BMBF

Verbmobil

Verbundvorhaben

GLR-Parsing von Worthypothesengraphen

Steffen Staab

IMMD VIII/UER

Vm Report 85

Mai 1995

Mai 1995

Steffen Staab

IMMD VIII – Künstliche Intelligenz
Universität Erlangen-Nürnberg
Am Weichselgarten 9
D-91058 Erlangen

Tel.: 09131/8599 - 07
Fax: 09131/8599 - 05

Der Autor ist jetzt bei:
Universität Freiburg
Arbeitsgruppe Computerlinguistik/Linguistische Informatik
Platz der alten Synagoge 1
79085 Freiburg

Tel.: (0761)203-3357
Fax: (0761)203-3251
e-mail: `staab@coling.uni-freiburg.de`

Gehört zum Antragsabschnitt: 15.7: Architektur integrierter Parser für gesprochene Sprache

Die vorliegende Arbeit wurde im Rahmen des Verbundvorhabens Verbmobil vom Bundesministerium für Bildung, Wissenschaft, Forschung und Technologie (BMBF) unter dem Förderkennzeichen 01 IV 101 H9 gefördert. Die Verantwortung für den Inhalt dieser Arbeit liegt bei den Autoren.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

1.1 Problemstellung

Beim Erkennen gesprochener Sprache müssen eine Anzahl verschiedener Techniken angewandt werden, damit aus dem im Allgemeinen mehrdeutigen akustischen Signal die Bedeutung eines Satzes erkannt werden kann.
Daß das Eingangssignal nicht eindeutig ist, hat viele Ursachen:

- **Koartikulation:** Es gibt keine klaren Phonem-/Wort- und manchmal auch Satzgrenzen in der gesprochenen Sprache.
- **Sprecher** variieren in Tempo, Stimmlage, Akzent.
- **Ellipsen, Fragmente, Häsitationen, Reparaturen,** sind häufig in gesprochener Sprache.
- **Umgebungsgeräusche** erschweren die korrekte Analyse des akustischen Signals.
- **Übermittlungsprobleme:** z.B. durch Telefonleitungen wird das akustische Signal stark beeinträchtigt.

Aufgrund dieser Ursachen entsteht eine Vielzahl von Ambiguitäten, aus denen eine Analyse erstellt werden muß.

1.2 Lösungsansätze

Die prominentesten Ansätze zur Analyse eines Sprachsignals integrieren zwei Verarbeitungsstufen:

1. In der *Spracherkennung* werden statistische Methoden zur Mustererkennung (Decoding) eingesetzt.
2. Im Bereich des *Sprachverstehens* versucht man mittels symbolischer Sprachbeschreibungen, die beim Decoding erkannten Symbole zu verarbeiten.

Das Problem bei der Kopplung dieser beiden Verarbeitungsstufen besteht nun darin, daß eine vollständige Analyse der erkannten Symbole zu zeitaufwendig ist. Die effiziente Integration kann aber dadurch erreicht werden, daß die Spracherkennung nicht nur erkannte Symbole an die Sprachverstehensverarbeitung weitergibt, sondern auch Information, die den Prozeß des Sprachverstehens steuert. Dabei wird es dann auch nicht das Ziel sein *alle* prinzipiell möglichen Analysen zu erstellen, sondern "die beste Analyse" in den allermeisten Fällen wäre ein ausreichendes Ergebnis.

"Die beste Analyse" wäre sicherlich eine, die die Intention des Sprechers erkennt und eine "sinnvolle" Reaktion zeigt. In dieser Arbeit wird "die beste Analyse" jedoch lediglich die sein, welche eine aus der Spracherkennung resultierende Bewertungszahl maximiert. Das Problem besteht dann im wesentlichen aus zwei Teilen:

1. Wie definiert man die Bewertungszahl?
2. Welche Strategie verfolgt man, um in den allermeisten Fällen die Analyse mit der maximalen Bewertungszahl zu erhalten?

Die Bewertungszahl, die den Sprachverstehensprozeß steuern soll, kann sich z.B. aus den folgenden Komponenten zusammensetzen:

- **Akustische Bewertungen:** Mit Hilfe statistischer Wortmodelle wird die Wahrscheinlichkeit berechnet, daß *ein bestimmtes Wort* zu einem bestimmten Zeitpunkt geäußert wurde.
- **Statistische Sprachmodelle:** Probabilistische reguläre oder kontextfreie Grammatiken ermöglichen die Bewertung *einer Wortsequenz.*

Im Constraint-Verarbeitungssystem kommen dann vor allem zwei Methoden zur Steuerung des Sprachverstehensprozesses zum Einsatz:

- Die **Bestensuche (Best-First-Search)** analysiert die Ambiguitäten in einer Reihenfolge, die z.B. durch die kombinierte Bewertungszahl von akustischer Bewertung und statistischem Sprachmodell bestimmt wird. Sie findet viel früher eine – zudem meist recht gute – Analyse, als eine Breitensuche (Breadth-First-Search).
- **Strahlsuchverfahren (Beam-Search)** benutzen ähnliche oder gleiche Bewertungen wie die Besten-Suchverfahren, arbeiten aber nicht nur auf einer Möglichkeit, sondern auf einer Menge von gut bewerteten Pfaden.

Es gibt zahlreiche Systeme, die diese Techniken erfolgreich anwenden und verfeinern, z.B.:

- [Shi87] zeigt die erfolgreiche Verwendung von n-gram-Modellen[1].
- In [PNea87] wird ein Strahlsuchverfahren vorgestellt.
- Dieses wird in [PN89] mit einem n-gram-Modell kombiniert.
- Wrigley und Wright [WW91a], [Wri90], [WW91b] benutzen eine probabilistische, kontextfreie Grammatik als stochastisches Sprachmodell.

[1] Siehe dazu später S. 29ff.

- In [Sen89] wird eine Bestensuche propagiert.
- In [ADNS94] und [Ney93] werden Strahlsuchstrategien über die Menge der Phonemhypothesen beschrieben, und wie, daran folgend, ein N-Gram Modell verwendet werden kann.
- Ludwig Schmid benutzt in [Sch94] den A*-Algorithmus zur Bestensuche.

1.3 Ziel

Für die Kopplung von Spracherkennung und Sprachverstehen, wie in Kapitel 1.2 beschrieben, werden auf der Seite der Constraintverarbeitung vor allem der Chartparser und seine Variationen, Earley- und CYK-Algorithmus (siehe [Ear68]) benutzt.

M. Tomita hat in [Tom85] einen Parsingalgorithmus vorgeschlagen, welcher das LR-Parsing für deterministische kontextfreie Sprachen[AU72] verallgemeinert, und mit dem auch ambige kontextfreie Grammatiken geparst werden können. [2] Dieser Algorithmus (GLRP, Generalized LR-Parsing) arbeitet auf Grammatiken für natürliche Sprachen oft effizienter (siehe [Tom87, Sha91, BC93]) als Earley oder CYK.

Das Hauptziel dieser Arbeit ist die Untersuchung, wie sich das GLR-Parsing als Symbolverarbeitungsmechanismus mit Spracherkennung kombinieren läßt.

Dazu wird in dieser Studienarbeit in:

Kapitel 2: die ursprüngliche Version von Tomita's Algorithmus dargestellt und implementiert. Eine Unstimmigkeit wird diskutiert, die bei Tomita's Algorithmus auftritt.

Kapitel 3: eine Version behandelt und realisiert, die einen vollständigen Parse eines Lattice[3] liefert.

Kapitel 4: ein kurzer Überblick über die statistischen Modelle gegeben, die zur Steuerung des Parsing-Prozesses verwendet werden.

Kapitel 5: eine Agenda-Strategie für GLRP vorgestellt und implementiert, welche als Grundlage für die beiden folgenden Punkte dient.

Kapitel 6: ein verbesserter Ansatz für eine A*-Suche durch das Lattice vorgeschlagen.

Kapitel 7: GLRP mit einem Strahlsuchverfahren gesteuert.

Als Basis für verschiedene Experimente wurden die folgenden Module realisiert:

- *Die Grammatik* umfaßt 1564 kontextfreie Regeln, die einfache Aussagesätze mit freier Wortstellung beschreiben (Kap. C.1).
- *Das Lexikon* umfaßt 500 Wortformen mit durchschnittlich 3,4 möglichen Kategorien pro Wortform (Kap. C.2).

[2] Sein Verfahren war zunächst auf nicht-zyklische Grammatiken beschränkt, allerdings wird in [NF91] eine entsprechende Erweiterung vorgestellt. Eine weitere Verallgemeinerung betrifft das LR-Parsing von Tree Adjoining Grammars in [Sch90].

[3] Ein Lattice ist die Menge aller beim Decoding erkannten Symbole. Genaueres dazu in Kapitel 3.

- *Der Tabellenkonstruktor* erlaubt das effiziente Erstellen von SLR- und LALR-Tabellen (Kap. D).

Üblicherweise werden für Sprachverarbeitungssysteme Unifikationsgrammatiken so verwendet, daß mit der Abarbeitung der Grammatikregeln auch gleichzeitig Semantik und Teile der Pragmatik einer Äußerung erfaßt werden. Hier werden im folgenden lediglich kontextfreie Grammatiken benutzt, was allerdings keine konzepionelle Einschränkung darstellt, da GLR-Parsing, wie in [Tom87] gezeigt wird, auch auf Unifikationsgrammatiken erweiterbar ist.

Kapitel 2

Der Tomita-Parser

2.1 Der LR-Parsing-Algorithmus

LR-Parsing in seinen verschiedenen Varianten, SLR, LR und LALR, wird wegen seiner Effizienz bei fast jedem Compiler zur Syntaxanalyse eingesetzt und auch Compilergeneratoren wie Yacc [Joh75] erzeugen LR-Parser.

Ähnlich einem Pushdown-Automat [HU79] wird ein LR-Parser mit einem Stack und einer Zustandsübergangsfunktion gesteuert, $\delta : \Gamma \times \Sigma \longrightarrow \Omega$, wobei Γ das Stack- und Σ das Eingabealphabet (Menge der möglichen Wortkategorien) darstellt und Ω die Menge der möglichen Aktionen ist.[1]

Die Zustandsübergangsfunktion kann mit Hilfe einer zweidimensionalen Tabelle beschrieben werden, da es nur endlich viele Stacksymbole und Eingabesymbole gibt. Allerdings arbeitet der LR-Parser im Gegensatz zum Pushdown-Automat nur mit einer deterministischen Funktion, also nicht mit mehrdeutigen Tabelleneinträgen.

Eine ausführliche Beschreibung der verschiedenen Tabellenkonstruktoren findet man z.B. in [AU72]. Im folgenden wird anhand von Abbildung 2.1 der Mechanismus für die Konstruktion einer SLR-Tabelle - die einfachste Methode - veranschaulicht.

[1] Ein Pushdown-Automat (PDA) hat zusätzlich eine finite Menge von Zuständen. Dafür löst ein Eingabesymbol bei einem PDA genau eine Folgeaktion aus, während beim LR-Parsing sukzessive mehrere Folgeaktionen bestimmt werden können. Es wird später leicht vorzustellen sein, daß das der Ansatzpunkt sein könnte um die Äquivalenz von DPDA und LRP bzw. NPDA (= PDA) und GLRP zu zeigen.

Grammatik $\mathcal{G}$
Startsymbol S
Nichtterminale $N = \{S, A, B\}$
Terminale $T = \{a, b\}$

0.	S	$\rightarrow$	AB
1.	A	$\rightarrow$	aA
2.	A	$\rightarrow$	a
3.	B	$\rightarrow$	b

Abbildung 2.1: Grammatik 1

2.1.1 Parsetabelle

Zunächst müssen zur Erklärung einige Begriffe definiert werden:

Eine markierte Regel ist eine Produktion aus der Grammatik, die auf der rechten Seite eine Markierung vor, nach oder zwischen ihren Symbolen hat:
$P = (H \rightarrow \alpha.\beta)$, wobei $\alpha, \beta \in (N \cup T)^* \wedge H \in N$.

Die intuitive Bedeutung der Markierung ist, daß als nächstes im weiteren Parsingprozeß die Kategorie erwartet wird, die unmittelbar nach der Markierung folgt.

Eine markierte Regel Q ist die direkte Ableitung einer anderen markierten Regel P:
$P \rightarrow Q \Leftrightarrow P = (H \rightarrow \alpha.K\beta) \wedge \exists (K \rightarrow \delta) \in \mathcal{G} \wedge Q = (K \rightarrow .\delta)$.

Eine markierte Regel R ist in der Hülle von P:
$R \in \mathrm{Cl}(P) \Leftrightarrow P \overset{*}{\rightarrow} R \Leftrightarrow R = P \vee \exists Q : P \overset{*}{\rightarrow} Q \wedge Q \rightarrow R$

Die markierte Regel R ist in der Hülle von einer Menge von markierten Regeln, wenn sie in der Hülle mindestens einer dieser Regeln ist:
$R \in \mathrm{Cl}(\{P_1, \cdots, P_n\}) \Leftrightarrow \exists i : R \in \mathrm{Cl}(P_i)$.

Die Menge der initialen markierten Regeln $\mathcal{R}_I$ besteht aus den Regeln, die direkt vom Startsymbol abgeleitet werden können und die Markierung zu Beginn der rechten Seite haben. In der Grammatik von Abb. 2.1 ist das $\{(S \rightarrow .AB)\}$.

Der Zustandsübergang δ von einer markierten Regel R $(R = (H \rightarrow \alpha.a\beta))$ mit einem Symbol v ist wie folgt definiert:

$$\delta(R, v) = \begin{cases} \emptyset & \text{gdw.} \quad v \neq a \\ \{(H \rightarrow \alpha a.\beta)\} & \text{gdw.} \quad v = a \end{cases}$$

Ein Zustand besteht aus einer Menge von markierten Regeln.

Der Zustandsübergang δ von einem Zustand I mit einem Symbol v ist wie folgt definiert:

$$\delta(I, v) = \bigcup_{\forall R \in I} \delta(R, v)$$

Nun kann man den Algorithmus zur Konstruktion der SLR-Tabellenzustände wie in Abb. 2.2 beschreiben.

Die zugrundeliegende Idee ist dabei die folgende:

- Eine markierte Regel beschreibt, wie weit die Regel bereits abgearbeitet wurde.
- Ein Zustand besteht aus einer Menge von markierten Regeln. Ein Zustand beschreibt also, welcher linke Kontext bereits geparst wurde, und welche Regeln dabei wie weit benutzt wurden.
- Für jedes Wort aus der Grammatik muß es einen Weg durch das Zustandsübergangsdiagramm geben, analog zu einem Pushdown-Automat. Alle erforderlichen Zustände müssen vom Tabellenkonstruktor im voraus berechnet werden.

Konstruktor:				Input: Grammatik $\mathcal{G}$ Output: SLR-Tabellenzustände
1.	Initialisiere			
2.		Zustand $S_0 = \mathrm{Cl}(\mathcal{R}_{\prime})$		
3.		Die Menge der neuen Zustände $\mathcal{I} = \{S_0\}$		
4.		Die Menge der abgearbeiteten Zustände $\mathcal{Z} = \emptyset$		
5.	Bis $\mathcal{I} = 0$:			
6.		Nimm beliebigen Zustand I aus $\mathcal{I}$		
7.		Für alle Symbole v aus $(N \cup V)$:		
8.			Berechne $J = \mathrm{Cl}(\delta(I, v))$	
9.			Wenn J bereits existiert, d.h. $J \in \mathcal{I} \cup \mathcal{Z} \cup \{I\}$	
10.				dann: tue nichts
11.				sonst: füge J zu $\mathcal{I}$ hinzu.
12.		Füge I zu $\mathcal{Z}$ hinzu.		
$\mathcal{Z}$ ist die Menge aller SLR-Tabellenzustände.				

Abbildung 2.2: Der Konstruktor der SLR-Tabellenzustände

In einem gegebenen Zustand I bestimmt die Menge der gegebenen markierten Regeln die Menge der Produktionen, die für die nächsten Schritte in Betracht kommen, und die Markierungen geben die Symbole an, die als nächstes erwartet werden.

In den nächsten Zustand gelangt man, indem ein terminales oder ein nichtterminales Symbol v *geschoben* wird. Der neue Zustand ist dabei mit dem Zustandsübergang $F(I, v)$ definiert. Dabei ist es durchaus möglich, daß man von zwei verschiedenen Zuständen aus beim Schieben des gleichen Symbols in denselben Zustand gelangt, wobei zwei Zustände genau dann identisch sind, wenn die Elemente in ihren Hüllen identisch sind.

In Abb. 2.3 wird das Zustandsübergangsdiagramm für die Grammatik in Abbildung 2.1 gezeigt.

Die Berechnung beginnt im Zustand 0 mit der Hülle der einzigen initialen markierten Regel, $(S \rightarrow .AB)$. $F(0, a) = 1$, $F(0, A) = 2$, $F(1, a) = 1$ usw. Die Berechnung neuer Zustände wird solange fortgesetzt, bis kein neuer Zustand mehr gefunden werden kann.

Ein solches Zustandsübergangsdiagramm kann dann umgesetzt werden in eine Tabelle, welche anhand von Zustandsnummern, die entsprechenden Übergänge beschreibt (siehe Tab. 2.1). In der Tabelle wird jedem Paar aus Zustand und Symbol der neue Zustand zugeordnet. Allerdings werden bei den Einträgen auf der Seite der terminalen Symbole nicht nur Übergänge beschrieben, sondern auch mögliche *Reduktionen*, weswegen die Übergänge zusätzlich mit einem "s" und Reduktionen mit einem "r" gekennzeichnet werden.

Wenn sich nämlich in einem Zustand eine markierte Regel befindet, die die Markierung ganz am Ende stehen hat, dann bedeutet das, daß diese Regel vollständig abgearbeitet wurde und die n letzten Symbole durch den Kopf dieser Regel ersetzt werden müssen. Die n letzten Symbole sind dabei diejenigen, die den Symbolen auf der rechten Seite dieser Regel entsprechen.

In Zustand 5 von Abb. 2.3 erfolgt also immer eine Reduktion mit der Regel 1, (A $\rightarrow$ aA) – unabhängig von dem terminalen Symbol, welches nachfolgt.

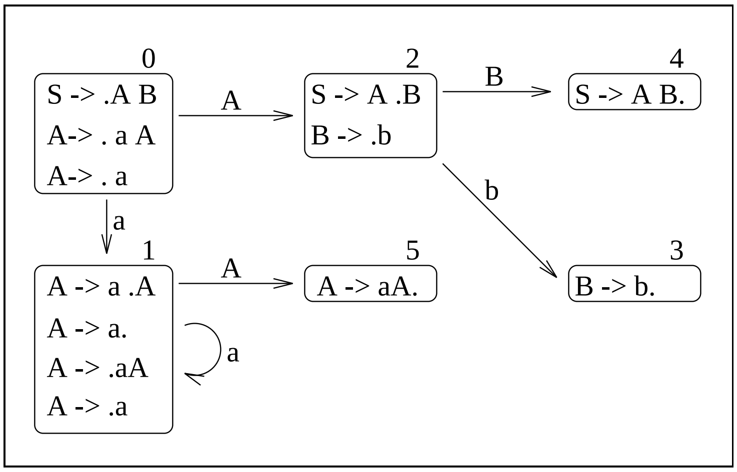

Abbildung 2.3: Zustandsübergangsdiagramm für Grammatik in Abb. 2.1 .

	a	b	$	A	B
0	s1			2	
1	s1,r2	r2	r2	5	
2		s3			4
3	r3	r3	r3		
4			Acc		
5	r1	r1	r1		

Tabelle 2.1: SLR-Tabelle für Grammatik in Abb. 2.1.
“$” ist das Satzendezeichen, welches jede Eingabe abschließen muß.

	a	b	$	A	B
0	s1			2	
1	s1	r2		5	
2		s3			4
3			r3		
4			Acc		
5		r1			

Tabelle 2.2: LR-Tabelle für Grammatik in Abb. 2.1.

2.1.2 Algorithmus

Der LR-Parsing-Algorithmus muß zwei Aktionen ausführen können, um sich auf dem Zustandsübergangsdiagramm zu "bewegen":

1. Das *Schieben* eines Symbols vom augenblicklichen Zustand.

2. Die *Reduktion* über die n letzten Symbole.

Zu diesem Zweck wird der Algorithmus mit einem Stack betrieben. Oben auf dem Stack befindet sich jeweils der aktuelle Zustand. Wenn nun ein Eingabesymbol geparst wird, dann wird anhand des aktuellen Zustands und des Eingabesymbols die zu erledigende Aktion aus der Zustandsübergangstabelle entnommen und ausgeführt.

Wenn eine Schiebeaktion anliegt, wird der neue Zustand auf dem Stack abgelegt. Wenn eine Reduktion ansteht, dann werden die letzten n Zustände vom Stack geholt und anhand des (n+1)-letzten Zustands und des soeben reduzierten Nichtterminals wird der neue Zustand bestimmt, mit dem erneut versucht wird, das Eingabesymbol zu kombinieren.

Probleme ergeben sich nun allerdings, wenn die Tabelle nicht so konstruiert werden kann, daß jeder Tabelleneintrag eindeutig ist. In der SLR-Tabelle 2.1 z.B. stehen im Eintrag $F(1, a)$ die zwei Einträge "schiebe in Zustand 1" und "reduziere mit Regel 2".

Für deterministische kontextfreie Sprachen wie "a^+b" aus Abb. 2.1 kann man immer eine Grammatik finden, die mit einem ausgefeilterem Tabellenkonstruktionsmechanismus - z.B. LR(1) - eine Tabelle mit eindeutigen Einträge generiert, wie z.B. in Tab. 2.2.[2]

Bei kontextfreien Sprachen jedoch, die *inhärent* ambig sind, wie z.B. der Ausschnitt einer deutschen Grammatik, den die Abb. 2.4 beschreibt, sind solche Mehrfacheinträge nicht vermeidbar.

Eine derartige Tabelle kann dann nicht von Knuth's LR-Parsing-Algorithmus verwendet werden. Statt dessen muß dann ein Algorithmus benutzt werden, wie ihn Tomita vorgeschlagen hat [Tom85].

[2] [HU79], S.233: "The LR-grammars have the property that they generate exactly the deterministic context free languages."

Grammatik $\mathcal{G}$
Startsymbol S
Nichtterminale N = {S, NP, VP, PP}
Terminale T = {det, n, v, p}

0.	S	$\rightarrow$	NP VP
1.	S	$\rightarrow$	S PP
2.	NP	$\rightarrow$	NP PP
3.	NP	$\rightarrow$	det n
4.	NP	$\rightarrow$	n
5.	VP	$\rightarrow$	v NP
6.	PP	$\rightarrow$	p NP

Abbildung 2.4: Grammatik 2

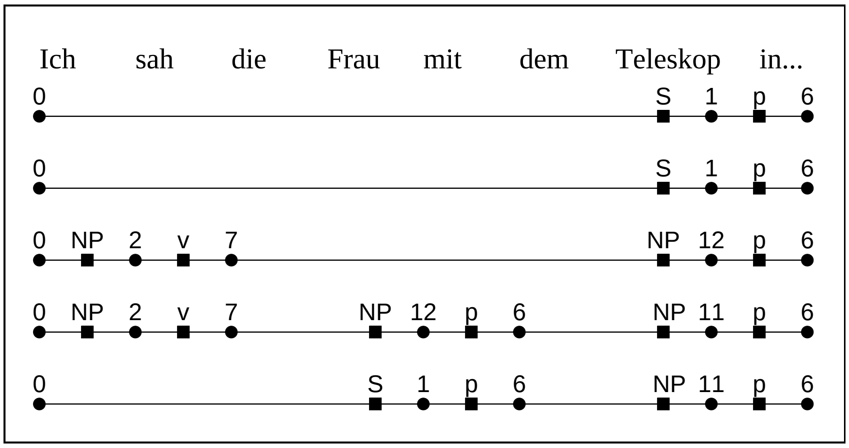

Abbildung 2.5: Eine Stackliste zur Simulation des Nichtdeterminismus

2.2 Der verallgemeinerte LR-Parsing-Algorithmus

2.2.1 Wirkungsweise

Die grundlegende Idee des GLRP-Algorithmus besteht darin, den Parsing-Prozeß nichtdeterministisch ablaufen zu lassen, um damit den mehrdeutigen Tabelleneinträgen Rechnung zu tragen.

Die Simulation dieses Nichtdeterminismus könnte dann dadurch geschehen, daß sich ein Prozeß bei jeder Verzweigungsstelle teilt, und jeder neue Prozeß eine Kopie des Stacks bekommt.

Allerdings wäre ein solcher Parsing-Algorithmus höchst ineffizient für Grammatiken wie in Abb. 2.4, denn mit jeder neuen Präpositionalphrase (PP) würde sich die Anzahl der Prozesse mindestens verdoppeln, was zu exponentiellem Laufzeitverhalten führen würde.

Bei genauerer Betrachtung fällt aber auf, daß sich viele Stacks der verschiedenen Prozesse nur in wenigen Knoten unterscheiden (siehe das Beispiel in Abb. 2.5). Wenn gleiche Knoten der Stacks so zusammengefaßt werden, daß sich anstatt eines Stacks ein graph-strukturierter Stack als Steuerungsstruktur ergibt, dann reduziert sich die Anzahl der aktiven Stackenden erheblich (Abb. 2.6). Das Laufzeitverhalten in Abhängigkeit von der Länge der Eingabe wird

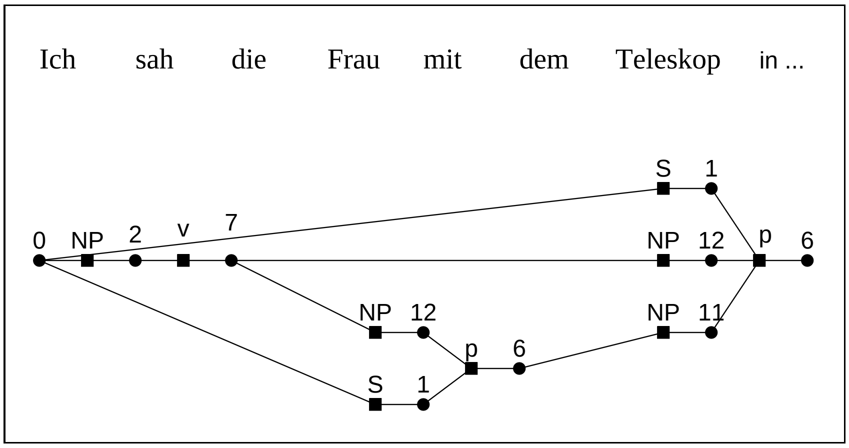

Abbildung 2.6: Ein graph-strukturierter Stack

zu $\mathcal{O}(1+\rho)$, wobei ρ die Länge der rechten Regelseite der längsten Produktion ist ([Kip91]). Damit ist die theoretische Komplexität des GLR-Parsers schlechter als die eines Chartparsers. Aber diese Komplexität wird vor allem durch den Aufbau des Parse Forest bestimmt.[3]

Der *Parse Forest* ist die zweite zentrale Datenstruktur beim GLRP. Während im graph-strukturierten Stack vor allem Steuerinformationen abgespeichert werden[4], die regeln, welche Aktionen ausgeführt werden müssen, findet man im Parse Forest das Ergebnis der ausgeführten Aktionen, die Parse Bäume, und damit die syntaktische Struktur des geparsten Satzes. Das Besondere bei Parse Forests ist, daß sie keine reine Aufzählung aller möglichen Parse Trees enthalten. Dies ist wichtig, da eine reine Aufzählung aller Bäume bei kontextfreien Grammatiken zu exponentiellem Laufzeitverhalten führen würde, da bei ambigen Grammatiken, wie in Abb. 2.4, die Anzahl der möglichen Ableitungsbäume exponentiell mit der Länge der Eingabe wächst.

In einem Parse Forest liegen die Analysebäume in einer gepackten Form vor, aus der die einzelnen Bäume mit einer trivialen Aufzählung herausgelesen werden können[5].

In der Praxis hat sich gezeigt, daß sich Parse Forests sehr viel besser verhalten als ihre theoretische Komplexität zunächst erwarten läßt, und Johnson [Joh91] vermutet, daß sich die Komplexität des GLRP, ähnlich wie beim Chartparser, durch eine weitere Kompaktifizierung des Parse Forest auf $\mathcal{O}(n^3)$ verringern läßt.[6]

Außerdem scheint es, daß natürliche Sprachen nur "wenig ambig" sind, d.h. daß nur wenige Tabelleneinträge wirklich mehrdeutig sind, und der größte Teil des Parsingvorgangs daher deterministisch ablaufen kann. Das bedeutet aber wiederum, daß ein großer Teil des Parsingprozesses mit dem äußerst effizienten LR-Parsing-Algorithmus und geringem Verwaltungs-

[3]Ein GLR-Parser, der keinen Parse Forest erzeugt, kann in $\mathcal{O}(n^3)$ arbeiten. Vgl. [Joh91, LAKA92]

[4]Der graph-strukturierte Stack enthält auch alle grammatikalischen Informationen. Allerdings kann man die Ableitungsbäume dem Stack nicht einfach entnehmen.

[5]Vergleiche zur Arbeitsweise des Parse Forest Kap. 2.3

[6]Vgl. zur weiteren Kompaktifizierung des Parse Forest [BL89]. Die Frage ist offen, ob sich diese Kompaktifizierung auch mit Unifikationsgrammatiken durchführen läßt. Mit stochastischen Grammatiken, die alle Bewertungen mitführen, geht dies sicherlich nicht. Dies wäre bei stochastischen Grammatiken aber nötig, um die beste Ableitung ohne beträchtlichen Mehraufwand herauszufinden.

mehraufwand betrieben werden kann ([Tom85], S. 3f).

Eine andere Beobachtung macht Shann [Sha91], der feststellt, daß das GLR-Parsing vor allem durch die effiziente Handhabung des Parse Forest bei stark ambigen Sätzen Vorteile besitzt.

Allerdings weisen sowohl Tomita als auch Shann darauf hin, daß vergleichende Experimente schwierig sind, da sowohl Implementation als auch die Testmenge das Ergebnis stark beeinflussen.

Die Debatte über die Angemessenheit von GLRP für das Parsing natürlicher Sprachen ist kontrovers. Argumente für Tomita's Standpunkt finden sich z.B. in [Shi83, Per85, BW84], Argumente dagegen in [AJ90], Argumente für eine strukturelle Gleichheit - und damit auch eine Gleichheit im Verhalten - zwischen GLR- und Chartparsing kann man z.B. [Lee92] entnehmen.

2.2.2 Beschreibung des GLRP

Eine formale Spezifikation des GLRP ist in [Tom85], S. 65ff zu finden, weswegen in dieser Studienarbeit hier darauf verzichtet wird. Dafür soll der Leser in diesem Kapitel in die Lage versetzt werden, sich anhand einer weniger formalen Spezifikation mit dem Algorithmus vertraut zu machen. Der hier beschriebene Algorithmus lehnt sich in Terminologie und Struktur sehr stark an die erfolgte Implementation an, bleibt aber den Grundzügen von Tomitas ursprünglicher Spezifikation sehr nah.

Bereits an dieser Stelle ist anzumerken, daß Tomita eine Unstimmigkeit hat, was seine formale Spezifikation und seinen Beispieltrace in [Tom85], Kapitel 3 anbelangt. In dieser Arbeit fiel die Entscheidung zugunsten der Strategie, die im Beispieltrace verfolgt wird.

Der genaue Unterschied und die daraus resultierenden Probleme werden in Kapitel 2.4 erörtert.

Die Steuerstruktur

Der GLRP-Algorithmus arbeitet strikt von rechts nach links und parst ein Wort jeweils vollständig, bevor das nächste bearbeitet wird. Deswegen arbeitet der Algorithmus wortinkrementell und läßt sich auch dementsprechend in Definition 2.1 beschreiben.

Dabei wird mit dem bereits erwähnten graph-strukturierten Stack (DAG) und einem Parse Forest gearbeitet. Beides sind monoton wachsende Strukturen.

Der *DAG* besteht aus einem Stack von DAG-Knotenmengen. Direkt zugegriffen wird nur auf die oberste Knotenmenge, allerdings sind die Knoten untereinander durch Links verbunden, über die man auch Knoten aus weiter unten liegenden Mengen erreicht. Anfangs ist der Stack mit einer einzigen Knotenmenge initialisiert, die nur den Startknoten enthält.

Die Merkmale eines *DAG-Knotens* sind sein Zustand und die Menge seiner Links. Der Startknoten hat den Startzustand (bei dem implementierten Tabellenkonstruktor ist das immer der Zustand 0) und eine leere Menge von Links. Die Komponenten eines *Links* bestehen aus einer Referenz auf einen Parse-Knoten und aus einer Menge von vorhergehenden DAG-Knoten.

Der *Parse Forest* besteht aus einer Menge von Parse-Knoten, welche anfangs leer ist. Jeder *Parse-Knoten* hat einen Eintrag für eine Kategorie und eine Referenz. Diese Referenz enthält im terminalen Fall einen Verweis auf das geparste Wort, im nichtterminalen Fall eine Menge von Referenzlisten auf andere Parse-Knoten.

Die global verwendeten Variablen sind dabei:

- **DAG:** Der graph-strukturierte Stack.
- **F:** Der Parse Forest.
- **CURRENT:** Die Menge der jeweils aktiven DAG-Knoten.
- **REDUCES, ε-REDUCES, SHIFTS, ACCEPTS:** Die Menge der jeweils noch zu erledigenden Aktionen. Dabei bestehen die einzelnen Aktionen aus unterschiedlichen Komponenten:
  - Eine Aktion in REDUCES besteht aus einem DAG-Knoten und einer Regel.
  - In ε-REDUCES wird eine Aktion durch einen DAG-Knoten und die jeweilige ε-Regel charakterisiert.
  - Die Bestandteile einer SHIFT-Aktion sind DAG-Knoten, zu schiebende Kategorie und aktuelles Wort.
  - Eine ACCEPT-Aktion besteht nur aus einem DAG-Knoten.
- **TAB:** Die Parsing-Tabelle.

Definition 2.1 *ParseWord(WORD)*

1. *Initialisiere REDUCES = ε-REDUCES = SHIFTS = ACCEPTS = $\emptyset$.*
2. *Die Menge der aktiven DAG-Knoten CURRENT = Top(DAG)*[7]*.*
3. *Solange bis es keine aktiven DAG-Knoten in CURRENT mehr gibt:*
   (a) *Bestimme alle Aktionen (REDUCES, ε-REDUCES, SHIFTS, ACCEPTS) die mit den Kategorien von WORD und den DAG-Knoten in CURRENT möglich sind mit Hilfe der Parsing-Tabelle.*
   (b) *Entferne alle DAG-Knoten aus CURRENT.*
   (c) *Wenn es Aktionen in REDUCES gibt, dann erledige diese, setze REDUCES auf $\emptyset$ und fahre anschließend mit 3a fort.*
   (d) *Wenn es Aktionen in ε-REDUCES gibt, dann rufe ε-Reduce(), setze ε-REDUCES auf $\emptyset$ und fahre anschließend mit 3a fort.*
   (e) *Wenn es Schiebeaktionen in SHIFTS gibt, dann rufe Shift() und setze SHIFTS auf $\emptyset$.*
   (f) *Wenn es eine Akzeptiere-Aktion in ACCEPTS gibt, dann vermerke den entsprechenden Parse-Knoten als Wurzel des Parse Forest und setze ACCEPTS auf $\emptyset$.*

Reduktionen

Reduktionen werden also, wenn sie auftreten, immer zuerst abgearbeitet. Da Reduktionen völlig unabhängig voneinander arbeiten, genügt es, in Definition 2.2 eine einzelne Reduktion zu beschreiben. VERTEX ist dabei der DAG-Knoten, von dem aus die Reduktion startet, und RULE ist die Grammatikregel, die zur Reduktion benutzt wird.

[7] "Top" referenziert den obersten Eintrag eines Stacks, ohne ihn zu entfernen.

Definition 2.2 *Reduce(VERTEX, RULE)*

1. *Setze DERIVATION_LENGTH auf die Anzahl der Symbole auf der rechten Seite von RULE.*

2. *Setze DERIVATION_LINKS zunächst auf $\emptyset$ und erweitere es dann wie folgt:*

   - *Suche entlang aller Links in VERTEX rekursiv nach den Links, die den Abstand DERIVATION_LENGTH von VERTEX haben.*
     Die Links, die in VERTEX selbst sind haben dabei den Abstand 1, die Links, die diesen vorangehen (mit anderen DAG-Knoten als "Zwischenstation") haben den Abstand 2, usw.
     Dabei erhält man eine Menge von Link-Pfaden der Länge DERIVATION_LENGTH.

   - *Erstelle bei dieser rekursiven Suche eine Referenzliste für jeden Pfad. Die Referenzliste eines Pfades besteht aus den Referenzen auf den Parse Forest, die in den Links stehen, die auf diesem Pfad aufgesucht worden sind.*
   - *Jeder Pfad P_i erreicht einen Link L_i mit Abstand DERIVATION_LENGTH von VERTEX. Füge jedes Paar (P_i, L_i) zu DERIVATION_LINKS hinzu.*

3. *Für jedes Paar (P_i, L_i) in DERIVATION_LINKS: ReduceBundle((P_i,L_i), RULE)*

Jeder Pfad soll möglichst nur einmal reduziert werden, denn jede Reduktion beansprucht vergleichsweise viel Rechenzeit, vor allem wenn man beabsichtigt eine Unifikationsgrammatik zu benutzen. Diese Reduktion eines Pfades erfolgt in "ReduceBundle".

Definition 2.3 *ReduceBundle((P_i,L_i), RULE)*

1. *Setze HEAD auf den Kopf der Regel RULE.*

2. *Erzeuge einen Parse-Knoten NODE im Parse Forest F mit der Referenzliste (P_i) und der Kategorie HEAD.*

3. *Setze PREDECESSORS auf die Menge der DAG-Knoten, die L_i vorangehen.*

4. *Partitioniere PREDECESSORS in W_SETs. W_SET_k besteht aus den DAG-Knoten, deren Zustand, gepaart mit HEAD, im Tabellen-Lookup den Zustand k ergibt. Formal ausgedrückt: $W\_SET_k = \{\nu \mid TAB(State(\nu), HEAD) = k\}$*

5. *Für jedes nicht leere W_SET_k: ReduceWSet(W_SET_k, P_i, HEAD, NODE)*

Im Allgemeinen können nicht alle DAG-Knoten in "PREDECESSORS" gleichzeitig reduziert werden, da verschiedene DAG-Knoten mit verschiedenen Zuständen verschiedene Nachfolgezustände zur Folge haben können. Diejenigen DAG-Knoten, aus denen ein gemeinsamer Zustand resultiert, werden in "ReduceWSet" gemeinsam reduziert, damit der graphstrukturierte Stack eine größtmögliche Strukturteilung erhält.

Der Unterschied zwischen der Reduktion mit W_SETs und der ohne wird deutlich in den Abbildungen 2.7 und 2.8. Eine nachfolgende Reduktion an dem DAG-Knoten mit Zustand 6 müßte im ersten Fall 3 verschiedene Links aufsuchen und getrennt reduzieren, während im zweiten Fall evtl. weniger Pfade und damit weniger Reduktionen notwendig werden.

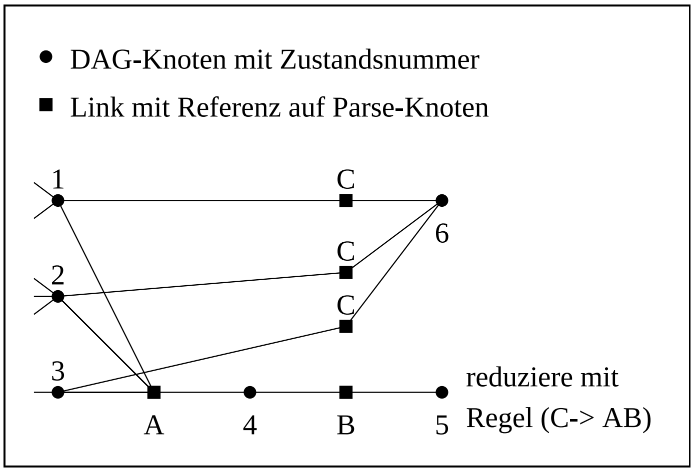

Abbildung 2.7: Reduktion einzelner DAG-Knoten

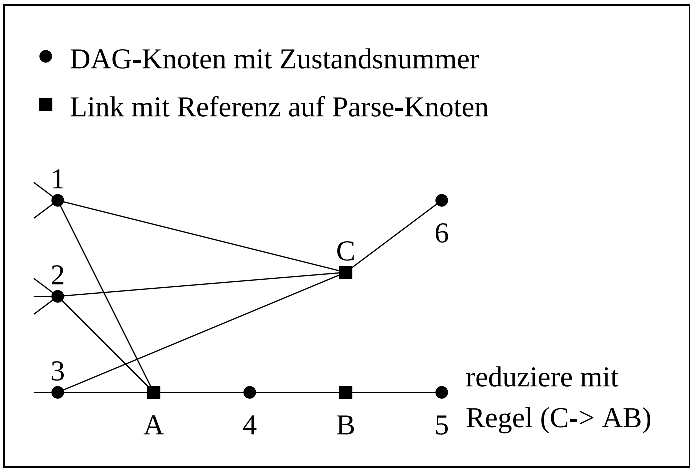

Abbildung 2.8: Reduktion mit Partitionierung in W_SETs — maximale Strukturteilung

Definition 2.4 *ReduceWSet(W_SET$_k$, P_i, HEAD, NODE)*

1. *Suche in Top(DAG) nach einem DAG-Knoten mit dem Zustand k. Falls kein solcher DAG-Knoten in Top(DAG) existiert, dann gehe nach 3, ansonsten sei ν dieser DAG-Knoten.*

2. *Benutze beim Reduzieren den DAG-Knoten ν:*

   (a) *Durchsuche alle Links, die es in ν gibt und teste, ob es einen Link L mit einer Menge von Vorgänger-DAG-Knoten gibt, die identisch ist mit der Menge in W_SET$_k$. Wenn dies der Fall ist, dann wurde eine lokale Ambiguität im DAG* und *eine im Parse Forest entdeckt, ansonsten gehe nach 2c.*

   (b) *Der gefundene Pfad P_i ist ambig mit den Pfaden, die in der Parse Forest-Referenz R von Link L stehen. Deswegen ergänze R um P_i. Return.*

   (c) *Die Menge der Links eines bereits existierenden DAG-Knotens muß um einen neuen Link erweitert werden:*

      i. *Erzeuge einen Link L mit einer Referenz auf NODE und W_SET$_k$ als seinen Vorgänger-DAG-Knoten.*

      ii. *Erweitere die Menge von Links des DAG-Knoten ν um den Link L.*

      iii. *Wenn ν in CURRENT ist, dann wurden von ν aus noch keine Reduktionen gestart und neugestartete Reduktionen passieren automatisch den neuen Link. Aber wenn ν nicht mehr aktiv ist, dann haben möglicherweise einige Reduktionen die anderen, alten Links von ν bereits passiert. Dann finde die erfolgten Reduktionen anhand der Parse-Tabelle und addiere diese als spezielle Reduktionen von Knoten ν aus über Link L zu der Menge der Reduktionen in RE-DUCES.*
      Diese speziellen Reduktionen verlaufen genau wie andere auch. Der einzige Unterschied besteht darin, daß beim ersten Suchschritt zum Erstellen der Pfade nicht alle Links von ν passiert werden, sondern nur der neue Link L.
      Wenn ν außerdem bereits akzeptiert wurde, dann muß NODE als eine weitere mögliche Wurzel des Parse Forest vermerkt werden.

      iv. *Return.*

3. *Erzeuge beim Reduzieren einen neuen DAG-Knoten:*

   (a) *Erzeuge einen Link L mit einer Referenz auf NODE und W_SET$_k$ als seinen Vorgänger-DAG-Knoten.*

   (b) *Erzeuge einen DAG-Knoten ν mit $\{L\}$ als seiner Menge von Links und k als seinem Zustand.*

   (c) *Füge ν zu Top(DAG) und zu CURRENT hinzu. ν ist jetzt also ein neuer aktiver DAG-Knoten.*

ε-Reduktionen und Schiebeaktionen

sind sich sehr ähnlich. Sie unterscheiden sich vor allem dadurch, daß im ersten Fall nichtterminale, im zweiten Fall dagegen terminale Symbole geschoben werden.

Definition 2.5 *ε-Reduce()*

1. *Gruppiere ε-REDUCES nach den verschiedenen Regeln, aus denen die einzelnen ε-Reduktionen bestehen in ε-$REDUCES_{RULE_1}$ bis ε-$REDUCES_{RULE_n}$.*
2. *Für alle ε-$REDUCES_{RULE_i}$:*
   (a) *Erzeuge einen Knoten NODE im Parse Forest mit dem Kopf der Regel $RULE_i$ als Kategorie ($HEAD_i$) und einer Referenzliste mit einer leeren Liste als einzigem Eintrag.*
   (b) *Gruppiere die DAG-Knoten aus ε-$REDUCES_{RULE_i}$ nach den verschiedenen Zuständen, die sich aus dem Tabellenlookup des jeweiligen DAG-Knoten-Zustandes und dem $HEAD_i$ ergeben in verschiedene W_SETs (analog Def. 2.3 Schritt 4).*
   (c) *Für alle W_SET_k:*
      i. *Erzeuge einen neuen Link L mit einer Referenz auf NODE und W_SET_k als seiner DAG-Knotenmenge.*
      ii. *Erzeuge einen neuen DAG-Knoten ν mit $\{L\}$ als seiner Menge von Links und k als seinem neuen Zustand.*
      iii. *Addiere ν zu CURRENT und zu Top(DAG).*

Definition 2.6 *Shift()*

1. *Erzeuge einen neuen Stackeintrag auf dem DAG, der mit einer leeren DAG-Knotenmenge initialisiert ist.*
2. *Gruppiere SHIFTS nach den Wortkategorien CAT_i, welche die verschiedenen Möglichkeiten für das zu parsende Wort darstellen, in $SHIFTS_{CAT_1}$ bis $SHIFTS_{CAT_n}$.*
3. *Für alle $SHIFTS_{CAT_i}$:*
   (a) *Erzeuge einen Knoten NODE im Parse Forest mit der jeweiligen Kategorie CAT_i und einer Referenz auf das zu parsende Wort.*
   (b) *Gruppiere die DAG-Knoten aus $SHIFTS_{CAT_i}$ nach den verschiedenen Zuständen, die sich aus dem Tabellenlookup des jeweiligen DAG-Knoten-Zustandes und der CAT_i ergeben, in verschiedenen W_SETs (analog Def. 2.3 Schritt 4).*
   (c) *Für alle W_SET_k:*
      i. *Erzeuge einen neuen Link L mit einer Referenz auf NODE und W_SET_k als seiner DAG-Knotenmenge.*
      ii. *Erzeuge einen neuen DAG-Knoten ν mit $\{L\}$ als seiner Menge von Links und k als seinem neuen Zustand.*
      iii. *Addiere ν zu Top(DAG).*

Akzeptanz einer Eingabe

Wenn anhand des Satzendezeichens, eines DAG-Knotens und dem dazugehörigen Tabelleneintrag die Akzeptanz des Satzes festgestellt wird, dann muß dies lediglich vermerkt werden, und alle Parse-Knoten der Links, die von dem zu akzeptierenden DAG-Knoten ausgehen, müssen als mögliche Wurzeln des Parse Forest registriert werden.

2.3 Der Parse Forest

Im Parse Forest wird, wie auf Seite 11 bereits angeführt, das Ergebnis der syntaktischen Analyse effizient festgehalten. Durch die kombinierten Prinzipien von *Subtree Sharing* und *Local Ambiguity Packing* können Speicherbedarf und Konstruktionszeit polynomial in Abhängigkeit von der Länge der Eingabe gehalten werden.

Subtree Sharing

Das Prinzip beim Subtree Sharing ist, daß ein untergeordneter Parse-Knoten von mehreren übergeordneten referenziert werden kann, anstatt daß jeder übergeordnete Knoten auf individuelle Subknoten verweisen muß.
In Abb. 2.13 z. B. wird richtigerweise die rechte Ableitung ($D \rightarrow d$) von verschiedenen übergeordneten Knoten referenziert und muß nur einmal im Parse Forest kreiert und gespeichert werden.
Dagegen könnten in Abb. 2.11 die mittlere und die rechte Ableitung von ($D \rightarrow d$) mit einer einzigen Ableitung subsumiert werden, wenn korrektes Subtree Sharing vorläge.

Local Ambiguity Packing

Beim Local Ambiguity Packing werden verschiedene Teilableitungsbäume zu einem Parse-Knoten zusammengefaßt, falls sie die gleiche Kategorie besitzen und den gleichen Abschnitt aus der Eingabe abdecken.
In Abb. 2.11 wird Local Ambiguity Packing benutzt, da die Ableitungsbäume, die "S" als Wurzel haben, in einem Knoten subsumiert werden.
Die zwei dick umrandeten "D"s hingegen dürfen nie zu einem Knoten zusammengefaßt werden. Sie besitzen zwar die gleiche Kategorie, aber überdecken nicht den gleichen Eingabezeitraum. Durch das Packen in einen Knoten wären die jeweils überdeckten Zeiträume — die jeweiligen "d"s — nicht mehr voneinander zu unterscheiden, so daß falsche Ableitungsbäume aus dem Parse Forest ausgelesen würden.

2.4 Fehlerhafter Aufbau des Parse Forest

Sowohl die formale Spezifikation als auch die Implementation in Tomita's Dissertation sorgen sich nicht ausreichend um ein Problem: die maximale Strukturteilung im Parse Forest mittels bestmöglichem Subtree Sharing.
Im Gegensatz zur Spezifikation in Kap. 2.2.2 wird im Parse Forest nicht ein Parse-Knoten pro "Bundle" erzeugt, welches reduziert wird, sondern ein Parse-Knoten pro W_SET, welches reduziert wird (Ausnahme: bei lokaler Ambiguität, die wie in Def. 2.4 Schritt 2a erkannt wird).
Das ließe aber bei der Grammatik in Abbildung 2.9 bei Eingabe von "d d"[8] folgendes geschehen (siehe dazu Abb. 2.10):

- Das zweite "d" würde – wie gewünscht — nur einmal zu "D" reduziert.

[8] Hier wird angenommen, daß Eingabewort und Kategorie identisch sind.

Grammatik $\mathcal{G}$
Startsymbol S
Nichtterminale $N = \{d\}$
Terminale $T = \{S, C, D, E, F\}$

$$\begin{array}{llcl}
0. & S & \rightarrow & EC \\
1. & S & \rightarrow & EDDDD \\
2. & S & \rightarrow & IC \\
3. & S & \rightarrow & IDDDD \\
4. & E & \rightarrow & \varepsilon \\
5. & I & \rightarrow & \varepsilon \\
6. & C & \rightarrow & dD \\
7. & C & \rightarrow & F \\
8. & F & \rightarrow & DD \\
9. & D & \rightarrow & d
\end{array}$$

Abbildung 2.9: Grammatik 3

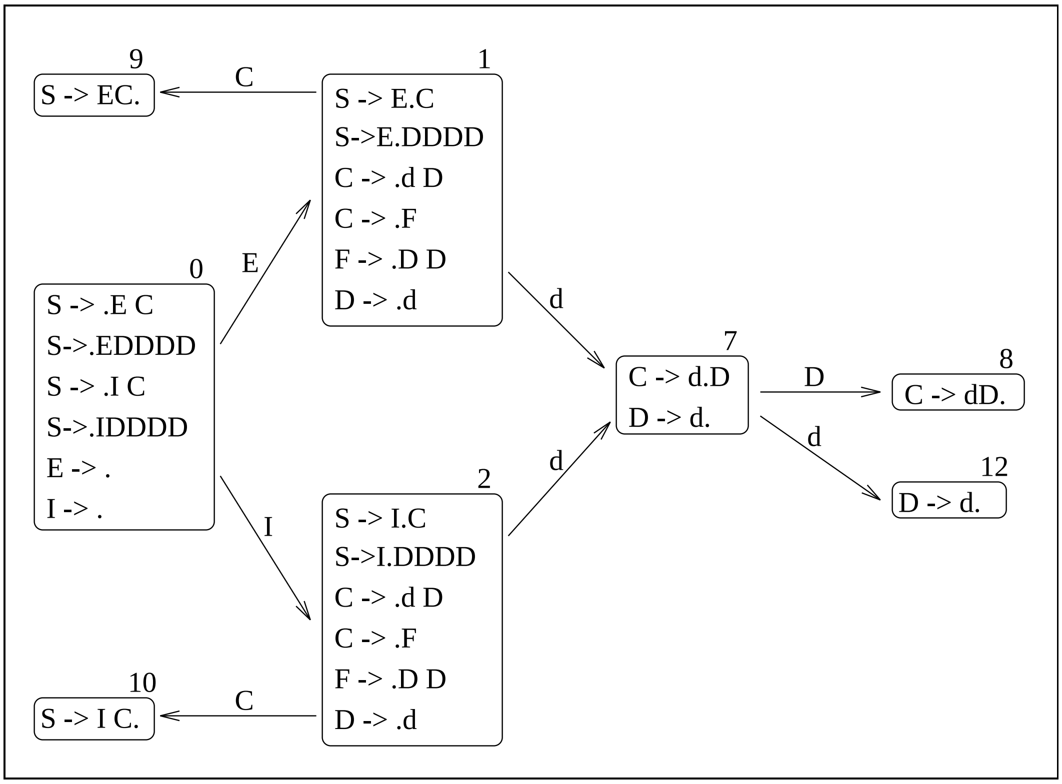

Abbildung 2.10: Teil des Zustandübergangsdiagramm zur Grammatik in Abb. 2.9

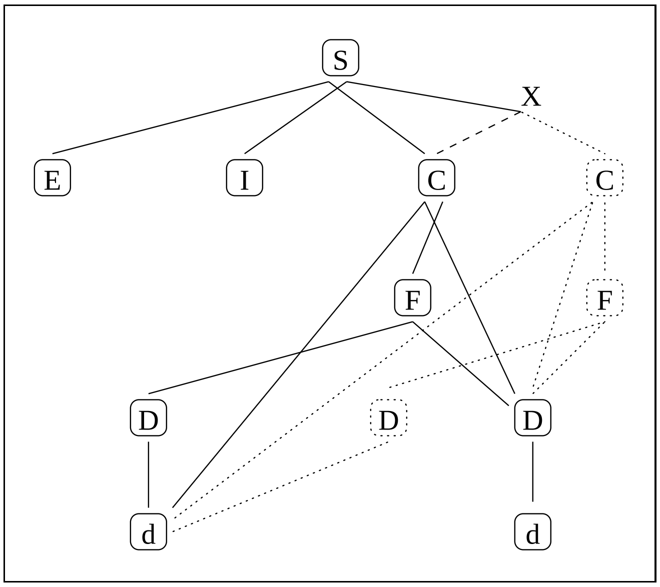

Abbildung 2.11: Parse Forest mit mangelhaftem Subtree Sharing (1)
Alle gepunkteten Referenzen und gepunktet umrahmten Symbole werden überflüssigerweise kreiert. Wenn an der mit "X" markierten Stelle die Referenz entlang der gestrichelten Linie verlaufen würde anstatt entlang der gepunkteten, dann wäre der Parse Forest minimal.

- Anschließend würde "d D" zu "C" reduziert. Da aber die Vorgängerknoten des DAG-Knoten mit Zustand 7 bei der Reduktion zu "C" zwei verschiedenen W_SETs angehören (W_SET_9 bzw. W_SET_{10}), würden zwei Parse-Knoten erzeugt, obwohl einer völlig ausreichend wäre.

Der Parse Forest bei Eingabe "d d" würde dann aussehen, wie in Abb. 2.11 — inklusive aller überflüssigen (gepunkteten) Teile.

In seinem Beispieltrace in [Tom85], Kapitel 3 benutzt Tomita demzufolge auch eine leicht geänderte Strategie. Anstatt für jedes W_SET einen eigenen Parse-Knoten einzuführen wird nur ein Parse-Knoten für jedes Paar, (P_i, L_i), aus Pfad und Link eingeführt.

Auf diese Weise wurden auch der sequentielle Parser und der traditionelle Lattice-GLRP dieser Studienarbeit realisiert und auf diese Weise erfolgte auch die Beschreibung in Kapitel 2.2.2 in dieser Studienarbeit. Allerdings kann diese Strategie bei Grammatiken wie in Abb. 2.9 erneut zu einem fehlerhaften Aufbau des Parse Forest führen.[9]

Das Problem wird deutlich an Abb. 2.12:

- "$C \rightarrow dD$" wird nun korrekterweise nur einmal reduziert, da die verschiedenen W_SETs keine Differenzierung bewirken.

- Die Reduktion "$F \rightarrow DD$" erfolgt aber nach wie vor doppelt, da hierbei völlig unterschiedliche DAG-Knoten betroffen sind — das eine Mal wird der DAG-Knoten mit

[9]Was ich allerdings sehr spät erst erkannt habe, da der Fehler nicht eben häufig auftritt und gerade bei kleinen Testbeispielen sehr rar ist. Deutlich wurde er in meinen Versuchen erst durch die systematische Änderung der Reduktionsreihenfolgen im Agendaverfahren.

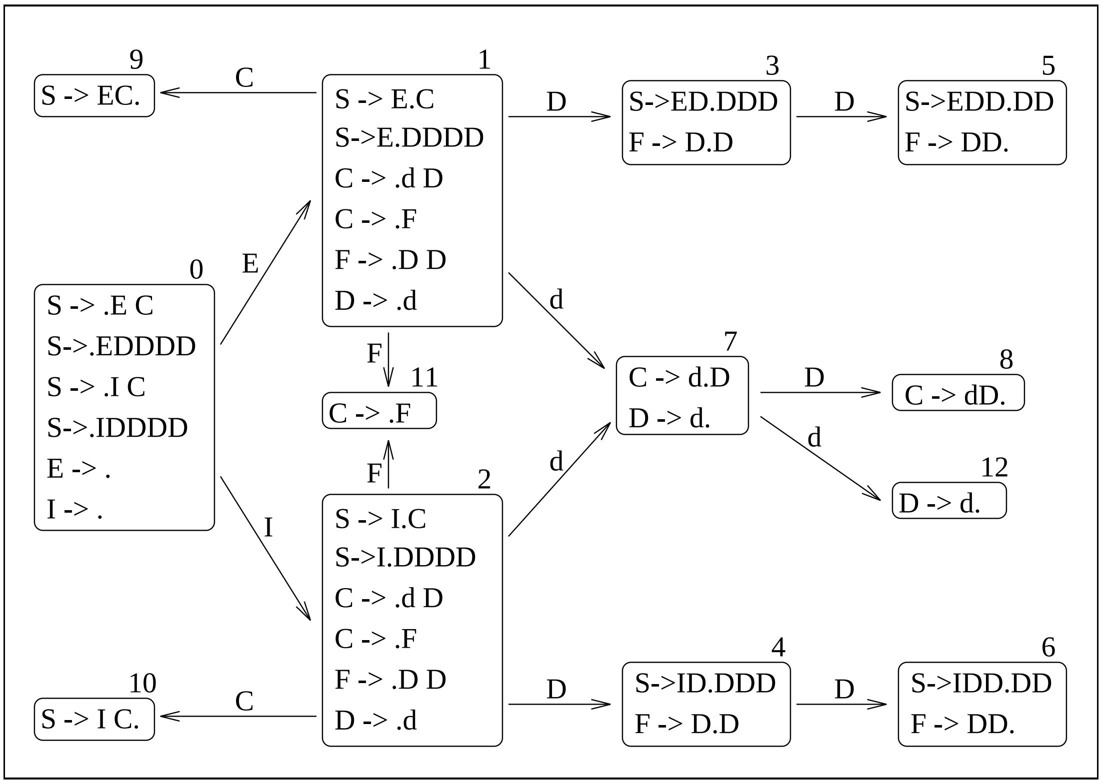

Abbildung 2.12: Teil des Zustandübergangsdiagramm zur Grammatik in Abb. 2.9 (Erweitert im Vergleich zur Abb. 2.10)

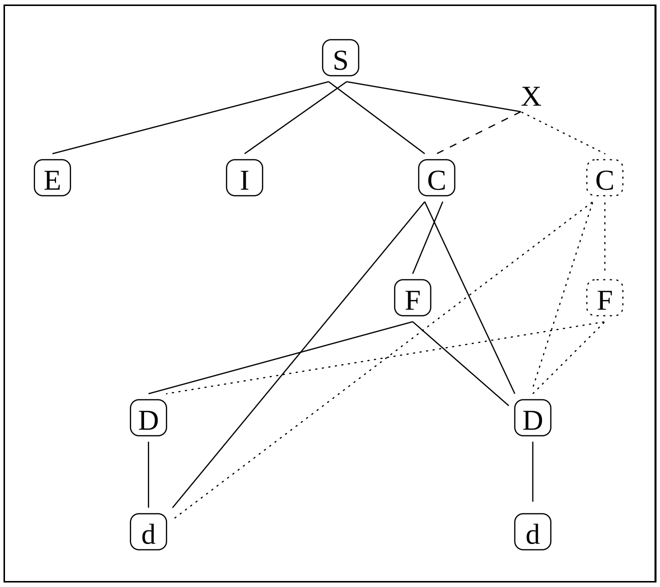

Abbildung 2.13: Parse Forest mit mangelhaftem Subtree Sharing (2)
In diesem Parse Forest gibt es zwei "C"-Knoten. Das ist kein Widerspruch zur Behauptung das "$C \rightarrow dD$" nur einmal reduziert wird, da die doppelte Reduktion von "$C \rightarrow F$" diese Verdopplung bewirkt hat.

Zustand 5 reduziert, das andere Mal der DAG-Knoten mit Zustand 6. Das bedeutet, daß eine Erkennung eines solchen gemeinsamen Subtrees *nur anhand des DAGs* von vorneherein ausgeschlossen ist.

Die zunächst völlig unbedeutenden Regeln "$S \rightarrow EDDD$" bzw. "$S \rightarrow IDDD$" sorgen nämlich dafür, daß die beiden Versionen von "$F \rightarrow DD$" getrennt im DAG behandelt werden müssen.

- Außerdem kann jetzt die Entdeckung der lokalen Ambiguitäten "$C \rightarrow dD$" und "$C \rightarrow F$" zusätzliches Durcheinander in den Parse Forest bringen. Je nachdem welche der beiden Reduktionen zuerst ausgeführt werden, ergibt sich ein Parse Forest wie in Abb. 2.13 oder wie in Abb. 2.14. Beide sind nicht minimal und in Abb. 2.14 erhält man sogar bei trivialer Aufzählung der Parse Bäume zwei der vier Bäume doppelt.

Resümee:

Es wird also deutlich, daß es keine Möglichkeit gibt, lokal im graph-strukturierten Stack eine Entscheidung darüber zu treffen, ob ein Parse-Knoten im Forest bereits existiert oder nicht.[10] Die logische Folgerung ist daher, daß im Parse Forest selbst getestet werden muß, ob ein Knoten mit passender Kategorie, Anfangs- und Endezeit existiert. Allein diesem Ergebnis

[10] Die einzige mir bekannte PD-Version eines Tomita-Parsers von Mark Hopkins (anonymous ftp at iecc.com, /pub/file/tomita.tar.gz) benutzt ebenfalls eine Suche im Parse Forest, um herauszufinden, ob der entsprechende Eintrag bereits existiert. Mir ist kein Artikel bekannt, der sich direkt mit diesem Problem beschäftigt. Allerdings wird in [BC93] diese Schwierigkeit angesprochen, weil dort durch mehrfache Reduktionen über gleiche Parse-Knoten-Pfade unnötige, zusätzliche Unifikationen entstanden.

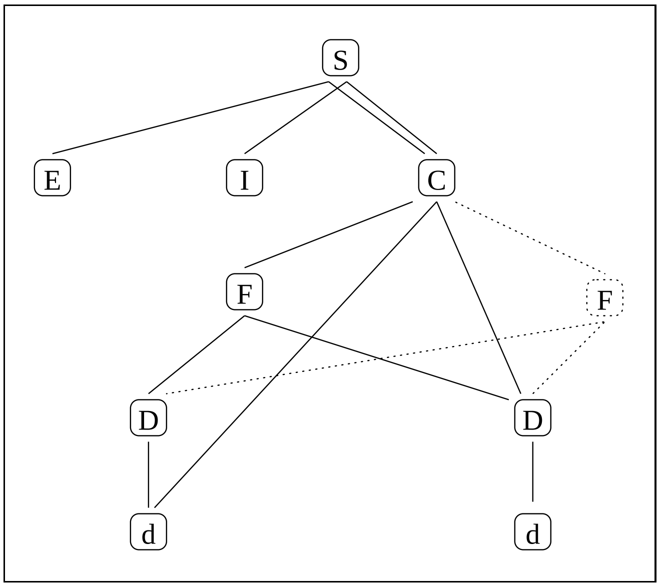

Abbildung 2.14: Parse Forest mit mangelhaftem Subtree Sharing (3)

zufolge kann der Parse Forest erweitert werden. Trotzdem ist der Redundanzcheck im DAG nicht unnötig, da auch dort das Kriterium der maximalen Strukturteilung beibehalten werden sollte.

Kapitel 3

Lattice-Parsing

3.1 Schnittstelle

Die Eingabedaten für einen Parser für gesprochene Sprache unterscheiden sich erheblich von der sequentiellen Eingabe, mit der das Parsing in Kapitel 2 beschrieben wurde.

Das Sprachsignal wird zunächst von einem Decoder in eine Menge von Symbolen umgewandelt. Diese Symbole, die z.B. Silben oder Wortformen beschreiben, konkurrieren untereinander. D.h. der Decoder wird im Allgemeinen nicht in der Lage sein dem akustischen Signal eindeutig eine symbolische Beschreibung zuzuordnen.

Wenn die konkurrierenden Symbole Wortformen repräsentieren, dann kann die Ausgabe eines Decoders wie in Abb. 3.1 veranschaulicht werden.

Diese Darstellung zeigt ein **Wort-Lattice** : die allgemeinste Form einer Schnittstelle zwischen einem Worthypothesenerkenner und einem Parser. Das Lattice ist definiert als eine Menge von Worthypothesen in einem gegebenen Zeitraum. Jede Worthypothese ist ein Tripel <from,to,key>, welches aus Wortanfangszeitpunkt, Wortendezeitpunkt und Schlüssel zum Lexikon besteht.

Andere mögliche Schnittstellen sind z.B.:

- **Ein verbundener Wortgraph:** Ein solcher Wortgraph hat einen Anfangsknoten, von dem aus Verbindungen zu allen nachfolgend möglichen Worthypothesen[1] gehen und mindestens eine Verbindung jeder Worthypothese zu einer nachfolgenden Worthypothese oder zum Endknoten.

- **N-Beste Ketten[2]:** Viele Verfahren extrahieren aus dem Lattice zunächst die n vielversprechendsten Ketten, um sie nacheinander in einem üblichen sequentiellen Parser zu analysieren.

Wichtige Varianten der Kopplung von Decoder und Parser bestehen aus interagierenden Ansätzen, wie sie Hauenstein & Weber [HW94] beschreiben. Im folgenden werden diese nicht berücksichtigt werden, aber die Prediktionsmöglichkeiten der Parsingtabelle würden solche Ansätze unterstützen, wie z.B. in [KKS89] deutlich wird.

[1] Bzw. andere Darstellungen, falls der Decoder kein Worthypothesenerkenner ist.

[2] Vgl. Kap. 6

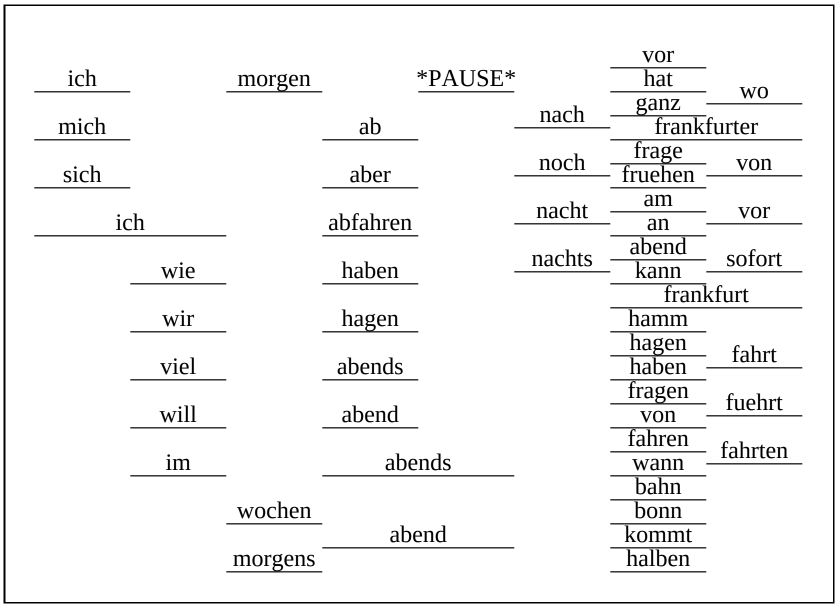

Abbildung 3.1: Ein Wort-Lattice: Eingabe an einen Parser

3.2 Parser

M. Tomita hat in [Tom86] aufgezeigt, wie sich der sequentielle GLR-Parser auf einen GLRP für Lattices erweitern läßt.

Die Eigenheiten der Lattice-Version zeigen sich vor allem anhand von zwei Punkten:

1. Es gibt Prädikate, um Verbundenheitsbedingungen innerhalb des Lattice während der Laufzeit des Parsers zu überprüfen:

   - *Adjacent?(w_1,w_2):* Testet, ob zwei Worthypothesen aufeinanderfolgen können. Im einfachsten Fall ist dies nur der Test, ob die Startzeit von w_2 gleich der Endzeit von w_1 ist. Mit komplizierteren Modellen könnte man z.B. bei Verschleifungen auch überlappende Worthypothesen in einer Worthypothesensequenz erlauben, oder Sprechpausen berücksichtigen, etc.
   - *Begin?(w):* Überprüft, ob w eine mögliche Anfangsworthypothese sein könnte. Dieser Test sollte "false" ergeben, wenn sich mit hoher Wahrscheinlichkeit ein anderes wichtiges akustisches Ereignis vor w ereignet hat.
   - *End?(w):* Überprüft, ob w eine Worthypothese sein könnte, die das Ende des Satzes darstellt. Analog zu "Begin?" sollte die Wahrscheinlichkeit, daß nach w noch etwas gesprochen wurde, sehr gering sein.

   Das Problem, wie diese Prädikate zur Verfügung gestellt werden können, wird hier nicht weiter diskutiert werden. Im weiteren wird angenommen werden, daß diese Tests nur aufgrund der Wortanfangs- und Wortendezeitpunkte der Hypothesen bewertet werden.

2. Es hat sich gezeigt, daß Lookahead-Strategien, die bei sequentiellen Parsern den Suchraum einengen, weil Reduktionen nur ausgeführt werden, wenn das nachfolgende Einga-

besymbol eine bestimmte Kategorie besitzt, bei Speech-Parsern nicht den gewünschten Effekt erzielen[3].

Die Ursache liegt darin begründet, daß die möglichen Nachfolger einer Hypothese so zahlreich sind, daß ohnehin fast jede mögliche Reduktion stattfindet.

Es bietet sich beim Lattice-Parser also an, die Steuerstruktur umzustellen:

1. Zuerst werden alle möglichen SHIFTS ausgeführt. Ob eine Schiebeaktion erfolgen kann, hängt dabei nicht nur von DAG-Knotenzustand und neuer Kategorie ab, sondern auch von den Verbundenheitsprädikaten, die in Punkt 1 beschrieben wurden.

   Eine Schiebeaktion mit dem Start-DAG-Knoten kann demzufolge nur erfolgen, wenn *Begin?(w)* "true" ist. Dagegen kann eine Schiebeaktion mit einem DAG-Knoten, der aus der Worthypothese w_1 entstanden ist, nur stattfinden, wenn *Adjacent?*(w_1,*w*) "true" ist.

2. Danach werden bei jedem neuen aktiven DAG-Knoten *alle* Reduktionen ausgeführt, die aufgrund des jeweiligen Zustands des DAG-Knoten und bei – hypothetisch – allen nächsten Eingabesymbolen in Frage kommen.

3. Als nächstes werden alle ε-Reduktionen ausgeführt, die aufgrund der jeweiligen Zustände der aktiven DAG-Knoten – analog zu den sonstigen Reduktionen – überhaupt möglich sind.

4. Schließlich wird ein DAG-Knoten akzeptiert, wenn die Worthypothese, aus der er entstanden ist, das Prädikat *End?* erfüllt.

Außer den beiden genannten Änderungen bieten sich zwei Implementationsalternativen an, wenn die Verbundenheitsprädikate so sind, daß Entscheidungen nur anhand von Wortanfangs- und endezeitpunkten getroffen werden:

Zum einen kann bei jedem Aufruf von "Shift()" ein neuer Eintrag im graph-strukturierten Stack erzeugt werden. Das erlaubt einfache Veränderungen der Verbundenheitsprädikate, denn DAG-Knoten, die aus einer bestimmten Hypothese resultieren, werden, wie bisher schon beim sequentiellen Parser, im gleichen DAG-Knoten abgespeichert wie die elementareren Knoten, aus denen sie entstanden sind. In Abb. 3.2 z.B. wird der DAG-Knoten nach "NP" im gleichen Stackeintrag gespeichert, wie der nach "pron", da "NP" aus "pron" entstanden ist. Die Prädikate *End?* bzw. *Adjacent?* können also einen Stackeintrag auf einmal testen, da ein Stackeintrag genau einer Worthypothese entspricht.

Zum anderen können die DAG-Knoten aber auch nur nach Zeitpunkten in den DAG einsortiert werden. Wenn die Prädikate sowieso nur auf Zeitpunkte testen, dann ist diese Methode wesentlich effizienter, da viele parallele Worthypothesen oft gleiche oder ähnliche Kategorien haben, die gleiche Aktionen auslösen und diese Aktionen in diesem zweiten Fall oft die gleichen DAG-Knoten betreffen, während bei ersterer Methode in der Regel verschiedene DAG-Knoten betroffen sind. Beim Vergleich der Abbildungen 3.2 und 3.3 wird deutlich, daß die Strukturteilung bei dem Diagramm in Abb. 3.3 wesentlich ausgeprägter ist, was sich vor allem bei kleinen Grammatiken und vielen parallelen Worthypothesen enorm auf die Parsegeschwindigkeit auswirkt. Im Extremfall wird bei größerer Strukturteilung für jede neue parallele Worthypothese

[3] Vgl. [Ney91]

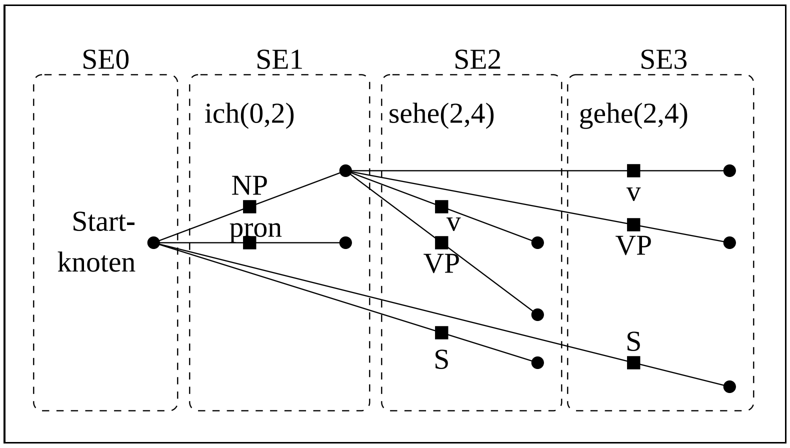

Abbildung 3.2: Ein Stackeintrag für jede Worthypothese

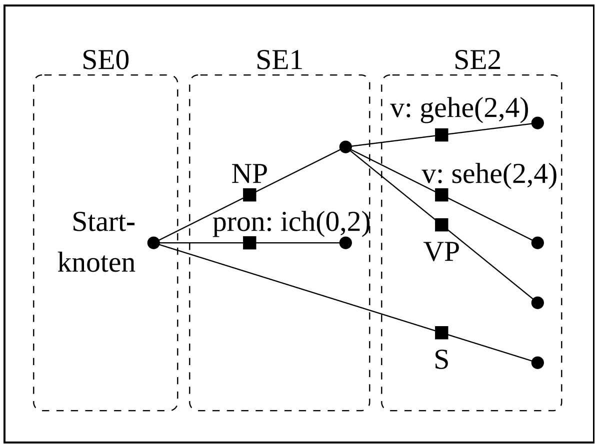

Abbildung 3.3: Ein Stackeintrag für jeden vorhandenen Wortendezeitpunkt

einfach ein Knoten in den Parse Forest eingebracht. Gleichzeitig entfällt jegliche Reduktion, weil die entsprechende Kategorie bereits geparst wurde, während bei geringerer Strukturteilung die gleichen Reduktionen immer wieder ausgeführt werden müssen.

Die erste Methode wird in der implementierten "einfachen" Lattice-Version benutzt, die zweite in der implementierten Agenda-Version. Generell sank die Parsezeit bei der zweiten Methode — bei einem Beispiel sogar von 10 Minuten auf 10 Sekunden. Allerdings ist dieses empirische Ergebnis nur ein Fingerzeig, da nicht nur bei dieser Wahlmöglichkeit Code verändert wurde.

Die theoretische Komplexität des GLR-Lattice-Parsers läßt sich analog zu [Web94] als $\mathcal{O}(n^{1+\rho})$ bestimmen, wobei n die Anzahl der Wortendehypothesen und ρ die Länge der längsten rechten Regelseite ist.

Kapitel 4

Statistische Methoden

Auch wenn sich im GLRP selbst viele Optimierungen finden lassen – und die realisierten Versionen sind da sicherlich noch sehr verbesserungsbedürftig – verbietet die typische Größe eines Wort-Lattice von ca. 500 bis 1000 Worthypothesen bei einer Komplexität von mindestens $\mathcal{O}(n^3)$ eine vollständige Analyse des Suchraums. Vor allem die angestrebte Benutzung einer Unifikationsgrammatik erzwingt, daß nicht zu viele grammatikalischen Operationen ausgeführt werden, da die einzelnen Unifikationsschritte sehr zeitaufwendig sind.

Spracherkennung und Sprachverstehen können aber effizient integriert werden, wenn statistische Information aus der Spracherkennung die weitere Suche in der Sprachverarbeitung steuert. Wie diese Steuerung aussehen kann, wird in den Kapiteln 6 und 7 dargelegt.

Die statistischen Methoden, die in dieser Arbeit benutzt werden, beruhen auf der folgenden Überlegung[1]:

> Maximiere die Wahrscheinlichkeit $P(w_1, \ldots, w_n \mid x_1, \ldots, x_T)$, daß die Wortfolge $w_1, \ldots, w_n$ gesprochen wurde, wenn die Akustikvektoren $x_1, \ldots, x_T$ erkannt wurden. (4.1)

Durch die Anwendung von Bayes Theorem über bedingte Wahrscheinlichkeiten kann man das Problem umformulieren: [2]

> Bestimme die Wortsequenz $\overline{w}$, die
> $$P(w_1, \ldots, w_n) \cdot P(x_1, \ldots, x_T \mid w_1, \ldots, w_n) \tag{4.2}$$
> maximiert.

Der erste Faktor, $P(w_1, \ldots, w_n)$, ist dabei vollständig unabhängig vom Sprachsignal und kann durch ein Sprachmodell beschrieben werden. Im folgenden werden dazu N-Grammatiken verwendet. Der zweite Faktor hingegen umfaßt nur die Bewertung des Sprachsignals.

4.1 Akustische Bewertungen

Durch die konkrete Einführung von Wortgrenzen ($[t_i]$ ist die Menge der entsprechenden Zeitpunkte) kann man schreiben:

$$P(x_1, \ldots, x_T \mid w_1, \ldots, w_n) = \sum_{[t_i]} P(x_1, \ldots, x_T, t_1, \ldots, t_n \mid w_1, \ldots, w_n). \tag{4.3}$$

[1] Vgl. [Ney93]

[2] Diese Betrachtung ist nur eine Näherung, da die beiden Teile nicht wirklich stochastisch unabhängig sind.

Wenn es keine Interdependenzen gibt, kann man statt dessen formulieren:

$$\sum_{[t_i]} \prod_{i=1}^{n} P(x_{t_{i-1}+1}, \ldots, x_{t_i} \mid w_1, \ldots, w_n). \tag{4.4}$$

Was gleichbedeutend ist mit

$$\sum_{[t_i]} \prod_{i=1}^{n} P(x_{t_{i-1}+1}, \ldots, x_{t_i} \mid w_i), \tag{4.5}$$

da nur w_i die Akustikvektoren im Zeitraum $(t_{i-1}+1, t_i)$ beeinflußt. Eine Maximumsapproximation liefert die Näherung

$$P(x_1, \ldots, x_T \mid w_1, \ldots, w_n) \approx \max_{[t_i]} \prod_{i=1}^{n} P(x_{t_{i-1}+1}, \ldots, x_{t_i} \mid w_i). \tag{4.6}$$

Der Decoder, welcher sehr häufig auf Hidden Markov Modells beruht, ermöglicht es nun, diese stochastischen Überlegungen im Sprachverstehensprozeß zu benutzen, indem er nicht nur mögliche Worthypothesen erkennt, sondern zu den jeweiligen Worthypothesen auch die Wahrscheinlichkeit $P(\overline{x_i} \mid w_i)$ bestimmt.

Hidden Markov Modells haben sich dabei als sehr erfolgreich erwiesen, weil kein explizites Modell vorgegeben werden muß, sondern die Wortmodelle anhand eines Testsets trainiert werden können. Einführungen zu diesem Thema findet man z.B. in [RJ86] oder in [Rab88].

Für die Schnittstelle zwischen Decoder und Parser bedeutet das, daß Worthypothesen nun als 4-tupel <from,to,key,score> definiert werden, wobei der letzte Eintrag die Wahrscheinlichkeit $P(\overline{x}|w)$ beschreibt.

4.2 N-Grammatiken

Die Wahrscheinlichkeit $P(w_1, \ldots, w_n)$ läßt sich aufgliedern in

$$P(w_n \mid w_1, \ldots, w_{n-1}) \cdot P(w_{n-1} \mid w_1, \ldots, w_{n-2}) \cdot \ldots \cdot P(w_2 \mid w_1) \cdot P(w_1). \tag{4.7}$$

Um die Wahrscheinlichkeiten $P(w_i \mid w_1, \ldots, w_{i-1})$ approximativ zu bestimmen, verwendet man nur die letzten k Worte. In der Praxis ist $k \leq 3$, da ansonsten der Speicherverbrauch zu sehr ansteigt, selbst wenn – was häufig bei größeren Lexika passiert – anstatt von Wortformen Wortkategorien zur Bestimmung der Wortsequenzwahrscheinlichkeit herangezogen werden.

In den folgenden Implementationen werden ausschließlich Bigramme verwendet, die $P(w_i \mid w_1, \ldots, w_{i-1})$ wie folgt approximieren:

$$P(w_i \mid w_1, \ldots, w_{i-1}) \approx P(w_i \mid w_{i-1}). \tag{4.8}$$

Die Wahrscheinlichkeiten $P(w_i \mid w_j)$ werden ermittelt, indem in einer Stichprobe die Häufigkeit des Aufeinanderfolgens von zwei Worten ausgezählt werden:

$$P(w_i \mid w_j) \approx \frac{\#(w_j, w_i)}{\#(w_j)} \tag{4.9}$$

Nachdem alle möglichen Kombinationen ausgewertet worden sind, muß noch berücksichtigt werden, daß auch nichtbeobachtete Wortfolgen später auftauchen können und diese daher nicht die Wahrscheinlichkeit 0 haben dürfen. Die Aufgabe des *Smoothing-Prozesses* ist es deshalb, diesen nicht beobachteten Kombinationen einen geeigneten Wert zwischen 0 und $\min_{i,j} \frac{\#(w_j,w_i)}{\#w_j}$ zuzuweisen und anschließend alle Werte so zu korrigieren, daß $\forall j : \sum_i P(w_i|w_j) = 1$.
Die Kombination von akustischer Bewertung (Gl. 4.6) mit einem N-Gram Modell (Gl. 4.8) in Gleichung 4.2 erlaubt es nun, die Wahrscheinlichkeit einer Wortsequenz näherungsweise anzugeben. Diese Wahrscheinlichkeit wird im weiteren benutzt werden, um den Parsing Prozeß zu steuern.

Kapitel 5

Agenda-gesteuertes GLR-Parsing

Wie in Kapitel 4 bereits ausgeführt wurde, ist es nicht möglich, den Raum aller möglichen Worthypothesensequenzen durch ein großes Lattice vollständig zu durchsuchen, und es wurde auch bereits ein Bewertungskriterium genannt, nach welchem die Suche gesteuert werden könnte. Ungeklärt ist bisher aber noch, wie sich ein derartiges Kriterium zur Steuerung eines GLR-Parsers einsetzen ließe, der zunächst eine strikt links-rechts gerichtete Abarbeitung impliziert.

Einen Ansatz gibt es von L. Schmid [Sch94], der Parser und Bewertung nicht vollständig integriert, sondern zunächst eine A*-Suche über den vollständigen Lattice betreibt, um anschließend einzelne Pfade in linkskompaktierter Form mit einem sequentiellen GLR-Parser zu analysieren.[1]

Ein weiterer Ansatz, der sich zwar nur mit sequentiellem Parsing beschäftigt, aber sicherlich erweiterungsfähig wäre, wird in [SJNWC91] von Su, Wang, Su and Chang vorgestellt. Dort wird eine Bestensuchstrategie angewendet, die sich an den nichtdeterministischen Wahlpunkten zwischen Shift-Reduce- bzw. Reduce-Reduce-Konflikten mittels einer Bewertungsfunktion für einen Weg entscheidet und später evtl. backtrackt, um bei einem sequentiellen Parse möglichst schnell eine erste Analyse zu erhalten.

Dieses sind gut funktionierende Methoden, die aber in ihren Möglichkeiten beschränkt sind, weil sie vergleichsweise rigide Ablaufstrukturen in der Problembearbeitung erfordern.

Beim Chartparsing hingegen wird fast immer die viel flexiblere Form einer Agendastrategie[2] gewählt. Die Agenda, als zentrale Datenstruktur, welche die Abarbeitungsreihenfolge von Problemen und deren Subproblemen steuert, erlaubt es, Strategien und Bewertungsfunktionen einfach zu variieren, und sogar während des Programmlaufs, die Strategie zu ändern. Diese Flexibilität benötigt einen vernachlässigbar kleinen Verwaltungsmehraufwand und zeitigt daher keine Leistungseinbußen.

Im folgenden soll nun eine Agendastrategie für GLR-Parsing vorgestellt werden, die den Vorteil der Flexibilität eines Chartparsers mit dem Vorteil des GLR-Parsings, nämlich der effizienten Nutzung vorkompilierter Information, verbindet und dabei die Funktionalität der in [SJNWC91, Sch94] vorgestellten Arbeiten subsumieren kann.

---

[1] Genauer siehe Kap. 6.1

[2] Siehe [Gö88] zur Flexibilität einer Agendastrategie beim Chartparsing.

5.1 Vorüberlegungen zur Agendasteuerung

Beim Entwurf einer Agendasteuerung für GLRP müssen vor allem vier ineinandergreifende Punkte berücksichtigt werden:

1. **Design:** Es muß geklärt werden, welche Agendaaktionen auftreten können, da die Festlegung dieser Aktionen die Granularität bestimmt, mit der die Steuerung durch die Agenda erfolgen kann bzw. muß[3].

2. **Korrektheit:** Die Aktionen müssen ausformuliert werden.

3. **Effizienz:** Die Effizienz darf durch neue Ablaufstrukturen nicht beeinträchtigt werden.

4. **Vollständigkeit:** Die Interaktion der atomaren Aktionen muß beschrieben werden. Welche Aktion löst welche Folgeaktionen aus?

Diese Überlegungen spiegeln sich — grob betrachtet — in den folgenden 3 Sektionen wieder. Dabei werden — ausgehend von den ersten Designentscheidungen — die Veränderungen des traditionellen Lattice-GLRP schrittweise beschrieben, die schließlich zu einer korrekten, effizienten und vollständigen Agendaversion führen.

5.2 Elementare Aktionen

Durch die bisherige Struktur des GLRP-Verfahrens bietet es sich an, folgende Agendaaktionen zu verwenden:

- Reduktion
- SchiebeTerminal
- ε-Reduktion
- Akzeptanz

Damit eine Analyse des Lattice in flexibler Reihenfolge möglich ist, ist es sinnvoll, auch eine

- NeueWorthypothese

als eigenständige Aktion auf die Agenda legen zu können.

Da der DAG nicht mehr nur auf seiner rechten Seite wächst, sondern nun auch *im* DAG neue Knoten und Links hinzukommen können, ist es ferner notwendig, die Auswirkungen zu bedenken:

1. *Neue DAG-Knoten* interagieren nicht mit anderen DAG-Knoten. Sie können aber sowohl neue Reduktionen veranlassen — welche dann nach dem bisherigen Schema ablaufen — als auch neue Schiebeaktionen, da es ja bereits Worthypothesen rechts des neuen DAG-Knoten geben kann.

[3]Für die spätere Steuerung mittels statistischer Information und — als Fernziel — die Benutzung einer Unifikationsgrammatik darf die Strukturierung nicht zu grobmaschig geraten, weil dadurch Flexibilität verlorenginge. Andererseits können sinnvolle Strategien nur implementiert werden, wenn die Anzahl der Operationen überschaubar bleibt.

2. *Neue Links* an neuen DAG-Knoten erfordern keine neuartige Behandlung, aber neue Links *an schon vorhandenen DAG-Knoten* müssen gesondert berücksichtigt werden. Es ist nämlich möglich, daß bereits ein oder mehrere Reduktionen den DAG-Knoten und die vorhergehenden Links passiert haben. Diese Reduktionen konnten dann natürlich noch nicht den Weg über den gerade erst hinzugefügten Link genommen haben.

   Deswegen muß nun festgestellt werden, welche Reduktionen bereits erfolgt sind, und die entsprechenden Reduktionen müssen nun auch über den neuen Link hinweg erfolgen.

   Ist allerdings der Link der letzte im Reduktionspfad, so ist keine weitere Suche mehr nötig. In diesem Spezialfall genügt die Reduktion eines "Bundle"[4].

Deshalb würde es sich empfehlen, "ReduceBundle" als weitere Aktion in der Agenda zu erlauben. Für die Bewertungsstrategie in Kapitel 7 und für eine einheitliche Betrachtung des GLRP ist es aber sinnvoller, eine Subaktion von "ReduceBundle" als Agendaaktion zu verwenden:

- SchiebeNichtTerminal

Eine solche Aktion wird für jeden Vorgänger-DAG-Knoten des Links in "ReduceBundle" ausgelöst. Wie diese den Ablauf verändert, wird im folgenden Abschnitt beschrieben.

5.3 Korrektheit und Effizienz

Ähnlich wie beim Chartparser sollte die Einführung einer Agendastrategie keine drastische Verschlechterung des Parserlaufzeitverhaltens, oder gar einen fehlerhaften Algorithmus zur Folge haben. Beim Entwurf eines agendagesteuerten Ansatzes muß also bedacht werden, daß keine der Agendaaktionen sich wesentlich schlechter verhalten darf als ihr Pendant in der strikten links-rechts Version, und gleichzeitig die Korrektheit bewahrt werden muß.

Um dies zu belegen, werden in den Tabellen 5.1 und 5.2 die Aktionen verglichen, die kritisch sind, weil dort Strukturen verändert werden.

Zum einen wird dabei deutlich, daß die neue Version nur Strukturen im DAG bzw. im Parse Forest kreiert, die korrekte Unterstrukturen eines entsprechenden traditionellen GLRP wären. Zum anderen zeigt sich, daß sich der einzige Mehraufwand aus der Umdefinition der Reduktion und der Aktion SchiebeTerminal ergibt[5]:

- Reduktionen, die einen DAG-Knoten passieren, müssen dort vermerkt werden. Da aber der Mehraufwand pro Reduktion und besuchtem DAG-Knoten konstant ist, verschlechtert sich das Laufzeitverhalten des Parsers höchstens um einen kleinen Faktor.

  Daß sich der Speichermehraufwand in Grenzen hält, begründet sich wie folgt:

  – Jeder reduzierte Pfad hat einen Eintrag im Parse Forest. Die Worst Case Betrachtung geht davon aus, daß sich jeder Eintrag von jedem anderen Eintrag unterscheidet und somit (zumindest bei einem gewöhnlichen Parse Forest) kein Eintrag zwei verschiedene Pfade durch den DAG repräsentieren kann.

  – Jeder Pfadeintrag besteht aus so vielen Referenzen, wie der Pfad lang ist.

[4] Vgl. Def. 2.3

[5] Vergleiche dazu immer Abb. 5.3.

Agendaaktion	Funktionsweise des GLR-Lattice-Parsers ohne Agenda	Funktionsweise der Agendaversion
NeueWorthypothese	Eine neue Worthypothese mit Anfangszeitpunkt t_i wird bearbeitet, indem versucht wird, alle DAG-Knoten mit Zeitstempel t_i mit allen Kategorien der neuen Worthypothese zu kombinieren. Das löst neue SchiebeTerminals aus. Die Wahl der Abarbeitungsreihenfolge sorgt dafür, daß alle in Frage kommenden SchiebeTerminals auch festgestellt und ausgeführt werden.	Im Prinzip unverändert. Aber die Suche nach den entsprechenden DAG-Knoten beschäftigt sich nicht mehr mit den obersten Einträgen im Stack, sondern die Stack-Struktur verschwindet. Dafür werden die DAG-Knoten in Einträgen gehalten, die über Zeitpunkte t_j referenziert werden können. Außerdem muß die Worthypothese jetzt im DAG abgespeichert werden, damit getestet werden kann, ob eine SchiebeTerminal-Aktion mit dieser Worthypothese notwendig wird, wenn ein neuer DAG-Knoten *im DAG* entsteht. Durch die neue DAG-Struktur bedeutet das aber keinen Mehraufwand, mit Ausnahme des geringen Speicherbedarfs für die Worthypothesen.
SchiebeTerminal	Die SchiebeTerminal-Aktionen, die durch eine Worthypothese ausgelöst werden, werden nach der zu schiebenden Kategorie und nach Folgezustand sortiert.	Die Sortierung geschieht nicht mehr zentral bei Aufruf von "Shift()". Stattdessen wird jede SchiebeTerminal-Aktion einzeln auf die Agenda gelegt. Wenn ein SchiebeTerminal von der Agenda geholt wird, wird überprüft, ob eine ähnliche Aktion bereits erfolgt ist. Diese Überprüfung ist kein Mehraufwand im Vergleich zur früheren Sortierung.
ε-Reduktion	ε-Reduktionen werden fast wie SchiebeTerminals behandelt. Der Hauptunterschied zu SchiebeTerminal-Aktionen ist, daß hier Nichtterminale mit Eingabelänge 0 und ohne weitere Referenz geschoben werden.	Die Änderung verläuft analog zu "Shift()" und SchiebeTerminal.
Akzeptanz	Die Parseknoten der Links des akzeptierten DAG-Knoten werden als Wurzeln des Parse Forest markiert.	Unverändert.

Tabelle 5.1: Vergleich von Agendaaktionen mit dem traditionellem Lattice-GLRP (1)

Agendaaktion	Funktionsweise des GLR-Lattice-Parsers ohne Agenda	Funktionsweise der Agendaversion
Reduktion	Ausgehend von dem DAG-Knoten, der mit Regel R reduziert wird, wird rückwärts gesucht, um alle Paare (P_i, L_i) zu finden (Vgl. Def.2.2). Diese werden dann mit ReduceBundle reduziert.	Im Prinzip unverändert. Bei der Suche rückwärts wird aber zusätzlich an jedem DAG-Knoten abgespeichert, welche Reduktionen den Knoten bereits passiert haben. Dies ist nötig, damit die Effizienz nicht durch weiteres Suchen verschlechtert wird. Die Rückwärtssuche ist nämlich ein Prozeß, der bei Effizienzbetrachtungen sehr negativ ins Gewicht fällt (Vgl. dazu [Kip91]). Die Einzelschritte werden in Abb. 5.3 erklärt.
SchiebeNicht-Terminal Ausgangs-situation: ReduceBundle	Die möglichen Vorgängerknoten von L_i werden in W_SETs partitioniert. Die Reduktionen einer Regel, ausgehend von den DAG-Knoten eines W_SET, resultieren in einem gemeinsamen Zustand (Vgl. Def. 2.3).	Analog zum SchiebeTerminal erfolgt keine zentrale Aufteilung in W_SETs. Jeder Vorgänger-DAG-Knoten des letzten Links in der Reduktion löst eine eigene SchiebeNichtTerminal-Aktion aus. Bei deren Abarbeitung wird zuerst überprüft, ob bereits eine ähnliche Aktion stattgefunden hat. Da auch der Link L_i jederzeit um einen Vorgänger erweiterbar sein muß, ist es außerdem nötig die Reduktion, die die SchiebeNichtTerminal-Aktionen ausgelöst hat, hier zusammen mit den resultierenden DAG-Knoten abzuspeichern.

Tabelle 5.2: Vergleich von Agendaaktionen mit dem traditionellem Lattice-GLRP (2)

  - Eine Reduktion, die einen DAG-Knoten passiert, veranlaß37t die Speicherung eines Quadrupels. Dieses Quadrupel besteht aus dem Zeitpunkt des Ergebnis-DAG-Knotens, der Regel, dem "car" des aktuellen Pfades und dem "cdr" des aktuellen Pfades.
  - Zwischen jedem solchen Quadrupel besteht eine bijektive Beziehung zu einer Referenz eines Pfadeintrags im Parse Forest. Also ist der Speichermehraufwand kleiner als die vierfache Größe des Parse Forest.

- Eine Reduktion, die einen Link passiert, welcher bei der Konstruktion eines Pfades der letzte ist, muß in diesem Link ebenfalls vermerkt werden — zusammen mit den jeweiligen Paaren aus neuem DAG-Knoten und neuem Link, die aus dieser Reduktion mit diesem Link als letztem Glied entstanden sind.

  Da jeder Link im DAG nur einmal neu erstellt wird, wird das Tripel <Reduktionsregel, neuer DAG-Knoten, neuer Link> weniger als einmal pro existierendem Link abgespeichert. Der Speichermehraufwand ist demnach wesentlich kleiner als die dreifache Größe des traditionellen DAGs.

Zusammenfassend kann man also sagen:

Satz 5.1 *Des Laufzeitverhalten der Agendaversion des GLRP ist $\mathcal{O}(n^{1+\rho})$. Der Speicheraufwand ist kleiner als der fünffache Aufwand für die bisherige Lattice-Version.*

5.4 Erzeugung von Aktionen

Im vorhergehenden Abschnitt wurde sichergestellt, daß alle erzeugten Strukturen korrekt sind. Nun müssen Überlegungen folgen, die garantieren, daß jede Struktur eines traditionellen GLRP auch von der Agendaversion kreiert werden kann:

Die Erzeugung und Verarbeitung neuer Aktionen im traditionellen Lattice-GLRP konnte stufenweise erfolgen, da keine rechten Kontexte benutzt wurden. Eine Worthypothese löste SchiebeTerminal-Aktionen auf dem DAG aus. Diese erzeugten neue DAG-Knoten, die evtl. im folgenden Reduktionszyklus benutzt werden konnten, und jede einzelne Reduktion erzeugte eine Reihe von ReduceBundles als Unteraktionen. In dieser Stufe gab es die einzige Rückkopplung, da ein ReduceBundle neue (ε-)Reduktionen auslösen konnte[6]. Die Verarbeitungsrichtung verlief entgegengesetzt zur Richtung der Erzeugung, wobei allerdings jede Stufe zunächst vollständig abgearbeitet wurde. Im Agendamodell — und somit unter eventueller Nutzung eines rechten Kontextes — gibt es neue Möglichkeiten zur Erzeugung von Aktionen (Abb. 5.2), kausale Verknüpfungen, die erweitert wurden und Kopplungen, die auch bereits im traditionellen Lattice-GLRP vorkamen. An die Stelle von "ReduceBundle" ist eine Menge von SchiebeNichtTerminal-Aktionen gerückt. Die Reihenfolge der Verarbeitung wird jetzt allein durch die Agendastrategie bestimmt.

Die neuen Möglichkeiten zur Erzeugung von Aktionen lassen sich am einfachsten danach charakterisieren, ob ein neuer DAG-Knoten kreiert oder ein alter benutzt wird. Beim traditionellen GLRP gab es diese Unterscheidung nur bedingt, da der DAG nur im obersten Stackeintrag erweitert wurde. Die "neuen" SchiebeTerminal-, SchiebeNichtTerminal-Aktionen

[6] Vgl. Abb. 5.1.

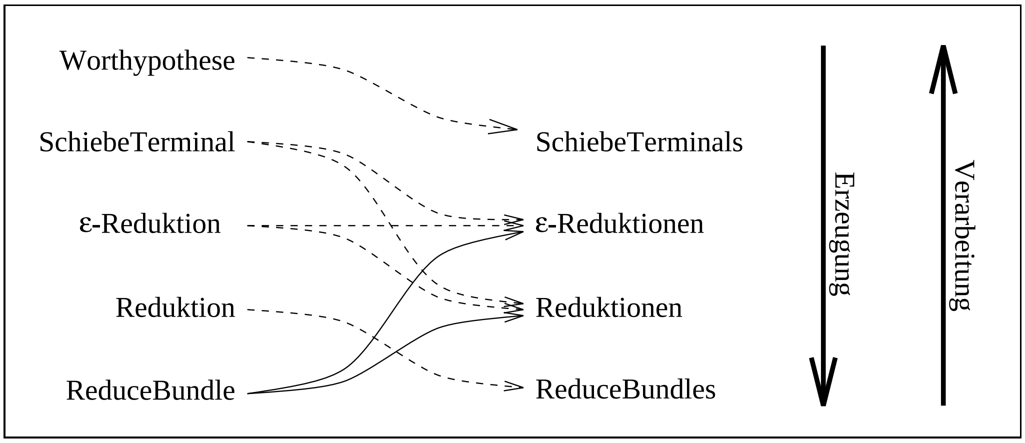

Abbildung 5.1: Verarbeitungsreihenfolge im traditionellen Lattice-GLRP
Die Pfeile beschreiben welche Aktionen welche andere Aktionen zur Folge haben können. Die durchgezogenen Pfeile stellen den Anteil dar, der für die — geringe — Rückkopplung verantwortlich ist.

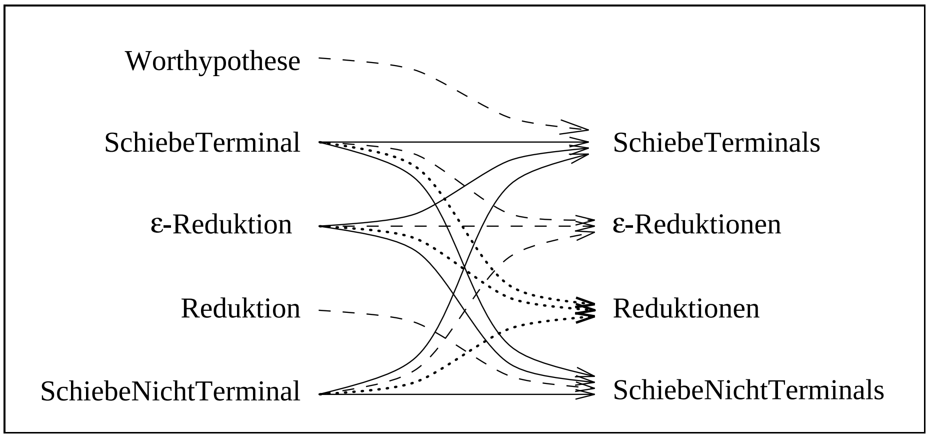

Abbildung 5.2: Interaktion der Agendaaktionen
Durchgezogene Pfeile: neue Möglichkeiten zur Erzeugung von Aktionen.
Gepunktete Pfeile: modifizierte kausale Verknüpfungen.
Gestrichelte Pfeile: Kopplungen, die es bereits im traditionellen GLRP gab.

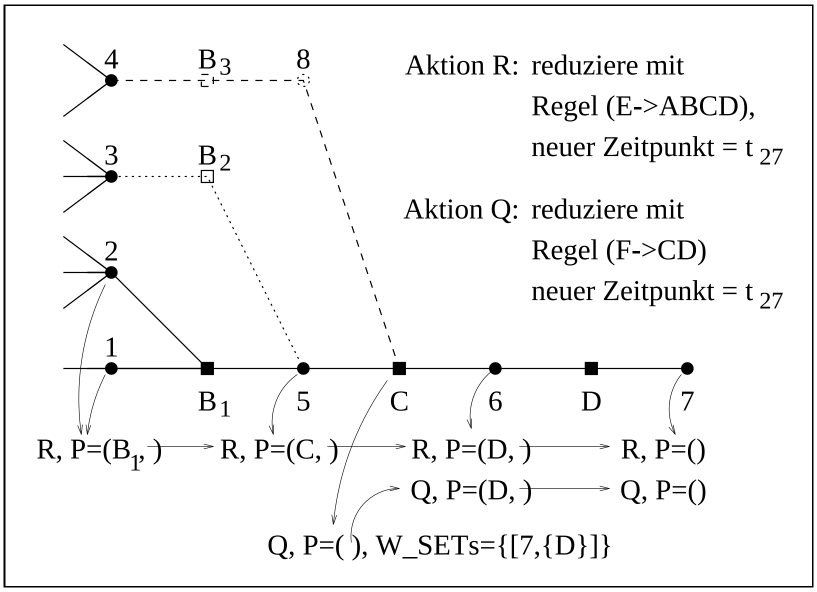

Abbildung 5.3: Reduktionen mit rechtem Kontext

und ε-Reduktionen hingegen erzeugen alle auch innerhalb des DAGs neue DAG-Bestandteile. Deshalb gelten für alle drei Aktionen die folgenden Überlegungen gleichermaßen:

1. Es wird ein neuer DAG-Knoten kreiert: Dieser Fall läßt sich wie früher behandeln, evtl werden aber zusätzlich neue *SchiebeTerminal-Aktionen* erzeugt.

   Denn es ist möglich, daß eine Worthypothese, die mit einem soeben neu erzeugten DAG-Knoten kombinieren würde, bereits von der Agenda genommen wurde. Solche Worthypothesen werden deshalb im DAG vermerkt und bei neuen DAG-Knoten wird getestet, ob sie Kategorien der abgespeicherten Worthypothesen erwarten. Wenn dies der Fall ist, wird eine bzw. werden mehrere Schiebeaktionen ausgelöst.

   Reduktionen und ε-Reduktionen werden, wie bereits bei der traditionellen Lattice-Version, bei jedem neuen DAG-Knoten angestoßen.

2. Der DAG-Knoten existiert bereits:

   (a) Wenn der benötigte Link noch nicht existiert (Abb. 5.3, Bsp. 1: Link B_2), dann muß er kreiert werden und der Menge der Links des existenden DAG-Knotens ([5,{B_1}]}) hinzugefügt werden.

   (b) Wenn der Link bereits existiert (Abb. 5.3, Bsp. 2: Link C mit altem Vorgänger-DAG-Knoten [5,{B_1}]), dann muß die Menge der DAG-Knoten-Vorgänger um den kürzlich kreierten Vorgänger ([8,{B_3}]) erweitert werden.

   (c) Alle Reduktionen, die den existenden DAG-Knoten (Bsp. 1: [5,{B_1}]; Bsp. 2: [6,{C}]) bereits passiert haben, müssen nun auf allen Pfaden fortgesetzt werden, die noch nicht durchsucht wurden. Dabei sind 2 Möglichkeiten zu unterscheiden:

      i. Der benötigte Link ist das letzte Glied der jeweiligen Reduktion:

Dann müssen alle SchiebeNichtTerminal-Aktionen, die nun neu möglich sind, auf die Agenda gebracht werden.
In Bsp. 1 gibt es keine solchen Aktionen.
In Bsp. 2 hätte diese SchiebeNichtTerminal-Aktion die Parameter Link C, DAG-Knoten [8, $\{B_3\}$], Regel Q und Forest-Knoten N. N wäre dabei der Knoten im Parse Forest, der der Reduktion (F $\rightarrow$ CD) entspräche und bereits vorher der SchiebeNichtTerminal-Aktion mit den Parametern C, [8, $\{B_3\}$] und Q übergeben worden wäre.

ii. Der Link ist nicht das letzte Glied in der Reduktion:
Dann muß eine Reduktion mit entsprechend vorinitialisiertem Pfad in die Agenda eingebracht werden, damit die Rückwärtssuche auf dem neuen Weg fortgesetzt werden kann.
In Bsp. 1 hätte diese Reduktion die Parameter Regel R, Pfad (B_2CD) und aktueller DAG-Knoten [3,$\{\ldots\}$], in Bsp. 2 die Parameter R, (CD) und [8, $\{B_3\}$].

5.5 Die vereinheitlichte Sichtweise des GLRP

Tomitas Darstellung des GLRP läßt die verschiedenen Subroutinen für das Schieben von Terminalen, für Reduktionen und für ε-Reduktionen als streng separierte Einheiten erscheinen.

Eine Revision der bisherigen Betrachtungen, sowie ein Vergleich mit dem Chartparsing, ermöglicht eine kompaktere Beschreibung des Agenda-GLRP, die zudem auch die Prinzipien von Tomita's GLRP viel klarer und homogener werden läßt:

- Der DAG ist die zentrale Datenstruktur zur Steuerung der Aktionen. Gleichzeitig stellt er auch eine syntaktische Beschreibung der Eingabe dar. Damit die Ableitungsbäume aber mit einer trivialen Aufzählung herausgelesen werden können, wird der Parse Forest erstellt.

- Die elementare Funktion des GLRP ist die *Schiebeaktion*.

  Sie subsumiert SchiebeTerminal-, SchiebeNichtTerminal-Aktionen und ε-Reduktionen und entspricht ungefähr dem Einfügen einer *passiven Kante* beim Chartparsing[7].

- Eine Schiebeaktion erweitert den DAG um einen Link und einen DAG-Knoten. Dabei gilt aber das Prinzip der maximalen Strukturteilung, welches bedeutet, daß keine redundanten Links und DAG-Knoten eingeführt werden.

- Eine *passive Kante* ($\approx$ Link) wird beim GLRP nicht global eingeführt wie beim Chartparsing. Stattdessen gibt es verschiedene linke Kontexte (= DAG-Knoten) die mit dieser passiven Kante kombinieren können.[8]

  Die Erzeugung einer oder mehrerer passiven Kante(n), wird durch ein Ereignis bestimmt; die passenden Kontexte werden durch eine Suche gefunden:

[7] Diese passive Kante wird beim GLRP aber nicht nur eingeführt, sondern es erfolgt auch automatisch ein Teil der Kombination von aktiver mit passiver Kante. Vgl. auch Kap. 7.2.

[8] Das ist gleichzeitig der Vor- und Nachteil des GLRP. Vorteilhaft ist, daß nicht versucht wird, diese passive Kante mit allen anderen aktiven Kanten zu kombinieren. Nachteilig ist es insofern, als möglicherweise mehrere gleiche Kanten für verschiedene linke Kontextsituationen benötigt werden. Vgl. dazu auch Kap. 2.4.

  - Eine neue Worthypothese ist ein Ereignis, welches eine Suche nach der Kombinierbarkeit mit den DAG-Knoten des Anfangszeitpunkts dieser Hypothese auslöst. Dabei werden passende linke Kontexte für eine terminale Kante gefunden.
  - Ein neuer DAG-Knoten ist ein Ereignis, welches eine Rückwärtssuche (Completerschritt) von diesem DAG-Knoten aus startet. Dabei werden passende linke Kontexte für eine nichtterminale Kante gefunden.

- Das letzte Prinzip ist, daß durch die Strukturteilung keine Rückwärtssuchschritte, und damit neue passive Kanten, vergessen werden dürfen. Nur in der Ausführung dieses Prinzips unterscheidet sich letztendlich die Agendaversion von dem bisherigen GLRP.

Die Ablaufstrukturen im GLRP sind also längst nicht so schwer zu durchschauen, wie es angesichts der Beschreibung in [Tom85] zunächst erscheint.

5.6 Resümee zur Agendaversion

Die Überlegungen dieses Kapitels wurden in der Implementierung einer Agendaversion eines Lattice-GLRP erfolgreich umgesetzt. Außerdem führte die vereinheitlichte Betrachtung des GLRP zu einem um ca. 15% kompakteren Code im Vergleich zum traditionellen Algorithmus.

Der realisierte traditionelle Lattice-Parser und die Agendaversion sind nicht leicht zu vergleichen, da einige Implementationsentscheidungen überdacht wurden, was generell zu einer verbesserten Laufzeit führte. Aber es sind sicherlich auch noch viele Optimierungen der derzeitigen Version möglich[9].

Einige wenige vergleichende Experimente zwischen einer Agendastrategie mit zufälligem Zugriff auf die gerade zur Verfügung stehenden Aktionen und einer Strategie, deren Ablaufstrukturen der eines traditionellen Lattice-Parsers sehr ähnlich sind ergaben eine Verdopplung der Laufzeit bei der Zufallsstrategie. Es besteht aber die begründete Vermutung, daß dieser Unterschied durch ein Tuning aller suchenden Funktionen erheblich reduziert werden kann.

[9] Z. B. Agendazugriff, DAG-Verwaltung, ...

Kapitel 6

N-Beste-Verfahren

Die einfachste Schnittstelle zwischen einem Decoder und einem Parser sind N-Beste Ketten[1]. Es gibt sehr effiziente Algorithmen, um aus einem Wort-Lattice, welches der Decoder liefert, und einem Bewertungsschema, wie es z.B. in Kap. 4 beschrieben wurde, die N bestbewerteten Ketten zu extrahieren[2], damit sie anschließend in einem sequentiellen Parser analysiert werden können.

Die entsprechenden Algorithmen sind vergleichsweise einfach zu implementieren, haben aber auch Nachteile:

- Das Lattice muß als Ganzes vorliegen, bevor mit der Extraktion der N-Besten Ketten und der syntaktischen Analyse begonnen werden kann. Es ist also keine inkrementelle Kopplung von Decoder und Parser möglich.

- Lattices und verbundene Wortgraphen haben den Vorteil, daß die Information des Decoders optimal codiert ist, weil es bei der Darstellung der möglichen Wortfolgen keine Redundanzen gibt.

  Dagegen sind bei N-Besten Ketten Redundanzen sehr häufig, da sich viele ähnlich bewertete Ketten nur in ein oder zwei Worthypothesen unterscheiden. Dies hat zur Folge, daß der Parser sehr viel redundante Arbeit erledigen muß, wenn nicht eine der allerersten Ketten bereits zu einer erfolgreichen Analyse führt, was aber bei großen Lexika äußerst selten der Fall ist.

Um eine inkrementelle Kopplung ausführen zu können, muß man ein Verfahren benutzen, wie es in Kapitel 7 dargestellt wird. Wenn dies aber nicht nötig ist, dann bietet es sich an, nur auf die Vermeidung von redundanter Parsing-Arbeit zu achten:

6.1 Das A*-Verfahren von Schmid

L. Schmid hat in [Sch94] ein Verfahren vorgeschlagen, welches hilft, redundante Parsing-Arbeit zu verringern. Die Methode besteht aus zwei Stufen:

[1] Vgl. Kap. 3.1.

[2] Vgl. z.B. [CS89, Tho90].

1. Das Lattice wird in einer Rückwärtssuche durchschritten, bei der zu jedem Worthypothesenende die genauen Wahrscheinlichkeiten der besten Fortsetzungen berechnet werden.

2. Anschließend wird eine A*-Suche[3] mit einem Parsing-Prozeß gekoppelt. Die Wahrscheinlichkeiten aus Schritt 1 bestimmen den Suchpfad, den der Parsing-Prozeß beschreitet.

   Wenn ein Pfad fehlschlägt, dann werden die k letzten Parseschritte bis zu dem Verzweigungspunkt zurückgenommen[4], an dem die nächstbeste Wortsequenz anknüpfen kann.

Dieser Algorithmus entnimmt einem Lattice also nicht wirklich N Ketten, sondern einen Suchbaum, der viele Linksredundanzen nur einmal parsen muß. Die exakt berechneten Wahrscheinlichkeiten erlauben eine optimale Fortsetzung jeder Wortsequenz zu einem ganzen Parse, falls ein Parse überhaupt möglich ist. Wenn eine Wortsequenz keine parsebare Fortsetzung hat, dann wird die nächstbestbewertete Wortsequenz untersucht.

Dieser Algorithmus ist wirkungsvoller als die direkte Benutzung von N-Besten Ketten, hat aber auch einen hohen Anteil an redundanter Arbeit:

- **Linke Kontexte** werden wiederholt geparst, wenn eine Situation wie in Abb. 6.1 auftritt.[5]

  In diesem Beispiel wird aufgrund der jeweiligen Fortsetzungswahrscheinlichkeiten zuerst die Sequenz, "ich gehe hort", geparst. Diese wird als syntaktisch ungültig erkannt, woraufhin die nächstbeste Wortfolge geparst wird. Die nächstbeste Wortfolge, was sich mit Hilfe einer priorisierten Liste erkennen läßt, ist die Sequenz "ich sehe hort". Die Worthypothese "ich" muß nun nicht erneut geparst werden, da entlang des A*-Suchbaum die Hypothese "ich" nur einmal erscheint.

  Nachdem die zweite Möglichkeit abgelehnt wurde, kommt schließlich die dritte Wortfolge an die Reihe und wird erfolgreich geparst. Für diesen Parse mußte aber die Hypothese "gehe" erneut analysiert werden, da die Pfade "ich sehe hort" und "ich gehe fort" nur die Worthypothese "ich" gemeinsam haben. Der linke Kontext, "ich", konnte also erfolgreich wiederverwendet werden, "ich gehe" mußte aber zweimal bearbeitet werden.

- **Rechte Kontexte** werden wiederholt geparst, wenn sich Wortsequenzen im Anfangsteil ihrer Worthypothesen unterscheiden.

  Im schlechtesten Fall entartet auch die A*-Methode zu einer reinen Suche über N-Beste Ketten mit sehr viel redundanter Information. Ein typisches Beispiel hierfür ist Abb. 6.2, in der sich die beiden Worthypothesensequenzen nur in ihren beiden ersten Hypothesen unterscheiden, damit aber bereits zwei vollständig separate Parses auslösen.

[3] Vgl. [Nil82] zur A*-Suche. Die Wahrscheinlichkeiten entsprechen hier den Kostenabschätzungen für die verschiedenen Pfade.

[4] Das geht sehr effizient mit GLRP. Vgl. dazu die Implementierung des sequentiellen Parsers und [Tom85], Kap. 7.3.

[5] Dem könnte man entgegenwirken, wenn die "UNDO"-Funktion nicht, wie bei [Tom85] beschrieben, alle gewonnenen Informationen wegwerfen, sondern über ein entsprechendes Referenzierungsschema, mit den Teilzweigen des A*-Baums als Indizes, retten würde. Dies würde auch viel eher dem Charakter einer A*-Suche entsprechen. Dieser Punkt wird bei Schmid aber leider nicht ausgeführt.

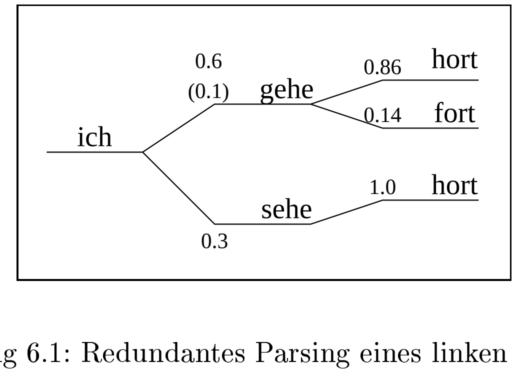

Abbildung 6.1: Redundantes Parsing eines linken Kontextes

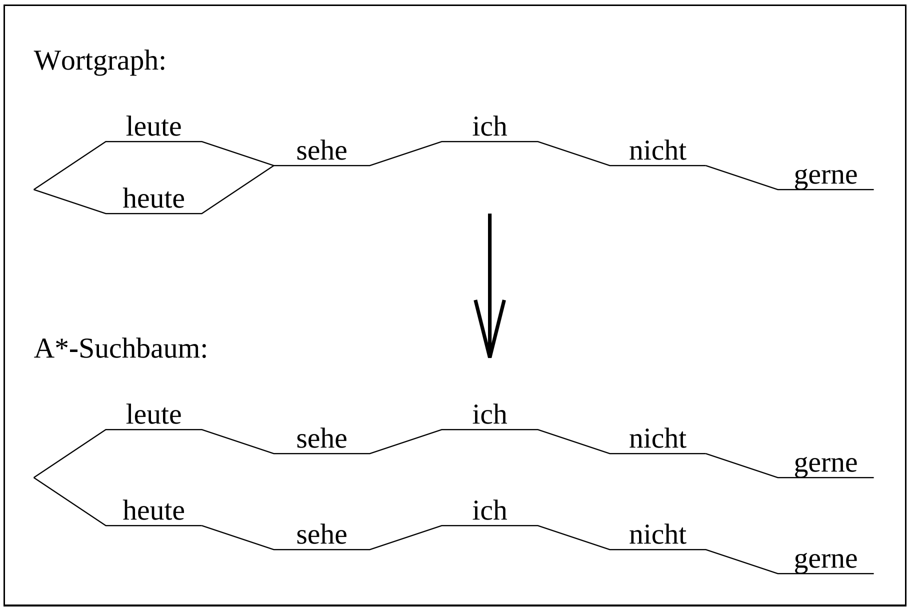

Abbildung 6.2: Redundantes Parsing eines rechten Kontextes

6.2 Das verbesserte A*-Verfahren

Mit Hilfe eines agendagesteuerten GLRP kann man ein Verfahren angeben, welches einen größeren Anteil an Kontexten wiederverwendet, anstatt sie erneut zu parsen.

Das Verfahren von Schmid wird dazu nicht wesentlich verändert. Der einzige signifikante Unterschied besteht darin, daß keine Information, die bereits in einem Parse gewonnen wurde, wieder vernichtet wird.

Neue Worthypothesen werden in der Reihenfolge geparst, die der A*-Suchbaum vorgibt, wobei aber identische Worthypothesen nur einmal geparst werden müssen, auch wenn eine Worthypothese in verschiedenen Zweigen des Suchbaums mehrfach auftaucht, wie z.B. “sehe” in Abb. 6.2. Daß eine Worthypothese bereits geparst wurde, wird durch einen billigen Test auf Identität festgestellt, und die Worthypothesenreihenfolge wird durch die entsprechende Ordnung auf der Agenda bestimmt.[6]

Die Agenda wird für dieses Verfahren mit einer einfachen Strategie (z.B. Last In First Out) betrieben, die alle Möglichkeiten einer Worthypothese vollständig abarbeitet, bevor die nächste angegangen wird.

Diese Methode hat aber auch einen Nachteil. Wenn nämlich ein Parse einer Worthypothesenfolge existiert, dann wird er zwar auch gefunden, es kann aber auch passieren, daß eine schlechter bewertete Wortsequenz vor einem besser bewerteten Pfad erfolgreich geparst wird.

In dem Beispiel in Abb. 6.3 sind 4 Pfade durch den Wortgraphen möglich. Der beste Pfad, “a b_1 c d_1 e” erweist sich nach dem Parsen von “e” als ungrammatikalischer Satz. Deswegen wird der zweitbeste Pfad, “a b_2 c d_2 e” versucht. Um diesen Pfad zu parsen, müssen lediglich die bisher nicht geparsten Worthypothesen “b_2” und “d_2” auf die Agenda gelegt werden. Nachdem aber “b_2” bearbeitet wurde, wurde auch schon der ganze Pfad “a b_2 c d_1 e” bearbeitet und kann daher auch — fälschlicherweise — akzeptiert werden.

Für dieses Problem kann man sich mehrere Lösungsmöglichkeiten denken:

- Man kann den schlechter bewerteten Satz als Lösung akzeptieren. Die Bewertung einer Wortsequenz erfolgt häufig mit Bi- und Trigrammen und hat daher nur sehr lokale Auswirkungen. Beispiele wie in Abb. 6.3 werden also nicht allzu häufig auftreten, und falls doch, dann wird der schlechter bewertete Pfad sehr oft nur sehr wenig schlechter bewertet sein als der bessere.

- Nach der Akzeptanz werden noch ein paar weitere Pfade geparst. Anschließend wird wie in [WW91b] die beste Ableitung — gemessen mit einer stochastischen CFG — extrahiert.

Die sauberste Variante ist aber wahrscheinlich die folgende:

- Wenn eine Eingabe akzeptiert wurde, wird im Parse Forest überprüft, ob der zuletzt aktive Zweig des A*-Suchbaums die Akzeptanz ausgelöst hat, oder ob eine unbeabsichtigte Kombination die Ursache war. Dies geht sehr effizient, da sich zusätzlicher Zeit- und Speicherbedarf linear zur Größe des Parse Forest verhalten — mit einer sehr kleinen Konstanten, da hierfür nur sehr einfache bool'sche Operationen ausgeführt werden

[6] Die Agendaanordnung der Hypothesen wäre für das Beispiel von Abb. 6.1: “ich”, “gehe”, “hort”, “sehe”, “fort”, für Abb. 6.2: “heute”, “sehe”, “ich”, “nicht”, “gerne”, “leute”.

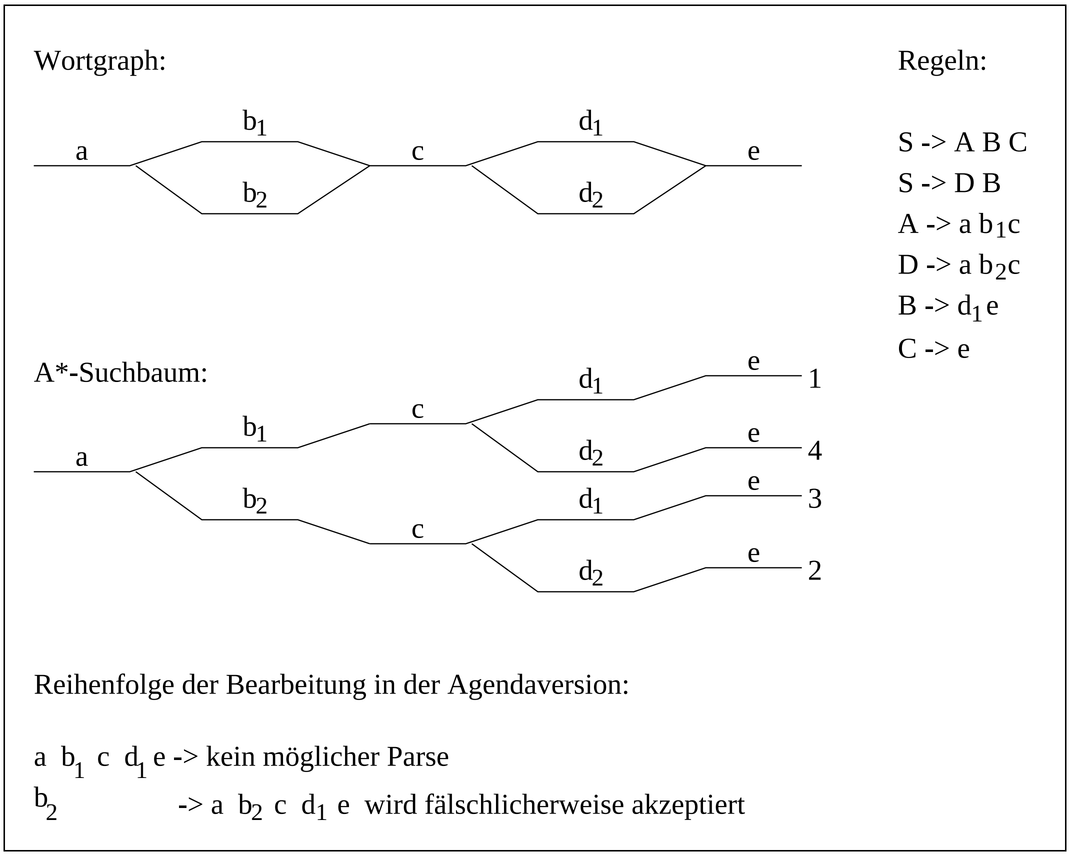

Abbildung 6.3: Akzeptanz eines schlechterbewerteten Pfades

müssen. Die bool'schen Variablen, die die Präsenz einer Worthypothese bzw. einer Worthypothesensequenz anzeigen, werden zur Wurzel des Parse Forest propagiert, indem die einzelnen Variablen für die alternativen Pfade ge-oder-t werden, innerhalb eines Pfades aber alle Worthypothesen gleichzeitig präsent sein müssen.

Abbildung 6.4 zeigt, wie dieser Test der schlechterbewerteten Sequenz von Abb. 6.3 ablehnen würde, während Abb. 6.5 die Akzeptanz der bestbewerteten parsebaren Worthypothesenfolge im weiteren Parse verdeutlicht. Außerdem könnte man sich auch über-

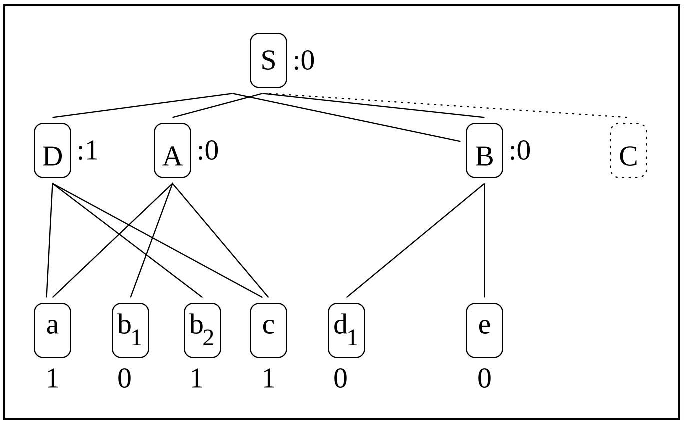

Abbildung 6.4: Bool'scher Test im Parse Forest — gescheitert

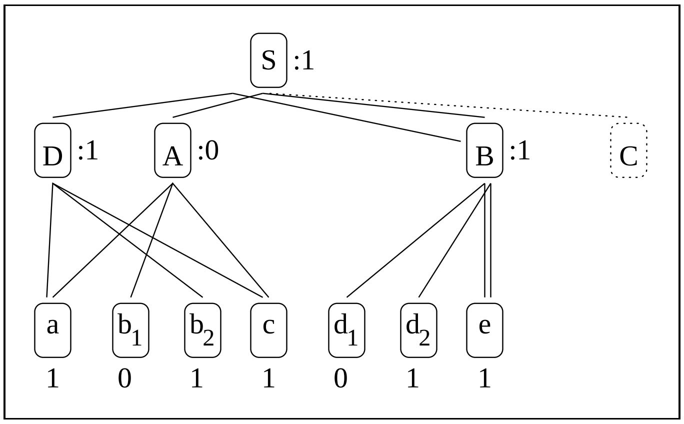

Abbildung 6.5: Bool'scher Test im Parse Forest — erfolgreich

legen, die bool'schen Operationen ständig dynamisch mitzuberechnen und Agendaaktionen zu definieren, die, falls sie die Reduktion augenblicklich "ungültiger" Parse-Knoten beinhalten, auf spätere Zeitpunkte verschoben werden. Dies soll jedoch späteren Arbeiten überlassen bleiben.

Kapitel 7

Beam-Search

Eine inkrementelle Kopplung von Decoder und Parser läßt sich mit keinem Ansatz verbinden, der den kompletten Wortgraph benötigt, um Pfadbewertungen zu finden und damit den Parsingprozeß zu steuern.[1]

Die inkrementelle Integration erfordert das Durchsuchen verschiedener Worthypothesensequenzen aufgrund von Informationen des jeweiligen linken Kontextes. Dafür kommen zunächst Breiten- und Bestensuche als Basisstrategien in Frage. Die Komplexität der Aufgabe verbietet aber eine Breitensuche[2].

Eine Bestensuche bereitet hingegen auch Probleme, da die Vergleichbarkeit verschiedener Aktionen, die sich über verschiedene und auch verschieden lange Eingabesequenzen erstrecken, nicht gewährleistet sein kann. So ist es zum Beispiel möglich, daß eine schlechte Akustik während einiger Frames zu einer schlechten Bewertung aller dort befindlichen Worthypothesen führt, was die Bestensuche zu einer Breitensuche im Anfangsabschnitt des Eingabesignals gerieten ließe.

Der erfolgreiche Kompromiß zwischen diesen beiden Ansätzen ist der *Beam-Search*. Diese Strategie beruht zum einen darauf, daß vor allem Sequenzen mit gleichem Endzeitpunkt bewertet und verglichen werden, zum anderen aber auch nicht nur der am besten bewertete Pfad, sondern ein Teil der bestbewerteten Pfade durchsucht wird. Diese Strategie wurde verschiedentlich und erfolgreich mit Chartparsing kombiniert[3], soweit bekannt aber noch nie mit GLRP.

7.1 Metrik

Eine Metrik für Beam-Search muß folgendes leisten:

- N-Gram Modelle sollen in die Bewertungen eingehen.
- Akustische Bewertungen von Worthypothesen sollen benutzt werden.

[1] Dieses Kapitel ist sehr eng angelehnt an Weber [Web94]. Die Ausführungen dort sind allerdings sehr viel umfangreicher. Seine Argumente wurden direkt als Grundlage für Implementationsentscheidungen benutzt.

[2] Vgl. Kapitel 4.

[3] Vgl. z.B. [Web94, CCL93].

- Die Bewertungen von Aktionen sollen vergleichbar sein. Es müssen also Normalisierungen vorgenommen werden, damit Sequenzen mit mehreren Worten oder langdauernde Worthypothesen keine prinzipiellen Nachteile erfahren.

- Der vollständige linke Kontext soll berücksichtigt werden.

Sowohl die Wahrscheinlichkeiten, die der Decoder für Worthypothesen, als auch die, die das N-Gram-Modell für Wortketten berechnet, sind nur wirklich vergleichbar, wenn die jeweilige Länge in die Bewertung miteinfließt, da verschieden viele Multiplikationen ausgeführt werden müssen.

Die normalisierte akustische Wahrscheinlichkeit einer Worthypothesenkette $\overline{w}$ kann man also intuitiv, wie folgt, definieren:

$$\log P_{normal}(\overline{w} \mid \text{HMM}) = \frac{\log P(\overline{w} \mid \text{HMM})}{\#(\text{Frames})} \tag{7.1}$$

Die entsprechende normalisierte Bigram-Wahrscheinlichkeit einer Sequenz $\overline{w}$ kann man folgendermaßen festlegen:

$$\log P_{normal}(\overline{w} \mid \text{N-Gram}) = \frac{\log P(\overline{w} \mid \text{N-Gram})}{\#(\text{BigramOperationen})} \tag{7.2}$$

Die kombinierte normalisierte Bewertung einer Worthypothesenkette $\overline{w}$ wird dann mit Hilfe eines Parameters λ einjustiert, weil die beiden Bewertungsschemata auch nicht direkt vergleichbar sind und deswegen die Kombination am besten anhand von Experimenten optimiert wird[4]:

$$\log P_{normal}(\overline{w} \mid \text{N-Gram, HMM}) = \left\{ \begin{array}{c} \lambda \cdot \log P_{normal}(\overline{w} \mid \text{N-Gram}) \\ + \\ \log P_{normal}(\overline{w} \mid \text{HMM}) \end{array} \right. \tag{7.3}$$

Die bisher angegebene Metrik unterstützt noch nicht die effiziente Einbeziehung des linken Kontextes. Zu diesem Zweck werden die Bewertungen — analog zu [Web94] — in *Inside-* und *Outside-Bewertungen* unterschieden und auf den Links des DAGs definiert.

Die Inside-Bewertung eines Links L, der die Worthypothesensequenz $\overline{w}$ überspannt, ist gegeben durch[5]:

$$\log P_{\text{inside}}(L) = \begin{cases} \log P_{normal}(\overline{w} \mid \text{N-Gram, HMM}), & \text{falls } \mathit{length}(\overline{w}) > 0 \\ 0, & \text{falls } \mathit{length}(\overline{w}) = 0 \end{cases} \tag{7.4}$$

Die Outside-Bewertung eines Links L mit Vorgänger-DAG-Knoten K wird rekursiv definiert. Wenn K der Start-DAG-Knoten ist, dann gilt:

$$\log P_{outside}(L) = \text{Normalize}\left(\begin{array}{l} \text{Denormalize}(0)+ \\ \text{Denormalize}(P_{inside}(L))+ \\ \lambda \cdot \log P(\text{first\_word}(L) \mid \\ \qquad\qquad \text{*BEGIN-MARKER*,N-Gram}) \end{array} \right). \tag{7.5}$$

[4] Auch andere Bewertungsschemata sind denkbar.

[5] 0 entspricht der Wahrscheinlichkeit 1, da $\log 1 = 0$.

Ansonsten:

$$\log P_{outside}(L) = \max_{l \in \mathrm{Links}(K)} \mathrm{Normalize}\left(\begin{array}{l} \mathrm{Denormalize}(P_{outside}(l)) + \\ \mathrm{Denormalize}(P_{inside}(L)) + \\ \lambda \cdot \log P(\mathrm{first\_word}(L) | \\ \qquad\qquad \mathrm{last\_word}(l), \mathrm{N\text{-}Gram}) \end{array} \right). \quad (7.6)$$

Die Funktion "Denormalize" bewirkt dabei eine Extraktion des akustischen und des N-Gram-Scores, sowie deren Denormalisierung. Dies ist nötig, da die normalisierten Scores nicht direkt miteinander kombiniert werden können. "Normalize" ist die dazugehörige inverse Funktion.

Diese Definition gilt zunächst für Bigramme; für Trigramme oder andere Modelle müßte aber nur der Term mit der bedingten Wahrscheinlichkeit geändert werden.

Wenn ein Link mehrere Wortfolgen repräsentiert, dann ist es sinnvoll, für die Bewertung in Gleichung 7.6 die maximale Bewertung zu nehmen, da ja die beste Analyse gefunden werden soll. Damit aber wirklich jede Bewertung bestimmt werden kann, muß jeder rechte und linke Rand einer Wortsequenz eines Links mit der jeweiligen besten Inside-Bewertung mitgeführt werden. Dieser Rand umfaßt im Bigram-Fall ein, im Trigramm-Fall zwei Worte. Das bedeutet also, daß sich die theoretische Komplexität der Suche über das ganze Lattice hinweg sogar verteuert.[6] Experimente mit Chartparsing-Ansätzen haben aber gezeigt, daß die beste Analyse eines Lattice mit Hilfe stochastischer Bewertungen viel effizienter gefunden werden kann als ohne.

7.2 Agendastrategie

Den Paaren von aktiver und passiver Kante auf der Agenda des Chartparsers entsprechen die verschiedenen Schiebeaktionen im GLRP. So wie beim Chartparser z.B. bei H. Weber diese Kantenpaare bewertet und abgearbeitet werden, werden bei dieser GLRP-Beam-Variante die neuen Outside-Bewertungen eines Links[7] berechnet, und die dazugehörige Aktion wird mit dieser Outside-Bewertung in die Agenda eingebracht.

Es ergeben sich also zwei Gruppen von Agendaaktionen:

1. Bewertete Aktion:
   - *Schiebeaktion*:
     Diese subsumiert:
     - *Schieben eines Nichtterminals*
     - *Schieben eines Terminals*
     - *ε-Reduktion*
2. Unbewertete Aktionen (Steueraktionen):
   - *Reduktion (Completerschritt)*
   - *neue Worthypothese*

[6] Bei stochastischen CFGs müssen sogar alle ganzen Pfade mitgeführt werden. In diesem Fall ist der Worst Case exponentiell.

[7] Genauer: die Outside-Bewertungen eines Links mit dem jeweiligen speziellen Vorgänger-DAG-Knoten. Vgl. dazu Tab. 5.2, S. 36.

- *Akzeptanz*

Letztere werden, wenn sie auftreten, nicht anhand einer Bewertung in die Agenda einsortiert, sondern ihre Abarbeitung unterliegt einem festen Schema, für das es mehrere Möglichkeiten gibt. Das implementierte Schema veranlaßt die sofortige Bearbeitung einer Akzeptanz, es verarbeitet anstehende Reduktionen vor Schiebeaktionen mit LIFO-Strategie, und arbeitet neue Worthypothesen schubweise, nach ihren Wortendezeitpunkten geordnet, ab.

Die Schiebeaktion dagegen ist die einzige Aktion, die bewertet wird. Der hierbei benutzte Score ist die Outside-Bewertung des neuen Links mit dem jeweiligen Vorgänger-DAG-Knoten, wie in Kapitel 7.1 geschildert.

Definition 7.1 *Beam-GLRP(Lattice)*

1. *Initialisiere* $Agenda_{work} = Agenda_{pruned} = \emptyset$
2. *Setze aktuellen Wortendezeitpunkt* T_0 *auf den Wortendezeitpunkt der nächsten Worthypothese.*
3. *Bearbeite alle Hypothesen mit Wortendezeitpunkt* T_0*. Füge die dabei anfallenden Schiebeaktionen in* $Agenda_{work}$ *ein.*
4. *Wenn* $Agenda_{work} = \emptyset$ *dann:*
   (a) *Wenn* $Agenda_{pruned} = \emptyset$*, dann gibt es keinen Parse.*
   (b) *Ansonsten: Setze* $Agenda_{work}$ *auf* $Agenda_{pruned}$*,*
   $Agenda_{pruned} = \emptyset$.
5. *Setze MaxBeamValue auf die beste Bewertung in* $Agenda_{work}$.
6. *Prune:*
   $Agenda_{pruned} = Agenda_{pruned} \cup \; Bottom_{beamwidth}(MaxBeamValue,\ Agenda_{work}) \wedge$
   $Agenda_{work} = Top_{beamwidth}(MaxBeamValue,\ Agenda_{work})$.
7. *Bearbeite alle Agendaaktionen von* $Agenda_{work}$*. Füge dabei neue Aktionen in* $Agenda_{work}$ *ein, wenn sie innerhalb der Strahlbreite von MaxBeamValue liegen, ansonsten addiere diese Aktionen zu* $Agenda_{prune}$.
8. *Gehe zu Schritt 2.*

7.3 Experimente

7.3.1 Kondition und Ziel der Experimente

Um die Realisierbarkeit des GLRP-Beam-Parsers zu belegen, wurden einige Experimente durchgeführt. In diesen Versuchsreihen wurde auf optimierte Ergebnisse weniger Wert gelegt, stattdessen wurde versucht, die Effizienz des Algorithmus zu belegen und die Schwachstellen zu erklären.

Es wurden 10 Wort-Lattices getest, die decodierte Signale aus dem Themenbereich Bahnauskunft repräsentierten.[8] Worthypothesenfamilien[9] in diesen Lattices waren bereits zu einzelnen Worthypothesen reduziert und die Konnektivität wurde anhand logischer Zeitpunkte beschrieben.

Die CFG-Analysen wurden mit Hilfe einer SLR-Tabelle erstellt, die aus der realisierten Grammatik berechnet wurde. Ebenso wurde das eigens erstellte Lexikon verwendet. Bei der folgenden Betrachtung der Ergebnisse muß bedacht werden, daß Grammatik und Lexikon nicht sorgfältig entworfen wurden, sondern innerhalb weniger Tage erstellt wurden, um eine solche Versuchsreihe überhaupt zu ermöglichen. Dies wirkte sich nachteilig auf die Parsing-Performance aus, da die Grammatik stark überproduzierte.

Für die Beam-Search-Versuche wurde ein Bigram-Modell mit relativ hoher Perplexität verwendet[10], so daß das statistische Sprachmodell alleine nicht in der Lage ist, die Suche auf wenige Worthypothesensequenzen zu reduzieren und damit das Ergebnis zu "schönen".

7.3.2 Auswertung der Versuchsreihen

Als erstes zeigt Tabelle A.1 eine Versuchsreihe mit einem Agendamodell mit LIFO-Strategie, welches die Lattices vollständig analysiert. Dort zeigt sich, daß die Parse-Zeit mit der Anzahl der Worthypothesen i. Allg. ansteigt. Die Lattices mit mehr als 150 Worthypothesen können nicht mehr in sinnvoller Zeit und mit der systembedingten maximalen Prozeßgröße geparst werden.

Die Tabellen A.2 bis A.5 zeigen Testreihen bei gleichbleibendem Kombinationsparameter, $\lambda^{11} = 1$, und verschiedenen Strahlweiten. Verständlicherweise wird das Erkennungsresultat optimal, wenn der Beam ungefähr die Menge von Aktionen wegschneidet, die nicht zur Erkennung notwendig sind (Tab. A.3 und A.4). Ein zu weiter Beam führt zu einer zu ausgeprägten Breitensuche (Tab. A.2), ein zu enger Beam zu einer Art Bestensuche (Tab. A.2).

Dabei schneidet die Bestensuche sehr schlecht ab, was aber bei einer Überprüfung der einzelnen Aktionen leicht erklärbar ist: die Normalisierung ist nicht gut genug, um wirklich die Vergleichbarkeit von Aktionen in verschiedenen Zeitabschnitten zu garantieren. Manchmal ist die Vergleichbarkeit ausreichend und dann bleibt die Strategie von Definition 7.1 effizient (Lattices 0 und 3 in allen Versuchsreihen), häufig ist dies aber auch nicht der Fall (u. a. Lattice 7).

Die logische Folgerung wäre, die Strahlsuchstrategie konsequenter durchzuführen als in Def. 7.1. Das bedeutet, daß alle gepruneten Agendaaktionen, nach Bewertung *und* Zielzeitpunkten sortiert, zu sammeln wären. Beim Fehlschlagen des ersten Beams würden die noch vorhandenen Agendaaktionen mit einem zweiten, weiteren Beam abgearbeitet, usw. Natürlich wäre auch eine verbesserte Normalisierung ein wünschenswertes Ziel. Beides soll aber aus Platz- und Zeitgründen nicht mehr in dieser Arbeit betrachtet werden.

Als nächstes kann man feststellen — was auch intuitiv einleuchtend ist — daß λ auf keinen extremen Wert eingestellt werden sollte, da dies die Erkennungsrate verschlechtert (Tab. A.6,

[8] Die Lattices erhielt ich dankenswerterweise von Gerhard Hanrieder.

[9] Vgl. dazu Kap. E.2

[10] Das Programm zur Berechnung der Bigram-Bewertungen wurde von Tim Geisler erstellt, die Bewertungen selbst erhielt ich von Hans Weber.

[11] Vgl. S. 49.

A.7 und A.10). Denn durch ein zu großes λ wird die akustische, und durch ein zu kleines die Bigram-Bewertung vernachlässigt.

Die optimale Einstellung von λ in dieser Versuchsreihe scheint in der Nähe von 0,2 zu liegen (Tab. A.8 und A.9). Dort verbessert sich die Erkennungsrate auf 80% und zeigt in sieben der acht erfolgreich analysierten Lattices eine Effizienz, die wesentlich besser ist, als die der Version, die das ganze Lattice analysiert.

7.3.3 Resümee

Die Versuchsreihen (insbes. Tab. A.9) zeigen also überwiegend positive Ergebnisse für den Beam-GLRP. Trotzdem existieren noch Schwierigkeiten — z.B. wurde Lattice 7 nie erfolgreich geparst — aber es gibt Gründe zu vermuten, daß sich diese Probleme beseitigen lassen, denn:

- es fehlte eine Feinabstimmung von λ und Strahlweite.
- das verwendete Bigram-Modell deckte nicht alle verwendeten Wortformen ab. Wenn wichtige Worte dadurch eine schlechte Bewertung erhielten, konnte dies den Beam in die völlig falsche Richtung lenken.
- die Überproduktivität der Grammatik führte zu einer großen Anzahl von erzeugten Wortseqenzen. Dies wirkt sich beim stochastischen Parsen noch sehr viel nachteiliger aus als beim nichtstochastischen, da der zugrunde liegende Algorithmus prinzipiell exponentiell ist.[12]
- es wurde bis jetzt kein Profiling durchgeführt.

Alles zusammengefaßt kann man also sagen, daß die Experimente die folgende These, welche die Quintessenz dieser Arbeit enthält, stützen:

> Das GLR-Parsing in der in dieser Arbeit vorgestellten Agenda-Version läßt sich effizient zum Parsing von Wort-Lattices einsetzen. Dabei kann der Parsing-Prozeß mittels stochastischer Informationen gesteuert werden.

[12] Im Bigram-Fall müßte der Algorithmus noch nicht exponentiell sein, da keine ganzen Wortketten mitgeführt werden müßten, sondern Ränder von Wortketten ausreichend wären. Da es aber einfacher war, den Algorithmus mit ganzen Wortsequenzen zu implementieren und ein enstprechender Algorithmus mit stochastischen CFGs dies sowieso bräuchte, ist der implementierte Algorithmus exponentiell im Worst Case.

Anhang A

Experimente mit dem Beam-Search

Lattice	Anzahl der Hypothesen	Zeit (non-gc)[1]	Anzahl der Parse-Knoten	Anzahl der Analyse-Bäume
0	56	3,3s	317	160
1	103	22,6s	640	994804
2	121	44,3s	1094	12519604
3	83	10,6s	673	7998487
4	138	11,0s	791	140469256
5	202	∞[2]		
6	81	8,2s	585	13280
7	175	∞		
8	164	∞		
9	199	∞		

Tabelle A.1: Versuchsreihe 1: mit LIFO-Strategie und vollständigem Parse

[2]non-gc: Zeit ohne Garbage Collection

[2]∞: nicht parsebar in “sinnvoller Zeit” ($\approx$ Realtime: 20min $\geq$ maximale Ausdauer des Testers) und mit der systembedingten maximalen Prozeßgröße.

Lattice	Anzahl der Hypothesen	Zeit (non-gc)	Einsatz der Pruned-Agenda	Anzahl der Parse-Knoten	Anzahl der Analyse-Bäume
0	56	2,0s	Nein	191	8
1	103	3,5s	Nein	201	100
2	121	3,4s	Nein	306	216
3	83	1,0s	Ja	151	2
4	138	∞	Nein		
5	202	∞	Nein		
6	81	1,9s	Ja	160	8
7	175	∞	Ja		
8	164	∞	Nein		
9	199	∞	Nein		

Tabelle A.2: Versuchsreihe 2: $\lambda = 1$ und Strahlweite $= 0,25$

Lattice	Anzahl der Hypothesen	Zeit (non-gc)	Einsatz der Pruned-Agenda	Anzahl der Parse-Knoten	Anzahl der Analyse-Bäume
0	56	1,8s	Ja	126	8
1	103	1,8s	Nein	149	48
2	121	2,5s	Nein	241	36
3	83	2,0s	Ja	151	2
4	138	∞	Nein		
5	202	∞	Ja		
6	81	2,7s	Ja	160	8
7	175	∞	Ja		
8	164	∞	Nein		
9	199	8,8s	Nein	337	324

Tabelle A.3: Versuchsreihe 3: $\lambda = 1$ und Strahlweite $= 0,1$

Lattice	Anzahl der Hypothesen	Zeit (non-gc)	Einsatz der Pruned-Agenda	Anzahl der Parse-Knoten	Anzahl der Analyse-Bäume
0	56	2,2s	Ja	126	8
1	103	1,3s	Nein	121	4
2	121	2,6s	Nein	222	36
3	83	1,5s	Ja	151	2
4	138	48,2s	Nein	267	1584
5	202	∞	Ja		
6	81	2,7s	Ja	160	8
7	175	∞	Ja		
8	164	∞	Ja		
9	199	3:16min	Ja	588	32

Tabelle A.4: Versuchsreihe 4: $\lambda = 1$ und Strahlweite $= 0,05$

Lattice	Anzahl der Hypothesen	Zeit (non-gc)	Einsatz der Pruned-Agenda	Anzahl der Parse-Knoten	Anzahl der Analyse-Bäume
0	56	1,4s	Ja	126	8
1	103	2,1s	Ja	190	40
2	121	1,6s	Nein	222	36
3	83	0,9s	Ja	151	2
4	138	∞	Ja		
5	202	∞	Ja		
6	81	1,9s	Ja	160	8
7	175	∞	Ja		
8	164	∞	Ja		
9	199	∞	Ja		

Tabelle A.5: Versuchsreihe 5: $\lambda = 1$ und Strahlweite $= 0,02$

Lattice	Anzahl der Hypothesen	Zeit (non-gc)	Einsatz der Pruned-Agenda	Anzahl der Parse-Knoten	Anzahl der Analyse-Bäume
0	56	2,4s	Ja	126	8
1	103	2,1s	Nein	160	48
2	121	3,7s	Nein	293	108
3	83	1,6s	Ja	151	2
4	138	∞	Nein		
5	202	∞	Nein		
6	81	2,6s	Ja	161	8
7	175	∞	Ja		
8	164	∞	Nein		
9	199	24,2s	Nein	378	252

Tabelle A.6: Versuchsreihe 6: $\lambda = 5$ und Strahlweite $= 0,25$

Lattice	Anzahl der Hypothesen	Zeit (non-gc)	Einsatz der Pruned-Agenda	Anzahl der Parse-Knoten	Anzahl der Analyse-Bäume
0	56	1,5s	Ja	126	8
1	103	1,2s	Nein	121	4
2	121	2,1s	Nein	222	36
3	83	1,3s	Ja	151	2
4	138	∞	Ja		
5	202	1:54min	Ja	928	48
6	81	2,0s	Ja	161	8
7	175	∞	Ja		
8	164	∞	Ja		
9	199	∞	Ja		

Tabelle A.7: Versuchsreihe 7: $\lambda = 5$ und Strahlweite $= 0,05$

Lattice	Anzahl der Hypothesen	Zeit (non-gc)	Einsatz der Pruned-Agenda	Anzahl der Parse-Knoten	Anzahl der Analyse-Bäume
0	56	2,1s	Ja	138	8
1	103	3,3s	Nein	177	96
2	121	2,6s	Nein	269	36
3	83	1,2s	Ja	155	8
4	138	∞	Nein		
5	202	∞	Nein		
6	81	2,7s	Ja	166	8
7	175	∞	Ja		
8	164	∞	Nein		
9	199	38,6s	Nein		

Tabelle A.8: Versuchsreihe 8: $\lambda = 0,2$ und Strahlweite $= 0,05$

Lattice	Anzahl der Hypothesen	Zeit (non-gc)	Einsatz der Pruned-Agenda	Anzahl der Parse-Knoten	Anzahl der Analyse-Bäume
0	56	2,2s	Ja	138	8
1	103	1,0s	Nein	124	12
2	121	1,7s	Nein	222	18
3	83	1,2s	Ja	155	8
4	138	2:53min	Nein	287	480
5	202	∞	Nein		
6	81	2,2s	Ja	166	8
7	175	∞	Ja		
8	164	3,3s	Nein	323	4
9	199	2,3s	Nein	266	24

Tabelle A.9: Versuchsreihe 9: $\lambda = 0,2$ und Strahlweite $= 0,02$

Lattice	Anzahl der Hypothesen	Zeit (non-gc)	Einsatz der Pruned-Agenda	Anzahl der Parse-Knoten	Anzahl der Analyse-Bäume
0	56	1,5s	Ja	121	4
1	103	2,4s	Nein	188	80
2	121	2,0s	Ja	271	6
3	83	1,8s	Ja	169	32
4	138	∞	Nein		
5	202	1:00min	Nein	932	48
6	81	∞	Ja		
7	175	∞	Nein		
8	164	∞	Nein		
9	199	8,2s	Nein	331	80

Tabelle A.10: Versuchsreihe 10: $\lambda = 0,05$ und Strahlweite $= 0,02$

Anhang B

Beschreibung des Programmpaketes

Die Implementation aller Parser und des Tabellenkonstruktors erfolgte in “Allegro Common Lisp”. Der Präprozessor für die Vorverarbeitung der Grammatik wurde in “Scheme” realisiert.

Alle Sourcen sind auf dem FTP-Server “faui80.informatik.uni-erlangen.de” mit Anonymous-Login im File “/pub/lisp/parser/glr-lattice-parser.tar.gz” frei zugänglich.

Die entpackten Files sind in folgende Verzeichnisse untergliedert:

a_star: Eine Version des Tomita-Parser, die von einem A*-Algorithmus gesteuert werden könnte, wie [Sch94].

beam: Eine Beam-Search-Version des Tomita-Parser. Für den Pruning-Prozeß werden die statistische Methoden aus Kapitel 4 benutzt. Diese Version arbeitet wie in den Kapiteln 5 und 7 beschrieben.

biggrammar: Grammatik(1564 Regeln), Lexikon(500 Wortformen) und Scheme-Präprozessor. Eine Kopie der SLR-Tabellenversion.

grammar: Arbeitsverzeichnis für die jeweils aktuelle Grammatik, Tabellenversion und Lexikon.

hypotheses: Eine Menge von Test-Wort-Lattices.

lattice: Die Lattice-Version des Tomita-Parser aus [Tom86].

lifo: Eine LIFO-Strategie auf der Agenda. Agendaversion wie in Kap. 5 beschrieben.

random: Eine Zufallszugriffsstrategie auf die Agenda, ältere Agendaversion.

sequential: Die ursprüngliche Version des Tomita-Parsers aus [Tom85].

smallgrammar: Die kleine Grammatik aus [Tom85], Abb. 2.3., mit kleinem Lexikon und Scheme-Präprozessor. Eine Kopie der beiden Tabellenversionen.

tableconstr: Der Tabellenkonstruktor fuer SLR- und LALR(1)-Tabellen.

Weitere Anleitungen zur Bedienung und zu benutzten Formaten entnehmen Sie bitte den folgenden Kapiteln des Anhangs.

Anhang C

Präprozessor

Der Scheme-Präprozessor (“~snstaab/studienarbeit/src/smallgrammar/preprocess.scm” bzw. “~snstaab/studienarbeit/src/biggrammar/preprocess.scm”) bearbeitet nach dem Laden automatisch die Dateien, die am Kopf des Programms als Grammatik- bzw. Lexikon-Dateien ausgewiesen sind.
Als Resultat werden Grammatik und Lexikon in die Dateien geschrieben, die in den Variablen “output_grammar” und “output_lexicon” definiert sind.
Die Grammatik und das Lexikon können direkt von Tabellenkonstruktor und Parser benutzt werden.

C.1 Grammatikformat

Der Präprozessor benötigt das im folgenden gegebene Format, damit er die vom Benutzer gegebene Grammatik benutzen kann.
Die Regeln der Grammatik werden in einem oder mehreren Files wie folgt definiert:

```
(set! regeln (append! regeln '(
RULE*
)))
```

“RULE” ist dabei von der Form “(V A*)”, V $\in$ N und A $\in (N \cup T)$, wobei N die Menge der Nichtterminale und T die Menge der Terminale bezeichnet.
Ein konkretes Beispiel (“ snstaab/studienarbeit/src/smallgrammar/rules.scm”):

```
(set! regeln (append! regeln '(
(ss s)
(s np vp)
(s s pp)
(np *n)
(np *det *n)
(np np pp)
(pp *prep np)
(vp *v np)
(np))))
```

Alle Symbole, die am Kopf einer Regel stehen, werden vom Präprozessor zu Nichtterminalen erklärt, alle anderen Symbole zu Terminalen. Die Regel mit dem Kopf “ss” bezeichnet das Startsymbol (in diesem Beispiel “s”).

C.2 Lexikonformat

Der Präprozessor benötigt das im folgenden gegebene Format, damit er das vom Benutzer gegebene Lexikon benutzen kann.

```
(set! lexikon (append! lexikon '(
[RULE1 | RULE2]*
)))
```

“RULE1” hat die Form “((Kategorie$_1 \cdots$Kategorie$_m$) $<$ (Wort$_1 \cdots$Wort$_n$))”. Dabei werden alle Kategorien$_i$ allen Wörtern Wort$_j$, für $i = 1$ bis m und $j = 1$ bis n, zugewiesen. Werden Wortformen an verschiedenen Stellen kategorisiert, so erfolgt eine Wornung. Dieses Format eignet sich besonders, um gleichartige Kategorisierungen vieler Wörter kompakt zu erfassen. Z. B. haben die meisten femininen Nomina im Singular wie “Frau” nur eine Wortform. Allen diesen Nomina können mit dem Format von “RULE1” gleichzeitig alle vier verschiedenen Formen des Singulars zugewiesen werden.

Dagegen eignet sich das andere Format, “RULE2”, besser für Kategorien, die nur wenige Ausprägungen haben, deren Ausprägungen aber evtl. vielfach kategorisiert sind. “RULE2” hat die Form “(Kategorie $>$ Wort$_1 \cdots$Wort$_n$). Typische Beispiele sind Artikel und Pronomen. Eine Wortform kann an verschiedenen Stellen kategorisiert werden, ohne daß eine Warnung erfolgt.

Ein Beispiel hierfür könnte wie folgt aussehen:

```
(set! lexikon (append! lexikon '(
((nomen_m_sg_nom nomen_m_sg_dat nomen_m_sg_akk) < (mann park))
((v_3_sg v_2_pl) < (geht steht))
(pron_1_sg > ich)
(pron_2_sg > du)
(def_art_f_1_sg > die)
(def_art_f_2_sg > der)
(def_art_f_3_sg > der)
(def_art_f_4_sg > die)
)))
```

Anhang D

Tabellenkonstruktor

Die Bedienung des Tabellenkonstruktor ist ähnlich aufgebaut wie die der verschiedenen Parser. Zunächst müssen der "loader.lisp" des Tabellenkonstruktors geladen werden, und die Funktionen des "CONSTRUCTOR"-Packages, die im aufrufenden Package benötigt werden, durch "(use-package "CS")" bereitgestellt werden.
Anschließend stellt der Tabellenkonstruktor folgende Befehle bereit:

- *(load_constructor)* bzw. *(lc)*: Lädt die Tabellenkonstruktor-Files, sowie die Grammatik. Dafür müssen die Pfadvariablen in "loader.lisp" zuvor korrekt gesetzt worden sein.
- *(compile_constructor)* bzw. *(cc)*: Kompiliert alle Tabellenkonstruktor-Files, nicht jedoch die Grammatik. Die Pfadangaben in "loader.lisp" müssen korrekt sein.
- *(compute_table method)*: Berechnet die Parsing-Tabelle. Als Parameter können die Symbole :slr und :lalr angegeben werden (gequoted!).

Der Tabellenkonstruktor erstellt ein File "tab.lisp" im Verzeichnis "path_grammar", wo er auch erwartet, das Grammatik-File "grammar.lisp" zu finden. Für große Grammatiken gerät der Tabellenkonstruktor während des Programmlaufs leicht an die "MAX IMAGE SIZE" des Lisp-Systems. Auf der faui80 gab dies bei SLR-Tabellen mit einer 1500-Regel-Grammatik keine Schwierigkeit, sehr oft aber bei LALR-Tabellen.[1]
Die Tabellenkonstruktion geschieht in vertretbarer Zeit. Bei der bereits erwähnten Grammatik mit mehr als 1500 Regeln benötigte die faui80 ca. 45 Minuten für die Erstellung einer SLR- und 90 Minuten für die Berechnung einer LALR-Tabelle. Dabei wurde eine Tabelle mit ca. 4600×440 Einträgen kreiert.
Nachdem der Tabellenkonstruktor das File "tab.lisp" erstellt hat, empfiehlt es sich dieses zu compilieren[2], damit bei nachfolgenden Parseläufen gleich das compilierte File eingeladen werden kann, was wesentlich schneller erfolgt als ein Laden der uncompilierten Tabelle.
Die resultierende SLR-Tabelle hat dabei nur die Hälfte bis ein Drittel der Größe der entsprechende LALR-Tabelle, was sich dadurch begründet, daß Reduktionen in einem bestimmten

[1] Um diese Problem zu lösen, müßte die Steuerstruktur umgestellt werden. Im Augenblick werden alle Closures zunächst berechnet und erst anschließend wird die Tabelle abgespeichert. Eine Verzahnung dieser beiden Prozesse würde ermöglichen, daß ein Großteil des Speichers früher wieder freigegeben werden kann.

[2] Das Compilieren der LALR-Tabelle für die große Grammatik führte auch fast immer zu Problemen, weswegen eine nächste Version des Tabellenkonstruktors eine entsprechende Vorkompilation gleich integrieren sollte.

Zustand unabhängig vom nachfolgenden Symbol erfolgen. Viele Zustände lassen überhaupt keine Schiebeaktionen zu, diese Zustände können vollständig aus der SLR-Tabelle genommen werden, bis auf einen Eintrag, welcher die Reduktionen vermerkt (*RED-VEC*), die im jeweiligen Zustand möglich sind. Damit trotz des Löschens von ganzen Tabellenzeilen die richtigen Zustände referenziert werden, werden im Vektor "*INDIR-REF*" diese Löschungen korrigiert.

Der sequentielle Parser erkennt am Vorhandensein dieses Vektors, ob es sich um eine SLR- oder eine LALR-Tabelle handelt.

Bei den Worthypothesenparsern wird kein Lookahead benutzt und eine LALR-Tabelle ist hier deshalb nutzlos.

Anhang E

Handhabung der implementierten Parser

E.1 Funktionen

Zunächst muß der "loader.lisp" des jeweiligen Pakets geladen werden. In diesem sind Pfade deklariert, die angepaßt werden müssen, wenn das Paket an eine andere Stelle im Hierarchiebaum gebracht wurde. Außerdem werden dort die Packages "LATTICE-PARSER" (abgekürzt "LP") und "GRAMMAR" definiert.

Wenn die Parsing-Funktionen im augenblicklichen Package (z.B. "USER") direkt zugänglich sein sollen, dann muß nun zunächst ein "(use-package "LP")" eingegeben werden, damit die exportierten Funktionen zugänglich werden.

Warnung! Der sequentielle Parser ist nur von dem Package aus benutzbar, in welches er geladen wurde. Denn in dieses wird das Lexikon geladen, und bei direkter Eingabe von Symbolen werden diese jeweils dem augenblicklichen Package zugerechnet.

Anschließend stehen folgende Funktionen bereit:

- *(load_parser)* bzw. *(lp)*: Lädt Parser, Tabelle, Grammatik und Lexikon. Der Parser erwartet, daß Tabelle, Grammatik und Lexikon sich im Verzeichnis "path_grammar" befinden, welches in "loader.lisp" definiert wird, und daß alle Parser-Files im Verzeichnis "path_parser" zu finden sind, welches ebenfalls in "loader.lisp" definiert wird.

- *(compile_parser)* bzw. *(cp)*: Compiliert alle Parser-Files neu (nicht Grammatik und Tabelle!). Erwartet wird, daß "path_parser" korrekt gesetzt ist.

- *(parse_lattice name)*: Parst den Wortlattice mit Name "name", der sich in Verzeichnis "path_lattices" befindet, welches in "loader.lisp" definiert ist. Gilt nicht für den sequentiellen Parser.

- *(parse_sentence)* bzw. *(ps)*: Parst einen Satz, der im folgenden mit "(" und ")" eingegeben wird. Gilt nur für den sequentiellen Parser.

- *(incremental_parse)* bzw. *(ip)*: Parst einen Satz wortweise. "Unparsing" der letzten Worte ist möglich. Gilt nur für den sequentiellen Parser.

- *(send *forest* 'dump &optional flag)*: Gibt einen Dump des Parse Forest aus, der während des letzten Parse erzeugt wurde. Wenn "flag" auf "true" steht, werden nicht nur die Knoten ausgegeben, die erfolgreich in den Parse Forest eingebaut wurden, sondern alle, die während des letzten Parse erzeugt wurden.
- *(send *dag* 'dump)*: Gibt einen Dump des graph-strukturierten Stacks aus.
- *(send *forest* 'prune_bottom_up nr)*: Entfernt alle Parse-Knoten aus dem Forest, die nicht einen Teil der Wortsequenz mit der Nummer "nr" abdecken. Gilt nur für die Beam-Version.

Unbekannte Worte werden im Parser weitergeparst, indem jede Kategorie als mögliche Kategorie der unbekannten Wortform angenommen wird. In stark selektiven linken Kontexten bleibt der weitere Parsing-Prozeß effizient. Häufig aber ziehen schon wenige unbekannte Worte eine Explosion der Anzahl der Analysemöglichkeiten nach sich.

Als besonderer Key existiert das Symbol "*PAUSE*" welches vom Lattice-Parser als eine Sprechpause interpretiert und verarbeitet wird. Die Agenda-Versionen können dieses Symbol nicht korrekt bearbeiten.

E.2 Format der Word Lattices

Die Wort-Lattices, die die Parser verarbeiten können, haben das folgende Format:

```
BEGIN_LATTICE
Worthypothese*
END_LATTICE
```

Eine Worthypothese besteht aus diesem Schema: [LogT1 LogT2 Key Prob RealT1 RealT2]*. Die Bedeutung der einzelnen Komponenten ist:

- *LogT1:* Beschreibt den logischen Zeitpunkt zu dem die Worthypothese beginnt. Bei den hier verwendeten Tests werden ausschließlich die logischen Zeitpunkte zum Parsing herangezogen, da hierbei schon Familien von Worthypothesen zu einer einzigen Worthypothese zusammengefaßt sind.

  Solche Familien sind Worthypothesen mit gleichem realem Anfangszeitpunkt, gleichem Key und nahe beieinanderliegenden realen Endezeitpunkten. Daß solche Familien entstehen, liegt an der Art, in der die Hidden Markov Modelle Wortendehypothesen ausgeben.

  Das Zusammenfassen zu einer Instanz reduziert den Suchraum etwa um den Faktor 5 und vermindert somit die Parse-Dauer erheblich, ohne daß die Erkennungsrate sinkt.[1]
- *LogT2:* Der logische Zeitpunkt, zu dem die Worthypothese endet.
- *Key:* Der Schlüssel, die Wortform, mit der auf das Lexikon zugegriffen wird.
- *Prob:* Die akustische Bewertung.

[1] Eine genauere Behandlung dieses Themas erfolgt in [Web94]. Dort wird auch eine dynamische Lösung zur Behandlung dieses "Timemap"-Problems vorgestellt.

- *RealT1:* Der Frame, in dem die Worthypothese begann.
- *RealT2:* Der Frame, in dem die Worthypothese endete.

Die verwendeten Beispiele finden sich in "~snstaab/studienarbeit/src/hypotheses".

E.3 Ausgabe des Parse Forest

Die Ausgabe des sequentiellen Parsers bei der realisierten Grammatik, SLR-Tabell, Lexikon und Eingabe von "ich sehe die Frau" sieht, wie folgt, aus:

```
>  (parse_sentence)

Bitte geben Sie einen Satz ein:
(in der Form "(wort1 ... wortn)")

(ich sehe die frau)

Parsing: ICH
Parsing: SEHE
Parsing: DIE
Parsing: FRAU
Parsing: $
; cpu time (non-gc) 317 msec user, 117 msec system
; cpu time (gc)     0 msec user, 0 msec system
; cpu time (total)  317 msec user, 117 msec system
; real time  977 msec
; space allocation:
 3,245 cons cells,;  208 symbols, 184,952 other bytes
Number of trees: 1
Forest:  Roots (12)   48 nodes before pruning.
12: S ((11))
11: KERN_S ((2 3 4 10))
10: AKKUSATIV ((9))
9: AKKUSATIV_NP ((8))
8: AKKUSATIV_NP_F_SG ((5 6 7))
7: AKKUSATIV_NOUN_F_SG [FRAU]
6: ADJ_AKK_F_SG_DEF* (())
5: ART_AKK_F_SG_DEF [DIE]
4: ADV* (())
3: V_1_SG_A [SEHE]
2: NOMINATIV_1_SG ((1))
1: NOMINATIV_PRON_1_SG [ICH]
"ACCEPTED"
```

Das Resultat des Parse setzt sich folgendermaßen zusammen:

- Geparste Worte

- Parse-Zeit
- Anzahl der syntaktisch möglichen Bäume
- Die Wurzel des Satzes und die Anzahl der Parse-Knoten bevor alle unwichtigen Parse-Knoten weggeschnitten wurden.
- Die verbleibenden Parse-Knoten.
- Zurückgeliefert wird "ACCEPTED" oder "NOT ACCEPTED".

Ein Parse-Knoten besteht aus:

- Seiner Knotennummer
- Seiner Kategorie
- Der Referenz auf das jeweilige Wort oder eine Menge von Parse-Knoten-Pfaden.

Das Format dieser Ausgabe ist nahezu konstant in allen Parser-Versionen.

Literaturverzeichnis

[ADNS94] X. Aubert, C. Dugast, H. Ney, V. Steinbiß. *Large vocabulary continuous speech recognition of wall street jounal data.* In Proc. International conference on acoustics, speech and signal processing (ICASSP) 1994, pp. II 129 – II 132, 1994.

[AJ90] Abney, Johnson. *Eager Parsing.* In CUNY conference on sentence processing, New York, 1990.

[AU72] Aho, Ullman. *Theory of parsing, translation and compiling.* Prentice-Hall, Englewood Cliffs, 1972.

[BC93] Ted Briscoe, John Carroll. *Generalized Probabilistic LR Parsing of Natural Language (Corpora) with Unification-Based Grammars.* Computational Linguistics, pp. 25–59, March 1993.

[BL89] Billot, Lang. *The Structure of shared forests in ambiguous parsing.* In Proc. the 27th meting of the association for computational linguistics, Vancouver, Canada, 1989.

[BW84] Berwick, Weinberg. *The Grammatical Basis of Linguistic Performance.* MIT Press, Cambridge, Mass., 1984.

[CCL93] Lee-Feng Chien, Keh-Jiann Chen, Lin-Shan Lee. *A best-first language processing model integrating the unification grammar and markov language model for speech recognition applications.* In IEEE Transactions on Speech and Audio Processing, volume 1,2, pp. 221–240, 1993.

[CS89] Yen-Lu Chow, Richard Schwartz. *The n-best algorithm: An efficient procedure for finding top n sentence hypothesis.* In DARPA Speech and Natural Language Workshop, pp. 199–202, Cape Cod, Mass., October 1989.

[Ear68] J. Earley. *An efficient context-free parsing algorithm.* Ph.D. thesis, Computer Science Department, Carnegie-Mellon University, 1968.

[Gö88] Günther Görz. *Strukturanalyse natürlicher Sprache.* Addison Wesley, Bonn, 1988.

[HU79] J. E. Hopcroft, J. D. Ullman. *Introduction to automata theory, languages, and computation.* Addison-Wesley, Reading, Mass., 1979.

[HW94] A. Hauenstein, H. Weber. *An investigation of tightly coupled time synchronous speech language interfaces.* In Proceedings of the CONVENTS 94, Wien, September 1994.

[Joh75] S. C. Johnson. Yacc – yet another compiler compiler. Technical Report CSTR 32, Bell Laboratories, 1975.

[Joh91] M. Johnson. The computational complexity of GLR parsing. In M. Tomita (ed.), Generalized LR Parsing, pp. 35–42. Kluwer Academic Publishers, 1991.

[Kip91] J. R. Kipps. GLR parsing in time $O(n^3)$. In M. Tomita (ed.), Generalized LR Parsing, pp. 43–60. Kluwer Academic Publishers, 1991.

[KKS89] K. Kita, T. Kawabata, H. Saito. *Hmm continuous speech recognition using predictive LR parsing.* In IEEE International Conference on Acoustics, Speech and Signal Processing, pp. 703–706, 1989.

[LAKA92] R. Leermakers, L. Augusteijn, F.E.J. Kruseman-Aretz. *A functional LR parser.* In Theoret. Comput. Sci 104, pp. 313–323, 1992.

[Lee92] R. Leermakers. *Recursive ascent parsing: from Earley to Marcus.* In Theoret. Comput. Sci 104, pp. 299–311, 1992.

[Ney91] Hermann Ney. *Dynamic programming parsing for context-free grammars in continuous speech recognition.* In IEEE Transactions on Signal Processing, pp. 336–340, February 1991.

[Ney93] H. Ney. *Architecture and search strategies for large-vocabulary continuous-speech recognition.* In NATO-ASI BUBIÓN, pp. 59 – 84, 1993.

[NF91] R. Nozohoor-Farshi. GLR parsing for ϵ -grammars. In M. Tomita (ed.), Generalized LR Parsing, pp. 61–76. Kluwer Academic Publishers, 1991.

[Nil82] N. Nilson. *Principles of Artificial Intelligence.* Tioga Publishing Company, 1982.

[Per85] Pereira. A new characterization of attachment preferences. In Karttunen Dowty, Zwicky (eds.), Natural Language Parsing. Cambridge University Press, Cambridge, England, 1985.

[PN89] A. Päseler, H. Ney. *Continuous-speech recognition using a stochastic language model.* In Proc. International conference on acoustics, speech and signal processing (ICASSP), 1989.

[PNea87] A. Päseler, H. Ney, et al. *A data driven organization of the dynamic programming beam search for continuous speech recognition.* In Proc. International conference on acoustics, speech and signal processing (ICASSP), 1987.

[Rab88] L. R. Rabiner. Mathematical foundations of Hidden Markov Models. In H. Niemann, M. Lang, G. Sagerer (eds.), Recent advances in speech understanding and dialog systems, pp. 183–205. Springer, Heidelberg, 1988.

[RJ86] L.R. Rabiner, B. H. Juang. *An introduction to hidden Markov Models.* IEEE ASSP Magazine, pp. 4–16, January 1986.

[Sch90] Y. Schabes. *Mathematical and Computational Aspects of Lexicalized Grammars.* Dissertation, University of Pennsylvania, Philadelphia,PA, 1990.

[Sch94] Ludwig Schmid. *Parsing word graphs using a linguistic grammar and a statistical language model.* In Proc. International conference on acoustics, speech and signal processing (ICASSP) 1994, pp. II 41 – II 44, August 1994.

[Sen89] S. Seneff. *TINA: a probabilistic syntactic parser for speech understanding systems.* In Proc. International conference on acoustics, speech and signal processing (ICASSP), 1989.

[Sha91] P. Shann. Experiments with GLR and chart parsing. In M. Tomita (ed.), Generalized LR Parsing, pp. 17–34. Kluwer Academic Publishers, 1991.

[Shi83] Shieber. *Sentence disambiguation by a shift-reduce parsing technique.* In ACL Proc., The 21st annual meeting of the ACL, 1983.

[Shi87] K. Shikano. *Improvement of word recognition results by trigram model.* In Proc. International conference on acoustics, speech and signal processing (ICASSP), pp. 1261–1264, 1987.

[SJNWC91] K.-Y. Su, M.-H.Su J.-N. Wang, J.-S. Chang. GLR-parsing with scoring. In M. Tomita (ed.), Generalized LR Parsing, pp. 93–112. Kluwer Academic Publishers, 1991.

[Tho90] H. Thompson. *Best-first enumeration of paths through a lattice – an active chart parsing solution.* Computer Speech and Language, 4: 263–274, 1990.

[Tom85] M. Tomita. *Efficient parsing for natural language.* Kluwer, 1985.

[Tom86] M. Tomita. *An efficient word lattice parsing algorithm for continuous speech recognition.* In Proc. International conference on acoustics, speech and signal processing (ICASSP) 1986, pp. 115–120, August 1986.

[Tom87] M. Tomita. *An efficient augmented context-free parsing algorithm.* Computational linguistics, 13(1-2): 31–46, 1987.

[Web94] H. Weber. *LR-inkrementelles, probabilistisches Chartparsing von Worthypothesenmengen mit Unifikationsgrammatiken: Eine enge Kopplung von Suche und Analyse.* Dissertation, Universität Hamburg, 1994.

[Wri90] J. H. Wright. *LR parsing of probabilistic grammars with input uncertainty for speech recognition.* Computer Speech and Language, 4: 297–323, 1990.

[WW91a] J. H. Wright, E. N. Wrigley. Glr parsing with probability. In M. Tomita (ed.), Generalized LR Parsing, pp. 113–128. Kluwer Academic Publishers, 1991.

[WW91b] E. N. Wrigley, J. H. Wright. *Computational requirements of probabilistic LR parsing for speech recognition using a natural language grammar.* In Proc. Eurospeech '91, 2nd European Conference on speech communication and technology, Genova, pp. 761–764, 24-26 September 1991.

Index