Entwicklungsumgebungen für NL-Parser

Magisterarbeit im Fach
Computerlinguistik und Künstliche Intelligenz
zur Erlangung des
Magister Artium

im Fachbereich
Sprach- und Literaturwissenschaft
der Universität Osnabrück

vorgelegt von:
Wilfried Teiken
aus Haren (Ems)

1998

Inhaltsverzeichnis

1. Einleitung **1**

1.1. Motivation 1

1.2. Ein Überblick 2

2. Parsing natürlicher Sprache **5**

2.1. Definition des Begriffs Grammatik 5

2.2. Definition des Begriffs Lexikon 7

2.3. Definition des Begriffs Parsing 7

2.4. Besonderheiten beim Parsing natürlicher Sprache 10

2.5. Parsingalgorithmen 10

2.5.1. Grundlegende Parsingprinzipien 11

2.5.2. Deterministisches Parsing 12

2.5.3. Chart-Parsing 15

2.5.4. Der Earley-Parser 16

2.6. Exkurs: Features und Featurestrukturen 20

2.6.1. Definition von Featurestrukturen 20

2.6.2. Featurestrukturen als Verband 24

2.6.3. Zusammengesetzte Featurestrukturen 25

2.6.4. Getypte Featurestrukturen 26

2.7. Unifikationsgrammatiken 28

3. Der Parser **31**

3.1. Das Lexikon 31

3.1.1. Einfache Lexikoneinträge 32

3.1.2. Defaults 34

3.1.3. Templates 35

3.1.4. Explizite Wertlöschung 36

3.1.5. Morphologische Fähigkeiten des Lexikons 36

3.1.6. Mehrworteinträge im Lexikon 40

3.2. Der Parserkern 40

3.3. Modularisierung von Grammatik und Lexikon 42

3.4. Parsing fehlerhafter Eingaben 43

3.4.1. Ein unifikationsbasierter Ansatz 44

3.4.2. Ein regelbasierter Ansatz 46

3.4.3. Ein weiterer regelbasierter Ansatz 48

3.5. Funktionale Ausdrücke 48

3.6. Formale Betrachtung der Systemerweiterungen 49

3.6.1. Typisierte Features 49

3.6.2. Wertdisjunktionen 50

3.6.3. Neue Featuretypen 50

3.6.4. Mehrwortlexeme 52

3.6.5. Unifikationsbasierte Fehlererkennung 52

3.6.6. Regelbasierte Fehlererkennung 52

3.6.7. Funktionen und Deklarativität 53

4. Entwicklungsumgebungen **55**

4.1. Anforderungen an eine Entwicklungsumgebung 55

4.2. Bearbeitung und Visualisierung der Daten 56

4.3. Testumgebungen für Parser 57

4.3.1. Testsatzsammlungen 58

5. Die Entwicklungsumgebung **65**

5.1. Ein Systemüberblick 65

5.2. Visualisierung der Daten 66

5.3. Definition der Testsatzmengen 69

5.4. Auswertung der Testläufe 71

6. Ausblick **73**

A. Notation **75**

A.1. Verwendete Begriffe 75

A.2. Konstanten und Symbole 77

A.3. Operatoren 78

A.4. Strukturen 78

B. Die Syntax des Formalismus **81**

B.1. Der Grammatikformalismus 81

B.1.1. Deklarationen 81

B.1.2. Featurestrukturen 84

B.1.3. Bezeichner und Strings 87

B.1.4. Pfade 87

B.1.5. Funktionsaufrufe 88

B.1.6. Listen 89

B.1.7. Kommentare 89

B.1.8. Die Grammatik 90

B.1.9. Das Lexikon 94

B.2. Funktionsbibliothek 96

C. Beispiele **99**

C.1. Die Schnittstellendatei 99

C.2. Defaults und Templates 100

C.2.1. Defaults 100

C.2.2. Templates 101

C.3. Ein Lexikon 103

C.4. Eine Grammatik 109

C.4.1. Die Grammatikmodule 109

1. Einleitung

1.1. Motivation

Eine der immer wiederkehrenden Aufgaben für einen Computerlinguisten ist die Entwicklung eines Parsers. Dabei wird bei dieser Aufgabe das Rad immer wieder neu erfunden, da bisher kaum Methoden zur Grammatikerstellung entwickelt wurde, wie sie z.B. inzwischen in der Informatik (mehr oder weniger) etabliert sind. Dies führt dann oft zu Systemen, die nicht wartbar[1] und nur in den seltensten Fällen wiederverwertbar[2] sind. Ein Grammatikformalismus muß Methoden zur Verfügung stellen, um diese Probleme zu lösen. Eine sinnvolle Entwicklungsumgebung für Grammatiken muß den Grammatikentwickler außerdem dabei unterstützen, den Überblick bei der Grammatikentwicklung zu behalten. Dies beinhaltet sowohl eine die Abhängigkeiten darstellende Visualisierung der Grammatik als auch Tools zur Analyse der Grammatik, d. h. die Struktur einer Grammatik muß verständlich und übersichtlich dargestellt werden.

Ein weiteres Problem bei der Entwicklung von großen Grammatiken ist die Koordination der Zusammenarbeit. Ein System, das die Zusammenarbeit mehrerer Entwickler unterstützen will, muß eine Aufteilung der Grammatik in mehrere *Module*[3] ermöglichen. Dabei muß jeder Entwickler einen bestimmten Stand der Grammatik für seine Arbeit festschreiben können, um unabhängig von der Arbeit anderer zu bleiben. Falls dies nicht möglich ist stört jede Änderung, die in anderen Modulen vorgenommen wird, die eigene Arbeit.

Als letzte Anforderung bleibt noch ein Mechanismus, um automatische Tests des entwickelten Systems durchzuführen. Ein Softwarepaket mit einer Komple-

[1] Unter der Wartbarkeit einer Grammatik verstehe ich analog zur Wartbarkeit eines Softwaresystems die Eigenschaft an neue Anforderungen anpaßbar zu sein.

[2] Wiederverwertbarkeit heißt im folgenden, daß ein Teil des Systems aus dem Ganzen herauslösbar ist, um in einem anderen System die gleiche Aufgabe zu erfüllen.

[3] Ein Modul ist ein Teil eines Gesamtsystems, das klar definierte Schnittstellen, d. h. Kommunikationsmethoden, besitzt.

xität wie sie bei größeren Grammatiken vorliegt, kann nicht mehr allein durch intuitive Tests mit ad hoc gewählten Sätzen verifiziert werden. Es müssen Tests in einem *Batchverfahren*[4] durchgeführt werden könne, die systematisch die implementierten Phänomenbereiche abdecken. Daher muß ein Entwicklungssystem eine Unterstützung bei der Entwicklung und der Anwendung der Tests zur Verfügung stellen.

Diese Anforderungen erinnern an die Anfänge der Softwareindustrie, die vor dem Durchbruch des *Software Engineering* mit Problemen zu kämpfen hatte, die denen bei der Grammatikentwicklung gleichen. Daher liegt die Idee nahe, die in der Softwareentwicklung erfolgreich angewendeten Konzepte auf die Grammatikentwicklung zu übertragen. Durch diese ingenieursmäßige Herangehensweise wurde der Begriff *Grammar Engineering* geprägt.

Im Rahmen dieser Magisterarbeit ist ein System entwickelt worden, das versucht, zumindest einen Teil dieser Anforderungen zu erfüllen. Es stellt Tools zur Grammatikentwicklung zur Verfügung, die denen der Softwareentwicklung entsprechen. Ein Schwerpunkt der Implementation liegt darauf, das System einfach bedienbar zu halten, da es einem Anwenderkreis gerecht werden muß, der den Computer als reines (und möglichst einfach zu bedienendes) Werkzeug sieht, das nicht die zentrale Rolle des Computers in der Informatik spielt.

1.2. Ein Überblick

In dieser Arbeit werden zwei Teilaspekte, die für ein solches System wesentlich sind, untersucht. Der eine Aspekt sind die Methoden zur Analyse der Spracheingaben[5], d.h. die dabei verwendeten Algorithmen. Der zweite Aspekt ist die Unterstützung des Entwicklers bei der Erstellung eines solchen Systems. Dies beinhaltet Methoden zur Visualisierung der Zusammenhänge in einem so komplexen Systems und eine möglichst effiziente Hilfe bei der Suche nach Fehlern, die bei der Entwicklung komplexerer Systeme nahezu zwangsläufig auftreten. Ein wichtiges Mittel bei der systematischen Fehlersuche oder der Erstellung solcher Systeme sind Testumgebungen, die das korrekte Verhalten eines Sprachverarbeitenden Systems anhand von Testeingaben überprüfen.

Der erste Teil dieser Arbeit befasst sich mit dem Parsing natürlicher Sprache und den dabei verwendeten Algorithmen. Daran schließt sich in Kapitel 3 ei-

[4] Ein Batchverfahren ist eine automatische Durchführung einer großen Menge an Einzelaufgaben ohne Interaktion mit einem Benutzer. In diesem Falle ist ein Batchverfahren die Durchführung eines Analyse einer großen Menge an Testsätzen.

[5] Im weiteren wird von Eingaben in maschinenlesbarer Form ausgegangen.

ne Beschreibung der Implementation eines Parsers an. Diese Implementation ermöglicht z. B. die Aufteilung der Grammatik in mehrere, voneinander unabhängige Module. Die beiden nachfolgenden Kapitel befassen sich mit Testumgebungen für Grammatiken. In Kapitel 4 wird ein Überblick über bestehende Entwicklungsumgebungen für Grammatiken gegeben, die an verschiedenen Instituten entwickelt wurden. Die bei der Analyse der verschiedenen Systeme gemachten Erfahrungen führten zu einer Implementation einer Entwicklungsumgebung für Grammatiken, wie sie der in Kapitel 3 vorgestellte Parser verwendet. Diese Implementation wird in Kapitel 5 vorgestellt.

Im Anhang befinden sich eine etwas umfangreichere Beschreibung der Syntax des entwickelten Grammatikformalismus sowie eine Beispielgrammatik, die die Verwendung der verschiedenen Möglichkeiten des Systems aufzeigt.

Die Implementation stellt eine Systemumgebung dar, mit der eine Sprachverarbeitungskomponente – der Teil eines Systems der die Sprachanalyse durchführt – entwickelt werden kann. Das System in seinem momentanen Zustand wird in einem Bibliotheksinterfacesystem zur Analyse der Spracheingaben verwendet und dabei auf seine Praxistauglichkeit hin untersucht. Die bei diesem Einstatz gemachten Erfahrungen werden in die Weiterentwicklung der Bedienungsoberfläche integriert, um die Effizienz[6] des Systems zu steigern.

Bei all diesen Verfahren steht der informatische Aspekt der Sprachverarbeitung im Vordergrund, d.h. es wurde im Rahmen dieser Arbeit keine komplexe Grammatik entwickelt, die hier vorgestellten Grammatiken dienen lediglich zur Demonstration der Möglichkeiten des im Rahmen dieser Magisterarbeit erstellten Systems. Ein besonderer Dank geht an dieser Stelle noch an Dr. Hagen Langer, auf dessen Ideen – und deren erste Implementationen – ein großer Teil des in dieser Arbeit vorgestellten Parsers beruht.

[6] Mit dem Begriff Effizienz ist in diesem Fall nicht die Ausführungsgeschwindigkeit sondern die optimale Unterstützung des Entwicklers gemeint.

2. Parsing natürlicher Sprache

Am Anfang dieser Arbeit soll zunächst versucht werden, den Begriff Parsing zu klären. In einem ersten Ansatz könnte man Parsing etwa so definieren:

> Parsing ist die Generierung einer Menge von strukturellen Beschreibungen einer Eingabe anhand vorgegebener Regeln.

Um diese Definition formaler fassen zu können, müssen zunächst noch einige andere Begriffe geklärt werden.

Definition 2.0.1 (Eingabe) *Die Eingabe $\mathcal{I}$ beim Parsing ist eine Sequenz aus dem Eingabealphabet Σ, also $\mathcal{I} = a_1 a_2 \cdots a_n \in \Sigma^*$.*

Neben der Eingabe muß nun noch die Art der Regeln geklärt werden, die beim Parsing benutzt werden. Dazu sind noch die Begriffe *Grammatik* und *Lexikon* notwendig. Diese Begriffe werden so definiert, wie es im Rahmen dieser Arbeit am sinnvollsten erscheint, also für die Analyse von geschriebener, in maschinenlesbarer Form vorliegender Sprache. In anderen Kontexten können diese Definitionen durchaus unzureichend sein.

2.1. Definition des Begriffs Grammatik

Definition 2.1.1 ((kontextfreie) Grammatik) *Eine Grammatik ist eine Struktur*

$$G =< V_N, V_T, S, R >, V_N \cap V_T = \emptyset, S \in V_N.$$

Dabei ist V_N eine Menge von nichtterminalen Symbolen, die durch Regeln expandiert werden können, V_T ist eine Menge von nicht weiter expandierbaren – terminalen – Symbolen. R ist eine Menge von Regeln der Form

$$L \rightarrow R_1 \ldots R_n, \text{ mit } L \in V_N,\ R_i \in V = V_N \cup V_T\ \forall 1 \leq i \leq n,\ n \geq 0^1.$$

[1] In einigen Fällen wird auch der Doppelpunkt statt des $\rightarrow$ benutzt.

Die Ersetzungsregeln erlauben in einer Symbolkette aus V^* die Substitution eines Symbols L durch die Symbolkette $R_1 \dots R_n$[2]. Als Startkette dient die Kette S, d. h. die Kette die aus genau einem Symbol – S – besteht. Diese wird expandiert bis die Symbolkette nur noch aus terminalen Symbolen besteht. Grammatiken diesen Typs heißen kontextfreie Grammatiken (*kfG*), da die Regeln immer nur ein LHS-Symbol enthalten[3]. Der Kontext[4] des Symbols hat daher keinen Einfluß auf die möglichen Ableitungen.

Die Ersetzungsoperation, auch Ableitungsschritt genannt, wird in der Regel durch das Symbol $\Rightarrow$ gekennzeichnet. $L \Rightarrow R_1 \dots R_n$ erzeugt aus der Kette L die Kette $R_1 \dots R_n$. Bei einer endlichen Folge von Ersetzungen wird das Symbol $\Rightarrow^*$ verwendet. Wird dabei immer das am weitesten rechts stehende Nichtterminal ersetzt, dann wird von einer Rechtsableitung – symbolisiert durch $\Rightarrow_{rm}$ – gesprochen.

Definition 2.1.2 (Sprache (einer Grammatik)) *Die Menge*

$$L = \{w \mid w \in V_T^* \wedge S \Rightarrow^* w\},$$

also die Menge aller erzeugbaren terminalen Ketten, wird als die von G erzeugte Sprache $L(G)$ bezeichnet.

Sprachen werden wie Grammatiken in verschiedene Kategorien unterteilt. Die zu den kfG gehörenden Sprachen heißen kontextfreie Sprachen[5] (*kfS*). Dabei ist allerdings zu beachten, daß eine Sprache nur dann nicht kontextfrei ist, wenn sie durch keine kfG erzeugt werden kann. Dies ist nicht direkt gegeben, wenn eine Grammatik, die nicht kontextfrei ist, die Sprache erzeugt, da es eine kontextfreie Grammatik mit der gleichen Sprache geben könnte.

Bei einigen Definitionen von kfG werden Regeln der Form $A \to \varepsilon$, auch ε-Produktionen oder Tilgungsregeln genannt, explizit verboten. Dies hat – außer im Fall $\varepsilon \in L(G)$ – keine Auswirkungen auf die Sprache, da man zu einer

[2] Da das Symbol L auf der linken Seite des Ersetzungspfeils steht wird es *left hand symbol* (*LHS*) genannt. Die Symbole R_i heißen *right hand symbols* (*RHS*).

[3] Der Typ der Grammatik hängt im Wesentlichen von den erlaubten Symbolen auf der rechten und linken Seite des Ableitungspfeils ab. Mehr zu Produktionssystemen und verschiedenen Grammatiktypen findet sich in (Hopcroft und Ullman, 1979) oder (Chomsky, 1956) und (Chomsky, 1963).

[4] Unter dem Kontext eines Symbols versteht man die Symbole links und rechts dieses Symbols.

[5] Die Ketten der Form $a^n b^m c^n$ können z. B. von einer kfG erzeugt werden, die Ausdrücke der Form $a^n b^m c^n d^m$ nicht.

kontextfreien Grammatik mit ε-Produktionen auch immer eine Grammatik ohne Tilgungsregeln finden kann, die die gleiche Sprache erzeugt (vgl. Hopcroft und Ullman (1979)). Aus verschiedenen Gründen, die sowohl in linguistischer Motivation als auch in der Struktur der sprachverarbeitenden Algorithmen zu suchen sind, wird bei einigen Parsingsystemen die Anforderung der Tilgungsregelfreiheit gestellt. Falls im weiteren Verlauf der Arbeit die ε-Produktion erlaubt sein soll, wird dies explizit erwähnt.

2.2. Definition des Begriffs Lexikon

Definition 2.2.1 (Lexikon) *Das Lexikon ist eine Funktion*

$$\mathcal{L}ex : W \rightarrow \wp(V_T \times \Phi),$$

die einem Wort[6] $w \in W$ *eine Menge von Tupeln aus jeweils einem terminalen Symbol der Grammatik und einer Repräsentation des Wortes zuordnet.*

Diese Menge aus Tupeln kann als die Menge der möglichen Interpretationen aufgefaßt werden[7]. Im einfachsten Fall eines Lexikons ist die Repräsentation des Wortes leer, sie kann aber, wie im weiteren Verlauf der Arbeit gezeigt wird, komplexe Informationen enthalten. Die genaue Struktur ist für die Definition des Begriffs Parsing allerdings unwichtig. Die terminalen Symbole der Grammatik werden, da sie den Worten durch das Lexikon zugeordnet werden, auch als *lexikalische Kategorien* bezeichnet. Die Einträge im Lexikon werden *Lexeme* genannt.

2.3. Definition des Begriffs Parsing

Nachdem nun diese Begriffe geklärt sind, kann ein erneuter Versuch einer präzisen Definition von *Parsing* unternommen werden. Dabei ist aber noch unklar, wie die Eingabe genau aussieht. Die Eingabe besteht zunächst aus einer Folge von ASCII-Zeichen[8], diese Zeichen sind jedoch als Eingabealphabet ungeeignet

[6] Mit Wort ist hier nicht unbedingt ein Wort im linguistischen Sinn gemeint, es ist eher als eine Teilfolge der Eingabe anzusehen.

[7] Das Wort *hit* im Englischen kann sowohl ein Nomen als auch ein Verb sein. Daher sind hier mindestens zwei Elemente in der Ergebnismenge des Lexikons.

[8] Ein ASCII-Zeichen ist eine genormte Darstellung eines alphanumerischer Zeichen nach dem American Standard Code for Information Interchange. In diesem Standard ist für jedes Zeichen eine Zahl definiert, mit der es repräsentiert wird.

da man sich beim Parsing auf der Wortebene und nicht auf der Buchstabenebene befindet. Daher wird die Eingabe des Parsers durch eine Vorverarbeitung aus den tatsächlich zu analysierenden Daten erzeugt. Das Eingabealphabet beim Parsing natürlicher Sprache kann dabei entweder als die Menge W aller Worte, die das System kennt, oder als die Menge V_T aller lexikalischer Kategorien aufgefasst werden. Die Definition des Eingabealphabetes als die Menge aller lexikalischen Kategorien reduziert die Größe des Eingabealphabetes deutlich; außerdem schafft es eine klare Schnittstelle zwischen dem Lexikon, das die Analyse der Worte übernimmt, und dem Parser, der die Satzstruktur dieser voranalysierten Worte feststellt.

Die Trennung der Worte kann relativ sicher durch Leerzeichen[9] erfolgen. Die so separierten Worte werden dann durch das Lexikon in die für den Parser gültigen Symbole umgewandelt. Der Schritt der Wortseparation ist beim Parsing maschinenlesbarer Sprache – im Gegensatz zu gesprochener Sprache – eindeutig, daher kann immer von einer eindeutigen Tokenfolge ausgegeangen werden. Bei der Analyse gesprochener, natürlicher Sprache können die Wortgrenzen durch Verschleppung oder Auslassung von Silben nicht immer korrekt erkannt werden. Außerdem muß verschiedenen Lautfolgen ein Wort zugeordnet werden, oder eine Lautfolge kann verschiedenen Worten entsprechen. Die beim Verbmobil-Projekt verwendete Spracherkennungskomponente verwendet teilweise Neuronale Netze, um Worthypothesen zu bilden und dann mit dem Lexikon abzugleichen. Die Fehlerrate des Systems bei der Worterkennung lag 1996 bei 13,8% (vgl. Finke et al. (1997)). Die generierten Worthypothesen können daher mehrdeutig sein, d. h. ein Input kann durch verschiedene Token-Folgen dargestellt werden. Damit ergeben sich aus einem zu analysierenden Datum mehrere Eingaben für den Parser. Diese Sichtweise entsteht allerdings nur durch die Wahl des Eingabealphabetes, denn mit der Menge der Laute als Eingabealphabet ist hier ein direktes Parsig der Eigabe ohne einen Vorverarbeitungsprozess möglich.

Definition 2.3.1 (Parsing) *Parsing ist die Überführung einer Eingabe I in eine Menge von strukturellen Beschreibungen $\varphi \subset \Phi$:*

$$P : \Sigma^* \rightarrow \wp\Phi.$$

Wenn das Eingabealphabet Σ des Parsers aus den lexikalischen Kategorien besteht – d. h. es findet eine Trennung zwischen Parser und Lexikon statt – dann kann man diese Definition noch etwas erweitern:

[9] Als Leerzeichen zählen hierbei alle Zeichen, die auf dem Papier nicht sichtbar sind, sowie die Zeilenenden. Diese Zeichen werden in der Informatik oft *Whitespace* genannt.

Definition 2.3.2 (Parsing unter Verwendung eines Lexikons) *Es sei W eine Wortmenge und $\mathcal{L}ex : W \to V_T$ eine Lexikonfunktion. Dann ist Parsing ist die Überführung einer Eingabe $I \in W^*$ in eine Menge von strukturellen Beschreibungen $\varphi \subset \Phi$:*

$$P \circ \mathcal{L}ex : W^* \to \wp\Phi$$

Jedes Element der Ergebnismenge ist eine mögliche strukturelle Beschreibung der Eingabe, enthält sie mehr als ein Element, dann ist die Eingabe ambig.

Definition 2.3.3 (Ambiguität) *Eine Eingabe I ist genau dann ambig (bzgl. einer Grammatik), wenn durch die Grammatik mehr als eine strukturelle Beschreibung erzeugt werden kann, d. h.*

$$\mid P(I) \mid > 1.$$

Eine Grammatik ist genau dann ambig, wenn sie für mindestens eine Eingabe ambig ist.

Ist die Ergebnismenge beim Parsing leer, dann gibt es im Rahmen der gegebenen Grammatik keine strukturelle Beschreibung der Eingabe. In diesem Fall wird die Eingabe als *inkorrekt* bezeichnet, im anderen Fall heißt der Input *korrekt* oder *erzeugbar*. Im weiteren Verlauf der Arbeit wird, soweit nicht anders erwähnt, davon ausgegangen, daß die untersuchten Eingaben erzeugbar[10] sind.

Die strukturelle Beschreibung kann auf verschiedene Arten repräsentiert werden, z. B. durch Strukturbäume und Featurestrukturen (vgl. Abschnitt 2.6), wie sie in der Linguistik üblich sind. Falls man den Parsingvorgang durch ein entsprechend gewähltes Deduktionssystem realisiert, kann man einen so gefundenen Beweis ebenfalls als Strukturbeschreibung auffassen (vgl. Sikkel (1997)).

Die Struktur der Grammatik wird durch die Art des Parsers bestimmt. Man kann sie durch logische Formeln in einem automatischen Beweiser, als deterministischer Kellerautomat (*DKA*) oder durch Ersetzungsregeln für einen speziellen Parsing-Algorithmus repräsentieren. Im Falle von *DCG*-Grammatiken entspricht die Grammatik im Wesentlichen einem *Prolog*-Programm, das von einem entsprechenden Interpreter ausgeführt wird. Es muß daher nicht notwendigerweise ein expliziter Algorithmus für das Parsing vorhanden sein, da er oft implizit durch das System vorgegeben ist. Der Vorgang des Parsens ist eine

[10] Es wird hier der Begriff erzeugbar verwendet, um Mißverständnisse zwischen korrekt im grammatischen Sinne und korrekt im Sinne einer formalen Sprache zu unterscheiden.

spezielle Form der Suche (vgl. Naumann und Langer (1994)), sei es die Suche nach einem Beweis, die Suche nach den möglichen Ersetzungen oder – im Fall von *Prolog* – die Suche nach der richtigen Lösung im von den Prologklauseln aufgespannten Lösungsraum. Der Vorteil der expliziten Parsingalgorithmen ist dabei, daß sie eine für das Parsing möglichst optimale (Such-)Strategie verwenden können.

Systeme wie *YACC*[11] zeigen, daß die verschiedenen Betrachtungsweisen ineinander überführbar sind. Dieses System kann z. B. eine regelorientierte Darstellung einer Grammatik[12] in einen DKA übersetzen. Im weiteren Verlauf dieses Kapitels werden einige explizite Parsingalgorithmen vorgestellt; die impliziten Algorithmen sind für diese Arbeit nur bedingt relevant und werden hier nicht näher betrachet. In (Gazdar und Mellish, 1989) werden Ansätze dargestellt, in *Prolog* einen Parser als Automat oder durch DCG's zu implementieren.

2.4. Besonderheiten beim Parsing natürlicher Sprache

Beim Parsing formaler Sprachen, z. B. von Programmiersprachen, enthält die Ergebnismenge in der Regel genau eine Beschreibung, da die Semantik eines syntaktisch ambigen Programms nicht eindeutig definiert ist. Hat das Programm keine strukturelle Repräsentation, kann das Programm nicht übersetzt werden. Die Grammatiken, die zum Parsen einer formalen Sprache dienen, genügen daher meistens bestimmten Bedingungen, die in Abschnitt 2.5.2 näher erläutert werden.

Beim Parsing natürlicher Sprache ist die Existenz mehrerer struktureller Beschreibungen für einen Input, hier normalerweise ein Satz, der Normalfall. Als weitere Besonderheit beim Parsing natürlicher Sprache ist die Anforderung, in bestimmte Situationen auch inkorrekte (in diesem Fall eher fehlerhafte) Eingaben analysieren zu können (vgl. Abschnitte 3.4 und 5.3).

2.5. Parsingalgorithmen

In diesem Abschnitt wird eine kurze Darstellung von Parsingalgorithmen gegeben. Die Darstellung der verschiedenen Parsingalgorithmen ist an die Notation

[11] Yet Another Compiler Compiler, (vgl. Levine et al. (1992))

[12] Die Grammatiken, die *YACC* verarbeiten kann, müssen den LALR(1)-Bedingungen (vgl. Abschnitt 2.5.2) entsprechen, die eine Subklasse der kontextfreien Grammatiken bilden.

in (Sikkel, 1997) angelehnt; die Parsingverfahren werden dabei im Wesentlichen über sie beschreibende Deduktionssysteme charakterisiert. Daher kann, ähnlich wie bei Deduktionssystemen, die Realisierung des Algorithmus auf verschiedene Arten erfolgen, so daß die Effizienz des Algorithmus dabei stark von der Wahl der Datenrepräsentation abhängt. Es werden vor allem Chart-Parser betrachtet, über die Sikkel sagt: „One could say that the canonical implementation of a parsing schema is a chart parser." (Sikkel, 1997, S. 13)

Eine praktische Einführung in die verschiedenen Algorithmen, die sich eng an deren Implementierung in *LISP* oder *Prolog* anlehnt, wird in (Naumann und Langer, 1994) gegeben.

2.5.1. Grundlegende Parsingprinzipien

Es gibt einige grundlegende Prinzipien, die zur Klassifikation von Parsingalgorithmen genutzt werden können. Dies sind:

- Analyserichtung
- Verarbeitungsrichtung

Analyserichtung

Top-Down-Parser versuchen durch Expansion die Eingabekette aus dem Startsymbol abzuleiten, d. h. durch die Anwendung von Regeln der Form $X \rightarrow Y_1, \ldots, Y_n$ wird aus einer Kette, die nur aus dem Startsymbol besteht, schrittweise die Eingabekette $E_1, \ldots, E_n$. Ein Beispiel für einen solchen Parser ist der Earley-Parser, wie er in Kapitel 2.5.4 vorgestellt wird.

Bottom-Up-Parser reduzieren die Eingabekette schrittweise bis auf das Startsymbol. Hier werden die Regeln in der anderen Richtung, $Y_1, \ldots, Y_n \rightarrow X$, interpretiert. Es gibt natürlich auch Parsingverfahren, die diese beiden Vorgehensweisen kombinieren. Die Effizienz der Algorithmen hängt, wie bei Deduktionssystemen, von der geschickten Regelauswahl ab. Im Falle einer zu einem DKA kompilierten Grammatik ist die Top-Down-Richtung der Analyse komplett in die Zustände eingearbeitet. Die Analyse des Input durchläuft die so errechneten möglichen Zustände. Die weitere Verarbeitungsrichtung schließt daher im Bottom-Up-Verfahren alternativen aus (vgl. Abschnitt 2.5.2). Der in Kapitel 2.5.4 vorgestellte Parser arbeitet ebenfalls Bottom-Up.

Verarbeitungsrichtung

Ein zweites Merkmal ist die horizontale Analyserichtung beim Parsing. Viele Algorithmen verarbeiten die Eingabe von links nach rechts. Einige andere Systeme versuchen auch eine parallele Verarbeitung der Eingabe. Die horizontale Analyserichtung kann sich bei einem fehlerhaften Input, also einem Input, der durch die Grammatik nicht beschrieben werden kann, sehr stark auf das Analyseverhalten auswirken. Bei einem strikt von links nach rechts arbeitenden Algorithmus muß das System die Arbeit einstellen, falls es ein unbekanntes Token in der Eingabekette findet. Alle Token rechts von der Fehlerstelle bleiben daher unanalysiert. Ein parallel arbeitendes System kann dagegen auch in diesem Fall für den Rest des Inputs Teilstrukturen identifizieren. Natürlich können auch sequentiell arbeitende Systeme den fehlerhaften Input ignorieren; diese Fähigkeit muß den Algorithmen dann aber explizit gegeben werden, während die bei den parallel arbeitenden Systemen automatisch vorhanden ist. Der Earley-Parser ist ein klassischer unidirektionaler, der Inselparser (vgl. Naumann und Langer (1994)) bidirektional arbeitender Parser.

2.5.2. Deterministisches Parsing

Ein Sonderfall beim Parsing ist das sogenannte *deterministische* Parsing. In diesem Fall wird davon ausgegangen, daß weder globale noch lokale Strukturambiguitäten auftreten dürfen. Bei deterministischem LR-Parsing, d. h. dem deterministischen Parsen einer Eingabe von links nach rechts, muß das System bei jedem Symbol, das verarbeitet wird, entscheiden können, ob

- das Symbol auf den Stack gelegt wird (*Shift*) oder
- durch eine Regel $A \rightarrow B_1 \ldots, B_m$ die obersten m Symbole auf dem Stack zum Symbol A reduziert werden (*Reduce*). Dabei müssen die Symbole auf dem Stack natürlich mit den RHS-Symbolen der Regel übereinstimmen. Die m Symbole werden dabei vom Stack heruntergenommen, das Symbol A wird dort abgelegt.

Auf dem Stack liegen also alle Symbole, die noch nicht verarbeitet werden konnten. Das System kann dabei neben dem gerade zu verarbeitenden Symbol noch n Symbole als *Lookahead* benutzen, d. h. die Symbole dürfen, obwohl sie noch nicht verarbeitet sind, verwendet werden, um sich für Shift oder Reduce zu entscheiden. Die bekanntesten Grammatiksubtypen sind die $LR(n)$- und

die $LALR(n)$-Grammatiken, wobei n dabei der Länge des erlaubte Lookahead entspricht. Ein so arbeitender Parser heißt *Shift-Reduce-Parser*[13].

Die nötigen Einschränkungen gegenüber den kfG können durch folgendes Beispiel veranschaulicht werden:

$$\texttt{S} \rightarrow \texttt{A B} \tag{2.1}$$
$$\texttt{A} \rightarrow a \tag{2.2}$$
$$\texttt{A} \rightarrow \texttt{A } a \tag{2.3}$$
$$\texttt{B} \rightarrow ab \tag{2.4}$$

Bei $n = 0$ ist diese Grammatik (lokal) ambig, denn wenn ein a auf dem Stack liegt, kann nicht entschieden werden, ob es durch Regel (2.2) zu einem `A` reduziert werden soll, oder ob das nächste Zeichen – b – zusätzlich auf den Stack gelegt werden soll. Bei $n = 1$ wird diese Grammatik (lokal) eindeutig, da durch den Lookahead eines a oder b klar wird, ob das a entweder sofort durch Regel (2.2) zu einem `A` reduziert oder nach einem weiteren Shift bei dem nächsten Zeichen Regel (2.4) anzuwenden ist. Mit jedem weiteren a am Anfang der Regel (2.4) steigt das benötigte n um 1. Für Grammatiken, die die $LR(n)$-Bedingungen erfüllen, können allerdings grundsätzlich $LR(1)$-Grammatiken gefunden werden, die die gleiche Sprache erzeugen (vgl. Hopcroft und Ullman (1979)). Die Klasse der $LR(0)$-Grammatiken ist eine echte Teilmenge der $LR(1)$-Grammatiken, da sie die Teilmenge der präfixfreien Sprachen aus der Menge der kontextfreien Sprachen[14] erzeugen. Präfixfrei bedeutet hierbei, daß keine Sequenz, die in der Sprache vorkommt, der Anfang einer längeren Sequenz der Sprache sein darf. Im vorliegenden Fall ließe sich sogar eine $LR(0)$-Grammatik finden, indem Regel (2.4) durch $\texttt{B} \rightarrow b$ ersetzt wird.

Formal läßt sich die $LR(k)$-Eigenschaft folgendermaßen ausdrücken:
Eine kontextfreie Grammatik $G =< V_N, V_T, S, R >$[15] ist genau dann eine

[13] Es sind allerdings nicht alle Shift-Reduce-parser deterministisch. Dieser Begriff bezieht sich auf die Vorgehensweise, Symbole auf einen Stack zu legen und dann die obersten Stackelemente zu reduzieren. Der Tomita-Algorithmus (vgl. Tomita (1985)) verwendet einen graphorientierten statt eines linearen Stack, die grundlegende Funktionsweise entspricht aber der eines Shift-Reduce-Parsers.

[14] Die $LR(1)$-Sprachen werden auch die deterministischen kontextfreien Sprachen (*dkfS*) genannt.

[15] Dabei wird vorausgesetzt, daß es nur eine Regel mit S als LHS gibt. Ist dies nicht der Fall, dann kann G in die Grammatik $G' =< V_N \cup \{S'\}, V_T, S', R \cup \{S' \rightarrow S\} >$ umgewandelt werden, die diese Bedingung erfüllt.

$LR(k)$-Grammatik, wenn aus

$$\begin{aligned} S &\Rightarrow^*_{rm} \alpha A w \Rightarrow_{rm} \alpha\beta w \\ S &\Rightarrow^*_{rm} \gamma B x \Rightarrow_{rm} \gamma\beta y \end{aligned}$$

mit $\text{init}_k(w) = \text{init}_k(y)$[16] immer folgt, daß

$$\alpha A y = \gamma B x, \text{ mit } \alpha = \gamma, A = B \text{ und } x = y.$$

Die $LR(1)$-Grammatiken[17] sind äquivalent zu den DKA (vgl. Knuth (1965)), daher liegt es nahe zum Parsen der Grammatik einen DKA zu verwenden. Die Transformation der Grammatiken in den DKA kann zudem automatisch erfolgen, dies wird z. B. in (Naumann und Langer, 1994) gezeigt. Ein Problem der so generierten DKA ist die hohe Anzahl von Zuständen, aus denen der DKA besteht. Daher wurde nach Unterklassen der $LR(1)$-Grammatiken gesucht, die bessere Eigenschaften in Bezug auf die Umwandlung in einen DKA besitzen. In (De Remer, 1969) und (De Remer, 1970) wurden entsprechende Unterklassen vorgestellt: die $SLR(1)$ und $LALR(1)$-Grammatiken. Da die oben vorgestellte Grammatik die $LALR(1)$-Eigenschaft besitzt, kann sie von *YACC* bearbeitet werden. Der erzeugte Zustandsgraph findet sich in Abbildung 2.1.

Die Zustände des DKA kodieren die Information darüber, welche Regeln bei welchem Input angewendet werden können. Mit jedem Knoten ist eine Menge von Regeln verknüpft, bei denen der Stack noch zu den RHS-Symbolen paßt. Der DKA stellt eine statische Top-Down-Analyse eines Input dar, die dann bei der Bottom-Up-Analyse des tatsächlichen Inputs genutzt wird, um ohne weitere Referenz auf die Regeln den Input analysieren zu können. Die einzige Information, die aus den Regeln noch genutzt wird, ist die Anzahl der RHS-Symbole und das LHS-Symbol.

Bei beliebigen kontextfreien Grammatiken, die nicht den LR-Bedingungen entsprechen, kann es in einem Zustand für ein Eingabesymbol mehrere mögliche Folgezustände geben. Im Tomita-Algorithmus (vgl. Tomita (1985)) wird das beim deterministischen Parsen verwendete Prinzip erweitert, indem statt eines linearen ein graphorientierter Stack verwendet wird. Dies erlaubt parallele Analyse mehrerer Hypothesen für einen gegebenen Input, indem bei mehreren möglichen Folgezuständen ein paralleler Stack aufgebaut wird. Der Tomita-Algorithmus bildet eine Brücke zwischen dem vor allem für formale Sprachen wichtigen deterministischen Parsen und der Analyse natürlicher Sprachen.

[16] $\text{init}_k(w)$ bezeichnet die Kette der ersten k Symbole von w.

[17] Ab jetzt werden nur noch $LR(1)$-Grmmatiken betrachtet, da die Menge der durch $LR(k)$-Grammatiken beschreibbaren Sprachen für $k > 1$ identisch zu den Sprachen bei $k = 1$ ist (vgl. Hopcroft und Ullman (1979)).

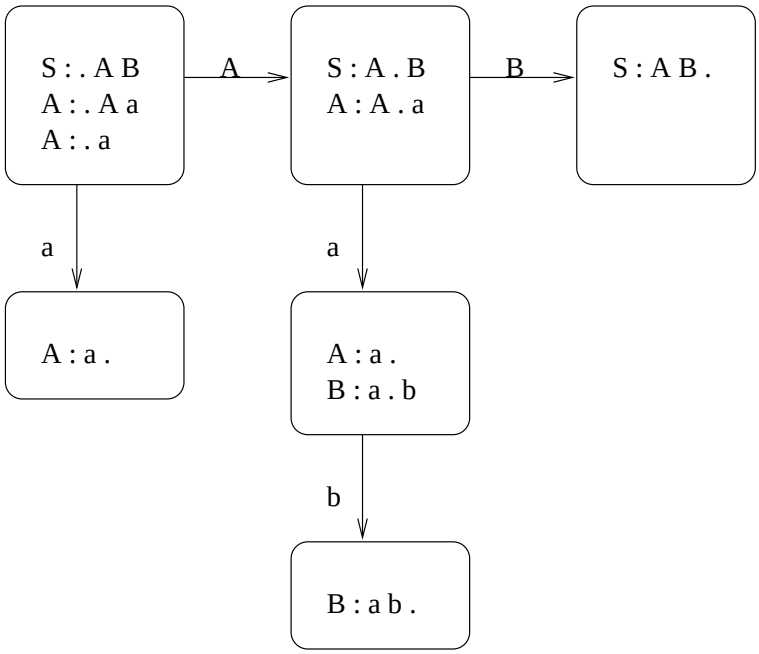


Abbildung 2.1.: Der zur Beispielgrammatik gehörende DKA

2.5.3. Chart-Parsing

Die naive Möglichkeit, nach einer Baumstruktur zu suchen, ist die Aufzählung aller möglichen Bäume und die anschließende Verifikation des so errechneten Baumes anhand des gegebenen Input. Neben dem Problem bei der Aufzählung aller Bäume zu einer Grammatik[18] ist der Nachteil dieses Verfahrens die Mehrfachanalyse eines Satzteiles, falls eine Hypothese während des Parsens sehr spät scheitert (vgl. (Naumann und Langer, 1994, S. 102)). Um diese Mehrfachanalyse zu vermeiden, werden die bisherigen Analyseergebnisse zwischengespeichert, um bei späteren Analysen des Satzteiles auf diese Informationen zurückgreifen zu können.

Eine Chart ist eine Struktur, die Objekte, die als Kanten oder auch Chartitems bezeichnet werden, speichert. Jede Kante repräsentiert eine (teilweise) erkannte Konstituente durch Informationen über den Anfangspunkt, den Endpunkt, die verwendete Regel und den momentan erkannten Teil der Regel.

[18] Durch eine Grammatik können oft unendlich viele Bäume erzeugt werden. Daher muß die Aufzählung so gewählt werden, daß dabei möglichst alle Variationen nach ihrer Komplexität geordnet aufgezählt werden. Die Problemfälle sind Links- oder Rechtsrekursion beim Top-Down-Parsing und ε-Produktionen beim Bottom-Up-Parsing.

Formal läßt sich ein Chartitem folgendermaßen darstellen:

$$[A \to \alpha \bullet \beta, i, j],$$

dabei ist $A \to \alpha\beta\}$[19] eine Regel der Grammatik, i die Anfangsposition der Konstituente und j der Endpunkt. Der Anfangspunkt für eine Kante, die bei einem Input $a_1 \cdots a_n$ die Elemente $a_{i+1} \cdots a_j$ umfaßt, ist i, der Endpunkt ist j. Das Zeichen „$\bullet$“ markiert, welcher Teil der Regel schon korrekt erkannt wurde, er wird oft als *Dot* bezeichnet. Alle Kanten mit $\beta \neq \varepsilon$, also alle Kanten mit noch nicht vollständig erkannten Regeln, heißen aktiv. Die komplett erkannten Kanten, die die gültige Konstituenten des Satzes darstellen, heißen passiv. Die Kanten werden während des Parsings in die Chart eingetragen. Falls auf die Informationen schon einmal bearbeiteter Satzteile zurückgegriffen werden muß, reicht ein Durchsuchen der Chart statt der Neuanalyse der Satzteile.

Vom mathematischen Standpunkt aus betrachtet stellen Chartitems Kongruenzklassen von Teilbäumen dar. Bei einem Chartitem $[A \to \alpha\bullet, i, j]$ ist die genaue Baumstruktur nicht mehr kodiert. Durch das Chartitem sind alle Teilbäume, die die Regel $A \to \alpha$ bis zum Input $a_i, \ldots, a_j$ expandieren, äquivalent (vgl. Sikkel (1997)).

Ein Chartparser für eine Grammatik $< V_N, V_T, S, R >$ und einen Input $a_1, \ldots, a_n$ ist ein Deduktionssystem $< \mathcal{I}, H, D >$. $\mathcal{I}$ ist die Domäne (hier die Menge der möglichen Items), H eine Menge von Hypothesen (hier initiale Items, die den Input repräsentieren) und $D \subset \wp_{fin}(H \cup \mathcal{I}) \times \mathcal{I}$ eine Menge von Ableitungsregeln. Die Ableitungsregeln ergeben sich aus dem verwendeten Parsingalgorithmus. Aus den Hypothesen, also den initialen Chartitems, die aus dem Input gewonnen werden, wird durch die Ableitungsregeln versucht, das Chartitem $[S \to \alpha\bullet, 0, n]$ abzuleiten.

2.5.4. Der Earley-Parser

In diesem Abschnitt werden verschiedene Varianten des Earley-Parsers vorgestellt, die die verschiedenen Parsingprinzipien deutlich machen. Im Vergleich der sehr ähnlichen Vorgehensweisen wird deutlich, daß die Grundstruktur eines Algorithmus nicht fest mit der Wahl der Parsingprinzipien verbunden ist.

[19] $\alpha, \beta, \gamma \in (V_N \cup V_T)^*;\ a, b \in V_T;\ A, B \in V_N$.

Der klassische Earley-Parser

Der *Earley-Parser* dient zur Analyse von Sprachen, die mit einer kfG beschrieben werden können.

Definition 2.5.1 (Earley-Parser) *Der Earley-Parser ist ein System* $< \mathcal{I}, \mathcal{K}, D >$ *mit folgenden Elementen:*

$$
\begin{aligned}
\mathcal{I} &= \{[A \to \alpha \bullet \beta, i, j] | A \to \alpha\beta \in R \wedge 0 \leq i \leq j\} \\
D^{init} &= \{\vdash [S \to \bullet\gamma, 0, 0]\} \\
D^{scan} &= \{[A \to \alpha \bullet a\beta, i, j], [a, j, j+1] \vdash [A \to \alpha a \bullet \beta, i, j+1\} \\
D^{compl} &= \{[A \to \alpha \bullet B\beta, i, j], [B \to \gamma\bullet, j, k] \vdash [A \to \alpha B \bullet \beta, i, k]\} \\
D^{pred} &= \{[A \to \alpha \bullet B\beta, i, j] \vdash [B \to \bullet\gamma, j, j]\} \\
D &= D^{init} \cup D^{scan} \cup D^{compl} \cup D^{pred} \\
\alpha, \beta, \gamma &\in V^* \\
a &\in V_T \\
A, B &\in V_N
\end{aligned}
$$

Die Funktion $\mathcal{K}$ ist eine Funktion, die einer Eingabekette $a_1, \ldots, a_n$ eine Hypothese $H = \mathcal{K}(a_1, \ldots, a_n)$ zuordnet. Diese Funktion ist definiert als:

$$\mathcal{K}(a_1, \ldots, a_n) = \{[a, i-1, i] \mid a = a_i \wedge 0 < i \leq n\}.$$

Damit hat man ein instantiiertes Parsingsystem wie in Abschnitt 2.5.3 beschrieben.

Der Parsingvorgang kann grob in drei Schritte eingeteilt werden:

1. Im *Prediction*-Schritt werden Top-Down die möglichen Hypothesen erzeugt. Dies geschieht durch die Ableitungen in der Menge D^{pred}. Bei allen so erzeugten Items steht der Dot am Anfang der Regel, d. h. es wurde noch keine der Konstituenten erkannt.

2. Bei allen Items, bei denen das nächste Terminal in der Inputkette mit dem in der Regel erwarteten Terminal übereinstimmt[20], wird der Dot um

[20] Wichtig ist dabei, daß der rechte Rand des Chartitems und die Satzposition des Wortes übereinstimmen. So kann eine Regel nicht mit Lücken, d. h. durch das Überspringen von Worten, erkannt werden.

ein Element nach rechts versetzt. Damit steht er hinter dem erkannten Terminal. Dies ist der *Scan*-Schritt, in dem ein neues Chartitem erzeugt wird, dessen rechter Rand nicht mehr bei j sondern bei $j+1$ liegt.

3. In einem dritten Schritt wird bei allen Items, bei denen alle Konstituenten korrekt erkannt wurden, das LHS-Symbol an die *darüberliegenden* Regeln weitergereicht. Bei diesen Regeln, die als nächstes Symbol dies durch Regelkomplettierung gefundenes Nichtterminal als *aktives Symbol*[21] erwarten, wird ebenfalls der Dot versetzt. Dies muß evtl. rekursiv wiederholt werden, wenn dadurch bei der so expandierten Regel alle Konstituenten erkannt wurden. Dabei wird aus einem Item von i bis j und einem Item von j bis k, das gerade komplett erkannt wurde, ein neues Item von i bis k erzeugt.

Durch iterierte Abarbeitung dieser Schritte wird das Startsymbol `S` zum Satz expandiert, indem von links nach rechts die Chart mit neuen Items aufgefüllt wird. Kann ein Symbol an der Position i nicht erkannt werden, können für die weiteren erkannten Terminale keine Regeln mehr vorhergesagt werden, da in der Chart ein Loch an der Stelle i existiert. Es gibt daher kein Item mit $i+1$ als rechtem Rand, daher auch keine Prediction mehr mit $i+1$ als linkem Rand.

Wenn auch Grammatiken mit ε-Produktionen benutzt werden sollen, kann die Menge D des Parsers noch um folgendes Element erweitert werden:

$$D^{init-\varepsilon} \quad = \quad \{\vdash [A \to \varepsilon\bullet, i, i] \mid A \to \varepsilon \in R\}.$$

Damit wird jede Konstituente, die in ein ε überführt werden kann, an jeder Position ohne Prämisse ableitbar.

Bottom-Up Earley-Parser

Es gibt eine Bottom-Up-Variante des Earley-Parsers, der eine parallele Bearbeitung der Worte erlaubt. Der Algorithmus hat folgende Form:

[21] Das aktive Symbol ist das erste noch nicht erkannte Symbol einer aktiven Kante.

Definition 2.5.2 (Bottom-Up Earley-Parser)

$$
\begin{aligned}
\mathcal{I} &= \{[A \to \alpha \bullet \beta, i, j] | A \to \alpha\beta \in R \wedge 0 \leq i \leq j\} \\
D^{init} &= \{\vdash [A \to \bullet\alpha, j, j]\} \\
D^{scan} &= \{[A \to \alpha \bullet a\beta, i, j], [a, j, j+1] \vdash [A \to \alpha a \bullet \beta, i, j+1\} \\
D^{compl} &= \{[A \to \alpha \bullet B\beta, i, j], [B \to \gamma\bullet, j, k] \vdash [A \to \alpha B \bullet \beta, i, k]\} \\
D &= D^{init} \cup D^{scan} \cup D^{compl} \\
\alpha, \beta, \gamma &\in V^* \\
a &\in V_T \\
A, B &\in V_N
\end{aligned}
$$

Die wesentliche Änderung gegenüber dem klassichen Earley-Algorithmus ist, daß die Prediction-Schritte wegfallen; stattdessen erlaubt die neue Menge D^{init} an jeder Stelle, eine neue Regel einzuführen. Gerade dieses Wegfallen des Prediction-Schrittes verwandelt die Analyse in eine Bottom-Up-Analyse, da die wesentliche Information von den Worten her zu immer größeren Strukturen zusammengesetzt wird, während die wesentliche Information im klassischen Algorithmus die Top-Down-Vorhersage der möglichen Regeln ist.

Left-Corner Earley-Parser

Der *Left-Corner* Earley-Parser ist eine leichte Modifikation der Bottom-Up-Variante, bei der überflüssige Chartitems der Form $[A \to \bullet\alpha, i, i]$, die nie expandiert werden können, vermieden werden sollen. Daher werden die gültigen Items so eingeschränkt, daß mindestens eine Konstituente einer Regel vorhanden sein muß, bevor das Item in die Chart eingetragen wird. Dies ist die linke Ecke, die der Variante ihren Namen gibt.

Definition 2.5.3 (Left-Corner Earley-Parser (LC-Earley))

$$
\begin{aligned}
\mathcal{I} &= \{[A \to A\alpha \bullet \beta, i, j] | A \to X\alpha\beta \in R \wedge 0 \leq i \leq j\} \\
D^{init} &= \{\vdash [A \to \bullet\alpha, j, j]\} \\
D^{LC(a)} &= \{[a, j-1, j] \vdash [B \to a \bullet \beta, j-1, j]\} \\
D^{LC(A)} &= \{[A \to \alpha\bullet, i, j] \vdash [B \to A \bullet \beta, i, j]\} \\
D^{scan} &= \{[A \to \alpha \bullet a\beta, i, j], [a, j, j+1] \vdash [A \to \alpha a \bullet \beta, i, j+1]\} \\
D^{compl} &= \{[A \to \alpha \bullet B\beta, i, j], [B \to \gamma\bullet, j, k] \vdash [A \to \alpha B \bullet \beta, i, k]\} \\
D &= D^{init} \cup D^{scan} \cup D^{compl} \cup D^{LC(a)} \cup D^{LC(A)} \\
\alpha, \beta, \gamma &\in V^* \\
a &\in V_T \\
A, B &\in V_N
\end{aligned}
$$

Hier werden tatsächlich alle Regeln Bottom-Up in die Chart eingetragen. Aus erkannten Konstituenten wird eine Voraussage für mögliche Regeln getroffen, die die erkannte Konstituente als erstes RHS-Symbol benutzen.

2.6. Exkurs: Features und Featurestrukturen

Ein Hilfsmittel, das in vielen Bereichen der Sprachverarbeitung zur Speicherung von Informationen verwendet wird, sind *Features* und *Featurestrukturen*. In diesem Abschnitt soll ein kurzer Überblick über Featurestrukturen und ihre Möglichkeiten gegeben werden.

2.6.1. Definition von Featurestrukturen

Definition 2.6.1 (Features) *Features, auch Attribute genannt, sind Elemente einer Menge $\mathcal{F}eat$. Sie werden mit $f, g, h, \ldots$ bezeichnet.*

Eine Featurestruktur hat als *Basisobjekt* ein Objekt X, das zur Identifikation benutzt wird. Die genaue Struktur des Basisobjektes ist für die Featurestruktur unwichtig, sie müsses lediglich eindeutig identifizierbar sein.

Definition 2.6.2 (Values) *Attributwerte oder Values sind Elemente einer Menge $\mathcal{V}al$. Für Values werden die Symbole $c, d, e, \ldots$ benutzt.*

Definition 2.6.3 (Featurepfad) *Featurepfade sind Elemente aus $\mathcal{F}eat^*$; sie werden als $\langle Xfgh\rangle$ notiert. Die Featurepfade werden durch die Symbole $\pi, \rho, \ldots$ dargestellt.*

Das Symbol μ kann für Featurepfade und Values stehen.

Definition 2.6.4 (Featurestruktur) *Eine Featurestruktur χ ist eine der folgenden Strukturen:*

- *eine partielle Funktion $\chi_X : \mathcal{F}eat^* \rightarrow \mathcal{V}al$,*
- *ein Constraintset $\chi(X) \subset VC \cup CC$,*

$$\begin{aligned} VC &= \{\langle X\pi\rangle \doteq c \mid \pi \in \mathcal{F}eat^* \wedge c \in \mathcal{V}al\}, \\ CC &= \{\langle X\pi\rangle \doteq \langle X\varrho\rangle \mid \pi, \varrho \in \mathcal{F}eat^*\} \end{aligned}$$

  oder

- *ein gerichteter, azyklischer Graph. Dieser Graph muß folgende Bedingungen erfüllen:*

  1. *es gibt eine Wurzel mit dem Bezeichner (Label) X,*
  2. *alle Kanten sind mit einem Label $f \in \mathcal{F}eat$ versehen,*
  3. *bei allen paarweise verschiedenen Kanten, die von einem Knoten ausgehen, sind die Label nicht identisch,*
  4. *Blätter können mit Labeln $c \in \mathcal{V}al$ ausgezeichnet werden,*
  5. *kein innerer[22] Knoten trägt ein Label.*

Eine einfache Featurestruktur ist in Abbildung 2.2 als Graph (*Featuregraph*) abgebildet.

Die Wurzel wird mit dem Basisobjekt bezeichnet, die Blätter sind entweder unbenannt oder durch Elemente aus $\mathcal{V}al$ gekennzeichent. An jeder Kante steht ein Wert $f \in \mathcal{F}eat$, die Knoten zwischen zwei Kanten sind ebenfalls anonym.

Als Constraintset hat diese Struktur die Form:

[22] Innere Knoten sind alle Knoten, die weder Blatt noch Wurzel sind.

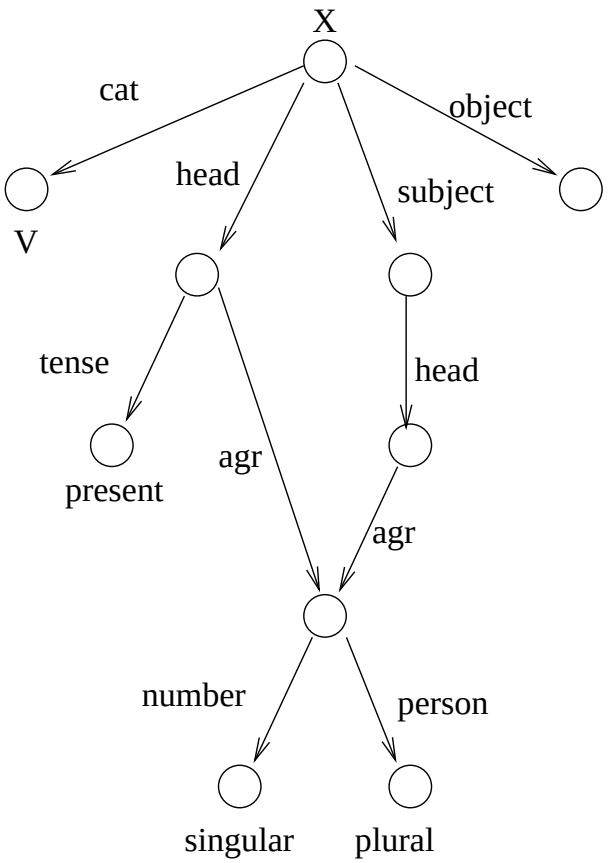


Abbildung 2.2.: Ein einfacher Featuregraph

$$
\begin{aligned}
\{\quad & \langle X \text{ CAT}\rangle \doteq V, \\
& \langle X \text{ HEAD TENSE}\rangle \doteq \text{present}, \\
& \langle X \text{ HEAD AGR attrnameNUMBER}\rangle \doteq \text{singular} \\
& \langle X \text{ HEAD AGR PERSON}\rangle \doteq \text{third} \\
& \langle X \text{ SUBJECT HEAD AGR person}\rangle \doteq \langle X \text{ HEAD AGR}\rangle \\
& \langle X \text{ OBJECT}\rangle \doteq [] \qquad\qquad \}
\end{aligned}
$$

Dabei steht auf der linken Seite des „$\doteq$“ für jedes Blatt ein Pfad, auf der rechten Seite der Attributwert. Bei anonymen Blättern wird dazu das Symbol „[ ]“ verwendet. Wenn ein Knoten durch mehrere Pfade erreichbar ist, wird jeder weitere Pfad durch ein Constraint repräsentiert, in dem auf der linken Gleichungsseite der Originalpfad und auf der rechten der alternative Pfad steht[23]. In dem Fall, daß zwei Attribute denselben Wert haben, also eine Pfadkoreferenz vorliegt, spricht man von *Token-Identity* oder Referenzidentität. Haben

[23] Dies wird auch als Pfadkoreferenz bezeichnet.

zwei Attribute den gleichen Wert aber unterschiedliche Token, d. h. es liegt keine Pfadkoreferenz vor, spricht man von *Type-Identity* oder Wertidentität.

In einer weiteren, häufig verwendeten Notation für Featurestrukturen hat die Struktur die Form

$$\begin{bmatrix} \text{CAT} & \text{V} \\ \text{HEAD} & \begin{bmatrix} \text{TENSE} & \text{present} \\ \text{AGR} & \boxed{1}\begin{bmatrix} \text{NUMBER} & \text{singular} \\ \text{PERSON} & \text{third} \end{bmatrix} \end{bmatrix} \\ \text{SUBJECT} & \begin{bmatrix} \text{HEAD} & \begin{bmatrix} \text{AGR} & \boxed{1} \end{bmatrix} \end{bmatrix} \\ \text{OBJECT} & [\,] \end{bmatrix}$$

Die umrahmten Zahlen entsprechen dabei den Pfadkoreferenzen bei Constraintsets, die gemeinsam genutzte Teilstrukturen definieren. Falls, wie im obigen Fall, verschachtelte Strukturen vorhanden sind, bei denen jede Teilstruktur nur ein Attribut enthält, können diese über die Notation

$$\left[\text{SUBJECT} \mid \text{HEAD} \mid \text{AGR} \quad \boxed{1}\right]$$

abgekürzt werden.

Im weiteren Verlauf dieser Arbeit werden die vorgestellten Darstellungen als äquivalent behandelt und je nach Situation ausgewählt[24] , da sie sich von Fall zu Fall unterschiedlich gut zur Veranschaulichung von Phänomenen eignen. Die Sichtweise auf Featurestrukturen als partielle Funktionen ist in diesem Fall allerdings gesondert zu behandeln, da über diesen Ansatz Pfadkoreferenzen nicht adäquat formalisiert werden können[25].

[24] Ein formaler Beweis der Äquivalenz zwischen Graphen und Constraints findet sich in (Sikkel, 1997). Dort werden auch Algorithmen zur Transformation der verschiedenen Darstellungsformen angegeben. Die Interpretation von Featurestrukturen als Funktionen wird in (Johnson, 1988) eingeführt.

[25] Wenn für zwei (partielle) Funktionen f und g gilt, daß $\text{dom}(f) = \text{dom}(g)$ und $f(x) = g(x)\ \forall x \in \text{dom}(f)$, dann gilt $f = g$. Daher kann man bei zwei Featurestrukturen in dieser Darstellung nicht zwischen Wert- und Referenzidentität unterscheiden.

2.6.2. Featurestrukturen als Verband

Die Featurestrukturen lassen sich in einen Verband (vgl. Definition A.4.1) einordnen, der genutzt werden kann, um Featurestrukturen über einen Unifikationsoperator „$\sqcup$“ zu einer neuen Featurestruktur zusammenzufügen. Dazu sind zunächst noch einige Definitionen notwendig:

Definition 2.6.5 (Abschluß) *Der Abschluß einer Featurestruktur $\chi(X)$, $closure(\chi(X))$, ist das kleinste Constraintset*[26]*, das folgende Bedingungen erfüllt:*

1. $\langle X\rangle\doteq[\,] \in closure(\chi(X))$;
2. $\langle X\pi\rangle\doteq\mu \in \chi(X) \rightarrow \langle X\pi\rangle\doteq\mu \in closure(\chi(X))$;
3. $\langle X\pi\rangle\doteq\langle X\pi'\rangle \in closure(\chi(X)) \wedge \langle X\pi\varrho\rangle\doteq\mu \in \chi(X)$
   $\rightarrow \langle X\pi'\varrho\rangle\doteq\mu \in closure(\chi(X))$;
4. $\langle X\pi\rangle\doteq\langle X\varrho\rangle \in closure(\chi(X)) \rightarrow \langle X\varrho\rangle\doteq\langle X\pi\rangle \in closure(\chi(X))$;
5. $\langle X\pi\varrho\rangle\doteq\mu \in closure(\chi(X)) \rightarrow \langle X\pi\rangle\doteq[\,] \in closure(\chi(X))$.

Dabei sorgt (3) für die Aufnahme der transitiven Hülle in den Abschluß. Die Forderung (4) sorgt dafür, daß die Symmetrie der Pfadkoreferenzen ebenfalls explizit im Abschluss ausgedrückt ist.

Definition 2.6.6 (Konsistenz) *Eine abgeschlossene Featurestruktur χ ist konsistent, wenn gilt:*

1. *χ ist endlich;*
2. $\langle X\pi\rangle\doteq c \in \chi(X) \wedge \langle X\pi\rangle\doteq d \in \chi(X) \rightarrow c = d$;
3. $\langle X\pi\rangle\doteq c \in \chi(X) \wedge \langle X\pi\varrho\rangle\doteq\mu \in \chi(X) \rightarrow \varrho = \varepsilon$;
4. $\langle X\pi\varrho\rangle\doteq\langle X\pi\rangle \notin \chi(X)\ \forall\pi \in \mathcal{F}eat^*, \varrho \neq \varepsilon$.

[26] Dieses Constraintset kann unendlich groß sein (vgl. Sikkel (1997)).

Eine nicht abgeschlossene Featurestruktur ist genau dann konsistent, wenn ihr Abschluß konsistent ist.

Die Menge aller konsistenten Featurestrukturen wird als $\mathcal{CCS}$[27] *bezeichnet.*

Definition 2.6.7 (Subsumption) *Die Subsumptionsrelation $\preceq$ ist definiert als:*

$$closure(\chi_1(X)) \subseteq closure(\chi_2(X)) \rightarrow \chi_1(X) \preceq \chi_2(X).$$

Definition 2.6.8 *Das Symbol $\top$ definiert die (inkonsistente) Featurestruktur, die von allen Featurestrukturen subsumiert wird*[28]*:*

$$\top = VC \cup CC.$$

Definition 2.6.9 (Feature-Verband) *Die konsistenten Featurestrukturen $\mathcal{CCS}$ zusammen mit der Relation $\preceq$, dem Element $\top$ und dem Element $\bot = [\,]$ definieren einen Verband von Featurestrukturen.*

Definition 2.6.10 (Unifikationsoperator) *Die obere Grenze für zwei Featurestrukturen wird durch $\chi_1(X) \sqcup \chi_2(X)$ angegeben. Der so definierte Operator heißt Unifikationsoperator. Ohne diese formale Betrachtung kann man es auch so ausdrücken, daß die Struktur $\chi_1(X) \sqcup \chi_2(X)$ die allgemeinste Featurestruktur ist, die die Informationen von $\chi_1(X)$ und $\chi_2(X)$ enthält.*

Der Operator $\sqcup$ kann genutzt werden, um aus zwei Featurestrukturen eine neue Featurestruktur mit der vereinten Information zu erzeugen.

2.6.3. Zusammengesetzte Featurestrukturen

Wenn man sich bei den so definierten Featurestrukturen nicht auf ein Basisobjekt beschränkt, kann man analog zu den einfachen Featurestrukturen zusammengesetzte Featurestrukturen mit den Basisobjekten X_i definieren.

[27] Eine Abkürzung für *Consistent Constraint Sets*.

[28] Die Struktur $\top$ ist keine konsistente Featurestruktur, da sie nicht endlich ist. In der Literatur findet sich oft das Symbol $\bot$ für diese Struktur; dies widerspricht dann jedoch seiner Bedeutung im Verband.

Definition 2.6.11 (Zusammengesetzte Featurestrukturen) *Eine zusammengesetzte Featurestruktur ist*

- *ein Constraintset* $\chi(X_1, \ldots, X_n) \subset VC \cup CC$,

$$\begin{aligned} VC &= \{\langle X_i\pi\rangle \doteq c \mid \pi \in \mathcal{F}eat^* \wedge c \in \mathcal{V}al\}, \\ CC &= \{\langle X_i\pi\rangle \doteq \langle X_j\varrho\rangle \mid \pi, \varrho \in \mathcal{F}eat^*\} \end{aligned}$$

  oder

- *ein gerichteter, azyklischer Graph. Dieser Graph muß folgende Bedingungen erfüllen:*

  1. *es gibt n Wurzeln mit den Labeln* $X_1, \ldots, X_n$,
  2. *alle Kanten sind mit einem Label* $f \in \mathcal{F}eat$ *versehen,*
  3. *bei allen paarweise verschiedenen Kanten sind die Label nicht identisch,*
  4. *Blätter können mit Labeln* $c \in \mathcal{V}al$ *ausgezeichnet werden,*
  5. *kein innerer Knoten trägt ein Label.*

Analog zum vorigen Abschnitt lassen sich auch hier Subsumption, ein Featureverband und die Unifikation definieren. Auf die formale Ausführung soll an dieser Stelle verzichtet werden; sie sind in (Sikkel, 1997) nachzulesen.

Der Ausdruck $\chi(X_1, \ldots, X_n) \mid X_i$ bezeichnet die Teilstruktur X_i der zusammengesetzten Featurestruktur.

2.6.4. Getypte Featurestrukturen

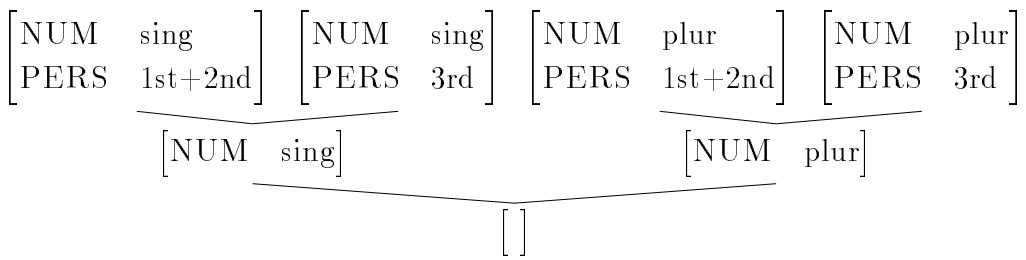
Man kann die so definierten Featurestrukturen noch um viele Eigenschaften erweitern. Eine der für Parser wesentlichen Erweiterungen sind die getypten Featurestrukturen. Die getypten Featurestrukturen sind ein Mittel, effizienter festzustellen, ob $\chi \sqcup \xi = \top$, d. h. daß es keine konsistente Featurestruktur gibt, die die Informationen von χ und ξ enthält. Die Typen bilden dabei wiederum ein hierarchisches System, das durch einen Verband $(\mathcal{T}ype, \preceq_{\mathcal{T}ype})$ dargestellt wird.

Jede Featurestruktur wird dabei als ein Tupel $(\chi, t), t \in \mathcal{T}ype$ betrachtet; diese kann man wiederum in einem Verband $(\mathcal{TF}, \preceq_{\mathcal{TF}})$ anordnen mit

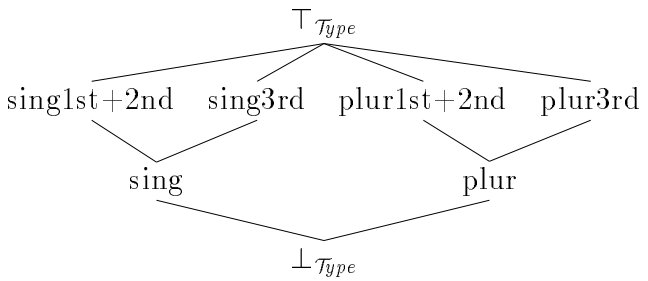
$$(\chi_1, t_1) \preceq_{\mathcal{TF}} (\chi_2, t_2) \iff t_1 \preceq_{\mathcal{T}ype} t_2 \wedge \chi_1 \preceq \chi_2.$$

Die Elemente $\top_{\mathcal{TF}}$ und $\bot_{\mathcal{TF}}$ sind offensichtlich $(\top_{\mathcal{T}ype}, \top)$ und $(\bot_{\mathcal{T}ype}, \bot)$. Eine Unifikation zweier Featurestrukturen (χ_1, t_1), (χ_2, t_2) ergibt nur dann nicht $\top_{\mathcal{TF}}$, falls es einen Typen $t = t_1 \sqcup_{\mathcal{T}ype} t_2 \neq \bot_{\mathcal{T}ype}$ gibt, und die beiden Featurestrukturen zusätzlich noch verträglich sind.

Man kann aus Featurestrukturen automatisiert Typen extrahieren, indem man Teilstrukturen statisch in Typen umwandelt. Damit ist ohne eine aufwendige Unifikation direkt anhand der Typen ersichtlich, ob die Featurestrukturen verträglich sind. Dies kann z. B. genutzt werden, um Person und Numerus in einem Typensystem auszudrücken:



Diese Teilstrukturen können in einen Verband überführt werden:



Jede Featurestruktur bekommt einen Typ zugeordnet. Das korrekte *Agreement* (vgl. Abschnitt 3.4.1) bzgl. Person und Numerus kann dann anhand des Typs entschieden werden, ohne daß die aufwendigere Unifikation über die Featurestrukturen durchgeführt werden muß. Der Universaltyp $\bot_{\mathcal{T}ype}$ ist in diesem System mit allen Typen verträglich. Wird er verwendet dann entspricht die Unifikation der Standardunifikation, da gilt $\bot_{\mathcal{T}ype} \sqcup_{\mathcal{T}ype} t = t \; \forall t \in \mathcal{T}ype$.

In dem oben gegebenen Beispiel gilt:

$$\text{sing3rd}\left[\cdots\right] \sqcup \text{plur3rd}\left[\cdots\right] = \top,$$

d. h. die Koreferenz zweier inkompatibler Typen schlägt immer fehl.

2.7. Unifikationsgrammatiken

Die bisher vorgestellten Parsingsysteme arbeiten nur auf kontextfreien Grammatiken, die ohne weitere Hilfsmittel nicht zur Beschreibung natürlicher Sprache geeignet sind. Bei vielen Parsingsystemen wird daher die kontextfreie Grammatik um Strukturen erweitert, die weitere Informationen über die Konstituenten tragen. Die in Abschnitt 2.6 eingeführten Featurestrukturen werden in Unifikationsgrammatiken[29] verwendet, um zusätzliche Informationen zu jeder Konstituente abzulegen. Die Basisobjekte entsprechen dabei den Konstituenten. Jede Konstituente einer Regel ist ein Basisobjekt mit zugehöriger Featurestruktur. So kann z. B. die Regel S $\rightarrow$ NP VP auf folgende Art erweitert werden:

$$\begin{aligned} \text{S} \rightarrow\ & \text{NP VP} \\ & \langle \text{VP SYN AGR} \rangle = \langle \text{NP SYN AGR} \rangle \\ & \langle \text{S SYN AGR} \rangle = \langle \text{VP SYN AGR} \rangle \end{aligned}$$

Diese Regel fordert neben der Konstituentenstruktur auch noch ein Agreement zwischen `NP` und `VP`. Die Informationen dieser Struktur werden dann zusätzlich in die Featurestruktur des `S`-Knotens aufgenommen.

Definition 2.7.1 (Unifikationsgrammatik) *Eine Unifikationsgrammatik ist eine Struktur*

$$< G, \Phi, \varphi_0, W, \mathcal{L}ex > .$$

G ist eine kontextfreie Grammatik (vgl. Abschnitt 2.1), Φ ist ein Verband aus zusammengesetzten Featurestrukturen (vgl. Abschnitt 2.6.9) und φ_0 eine Funktion, die jeder Regel der kontextfreien Grammatik $G =< V_N, V_T, S, R >$ eine Featurestruktur zuordnet, also

$$\varphi_0(A \rightarrow \gamma) = \varphi \in \Phi.$$

Die Menge W enthält alle Worte, die der Parser erkennen kann. Diesen wird durch die Funktion $\mathcal{L}ex$ jeweils eine Menge von Featurestrukturen zugeordnet.

[29] Zur Familie der Unifikationsgrammatiken gehören z. B. GPSG (vgl. Gazdar et al. (1985)), HPSG (vgl. Pollard (1985)) und LFG (vgl. Sells (1985)).

Jede der Featurestrukturen entspricht einer möglichen Lesart eines Wortes w. Ist die Lesart in einem Kontext klar, wird sie auch als $\varphi(w)$ notiert. Die Verbindung zwischen den terminalen Symbolen und den Featurestrukturen wird über ein dediziertes Feature CAT geschaffen, das als Wert ein Symbol $c \in \Sigma$ haben muß.

Um die Featurestrukturen in den vorgestellten Earley-Parser zu integrieren, müssen zu jedem Chartitem auch entsprechende Featurestrukturen gespeichert werden. Dies ergibt die Notation $[A \to \alpha \bullet \beta, \varphi(A \to \alpha \bullet \beta), i, j]$, mit $\varphi(A \to \alpha \bullet \beta)$ als die zum Item gehörende Featurestruktur. Um die Itemnotation übersichtlicher zu gestalten, wird das Item als $[A \to \alpha \bullet \beta, i, j]_\xi$ geschrieben, die Featurestruktur wird dann als $\varphi(\xi)$ referenziert. Die Featurestrukturen, die zu Regeln gehören, werden über den Ausdruck $\varphi_0(A \to \alpha\beta)$ angesprochen. Mit dem Symbol $\varphi(A_\xi)$ ist die Teilstruktur von $\varphi(\xi)$ gemeint, die zum Symbol A gehört, also eine Kurznotation zu $\varphi(\xi) \mid A$.

Definition 2.7.2 (Earley-Parser mit Unifikation)

$$
\begin{aligned}
\mathcal{I} &= \{[A \to \alpha \bullet \beta, i, j]_\xi | A \to \alpha\beta \in R \wedge 0 \leq i \leq j \\
&\qquad \wedge \varphi_0(A \to \alpha\beta) \preceq \varphi(\xi) \wedge \varphi(\xi) \neq \top\} \\
D^{init} &= \{\vdash [S \to \bullet\gamma, 0, 0]_\xi \\
&\qquad \mid \varphi(\xi) = \varphi_0(S \to \gamma)\} \\
D^{scan} &= \{[A \to \alpha \bullet a\beta, i, j]_\eta, [a, j, j+1]_\zeta \vdash [A \to \alpha a \bullet \beta, i, j+1]_\xi \\
&\qquad \mid \varphi(\xi) = \varphi(\eta) \sqcup \varphi(a_\zeta)\} \\
D^{compl} &= \{[A \to \alpha \bullet B\beta, i, j]_\eta, [B \to \gamma\bullet, j, k]_\zeta \vdash [A \to \alpha B \bullet \beta, i, k]_\xi \\
&\qquad \mid \varphi(\xi) = \varphi(\eta) \sqcup \varphi(B_\zeta)\} \\
D^{pred} &= \{[A \to \alpha \bullet B\beta, i, j]_\eta \vdash [B \to \bullet\gamma, j, j]_\xi \\
&\qquad \mid \varphi(\xi) = \varphi_0(B \to \gamma)\} \\
D &= D^{init} \cup D^{scan} \cup D^{compl} \cup D^{pred} \\
\alpha, \beta, \gamma &\in V^* \\
a &\in V_T \\
A, B &\in V_N
\end{aligned}
$$

Die Funktion $\mathcal{K}$, die aus dem Input die initialen Chartitems generiert, hat die Form

$$\mathcal{K}(a_1, \ldots, a_n) = \{[a, i-1, i]_\xi \mid a = a_i \wedge 0 < i \leq n \wedge \varphi(\xi) \in \mathcal{L}ex(a_i)\}.$$

Die Ableitungsregeln entsprechen, um die Unifikationsvorschrift erweitert, denen des kontextfreien Earley-Parsers. Beim *Prediction*-Schritt wird die zur Regel gehörende Grundstruktur an das Item gebunden. Beim *Scan* wird die Featurestruktur des terminalen Symbols mit der Featurestruktur der bearbeiteten Regel unifiziert, beim *Complete* die Featurestruktur des LHS-Symbols der komplettierten Regel mit der Featurestruktur der bearbeiteten Regel. Falls bei der Unifikation zweier Featurestrukturen das Element $\top$ entsteht, dann gibt es keine konsistente Featurestruktur, die eine Unifikation der Teilstrukturen darstellt. Damit gilt die Unifikation als gescheitert, und die Regel wird als nicht anwendbar betrachtet. Daher sind die Items mit $\varphi(\xi) \neq \top$ aus der Menge der erlaubten Items ausgeschlossen. Auf diese Art können unerwünschte Konstruktionen, z. B. Sätze mit inkorrektem Agreement, ausgeschlossen werden.

Während beim reinen kontextfreien Earley-Parser für einen Bereich des Inputs von i bis j und eine Regel immer nur ein Chartitem der Form $[A \rightarrow \gamma, i, j]$ nötig war, muß beim unifikationsbasierten Earley-Parser bei einem Item zwischen verschiedenen Featurestrukturen unterschieden werden. Daher können, im Gegensatz zum einfachen Parser, für eine Regel mehrere Items, die in dem Inputbereich i bis j gelten, vorhanden sein.

3. Der Parser

In diesem Kapitel wird der Parser, der einen wesentlichen Teil des im Rahmen dieser Arbeit entwickelten Systems darstellt, näher beschrieben. Dabei wird nicht intensiv auf die Implementation des Systems eingegangen, da dies allein durch dessen Größe den Rahmen dieser Arbeit sprengen würde. Jede Systemkomponente wird grob in ihrer Leistungsfähigkeit beschrieben. Den Rahmen für diese Beschreibung bildet eine kleine Beispielgrammatik.

Die Implementation entstand ursprünglich im Rahmen des Studienprojektes *Semf* (vgl. Bongardt et al. (1997)), in dem eine Komponente eines Sprachlehrsystems entwickelt wurde. Dabei diente als Grundlage ein Prototyp eines Systems von Dr. Hagen Langer. Im Rahmen des RECALL-Projektes, das sich ebenfalls mit Sprachlehrsystemen beschäftigt hat, wurde das System nach einigen Modifikationen ebenfalls eingesetzt. Die aktuelle Version stellt eine Fortentwicklung des im RECALL-Prototypen eingesetzten Parsers dar, die im Bereich des Lexikons und der funktionalen Ausdrücke (vgl. Abschnitt 3.5) erweitert wurde. Außerdem wurde die Möglichkeit automatisierter Grammatiktests geschaffen. Diese wird in Kapitel 5.3 näher erläutert.

3.1. Das Lexikon

Das Lexikon ist eine Komponente, die zwar nicht direkt zum Parser gehört, ohne die der Parser aber nicht arbeiten kann.

Die erste Aufgabe der lexikalischen Komponente ist die Zerlegung der Eingabe in eine Abfolge von Worten. Die Eingabekette ist ein ASCII-String[1], der an bestimmten Trennzeichen zerlegt wird. Die Zeichenfolgen zwischen zwei Trennzeichen werden dabei als Worte interpretiert. Als Trennzeichen gelten dabei Leerzeichen, Tabulatorzeichen, Zeilentrenner und das „'“ (Apostroph).

[1] Ein ASCII-String ist eine Abfolge von ASCII-Zeichen.

Das Lexikon übernimmt die Transformation der Eingabe $\mathcal{E} \in W^*$ in eine Abfolge von Strukturbeschreibungen

$$\mathcal{I} \in \{(k, \chi) | k \in \Sigma, \chi \in \mathcal{CCS}\}^*.$$

Da die Zuordnung von Worten zu den lexikalischen Kategorien nicht eindeutig sein muß, kann dabei an einer Position I_n innerhalb der Kette eine Menge mit mehr als einer Strukturbeschreibung für ein Wort stehen.

Während des Parsings wird jedes Wort der Eingabemenge in eine Menge von Tupeln übersetzt, die jeweils aus der lexikalischen Kategorie des Wortes und einer das Wort beschreibenden, konsistenten Featurestruktur bestehen. Wenn für ein Wort mehr als eine Beschreibung gefunden wird, muß der Parser dazu mehrere alternative Analysevarianten erzeugen. Da die Generierung dieser I_n zu dem Zeitpunkt erfolgt, an dem der Parser alle Eingaben $I_1, \ldots, I_{n-1}$ schon verarbeitet hat, können anhand der Regelstruktur einige Alternativen direkt eliminiert werden. Da alle Alternativen durch den Parser parallel verfolgt werden, kann an diesem Punkt entschieden werden, ob es Teilsequenzen $i_1 \cdots i_{n-1}, i_k \in I_k \; \forall i = 1, \ldots, n-1$ gibt, die mit einem $i_n \in I_n$ fortgesetzt werden können. Ist dies nicht der Fall, dann kann eine solche Alternative i_n ignoriert werden.

Die Wahl der Repräsentation einer Wortanalyse als Tupel aus Kategorie und Beschreibung beseitigt die Notwendigkeit, die Kategorie in einem speziell behandelten Attribut abzulegen.

Es folgen nun einige Beispiele zur Erläuterung des Lexikonformates, auf die genaue Syntax des Lexikons wird in Angang B eingegangen. In Anhang C.3 findet sich ein komplettes Beispiellexikon.

3.1.1. Einfache Lexikoneinträge

Ein einfacher Lexikoneintrag hat die Struktur

```
"I" PRO
[
 NUM=SING;
 PERS=FIRST;
 DETINFO = [
   DET_TYPE=FORBIDDEN;
 ];
].
```

Am Anfang des Eintrages steht das Wort in Anführungszeichen, gefolgt von der lexikalischen Kategorie und der zu dem Wort gehörenden Featurestruktur. Die Feature werden als Feature-Wertpaare, durch ein Gleichheitszeichen getrennt, aufgelistet. Das Feature DETINFO zeigt, daß ein Feature auch eine komplette Featurestruktur als Wert besitzen kann. Falls – wie in diesem Fall – nur wenige Werte in der Subfeaturestruktur vorhanden sind, kann dies alternativ auch anders formuliert werden. Die beiden folgenden Strukturen sind äquivalent:

```
[
 TYPE->SOLID=NO;
 TYPE->COUNTABLE=NO;
]
[
 TYPE = [
   SOLID=NO;
   COUNTABLE=NO;
 ];
].
```

Die obige Form verwendet Featurepfade (vgl. B.1.4) statt der Syntax für Unterstrukturen.

Aus technischen Gründen können die Featurestrukturen momentan keine Pfadkoreferenzen enthalten. Dies macht Lexikoneinträge mit Featurestrukturen wie

$$\begin{bmatrix} \text{CAT} & \text{V} \\ \text{HEAD} & \begin{bmatrix} \text{TENSE} & \text{present} \\ \text{AGR} & \boxed{1}\begin{bmatrix} \text{NUMBER} & \text{singular} \\ \text{PERSON} & \text{third} \end{bmatrix} \end{bmatrix} \\ \text{SUBJECT} & \begin{bmatrix} \text{HEAD} & \begin{bmatrix} \text{AGR} & \boxed{1} \end{bmatrix} \end{bmatrix} \\ \text{OBJECT} & [\,] \end{bmatrix}$$

unmöglich. Die Koreferenz zwischen der `NP` und der `VP`, wie sie hier im Lexikoneintrag kodiert ist, muß daher im vorliegenden System in der Grammatik kodiert werden. Es ist eine Frage des Stils ob die Information über solche Koreferenzen lexikalisiert – d. h. in das Lexikon transferiert – wird oder in den Regeln steht. In einigen Formalismen ist diese starke Lexikalisierung üblich, so z. B. in der *HPSG* oder der *LFG*. Eine Realisierung dieser Funktionalität

im aktuellen Entwicklungsstand des Systems würde aufgrund der bisherigen Implementation das Lexikon erheblich vergrößern, da jede Koreferenz explizit abgelegt wird. Werden in einem Lexikon daher viele Koreferezen verwendet, z.B. durch die Nutzung von Templates (vgl. Abschnitt 3.1.3), steigt der Aufwand für die Verarbeitung und der Speicherbedarf für das Lexikon drastisch. Aus diesem Grund ist diese Möglichkeit der Koreferenzen, zumal sie keinen Einfluß auf die Mächtigkeit[2] des Formalismus hat, nicht implementiert. Bei einer Reimplementierung der Lexikonrepräsentation könnte diese Funktionalität allerdings ohne Einbußen bei Verarbeitungseffizienz und Speicherplatzbedarf realisiert werden.

3.1.2. Defaults

Bei der Definition einer Grammatik können neben den Lexikoneinträgen auch sogenannte *Defaults* angegeben werden. Defaults sind Featurestrukturen, die direkt den verschiedenen Kategorien zugeordnet werden können. Jeder Lexikoneintrag, dem eine solche Kategorie zugeordnet wird, erbt den Defaulteintrag, d. h. der Lexikoneintrag erhält eine Kopie dieses Defaults als Featurestruktur. Sie erlauben daher eine Beschreibung der Eigenschaften einer Kategorie (vgl. Shieber (1986)).

```
#default V
[
 TYPE=[CAT=V;];
 SUBCAT=<>;
]

"hits" V
[
 SUBCAT=<TYPE=[CAT=N; SOLID=YES;]>;
]
```

Der Defaulteintrag besteht aus dem Schlüsselwort `#default`, der Kategorie und der Featurestruktur, die den Default darstellt. Die Lexikoneinträge, die

[2] Dies bezieht sich auf die gesamte Ausdrucksmächtigkeit des Systems. In spezifischen Fällen kann eine Realisierung eines Lexikons bzw. einer Grammatik erheblich aufwendiger werden, aber sie ist generell machbar. Auf einen formalen Nachweis der Äquivalenz in der Ausdrucksmächtigkeit soll an dieser Stelle, obwohl er durchaus nicht trivial ist, verzichtet werden. Für eine zukünftige Version des Systems ist zudem eine Aufhebung dieser Beschränkung vorgesehen.

die Defaults nutzen, unterscheiden sich nicht von den einfachen Lexikoneinträgen, die alleinige Anwesenheit eines Defaults modifiziert die nachfolgenden[3] Einträge. Wie hier in dem Beispiel können in einem Lexikoneintrag, der einen Default erbt, noch zusätzliche Strukturen definiert oder einzelne Features, die im Default definiert sind, durch neue Werte überschrieben werden.

3.1.3. Templates

Ein weiteres Hilfsmittel zur effizienten Lexikonerstellung sind *Templates*, d. h. Benamungen von Teilfeaturestrukturen, die über diesen Namen in eine bestehende Struktur eingefügt werden können. Dies ist einerseits ein Hilfsmittel, um Tipparbeit zu ersparen, andererseits können darüber auch generalisierte Eigenschaften mehrerer Worte ausgedrückt werden. Das Template

```
#template sing3rd
[
 NUM=SING;
 PERS=THIRD;
]

"a" DET
[
#use sing3rd;
 TYPE=[INITIAL=CONSONANT;];
]
```

stellt z. B. die Generalisierung der Eigenschaft dar, ein Verb in der dritten Person singular zu sein (vgl. Shieber (1986)). Bei dem Verb *a* wird das Template über die `#use`-Anweisung eingebunden.

Defaults und Templates können neben der Interpretation als Featurestruktur, die in eine Gesamtstruktur eingefügt wird, auch zur Generierung von Typen, wie sie in Abschnitt 2.6.4 beschrieben werden, benutzt werden. Ein formaler Ansatz in dieser Richtung findet sich in (Smolka, 1988).

[3] „Nachfolgend“ bezieht sich hier auf die Position in der Grammatikdatei. Aus technischen Gründen wirkt ein Defaulteintrag immer nur auf die nachfolgenden Lexikoneinträge.

3.1.4. Explizite Wertlöschung

In einigen Fällen soll ein vorgegebenes Attribut-Wertpaar eines Defaults gelöscht werden, d. h. jede Definition eines Wertes für dieses Attribut ist zu entfernen. Dies wird durch die Koreferenz

```
"the" DET
[
#use sing3rd;
 NUM=undefined;
 TYPE=[INITIAL={CONSONANT, VOWEL};];
]
```

erreicht. Damit ist für das Attribut NUM kein Wert mehr gespeichert, d. h. es gibt kein Feature-Wertpaar mehr mit dem Attribut NUM. Dies kann in einigen Fällen sinnvoll sein, im allgemeinen zeugt es aber eher von einem Fehler in den Vererbungsmechanismen. Dieser Effekt könnte auch durch die Anweisung

```
NUM={SING,PLUR}
```

erreicht werden, doch bei einigen Featuretypen, z. B. bei den Subfeaturestrukturen, ist diese Vorgehensweise nicht möglich.

3.1.5. Morphologische Fähigkeiten des Lexikons

Bei der Erstellung eines Lexikons gibt es zwei grundsätzliche Herangehensweisen:

- Man implementiert ein Vollformenlexikon, bei dem jede Flexionsform eines Wortes als eigenständiger Eintrag im Lexikon vorhanden sein muß.
- Man schafft eine morphologische Komponente im Lexikon, die aus einem Grundeintrag verschiedene Flexionsformen ableitet.

Das vorgestellte System wählt den zweiten Weg, um eine möglichst effiziente Erstellung des Lexikons zu ermöglichen[4]. Das Grundprinzip des Lexikons folgt

[4] Man kann natürlich die morphologische Komponente in einem Compiler realisieren, der für das lauffähige System alle Vollformen generiert. Dies beschleunigt zwar den Zugriff, erhöht aber auch die Lexikongröße.

der Trennung eines Wortes in einen bedeutungstragenden Stamm und einen die Flexionsinformation tragenden Suffix[5]. Dies ist natürlich nur eine relativ grobe morphologische Aufteilung, die viele Aspekte unberücksichtigt lässt. So werden im Moment z. B. keine Prefixe behandelt, die eine weitere Strukturierung des Lexikons erlauben. Eine Implementierung eines detaillierten Regelsystems zur Lexikongenerierung findet sich in (Ritchie et al., 1992).

Die Featurestruktur für den Stamm enthält im Wesentlichen eine Repräsentation der Semantik, während die Featurestruktur des Suffix die Repräsentation der Flexionsinformation enthält. Wird ein Wort im Lexikon gesucht, dann erzeugt die Lexikonkomponente, falls möglich, die zurückgegebene Featurestruktur durch die Unifikation der beiden Teilstrukturen. Scheitert die Unifikation, dann wird diese Kombination aus Stamm und Suffix ignoriert. Dies schafft die Möglichkeit, einfache Flexionsmuster im Lexikon geeignet darzustellen. Falls ein Wort unregelmässig flektiert wird, können – wie bei Vollformenlexika – alle flektierten Wortformen einzeln im Lexikon aufgenommen werden.

Ein Problem bei Sprachlehrsystemen, für die der Parser ursprünglich entwickelt wurde, ist, daß ein Wort auch dann erkannt werden sollte, wenn es inkorrekt flektiert wird. Dies tritt durch eine Übergeneralisierung des Schülers bei der Flexionsbildung auf.

Dabei soll das Wort nicht nur erkannt, sondern auch die Information über diese falsche Flexion geliefert werden. Dies wird über spezielle *Korrekturregeln* erreicht, die einzelne Flexionsmuster als inkorrekt markieren. Diese Korrekturregeln enthalten eine Featurestruktur, die die Fehlerinformation enthält. Der Algorithmus zur Erkennung eines Wortes arbeitet wie folgt:

1. Wähle alle Vollformen, die mit dem gesuchten Wort übereinstimmen. Die Featurestrukturen werden aus dem Eintrag extrahiert.

2. Wähle alle Korrekturregeln, deren Korrektureintrag mit dem gesuchten Wort übereinstimmt. Die Featurestrukturen werden aus dem Stamm, zu dem der Korrektureintrag gehört, und dem Suffix, der bei der Korrekturregel eingetragen ist, durch Unifikation gebildet.

[5] Die Aufteilung in einen bedeutungstragenden Stamm und einen flexionsinformation tragenden Suffix ist willkürlich und nicht strikt durchzuhalten. Die einfache Beobachtung, daß der Suffix vor allem die Flexionsinformation trägt führte lediglich zu der aktuellen Implementation. Die Featurestrukturen, die an Stamm und Suffix gebunden werden, sind natürlich frei wählbar, daher könnte im Suffix auch durchaus semantische Information abgelegt werden.

3. (a) Wähle alle Stammeinträge, die den Anfang des gesuchten Wortes bilden.

   (b) Wähle alle Suffixe, die zusammen mit dem Stamm das gesuchte Wort bilden und die gleiche Kategorie besitzen.

      i. Falls die Featurestrukturen aus Stamm und Suffix nicht unifizierbar sind, ignoriere diese Kombination aus Stamm und Suffix.

      ii. Falls es zu diesem Wort und diesem Suffix eine Korrekturregel gibt, gebe die Unifikation aller drei Featurestrukturen zurück. Diese enthält daher zusätzlich zu Stamm- und Flexionsinformation auch die Fehlerinformation im Korrektureintrag.

      iii. Die Rückgabestruktur wird aus der Unifikation der Stamm- und Suffixfeaturestrukturen gebildet.

Gegeben sei ein Lexikon mit folgenden Einträgen:

```
"~":foo N                              /* Suffix, Eintrag 1 */
[
  NUM=SING;
 DETINFO->DET_TYPE=REQUIRED;
]

"~s":nomplur N                         /* Suffix, Eintrag 2 */
[
 NUM=PLUR;
 TYPE->COUNTABLE=YES;
]
"~":foo N                              /* Suffix, Eintrag 3 */
[
 NUM=PLUR;
 TYPE->COUNTABLE=NO;
]

"astronaut~" N                         /* Stamm, Eintrag 4 */
[
 DETINFO=[INITIAL=VOWEL;];
]

"army~" N                              /* Stamm, Eintrag 5 */
[
]
```

```
"army~" / nomplur "armies" N          /* Korrektur, Eintrag 6 */
[
  ERROR=[ AKTERRS=<"irregular form, correct is armies">; ];
]
"water~" N                            /* Stamm, Eintrag 7 */
[
 TYPE->SOLID=NO;
 TYPE->COUNTABLE=NO;
]
```

Die genaue Syntax der Lexikoneinträge ist in B.1.9 näher Erläutert.

Mit diesen Einträgen liefert das Lexikon folgende Featurestrukturen:

Wort	Einträge	Zusammengesetzte Featurestruktur
astronaut	4,1	Stamm (4) und Endung (1)
astronauts	4,2,(3)	Stamm (4) und Endung (2)
water	7,(2),3	Stamm (4) und Endung (3)
army	5,1	Stamm (5) und Endung (1)
armies	6	Korrigierter Stamm (5) und Korrigierte Endung (2)
armys	5,2,6,(3)	Stamm (5), Endung (2) und Korrektureintrag (6)

In der ersten Spalte stehen dabei die Worte, in der zweiten die Lexikoneinträge, die das System vergleicht – eingeklammert wenn sie aufgrund der Featurestruktur nicht verwendet werden – und in der dritten Spalte die miteinander unifizierten Featurestrukturen, die vom Lexikon zurückgegeben werden.

Bei der Suche nach der Pluralendung werden immer die Suffixe 2 und 3 verwendet, und je nach der Markierung COUNTABLE kann entweder der Eintrag 2 (bei *astronaut*) oder der Eintrag 3 (bei *water*) verwendet werden. So können Suffixe systematisch auf Wortklassen, die sich nicht durch die lexikalische Kategorie unterscheiden, eingeschränkt werden.

Beim Wort *armies* findet das Lexikon den entsprechenden Korrektureintrag. Daher sucht es sich die passende Featurestruktur zu dem Wort aus dem Stammeintrag, der zu dem Korrektureintrag gehört, und dem zugehörigen Suffix. Dies vermeidet Redundanzen in den Featurestrukturen bei solchen zusammenhängenden Einträgen. Bei der Suche nach dem inkorrekten Wort **armys* findet es korrekt den Stamm und den Suffix, aber durch den Korrektureintrag 6 ist klar, daß diese Kombination aus Stamm `army` (5) und Suffix mit dem Bezeichner `nomplur` (2) nicht erlaubt ist. Die zurückgelieferte Featurestruktur besteht aus der Stamm- und der Suffixinformation sowie der Fehlerinformation im Korrektureintrag. So kann das durch ein übergeneralisiertes Flektionsmuster entstandene Wort erkannt und dem System entsprechend gemeldet werden.

Dies erlaubt z. B. einem Sprachlehrsystem eine adäquate Reaktion auf diesen Fehler. Dieses System der miteinander verbundenen Einträge erlaubt es, die Zusammenhänge ohne redundante Information im Lexikon auszudrücken.

3.1.6. Mehrworteinträge im Lexikon

Um die Grammatik von Regeln zu entlasten, die Wortgruppen wie *on top of* im Englischen analysieren, kann das Lexikon des aktuellen Systems Lexikoneinträge enthalten, die mehr als ein Wort umfassen. Bei einer Wortanalyse wird dem Lexikon immer der komplette Input übergeben, damit im Lexikon entschieden werden kann, wieviele Worte dieser Eintrag umfaßt. Dabei kann das letzte Wort eines Mehrworteintrages wiederum ein Stammeintrag sein, der mit einem Suffix kombiniert wird. Bei Suffixen sind allerdings keine Mehrworteinträge erlaubt, d.h. ein Suffix darf nicht aus mehr als einem Wort bestehen.

3.2. Der Parserkern

Der Parserkern ist die Implementation eines unifikationsbasierten Earley-Parsers, der auf einer kontextfreien Grammatik operiert, in der Tilgungsregeln erlaubt sind. Eine kleine Beispielgrammatik ist in Anhang C.4 angegeben. Der Grammatikformalismus ist angelehnt an den Formalismus *PATR-II* (vgl. Shieber et al. (1983)).

Der Parserkern arbeitet mit Featurestrukturen, bei denen die Features getypt sind, d.h. es gibt unterschiedliche Featuretypen, die unterschiedliche Wertemengen haben. Dies macht einerseits eine effiziente Speicherung der Featurestrukturen möglich, andererseits vermindert dies unentdeckte Tippfehler in der Grammatik, da die vorgegebenen Featurewerte mit den Vorgaben verglichen werden. Es wird zwischen atomaren und komplexen Features unterschieden, wodurch komplexe Features kann niemals atomare und atomare Features niemals komplexe Werte annehmen können. Die atomaren Features untergliedern sich noch in

Standardfeatures , deren Wertebereich $\mathrm{im}(F)$ bei der Deklaration festgelegt wird. Wenn ein Standardfeature mit einem Wert belegt wird, kann ein einzelner Wert $f \in \mathrm{im}(F)$ oder eine Disjunktion von Werten, d. h. eine Menge $\{f_1, \ldots, f_n\} \subset \mathrm{im}(F)$ verwendet werden. Die Disjunktion entspricht mehreren Alternativen für ein Feature F, die sich jedoch gegenseitig ausschließen. Die Unifikation zweier Featurewerte entspricht der Schnittmengenbildung über die disjunktiven Wertemengen,

d. h. $f_1 \doteq f_2 \Rightarrow f1 := f_1 \cap f_2, f2 := f1$. Ist $f_1 \cap f_2 = \emptyset$, dann ist die Unifikation gescheitert.

Stringfeatures , die sich im Wesentlichen durch den vordefinierten Wertebereich von den Standardfeatures unterscheiden. Der Wertebereich umfaßt alle auf dem System darstellbaren[6] Zeichenketten.

Booleanfeatures mit einem vorgegebenen Wertebereich ({TRUE, FALSE}). Die Unifikation entspricht der Standardunifikation.

Zähler , deren Wertebereich alle auf dem System darstellbaren Integerzahlen umfaßt. Es sind jedoch keine Disjunktionen von Werten erlaubt; Zähler haben immer nur einen einzigen Wert. Die Unifikation zweier Features ist ein Test auf Wertidentität der Zahlen. Sie schlägt fehl, wenn die Zahlen unterschiedlich sind.

Pfadfeatures , die einen Pfad in einer Featurestruktur zu speichern können. Dieser Pfad kann z.B. genutzt werden um die Lage eines Zielpfades zu beschreiben (vgl. B.1.2). Die Unifikation zweier Pfadfeatures gelingt nur, wenn die Pfade komplett identisch sind.

Neben einem einzigen atomaren Wert können atomare Featuretypen auch Disjunktionen als Wert besitzen[7], die eine kompakte Kodierung erlauben, falls bei einer Wortform mehrere syntaktische Analysen möglich sind. So kann der Artikel *the*, der im Singular und im Plural benutzt weren kann, durch eine einzige Featurestruktur repräsentiert werden (vgl. Abschnitt 3.6.2).

Neben diesen atomaren Featuretype gibt es als komplexe Featuretypen

Featurelisten , deren Unifikation der Mengenvereinigung entspricht:

$$f_1 \doteq f_2 \Rightarrow f_1 := f_1 \cup f_2, f_2 := f_1.$$

Die Unifikation kann daher nicht fehlschlagen[8]. Diese Features dienen zur Darstellung von Attributen, die bei der Gruppierung von Konstituenten kumuliert werden. Diese Aufgabe kann allerdings auch durch ein Listenfeature gelöst werden.

[6] Darstellbar heißt in diesem Zuammenhang, daß der String auf dem Rechner speicherbar ist, er muß nicht notwendigerweise eine Repräsentation auf dem Bildschirm haben

[7] Die Ausnahme sind hier Zähler und Pfadfeatures.

[8] Es kann aber ein Systemfehler entstehen, wenn die Listen zu lang werden. In diesem Fall wird das Parsing abgebrochen.

Stringlisten , die das Äquivalent der Featurelisten für Strings darstellen.

Listen , die aus endlichen Featuresequenzen bestehen, d. h. die eine endliche Anzahl beliebiger Attribut-Wert-Paare aufnehmen können. Für diese Listen sind Funktionen definiert, die die üblichen Listenoperationen (auf Elemente zugreifen, Teillisten extrahieren, oder Elemente anhängen) implementieren.

Subfeature werden zur Strukturierung der Featurestrukturen genutzt, da ihr Wertebereich die Menge der Featurestrukturen umfaßt. Da diese eingebetteten Featurestrukturen wiederum ein (oder mehrere) Subfeature mit weiter eingebetteten Strukturen enthalten können, erlaubt dies beliebig tief verschachtelte Featurestrukturen.

Eine Grammatikregel besteht aus einem Phrasenstrukturskelett und damit verbundenen Featurestrukturen, in denen Koreferenzen zwischen den einzelnen Teilstrukturen gefordert werden können. Die genaue Form der Grammatikregeln wird in Anhang B.1.8 erläutert.

3.3. Modularisierung von Grammatik und Lexikon

Da die Entwicklung einer komplexen Grammatik eine sehr aufwendige Aufgabe ist, wird sie oft von mehreren Personen gleichzeitig bearbeitet. Daher ist die Möglichkeit der parallelen Arbeit an einer einzigen Grammatik eine Anforderung, die ein System unbedingt erfüllen muß. Der erste Schritt in diese Richtung ist die Möglichkeit, eine Grammatikbeschreibung auf mehrere Teildateien zu verteilen. Dies erlaubt zwar ein gleichzeitiges Arbeiten an der Grammatik, d. h. die Quellen können gleichzeitig von verschiedenen Personen editiert werden, die Schnittstellen der Module sind aber noch nicht klar definiert.

Ein wesentlicher Bestandteil der Modularisierung einer Grammatik ist die Definition der Schnittstellen. Dies umfaßt die Festlegung aller Namen und Inhalte der benutzten Features, die Deklaration aller lexikalischen Kategorien sowie aller nichtterminalen Kategorien, die in allen Modulen verwendet werden. Diese Information bildet die Schnittstelle der einzelnen Grammatikmodule untereinander. Sie muß zum einen dokumentiert und zum anderen im Grammatikformalismus ausgedrückt werden. Diese Definitionen bilden die Grundlage für das Zusammenspiel der einzelnen Grammatikdateien. Eine Beispieldatei ist in

Anhang C.1 angegeben. Die Schnittstellendefinition wird in jedes Grammatikmodul eingebunden, um die Konsistenz der Definitionen über alle Module hinweg zu sichern. Diese Datei darf daher nur dann geändert werden, wenn alle Entwickler auf die Änderung oder Erweiterung der ihrer Module zugrundeliegenden Definitionen vorbereitet sind. Die in dieser Datei deklarierten Symbole sind die einzigen Symbole, die in mehr als einem Modul bekannt sind, alle anderen Deklarationen wirken sich nur auf die Dateien aus, in denen sie auftauchen. Die globalen – also in allen Modulen verfügbaren – Kategorien werden durch die Anweisungen `#vn-global` für alle nichtterminalen Kategorien und `#vt` für alle lexikalischen Kategorien festgelegt.

In den einzelnen Modulen können nun nach Belieben neue Features und Kategorien deklariert werden, die zur Erledigung der Aufgabe benötigt werden. So kann jeder Entwickler unabhägig und ohne weitere Absprachen – von Schnittstellenmodifikationen abgesehen – arbeiten. So wird z. B. in der Datei `testgram1-VP.y` im Anhang C.4.1 ein lokales Symbol `VBAR` benutzt, das in keinem der anderen Module benötigt wird. Diese Definition gilt, da sie lokal ist, in keinem anderen Modul. Wenn ein anderes Modul ebenfalls eine solche lokale Kategorie einführt, dann hat dies keine Auswirkungen auf die Definition im Modul `VP`. Die einzelnen Module werden dann vom Parser zusammengefügt und bilden die Grammatik und das Lexikon, die zum Parsen verwendet werden. Die Schnittstellen sollten möglichst universell gewählt werden, damit ein Modul auch in einem späteren Kontext wiederverwendet werden kann. Ein einmal geschriebenes Modul zur Bearbeitung von `NP`s kann so in jedes Projekt integriert werden, ohne es neu erstellen zu müssen. Eine Grundvoraussetzung für ein solches Vorgehen ist natürlich, daß die verwendeten Featurestrukturen miteinander verträglich sind.

Die Beispielgrammatik soll durch die Verwendung von drei Grammatikmodulen und einem Lexikonmodul das Zusammenspiel mehrerer Dateien verdeutlichen. Die Aufteilung der Module hängt natürlich von der gegebenen Aufgabenteilung ab; in diesem Beispiel ist sie eher willkürlich gewählt.

3.4. Parsing fehlerhafter Eingaben

Da das vorgestellte System einem Sprachlehrsystem entstammt, sind einige Erweiterungen gegenüber einem normalen Parsersystem integriert, die teilweise auf den Ideen aus (Schwind, 1994) basieren. Schwind geht in diesem Werk davon aus, daß die Ungrammatikalität einer Eingabe sich im Scheitern einer Unifikation ausdrückt. Ein typischer Fehler in diesem Kontext ist ein Agreementfehler, wie er bei **the child play* vorhanden ist. In diesem Beispiel sind die

Agreementconstraints, d. h. die Forderung auf gleiche Person, gleichen Numerus und gleichen Kasus, zwischen `NP` und `VP` verletzt, da die `NP` in der dritten Person, die `VP` aber in der ersten oder zweiten Person steht.

In (Schwind, 1994) wird der Ansatz verfolgt, diesen Fehler zu korrigieren, damit das Parsing mit dieser korrigierten Eingabe gelingt. Dies erfordert natürlich einen entsprechenden Mechanismus, der aufgrund der verletzten Constraints eine Alternative für die Eingabe erzeugt. Falls die Sätze komplexer werden, kann diese Korrektur allerdings an verschiedenen Satzpositionen erfolgen. Die korrekte Modifikationsposition hängt dabei von vielen Bedingungen, z. B. dem Diskurskontext, ab.

Ein anderer Ansatz findet sich in (Langer, 1990), bei dem ebenfalls eine Korrektur der Eingabe versucht wird. Bei diesem Ansatz wird jedoch nicht wie bei Schwind versucht einzelne Constraints zu lockern, sondern es werden zunächst alle Constraints mit Variablen als Wertet statt der konstanten Werte verwendet. Nachdem die Unifikation durchgefuhrt wurde ist durch die durchgeführte Unifikation klar, welche Featurewerte miteinander unifiziert wurden. Danach wird nach passenden lexikalischen Einträgen gesucht, die die geforderten Agreementconstraints erfüllen können. So kann für die einzelnen Lexeme ein korrekter Ersatz ausgewählt werden. Der in (Langer, 1990) vorgestellte Ansatz behandelt zudem noch die Analyse von Wortwiederholungen und expliziten Korrekturen, wie sie z. B. bei gesprochener Sprache immer wieder auftreten. Diese Ideen wurden aber in dem vorgestellten System nicht weiter behandelt.

3.4.1. Ein unifikationsbasierter Ansatz

Aud diesen Ideen basiert einer der im vorgestellten System implementierten Ansätze. Dabei wird die Information über einen erkannten Fehler im Satz abgespeichert, während die Reaktion auf den Fehler, z. B. die Korrektur des Fehlers, einer Komponente mit mehr Sprach- und Kontextwissen überlassen wird. Im *Semf*-System (vgl. Bongardt et al. (1997)) wurde die Reaktion durch eine Komponente gesteuert, die durch Wissen über ein zu beschreibendes Szenario einige Fehlerhypothesen aufgrund semantischer Analysen ausschließen konnte. Auch im RECALL-System wird die Fehlerauswertung von einer Komponente, die über ein Lernermodell, d. h. eine Modellierung des momentanen Sprachwissens des Lernenden, und Kontextinformation aus der Übung verfügt, getroffen.

Wenn eine Unifikation scheitert, dann werden die größten Teilmengen der Constraints gebildet, die noch erfüllt werden können. Die Menge $\mathcal{A}$ ist die größte Menge für die gilt:

$$\forall \chi, \xi \in \mathcal{A} : (\chi \neq \xi \rightarrow (\chi \not\sqsubseteq \xi \wedge \xi \not\sqsubseteq \chi)$$

Diese Menge enthält in der Regel mehrere Elemente, da z. B. bei einem Agreementfehler das falsche Agreement in jeder beteiligten Konstituente vermutet werden kann. Für jede gescheiterte Pfadkoreferenz einer solchen Menge wird eine Fehlerinformation zur aktuellen Featurestruktur hinzugefügt. Diese Informationen werden in einem speziellen Feature, AKTERR, gesammelt. Dabei entsteht für jede Menge aus $\mathcal{A}$ ein Chartitem für diese Regel, d. h. die Anzahl der Analysen für diese Konstituente ist die Anzahl der Elemente in $\mathcal{A}$.

Jede Regel gibt neben der Information über die Fehler in dieser Regel auch die Information über die Fehler in den anderen Regeln weiter. Dies geschieht über eine rekursive Liste mit dem Namen PREVERRS, in der für jede Regelkonstituente die *Fehlerstrukturen* der Konstituente abgelegt ist. Bei einer Regel mit drei Konstituenten bedeutet dies:

$$\text{PREVERRS} =< \$1\text{:ERROR}, \$2\text{:ERROR}, \$3\text{:ERROR} >;$$

Damit eine solche Pfadkoreferenz nicht zu jeder Regel hinzugefügt werden muß, gibt es spezielle Templates mit den Namen `init1`, `init2`, ... Diese Templates können Pfadkoreferenzen enthalten, die bei Regeln mit 1, 2, ... Konstituenten eingefügt werden. In diesen Templates können entsprechende Koreferenzen definiert werden, die die Information weiterleiten.

In speziellen Strukturen – definiert durch die Anweisung `#relax` – wird eine Featurestruktur abgelegt, die einen speziellen Fehlertyp beschreibt. Koreferenzen, deren Rücknahme einen solchen Fehler beschreibt, werden mit dem Namen der Struktur, die dem Fehlertyp entspricht, markiert. Über diesen Mechanismus gelangt die Fehlerinformation bis zur Featurestruktur des Satzes und kann ausgewertet werden.

```
#relax NUMERUS ERROR -> AKTERRS
[
 HEAD: [
      INFO=strcat("wrong numerus ",
                  ERRCAT:NUM,
                  " in cat:",
                  getCat[ERRCAT:](),
                  " needed ",
                  HEAD:NUM);
 ]
```

```
]

S : NP VP
#assert(NUMERUS 1|NUMERUS 2)
[
 S:[
   NUM = NP:NUM #opt NUMERUS 1;
   NUM = VP:NUM #opt NUMERUS 2;
 ]
]
```

Mit diesen beiden Anweisungen wird eine Fehlerstruktur erzeugt, die die Information über das fehlerhafte Agreement in einer Liste an der Position `ERROR->AKTERRS` einfügt. In dieser speziellen Anwendung kann mit `HEAD` auf den Kopf der Regel, in dem der Fehler auftritt, und mit `ERRCAT` auf die Kategorie, die den Fehler verursacht, zugegriffen werden. In der Grammatikregel werden dann die Koreferenzen, die abgeschwächt werden können, mit dem Schlüsselwort `#opt` versehen. Nach dem `#opt` folgen die Bezeichnung der Fehlerstruktur sowie eine Nummerierung. Diese Nummerierung erlaubt die Identifikation einer abschwächbaren Koreferenz in den `#assert`-Anweisungen. Diese Anweisungen steuern die Koreferenzrücknahme, indem mindestens eine der Koreferenzen, die in der `#assert`-Anweisung genannt wird, bei der Unifikation tatsächlich gelingen muß. Im obigen Beispiel wird dadurch verhindert, daß beide Koreferenzen gleichzeitig zurückgenommen werden.

3.4.2. Ein regelbasierter Ansatz

Neben dem unifikationsbasierten Ansatz zur Erkennung fehlerhaften Inputs gibt es noch regelbasierte Ansätze. In diesen regelbasierten Ansätzen werden *Fehlergrammatiken* erstellt, d. h. Grammatiken, die Regeln enthalten, die speziell auf einen Fehler abgestimmt sind. Dabei müssen natürlich alle Fehler komplett antizipiert werden, d. h. jeder Fehler, mit dem das System umgehen kann, wird als eigene Regel formuliert. Dieses Vorgehen wird u.a. in (Krüger-Thielmann, 1992) vorgeschlagen, die sich vor allem mit speziellen Fehlern bei deutschen Schülern in der Fremdsprache Französisch beschäftigt hat. Dies liefert – neben dem Nachteil, auf spezielle Fehler beschränkt zu sein – den Vorteil, eine sehr detaillierte Analyse zu ermöglichen.

Diese Vorgehensweise ist vor allem bei falschen Wortstellungen sinnvoll, denn zum einen kann man sich hier auf die typischen Wortstellungsfehler konzentrieren und zum anderen kann bei Wortstellungsfehlern die Koreferenzrücknahme

nicht funktionieren, da schon das Phrasenstrukturskelett – es handelt sich ja z.B. um vertauschte Worte – nicht stimmt. Die *Fehlerregeln*, also die Regeln, die eine fehlerhafte Wortstellung berücksichtigen, werden neben den Regeln mit der korrekten Wortstellung beim Parsing verwendet. Außer bei Wortstellungsfehlern sind regelbasierte Ansätze eine Alternative zur Koreferenzrücknahme, falls sich ein Fehler nicht durch das Scheitern einer Unifikation ausdrückt. Dies kann z. B. beim gleichzeitigen Scheitern mehrerer Unifikationen auftreten; bei denen die einzelnen Unifikationen nicht mit Fehlerklassen in Verbindung gebracht werden können, sondern zusammen einen Fehlertyp darstellen. In diesem Fall kann eine ganze Fehlerregel, deren Phrasenstrukturskelett mit dem der primären Regel übereinstimmt, als Ersatzregel deklariert werden. Falls die Unifikation bei der Primärregel scheitert wird die Ersatzregel, die eine völlig andere Featurestuktur besitzen kann, an dieser Stelle verwendet. Da auch diese Unifikation wieder scheitern kann stehen wiederum die Möglichkeiten von Constraintabschwächung oder Ersatzregeln für diese Ersatzregeln zur Verfügung. Diese Regeln sind nicht Bestandteil der Regelmenge R der Grammatik. Sie werden unabhängig von dieser Menge gespeichert, um zu verhindern daß bei der Anwendung der Primärregel auch noch zusätzlich die Ersatzregel benutzt wird.

Im Falle einer falschen Wortstellung sind die Fehlerregeln Bestandteil der Regelmenge R, da ihnen ein völlig anderes Phrasenstrukturskelett zugrunde liegt. Damit ist eine parallele Anwendug von Primarregel und Fehlerregel ausgeschlossen.

```
VP : VBAR #!subcat-error
[
 VBAR:[
   SUBCAT=<>;
 ]
 VP:[
#use kongr1;
   SUBCAT=VBAR:SUBCAT;
   TYPE=VBAR:TYPE;
 ]
]

VP : VBAR #!subcat-error
[
 VP:[
#use kongr1;
```

```
    SUBCAT=VBAR:SUBCAT;
    TYPE=VBAR:TYPE;
    ERROR->AKTERRS=<"Invalid verb usage, nonemtpty subcat list">;
  ]
 ]
```

In diesem Beispiel werden die beiden Regeln über die Anweisung `#!subcat-error` miteinander verbunden. Damit ist die zweite Regel eine Alternativregel zu der ersten, die verwendet wird, falls die erste Regel nicht korrekt angewandt werden kann. Damit lassen sich auch Fehler modellieren, die mit einer Koreferenzrücknahme nur schwer oder gar nicht modellierbar sind.

3.4.3. Ein weiterer regelbasierter Ansatz

Ein weiterer regelbasierter Ansatz nutzt die Informationen in der Chart aus, um eine Regelanwendung zu kontrollieren. In diesem Fall kann jeder Regel eine Menge an Regeln zugeordnet werden, deren Anwendung die Ausgangsregel blockieren. Falls eine Regel $A \rightarrow \alpha$ auf einem Eingabebschnitt i, j angewandt werden soll, wird geprüft, ob eine der blockierenden Regeln auf diesem Abschnitt schon angewandt wurde. Existiert ein solches Chartitem $[B \rightarrow \beta, i, j]$, dann wird die Regel $A \rightarrow \alpha$ nicht angewandt. Es ist klar, daß diese Vorgehensweise von der Regelreihenfolge und dem genauen Ablauf des Parsings abhängt, d. h. in einigen Fällen fehlschlägt; dennoch war dies im *Semf*-Projekt in einigen Fällen der einzige Weg bei einer Fehlergrammatik bestimmte Regelabhängigkeiten auszudrücken. Aufgrund der Probleme soll dieser Ansatz hier jedoch nicht weiter erläutert werden.

3.5. Funktionale Ausdrücke

Eine Erweiterung gegenüber klassischen unifikationsbasierten Parsern sind die funktionalen Ausdrücke, die das System beherrscht. Funktionale Ausdrücke sind Funktionen, die in *C* implementiert sind und auf alle internen Datenstrukturen des Parsers und des Lexikons zugreifen können. Über diese Funktionen kann erheblich mehr implementiert werden, als es mit einfachen Unifikationsgrammatiken möglich ist, da die Funktionen völlig neue Operatoren definieren. Im Sinne der zusammengesetzten Merkmalsstrukturen definieren Funktionen neue Bäume, deren Wurzeln, die Funktionsergebnisse, durch Pfadkoreferenzen in die Featurestruktur eingebunden werden.

Im Anhang (vgl. Anhang B.2) sind die Funktionen aufgeführt, die beim Parser vordefiniert sind. Über eine festgelegte Schnittstelle können neue Funktionen definiert werden, die dann ebenfalls in einer Grammatik zur Verfügung stehen. Es ist jedoch zu beachten, daß Funktionen den bisher deklarativen Rahmen der Featurestrukturen sprengen. Näheres dazu findet sich in Abschnitt 3.6.7.

3.6. Formale Betrachtung der Systemerweiterungen

In diesem Abschnitt werden die Modifikationen, die zur Repräsentation des Systems an den bisher vorgestellten formalen Beschreibungen vorgenommen werden müssen, näher betrachtet. Dabei wird in den folgenden Unterabschnitten jeweils eine Erweiterung unter formalen Gesichtspunkten analysiert.

3.6.1. Typisierte Features

Eine der zu betrachtenden Erweiterungen im formalen System des Parsers sind typisierte Features. Diese Typen erzeugen weitere Constraints bei den konsistenten Featurestrukturen. Das weitere Vorgehen in Abschnitt 2.6 wird durch diese neue Definition nicht beeinflußt.

Definition 3.6.1 (Typen) *Die Menge $\mathcal{T}ype$ ist die Menge aller verfügbaren Typen.*

Definition 3.6.2 (Typenzuordnug) *Die Funktion*

$$\mathcal{T}ypeof : \mathcal{F}eat \cup \mathcal{V}al \cup \{[\ ]\} \rightarrow \mathcal{T}ype \oplus \{Subfeature\}$$

ordnet jedem Feature und jedem möglichen Wert einen Typ zu. Der leeren Featurestruktur wird ein Typ zugewiesen, der von allen anderen Typen in $\mathcal{T}ype$ verschieden ist.

Definition 3.6.3 (Konsistenz bei getypten Features) *Eine abgeschlossene Featurestruktur χ ist konsistent, wenn gilt:*

1. *χ ist endlich;*

2. $\langle X\pi f\rangle\doteq c \in \chi(X) \rightarrow \mathcal{T}ypeof(f) = \mathcal{T}ypeof(c)$;
3. $\langle X\pi f\rangle\doteq\langle X\varrho g\rangle \in \chi(X) \rightarrow \mathcal{T}ypeof(f) = \mathcal{T}ypeof(g)$;
4. $\langle X\pi\rangle\doteq c \in \chi(X) \wedge \langle X\pi\rangle\doteq d \in \chi(X) \rightarrow c = d$;
5. $\langle X\pi\rangle\doteq c \in \chi(X) \wedge \langle X\pi\varrho\rangle\doteq\mu \in \chi(X) \rightarrow \varrho = \varepsilon$;
6. $\langle X\pi\varrho\rangle\doteq\langle X\pi\rangle \notin \chi(X)\ \forall\pi \in \mathcal{F}eat^*, \varrho \neq \varepsilon$.

3.6.2. Wertdisjunktionen

Eine formale Betrachtung disjunktiver Featurestrukturen findet sich ansatzweise in (Sikkel, 1997). Grundsätzlich ist die Verwendung der Strukturen ein Optimierungsproblem, bei dem die Anzahl der Chartitems reduziert wird. Man kann alle disjunktiven Einträge ausmultiplizieren, d. h. aus einer Featurestruktur, die eine Disjunktion enthält, diese auflösen. Dies ist möglich, da nur atomare Wertte in den Disjunktionen vorkommen. Dabei wird für jedes Element der Disjunktion eine neue, bei dem bearbeiteten Attribut nur mit diesem Element versehene, Featurestruktur erzeugt. Dies muß iterativ für jede Disjunktion in einer Featurestruktur durchgeführt werden:

$$\begin{bmatrix}\text{PERS} & \{\text{FIRST+SECOND, THIRD}\}\\ \text{NUMBER} & \{\text{SING, PLUR}\}\end{bmatrix}$$

$$\Rightarrow \begin{bmatrix}\text{PERS} & \text{first+second}\\ \text{NUMBER} & \{\text{sing, plur}\}\end{bmatrix}, \begin{bmatrix}\text{PERS} & \text{third}\\ \text{NUMBER} & \{\text{sing, plur}\}\end{bmatrix}$$

$$\Rightarrow \begin{bmatrix}\text{PERS} & \text{first+second}\\ \text{NUMBER} & \text{sing}\end{bmatrix}, \begin{bmatrix}\text{PERS} & \text{third}\\ \text{NUMBER} & \text{sing}\end{bmatrix},$$
$$\begin{bmatrix}\text{PERS} & \text{first+second}\\ \text{NUMBER} & \text{plur}\end{bmatrix}, \begin{bmatrix}\text{PERS} & \text{third}\\ \text{NUMBER} & \text{plur}\end{bmatrix}$$

3.6.3. Neue Featuretypen

Es gib zwei neue Arten von Featuretypen, die berücksichtigt werden müssen:

1. konjunktive Featuremengen und
2. Listen.

Diese beiden Typen müssen noch in die gegebene formale Beschreibung der Algorithmen und Datenstrukturen eingebettet werden.

Neben den Constraints der Form $\langle X\pi\rangle \doteq c$ und denen der Form $\langle X\pi\rangle \doteq \langle X\varrho\rangle$ sind nun auch Constraints der Form $\langle X\pi\rangle \doteq \{c_1, \ldots, c_n\}$ zugelassen. Der Abschluß wird erweitert um die Bedingung

$$\begin{aligned} \langle X\pi\rangle \quad &\doteq \quad \{c_1, c_2, \ldots, c_2\} \in \chi(X) \\ &\rightarrow \quad \langle X\pi\rangle \doteq C \in \text{closure}(\chi(X))\ \forall C \in \wp(\{c_1, c_2, \ldots, c_2\}). \end{aligned}$$

Mit dieser Erweiterung sind alle Teilmengen der konjunktiven Menge ebenfalls im Abschluß, und die Subsumptionsrelation kann wie bisher definiert werden. Es muß nun nur noch die Konsistenz einer Untersuchung unterzogen werden. Die Konsistenz wird erweitert um

$$\langle X\pi\rangle \doteq C \in \chi(X) \wedge \langle X\pi\rangle \doteq D \in \chi(X) \rightarrow \langle X\pi\rangle \doteq E \wedge (C \cup D) \subseteq E.$$

Damit kann die Subsumption wie gehabt definiert werden. Es ergibt sich auch für diese erweiterte Struktur ein Verband mit den erwünschten Eigenschaften.

Die Listen sind äquivalent zu speziellen Featurestrukturen. Die Struktur

$$\texttt{< } \mathtt{F}_1 \texttt{ = } \mathtt{V}_1\texttt{, } \mathtt{F}_2 \texttt{ = } \mathtt{V}_2\texttt{, ..., } \mathtt{F}_n \texttt{ = } \mathtt{V}_n \texttt{ >}$$

kann man übersetzen in

$$\begin{bmatrix} \text{FIRST} & \begin{bmatrix} \text{F}_1 & \text{V}_1 \end{bmatrix} & & \\ \text{NEXT} & \begin{bmatrix} \text{FIRST} & \begin{bmatrix} \text{F}_2 & \text{V}_2 \end{bmatrix} & & \\ \ldots & & \begin{bmatrix} \text{FIRST} & \begin{bmatrix} \text{F}_n & \text{V}_n \end{bmatrix} \\ \text{NEXT} & \top \end{bmatrix} \ldots \end{bmatrix} \end{bmatrix} \end{bmatrix}.$$

Wenn man diese Übersetzung aller Featurestrukturen durchführt, kann man die Featurelisten mit den bisherigen Mitteln ausdrücken. Die Listen sind daher nichts anderes als notationelle Varianten verketteter Featurestrukturen, die lediglich zur besseren Lesbarkeit dienen. Auf der Implementationsseite ist die Einschränkung sinnvoll, da sie eine erheblich effizientere Implementierung der Unifikation von Listen erlaubt.

3.6.4. Mehrwortlexeme

Mehrwortlexeme erfordern im unifikationsbasierten Earley-Parser einige kleine Modifikationen. Die modifizierten Elemente sind hier aufgeführt:

$$
\begin{aligned}
D^{scan} &= \{[A \to \alpha \bullet a\beta, i, j]_\eta, [a, j, k]_\zeta \vdash [A \to \alpha a \bullet \beta, i, k]_\xi \\
&\qquad \mid \varphi(\xi) = \varphi(\eta) \sqcup \varphi(a_\zeta)\} \\
\alpha, \beta, \gamma &\in V^* \\
a &\in V_T \\
A, B &\in V_N \\
\mathcal{K}(a_1, \ldots, a_n) &= \{[a, i, i+l]_\xi \mid a = a_{i+1} \cdots a_{i+1} \wedge 0 \leq i < n \wedge 0 < l \leq n - i \\
&\qquad \wedge \varphi(\xi) \in \mathcal{L}ex(a_{i+1} \cdots a_{i+l})\}.
\end{aligned}
$$

3.6.5. Unifikationsbasierte Fehlererkennung

Um eine unifikationsbasierte Fehlererkennung zu realisieren, muß die Unifikationsoperation modifiziert werden, da die neue Unifikation – im Gegensatz zur Standardunifikation – statt einer einzigen Featurestruktur eine Menge möglicher Featurestrukturen als Ergebnis liefert. Diese neue Unifikation in einem formalen Rahmen zu beschreiben würde allerdings den Rahmen dieser Arbeit sprengen. Die verschiedenen Möglichkeiten können zwar auch auf Regelbasis dargestellt werden, dies gibt jedoch den dynamischen Prozess der Operation nicht wieder. Das Problem dabei ist, daß der Subset an Regeln erkannt werden muß, der die Menge der speziellsten Featurestrukturen, die gebildet werden können, benutzt. Diese Aufgabe läßt sich deklarativ nur schwer lösen, wenn die Featurestrukturen nicht bekannt sind. Ist dies der Fall, müßte die Menge der Ableitungsregeln temporär modifiziert werden, um die korrekten Chartitems zu bilden. Auch dies ist nur schwer mit den gegebenen Mitteln modellierbar.

3.6.6. Regelbasierte Fehlererkennung

Bei der regelbasierten Fehlererkennung gibt es für jede Regel $A \to \alpha$ eine Ersatzregelmenge, in der Regeln mit gleicher Struktur – aber unterschiedlichen Featureannotationen – stehen. Diese Regelmenge wird mit $\delta(A \to \alpha)$ bezeichnet. Dazu muß der Earley-Parser etwas modifiziert werden indem z. B. alle Chartitems mit zwei Featurestrukturen ausgestattet werden.

Definition 3.6.4 (Earley-Parser mit Unifikation und Fehlerregeln)

$$
\begin{array}{rcl}
\mathcal{I} & = & \{[A \to \alpha \bullet \beta, i, j]_{\xi_1,\xi_2} | A \to \alpha\beta \in R, \wedge 0 \leq i \leq j \\
 & & \wedge \varphi_0(A \to \alpha\beta) \preceq \varphi(\xi_1)\} \\
D^{init} & = & \{\vdash [S \to \bullet\gamma, 0, 0]_{\xi_1,\xi_2} \\
 & & \quad | \, \varphi(\xi_1) = \varphi_0(S \to \gamma) \wedge \varphi(\xi_2) = \bot\} \\
D^{scan} & = & \{[A \to \alpha \bullet a\beta, i, j]_{\eta_1,\eta_2}, [a, j, j+1]_{\zeta_1,\zeta_2} \vdash [A \to \alpha a \bullet \beta, i, j+1]_{\xi_1,\xi_2} \\
 & & \quad | \, \varphi(\xi_1) = \varphi(\eta_1) \sqcup \varphi(a_{\zeta_1}) \wedge \varphi(\xi_2) = \varphi(\eta_2) \sqcup \varphi(a_{\zeta_2})\} \\
D^{compl} & = & \{[A \to \alpha \bullet B\beta, i, j]_{\eta_1,\eta_2}, [B \to \gamma\bullet, j, k]_{\zeta_1,\zeta_2}, \varphi(\zeta_1) \neq \top \\
 & & \quad \vdash [A \to \alpha B \bullet \beta, i, k]_{\xi_1,\xi_2} \\
 & & \quad | \, \varphi(\xi_1) = \varphi(\eta_1) \sqcup \varphi(B_{\zeta_1}) \wedge \varphi(\xi_2) = \varphi(\eta_2) \sqcup \varphi(B_{\zeta_1})\} \\
D^{fail} & = & \{[A \to \alpha\bullet, i, j]_{\zeta_1,\zeta_2}, \varphi(\zeta_1) = \top \\
 & & \quad \vdash [(A \to \alpha)'\bullet, i, j]_{\xi_1,\xi_2} \mid (A \to \alpha)' \in \delta(A \to \alpha) \\
 & & \quad \wedge \varphi(\xi_1) = \varphi(\zeta_2) \sqcup \varphi_0((A \to \alpha)') \wedge \varphi(\xi_2) = \varphi(\zeta_2) \\
D^{pred} & = & \{[A \to \alpha \bullet B\beta, i, j]_{\eta_1,\eta_2} \vdash [B \to \bullet\gamma, j, j]_{\xi_1,\xi_2} \\
 & & \quad | \, \varphi(\xi_1) = \varphi_0(B \to \gamma) \wedge \varphi(\xi_2) = \bot\} \\
D & = & D^{init} \cup D^{scan} \cup D^{compl} \cup D^{fail} \cup D^{pred} \\
\alpha, \beta, \gamma & \in & V^* \\
a & \in & V_T \\
A, B & \in & V_N
\end{array}
$$

Die Funktion $\mathcal{K}$, die aus dem Input die initialen Chartitems generiert, hat die Form

$$\mathcal{K}(a_1, \ldots, a_n) = \{[a, i-1, i]_{\xi,\xi} \mid a = a_i \wedge 0 < i \leq n \wedge \varphi(\xi) \in \mathcal{L}ex(a_i)\}.$$

3.6.7. Funktionen und Deklarativität

In diesem Abschnitt wird die Einschränkung der Deklarativität des Unifikationsmechanismus betrachtet. Diese Einschränkungen sind wesentlich bei der Erstellung der Grammatiken, da bei einem nur eingeschränkt deklarativen Mechanismus die Constraintreihenfolge einen wesentlichen Einfluß auf das Unifikationsergebnis hat.

Bei einem Constraintset ist die Reihenfolge der Constraints unwichtig, da sie eine Struktur definieren, die in eine Unifikationsoperation eingeht. Bei der Im-

plementierung werden diese einzelnen Constraints in einer spezifischen Reihenfolge abgearbeitet. Der Unifikationsalgorithmus ist jedoch unabhängig von der Reihenfolge. Eine Funktion schafft einen neuen Baum in der zusammengesetzten Featurestruktur, der allerdings zu dem Zeitpunkt der Bearbeitung dieses speziellen Constraints erzeugt wird. Legt diese Funktion eine Kopie einer Featurestruktur an[9] statt eine Pfadkoreferenz zu fordern, dann kopiert sie diese Struktur in der Form, die der Unifikationsalgorithmus bisher erzeugt hat. Spätere Modifikationen, die durch andere Constraints entstehen, haben keinen Einfluß auf diese Kopie. Somit ist die Reihenfolge der Pfadkoreferenzen für das Ergebnis wesentlich, sobald eine Funktion benutzt wird. In einer geplanten Neuimplementierung der Featurerepräsentation können diese Schwierigkeiten aber zumindest zum Teil gelöst werden, weil die Funktionen in der neuen Implementierung expliziten Zugriff auf die Koreferenzbeziehungen haben werden.

[9] Ob eine Funktion eine Kopie der Struktur anlegt oder nicht, muß im einzelnen Fall nachgeprüft werden.

4. Entwicklungsumgebungen

Der bisher vorgestellte Parser bildet einen Teil eines Sprachanalysesystems, hier auch *NLP-System* genannt[1], das zusammen mit einer Grammatik und einem Lexikon für verschiedenste Zwecke eingesetzt werden kann. Der Schwerpunkt in den folgenden Kapiteln liegt auf der effizienten Erstellung von Grammatiken und Lexika, die ein wesentlicher Bestandteil bei der Entwicklung vieler natürlichsprachlichen Systeme ist. Dabei liegt der Schwerpunkt auf der technischen Ebene, d. h. es liegt die Erstellung eines den Anforderungen angemessenen Systems im Vordergrund. Diese Herangehensweise wird auch als *linguistic Engineering* bezeichnet (vgl. Nijholt (1994)).

4.1. Anforderungen an eine Entwicklungsumgebung

Um ein solches NLP-System effizient erstellen zu können bedarf es allerdings noch vieler Werkzeuge, die diese Arbeit unterstützen. Diese Werkzeuge lassen sich grob in drei Kategorien einteilen:

Werkzeuge zur Generierung von Grammatiken und Lexika Die Generierungstools dienen der Verwertung bisheriger Informationsquellen, z. B. Korpora und maschinenlesbare Lexika, aus denen möglichst automatisch Teile der Grammatik und des Lexikons extrahiert werden sollen. Dieses Gebiet umfaßt einfache Konverter, die die Information aus einem systemfremden in das systemeigene Format umwandeln, Systeme, die gegebene Informationen extrahieren und standardisieren (vgl. Hötker et al. (1994)), sowie Systeme, die versuchen, aus Korpora neue Informationen zu gewinnen, die durch einen Linguisten aufbereitet werden können.

[1] Die NLP-Systeme umfassen neben den Analysesystemen auch Sprachgenerierungssysteme und alle anderen Systeme, die sich mit Sprachverarbeitung im weitesten Sinne befassen.

Bearbeitungs- und Visualisierungstools Die Bearbeitungs- und Visualisierungstools setzen auf der bearbeitenden Ebene auf, d. h. sie stellen ein Werkzeug dar, mit dem eine bestehende Grammatik bzw. ein bestehendes Lexikon geändert und ereitert werden kann. Dieser Bereich wird von dem vorgestellten System zumindest teilweise abgedeckt. Der Schwerpunkt liegt hierbei jedoch nicht auf der Bearbeitung, sondern auf der Visualisierung der Daten. Dies ist dabei vor allem ein Hilfsmittel, um die Testumgebung zu unterstützen und nur indirekt ein Tool zur Grammatikerstellung.

Testumgebungen Testumgebungen sind Programme, die zumindest teilweise automatische Verifikation eines NLP-Systems erlauben. Der wesentliche Bestandteil ist dabei eine Testsatzsammlung, in der sowohl korrekte als auch inkorrekte Sätze abgelegt sind. Diese Sätze werden automatisch an das NLP-System zur Analyse weitergereicht, und das Ergebnis der Analyse wird mit dem erwarteten Ergebnis verglichen. Über diesen Vergleich kann die korrekte Abdeckung eines Sprachfragmentes durch ein NLP-System überprüft werden.

Eine Entwicklungsumgebung sollte idealerweise alle diese Funktionen zur Verfügung stellen, um eine möglichst effiziente Arbeit zu ermöglichen. Diese Entwicklungsumgebung kann sich jedoch aus Einzelsstemen zusammensetzen, die jeweils einen Teil der Aufgaben erfüllen. Eine natürliche Aufgabentrennung ergibt sich z. B. durch die oben vorgestellte Aufteilung der Tools in funktional abgetrennte Bereiche.

Der Schwerpunkt in dieser Arbeit liegt auf dem dritten und auf Teilen des zweiten Bereichs. Diese sollen im Weiteren näher betrachtet werden.

4.2. Bearbeitung und Visualisierung der Daten

Die Bearbeitung einer Grammatik kann auf verschiedene Art erfolgen. Einige Systeme verwenden einen speziellen Editor, der die Korrektheit – auf der Ebene des Grammatikformalismus – direkt bei der Eingabe prüft und, im Idealfall, alle verwendeten Symbole kennt und auf Anfrage eine Erklärung bieten kann. Ein solcher Editor kann den Entwickler direkt unterstützen, da er eine Repräsentation des Wissens über die Domäne, in der er eingesetzt wird, besitzt. Viele Systeme benutzen allerdings einen normalen Texteditor, in dem die Grammatik wie ein beliebiger anderer Text eingegeben wird. Diese Art von Editoren

kann natürlich keine vergleichbare Unterstützung bei der Grammatikerstellung liefern.

Ein weiteres Werkzeug bei der Grammatikerstellung sind Visualisierungstools, die die Daten aufbereiten und unter speziellen Gesichtspunkten anzeigen. Dabei ist Visualisierung nicht gleichbedeutend mit grafischer Anzeige. Ein solches Tool hat die Aufgabe, die Zusammenhänge innerhalb eines Systems so anzuzeigen, daß sie einfacher durchschaubar sind. Dies kann durch die Anzeige aller Ersetzungsalternativen für ein Symbol, die Auflistung aller Lexikoneinträge zu einem terminalen Symbol oder die grafische Anzeige der Parsingbäume zu einem Satz versucht werden. Die Tools sollten nicht nur die Grammatik als statisches Objekt, sondern auch den Ablauf eines Parsings darstellen können. Welche Visualisierungen in einem System sinnvoll sind und welche nicht, zeigt sich erst im praktischen Einsatz eines Systems.

Eine wesentliche Aufgabe hat die Visualisierung bei der Analyse von unerwünschten Parsingergebnissen, d. h. bei der Suche nach fehlerhaften Regeln in der Grammatik oder nach fehlerhaften Lexikoneinträgen. Dabei muß schnell festgestellt werden können, welche Regel nicht wie erwartet reagiert. Dies muß nicht immer die Regel sein, in der der Fehler auftritt, da die eigentliche Ursache in einer vorher angewandten Regel liegen kann. Eine generelle Richtlinie, wie solche Tools aussehen sollten, gibt es leider nicht. Es gibt lediglich einige Minimalanforderungen, die ein System erfüllen sollte:

- Das Ergebnis jeder Regelanwendung muß sichtbar sein.
- Alle lexikalischen Analysen, die für ein Wort gefunden werden, müssen untersucht werden können.
- Es sollten alle Daten, die in der Grammatik vorliegen, abrufbar sein. Dies umfaßt Lexikoneinträge, Regeln, Feature- und Typendefinitionen.

In Richtung der Visualisierungstools gibt es bis jetzt einige Ansätze, die in einige Entwicklungsumgebungen integriert sind. Eine Übersicht über verschiedene Entwicklungsumgebungen findet sich in (Volk, 1995). Dort ist erwähnt, welche Möglichkeiten die verschiedenen Systeme bieten.

4.3. Testumgebungen für Parser

Der letzte wesentliche Bestandteil ist eine in die Entwicklungsumgebung integrierte Testumgebung. Der Begriff Testumgebung umfaßt dabei sowohl gram-

matikinterne Tests, in denen inkonsistente Featurestrukturen, unerreichbare oder nicht expandierbare Symbole und sonstige grammatikinhärente Phänomene gefunden werden sollen, als auch Anwendungstests, bei denen für einzelne Sätze geprüft wird, ob der Satz analysierbar ist oder nicht. So kann systematisch die Leistungsfähigkeit eines NLP-Systems ermittelt werden. Ein solcher Test ist wesentlich, um die Einsatzfähigkeit eines Systems zu gewährleisten.

4.3.1. Testsatzsammlungen

Definition 4.3.1 (Testsatzsammlung) *Eine Testsatzsammlung ist eine Menge von Sätzen – sowohl grammatisch korrete als auch inkorrekte – zusammen mit einer Annotierung der Sätze durch syntaktische und evtl. semantische Merkmale.*

Eine etwas detailliertere und strengere Definition des Begriffs Testsatzsammlung findet sich in (Volk, 1995). Die wesentliche Information über eine Testsatzsammlung findet sich aber auch in der hier gegebenen Definition.

Eine Testsatzsammlung dient im Wesentlichen folgenden Zwecken:

1. Dem Test des Systems anhand gegebener Vorgaben über zu verarbeitende Sätze. Hierbei wird für alle Vorgaben geprüft, ob die Analyse des Satzes mit dem System möglich ist. So kann schnell verifiziert werden, ob das System alle Anforderungen tatsächlich erfüllt, bzw. welcher Teil der Anforderungen noch nicht erfüllt werden kann.

2. Der Verifikation des Systems und der Dokumentation seiner Entwicklung. Wenn zu jeder Entwicklungsstufe ein Protokoll des Testlaufes, bei dem alle Testsätze dem System zur Analyse übergeben werden, gespeichert wird, dann dokumentiert dies automatisch die Veränderungen – aus der Sicht des Anwenders – des Systems von einer Entwicklungsstufe zur nächsten. Jeder Satz, der im Gegensatz zu einer älteren Version, analysiert werden kann, definiert einen Fortschritt während jeder Satz, den eine ältere Version bearbeiten konnte während eine neue Version die Bearbeitung abbricht, auf einen Fehler hindeutet. Sind in einer Testsatzsammlung auch Sätze enthalten, die das System nicht analysieren können soll, etwa bei Grammatikcheckern, dann kehrt sich dieses Verhältnis für die entsprechenden Sätze natürlich um.

Die Annotation der Testsätze muß dazu die Informationen, die zur Überprüfung der Analyseergebnisse notwendig sind, enthalten. Diese Annotierung bildet die wesentliche Information der Testsatzsammlung. Ihre Vollständigkeit und Konsistenz ist wesentlich für die Einsatzfähigkeit einer Testsatzsammlung. In (Volk, 1995) und (Fouvry und Balkan, 1996) finden sich Konzepte, nach denen eine Testsatzsammlung erstellt werden sollte.

Erstellung einer Testsatzsammlung

Wenn ein NLP-System erstellt wird, sollte vor der Implementation der Grammatik die genaue Definition des Sprachfragmentes erfolgen, mit dem das System umgehen können muß. Zu jedem Phänomen des Sprachfragmentes sollte mindestens ein dieses Phänomen benutzender Beispielsatz vorhanden sein. Diese Beispielsammlung bildet einen Anforderungskatalog für die Fähigkeiten, die das NLP-System besitzen muß.

Die Beispielsätze bilden eine erste Testsatzmenge, die das System verarbeiten können sollte. Die Sätze müssen, z. B. im Sprachlehrkontext, nicht unbedingt korrekt sein, da die Analysierbarkeit – im Sprachlehrkontext mit Informationen über die Fehler – und nicht die grammatische Korrektheit das wesentliche Kriterium ist.

Neben den Sätzen, bei denen eine Analyse des Systems möglich sein muß, sollten in der Beispielsammlung auch Sätze vorhanden sein, bei denen das System keine erfolgreiche Analyse liefern darf. Dies ist, wie bereits erwäht, vor allem im Kontext von Grammatikcheckern wichtig, da so eine Übergeneralisierung der Grammatik erkannt werden kann. Die analysierbaren und die nicht analysierbaren Sätze können so deutlich voneinander abgegrenzt werden. Die so erstellte Testsatzmenge kann dann inkrementell erweitert werden, bis für alle erwartbaren Klassen von Eingaben ein Beispielsatz vorhanden ist.

Eine Idee zur systematischen Erweiterung der Testsatzsammlung, die in (Arnold et al., 1994) vorgestellt wird, ist die Generierung von Testsätzen durch einen möglichst simplen Formalismus. Dieser unterliegt nicht den strengen Restriktionen der Grammatik, denn:

> „you do not care about all the overgeneration that is usually a problem – this is all useful (negative) test data – and anyway, it is easy to go through the TS database that is produced by hand and remove really silly examples.“ (Arnold et al. (1994), S. 14).

Arnold geht dabei von einer Situation aus, in der nach Entwicklungsstand eines NLP-Systems durch das Generierungssystem *Auto-TSG* eine Testsatzsammlung erstellt wird. Diese Satzsammlung wird von Hand durch Annotationen, unter anderem auch über die Korrektheit, angereichert. Diese manuelle Annotation macht diesen Ansatz für große Testsatzsammlungen wenig akzeptabel.

Bestehende Testsatzsammlungen

Um die aufwendige Erstellung einer Testsatzsammlung nicht für jedes NLP-System erneut leisten zu müssen, gibt es vorgefertigte, oft frei verfügbare Testsatzsammlungen die zum Test von NLP-Systemen verwendet werden können. Das Projekt *Test Suite for Natural Language Processing* (*TSNLP*) setzt sich speziell mit Testsatzsammlungen auseinander. Dieses Projekt hat eine Methodik vorgestellt, die die Entwicklung einer möglichst universellen Testsatzsammlung ermöglicht. Eine solche Sammlung wurde in diesem Projekt schon erstellt und ist frei verfügbar.

Die Anforderungen an eine solche Testsatzsammlung sind nach (Volk, 1995):

Repräsentativität Eine Testsatzsammlung ist dann repräsentativ für ein Sprachfragment, wenn eine Grammatik bei der Testsatzsammlung den gleichen Prozentsatz an Sätzen verarbeiten kann wie in einem Textkorpus, der das gleiche Sprachfragment beschreibt. Dies ist natürlich nur dann gewährleistet, wenn die Häufigkeitsverteilung bei verschiedenen Phänomenen mit der Verteilung in einem natürlichen Textkorpus übereinstimmt. Dies ist manchmal kein erwünschter Effekt, wenn sehr unterschiedliche Phänomene mit unterschiedlichen Häufigkeiten vorhanden sind. In dieser Situation kann diese Forderung zu häufigen Testsätzen mit immer ähnlichen einfachen Phänomenen und wenigen Testsätzen mit komplexen Phänomenen führen.

Vollständigkeit Eine Testsatzsammlung sollte idealerweise alle syntaktischen Phänomene eines Sprachfragmentes erfassen.

Annotierung Die einzelnen Sätze sollten möglichst vollständig mit Informationen über die Phänomene des Satzes annotiert werden. Bei inkorrekten Sätzen muß eindeutig der Fehler, den der Satz produziert, beschrieben werden. Um diese Annotierungen sinnvoll verwenden zu können muß sie natürlich konsistent in der gesamten Satzmenge verwendet werden.

Handhabbarkeit Die Sätze in der Satzsammlung sollten anhand eines durchschaubaren Systems sortiert sein, um gezielt zu einem Phänomen einen Satz in der Testsatzsammlung finden zu können. Außerdem ist so gewährleistet, daß Teilmengen der Phänomene ausgewählt werden können. Diese Anforderungen sollen einen effizienten Zugriff auf die Sätze der Testsatzsammlung garantieren. Neben dem Zugriff auf die Satzsammlung muß diese auch noch erweitert werden können. Dies setzt ebenfalls eine klare Strukturierung der Daten voraus, denn nur so kann ein neuer Satz in die bestehende Ordnung eingefügt werden. (Volk, 1995) schlägt einen Baum vor, der die verschiedenen Phänomene strukturiert. Allen Testsätzen wird hierbei ein Blatt des Baumes zugeordnet, das dem Phänomen entspricht. Bei mehreren Phänomenen innerhalb eines Satzes werden dem Satz mehrere Blätter zugeordnet. So können gezielt einzelne Phänomene oder Sätze mit Phänomenkombinationen aus der Testsatzmenge herausgesucht werden.

Begrenztes Vokabular Das Vokabular einer Testsatzsammlung sollte, soweit möglich, beschränkt sein. Dies dient der Reduktion des für Tests notwendigen Lexikons. Nur durch ein möglichst kleines Lexikon können evtl. Fehler auf die Grammatik begrenzt werden, da ein sehr großes Lexikon seinerseits wieder viele Fehler enthalten kann, die unerwartete Analysen produzieren. Diese Forderung dient also einer praktikablen Eingrenzung von Fehlern auf die Grammatik, die bei den Testsatzsammlungen der wesentliche Untersuchungsgegenstand ist. Eine Ausnahme bilden hier natürlich die Testsätze, die im Anforderungskatalog enthalten sind. Diese Sätze können so gewählt sein, daß sie einen möglichst großen Teil des Vokabulars abdecken, das das System beherrschen soll. Diese Sätze sollten dann allerdings auf der grammatischen Ebene möglichst einfach gehalten sein, damit hier die Fehlerquelle möglichst auf der lexikalischen Seite liegt.

Die hier gestellten Anforderungen können, da sie teilweise auch gegensätzlicher Natur sind, nie vollständig erreicht werden. Das Ziel bei der Erstellung einer Testsatzsammlung ist daher, die Anforderungen möglichst optimal zu erfüllen. Die Gewichtung der Anforderungen kann auch von Anwendung zu Anwendung variieren, daher gibt es für spezielle Anwendungstypen spezielle Testsatzsammlungen. Ein Beispiel ist die *Madeline Bates Test Suite*, die spezielle Testsätze für natürlichsprachliche Datenbankanfragen enthält.

Datenaustausch mit bestehenden Testsatzsammlungen

Der Zugriff auf die Daten einer Testsatzsammlung geschieht in der Regel über eine fest definierte Datenbankschnittstelle (vgl. Oepen et al. (1996)) oder einen sonstigen standardisierten Formalismus wie *SGML*, durch den die Informationen strukturiert werden (vgl. Volk et al. (1994)). Diese Datenbanken liefern zu gegebenen Selektionskriterien die gewünschten annotierten Testsätze.

Um diese Testsätze verwenden zu können, müssen die verschiedenen Notationen ineinander überführt werden. Es muß daher ein Konvertierungssystem vorhanden sein, das möglichst automatisch die Annotationen der Testsatzsammlung in ein für die Testumgebung verständliches Format umwandelt. Diese Aufgabe reicht von der Umbenennung einiger Featurenamen bis hin zur Konvertierung der semantischen Repräsentation eines Satzes. Der letztere Teil kann relativ aufwendig werden, da sehr unterschiedliche Formalismen benutzt werden können. Bei den syntaktischen Repräsentationen sind die üblicherweise verwendeten Formalismen ähnlich; es sind in der Regel Featurestrukturen. Daher besteht hier oft eine einfachere Möglichkeit zur Konvertierung der Formate. Weiterhin muß die annotierte Information eventuell gefiltert werden, um für die Testumgebung irrelevante Daten aus den Annotationen zu entfernen. Das *TSNLP*-Projekt befasich vor allem mit der Fragestellung, wie eine möglichst systemunabhängige Notation für die notwendigen Annotationen gefunden werden kann. Eine solche systemunabhängigie Notation sollte relativ problemlos in das Format einer Testumgebung konvertierbar sein. Bei der Spezifikation der Grammatik sollte allerdings auf eine grundsätzliche Kompatibilität der verwendeten Strukturen geachtet werden, um die Konvertierung der Formate möglichst einfach zu gestalten.

Neben der Konvertierung der Annotationen muß natürlich auch noch das verwendete Vokabular angepasst werden. Dazu gibt es zwei grundlegende Ansätze:

- Die Anreicherung des systemeigenen Vokabulars um das in den Testsätzen verwendete Vokabular.
- Die Anpassung der Testsätze auf das systemeigene Vokabular.

Die erste Variante hat den Vorteil, daß die Testsätze ohne weitere Modifikation verwendet werden können. Der Aufwand für Konvertierungen wird daher gering gehalten. Bei NLP-Systemen, die ein domänenspezifisches Vokabular verwenden, ist diese Vorgehensweise natürlicht nicht akzeptabel, da hier ein entscheidender Bereich, das domänenspezifische Systemlexikon, nicht in den

Test eingeht. Die zweite Variante löst dieses Problem, es tritt aber das Problem der Testsatzkonvertierung auf. Dabei müssen alle Worte in den Testsätzen, die das System nicht kennt, durch dem System bekannte Worte ersetzt werden. Das Problem liegt in diesem Fall darin, die geeigneten Worte für die Ersetzung zu identifizieren. In (Arnold et al., 1994) wird ein halbautomatisches Vorgehen vorgeschlagen, bei dem der Texteditor *EMACS* um die benötigte Funktionalität erweitert wird. Nachdem der Benutzer für ein Wort einen entsprechenden Ersatz gefunden hat, versucht das System, eine entsprechende Korrektur aller weiteren Testätze vorzunehmen. Auch dieser Ansatz ist Bestandteil des *TSNLP*-Projektes.

5. Die Entwicklungsumgebung

Im vorliegenden System sind eine Visualisierungskomponente und eine Testumgebung realisiert. Die Visualisierungskomponente kann sowohl Regeln und Lexikoneinträge als auch Strukturbäume darstellen. Die Testumgebung dient zur Definition von Testsätzen, zur Annotation der Testsätze mit erwarteten Ergebnissen und zur Ausführung und Kontrolle der Testläufe.

5.1. Ein Systemüberblick

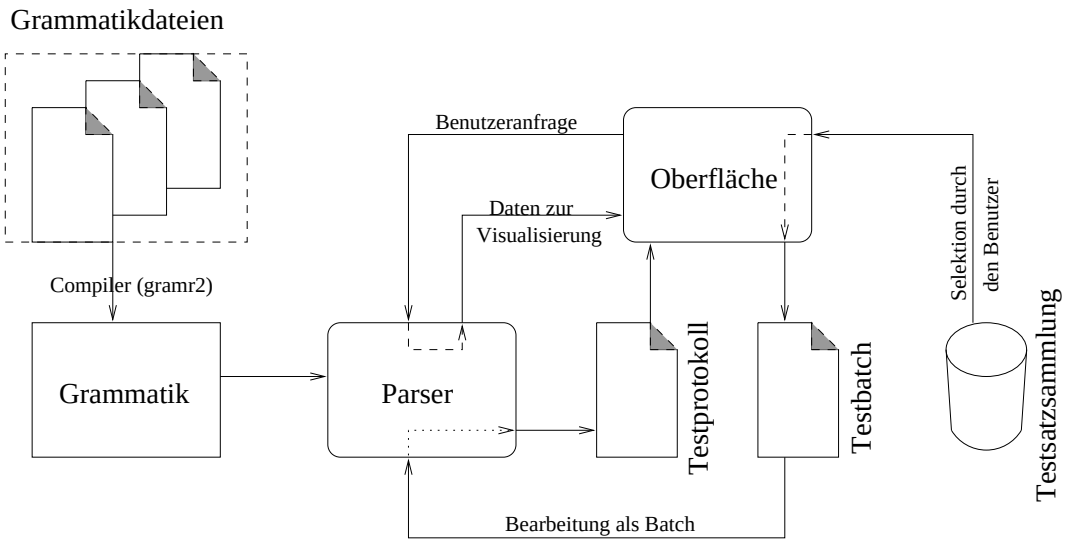


Abbildung 5.1.: Eine Systemskizze

Die Entwicklungsumgebung ist eine in Java geschriebene Oberfläche, der in einer Client-Server-Architektur mit dem in Kapitel 3 vorgestellten System kommuniziert. Die Visualisierung der Grammatik findet durch einen Dialog zwischen der Oberfläche und dem zugrundeliegenden System statt, das die

Grammatikdaten für die Darstellung aufbereitet. Will der Benutzer etwa eine Regel anzeigen lasen, dann erfragt die Oberfläche die Regelstruktur vom Parser und stellt sie entsprechend dar. So muß die Oberfläche niemals selbst die Grammatikdateien analysieren.

Die Testsatzmengen, mit denen der Systemtest durchgeführt werden soll, werden aus der Testsatzmenge heraus generiert und dann in einem Batchverfahren, d. h. ohne weitere Interaktion mit dem Benutzer, vom Parser bearbeitet. Das Testprotokoll wird dann wiederum von der Oberfläche analysiert und für die Darstellung aufbereitet. Durch diese Architektur ist auch ein kompletter Offlinebetrieb denkbar, in dem große Testsatzmengen für eine Verarbeitung erstellt und dann nachts auf einer rechenstarken Maschine bearbeitet werden. Das Protokoll kann dann zu einem späteren Zeitpunkt durch die Oberfläche ausgewertet werden.

In der Oberfläche können auch die Grammatikdateien editiert werden, diese werden dann auf Knopfdruck vom System übersetzt – bei Fehlern werden diese natürlich vom System angezeigt – und an den Parserprozess übergeben. Dieser Prozess bereitet die Daten dann widerum für die Anzeige in der Oberfläche auf. Das gesamte System ist in einer Unix-Umgebung implementiert, kann aber mit einigen Modifikationen auch in eine Client-Server-Archtektur umgewandelt werden bei der der Parserprozess weiterhin auf einem Unix-Server läuft, während die Oberfläche auf einem Arbeitsplatzrechner unter Windows oder einem anderen Betriebssystem ausgeführt wird. Damit ist dann eine echte Verteilung der Systemumgebung auf mehrere Rechner möglich, die mehreren Grammatikentwicklern einen einfachen gleichzeitigen Zugriff auf die gemeinsam genutzten Grammatikdaten erlaubt. So kann eine Entwicklungsumgebung realisiert werden, in der eine kooperative Grammatikentwicklung möglich ist. Durch die Integration von Versionskontrollmechanismen und gegenseitige Absicherung gegen ungewollte Modifikationen einer Datei durch einen anderen Entwickler – wie sie in Programmierumgebungen schon zum Standard gehört – kann damit eine effiziente parallele Arbeit ermöglicht werden.

5.2. Visualisierung der Daten

Die Visualisierungskomponente stellt die Komponenten des Systems in verschiedenen Fenstern dar. Es gibt Fenster zur Darstellung von

- Regeln und den zugehörigen Featurestrukturen,

- Fehlerstrukturen für die Koreferenzenzrücknahme und den zugehörigen Featurtestrukturen,
- Lexikoneinträgen und den zugehörgen Featurestrukturen,
- Wortanalysen und den zugehörigen Featurestrukturen, d. h. gegebenenfalls werden die Featurestrukturen des Stamms, des Suffixes und eventuell des Korrektureintrags kombiniert,
- Parsingergebnissen über Teilstrukturen mit der zugehörigen Top-Level-Featurestruktur.

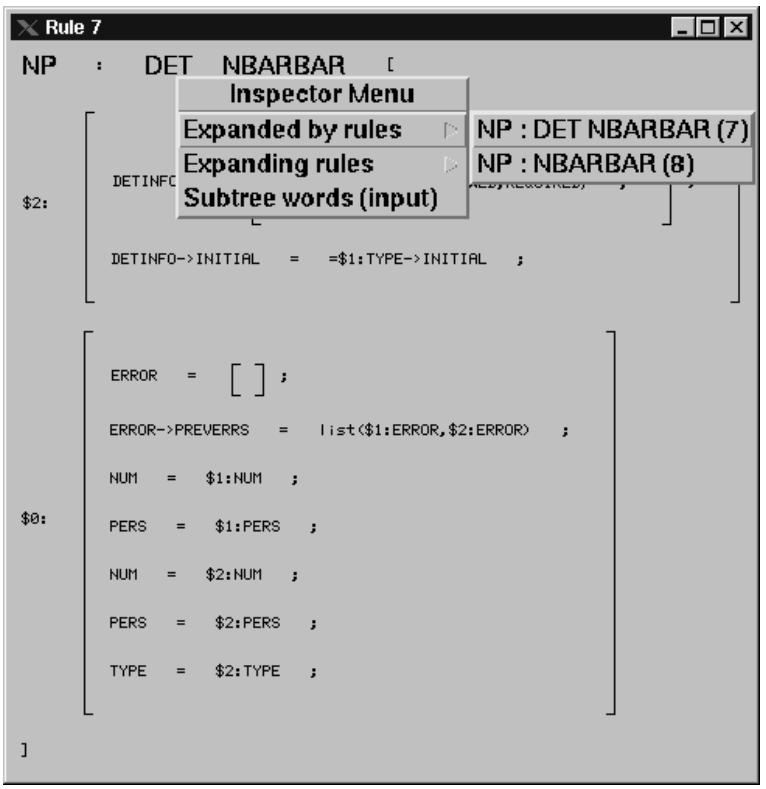


Abbildung 5.2.: Ein Fenster, das eine Regel zeigt. In dem Kontextmenü sind die Verweise zu anderen Regeln aufgelistet.

Bei der Anzeige von Parsingergebnissen und Wortanalysen kann zwischen den verschiedenen Ausgabefenstern hin- und hergeschaltet werden. Fast alle Elemente – z.B. Kategorienamen oder Worte – in den Fenstern sind anklickbar. Über *Kontextmenüs*, d. h. durch Auswahlboxen, die beim Drücken der rechten

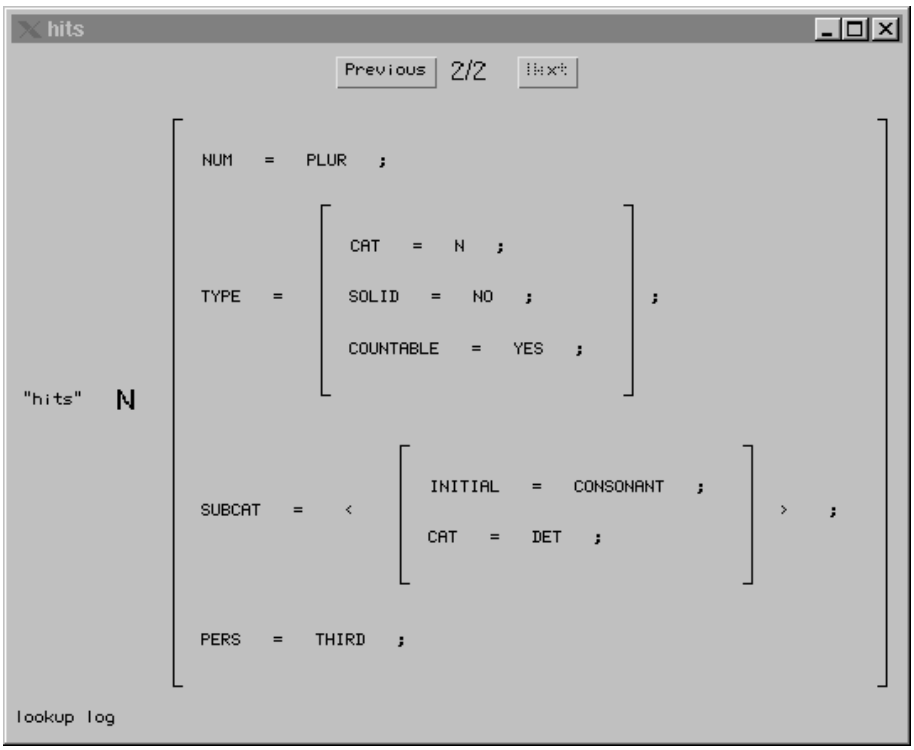


Abbildung 5.3.: Ein Fenster, das eine Wortanalyse zeigt.

Maustaste erscheinen, kann von einem Fenster ein neues Fenster geöffnet werden das weitere Informationen über dieses Element darstellt. Bei den nichtterminalen Symbolen können über ein solches Menü alle Regeln, die das Symbol expandieren bzw. durch die das Symbol erzeugt wird, in einem entsprechenden Fenster dargestellt werden. Dies ermöglicht ein schnelles und zielorientiertes Durchsuchen der Grammatik nach bestimmten Fehlerquellen. Bei einem terminalen Symbol kann über das Menü eine Detailanzeige eines zu dieser Kategorie passenden Lexikoneintrages ausgewählt werden. Möglichst alle Verbindungen, die in der Grammatik bestehen, werden über solche Kontextmenüs in den Fenstern angeboten. Neben dieser Darstellung der Daten kann auch eine Baumstruktur angezeigt werden. Diese wird entweder für einen Parse aus dem Analysefenster heraus erzeugt, d. h. sie stellt den zum Parse gehörenden Baum dar, oder der Baum wird von Hand durch direkte Manipulation durch die Maus in einem Baumeditor zusammengestellt. Dies erlaubt die Konstruktion eines Baumes anhand der vorhandenen Phrasenstrukturregeln. Für einen solche Baum kann durch das System überprüft werden, ob eine gültige Featurestruktur errechnet werden kann. Dazu wird für jeden Teilbaum überprüft, ob für ihn eine Featurestruktur gefunden werden kann. Dabei wird jeder Teilbaum

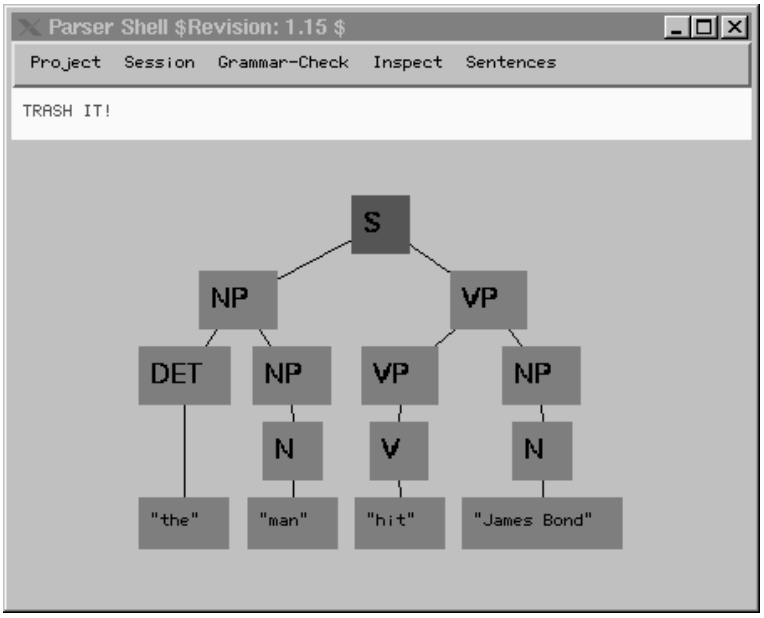


Abbildung 5.4.: Die Baumansicht einer teilweise (bei S) gescheiterten Analyse.

farblich – rot bei gescheiterten und grün bei erfolgreichen Teilbaumanalysen – markiert (vgl. Abbildung 5.4). Durch einen Mausklick auf das Symbol werden alle Featurestrukturen für diesen Teilbaum angezeigt. Über diesen Mechanismus ist die fehlerhafte Stelle bei einem gescheiterten Parse direkt ersichtlich. Durch die Featurestrukturen an den Konstituenten der gescheiterten Regeln und anhand eines Fehlerprotokolles zu der Analyse kann dann nachvollzogen werden, warum die Regel scheitert. Falls die Ursache in einer vorher benutzten Regel liegt, kann dies über die mit den Konstituenten verbundene Information in den tiefer im Baum liegenden Teilstrukturen nachgeprüft werden.

5.3. Definition der Testsatzmengen

Die Visualisierungskomponente ist vor allem ein Werkzeug, um bei einer nicht korrekt funktionierenden Satzanalyse den Fehler zu finden. Diese nicht korrekt analysierten Testsätze können wiederum über die Testsatzmenge ermittelt werden.

Zur Erstellung der Testsatzmenge und der notwendigen Annotationen ist ein spezielles Editorfenster vorhanden (vgl. Abbildung 5.6), in dem zu jedem Satz

die erwarteten und die unerwünschten Konstituenten eingetragen werden. Unerwünscht sind die Konstituenten, die bei einem Parsing nicht erkannt werden sollen. Bei dem inkorrekten Satz **the man storm the castle.* wäre die Konstituente `S` auf dem Bereich $0,5$ unerwünscht.

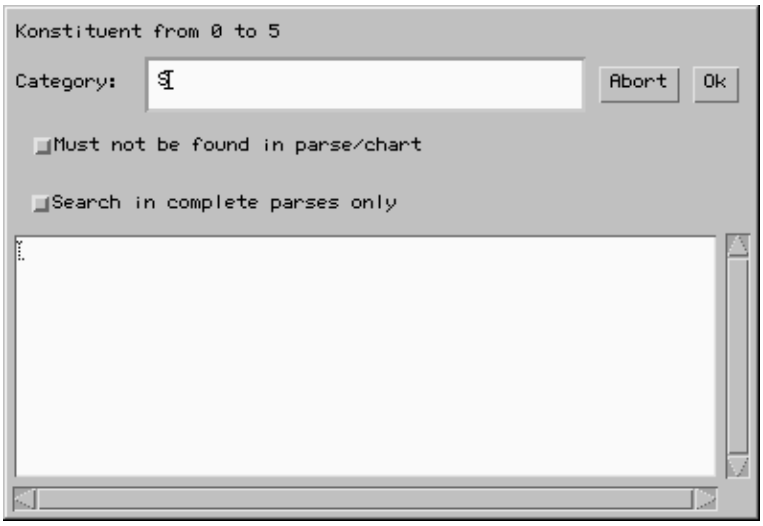


Abbildung 5.5.: Das Fenster für die Definition der Annotationen.

Alle Testsätze werden in einer Liste eingetragen, jeder einzelne Satz kann in dem abgetrennten Editierbereich mit Konstituenten annotiert werden. Für jede Konstituente wird der Bereich, den sie umfassen soll, mit der Maus in einer Tabelle markiert. Danach öffnet sich ein Fenster (vgl. Abbildung 5.5) in dem die Eigenschaften der Konstituente eingetragen werden. Die Eigenschaften sind:

- die Kategorie der Konstituente;
- die Forderung nach dem Vorhandensein bzw. Nichtvorhandensein der Konstituente;
- die Forderung nach dem Vorhandensein in einer abgeschlossenen Analyse. So kann überprüft werden, ob eine Konstituente in einer vollständigen Analyse als Teilstruktur benutzt wird;
- eine Featurestruktur, die mit dem Chartitem der Konstituente verträglich, d. h. unifizierbar, sein muß.

Falls ein Satz neu angelegt wird, kann der Parser, auch wenn der Satz noch nicht analysiert werden kann, schon Teilstrukturen des Satzes erkennen. Beim Anlegen von Testsätzen gibt es mit dem GET CHART-Knopf eine Schnittstelle zum Parser, über den schon erkannte Teilstrukturen automatisch in die Tabelle eingetragen werden. Es werden dabei der Bereich, die Kategorie und die erkannte Featurestruktur als Einträge in die Tabelle eingefügt. Diese Einträge können noch nachbearbeitet oder, falls sie überflüssig erscheinen, gelöscht werden. Dies erspart dem Entwickler Arbeit bei der Erstellung der Testsatzmengen. Falls ein Konvertierungsprogramm vorhanden ist, das eine Testsatzmenge aus einer bestehenden Testsatzsammlung extrahieren kann, lassen sich an dieser Stelle auch ganze Testsatzmengen aus einer konvertierten Beschreibung importieren.

5.4. Auswertung der Testläufe

Wenn die Testsatzliste erstellt ist, kann daraus eine Datei, *Testset* genannt, für den Parser generiert werden. Aus der Entwicklungsumgebung heraus wird dieses Testset an den Parser übergeben und das Testergebnis angezeigt. Die Anzeige erfolgt in einem Fenster, das dem zur Definition der Testsatzmengen gleicht. In der Testsatzliste sind nun alle Sätze markiert, bei denen das Analyseergebnis nicht mit dem erwarteten Analyseergebnis übereinstimmt. Für jeden Satz werden alle erwartungsgemäßen Konstituenten grün und alle nicht den Erwartungen entsprechenden Konstituenten rot hinterlegt. So kann auf einen Blick festgestellt werden, was bei einem Satz nicht korrekt ist.

Aus dieser Testsatzansicht heraus kann die Analyse des Satzen in die Parsingansicht übernommen werden. In dieser Ansicht ist dann durch die oben beschriebenen Hilfsmittel die Suche nach dem Fehler möglich.

Um die Entwicklung einer Grammatik zu verfolgen, gibt es neben der Auswertung eines Testlaufes auch noch die Möglichkeit, zwei Testläufe miteinander zu vergleichen. Bei diesem Vergleich werden automatisch alle Testsätze, deren Analyse bei den verschiedenen Versionen unterschiedlich war, verglichen. Alle korrekten Analysen, die in der Vergleichsdatei als inkorrekt markiert sind, werden dabei als Entwicklungsfortschritt betrachtet und entsrechend aufgelistet. Bei allen inkorrekten Analysen, die im Vergleichsset als korrekt markiert sind, wird eine Warnung an den Benutzer abgegeben. Durch diese Vergleichsfunktion ist es möglich jede Grammatikänderung systematisch auf unerwünschte Nebeneffekte zu untersuchen.

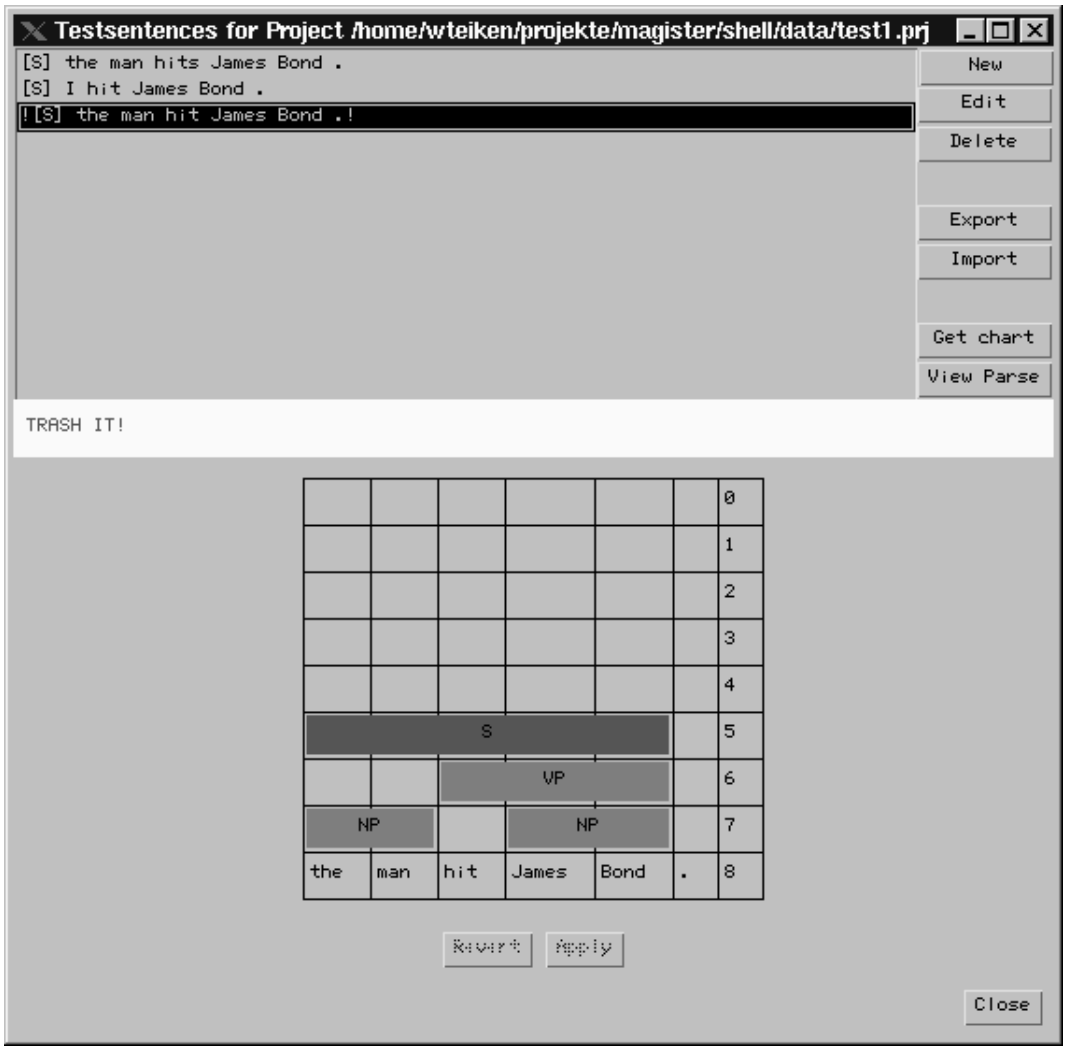


Abbildung 5.6.: Das Fenster für die Eingabe und Kontrolle der Testsätze.

6. Ausblick

Die im Rahmen dieser Arbeit vorgestellten Komponenten sollen die Arbeit eines Grammatikentwicklers vereinfachen. Bei der momentanen Version des Systems liegt dabei der Fokus auf der Erkennung und Analyse von Fehlern in der Grammatik.

Ein wesentlicher Bestandteil bei der Grammatikentwicklung ist natürlich die Erstellung der Grammatik in einem Editor. Um den Entwickler besser zu unterstützen, sollte das System um einen speziellen Grammatikeditor, der z. B. alle definierten Features kennt und deren Verwendung direkt bei der Eingabe prüft, erweitert werden. Mit einem solchen Werkzeug können viele Fehler schon bei der Erstellung der Grammatik vermieden werden, die im Moment noch aufwendig durch Testsatzmengen gefunden werden müssen. Der Einsatz von syntaxsensitiven Editoren – in der Programmentwicklung inzwischen zum Standard geworden – zeigt die Vorteile eines solchen Werkzeuges. Der Entwickler erhält schon bei der Erstellung des Quelltextes ein direktes Feedback über die syntaktische Korrektheit des Programmes. Die Funktionalität einiger Editoren geht soweit, daß die korrekte Verwendung einer Variablen oder Funktion schon bei der Nennung im Quelltext überprüft wird. Dies reduziert die durch Tippfehler entstandenen – aber oft syntaktisch nicht erkennbaren – Systemfehler, die ansonsten mit viel Mühe später gesucht werden müssen. Ein bekanntes Beispiel sind die Programmiersprachen, bei denen Variablen nicht deklariert werden müssen. In diesem Fall führt ein Tippfehler bei einem Variablennamen zu einer neuen Variablen, was natürlich nicht der Absicht des Programmierers entspricht. Diese Modifikationen auf der Bearbeitungsseite zusammen mit der bei der Architekturübersicht angedeuteten Verteilung des Systems auf mehrere Rechner wäre dies ein weiterer Schritt in Richtung einer integrierten Entwicklungsumgebung für Grammatiken, die gleichzeitig gute Rahmenbedingungen für kooperative Grammatikentwicklung unterstützt.

Ein weiterer Ansatz im Rahmen der Fehlersuche wäre es, durch maschinelles Lernen die konsistente Verwendung eines Features zu prüfen. Darüber könnte man in einer Grammatik ungewöhnliche Verwendungen eines Features, z. B. an

einer für dieses Feature unüblichen Position innerhalb einer komplexen Featurestruktur, aufspüren. Ich konnte im Verlauf der Arbeit allerdings keinerlei Hinweis darauf finden, ob diese Idee schon weiterverfolgt wurde.

Ein System, das all diese Funktionen in sich vereinigt, wäre dann eine wirkliche Hilfe bei der Erstellung von Grammatiken und Lexika. Durch eine Anbindung an bestehende Testsatzsammlungen wäre dann eine Mächtigkeit der Entwicklungsumgebung gegeben, wie sie in der Programmierung schon seit einiger Zeit, zumindest im professionellen Bereich, zum Standard geworden ist.

Ein weiterer Bereich, in dem dieses System eingesetzt werden könnte, ist die Lehre. Ein System, das eine grafische Interaktion mit einer Grammatik erlaubt, kann sehr schnell ein Gefühl für die Leistungsfähigkeit eines solchen Formalismus vermitteln. Vor allem die Möglichkeit, die Ergebnisse eines Parserdurchlaufs schnell wieder mit den benutzten Regeln in Verbindung zu bringen, kann das Verständnis für die Abläufe innerhalb eines Parsers fördern. Dies wäre eine sinnvolle Nutzung der Möglchkeiten, die der Computereinsatz – und in diesem Falle besonders die Nutzung des WWW mit seiner Client-Server orientierten Struktur – bietet. Daher wäre es interessant, die Reaktionen auf ein solches System in der Lehre zu testen und vielleicht Modifikationen in dieser Richtung zur besseren Unterstützung der Lehre vorzunehmen. Es gibt inzwischen einige Forschungsprojekte, die sich mit entsprechenden Ansätzen befassen und deren Ergebnisse Antworten auf die Frage nach dem Nutzen eines solchen Systems in der Lehre liefern können. Die technischen Möglichkeiten scheinen zunächst noch unbegrenzt, es gilt diese sinnvoll auszuschöpfen.

A. Notation

A.1. Verwendete Begriffe

Um Mißverständnisse zu vermeiden werde ich in diesem Abschnitt kurz einige Begriffe erläutern, die in der Arbeit verwendet werden:

Atom Ein *Atom* ist eine unzerlegbare Informationseinheit, z. B. ein einzelnes Symbol; im Gegensatz zu zerlegbaren Informationseinheiten wie Listen, die sich aus verschiedenen Atomen zusammensetzt.

Alphabet: Ein *Alphabet* ist eine Menge von Zeichen.

Batchverfahren Ein Batchverfahren ist die automatische Durchführung einer zu einem sog. *Batch* zusammengefassten Aufgabenmenge. Diese Durchführung wird ohne weitere Benutzerinteraktion durchgeführt, typischerweise wird der Systembenutzer nach der kompletten Abarbeitung des Batches über den Abschluss informiert und kann erst in diesem Augenblick anhand von Protokolldateien den Erfolg oder Mißerfolg der Programmläufe überprüfen.

Compiler: Ein *Compiler* ist ein Programm, daß eine Darstellungsform eines Systemes in eine andere überführt. Ein Beispel für ein Compiler sind Programmiersprachen, bei denen eine Programmquelle – z. B. in *C* – in ein ausführbares Programm übersetzt wird.

Constraint: Ein *Constraint* ist eine Bedingung, die mit einem Objekt verknüpft ist. Das Objekt muß dieses Constraint erfüllen, um gültig zu sein. Oft, allerdings nicht immer, sind Constraints logische Formeln.

Constraintset: Die konjunktive Verknüpfung mehrerer Constraints zu einer Menge nennt man *Constraintset*.

Deklarativ: Ein Formalismus ist *deklarativ*, wenn implementatorische Details, z. B. die Abarbeitungsreihenfolge von Anweisungen, keinen Einfluß haben. Programmiersprachen sind im Wesentlichen *imperativ*, da die Reihenfolge der einzelnen Anweisungen entscheidend für die Bedeutung ist[1].

Grammatik: Siehe Abschnitt 2.1.

Instanz: Eine *Instanz* ist eine Ausprägung eines Konzeptes. Ein realer Stuhl ist z. B. eine Instanz des Konzeptes „Stuhl".

Interface: Siehe *Schnittstelle*.

Imperativ: Ein Formalismus ist *Imperativ*, wenn jede Anweisung in einer vorgegebenen Reihenfolge abgearbeitet wird. Daher verändert die Vertauschung zweier Anweisungen im allgemeinen die Semantik des Systems.

Lesart: Eine *Lesart* eines Wortes ist eine mögliche Wortanalyse. Man unterscheidet zwischen syntaktischen Lesarten, wie bei *hit* als Verb oder Nomen, und semantischen Lesarten, wie bei *bank* als Möbel oder Geldinstitut.

Modul: Ein *Modul* ist ein funktionaler Bestandteil eines Systems, das in einem gewissen Rahmen unabhängig vom Rest des Systems arbeitet. Ein modularisiertes System besteht aus einzelnen Komponenten, die über *Schnittstellen* miteinander kommunizieren.

NLP: Eine Abkürzung für *Natural Language Processing*, also die Verarbeitung natürlicher Sprache.

Phrasenstrukturskelett: Das *Phrasenstrukturskelett* ist die kontextfreie Regel einer Unifikationsgrammatik, an das Regelconstraints gebunden werden.

Regel: Siehe Abschnitt 2.1.

Schnittstelle: Eine *Schnittstelle* ist die Definition des Datenaustausches zwischen zwei Modulen. Eine Schnittstellenbeschreibung umfaßt neben den

[1] *Prolog* geht den Weg, möglichst weitgehend deklarativ zu sein. Es gibt zwar eine vorbestimmte Abarbeitungsreihenfolge, d. h. eine imperative Komponente, die ein Programm allerdings möglichst wenig ausnutzen sollte. Ein Programm, das sich nicht auf den imperativen Bestandteil von *Prolog* verläßt, ist direkt äquivalent zu einer Formelmenge in einem Deduktionssystem, d. h. deklarativ. Bei Ein-/Ausgabeoperationen ist die Forderung nach Deklarativität allerdings nicht sinnvoll, da es hier, um eine sinnvolle Ausgabe zu erzeugen, in der Regel auf die Reihenfolge der Operationen ankommt.

Funktionsdefinitionen auch die Festlegung aller Datenstrukturen, die zum Datenaustausch benutzt werden können. Es wird für eine Schnittstelle auch oft der Begriff *Interface* verwendet.

Semantik: Mit dem Begriff *Semantik* wird die Bedeutung einer Struktur bezeichnet. Damit kann z. B. die Bedeutung eines Satzes oder die Bedeutung eines Programmes gemeint sein.

Stack: Ein *Stack* bezeichnet eine Datenstruktur, die zur Speicherung von Objekten dient. Der Stack kann mit einem Stapel verglichen werden, bei dem man ein Objekt oben ablegen (*push*) oder das oberste Element herabnehmen kann (*pop*). Außerdem kann das oberste Element ausgelesen werden, ohne es vom Stack zu nehmen. Daher kann immer nur auf das zuletzt gespeicherte Element zugegriffen werden.

Token: Ein *Token* ist eine Instanz eines *Types*. Es kann z. B. die Verwendung eines Konzeptes sein.

Type: Ein *Type* ist das Konzept, das hinter einem Bezeichner steht.

Unifikationsgrammatik: Siehe Abschnitt 2.7

A.2. Konstanten und Symbole

Σ	Das Symbol $\Sigma = \{a_1, a_2, \dots\}$ wird als Eingabealphabet bezeichnet; es stellt die Menge aller Symbole dar, die vom Parser verstanden werden.
ε	ist die leere Kette, also die Kette aus 0 Elementen.
$\mathcal{CCS}$	ist die Menge aller konsistenten Featurestrukturen.

A.3. Operatoren

$\mathbb{N}_0$	ist die Menge aller nicht-negativen, ganzen Zahlen, $\mathbb{N}_0 = \{0, 1, \dots\}$.
*	Der Operator * liefert zu einer Menge Σ die Menge aller (endlichen) Sequenzen aus Σ, also $\Sigma^* = \{a_1 a_2 \cdots a_n \mid a_1, \dots, a_n \in \Sigma, n \in \mathbb{N}_0\}$.
$\mathrm{dom}(f)$	bezeichnet die Domäne einer Funktion f, also die Menge aller Werte, für die diese Funktione definiert wird, z. B. die Menge aller x, für die ein $f(x)$ existiert.
$\mathrm{im}(f)$	bezeichnet die Bildmenge einer Funktion f, also die Menge aller Werte, die diese Funktion annehmen kann.
$\Rightarrow$	ist der Ableitungsoperator (vgl. Abschnitt 2.1).
$\Rightarrow_{rm}$	steht für die Rechtsableitung, eine Ableitung bei der immer das am weitesten rechts stehende Terminal ersetzt wird.
$\Rightarrow^*$	ist eine Ableitungssequenz, d. h. die iterierte Anwendung des Ableitungsoperators.
$\wp(X)$	steht für die Menge aller Teilmengen von X, $\wp(X) = \{A \subseteq X\}$ (Potenzmenge).
$\subseteq_{fin}$	bezeichnet eine endliche Teilmenge einer Menge.
$\wp_{fin}(X)$	ist die Menge aller endlichen Teilmengen von X, $\wp_{fin}(X) = \{A \subseteq_{fin} X\}$.

A.4. Strukturen

Definition A.4.1 (Verband) *Es sei X eine Menge und $\leq$ eine partielle Ordnung auf X. Die Struktur $(X, \leq)$ ist ein Verband, wenn*

1. *es Elemente $\top, \bot \in X$, genannt Top und Bottom, gibt so daß gilt: $\bot \leq x \leq \top \; \forall x \in X$.*
2. *Für zwei Elemente $x, y \in X$ gibt es eine kleinste obere Schranke $x \sqcup y$ mit*
   (a) $x \leq x \sqcup y \wedge y \leq x \sqcup y$
   (b) $(\forall z(x \leq z \wedge y \leq z) \rightarrow x \sqcup y \leq z)$.
3. *Für zwei Elemente $x, y \in X$ gibt es eine größte untere Schranke $x \sqcap y$ mit*

(a) $x \sqcap y \leq y \wedge x \sqcap y \leq y$

(b) $(\forall z(z \leq x \wedge z \leq y) \rightarrow z \leq x \sqcup y)$.

B. Die Syntax des Formalismus

B.1. Der Grammatikformalismus

Das System stellt einen Formalismus zur Verfügung, in dem die Grammatiken beschrieben werden. Die Grammatikbeschreibung wird in einer *ASCII*-Datei hinterlegt und dann von einem speziellen Compiler, *gramr2*, in das interne Format des Parsers umgewandelt. Aus Geschwindigkeitsgründen verwendet der Parser numerische Repräsentationen der Featurestrukturen; diese werden vom Grammatikcompiler generiert.

Bei der Übersetzung wird soweit wie möglich die Konsistenz der Grammatik geprüft. Dies betrifft sowohl die Syntax der Regeln, als auch Teile der Semantik. Dadurch werden viele Fehler schon vor dem Test der Grammatik beim Parsen beseitigt.

B.1.1. Deklarationen

Deklaration eines Features

Jedes Feature, das in der Grammatik benutzt wird, muß im Deklarationsteil eingeführt werden. Da das System verschiedene Featuretypen kennt, muß bei der Deklaration eines Features auch einer der später beschriebenen Typen festgelegt werden. Die Deklaration der verschiedenen Featuretypen erfolgt mit den folgenden Anweisungen:

Standardfeatures `(#attrdef` *name* $val_1 \ldots val_n$ `)`

Dabei ist *name* der Name des Features und die val_i sind die verschiedenen Werte, die das Feature annehmen kann.

```
(#attrdef PERS FIRST SECOND THIRD)
```

Stringfeatures (#strdef *name*)
haben einen vordefinierten Wertebereich, dieser muß daher nicht angegeben werden.

```
(#strdef INFO)
```

Boolean (#boolean *name*)
sind ebenfalls Features mit einem vorgegebenen Wertebereich

```
(#boolean DET_PRESENT)
```

Featurelisten (#attrlistdef *name* val_1 ... val_n)

```
(#attrlistdef ATTRIBUTES BIG SMALL MEDIUM
              RED GREEN BLUE YELLOW BLACK WHITE)
```

Stringlisten (#strlistdef *name*)

```
(#strlistdef INFOS)
```

Zähler (#counterdef *name*)

```
(#counterdef ERRORCOUNT)
```

Pfadefeatures (#pathdef *name*)

```
(#pathdef DESTPATH)
```

Listen (#listdef *name*)

```
(#listdef SUBCAT)
```

Subfeature (#subfeature *name*)

```
(#subfeature SYN)
```

Die Featuretypen und die wichtigsten Eigenschaften auf einen Blick:

Typ	Schlüsselwort	Wertebereich	Unifikationsoperation
Standard	`attrdef`	Anzugeben	Schnittmenge
Strings	`strdef`	Alle Strings	Schnittmenge
Boolean	`boolean`	TRUE, FALSE	Schnittmenge
Featurelisten	`attrlistdef`	Anzugeben	Vereinigung
Stringliste	`strlistdef`	Alle Strings	Vereinigung
Counter	`counterdef`	Alle Integer	Gleichheit
Pfad	`pathdef`	Pfade	Gleichheit
Subfeature	`subfeature`	Featurestrukturen	elementweise Rekursiv
Liste	`listdef`	Listen	elementweise

Funktionsdeklarationen

Der Grammatikformalismus bietet die Möglichkeit, Funktionen für spezielle Koreferenzen einzusetzen. Um Laufzeitfehler[1] zu vermeiden, müssen alle Funktionen vor ihrer Anwendung deklariert werden.

Der Rückgabetyp der Funktion wird durch ein Schlüsselwort bestimmt, die Schlüsselworte für die verschiedenen Rückgabewerte sind in der Tabelle aufgelistet:

Typ	Deklaration
Standard	`attrfun`
String	`strfun`
Boolean	`booleanfun`
Featureliste	`attrlistfun`
Stringliste	`strlistfun`
Counter	`counterfun`
Subfeature	`subfeaturefun`
Pfad	`pathfun`
Liste	`listfun`
Laufzeit	`varfun`

So wird z. B. eine Funktion FUNKTION1, die Standardfeatures zurückgibt, mit der Anweisung

```
(#attrfun funktion1)
```

[1] Laufzeitfehler meint hier einen Fehler in der Grammatik, der beim Parsing auftritt und nicht behoben werden kann. Dies kann z. B. der Versuch sein, in der Grammatik eine Funktion aufzurufen, die nicht definiert ist.

angelegt.

Der Typ `varfun` kann für Funktionen genutzt werden, bei denen der Rückgabewert zur Compilezeit der Grammatik nicht bekannt ist. Dies ist z. B. bei der Listenzugriffsfunktion `litem` (vgl. Anhang B.2) der Fall. Diese Funktionen müssen selbst den korrekten Rückgabewerttyp prüfen und ggf. einen Laufzeitfehler auslösen, d. h. sie müssen einen Systemfehler melden, falls der Rückgabetyp nicht dem des Zielfeatures entspricht.

B.1.2. Featurestrukturen

Die Featurestrukturen werden als Constraintsets angegeben, die aus direkten Wertangaben und Pfadkoreferenzen bestehen. Die Wertangaben sind Anweisungen der Form

Attribut = *Wert*;

wobei *Attribut* der Name eines Attributes und *Wert* je nach Attributtyp ein unterschiedlicher Wert sein kann. Die einzelnen Anweisungen werden durch eckige Klammern („[“ am Anfang, ein „] ;“ am Ende) zusammengefaßt. Dabei gibt es folgende Formen:

Standardfeatures können als Werte entweder einzelne Atome oder Mengen von Atomen haben:

```
NUM=SING;
NUM={SING, PLUR};
```

Stringfeatures können einen oder mehrere Strings als Wert besitzen. Die Strings sind dabei immer in doppelte Anführungszeichen eingefaßt:

```
INFO="Inkorrekte Form";
INFO={"Inkorrekte Form", "Falsche Position"};
```

Booleanfeatures besitzen genau drei mögliche Werte:

```
COUNTABLE=YES;
COUNTABLE=NO;
COUNTABLE={YES, NO};
```

Zähler haben als Werte immer einzelne Zahlen:

```
COUNTER=1;
COUNTER=-2;
```

Pfadfeatures können nur Pfade aufnehmen:

```
DESTPATH=SYN->AGR->NUM;
```

Featurelisten entsprechen syntaktisch den Standardfeatures:

```
ATTRIBUTES=GREEN;
ATTRIBUTES={YELLOW, BIG};
```

Stringlisten `COMMENTS="Kommentar 1";`

```
COMMENTS={"Kommentar 1", "Kommentar 2"};
```

Listen können beliebige Feature-Wertpaare aufnehmen; dabei gibt es jedoch einige Sonderformen, die in Anhang B.1.6 näher erläutert werden.

```
SUBCAT=<TYPE=[CAT=N;], TYPE=[CAT=N; SOLID=YES;]>;
```

Subfeature haben komplette Featurestrukturen als Wert.

```
ERROR = [
  INFO = "Agreementfehler";
  LAST= [
    ...
  ];
];
```

Dabei gibt es die Sonderform, alle Pfade der Subfeature einzeln aufzulisten. Obige Featurestruktur ist äquivalent zu:

```
ERROR->INFO = "Agreementfehler";
ERROR->INFO->LAST= [
    ...
];
```

Damit lassen sich in einigen Fällen Featurestrukuren übersichtlicher notieren.

Dabei kann natürlich die Zuweisung auch durch eine Funktion erfolgen, die den korrekten Rückgabetyp für das Feature liefert:

```
INFO=strcat("Text, " aus ", "einzelteilen");
```

Die dritte Möglichkeit für Constraints sind Pfadkoreferenzen, bei denen auf der linken Seite des Gleichheitszeichens ein Feature und auf der rechten Seite eine Pfadangabe steht (vgl. Anhang B.1.4). Dies ist allerdings nur bei Featurestrukturen die zu Regeln oder Fehlerbeschreibungen gehören erlaubt.

Eine weitere Möglicheit des Parsers ist die Auswahl des linken Features für die Unifikation durch eine Funktion. Die benutzte Funktion muß in diesem Fall allerdings einen Pfad als Wert zurückliefern:

```
evalPath(NP:, #DESTPATH) = NP:AGR;
```

Diese Anweisung führt eine Koreferenz zwischen dem Feature, das in der Position steht, die durch den Pfad des Features DESTPATH der Komponente `S` der Regel gegeben ist, und dem Feature AGR in der Komponente `NP` durch. So lassen sich indirekte Koreferenzen, wie sie z. B. in der *LFG* benutzt werden, durchführen[2].

Neben diesen expliziten Koreferenzen können natürlich durch die Anweisung `#use` zusätzlich noch Templates eingebunden werden. Neben diesen Koreferenzen gibt es noch die Sonderform `ANY = ...`, die für Funktionen gedacht ist. Damit können Funktionen aufgerufen und das Funktionsergebns ignoriert werden.

Wenn eine Koreferenz optional ist (vgl. Anhang B.1.8), dann wird die Regel evtl. auch dann angewendet, wenn die Koreferenz nicht erfüllbar ist. In diesem Falle wird die Regelanwendung nicht abgebrochen, sondern die Koreferenzforderung wird *zurückgenommen* (vgl. Abschnitt 3.4.1). Die Syntax für optionale Koreferenzen lautet: `<koreferenz> #opt <name> <nummer>`,

es ist also eine *normale* Koreferenz erweitert um eine Kennung. Der Name ist der einer Fehlerstruktur (vgl. Angang B.1.8), die definiert sein muß. Die *Nummer* ermöglicht es, eine Metaregel mehrfach zu benutzen, aber trotzdem über die *Asserts* eine bestimmte Kombination an Koreferenzen fordern zu können, da auch bei den *Asserts* diese Nummer mit angegeben wird. Wird keine Nummer angegeben, dann wird 0 als Default angenommen.

[2] Die Notation in der *LFG* für diesen Ausdruck wäre (↑(↓ DESTPATH))=(↓ AGR)

Um bei diesen Rücknahmen unsinnige Kombinationen von Auslassungen zu vermeiden, können bestimmte Bedingungen für solche Koreferenzrücknahmen als *Asserts* (vgl. B.1.8) vorgegeben werden. Damit kann das System u.U. deutlich effizienter mit den Alternativen umgehen.

B.1.3. Bezeichner und Strings

Bezeichner sind Zeichenfolgen, die dem folgenden regulären Ausdruck genügen:

```
[A-Za-z0-9]+([A-Za-z0-9_\-]*[A-Za-z0-9]){0,1} .
```

Bezeichner werden benutzt, um Konstrukte zu benennen, z. B. Features, Values und Funktionen.

Strings (sowohl als Values für Stringfeatures als auch für die Einträge im Lexikon) werden in doppelte Anführungszeichen[3] eingeschlossen. Um ein Anführungszeichen in einem String darzustellen, wird dieses durch einen \ (Backslash) geschützt[4]. Der Backslash ermöglicht es, spezielle Zeichen wie einen Zeilenvorschub oder aber die Anführungszeichen in einem String zu kodieren. Es gibt folgende Sequenzen:

- `\"` Doppelte Anführungszeichen,
- `\\` Backslash,
- `\n` Zeilenvorschub,
- `\t` Tabulator.

B.1.4. Pfade

Pfade sind Ausdrücke der Form Bezeichner$_1$ `->` ... `->` Bezeichner$_n$, dabei muß jeder Bezeichner ein gültiger, d. h. definierter, Featurename sein. Neben dieser syntaktischen Einschränkung gibt es die semantische Einschränkung, daß alle Bezeichner bis auf Bezeichner$_n$ Feature des Typs Subfeature benennen sollten.

Oft muß neben dem Pfad auch die Featurestruktur, für die der Pfad gültig ist, angegeben werden. Dies geschieht durch `$`n`:` vor dem Pfad oder aber durch den Namen der Konstituente, zu dem die Featurestruktur gehört. Dies ist aber

[3] Für die Anführungszeichen wird auch der Terminus *Quote* verwendet; das Einschließen von Zeichenketten in Quotes wird auch als *quoting* bezeichnet.

[4] Dieser Schutz eines Zeichens wird oft durch den Begriff *Escaping* oder *Escape-Sequenz* ausgedrückt.

nur dann möglich, falls der Featurename eindeutig ist[5]. Das n gibt dabei die Nummer der Konstituente an, zu der die Featurestruktur gehört. Als besondere Form der Referenz ist dabei auch ein leerer Pfad erlaubt; dann wird auf die komplette Featurestruktur der Konstituente referiert.

```
$0: SYN->AGR->NUM;
HEAD: SYN->AGR->NUM;
```

B.1.5. Funktionsaufrufe

Funktionsaufrufe sind Ausdrücke der Form

funktor`[`*Strukturbezeichner*`:]`(p_1, p_2, ... , p_n).

Der Funktor ist der Name der Funktion, die mit den Parametern p_n aufgerufen wird. Der *Strukturbezeichner* ist der Ausdruck, der wie bei einem Pfad die Featurestruktur auswählt, deren Chartinformation an die Funktion übergeben werden soll. Dies ist bei Funktionen wie `getPosition` wichtig, die die Satzposition einer Konstituente ermitteln. Wird dieser Index ausgelassen, dann wird als Defaultwert die Chartinformation zum Regelkopf übergeben.

Die Parameter p_i sind Ausdrücke die jeweils einer der folgenden Formen entsprechen müssen:

- Eine Zahl: `23`
- Ein einzelner String: `"Info"`
- Eine Liste von Strings, die in geschweifte Klammern eingefaßt sind: `{"Info 1", "Info 2"}`
- Ein Pfad: `#AGR->NUM`
- Eine Liste (vgl. Anhang B.1.6):
  `<TYPE=[CAT=N;], TYPE=[CAT=N; SOLID=YES;]>`
- Ein verschachtelter Funktionsaufruf: `strcat("A", "B")`
- Eine Wertkoreferenz (vgl. Anhang B.1.2): `NUM = SING`

[5] Um einen Konstituentennamen eindeutig zu machen, kann man ein `.`n anhängen, das lediglich zur Identifikation dient. So kann z. B. die Regel `NP : NP.1 VP` formuliert werden, in der die beiden beteiligten `NP`en durch die angehängte Nummer unterschieden werden können. Das System betrachtet natürlich auch `NP.1` als eine normale `NP`.

- Ein Featurepfad, der ausgewertet wird: `S:SYN->AGR`

Wenn ein Parameter in einer der ersten fünf Formen angegeben wird, dann wird aus dem Parameter ein Attribut-Wertpaar generiert, indem als Attribut ein generischer Zähler oder ein generischer String-, Pfad- oder Listenfeature – abängig vom Typ – gewählt wird. Daher ist dieses Verfahren nur möglich, wenn es keine featurespezifischen Werte sind, wie z. B. bei Standardfeatures. Es können auch Funktionen verwendet werden, die als Resultat ein solches Atom liefern; auch diese werden in der oben angegebenen Form behandelt. Bei den übrigen Varianten wird das angegebene Feature-Wertpaar übergeben.

```
strcat("Info 1", "Info 2");
strcat("Es wird der Numerus ", NUM=SING, " benoetigt");
getPosition[S]();
```

Im zweiten Beispiel wird automatisch der Wert SING in einen String konvertiert bevor die Funktion strcat ihn benutzt.

B.1.6. Listen

Die Syntax für eine Liste ähnelt der Syntax von Funktionsaufrufen; eine Liste ist ein Ausdruck der Form

$<e_1, e_2, \ldots, e_n>$,

die e_i entsprechen in Syntax und Semantik den p_i bei den Funktionsaufrufen. Dabei sind allerdings bei Listen weder Funktionsaufrufe noch Pfadkoreferenzen erlaubt.

B.1.7. Kommentare

An jeder beliebigen Stelle kann ein Kommentar eingefügt werden. Ein Kommentar ist ein Textabschnitt der Form `/* <Kommentar> */`, der Text zwischen den Zeichensequenzen `/*` und `*/` wird dabei ignoriert. Ein solcher Kommentar kann sich auch über mehrere Zeilen erstrecken.

Die zweite Art der Kommentare ist ein `//` am Anfang des Kommentars. Alles hinter diesem *Doppelslash* bis hin zum Zeilenende wird als Kommentar angesehen und vom System ignoriert.

B.1.8. Die Grammatik

Fehlerstrukturen

Die *Fehlerstrukturen* sind Metaregeln, die im Falle des Scheiterns einer optionalen Koreferenz angewendet werden. Wenn eine Koreferenz scheitert, dann sollte die Information über die Ursache des Scheiterns (z.B. ein Agreementfehler) in der Featurestruktur zu dem Parse gespeichert werden. Dies geschieht mittels der Metaregeln, die die Fehlerinformation in die aktuelle Featurestruktur einfügen. Die Syntax für die Fehlerstruktur ist:

```
#relax <name> <basisfeature> -> <listenfeature>
[ <koreferenzen> ]
```

Asserts

Asserts oder auch *Assertionen* sind Ausdrücke, die angeben welche Kombinationen von Regelrelaxionen erlaubt sind. Im Moment sind nur Asserts erlaubt, die eine Oderverknüpfung ausdrücken, also Ausdrücke der Form:

```
#assert(<name_1> <nr_1>|<name_2> <nr_2>|...|<name_n> <nr_n>) .
```

Dieser Ausdruck garantiert, daß in einer Regel, die eine oder mehr der Regelrelaxionen $\texttt{name}_1$ bis $\texttt{name}_n$ verwendet, mindestens eine der aufgeführten Koreferenzen auch erfolgreich ist. Über diesen Mechanismus kann die Gesamtanzahl der Relaxionen pro Regel reduziert werden. Die so definierten Asserts gelten für alle Regeln in einem Modul. Zur Bedeutung der Nummern vgl. B.1.2. Wird keine Nummer angegeben, dann wird 0 als Default angenommen.

Templates

Im Regelabschnitt können neben den Regeln auch sog. *Templates* definiert werden. *Templates* sind abfolgen von Koreferenzen, die unter einem Namen abgespeichert und in beliebigen Regeln unter diesem Namen verwendet werden können. Sie verhalten sich ähnlich wie Makros in Programmiersprachen. Die Syntax für ein *Template* lautet:

```
#template <name>
[
  <koreferenzen>
]
```

In einer Regel kann das *Template* dann mit `#use <name>` verwendet werden; die Semantik ist identsch zu der Regel, in der alle Koreferenzen des *Templates* direkt eingetragen sind. *Templates* können wiederum schon definierte *Templates* enthalten.

Es gibt spezielle *Templatenamen* – z. B. `foo` – der beliebig oft überschrieben werden kann. Dies ist für Situationen gedacht, in denen man Koreferenzabfolgen in aufeinanderfolgenden Regeln mehrfach verwenden will, aber keine weitere Verwendung im Rest der Grammatik erfolgt. Anstatt hier jeweils einen eindeutigen Namen zu vergeben, kann einfach der Name `foo` mehrfach verwendet werden. Falls auf die letzte Featurestruktur als Template zugegriffen werden soll, kann dies auch über das spezielle Template `last` geschehen, was zur Übernahme der komplette Featurestruktur (incl. aller Templates und Defaults) des letzten Lexikoneintrages führt.

Weitere, spezielle *Templatename* sind `init1`, `init2`, dots. Wenn einer dieser speziellen Templates definiert ist, dann wird bei allen nachfolgenden Regeldefinitionen mit n Konstituenten das entsprechende Template am Anfang von `$0` eingefügt.

Regeln

Die Regeln haben die Grundform:

```
<HEAD> : <NT_1>...<NT_n> &<flags> *<function> #<groups>
         <asserts>
         [
           <koreferenzen>
         ]
```

Jede Regel besteht aus einem Phrasenstrukturskelett(PSG), das die Kategorien der Regel angibt, sowie aus optionalen Parametern und den zum PSG gehörenden Koreferenzen. Es kann aber auch eine leere Koreferenzliste angegeben werden.

Die Parameter definieren verschiedene zusätzliche Eigenschaften für die Regel, die deren Anwendung während des Parses beeinflussen. Die einzelnen Optionen werden jeweils durch ein Symbol `&`, `,` `#` eingeleitet; außerdem müssen sie in der angegebenen Reihenfolge erscheinen. Sie können aber unabhängig von einander angegeben oder aber weggelassen werden.

Bei der Syntax für Konstituentennamen gibt es noch die Besonderheit, daß das Zeichen . (Punkt) in dem Bezeichner benutzt werden darf. Wenn dieses Zeichen auftritt, dann trennt es einen Konstituentenbezeichner von einem Affix, das die Konstituente mit einem eindeutigen Namen versieht[6]. Wenn jede Konstituente einen eindeutigen Namen hat, dann kann auf die Featurestrukturen auch über den eindeutigen Namen der Konstituente statt über ihren Index in der Regel zugegriffen werden. Dies ist vor allem dann notwendig, wenn durch einen Präprozessor einzelne Konstituenten eliminiert werden (zur Implementation von optionalen Konstituenten), denn dann ändert sich der Index aller anderen Konstituenten.

Die Flags sind binäre Masken, die beim Parse mit einer vorgegebenen Maske verknüpft[7] werden. Die einzelnen Flags (numerische oder im Deklarationsteil vordefinierte Symbole) werden schon bei der Grammatikübersetzung miteinander verknüpft[8]. Wenn mehrere Flags vorhanden sind, dann entspricht das einer Regel, die für mehrere Satztypen verwendet werden kann.

Die Funktion dient zur dynamischen Kontrolle des Parses. Sie kann aufgrund der in den Konstituenten vorliegenden Information[9] die Anwendung dieser Regel verhindern. Über diesen Mechanismus können ungewünschte Parsehypothesen vermieden werden. Wenn die Funktion die Regel nicht zur Anwendung freigibt, dann reagiert der Parser, als sei eine notwendige Koreferenz fehlgeschlagen; es wird keine der für die Regel eingetragenen Koreferenzen ausgeführt.

Gruppen dienen dazu, Regeln miteinander zu verbinden. In einigen Fällen soll eine Regel nur dann angewendet werden, wenn eine andere Regel gescheitert ist. Über diesen Mechanismus können dann verschiede Regeln alternativ zueinander angewendet werden. Dies kann sowohl ungewünschte Hypothesen vermeiden als auch Rechenzeit einsparen, indem bestimmte, zum scheitern verurteilte Hypothesen von vornherein ausgeschlossen werden. Dazu stellt das System mehrere voneinander getrennte Mechanismen zur Verfügung.

[6] z.B. bei `VP : VP PP` wäre hier `VP : VP.Kopf PP`

[7] Es findet eine binäre *Undverknüpfung* der Regelmaske mit einer beim Parse vorgegebenen Maske statt. Ist das Ergebnis der Verknüpfung 0, dann wird die Regel ignoriert.

[8] Alle Teilflags werden über ein logisches *Oder* verbunden. Sind keine Flags angegeben, so wird als Defaultwert *(Hex)FFFF* verwendet, die Regel wird also immer beachtet.

[9] z.B. die bisherige Fehleranzahl

Der erste Mechanismus bildet für jeden verwendeten Regelnamen eine Regelmenge. Die erste Regel dieser Menge, die in der Grammatik auftaucht, ist die *Masterregel* dieser Menge. Wenn die Masterregel auf einem bestimmten Teil des Inputs benutzt wurde, dann wird keine weitere Regel[10] dieser Regelmenge auf diesem Abschnitt angewandt. Wenn beim Input *Der Mann geht über die Straße.* die NP von der Position 0 bis zur Position 2 durch die Masterregel erkannt wurde, dann wird keine weitere Regel der zugehörigen Regelmenge auf dem Teilinput 0...2 angewendet. Über diesen Mechanismus kann eine Subsumptionshierarchie Über die Regeln definiert werden, die immer die speziellste Regel anwendet, die in einem gegebenen Kontext anwendbar ist. Die Blockade der Regelgruppe wird jedoch immer nur durch die *Masterregel* ausgelöst; bei differenzierteren Hierarchien müssen die Gruppen also wiederum in Teilgruppen aufgegliedert werden. Für jede Hierarchiestufe werden somit 2^n Gruppen benötigt, wobei n die Hierarchietiefe angibt. Alle Regeln werden anfangs in die Chart eingetragen. Wenn die Kanten der Regeln vom aktiven in den passiven Status übergehen wird überprüft, ob sie durch die Masterregel blockiert werden.

Erweiterte Masterregeln erweitern diese starre Struktur um Regeln, die zwar durch die *Masterregel* blockiert werden, ihrerseits aber alle *Slaveregeln*[11] blockieren. Die *erweiterten Masterregeln* blockieren sich jedoch nicht gegenseitig. Durch diese Erweiterung können Regelabhängigkeiten modelliert werden, die nicht direkt einer Subsumptionshierarchie entsprechen. Syntaktisch werden die *erweiterten Masterregeln* durch ein + (Pluszeichen) vor dem Gruppennamen gekennzeichnet.

Das Problem bei dieser Art der Regelverknüpfungen ist jedoch, daß sie stark von der Regelreihenfolge und teilweise auch vom Ablauf des Parsings abhängen.

Erweiterte Slaveregeln lösen dieses Problem, indem sie in dem Augenblick, wenn eine Regelanwendung fehlschlägt, als Alternative zu diesen Regeln nachträglich in die Chart eingefügt werden. Dies kann auch rekursiv passieren, indem eine *erweiterte Slaveregel* wiederum eigene *erweiterte Slaveregeln* besitzt. Dies Verfahren garantiert eine Unabhängigkeit vom Parsing, weil die Regeln direkt als Alternativen zur ursprünglichen Regel angewendet werden. Es können über diesen Mechanismus allerdings nur strikte Hierarchien modelliert werden, allerdings können parallel zu diesem Mechanismus noch zusätzlich die *normalen* und die *erweiterten Masterregeln* verwendet werden. Syntaktisch

[10] In diesem Kontext heißen die weiteren Regeln der Menge auch *Slaveregeln*

[11] In diesem Fall zählen die *erweiterten Masterregeln* nicht zu den *Slaveregeln*.

werden die *erweiterten Slaveregeln* durch ein ! (Ausrufungszeichen) gekennzeichnet. Es sollte niemals ein Gruppenbezeichner für *erweiterte Slaveregeln* und *normale* oder *erweiterte Masterregeln* gleichzeitig verwendet werden.

Asserts können einerseits global gültig (vgl. B.1.8), andererseits auch nur für eine einzige Regel gültig definiert werden. Diese lokalen *Asserts* werden zwischen den Optionen und dem Koreferenzteil der Regel plaziert. Die Syntax ist identisch zu der Syntax der globalen Asserts; sie gelten aber nur in der Regel, die sie enthält.

B.1.9. Das Lexikon

Der Lexikonabschnitt der Grammatik wird durch das Schlüsselwort `%lexicon` eingeleitet und durch `%eolex` beendet.

Das Lexikon selbst besteht aus einer Abfolge von *Defaults*, *Templates*, *Standard-*, *Suffix-* und *Korrektureinträgen*. Diese Einträge können in beliebiger Abfolge im Lexikon erscheinen.

Die Featuredefinitionen haben ebenfalls die Form von Koreferenzen, allerdings sind bei den hier benutzten Koreferenzlisten weder Funktionsaufrufe noch Featurereferenzen erlaubt, sondern lediglich direkte Wertangaben.

Defaults

Defaults definieren die *Standardeigenschaften* von Kategorien. Wenn ein *Default* für eine Kategorie definiert wurde, dann erben alle Lexikoneinträge[12] für diese Kategorie die im *Default* definierten Features. Diese können dann in dem Lexikoneintrag explizit überschrieben werden.

Ein *Defaulteintrag* hat die Syntax:

```
#default <kategorie>
[
  <koreferenzen>
]
```

[12] Genaugenommen erben nur die Standardenträge die Eigenschaften, für Suffixe und Korrekturen sind die *Defaults* nicht sinnvoll.

Templates

Die *Templates* im Lexikon entsprechen den *Templates* im Regelabschnitt(vgl. B.1.8). Es sind Abkürzungen für Koreferenzabfolgen, oder in diesem Abschnitt eher Definitionsabfolgen. Auch hier gibt es die Sonderbedeutung des Namens *foo* als beliebig oft definierbarer Eintrag.

Standardeinträge

Die Standardeinträge dienen zur Definition von Worten und von Wortstämmen, die mit Suffixen zusammen ein simples morphologisches Lexikon bilden. Die Syntax für Standardlexikoneinträge ist:

```
"<Lexikoneintrag>" <kategorie> [
  <koreferenzen>
                 ]
```

Der *Lexikoneintrag* kann dabei auch mehrere Worte umfassen; es können so Mehrwortbegriffe als ein Lexikoneintrag definiert werden, wie etwa *Tower Bridge*. Wenn ein Wort (oder das letzte der Worte eines Mehrwortbegriffes) mit einem ~ (Tilde) endet, dann wird dieser Begriff als Stamm aufgefaßt, der durch ein Suffix zu einem Wort komplettiert werden muß. Es wird mit allen Endungen versucht, eine Wortkomplettierung zu bilden.

Suffixeinträge

Suffixeinträge definieren die Endungen, die für eine Kategorie möglich sind. Die hier definierten Featurestrukturen werden mit denen der Wortstämme unifiziert. Sie liefern zusätzliche Information für den Lexikoneintrag. So kann etwa die Semantik eines Wortes im Stamm, die Flektionsinformation aber im Suffix definiert werden. Auch bei den Suffixen können Templates verwendet werden, die *Defaults* für eine Kategorie werden allerdings nicht beachtet.

Die Suffixeinträge müssen immer mit einem ~ beginnen; es gilt die Syntax:

```
"~<suffix>":<name> <Kategorie> [ <koreferenzen> ]
```

Dabei ist `<name>` eine Benamung für diesen Suffix, die für evtl. Korrektureinträge wichtig ist. Wenn keine Korrektureinträge für dieses Suffix gelten sollen, so kann auch hier beliebig oft der Name `foo` verwendet werden, auf den aber nicht referenziert werden kann.

Korrekturregeln

Wenn eine Stamm-/Suffixkombination in bestimmten Fällen nicht erwünscht ist – etwa bei unregelmäßigen Verbformen – dann kann ein sog. Korrektureintrag der Form:

```
<Stamm>/<Name>~<Suffix> <Kategorie> [<koreferenzen ]
```

benutzt werten. Dies verbietet die Kombination aus Stamm und dem Suffix mit dem Namen `name` und erzeugt ein korrigiertes Wort bestehen aus Stamm und dem korrekten Suffix `Suffix`. Beginnt `Suffix` nicht mit einer `~`, dann kann hier statt eines korrekten Suffix auch eine komplett unregelmäßige Form unabhängig vom Stamm angegeben werden. Die Koreferenzen für dieses Wort werden aus dem Stamm und dem in diesem Falle inkorrekten Suffix generiert. Die entsprechende Information muß somit nicht mehrfach kodiert werden. Die Koreferenzliste in diesem Eintrag dient zur Fehleridentifikation. Wenn jemand die Kombination aus Stamm und falschem Suffix benutzt, dann wird der Eintrag zwar identifiziert, aber es wird zu den Featurestrukturen aus Stamm und Suffix die hier abgelegte Struktur hinzugefügt. So kann eine Fehlerinformation in der Featurestruktur abgelegt werden, die den Fehler der Übergeneralisierung von Grammatikregeln kennzeichet.

Wird der Stamm leergelassen, also ein "" angegeben, dann bezieht sich der Korrektureintrag auf den letzten Stammeintrag, der vor diesem Eintrag im Lexikon steht.

B.2. Funktionsbibliothek

Es gibt eine Grundmenge an Funktionen für verschiedene Zwecke, die im System enthalten ist. Die Bibliothek ist unterteilt in mehrere Funktionsgruppen, die verschiedene Aufgabenbereiche erfüllen. Im Moment können die Funktionen keine Disjunktionen als Parameter verarbeiten. Man könnte Funktionen so implementieren, daß das Ergebnis einer Disjunktion wiederum eine Disjunktion ist. Im Moment ignorieren die Funktionen alles bis auf das erste Element der Disjunktion. Es wird aber auch kein Laufzeitfehler generiert, wenn eine Disjunktion als Parameter übergeben wird.

Stringbearbeitung

`strcat`$(s_1, s_2, \ldots)$ dient zur Konkatenation von Strings; das Funktionsergebnis ist ein String, der die Aneinanderreihung der als Parameter über-

gebenen Teilstrings bildet. Es können beliebig viele Strings übergeben werden.

`strcut`(s, a, l) kann genutzt werden, um aus einem String einen Teil herauszuschneiden. Es wird aus dem String s der Teil ab dem a-ten Zeichen (Zählung beginnt bei 0) ein Teilstring der Länge l als Ergebnis geliefert. Ist der String nicht so lang, wie die Länge vorgibt, ist der Ergebnisstring kürzer als die Länge l. Ist der Parameter a negativ, beginnt die Zählung am Ende des Strings, d. h. -1 beginnt beim letzten Zeichen, -2 bei vorletzten etc. Ist der Parameter l negativ oder Null, dann wird nicht die Länge, sondern die Endposition angegeben, also 0 für bis zum Ende, -1 für alles bis auf das letzte Zeichen.

`strlen`(s) hat als Resultat die Länge des Strings s.

Listenbearbeitung

`list`$(e_1, e_2, \ldots, e_n)$ erzeugt eine Liste aus seinen Parametern e_i[13]. Diese Parameter können wiederum jeden beliebigen Typ haben. Jeder Parameter ist ein Element der – n Elemente langen – Ergebnisliste.

`append`$(l, e_1, \ldots, e_n)$ hängt an die Liste l die Elemente e_i an, die Ergebnisliste ist also um n Elemente länger als die Liste l und besteht aus den Elementen von l erweitert um die Elemente e_i.

`concat`$(l_1, \ldots, l_n)$ hängt die Listen l_i aneinander.

`litem`(l, p) gibt das p-te Element einer Liste zurück. Ist dieses Element nicht in der Liste vorhanden, dann wird ein ungültiges Feature zurückgegeben. Ein solcher Wert ist mit nichts unifizierbar außer mit einem *leeren* Wert, d.h. er kann nur dann unifiziert werden, wenn das Zielfeature, mit dessen Wert unifiziert werden soll, nicht vorhanden ist. Die Zählung der Elemente beginnt bei 0. Ist die Zahl p negativ, dann wird ab dem Listenende rückwärts gezählt: -1 ist das letzte Element, -2 das vorletzte etc.

`lrange`(l, a, z) gibt eine Teilliste der Liste l zurück. Die Teilliste startet beim Element a[14] und hat die Länge z. Ist die Länge negativ oder Null, dann wird auch hier die Länge als Abstand vom Ende interpretiert: 0 steht für alles bis zum Ende, -1 für die Restliste ohne das letzte Element.

`llen`(l) gibt die Länge der Liste l, d. h. die Anzahl ihrere Elemente, zurück.

[13] e_i ist eine Kurzschreibweise für alle Elemente $e_i \in \{e_1, \ldots, e_n\}, 1 \leq i \leq n$.
[14] Auch hier werden negative Offsets ab dem Listenende gezählt-

Rechenfunktionen

`add`$(a_1, a_2, \ldots, a_n)$ liefert als Ergebnis $\sum_{i=1}^{n} a_i$.

`sub`$(a, s_1, s_2, \ldots, s_n)$ liefert als Ergebnis $a - \sum_{i=1}^{n} s_i$.

`mul`$(m_1, m_2, \ldots, m_n)$ liefert als Ergebnis $\prod_{i=1}^{n} m_i$.

`div`$(d, d_1, d_2, \ldots, d_n)$ liefert als Ergebnis $\frac{d}{\prod_{i=1}^{n} d_i}$.

Pfadfunktionen

`createPath`$(p_1, \ldots, p_n)$ erzeugt einen Pfad, der aus den Teilpfaden p_i zusammengesetzt ist. Alle Teilpfade p_i bilden einen neuen Pfad, der als Ergebnis zurückgeliefert wird.

`evalPath`(f, p, i) extrahiert Teilstrukturen. Es wird in der Featurestruktur f der Pfad p ausgewertet, und das entsprechende Feature wird zurückgeliefert. Ist dieser Pfad in der entsprechenden Featurestruktur nicht vorhanden, wird ein ungültiges Feature wie bei `litem` zurückgegeben. Ist das so gefundene Feature wieder ein Pfadfeature, dann kann dieses nocheinmal ausgewertet werden. Die maximale Anzahl an rekursiven Auswertungen gibt der Parameter i an. Dieser Parameter ist optional; wird er ausgelassen, dann gilt der Standardwert 1.

Tools

`print`$(p_1, \ldots, p_n)$ gibt die Funktionsparameter aus. Je nach Parsereinstellung wird die Ausgabe in einer Datei abgespeichert oder auf dem Bildschirm ausgegeben. Diese Funktion ist vor allem zum *Debuggen*, d. h. zur Fehlersuche, der Grammatiken gedacht. Oft wird sie in Kombination mit dem Pseudofeature `any` benutzt, da es keinen sinnvollen Rückgabewert gibt.

`getPosition[$`k`:]()` liefert die Position der Konstituente k im Input. Bei der Regel `S : NP VP` liefert der Aufruf `getPosition[S:]()` den Bereich, den der erkannte Satz ausfüllt, `getPosition[NP:]()` den Bereich der `NP`-Konstituente. Der Bereich wird als String der Form `3-5` zurückgeliefert. In diesem Fall wäre der Bereich von der Position 3 bis zur Position 5.

`getCat[$`k`:]()` liefert den Namen der Konstituente k der Grammatikregel.

C. Beispiele

C.1. Die Schnittstellendatei

testgram1-attr.y

```
/*
 *
 * $Id: testgram1-attr.y,v 1.1 1998/01/26 00:35:54 wteiken Exp $
 *
 * Globale Attribut- und Kategoriedefinitionen
 *
 * Wilfried Teiken <wteiken@cl-ki.uni-osnabrueck.de>
 *
 */

(#attrdef NUM  SING PLUR)
(#attrdef PERS FIRST SECOND THIRD)

(#subfeature ERROR)
(#listdef PREVERRS)
(#listdef AKTERRS)
(#strdef INFO)

(#listdef SUBCAT)

(#subfeature TYPE)
(#subfeature DETINFO)
(#attrdef CAT DET PRO N V PP)
(#attrdef INITIAL CONSONANT VOWEL)
(#attrdef DET_TYPE FORBIDDEN ALLOWED REQUIRED)
(#boolean SOLID)
(#boolean COUNTABLE)
(#boolean ALT)
```

```
/*
 * Das Startsymbol der Grammatik
 */
(#startsymbol S)

/*
 * Die Definition globaler Symbole aus VT und VN.
 */
(#vt DET N V PRO PG DEL)
(#vn-global NP VP PP)
```

C.2. Defaults und Templates

C.2.1. Defaults

testgram1-default.y

```
/*
 *
 * $Id: testgram1-default.y,v 1.2 1998/01/28 00:26:28 wteiken Exp wteiken $
 *
 * Defaults fuer lexikalische Kategorien
 *
 * Wilfried Teiken <wteiken@cl-ki.uni-osnabrueck.de>
 *
 */

#default N
[
 TYPE=[CAT=N; SOLID=YES; COUNTABLE=YES;];
 DETINFO=[DET_TYPE={ALLOWED, REQUIRED}; INITIAL=CONSONANT; ];
 PERS=THIRD;
]

#default DET
[
 TYPE=[CAT=DET;];
 SUBCAT=<>;
]

#default V
```

```
[
 TYPE=[CAT=V;];
 SUBCAT=<>;
]

#default PRO
[
 PERS=THIRD;
 NUM=SING;
 TYPE=[CAT=PRO;];
]
```

C.2.2. Templates

testgram1-templ.y

```
/*
 *
 * $Id: testgram1-templ.y,v 1.2 1998/01/28 00:26:28 wteiken Exp wteiken $
 *
 * Wilfried Teiken <wteiken@cl-ki.uni-osnabrueck.de>
 *
 */

/*
 * Templates fuer das Lexikon
 */

#template sing3rd
[
 NUM=SING;
 PERS=THIRD;
]

#template plur3rd
[
 NUM=PLUR;
 PERS=THIRD;
]

/*************************************************************/

/*
```

```
 * Templates fuer Grammatikregeln
 */

#template kongr1
[
 NUM = $1:NUM;
 PERS=$1:PERS;
]

#template kongr2
[
#use kongr1;
 NUM = $2:NUM;
 PERS=$2:PERS;
]

/*****************************************************************/

/*
 * Diese speziellen Templates werden automatisch in alle Regeln
 * eingefuegt. Die Zahl hinter der Regel gibt an, in welche Regeln
 * sie eingefuegt werden sollen.
 */

/* Regeln der Form X : A */
#template init1
[
 ERROR->PREVERRS=list($1:ERROR);
]

/* Regeln der Form X : A B */
#template init2
[
 ERROR->PREVERRS=list($1:ERROR, $2:ERROR);
]

/* Regeln der Form X : A B C */
#template init3
[
 ERROR->PREVERRS=list($1:ERROR, $2:ERROR, $3:ERROR);
]
```

C.3. Ein Lexikon

testgram1-lex.yl

```
/*
 *
 * $Id: testgram1-lex.yl,v 1.2 1998/01/28 00:26:28 wteiken Exp wteiken $
 *
 * Das Lexikon
 *
 * Wilfried Teiken <wteiken@cl-ki.uni-osnabrueck.de>
 *
 */

#include "testgram1-attr.y"
#include "stdfunc.y"

(#module LEX)

%eoattrdefs

/*****************************************************************/

%lexicon

#include "testgram1-templ.y"
#include "testgram1-default.y"

/*
 * Artikel
 */

/*
 * In diesem Beispiel wird ein Templatewert ueberschrieben
 */
"the" DET
[
#use sing3rd;
 NUM={SING,PLUR};
 TYPE=[INITIAL={CONSONANT, VOWEL};];
]

"a" DET
[
```

```
#use sing3rd;
 TYPE=[INITIAL=CONSONANT;];
]

"an" DET
[
#use last;
 TYPE=[INITIAL=VOWEL;];
]

/*
 * Nomen (Endungen)
 */

/*
 * Bei Nomina im Singular muss ein Artikel vorhanden sein
 */
"~":foo N
[
 NUM=SING;
 DETINFO->DET_TYPE=REQUIRED;
]

/*
 * Die "s"-Pluralbildug erfolgt nur bei zaehlbaren Nomina.
 */
"~s":nomplur N
[
 NUM=PLUR;
 TYPE->COUNTABLE=YES;
]

/*
 * Bei unzaehlbaren Nomina wird auch im Plural kein "s" angehaengt.
 */
"~":foo N
[
 NUM=PLUR;
 TYPE=[COUNTABLE=NO;];
]

/*
 * Nomen (Stammeintraege)
```

```
 */

"astronaut~" N
[
 DETINFO=[INITIAL=VOWEL;];
]

"man" N
[
#use sing3rd;
]

/*
 * Mit "#use last" kann die Featurestruktur des letzten Eintrages
 * als Template verwendet werden.
 */

"men" N
[
#use last;
 NUM=PLUR;
]

"army~" N
[
]

/*
 * Bei "army" ist die Pluralbildung unregelmaessig, daher erfolgt
 * direkt ein Korrektureintrag fuer das letzte Lexem. Dabei wird
 * eine komplett neue Form abgelegt.
 */
""/nomplur "armies" N
[
  ERROR=[ AKTERRS=<"irregular form, correct is armies">; ];
]

"stone~" N
[
]

"asteroid~" N
[
```

```
 DETINFO->INITIAL=VOWEL;
]

"castle~" N
[
]

"knight~" N
[
]

"mountain~" N
[
]

"hit~" N
[
 TYPE->SOLID=NO;
]

"water~" N
[
 TYPE->SOLID=NO;
 TYPE->COUNTABLE=NO;
]


"face~" N
[
]

"heart~" N
[
]

"James Bond" N
[
#use sing3rd;
 DETINFO->DET_TYPE=FORBIDDEN;
]

/*
 * Pronomina
```

```
 */
"I" PRO
[
 NUM=SING;
 PERS=FIRST;
 DETINFO->DET_TYPE=FORBIDDEN;
]

/*
 * Praepositionalgruppen
 */

"in" PG []

"on top of" PG []

/*
 * Verben (Endungen)
 */

"~":foo V
[
 NUM=SING;
 PERS={FIRST, SECOND};
]

"~":foo V
[
 NUM=PLUR;
 PERS={FIRST, SECOND, THIRD};
]

"~s":3rdsing V
[
 NUM=SING;
 PERS=THIRD;
]

/*
 * Verben (Stammeintraege)
 */
"walk~" V
[
```

```
]

"see~" V
[
 SUBCAT=<TYPE=[CAT=N;]>;
]

/*
 * Es koennen nur feste Gegenstaende getroffen werden.
 */
"hit~" V
[
 SUBCAT=<TYPE=[CAT=N; SOLID=YES;]>;
]

"storm~" V
[
 SUBCAT=<TYPE=[CAT=N; SOLID=YES;]>;
]

"miss~" V
[
 SUBCAT=<TYPE=[CAT=N; SOLID=YES;]>;
]

/*
 * Der Korrektureintrag fuer die unregelmaessige Form von "miss".
 * Es kann statt einer komplett neuen Form auch eine korrigierte
 * Endung angegeben werden.
 */
""/3rdsing "~es" V [
  NUM=SING;
  PERS=THIRD;
  ERROR=[ AKTERRS=<"irregular form, correct is misses">; ];
]

"stand~" V
[
]

%eolex
```

C.4. Eine Grammatik

C.4.1. Die Grammatikmodule

Das Hauptmodul

testgram1.y

```
/*
 *
 * $Id: testgram1.y,v 1.2 1998/01/28 00:26:28 wteiken Exp wteiken $
 *
 * Wilfried Teiken <wteiken@cl-ki.uni-osnabrueck.de>
 *
 */

#include "testgram1-attr.y"
#include "stdfunc.y"

%eoattrdefs

/*****************************************************************/

%rules

/*
 * Die Definition der Fehlerstrukturen fuer Koreferenzruecknahmen.
 */

#relax NUMERUS ERROR -> AKTERRS
[
 HEAD: [
      INFO=strcat("wrong numerus ",
                  ERRCAT:NUM,
                  " in cat:",
                  getCat[ERRCAT:](),
                  " needed ",
                  HEAD:NUM);
 ]
]

#relax PERS ERROR -> AKTERRS
[
```

```
 HEAD: [
      INFO=strcat("wrong person ",
                  ERRCAT:PERS,
                  " in cat:",
                  getCat[ERRCAT:](),
                  " needed ",
                  HEAD:PERS);
 ]
]

/*****************************************************************/

#include "testgram1-templ.y"

S : NP VP #!test1
#assert(NUMERUS 1|NUMERUS 2)
#assert(PERS 1|PERS 2)
[
 $0:[
   NUM = $1:NUM #opt NUMERUS 1;
   NUM = $2:NUM #opt NUMERUS 2;
   PERS=$1:PERS #opt PERS 1;
   PERS=$2:PERS #opt PERS 2;
 ]
]

/*
 * Diese Regel zeigt die alternative Verwendung der (eindeutigen)
 * Konstituentennamen in einer Regel.
 */
S : NP VP
#assert(NUMERUS 1|NUMERUS 2)
#assert(PERS 1|PERS 2)
[
 S:[
   ALT = YES;
   NUM = NP:NUM #opt NUMERUS 1;
   NUM = VP:NUM #opt NUMERUS 2;
   PERS=NP:PERS #opt PERS 1;
   PERS=VP:PERS #opt PERS 2;
 ]
]
```

```
%eorules
```

VP-Analyse

testgram1-VP.y

```
/*
 *
 * $Id: testgram1-VP.y,v 1.2 1998/01/28 00:26:28 wteiken Exp wteiken $
 *
 * VP-Analyse
 *
 * Wilfried Teiken <wteiken@cl-ki.uni-osnabrueck.de>
 *
 */

/*****************************************************************/

/*
 * In diesem Modul wird das lokale Symbol "VBAR" eingefuehrt.
 */

/*****************************************************************/

#include "testgram1-attr.y"
#include "stdfunc.y"

(#module VP)

%eoattrdefs

/*****************************************************************/

%rules

#include "testgram1-templ.y"

VP : VP.1 PP
[
 VP:[
#use kongr2;
   SUBCAT=VP.1:SUBCAT;
   TYPE=VP.1:TYPE;
```

```
 ]
]

/*
 * Diese Regel darf nur bei einer VBAR verwendet werden, deren
 * Subkatliste gesaettigt ist, d.h. es ist kein Element mehr im
 * "SUBCAT"-Feature vorhanden.
 */
VP : VBAR #!subcat-error
[
 VBAR:[
   SUBCAT=<>;
 ]
 VP:[
#use kongr1;
   SUBCAT=VBAR:SUBCAT;
   TYPE=VBAR:TYPE;
 ]
]

/*
 * Dieser Eintrag erlaubt das Parsing von Saetzen, bei denen
 * die Subkatliste nicht komplett gesaettigt wird. Die beiden
 * Regeln sind ueber das Symbol * "!subcat-error" miteinander
 * verbunden.
 */
VP : VBAR #!subcat-error
[
 VP:[
#use kongr1;
   SUBCAT=VBAR:SUBCAT;
   TYPE=VBAR:TYPE;
   ERROR->AKTERRS=<"Invalid verb usage, nonemtpty subcat list">;
 ]
]

/*
 * Diese Regel erweitert eine VBAR, deren Subcatliste noch mindestens
 * ein Element enthaelt. Dieses Element muss mit der vorhandenen NP
 * vertraeglich sein. Bei der resultierenden VBAR ist die Subkatliste
 * im ein Element verkuerzt.
 */
```

```
VBAR : VBAR.1 NP
[
 NP:[
   TYPE=litem(VBAR.1:SUBCAT, 0);
 ]
 VBAR:[
#use kongr1;
   SUBCAT=lrange(VBAR.1:SUBCAT, 1, 0);
   TYPE=VBAR.1:TYPE;
 ]
]

VBAR : V
[
 VBAR:[
#use kongr1;
   SUBCAT=V:SUBCAT;
   TYPE=V:TYPE;
 ]
]

%eorules
```

NP-Analyse

testgram1-NP.y

```
/*
 *
 * $Id: testgram1-NP.y,v 1.2 1998/01/28 00:26:28 wteiken Exp wteiken $
 *
 * NP-Analyse
 *
 * Wilfried Teiken <wteiken@cl-ki.uni-osnabrueck.de>
 *
 */

/*****************************************************************/

/*
 * In diesem Modul werden die lokalen Symboln "NBAR" und "NBARBAR"
 * eingefuehrt.
```

```
 */

/*****************************************************************/

#include "testgram1-attr.y"
#include "stdfunc.y"

(#module NP)

%eoattrdefs

/*****************************************************************/

%rules

#include "testgram1-templ.y"

/*
 * Die Koreferenz durch das "==" prueft lediglig die
 * Koreferenzierbarkeit, es wird aber keine Pfadkoreferenz erzeugt.
 * Dies kann durch das System effizienter bearbeitet werden.
 */

NP : DET NBARBAR
[
 NBARBAR:[
   DETINFO->DET_TYPE={ALLOWED, REQUIRED};
   DETINFO->INITIAL==DET:TYPE->INITIAL;
 ]
 NP:[
#use kongr2;
   TYPE=NBARBAR:TYPE;
 ]
]

NP : NBARBAR
[
 NBARBAR:[
   DETINFO->DET_TYPE={ALLOWED, FORBIDDEN};
 ]
 NP:[
#use kongr1;
   TYPE=NBARBAR:TYPE;
```

```
 ]
]


NBARBAR : NBAR PP
[
 NBARBAR:[
#use kongr1;
   TYPE=NBAR:TYPE;
   DETINFO=NBAR:DETINFO;
 ]
]

NBARBAR : NBAR
[
 NBARBAR:[
#use kongr1;
   TYPE=NBAR:TYPE;
   DETINFO=NBAR:DETINFO;
 ]
]

NBAR : N
[
 NBAR:[
#use kongr1;
   TYPE=N:TYPE;
   DETINFO=N:DETINFO;
 ]
]

NBAR : PRO
[
 NBAR:[
#use kongr1;
   TYPE=PRO:TYPE;
   DETINFO->DET_TYPE=FORBIDDEN;
 ]
]

%eorules
```

PP-Analyse

testgram1-PP.y

```
/*
 *
 * $Id: testgram1-PP.y,v 1.2 1998/01/28 00:26:28 wteiken Exp wteiken $
 *
 * PP-Analyse
 *
 * Wilfried Teiken <wteiken@cl-ki.uni-osnabrueck.de>
 *
 */

/*****************************************************************/

/*
 * Definition der globalen Strukturen laden
 */
#include "testgram1-attr.y"
#include "stdfunc.y"

%eoattrdefs

/*****************************************************************/

%rules

#include "testgram1-templ.y"

PP : PG NP
[
 PP: [
   NUM=NP:NUM;
   PERS=NP:PERS;
  ]
]

%eorules
```

Literaturverzeichnis

Arnold, Doug; Rondell, Martin und Fouvry, Frederik (1994): "Design and Implementation of Test Suite Tools". Report to LRE 62-089 D-WP5.1, University of Essex, UK.

Balkan, Lorna und Fouvry, Frederik (1995): "Corpus-Based Test Suite Generation". Report to LRE 62-089 D-WP5.2, University of Essex, UK.

Balkan, Lorna; Meijer, Siety; Falkedal, Kirsten; Arnold, Doug; Estival, Dominique; Lehmann, Sabine; Regnier-Prost, Sylvie; Dauphin, Eva und Netter, Klaus (1994a): "Issues in Test Suite Design". Report to LRE 62-089 D-WP2.1.2, University of Essex, UK.

Balkan, Lorna; Meijer, Siety; Falkedal, Kirsten; Arnold, Doug; Estival, Dominique; Lehmann, Sabine; Regnier-Prost, Sylvie und Dauphin, Eva (1994b): "Test Suite Design — Guidelines and Methology". Report to LRE 62-089 D-WP2.1.1, University of Essex, UK.

Bongardt, Gerd; Dalinghaus, Klaus; Dittmann, Henrik; Huber, Michael; Krumeich, Alexander; Peylo, Christoph; Rehm, Georg; Rother, Andreas und Teiken, Wilfried (1997): "Studienprojekt SemF — Abschlußbericht". Universität Osnabrück.

Chomsky, N. (1956): "Three models for the description of language". IRE Trans. on Information Theory 2 (3): S. 113 – 124.

Chomsky, N. (1963): "Formal properties of grammars". In: Handook of Math. Psych., New York: John Wiley and Sons, Band 2, S. 323 – 418.

De Remer, F.L. (1969): "Generating parsers for BNF grammars". In: Proceedigs 1969 Spring Joint Computer Conference. Montvale, N.J.: AFIPS Press, S. 793 – 799.

De Remer, F.L. (1970): "Simple LR(k) grammars". Comm. ACM 14 (7): S. 453 – 460.

DÖRRE, JOCHEN UND RAASCH, INGO (1991): The Stuttgart Type Unification Formalism, Band 168 von IWBS Report. Stuttgart: IBM Deutschland GmbH.

DÖRRE, JOCHEN UND SEIFFERT, ROLAND (1991): Sorted Feature Terms and Relational Dependencies, Band 153 von IWBS Report. Stuttgart: IBM Deutschland GmbH.

EARLEY, J. (1970): "An efficient context-free parsing algorithm". Comm. ACM 13 (2): S. 94 – 102.

ESTIVAL, DOMINIQUE; FALKEDAL, KIRSTEN; BALKAN, LORNA; MEIJER, SIETY; REGNIER-PROST, SYLVIE; NETTER, KLAUS UND OEPEN, STEPHAN (1994): "Survey of existing test suites". Report to LRE 62-089 D-WP1, University of Essex, UK.

FINKE, MICHAEL; GEUTNER, PETRA; HILD, HERMANN; KEMP, THOMAS; RIES, KLAUS UND WESPHAL, MARTIN (1997): "The Karlsruhe-Verbmobil Speech Recognition Engine". In: IEEE International Conference on Acoustics, Speech and Signal Processing, Munich, 1997. Noch nicht veröffentlicht.

FOUVRY, FREDERIK UND BALKAN, LORNA (1996): "Test suites for quality evaluation of NLP products". In: Le traitement automatique du langage et les applications industrielles / Natural Language Processing and Industrial Applications. NLP+IA/TAL+AI 96, 4-6 juin 1996. Hôtel Beauséjour, Moncton, N.-B., Canada. Gretal: Groupe d'étude sur le traitement automatique des langues, Université de Moncton, New Brunswick, Canada, Band Volume I, S. 86–92.

GAZDAR, C. UND MELLISH, C. (1989): Natural Language Processing in Prolog. Addison-Wesley.

GAZDAR, G.; KLEIN, E. UND PULLUM, G.K. (1985): Generalized Phrase Structure Grammar. Camebridge, Mass.: Havard University Press.

GÖRZ, GÜNTHER (HRSG.) (1995): Einführung in die künstliche Intelligenz. Addison-Wesley.

GRASSMAN, WINFRID K. UND TREMBLAY, JEAN-PAUL (1996): Logic and Discrete Mathematics. Prentice Hall.

HOPCROFT, J.E. UND ULLMAN, J.D. (1979): Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie. Addison-Wesley.

Hötker, Wilfried; Kanngiesser, Siegfried und Ludewig, Petra (1994): "Unterstützung der Wiederverwertung lexikalischer Ressourcen durch dynamische Wissensstrukturierung". Künstliche Intelligenz (KI) (4).

Johnson, Mark (1988): Attribute-value logic and the theory of grammar. Nummer 16 in CSLI Lecture Notes. Stanford: CSLI Publications.

Knuth, D.E. (1965): "On translation of languages from left to right". Information and Control 8 (6): S. 607 – 639.

Krüger, Anja und Ronthaler, Marc (1995): "Intelligente Systemreaktionen in CALL-Software". Auftragsarbeit für die IBM Deutschland GmbH.

Krüger-Thielmann, Karin (1992): Wissensbasierte Sprachlehrsysteme: Neue Möglichkeiten für den computergestützten Sprachunterricht. Gießener Beiträge zur Fremdsprachendidaktik. Narr.

Langer, Hagen (1990): "Syntactic normalization of spontaneous speech". In: COLING-90. Band 3, S. 180 – 183.

Levine, John; Mason, Tony und Brown, Doug (1992): Lex & Yacc. O'Reilly.

Naumann, Sven und Langer, Hagen (1994): Parsing. B.G.Teubner.

Nijholt, Anton (1994): "Linguistic engineering: A survey". In: Proceedings of the 7th Twente Workshop on Language Technology (TWLT 7). S. 1 – 22.

Oepen, Stephan; Netter, Klaus und Klein, Judith (1996): "tsnlp — Test Suites for Natural Language Processing". In: Linguistic Databases, herausgegeben von Nerbonne, John, Center for the Study of Language and Information, CSLI Lecture Notes. Forthcoming.

Pollard, C. (1985): Lecture Notes on Head-driven Phrase-structure Grammar. Stanford University. Unpublished manuscript.

Ritchie, Graeme D.; Russel, Graham J.; Black, Alan W. und Pulman, Stephen G. (1992): Computational Morphology. ACL. MIT Press.

Schwind, C. (1994): "Error analysis and explanation in knowledge based language tutoring". In: Proceedings of the 7th Twente Workshop on Language Technology (TWLT 7). S. 77 – 91.

Sells, Peter (1985): Lectures on Contemporary Syntactic Theories. CSLI Lecture Notes. Stanford: CSLI Publications.

Shieber, Stuart M. (1986): An Introduction to Unification-Based Approaches to Grammar. Nummer 4 in CSLI Lecture Notes. Stanford: CSLI Publications.

Shieber, Stuart M.; Uszkoreit, Hans; Pereira, Fernando C. N.; Robinson, Jane J. und Tyson, Mabrys (1983): "The formalism and implementation of PATR-II". In: Research on Interactive Acquisition and Use of Knowledge., herausgegeben von Grosz, Barbara und Stickel, Mark E., SRI International, S. 39 – 79.

Sikkel, Klaas (1997): Parsing Schemata. EACTS. Springer.

Smolka, Gert (1987): A Feature Logic with Subsorts, Band 33 von IWBS Report. Stuttgart: IBM Deutschland GmbH.

Smolka, Gert (1988): Feature Constraint Logik for Unification Grammars, Band 93 von IWBS Report. Stuttgart: IBM Deutschland GmbH.

Tarnowski, Michael (1991): Mathematische Grundlagen der formalen Linguistik – Ein Kompendium –, Band 174 von IWBS Report. Stuttgart: IBM Deutschland GmbH.

Tomita, M. (1985): Efficient Parsing for Natural Language. Boston, Mass.: Kluwer.

Volk, M.; Jung, M.; Richarz, D.; Fitschen, A.; Hubrich, J.; Lieske, C.; Pieper, S.; Ridder, H. und Wagner, A. (1994): GTU - A Workbench for the development of natural language grammars. Nummer 12 in Informatik Fachbericht. University of Koblenz.

Volk, Martin (1995): Einsatz einer Testsatzsammlung im Grammar Engineering, Band 30 von Sprache und Information. Tübingen: Max Niemeyer Verlag.