahren bei Umbenennungsregeln leider nicht anwendbar, da es sich hier nicht nur um Hilfssymbole in Ableitungsfolgen sondern um Pfade in TAG-Bäumen handelt, und die Entfernung der Hilfsknoten in Analogie zur kontextfreien Methode problematisch ist, da in ihnen adjungiert werden kann. Zur Transformation dieser sogenannten "Ketten" wird daher ein aufwendigerer Algorithmus benötigt.

**Transformation der Fälle (2), (3), und (4) :**

Betrachten wir zunächst die reinen TAGs. Zwischen jeden Terminalknoten und seinen Vater wird jeweils ein nichtterminaler Hilfsknoten mit neuem Knotenlabel eingefügt. Dieser darf bisher nicht aufgetreten sein, damit in den neuen Knoten keine Adjunktionen möglich sind. Abbildung 3.11 verdeutlicht an einem Beispiel dieses Vorgehen.

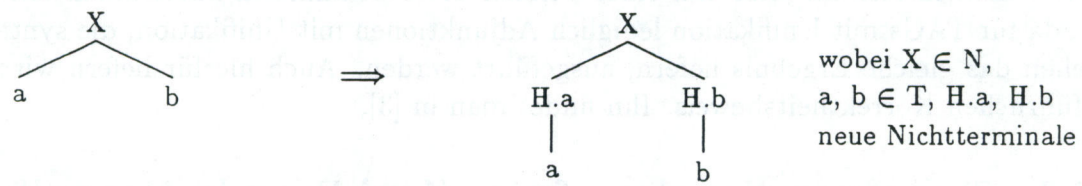



Abbildung 3.11: Einführung von Hilfsknoten

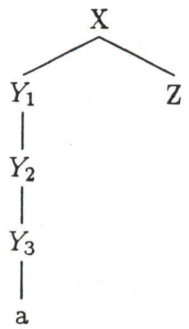
Bei TAGs mit Unifikation muß berücksichtigt werden, daß die Vererbungsschritte trotz der eingefügten Hilfsknoten weiterhin analog ablaufen. Genau dieses Problem trat aber bereits bei der Transformation der Grammatik in Zwei-Form auf, so daß wir an dieser Stelle auf die erforderlichen Änderungen der Regeln nicht mehr einzugehen brauchen, sondern analog wie in Kapitel 3.3.1 beschrieben vorgehen.

**Elimination von Ketten :**

Wie bei den bisher behandelten Fällen läßt sich auch für Ketten das Eliminationsverfahren der reinen TAGs übernehmen. Bei Hinzunahme der Unifikation sind lediglich einige Veränderungen an den DAGs vorzunehmen, die das TAG-Verfahren jedoch nicht beeinflussen. Daher folgt an dieser Stelle zunächst der Algorithmus für TAGs.

Eine Kette kann drei unterschiedliche Positionen in einem TAG-Baum einnehmen:

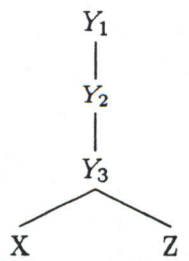
1. Sie steht im Innern eines Baumes (siehe Abbildung 3.12, wobei o.B.d.A. der Sohn des letzten Kettengliedes ein Terminal ist), d.h. der Anfang der Kette ist nicht Wurzel des Baumes, und das Ende nicht Fußknoten.




$X, Y_1, Y_2, Y_3, Z \in N$,
$a \in T$

Abbildung 3.12: Ketten im Innern eines Baumes

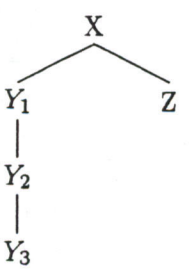
2. Sie beginnt an der Wurzel eines Baumes (siehe Abbildung 3.13).




$Y_1, Y_2, Y_3, X, Z \in N$

Abbildung 3.13: Kette an der Wurzel eines Baumes

3. Sie endet am Fußknoten eines auxiliaren Baumes (Abbildung 3.14).




$Y_1, Y_2, Y_3, X, Z \in N$,
$Y_3$ ist Fußknoten

Abbildung 3.14: Kette am Fußknoten eines auxiliaren Baumes

Im ersten Schritt des Algorithmus werden alle Ketten durch einen einzigen Knoten (auch *Kettenknoten* genannt) ersetzt, wodurch die Bäume kettenfrei werden. In Abbildung 3.15 ist dies für die obigen drei Fälle durchgeführt.

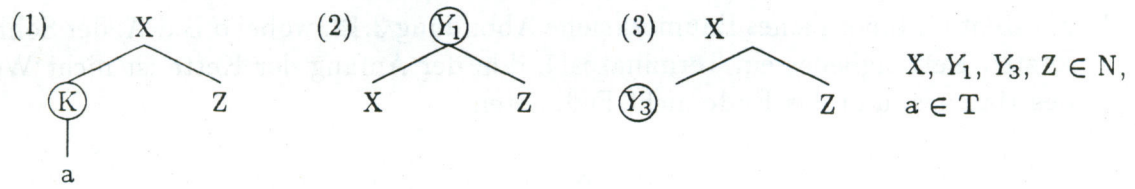



Abbildung 3.15: Ersetzen einer Kette durch den Kettenknoten

In Fall (1) aus Abbildung 3.15 sei K ein Nichtterminal, das bisher nicht aufgetreten ist, damit dort keine Bäume adjungiert werden können, die bislang an dieser Stelle nicht adjungierbar waren. In Fall (2) aus Abbildung 3.15 muß $Y_1$ als Label der Wurzel erhalten bleiben, damit dieser Baum (falls es ein auxiliarer Baum ist) weiterhin adjungiert werden kann, bzw. die Wurzel eines initialen Baumes das Startsymbol ist. Entsprechend muß in Fall (3) $Y_3$ als Fußknotenlabel erhalten bleiben.

Das eigentliche Problem bei der Kettenelimination ist nun, daß auch weiterhin alle Adjunktionsmöglichkeiten in die (ehemalige) Kette repräsentiert sein müssen. Dazu werden für jede Kette neue auxiliare Bäume (K-Bäume) definiert, und zwar so, daß jeder neue auxiliare Baum gerade eine mögliche Kombination von einmaligen Adjunktionen in die Kette darstellt. Es entstehen also genau soviel neue Bäume, wie es verschiedene Kombinationen von einmaligen Adjunktionen entlang der Kette gibt. Damit diese Bäume in den neuen Kettenknoten für die Kette adjungiert werden können, erhalten Wurzel und Fußknoten den Label des Kettenknoten als Knotenlabel . Der Kettenknoten bekommt zusätzlich als Constraint, daß nur diese neuen Bäume in ihn adjungiert werden dürfen.

Im zweiten Schritt bildet man also alle möglichen Kombinationen von Adjunktionen entlang der Kette, wie es in Abbildung 3.16 dargestellt ist.

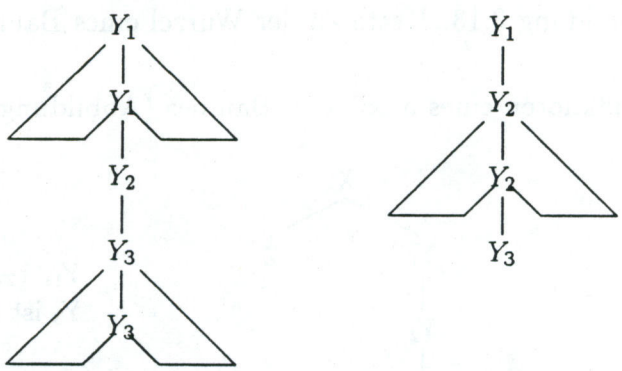



Abbildung 3.16: Adjunktionen entlang einer Kette

Es fällt auf, daß die Kette durch bloßes Adjungieren nicht verschwindet, sondern sich allenfalls in einzelne Teilketten zerlegt wie in Abbildung 3.16 (b). Dies ist jedoch nicht weiter tragisch, denn an dieser Stelle können die Teilketten problemlos durch ihren Endknoten ersetzt werden, da ja alle Kombinationen von Adjunktionen gerade gebildet werden, und in den entfernten Knoten keine Adjunktion stattfinden soll (siehe Abbildung 3.17 für die

entsprechend modifizierten Bäume aus Abbildung 3.16).

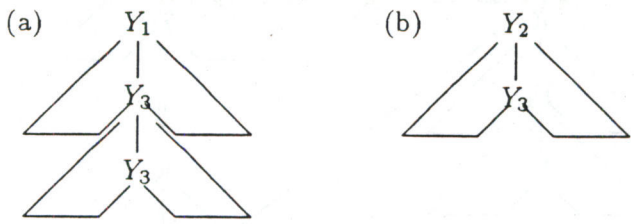



Abbildung 3.17: Entfernung der Zwischenknoten

Nun muß der Label von Wurzel und Fußknoten der neu erzeugten auxiliaren Bäume auf den Label des Kettenknotens geändert werden, damit die Bäume dort auch adjungierbar sind. In Abbildung 3.18 ist dies für den Fall, daß die Kette inmitten eines Baumes stand (Abbildung 3.12), durchgeführt.

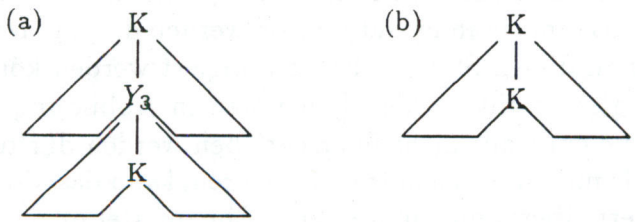



Abbildung 3.18: Einsetzen des Kettenknotens an Wurzel- und Fußknoten der erzeugten auxiliaren Bäume

Jetzt muß noch sichergestellt werden, daß in bestimmten Kettenknoten mehrfach adjungiert werden kann. Da Wurzel und Fußknoten der neu erzeugten Bäume neue Labels tragen, müssen auch die in ihnen adjungierbaren Bäume zusätzlich mit entsprechend geändertem Wurzel- bzw. Fußknotenlabel zur Verfügung stehen, wie es in Abbildung 3.19 veranschaulicht ist.

Auch in diesen Bäumen wird an der Wurzel die Constraintmenge so eingesetzt, daß statt der ursprünglich adjungierbaren Bäume nun die entsprechenden Bäume mit der neuen Wurzel- und Fußknotenbeschriftung adjungiert werden dürfen. Der Fußknoten jedes eine Adjunktionsfolge in die Kette repräsentierenden Baumes erhält das NA-Constraint. Durch diese Constraints wird dann sichergestellt, daß die gesamte Kette nicht zyklisch adjungiert werden kann. Zusätzlich ist es ausgeschlossen, daß weitere Adjunktionen nicht der Knotenreihenfolge entlang der Kette gehorchen, da in der Wurzel nur Bäume eingesetzt werden dürfen, die zum obersten Knoten, in dem bei der Erzeugung adjungiert wurde, gehören, und im Fußknoten gar nicht adjungiert werden darf.

Für Fall (2) und (3) der Kettenpositionen innerhalb eines Baumes (Abbildung 3.13 bzw. Abbildung 3.14) ist ein zusätzlicher Punkt von Bedeutung: Da in Abbildung 3.13 $Y_1$ als Label für die Kette erhalten bleibt, muß sichergestellt werden, daß die erzeugten $Y_1$-Bäume nur in den Kettenknoten adjungiert werden können und nicht in einen anderen

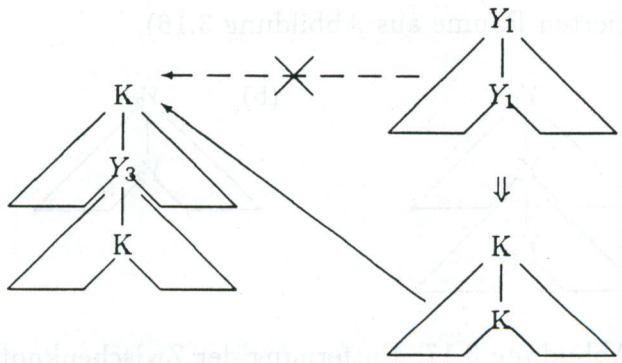



Abbildung 3.19: Erzeugung elementarer K-Bäume für Mehrfachadjunktionen in einem Knoten der Kette

$Y_1$-Knoten an anderer Position im Baum oder in einem anderen Baum. Also erhalten vor der Erzeugung der neuen $Y_1$-Bäume alle anderen $Y_1$-Knoten als Constraint, daß nur die bisher vorhandenen $Y_1$-Bäume in ihnen adjungiert werden dürfen. Damit ist sichergestellt, daß die neuen Bäume an diesen Stellen nicht adjungiert werden können, da sie jetzt erst erzeugt werden. Das gleiche gilt für den Fußknoten in Abbildung 3.14 als letztes Glied der Kette, dessen Name ebenfalls nicht überschrieben werden durfte.

Will man nun TAGs mit Unifikation transformieren, kann das Verfahren für TAGs ohne Unifikation unverändert übernommen werden. Ein im Detail noch ungeklärter Punkt ist allerdings die Frage, was mit den DAGs passiert, wenn mehrere Knoten zu einem einzigen verschmelzen, wie dies bei der Elimination der Ketten an zwei Stellen auftritt: Ersetzen der Kette durch einen einzigen Knoten und Entfernung der Teilketten in den neu erzeugten auxiliaren Bäumen. Auf jeden Fall ist es wünschenswert, daß die DAG-Strukturen der einzelnen Knoten nicht verloren gehen, damit insbesondere die einzelnen Vererbungsschritte weiterhin nachvollziehbar sind, sowie die Einstiegsmöglichkeiten in die DAGs erhalten bleiben.

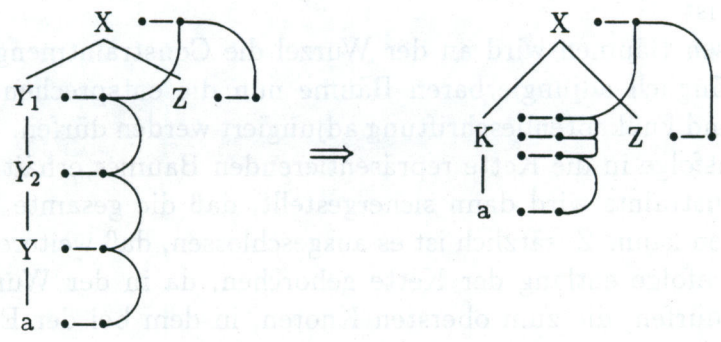



Abbildung 3.20: Entstehung dicker DAGs

Der neue Knoten erhält gemäß Abbildung 3.20 als DAG eine Liste der DAGs der verschmolzenen Knoten zugeordnet ("dicker DAG").

Damit wird es möglich, daß ein Knoten mehr als einen DAG besitzt. Dieser Fall darf jedoch nur innerhalb der Transformation der Grammatik in Normalform entstehen und dort nur bei der notwendigen Verschmelzung von Knoten innerhalb der Kettenelimination. Für den Grammatikschreiber ergeben sich dadurch dann keine Konsequenzen, denn er darf weiterhin nur Knoten mit höchstens einem DAG definieren. Mehrere DAGs an einem Knoten dienen lediglich dazu, die Verschmelzung von Knoten bei der Kettenelimination in den Griff zu bekommen.

Zu Beginn dieses Kapitels haben wir bereits die Problematik angeschnitten, die sich aus der bei der Kettenelimination erforderlichen Verschmelzung von Knoten für ihre DAGs ergibt. Da mittlerweile geklärt ist, warum und an welchen Knoten "dicke DAGs" auftreten, definieren wir nun die Adjunktion mit Unifikation in allen möglichen Fällen unter Bezugnahme auf Definition 16.

**Definition 20 :** Es gibt insgesamt vier mögliche Fälle von Adjunktionen, die folgendermaßen ablaufen :

a) **Anhängeknoten, Wurzel und Fußknoten mit normalem DAG :**

Die Adjunktion erfolgt gemäß Definition 16.

b) **Anhängeknoten mit "dickem DAG", Wurzel und Fußknoten mit normalem DAG :**

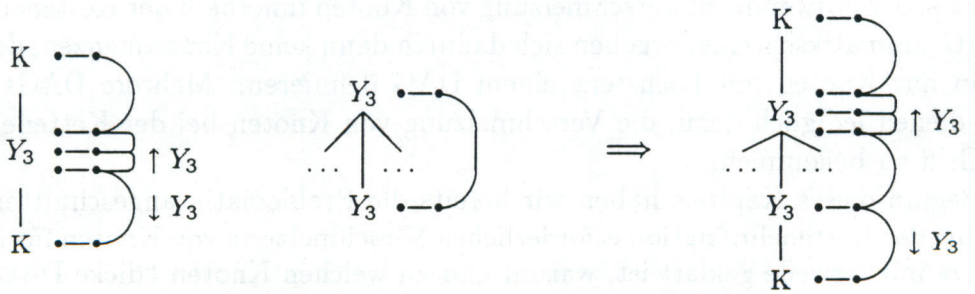
Dann ist der Anhängeknoten ein Knoten innerhalb eines neu erzeugten auxiliaren Baumes (K-Baumes). Er trägt den Label des untersten Knotens der Kette, die er repräsentiert.

Die Adjunktion wird durch Anwendung von Definition 16 auf den untersten DAG des Anhängeknotens durchgeführt. Da alle DAGs im "dicken DAG" über DAG-Kanten verbunden sind, ist klar, daß so DAG-Kanten über den gesamten Ableitungsbaum entstehen. Man stellt sich die Situation auf der Seite der Unifikation so vor, daß bei der Adjunktion mit "dicken DAGs" der betreffende Knoten als Kette von Hilfsknoten interpretiert wird, so daß jeweils ein Knoten und ein DAG korrespondieren und die Adjunktion wie in Definition 16 ausgeführt wird (siehe Abbildung 3.21).

c) **Anhängeknoten, Wurzel und/oder Fußknoten mit "dickem DAG" :**

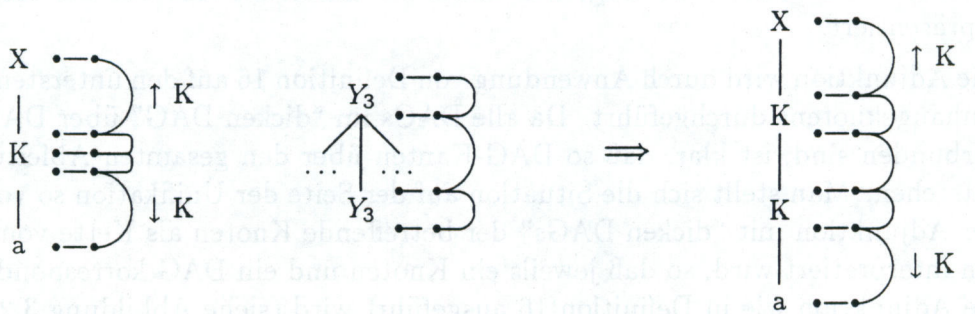
Dieser Fall tritt auf, wenn eine Kette durch einen einzigen Knoten (auch *K-Knoten* genannt) ersetzt wurde, in den jetzt einer der neu erzeugten auxiliaren Bäume (K-Bäume) adjungiert werden soll.

Bei der Adjunktion entfernt man den DAG des Anhängeknotens, hängt den auxiliaren Baum ein und setzt die DAG-Kanten nach oben und unten neu. Dies ist insofern korrekt, als innerhalb des auxiliaren Baumes (K-Baum) die DAG- Entwicklung der Kette bereits enthalten ist und diese nicht doppelt erscheinen darf (siehe Abbildung 3.22).




*$Y_3$ ist der Anhängeknoten, der dadurch entstanden ist, daß eine Teilkette innerhalb eines neu erzeugten auxiliaren Baumes durch ihren untersten Knoten ersetzt wurde. Nach der Adjunktion besitzt deshalb die Wurzel des eingehängten auxiliaren Baumes den "dicken DAG", da die Adjunktion in den untersten Knoten der ersetzten Kette zu interpretieren ist.*

Abbildung 3.21: Beispiel für Fall b)




*Hier ist der Knoten K mit "dickem DAG" Repräsentant einer ehemaligen Kette. In ihn wird einer der neu erzeugten auxiliaren Bäume adjungiert. Man beachte, daß der DAG von K bei der Adjunktion entfernt wird, da der neue auxiliare Baum durch Adjunktionen in die von K repräsentierte Kette enstand und daher eine um die durchgeführten Adjunktionen modifizierte DAG-Struktur der Kette besitzt. Entsprechend sieht das Ergebnis der Adjunktion aus.*

Abbildung 3.22: Beispiel für Fall c)

d) **Anhängeknoten mit normalem DAG, Wurzel und/oder Fußknoten mit "dickem DAG"**

Dann stand in einem elementaren auxiliaren Baum an der Wurzel und/oder dem Fußknoten eine Kette, die mittlerweile durch einen Knoten mit "dickem DAG" repräsentiert wird.

Zur Adjunktion wendet man Definition 2 auf den obersten DAG der Wurzel und den untersten DAG des Fußknotens an.

Die Fälle (b), (c) und (d) in der Definition beschreiben Adjunktionen, die nur aufgrund der Transformation in Normalform entstehen. Sie sind für den Grammatikschreiber nicht von Interesse, da er die eingeführten "dicken DAGs" nicht verwenden darf.

Wie wir gesehen haben, wird durch die vorgestellte Transformation u.U. eine Vielzahl neuer Bäume erzeugt. Dies ist natürlich ein Ansatz für Kritik. Ist die Verbesserung der Laufzeit eines Parsers von $O(n^6)$ auf $O(n^4 log(n))$ nicht teuer erkauft? Theoretisch gesehen ist dies sicher richtig. Die Anzahl der neuen Bäume hängt jedoch, wie wir gesehen haben, wesentlich davon ab, wie oft $\epsilon$ als Terminal verwendet wird und wie lang vorhandene Ketten sind. Berücksichtigt man die Hauptaufgabe von TAGs mit Unifikation, natürliche Sprache zu analysieren, so läßt sich sagen, daß viele $\epsilon$-Blätter und lange Ketten linguistisch irrelevant sind, so daß davon ausgegangen werden kann, daß sich die Anzahl neuer Bäume von der Natur der Sache her automatisch in Grenzen hält.

Einen weiteren Ansatz für kritische Worte bietet die sehr unschöne Einführung von "dicken DAGs" bei der Verschmelzung von Knoten. Ein anderer Lösungsansatz für dieses Problem wäre der Versuch, ein Verfahren zu entwickeln, das aus den DAGs der verschmelzenden Knoten einen einzigen DAG konstruiert. Diesen Weg haben wir allerdings nicht weiter verfolgt, da sofort zwei Probleme ins Auge fallen: Wie könnten dabei die einzelnen Vererbungsschritte entlang der Knoten erhalten bleiben und wie sollen umgekehrt aus diesem DAG die DAGs der einzelnen Knoten rekonstruiert werden können?

Die Grammatik ist nun in Normalform, da alle bisherigen und alle neu erzeugten Bäume es sind und sichergestellt ist, daß alle Adjunktionen, die vor der Transformation möglich waren, entweder weiterhin möglich sind oder durch neue auxiliare Bäume repräsentiert werden. Einen kompletten Beweis hierfür findet man in [3].

Damit haben wir die Grammatik in der Form vorliegen, wie sie unser Parser benötigt, mit dem wir uns im folgenden beschäftigen.

## 3.4 Parsing von TAGs ohne Unifikation

Bei den folgenden Betrachtungen wird von jetzt ab davon ausgegangen, daß eine TAG in der für den Algorithmus von Harbusch nötigen Normalform als Ergebnis des Transformationsalgorithmus aus Abschnitt 3.3.3 vorliegt.

Hauptidee des Algorithmus vno Harbusch ist, zuerst eine komplette kontextfreie Strukturanalyse des Eingabesatzes durchzuführen. Die TAG wird dazu wie beim Ansatz von Vijay-Shanker und Joshi ebenenweise als kontextfreie Grammatik beschrieben. In der

Menge der kontextfreien Ergebnisse sind auf jeden Fall alle korrekten TAG-Ergebnisse enthalten, die es dann noch herauszufiltern gilt. Auf einer graphisch adäquaten Darstellung, die gleiche Teile von verschiedenen Ableitungen gemeinsam repräsentiert, werden nun iterativ alle Adjunktionen gesucht und eliminiert, bis man nur noch initiale Bäume erhält.

Der generelle Unterschied zum Algorithmus von Vijay-Shanker und Joshi ist, daß als Eingabesatz nicht die Initialisierung in einer 4-dimensionalen Matrix gewählt wird, die alle Teilbäume beinhaltet, die TAG-gemäß ableitbar sind. Es werden alle zu ignorierenden auxiliaren Bäume in der Dreiecksmatrix gesucht. Diese sind völlig lokal definiert, d.h. um einen solchen Baum zu finden, muß man nur das Aussehen des auxiliaren Baumes kennen.

Die anschließend aufgeführten Analyseschritte verdeutlichen grob das Vorgehen und die wesentlichen Ideen des Algorithmus von Harbusch zum Parsing von TAGs in Reinform. Da das Hauptgewicht dieser Arbeit auf den von uns erarbeiteten Erweiterungen des Algorithmus von Harbusch liegt, verweisen wir den interessierten Leser für Details auf [3].

Schritte der Analyse: Sei w ein Eingabewort aus n ($\geq 0$) Terminalen und G eine TAG in der beschriebenen Normalform.

**Behandlung des leeren Wortes:**

Der Fall, daß die Eingabe aus dem leeren Wort $\epsilon$ besteht und $\epsilon$ in der durch die TAG definierten Sprache liegt, wird gesondert abgefangen, da mit der Normalformdefinition gilt:
$w \in L(G) \Leftrightarrow$ der $\epsilon$-Baum ist in I(G).

**Bestimmung des kontextfreien Kerns der TAG:**

Der kontextfreie Kern der TAG stellt die zur TAG korrespondierende kontextfreie Grammatik dar. Jeder innere Knoten eines elementaren Baumes wird als linke Seite einer kontextfreien Regel interpretiert; der Sohn bzw. die Söhne dieses Knotens bilden die rechte Seite. Zusätzlich zu den Knotennamen werden noch die bereits im vorangegangenen Kapitel eingeführten, eindeutigen Knotennummeren vermerkt. Zur Veranschaulichung der Definition sieht man in Abbildung 3.24 den kontextfreie Kern einer TAG in der von uns verwendeten Normalform (in Abbildung 3.23).

Ausgehend von der bei uns verwendeten Normalform für TAGs ist die korrespondierende kontextfreie Grammatik in Chomsky-Normalform, womit die Voraussetzungen für eine Analyse des Eingabesatzes mit dem CYK-Algorithmus (siehe [8]) geschaffen sind.

**CYK-Analyse des kontextfreien Kerns:**

Auf den CYK-Algorithmus wurde bereits im Zusammenhang mit dem Ansatz von Vijay-Shanker und Joshi in Abschnitt 3.1 näher eingegangen. An dieser Stelle sollen nur die für den Algorithmus von Harbusch wichtigen Erweiterungen betrachtet werden. Die grundlegenden Notationen aus Abschnitt 3.1 bleiben erhalten.

initialer Baum $\alpha$ mit Nummer 0 auxiliarer Baum $\beta$ mit Nummer 1

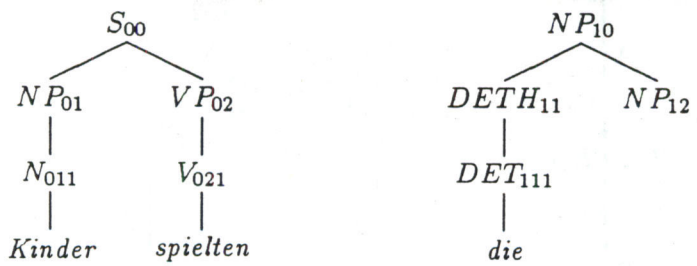



Abbildung 3.23: Beispiel für eine TAG in Chomsky-Normalform

$$
\begin{aligned}
\text{S(00)} &\longrightarrow \text{NP(01) VP(02)} \\
\text{NP(01)} &\longrightarrow \text{N(011)} \\
\text{VP(02)} &\longrightarrow \text{V(021)} \\
\text{NP(10)} &\longrightarrow \text{DETH(11) NP(12)} \\
\text{DETH(11)} &\longrightarrow \text{DET(111)}
\end{aligned}
$$

Abbildung 3.24: Kontextfreier Kern zur TAG der vorangehenden Abbildung

Das CYK-Verfahren wird insoweit leicht modifiziert, als sich ein Eintrag in einem Element der CYK-Matrix nun aus dem Nichtterminal-Label (wie bisher) und den folgenden Komponenten zusammensetzt:

- Knotennummer (wie oben beschrieben),
- je eine Kante für den linken bzw. rechten Sohn, womit der Zusammenhang zwischen linker und rechter Seite einer kontexfreien Regel explizit repräsentiert wird,
- Baumart $\in$ {LA, LI, TA, TI, NIL}: die Baumart gibt an, ob der Unterbaum, gemäß des TAG-Baumes zur aktuellen Knotennummer, vollständig vorhanden ist. LA oder LI stehen für lokal-auxiliar oder lokal-initial, falls der Knoten ein innerer Knoten des Baumes ist. TA oder TI stehen für total-auxiliar oder total-initial, falls der Knoten die Wurzel des Baumes ist,
- Fußknotenpointer (Pointer auf den Fußknoten, falls er in diesem Unterbaum liegt, bzw. NIL, falls nicht oder falls der Baum initial ist).

Die neuen Positionen werden während jedem Aufbauschritt eines Matrixelementes im CYK-Algorithmus besetzt. Die Knotennummern und die Ursprungs- und Zielknoten der Kanten erhält man über die Zuordnung jeder kontextfreien Regel zu einem Knoten und seinen Söhnen in der TAG.
Die Baumart wird induktiv, ausgehend von allen Blättern (d.h. auch Fußknoten) berechnet.

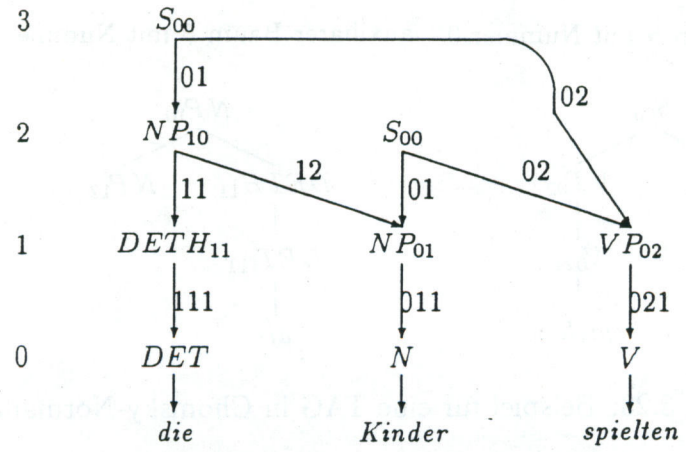



Abbildung 3.25: CYK-Matrix zur Eingabe 'die Kinder spielten'

Bei der Initialisierung der Fußknoten wird die Fußknotenposition gespeichert und bis zur Wurzel des Baumes vererbt.

Die Modifikationen am CYK-Algorithmus kosten nicht mehr Zeit, als der ursprüngliche Algorithmus, da die Baumart und der Fußknotenpointer nur für innerste Bäume 1-ter Stufe berechnet wird, d.h. für auxiliare Bäume in denen keine weiteren Adjunktionen stattgefunden haben. Während der kontextfreien Analyse kann jeder Knoten nur konstant (bezüglich der definierten Grammatik) viele Nummern haben. Jede Nummer spiegelt die Stellung in einem elementaren Baum wider, also kann jeder Knoten nur konstant viele Beschriftungen mit einer Baumart und einem Fußknotenpointer haben. Damit ist die Dreiecksmatrix für die Iteration initialisiert, die alle erkannten auxiliaren Bäume (Wurzel trägt Baumart TA und Fußknotenpointer zeigt auf den Fußknoten des Baumes in der Dreiecksmatrix) eliminiert.

Für unsere Beispielgrammatik aus Abbildung 3.24 (bzw. den dazu korrespondierenden kontextfreien Kern) und den Eingabesatz 'Die Kinder spielten' wird die Dreiecksmatrix in Abbildung 3.25 aufgebaut.

Der leichteren Lesbarkeit wegen wird die Nummer des Zielknotens an die Kante selbst geschrieben.

Wenn ein Eingabesatz kontextfrei akzeptiert wird, ist erst eine notwendige Bedingung für die Existenz einer Ableitungsfolge zu einer TAG erfüllt. Weitere Analysen müssen sich anschließen. Dazu sind die erweiterten Einträge in der CYK-Matrix (wie oben beschrieben) sehr hilfreich.

Besonderes Augenmerk gilt denjenigen Matrixeinträgen, die sich durch die Baumart "TA" auszeichnen. Zusammen mit dem Fußknotenpointer eines derartigen Eintrags ist ein kompletter auxiliarer Baum der TAG innerhalb der CYK-Matrix eindeutig beschrieben. In einem solchen Baum haben keine Adjunktionen mehr stattgefunden. Bezogen auf eine TAG-Ableitungsfolge (in Zukunft kurz: Ableitungsfolge) werden

von jetzt ab derartige elementare Bäume als *innerste Bäume* bezeichnet. Ignoriert man die innersten Bäume einer Ableitungsfolge mit der Umkehrung der Adjunktionsdefinition, so erhält man eine modifizierte Ableitungsfolge, in der nun zu einem früheren Zeitpunkt an Adjunktionen beteiligte auxiliare Bäume als innerste Bäume angesehen werden können. Der Begriff *innerster Baum* ist also temporär und bezieht sich immer auf eine konkrete Ableitungsfolge, wie sie bei fortgesetzter *Elimination* (= Ignorieren eines innersten Baumes) als Zwischenergebnis vorliegt.

**Iteration zur Elimination innerster Bäume:**

Mit den Vorüberlegungen des vorangehenden Analyseschrittes ist intuitiv klar, daß man mit einer Iteration über die Anzahl der Eliminationen entscheiden kann, ob ein (kontextfreier) Ableitungsbaum korrekt bez. einer TAG ist. Die Iteration terminiert nach spätestens n (= Länge der Eingabe), da jeder elementare Baum einer TAG in unserer Normalform ein Terminal produziert, d.h. n-1 Adjunktionen stattgefunden haben können.

Angenommen, eine Ableitungsfolge sei durch m-malige Adjunktion ($m \leq n-1$) entstanden, wobei $\alpha$ der initiale Baum ist. Damit ist m die maximale Schachtelungstiefe der Adjunktionen dieser Ableitungsfolge. In einem Iterationsschritt werden die jeweils innersten Bäume der Ableitungsfolge mit der Umkehrung der Adjunktionsdefinition eliminiert. Die so modifizierte Ableitungsfolge ist Ausgangsbasis für den nächsten Iterationsschritt. Das Verfahren terminiert (spätestens nach n Schritten), wenn die Ableitungsfolge nur noch aus dem initialen Baum $\alpha$ besteht.

Obige Überlegungen müssen jetzt noch auf die abstrakten Datenstrukturen unseres Parsingalgorithmus (d.h. auf die CYK-Matrix) übertragen werden. Wenn Ableitungsfolgen existieren, so sind sie in der CYK-Matrix kodiert. Wir wissen, daß ein innerster Baum innerhalb der CYK-Matrix durch seine Wurzel und den Fußknoten eindeutig beschrieben ist. Da die Adjunktion eine lokale Operation darstellt, muß man in der CYK-Matrix weder Wissen über den Teilbaum über dem Wurzelknoten noch Wissen über den Teilbaum unter dem Fußknoten haben. Ein innerste Baum (bzw. alle seine Knoten) können eliminiert werden, ohne daß dadurch Wechselwirkungen mit den übrigen Matrixeinträgen entstehen.

Konkret wird die Elimination realisiert, indem man den Matrixeintrag, der den Fußknoten repräsentiert, in den Eintrag kopiert, der für die Wurzel steht. Dadurch sind alle Teilbäume, die unter dem Fußknoten als erkannt galten, nun unter der Wurzel bekannt.

Im nächsten Iterationsschritt sind die elementaren Bäume zu finden, die durch die Modifikation der CYK-Matrix nun zu innersten Bäumen geworden sind. Dazu braucht man nur die lokalen Nachbarschaften der zuletzt eliminierten Bäume zu betrachten. Aufwendige Untersuchungen auf der CYK-Matrix als Ganzes sind nicht nötig.

Das Beispiel aus Abbildung 3.26 soll die Arbeitsweise der Iteration veranschaulichen.

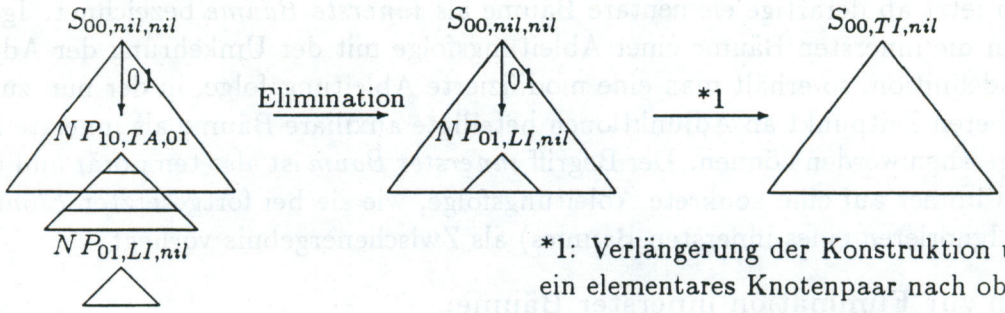



Abbildung 3.26: Beispiel der Iteration zur Elimination innerster Bäume

Ausgangspunkt ist die CYK-Matrix für das vorangehende Beispiel nach Beendigung der CYK-Analyse (siehe Abbildung 3.25). Der besseren Darstellbarkeit wegen erfolgt die Veranschaulichung an der Struktur des Ableitungsbaumes und nicht anhand der CYK-Matrix.

Der schraffierte Baum ist als innerster Baum erkannt. Das Dreitupel an den Knotenlabels bezeichnet die Knotennummer, die Baumart und den Pointer auf den Fußknoten.

In einem ersten Schritt wird die Adjunktion rückgängig gemacht, der innerste Baum also eliminiert. Man bemerke, daß sich die Information am Knoten 00 zu diesem Zeitpunkt noch nicht ändert. Erst im nächsten Schritt orientiert sich der Parser vom Knoten 01 ausgehend um ein elementares Kantenpaar zum Knoten 00 nach oben. Im Beispiel wird davon ausgegangen, daß der Knoten 02 zu diesem Zeitpunkt eine vollständige Untergeschichte besitzt, d.h. die Baumart berechnet ist (hier "LI"). Das bedeutet, daß dem Knoten 00 die Baumart "TI" zugewiesen werden kann, womit die Abbruchbedingung der Iteration erfüllt und die Eingabe akzeptiert ist.

Der Algorithmus von Harbusch parst mit einer Laufzeit von $O(n^4(logn))$ Zeiteinheiten (n = Länge des Eingabesatzes) und stellt gegenüber dem bisher in Abschnitt 3.1 behandelten Algorithmus von Vijay-Shanker und Joshi (siehe auch [12]) mit einer Laufzeit von $O(n^6)$ eine deutliche Verbesserung dar.

Eine exakte Argumentation, die die Laufzeit des Algorithmus von Harbusch belegt, würde an dieser Stelle zu weit führen und ist in [3] nachzulesen. Intuitiv kann sich der Leser das gegenüber dem Algorithmus von Vijay-Shanker und Joshi deutlich bessere Laufzeitverhalten wie folgt vorstellen:

Die Laufzeit des Algorithmus von Harbusch wird von der Iteration zur Elimination innerster Bäume dominiert. Dem Vorgehen des Algorithmus liegt die Lokalität von Adjunktionen zugrunde. Um einen auxiliaren Baum in der Dreiecksmatrix zu finden, in dem keine weiteren Adjunktionen stattgefunden haben (existiert, wenn die Ableitungsfolge länger als 1 ist), muß man weder Wissen über den Teilbaum über dem Wurzelknoten, noch Wissen über den Teilbaum unter dem Fußknoten haben. Die Information unter dem Fußknoten (in den beiden zusätzlichen Dimensionen abgelegt) muß bei Vijay-Shanker und

Joshi aber gerade in der 4-dimensionalen Matrix aufgebaut und verwaltet werden, was beim Algorithmus von Harbusch entfällt.

Der erhöhte Aufwand im Vergleich zur kontextfreien Analyse erklärt sich dadurch, daß in jeder Zelle der Dreiecksmatrix nicht nur die konstante Anzahl der Nichtterminale verwaltet wird, sondern die erweiterten Knotennummern, wobei jedes Nichtterminal n unterschiedliche Fußknotenpointer haben kann (die Verwaltung kostet O(log n)). Dadurch erhöht sich die Laufzeit auf $O(n^4 \log n)$.

## 3.5 Parsing von TAGs mit Unifikation

Der Algorithmus von Harbusch wird im folgenden um die Unifikation erweitert, wie sie in Abschnitt 2.1.2 vorgestellt wurde.

Da die grundlegende Struktur des Parsingalgorithmus durch Unifikation nicht geändert wird, soll an dieser Stelle nur auf die entsprechenden Erweiterungen eingegangen werden.

Ziel der Erweiterungen ist es gewesen, bereits während der Analyse parallel zum in der CYK-Matrix kodierten Ableitungsbaum die zu diesem gehörende DAG-Struktur aufzubauen.

Eine Analyse des Algorithmus von Harbusch besteht darin, eine kontextfreie Analyse des Eingabesatzes mit der zur TAG gehörenden kontextfreien Grammatik durchzuführen. Korrekte TAG-Ableitungsbäume sind dann (falls vorhanden) in der CYK-Matrix kodiert. Von daher liegt es nahe, einen bereits existierenden Unifikationsformalismus für kontextfreie Grammatiken mit Unifikation in diesen Schritt der Analyse zu integrieren.

Bei den folgenden Überlegungen wird nicht konkret auf den Unifikationsformalismus eingegangen. Wir gehen davon aus, daß dieser den in der abstrakten Beschreibung formulierten Anforderungen genügt. Die Implementierung von PATR unterstützt diese Sicht als "Black Box". Man ruft an beliebiger Stelle das Modul mit zwei zu unifizierenden DAGs auf und erhält als Ergebnis den unifizierten DAG oder "FAIL" zurück (gemäß der Unifikationsdefinition aus Abschnitt 2.1.2).

Auf abstrakter Beschreibungsebene besteht eine Regel einer kontextfreien Grammatik mit Unifikation aus der rein kontextfreien Regel und einer zugehörigen Unifikationsregel. Die Unifikationsregel beschreibt einen nur auf die Regel bezogenen DAG. Betrachtet man die kontextfreie Regel als Baum der Tiefe 1 mit der linken Regelseite als Wurzel (bzw. Vater), so kann man jedem Knoten dieses Baumes einen DAG zuordnen, der eine Subgraphen des zur gesamten kontextfreien Regel gehörenden DAGs darstellt.

Diese Betrachtungsweise ist bei einer anschaulichen Beschreibung des Aufbaus eines Ableitungsbaumes hilfreich. Der Baum entsteht durch die Anwendung der kontextfreien Grammatik. Mit jedem Knoten im Baum ist ein DAG assoziiert, der gemäß der kontextfreien Regel Verbindungen zu den DAGs des Vaterknotens und seiner Brüder haben kann. Bei der kontextfreien Regelanwendung werden Nichtterminale auf der rechten Seite einer Regel durch gleiche Nichtterminale auf der linken Seite einer Regel ersetzt. Für die mit diesen Knoten assoziierten DAGs heißt das, daß sie identifiziert werden. So entstehen Verbindungen zwischen DAGs über eine kontextfreie Regel, d.h. über eine Spezifikationsliste

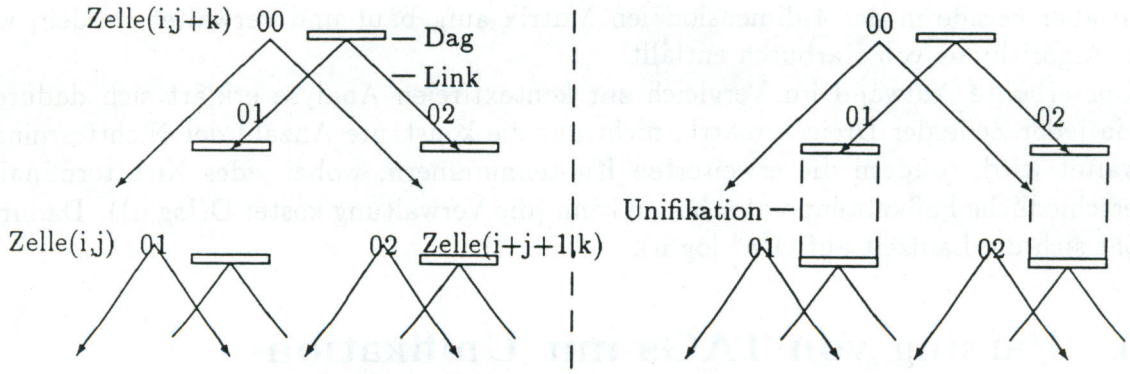



Abbildung 3.27: Einbinden lokaler DAGs

hinaus.

Nun übertragen wir diesen Zusammenhang auf TAGs mit Unifikation. In einem elementaren Baum einer TAG mit Unifikation ist zu jedem inneren Knoten des Baumes eine Unifikationsregel vermerkt, die sich nur lokal auf den Knoten selbst und seine Söhne bezieht. Die Anbindung der Spezifikationen erfolgt also wie im kontextfreien Fall. Da bereits ein ganzer Baum mit Spezifikationen für jeden Vater und seine Söhne versehen ist, findet hier bereits die Identifikation statt wie im kontextfreien Fall beim Aufbau eines Ableitungsbaumes. Es bleibt zu beschreiben, wie das Einfügen von neuen DAGs in dieses Netz erfolgt.

Im Rahmen der CYK-Analyse wird parallel zur kontextfreien Struktur des Ableitungsbaumes die korrespondierende DAG-Struktur aufgebaut. Dazu wird die Information eines Eintrags in einem Element der CYK-Matrix um eine zusätzliche Komponente, den DAG, ergänzt. Um das Vorgehen bei der Identifikation von DAGs beim Aufbau einer baumübergreifenden Struktur zu illustrieren, sei auf der linken Seite von Abbildung 3.27 exemplarisch die folgende bei der CYK-Analyse immer wiederkehrende Situation herausgegriffen.

Angenommen, im Element (i,j) der CYK-Matrix gibt es einen Eintrag mit der Knotennummer 01, im Element (i+j+1,k) einen Eintrag mit der Knotennummer 02 und ferner eine kontextfreie Regel, deren reine Nummernnotation 00 → 01 02 lautet. Bei der rein kontextfreien Analyse (also ohne DAGs) wird nun in das Element (i,j+k) der CYK-Matrix ein Eintrag mit der Knotennummer 00 geschrieben.

Im Fall der Unifikation kann aufgrund der strengen breadth-first-, bottom-up-Analyse davon ausgegangen werden, daß die Einträge in (i,j) und (i+j+1,k) bereits DAGs besitzen, die gemäß ihrer Untergeschichte komplette Teilbäume überspannen. Weiter ist es möglich, lokal zu einer kontextfreien Regel anhand der zugehörigen Unifikationsregel einen DAG aufzubauen, der Teil-DAGs für jeden Vater- bzw. Sohnknoten besitzt, die untereinander durch DAG-Kanten verbunden sind. Dieser lokale DAG kann nun in die baumübergreifende Struktur eingebunden werden (siehe rechte Seite von Abbildung 3.27), indem der DAG zum Sohnknoten 01 mit dem DAG in (i,j) und der entsprechende DAG zum Sohnknoten 02 mit dem DAG in (i+j+1,k) unifiziert wird. Schlägt mindestens eine

der beiden Unifikationen fehl, ist die Regel 00 → 01 02 zu verwerfen.

Allgemein läßt sich sagen, daß, wenn eine kontextfreie Regel bei der Analyse als zutreffend erkannt wird, immer Unifikationen in der oben beschriebenen Weise durchgeführt werden.

Durch dieses Vorgehen ist sichergestellt, daß am Ende der CYK-Analyse zu jedem in der CYK-Matrix kodierten Ableitungsbaum ein "kompletter" DAG existiert.

Im sich nun anschließenden Schritt des Parsingalgorithmus, der Iteration zur Elimination innerster Bäume, sind aufgrund der oben beschriebenen Ergebnisse in punkto Unifikation keine Erweiterungen notwendig.

Vergleicht man den hier vorgestellten Parsingansatz mit unserer formalen Definition von TAGs mit Unifikation, so treten die dort beschriebenen Probleme hier nicht auf. Da die Adjunktionen nicht ausgeführt werden, d.h. in eine bestehende DAG-Struktur eingefügt werden müssen, sondern in einem vorausberechneten Strukturbaum gefunden werden müssen, verläuft die Unifikation wie im kontextfreien Fall bei der Berechnung der Dreiecksmatrix.

Sowohl beim Algorithmus von Harbusch als auch beim Ansatz von Vijay-Shanker und Joshi werden Ableitungsbäume im breadth-first-, bottom-up-Verfahren ausgehend von kontextfreien Regeln aufgebaut. Die eindeutige Beziehung zwischen TAG und zugehöriger kontextfreier Grammatik (mit eindeutigen Knotennummern) stellt die Korrektheit des Vorgehens sicher.

Gleiches gilt auch für die Erweiterung um Unifikation. Jeder innere Knoten eines elementaren Baumes besitzt eine eigene Unifikationsregel, die nur den Knoten selbst und seine Söhne betrifft. D.h. die Zerlegung eines elementaren Baumes einer TAG mit Unifikation in kontextfreie Regeln mit Unifikation vollzieht sich immer noch auf eindeutige Weise. Der Aufbau eines Ableitungsbaumes ausgehend von kontextfreien Regeln mit Unifikation stellt die Umkehrung dieser Zerlegung dar.

Daraus folgt, daß der den Baum überspannende DAG bei einem **konstruierten** Ableitungsbaum identisch ist mit dem DAG beim gleichen, jedoch **analysierten** Ableitungsbaum.

Wie im Abschnitt 2.1 genauer beschrieben, geht bei unserer formalen Definition von TAGs mit Unifikation die Monotonie der Unifikation verloren, da bereits propagiertes Wissen zurückgenommen oder modifiziert werden kann. Demgegenüber gestaltet sich die Umsetzung von Unifikation bei unserem Parsing-Ansatz vergleichsweise einfach. Durch die strenge bottom-up-Analyse des CYK-Verfahrens ist die Monotonie der Unifikation immer gewährleistet. Aufwendige Neuberechnungen von DAGs sind nicht nötig.

In Abschnitt 3.1.2 haben wir bereits darüber spekuliert, daß sich unsere Erweiterungen zum Parsing von TAGs mit Unifikation in gleicher Weise in den Basis-Algorithmus von Vijay-Shanker und Joshi integrieren lassen. Die oben aufgeführten Überlegungen unterstreichen diese Ansicht.

Damit haben wir grob die Umsetzung unserer Definition von TAGs mit Unifikation in einem Algorithmus beschrieben. Im nächsten Abschnitt erläutern wir die Funktionalität der einzelnen Komponenten einer Implementierung dieses Algorithmus.

# Kapitel 4

# Implementierung

Das Programmpaket zur Analyse von TAGs mit Unifikation setzt sich aus folgenden Teilen zusammen: Grammatikeingabe, Normalformtransformation und Parsingalgorithmus. Diese Teile sind modular realisiert worden. Die grundlegenden Objekte und Funktionen für Unifikation, Lexikonverwaltung und -zugriff haben wir aus D-PATR übernommen. Der von Karttunen (siehe [7]) stammende Formalismus wurde von Wolfgang Finkler und Günther Neumann nach Zeta-LISP übertragen (SB-PATR, siehe [1]) und von uns nach Common-LISP portiert. Einzelheiten der Implementierung dieses Moduls sind in der angegebenen Literatur nachzulesen.

Zu jedem Modul, das einer Datei gleichen Namens entspricht, geben wir im folgenden einen Überblick an. Zunächst beschreiben wir als "Erweiterungen der Definition", was an Änderungen und Anpassungen des von uns definierten Formalismus nötig war, um ihn im Computer zu realisieren. Im Abschnitt "Realisierung" jedes Moduls erläutern wir benutzte Datenstrukturen und Operatoren vor der eigentlichen Beschreibung des Algorithmus, damit wir dort auf der Basis definierter Begriffe argumentieren können. Der Leser, der vorwiegendes Interesse an der Realisierungsidee hat, kann sich durchaus am Abschnitt "Algorithmus" orientieren und die beiden vorhergehenden als Nachschlagewerk betrachten. Anschließend findet man wesentliche Funktionen als Struktogramme mit umgangssprachlichen Einträgen und am Ende Aussagen über die Handhabung jedes Moduls, die ausreichen, um mit den vorher beschriebenen Werkzeugen sinnvoll umzugehen. Im folgenden benutzen wir "umgangsspachliche" Begriffe aus den Programmiersprachen C und LISP, die sich im allgemeinen Sprachgebrauch eingebürgert haben. Sollte der Leser mit den Ausdrücken nicht vertraut sein, so kann er sie in [11] nachlesen.

Wir beginnen mit der Beschreibung der Handhabung von Ein- und Ausgabe.

## 4.1 Ein- und Ausgabe

### 4.1.1 Ein- und Ausgabe und Aufbau der TAG-Grammatik

Die Funktionen zur Ein- und Ausgabe und zum Aufbau der TAG-Grammatik sind in der Datei "grammar-io.l" zusammengefaßt.

### Erweiterungen der Definition

Die Realisierung von Ein- und Ausgabe und der Aufbau der TAG-Grammatik erforderten keine Anpassung der Definition, wie sie in Kapitel 2 eingeführt wurden.

### Datenstrukturen

Wir greifen hier den Operatoren zur Eingabe und Speicherung der Grammatik vor, da die Überlegungen zur Datenstruktur der Planung der entsprechenden Funktionen vorausgehen. Die eingelesene und compilierte Version der TAG-Grammatik wird unter der globalen Variable *grammar* als Struktur abgelegt.

*(defstruct grammar*
*tlist, ntlist, treelist, chntlist, chtreelist)*

*Tlist* und *ntlist* sind die Listen der Terminale und Nichtterminale aus der Eingabe bis auf eine obligatorische Erweiterung der Terminale um das leere Wort "eps". *Chntlist* sammelt die neue Menge von Nichtterminalen, wie sie bei der Normalform-Transformation entstehen. *Treelist* ist die Liste der TAG-Bäume vor, *chtreelist* nach der Umwandlung in die Normalform. Ihre Struktur ist folgendermaßen vereinbart:

*(defstruct tree*
*rootp, number, i-or-a, leafplist, footp)*

Im Slot *rootp* steht ein Verweis auf den Wurzelknoten des Baumes, in *footp* NIL oder ein Verweis auf den Fußknoten des auxiliaren Baumes, in *leafplist* die Liste der Blätter (Terminale), in *number* die Nummer des Baumes als String ("0", ...,"n") und in *i-or-a* ein 'a bei auxiliaren Bäumen, NIL sonst. Die Knoten, auf die hier verwiesen wird, sind wiederum als Struktur realisiert:

*(defstruct node*
*number, constraints, dag, rule, defs, prep, succplist, label, hprep,*
*hsuccplist)*

*Number* enthält die Nummer des Knotens als String, die (wie schon im 3. Kapitel beschrieben) nach dem Pfad von der Wurzel bis zu diesem Knoten aufgebaut ist ("n1.n2.n3. ..."). In *prep* wird der Vater, in *succplist* eine Liste der Söhne dieses Knotens gespeichert. Der Slot *label* erhält den Stringnamen des Knotens (d.h. den benutzerdefinierten Namen bzw. auch Hilfssymbole nach der Transformation in Normalform).

*Dag* und *rule* erhalten identische Kopien der zum Knoten gehörenden Unifikationsliste, in *dag* wird später die "compilierte" Version als Graph mit DAG-Kanten stehen. In der Zwischenzeit kann man *dag* als Hilfsspeicher für die Zweiform-Transformation benutzen, bei der auch die Regeln angepaßt werden müssen.

Im Slot *defs* wird in der Wurzel jedes auxiliaren Baums die Vererbungsrichtung jedes

im Wurzel- und Fußknoten sichtbaren Attributpräfixes gespeichert (siehe Kapitel 4.3, wo die Vererbungsrichtung berechnet und bei der Value-Definition während einer Adjunktion benutzt wird).

*Constraints* werden erst bei der Umwandlung in unsere Normalform wichtig und können bedeuten:

**('SA)** keine Adjunktion,

**('OA n1 n2 ...)** obligatorische Adjunktion der Bäume mit Nummer n1, n2, ... und

**('SA n1 n2 ...)** selektive Adjunktion der Bäume mit Nummer n1, n2, ....

Die Slots *hprep* und *hsuccplist* werden benötigt, um die bei der Realisierung der Kettenfreiheit durch Zusammenziehen von Ketten verschwindenden Knoten abzulegen. Dieser versteckte Erhalt der Ketten vereinfacht den späteren Umgang mit den Hilfsknoten, die die jeweilige Kette ersetzen.

### Operatoren

**read-input:** kommuniziert über die Funktion *read-help* mit dem Benutzer. Näheres darüber findet man im Abschnitt "Handhabung". Nach Einlesen eines Baumes als String wird dieser in eine geschachtelte Listenstruktur mit Strings als atomare Einheiten umgewandelt und an die Funktion *tree-to-grammar* übergeben.

**tree-to-grammar:** erhält den Listen-Baum der Eingabestruktur als Eingabeparameter, ruft *compile-tree* mit der neuen Baumnummer, dem Baum und 0 als erste Ergänzung der Blattnummer auf. Bei erfolgreicher Umwandlung wird das Ergebnis von *compile-tree* in *grammar* abgespeichert.

**compile-tree:** geht den Baum Knoten für Knoten durch, ordnet die erkannten Knoten nach Stellung im Baum und speichert das vorläufige Ergebnis jeweils in einer lokalen Variablen.

**fill-node:** erzeugt eine Knotenstruktur und verzeigert den Knoten mit dem Vorgänger im Baum.

### Algorithmus

Das gesamte Modul wird über den Aufruf der Funktion *read-input* gestartet. Hier wird nacheinander die Liste der Terminale und Nichtterminale abgefragt und so lange Bäume eingelesen, bis "E" (für Ende) eingegeben wird.

Nach dem Einlesen eines Baums wird dieser mithilfe eines einfachen Parsers von einem String in eine Liste umgewandelt, als Parameter an *tree-to-grammar* übergeben und bei erfolgreicher Umwandlung in Grammatik abgespeichert.

Die Funktion *tree-to-grammar* bildet den Einstieg in die Compilierungs-Funktionen, indem sie *compile-tree* für den eingegebenen Baum aufruft und bei Erfolg den umgeformten Baum abspeichert.

*Compile-tree* arbeitet den Baum Knoten für Knoten ab:

- der Label wird auf Vorhandensein in *tlist* oder *ntlist* getestet.
- hat der Knoten eine besondere Stellung im Baum (Wurzel, Blatt, Fußknoten), so wird er im entsprechenden Slot des kreierten Ausgabebaums abgelegt.
- Für jeden neuen Knoten wird die Funktion *fill-node* mit dem Namen des Knotens, der Spezifikationsliste, der aktuellen Knotennummer und ihrer Ergänzung aufgerufen, die eine aktuelle Knotenstruktur bildet und mit dem Vorgänger verzeigert.

### Handhabung

Die Eingabe erfolgt vom Benutzer über das Terminal. Erwartet wird die Eingabe einer TAG-Grammatik in folgenden Teilen:

- Liste aller vorkommenden Terminale (bzw. Präterminale).
  **(t1 t2 t3 ...)**
- Liste aller vorkommenden Nichtterminale.
  **(nt1 nt2 nt3 ...)**
- Jeweils ein Baum als Liste
  **(nt1 (Unifikationsliste) (Liste der Söhne))**
  wobei **nt1** die Wurzel des Baums bezeichnet und die Liste der Söhne NIL oder wiederum als Liste
  **(nt11 (Unif.) (Söhne) nt12 (Unif.) (Söhne) ...)**
  aufgebaut ist.
- Die Unifikationsliste besteht aus einer Auflistung der Unifikationsregeln, die auf Vater und Söhne nur als 0 und 1 ...n referenzieren. Die Liste kann auch leer sein oder einfach weggelassen werden (dann werden TAGs ohne Unifikation bearbeitet). Jede Unifikationsregel ist von der Form
  **(Pfad Value)** oder
  **(Pfad Pfad)**
  Ein Pfad ist entweder die Nummer eines Konstituenten in Klammern oder eine Liste aus dieser Nummer und den Attributen bis zum gewünschten Punkt des Knoten-Features (siehe [1]).

Beispiel:
Terminale **(a b)**
Nichtterminale **(S F)**
1. Baum **(S (((0 wert) (1 wert)) ((0 attr) y)) (F (((0 wert) x)) (a b)))**
ist die korrekte Eingabe für den Baum in Abbildung 4.1.

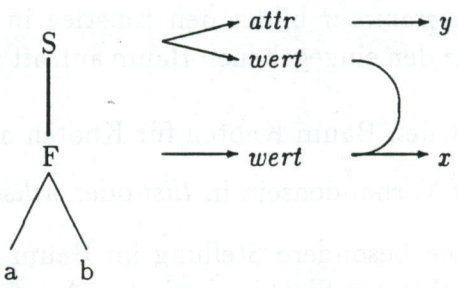



Abbildung 4.1: Beispiel eines Eingabebaums

Beendet wird die Eingabe mit **E**. Die compilierten Bäume werden in der Variablen *grammar* abgelegt.

Sollten bei der Eingabe der Bäume Konstituenten auftreten, die nicht in *tlist* oder *ntlist* gespeichert sind, so erhält der Benutzer die Gelegenheit, den Namen zu ändern oder neu abzulegen:

**"x" is not in tlist or ntlist**
**Shall I substitute?**
Y → **What label?**
"y" → x durch y ersetzt, falls y bekannt
N → **Shall I put it in tlist?**
Y → tlist vereinigt mit ("x")
N → **I will put it in ntlist**
ntlist vereinigt mit ("x")

Stimmen *root*- und *foot*-Label eines eingegebenen auxiliaren Baumes nicht überein, so erscheint die Fehlermeldung

**Root and foot don't match**

und die Information

**Tree couldn't be saved.**

Treten in einem auxiliaren Baum mehrere Fußknoten auf, so erfährt der Benutzer:

**More than one foot node**
**Tree couldn't be saved**

Ebenfalls nicht abgespeichert wird ein auxiliarer Baum, dessen Blätter nur aus leeren Worten bestehen.

Als weitere Fehlermeldungen können nur solche des LISP-Systems auftreten, wenn die eingegebenen Parameter nicht die geforderten Voraussetzungen erfüllen.

### 4.1.2 Lexikon

#### Erweiterungen der Definition

In der abstrakten Version des Algorithmus spielt das Lexikon nur eine untergeordnete Rolle. Eine gegenseitige Abhängigkeit besteht nur insoweit, als daß das Lexikon Auskunft über die lexikalische Kategorie des Terminals geben muß und weiter einen zugehörigen DAG liefert, dessen Datenstruktur mit der im Parser verwendeten übereinstimmt.

Da alle DAGs ohnehin mit den im SB-PATR-Modul (siehe [1]) verwendeten Mechanismen und Datenstrukturen aufgebaut werden, lag es nahe, auch das dort verwendete Lexikon einzusetzen.

In seiner gegenwärtigen Form verfügt das Lexikon nicht über einen morphologischen Analyser, der gebeugte oder abgeleitete Formen des Terminals mit Einträgen in einem morphologischen Lexikon in Beziehung bringt. Im Hinblick auf eine bessere Lösung splittet SB-PATR das Lexikon momentan in zwei Teile auf: Wörter und Stämme. Das Format der Einträge in beiden Lexika ist das gleiche, aber beim Abarbeiten eines Eingabesatzes wird nur das Wort-Lexikon hinzugezogen. Die Einträge des Wort-Lexikons können wiederum einen Verweis auf das Stamm-Lexikon enthalten. Eine detailliertere Beschreibung der beiden Lexika würde hier sicher zu weit führen; diesbezüglich sei auf die oben erwähnte Literatur verwiesen.

#### Datenstrukturen

Neben der globalen Variablen *grammar* und der gleichnamigen Structure werden lediglich indirekt die elementaren Datenstrukturen des SB-PATR-Moduls verwendet.

#### Operatoren

**look-up-or-fake:** stellt die Lexikonschnittstelle des SB-PATR-Moduls dar.

**readings-from-lex:** stellt die auf *look-up-or-fake* aufgesetzte Lexikonschnittstelle des Parsingalgorithmus dar.

#### Algorithmus

Die hier eingesetzte Lexikonschnittstelle des SB-PATR-Moduls liefert zu einem gegebenen Terminal eine Liste von DAGs. Jeder DAG entspricht einer Lesart des Terminals. Auf diese Schnittstelle aufgesetzt wurde die von uns neu definierte Lexikonschnittstelle des Parsingalgorithmus. Wieder wird eine Liste von Lesarten returniert. Eine Lesart ist diesmal jedoch ein Zweitupel, bestehend aus der lexikalischen Kategorie in Form eines Strings und einem DAG. Zusätzlich übt die aufgesetzte Schnittstelle eine Filterwirkung

aus, da nur solche Zweitupel Bestandteil der Ausgabe sind, deren lexikalische Kategorie in der Terminal-Liste der definierten Grammatik aufgeführt ist.

Wegen der Einfachheit der Funktion *readings-from-lex* erübrigt sich die Darstellung des Algorithmus in Form eines Struktogramms. Der LISP-Code der von uns definierten Lexikonschnittstelle findet sich in der Datei "cyk.l".

#### Handhabung

#### Parameter

Der Parameter der Funktion *readings-from-lex* ist ein Symbol, das für ein Terminal steht. Um eine korrekte Arbeitsweise zu garantieren, müssen die Dateien des SB-PATR-Moduls geladen und an die globale Variable *grammar* eine TAG gebunden sein.

#### Ausgabe

Die Ausgabe besteht aus einer Liste 2-elementiger Listen (genaueres dazu findet sich oben unter 'Algorithmus').

#### Fehlermeldungen

An Fehlermeldungen können nur solche des SB-PATR-Moduls auftreten, sofern die im Abschnitt 'Parameter' geforderten Voraussetzungen nicht erfüllt wurden.

## 4.2 Transformation in Normalform

Im folgenden werden die Implementationsschritte zum Erreichen der Normalform beschrieben. Den Anfang machen für die Transformation notwendige Hilfsfunktionen.

### 4.2.1 Erzeugen von DAGs und DAG-Kanten

Die Funktionen zum Erzeugen von DAGs und DAG-Kanten sind in der Datei "dag-unify.l" zusammengefaßt.

#### Erweiterungen der Definition

Wie schon gezeigt, macht es die Umformung der Grammatik in eine kettenfreie Grammatik nötig, an einem Knoten nicht nur einen sondern eine ganze Liste von DAGs abzulegen (siehe Kapitel 2.2). Zum leichteren Umgang mit der DAG-Liste wird parallel dazu eine Reihe von Hilfsknoten angelegt, von denen jeder einen DAG trägt, der identisch mit einem DAG der Liste ist. Diese Erweiterung macht einige zusätzliche Hilfsfunktionen zum Umgang mit DAGs nötig.

### Datenstrukturen

Es wird auf den *grammar*-Strukturen mit Listen von DAGs anstelle von einzelnen DAGs gearbeitet.

### Operatoren

Das Modul "dag-unify.l" verwendet grundlegende Funktionen der Implementierung von SB-PATR durch Wolfgang Finkler und Günter Neumann an der Universität des Saarlandes (siehe [1]), die von uns zunächst von Zeta-LISP nach Common-LISP übertragen wurde.

**dag-to-rule:** hängt vor die Liste der Unifikationsregeln des Knotens node eine Liste der Konstituenten (Label des Vaters und der Söhne). Die Funktion ist nur anwendbar vor der Umwandlung in "dicke DAGs", d.h. vor und während der Transformation in Zweiform. Der Einfachheit halber haben wir hier auf Listen von DAGs verzichtet, die erst zum Abschluß der Zweiformtransformation eingeführt werden und Voraussetzung für die folgenden Transformationsschritte sind.

**rule-unify:** unifiziert die DAG-Teile mit Pfadanfang 1 ...n eines Knotens node mit n Söhnen mit den DAG-Teilen nach Attribut 0 des jeweiligen Sohns. Die Funktion ist nur anwendbar vor der Umwandlung in "dicke DAGs".

**fat-dag-to-node, fat-rule-unify:** übernehmen die analogen Funktionen für Knoten, die Listen von DAGs haben. Nun wird der letzte DAG des Knotens um die Konstituentenregel ergänzt und mit den jeweils ersten DAGs der Nachfolgeknoten unifiziert.

**h-dag-to-node, h-rule-unify:** übernehmen die analogen Funktionen für Knoten, die in einem Hilfsknotenzyklus stehen, wie er bei der Bildung der Kettenfreiheit entsteht. Sie verwenden das Wissen, daß jeder solche Knoten nur einen DAG und einen Nachfolger haben kann.

### Algorithmus

Die Funktionen *dag-to-rule* und *rule-unify* werden erst am Ende des Moduls "two-form-tr.l" aufgerufen, das einen Baum in Zweiform umwandelt. Der Grund dafür ist, daß bei der Zweiformtransformation selber noch die Regeln (in *node-rule*) verändert werden, ansonsten aber auf der reinen Baumstruktur (ohne DAGs) gearbeitet wird. Eine Erzeugung und Verzeigerung der DAGs macht demnach erst am Ende dieses Transformationsschrittes Sinn. Die Funktionen *fat-dag-to-node, fat-rule-unify, h-dag-to-node* und *h-rule-unify* werden zur Realisierung der Adjunktion benötigt, wenn ausgehend von der Wurzel des eingehängten Auxiliarbaumes nach oben bis zur Wurzel des Anhängebaumes noch einmal alle DAGs aufgelöst und neu verzeigert werden (siehe auch theoretische und praktische Überlegungen zur Adjunktion). Sicher sind diese Funktionen zu grob und überschreiten

den Aufwand, der nötig wäre, um die DAG-ups nach oben im Baum neu zu setzen. Direkte Zugriffe auf für die Hochvererbung in Frage kommende Pfade wären sinnvoller, jedoch auch schwieriger zu realisieren, da für jede DAG-Kante der Baum nach oben auf Weiterpropagierung untersucht werden müßte. Es gibt demnach noch sinnvolle Verbesserungen dieser Funktionen, die wir aber aus Zeitgründen nicht in Angriff genommen haben.

**rule-unify**

| | |
|---|---|
| Eingabe: node | {* Knoten mit compiliertem DAG *} |
| i := 1 | |
| ↓ i = Länge der Nachfolgeliste von node | |
| (unify $T_1$ $T_2$) | {* $T_1$ ist der i-Teil des DAGs von node, $T_2$ der 0-Teil des i-ten Nachfolgers von node *} |
| i := i+1 | |
| Ausgabe: node | {* Knoten mit Links zu den Söhnen*} |

Abbildung 4.2: Struktogramm zu rule-unify

**fat-rule-unify**

| | |
|---|---|
| Eingabe: node | {* Knoten mit compiliertem dicken DAG *} |
| i := 1 | |
| ↓ i > Länge der Nachfolgerliste von node | |
| (unify $T_1$ $T_2$) | {* $T_1$ ist der i-te Teil des letzten DAGs von node, $T_2$ der 0-te des ersten DAGs des i-ten Nachfolgers von node *} |
| i := i+1 | |
| Ausgabe: node | |

Abbildung 4.3: Struktogramm zu fat-rule-unify

**h-rule-unify**

| | |
|---|---|
| Eingabe: node | {* Knoten mit Hilfsknotenzyklus *} |
| Eingabe: n | {* Position des aktuellen Hilfsknotens *} |
| (unify $T_1$ $T_2$) | {* $T_1$ ist der 1-te Teil des ersten DAGs von node, $T_2$ der 0-te des ersten DAGs des Hilfsnachfolgers von node *} |

Abbildung 4.4: Struktogramm zu h-rule-unify

#### Handhabung

Die genaue Verwendung der Funktionen dieser Datei ist für den Benutzer uninteressant, denn sie werden bei einzelnen Transformationsschritten und nur von bestimmten Programmen beim Erzeugen neuer DAGs und beim Adjungieren aufgerufen.

Fehlermeldungen treten entweder beim Compilieren oder Unifizieren der DAGs auf und sind dann in den entsprechenden Dateien aus SB-PATR zu finden.

### 4.2.2 Berechnung der Vererbungsrichtung im auxiliaren Baum

Die Funktionen zur Berechnung der Vererbungsrichtung im auxiliaren Baum sind in der Datei "dag-direct" zusammengefaßt. Die Berechnung der Vererbungsrichtung im auxiliaren Baum wird vor der Transformation ausgeführt, so daß sie nur einmal für jeden elementaren Baum gemacht werden muß und zum Adjunktionszeitpunkt zur Verfügung steht.

#### Erweiterungen der Definition

Zur Berechnung der Vererbungsrichtung waren keine Erweiterungen der Definition notwendig.

#### Datenstrukturen

Es werden Listen erzeugt, die als erstes Element den minimalen Präfix eines DAG-Pfades im Baum haben; die folgenden Elemente sind die Nummern der Knoten, in denen Werte, die im Anhängeknoten hinter diesem Präfix gespeichert sind, abgelegt werden sollen. Treten im Baum "dicke DAGs" (zur Definition siehe Kapitel 2.2) auf, und soll der Wert in einen DAG einer solchen Liste gesetzt werden, so beginnt die Knotennummer mit h und bezieht sich auf eine Nummerierung des Baumes, die Hilfsknoten den regulären Knoten vorzieht.

*((Präfix1 "n11" "n12" ...) (Präfix2 "n21" "n22" ...) ...)*

Im Slot *dag-defs* der auxiliaren Wurzel wird die Liste dieser Ergebnisse abgelegt.

#### Operatoren

**get-def-nodes:** liest alle erkennbaren Pfade aus Wurzel und Fußknoten, verfolgt sie durch den Baum und schreibt die Vererbungsknoten mit dem übriggebliebenen Präfix in *node-defs*.

**get-dir:** untersucht den Baum von der Wurzel aus nach den Vererbungsknoten für einen Pfadpräfix.

**get-r-f-dir:** vereinigt die von Wurzel und Fußknoten aus gefundenen Vererbungsknoten.

**rpath-dir:** untersucht den Baum von der Wurzel aus nach der Vererbungsrichtung eines Pfadpräfix.

**fpath-dir:** untersucht den Baum vom Fußknoten aus nach der Vererbungsrichtung eines Pfadpräfix.

**direction:** untersucht im aktuellen Knoten die Vererbungsrichtung und das verbleibende Präfix.

### Algorithmus

Die theoretischen Überlegungen zum Verwendungszweck und zur Berechnung der Vererbungsrichtung sind bereits in Kapitel 2.2.2 dargestellt worden, so daß hier nur noch der oberflächliche Ablauf der Richtungsberechnung mit den Funktionsnamen belegt werden muß .

Für jeden Pfad, der in den Unifikationsregeln der Wurzel des untersuchten auxiliaren Baumes gefunden wird (*get-def-nodes*) führt man folgende Untersuchung durch:

1. Der Baum wird von der Wurzel aus untersucht (*get-dir*). Durch den Aufruf von *rpath-dir* wird zunächst festgestellt, welche Vererbungsrichtung von der Wurzel aus vorliegt und ob ein durchgängiger Pfad von der Wurzel bis zum Fußknoten gefunden wurde.

   **(dir T)** bedeutet, daß ein durchgängiger Pfad gefunden wurde. Für *dir* = *'d* oder *dir* = *'b* ist der Ergebnisknoten der Fußknoten, für *dir* = *'u* die Wurzel.

   **(dir node)** bedeutet, daß der Pfad im Baum bei *node* aufhört. Für *dir* = *'u* oder *'b* ist der Ergebnisknoten die Wurzel, für *dir* = *'d* der lokale Knoten *node*.

   **(NIL node)** bedeutet, daß der Pfad zwar in der Wurzel, aber in keinem ihrer Söhne auftritt. Der Ergebnisknoten ist die Wurzel.

2. Nach Eintritt der letzten beiden Fälle ist es nötig, den Baum noch einmal vom Fußknoten aus zu untersuchen (*get-r-f-dir*). Die Vererbungsrichtung stellt hier die Funktion *fpath-dir* fest.

   **(dir node)** bedeutet, daß der Pfad im Baum bei *node* aufhört. Für *dir* = *u* ist der Ergebnisknoten der lokale Knoten *node*, für *dir* = *d* oder *b* der Fußknoten.

   **(NIL node)** bedeutet, daß der Vorgänger des Fußknotens kein Präfix des Pfades besitzt. Der Ergebnisknoten ist der Fußknoten.

**get-def-nodes**

| Eingabe: tree {* auxiliarer TAG-Baum *} |
|---|
| pp := NIL |
| parts := $E_1$ {* Liste der kürzesten Pfade (Präfixe) *} |
| für jeden Pfad erg aus parts, jeweils ohne die erste Zahl ((0 a x) → (a x)) |
| erg := (get-dir tree erg) {* Liste der Knoten, an denen def stehen soll *} |
| erg ≠ () → (node-defs (tree-rootp tree)) := erg ∪ (node-defs (tree-rootp tree)) |
| Ausgabe: (node-defs (tree-rootp tree)) |

Abbildung 4.5: Struktogramm zu get-def-nodes

$E_1$

| list := Regelliste von Wurzel (R), Vorgänger des Fußknotens (F-V) und Fußknoten (F) | | | | |
|---|---|---|---|---|
| erg := NIL | | | | |
| für jede Regel aus list | | | | |
| Regel | | | | |
| aus R: ((0 x) (1/2 y)) oder ((0 x) (0 y)) | aus R: ((0 x) y) | ((1/2 x) (i y)) | aus F-V: ((i x) (1/2 y)) | aus F: ((0 x) z) |
| $E_2$ | $E_3$ | $E_4$ | $E_5$ | $E_4$ |
| Ausgabe: erg | | aus F-V: | | |

Abbildung 4.6: Struktogramm zu E1 (get-def-nodes)

$E_2$

| pp := (has-pref (notnum (caar list) erg) {* untersucht 1. Regelhälfte (RH) *}<br>∘ pp ≠ T {* 1. RH hat kein Präfix in erg *} → erg := erg ∪ Präfix aus 1. RH<br>∘ pp ≠ T {* 1. RH ist Präfix von pp in erg *} → erg := erg \ pp |
|---|
| pp := (has-pref (notnum (cadar list)) erg) {* untersucht 2. Regelhälfte (RH) *}<br>∘ pp ≠ T {* 2. RH hat kein Präfix in erg *} → erg := erg ∪ Präfix aus 2. RH<br>∘ pp ≠ T ∧ pp ≠ NIL {* 2. RH ist Präfix von pp in erg *} |
| erg := erg \ pp |

Abbildung 4.7: Struktogramm zu E2 (get-def-nodes)

$E_3$

| |
|---|
| pp := (has-pref (notnum (caar list)) erg) {* untersucht 1. Regelhälfte *} |
| pp ≠ T {* 1. Regelhälfte hat kein Präfix in erg *}<br>→ erg := erg ∪ Präfix aus 1. Regelhälfte |
| pp ≠ T ∧ pp ≠ NIL {* 1. Regelhälfte ist Präfix von pp in erg *}<br>→ erg := erg \ pp |

Abbildung 4.8: Struktogramm zu E3 (get-def-nodes)

$E_4$

| | |
|---|---|
| pp := (has-pref (notnum (caar list)) erg) {* untersucht 1. Regelhälfte *} | |
| pp ≠ T {* 1. Regelhälfte hat kein Präfix in erg *}<br>→ erg := erg ∪ Präfix aus 1. Regelhälfte | |
| pp ≠ T ∧ pp ≠ NIL {* 1. Regelhälfte ist Präfix von pp in erg *}<br>→ erg := erg \ pp | |
| (car list) = ((foot-number x) (foot-number y))<br>ja | nein |
| pp := (has-pref (notnum (cadar list)) erg) {* 2. Regelhälfte *} | — |
| pp ≠ T {* 2. Regelhälfte hat kein Präfix in erg *}<br>→ erg := erg ∪ Präfix aus 2. Regelhälfte | |
| pp ≠ T ∧ pp ≠ NIL {* 2. Regelhälfte *}<br>→ erg := erg \ pp | |

Abbildung 4.9: Struktogramm zu E4 (get-def-nodes)

$E_5$

| |
|---|
| pp := (has-pref (notnum (cadar list)) erg) {* untersucht 2. Regelhälfte *} |
| pp ≠ T {* 2. Regelhälfte hat kein Präfix in erg *}<br>→ erg := erg ∪ Präfix aus 2. Regelhälfte |
| pp ≠ T ∧ pp ≠ NIL {* 2. Regelhälfte ist Präfix von pp in erg *}<br>→ erg := erg \ pp |

Abbildung 4.10: Struktogramm zu E5 (get-def-nodes)

**get-dir**

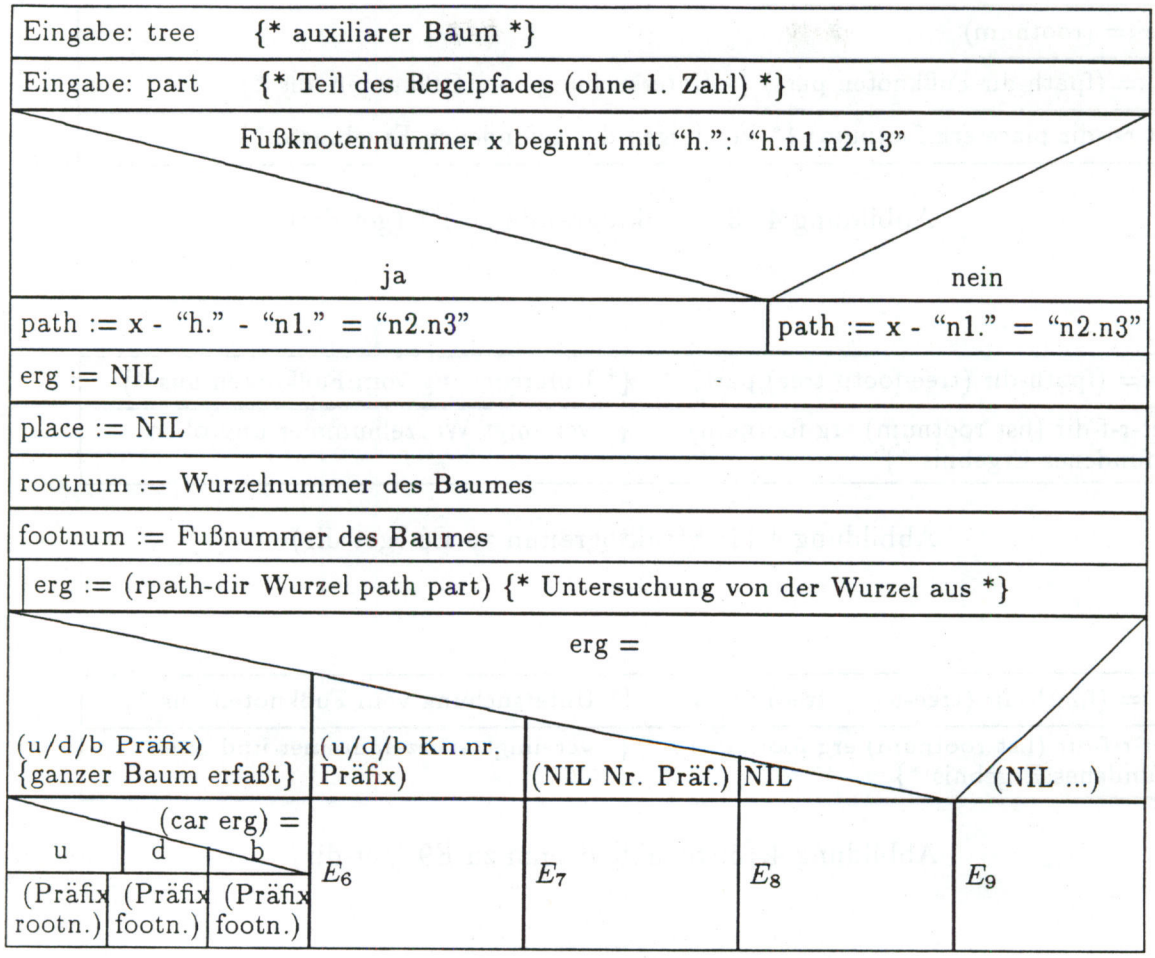



Abbildung 4.11: Struktogramm zu get-dir

$E_6$

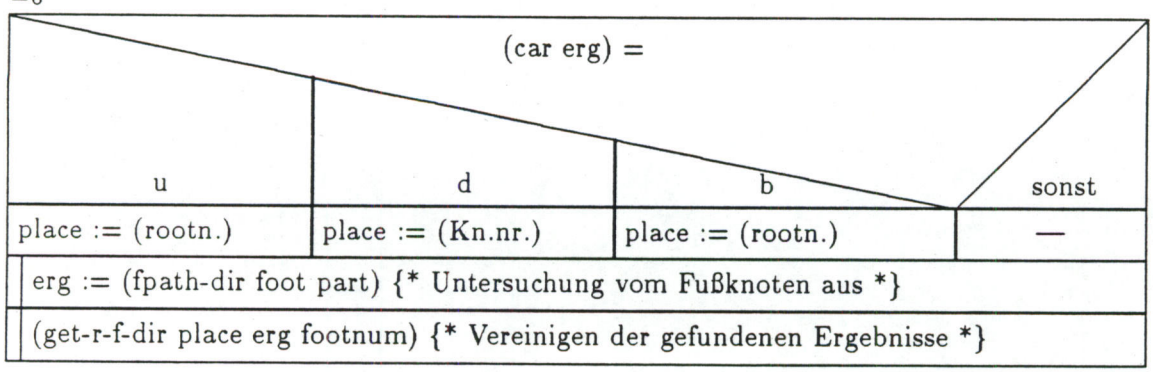



Abbildung 4.12: Struktogramm zu E6 (get-dir)

$E_7$

| place := (rootnum) |
|---|
| erg := (fpath-dir Fußknoten part) {* Untersuchung vom Fußknoten aus *} |
| (get-r-f-dir place erg footnum) {* Vereinigen der gefundenen Ergebnisse *} |

Abbildung 4.13: Struktogramm zu E7 (get-dir)

$E_8$

| erg := (fpath-dir (tree-footp tree) part) | {* Untersuchung vom Fußknoten aus *} |
|---|---|
| (get-r-f-dir (list rootnum) erg footnum) gefundenes Ergebnis *} | {* vereinigt Wurzelnummer und oben |

Abbildung 4.14: Struktogramm zu E8 (get-dir)

$E_9$

| erg := (fpath-dir (tree-footp tree) part) | {* Untersuchung vom Fußknoten aus *} |
|---|---|
| (get-r-f-dir (list rootnum) erg footnum) gefundenes Ergebnis *} | {* vereinigt Wurzelnummer und oben |

Abbildung 4.15: Struktogramm zu E9 (get-dir)

**get-r-f-dir**

| Eingabe: place {* Liste bereits erkannter Knoten *} | | |
|---|---|---|
| Eingabe: erg {* Liste aus Richtung T/Knotennummer Präfix *} | | |
| Eingabe: num {* Nummer des Fußknotens *} | | |
| | erg = | |
| (u/d/b "nr" Präfix) | (NIL "fnr" Präfix) | sonst |
| $E_{10}$ | (Präfix place ∪ num) | — |

Abbildung 4.16: Struktogramm zu get-r-f-dir

$E_{10}$

| | (car erg) = | | |
|---|---|---|---|
| u | d | b | sonst |
| (Präfix place ∪ "nr") | (Präfix place ∪ num) | (Präfix place ∪ num) | — |

Abbildung 4.17: Struktogramm zu E10 (get-r-f-dir)

**rpath-dir**

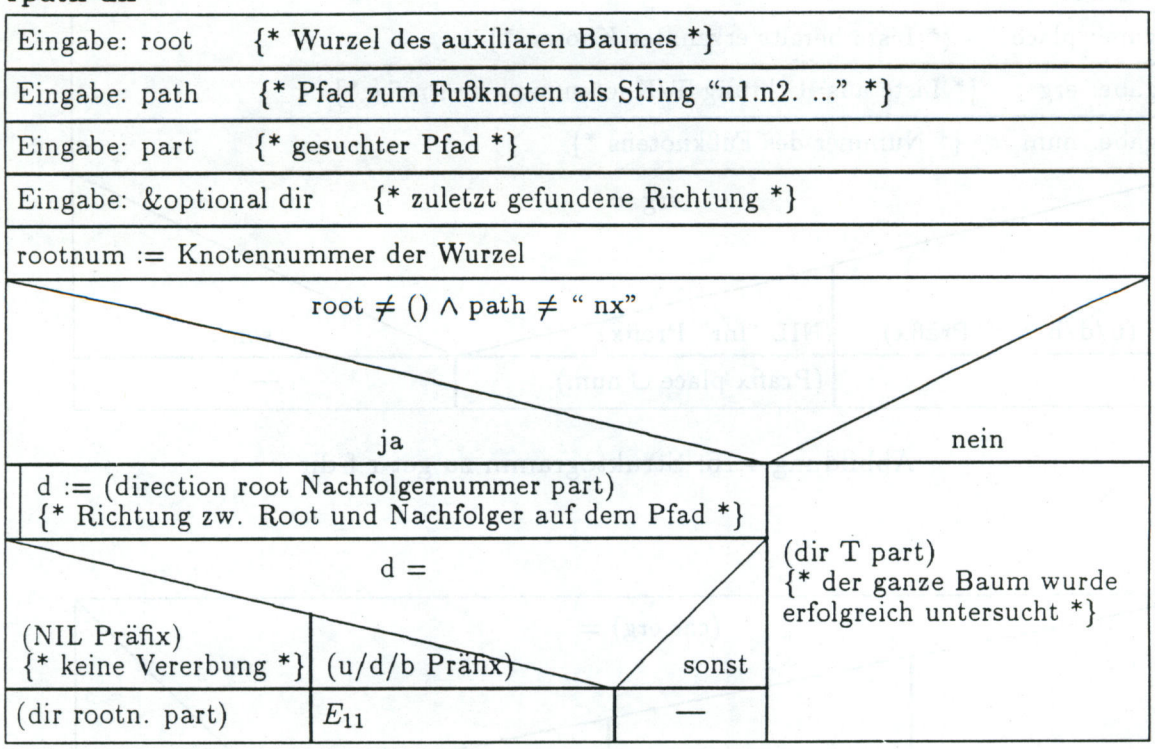



Abbildung 4.18: Struktogramm zu rpath-dir

$E_{11}$

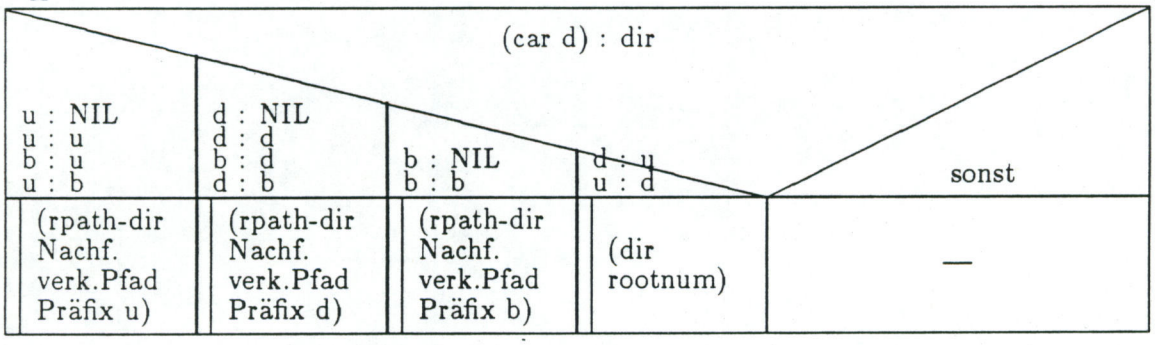



Abbildung 4.19: Struktogramm zu E11 (rpath-dir)

**fpath-dir**

| Eingabe: foot {* Fußknoten des auxiliaren Baumes *} | |
|---|---|
| Eingabe: part {* gesuchter Präfix *} | |
| Eingabe: &optional dir {* zuletzt gefundene Richtung *} | |
| footnum := Nummer des Fußknotens | |
| foot ≠ NIL ∧ Vorgänger ≠ NIL | |
| ja | nein |
| d := (direction Vorg. Bezugsnr. part)<br>{* Richtung zw. Foot und Vorgänger *} | (dir T part)<br>{* ganzer Baum untersucht *} |
| $E_{12}$ | |

Abbildung 4.20: Struktogramm zu fpath-dir

$E_{12}$

| (car d) : dir | | | | |
|---|---|---|---|---|
| u : NIL<br>u : u<br>u : b<br>b : u | d : NIL<br>d : d<br>d : b<br>b : d | b : NIL<br>b : b | d : u<br>u : d | sonst |
| (fpath-dir Vorgänger Präfix u) | (fpath-dir Vorgänger Präfix d) | (fpath-dir Vorgänger Präfix b) | (dir footnum part) | — |

Abbildung 4.21: Struktogramm zu E12 (fpath-dir)

**direction**

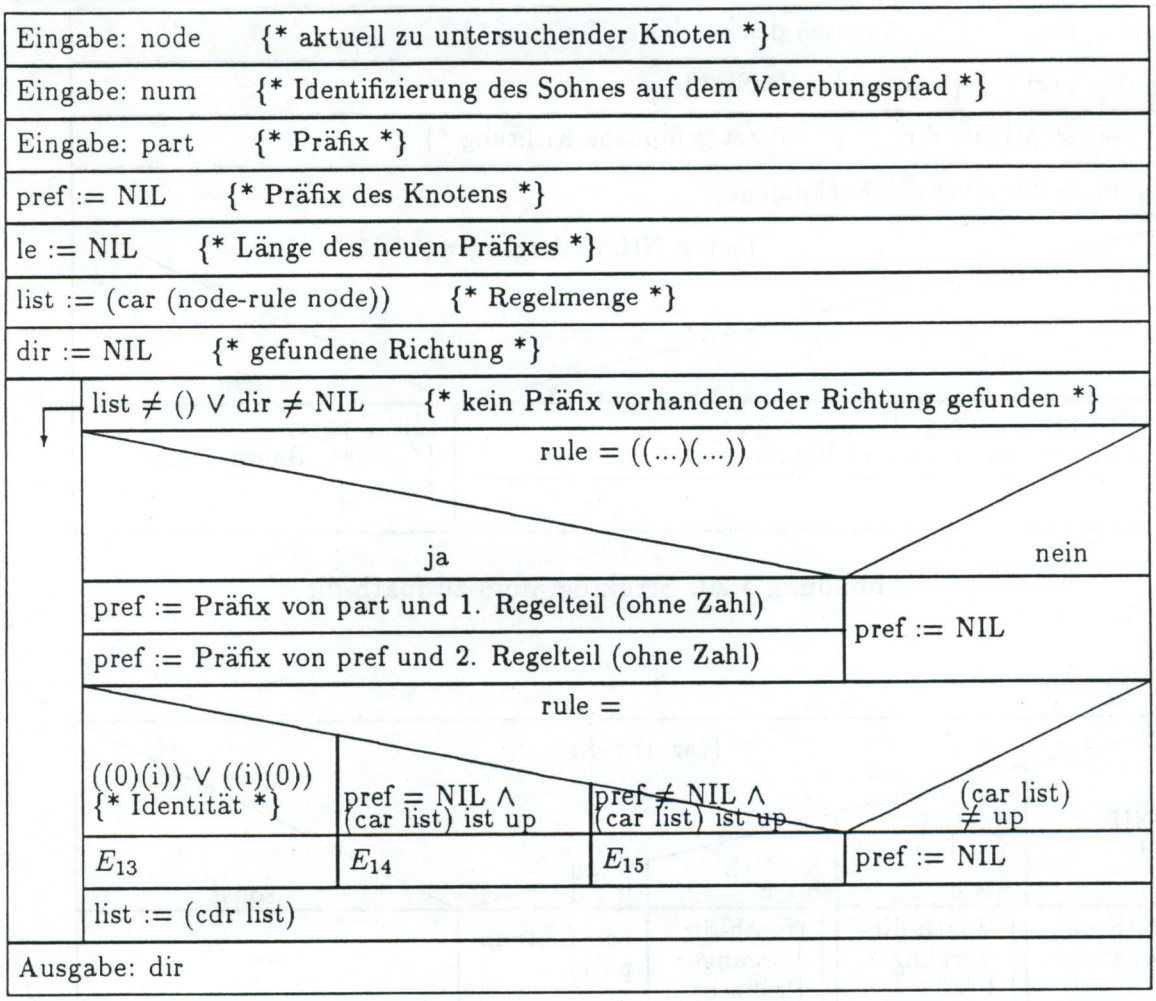



Abbildung 4.22: Struktogramm zu direction

$E_{13}$

rule = ((0)(num)) ∨ ((num)(0)) →
- dir := 'b
- pref := part {* alter Präfix *}

Abbildung 4.23: Struktogramm zu E13 (direction)

$E_{14}$

| rule = | | | |
|---|---|---|---|
| ((num)(0 x)) | ((0 x) (num)) | ((0)(num x)) | ((num x)(0)) |
| pref := Präfix aus part und x | pref := Präfix aus part und x | pref := Präfix aus part und x | pref := Präfix aus part und x |
| dir := 'd | dir := 'd | dir := 'u | dir := 'u |

Abbildung 4.24: Struktogramm zu E14 (direction)

$E_{15}$

| rule = | | | | | |
|---|---|---|---|---|---|
| ((0 x y) (num x)) | ((0 x) (num x y)) | ((0 x) (num x)) | ((num x) (0 x y)) | ((num x y) (0 x)) | ((num x) (0 x)) |
| dir := 'd | dir := 'u | dir := 'b | dir := 'd | dir := 'u | dir := 'b |

Abbildung 4.25: Struktogramm zu E15 (direction)

### Handhabung

Die einzelnen Funktionen dieser Datei sind für den Benutzer nicht zugänglich, die Steuerfunktion *get-def-nodes* wird am Ende der Umwandlung der Grammatik in 2-Form für alle auxiliaren Bäume aufgerufen.

Entstehen beim Umwandeln in eine $\epsilon$-freie oder später eine kettenfreie Grammatik neue auxiliare Bäume, so muß die Vererbungsrichtung neu berechnet werden. Da hier aber "dicke DAGs" an Knoten auftreten können, werden über den Aufruf *fat-get-def-nodes* zunächst in einer Baumkopie alle Knoten mit "dicken DAGs" wieder zu Ketten auseinandergezogen (mit bestimmten erkennbaren Nummern), die Vererbungsrichtung berechnet und an die Wurzel des Baumes geschrieben.

Man muß die Untersuchungen zur Vererbungsrichtung bereits vor der Ausführung der Normalformtransformation gemacht haben, da hier adjungiert ,also auf die Vererbungsrichtung in den gegebenen und neu entstehenden Bäumen Bezug genommen wird.

## 4.2.3 Adjunktion

Wir müssen bereits bei der Implementierung der Normalform-Transformation auf die Adjunktion eingehen, da sie in einzelnen Schritten ausgeführt wird und sich vom Vorgehen bei

der Analyse durch die Verarbeitungsrichtung unterscheidet. Hier werden Adjunktionen ausgeführt, bei der Analyse später müssen Adjunktionen erkannt werden. Die Funktionen zum Ausführen der Adjunktion sind in der Datei "tree-adj.l" zusammengefaßt.

### Erweiterungen der Definition

Durch die Einführung der "dicken Dags" (siehe Kapitel 2.2) entstehen drei neue Adjunktionsfälle, je nachdem, ob Wurzel und/oder Fußknoten und der Anhängeknoten "dünne" oder "dicke DAGs" haben. In Zukunft rede ich zur Abgrenzung statt von DAGs (normalen DAGs, keine Liste) von "dünnen DAGs".

### Datenstrukturen

Bei der Adjunktion eines auxiliaren Baumes in einem Knoten X werden dessen Regeln noch einmal vom auxiliaren Baum aus auf Vater und Söhne angewendet und die Definitionen in die Knoten, die berechnet wurden und jetzt in *node-defs* stehen, übertragen. Dazu wird *node-rule* (bei "dicken Dags" die letzte *node-rule*) von X in Sinneinheiten zerlegt:

- Liste der Definitionen *((0 x) y), ((0 x) (a b))*
- Liste der Tests *((1 x) (2 y))*
- Liste der Downs (↓X) *((0 x) (1 y)), ((0) (1)), ((0) (1 y)), ((0 x) (1))*
- Liste der Ups (↑X) (vom Vater aus für den Sohn i) *((0 x) (i y)), ((0) (i)), ((0) (i y)), ((0 x) (i))*

### Operatoren

**dag-parts:** erzeugt eine Liste aus: einer Liste von Definitionen, einer Liste von Downs, einer Liste von Ups und einer Liste von Tests.

**norm-apply-defs:** schreibt die Values an die Knoten, die durch die Untersuchung der Vererbungsrichtung im Baum bestimmt wurden.

**write-tests:** schreibt die Test-Regeln in den Fußknoten.

**n-d-down:** löscht, compiliert und unifiziert noch einmal alle Dags im Ergebnisbaum. Dieses teure Wiederaufbauen ist notwendig, da wir auf der Basis von PATR programmiert und so keinen direkten Zugriff auf 'Links' und keine Möglichkeit hatten, sie destruktiv zu durchtrennen.

**adjoinfu:** adjungiert den angegebenen auxiliaren Baum in den vorgegebenen Knoten des Anhängebaumes.

**norm-adjoinfu:** steuert die verschiedenen Adjunktionsarten für "dicke DAGs" in "dicke DAGs", "dünne" in "dicke", "dünne" in "dünne" und "dicke" in "dünne DAGs".

### Algorithmus

Die theoretischen Überlegungen zu den verschiedenen Adjunktionsarten wurden bereits beim ersten Auftreten der Idee von den "dicken DAGs" gemacht und sind in Kapitel 3.3.3 nachzulesen. Im folgenden werden die den einzelnen Adjunktionsarten entsprechenden Programmabläufe erklärt.

### 1. Normalfall: Wurzel, Fußknoten und Anhängeknoten mit "dünnen DAGs"

Beim Aufruf von *adjoinfu* wird zuerst eine Kopie der Bäume und des Anhängeknotens erzeugt, um nicht destruktiv zu arbeiten.

Mit Hilfe von *dag-parts* werden die Regeln des Adjunktionsknotens in Teile zerlegt und durch die *apply*-Funktionen auf den auxiliaren Baum übertragen. Die Pointer zwischen dem aktuellen Knoten, seinem Vater und seinen Söhnen werden umgehängt, die Nummern des Baums und seiner Knoten erneuert und die Liste seiner Blätter berechnet (siehe Abbildung 4.26).

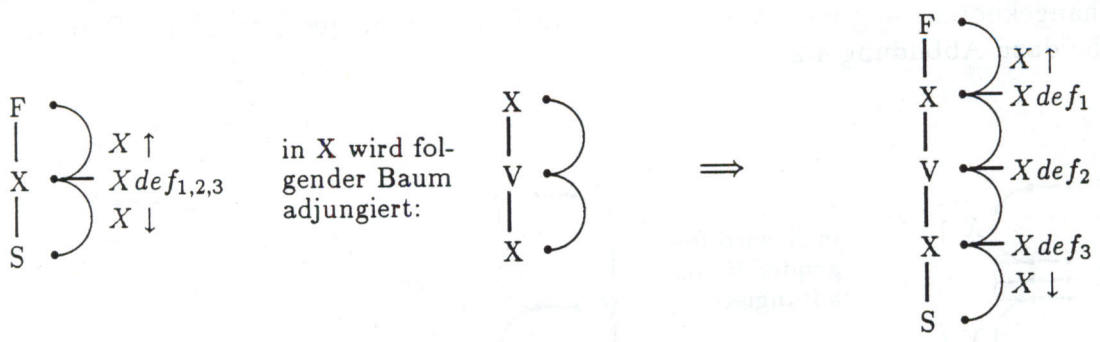



Abbildung 4.26: Normalfall der Adjunktion: nur "dünne DAGs"

### 2. Spezialfall: Wurzel, Fußknoten mit "dünnen DAGs", Anhängeknoten mit "dickem DAG"

Will man in einem Knoten mit dickem DAG einen "normalen" auxiliaren Baum anhängen, so wird der letzte DAG (also der letzte Hilfsknoten zum Anhängeknoten) als virtueller Anhängeknoten betrachtet. Damit bleibt nach der Adjunktion der "dicke DAG" an der Wurzel des ehemaligen auxiliaren Baumes (siehe Abbildung 4.27). Dieses Vorgehen wird durch die Überlegung gerechtfertigt, daß ein Knoten mit "dickem DAG" stets nur höhere Knoten repräsentiert (siehe Kapitel 3.3.3, Kettenfreiheit).

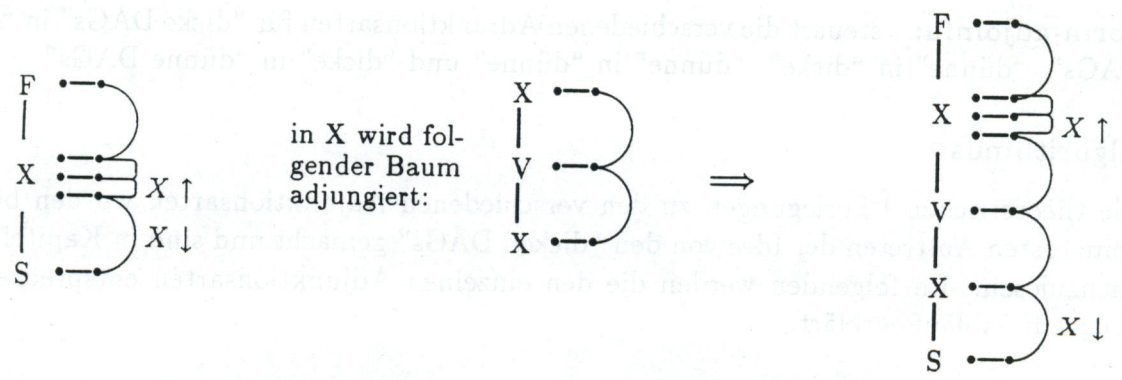



Abbildung 4.27: Spezialfall der Adjunktion: Anhängeknoten mit "dickem DAG"

**3. Spezialfall: Wurzel und/oder Fußknoten und Anhängeknoten mit "dicken DAGs"**

Dieser Fall tritt nur auf, wenn ein neuer auxiliarer K-Baum zur Kettenfreiheit oben angegebener Art in den äquivalenten dicken K-Knoten adjungiert wird. Nach Aufbau des K-Baumes sind alle internen Zusammenhänge in "dicken DAGs" des Anhängeknotens noch einmal im Baum selber repräsentiert. Deshalb fällt bei der Adjunktion der DAG des Anhängeknotens weg und es wird nur zu Vater und Sohn geglättet (Ups, Downs, Tests). Siehe dazu Abbildung 4.28.

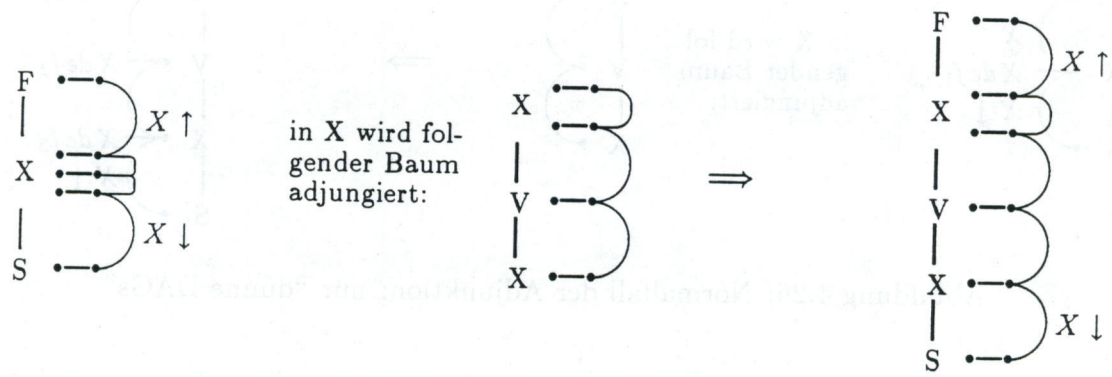



Abbildung 4.28: Spezialfall der Adjunktion: Nur "dicke DAGs"

**4. Spezialfall: Wurzel und/oder Fußknoten mit "dicken DAGs", Anhängeknoten mit "dünnem DAG"**

Liegen Ketten an Wurzel und/oder Fußknoten eines auxiliaren Baumes, so erhält dieser nach der Transformation in einen kettenfreien Baum "dicke DAGs" an Wurzel und/oder Fußknoten, kann aber trotzdem noch in die ursprünglichen Knoten adjungiert werden. Dieser Fall entspricht im großen und ganzen der normalen Adjunktion, wenn man stets den ersten DAG der Wurzel und den letzten des Fußknotens betrachtet (siehe Abbildung 4.29).

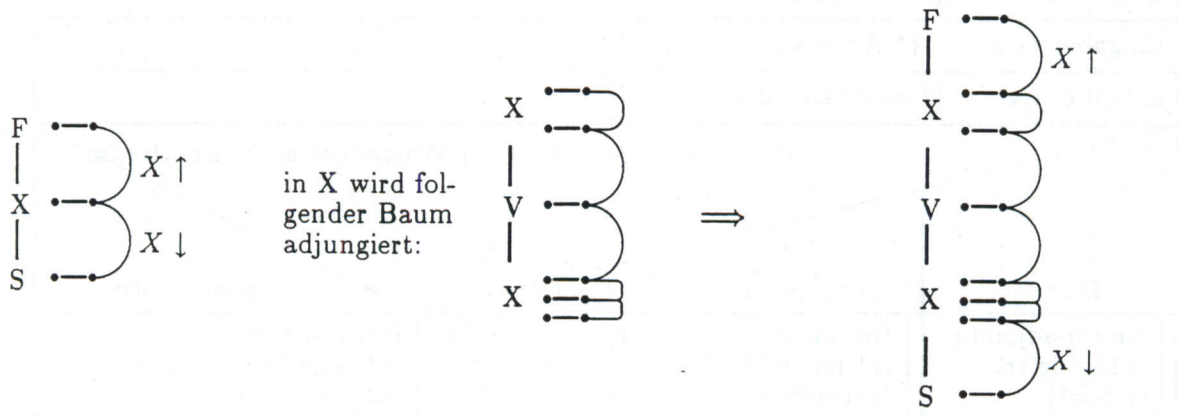



Abbildung 4.29: Spezialfall der Adjunktion: Wurzel und/oder Fußknoten mit "dickem DAG"

**n-d-down**

| | | | |
|---|---|---|---|
| Eingabe: nodelist {* Liste der Wurzel des Ergebnisbaums *} | | | |
| prep := Vorgänger von nodelist | | | |
| i := Identifizierungszahl von (car nodelist) | | | |
| n := 0 | | | |
| hnode := (node-hprep (car nodelist)) | | | |
| n = Länge des DAGs des 1. Knotens | | | |
| untersucht wird | | | |
| DAG in dünnem DAG | Höchster DAG in fat DAG | unterster DAG | innerer DAG |
| lösche, erstelle und compiliere den DAG des 1. Knotens, unifiziere ihn mit dem entspr. DAG-Teil des Vorgängers | lösche, erstelle und compiliere den 1. DAG des 1. Knotens, unifiziere ihn mit dem entspr. DAG-Teil des Vorgängers | lösche, erstelle und compiliere den 1. DAG des 1. Knotens, unifiziere ihn mit dem Vorgänger in fat DAG | lösche, erstelle und compiliere den 1. DAG des 1. Knotens, unifiziere ihn mit dem Vorgänger in fat DAG |
| (n-d-down (Nachfolger des 1. Knotens) ∪ Rest | | | |

Abbildung 4.30: Struktogramm zu n-d-down

**adjoinfu**

| Eingabe: tr1 {* Adjunktionsbaum *} | | | |
|---|---|---|---|
| Eingabe: node {* Adjunktionsbaum *} | | | |
| Eingabe: tr2 {* auxiliarer Baum *} | | | |
| DAG von node : DAG der Wurzel des auxiliaren Baums | | | |
| fat : fat | norm : fat | fat : norm | norm : norm |
| (norm-adjoinfu tr1 node tr2 'fat-fat) | (norm-adjoinfu tr1 node tr2 'norm-fat) | (norm-adjoinfu tr1 node tr2 'fat-norm) | (norm-adjoinfu tr1 node tr2 'norm-norm) |

Abbildung 4.31: Struktogramm zu adjoinfu

**norm-adjoinfu**

| |
|---|
| Eingabe: tr1 {* Adjunktionsbaum *} |
| Eingabe: node {* Adjunktionsknoten *} |
| Eingabe: tr2 {* auxiliarer Baum *} |
| Eingabe: par {* Code für die Art der Adjunktion *} |
| tree1 := Kopie von tr1 |
| tree2 := Kopie von tr2 |
| node := Adresse des äquivalenten Knotens zu tr1 in tree1 |
| erg := (dag-parts (car (node-rule node)))<br>{* Liste aus defs, downs, ups aus der letzten node-rule von node *} |
| defs := (car erg) |
| defparts := (list-parts defs) {* ('+) falls ((0 x) (0 y)), ('-) falls ((0 x) y) *} |
| downs := (cadr erg) |
| tests := (caddr erg) |
| error := NIL |
| $E_1$ {* Definitionen werden gesetzt *} |
| $E_2$ {* Nachfolger des Fußknotens werden gesetzt und verzeigert *} |
| (write-tests tests tree2) {* Tests werden an Foot geschrieben *} |
| $E_3$ {* Vorgänger von Root werden gesetzt und gelinkt *} |
| (n-d-down (list (tree-rootp tree2))) {* neue vollst. Regel-Compilierung *} |
| Blätter und Nummern werden korrigiert |
| error = NIL<br>ja nein |
| Ausgabe: tree1 \| Ausgabe: NIL |

Abbildung 4.32: Struktogramm zu norm-adjoinfu

### Handhabung

Vor dem Aufruf von *adjoinfu* müssen die DAGs der Bäume compiliert und verzeigert und die Vererbungsrichtung in den auxiliaren Bäumen untersucht sein. Der Aufruf erfolgt als

*(adjoinfu tree1 node tree2)*

wobei *tree1* der Anhängebaum, *node* der Anhängeknoten und *tree2* der auxiliare Baum sein sollen. Die Funktion testet nicht, ob die Adjunktion tatsächlich erlaubt ist, allerdings ist sie nicht destruktiv sondern arbeitet auf Kopien der Bäume. Der Benutzer muß im Vorfeld (durch Vergleich der Labels und der Constraints) selber abklären, ob adjungiert werden darf oder nicht.

Fehlermeldungen können beim Unifizieren der Dags auftreten. Während des Ablaufs der Normalform-Transformation werden diese Meldungen unterdrückt, für den Einzeltest mit der Adjunktionsfunktion können sie sichtbar gemacht werden, indem man beim Aufruf von *adjoinfu* als letzten (optionalen) Parameter T angibt. Bei einer fehlgeschlagenen Adjunktion gibt dann die Funktion "NIL" zurück, sowie eine Fehlermeldung, aufgrund derer man Rückschlüsse auf das Entstehen des Fehlschlages ziehen kann.

**'eq-bug:** Fehlschlag bei der Anwendung der Equals

**'bug2:** Fehlschlag bei der Anwendung der Downs

**'bug3:** Fehlschlag bei der Anwendung der Werte-Definitionen

## 4.2.4 Transformation in 2-Form

Die Funktionen zur Transformation einer Grammatik in 2-Form sind in der Datei "two-form-tr.l" zusammengefaßt.

### Erweiterungen der Definition

Erweiterungen der Definition waren zur Realisierung der 2-Form-Transformation nicht notwendig.

### Datenstrukturen

Werden im Verlauf der Umwandlung neue Vaterknoten eingeführt, so ist deren Slot *dag* noch unbenutzt und dient vorübergehend als Markierung dafür, daß sie neu sind, und zur Angabe der Nummerngrenzen (Integers) der Söhne des Ausgangsbaums, deren Vorfahren sie sind.

*('new anf.num endnum)*

### Operatoren

**grammar-to-2-form:** wandelt die Bäume einer Grammatik in 2-Form um.

**tree-to-2-form:** wandelt einen Baum in 2-Form um und compiliert seine *node-dags*.

**node-to-2-form:** erzeugt neue Knoten zwischen Root und seinen Söhnen, falls deren Anzahl die Zahl 2 übersteigt.

**change-dags:** erzeugt Vererbungsregeln zwischen neu geschaffenen Knoten und deren Söhnen und korrigiert die Regeln des alten Vaters mit ehemals mehr als zwei Söhnen.

**dag1-new-node:** erzeugt Regeln für einen neuen Vater zweier alter Söhne.

**dag2-new-node:** erzeugt Regeln für einen neuen Vater neuer Väter.

**dag-old-node:** korrigiert die Regeln des alten Vaters.

### Algorithmus

Die Theorie zur 2-Form-Transformation sind in Kapitel 3.3.1 nachzulesen, an dieser Stelle soll nur die Umsetzung auf LISP-Prozeduren erläutert werden.

Für jeden Baum der Grammatik (gesteuert durch *grammar-to-2-form*) wird in der Funktion *tree-to-2-form* folgendes ausgeführt:

1. Jeder Knoten mit mehr als zwei Söhnen erhält von links nach rechts neue Zwischenknoten, die die Grenzen der Söhne tragen (*node-to-2-form*). Siehe dazu Abbildung 4.33.

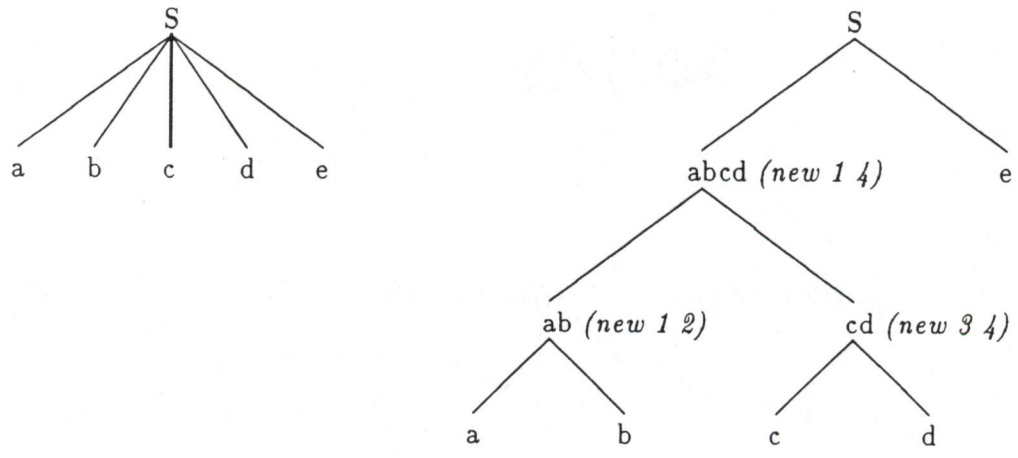



Abbildung 4.33: 1. Schritt der Transformation in Zweiform

2. Für jede Regel *((0 x) (i y))* des alten Vaters wird der Präfix *(0 x)* über Vererbungsregeln *((0 x) (1 x))* oder *((0 x) (2 x))* auf dem entsprechenden Pfad im Baum "heruntergereicht" und erst im letzten neuen Knoten *((0 x) (1/2 y) )* angewendet.

Für jede Regel *((i x) (j y))* des alten Vaters werden die Präfixe getrennt auf den entsprechenden Pfaden als *((0 x) (1/2 x))* und *((0 y) (1/2 y))* "heruntergereicht". Die Stelle der Unifikation bleibt beim alten Vater: *((1/2 x) (1/2 y))*. Dieser Schritt wird im Beispiel in Abbildung 4.34 ausgeführt.

Das Umschreiben der Regeln auf diese Art und Weise erscheint intuitiv richtig, da die einzige Veränderung eine im Baum vorgezogene, das heißt in tieferen Knoten stattfindende Unifikation zu sein scheint. Da aber die entsprechenden Pfade sowieso im alten Vater zusammengeführt worden wären und außerdem in den neuen Knoten nicht adjungiert werden darf, ist diese Regeländerung angebracht. Sie wurde einer anderen, einfacheren Alternative deshalb vorgezogen, weil hier die Präfixe im Baum und damit das Ergebnis der Untersuchung der Vererbungsrichtungen erhalten bleiben. Einen Beweis für die Korrektheit der Umformungen haben wir an dieser Stelle nicht geführt.

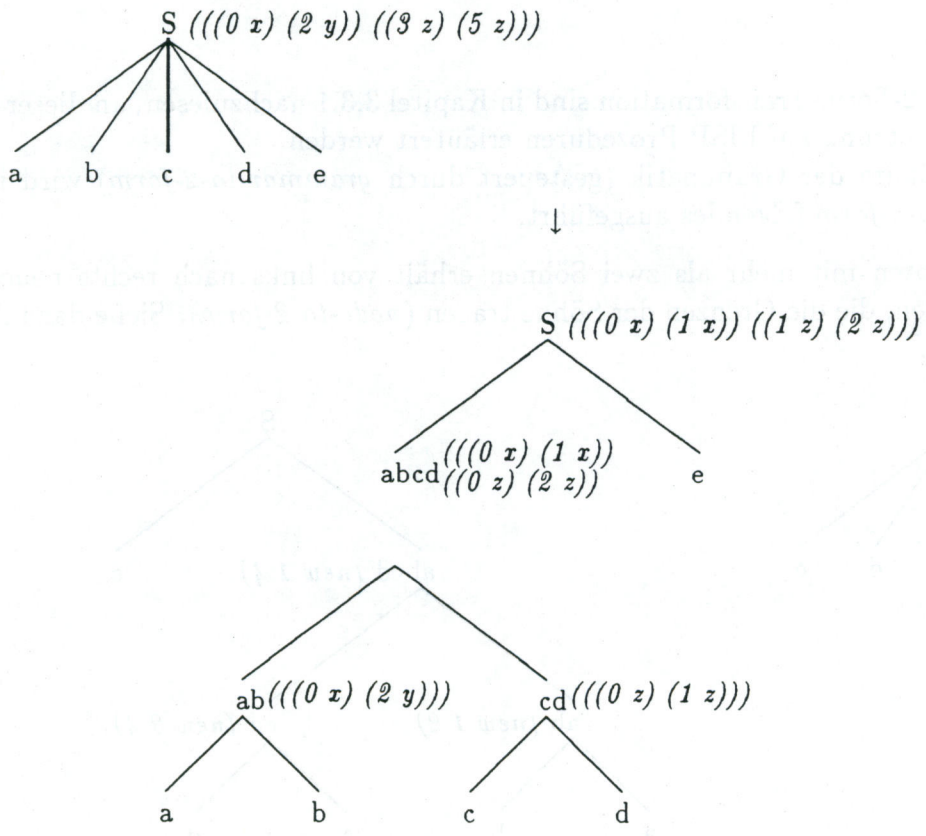



Abbildung 4.34: 2. Schritt der Transformation in Zweiform

3. Die Nummern der Knoten werden neu gesetzt.

4. Die Angaben im Slot *node-dag* der neuen Knoten *(new n1 n2)* werden gelöscht.

5. Die DAGs werden compiliert und gelinkt.

**grammar-to-2-form**

| |
|---|
| für jeden tree aus chtreelist von grammar |
| (tree-to-2-form Wurzel von tree) {* wandelt Baum in 2-Form um *} |
| für jeden tree aus chtreelist von grammar |
| tree ist auxiliarer Baum<br>→ (get-def-nodes (tree)) {* berechnet Vererbungsrichtungen im Baum *} |

Abbildung 4.35: Struktogramm zu grammar-to-2-form

**tree-to-2-form**

| | |
|---|---|
| Eingabe: root {* Wurzel eines TAG-Baumes *} | |
| root hat Nachfolger | |
| ja | nein |
| root hat mehr als 2 Nachfolger<br>→ root := (node-to-2-form root) {* Einführen von Zwischenkn. *} | — |
| für jeden son in der Nachfolgerliste von root | |
| (tree-to-2-form son)<br>{* Baum mit "Zwischenwurzel" son<br>wird in 2-Form umgewandelt *} | |
| root ist die Wurzel | |
| ja | nein |
| (change-dags (list root)) {* erzeugt Identitätsregeln zw.<br>Hilfsknoten und Söhnen und korrigiert die Zugriffe *} | — |
| (change-numbers Wurzel Wurzelnummer)<br>{* erneuert die Baumnummerierung *} | |
| (del-new-dag-part (list root))<br>{* entfernt die Hilfsnotierungen (new n1 n2) *} | |
| (set-tree-links (list root)) {* ergänzt Unifikations-<br>regel im Slot 'Dag' durch Komponentennamen der KF-<br>Regel, compiliert und linkt die Dags *} | |
| Ausgabe: root | |

Abbildung 4.36: Struktogramm zu tree-to-2-form

**node-to-2-form**

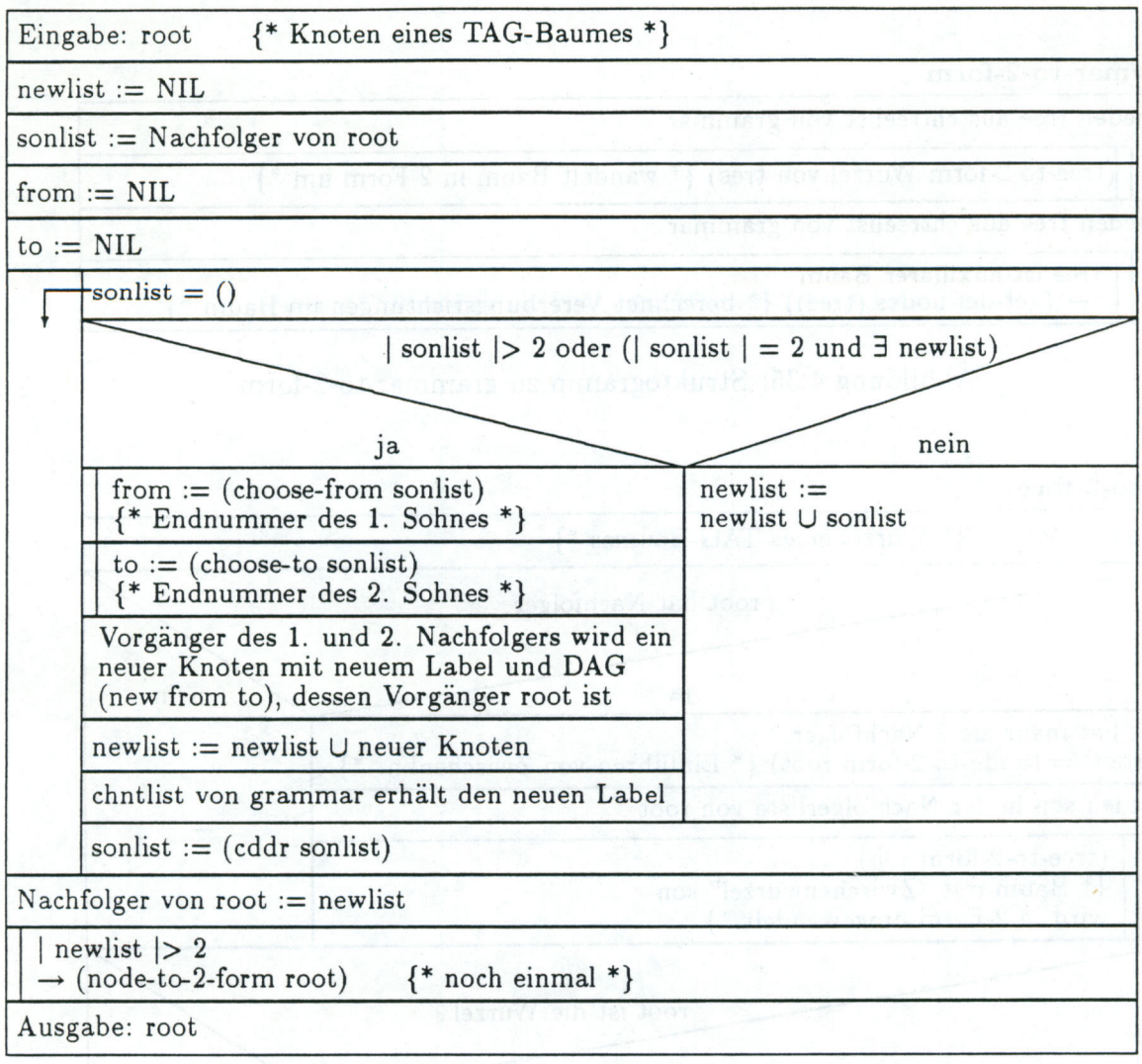



Abbildung 4.37: Struktogramm zu node-to-2-form

**change-dags**

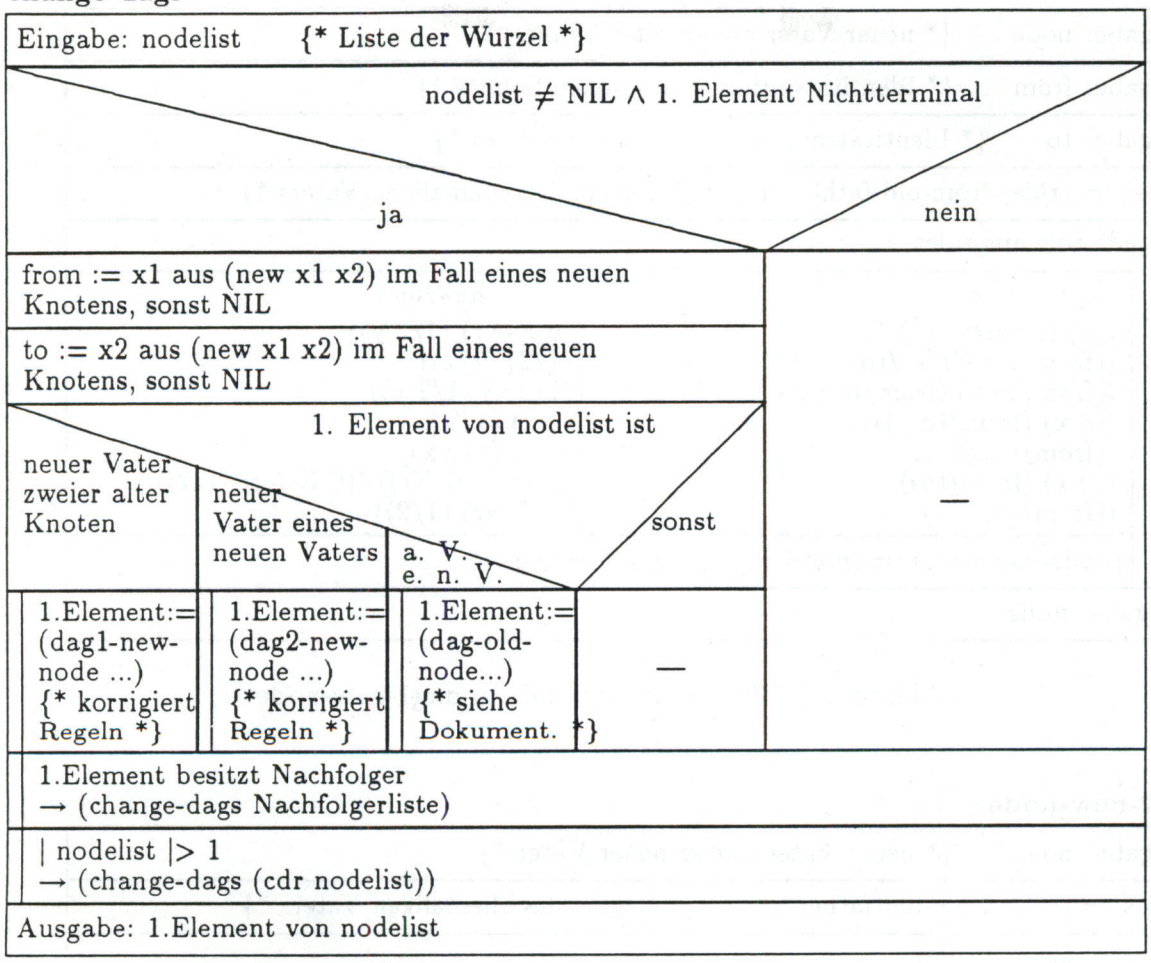



Abbildung 4.38: Struktogramm zu change-dags

**dag1-new-node**

Eingabe: node {* neuer Vater zweier alter Söhne *}

Eingabe: from {* Identitätsnummer des linken Sohnes *}

Eingabe: to {* Identitätsnummer des rechten Sohnes *}

rules := (rules-from-old-father...) {* Regeln des ehemaligen Vaters *}

für jede rule aus rules

| rule | → | newrule) |
|---|---|---|
| ((0) (from/to y)) | | ((0 K-Nr) (1/2 y)) |
| ((from/to) (from/to)) | | ((1/2) (1/2)) |
| ((from/to x) (from/to y)) | | ((1/2 x) (1/2 y)) |
| ((0 x) (from/to y)) | | ((0 x) (1/2 y)) |
| ((from/to x) ...) | | ((0 x) (1/2 x)) |
| ((0 x) (from/to)) | | ((0 x) (0 K-Nr)) ((0 K-Nr) (1/2)) |
| ((from/to) ...) | | ((0 K-Nr) (1/2)) |

(node-dag node) := (node-dag node) ∪ newrule

Ausgabe: node

Abbildung 4.39: Struktogramm zu dag1-new-node

**dag2-new-node**

Eingabe: node {* neuer Vater zweier neuer Väter *}

rules := (rules-from-old-father...) {* Regeln des ehemaligen Vaters *}

für jede rule aus rules

| rule | → | newrule |
|---|---|---|
| ((0 x) (Nachf y)) | | ((0 x) (1/2 y)) |
| ((i y) (Nachf z)) | | ((0 z) (1/2 z)) |
| ((0 x) (Nachf)) | | ((0 x) (1/2 x)) |
| ((0) (Nachf x)) | | ((0 K-Nr) (1/2 K-Nr)) |
| ((Nachf) (0)) | | ((0 K-Nr) (1/2 K-Nr)) |
| ((Nachf) (i)) | | ((0 K-Nr) (1/2 K-Nr)) |
| ((Nachf1 x) (Nachf2 y)) | | ((1/2 x) (1/2 y)) |
| ((Nachf1) (Nachf2 y)) | | ((1/2 K-Nr) (1/2 y)) |
| ((Nachf1) (Nachf2)) | | ((1/2 K-Nr) (1/2 K-Nr)) |

(node-dag node) := (node-dag node) ∪ newrule

Ausgabe: node

Abbildung 4.40: Struktogramm zu dag2-new-node

**dag-old-node**

Eingabe: node {* alter Vaterknoten *}

(node-defs node) := (node-dag node) {* Zwischenspeicherung *}

i := 0

rule := 0-te Regel aus (node-dag node)

newrule := NIL {* neue Regel für node *}

i = Länge von (node-dag node)

Nachf steht für neue, NACHF für alte Sohn-Kanten

| rule | → | newrule |
|---|---|---|
| ((0 x) (Nachf y)) | | ((0 x) (1/2 x)) |
| ((0 x) (Nachf)) | | ((0 x) (1/2 x)) |
| ((0) (Nachf x)) | | ((0) (1/2 K-Nr)) |
| ((0 x) (Nachf y)) | | ((0 x) (1/2 y)) |
| ((Nachf1 x) (Nachf2 y)) | | ((1/2 x) (1/2 y)) |
| ((NACHF1 x) (NACHF2 y)) | | ((1/2 x) (1/2 y)) |
| ((NACHF1 x) (Nachf2 y)) | | ((1/2 x) (1/2 y)) |
| ((Nachf1 x) (NACHF2 y)) | | ((1/2 x) (1/2 y)) |
| ((0 x) (NACH)), ((NACHF) (0 x)) | | ((0 x) (1/2)) |
| ((NACHF1) (NACHF2 x)) | | ((1/2) (1/2 x)) |
| ((Nachf1 x) (NACHF2)) | | ((1/2 x) (1/2)) |
| ((NACHF1) (NACHF2)) | | ((1/2) (1/2)) |
| ((0) (Nachf)) | | ((0) (1/2 K-Nr)) |
| ((Nachf1) (Nachf2 x)), ... | | ((1/2 K-Nr) (1/2 x)) |
| ((Nachf1) (Nachf2)) | | ((1/2 K-Nr) (1/2 K-Nr)) |
| ((NACHF1) (Nachf2)), ... | | ((1/2) (1/2 K-Nr)) |

i-te Regel aus (node-dag node) := newrule

i := i+1

rule := i-te Regel aus (node-dag node)

newrule := NIL

Ausgabe: node

Abbildung 4.41: Struktogramm zu dag-old-node

### Handhabung

Beim Aufruf von *grammar-to-2-form* muß die Grammatik in der Variable *grammar* die ungeänderte Eingabeform besitzen, DAGs sollen noch nicht als Listen vorliegen und ebensowenig compiliert und unifiziert sein.

Das Ergebnis der Umformung wird unter dem Slot *chtreelist* der Variable *grammar* abgelegt. Jetzt sind Regeln und DAGs als Listen mit einem Element (für spätere dicke DAGs) angelegt, die DAGs sind compiliert, die DAG-Kanten im Baum gezogen und die Vererbungsrichtungen für die auxiliaren Bäume berechnet.

Fehlermeldungen können beim Aufruf von *compile-patr-rule* und *unify* auftreten und sind in den jeweils verwendeten Dateien nachzulesen.

## 4.2.5 Epsilon-Freiheit

Die Funktionen zur Transformation einer Grammatik in eine epsilonfreie Grammatik sind in der Datei "eps-free.l" zusammengefaßt.

### Erweiterungen der Definition

Erweiterungen der Definition waren zur Realisierung der Transformation nicht notwendig.

### Datenstrukturen

Die Variable $F$ wird mit der Menge aller Bäume belegt, die durch das Abschneiden der $\epsilon$-Blätter falsche Fußknoten erhalten haben.

Die Variable $X$ wird mit der Menge der auxiliaren Bäume belegt, die durch das Abtrennen der korrekten Baumteile in $F$ entsteht.

In *SA-new* wird zu einem falschen auxiliaren Baum (auxiliarer Baum mit falschem Fußknoten) die Menge aller korrigierten (abgeschnittenen und adjungierten) auxiliaren Bäume gesammelt.

In *G-new* werden die korrigierten Bäume abgelegt, um später der Grammatik angehängt zu werden.

### Operatoren

**grammar-to-e-free:** ruft für jeden Baum der Grammatik die Umwandlungsfunktionen auf und initialisiert die lokalen Strukturen.

**new-e-tree:** erzeugt, falls noch nicht vorhanden, einen neuen Baum S — $\epsilon$, um das leere Wort in der Sprache zu behalten.

**one-ob-adj:** erzeugt eine Menge neuer Bäume, die aus dem alten Baum durch jeweils eine Adjunktion an jedem möglichen Knoten mit jedem möglichen auxiliaren Baum hervorgeht.

**new-aux-trees:** steuert die drei folgenden Funktionen.

**delete-e-nodes:** schneidet alle $\epsilon$-Blätter der Knoten der Liste ab.

**false-foot-trees:** sammelt alle Bäume mit falschen Fußknoten.

**new-aux:** trennt die Bäume in $F$ in korrekten Teil und neuen auxiliaren Baum, speichert diesen als selektive Adjunktion in entsprechenden Knoten des korrekten Teils, schreibt die korrekten Bäume in die Grammatik, die falschen in $X$, setzt neue Nummern und macht die Grammatik kettenfrei.

**adj-new-aux-trees:** steuert den Algorithmus zur Korrektur neuer auxiliarer Bäume.

**X-new:** ruft die Cut- und Adjoinfunktionen für alle Bäume in $X$ auf und schreibt die Ergebnisse nach $F$.

**adj-and-cut:** erzeugt die Menge aller Bäume, die aus einem auxiliaren Baum durch iteratives Abschneiden des Pfades vom ersten falschen Fußknoten bis zum ersten korrekten Vaterknoten, Adjunktion im jeweils letzten Knoten und Abschneiden des neuen Fußknotens entsteht.

**F-new:** schreibt Bäume, die immer noch einen falschen Fußknoten haben, aus $X$ nach $F$, richtige nach *G-new*.

**correct-trees:** korrigiert die Bäume der Grammatik.

### Algorithmus

Die Umformung einer Grammatik in eine epsilon-freie Grammatik ist der zweite Schritt zur Umformung in unsere Normalform, die theoretischen Betrachtungen zu dieser Umformung wurden bereits in Kapitel 3.3 dargestellt. Eine grobe Gliederung der Umformung beinhaltet die Schritte:

1. Reine $\epsilon$-Bäume werden entfernt.
2. Alle $\epsilon$-Blätter werden entfernt (*new-aux-trees*).
3. Korrektur der neuen auxiliaren Bäume (*adj-new-aux-trees*) und der ganzen Grammatik.

Anstelle aller Bäume mit leerem Blattwort erhält die Grammatik einen Baum S — $\epsilon$, bei dem im Knoten S nicht adjungiert werden darf (*new-e-tree*). Die Bäume

mit leerem Blattwort ersetzt man durch neue Mengen von Bäumen, die aus ihnen durch Adjunktion jedes möglichen auxiliaren Baumes an jedem möglichen Knoten hervorgeht.

Die Grammatik besitzt jetzt (außer dem Baum S — $\epsilon$) nur Bäume mit mindestens einem Blatt ungleich $\epsilon$, da jeder auxiliare Baum ein solches besitzen muß. Nun muß folgendes geschehen:

1. Von allen Bäumen werden die $\epsilon$-Blätter abgeschnitten (*delete-e-nodes*).
2. Dabei entstehende Bäume mit falschen Fußknoten werden in der Menge $F$ gesammelt (*false-foot-trees*).
3. Die falschen Fußknoten werden im Baum nach oben verfolgt, bis ein Knoten gefunden wird, der Vater eines korrekten Blattes ist. Dort wird der Baum durchtrennt, der korrekte Teil in die Grammatik zurückgeschrieben, die *constraints* seiner entsprechenden Knoten auf selektive Adjunktion gesetzt und mit dem (den) neuen auxiliaren Baum (Bäumen) in $X$ verzeigert (siehe Abbildung 4.42).

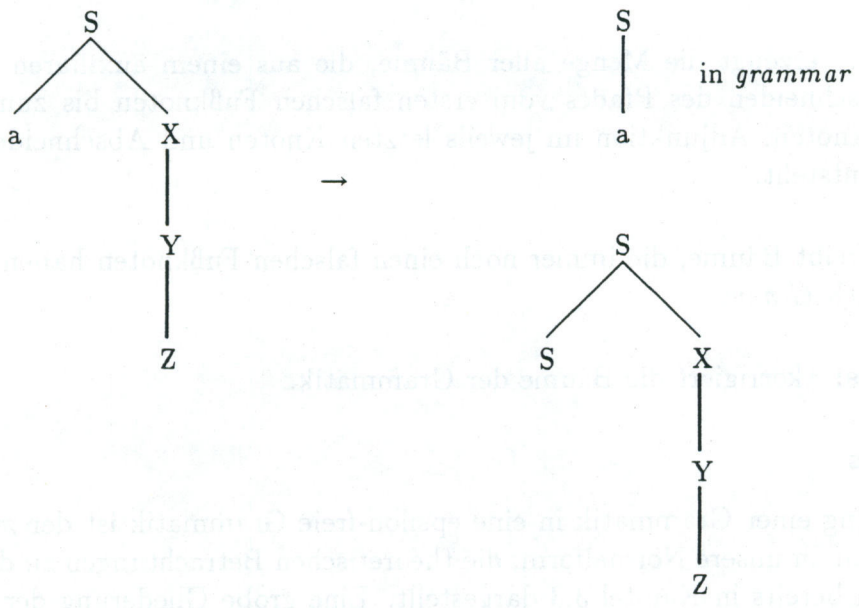



Abbildung 4.42: Behandlung "falscher Fußknoten" bei der Epsilonfreiheit

4. Die Bäume in *grammar* und $X$ werden bezüglich ihrer Nummern und Blattlisten korrigiert.
5. Die Bäume in *grammar* werden kettenfrei gemacht.

Die Korrektur der neuen auxiliaren Bäume läuft folgendermaßen ab:

1. Die in $X$ gesammelten auxiliaren Bäume, die bereits korrekte Struktur haben, werden in *G-new* geschrieben, die anderen in die Menge $F$.

2. Solange $F$ noch Bäume enhält:
   Tue für alle Bäume von $F$:

   - Suche den ersten falschen Fußknoten.
   - Verfolge ihn nach oben bis zum korrekten Vaterknoten und mache dabei folgendes:
     - schneide den letzten Knoten ab (zuerst NIL)
     - adjungiere im jetzt letzten Knoten alle möglichen auxiliaren Bäume
     - schneide den auxiliaren Fußknoten ab

     Siehe dazu auch Abbildung 4.43. Die so entstehende Menge (teilweise) korrekter auxiliarer Bäume wird wiederum per Pointer im *node-constraint* des Wurzelknotens des alten auxiliaren Baums hinter dem Code-Symbol "Z" abgelegt. Sie stehen in $X$, werden erneut in korrekte und falsche Bäume aufgeteilt und in *G-new* oder $F$ gespeichert.

3. Die Bäume, die als selektive Adjunktion in Knoten der Grammatik gespeichert sind, werden durch ihre Nummern ersetzt, nachdem sie bis zur endgültig korrekten Menge über das Code-Symbol "Z" verfolgt wurden.

   Nummern und Blattlisten der Bäume werden korrigiert.

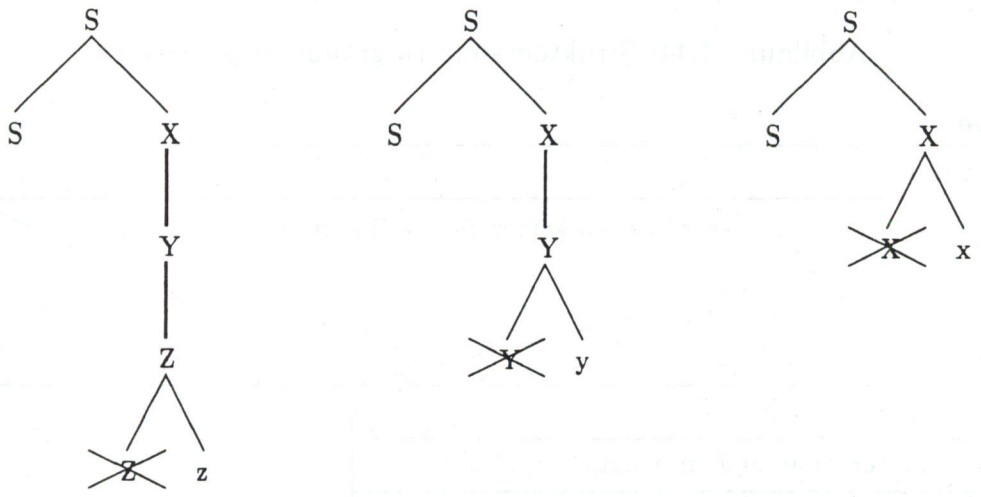



Abbildung 4.43: Alternative neue auxiliare Bäume bei der Epsilonfreiheit

**grammar-to-e-free**

| | | |
|---|---|---|
| F := NIL {* Initialisierung *} | | |
| X := NIL {* Initialisierung *} | | |
| G-new := NIL {* Initialisierung *} | | |
| SA-new := NIL {* Initialisierung *} | | |
| für i von 0 bis zur Länge der Grammatik | | |
| | tree $\epsilon$-Baum aber $\neq$ S — $\epsilon$ | |
| | ja | nein |
| | (new-e-tree (root)) {* S — $\epsilon$ dazu *} | — |
| | (one-ob-adj...) {* füge alle Bäume, die durch 1 Adjunktion an tree entstehen, zur Grammatik *} | |
| | lösche tree | |
| new-aux-trees {* schneidet die $\epsilon$-Blätter ab *} | | |
| (adj-new-aux-trees) {* korrigiert neu entstandene auxiliare Bäume *} | | |
| (correct-trees) {* korrigiert die Grammatik *} | | |

Abbildung 4.44: Struktogramm zu grammar-to-e-free

**new-e-tree**

| | |
|---|---|
| Eingabe: root | |
| es gibt noch keinen S — $\epsilon$-Baum in chtreelist | |
| ja | nein |
| num := neue Baumnummer | Ausgabe: chtreelist |
| footnode := Knoten "num.0.0" mit Label "eps" | |
| rootnode := Knoten "num.0" mit Label von root und constraints (SA) | |
| rootnode und footnode werden verpointert | |
| der Baum mit Wurzel rootnode und Fußknoten footnode wird an die Grammatik gehängt | |
| Ausgabe: neue chtreelist | |

Abbildung 4.45: Struktogramm zu new-e-tree

**one-ob-adj**

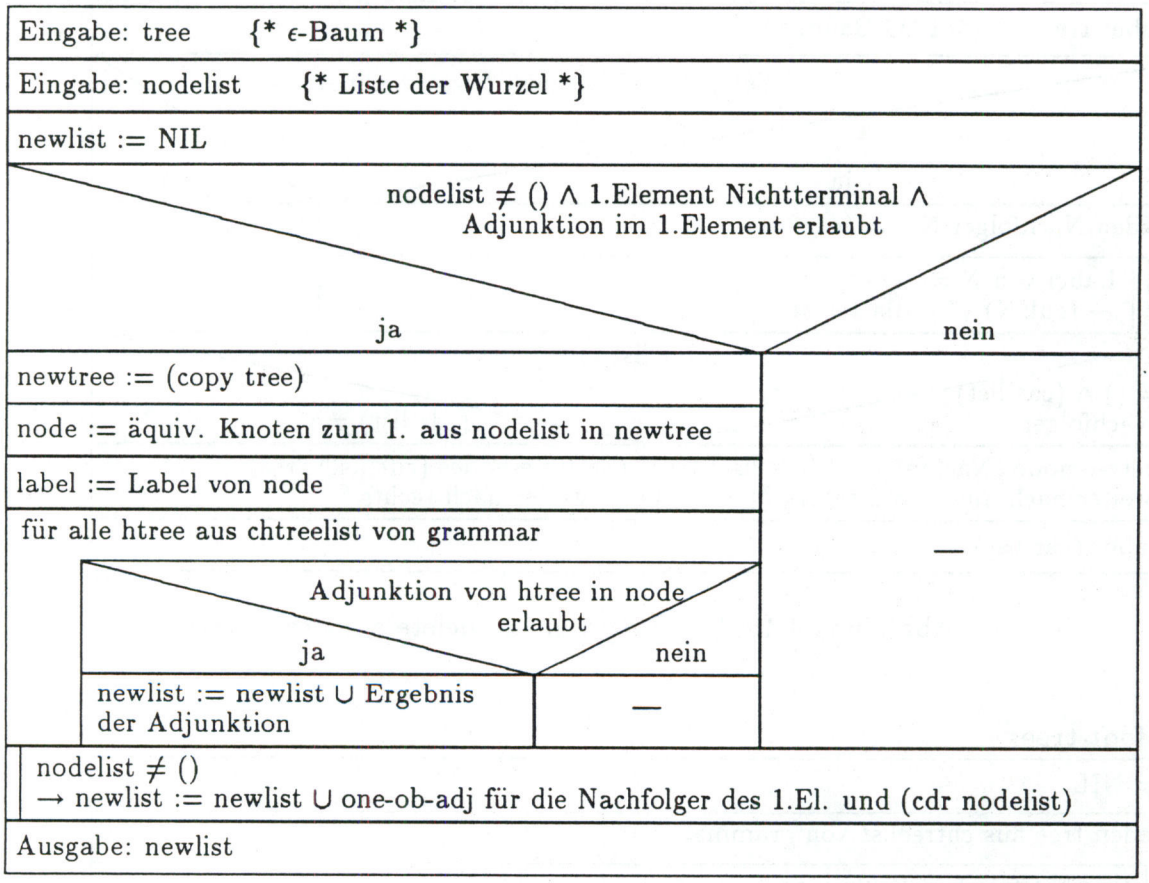



Abbildung 4.46: Struktogramm zu one-ob-adj

**new-aux-trees**

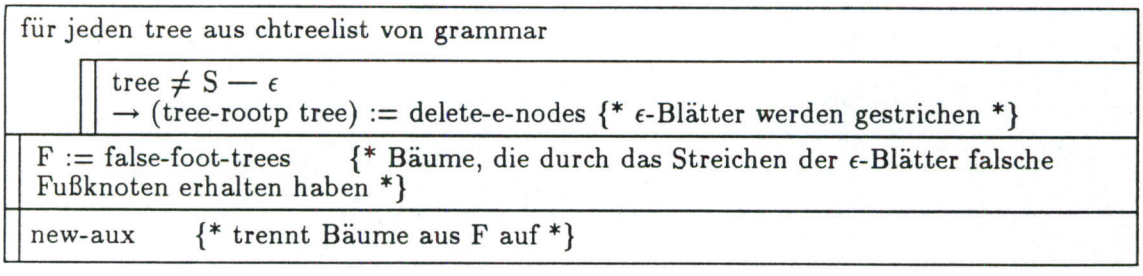



Abbildung 4.47: Struktogramm zu new-aux-trees

**delete-e-nodes**

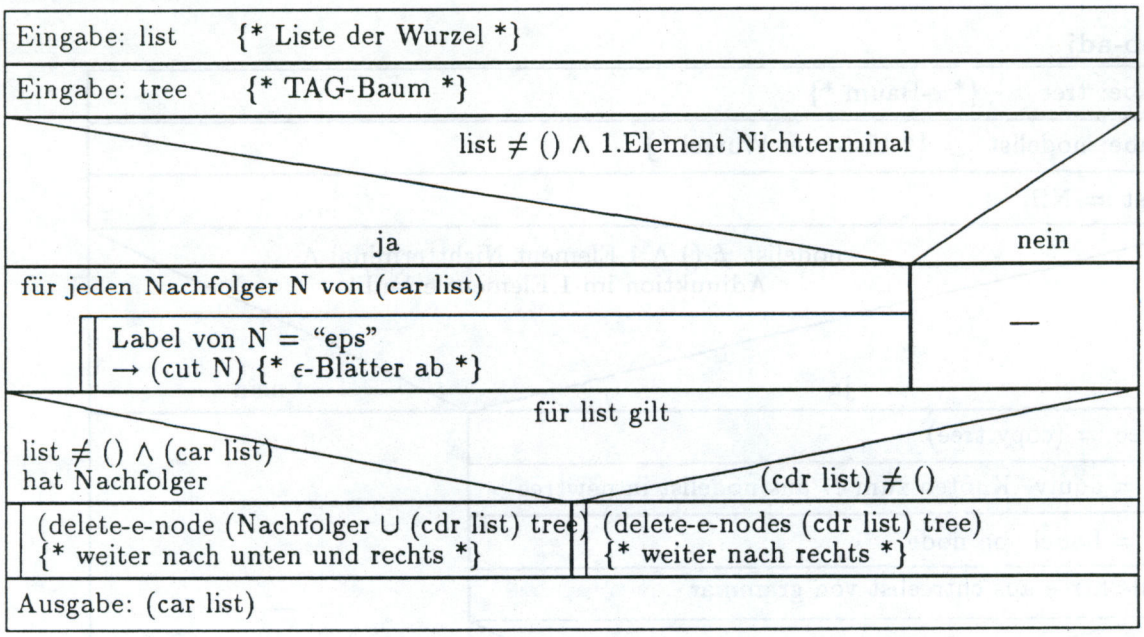



Abbildung 4.48: Struktogramm zu delete-e-nodes

**false-foot-trees**

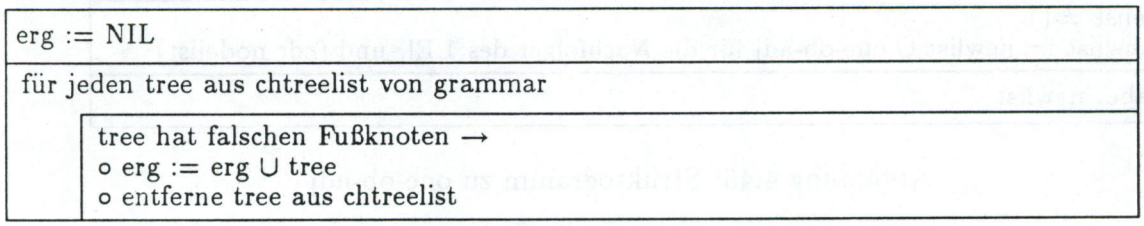



Abbildung 4.49: Struktogramm zu false-foot-trees

**new-aux**

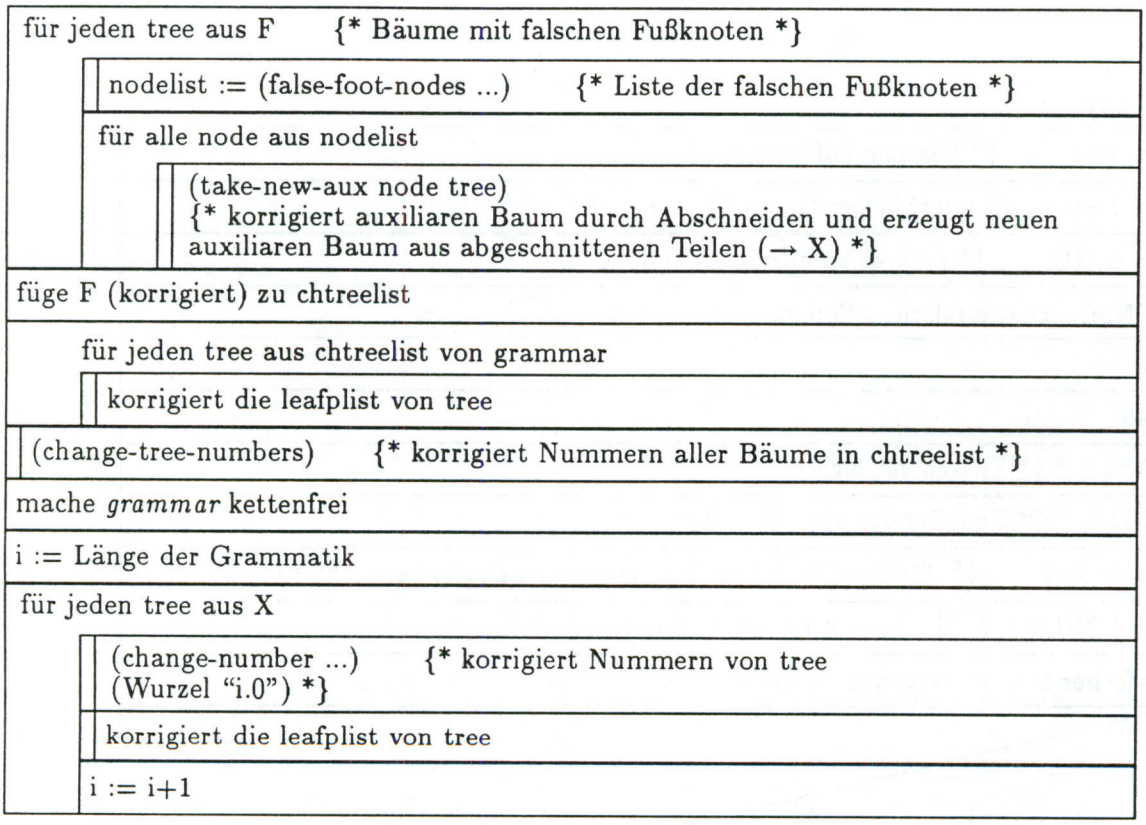



Abbildung 4.50: Struktogramm zu new-aux

**adj-new-aux-trees**

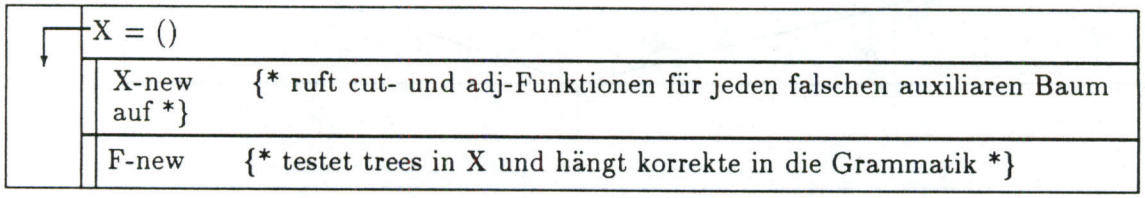



Abbildung 4.51: Struktogramm zu adj-new-aux-trees

**adj-and-cut**

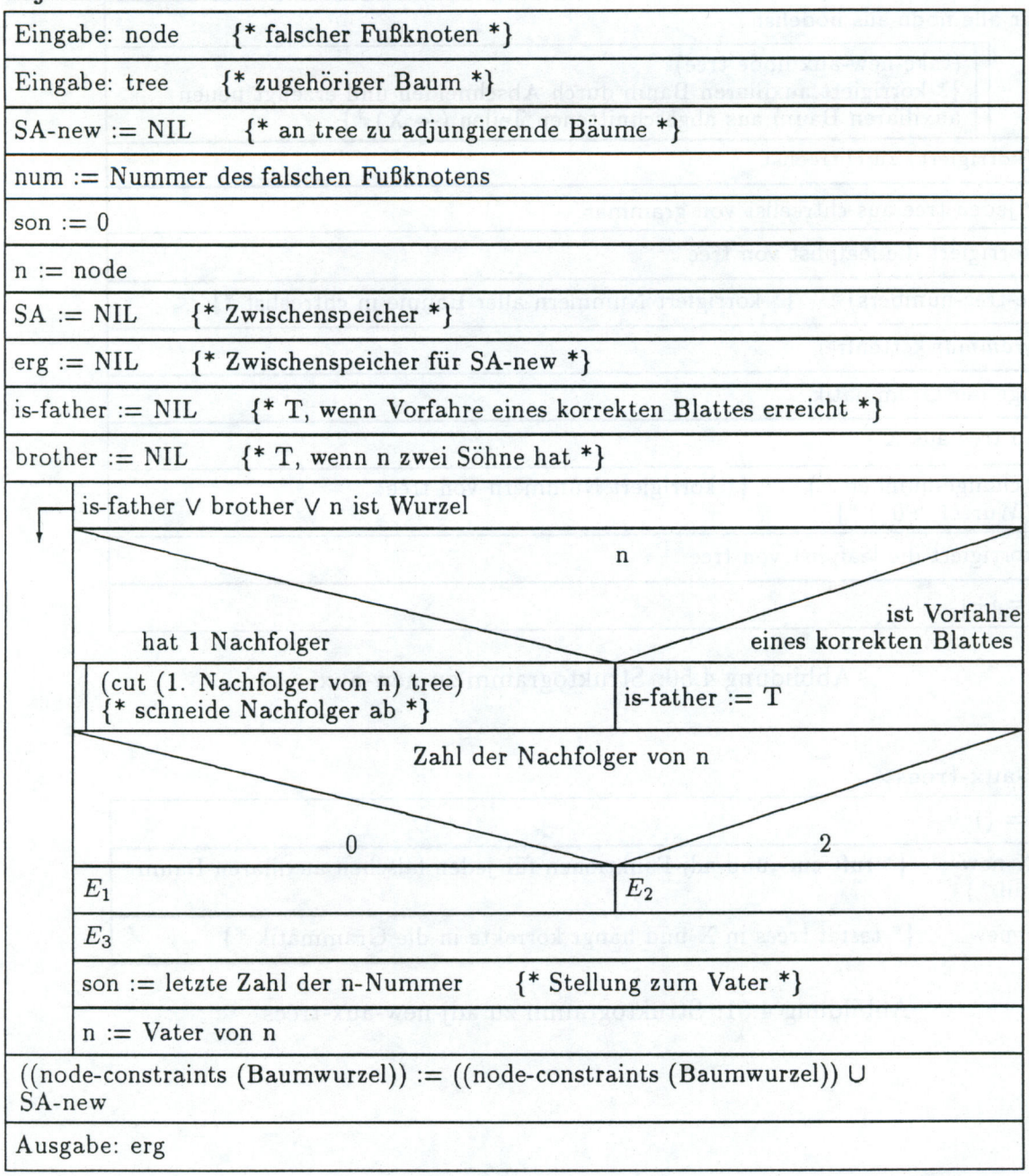



Abbildung 4.52: Struktogramm zu adj-and-cut

$E_1$

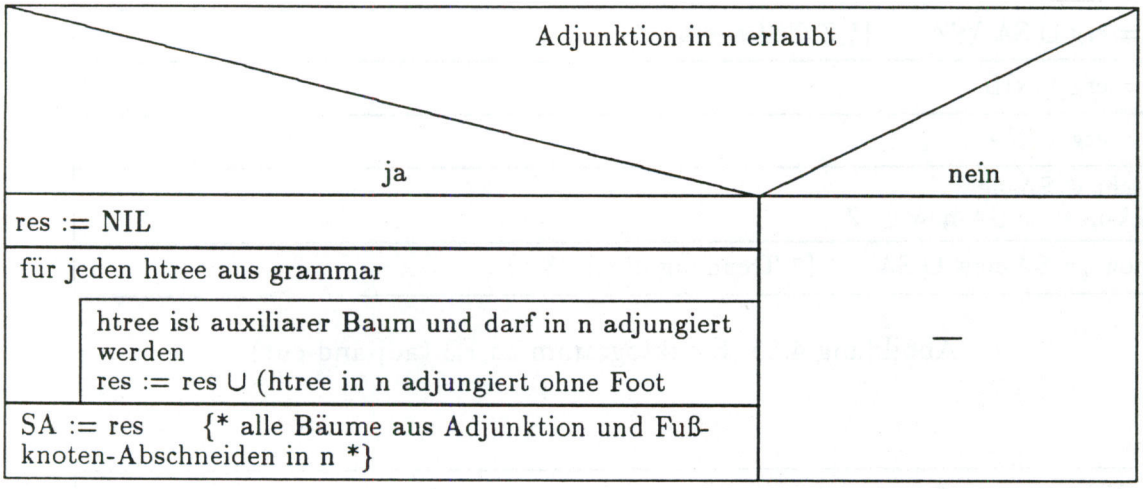



Abbildung 4.53: Struktogramm zu E1 (adj-and-cut)

$E_2$

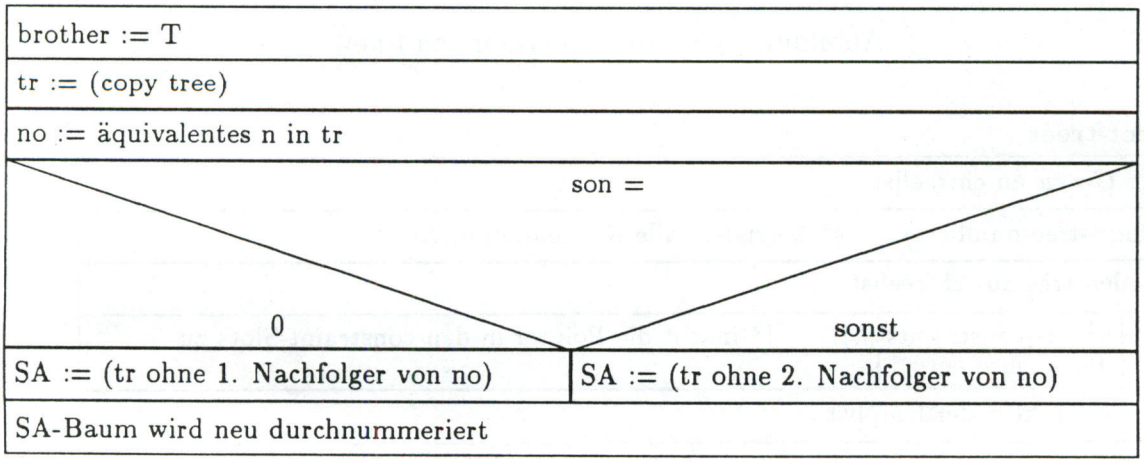



Abbildung 4.54: Struktogramm zu E2 (adj-and-cut)

$E_3$

| |
|---|
| erg := erg ∪ SA \ 'Z {* Z Codesymbol *} |
| erg := erg \ NIL |
| erg := erg \ 'NA |
| 'Z nicht ∈ SA-new<br>→ SA-new := SA-new ∪ 'Z |
| SA-new := SA-new ∪ SA {* Trennung durch 'Z *} |

Abbildung 4.55: Struktogramm zu E3 (adj-and-cut)

**F-new**

| | |
|---|---|
| X := NIL | |
| für jeden tree aus F | |
| | tree hat falschen Fußknoten<br>→ X := X ∪ (1.Element) |

Abbildung 4.56: Struktogramm zu f-new

**correct-trees**

| | |
|---|---|
| hänge *G-new* an chtreelist | |
| (change-tree-numbers) {* korrigiere alle Knotennummern *} | |
| für jeden tree aus chtreelist | |
| | (right-constraints ...) {* macht die Pointer in den constraint-Slots zu Baumnummern *} |
| | Korrektur der leafplist |

Abbildung 4.57: Struktogramm zu correct-trees

### Handhabung

Beim Aufruf von *grammar-to-e-free* muß die Grammatik aus Bäumen bestehen, deren DAGs compiliert und unifiziert sind und wo die Vererbungsrichtungen in den auxiliaren Bäumen berechnet sind. Das ist nach dem Aufruf von *grammar-to-2-form* auf die Eingabebäume der Fall.

Nach der Umformung sind die gleichen Bedingungen erfüllt, allerdings können hier zum erstenmal "dicke DAGs" auftauchen, da die Grammatik im Laufe der Umformung kettenfrei gemacht werden muß .

Fehlermeldungen können bei der Adjunktion auftreten (siehe Kapitel 4.4).

### 4.2.6 Transformation der $\epsilon$-freien Zwei-Form in Normalform

Die Funktionen zur Transformation einer $\epsilon$-freien Grammatik in Zwei-Form in die von uns verwendete Normalform sind in den Dateien "normalform" und "chain-free" abgelegt.

#### Erweiterungen der Definition

Im Modul "chain-free" wird die erforderliche Elimination von Ketten durchgeführt. Wie aus Kapitel 3 bekannt ist, müssen hierzu "dicke DAGs" bei Verschmelzung von Knoten einer Kette eingeführt werden, woraus eine umfangreichere Definition der Adjunktion mit Unifikation resultiert, da die beteiligten Knoten "dicke DAGs" tragen können. Diese Zusammenhänge sowie die erweiterte Definition sind Kapitel 2 zu entnehmen.

#### Datenstrukturen

Zur Realisierung der auftretenden "dicken DAGs" haben wir die Struktur *node* um zwei Slots (*hprep, hsuccplist*) erweitert. Sie dienen dazu, die alten Knoten, die verschmolzen sind, unter neuem Slot weiter bestehen zu lassen, um die Entstehungsgeschichte eines Knotens mit "dickem DAG" sofort parat zu haben. Dabei bilden *hsuccplist* und *hprep* einen Zyklus vom neuen Knoten in die alte Kette und wieder zurück zum neuen Knoten. Er selbst erhält als DAG eine Liste der DAGs der verschmolzenen Knoten und als Regel eine Liste der Regeln der verschmolzenen Knoten. Aus Einheitlichkeitsgründen besitzt auch jeder andere Knoten als DAG eine Liste, die nur diesen DAG umfaßt, ebenso auch eine Liste seiner Regel.

Bevor die Grammatik transformiert werden kann, müssen zunächst alle Knoten bekannt sein, deren Söhne nicht der Normalform entsprechen. Zur Verwaltung dieser Knoten dient die Struktur *mistake-protocol*, die für jeden Baum, der noch inkorrekte Knoten enthält, angelegt wird.

*(defstruct mistake-protocol*
*treep, case2, case3, case4, case6)*

Der Slot *treep* enthält den Baum selbst. Die Slots *case2, case3, case4, case6* enthalten alle Knoten des Baumes, die dem jeweiligen Fall angehören, bezogen auf Abbildung 4.58, die die Klassifizierung von inneren Knoten gemäß ihrer Söhne zeigt. Alle angelegten *mistake-protocol's* werden in der Variable *mistake-trees* abgelegt *(defvar mistake-trees)*.

Die Variable *root-constraints* enthält die Constraints für die Wurzeln der bei der Kettenelimination neu erzeugten auxiliaren Bäume *(defvar root-constraints)*.

Die Variable *chain-constraint* enthält alle Baumnummern von Bäumen, die bereits in der Baummenge der Grammatik enthalten sind, obwohl derselbe Baum noch einmal innerhalb der Kettenelimination erzeugt wird. Dies benötigt man für das Constraint des Kettenknotens *(defvar chain-constraint)*.

In der Variable *special-chain-dates* werden alle Knoten einer Kette und der sie ersetzende Kettenknoten für solche Ketten abgelegt, die eine Spezialposition einnehmen,

nämlich entweder an der Wurzel eines Baumes oder am Fußknoten eines auxiliaren Baumes stehen *(defvar special-chain-dates)*. Diese Ketten werden im folgenden Spezialketten genannt.

**Operatoren**

**find-mistake-trees:** sucht alle Knoten der Grammatik in Zwei-Form mit Söhnen, die nicht der Normalform entsprechen.

**tree-test:** legt für einen Baum ein *mistake-protocol* an und testet seine Knoten auf Normalform.

**node-test:** testet die Söhne eines Knotens und belegt ggf. das *mistake-protocol.*

**grammar-to-chain-free:** wandelt die Grammatik in Normalform um.

**tree-to-harbusch:** wandelt alle Knoten eines Baumes, die durch Einführung von Hilfsknoten korrigiert werden können, in Normalform um (case2, case3, case4).

**find-chains:** sucht aus den Knoten in *case6* eines *mistake-protocol* die Ketten heraus.

**tree-chains-to-harbusch:** wandelt alle Ketten eines Baumes in Normalform um.

**case6-to-harbusch:** wandelt eine Kette in Normalform um.

**make-chain-to-node1:** ersetzt eine Kette durch ihren Kettenknoten.

**first-trees:** führt alle möglichen einmaligen Adjunktionen in der Kette durch.

**eliminate-firstchains:** entfernt Ketten an der Wurzel der neu erzeugten auxiliaren Bäume.

**set-root-constraints:** setzt die Constraints an den Wurzeln der neu erzeugten auxiliaren Bäume.

**first-combination:** führt ausgehend von bereits durch Adjunktionen erzeugten Bäumen eine weitere Adjunktion im Sinne einer Initialisierung für die Funktion *combination* in der Kette durch.

**combination:** berechnet ausgehend vom Ergebnis der Funktion *first-combination* die restlichen Kombinationen dieser Adjunktion in der Kette.

**add-simple-k-trees:** fügt die in Wurzel- und Fußknotenlabel geänderten auxiliaren Bäume in die Baummenge der Grammatik ein.

**add-k-trees:** fügt die durch Adjunktionen in der Kette erzeugten auxiliaren Bäume in die Baummenge der Grammatik ein.

**add-tree-to-chtreelist:** entfernt die nach den Adjunktionen entstandenen Teilketten in den neu erzeugten auxiliaren Bäumen und fügt sie in die Baummenge der Grammatik ein.

**make-chain-to-node2:** eliminiert eine nach den Adjunktionen entstandene Teilkette.

### Algorithmus

Im letzten Transformationsschritt muß die in $\epsilon$-freier Zwei-Form vorliegende Grammatik in Normalform umgeformt werden. Die vollständige Beschreibung des Algorithmus ist in Kapitel 3 ausgeführt. An dieser Stelle soll die Umsetzung desselben auf die verwendeten Funktionen erläutert werden.

In den Bäumen der Grammatik können sechs verschiedene Ableitungsschritte vorkommen, die bei der Transformation unterschiedlich behandelt werden müssen. Sie sind noch einmal in Abbildung 4.58 aufgeführt.

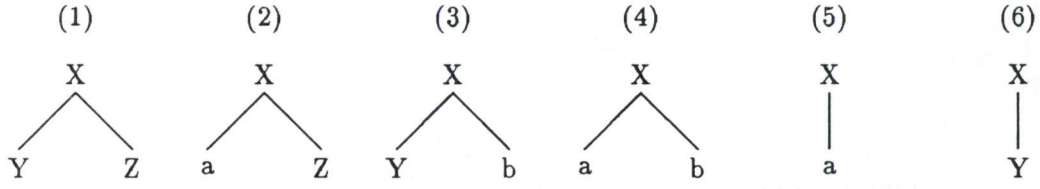



Abbildung 4.58: mögliche Sohnkonstellationen für innere Knoten

Die Fälle (1) und (5) aus Abbildung 4.58 sind bereits in Normaform. Die Fälle (2), (3) und (4) lassen sich durch Einführung nichtterminaler Hilfsknoten vor den Terminalen und entsprechende Regeländerung der beteiligten Knoten leicht transformieren (gesteuert durch die Funktion *tree-to-harbusch*). Der Algorithmus zur Transformation der Ketten in Fall (6) aus Abbildung 4.58 wurde wie folgt umgesetzt (gesteuert durch die Funktion *case6-to-harbusch*). Um die Normalform zu erhalten, wird die Kette durch einen einzigen Knoten, den Kettenknoten (mit "dickem DAG") ersetzt. Damit die dadurch wegfallenden Adjunktionsmöglichkeiten erhalten bleiben, werden neue auxiliare Bäume erzeugt, wobei jeder von ihnen gerade eine Kombination von Adjunktionen in die Kette repräsentiert (gesteuert durch die Funktion *case6-to-harbusch*). Das Entfernen der Kette veranschaulicht Abbildung 4.59.

In den neuen auxiliaren Bäumen können an der Wurzel nicht mehr die bisherigen Bäume adjungiert werden, da der Kettenknoten einen neuen Label besitzt, den auch die

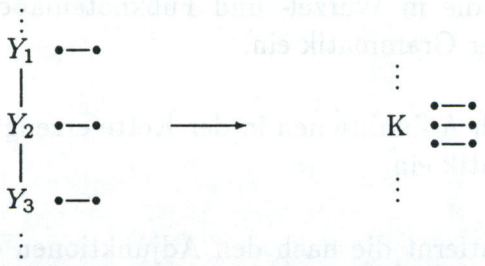



Abbildung 4.59: Ersetzen einer Kette durch den Kettenknoten

neuen Bäume tragen müssen. Daher werden alle in den Knoten der Kette adjungierbaren auxiliaren Bäume durch die Funktion *add-simple-k-trees* auch als Bäume mit diesem Label an Wurzel und Fußknoten in die Baummenge der Grammatik eingefügt. Jetzt werden die neuen auxiliaren Bäume erzeugt, die die weggefallenen Adjunktionsmöglichkeiten repräsentieren. Die Funktion *first-trees* führt zunächst alle möglichen einmaligen Adjunktionen in den Knoten der Kette aus. Das Aussehen der dadurch entstandenen Bäume zeigt Abbildung 4.60.

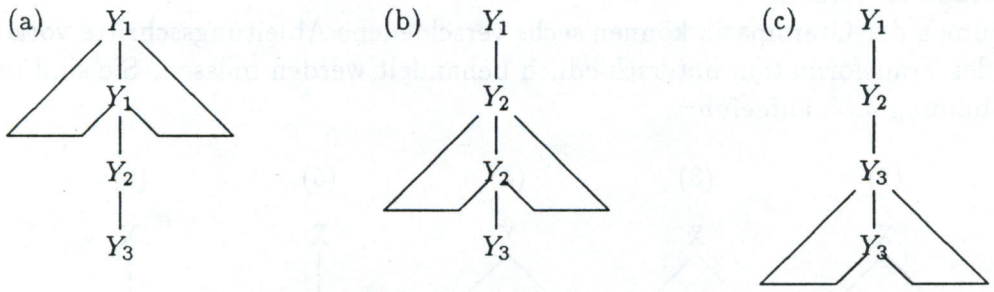



Abbildung 4.60: Bäume nach Aufruf von *first-trees*

Da im folgenden alle weiteren Adjunktionen in diese Bäume unterhalb der bereits ausgeführten Adjunktion erfolgen, werden an dieser Stelle alle Teilketten an der Wurzel durch die Funktion *eliminate-firstchains* und das erforderliche Constraint an der Wurzel eingesetzt, da die dort adjungierbaren Bäume bereits durch die Funktion *add-simple-k-trees* erzeugt wurden. Dadurch erhalten die Bäume die Gestalt aus Abbildung 4.61.

Die Bäume aus Abbildung 4.61 stellen einerseits bereits eine mögliche Kombination von Adjunktionen dar (nämlich jeweils genau eine) und werden daher nach den noch erforderlichen Korrekturen durch die Funktion *add-k-trees* zur Baummenge der Grammatik hinzugefügt. Zum anderen dienen sie als Ausgansbäume für weitere Adjunktionen. Die Funktionen *first-combination* und *combination* berechnen alle hier möglichen Kombinationen. Man erhält die Bäume aus Abbildung 4.62. Baum (c) aus Abbildung 4.61 braucht nicht mehr betrachtet zu werden, da dort im untersten Knoten adjungiert wurde.

Die Bäume aus Abbildung 4.62 stellen wiederum sowohl eine mögliche Kombination von Adjunktionen als auch den Ausgangspunkt für weitere Adjunktionen dar. Diese Iteration wiederholt sich solange, bis die Kombination erreicht ist, bei der in jedem Knoten

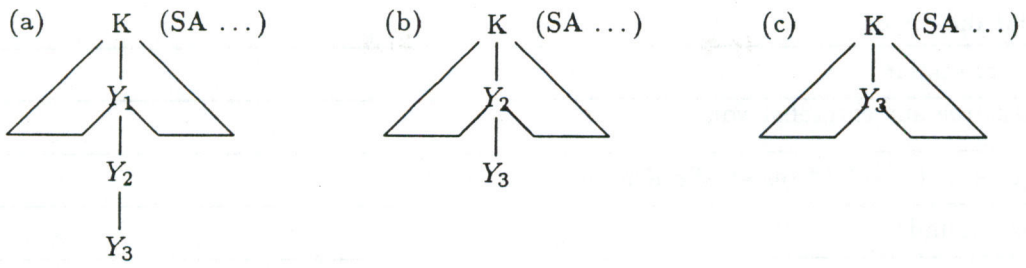



Abbildung 4.61: Bäume nach Entfernung von Teilketten an der Wurzel

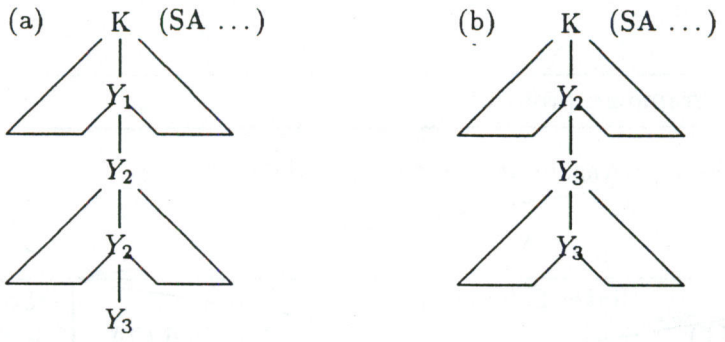



Abbildung 4.62: Bäume nach Durchführung einer weiteren Adjunktion

der Kette eine Adjunktion stattgefunden hat.

Schließlich muß noch beachtet werden, ob die behandelte Kette eine Spezialposition im Baum einnahm, also an Wurzel oder Fußknoten eines Baumes stand. In diesem Fall fand bei der Einführung des Kettenknotens keine Labeländerung statt, so daß alle bisher möglichen Adjunktionen bzgl. dieses Labels in allen Bäumen per Constraint abgedeckt werden müssen, da die durch die Kettenelimination erzeugten Bäume in diesen Knoten nicht adjungiert werden dürfen. Der Kettenknoten erhält durch die Funktion *case6-to-harbusch* als Constraint, daß wie üblich nur die neuen Bäume in ihm adjungiert werden dürfen.

**find-mistake-trees**

| Eingabe: grammar |
|---|
| für jeden tree aus chtreelist von grammar |
| (tree-test tree) {* testet alle Knoten von tree *} |
| Ausgabe: "found" |

Abbildung 4.63: Struktogramm zur Funktion find-mistake-trees

**node-test**

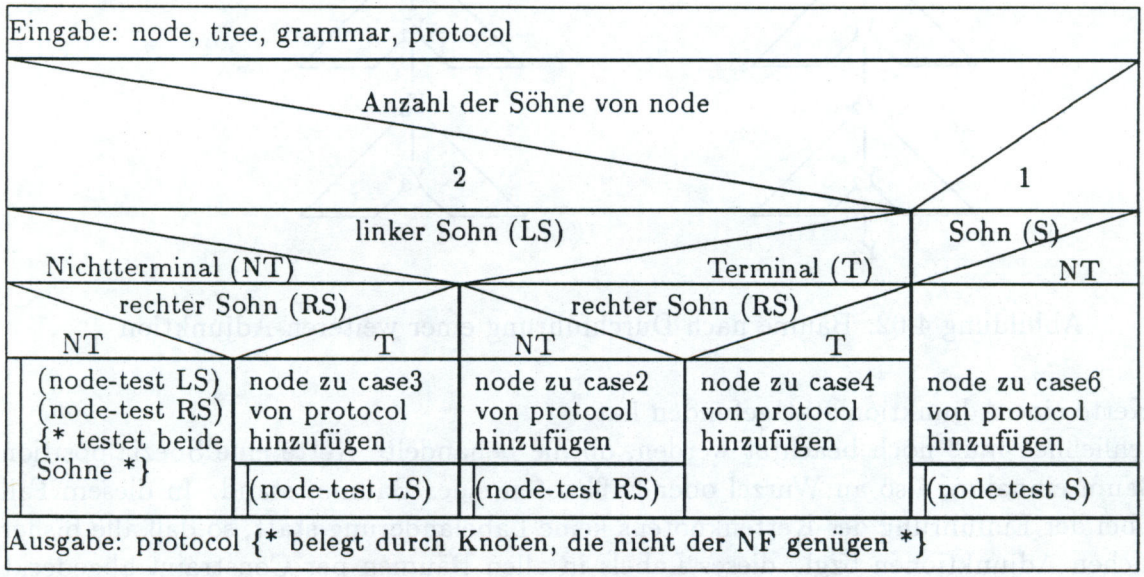



Abbildung 4.64: Struktogramm zur Funktion node-test

**grammar-to-chain-free**

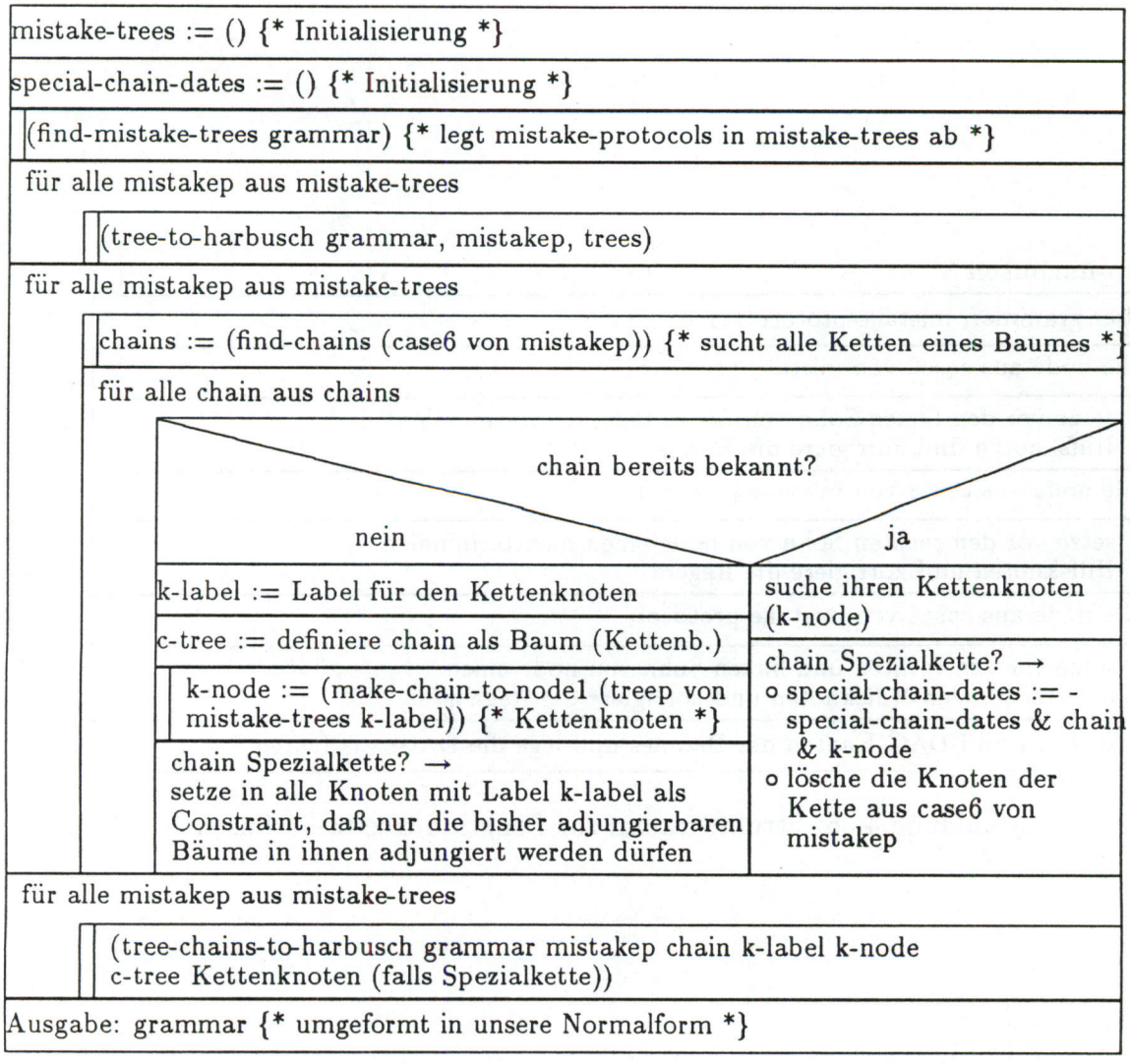



Abbildung 4.65: Struktogramm zur Funktion grammar-to-chain-free

**tree-to-harbusch**

| |
|---|
| Eingabe: grammar, mistake-protocol, tree |
| für alle node aus case2 von mistake-protocol |
| setze vor den linken Sohn von node einen nichtterminalen Hilfsknoten und korrigiere die Regeln |
| für alle node aus case3 von mistake-protocol |
| setze vor den rechten Sohn von node einen nichtterminalen Hilfsknoten und korrigiere die Regeln |
| für alle node aus case4 von mistake-protocol |
| setze vor den rechten und linken Sohn von node einen nichtterminalen Hilfsknoten und korrigiere die Regeln |
| erneuere DAGs und DAG-Kanten des Baumes und lege die DAGs als Listen an |

Abbildung 4.66: Struktogramm zur Funktion tree-to-harbusch

**case6-to-harbusch**

Eingabe: grammar, tree, chain, Kettenbaum (chain-tree), Kettenlabel (k-label), Kettenknoten (k-node), Kettenknoten falls Spezialkette (special)

root-constraints := (), chain-constraint := () {* Initialisierung *}

auxtrees := Liste: i-tes Element sind die im i-ten Knoten der Kette adjungierb. Bäume

simple-k-trees := ändere in allen Bäumen aus auxtrees Wurzel und Fußknoten auf k-label

(add-simple-k-trees grammar simple-k-trees k-label) {* belegt auch root-constraints *}

k-trees := (eliminate-firstchains (first-trees auxtrees k-label chain-tree chain) k-label) {* adjungiert jeden Baum aus auxtrees einmal in chain-tree am passenden Knoten und entfernt Teilketten an der Wurzel der entstandenen Bäume *}

(set-root-constraints grammar k-trees root-constraints) {* Constraints an Wurzeln *}

constraint-counter := 1 + Anzahl der Bäume der Grammatik

(add-k-trees grammar k-trees k-label) {* Bäume mit einer durchgef. Adjunktion *}

up-k-trees := k-trees ohne letzte Liste {* Bäume, in die noch adjungiert wird *}

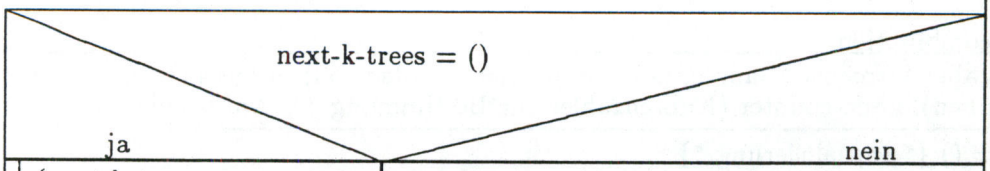
für alle downtrees von (cdr auxtrees) unter cdr bis NIL {* noch zu adj. Bäume *}

next-k-trees := () {* Initialisierung *}

n := 0

dtrees := downtrees

für alle uptreelist aus up-k-trees




(next-k-trees := (first-combination uptree-list dtrees Knotenzähler) {* Knotenzähler bestimmt Anhängeknoten *}

n := n + 1

next-k-trees := erste n Listen von next-k-trees & (combination uptreelist dtrees (nthcdr n next-k-trees) Knotenzähler) {* Knotenzähler bestimmt Anhängekn. *}

{* erzeugt alle Kombinationen von Adjunktionen eines Baumes aus dtrees in einen Baum aus uptreelist, wobei die nächste Adjunktion immer unter bereits stattgefundenen ausgeführt wird; damit wird next-k-trees eine Liste, deren i-tes Element (von hinten) eine Liste der erzeugten Bäume ist, deren unterste Adjunktion im i-ten Knoten von unten stattfand *}

dtrees := (cdr dtrees)

(add-k-trees grammar next-k-trees k-label) {* fügt die aktuellen Bäume hinzu *}

up-k-trees := next-k-trees ohne letzte Liste

constraint := chain-constraint {* belegt durch Baumnummern von Bäumen, die schon in der Baummenge waren *}

i := constraint-counter

i = 1 + Anzahl der Bäume der Grammatik

constraint := constraint & i

i := i + 1

node-constraint von k-node := constraint

chain ist mehrfach auftretende Spezialkette →
- o suche die anderen Kettenknoten dieser Kette in special-chain-dates
- o node-constraint dieser Knoten := constraint

first-trees

| |
|---|
| Eingabe: Liste der Baumlisten (auxtrees), k-label, Kettenbaum (chain-tree), chain |
| erg := () {* Initialisierung *} |
| adj-node := Wurzel von chain-tree {* Initialisierung des Anhängeknotens |
| für alle treelist aus auxtrees |
| help := () {* Initialisierung *} |
| für alle tree aus treelist |
| erg-tree := adjungiere tree in adj-node von chain-tree |
| help := help & erg-tree |
| erg := erg & help |
| adj-node := Nachfolger von adj-node |
| Ausgabe: erg {* Liste der nach Anhängeknoten geordneten Baumlisten *} |

Abbildung 4.68: Struktogramm zur Funktion first-trees

first-combination

| |
|---|
| Eingabe: k-treelist (Bäume, in die adjungiert werden soll), n-trees (Liste von zu adj. Bäumen), node-counter (Knotenzähler zur Bestimmung des Anhängeknotens |
| erg := () {* Initialisierung *} |
| für jede n-treelist aus n-trees |
| erg := erg & Liste aller Kombinationen von Bäumen aus k-treelist mit einer Adj. von Bäumen aus n-treelist, adjungiert im zu n-treelist gehörigen Knoten |
| Ausgabe: erg |

Abbildung 4.69: Struktogramm zur Funktion first-combination

**combination**

| Eingabe: k-treelist, n-trees, nextlist (next-k-trees für case6-to-harbusch), node-counter |
|---|
| n-trees nicht leer →<br>nextlist := nextlist & Kombination von Bäumen aus k-treelist mit Bäumen aus der ersten Liste in n-trees & (combination k-treelist (cdr n-trees) (cdr nextlist) (node-counter - 1)) {* Dadurch behalten alle Bäume einer Teilliste aus nextlist die Eigenschaft, daß die unterste Adjunktion im gleichen Knoten stattfand *} |
| Ausgabe: nextlist {* enthält alle Bäume, die durch eine weitere Adjunktion entstehen *} |

Abbildung 4.70: Struktogramm zur Funktion combination

**add-simple-k-trees**

Eingabe: grammar, Liste der Baumlisten (auxtrees), k-label

n := 1 + Anzahl der Bäume der Grammatik

constraint-list := () {* Sammelt Nummern der geänderten auxiliaren Bäume *}

für alle treelist aus auxtrees

- constraint := (SA)
- für alle tree aus treelist
  - setze Label des Fußknotens von tree auf k-label
  - setze constraint des Fußknotens von tree auf (NA) {* Nulladjunktion *}
  - korrigiere DAGs, DAG-Kanten und Regeln {* wegen neuer Labels *}
  - tree bereits in Baummenge der Grammatik

| ja | nein |
|---|---|
| constraint := constraint & Nummer des bereits entfernten Baumes | constraint := constraint & n |
| | n := n + 1 |

- constraint-list := constraint-list & constraint
- für alle tree aus treelist
  - tree noch nicht in Baummenge der Grammatik →
    - o node-constraint von Baumwurzel := constraint
    - o Baumnummer von tree := 1 + Anzahl der Bäume der Grammatik
    - o erzeuge daraus die Knotennummern der Knoten von tree
    - o füge tree zur Baummenge hinzu

root-constraints := constraint-list

Ausgabe: root-constraints {* Constraints für die Wurzeln der noch zu erz. Bäume *}

Abbildung 4.71: Struktogramm zur Funktion add-simple-k-trees

**add-tree-to-chtreelist**

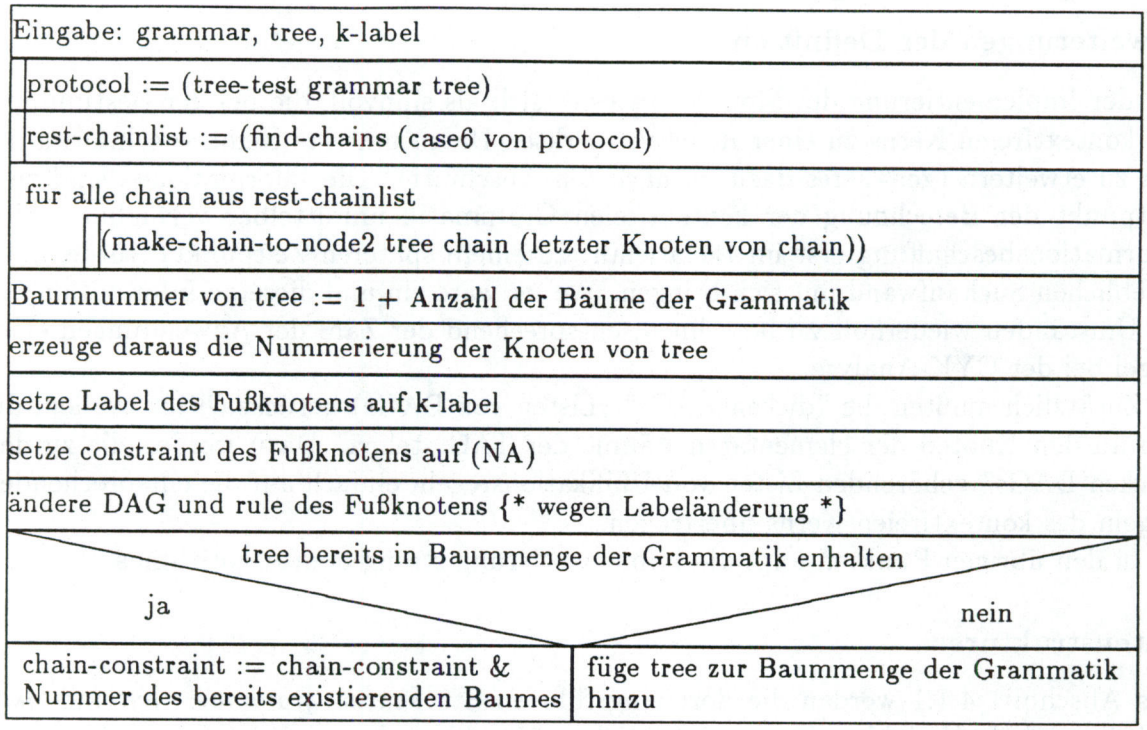



Abbildung 4.72: Struktogramm zur Funktion add-tree-to-chtreelist

### Handhabung

Das Modul für den letzten Transformationsschritt ruft man mit der Funktion *grammar-to-chain-free* auf. Sie steuert alle erforderlichen Teilschritte in der angegebenen Weise. Die Grammatik in der Variable *grammar* muß allerdings in $\epsilon$-freier Zwei-Form vorliegen, da das Modul darauf aufbaut. DAGs und Unifikationsregeln der Knoten müssen bereits als Listen vorliegen, da wir einheitlich damit arbeiten. Diese Voraussetzungen sind dadurch erfüllt, daß das Modul nur im Anschluß an "eps-free" läuft. Die Transformation ist beendet, wenn die neu belegte Variable *grammar* zurückgegeben wird.

## 4.3 Beschreibung unseres Parsingalgorithmus

In diesem Abschnitt gehen wir davon aus, daß die Eingabegrammatik mit dem Programmpaket zur Normalformtransformation in unsere Normalform gebracht wurde.

Im folgenden werden die einzelnen Schritte der Analyse, die bereits in Abschnitt 3.2 beschrieben wurden, mit konkreten Funktionsnamen des Programmes belegt. Zunächst wird der kontextfreie Kern der TAG berechnet (siehe Abschnitt 4.3.1), mit dem dann eine kontextfreie Analyse mit den CYK-Algorithmus durchgeführt wird (Abschnitt 4.3.2). In Abschnitt 4.3.3 wird abschließend die Iteration erläutert.

### 4.3.1 Berechnung des kontextfreien Kerns der TAG

#### Erweiterungen der Definition

Bei der Implementierung des Moduls erwies es sich als sinnvoll, die bei der Bestimmung des kontextfreien Kerns zu einer Regel vermerkten Daten um eine Reihe von Informationen zu erweitern (genaueres dazu im nächsten Abschnitt). Die Informationen sind zum Zeitpunkt der Berechnung der kontextfreien Grammatik unmittelbar verfügbar. Eine Informationsbeschaffung erst im Bedarfsfall (zu einem späteren Zeitpunkt) würde einen erheblichen Suchaufwand mit sich bringen. Weiter wäre ein und dieselbe Information unter Umständen wiederholt zu berechnen, entsprechend der Zahl der Anwendungen einer Regel bei der CYK-Analyse.

Zusätzlich mußten die "dicken DAGs" (Listen von DAGs) berücksichtigt werden, die nun an den Knoten der elementaren Bäume der TAG stehen. Dazu werden die zu den "dicken DAGs" gehörenden Listen von Unifikationsregeln einfach auf die entsprechenden Regeln des kontextfreien Kerns übertragen.

In den übrigen Punkten folgt die Implementierung streng dem Algorithmus.

#### Datenstrukturen

Aus Abschnitt 4.1.1 werden die dort eingeführten Strukturen *grammar*, *tree* und *node* und die globale Variable *grammar* verwendet. Neu hinzu kommen in diesem Abschnitt die globalen Variablen *nodehash* und *cfgram*:

```
(defvar nodehash (make-hash-table
    :test #'equal    :size 100    :rehash-size 1.5    :rehash-theshold 0.95))

(defvar cfgram NIL)
```

*Nodehash* erlaubt mit der eindeutigen Knotennummer als Schlüssel (siehe Slot *number* der Structure *node*) den direkten Zugriff auf die Baumknoten einer TAG. An *cfgram* ist nach Abarbeitung des Moduls die zur TAG korrespondierende kontextfreie Grammatik gebunden, zu deren Darstellung die nachfolgenden Strukturen dienen:

```
(defstruct CFG
    tlist, ntlist, trules, brules)
```

Die Slots *tlist* und *ntlist* entsprechen den gleichnamigen Slots der Struktur *grammar*. An die Slots *trules* bzw. *brules* ist jeweils eine Liste von Strukturen des Typs *CFR* gebunden. *CFR* steht für kontextfreie Regel, wobei an den beiden Slots noch nach *terminal-rules* (→ *trules*) und *branching-rules* (→ *brules*) unterschieden wird. *Terminal-rules* sind Regeln, deren rechte Seite aus einem Terminal (korrekter wäre Präterminal) besteht. Die rechte Seite von *branching-rules* setzt sich aus zwei Nichtterminalen zusammen.

*(defstruct CFR*
*cf-rule, readings)*

Der Slot *cf-rule* steht für eine kontextfreie Regel. An ihn ist eine Liste von Strings gebunden, von denen jeder für ein Terminal- bzw. Nichtterminal-Label steht. Die rechte Seite wird von den übrigen Strings gebildet. An den Slot *readings* ist eine Liste von Strukturen des Typs *reading* gebunden. Die Einführung von Lesarten einer kontextfreien Regel rührt daher, daß mehrere aus der TAG extrahierte Regeln zwar die gleichen Labels aufweisen, sich jedoch in der Nummernnotation unterscheiden.

*(defstuct reading*
*number, patr-rule, dag, constraints, footn-father, i-or-a, root)*

Die Slots *patr-rule*, *dag*, *constraints* und *i-or-a* entsprechen in der Reihenfolge den Slots *rule*, *dag*, *constraints* und *i-or-a* der Struktur *node*. An *number* ist ein String gebunden, der für die eindeutige Knotennumer der linken Seite der kontextfreien Regel steht. Die Syntax dieser Nummer entspricht der am Slot *number* der Struktur *node*. Auf eine Darstellung der Nummern der rechten Seite einer Regel wurde verzichtet, da sich diese aus der Nummer der linken Seite eindeutig berechnen lassen. *Footn-father* ist "0" falls der linke, "1" falls der rechte bzw. NIL falls kein Sohn Fußknoten ist. Der Slot *root* ist T, falls die linke Seite der kontextfreien Regel gleichzeitig die Wurzel eines Baumes der TAG ist (sonst NIL).

### Operatoren

**catch-CFG:** stellt die Schnittstelle des Moduls dar und realisiert über den Aufruf des Operators *catch-rule* die Extrahierung der kontextfreien Grammatik.

**catch-rule:** leitet von einem Knoten eines TAG-Baumes ausgehend die zugehörige Regel ab, baut diese in die kontextfreie Grammatik ein und arbeitet so durch rekursiven Aufruf Ast für Ast eines Baumes ab.

### Algorithmus

Auf die Hauptmerkmale des Algorithmus wurde bereits in Abschnitt 3.4 eingegangen. Die Realisierung der Operatoren *catch-CFG* und *catch-rule* ist den Struktogrammen der Abbildungen 4.73 und 4.74 zu entnehmen. Der entsprechende LISP-Code findet sich in der Datei "contfree.l".

### Handhabung

### Parameter

Der Aufruf des Moduls erfolgt ohne Prameter (→ Funktion *catch-CFG*). Um eine korrekte Arbeitsweise zu garantieren, muß an die globale Variable *grammar* eine TAG in

**catch-CFG**

| |
|---|
| cfgram := structure des Typs CFG mit folgender Slot-Initialisierung: tlist := (grammar-tlist grammar), ntlist := (grammar-ntlist grammar), trules, brules := NIL |
| für alle tree aus der Grammatik |
| (catch-rule (list (tree-rootp tree)) (tree-footp tree) (tree-i-or-a tree) T) |

Abbildung 4.73: Struktogramm zur Funktion catch-CFG

**catch-rule**

| |
|---|
| Eingabe: nodelist {* Liste von Wurzelknoten von Teilbäumen elementarer Bäume *}<br>Eingabe: footp {* Pointer auf den Fußknoten des 1. Teilbaums aus nodelist *}<br>Eingabe: aux {* Flag: 1. Element aus nodelist ist Teil eines initialen Baums *}<br>Eingabe: root {* Flag, das angibt, ob das 1. Element aus nodelist Wurzel eines elementaren Baumes ist *} |
| nodelist ≠ ()<br>ja |
| node := (car nodelist) |
| trage node mit seiner Nummer als Schlüssel in nodehash ein |
| LS := (node-label node)<br>RS1 := (node-label (car (node-succplist node)))<br>RS2 := (node-label (cadr (node-succplist node)))<br>cf-rule := (list LS RS1 RS2) {* entspricht der kontextfreien Regel LS → RS1 RS2 *} |
| füge cf-rule in die kontextfreie Grammatik cfgram ein |
| (catch-rule (append (cdr nodelist)) footp aux NIL) |

Abbildung 4.74: Struktogramm zur Funktion catch-rule

Normalform gebunden sein. In jedem Fall muß die Eingabe die für die Transformation der Grammatik zuständigen Moduln durchlaufen haben, auch wenn dies aufgrund der Grammatik-Struktur nicht als notwendig erscheinen mag (z.B. wenn die Grammatik bereits in Chomsky-Normalform ist).

#### Ausgabe

Der Aufruf der Funktion *catch-CFG* hat nur Seiteneffekte, die sich auf globale Strukturen beziehen, die in nachfolgenden Programmteilen benötigt werden. Insbesondere sind dies die Variablen *cfgram* und *nodehash*.

#### Fehlermeldungen

Als Fehlermeldungen können nur solche des LISP-Systems auftreten, wenn die im Abschnitt 'Parameter' geforderten Voraussetzungen nicht erfüllt wurden.

### 4.3.2 CYK-Analyse des kontextfreien Kerns einer TAG

#### Erweiterungen der Definition

Die bei der Implementierung des Moduls vorgenommenen Erweiterungen gegenüber dem Parsingalgorithmus beschränken sich im wesentlichen, wie im vorangehenden Modul, auf zusätzlich in die Einträge der CYK-Matrix aufgenommene Informationen, die eine effizientere Arbeitsweise gestatten.

Zusätzlich mußte der neuen Struktur der "dicken DAGs" Rechnung getragen werden. Das heißt, daß an jedem Eintrag in der CYK-Matrix nun ein "dicker DAG", also eine Liste von DAGs, anliegt. Zu einer Regel 00 $\rightarrow$ 01 02 des kontextfreien Kerns der TAG läßt sich ebenfalls lokal ein "dicker DAG" aufbauen. Die Teil-DAGs für die Sohnknoten sind dann entsprechend diejenigen des letzten DAGs in der Liste von DAGs, die den "dicken DAG" repräsentieren.

In Anlehnung an das Beispiel (Abbildung 3.27) aus Abschnitt 3.5 kann der lokale "dicke DAG" in die baumübergreifende Struktur eingebunden werden, indem der oben beschriebene Teil-DAG zum Sohnknoten 01 mit dem ersten Element der Liste von DAGs in (i,j) und der entsprechende Teil-DAG zum Sohnknoten 02 mit dem ersten Element der Liste von DAGs in (i+j+1,k) unifiziert wird.

#### Datenstrukturen

Aus Abschnitt 4.3.1 werden die dort eingeführten Structures *CFG*, *CFR* und *reading* und die globalen Variablen *cfgram* und *nodehash* übernommen.

Bei der Umsetzung der CYK-Analyse kommt der CYK-Matrix im Hinblick auf die Datenstrukturen eine zentrale Rolle zu. Der Matrix-Begriff legt nahe, die CYK-Matrix als 2-dimensionalen ARRAY zu implementieren. Da es bei der CYK-Analyse jedoch oft vorkommt, daß Matrix-Elemente nicht besetzt werden und die CYK-Matrix Dreiecksform besitzt, also nur etwas mehr als die Hälfte eines 2-dimensionalen ARRAYS tatsächlich

beansprucht wird, fiel die Entscheidung zugunsten eines Hashtables.

```
(defvar cykhash (make-hash-table
    :test #'equal    :size 100    :rehash-size 1.5    :rehash-theshold 0.95))
```

Als Schlüssel dienen 2-elementige Listen ganzer Zahlen. Die erste bezeichnet den Zeilen-, die zweite den Spaltenindex der CYK-Matrix. Der Wert zu einem Schlüssel besteht aus einer Liste sogenannter CYK-Zellen.

```
(defstuct cykcell
    label, cyknodes)
```

Eine solche CYK-Zelle faßt mehrere der im Parsingalgorithmus beschriebenen Einträge zu einer logischen Einheit zusammen. Allen Einträgen einer Einheit ist das gleiche Nichtterminallabel gemeinsam, das an den Slot *label* gebunden ist. Unter dem Slot *cyknodes* ist eine Liste von Strukturen des Typs *cyknode* abgelegt, die die eigentlichen Einträge der CYK-Matrix darstellen.

```
(defstruct cyknode
    label, cykhash-key, cyknodehash-key, edges-from, adjoining, number, root,
    i-or-a, subtree, footp, dag, rule, constraints, LP, RP)
```

Die Slots *label*, *number*, *i-or-a*, *dag*, *rule* und *constraints* entsprechen den gleichnamigen Slots der Strukturen *node* in Abschnitt 4.1.1, *root* dem der Struktur *reading* im Abschnitt 4.3.1. An den Slot *edges-from* ist eine Liste von Pointern auf CYK-Knoten gebunden, von denen aus Kanten in den Knoten eingehen. Dem Slot *adjoining* kommt erst im Modul 4.3.3 (siehe Abschnitt 4.3.3) eine Bedeutung zu. Dort wird an dieser Stelle ein Pointer auf den Wurzelknoten des Baumes abgelegt, der im Knoten adjungiert wurde. Die Information über die Untergeschichte eines CYK-Knotens gemäß seiner Position im TAG-Baum wird analog zum Parsingalgorithmus als Symbol am Slot *subtree* vermerkt. Entsprechend liegt unter *footp* eine Liste von Pointern auf Fußknoten (falls vorhanden) oder NIL. An den Slots *LP* und *RP* stehen Listen von Strukturen des Typs *edge* wobei in *LP* linke bzw. in *RP* rechte Kanten abgelegt sind. Kanten aus *LP* und *RP* mit gleicher Indexposition bilden ein Kantenpaar. Unter dem slot *cykhash-key* ist der Schlüssel zum Hashtable *cykhash* abgelegt, über den die CYK-Zelle erreicht werden kann, deren Bestandteil der CYK-Knoten selbst ist. Der Slot *cyknodehash-key* liefert den Schlüssel zum Hashtable *cyknodehash* unter dem er selbst dort abgelegt ist.

```
(defvar cyknodehash (make-hash-table
    :test #'eq    :size 500    :rehash-size 1.2    :rehash-theshold 0.9))
```

Wie bereits oben erwähnt, kann über *cyknodehash* jeder in der CYK-Matrix vorkommende Knoten direkt erreicht werden. Als Schlüssel dient dazu ein intern erzeugtes, eindeutiges

Symbol, dem nach außen keinerlei Bedeutung zukommt.

An dieser Stelle sei erwähnt, daß alle im Zusammenhang mit den Slots der Struktur *cyknode* vorkommenden Pointer auf Structures des gleichen Typs keine Pointer im strengen Sinn darstellen. Vielmehr sind es Symbole, die als Schlüssel zum HASHTABLE *cyknodehash* dienen. Auf diese indirekte Verpointerung kann zweifelsohne zugunsten einer direkten verzichtet werden, die zudem effizienter ist. Andererseits ist die hier gewählte Lösung eine Möglichkeit, mit geringem Aufwand die Auswirkungen der einzelnen Programmschritte auf die CYK-Matrix nachvollziehbar zu machen, was für die Programmentwicklung unerläßlich ist.

*(defstuct edge*
*cykhash-key, cyknodehash-key, dag)*

Alle Slots der Struktur *edge* entsprechen den gleichnamigen Slots der Struktur *cyknode*. Der Inhalt ist auf den Zielknoten bezogen, auf den die Kante zeigt.

*(defvar word NIL)*

*(defvar word-length NIL)*

An die globale Variable *word* wird die zu parsende Eingabe in Form einer Liste von Symbolen gebunden. *Word-length* gibt Auskunft über die Länge der Eingabe.

*(defstruct sat*
*TI, TA, LI, LA, rest)*

Die Struktur *sat* ist im Zusammenhang mit der Iteration zum Finden aller innersten Bäume zu sehen. Die Definition erfolgte bereits in diesem Abschnitt, da der Parsingalgorithmus die Initialisierung einer solchen "Menge aktiver Bäume" im Rahmen der CYK-Analyse vorsieht. An jeden Slot wird eine Liste von Pointern auf Strukturen des Typs *cyknode* gebunden. Je nach der Information am Slot *subtree* eines CYK-Knotens wird selbiger in der Struktur *sat* unter einen gleichnamigen Slot eingehängt. CYK-Knoten, deren *subtree* NIL ist, werden unter dem Slot *rest* zusammengefaßt.

*(defvar SAT0 (make-sat))*

Die globale Variable *SAT0* steht für die "Menge aktiver Bäume", die im Rahmen der CYK-Analyse initialisiert wird.

### Operatoren

**cyk:** stellt die Schnittstelle des Moduls dar und führt mit Hilfe der nachfolgenden Operatoren die CYK-Analyse der Eingabe durch.

**make-Vi0:** kreiert, falls möglich, das Element (i 0) der CYK-Matrix.

**make-Vi1:** kreiert, falls möglich, das Element (i 1) der CYK-Matrix.

**make-Vij:** kreiert, falls möglich, das Element (i j) der CYK-Matrix für $j > 1$. Im Gegensatz zu *make-Vi0* und *make-Vi1* kann *make-Vij* im Laufe der CYK-Analyse für ein und dasselbe Matrixelement mehrfach aufgerufen werden. Mit jedem Aufruf können zusätzliche Einträge in diesem Matrixelement abgelegt werden.

**accept-word-cyk:** fällt die Entscheidung über Akzeptieren oder Nicht-Akzeptieren der Eingabe gemäß der CYK-Analyse. Die Eingabe wird akzeptiert, falls im Element (1 n) (n = Länge der Eingabe) der CYK-Matrix mindestens ein CYK-Knoten existiert, der die Wurzel eines elementaren Baumes der TAG ist.

### Algorithmus

Die Hauptmerkmale des Algorithmus wurden bereits in den Abschnitten 3.4 und 3.5 vorgestellt. Die Realisierung der oben eingeführten Operatoren ist den nachfolgenden Struktogrammen zu entnehmen, wobei das Struktogramm aus Abbildung 4.75 in seiner Grundstruktur und der Terminologie der in Pseudo-Pascalcode verfaßten Version des CYK-Algorithmus aus [4] S. 139-140 folgt. Die im Struktogramm aus Abbildung 4.76 verwendete Funktion *readings-form-lex* stellt die Lexikonschnittstelle dar, auf die in Abschnitt 4.1.2 näher eingegangen wird.

Der entsprechende LISP-Code findet sich in der Datei "cyk.l".

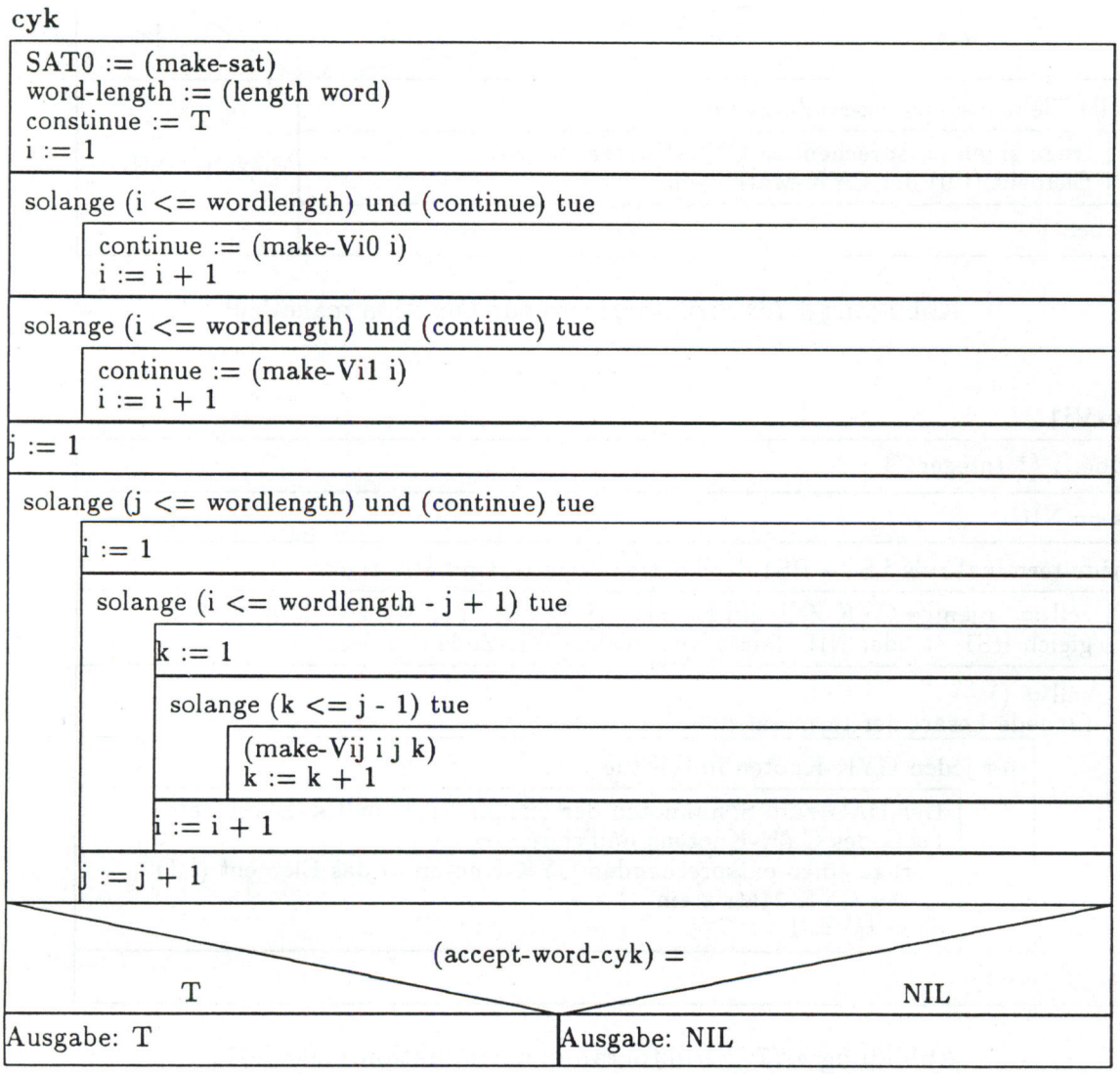



Abbildung 4.75: Struktogramm zur Funktion cyk

**make-Vi0**

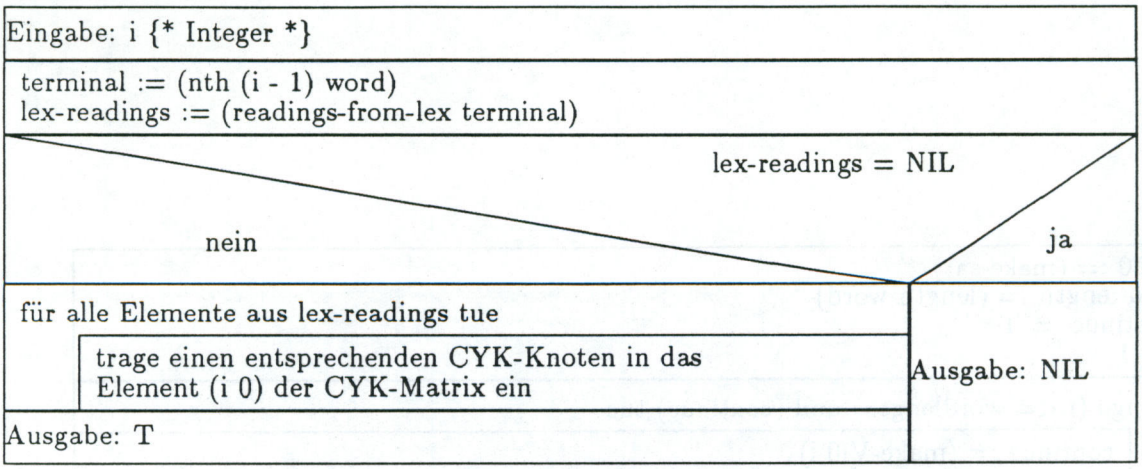



Abbildung 4.76: Struktogramm zur Funktion make-Vi0

**make-Vi1**

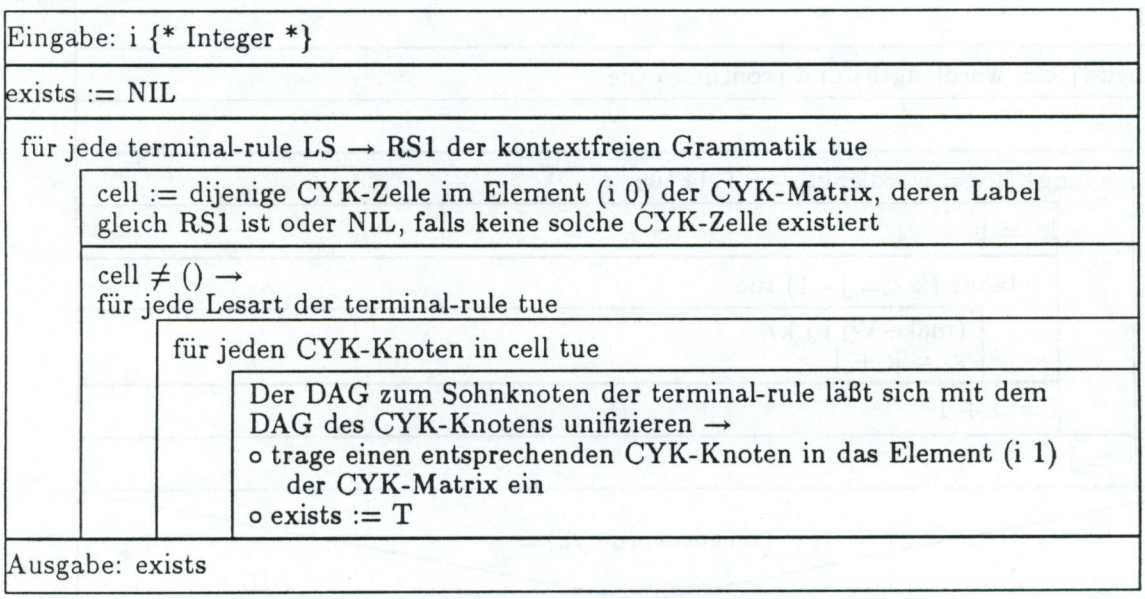



Abbildung 4.77: Struktogramm zur Funktion make-Vi1

**make-Vij**

Eingabe: i, j, k {* Integer *}

für jede branching-rule LS → RS1 RS2 der kontextfreien Grammatik tue

cell-1 := diejenige CYK-Zelle im Element (i k) der CYK-Matrix, deren Label gleich RS1 ist oder NIL, falls keine solche CYK-Zelle existiert
cell-2 := diejenige CYK-Zelle im Element ((i+k) (j-k)) der CYK-Matrix, deren Label gleich RS2 ist oder NIL, falls keine solche CYK-Zelle existiert

cell-1 ≠ () und cell-2 ≠ () →
für jede Lesart der branching-rule tue

für alle CYK-Knoten aus cell-1, die entweder die Nummer des linken Sohns haben oder als potentielle Adjunktionen im linken Sohn oder als Fußknoten in Frage kommen tue

Der DAG zum linken Sohn der branching-rule läßt sich mit dem DAG des CYK-Knotens aus cell-1 unifizieren → für alle CYK-Knoten aus cell-2, die entweder die Nummer des rechten Sohns haben oder als potentielle Adj. im rechten Sohn oder als Fußknoten in Frage kommen

Der DAG zum rechten Sohn der branching-rule läßt sich mit dem DAG des CYK-Knotens aus cell-2 unifizieren →
trage einen entsprechenden CYK-Knoten in das Element (i j) der CYK-Matrix ein

Abbildung 4.78: Struktogramm zur Funktion make-Vij

### Handhabung

#### Parameter

Der Aufruf des Moduls erfolgt ohne Parameter (→ Funktion *cyk*). Um eine korrekte Arbeitsweise zu garantieren, muß an die globale Variable *cfgram* der kontextfreie Kern einer TAG und an die globale Variable *word* die Eingabe in Form einer Liste von Symbolen gebunden sein.

### Ausgabe

Der Return der Funktion *cyk* ist T oder NIL, je nachdem, ob der erweiterte CYK-Parser die Eingabe akzeptiert oder nicht. Im Fall des Nicht-Akzeptierens der Eingabe wird zusätzlich noch eine der drei folgenden Nachrichten ausgegeben, die sich von selbst erklären:

1. "*terminal* is not in the lexicon"
   "Execution of the CYK-Parser aborted at level 0",
2. "There is no 'trule' to derive *termial*"
   "Execution of the CYK-Parser aborted at level 1",
3. "*Eingabe* is not in the language defined by 'cfgram'".

Wurde die Eingabe erfolgreich geparst, so bildet die an die globle Variable *cykhash* gebundene CYK-Matrix und die mit der "Menge aktiver Bäume" initialisierte globale Variable *SAT0* die Ausgangsbasis weiterer Analysen, wie sie im nachfolgenden Abschnitt beschrieben werden.

#### Fehlermeldungen

Als Fehlermeldungen können nur solche des LISP-Systems auftreten, wenn die im Abschnitt 'Parameter' geforderten Voraussetzungen nicht erfüllt wurden.

### 4.3.3 Iteration zur Elimination innerster Bäume

#### Erweiterungen der Definition

Im Gegensatz zu der exakten Version des Algorithmus aus [3] kommt die Implementierung des Moduls mit zwei "Mengen aktiver Bäume" aus. Die dritte Menge könnte auch in der exakten Version entfallen, da sie lediglich die Funktion eines Zwischenspeichers übernimmt. In allen übrigen Punkten folgt die Implementierung streng dem Algorithmus.

#### Datenstrukturen

Zusätzlich zu den im vorangehenden Abschnitt eingeführten Datenstrukturen und globalen Variablen, die komlett übernommen werden, werden in diesem Abschnitt keine mehr eingeführt.

#### Operatoren

**eit:** stellt die Schnittstelle des Moduls dar und realisiert durch Aufruf der nachfolgenden Operatoren die Iteration zum Auffinden innerster Bäume.

**SAT0-to-SAT1:** bewirkt den Übergang von der "Menge aktiver Bäume" *SAT0* zur "Menge aktiver Bäume" *SAT1.* Der Übergang entspricht der Elimination innerster Bäume.

**SAT1-to-SAT0:** bewirkt den Übergang von der "Menge aktiver Bäume" *SAT1* zur "Menge aktiver Bäume" *SAT0.* Der Übergang entspricht der Orientierung um ein elementares Kantenpaar nach oben, ausgehend von der Wurzel eines aktiven Baumes.

**accept-word-eit:** fällt die Entscheidung über Akzeptieren oder Nicht-Akzeptieren der Eingabe gemäß der TAG-Analyse. Eine Eingabe wird akzeptiert, falls nach Abschluß der Iteration zum Finden aller innersten Bäume im Element (1 n) (n = Länge der Eingabe) der CYK-Matrix mindestens ein CYK-Knoten existiert, der die Eigenschaft "TI" besitzt, also Wurzel eines kompletten initialen Baumes ist.

### Algorithmus

In den Abschnitten 3.4 und 3.5 wurde bereits auf die Grundstrukturen des Algorithmus eingegangen. Die Realisierung der oben eingeführten Operatoren ist den nachfolgenden Abbildungen 4.79 - 4.81 zu entnehmen.

Der zugehörige LISP-Code findet sich in der Datei "eit.l".

**eit**

| |
|---|
| counter := 1 |
| solange (counter < word-length) und in SAT0 noch "aktive Bäume" enthalten sind tue |
| (SAT0-to-SAT1) |
| (SAT1-to-SAT0) |
| counter := counter + 1 |
| Ausgabe: (accept-word-eit) |

Abbildung 4.79: Struktogramm zur Funktion eit

**SAT0-to-SAT1**

| |
|---|
| übertrage aus SAT0 alle CYK-Knoten mit Subtree LA und LI nach SAT1 |
| für alle CYK-Knoten aus SAT0 mit Subtree TA tue |
| für alle Fußknoten des CYK-Knotens tue |
| kreiere einen neuen CYK-Knoten, der alle Informationen des Fußknotens enthält außer: :adjoining = Pointer auf den TA-Knoten, :dag = DAG des TA-Knotens |
| trage den neuen CYK-Knoten in der CYK-Matrix in die CYK-Zelle des TA-Knotens ein und modifiziere die Informationen über ein- und ausgehende Kanten entsprechend der neuen Situation |
| hänge den neuen CYK-Knoten in SAT1 ein |
| lösche SAT0 |

Abbildung 4.80: Struktogramm zu SAT0-to-SAT1

**SAT1-to-SAT0**

übertrage aus SAT1 alle CYK-Knoten mit Subtree TA nach SAT0

für alle CYK-Knoten aus SAT1 mit lokal fertigen Unterbäumen (L-Knoten) tue

für alle in den L-Knoten eingehenden Kanten tue

from-node := der Ursprung (CYK-Knoten) einer solchen Kante

from-node ist Vater des L-Knotens gemäß der Nummernnotation und der Bruder des L-Knotens ist Wurzel eines vollständigen Teilbaums →
- kreiere einen neuen CYK-Knoten, der alle Informationen von from-node enthält außer: :subtree, :footp, :LP, :RP werden gemäß der neuen Situation berechnet
- trage den neuen CYK-Knoten in der CYK-Matrix in die CYK-Zelle von from-node ein und modifiziere die Informationen über ein- und ausgehende Kanten entsprechend der neuen Situation

hänge den neuen CYK-Knoten in SAT0 ein

lösche SAT1

Abbildung 4.81: Struktogramm zu SAT1-to-SAT0

### Handhabung

#### Parameter

Der Aufruf des Moduls erfolgt ohne Parameter (→ Funktion *eit*). Um eine korrekte Arbeitsweise zu garantieren, muß für eine Eingabe eine komplette CYK-Analyse mit Erfolg durchgeführt worden sein.

#### Ausgabe

Der Return der Funktion *eit* ist T oder NIL, je nachdem, ob die Eingabe in der durch die TAG definierten Sprache liegt oder nicht. Im Fall des Nicht-Akzeptierens wird zusätzlich noch die folgende Nachricht ausgegeben, die sich selbst erklärt:

"*Eingabe* is not in the language defined by the TAG 'grammar'",
"$x$ steps of the iteration of the EIT-algorithm have been made".

Eine Darstellung von Ableitungsbäumen nach erfolgreicher TAG-Analyse erfolgt nicht. Allerdings sind in der an die globale Variable *grammar* gebundenen CYK-Matrix alle Voraussetzungen geschaffen, die in der Matrix kodierten TAG-Ableitungsbäume weiter aufzuarbeiten.

#### Fehlermeldungen

Als Fehlermeldungen können nur solche des LISP-Systems auftreten, falls die im Abschnitt 'Parameter' geforderten Voraussetzungen nicht erfüllt wurden.

## 4.4 Interne Hilfsfunktionen

Der Abschnitt stellt eine Reihe von Funktionen zur Verfügung, die eine Darstellung elementarer Daten und Zwischenergebnisse ermöglichen, die bei der Abarbeitung des Algorithmus anfallen, nach außen jedoch nicht sichtbar werden.

### Erweiterungen der Definition

Die einzelnen Display-Funktionen sind kein Bestandteil des Algorithmus. Vielmehr entstanden sie während der Implementierung als Debugging-Werkzeuge, um die Programme auf ihre Korrektheit hin untersuchen zu können. Für jemanden, der an Erweiterungen oder Veränderungen des Kodes arbeitet, können die Funktionen dieses Moduls eine wertvolle Hilfe darstellen.

Weiterhin stellen die Display-Funktionen bis jetzt die einzige Möglichkeit dar, die Arbeitsweise des Programms anschaulich zu machen. Eine Darstellung der Ergebnisse von Grammatiktransformation und Strukturanalyse war nicht Bestandteil des Fortgeschrittenenpraktikums.

### Datenstrukturen

Aus den Abschnitten 4.3.1, 4.3.2, 4.3.3 werden alle globalen Variablen und dort definierten Strukturen verwendet.

### Operatoren

**display-grammar:** stellt die an die globale Variable *grammar* gebundene TAG mit all ihren elementaren Bäumen dar.

**display-nodehash:** stellt den gesamten Inhalt des Hashtables *nodehash* dar, also alle Knoten elementarer Bäume der TAG.

**display-cfgram:** stellt den an die globale Variable *cfgram* gebundenen kontextfreien Kern der TAG dar.

**display-cykhash:** stellt alle CYK-Knoten einer bestimmten Ebene der CYK-Matrix, die im Hashtable *cykhash* kodiert ist, dar.

**display-cyknodehash:** stellt den gesamten Inhalt des Hashtables *cyknodehash* dar, also alle CYK-Knoten der CYK-Matrix.

**display-sat:** stellt eine bestimmte Menge aktiver Bäume (Strukturen des Typs *sat*) dar.

### Algorithmus

Fast alle Display-Funktionen arbeiten komplexe Strukturen durch rekursiven Aufruf ab und führen dann ein "Pretty-Print" auf dem gewählten Output-Stream durch.

Der zugehörige LISP-Code findet sich in der Datei "display.l".

### Handhabung

### Parameter

Allen Display-Funktionen gemeinsam ist die Möglichkeit, die Ausgabe durch einen optionalen Parameter im Funktionsaufruf wahlweise auf den Bildschirm oder auf eine Datei zu leiten. Die Bildschirmausgabe stellt den Default-Fall dar. Da die erzeugten Daten oftmals sehr umfangreich sind und die Bildschirmausgabe vergleichweise langsam arbeitet, empfiehlt sich in den meisten Fällen die Ausgabe auf eine Datei. Lediglich die Funktionen *display-cykhash* und *display-sat* besitzen einen festen Parameter. Im ersten Fall wird ein Integer für die darzustellende Ebene der CYK-Matrix gefordert, im zweiten Fall eine Struktur des Typs *sat*. Um eine korrekte Arbeitsweise der einzelnen Funktionen zu garantieren, müssen im wesentlichen die oben aufgeführten globalen Variablen ihrem Sinn entsprechend gebunden sein.

### Ausgabe

Dem Return der Display-Funktionen kommt keine Bedeutung zu (weiteres siehe oben unter 'Parameter'). Die dargestellten Daten erklären sich ansonsten von selbst.

### Fehlermeldungen

An Fehlermeldungen können nur solche des LISP-Systems auftreten, falls die im Abschnitt 'Parameter' geforderten Voraussetzungen nicht erfüllt wurden. Weiter reagiert das LISP-System mit einer entsprechenden Fehlermeldung, wenn der für die Ausgabedatei gewählte Dateiname bereits in der $HOME-Directory vorhanden ist.

# Kapitel 5

# Bewertung

Durch die sehr genau umrissene Aufgabenstellung blieb nur wenig Spielraum bei der Implementierung. Das heißt, die einzelnen Moduln folgen mehr oder weniger streng den Vorgaben des Harbusch-Algorithmus [3] und des SB-PATR-Moduls [1]. Würde man, weg von der Unifkation des SB-PATR-Moduls, andere Formen der Unifikation verwenden, wären in diesem Zusammenhang effizientere Datenstrukturen und Werkzeuge denkbar, die sich positiv auf die Effizienz des Parsingalgorithmus auswirken würden. Gleiche Überlegungen lassen sich für das verwendete Lexikon anstellen. Allerdings muß bei der Realisierung der Lexikonschnittstelle immer der Form der eingesetzten Unifikation Rechnung getragen werden.

Abgesehen von der Unifkation bietet der jetzige Stand der Implementierung kaum noch Ansatzpunkte zur Effizienzsteigerung. Graduelle Verbesserungen ließen sich unter Umständen durch noch konsequenteres Hashen erzielen.

## 5.1 Vergleich mit dem Ansatz von Vijay

Der in diesem Fortgeschrittenenpraktikum eingeführten Definition von "TAGs mit Unifikation" ging bereits ein Ansatz von Vijay-Shanker und Joshi [12] voraus. Eine informale Beschreibung dieses Ansatzes ist in Kapitel 2.2.1 gegeben.

Ein Vorteil des Ansatzes von Vijay-Shanker und Joshi gegenüber dem von uns gewählten Verfahren ist sicher der geringere Rechenaufwand. Die Berechnung der Vererbungsrichtungen im zu adjungierenden auxiliaren Baum entfällt. Weiter ist eine Neuberechnung der DAGs vom Adjunktionsknoten aus aufwärts im Baum nicht notwendig (dem letzten Punkt sollte man nicht zu viel Bedeutung beimessen, da dieses Problem nicht auf Seiten der Definition sondern ausschließlich bei der Implementierung zum Tragen kommt. Effizientere Datenstrukturen und Werkzeuge für DAGs und Unifikation könnten hier sicher Abhilfe schaffen).

Ein Hauptnachteil des Ansatzes von Vijay-Shanker und Joshi ist darin zu sehen, daß zum Zeitpunkt des Adjungierens noch keine Aussage darüber gemacht werden kann, ob die Adjunktion auch korrekt im Sinne der Unifikation ist. Dies stellt sich erst heraus, wenn alle Adjunktionen (ohne Unifikation) durchgeführt worden sind und der den Ableitungsbaum

überspannende DAG berechnet wird.

Der bis hierher vorgenommene Vergleich der beiden Ansätze bezog sich lediglich auf die Definition des Adjungierens für "TAGs mit Unifikation" und somit auf die Generierung von Sprache. Die Implementierung des Parsers im Rahmen des Fortgeschrittenenpraktikums hat jedoch gezeigt, daß sich Unifikation beim Parsen an der Nahtstellen zweier Bäume vergleichsweise einfach und losgelöst von der aufwendigen Definition des Adjungierens realisieren läßt. Dies läßt vermuten, daß sich die Parsingalgorithmen zu unserem Ansatz und dem von Vijay-Shanker und Joshi in punkto Unifikation und DAGs kaum unterscheiden.

## 5.2 Laufzeitanalyse

Die Laufzeituntersuchungen wurden in der bereits in Abschnitt ... beschriebenen Umgebung an compiliertem LISP-Code vorgenommen. Die im folgenden aufgeführten Daten wurden mir Hilfe des in COMMON-LISP zur Verfügung stehenden Macros *time* gewonnen.

Laut Steele [11] sind Art und Erscheinungsbild der vom *time*-Macro ausgegebenen Daten "implementation-dependant". Die von uns eingesetzte COMMON-LISP-Version führt als Ausgabe von *time* neben dem eigentlichen Ergebnis der untersuchten LISP-Form eine "Real-Time" und eine "Run-Time" auf. Die COMMON-LISP-Funktionen *get-internal-run-time* und *get-internal-real-time* bilden dabei die Grundlage der Berechnungen. Eine weitere Basisgröße stellt die rechnerinterne Zeiteinheit dar, über die die COMMON-LISP-Konstante *internal-time-units-per-second* Auskunft gibt und die in der von uns eingesetzten Version den Wert 50 besitzt. Die Laufzeit wird also nur mit einer Genauigkeit von 0.02 Sekunden angegeben. Das bedeutet, daß Laufzeituntersuchungen mit Hilfe von *time* nur für größere Programmenheiten sinnvoll sind, deren Laufzeit deutlich über 0.02 Sekunden liegt. Auf die Programmeinheiten, deren Laufzeit nachfolgend untersucht wird, wurde bereits in Kapitel 4 eingegangen.

### 5.2.1 Transformation in Normalform

Für die Transformation in unsere Normalform haben wir einige Beispiele aus den Bereichen Adjunktion, Umwandlung in 2-Form, Umwandlung in eine $\epsilon$-freie Grammatik (ohne Ketten) und Umwandlung in eine kettenfreie Grammatik auf ihre Laufzeit untersucht. Um Anhaltspunkte für die Auswirkungen der DAG-Behandlung auf die Laufzeit zu bekommen, wurden die Beispiele mit (linkes Ergebnis) und "ohne" (rechtes Ergebnis) DAGs, d.h ohne Angabe von Regeln an den Knoten der Ausgangsbäume, laufengelassen.

**Beispiel 1:**

Bäume "ohne" DAGs werden in allen Beispielen um eine ähnliche Zeit schneller adjungiert, als Bäume mit Regeln an den Knoten. In den beiden vorgestellten Fällen macht der Unterschied etwa 30 bzw. 40 ms aus. Er ist verglichen mit der Gesamtlaufzeit gering, da

in Wirklichkeit die grundsätzlichen DAGs und Verbindungen sehr wohl vorhanden sind und nur die Aufteilung der Regel in Ups und Downs und die Suche der Vererbungsknoten im auxiliaren Baum wegfallen.

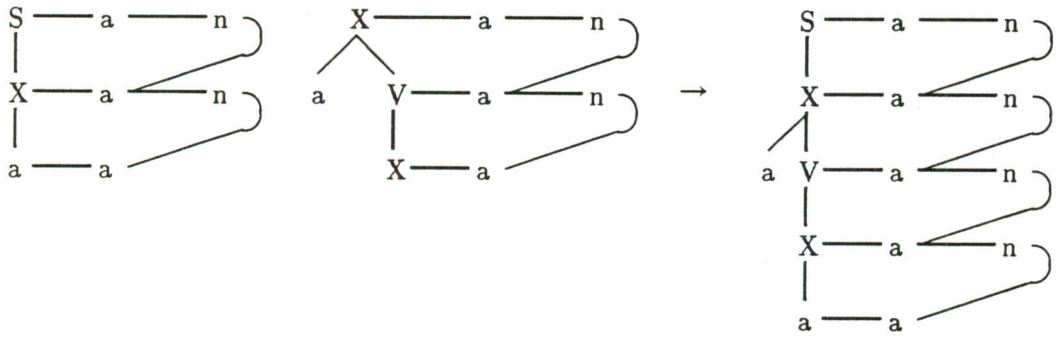



Abbildung 5.1: Laufzeitbeispiel 1 der Adjunktion

Laufzeit zu Abbildung 5.1:

```
(time (adjoinfu tree1 node tree2))
Real time:    1.76 s                Real time:    1.48 s
Run time:     1.76 s                Run time:     1.48 s
```

Laufzeit zu Abbildung 5.2:

```
(time (adjoinfu tree1 node tree2))
Real time:    3.58 s                  Real time:    3.18 s
Run time:     3.56 s                  Run time:     3.16 s
```

**Beispiel 2:**

Wie im ersten Beispiel werden hier die Laufzeiten mit und "ohne" DAGs gegenübergestellt. Der Baum stellt eine Schicht des Transformationsproblems dar (einen Vater mit mehr als zwei Söhnen, bei deren Zusammenfassung in Paare über mehrere Ebenen neue Zwischenknoten entstehen).
Laufzeit zu Abbildung 5.3:

```
(time (grammar-to-2-form))
Real time:    0.12 s                  Real time:    0.12 s
Run time:     0.12 s                  Run time:     0.12 s
T
```

**Beispiel 3:**

Für die Berechnung einer $\epsilon$-freien Grammatik werden die Unterschiede für Laufzeiten mit und "ohne" DAGs durch die vorkommenden Adjunktionen ausgemacht (wie in Beispiel 1 gesehen), weshalb hier der Einfachheit halber auf Regeln an Knoten verzichtet wurde.
Laufzeit zu Abbildung 5.4:

```
(time (grammar-to-e-free))
Real time:     4.82 s
Run time:      4.82 s
```

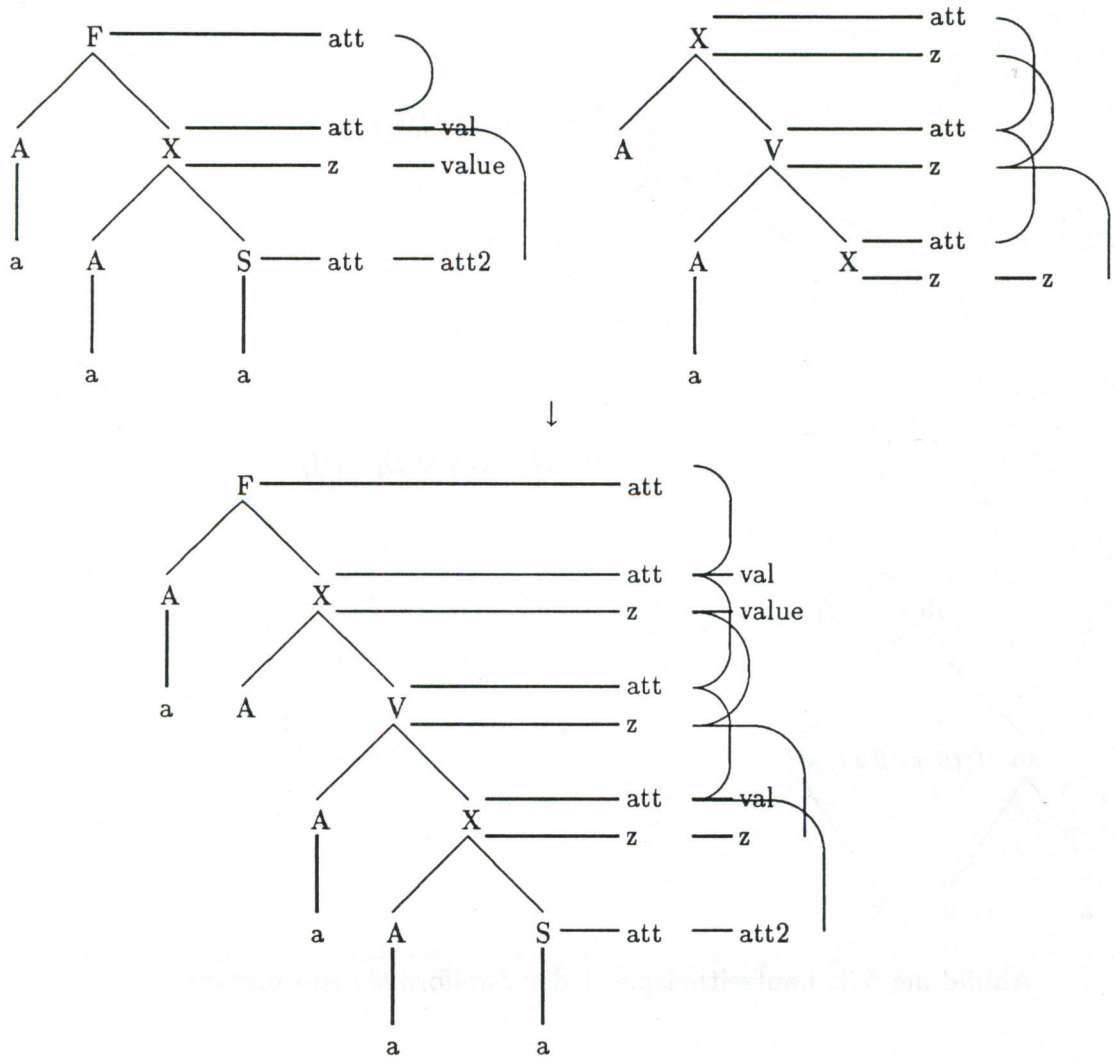



Abbildung 5.2: Laufzeitbeispiel 2 der Adjunktion

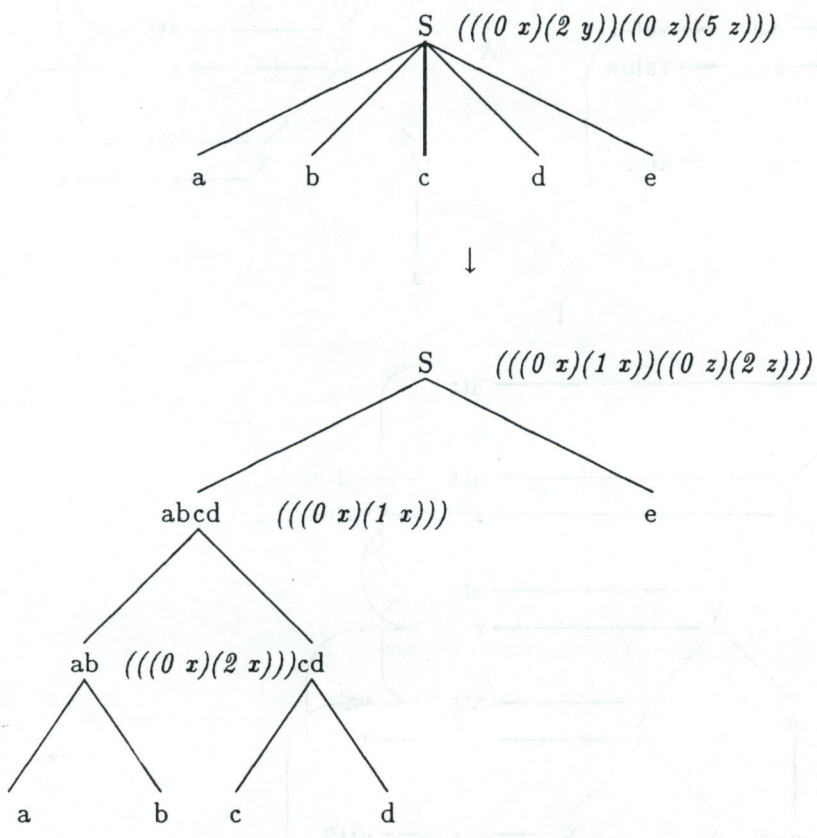



Abbildung 5.3: Laufzeitbeispiel 1 der Zweiformtransformation

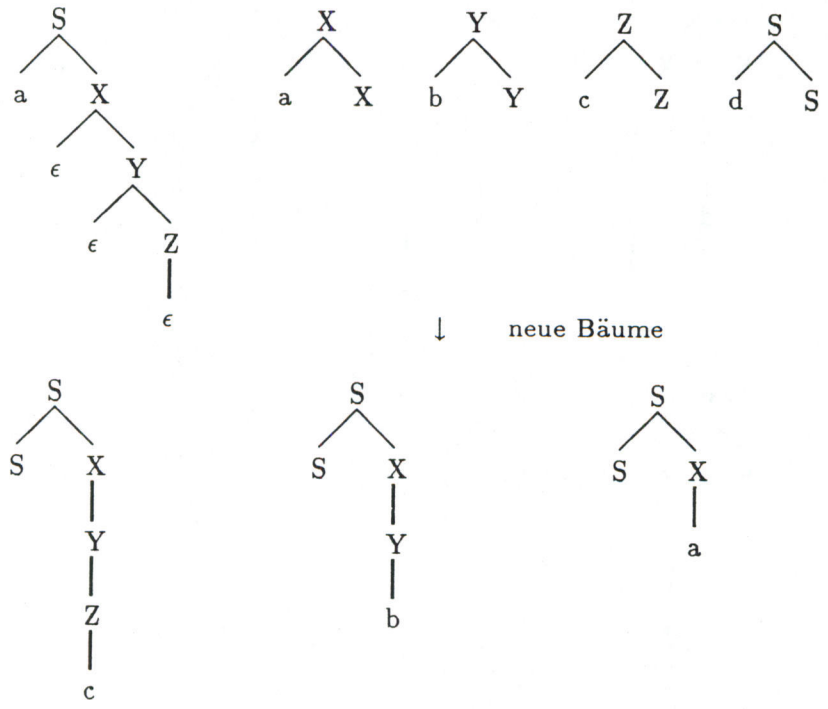



Abbildung 5.4: Laufzeitbeispiel 1 der Epsilonfreiheit

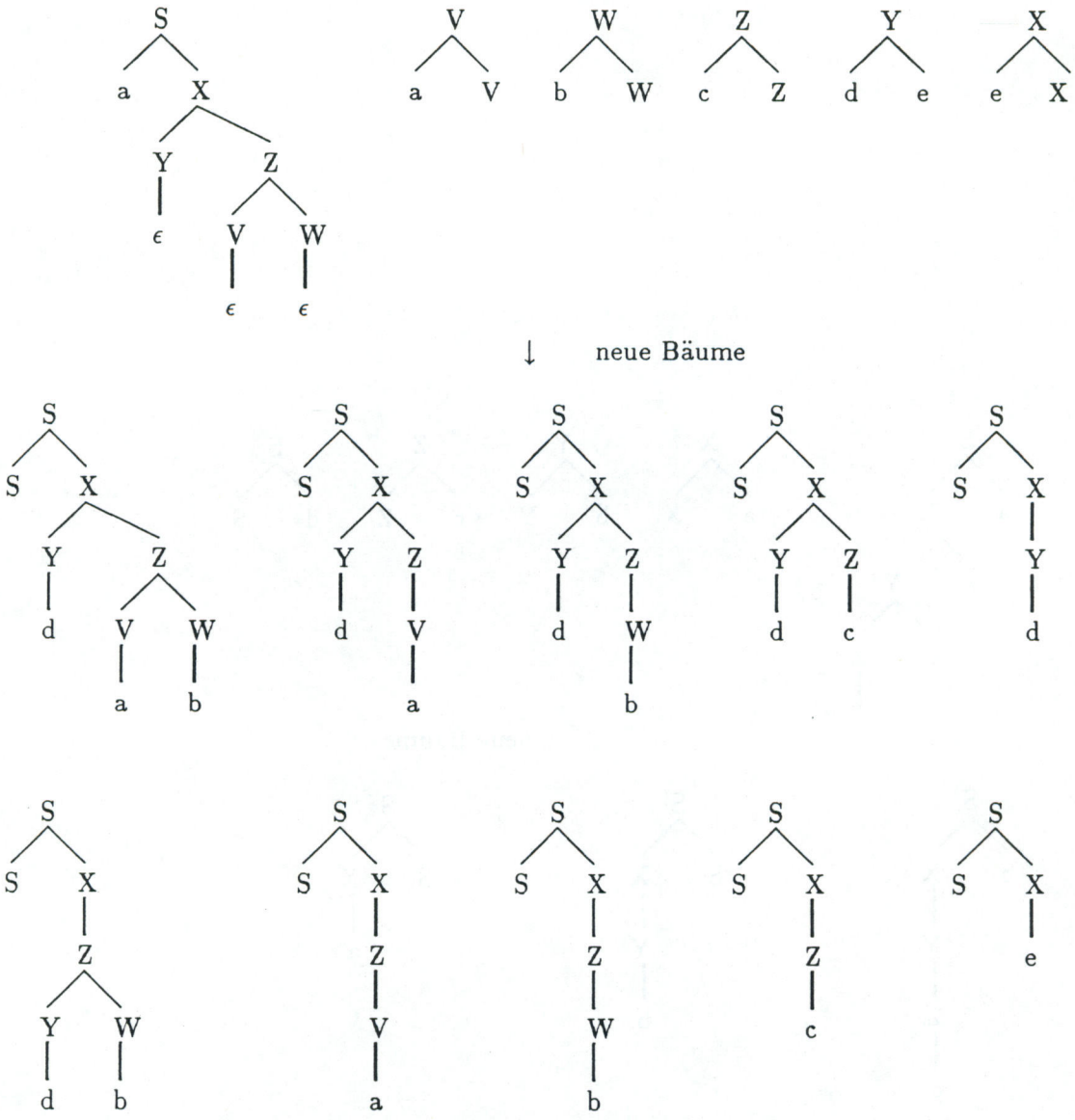



Abbildung 5.5: Laufzeitbeispiel 2 der Epsilonfreiheit

Laufzeit zu Abbildung 5.5:

```
(time (grammar-to-e-free))
Real time:    41.62 s
Run time:     41.32 s
T
```

**Beispiel 4 :**

Für die Elimination von Ketten ergeben sich die relativ gesehen geringen Laufzeitunterschiede mit und "ohne" DAGs wie in Beispiel 1 aus der Anzahl der durchgeführten

Adjunktionen. Die in dieses Modul integrierte Einführung von Hilfsknoten für die einfach zu korrigierenden Vater-Sohn-Beziehungen in Zwei-Form trägt ebenfalls nicht gravierend zur Laufzeit bei. Es ist klar, daß abgesehen davon die Laufzeit entscheidend von der Anzahl der neu erzeugten Bäume abhängt. Diese wiederum hängt exponentiell von der Anzahl der in den Knoten der Kette adjungierbaren Bäume bzw. der daraus resultierenden Anzahl neu erzeugter Bäume ab, wodurch die relativ hohe Laufzeit zu erklären ist.

Sei G eine Grammatik bestehend aus den Bäumen der Abbildung 5.6 (der Einfachheit halber "ohne" DAGs).

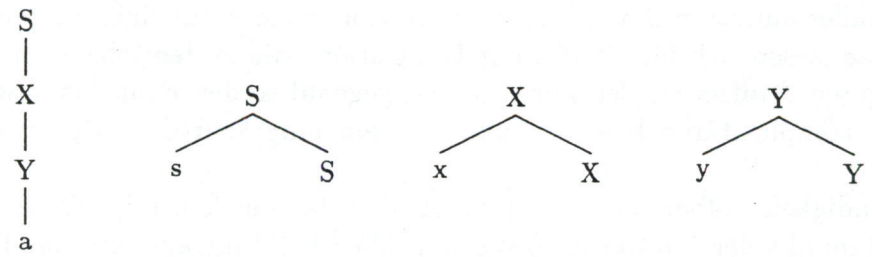



Abbildung 5.6: Laufzeitbeispiel der Kettenelimination

Laufzeit zu Abbildung 5.6 (es wurden neun neue Bäume erzeugt) :

```
(time (grammar-to-chain-free))
Real Time:    44.64 s                Real Time: 36.60 s
Run Time :    44.24 s                Run Time : 35.92 s
```

Erweitert man die Grammatik aus Abbildung 5.6 um je einen auxiliaren Baum für jeden Knoten der Kette (also insgesamt drei neue Bäume), so ergeben sich folgende Laufzeiten (es wurden 30 neue Bäume erzeugt) :

```
Real Time:    172.90 s               Real Time: 152.60 s
Run Time :    171.28 s               Run Time : 151.88 s
```

Diese Werte verdeutlichen eindrucksvoll, wie stark die Laufzeit von der Anzahl der in den Knoten der Kette adjungierbaren Bäume abhängt.

### 5.2.2 Parsing

Die für die Laufzeitbetrachtung des Parsens gewählten Programmeinheiten sind:

1. Berechnung des kontextfreien Kerns der TAG (→ Funktion *catch-cfg*)
2. CYK-Analyse des kontextfreien Kerns der TAG (→ Funktion *cyk*)
3. Iteration zum Auffinden aller innersten Bäume (→ Funktion *eit*)

Zusätzlich wurde für die Untersuchungen ein Parser für "TAGs ohne Unifikation" implementiert, der gleichfalls auf dem Harbusch-Algorithmus [3] aufbaut und die in Kapitel 4 beschriebenen Schnittstellen besitzt. Der Algorithmus verzichtet jedoch ganz auf Unifikation und verwendet konsequenterweise ein anderes Lexikon.

Aus der Gegenüberstellung der Laufzeiten der beiden Parser läßt sich ungefähr ablesen, wieviel Zeit beim Parsen von "TAGs mit Unifikation" auf die Unifikation selbst und die mit ihr verbundene Verwaltung der DAGs entfällt. Dies gilt insbesondere für die Funktion *cyk*. Die Laufzeiten der Funktion *eit* weisen in beiden Parsern hingegen nur graduelle Unterschiede auf, da beim Parsen von "TAGs mit Unifikation" während der Iteration zum Auffinden aller innersten Bäume keine Unifikationen mehr durchgeführt werden. Auf die gleiche Weise lassen sich für "TAGs mit Unifikation" die in den folgenden Beispielen deutlich geringeren Laufzeiten der Funktion *eit* gegenüber der Funktion *cyk* erklären, obwohl die Zeitkomplexitäten beider Funktionen ein umgekehrtes Verhältnis erwarten ließen.

Der Vollständigkeit halber sei noch erwähnt, daß das Lexikon für "TAGs mit Unifikaion" zum Zeitpunkt der Laufzeituntersuchung über 111 Einträge und das für "TAGs ohne Unifikation" über 20 Einträge verfügte. In den Beispielen wurde darauf geachtet, daß jedes eingesetzte Terminal in beiden Lexika die gleiche Anzahl Lesarten aufweist, was für die Vergleichbarkeit der Ergebnisse weitaus wichtiger ist, als die Anzahl der Lexikoneinträge selbst.

**Beispiel 1:**

Die folgende Grammatik stellt eine Simulation der Sprache $a^n b^n$ mit den Terminalen *in* und *an* dar. Beide Terminale besitzen je 2 Lesarten. Geparst wird der Terminalstring $a^5 b^5$.

Sind 2 Laufzeiten nebeneinander angegeben, so bezieht sich, wie in den nachfolgenden Beispielen auch, die linke immer auf den Parser für "TAGs mit Unifikation" und die rechte auf den Parser für "TAGs ohne Unifikation".

```
(Inprep Anprep)
(S H1 H2 H3 H4 H5)
(S (H1 (H2 (Inprep) S)
    H3 (Anprep)))
(S (H4 (Inprep)
    H5 (Anprep)))
E
(setq word '(in in in in in an an an an an))

(time (catch-cfg))
Real time:    0.10 s
Run time:     0.08 s
T
```

```
(time (cyk))
Real time:    41.64 s        Real time:     0.30 s
Run time:     40.50 s        Run time:      0.30 s
T

(time (eit))
Real time:     0.30 s        Real time:     0.20 s
Run time:      0.30 s        Run time:      0.20 s
T
```

**Beispiel 2:**

Die Unifikationsregeln der Grammatik stellen einen Zähler dar, der über die Anzahl der Adjunktionen der beiden auxiliaren Bäume Buch führt. Für den Parser für "TAGs ohne Unifikation" sind die Unifikationsregeln unbedeutend, sie werden überlesen.

```
(Adverb)
(A H1 H2)
(A ((((0 sc next) 0)
     ((0 nsc) 0)))
   (Adverb))
(A ((((0 sc) (2 sc))
     ((0 nsc next) (2 nsc))))
   (H1 (Adverb)
    A))
(A ((((0 sc next) (2 sc))
     ((0 nsc) (2 nsc))))
   (H2 (Adverb)
    A))
E
(setq word '(da da da da))

(time (catch-cfg))
Real time:     0.08 s
Run time:      0.06 s
T

(time (cyk))
Real time:    30.66 s        Real time:     0.18 s
Run time:     30.42 s        Run time:      0.18 s
T

(time (eit))
Real time:     1.40 s        Real time:     1.60 s
```

```
Run time:     1.40 s      Run time:     1.56 s
T
```

**Beispiel 3:**

Die Grammatik stellt ein Worst-Case-Beispiel dar, mit einer exponentiellen Zahl von Ableitungsbäumen für den gegebnen Terminal-String.

```
(Adverb)
(S)
(S (Adverb))
(S (S (Adverb)
    S))
(S (S
    S (Adverb)))
E
(setq word '(da da da))

(time (catch-cfg))
Real time:    0.06 s
Run time:     0.06 s
T

(time (cyk))
Real time:   30.40 s      Real time:    0.16 s
Run time:    30.30 s      Run time:     0.16 s
T

(time (eit))
Real time:    4.70 s      Real time:    1.10 s
Run time:     4.66 s      Run time:     1.10 s
T
```

Die folgenden Laufzeiten beziehen sich auf die gleiche Grammatik. Lediglich die Länge des Terminalstrings wurde von 3 auf 4 erhöht.

```
(setq word '(da da da da))

(time (cyk))
Real time:   84.44 s      Real time:    0.52 s
Run time:    83.30 s      Run time:     0.52 s
T

(time (eit))
Real time:   21.12 s      Real time:   25.36 s
```

```
Run time:     20.86 s       Run time:     25.04 s
T
```

## 5.3 Erweiterungsmöglichkeiten

Die Möglichkeiten der Verwendung anderer Formen der Unifikation als die durch den SB-PATR-Modul [1] vorgegebene ist in vorangegangenen Abschnitten schon wiederholt angesprochen worden. Derzeit wird versucht, eine sogenannte *verteilte Unifikation* zu implementieren, zu deren Definition es gehört, die einzelnen DAGs getrennt zu erhalten und ihre Verbindung durch explizite *Links* zu repräsentieren. Der Nachteil dieser Repräsentationsform ist, daß beim Lesen von Werten stets eine Suche durch alle verbundenen DAGs erfolgen muß, der Vorteil ist, daß ein destruktives Durchtrennen der Verbindungen – wie es für die Realisierung von Adjunktion mit Unifikation benötigt wird – sehr leicht möglich ist. Die praktische Verwendbarkeit dieses Ansatzes bleibt noch zu untersuchen.

Eine weiter Möglichkeit der Erweiterung ist in einer Werkbank für "TAGs mit Unifikation" zu sehen, deren Imlementierung ein Bestandteil der Diplomarbeit von Béla Buschauer sein wird. Ein erster Arbeitsschritt bei der Implementierung der Werkbank ist die Entwicklung eines Algorithmus zur Rücktransformation von TAG-Ableitungbäumen gemäß der ursprünglich vor der Normalformtransformation definierten Grammatik.

Als zusätzliche Schwerpunkte der Erweiterung seien noch die folgenden drei Stichpunkte genannt:

- "Dicke DAGs",
- die bisher bei der Normalformtranformation anfallenden Constraints in gleichwertige Unifikationsregeln übersetzen,
- paralleles Parsing von TAGs u.a.

# Literatur


[1] W. Finkler, G. Neumann : *SB-PATR: Systembeschreibung und Benutzeranleitung*, Memo in der Reihe KI-Labor, Fachbereich Informatik, Universität des Saarlandes, Saarbrücken 1988

[2] G. Gazdar, E. Klein, G. Pullum, I. Sag : *General Phrase Structure Grammar* Basil Blackwell, Oxford, 1985

[3] K. Harbusch : *Effiziente Analyse natürlicher Sprache mit TAGs* , Proceedings des Symposiums über 'Computerlinguistik und ihre theoretischen Grundlagen', Saarbrücken, Springer-Verlag, Berlin, Reihe 'Informatik-Fachberichte', 1988

[4] J.E. Hopcroft, J.D. Ullman : *Introduction to Automatam Theory, Languages, and Computation*, Addison-Wesley Publishing, Inc., Reading, Massachusetts, 1979

[5] A.K. Joshi, S. Levy, M. Takahashi : *Tree Adjoining Grammars*, Journal of Computer Systems and Science 10, 136-163, 1975

[6] A.K. Joshi : *An Introduction to Tree Adjoining Grammars*, Technical Report MS-CIS-86-64, LINC-LAB 31, Department of Computer and Information Science, Moore School, University of Pennsylvania, Philadelphia, 1985

[7] L. Karttunen : *D-PATR: A Development Environment for Unification Grammars*, Proceedings of the 11th International Conference on Computational Linguistics, COLING-86, Bonn, 1986

[8] T. Kasami : *An Efficient Recognition and Syntax Algorithm for Context-Free Languages*, Scientific Report, Air Force Cambridge Research Lab., Bedford, Massachusetts, 1965

[9] T. Kroch, A.K. Joshi : *Linguistic Relevance of Tree Adjoining Grammars*, Technical Report MS-CIS-85-16, Department of Computer and Information Science, Moore School, University of Pennsylvania, Philadelphia, 1985

[10] S.M. Shieber : *An Introduction to Unification-Based Approaches to Grammar*, CSLI-Stanford, Lecture Notes Number 4, Stanford, 1987

[11] G. Steele : *Common-LISP: The Language*, Digital Press, Burlington, 1984

[12] K. Vijay-Shanker, A.K. Joshi : *Some Computational Properties of Tree Adjoining Grammars*, Proceeding of the 23rd Annual Meeting of the Association for Computational Linguistics (ACL'85), University of Chicago, Chicago, Illinois, 1985

[13] D. H. Younger : *Recognition and Parsing of Context-Free Languages in Time* $On^{(3)}$, Information and Control,10:2, 189-208, 1967

Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH

DFKI
-Bibliothek-
PF 2080
6750 Kaiserslautern
FRG

## DFKI Publikationen

Die folgenden DFKI Veröffentlichungen oder die aktuelle Liste von erhältlichen Publikationen können bezogen werden von der oben angegebenen Adresse.
Die Berichte werden, wenn nicht anders gekennzeichnet, kostenlos abgegeben.

## DFKI Publications

The following DFKI publications or the list of currently available publications can be ordered from the above address.
The reports are distributed free of charge except if otherwise indicated.

### DFKI Research Reports

**RR-90-01**
*Franz Baader*: Terminological Cycles in KL-ONE-based Knowledge Representation Languages
33 pages

**RR-90-02**
*Hans-Jürgen Bürckert*: A Resolution Principle for Clauses with Constraints
25 pages

**RR-90-03**
*Andreas Dengel, Nelson M. Mattos:* Integration of Document Representation, Processing and Management
18 pages

**RR-90-04**
*Bernhard Hollunder, Werner Nutt:* Subsumption Algorithms for Concept Languages
34 pages

**RR-90-05**
*Franz Baader:* A Formal Definition for the Expressive Power of Knowledge Representation Languages
22 pages

**RR-90-06**
*Bernhard Hollunder:* Hybrid Inferences in KL-ONE-based Knowledge Representation Systems
21 pages

**RR-90-07**
*Elisabeth André, Thomas Rist:* Wissensbasierte Informationspräsentation:
Zwei Beiträge zum Fachgespräch Graphik und KI:
1. Ein planbasierter Ansatz zur Synthese illustrierter Dokumente
2. Wissensbasierte Perspektivenwahl für die automatische Erzeugung von 3D-Objektdarstellungen

24 pages

**RR-90-08**
*Andreas Dengel:* A Step Towards Understanding Paper Documents
25 pages

**RR-90-09**
*Susanne Biundo:* Plan Generation Using a Method of Deductive Program Synthesis
17 pages

**RR-90-10**
*Franz Baader, Hans-Jürgen Bürckert, Bernhard Hollunder, Werner Nutt, Jörg H. Siekmann:* Concept Logics
26 pages

**RR-90-11**
*Elisabeth André, Thomas Rist:* Towards a Plan-Based Synthesis of Illustrated Documents
14 pages

**RR-90-12**
*Harold Boley:* Declarative Operations on Nets
43 pages

**RR-90-13**
*Franz Baader:* Augmenting Concept Languages by Transitive Closure of Roles: An Alternative to Terminological Cycles
40 pages

**RR-90-14**
*Franz Schmalhofer, Otto Kühn, Gabriele Schmidt:* Integrated Knowledge Acquisition from Text, Previously Solved Cases, and Expert Memories
20 pages

**RR-90-15**
*Harald Trost:* The Application of Two-level Morphology to Non-concatenative German Morphology
13 pages

**RR-90-16**
*Franz Baader, Werner Nutt:* Adding Homomorphisms to Commutative/Monoidal Theories, or: How Algebra Can Help in Equational Unification
25 pages

**RR-90-17**
*Stephan Busemann:* Generalisierte Phasenstrukturgrammatiken und ihre Verwendung zur maschinellen Sprachverarbeitung
114 Seiten

**RR-91-01**
*Franz Baader, Hans-Jürgen Bürckert, Bernhard Nebel, Werner Nutt, Gert Smolka:* On the Expressivity of Feature Logics with Negation, Functional Uncertainty, and Sort Equations
20 pages

**RR-91-02**
*Francesco Donini, Bernhard Hollunder, Maurizio Lenzerini, Alberto Marchetti Spaccamela, Daniele Nardi, Werner Nutt:* The Complexity of Existential Quantification in Concept Languages
22 pages

**RR-91-03**
*B.Hollunder, Franz Baader:* Qualifying Number Restrictions in Concept Languages
34 pages

**RR-91-04**
*Harald Trost:* X2MORF: A Morphological Component Based on Augmented Two-Level Morphology
19 pages

**RR-91-05**
*Wolfgang Wahlster, Elisabeth André, Winfried Graf, Thomas Rist:* Designing Illustrated Texts: How Language Production is Influenced by Graphics Generation.
17 pages

**RR-91-06**
*Elisabeth André, Thomas Rist:* Synthesizing Illustrated Documents A Plan-Based Approach
11 pages

**RR-91-07**
*Günter Neumann, Wolfgang Finkler:* A Head-Driven Approach to Incremental and Parallel Generation of Syntactic Structures
13 pages

**RR-91-08**
*Wolfgang Wahlster, Elisabeth André, Som Bandyopadhyay, Winfried Graf, Thomas Rist:* WIP: The Coordinated Generation of Multimodal Presentations from a Common Representation
23 pages

**RR-91-09**
*Hans-Jürgen Bürckert, Jürgen Müller, Achim Schupeta:* RATMAN and its Relation to Other Multi-Agent Testbeds
31 pages

**RR-91-10**
*Franz Baader, Philipp Hanschke:* A Scheme for Integrating Concrete Domains into Concept Languages
31 pages

**RR-91-11**
*Bernhard Nebel:* Belief Revision and Default Reasoning: Syntax-Based Approaches
37 pages

**RR-91-12**
*J.Mark Gawron, John Nerbonne, Stanley Peters:* The Absorption Principle and E-Type Anaphora
33 pages

**RR-91-13**
*Gert Smolka:* Residuation and Guarded Rules for Constraint Logic Programming
17 pages

**RR-91-14**
*Peter Breuer, Jürgen Müller:* A Two Level Representation for Spatial Relations, Part I
27 pages

**RR-91-15**
*Bernhard Nebel, Gert Smolka:* Attributive Description Formalisms ... and the Rest of the World
20 pages

**RR-91-16**
*Stephan Busemann:* Using Pattern-Action Rules for the Generation of GPSG Structures from Separate Semantic Representations
18 pages

**RR-91-17**
*Andreas Dengel & Nelson M. Mattos:* The Use of Abstraction Concepts for Representing and Structuring Documents
17 pages

**RR-91-19**
*Munindar P. Singh:* On the Commitments and Precommitments of Limited Agents
15 pages

**RR-91-20**
*Christoph Klauck, Ansgar Bernardi, Ralf Legleitner* FEAT-Rep: Representing Features in CAD/CAM
48 pages

**RR-91-22**
*Andreas Dengel:* Self-Adapting Structuring and Representation of Space
27 pages

**RR-91-23**
*Michael Richter, Ansgar Bernardi, Christoph Klauck, Ralf Legleitner:* Akquisition und Repräsentation von technischem Wissen für Planungsaufgaben im Bereich der Fertigungstechnik
24 Seiten

**RR-91-24**
*Jochen Heinsohn:* A Hybrid Approach for Modeling Uncertainty in Terminological Logics
22 pages

**RR-91-25**
*Karin Harbusch, Wolfgang Finkler, Anne Schauder:* Incremental Syntax Generation with Tree Adjoining Grammars
16 pages

**RR-91-26**
*M. Bauer, S. Biundo, D. Dengler, M. Hecking, J. Koehler, G. Merziger:* Integrated Plan Generation and Recognition
- A Logic-Based Approach -
17 pages

**RR-91-27**
*A. Bernardi, H. Boley, Ph. Hanschke, K. Hinkelmann, Ch. Klauck, O. Kühn, R. Legleitner, M. Meyer, M. M. Richter, F. Schmalhofer, G. Schmidt, W. Sommer:* ARC-TEC: Acquisition, Representation and Compilation of Technical Knowledge
18 pages

**RR-91-30**
*Dan Flickinger, John Nerbonne:* Inheritance and Complementation: A Case Study of *Easy* Adjectives and Related Nouns
39pages

---

**DFKI Technical Memos**

**TM-89-01**
*Susan Holbach-Weber:* Connectionist Models and Figurative Speech
27 pages

**TM-90-01**
*Som Bandyopadhyay:* Towards an Understanding of Coherence in Multimodal Discourse
18 pages

**TM-90-02**
*Jay C. Weber:* The Myth of Domain-Independent Persistence
18 pages

**TM-90-03**
*Franz Baader, Bernhard Hollunder:* KRIS: Knowledge Representation and Inference System -System Description-
15 pages

**TM-90-04**
*Franz Baader, Hans-Jürgen Bürckert, Jochen Heinsohn, Bernhard Hollunder, Jürgen Müller, Bernhard Nebel, Werner Nutt, Hans-Jürgen Profitlich:* Terminological Knowledge Representation: A Proposal for a Terminological Logic
7 pages

**TM-91-01**
*Jana Köhler:* Approaches to the Reuse of Plan Schemata in Planning Formalisms
52 pages

**TM-91-02**
*Knut Hinkelmann:* Bidirectional Reasoning of Horn Clause Programs: Transformation and Compilation
20 pages

**TM-91-03**
*Otto Kühn, Marc Linster, Gabriele Schmidt:* Clamping, COKAM, KADS, and OMOS: The Construction and Operationalization of a KADS Conceptual Model
20 pages

**TM-91-04**
*Harold Boley:* A sampler of Relational/Functional Definitions
12 pages

**TM-91-05**
*Jay C. Weber, Andreas Dengel , Rainer Bleisinger:* Theoretical Consideration of Goal Recognition Aspects for Understanding Information in Business Letters
10 pages

**TM-91-08**
*Munindar P. Singh:* Social and Psychological Commitments in Multiagent Systems
11 pages

**TM-91-09**
*Munindar P. Singh:* On the Semantics of Protocols Among Distributed Intelligent Agents
18 pages

**TM-91-10**
*Béla Buschauer, Peter Poller, Anne Schauder, Karin Harbusch:* Tree Adjoining Grammars mit Unifikation
149 pages

**TM-91-11**
*Peter Wazinski:* Generating Spatial Descriptions for Cross-modal References
21 pages

---

**DFKI Documents**

**D-89-01**
*Michael H. Malburg, Rainer Bleisinger:*
HYPERBIS: ein betriebliches Hypermedia-Informationssystem
43 Seiten

**D-90-01**
DFKI Wissenschaftlich-Technischer Jahresbericht 1989
45 pages

**D-90-02**
*Georg Seul:* Logisches Programmieren mit Feature -Typen
107 Seiten

**D-90-03**
*Ansgar Bernardi, Christoph Klauck, Ralf Legleitner:* Abschlußbericht des Arbeitspaketes PROD
36 Seiten

**D-90-04**
*Ansgar Bernardi, Christoph Klauck, Ralf Legleitner:* STEP: Überblick über eine zukünftige Schnittstelle zum Produktdatenaustausch
69 Seiten

**D-90-05**
*Ansgar Bernardi, Christoph Klauck, Ralf Legleitner:* Formalismus zur Repräsentation von Geo-metrie- und Technologieinformationen als Teil eines Wissensbasierten Produktmodells
66 Seiten

**D-90-06**
*Andreas Becker:* The Window Tool Kit
66 Seiten

**D-91-01**
*Werner Stein , Michael Sintek:* Relfun/X - An Experimental Prolog Implementation of Relfun
48 pages

**D-91-03**
*Harold Boley, Klaus Elsbernd, Hans-Günther Hein, Thomas Krause:* RFM Manual: Compiling RELFUN into the Relational/Functional Machine
43 pages

**D-91-04**
DFKI Wissenschaftlich-Technischer Jahresbericht 1990
93 Seiten

**D-91-06**
*Gerd Kamp:* Entwurf, vergleichende Beschreibung und Integration eines Arbeitsplanerstellungssystems für Drehteile
130 Seiten

**D-91-07**
*Ansgar Bernardi, Christoph Klauck, Ralf Legleitner*
TEC-REP: Repräsentation von Geometrie- und Technologieinformationen
70 Seiten

**D-91-08**
Thomas Krause: Globale Datenflußanalyse und horizontale Compilation der relational-funktionalen Sprache RELFUN
137 pages

**D-91-09**
*David Powers and Lary Reeker (Eds):*
Proceedings MLNLO'91 - Machine Learning of Natural Language and Ontology
211 pages
**Note:** This document is available only for a nominal charge of 25 DM (or 15 US-$).

**D-91-10**
*Donald R. Steiner, Jürgen Müller (Eds.)*
MAAMAW'91: Pre-Proceedings of the 3rd European Workshop on „Modeling Autonomous Agents and Multi-Agent Worlds"
246 pages
**Note:** This document is available only for a nominal charge of 25 DM (or 15 US-$).

**D-91-11**
*Thilo C. Horstmann:*Distributed Truth Maintenance
61 pages

**D-91-12**
*Bernd Bachmann:*
$H^{iera}C_{on}$ - a Knowledge Representation System with Typed Hierarchies and Constraints
75 pages

**D-91-13**
International Workshop on Terminological Logics
*Organizers: Bernhard Nebel, Christof Peltason, Kai von Luck*
131 pages

**D-91-14**
*Erich Achilles, Bernhard Hollunder, Armin Laux, Jörg-Peter Mohren: KRIS : Knowledge Representation and Inference System*
- Benutzerhandbuch -
28 Seiten

# Tree Adjoining Grammars mit Unifikation

Béla Buschauer, Peter Poller, Anne Schauder,Karin Harbusch

**TM-91-10**
Technical Memo