Prolog und Computerlinguistik

Teil I – Syntax

Stefan Müller

Juni, 1994

© Stefan Müller

überarbeitete Ausgabe vom 28. Mai 2008

Fachbreich 10: Sprach- und Literaturwissenschaft
Universität Bremen
Postfach 33 04 40
28334 Bremen

fon: (+49)-(+421) 218 - 86 01
fax: (+49)-(0421) 218 - 42 83
mail: Stefan.Mueller@cl.uni-bremen.de
http://www.cl.uni-bremen.de/~stefan/

Inhaltsverzeichnis

1 Konzepte der Sprache Prolog **1**
1.1 Prädikate 1
1.2 Unifikation 2
1.3 Komplexe Prädikate 2
1.4 Operatoren 3
1.5 Listen 4
1.6 Rekursion 5
1.7 Backtracking 5
1.8 Cut 5
1.9 *Negation-as-Failure* 6

2 Formale Sprachen **7**
2.1 Grundlagen der Mengentheorie 7
2.2 Sprachen 8
2.3 Formale Systeme zur Definition von Sprachen 9
2.4 Grammatiken 10
2.5 Automaten 12
2.6 Typ-3-Beschreibungen 13
2.7 Typ-2-Beschreibungen 14
2.8 Typ-1-Beschreibungen 15
2.9 Typ-0-Beschreibungen 15
2.10 Hülleneigenschaften 15

3 Formale und natürliche Sprachen **18**
3.1 Natürliche Sprachen in der Chomsky-Hierarchie 18
3.2 Reguläre Sprachen 19
3.3 Deterministische kontextfreie Sprachen 20
3.4 Kontextfreie Sprachen 23
3.5 Indizierte Sprachen 24

4 Berechenbarkeit und Komplexität **27**
4.1 Probleme und Algorithmen 27
4.2 Berechenbarkeit 28
4.3 Komplexität 29
4.3.1 Komplexität von Algorithmen 29
4.4 Berechnung der Komplexität 31
4.5 Komplexität des Erkennens 34
4.6 Komplexität des Parsens 35
4.7 Idealisierungen 35

5 Eine einfache Grammatik für das Englische **38**
5.1 Das Schreiben von Grammatiken 38
5.2 Das Finden von Regularitäten 39
5.3 Die vier wichtigsten lexikalische Kategorien 40
5.3.1 Nomina 40
5.3.2 Adjektive 40
5.3.3 Verben 40
5.3.4 Präpositionen 41
5.4 Syntaktische Kategorien 41
5.4.1 Nominalphrasen 42
5.4.2 Verbphrasen 44
5.4.3 Präpositionalphrasen 44
5.4.4 Adjektivphrasen 44
5.5 Sätze 45
5.6 Komplemente und Modifikatoren 45
5.7 Unzulänglichkeiten dieser Grammatik 46
5.7.1 Morphologie 46
5.7.2 Konstituentenstellung 47
5.7.3 Subkategorisierungsregelmäßigkeiten 47
5.7.4 Fernabhängigkeiten (*Nonlocal Dependencies*) 48
5.7.5 Koordination 48

6 Definite Clause Grammars **52**
6.1 Die DCG Notation 52
6.2 Die Bedeutung der DCG 54
6.3 Die Mächtigkeit der DCGs 55
6.4 Die übersetzung von DCGs in logische Formeln 56

7 Fernabhängigkeiten in DCGs **61**
7.1 Beschreibung der UDCs 63
7.2 Lückenfädeln (*Gap Threading*) 64

7.3 Beschränkungen 66
7.4 Erweiterungen 67
7.5 Die DCG-Grammatik 67

8 Parse-Strategien und DCGs 71
8.1 Kriterien für einen guten Parser 71
8.2 Parse-Strategien 72
8.3 Top-Down-Parsen 73
8.4 Bottom-Up-Parsen 75
8.5 Tiefe-zuerst-Suche vs. Breite-zuerst Suche 76
8.6 Grammatikübersetzung 77
8.7 Der Standard-DCG-Parser 77

9 Interpretierendes Parsen 79
9.1 Compiler und Interpreter 79
9.2 Top-Down-Erkenner 80
9.3 Shift-Reduce-Erkenner 81
9.4 Left-Corner-Erkenner 82

10 *Well Formed Substring Table* **86**
10.1 Probleme mit Backtracking bei Parsern 86
10.2 Der Cocke-Kasami-Younger-Algorithmus 87
10.3 Mehrdeutigkeit 89
10.4 Vom Erkenner zum Parser 90
10.5 Nachteile des Bottom-Up Parsens 92

11 *Active Chart Parsing* **93**
11.1 Regeln mit Punkten 93
11.2 Bottom-Up-Einbeziehung 94
11.3 Top-Down-Einbeziehung – der Earley Algorithmus 97
11.4 Die Tagesordnung (*Agenda*) 101
11.5 Konstruktion eines Parsers 102

12 Kategorialgrammatik 105
12.1 Lexikongesteuerte Grammatiken 105
12.2 Kategorialgrammatik 106
12.3 Unechte Mehrdeutigkeiten I 109
12.4 Kombinationsgrammatik 109
12.5 Andere Vorteile der Kategorialgrammatik 111
12.6 Unechte Mehrdeutigkeiten II 113

13 Einführung in die Unifikation **117**
13.1 Merkmale und Unifikation 117
13.2 Term-Unifikation 120
13.3 Graph-Unifikation 122
13.4 Der Formalismus der Merkmalstrukturen 123
13.4.1 Definition der Merkmalstrukturen 123
13.4.2 Darstellung als Graph 125
13.4.3 *structure sharing* 125
13.4.4 Subsumption 128
13.4.5 Unifikation 130
13.4.6 Negation, Disjunktionen und Implikationen 132
13.4.7 Listen 135
13.4.8 Funktionen 136
13.4.9 Typisierte Merkmalstrukturen als Erweiterung des Grundformalismus 136

14 Eine PATR-II-Implementierung **143**
14.1 Der PATR-II-Formalismus 143
14.2 Subkategorisierung 144
14.3 PATR-II in Prolog 146
14.4 Die Repräsentation von Merkmalstrukturen 147
14.5 Implementierung der Graph-Unifikation 148
14.6 Typskelette und *Partial Execution* 149

15 Unifikationsgrammatiken und deren Lexikon **155**
15.1 Die Komplexität von Lexikoneinträgen 155
15.2 Templates 155
15.3 Redundanzregeln 156
15.4 Multiplikationsregeln / Lexikalische Regeln 157

16 Parsen mit Unifikationsgrammatiken **161**
16.1 Möglichkeiten für das Parsen mit UGs 161
16.2 Parsen mit komplexen Kategorien 162
16.3 Kopieren und *structure sharing* 163
16.4 Terminierung 167

17 *Constraint-Based Grammars* **170**
17.1 Prinzipien und Beschränkungen 170
17.2 *Generalized Phrase Structure Grammar* 170
17.2.1 Die Objektgrammatik 171
17.2.2 Die Metagrammatik 171

17.2.3 Eine GPSG-Beispiel-Grammatik 174
17.3 *Government and Binding* 177
17.4 *Head-Driven Phrase Structure Grammar* 180
17.4.1 Prinzipien – zwei Beispiele 181

A Abkürzungsverzeichnis 183

Danksagung

Hiermit möchte ich mich bei Christopher Mellish bedanken, der mir die Verwendung des Edinburgher Skripts *Computational Syntax* (Mellish und Whitelock, 1992) gestattete.[1]

Der größte Teil des ursprünglichen *Computational Syntax*-Skripts stammt von Pete Whitelock und das Kapitel 5 stammt von Graeme Ritchie.

Vorwort

Aufbau des Skripts

Dieses Skript ist als Postscriptfile verfügbar.

Dieses Skript ist als Einführung in die Prolog-Programmierung gedacht. Dabei liegt der Schwerpunkt auf der Implementierung linguistischen Wissens in Form von Grammatiken und auf der Implementierung von Programmen zur Verarbeitung dieses Wissens. Im Skript werden elementare Prolog-Kenntnisse vorausgesetzt.

Viele Grammatik-Formalismen werden an Hand englischer Beispiele erklärt, da diese Formalismen fürs Englische entwickelt wurden und mitunter für Deutsch wenig brauchbar sind.

Wie schon in der Danksagung erwähnt stammen die meisten Kapitel des vorliegenden Skripts aus Edinburgh. In Kapitel 13 gehe ich jedoch etwas genauer auf den Formalismus der Merkmalstrukturen ein als das in den Edinburgher Skripten getan wurde. Die entsprechenden Kapitel entstammen meinem Buch *Deutsche Syntax deklarativ. HPSG für das Deutsche* Müller (1999).

Dem Kapitel 14 über die Prolog-Implementation von PATR-II habe ich einen Abschnitt über Typskelette und *partial execution* hinzugefügt.

Da die *Generalized Phrase Structure Grammar* (GPSG) für Deutsch noch größere Bedeutung hat als für Englisch ist ihr in diesem Skript mehr Platz eingeräumt worden (siehe Kapitel 17.2). Die GPSG-Grammatik und die Erklärung der Prinzipien ist von Uszkoreit (1987) übernommen worden.

Vorlesungen und Übungen

Am Ende eines jeden Kapitels gibt es Kontrollfragen, die man nach der Lektüre des betreffenden Kapitels beantworten können sollte. Die Übungsaufgaben dienen zur Vertiefung des erworbenen Wissens und zur Prüfungsvorbereitung.

[1] Eine aktuelle Version seines Skripts ist unter http://www.dai.ed.ac.uk/staff/personal_pages/chrism/distrib/csyn.ps verfügbar Mellish u. a. (1994).

Studenten, die einen Abschluß machen wollen, müssen die Hausaufgaben abgeben. Die Bewertung dieser Hausaufgaben geht zu 40% in die Endnote ein.

Kapitel 1

Einige wesentliche Konzepte der Sprache Prolog

Das folgende Kapitel enthält eine Aufzählung einiger wichtiger Prolog-Konzepte. Es stellt keine Einführung dar. Sollten einige der Konzepte unklar sein, ist die am Ende des Kapitels angegebene Literatur zu Rate zu ziehen.

1.1 Prädikate

In Prolog gibt es Prädikate der Form

```
functor(ARG1, ... , Argn).
functor.
```

Das erste Prädikat ist n-stellig – es hat n Argumente. Das zweite Prädikat ist nullstellig.

Prädikatnamen bestehen aus einem kleinen Buchstaben gefolgt von kleinen oder großen Buchstaben oder '_'. Argumente können Konstante (Atome) oder Variablen sein. Variablen unterscheiden sich von Atomen dadurch, daß Variablen mit einem Großbuchstaben am Anfang geschrieben werden.

```
dumm(karl)
```

und

```
dumm(Karl).
```

sind beides Prolog-Prädikate. In `dumm(Karl)` ist `Karl` eine Variable, die außer ihrer Bezeichnung keine Information enthält.

Prädikate können Bestandteile der Prolog Datenbasis sein. Die Datenbasis kann durch das Einlesen Prolog-Code enthaltender Files initialisiert werden. Mit

`assert` und `retract` Befehlen kann man der Datenbasis Prädikate hinzufügen oder welche aus ihr entfernen.

1.2 Unifikation

Zwei strukturierte Terme unifizieren miteinander, wenn ihre Teile miteinander unifizieren. Zwei Atome unifizieren miteinander, wenn sie gleich sind. Die Unifikation eines Atoms mit einer Variablen ist das Atom. Die Unifikation ist kommutativ.

Das Operationssymbol der Unifikation bzw. des Unifikationstests ist das Gleichheitszeichen.

Anweisung	Ergebnis
X = karl	karl = karl
Karl = karl	karl = karl
Y = karl, verliebt_in(X,Y)	karl = karl, verliebt_in(X,karl)
X = karl, verliebt_in(X,X)	karl = karl, verliebt_in(karl,karl)

Siehe auch Kapitel 13.

1.3 Komplexe Prädikate

In Prolog kann man ausdrücken, daß eine Relation aus mehreren anderen Relationen folgt. Z.B. folgt daraus, daß Karl Maria und daß Maria Karl kennt, daß sie einander kennen:

$$\begin{aligned}&kennt(karl, maria) \wedge kennt(maria, karl) \rightarrow \\ &kennen_einander(karl, maria)\end{aligned} \tag{1.1}$$

Diesen Schluß kann man verallgemeinern:

$$\begin{aligned}&kennt(X, Y) \wedge kennt(Y, X) \rightarrow \\ &kennen_einander(X, Y)\end{aligned} \tag{1.2}$$

In Prolog wird solch eine Schlußfolgerung wie folgt ausgedrückt:

```
kennen_einander(X,Y) :- kennt(X,Y),kennt(Y,X).
```

Nehmen wir einmal folgende Prolog-Datenbasis an:

```
kennt(max,victor).
kennt(victor,max).
```

Prolog sollte alle Lösungen beim Aufruf von `kennen_einander` liefern.

```
?-kennen_einander(X,Y).
X = max
Y = victor;

X = victor
Y = max;

no
?-
```

1.4 Operatoren

In Prolog kann man Zeichenketten, die von Standard-Operatoren verschieden sind, als Operator deklarieren.

```
:- op(1000,xfy,plus).
```

ist eine mögliche Operatorendefinition. Nach einer solchen Definition ist es möglich, Terme wie `1 plus 2` aufzuschreiben. Dieser Term entspricht `plus(1,2)`. Die Operator-Definitionen sind rein syntaktisch. Der Operator wird durch sie eingeführt, und etwas wird über seine Eigenschaften bzgl. der Klammerung von Termen ausgesagt.

Es kann eine Semantik für die Operatoren festgelegt werden:

```
:- op(999,ist_gleich,xfy).

A ist_gleich B plus C :-
  A is B + C.
```

Die Definition solcher Operatoren ist herzlich sinnlos, da der `ist_gleich` Operator eine Teilmenge der Fälle des `is`-Operators erledigt und `plus` auch weniger als das Standard-`+` in Prolog kann.

Nach der Definition

```
:-op(200,kennt,xfy).
```

ist es möglich, statt `kennt(karl,maria)`

```
karl kennt maria.
```

zu schreiben.

1.5 Listen

In Prolog gibt es Listen der Form `[X,Y,Z]`. Für die interne Verarbeitung von Listen gibt es den '.'-Operator. Die Liste `[X,Y,Z]` hat intern die Form.

```
.(X,.(Y,.(Z,[]))).
```

Das entspricht dem Baum in Abbildung 1.1.

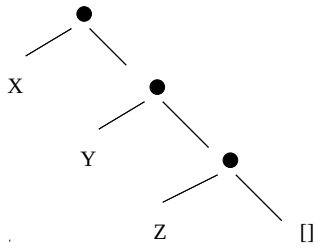

Abbildung 1.1: Interne Struktur von Listen

Es ist möglich, eine Liste in Anfang und Rest zu unterteilen:

```
X = [1,2,3,4],
X = [Y|Z].

Y = 1
Z = [2,3,4]
```

`|` ist der Operator, der das erste Element einer Liste abteilt. Will man die ersten beiden Elemente einer Liste weiterbearbeiten, so müssen zwei Elementen vor dem `|` auftauchen:

```
X = [1,2,3,4],
X = [Y,Z|Rest].

Y = 1,
Z = 2
Rest =[3,4]
```

Differenzlisten

Listen kann man zum schrittweisen Abarbeiten und zum Weiterreichen von Informationen benutzen. Ein einfaches Beispiel: Eine Liste von Wörtern ist ein Satz, falls sie aus einem Subjekt und einer Verbphrase besteht:

```
s(S1,S0) :- np(S1,S2),vp(S2,S0).

np([karl|X],X).
vp([schlaeft|X],X).
```

Der Aufruf von `s([karl,schlaeft],[]).` würde `yes` liefern.

1.6 Rekursion

Man wird kein Prolog-Programm finden, das ohne Rekursion auskommt. Ein Problem rekursiv zu lösen bedeutet, es in zwei oder mehrere Teilprobleme zu zerlegen, die Teilprobleme zu lösen und die Lösungen der Teilprobleme zur Gesamtlösung zusammenzufügen.

Das klassische Beispiel für ein rekursives Prädikat ist `append`. `append` soll zwei Listen verketten, bzw. Teile einer Liste liefern, je nachdem, welche Variablen beim Aufruf instantiiert sind:

```
append([],X,X).
append([X|T1],Y,[X|T2]) :- append(T1,Y,T2).
```

Die erste Klausel besagt: Die Verkettung einer leeren Liste mit einer Liste X ist X. Die zweite Klausel sagt: Um eine nichtleere Liste mit einer Liste Y zu verketten, merken wir uns das erste Element der Liste (X) und verketten den Rest (T1) mit Y. Das Ergebnis dieser Verkettung ist T2. An T2 hängen wir X wieder an und fertig ist die gesamte Verkettung.

1.7 Backtracking

Backtracking ist eine bestimmte Suchstrategie, die es Prolog ermöglicht, auf eine bestimmte Anfrage mehrere Antworten zu liefern.

1.8 Cut

Cuts sind nicht deklarativ und sollten deshalb äußerst sparsam verwendet werden. `xappend` ist eine prozedurale Variante von `append`. Im Gegensatz zu `append` findet `xappend` nicht alle Möglichkeiten eine Liste in zwei Teile zu teilen.

```
xappend([],X,X) :- !.
xappend([X|T1],Y,[X|T2]) :- xappend(T1,Y,T2).
```

Programme, die `xappend` verwenden, sind aber mitunter schneller, da sie weniger Backtrack-Informationen verwalten müssen, der Cut schneidet diese ab.

1.9 *Negation-as-Failure*

Es gibt in Prolog keine echte Negation. Das Fehlschlagen eines Ziels wird als ¬Ziel verstanden. Dem liegt die *Closed World Assumption* zugrunde. Will man zeigen, daß `not X` gilt, versucht man `X` zu beweisen. Ist `X` beweisbar, so muß `not X` fehlschlagen.

```
:- op(900, fy, not).

not X :- call(X), !, fail.
not X.
```

Lesetips

Wer mit den in diesem Kapitel kurz vorgestellten Begriffen nicht vertraut ist, sollte in den Büchern von Bratko (1987), Clocksin und Mellish (1987), Pereira und Shieber (1987) und Lehner (1990) nachlesen.

Die HU-Prolog Implementation wird zusammen mit einer Beschreibung ausgeliefert, in der die Grundbestandteile von HU-Prolog erklärt werden. Der Cut, `call` u.ä. werden mit Hilfe des Box-Modells gut erklärt.

Hausaufgabe

1. Schreiben sie ein Prolog-Prädikat, das folgendes leistet: Es wählt ein Element aus einer Liste aus (X) und gibt alle Elemente, die vor X in der Liste stehen zurück:

   % vor_x(+Liste,?X,?VorX).

   vor_x([1,2,3,4],X,VorX).
   X = 4, VorX = [1,2,3];
   X = 3, VorX = [1,2];
   X = 2, VorX = [1];

   no

Kapitel 2

Formale Sprachen

2.1 Grundlagen der Mengentheorie

Eine *Menge* ist eine Ansammlung von Objekten. Die Objekte werden *Elemente* genannt. Mengen schreibt man auf, indem man ihre Elemente in geschweiften Klammern aufzählt.

$$M_{Woche} = \{montag, dienstag, mittwoch, donnerstag, freitag, sonnabend, sonntag\} \tag{2.1}$$

ist die Menge der deutschen Wochentage. Das Symbol $\in$ wird als Abkürzung für *ist Element von* benutzt. Die folgenden Aussagen sind wahr:

$$\begin{aligned} &montag \in M_{Woche} \\ &dienstag \in M_{Woche} \\ &mittwoch \in M_{Woche} \\ &\ldots \end{aligned} \tag{2.2}$$

Die Frage *Wie oft ist* x *in S enthalten?* ist nicht sinnvoll. Ein Objekt kann nur Element einer Menge sein oder nicht. Es kommt nicht mehrfach in der Menge vor. Außerdem sind Mengen ungeordnet, so daß die Frage *Steht* x *in* M *vor* y*?* ebenfalls sinnlos ist. Zwei Mengen sind gleich, wenn sie dieselben Elemente enthalten. Die folgenden Mengen sind gleich:

$$\begin{aligned} &\{1,2,3\} \\ &\{2,3,1\} \\ &\{3,2,1\} \end{aligned} \tag{2.3}$$

Mengen werden im Computer oft in Form von Listen dargestellt. Listen sind natürlich geordnet. Manchmal benutzt man diese Ordnung bewußt, ordnet die Elemente z. B. alphabetisch, um dann Teilmengen oder Enthaltenseinsbeziehungen

schneller berechnen zu können. Um Mengen zu definieren, muß man nicht unbedingt alle Elemente der Menge aufzählen. Stattdessen kann man auch eine Menge von Elementen, die eine bestimmte Eigenschaft erfüllen, definieren. Die oben angegebene Menge läßt sich so auch wie folgt darstellen:

$$M_{Woche} = \{x | x \textit{ ist die deutsche Bezeichnung für einen Wochentag}\} \tag{2.4}$$

Man liest solche Mengenbeschreibungen wie folgt: *Die Menge aller* x, *wobei* x *die deutsche Bezeichnung eines Wochentags ist.*

Es gibt zwei wichtige Operationen über Mengen: *Durchschnitt* und *Vereinigung*. Der Durchschnitt zweier Mengen S_1 und S_2 ist die Menge der Objekte, die zu beiden Mengen gehören.

$$S_1 \cap S_2 = \{x | x \in S_1 \textit{ und } x \in S_2\} \tag{2.5}$$

Das heißt,

$$\{1,2,3,5\} \cap \{2,4,6,8\} = \{2\} \tag{2.6}$$

Die Vereinigung zweier Mengen ist die Menge der Objekte, die in beiden Mengen vorkommen:

$$S_1 \cup S_2 = \{x | x \in S_1 \textit{ oder } x \in S_2\} \tag{2.7}$$

Also:

$$\{1,2,3,5\} \cup \{2,4,8\} = \{1,2,3,4,5,8\} \tag{2.8}$$

Eine Menge S_1 ist eine *Teilmenge* einer anderen Menge S_2, $S_1 \subseteq S_2$, wenn jedes Element von S_1 auch Element von S_2 ist. Das heißt, daß jede Menge auch Teilmenge von sich selbst ist. Will man das ausschließen, spricht man von *echter Teilmenge* und benutzt das Symbol $\subset$.

2.2 Sprachen

Ein *Vokabular* ist eine Menge von Symbolen, die zu Phrasen verknüpft werden können. Wenn man Sätze als Wortsequenzen auffaßt, dann kann man das Vokabular des Deutschen $V_{Deutsch}$ wie folgt definieren:

$$V_{Deutsch} = \{Aachen, Aal, \ldots, Zytoplasma\} \tag{2.9}$$

Wenn S ein Vokabular ist, dann ist $S*$ die Menge, die alle möglichen endlichen Sequenzen von Elementen aus S enthält:

$$
\begin{aligned}
&V_{Deutsch}* = \\
&\{ \\
&\qquad \mathit{Aal\ Aal\ Aal\ Aal\ Aal\ Zytoplasma\ Aal}, \\
&\qquad \mathit{Aachen\ Zytoplasma\ fliegt\ Aal\ Aal\ ihn}, \\
&\qquad \mathit{Er\ riecht\ eine\ Laute} \\
&\qquad \mathit{Das\ klingt\ gut} \\
&\qquad \ldots \\
&\}
\end{aligned}
\tag{2.10}
$$

Der leere String ε ist ebenfalls Element dieser Menge.

Nicht jedes Element der Menge $V_{Deutsch}*$ ist eine vernünftige deutsche Phrase. Mathematisch wird eine Sprache über dem Vokabular S als Teilmenge von $S*$ definiert. Deutsch ist also eine Teilmenge von $V_{Deutsch}*$, also etwas wie:

$$
\begin{aligned}
&\{ \\
&\ \mathit{Er\ riecht\ eine\ Laute} \\
&\ \mathit{Das\ klingt\ gut} \\
&\ \ldots \\
&\}
\end{aligned}
\tag{2.11}
$$

Wenn α und β für Elemente des Vokabulars oder Sequenzen aus Elementen des Vokabulars stehen, dann bedeutet die Nacheinanderschreibung

$$\alpha\ \beta \tag{2.12}$$

die Verkettung der betreffenden Phrasen. Die Notation

$$\alpha^n \tag{2.13}$$

bedeutet die n-fache Wiederholung von α. In geschriebener Sprache werden Leerzeichen benutzt, um das Ende eines Wortes und den Anfang des nächsten zu markieren.

2.3 Formale Systeme zur Definition von Sprachen

Ein formales System, das zur Definition einer Sprache genutzt werden kann, besteht aus zwei Sachen:

1. Eine Charakterisierung des Wissens über eine Sprache, das von Sprache zu Sprache unterschiedlich ist.

2. Eine Spezifikation, wie man mit einem bestimmtem Teil dieses Wissens das Enthaltensein eines bestimmten Satzes in einer Sprache feststellen kann. Mit Hilfe einer solchen Spezifikation sollte man in der Lage sein, eine Prozedur, d. h. eine endliche Sequenz explizit formulierter Anweisungen, zu schreiben, die mechanisch ausgeführt werden kann. Die Prozedur sollte ein Algorithmus sein, der auf allen Eingaben terminiert, d. h. entscheidbar ist.

Wir werden zwei Alternativen, Sprachen zu definieren behandeln. Eine ist die Benutzung von Grammatiken, die Strings generieren, d. h. Sätze der entsprechenden Sprache, und die andere die Benutzung von Automaten, also Maschinen, die Sätze der Sprachen erkennen.

Wir werden vier verschiedene Grammatiktypen und die dazugehörigen Automaten kennenlernen. Diese vier Typen bilden eine Hierarchie. Die Sprachen, die mit einer Typ-n-Beschreibung beschrieben werden können, schließen die Sprachen, die mit einer Typ-n+1-Beschreibung beschrieben werden können, ein. Das heißt, je kleiner die Zahl, desto mächtiger die Grammatik. Diese Hierarchie wird *Chomsky-Hierarchie* genannt.

Diese Hierarchie ist für die Linguistik wichtig, da man, wenn man sich über die Komplexität natürlicher Sprache im klaren ist, weiß wie mächtig die Beschreibung sein muß, mit der die Sprache verarbeitet wird. Aus der Mächtigkeit der Beschreibung kann man die maximale Zeit berechnen, die man für das Erkennen von Sätzen brauchen kann.

2.4 Grammatiken

Def. 1 *Eine formale Grammatik G = [S,T,N,R] besteht aus vier Komponenten:*

- *T einer endlichen Menge von Terminalsymbolen,*
- *N einer endlichen Menge von Nichtterminalsymbolen,*
- *R einer endlichen Menge von Regeln der Form* $\alpha \rightarrow \beta$*, wobei* α *und* $\beta \in (N \cup T)*$ *sind und*
- *S einem besonderen Element von N – dem Startsymbol.*

Def. 2 (Satzform)

- *S ist eine Satzform, wenn S Startsymbol ist.*
- *Wenn x eine Satzform der Form* $\alpha\beta\gamma$ *ist und es eine Regel der Form* $\beta \rightarrow \delta$ *gibt, dann ist* $\alpha\delta\gamma$ *eine Satzform.*

Def. 3 *Eine Satzform, die nur Terminalsymbole enthält wird* **Satz** *genannt. Die Menge aller Sätze, die eine Grammatik G beschreibt, ist die* **Sprache L(G)**.

Eine Beispielgrammatik für Telefonnummern, die mindestens dreistellig sind, ist die folgende Grammatik.

N = {tel_nr, ziffer, ziffern}
T = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
S = tel_nr
R = {
tel_nr → ziffer ziffer ziffer ziffern

ziffern →
ziffern → ziffer ziffern

ziffer → 0

...

ziffer → 9
}

Die Definition von Satzform kann man dafür benutzen zu zeigen, daß 1 2 3 ein Satz der oben definierten Sprache ist, da die folgenden Sequenzen Satzformen sind.

tel_nr
ziffer ziffer ziffer ziffern
1 ziffer ziffer ziffern
1 2 ziffer ziffern
1 2 3 ziffern
1 2 3

Die erste Sequenz ist eine Satzform wegen des ersten Teils der Definition 2 und die anderen Sequenzen sind jeweils aus den vorigen Sequenzen abgeleitet. Das entspricht dem zweiten Teil der Definition.

Die Position einer Grammatik in der Chomsky-Hierarchie ist durch den Aufbau der Grammatikregeln bestimmbar. Je eingeschränkter die Möglichkeiten dafür sind, was auf linken und rechten Seiten von Regeln vorkommen darf, desto höher ist eine Grammatik in der Chomsky-Hierarchie einzuordnen, und desto kleiner ist die Menge der Sprachen, die durch entsprechenden Grammatiken beschrieben werden.

Typ	Automat		Grammatik	
	Speicher	Bezeichnung	Regeln	Bezeichnung
0	unbegrenzt	Turing Maschine	$\alpha \rightarrow \beta$	allgemeine Ersetzung
1	begrenzt	LBA	$\beta A\gamma \rightarrow \beta\delta\gamma$	kontextsensitiv
2	Stack	PDA	$A \rightarrow \beta$	kontextfrei
3	kein	FSA	$A \rightarrow xB, A \rightarrow x$	rechtslinear

Dabei sind A und B Nichtterminale, x ist ein String aus Terminalen, α, β, γ und δ sind Strings, die sich sowohl aus Terminal- als auch aus Nichtterminalsymbolen zusammensetzen können. δ darf nicht leer sein.

2.5 Automaten

Einen Automaten kann man sich aus drei Teilen bestehend vorstellen:

1. Ein Eingabeband, das in Felder aufgeteilt ist. Auf jedem Feld steht ein Symbol des Eingabealphabets. Es gibt einen Lese-Kopf, der über einem dieser Felder positioniert ist. Zu Beginn steht er über dem Feld, das am weitesten links auf dem Band steht.

2. Einem Hilfsspeicher, dessen Zustand durch zwei Funktionen bestimmt ist:

   (a) Eine Funktion, die aus der Menge der Speicherinhalte in die Menge der Symbole abbildet.

   (b) Eine Funktion, die die Menge der Speicherinhalte und einen Steuer-String in die Menge der Speicherinhalte abbildet.

3. Eine endliche Steuerkomponente, die die beiden oben erwähnten Komponenten steuert.

Ein Erkenner arbeitet schrittweise. Jeder Schritt besteht aus dem Lesen des Eingabesymbols, auf dem der Lese-Kopf gerade steht, und dem Anwenden der ersten Funktion. Das Funktionsergebnis, das Eingabesymbol und der aktuelle Zustand werden von der Steuerkomponente verarbeitet und so der neue Zustand bestimmt, in den der Automat übergeht. Zum Übergang in einen neuen Zustand gehört die Bestimmung der neuen Kopfposition. Der Kopf kann nach links, nach rechts oder garnicht bewegt werden. Zur Zustandsänderung gehört auch eine Veränderung des Speicherinhalts.

Die Arbeit des Automaten ist beendet, wenn der Lese-Kopf den rechten Rand des Bandes erreicht hat und der Automat in einem der Endzustände ist. Es kann auch Bedingungen für den Speicherinhalt im Endzustand geben. Wenn all diese

Bedingungen erfüllt sind, dann ist der Eingabe-String in der Menge der Sprache, die durch den Erkenner definiert wird. Sind sie nicht erfüllt, so ist der Eingabe-String nicht Element der Sprache.

Die Position eines Erkenners innerhalb der Chomsky-Hierarchie wird durch die Art des Speichers bestimmt, den der Automat hat. Je beschränkter die Speicherfunktionen sind, desto kleiner ist die Menge der Sprachen, die erkannt werden können.

2.6 Typ-3-Beschreibungen

Typ-3-Automaten werden auch *finite state automaton* genannt. Sie können durch ein *finite state transition network* beschrieben werden. In einem FSTN entspricht jeder Knoten einem Zustand des Automaten und Kanten entsprechen Symbolen, die vom Eingabeband gelesen werden können. Wenn ein Automat durch das Lesen eines x aus dem Zustand s_1 in den Zustand s_2 übergeht, dann gibt es im dazugehörigen Graphen eine Kante x, die vom Knoten s_1 zum Knoten s_2 geht. Ein Typ-3-Automat muß keinen Speicherinhalt berücksichtigen, um festzustellen in welchen Zustand er übergeht.

Die Typ-3-Grammatiken sind die, deren Ersetzungsregeln auf der rechten Seite maximal ein Nichtterminal haben. Wenn in Regeln auf der rechten Seite ein Nichtterminal vorkommt, muß dieses ganz rechts stehen. Solche Grammatiken heißen *rechtslinear*. Formal äquivalent dazu sind linkslineare Grammatiken. Bei ihnen muß das Nichtterminal ganz links in der rechten Seite stehen.

Diese Sprachen kann man durch reguläre Ausdrücke beschreiben. In regulären Ausdrücken kann man Symbole durch Hintereinanderschreibung, Wiederholung oder Alternativen kombinieren. Sie werden deshalb auch reguläre Sprachen genannt.

$$(det\ adj*\ noun) \mid pronoun \tag{2.14}$$

ist ein Beispiel für eine reguläre Sprache. Diese Sprache enthält das einzelne Symbol *pronoun* und Ketten der Form *det* gefolgt von einer beliebigen Anzahl von *adj* gefolgt von einem *noun*. Die äquivalente rechtslineare Grammatik hat die Form:

$$\begin{aligned} &S \rightarrow pronoun \\ &S \rightarrow det\ N \\ &N \rightarrow adj\ N \\ &N \rightarrow noun \end{aligned} \tag{2.15}$$

Dazu gibt es einen entsprechenden FSA. Der Startzustand ist 1, der Endzustand 3.

Startzustand	Eingabesymbol	Endzustand
1	pronoun	3
1	det	2
2	adj	2
2	noun	3

2.7 Typ-2-Beschreibungen

Typ-2-Grammatiken werden kontextfreie Grammatiken genannt. In Typ-2-Grammatiken darf auf der linken Seite nur ein einzelnes Nichtterminalsymbol stehen. Die Bezeichnung kontextfrei kommt daher, daß die Ersetzung des Nichtterminals nicht vom Kontext abhängt, in dem es auftaucht. Ein Typ-2-Automat wird Keller-Automat oder *Push Down Automaton* (PDA) genannt. Ein PDA ist ein *finite state automaton* mit einem Stack[1]. Bei einem Zustandsübergang kann der Automat ein Symbol auf den Stack legen. Bei der Bestimmung des Folgezustands kann das oberste Stack-Symbol überprüft und entfernt werden.

Eine typische kontextfreie Sprache, die nicht rechtslinear ist, ist die Sprache $a^n b^n$, d. h. die Sprache, die alle Strings enthält, die aus einer Zahl (n) von *a*s gefolgt von derselben Zahl von *b*s bestehen. Die Regeln für die CFG dieser Grammatik sind:

$$\begin{array}{l} \{ \\ S \rightarrow a\ S\ b \\ S \rightarrow \\ \} \end{array} \tag{2.16}$$

Den PDA zeigt die folgende Tabelle:

Startzustand	Eingabesymbol	Stack vorher	Endzustand	Stack nachher
0	a	Z	1	0Z
1	a	0...	1	00...
1	b	0...	2	...
2	b	0...	2	...
2	ε	Z	0	

[1]Stack oder Keller bedeutet folgendes: In den Keller schmeißt man Dinge durch eine Luke hinein. Will man ein bestimmtes Ding wieder herausholen, so muß man erst alle anderen Dinge, die danach in den Keller gebracht wurden, also oben auf liegen, herausnehmen, um an das ersehnte zu gelangen.

2.8 Typ-1-Beschreibungen

Typ-1-Grammatiken werden kontextsensitive Grammatiken genannt. Eine Regel einer solchen Grammatik muß mindestens ein Nichtterminal der linken Seite ersetzen und es können Kontexte angegeben werden, in denen die Ersetzung dieses Nichtterminals möglich ist. Das Nichtterminal darf nicht durch den leeren String ε ersetzt werden.

Eine analoge Beschränkung gibt es für die Typ-1-Automaten (*Linear Bound Automaton* LBA): Die Größe des Speichers eines LBAs darf die Länge des Eingabestrings nicht überschreiten.

Typische kontextsensitive Sprachen, die nicht kontextfrei sind, sind $a^n b^n c^n$, die Sprache, die aus allen Strings aus n *a*s gefolgt von n *b*s gefolgt von n *c*s besteht, und *ww* die Sprache, deren Strings aus zwei Hälften bestehen, wobei die erste Hälfte der zweiten gleicht. Die Grammatik für $a^n b^n c^n$:

$$\begin{array}{l} \{ \\ S \rightarrow aSBC \\ S \rightarrow a\,b\,C \\ b\,B \rightarrow b\,b \\ b\,C \rightarrow b\,c \\ c\,C \rightarrow c\,c \\ C\,B \rightarrow B\,C \\ \} \end{array} \tag{2.17}$$

2.9 Typ-0-Beschreibungen

Typ-0-Grammatiken sind allgemeine Ersetzungsgrammatiken, für die es keine Beschränkungen für die Regelform gibt. Der zu den Typ-0-Grammatiken gehörende Automat ist die Turing-Maschine. Sie hat einen unendlich großen Speicher. Alle Sprachen, für die es eine Erkennungsprozedur gibt, können durch Typ-0-Grammatiken oder Automaten definiert werden, aber das Erkennungsproblem ist für Typ-0-Automaten allgemein nicht entscheidbar (siehe Kapitel 4.5).

2.10 Hülleneigenschaften

Da wir Sprachen als Mengen von Strings definiert haben, können wir auch die Operationen Vereinigung und Durchschnitt auf Sprachen anwenden. Es ist oft einfacher Sprachen mit Hilfe dieser Operationen zu definieren als direkte Definitionen anzugeben. Deshalb ist es von Interesse, ob eine bestimmte Klasse von Sprachen bezüglich bestimmter Operationen abgeschlossen ist. z. B. stellt sich die

Frage, ob die Vereinigung zweier kontextfreier Sprachen wieder eine kontextfreie Sprache ist, oder welcher Klasse die Schnittmenge einer kontextfreien und einer regulären Sprache angehört. Es gibt Beweise für die folgenden Aussagen: Alle Klassen sind abgeschlossen bezüglich der Vereinigung mit sich selbst und bezüglich des Durchschnitts mit regulären Sprachen. Die Klasse der kontextfreien Sprachen ist nicht abgeschlossen bezüglich Durchschnitt mit sich selbst:

$$\begin{aligned} L_1 &= \{a^n b^n c^i | n \geq 1, i \geq 0\} \\ L_2 &= \{a^j b^n c^n | n \geq 1, j \geq 0\} \end{aligned} \tag{2.18}$$

L_1 und L_2 sind kontextfreie Sprachen aber

$$L_1 \cap L_2 = \{a^n b^n c^n | n \geq 1\} \tag{2.19}$$

ist nicht kontextfrei.

Kontrollfragen

1. In welchem Zusammenhang stehen eine Sprache und ihr Vokabular (entsprechend der Definition)?
2. Was braucht man, um eine Grammatik zu definieren?
3. Warum gehören verschiedene Automaten zu verschiedenen Grammatiktypen?
4. Was sind die Hauptbestandteile eines Automaten?
5. Was ist ein Typ-3-Automat? Welche Klasse von Grammatiken entspricht der Klasse der Typ-3-Automaten? Wie sehen die Regeln der dazugehörigen Grammatik aus?
6. Geben Sie ein typisches Beispiel für eine kontextfreie Sprache an, die nicht regulär ist.
7. Geben Sie ein typisches Beispiel für eine kontextsensitive Sprache an, die nicht kontextfrei ist.
8. Wieso ist die Theorie der formalen Sprachen für Linguistik wichtig?

Übungsaufgaben

1. * Zeigen Sie, daß der Satz *John eats John* ein durch die folgende Grammatik generierter Satz ist. Schreiben Sie die Werte von α, β, γ und δ aus der Definition der Satzform bei jedem Schritt auf.

   N = { S, NP, VP }, T = { John, eats }, Startsymbol = S,
   R = {
   S → NP VP,
   VP → V NP,
   NP → John,
   V → eats
   }

2. Zeigen Sie, daß der String *xxxx* in der Sprache, die durch die folgende Grammatik beschrieben wird, enthalten ist.

   N = { A, B, C, D, E, S }, T = {x}, Startsymbol = S,
   R = {
   S → ACxB,
   Cx → xxC,
   CB → DB,
   CB → E,
   xD → Dx,
   AD → AC,
   xE → Ex,
   AE →
   }

3. * Welche Sprache wird durch diese Grammatik definiert?

4. * Skizzieren Sie den Beweis dafür, daß die Sprache $a^n b^n$ nicht regulär ist.

Kapitel 3

Formale und natürliche Sprachen

3.1 Natürliche Sprachen in der Chomsky-Hierarchie

Wenn wir die natürlichen Sprachen in die Hierarchie der formalen Sprachen einordnen wollen, müssen wir zwei Dinge berücksichtigen. Einerseits wollen wir sie so hoch als möglich einordnen, da:

- je schwächer der Grammatik-Typ ist, desto stärkere Behauptungen kann man über die möglichen Sprachen aufstellen,
- je schwächer der Grammatik-Typ ist, desto größer kann die Effizienz der Parser für diese Grammatiken sein.

Andererseits kann es sein, daß wir Grammatiken von mächtigerem Typ benutzen müssen, da:

- wir mit schwächeren Grammatiken natürliche Sprachen nicht beschreiben können,
- da die Beschreibung mit einer Grammatik mächtigeren Typs kompakter, eleganter und präziser ist.

Natürlich sind Adäquatheit und Eleganz subjektive Kriterien, aber wir werden noch einige Konzepte einführen, die diese Begriffe präzisieren.

Eine formale Beschreibung einer Sprache, ein Automat oder eine Grammatik also, definiert nur eine Eigenschaft von Strings nämlich das Enthaltensein in einer Menge. Die Benutzung von Nichtterminalen in einer Grammatik, in der pro Schritt immer nur einzelne Nichtterminale ersetzt werden, führt zu einer hierarchischen Struktur über Strings, die den Ableitungsweg der Strings wiederspiegeln.

Die Kapazität einer Grammatik, eine Menge von Strings zu definieren, wird *schwache Kapazität* genannt. Für jede Sprache gibt es unendlich viele Grammatiken, die diese Sprache beschreiben und somit dieselbe schwache Kapazität haben.

Die Kapazität einer Grammatik, Strings Strukturen zuzuordnen, wird *starke Kapazität* genannt. *Starke Kapazität* ist ein Begriff, für den es eine exakte mathematische Definition gibt, den man aber eher in der Linguistik braucht, um Mehrdeutigkeiten und grammatische Beziehungen zwischen einzelnen Satzteilen beschreiben zu können. Eine elegante Grammatik ist eine Grammatik, deren starke Kapazität mit unseren linguistischen Vorstellungen übereinstimmt.

3.2 Reguläre Sprachen

Es ist ziemlich offensichtlich, daß die starke Kapazität regulärer Sprachen für die Beschreibung natürlicher Sprachen nicht ausreichend ist. Ein regulärer Ausdruck für einen einfachen Subjekt-Verb-Objekt-Satz ist:

$$((det\ adj*\ noun)) \mid pn)\ verb\ ((det\ adj*\ noun) \mid pn) \tag{3.1}$$

In diesem Ausdruck wird die Struktur der Phrasen nicht adäquat wiedergegeben. Reguläre Sprachen sind aber auch schwach inadäquat, da z.B. Einbettungen von Relativsätzen nicht beschrieben werden können. Ein Beispiel von **?** für Englisch: Die Menge

$$\{\ A\ white\ male\ (whom\ a\ white\ male)^n\ (hired)^n\ hired\ another\ white\ male. \mid n \geq 0\ \} \tag{3.2}$$

ist die Schnittmenge von Englisch und der regulären Menge

$$\{\ A\ white\ male\ (whom\ a\ white\ male)^i\ (hired)^j\ hired\ another\ white\ male. \mid i \geq 0,\ j \geq 0\ \} \tag{3.3}$$

(3.2) ist nicht regulär, da $a^n b^n$ nicht regulär ist. Reguläre Mengen sind abgeschlossen bezüglich Durchschnitt. Daraus folgt, daß (3.2) regulär sein müßte, wenn Englisch regulär wäre, da (3.3) regulär ist. Da (3.2) nicht regulär ist, kann Englisch nicht regulär sein.

Einbettungen in der Mitte sind ein Zeichen dafür, daß eine Sprache nichtregulär ist. Solche Einbettungen sind für Menschen nur bis zu einer bestimmten Tiefe verständlich (3.4a). Im folgenden Beispiel sind zum Vergleich Einbettungen mit nach rechts verzweigenden (3.4b) und nach links verzweigenden Strukturen (3.4c) zu sehen. Die Mitteleinbettungen sind schwieriger zu verstehen.

(3.4) a. The bath the plumber the firm your mother recommended sent put in is cracked.

b. On the table by the cupboard under the stairs in my house in London

c. My best friend's mother's hairdresser's Afghan hound's fur coat

Deutsche Beispiele für die rekursive Einbettung von Relativsätzen zeigt (3.5).

(3.5) a. Der Mann hat die Jacke, die die Frau, die gerade niest, liebt, umgefärbt.

b. Der Mann hat die Jacke, die die Frau, die den Mann, der seine Frau betrügt, kennt, liebt, umgefärbt.

Es gibt aber Sprachen, in denen solche Einbettungen üblich sind. Man kann außerdem auch für Englisch nicht genau sagen, wo die Akzeptanzgrenzen liegen. Würde man eine reguläre Grammatik schreiben, die Einbettungen der Tiefe 4 zuläßt, so wäre diese sehr viel größer und unübersichtlicher als eine entsprechende kontextfreie Grammatik.

3.3 Deterministische kontextfreie Sprachen

Nicht nur die regulären Sprachen können in linearer Zeit erkannt werden. Eine größere Sprachklasse, die ebenfalls in linearer Zeit erkannt werden kann, ist die Klasse der deterministisch kontextfreien Sprachen (DCFL). DCFL können von deterministischen PDAs erkannt werden. Das gängige Argument dafür, daß natürliche Sprachen keine DCFLs sind, ist, daß natürliche Sprachen mehrdeutig sind. Wenn man Sprachen als Mengen von Strings definiert, ist die Frage aber nicht, ob Sprache mehrdeutig ist, sondern ob sie inhärent mehrdeutig ist. Eine Sprache, für die es keine eindeutige (nicht mehrdeutige) Grammatik gibt, ist inhärent mehrdeutig. Eine Grammatik für eine Sprache kann mehrdeutig sein, ohne daß die Sprache selbst inhärent mehrdeutig ist. Die Grammatik ordnet dann Strings der Sprache mehrere verschiedene Ableitungsbäume zu, obwohl es Grammatiken für die Sprache gibt, die jedem String genau einen Baum zuordnen würden. Die Grammatik

$$\begin{aligned} &S \rightarrow if\ b\ then\ S\ else\ S \\ &S \rightarrow if\ b\ then\ S \\ &S \rightarrow a \end{aligned} \tag{3.6}$$

erzeugt für den Satz

(3.7) if b then if b then a else a

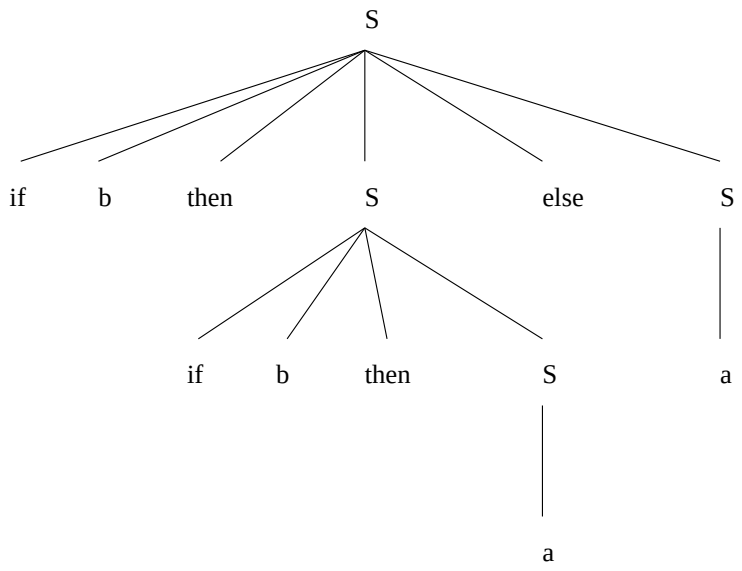

Abbildung 3.1: Erste Ableitung von *if b then if b then a else a*

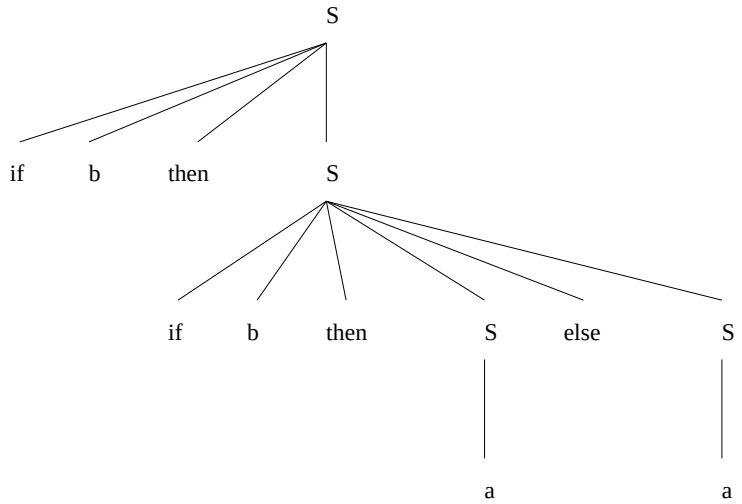

Abbildung 3.2: Zweite Ableitung von *if b then if b then a else a*

die beiden Ableitungsbäume aus Abbildung 3.1 und 3.2. Man kann aber auch eine eindeutige Grammatik angeben:

$$\begin{aligned}
&S1 \rightarrow if\ b\ then\ S1\\
&S1 \rightarrow if\ b\ then\ S2\ else\ S1\\
&S1 \rightarrow a\\
&S2 \rightarrow if\ b\ then\ S2\ else\ S2\\
&S2 \rightarrow a
\end{aligned} \tag{3.8}$$

Es kann keinen allgemeinen Algorithmus geben, der in der Lage ist, festzustellen, ob eine bestimmte CFG mehrdeutig ist oder ob eine gegebene CFG eine inhärent mehrdeutige Sprache definiert. Es ist möglich, daß eine große Teilmenge der natürlichen Sprache mit Hilfe nicht mehrdeutiger Grammatiken beschrieben werden kann. Das Hauptproblem ist wie so oft linguistischer Natur. Eine Veränderung der Grammatik verändert ihre starke Kapazität, und man will, daß ein String der semantisch mehrdeutig ist, auch verschiedene Ableitungsbäume hat, also auch syntaktisch mehrdeutig ist.

Diese Betrachtungsweise ist jedoch auch wieder nicht allzu dogmatisch zu sehen. Betrachtet man englische Komposita, die durch Hintereinanderschreibung komplexer oder atomarer Nomina gebildet werden, so müßte die Grammatik, die alle möglichen Strukturen liefert, die Form in (3.9) haben.

$$\begin{aligned}
&CN \rightarrow CN\ CN\\
&CN \rightarrow N
\end{aligned} \tag{3.9}$$

Die Zahl der möglichen Ableitungen, die ein Kompositum haben könnte, steigt wie die Catalan-Folge. $Catlan(n)$ ist die Zahl der binären Klammerungen eines Strings mit n Elementen.

1 2 5 14 42 132 469 1430 ...

(3.10) a. plastic baby pants

b. left engine fuel pump suction line

Für (3.10b) hätte man also 132 Parsebäume. Statt dessen benutzt man lieber Grammatiken der Form:

$$\begin{aligned}
&CN \rightarrow N\\
&CN \rightarrow N\ CN
\end{aligned} \tag{3.11}$$

und überläßt den Aufbau der genauen Struktur einer semantischen Komponente.

3.4 Kontextfreie Sprachen

Die Argumente in Bezug auf Mehrdeutigkeit sind dazu geeignet, zu zeigen, daß DCFGs nicht stark adäquat zur Beschreibung von natürlichen Sprachen sind. Gibt es Beweise dafür, daß deterministische kontextfreie Sprachen nicht schwach äquivalent mit natürlichen Sprachen sind?

Die Regel (3.12) für Sätze mit Subjekt-Verb-Kongruenz wird manchmal als Beweis gegen die schwache Adäquatheit angegeben.

$$S \rightarrow NP[num = \alpha]\ VP[num = \alpha] \tag{3.12}$$

Sie ist jedoch nichteinmal ein Argument gegen die starke Adäquatheit, da α nur die zwei Werte *singular* und *plural* annehmen kann. Man kann das also statt (3.12) die Regeln

$$\begin{aligned} S &\rightarrow NP[num = singular]\ VP[num = singular] \\ S &\rightarrow NP[num = plural]\ VP[num = plural] \end{aligned} \tag{3.13}$$

schreiben. Oder mit einer entsprechend erweiterten und großen Menge von Nichtterminalen:

$$\begin{aligned} S &\rightarrow NP_singular\ VP_singular \\ S &\rightarrow NP_plural\ VP_plural \end{aligned} \tag{3.14}$$

Dasselbe Argument läßt sich auf andere Phänomene anwenden, zu deren Beschreibung man Merkmale benutzt. Solange man eine endliche Menge von Werten für eine endliche Menge von Merkmalen benutzt, kann man die betreffende Grammatik in eine normale kontextfreie Grammatik umformen.

Es gibt jedoch Beispiele, die wirklich zeigen, daß es natürliche Sprachen gibt die nicht schwach kontextfrei sind. Hier niederländische Beispiele aus Johnson (1988):

(3.15) a. dat Jan de kinderen zag zwemmen

b. dat Jan Piet de kinderen zag helpen zwemmen

c. dat Jan Piet Marie de kinderen zag helpen laten zwemmen

Die Sätze bedeuten das folgende:

(3.16) a. daß Jan die Kinder schwimmen sah

b. daß Jan Piet den Kindern schwimmen helfen sah

c. daß Jan Piet Marie die Kinder schwimmen lassen helfen sah

Die niederländischen Sätze haben die Form $a^n b^n$ und könnten also von einer kontextfreien Grammatik generiert werden. In Schweizer-Deutsch (nicht im Niederländischen) gibt es aber auch Verben, die Dativobjekte verlangen. Ordnet man die Verben, die ein Akkusativobjekt verlangen, vor denen, die ein Dativobjekt verlangen, an, erhält man Sätze der Form $NP_a^n NP_d^m V_a^n V_d^m$. Die Sprache $a^n b^m c^n d^m$ kann nicht von einer kontextfreien Grammatik generiert werden.

Ein anderes Beispiel ist Bambara, eine Sprache, die in Senegal gesprochen wird. Es gibt in dieser Sprache Wörter der Form *w*-o-*w*, wobei *w* eine Zusammensetzung mehrerer Nomina sein kann, die dann im zweiten Teil des *w*-o-*w*-Wortes genauso wiederholt wird. Die Sprache $\{\ w\text{-}o\text{-}w \mid w \in \Sigma*\}$ ist nicht kontextfrei.

3.5 Indizierte Sprachen

Natürliche Sprachen gehören also nicht zu der Klasse der kontextfreien Sprachen. Man hat sich in der Vergangenheit bemüht, Beschreibungen zu finden, deren Mächtigkeit zwischen der der Typ-1- und Typ-2-Grammatiken liegt, in der Hoffnung, eine eingeschränktere Grammatikklasse als Typ-1 zu finden. Eine Grammatikklasse, die für die Beschreibung der nichtkontextfreien Phänomene adäquat zu sein scheint und dennoch nur eine echte Teilmenge der Typ-1-Grammatiken beschreibt, ist die Klasse der indizierten Grammatiken. Es ist nicht möglich, indizierte Grammatiken durch Ersetzungsregeln mit atomaren Symbolen zu beschreiben. Statt der atomaren Nichtterminalsymbole nimmt man Nichtterminale mit einem Stack und erlaubt nur Regeln einer bestimmten Form. Die einfachste Art indizierte Grammatiken zu beschreiben ist die, DCG-Regeln zu verwenden, in denen die Nichtterminale genau ein Argument haben, dessen Wert eine Liste ist, und für die außerdem das folgende gilt:

1. Ein Argument auf der rechten Seite oder alle Argumente auf der rechten Seite gleichen dem Argument auf der linken Seite. Gibt es Argumente, die dem auf der linken Seite nicht gleichen, so sind diese [ ].

   ```
   s(X) --> np([]), vp(X).
   s(X) --> np(X), vp(X).
   s(X) --> np(X).
   ```

2. Das Argument der linken Seite ist eine Liste mit einem Atom als Kopf und X ist der Rest. Auf der rechten Seite gibt es genau ein Argument das X ist, die anderen sind [ ].

   ```
   s([a|X]) --> np(X).
   s([i|X]) --> np(X),vp([]).
   s([a]) --> np([]),vp([]).
   ```

3. Ein Argument auf der rechten Seite oder alle Argumente auf der rechten Seite gleichen einer Liste mit einem Atom als Kopf und einem Wert X als Rest. X ist der Wert des Arguments der linken Seite. Die anderen Elemente der rechten Seite haben [ ] als Argument.

   ```
   s(X) --> np([a|X]).
   ```

Die Grammatik für $a^n b^n c^n$ sieht wie folgt aus:

```
s([]) --> [a], as([a]).
as(X) --> [a], as([a|X]).
as(X) --> bs(X).
bs([a|X]) --> [b], bs(X), [c].
bs([]) --> [].
```

und die für die Sprache $\{ww \mid w \in \Sigma*\}$:

```
s([]) --> [wordi], a([wordi]).
a(X) --> [wordi], a([wordi|X]).
a(X) --> b(X).
b([wordi|X]) --> b(X), [wordi].
b([]) --> [].
```

Wobei die Regeln mit den Terminalen `wordi` für jedes Wort vorhanden sein müssen.

Kontrollfragen

1. Was ist der Unterschied zwischen schwacher und starker Kapazität von Grammatiken?

2. Was ist Einbettung und warum kann man Einbettung nicht mit regulären Sprachen beschreiben?

3. Welche zwei Beispiele dafür, daß es nicht-kontextsensitive Sprachen gibt, wurden angegeben? In welcher Beziehung stehen diese Phänomene zu einfachen formalen Sprachen, die nicht kontextfrei sind?

4. Warum sind indizierte Sprachen interessant für die Computerlinguistik?

Übungsaufgaben

1. Schreiben Sie eine Grammatik, die die in (3.2) angegebene Sprache definiert.

2. Fertigen Sie eine Tabelle an, die die wichtigen Informationen über die Chomsky-Hierarchie enthält. Sie sollte folgende Spalten enthalten:

   - Chomsky-Typ-Nummer
   - Bezeichnung der Grammatiken dieses Typs
   - Bezeichnung der Automaten dieses Typs
   - eine formale Sprache diese Typs
   - NL-Phänomene, die mit Grammatiken diese Typs nicht beschrieben werden können

   Verwenden Sie beim Ausfüllen maximal 10 Wörter pro Spalte.

Hausaufgabe

2. Schreiben Sie eine indizierte Grammatik für die Sprache $a^m b^n c^m d^n$. Passen Sie diese Grammatik so an, daß sie Verbphrasen des Schweizer-Deutsch beschreiben kann. Das heißt, die Grammatik soll Verben mit Akkusativ und mit Dativ-Objekten erlauben. Es reicht aus, wenn Sie eine Grammatik schreiben, die nur Phrasen zuläßt, in denen erst alle Akkusativ-NPs, dann alle Dativ-NPs, alle Akkusativ-Verben und dann alle Dativ-Verben kommen, obwohl diese im richtigen Schweizer-Deutsch auch gemischt auftreten können, vorausgesetzt, die Reihenfolge der NPs stimmt mit der der VPs überein.

Kapitel 4

Berechenbarkeit und Komplexität

4.1 Probleme und Algorithmen

Die meisten Probleme in der Informatik sind algorithmische Probleme. Ein algorithmisches Problem ist eine Problembeschreibung, die die Form der Eingaben und die der Ausgaben spezifiziert. Die Eingaben sind die Informationen, die dem Problemlöser außer den Instruktionen zur Lösung des Problems noch zur Verfügung steht. Die Instruktionen müssen so sein, daß das Problem unabhängig von den Eingaben lösbar ist. Die Ausgaben sind das Ergebnis, das beim Lösen des Problems erarbeitet wurde. Wie die Ausgaben mit den Eingaben zusammenhängen, muß in der Beschreibung des Algorithmus angegeben werden. Ein *Algorithmus* ist die beste Art von Spezifikation für die Lösung eines algorithmischen Problems. Traditionell beschreibt ein Algorithmus das Vorgehen bei der Lösung eines Problems in einzelnen Schritten, die die Bedingungen der Endlichkeit, Definiertheit und Wirksamkeit erfüllen müssen.

- *Endlichkeit* Ein Problemlöser muß unabhängig von der Eingabe nach einer endlichen Anzahl von Schritten terminieren.
- *Definiertheit* Jeder Schritt des Algorithmus muß genau festgelegt sein, und es muß genau angegeben sein, welcher Schritt als nächstes auszuführen ist.
- *Effizienz* Jeder Schritt muß so einfach sein, daß man ihn in endlicher Zeit ausführen kann.

Zusätzlich verlangt man, daß ein Algorithmus bei jedem Schritt nur Informationen benutzt, die entweder zur Eingabe gehören oder in vorangegangenen Schritten errechnet wurden, und daß der Algorithmus die Ausgabe berechnet.

4.2 Berechenbarkeit

Ein algorithmisches Problem ist *berechenbar*, wenn es einen Algorithmus gibt, der für jede erlaubte Eingabe eine korrekte Ausgabe liefert. Ansonsten wird das Problem unberechenbar genannt.

Das scheint ein bißchen vage zu sein, da in der obigen Algorithmusdefinition der Begriff *Schritt* nicht genau definiert wurde. Eine genaue Definition kann man in Form eines Programms für einen hypothetischen, extrem einfachen Computer geben. Dieser Computer wird *Turing-Maschine* genannt. Der Schritt einer Turing-Maschine besteht aus dem Einlesen eines Symbols von einer bestimmten Position eines Eingabebandes, einer möglichen Bewegung des Schreib-Lese-Kopfes nach rechts oder links, der Änderung des Wertes, der auf dem Band steht und der Bestimmung des nächsten Schritts in Abhängigkeit vom gelesenen Symbol.

Diese Algorithmus-Definition ist natürlich nicht sonderlich sinnvoll, da keiner für Turing-Maschinen Programme schreiben wird. Es wird aber angenommen, daß die Turing-Maschine zu allen komplexeren Maschinen äquivalent ist, vorausgesetzt diese haben unbegrenzt viel Speicherplatz (*Churchsche Hypothese*). Wir können also mit Algorithmusschritten zufrieden sein, die mit den Computern, die wir benutzen, in endlicher Zeit durchführbar sind. Aber auch die entgegengesetzte Richtung ist wichtig: Für Probleme, deren Unberechenbarkeit bewiesen wurde, ist es nicht möglich, ein Programm zu schreiben, egal wie schnell der Computer, für den das Programm geschrieben werden soll, auch ist.

Eine wichtige Problemklasse ist die Klasse der Probleme, die eine einzige Eingabe haben und deren Ausgabe *ja* oder *nein* ist. Solche Probleme werden *Entscheidungsprobleme* genannt. Ein Entscheidungsproblem ist es zu bestimmen, ob eine bestimmte Eingabe eine bestimmte Eigenschaft hat, oder, was äquivalent ist, ob eine Eingabe Element einer Menge ist. Ein Entscheidungsproblem, das berechenbar ist wird *entscheidbar* genannt.

Eine Menge ist *rekursiv*, wenn das Enthaltensein in der Menge entscheidbar ist.[1] Die Menge der graden Zahlen in Dezimalschreibweise ist rekursiv, da einfach festzustellen ist, ob eine Zahl in Dezimaldarstellung gerade ist. Man muß sich nur die letzte Ziffer ansehen. Ist diese 0, 2, 4, 6 oder 8, so ist die Zahl gerade. Es sind jedoch nicht alle Mengen rekursiv. Es gibt z. B. keinen Algorithmus, der berechnet, ob zwei beliebige logische Ausdrücke äquivalent sind. Demzufolge ist es nicht entscheidbar, ob eine logische Formel zur Menge der zu einer anderen Formel äquivalenten Ausdrücke gehört.

Eine nichtrekursive Menge kann aber *rekursiv aufzählbar* sein. Eine Menge ist rekursiv aufzählbar, wenn es einen Algorithmus gibt, der für jedes positive n

[1] Der Begriff *rekursiv* hat in der Berechnungstheorie eine andere Bedeutung als in der Linguistik oder in der Informatik.

die ersten n Elemente der Menge angibt. Wenn eine Menge rekursiv aufzählbar ist, können wir einen Prozedur angeben, die testet, ob ein Element in der Menge enthalten ist. Wir lassen einfach den Aufzählalgorithmus arbeiten und vergleichen jedes aufgezählte Element mit dem, dessen Enthaltensein wir testen wollen. Diese Prozedur ist kein Algorithmus, da sie nur *ja* ausgibt, falls das Element enthalten ist. Ist es nicht enthalten, so terminiert die Prozedur nicht, es sei denn die Menge wäre endlich, aber dann ist sie sowieso rekursiv.

Die Menge der Folgerungen aus einer Aussage der Prädikatenlogik erster Stufe ist rekursiv aufzählbar.

4.3 Komplexität

Die Analyse der Berechenbarkeit eines Problems gibt uns Aufschluß darüber, ob ein Problem überhaupt lösbar ist. Die Analyse der Komplexität von Problemen gibt uns Informationen darüber, wie effektiv ein Problem gelöst werden kann.

4.3.1 Komplexität von Algorithmen

Der Einfachheit halber beschäftigen wir uns im folgenden nur mit algorithmischen Problemen mit genau einer Eingabe, obwohl alle Feststellungen, die man bei der Betrachtung von Problemen mit einer Eingabe machen kann, auch für Probleme mit mehreren Eingaben zutreffen. Die Anzahl der Schritte, die ein Algorithmus zur Lösung des Problems braucht, hängt gewöhnlich von der Komplexität der Eingabe ab. So hängt zum Beispiel die Anzahl der Schritte, die ein Algorithmus braucht, um zu bestimmen, ob ein String in einer Sprache ist, von der Länge des Strings ab. Die Komplexitätstheorie beschäftigt sich damit, wie der Arbeitsaufwand des Algorithmus von einem Maß für die Komplexität der Eingabe (normalerweise n) abhängt. Manchmal ist es möglich, eine genaue Formel für die Anzahl primitiver Operationen, deren Komplexität nicht von n abhängt, in Abhängigkeit von n zu finden (z. B. $25n^2 + 368$). Aber diese Genauigkeit ist wenig nützlich, da die Komplexität der einzelnen Schritte nicht berücksichtigt wird. Diese Komplexität der einzelnen Schritte hängt wesentlich von der Art der Maschine ab, auf der der Algorithmus läuft. Um die inhärente Komplexität eines Algorithmus unabhängig davon, auf welcher Maschine er läuft, zu bestimmen, ermittelt man, in welcher Größenordnung der Arbeitsaufwand in Abhängigkeit von n steigt. Das verschafft uns eine Vorstellung davon, wie gut der Algorithmus arbeitet, wenn wir größere Probleme mit ihm bearbeiten wollen, wobei von der Zeit, die primitive Operationen benötigen, abstrahiert wird. Im folgenden sind einige Beispiele für die Komplexität von Algorithmen angegeben:

- *konstant*: Der Arbeitsaufwand hängt nicht von n ab.

- *logarithmisch*: Der Arbeitsaufwand verhält sich wie $log_k(n)$, für eine Konstante k.
- *polynomial*: Der Arbeitsaufwand verhält sich wie n^k, für eine Konstante k. Das wird manchmal noch in *linear* (k = 1), *quadratisch* (k = 2), *kubisch* (k = 3), usw. unterteilt.
- *exponential*: Der Arbeitsaufwand verhält sich wie k^n, für eine Konstante k.

Diese Angaben beziehen sich nur auf Größenordnungen. Zum Beispiel sind Algorithmen mit der Komplexität $n/4000$ und $5000n$ linear. Wenn das n größer wird, werden solche konstanten Faktoren jedoch belanglos. Die Frage wie die höchste Potenz von n ist, die in der Komplexitätsformel vorkommt, wird mit wachsendem n immer wichtiger. Für jede lineare Funktion von n und jede quadratische Funktion von n gibt es ein n, ab dem der Wert der quadratischen Funktion größer als der der linearen ist. Genauso gibt es für eine polynomiale und eine exponentiale Funktion ein n, für das der Funktionswert der exponentialen Funktion größer als der der polynomialen ist. Das verdeutlicht die folgende Tabelle:

$log_e(n)$	0	0,7	1,6	2,3	3	4,6	6,9
n	1	2	5	$1*10$	$2*10$	$1*10^2$	$1*10^3$
n^2	1	4	$2*10$	$1*10^2$	$4*10^2$	$1*10^4$	$1*10^6$
n^3	1	8	$1*10^2$	$1*10^3$	$8*10^3$	$1*10^6$	$1*10^9$
$100*n^3$	$1*10^2$	$8*10^2$	$1*10^4$	$1*10^5$	$8*10^5$	$1*10^8$	$1*10^{11}$
n^4	1	$2*10$	$6*10^2$	$1*10^4$	$2*10^5$	$1*10^8$	$1*10^{12}$
n^{25}	1	$3*10^7$	$3*10^{17}$	$1*10^{25}$	$3*10^{32}$	$1*10^{50}$	$1*10^{75}$
$1000*n^{25}$	$1*10^3$	$3*10^{10}$	$3*10^{20}$	$1*10^{28}$	$3*10^{35}$	$1*10^{53}$	$1*10^{78}$
2^n	2	4	$3*10$	$1*10^3$	$1*10^6$	$1*10^{30}$	$1*10^{301}$

Die Tabelle zeigt, daß sich die Funktionen für $n > 500$ schon so verhalten, wie man es sich entsprechend ihrer Einordnung in Komplexitätsklassen vorstellen würde. $1000*n^{25}$ unterscheidet sich für kleine n nicht wesentlich von 2^n. Bei großen n sieht man die Unterschiede aber deutlich. Man kann folgende Reihenfolge angeben:

$$\begin{array}{l} \ldots \\ 2^n \\ \ldots \\ n^3 \\ n^2 \\ n \\ log_e(n) \\ k \end{array} \tag{4.1}$$

Nimmt man eine Funktion und kombiniert sie mit einer Funktion, die in dieser Aufzählung weiter unten steht, durch Multiplikation oder Addition, dann verhält sich die entstandene Funktion immer noch wie die erste Funktion, da die Funktion, mit der sie kombiniert wurde, ab einem bestimmten n keine Rolle mehr spielt. $25n^3 + 3887n^2 + 100$ bezeichnet man also einfach als kubisch. Das heißt, wir können in Komplexitätsformeln alle Teile bis auf den, der in (4.1) am weitesten oben steht, weglassen.

Die Unterteilung der Klasse polynomial in Teilklassen ist nicht so ohne weiteres zulässig, da verschiedene Klassen von Turing-Maschinen unterschiedlich funktionieren. Zum Beispiel kann ein Algorithmus auf einer Maschine mit freiem Speicherzugriff die Komplexität n^3 haben, auf einer Turing-Maschine, die nur einen Platz auf dem Band nach dem anderen lesen kann, dagegen die Komplexität n^6. Obwohl die Unterscheidung zwischen polynomialen und exponentiellen Algorithmen immer sinnvoll ist, ist es nur sinnvoll, die Art der Komplexität von polynomialen Algorithmen genauer anzugeben, wenn man sich damit auf eine bestimmte Art Maschine bezieht.

4.4 Berechnung der Komplexität

Um die Komplexität eines Algorithmus zu bestimmen, muß man sich entscheiden, in Bezug auf welche Eingabe die Komplexität berechnet werden soll, wie n bestimmt werden soll, und man muß herausfinden, wie der Arbeitsaufwand von n abhängt. Wir werden im folgenden drei Beispiele dafür geben, wie die Berechnung der Komplexität erfolgen kann. Dabei nehmen wir an, daß die Maschinen, auf denen die Algorithmen angewendet werden sollen, Maschinen mit wahlweisem Speicherzugriff sind.

Deterministische Prolog-Programme

Zuerst betrachten wir deterministische Prolog-Programme, d. h. Programme, die kein Backtracking machen. Die folgenden Programme sind zwei Varianten von `reverse`, einem Prädikat, das zu einer Liste eine Liste mit entgegengesetzter Reihenfolge konstruieren soll.

```
naive_rev([],[]) :- !.
naive_rev([H|T],Rev) :- naive_rev(T,Rev1),
        append(Rev1,[H],Rev).

   append([],X,X) :- !.
   append([X|Y],Z,[X|Y1]) :- append(Y,Z,Y1).
```

```
good_rev(L,Rev) :- rev1(L,[],Rev).

   rev1([],L,L) :- !.
   rev1([X|Y],L,Rev) :- rev1(Y,[X|L],Rev).
```

Wir betrachten zuerst die Komplexität von `append`. `append` wird benutzt, um zwei Listen zu einer dritten zu verketten. Angenommen n ist die Länge der Eingabeliste (des ersten Arguments). Die Prozedur benutzt das zweite Argument nur ganz zum Schluß einmal bei der Unifikation in der ersten Klausel. `append` arbeitet so, daß es sich immer wieder selbst aufruft und bei diesem erneuten Aufruf das erste Element der Liste wegläßt. Bei jedem Aufruf wird das erste Argument in Kopf X und Rest Y unterteilt und das dritte Argument als Liste, die mit X beginnt und deren Rest eine Variable Y1 ist, instantiiert. Die Komplexität dieser Operationen hängt nicht von n ab. Genauso hängt die Zeit, die benötigt wird, um `append` neu aufzurufen und dann die passende der beiden Klauseln auszuwählen, nicht von n ab. Man kann also sagen, daß die Komplexität von `append` linear ist. `append` braucht n Schritte, deren Komplexität nicht von n abhängt.

`naive_rev` teilt das erste Argument in Kopf und Rest, ruft `naive_rev` für den Rest auf, und hängt dann den Kopf an des Ergebnis von `naive_rev` an. Wenn die eingegebene Liste die Länge n hat, so rufen wir `append` für Listen der Länge $n-1, n-2, \ldots, 2$ und 1 auf. Die Komplexität von `append` ist linear, also ist die Komplexität von `naive_rev`:

$$n-1+n-2+\ldots+2+1$$

oder $n^2/2-n/2$, d. h. die Komplexität ist quadratisch.

Das Prädikat `good_rev` kann genauso wie `append` analysiert werden. Es hat lineare Komplexität. Das zeigt, daß `good_rev` `naive_rev` für lange Listen überlegen ist.

Ein nichtdeterministisches Prolog-Programm

Das zweite Beispiel ist ein nichtdeterministisches Prolog-Programm, d. h. ein Programm, das Backtracking verwendet, um alle Lösungen aufzuzählen.

```
p(X) :- p1(X).
p(X) :- p2(X).
p([]).

p1([a|L]) :- p(L).
p2([b|L]) :- p(L).
```

Das Prädikat `p` instantiiert eine Liste von Prolog-Variablen entweder mit `a` oder mit `b`:

```
?- p([A,B,C,D,E]).
A = a, B = a, C = a, E = a;
A = a, B = a, C = a, E = b;
A = a, B = a, C = b, E = a;
...
```

Wie groß ist die Komplexität in Abhängigkeit von der Länge der Eingabeliste? Da alle Lösungen aufgezählt werden sollen, müssen jedes mal, wenn `p` aufgerufen wird, beide Klauseln von `p` berücksichtigt werden. Jeder Aufruf von `p` hat eine konstante Anzahl von Operationen zur Folge (Instantiierung des Listenkopfes) und danach werden alle möglichen Instantiierungen des Restes der Liste durchgeführt. Alle möglichen Instantiierungen des Restes kommen in der Gesamtmenge aller Instantiierungen zweimal vor, nämlich einmal mit dem Kopf der Liste mit `a` instantiiert und einmal mit dem Kopf mit `b` instantiiert. Daraus folgt, daß sich der Arbeitsaufwand für jedes zusätzliche Element verdoppelt. Die Komplexität beträgt also $2 * 2 * \ldots * 2 = 2^n$, sie ist exponential. Bei dieser Abschätzung wurde der Aufwand, den Prolog treibt, um das Backtracking durchzuführen, nicht berücksichtigt. Beim Backtracken muß sich Prolog den Rückkehrpunkt merken und die Bindungen entsprechender Variablen auflösen. Es gibt maximal 2^n Backtrackpunkte und maximal n Variablen, d. h. dieser Teil der Arbeit ist auch nicht komplexer als $n * 2^n$. Das gesamte Programm hat also exponentiale Komplexität.

Ein normales Programm

Das letzte Beispiel entspricht einer Sortierprozedur, die in einer „normalen“ Programmiersprache geschrieben ist. Es wird angenommen, daß eine Menge von n Zahlen in einem Array aufeinanderfolgend angeordnet sind. Das i-te Element wird mit V(i) bezeichnet.

```
for x := n downto 1 do
  for y := 1 to x-1 do
    if V(y+1) < V(y) then change(V(y+1),V(y))
  endfor
endfor
```

Man muß nicht verstehen, wie das Programm funktioniert, um seine Komplexität zu verstehen. Wenn man annimmt, daß das Finden von V(i) für ein bestimmtes i und das Vergleichen zweier Zahlen in bezug auf n konstante Zeit brauchen, dann haben die Operationen innerhalb der inneren Schleife konstante Komplexität. Die

innere Schleife führt die Operationen x-1 mal aus und x geht von n bis 1. Es ergibt sich:

$$n-1+n-2+\ldots+2+1$$

also eine quadratische Komplexität.

4.5 Komplexität des Erkennens

Festzustellen ob ein String einer bestimmten Menge angehört oder nicht, ist ein Entscheidungsproblem. Im folgenden ist die Komplexität dieses Problems für Maschinen mit freiem Speicherzugriff für Sprachen verschiedenen Grammatiktyps angegeben:

Typ	Komplexität
0	nicht entscheidbar
1	e^n (exponential)
2	n^3 (kubisch)
3	n (linear)

Dabei ist n die Länge des Strings. Diese Angaben beziehen sich auf den ungünstigsten Fall (*worst case*), d. h. es gibt Typ-2-Grammatiken, so daß die Erkennung einiger von diesen Grammatiken definierten Strings kubischer Komplexität hat. Das heißt nicht, daß jede Grammatik dieses Typs nur Strings definiert, deren Erkennung ein Problem von kubischer Komplexität ist. Zum Beispiel sind die Typ-3-Grammatiken eine Teilmenge der Typ-2-Grammatiken, und Strings von Typ-3-Grammatiken sind mindestens in linearer Zeit erkennbar.

Daß die Erkennung von Typ-3-Grammatiken in linearer Zeit möglich ist, kann man sehen, wenn man sich den folgenden Algorithmus ansieht. Dieser Algorithmus gibt an, wie man ein *finite state transition network* durchwandern kann. Dabei soll der zu erkennende String n Elemente haben und das i-te Element w_i heißen.

S_0 := Menge der Startzustände des Netzes
for i := 1 to n do
  S_i := { s | es gibt einen Zustand s_1 in S_{i-1} und eine Kante w_i geht
        von s_1 nach s }
endfor
Wenn S_n einen Endzustand enthält, gib *yes* zurück

sonst *no*.

In S_i sind immer alle Zustände enthalten, in denen man nach dem Lesen des i-ten Wortes sein könnte. In jedem S_i gibt es maximal soviele Elemente wie das Netz Knoten hat. Das hängt nicht von n ab. Der Aufwand, das S_i zu berechnen, ist maximal, wenn man jedes Knotenpaar nehmen und überprüfen muß, ob es eine Kante w_i zwischen ihnen gibt. Die Berechnung von S_i ist also eine Operation, die in konstanter Zeit ausgeführt werden kann, d. h. die Zeit, die benötigt wird, um S_i zu berechnen hängt nicht von n ab. Also ist der Algorithmus insgesamt linear.

Auf die Komplexität der Typ-2-Grammatiken wird in Kapitel 10 eingegangen. Typ-1-Grammatiken – kontextsensitive Grammatiken also – beschreiben Mengen, die rekursiv sind. Das Erkennungsproblem für rekursive Mengen ist entscheidbar. Sprachen vom Typ 0 sind rekursiv aufzählbar aber nicht allgemein rekursiv. Das heißt, daß man für Typ 0 Sprachen im allgemeinen nur eine Prozedur schreiben kann, die bei Strings der Sprache nach einer gewissen Zeit *ja* ausgibt und ansonsten eventuell nicht terminiert.

Diese Ergebnisse erklären, warum Informatiker Grammatiken verarbeiten wollen, deren Typ größer oder gleich 2 ist (2 oder 3).

4.6 Komplexität des Parsens

Die Aufzählung aller Analysen eines Satzes ist immer komplexer als das Erkennen des Satzes, weil es sogar für Typ-3-Sprachen Strings gibt, bei denen die Anzahl der Analysen exponential mit der Länge des Strings steigt:

$$\begin{aligned} &S \rightarrow X \\ &S \rightarrow Y \\ &S \rightarrow \\ &X \rightarrow a\ S \\ &Y \rightarrow a\ S \end{aligned} \tag{4.2}$$

wobei a ein Terminalsymbol ist. Ein String der Länge n hat 2^n verschiedene Analysen, legt man die angegebene Grammatik zugrunde. Parsen ist also im ungünstigsten Fall exponential.

4.7 Idealisierungen

In der Theorie von Berechenbarkeit und Komplexität werden einige Idealisierungen angenommen, die in der Praxis nicht immer vorliegen.

1. *Die Wahl der Parameter* Es wird angenommen, daß man eine bestimmte Menge Parameter berücksichtigt, die in die Berechnung der Komplexität eingehen. Eine schlechte Auswahl dieser Parameter hat irreführende Ergebnisse zur Folge. Zum Beispiel haben wir durch Angabe eines Algorithmus bewiesen, daß die Erkennung von Typ-3-Sprachen in linearer Zeit möglich ist. Wir haben dabei aber nur die Länge des Strings als Parameter angenommen. Berücksichtigt man außerdem die Grammatikgröße (die Anzahl der Knoten im FSTN), dann stellt man fest, daß die Zeit auch noch quadratisch von diesem Parameter abhängt. Ob eine solche Idealisierung sinnvoll ist, hängt davon ab, ob man große Grammatiken benutzen will. Große Grammatiken bekommt man mitunter, wenn man Grammatiken automatisch generiert.

2. *Die Rolle von beliebig großen Problemen* Es wird angenommen, daß die Geschwindigkeit eines Algorithmus bei der Anwendung auf kleine Probleme nichts über seine Leistungsfähigkeit aussagt, und daß man beliebig große n in Betracht ziehen muß, um seine wirkliche Leistungsfähigkeit zu ermitteln. Aber in der Praxis kann man eine Obergrenze für Satzlängen annehmen. Sätze mit über 500 Wörtern sind für die Sprachverarbeitung irrelevant.

   Andererseits kann man sagen, daß es keine prinzipielle Grenze für die Länge natürlichsprachlicher Sätze gibt, und daß menschliche Parser (was immer das ist) und künstliche Parser deshalb so arbeiten müssen, als wären sie in der Lage, beliebig lange Sätze zu verarbeiten. Wenn nur kürzere Sätze benutzt werden, dann nur deshalb, weil die Fähigkeiten menschlicher Wesen begrenzt sind.

3. *Die Rolle der Analyse des ungünstigsten Falls* Die Ergebnisse der Komplexitätsanalysen waren jeweils für die ungünstigsten Grammatiken der entsprechenden Klassen und für die ungünstigsten Strings gültig. Man kann argumentieren, daß natürliche Sprachen nicht den Typ-1-Sprachen entsprechen, obwohl es Beispiele gibt, die zeigen, daß natürliche Sprachen eine Obermenge der Typ-2-Sprachen sind. Die Linguisten suchen derzeit noch nach Sprachklassen, die nicht die gesamte Klasse der Typ-1-Sprachen einschließen.

Kontrollfragen

1. Wozu braucht man das Konzept der Turing-Maschine?

2. Was ist ein entscheidbares Problem?
3. Ist logarithmische Komplexität besser als lineare?
4. Warum ist die Komplexität für das Parsen schlechter als für das Erkennen?

übungsaufgaben

1. * Wie hoch ist die Komplexität eines Tests, ob eine Formel der Aussagenlogik eine Tautologie ist, wenn man für den Test eine Wahrheitswertetabelle konstruiert?
2. * Diskutieren Sie die Komplexität der folgenden Prolog-Programme, die jeweils das kleinste Element einer Menge bestimmen sollen.

```
% 1. waehle ein Element und ueberpruefe ob es das
%    kleinste ist
smallest([X|List],X) :- less_than_all(X,List),!.
smallest([_|List],X) :- smallest(List,X).

   less_than_all(X,[]).
   less_than_all(X,[First|Rest]) :- X < First,
        less_than_all(X,Rest).


% 2. vergleiche jedes Element mit dem kleinsten
%    Element des Rests
smallest([X],X).
smallest([X|List],S) :- smallest(List,S1),
        smaller(X,S1,S).

   smaller(X,Y,X) :- X<Y.
   smaller(X,Y,Y) :- Y<X.
```

Kapitel 5

Eine einfache Grammatik für das Englische

5.1 Das Schreiben von Grammatiken

Beim Entwickeln einer Grammatik braucht man eine passende Menge grammatischer Kategorien um Wörter und Konstituenten, die beschrieben werden sollen, zu klassifizieren. Die mnemonischen Namen, die den Kategorien gegeben werden, sind willkürlich. Wichtig ist nur ihre Verwendung in den Regeln und im Lexikon. Würden wir in der in Kapitel 2.4 Grammatik das Nichtterminalsymbol *ziffer* an allen Stellen durch *tomatenbrot* ersetzen, so würde die Grammatik immernoch dieselbe Menge von Sätzen beschreiben. Die Grammatik wäre allerdings weniger leicht verständlich, und wahrscheinlich würde der Schreiber einer hinreichend komplizierten Tomatenbrot-Grammatik sie nach einem halben Jahr Arbeitsunterbrechung selbst nicht mehr verstehen. Es gilt also für das Schreiben von Grammatiken genauso wie für das Schreiben von Programmen: größte Disziplin bei der Vergabe von Namen.

Man könnte vielleicht annehmen, daß es eine bestimmte Menge von Kategoriesymbolen für die englische Sprache oder sogar DIE Grammatik des Englischen gibt. Weder das eine noch das andere ist der Fall. Obwohl es traditionelle Begriffe wie Nomen, Verb usw. gibt, gibt es keine offizielle Übereinkunft darüber, wie diese Begriffe zu verwenden sind. Es liegt also in der Verantwortung des Grammatikschreibers, diese Begriffe in konsistenter Weise zu benutzen. Um Verwirrungen zu vermeiden, ist es natürlich angebracht, die traditionellen Begriffe nicht entgegen den informalen Vorstellungen von ihnen zu gebrauchen. Die Menge der Grammatikkategorien, die in der Schule gelehrt werden, ist sehr ungenau, nicht formalisiert und nicht genau genug, um eine große, präzise und formale Grammatik einer natürlichen Sprache schreiben zu können, da man bei der Ausarbeitung

einer formalen Grammatik stärker differenzieren können muß, als das mit einem Dutzend (exklusiver) Klassen wie *Nomen*, *Verb* oder *Adjektiv* möglich ist.

Die Aufgabe des Grammatikschreibers ist es, herauszufinden, welche Wortarten und welche Konstituenten es in der entsprechenden Sprache gibt und wie sie zusammenhängen. Die Analyse von Daten, das Herausfinden von Regelmäßigkeiten ist die eigentliche Arbeit, die Namen für die verwendeten Kategorien zu vergeben ist das kleinere Problem.

5.2 Das Finden von Regularitäten

Was heißt das alles nun für jemanden, der ein lauffähiges Sprachverarbeitungssystem schreiben will? Es gibt viele Grammatiken, die mehr oder weniger gut für die entsprechende Applikation geeignet sind. Muß man eine eigene Grammatik schreiben, so soll diese so kurz und elegant wie möglich sein und so viel als möglich von der betreffenden Sprache beschreiben. Deshalb muß die Grammatik Regelmäßigkeiten der Sprache widerspiegeln. Es gibt eine Anzahl von Tests, die man benutzen kann, um benutzbare und erweiterbare Grammatiken zu schreiben.

- *Substituierbarkeit*: Wenn man einen Teil einer Phrase durch etwas anderes ersetzen kann und die Phrase dann noch grammatisch ist, kann man eine ähnlichkeit zwischen dem Aus- und dem Eingetauschten annehmen. Wenn solch eine Substitution in vielen verschiedenen Kontexten möglich ist, kann man annehmen, daß die beiden gegen einander austauschbaren Phrasen derselben Kategorie angehören. So könnte man z. B. festlegen, daß alles, was man in einem wohlgeformten Satz, der das Wort *John* enthält, für *John* einsetzen kann, eine Nominalphrase ist. Das funktioniert aber nur bis zu einem gewissen Punkt. Setzt man z. B. *John's friends* für *John* in den Satz *John was really mad.* ein, so erhält man einen ungrammatischen Satz, obwohl man *John's friends* in die Klasse der Nominalphrasen einordnen kann.

- *Koordination*: Ein weiterer Test besagt: Dinge, die sich koordinieren lassen, sind Konstituenten.

- *Semantik*: Wenn zwei Phrasen dieselbe Art von Bedeutung haben (z. B. wenn sich beide auf physikalische Objekte oder beide auf Handlungen beziehen), kann man ihnen dieselbe syntaktische Kategorie geben. Wenn eine Wortsequenz die Bedeutung einer anderen Wortsequenz modifiziert oder erweitert, dann kann es sinnvoll sein, sie als selbständige Phrasen zu behandeln, die an einem bestimmten Punkt im Phrasenstrukturbaum verknüpft werden. Manchmal will man semantische Mehrdeutigkeiten dadurch beschreiben, daß es mehrere syntaktisch verschiedene Analysen gibt. Es gibt

viele Möglichkeiten, wie semantische Faktoren den Aufbau einer Grammatik beeinflussen können. Wir werden im Teil 2 der Vorlesung darauf eingehen.

5.3 Die vier wichtigsten lexikalische Kategorien

Die vier wichtigsten lexikalischen Kategorien sind Nomen, Verb, Adjektiv und Präposition. Elemente der lexikalischen Kategorien sind einzelne Wörter.

5.3.1 Nomina

Nomen heißt eigentlich *Name*, aber das besagt überhaupt nichts, weil ein Nomen ziemlich abstrakt sein kann und manche Nomina nicht das sind, was man sich unter einem Namen vorstellt. Beispiele: *snow*, *unicorn*, *sideboard*, *measles*. Gewöhnliche Nomina haben typischerweise verschiedene Plural und Singularformen (z. B. *carton*, *cartons*). Es gibt aber auch Nomina, die nicht beide Formen haben (z. B. *sheep*, *deer*) und die Klasse der Massennomina, z. B. *furniture*, *toast*. Im folgenden ist nur noch von Zählnomina die Rede. Das einzige Merkmal, das wir zur Klassifizierung von Nomina brauchen ist also *number*.

num = { sing, plur }

5.3.2 Adjektive

Traditionell ist ein Adjektiv ein Beschreibungswort. Es kann die Bedeutung eines Nomens modifizieren. Beispiele: *blue*, *large*, *fake*, *main*. Adjektive haben eine Grundform, eine Adverbform, einen Komperativ und einen Superlativ (*great*, *greatly*, *greater* und *greatest*). Es gibt aber auch Ausnahmen. Manche Adjektive haben die letzten beiden Formen nicht (*unique*, *uniquely*) und andere Adjektive bilden diese Form mit *more* und *most* (*more beautiful*, *most beautiful*).

5.3.3 Verben

Traditionell ist ein Verb ein Tätigkeitswort, aber diese Bezeichnung ist nicht besonders gut, weil es auch Verben gibt, die eher passives Verhalten beschreiben. Außerdem beschreibt das Wort *Tätigkeit* sicher auch eine Tätigkeit. Es ist aber als Nomen einzuordnen. Zu diesem Problem und zur Definition von Wortklassen siehe auch (Engel, 1977, Kapitel 2.6). Beispiele für Verben: *run*, *know*, *be*, *have*. Es gibt zehn verschiedene Verbformen. Ein Verb kann aber maximal 8 morphologisch verschiedene Formen haben. Für die Beschreibung der unterschiedlichen

Merkmale braucht man das Merkmal *vform*.

vform = {fin, bse, inf, prp, pas, psp}

Finite Verben sind Verben, die in einem einfachen Satz mit einem Verb auftauchen können. Verben mit anderen Verbformen müssen mit finiten Verben kombiniert werden, um einen Satz bilden zu können. Ob ein Verb die Form *psp* – past participle – oder *pas* – passiv participle – hat, hängt von der Konstruktion ab, in der es verwendet wird (*has baked, was baked*).

Beschreibung	vform	be	write	bake
base form	bse	be	write	bake
infinitive	inf	to be	to write	to bake
finite present 1sing	fin	am	write	bake
finite present 3sing	fin	is	writes	bakes
finite present other	fin	are	write	bake
finite past sing	fin	was	wrote	baked
finite past plur	fin	were	wrote	baked
past participle	psp	been	written	baked
present participle	prp	being	writing	baking
passiv participle	pas	???	written	baked

Hilfsverben sind Verben, die zur Bildung komplexerer Konstruktionen benutzt werden. Sie stehen am Anfang der Verbphrase (*be, have, do, can, will, may, might, could, must, shall, should*). Die letzten acht werden auch Modalverben genannt. Das Wort *to*, das zur Bildung von Infinitiven benutzt wird, kann man im Englischen auch zu den Hilfsverben zählen.

5.3.4 Präpositionen

Präpositionen können vor Nominalphrasen stehen und mit dieser eine Präpositionalphrase bilden. Beispiele: *in, by, of, to*.

5.4 Syntaktische Kategorien

Die vier im Abschnitt 5.3 vorgestellten Kategorien zeichnen sich dadurch aus, daß sie im allgemeinen den wichtigsten Bestandteil von größeren phrasalen Kategorien bilden. Z. B. kann man mit Nomina Nominalphrasen und mit Verben Verbphrasen bilden. Es gibt nicht nur lexikalische und phrasale Kategorien (Maximalprojektionen) sondern auch noch Zwischenstufen. Um das zu beschreiben, benutzt

man in vielen syntaktischen Theorien *Bar-Level* (Zur $\overline{\text{X}}$-Theorie siehe insbesondere (Jackendoff, 1977)). Lexikalische Kategorien (z. B. Nomina) haben das Bar-Level 0, phrasale Kategorien haben Bar-Level 2 oder 3 und die Zwischenstufen haben das Level 1. Dieser Ansatz wird X-bar-Theorie genannt. Im folgenden wird ein maximales Bar-Level von 2 angenommen.

bar ={0,1,2}

Ein Vorteil dieser Notation ist, daß die Regularitäten in den Strukturen verschiedener Phrasen ausgedrückt werden. Die folgenden Regeln gelten für Nominal-, Verb-, Adjektiv- und Präpositionalphrasen. X steht jeweils für das Nomen, das Verb, das Adjektiv bzw. für die Präposition.

$$\begin{aligned} &X(2) \rightarrow Spezifikator\ X(1) \\ &X(1) \rightarrow X(1)\ Adjunkt \\ &X(1) \rightarrow Adjunkt\ X(1) \\ &X(1) \rightarrow X(0)\ Komplemente \end{aligned} \tag{5.1}$$

Ein Spezifikator ist ein Teil einer Phrase, der – wenn überhaupt – nur einmal und zwar zu Beginn einer Phrase auftaucht. Arbeitet man die Regeln von links nach rechts und vom höheren zum niedrigeren Bar-Level ab, so kann man nach dem Spezifikator mehrere optionale Modifikatoren finden und dann die lexikalische Kategorie mit ihren Komplementen. Modifikatoren und Komplemente werden weiter unten noch genauer erklärt.

Ein X auf der rechten Seite einer Regel hat entweder das gleiche Bar-Level wie auf der linken Seite oder ein kleineres. Stattet man die Regeln mit Merkmalen wie *num*, *vform* oder *pform* aus, dann müssen die Werte dieser Merkmale beim X auf der rechten Seite denen des X auf der linken Seite gleichen. Das X auf der rechten Seite wird Kopf (*head*) der Phrase genannt und die identischen Merkmale Kopfmerkmale (*head features*).

5.4.1 Nominalphrasen

Jede Phrase, die in einem Satz Subjekt oder Objekt sein kann, ist eine Nominalphrase. (z. B. *the big red block*, *most of the first three coaches*). Nominalphrasen werden normalerweise benutzt, um auf bestimmte Objekte zu referieren. Es gibt aber auch dummy-NPs wie *there* und *it*:

(5.2) a. There is a dog howling in the yard.
b. It is impossible for me to see you now.

Nominalphrasen werden durch Regeln der folgenden Form beschrieben:

$$
\begin{aligned}
&n(2, Per, Num) \rightarrow pronoun(Per, Num)\\
&n(2, Per, sing) \rightarrow proper_noun(Per, sing)\\
&n(2, Per, Num) \rightarrow det(Num)\ n(1, Per, Num)\\
&n(2, Per, plur) \rightarrow n(1, Per, plur)\\
&n(1, Per, Num) \rightarrow pre_mod\ n(1, Per, Num)\\
&n(1, Per, Num) \rightarrow n(1, Per, Num)\ post_mod\\
&n(1, Per, Num) \rightarrow n(0, Per, Num)
\end{aligned}
\tag{5.3}
$$

Die ersten zwei Regeln beschreiben zwei Wortklassen, die selbst Nominalphrasen sind. Pronomina werden benutzt, um auf Dinge zu referieren, die vorher erwähnt wurden. Pronomina können sich allerdings auch auf Dinge beziehen, die nicht erwähnt wurden (deiktisch) oder die erst nach der Benutzung des Pronomens im Diskurs auftauchen (Kataphora, kommen z. B. in der Literatur vor). Pronomina sind die einzigen Nominalphrasen des Englischen, die Kasus tragen.

(5.4) a. He gave the book to John.

b. John gave him the book.

c. John gave his book to Mary.

Entsprechend ihres Kasus kann man die Pronomina in unterschiedlichen Funktionen (Subjekt, Objekt, Besitzer) verwenden. Der Vollständigkeit halber müßten Nominalphrasen also noch ein Kasus-Merkmal haben.

Eigennamen (*proper nouns*) bezeichnen bestimmte Dinge, wohingegen normale Nomina Klassen von Dingen bezeichnen. Beispiele für Eigennamen sind: *John*, *Scotland*.

Determinatoren sind Spezifikatoren von Nominalphrasen. Zu den Determinatoren gehören Artikel (*the*, *a*, *an*), Possessivpronomina (*his*, *her*), Quantoren (*many*, *some*) possessive Nominalphrasen (*my father's*) und Demonstrativa (*this*, *that*).

Modifikatoren, die Nomina modifizieren können, stehen entweder vor dem Nomen (z. B. Adjektive) oder danach (z. B. Präpositionalphrasen und Relativsätze). Relativsätze sind Sätze, die das Nomen, dem sie folgen, näher bestimmen. Sie können mit einem Relativpronomen (*who*, *which*, *that*, *where* oder *when*) oder ohne beginnen. Das Relativpronomen kann auch in einer Präpositionalphrase enthalten sein (*to whom*). Ein Relativsatz ist in seiner Form dem normalen finiten Satz ähnlich, nur das irgendwo eine Nominalphrase fehlt. Das wird im Kapitel 7 genauer erklärt.

5.4.2 Verbphrasen

Eine Verbphrase besteht aus einem Verb und den dazugehörigen Objekten (*gave a parcel to the clerk, runs*). Verbphrasen werden durch Regeln wie die in (5.5) beschrieben.

$$\begin{aligned}
&v(2, Vform, Num, Per) \rightarrow v(1, Vform, Num, Per)\\
&v(1, Vform, Num, Per) \rightarrow adv\ v(1, Vform, Num, Per)\\
&v(1, Vform, Num, Per) \rightarrow v(1, Vform, Num, Per)\ verb_post_mods\\
&v(1, Vform, Num, Per) \rightarrow v(0, Vform, Num, Per)
\end{aligned} \tag{5.5}$$

Verbphrasen können durch Adverbien (*beautifully, quickly*) und Präpositionalphrasen (*on the desk*) modifiziert werden.

Vform ist ein Kopfmerkmal, es wird vom Hauptverb nach oben, d.h. zum nächsthöheren Bar-Level weitergereicht. Mit Hilfe des *Vform*-Merkmals kann man ausdrücken, daß in einem bestimmten syntaktischen Kontext eine Verbphrase gebraucht wird, deren Hauptverb eine bestimmte Form haben muß.

5.4.3 Präpositionalphrasen

Eine Präpositionalphrase mit einer bestimmten Präposition kann z. B. von einem Verb verlangt werden. Das Verb spezifiziert, welche Präposition in der Präpositionalphrase vorkommen muß. Um das ausdrücken zu können, braucht man das Merkmal *Pform*, dessen Wert ein Element aus der Menge der Oberflächenformen der Präpositionen ist:

pform = {of, to, with, about, ...}

Präpositionalphrasen werden durch Regeln der Form

$$\begin{aligned}
&p(2, Pform) \rightarrow p(1, Pform)\\
&p(1, Pform) \rightarrow adv\ p(1, Pform)\\
&p(1, Pform) \rightarrow p(0, Pform)\ n(2, _)
\end{aligned} \tag{5.6}$$

beschrieben. Das '_'-Zeichen steht dabei für einen nicht spezifizierten Wert. An seiner Stelle können alle für das entsprechende Merkmal möglichen Werte stehen.

5.4.4 Adjektivphrasen

Adjektivphrasen werden durch Regeln der Form

$$\begin{aligned}
&a(2) \rightarrow deg\ a(1)\\
&a(1) \rightarrow adv\ a(1)\\
&a(1) \rightarrow a(0)
\end{aligned} \tag{5.7}$$

beschrieben. Normalerweise enthalten Adjektivphrasen nur ein einzelnes Adjektiv. Es ist aber möglich eine Beschreibung der Gradation und bestimmte Adverbien anzufügen (*very commonly used*).

5.5 Sätze

Die Regel für einen englischen Satz ist:

$$s(Vform) \rightarrow n(2, Per, Num)\ v(2, Vform, Per, Num) \tag{5.8}$$

Das Subjekt muß mit dem Verb in Person und Numerus übereinstimmen. Das ist dadurch gewährleistet, daß Person und Numeruswerte in allen Verbphrasenregeln als Kopfmerkmale nach oben gereicht werden. Sie sind dann an der v(2)-Projektion vorhanden und man kann in der Satzregel auf die Werte Bezug nehmen.

5.6 Komplemente und Modifikatoren

Was auf welchem Bar-Level eingeführt wird, kann von Grammatik zu Grammatik unterschiedlich sein. Die allgemeine Idee ist jedoch relativ klar. Ein bestimmter Lexikoneintrag verlangt bestimmte Komplemente, die im Gegensatz zu Modifikatoren obligatorisch sind und nur einmal auftreten dürfen. Modifikatoren können normalerweise Kategorien einer bestimmten Klasse modifizieren, wogegen Komplemente meist wortspezifisch sind. Die Art der möglichen Komplemente eines Wortes wird im Lexikon mit Hilfe des *Subcat*-Merkmals spezifiziert. Das *Subcat*-Merkmal ist für die Auswahl bzw. für die Verwendung einer bestimmten Grammatikregel zuständig.

subcat = {intrans, trans, emotion, exchange_of_views,...}

Die ersten beiden Werte sind nach traditionellen Verbkategorien benannt. Ein intransitives Verb (z. B. *dream*) verlangt keine Komplemente.[1] Transitive Verben dagegen (z. B. *hit*) erwarten ein Objekt – eine Nominalphrase. Aber es gibt noch viele andere Möglichkeiten für die Kategorien von Verb-, Nomen-, oder Adjektivkomplementen.

Statt des X-bar-Levels 0 kann man auch das *Subcat*-Merkmal schreiben, da nur lexikalische Kategorien das *Subcat*-Merkmal haben. Es ist kein Kopfmerkmal.

[1] Diese Auffassung weicht von der z. B. in der HPSG vertretenen Auffassung ab, daß das Subjekt auch vom Verb abhängt und im Lexikon spezifiziert wird.

Man hat dann also eine Menge Regeln der folgenden Art:

$$
\begin{array}{lll}
n(1,Num) \rightarrow n(exchange_of_views,Num)\ p(2,with)\ p(2,about) & \%argument & \\
n(1,Num) \rightarrow n(plan,Num)\ v(2,inf,_) & \%plan & \\
a(1) \rightarrow a(prop)\ s(fin) & \%afraid & \\
a(1) \rightarrow a(emotion)\ p(2,of) & \%fond & \\
v(1,Vform,Num) \rightarrow v(ditrans,Vform,Num)\ n(2,_)\ n(2,_) & \%give & (5.9) \\
v(1,Vform,Num) \rightarrow v(obj_raising,Vform,Num)\ n(2,_)\ v(2,inf,_) & \%prefer & \\
v(1,Vform,Num) \rightarrow v(perfect,Vform,Num)\ v(2,psp,_) & \%have & \\
v(1,inf,_) \rightarrow v(to,inf)\ v(2,bse,_) & \%to & \\
p(1,Pform) \rightarrow p(takes_pp)\ p(2,of) & \%out &
\end{array}
$$

In den Regeln sind die Komplemente immer Maximalprojektionen, d.h. phrasale Kategorien mit dem Bar-Level 2. Eine andere wichtige Sache ist die, daß der Kopf die Menge der für ihn in Frage kommenden Komplemente einschränkt. Ein bestimmter Kopf ergibt nur mit einer bestimmten Menge von Komplementen eine wohlgeformte und sinnvolle Phrase. Der Kopf selegiert nicht nur nach syntaktischen sondern auch nach semantischen Kriterien. z. B. verlangt das Verb *kill* ein lebendes Objekt, wobei das Subjekt belebt oder unbelebt sein kann. *Murder* verlangt ein Subjekt, das für seine Handlung verantwortlich sein kann.

Das Verb *have* wird als Verb analysiert, das ein anderes Verb in der *psp*-Form verlangt. Andere Hilfsverben können ähnlich beschrieben werden.

Ein Wort kann mit verschiedenen Subkategorisierungsmustern auftreten. Es gibt z. B. eine Klasse von Verben wie *give*, die in der Form *give y x* aber auch in der Form *give x to y* vorkommen können. Diese Alternativen kann man einfach im Lexikon kodieren, oder aber man hat irgendeine Form lexikalischer Regeln, die dafür sorgen, daß beide Varianten aus einem einzigen Merkmal erzeugt werden.

5.7 Unzulänglichkeiten dieser Grammatik

Bis jetzt haben wir die Hauptbestandteile einer kontextfreien Grammatik für ein Fragment des Englischen angegeben. Die Anzahl der Sätze, die man mit der Grammatik beschreiben kann, wurde durch die Einführung einer endlichen Anzahl von Merkmalen mit endlich vielen Werten erweitert. Es gibt aber viele Dinge, die durch die Grammatik nicht korrekt modelliert werden. Auf einige wird in späteren Kapiteln noch eingegangen werden.

5.7.1 Morphologie

In unserer Grammatik müssen vollständig flektierte Wörter im Lexikon stehen. Das heißt, es gibt verschiedene, von einander unabhängige Lexikoneinträge für

bake, *bakes*, *baking* usw. Alle *Vform*- und *Subcat*-Werte sind in einem separaten Lexikoneintrag kodiert.

Es gibt aber allgemeine Prinzipien, die beschreiben, wie die verschiedenen Formen regulärer Verben oder Nomina gebildet werden können (z. B. wird bei normalen Nomina einfach die Endung *s* angehängt, um den Plural zu bilden). In einem guten System müßte man solche Regularitäten berücksichtigen, um ein großes, redundantes und schlecht wartbares Lexikon zu vermeiden.

Außerdem gibt es keine *derivationelle Morphologie*. Unter derivationeller Morphologie versteht man die semantische Ableitung eines Wortes aus einem oder mehreren anderen. Das Wort *computability* ist z. B. aus *compute+able+ity* zusammengesetzt.

5.7.2 Konstituentenstellung

Mit den Regeln kontextfreier Grammatiken kann man nur Symbole kombinieren, die genau in der Reihenfolge stehen, wie sie in den Regeln angegeben sind. Für Grammatiken des Englischen ist das kein Problem, da die Wortreihenfolge im Englischen relativ festgelegt ist. Es gibt jedoch Sprachen mit sehr viel freierer Wortstellung, wie z. B. das Deutsche. Im Deutschen gibt es sogar diskontinuierliche Komplemente. Das heißt, Konstituenten von Maximalprojektionen können abwechselnd mit Konstituenten anderer Maximalprojektionen auftreten.

(5.10) weil er_i es_j ihm_k zu lesen_j versprochen_k hat_i.

5.7.3 Subkategorisierungsregelmäßigkeiten

In unserer Grammatik werden mögliche Komplemente mit Hilfe des *Subcat*-Merkmals beschrieben, das eine bestimmte Regel der Form $v(1) \rightarrow v(subcat)\ldots$ auswählt, die zu dem Verb paßt.

Es spricht nichts dagegen, Verben mehrfach mit verschiedenen *Subcat*-Werten ins Lexikon einzutragen, wenn die verschiedenen Subkategorisierungsmöglichkeiten nur für bestimmte Verben auftreten.

Es könnte z. B. zwei verschiedene Einträge für *give* geben.

$$\begin{array}{ll} v(1) \rightarrow v(ditrans,_)\ n(2)\ n(2) & \%\ \textit{give a dog a bone} \\ v(1) \rightarrow v(trans_plus_to_pp,_)\ n(2)\ p(2,to) & \%\ \textit{give a bone to a dog} \end{array} \tag{5.11}$$

Es gibt aber andere Änderungen des *Subcat*-Merkmals, die für fast alle Verben gleich sind. Wenn man für jede Regel mit v(1) auf der linken Seite eine Passivregel

schreiben müßte, so könnte man mit Recht behaupten, der Grammatikformalismus sei nicht ausdrucksstark genug.

$$\begin{array}{l} v(1) \rightarrow v(trans, fin)\ n(2) \ldots \\ v(1) \rightarrow v(intrans, pas) \ldots \end{array} \tag{5.12}$$

Eine Möglichkeit diese Aktiv-Passiv-Regelmäßigkeiten zu behandeln, wird in Kapitel 15 diskutiert.

5.7.4 Fernabhängigkeiten (*Nonlocal Dependencies*)

Solche Aktiv-Passiv-Alternativen sind lokale Erscheinungen (nur die Art der Komplemente, die beim Verb stehen, ändert sich). Es ist möglich, diese Phänomene mit einer kontextfreien Grammatik zu beschreiben. Es gibt aber auch eine Klasse von Phänomenen, die nicht lokal sind, und bei denen es Abhängigkeiten über eine unbegrenzte Anzahl von Satzgrenzen hinweg gibt. Beispiel dafür sind die folgenden Sätze:

(5.13) a. This book, I could never manage to persuade my students to read.

b. The college that I expected John to want Mary to attend is very expensive.

c. What do you believe Mary told Bill that John had said?

d. This film is easy for me to persuade the children not to see.

In (5.13a) handelt es sich um eine Topikalisierung, in (5.13b) um Relativierung, in (5.13c) um die Bewegung der Fragepronomen und in (5.13d) um *tough*-Movement.

Man beschreibt diese Abhängigkeiten mit Hilfe einer Folge lokaler Abhängigkeiten. Dabei wird Information Schritt für Schritt durch den Syntaxbaum weitergegeben. Das wird in Kapitel 7 genauer erklärt werden.

5.7.5 Koordination

Man sagt etwas vereinfacht, daß man gleiche Konstituenten durch Koordination verbinden kann. Wir müßten also zu unserer Grammatik eine Menge Regeln hinzufügen, die das für jede Form von Konstituenten auf jedem Bar-Level ausdrückt. Das würde die Grammatik erheblich vergrößern und unübersichtlich machen. Außerdem gibt es auch Fälle, in denen Satzteile durch Konjunktionen verbunden werden, die keine Konstituenten sind. Darauf wird in Kapitel 12 genauer eingegangen.

Kontrollfragen

1. Geben Sie Beispiele für englische Verben, Nomina, Adjektive, Präpositionen und ein Hilfsverb an.

2. Welche Bar-Level haben folgende Kategorien in der X-Bar-Theorie: Nomina, Nominalphrasen, Verben, Verbphrasen, Präpositionen, Präpositionalphrasen, Adjektive, Adjektivphrasen?

3. Welche Phrase ist in dem Satz *John saw Mary.* das Subjekt und welche das Objekt?

4. Welche Werte kann das *Vform*-Merkmal annehmen?

5. Geben Sie Beispiele für mögliche Verb-, Nomen- und Adjektivkomplemente.

Übungsaufgaben

1. Schreiben Sie Beispielphrasen auf, die durch die oben angegebenen Regeln beschrieben werden.

2. * Welche Komplemente verlangen die folgenden Verben?

   asked, preferred, condescended, promised, tried, considered, accepted, forced, expected, wanted, believed, hoped

   Das kann man herausfinden, indem man sich Sätze aufschreibt, in die diese Verben passen:

   (5.14) a. John . . . to succeed.
   b. John . . . his friend to be careful.
   c. Nobody was . . . to be there.

   Schreiben Sie Grammatikregeln mit den entsprechenden Subkategorisierungsmerkmalen.

3. ** Schreiben Sie Grammatikregeln für andere Hilfsverben, die analog zu der für *have* angegebenen sind. Verwenden Sie dabei eigene *Subcat*-Merkmale.

4. ** Die folgenden Phrasen werden normalerweise als Nominalphrasen bezeichnet:

(5.15) a. the many hooligans
b. some of the hundred voters
c. more than seven people
d. many loyal voters

die in (5.16) dagegen nicht:

(5.16) a. * many the hooligans
b. of the hundred many voters
c. more voters loyal

Fertigen Sie eine Liste aller Möglichkeiten für die Bildung von Nominalphrasen an, die die folgenden lexikalischen Kategorien verwendet:

- Adjektive: *loyal, green, happy, ...*
- Nomina: *voters, hooligans, people, apples, ...*
- Quantoren: *some, many, all, ...*
- Artikel: *a, the, ...*
- Determinatoren: *the, a, his, her, its, ...*
- Präposition: *of*

Die Reihenfolge Determinator-Adjektiv-Nomen ist z. B. eine Möglichkeit. Sie werden feststellen, daß die angegebenen Klassen zu ungenau sind. Es gibt Sequenzen in denen *the* aber nicht *a* auftauchen darf. *Many* verhält sich anders als *some*. Verfeinern sie die Klassen wo nötig.

Hausaufgabe

3. Schreiben Sie eine X-Bar-Grammatik in DCG-Form, die dem in diesem Kapitel vorgestellten Ansatz entspricht und die folgenden Sätze akzeptiert:

(5.17) a. Mary gives John a book.
b. Mary has given John a book.
c. Mary is fond of John.
d. John is very fond of Mary.
e. John has forced Mary to go.
f. Mary forces John to go.
g. Mary forces John to be fond of the town.

und folgende Sätze nicht akzeptiert:

(5.18) a. * Mary has gives John a book.
b. * Mary gives John to go.
c. * John has forces Mary to go.
d. * Mary forces John fond of Helen.

Kapitel 6

Definite Clause Grammars als Grammatikformalismus

6.1 Die DCG Notation

Der Formalismus der *Definite Clause Grammars* (DCG) stellt Mittel zur Verfügung, die über die der kontextfreien Grammatiken hinausgehen. Für eine Grammatik, die den kontextfreien Regeln

$$\begin{aligned} &a \rightarrow bcd \\ &a \rightarrow \end{aligned} \tag{6.1}$$

entspricht, wobei *a*, *b* und *d* Nichtterminalsymbole sind und *c* ein Terminalsymbol ist, schreibt man:

```
a --> b, [c], d.
a --> [].
```

Mit DCGs kann man Nichtterminalsymbolen Merkmale zuordnen, und man kann die übereinstimmung dieser Merkmale bei verschiedenen Konstituenten ausdrükken. Hierbei ist wichtig, daß gleiche Merkmalnamen gleichen Werten entsprechen, und daß man bei Konstituenten, die mehrere Merkmale haben (z. B. *Num* und *Cas*), die Merkmale immer in einer bestimmten Reihenfolge vorfindet.[1]

```
s --> np(Num), vp(Num).
```

Werte von Merkmalen können beliebige Prolog-Terme sein. Sie können dazu benutzt werden, strukturelle Information aufzubauen. Parse-Bäume kann man z. B. wie folgt konstruieren:

[1] Die Argumente in Prolog-Termen und demzufolge auch in DCG-Termen sind geordnet.

```
s(s(NP,VP)) --> np(NP),vp(VP).
```

Dabei muß die interne Struktur der Nominalphrase in `NP` und die der Verbphrase in `VP` repräsentiert sein.

Soweit zu den grundlegenden Ausdrucksmöglichkeiten innerhalb des DCG-Formalismus. Es gibt noch einige andere, die im folgenden aufgezählt seien.

- Die Einheiten auf der rechten Seite einer Regel können sowohl Disjunktionen (getrennt durch Semikolon) als auch Konjunktionen (getrennt durch Kommata) sein. Wenn die Skopusverhältnisse unklar sind, werden runde Klammern verwendet.

  ```
  adjectives --> []; (adjective, adjectives).
  ```

- Variablen und sogar beliebige Terme können auf der rechten Seite von Regeln als Terminale und Nichtterminale genutzt werden.[2] Das ermöglicht es, Meta-Level-Konstruktionen zu definieren, in denen die Werte von Merkmalen Terminalsymbole sind. Z. B.:

  ```
  zero_or_more(X) --> [].
  zero_or_more(X) --> X, zero_or_more(X).

  % pick the ball up
  vp --> verb(Particle), np, [Particle].
  ```

- Dem Nichtterminal auf der linken Seite kann eine Menge von Terminalsymbolen folgen, so daß man in der Lage ist, Regeln der Form

  $$nt1\ t1\ t2\ t3 \rightarrow \ldots \tag{6.2}$$

  auszudrücken. Die folgende Regel beschreibt den Sachverhalt: Wenn nach einem positiven Verb gefolgt von einem *not* gesucht wird, so erfüllt ein negiertes Verb dieselbe Funktion:

[2] Die meisten Prolog-Interpreter haben einen DCG-Parser eingebaut. HU-Prolog verfügt nicht über einen built-in-DCG-Interpreter. Man muß ein Programm benutzen, das die DCG-Regeln in Prolog-Rufe umwandelt.

Der HU-Prolog-DCG-Compiler kann keine Meta-Variablen verarbeiten. Mit den eingebauten DCG-Interpretern von Sicstus- und Quintus-Prolog sind die angegebenen Beispiele aber verarbeitbar. Wer Lust hat, kann ja den HU-Prolog-Compiler verbessern.

```
% did not --> didn't
verb(positive), [not] --> verb(negative).
```

- Beliebige Prolog-Aufrufe können in Regeln aufgenommen werden. Sie sind in geschweiften Klammern zu schreiben. Das kann benutzt werden, um Berechnungen während der Abarbeitung der Grammatikregeln ausführen zu lassen. Will man die DCGs als Mittel zum Schreiben von vernünftigen Grammatiken benutzen, so sollte man, soweit es geht, auf die Verwendung solcher Prolog-Rufe verzichten, da diese die Deklarativität der Grammatik zerstören. Ein Beispiel für eine Anwendung ist die folgende Regel, in der unsinnige Werte für das *Num*-Merkmal ausgeschlossen werden.

  ```
  s --> np(Num), vp(Num), {legal_num(Num)}.
  ```

  Eine solche Regel ist an sich unnötig, da die Werte des *Num*-Merkmals im Lexikon stehen und dort nur korrekte Werte angegeben sein sollten. In den folgenden Kapitel werden noch sinnvollere Beispiele zu finden sein.

- Der Prolog-Cut kann auf rechten Regelseiten stehen. Das kann die Verarbeitungsgeschwindigkeit der Regeln erhöhen, hat aber keine deklarative Lesart, sondern zerstört diese oft sogar. Man sollte die Benutzung des Cuts also tunlichst vermeiden.

6.2 Die Bedeutung der DCG

Die Bedeutung der DCG ist die Sprache, die sie definiert. Um eine DCG als Grammatik interpretieren zu können, muß man definieren, wann ein Satz aus einer DCG-Grammatik ableitbar ist. Wie im Kapitel 2.4 definieren wir den Begriff der *Satzform*:

Def. 4 (Satzform)

- *$s(\alpha_1,\ldots,\alpha_n)$ ist eine Satzform, wenn s das n-argumentige Startsymbol ist und jedes α_i ein* ground-*Term ist (d.h. keine Variablen enthält).*
- *Wenn* x *eine Satzform der Form $\alpha\beta\gamma$ ist, und es gibt eine* ground-*Instanz einer DCG-Regel der Form β --> δ, dann ist $\alpha\delta\gamma$ eine Satzform.*

Das heißt, jede Regel steht für alle ihre *ground*-Instanzen, also für alle möglichen Instantiierungen der Merkmale. Eine Satzform, die keine Nichtterminale enthält, wird Satz genannt.

6.3 Die Mächtigkeit der DCGs

Wie im Kapitel 3 diskutiert, ist eine durch Merkmale erweiterte kontextfreie Grammatik äquivalent zu einer anderen (eventuell sehr großen) kontextfreien Grammatik, wenn die Anzahl der Merkmale und die Anzahl der Werte, die die Merkmale annehmen können, endlich ist. Somit kann man ziemlich komplexe DCGs schreiben ohne den Bereich der Typ 2 Sprachen zu verlassen.

Daß man mit DCGs eine größere Sprachklasse beschreiben kann, ist ebenfalls klar. Indizierte Sprachen mit DCGs zu beschreiben, ist ein leichtes (siehe Kapitel 3.5).

Benutzt man die Erweiterungen des DCG-Formalismus, kann man sogar Typ 0 Grammatiken beschreiben:

```
    /* Rule for the start symbol 's' */
s --> rewrite([s]).
    /* Apply rewrite rules until we */
    /* get the right list of nonterminals  */
rewrite(List) --> all_present(List).
rewrite(List) --> rewrite_once(List,New), rewrite(New).
    /* apply a single rewrite rule  */
rewrite_once(Old,New) --> rule(Old,New).
rewrite_once([S|Old],[S|New]) --> rewrite_once(Old,New).
    /* Find a list of Symbols in the string */
all_present([]) --> [].
all_present([S,Ss]) --> S, all_present(Ss).
    /* encoded version of the rules of the grammar */
rule([a,b,c|L],[d,e,f|L]) --> [].
    % encoding of 'a b c --> d e f'

...
```

Diese Kodierung benutzt keine DCG-Nichtterminale. Verlangt man eine direktere Kodierung, so kann man mit DCGs nicht einmal kontextsensitive Grammatiken ausdrücken. Das heißt, die obige DCG-Grammatik hat dieselbe schwache Kapazität wie eine entsprechende Typ 0 Grammatik. Das ist das Entscheidende in der Theorie der formalen Sprachen. Sie hat aber eine andere starke Kapazität, d.h. die Art und Weise, in der Teile von Strings beschrieben werden, ist anders.

6.4 Die übersetzung von DCGs in logische Formeln

Wie ist es möglich, daß Prolog – das ja ein Theorembeweiser ist – dazu genutzt werden kann, kontextfreie Grammatiken zu parsen? Um zu verstehen, wie das funktioniert, werden wir zeigen, wie man kontextfreie Regeln in logische Formen umwandeln kann. Einen Satz zu parsen, bedeutet dann nichts anderes, als die Gültigkeit der logischen Ausdrücke zu beweisen. Diese Sichtweise auf die Beziehung zwischen Logik und Grammatik wird auch *Parsing as Deduction* genannt.

Kontextfreie Grammatiken und DCGs kann man als Aussagen verstehen, indem man die Symbole der Grammatik als Prädikate, die für bestimmte Satzteile gelten, auffaßt. Die Regel

$$s \rightarrow np\ vp \tag{6.3}$$

kann man in die Aussage der Form in (6.4) umwandeln.

$$\forall\ NP\ \forall\ VP\ \forall\ S\ np(NP) \wedge vp(VP) \wedge concat(NP, VP, S) \rightarrow s(S) \tag{6.4}$$

Das ist wie folgt zu verstehen: Für alle Wortketten NP, VP und S gilt, wenn das Prädikat *np* für die Wortkette NP gilt und das Prädikat *vp* für die Wortkette VP, und S ist die Verkettung von NP und VP, dann ist das Prädikat *s* für S wahr.

Das Prädikat `concat` bzw. `append` sollte jeder, der schon Prolog programmiert hat, kennen. Bei der Abarbeitung von DCGs benutzt Prolog `append` aber nicht, sondern Differenzlisten der Art, wie sie in Kapitel 1.5 kurz vorgestellt wurden. In der obigen übersetzung der Grammatikregel in eine logische Form waren die Prädikate einargumentig. Prolog übersetzt die Regeln in zweiargumentige Prädikate. Hinzu kommen die Merkmale, die schon in der Grammatikregel zu den Kategoriebezeichnungen gehörten (Bar-Level, Subkategorisierung, Numerus, etc.).[3] Aus dem *s* in der obigen Regel würde

$$s(S1, S2) \tag{6.5}$$

s zu beweisen heißt, zu zeigen, daß es einen Satz am Anfang von S1 gibt und der Rest, der nicht verarbeitet wurde, S2 ist. Es wird noch darauf eingegangen werden, welcher Art S1 und S2 sein können und müssen. Vorerst zeigt S1 an, wo die Wortkette beginnt und S2 wo sie endet. übersetzt man die anderen Nichtterminale entsprechend, so erhält man für die Regel in (6.3) die folgende logische Form:

$$\forall\ S0\ \forall\ S1\ \forall\ S\ np(S0, S1) \wedge vp(S1, S) \rightarrow s(S0, S) \tag{6.6}$$

oder in Prolog-Notation:

[3] Ob die Differenzlistenargumente vor oder hinter den anderen Argumenten stehen, hängt von der konkreten Implementation des DCG-Compilers ab.

```
s(S0,S) :- np(S0,S1), vp(S1,S).
```

Eine DCG-Regel, die einen Lexikoneintrag beschreibt,

```
det --> [the]
```

könnte die Form (6.7) haben, aber man will nicht für jeden Lexikoneintrag ein Prädikat spezifizieren.

$$\forall\, S\; det(S) \leftarrow the(S) \tag{6.7}$$

Stattdessen bedient man sich der Differenzlisten und schreibt:

```
det([the|S],S).
```

Dieser Lexikoneintrag nimmt von einem String (einer Wortkette) das erste Wort weg, falls es *the* ist und gibt den Rest des Strings, also `S`, zurück. Dieser Rest wird dann weiterverarbeitet.

Wenn man eine Grammatik in Prolog übersetzt hat, kann man das Prolog-Ziel

```
:- s([all,cats,chase,fido],[]).
```

formulieren, um Prolog beweisen zu lassen, daß die Grammatik den Satz *All cats chase Fido.* akzeptiert. Voraussetzung für ein Prolog-`yes` ist natürlich, daß man den Satz mit der Grammatik ableiten kann.

Lesetips

In Lehner (1990) wird eine DCG für Deutsch vorgestellt. Das Problem der freien Wortstellung (Subjekt und Objekte) wird durch Benutzung von Prolog-Rufen gelöst.

Kontrollfragen

1. Welche fünf Erweiterungen das Basis-DCG-Formalismus wurden beschrieben?
2. Wie lautet die Definition des Begriffes Satzform für DCGs?
3. Was heißt, eine DCG generiert eine Sprache?
4. Welche schwache generative Kapazität haben DCGs (innerhalb der Chomsky-Hierarchie)?
5. Welche starke generative Kapazität haben DCGs (innerhalb der Chomsky-Hierarchie)?
6. Wie kann man DCGs in logische Ausdrücke übersetzen?

übungsaufgaben

1. Zeigen Sie, unter Anwendung der Definitionen, daß die Sprache, die durch die folgende Grammatik erzeugt wird, den Satz *robin eats vegetables* enthält.

```
s --> np(X), vp(X).
vp(X) --> v(X,Y), comp(Y).

np(sing) --> [robin].
np(plur) --> [vegetables].
v(sing,trans) --> [eats].
comp(trans) --> np(Z).
```

2. Die Standard-DCG-Implementation erzeugt eine übersetzung der folgenden Art:

```
s --> np, vp.
s(S0,S) :- np(S0,S1),vp(S1,S).
```

Eine DCG kann als Erkenner genutzt werden, indem man eine Anfrage der Form

```
:- s([john,smiled],[]).
```

stellt. Man kann sich auch alternative Implementationen vorstellen, die stattdessen etwas in der folgenden Art erzeugen:

```
s(S0,S) :- np(S1,S), vp(S0,S1).
s(S0,S) :- vp(S0,S1), np(S1,S).
s(S0,S) :- vp(S1,S), np(S0,S1).
```

Welche der drei übersetzungen geben die Bedeutung der Phrasenstrukturregel korrekt wieder? Welches Erkennungsverhalten haben sie zur Folge?

3. * Es ist aufwendig, sich für alle Subkategorisierungsmöglichkeiten neue *Subcat*-Namen auszudenken.

```
v(1) --> v(intrans), [].
v(1) --> v(trans),n(2).
v(1) --> v(ditrans), n(2), n(2).
```

Kann man DCG-Meta-Variablen verwenden, um die Einführung von *Subcat*-Werten zu vermeiden?

4. * DCGs können nicht nur zur Beschreibung der Bildung von Phrasen aus Wörtern benutzt werden, sondern auch zur Beschreibung der Wortbildung aus Buchstaben. Schreiben Sie Regeln, die aus dem Lexikoneintrag *plan* für das Verb und *quick* für das Adjektiv die folgenden Wörter und deren Kategorien ableiten können. *plan* und *planner* sind Nomina, *planning*, *plans*, *quicken*, *quickens* sind Verben, *quickest* und *quicker* sind Adjektive, und *quickly* ist ein Adverb.

5. ** Es gibt folgende Konstruktionen, die aus dem Satz *A cat chases Tigger.* abgeleitet sind. Man sagt, der Satz ist in einen Matrixsatz eingebettet.

   Subjektkontrollverben

   (6.8) a. A cat seems to chase Tigger.
   b. A cat tries to chase Tigger.

   Objektkontrollverben

   (6.9) a. Fido expected a cat to chase Tigger.
   b. Fido persuades a cat to chase Tigger.

   expect kann auch als Subjektkontrollverb auftreten. In jeder Gruppe von Beispielsätzen gibt es Verben, die vom kontrollierten Element verlangen, daß diese echte Nominalphrasen sind, d.h. referentielle Nominalphrasen, die sich auf etwas beziehen, also nicht *there* oder *it* wie in *It rains.*. Die, die echte Nominalphrasen verlangen, werden *equi*- und die anderen *raising*-Verben genannt.

   raising-Verben

   (6.10) a. There seems to be a cat in the garden.
   b. Tigger expected there to be a cat in the garden.

   equi-Verben

   (6.11) a. * There tries to be a cat in the garden.
   b. * Fido persuades there to be a cat in the garden.

   Schreiben Sie eine DCG, die die folgenden Sätze akzeptiert

(6.12) a. A cat chases Tigger.
b. A cat seems to chase Tigger.
c. A cat tries to chase Tigger.
d. Fido expected a cat to chase Tigger.
e. Fido persuades a cat to chase Tigger.
f. There is a cat in the garden.
g. There seems to be a cat in the garden.
h. Tigger expected there to be a cat in the garden.

und die Sätze in (6.13) zurückweist.

(6.13) a. * A cat persuades to chase Tigger.
b. * A cat tries Tigger to chase Fido.
c. * There tries to be a cat in the garden.
d. * Fido persuades there to be a cat in the garden.

Kapitel 7

Fernabhängigkeiten in DCGs

In Kapitel 5.6 haben wir gesehen, wie die Art und Anzahl der Komplemente mit Hilfe des *Subcat*-Merkmals im Lexikon kodiert werden können. Durch das *Subcat*-Merkmal wurde gewährleistet, daß eine bestimmte lexikalische Kategorie nur dann in einer rechten Seite auftauchen darf, wenn die anderen Elemente der rechten Seite den Subkategorisierungseigenschaften der lexikalischen Kategorie entsprechen. So kann die Akzeptanz ungrammatischer Sätze wie

(7.1) a. * Mary sneezed the dog.

b. * Fido chased.

c. * John persuades to eat a bone.

verhindert werden. Das linguistische Phänomen der Fernabhängigkeiten (*unbounded* oder auch *long-distance dependency constructions* (UDC) wurde in Kapitel 5.7.4 vorgestellt, und es wurden einige Beispiele aufgeführt. In solchen Abhängigkeiten werden die Subkategorisierungsanforderungen des Verbs nicht in der Konstituente erfüllt, die das Verb enthält. Diese Phänomene werden *unbounded* (unbegrenzt) genannt, da es keine Begrenzung für die Entfernung oder Anzahl der Satzgrenzen gibt, über die hinweg Konstituenten bewegt werden können.

(7.2) a. This book, I read.

b. This book, I managed to read.

c. This book, I believe Mary managed to read.

In allen drei Sätzen folgt das Objekt nicht unmittelbar dem Verb, wie es ansonsten in englischen Sätzen üblich ist, sondern steht am Satzanfang. Man markiert die Stelle, an der das Objekt eigentlich stehen müßte, mit einem *t* für *trace* (Spur) an dem unten ein Index steht. Der Index ist nötig, da es mitunter in einem Satz

mehrere Spuren gibt. Die Konstituente, die zu dem entsprechenden Trace gehört, wird Filler genannt und trägt denselben Index wie der Trace. Es gibt auch die Schreibweise $__i$ für Spuren. Eine Spur wird auch Lücke (*gap*) genannt.

(7.3) [This book]$_i$, I read t$_i$.

Einen möglichen Syntaxbaum für (7.2b) zeigt Abbildung 7.1. Wenn wir die Sätze

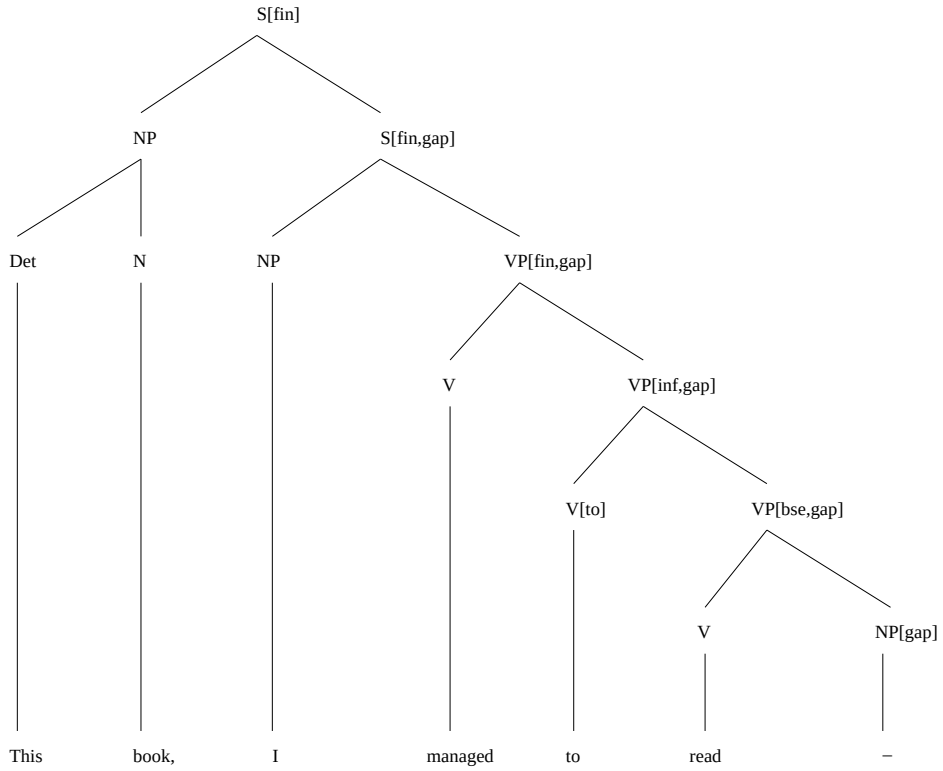

Abbildung 7.1: *This book, I managed to read.*

in (7.2) mit ihrer Satzstruktur aufschreiben, dann sieht man, daß es keine Beschränkungen für die Bewegung über Satzgrenzen hinweg gibt.

(7.4) a. [This book]$_i$, [$_s$ I read t$_i$].

b. [This book]$_i$, [$_s$ I managed [$_{vp}$ to read t$_i$]].

c. [This book]$_i$, [$_s$ I believe that [$_s$ Mary managed [$_{vp}$ to read t$_i$]]].

Die anderen Beispiele aus Kapitel 5.7.4 sehen wie folgt aus:

(7.5) a. [The college]$_i$, that [$_s$ I expected John [$_{vp}$ to want Mary [$_{vp}$ to attend t$_i$]]] is very expensive.

b. [What]$_i$ do you believe that [$_s$ Mary told Bill that [$_s$ John had said t$_i$]]]?

c. [This film]$_i$ is [$_{ap}$ easy for me [$_{vp}$ to persuade the children [$_{vp}$ not to see t$_i$]]].

7.1 Beschreibung der UDCs

Man kann UDCs mit Hilfe von drei Bezeichnungen beschreiben:

- der Anfang der Abhängigkeit (*the top of the dependency*), d. h. die Konstruktion, in der der Gap eingeführt wurde,
- das Ende der Abhängigkeit (*the bottom of the dependency*), die Stelle, an der der Gap entfernt wird, und
- die Mitte, durch die die Gap-Information weitergereicht wird.

Wie die Beispiele zeigen, können Gaps innerhalb vieler Konstruktionen eingeführt werden. Entfernt werden sie jedoch immer auf dieselbe Weise, nämlich durch das Fehlen einer Konstituente an einem Platz, an dem eine Regel diese Konstituente verlangen würde. Teilsätze wie (7.1b) und (7.1c) sind korrekt, wenn irgendwo innerhalb des Gesamtsatzes ein Gap konsumiert wird.

(7.6) a. This is the cat that Fido chased.

b. This is the cat that John persuades to eat a bone.

Unbounded dependencies sind mit kontextfreien Grammatiken schwierig zu beschreiben, weil man die Informationen des Gaps vom Anfang zum Ende der Abhängigkeit bewegen muß. Da CFG-Kategorien atomar sind, braucht man separate Regeln für eine VP ohne Gap, eine VP mit einem Gap und eine VP mit zwei Gaps. Um die Syntax der UDCs mit DCGs zu beschreiben, braucht es nur ein Merkmal, das die beiden Werte *gap* und *nogap* haben kann. Eine Nominalphrase mit dem Wert *gap* würde dem leeren String entsprechen.

```
n(2,gap) --> [].
```

Die folgenden Regeln zeigen, wie ein Gap in einen Satz eingeführt wird, der ein Nomen modifiziert (ein Relativsatz).

```
n(1) --> n(1),s(gap).
```

oder

```
n(1) --> n(1),rel.
rel --> [that],s(gap).
```

Wie kommt nun Informationsfluß zustande? Wie wird der Gap innerhalb der Elemente auf der rechten Seite verarbeitet? Es ist wichtig, daß der Gap nur zu einer der Töchter, d. h. zu einem der Elemente auf der rechten Seite, geht.

(7.7) * This is the cat$_i$ that Fido persuades t$_i$ to chase t$_i$.

Diese Tochter darf keine Kopftochter sein. Deshalb wird das Merkmal *gap* manchmal Fußmerkmal (*foot feature*) genannt. Um zu erreichen, daß der Gap nur zu einer Tochter gelangt, kann man eine Hilfsprozedur `legal_gaps` benutzen, die in jeder Regel aufgerufen wird:

```
legal_gaps(X,nogap,X).
legal_gaps(nogap,X,X).

s(X) --> n(2,Y), v(2,Z), {legal_gaps(Y,Z,X)}.
```

7.2 Lückenfädeln (*Gap Threading*)

Wir können die Hilfsprozedur weglassen, indem wir eine andere Form der Repräsentation der Gap-Information wählen. Die Repräsentation funktioniert auf die gleiche Weise wie Differenzlisten. Wir haben ein Merkmalpaar. Das eine Merkmal *GapIn* wird für die Gap-Information gebraucht, die in eine Konstituente hineingeht und das andere *GapOut* für die Information, die aus ihr herauskommt.

ein Argument	zwei Argumente	
gap	GapIn = gap,	GapOut = nogap
nogap	GapIn = X ,	GapOut = X

Es gibt keine Konstituenten mit den Werten GapIn = nogap und GapOut = gap. Die angegebenen Merkmalpaare entsprechen den folgenden Regeln:

```
n(1) --> n(1), s(gap,nogap).
n(2,gap,nogap) --> [].
```

Den Informationsfluß mit GapIn- und GapOut-Merkmal zeigt Abbildung 7.2.

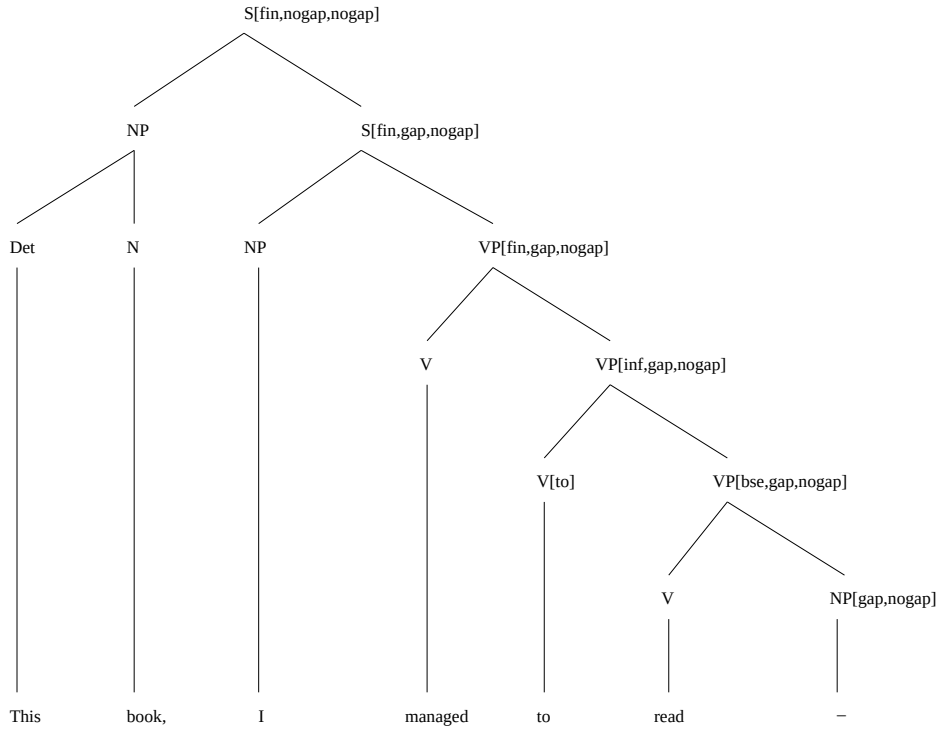

Abbildung 7.2: *This book, I managed to read.*

7.3 Beschränkungen

Kann ein Gap in jede Konstituente auf der rechten Seite eingegeben werden, die nicht die Kopftochter ist?

(7.8) a. Fido chased [$_{np}$ the dog that [$_s$ bit a cat]].

b. * The cat$_i$ that [$_s$ Fido chased [$_{np}$ the dog$_j$ that [$_s$ t bit t]]] sleeps.

Im zweiten Satz kann keiner der beiden Gaps in der Nominalphrase *the dog that t bit t* von *the cat* gefüllt werden.

(7.9) a. Bill believes that [$_s$ John claimed that [$_s$ Mary kissed someone]].

b. Bill believes [$_{np}$ John's claim that [$_s$ Mary kissed someone]].

c. Who$_i$ does Bill believe that [$_s$ John claimed that [$_s$ Mary kissed t$_i$]]?

d. * Who$_i$ does Bill believe [$_{np}$ John's claim that [$_s$ Mary kissed t$_i$]]?

In diesen Sätzen führt das *that* nicht Relativsätze sondern die Satz-Komplemente von *claim* bzw. *believe* ein. Die Gaps in den letzten beiden Sätzen gehören zum Wh-Fragewort. Im Satz (7.9d) befindet sich der Gap in der NP, was sehr viel schlechter ist als in (7.9c), wo sich der Gap im Satz befindet.

In der weiter unten angegebenen Grammatik gibt es eine vereinfachte Behandlung dieses Phänomens. Alle Nominalphrasen, die keine Gaps sind, haben gleiche Variablen für GapIn und GapOut. Dadurch kann kein auf höherem Level eingeführter Gap innerhalb einer Nominalphrase entfernt werden. Er wird stattdessen durch die Nominalphrase hindurch weitergereicht.

Es gibt einige Fälle im Englischen, in denen mehrere Gaps in einem Satz vorkommen:

(7.10) a. [Which violin]$_i$ is [this sonata]$_j$ easy to play t$_j$ on t$_i$?

b. [A violin this well crafted]$_i$, even [the most difficult sonata]$_j$ will be easy to play t$_j$ on t$_i$.

Solche Fälle könnte man mit einem *gap*-Merkmal beschreiben, dessen Wert eine Liste ist.[1]

Wir können nun die Regel für einen englischen Satz angeben:

[1] Es gibt noch wesentlich kompliziertere *unbounded dependency constructions* im Englischen.

(7.11) Who$_i$ did my talking to t$_i$ bother t$_i$?

In Pollard und Sag (1994) findet man eine kompliziertere Theorie als die hier dargestellte, die auch diese Phänomene korrekt beschreibt.

```
top_s --> s(fin,nogap,nogap).
```

In der in Kapitel 7.5 angegebenen Grammatik sind alle diskutierten Fakten realisiert. Das *Num*-Merkmal wurde zugunsten der Lesbarkeit weggelassen.

7.4 Erweiterungen

Es gibt außer Nominalphrasengaps noch Präpositionalphrasengaps:

(7.12) a. the man <u>who</u> I spoke to

b. the man <u>to whom</u> I spoke

Man kann das *Gap*-Merkmal strukturieren, so daß es noch Informationen über die Art des Gaps aufnehmen kann. Dann hat man die folgenden Regeln, durch die Gaps konsumiert werden:

```
np(gap(np),nogap) --> [].
pp(gap(pp),nogap) --> [].
```

Da unsere Grammatik nur Relativsätze ohne Relativpronomina zuläßt, müßten noch weitere Änderungen vorgenommen werden. Diese seien dem interessierten Leser als Übung überlassen:

(7.13) Those books, the wording on the covers of which everybody disagreed with, have been burned.

7.5 Die DCG-Grammatik

```
top_s --> s(fin,nogap,nogap).

s(Vform,G0,G) --> n(2,G0,G1),v(2,Vform,G1,G).

n(2,G,G) --> proper_noun.
n(2,G,G) --> det, n(1).
n(1) --> n(1), s(fin,gap,nogap).
n(1) --> n(0).

n(2,gap,nogap) --> [].

v(2,Vform,G0,G) --> v(1,Vform,G0,G).
```

```
v(1,Vform,G,G)  --> v(intrans,Vform).
v(1,Vform,G0,G) --> v(trans,Vform), n(2,G0,G).
v(1,Vform,G0,G) --> v(obj_equi,Vform), n(2,G0,G1),
                    v(2,inf,G1,G).
v(1,Vform,G0,G) --> v(scomp,Vform),[that],s(fin,G0,G).
v(1,inf,G0,G)   --> v(to,inf), v(2,bse,G0,G).

v(intrans,bse) --> [run].
v(intrans,fin) --> [runs].
v(trans,bse) --> [chase].
v(trans,fin) --> [chases].
v(obj_equi,bse) --> [persuade].
v(obj_equi,fin) --> [persuades].
v(scomp,bse) --> [believe].
v(scomp,fin) --> [believes].
v(to,inf) --> [to].

det --> [a].
det --> [the].

n(0) --> [cat].
n(0) --> [dog].

proper_noun --> [fido].
proper_noun --> [tigger].

test1 :- top_s([the,dog,fido,chases,persuades,a,cat,
                                 to,chase,tigger],[]).
test2 :- top_s([a,dog,chases,the,cat,fido,persuades,
                                          to,run],[]).
test3 :- top_s([tigger,chases,the,dog,fido,believes,
                              that,a,cat,chases],[]).
```

Lesetips

In (Pereira und Shieber, 1987) findet man weiteres zum *Gap-Threading* und seinen Ursprüngen.

Kontrollfragen

1. Wieso braucht man Fernabhängigkeiten für die Beschreibung von Relativsätzen?

2. Wo sind Anfang und Ende der Fernabhängigkeit im Satz *This is the cat that John persuades to eat a bone.*

3. Wenn X und Y die beiden Argumente für das *gap threading* in einer Phrase sind, welche Werte können sie dann annehmen?

Übungsaufgaben

1. Geben Sie in den folgenden Sätzen die Stelle an, an der die UDC aufgelöst wird, d. h. den Bottom bzw. das Ende der Fernabhängigkeit. Und geben Sie an, welche Konstruktion dafür verantwortlich ist, daß es einen Gap geben muß.

   (7.14) a. Who does he take to work each day?

   b. He wonders who his mother thinks he visited.

   c. This is the cat that ate the rat that lived in the house that Jack build.

   d. The man who the woman who the cat bit smiled at was happy.

2. Erklären Sie warum der Prolog-Parser den Satz *the man who the man saw slept* erkennt, den Satz *the man who saw slept* aber nicht, wenn folgende Grammatik zugrunde liegt:

```
top_s --> s(nogap,nogap).

s(G1,G3) --> np(G1,G2), vp(G2,G3).

np(gap,nogap) --> [].
np(G,G) --> [the,man], postmod.

postmod --> [].
postmod --> [who], s(gap,nogap).

vp(G,G) --> [slept].
vp(G1,G2) --> [saw], np(G1,G2).
```

3. * Erweitern Sie die Beispielgrammatik aus Kapitel 7.5 um Regeln, die einfache Topikalisierung und *tough*-Movement beschreiben können:

   (7.15) a. John, Mary nearly sat on.
   b. John is easy to persuade Mary to please.

Hausaufgabe

4. Erweitern Sie Ihre Grammatik von der Hausaufgabe 3 (Kapitel 5) so, daß sie die folgenden Sätze akzeptiert.

   (7.16) a. Mary gives John a book Helen is fond of.
   b. Mary is fond of the man John has forced to go.
   c. Mary is fond of the man John has forced to force Helen to go.
   d. The book, John gave to Mary.
   e. The town, Mary has forced John to be fond of.
   f. Mary is easy to please.
   g. Mary is easy to force John to be fond of.

   Benutzen Sie die in Kapitel 7.2 vorgestellte Technik des *gap threading*.

Kapitel 8

Parse-Strategien und DCGs

8.1 Kriterien für einen guten Parser

Ein Parser bekommt eine Grammatik und einen String als Eingabe und soll alle möglichen Analysen des Strings ausgeben. Hier sind die wichtigsten Kriterien zur Beurteilung von Parsern:

1. *Korrektheit* Ein Parser ist korrekt, wenn alle Analysen, die er zurückgibt, mögliche Analysen des Eingabestrings sind. Manchmal läßt man jedoch zu, daß der Parser unkorrekte Analysen liefert und sortiert diese nach dem Parsen aus. Das kann sinnvoll sein, wenn die fälschlicherweise gelieferten Analysen während des Parseprozeßes nur mit unverhältnismäßig großem Aufwand auszuschließen wären.

2. *Vollständigkeit* Ein Parser ist vollständig, wenn er jede mögliche Analyse eines Eingabestrings für eine bestimmte Grammatik liefert. Wenn eine Grammatik unendlich viele Analysen für einen String zuläßt, dann bedeutet Vollständigkeit, daß es keine Analyse gibt, die der Parser nicht irgendwann einmal findet. Der Parser würde in solch einem Fall natürlich nicht terminieren, sondern nach und nach die Analysen ausdrucken oder anderweitig zur Verfügung stellen.

   In manchen Anwendungen ist die Vollständigkeit des Parsers nicht notwendig. Man kann sich z. B. vorstellen, daß in zeitkritischen Anwendungen nicht genügend Zeit vorhanden ist, um alle möglichen Analysen in Betracht zu ziehen. Stattdessen werden dann alle zu einem bestimmten Zeitpunkt vorhandenen Ergebnisse berücksichtigt und mit der angenommenen besten Analyse weitergearbeitet.

3. *Effizienz* Ein Parser sollte nicht unnötigerweise ineffizient sein. Arbeit, die

nur einmal getan werden muß, um korrekte Ergebnisse zu erhalten, sollte nicht wiederholt werden.

8.2 Parse-Strategien

Eine fundamentale Dimension der Parse-Strategie ist die Analyserichtung. Man kann entweder *goal-driven* (Ziel-gesteuert) oder *data-driven* (Daten-gesteuert) parsen. Wir wollen beweisen, daß eine Wortkette ein *s* ist und unsere Daten sind die Wörter der Wortkette. Eine *goal-driven* Strategie beginnt mit dem Startsymbol der Grammatik *s*. Das wird mit den linken Seiten der Grammatikregeln verglichen und durch die rechte Seite einer passenden Regel – also einer Regel, auf deren linker Seite ein *s* steht – ersetzt. Der Prozeß wird mit den neuen Symbolen fortgesetzt. Ist die zu parsende Wortkette ein Satz der verarbeiteten Grammatik, so gelangen wir eventuell durch diesen Ersetzungsprozeß zu der Wortkette.

Eine *data-driven* Strategie beginnt bei den Wörtern im String und vergleicht diese mit den rechten Seiten von Grammatikregeln der Form PT → w, wobei PT ein Nichtterminal, das zu einer lexikalischen Kategorie gehört (auch Preterminal genannt), und w ein Wort ist. Diese Strings aus Preterminalen werden wieder mit rechten Regelseiten verglichen, und mit den Nichtterminalen der linken Seiten wird weitergearbeitet. Dieser Prozeß wird solange fortgesetzt bis ein String zu *s* reduziert wurde.

Da die Anwendung von kontextfreien Regeln eine hierarchische Struktur für einen String ergibt, d. h. einen Syntaxbaum, und da Syntaxbäume immer auf dem Kopf stehend aufgeschrieben werden (*s* steht ganz oben) wird die *goal-driven* Strategie Top-Down- und die *data-driven* Strategie Bottom-Up-Strategie genannt.

Eine andere Dimension ist die Reihenfolge der Verarbeitung. Werden Symbolketten von rechts nach links oder von links nach rechts verarbeitet? Es gibt auch Anwendungsbereiche, wie die Verarbeitung gesprochener Sprache, wo man eventuell in der Mitte bei sicher erkannten Wörtern oder Wortgruppen beginnt.

Außerdem gibt es noch verschiedene Möglichkeiten, Nichtdeterminismus zu behandeln. Wählen wir, wenn es mehrere Ersetzungsmöglichkeiten gibt, genau eine Alternative und gehen zurück, falls die Weiterverfolgung dieser Alternative sinnlos ist (Backtracking), oder verfolgen wir mehrere Alternativen gleichzeitig? Die erste Arbeitsweise wird Tiefe-zuerst-Suche (*depth-first-search*) und die zweite Breite-zuerst-Suche (*breadth-first-search*) genannt. Da wir momentan nur sequentielle Maschinen zur Verfügung haben, stellt sich nur die Frage, ob man den Nichtdeterminismus mit einer impliziten Steuerung behandelt oder explizit mit einer komplexen Datenstruktur, in der mehrere Möglichkeiten gespeichert sind.

Um das theoretische Minimum an Parsezeit zu erreichen – n^3 für n Wörter – brauchen wir eine komplexe Datenstruktur die *well-formed substring table* oder

die *chart* (siehe Kapitel 11).

8.3 Top-Down-Parsen

Ein Zustand eines Top-Down-von-links-nach-rechts-Erkenners wird durch zwei Dinge beschrieben:

1. eine Sequenz von Zielen – bestimmte Phrasen, nach denen gesucht wird, und
2. die Wortsequenz, in der die Phrasen gefunden werden müssen.

Der Erkenner arbeitet wie folgt: Er nimmt das erste Ziel aus der Liste, sucht eine Grammatikregel, die die dem Ziel entsprechende Kategorie auf der linken Seite hat, und ersetzt das Ziel durch die Sequenz der Symbole auf der rechten Seite. Wenn das erste Symbol in der Sequenz der Ziele dem Wort in der Wortsequenz gleicht, kann es aus beiden Listen entfernt werden. Wenn beide Listen leer sind, wurde die Wortsequenz erkannt. Es folgen die Sequenzen beim Parsen des Satzes *John likes Mary.*:

Ziele	Wörter
s	John likes Mary
np vp	John likes Mary
John vp	John likes Mary
vp	likes Mary
verb np	likes Mary
likes np	likes Mary
np	Mary
Mary	Mary

Bei der Abarbeitung der Sequenzen muß man die Entscheidung treffen, welche der Grammatikregeln für die Expansion des ersten Ziels anzuwenden ist. Nicht alle Regelanwendungen führen zum Ziel. Der Erkenner in unserem Beispiel expandiert sogar Preterminalsymbole top-down. Der Parser ersetzt z. B. *verb* nacheinander durch alle Lexikoneinträge der Kategorie *verb*. In der Praxis läßt man Top-Down-Erkenner nur bis zu den Preterminalen arbeiten. Statt der Wortsequenz wird eine Sequenz von Preterminalen verarbeitet und wie oben beschrieben schrittweise reduziert.

Um einen von-rechts-nach-links-Erkenner zu erhalten, müßte man die Abarbeitungsreihenfolge ändern. Ziele und Wörter werden vom Ende der Sequenzen entfernt bzw. ans Ende angehängt.

Man kann diese Herangehensweise auch so verallgemeinern, daß der Erkenner an mehreren Stellen des Strings gleichzeitig arbeiten kann.

Um aus einen Erkenner einen Parser zu machen, muß man ihn so erweitern, daß er sich irgendwo merkt, welche Regeln angewendet wurden, um das Ziel *s* und die entsprechenden Teilziele zu beweisen.

Top-Down-Parser können nicht mit allen Grammatiktypen gleich gut umgehen. Ein von-links-nach-rechts-Top-Down-Parser hätte z. B. mit einer Grammatikregel der Form

$$v(1, Vform, Num) \rightarrow v(1, Vform, Num)\ verb_mods \tag{8.1}$$

Schwierigkeiten:

Ziel	Wörter
...	...
v(1,...) ...	...
v(1,...) verb_mods ...	...
v(1,...) verb_mods verb_mods ...	...
v(1,...) verb_mods verb_mods verb_mods ...	...
...	...

usw. bis in alle Ewigkeit. Auch wenn durch eine geeignete Suchstrategie sichergestellt wird, daß diese Regel nicht als erste gewählt wird, so muß der Parser sie doch irgendwann anwenden, da von einem Parser verlangt wird, daß er alle möglichen Analysen liefert. Es könnte z. B. eine Eingabekette mit 534 Verbmodifikatoren vorliegen, die zu erkennen der Parser in der Lage sein muß. Er muß also die Regel anwenden, wenn sie anwendbar ist. Der Top-Down-Parser kann nicht erkennen, daß die unendliche Wiederholung der Regelanwendung sinnlos ist (z. B. weil es nicht möglich ist, daß noch 534 Modifikatoren folgen, weil die Wortsequenz nur noch drei Wörter enthält.) Regeln wie die in (8.1) werden *linksrekursiv* genannt.

Linksrekursion kann durch die Kombination von Regeln entstehen, die selbst nicht linksrekursiv sind. Will man z. B. die folgende Regel für Determinatoren in der Form von Possessivphrasen wie *John's* in *John's book* schreiben, so ergibt sich durch Kombination mit der Regel n(2) → det n(1) eine Linksrekursion.

$$det \rightarrow n(2)'s \tag{8.2}$$

Ein Top-Down-Parser ist auch für Fälle schlecht geeignet, in denen es viele Möglichkeiten zur Ersetzung der Ziele gibt, die für den zu parsenden Satz belanglos sind. Gibt es z. B. 600 Regeln, für die Kategorie *s*, so müssen diese vom Top-Down-Parser berücksichtigt werden, auch wenn 599 der Regeln mit einer NP anfangen und der zu parsende Satz mit einem Verb.

8.4 Bottom-Up-Parsen

Beim Top-Down-Erkenner beginnen wir mit den Zielen, die solange durch rechte Regelseiten ersetzt werden, bis wir Terminale erhalten, die mit dem String verglichen werden. Beim Bottom-Up-Parsen dagegen beginnt man beim String. Aus den Bestandteilen des Strings werden Phrasen gebildet, in der Hoffnung irgendwann ein *S* zu erhalten.

Bei jedem Schritt hat ein Bottom-Up-Erkenner eine Sequenz von Terminalsymbolen und Nichtterminalsymbolen, die dem entsprechen, was bis jetzt im String gefunden wurde (z. B. 'NP V Petra'). Der Erkenner vergleicht Anfänge dieser Sequenz mit rechten Seiten von Grammatikregeln. Wird eine rechte Seite gefunden, die mit einem Anfang der Sequenz übereinstimmt, so wird der Anfang aus der Sequenz entfernt, und statt seiner das Symbol auf der linken Seite der entsprechenden Regel eingetragen. Dabei muß der Erkenner zwei Dinge entscheiden. Erstens welche Regel angewendet wird und zweitens wie lang die Anfangssequenz sein soll. Eine Organisationsfrom ist der Shift-Reduce-Erkenner. Ein Zustand eines Shift-Reduce-Erkenners ist durch folgendes gekennzeichnet:

1. Eine Sequenz von Symbolen, die zum Anfang des Strings gefunden wurden und vollständige Phrasen repräsentieren.

2. Eine Sequenz von Symbolen, die die Wörter repräsentieren, die noch nicht berücksichtigt wurden.

Bei Shift-Reduce-Erkennern wird die Sequenz der Symbole von erkannten Phrasen als Stack bezeichnet. Bei jedem Schritt schiebt der Erkenner die Kategorie eines neuen Wortes auf den Stack (*shift*), oder er reduziert die Symbole auf dem Stack, indem er eine Symbolsequenz auf dem Stack findet, die einer rechten Regelseite entspricht und diese Sequenz durch die linke Seite der Regel ersetzt. Steht das Startsymbol *S* auf dem Stack und ist gleichzeitig die Wortsequenz leer, so ist der Eingabestring erfolgreich erkannt. Es folgen die Zustände eines Shift-Reduce-Erkenners bei der Eingabe des oben schon verwendeten Satzes.

Stack	Wortsequenz
	John likes Mary
np	likes Mary
np v	Mary
np v np	
np vp	
s	

In einem Shift-Reduce-Erkenner müssen zwei Entscheidungen gefällt werden. Bei jedem Schritt kann man entweder schieben oder reduzieren. Außerdem muß entschieden werden, welche Regel bei der Feststellung der Kategorie eines Wortes und welche Regel beim Reduzieren angewendet wird. Um aus dem Erkenner einen Parser zu machen, muß man ihn so erweitern, daß er sich die angewendeten Regeln merken kann. Das kann man erreichen, indem man erkannte Kategorien als Parse-Bäume repräsentiert, deren Spitze die entsprechende Kategorie bildet. Wenn eine Reduktion durchgeführt wird, bekommt man ein neues Symbol, das der linken Regelseite entspricht, erweitert durch eine Folge der Parse-Bäume der Elemente der entsprechenden rechten Seite.

Eine andere Form des Bottom-Up-Parsens – das Left-Corner-Parsen – nimmt Regeln in eine Hypothesenmenge auf, wenn die erste Kategorie der rechten Regelseite gefunden wurde. Diese Parser werden im nächsten Kapitel genauer erklärt.

Bottom-Up-Parser werden mit Grammatiken, die ε-Produktion enthalten, d. h. Regeln mit leeren rechten Seiten, nicht fertig. Der Grund dafür ist, daß der Parser, falls es eine Regel der Form $C \rightarrow \varepsilon$ existiert, an jeder Stelle des Strings beliebig viele Cs auf den Stack schieben kann. Ein Bottom-Up-Parser ist auch schlechter als ein Top-Down-Parser, wenn es große lexikalische Ambiguität gibt. Kann das erste Wort im String zu 10 verschiedenen Kategorien gehören, so muß der Bottom-Up-Erkenner alle berücksichtigen und sucht auch nach größeren Phrasen, die die entsprechende Kategorie enthalten, auch wenn nur eine wirklich zu Beginn eines S möglich wäre. Eine Kombination von Top-Down- und Bottom-Up-Parser kann diese Probleme verkleinern. Der Left-Corner-Parser ist solch eine Kombination von beiden und wird in Kapitel 9.4 vorgestellt.

8.5 Tiefe-zuerst-Suche vs. Breite-zuerst Suche

Tiefe-zuerst-Suche bedeutet, aus einer Menge alternativer Folgezustände einen auszuwählen und alle möglichen Nachfolgezustände dieses Zustands zu berücksichtigen, bevor man zurückkommt und eine andere Alternative wählt. Diese Strategie, nach der auch der Prolog-Interpreter arbeitet, wird auch Backtracking genannt. Backtracking kann effizient implementiert werden, weil der Interpreter nur einen Stack vorangegangener Entscheidungspunkte (*choice points*) verwalten muß. Stellt sich eine Entscheidung als falsch heraus, so muß der Interpreter zum nächsten (obersten) Entscheidungspunkt zurückkehren und nach einer erneuten Entscheidung die Arbeit fortsetzen. Die Größe des Stacks ist proportional zur Tiefe des Suchbaumes. Bei der Breite-zuerst-Suche simuliert der Interpreter das gleichzeitige Ausprobieren aller Möglichkeiten. Da heutige Maschinen zumeist sequentiell arbeiten bedeutet Breite-zuerst-Suche, daß etwas Zeit für die Abarbeitung der ersten Alternative, etwas Zeit für die zweite, usw. verbraucht wird. Bei

solch einer Vorgehensweise muß man den gesamten Suchbaum mit allen aktuellen Abarbeitungszuständen irgendwie repräsentieren, was meistens wesentlich aufwendiger ist als die Abspeicherung der Entscheidungspunkte für die Tiefe-zuerst-Suche.

Wenn der Suchraum unendlich ist (z. B. für einen Top-Down-Parser und eine Grammatik mit Linksrekursion), dann ist ein System mit Tiefe-zuerst-Suche nicht vollständig. Ein System mit Breite-zuerst-Suche dagegen findet immer alle möglichen Antworten, auch wenn die Suche eventuell nicht terminiert, weil der Suchraum unendlich ist.

8.6 Grammatikübersetzung

Um mit den Problemen der Linksrekursion und der ε-Produktionen fertig zu werden, kann man anstatt eine andere Parse-Strategie zu wählen auch die Grammatik verändern. Man kann eine kontextfreie Grammatik automatisch in eine Grammatik umwandeln, die schwach äquivalent ist aber keine Linksrekursion enthält. Genauso ist es möglich, automatisch eine schwach äquivalente Grammatik zu erzeugen, die keine ε-Produktionen enthält (Bar-Hillel, Perles und Shamir, 1961, S. 153, Lemma 4.1).

Leider ist das Ergebnis einer solchen Übersetzung nur schwach äquivalent zur Ausgangsgrammatik (abgesehen von dem Fall, daß diese schon in der gewünschten Form ist). Die automatische Umwandlung ist also nicht immer verwendbar, es sei denn man formt die Parsebäume entsprechend mit um.

8.7 Der Standard-DCG-Parser

Die Notation von DCG-Grammatiken ist konzeptuell unabhängig von den Strategien fürs Erkennen und Parsen. Die Probleme, die beim Interpretieren von DCGs entstehen, sind keine Folge der Notation. Man kann ohne weiteres einen Parser schreiben, der DCGs anderen Strategien folgend interpretiert (vergleiche Kapitel 9). Trotzdem haben die meisten Prolog-Versionen einen eingebauten Compiler, der DCGs nach Prolog übersetzt. Das bedeutet, daß es einen Standard-Weg gibt, DCGs zu parsen bzw. zu erkennen, der durch die Arbeitsweise von Prolog festgelegt ist.

Wie arbeitet Prolog? Prolog führt eine Tiefe-zuerst-Suche mit Backtracking, von links nach rechts und Top-Down durch. Um ein Ziel zu beweisen, wird eine Prolog-Klausel gesucht, deren Kopf mit dem Ziel matcht, und die Prädikate im Körper der gewählten Klausel werden Teil-Ziele, deren gleichzeitige Gültigkeit eine Bedingung für den Beweis des Top-Level-Ziels ist. Prolog versucht diese

Ziele von links nach rechts zu beweisen. Prolog berücksichtigt nur eine Beweismöglichkeit pro Ziel, nur wenn der Beweis eines späteren Ziels fehlschlägt, wird Backtracking bis zum letzten Entscheidungspunkt durchgeführt, um dann eine andere Alternative zu berücksichtigen. Es handelt sich also um eine Tiefe-zuerst-Suche.

Genauso arbeitet Prolog auf DCGs. Durch die Übersetzung der Grammatik in Prolog-Klauseln erhält man ein normales Prologprogramm, das genauso wie oben beschrieben abgearbeitet wird. Der Standard-DCG-Erkenner ist nicht für alle Grammatiken vollständig.

Kontrollfragen

1. Warum wäre es trivial, einen Parser zu schreiben, der korrekt aber nicht vollständig ist?
2. Welche Parser haben Probleme mit Linksrekursion?
3. Zählen Shift-Reduce-Parser zu Top-Down- oder Bottom-Up-Parsern?
4. Wenn der Suchraum beim Parsen unendlich ist, ist es dann besser einen Tiefe-zuerst- oder einen Breite-zuerst-Parser zu benutzen?
5. Arbeiten Standard-DCG-Parser Top-Down oder Bottom-Up?
6. Hat der Standard-DCG-Parser Probleme mit ε-Produktionen?

Übungsaufgaben

1. Wie könnte man eine kontextfreie Grammatik automatisch so umformen, daß man eine Grammatik erhält, die dieselbe Sprache definiert, aber keine ε-Produktionen enthält?

Kapitel 9

Interpretierendes Parsen

9.1 Compiler und Interpreter

Wenn DCGs nach Prolog übersetzt werden, wird eine Notation, die nicht direkt ausgeführt werden kann, in eine, die direkt ausgeführt werden kann, übertragen. Das nennt man Compilierung. Ein *Compiler* ist ein Programm, das ein Programm in einer *high-level*-Programmiersprache in ein Programm in einer *low-level*-Programmiersprache übersetzt. Das *low-level*-Programm kann dann von einer bestimmten Maschine direkt verarbeitet werden. Normalerweise ist die *high-level*-Programmiersprache so etwas wie LISP, Prolog oder C und die *low-level*-Programmiersprache entspricht dem Maschinencode einer SUN-Workstation oder etwas Ähnlichem. Man kann die DCG-Übersetzung auch als einen solchen Prozeß verstehen: DCG-Grammatiken werden in eine Notation übersetzt, die eine Prolog-Maschine direkt verarbeiten kann. Eine Prolog-Maschine haben wir, wenn wir ein Prolog-System laden und starten. Das Prolog-System kann Prolog-Programme ausführen, indem es sie in echten Maschinencode übersetzt oder einen *Interpreter* für Prolog verwendet.

Interpretation ist eine andere Möglichkeit, Programme einer *high-level*-Programmiersprache abzuarbeiten. Ein Interpreter ist ein Programm, das Programme in einer *high-level*-Programmiersprache Schritt für Schritt abarbeitet. Er simuliert die Arbeit eines compilierten Programms.

Im allgemeinen führt das Compilieren von Programmen zu effektiveren Resultaten als das Interpretieren, weil im Compilat direkt angegeben wird, welche Instruktion als nächste abzuarbeiten ist, wogegen der Interpreter immer wieder die Originalnotation anschauen muß, um den nächsten Schritt zu bestimmen. Andererseits ist es einfacher, einen Interpreter zu verstehen und seine Korrektheit zu überprüfen, als das Resultat einer Übersetzung zu durchschauen. Zum Beispiel wäre es ziemlich aufwendig, alle Differenzlistenargumente der Prolog-

Übersetzung zu prüfen. Es ist viel einfacher, das mit den Interpretern zu tun, die wir im folgenden kennenlernen werden.

Wir haben gesehen, daß die Standard-Prolog-Abarbeitung von DCGs bei bestimmten Grammatiken Probleme mit sich bringt. So terminiert Prolog bei linksrekursiven Grammatiken eventuell nicht. Deshalb ist es sinnvoll, alternative Parsestrategien zu untersuchen. Im folgenden werden verschiedene Bottom-Up-Interpreter, Shift-Reduce-Erkenner und Left-Corner-Erkenner vorgestellt. Diese kann man einfach zu Parsern erweitern, aber die Prinzipien sind an Hand der Erkenner besser zu erklären. Zuerst werden wir die Implementation eines Top-Down-Interpreters beschreiben, der dieselbe Suchstrategie wie das darunterliegende Prolog hat. Wir definieren dazu den Operator `--->`.

```
:- op(1100,xfx,'--->').
```

Außerdem nehmen wir an, daß die Regeln der zu verarbeitenden Grammatik sich in *Chomsky-Normal-Form* befinden, d.h. auf rechten Regelseiten befindet sich entweder ein Terminalsymbol oder zwei Nichtterminalsymbole. Es gibt zu jeder kontextfreien Grammatik eine schwach äquivalente Grammatik in Chomsky-Normal-Form. Das vereinfacht die Implementation des Shift-Reduce-Erkenners wesentlich. Es wäre schwierig, einen Shift-Reduce-Erkenner für DCGs mit beliebigen rechten Regelseiten zu schreiben. Für die anderen Erkenner ist das kein Problem.

9.2 Top-Down-Erkenner

Der folgende DCG-Interpreter arbeitet entsprechend der Standard-Erkennungs-Strategie:

```
recognise(NT,P0,P) :-
   (NT ---> Body),
   recognise_body(Body,P0,P).

recognise_body((Body1,Body2),P0,P) :-
   recognise_body(Body1,P0,P1),
   recognise_body(Body2,P1,P).

recognise_body([Word],P0,P) :-
   connects(P0,Word,P).
recognise_body(Body,P0,P) :-
   Body \= (_,_),
   Body \= [_],
   recognise(Body,P0,P).
```

```
connects([Word|P],Word,P).
```

Dieser Top-Down-Interpreter hat die Form einer in Prolog übersetzten DCG:

```
recognise(X) -->
   {X ---> Body},
   recognise_body(Body).

recognise_body((B1,B2)) -->
   recognise_body(B1), recognise_body(B2).
recognise_body([W]) --> [W].
recognise_body(B) --> {B \= (_,_), B \= [_]},
   recognise(B).

:- recognise(s,String,[]).
```

9.3 Shift-Reduce-Erkenner

Mit dem Erkenner aus dem vorigen Abschnitt hat man natürlich dasselbe Problem wie mit Prolog. Das liegt daran, daß aufgrund von linken Regelseiten Hypothesen über die rechte Regelseite aufgestellt werden. Das heißt, es wird eine von eventuell mehreren passenden Regeln mit entsprechender linken Seite ausgewählt. Ein Bottom-Up-Interpreter dagegen wählt die Regeln anhand der Symbole der rechten Regelseiten aus. Diese Symbole sind wirklich vorhandene Daten und keine Hypothesen.

Wie wir in Kapitel 8.4 gesehen haben, benutzt ein Shift-Reduce-Erkenner eine Hilfsdatenstruktur – einen Stack, der zu Beginn leer ist. Die Kategorien der Worte im String werden der Reihenfolge nach auf den Stack *gepusht*. Shift-Reduce-Erkenner werden so genannt, weil es für jeden Abarbeitungsschritt zwei Optionen gibt: Die Symbole, die sich oben auf dem Stack befinden, können gegen eine rechte Regelseite *gematcht* und durch die linke Seite der entsprechenden Regel ersetzt werden. Man nennt das Reduktion, da die Anzahl der Symbole auf dem Stack sich bei Grammatiken in CNF verkleinert. Die andere Möglichkeit besteht darin, das nächste Eingabewort anzusehen und seine Kategorie auf den Stack zu legen. Die Erkennung ist erfolgreich, wenn das Startsymbol der Grammatik auf dem Stack steht und alle Wörter verarbeitet wurden. Es folgt die Implementation eines Interpreters, der dieser Strategie folgt. Das Argument des Prädikates `recognise` ist der Stack – eine Liste von Kategorien in der umgekehrten Reihenfolge, in der sie im String vorkommen.

```
recognise([],[],[s]).

recognise([Y,X|Rest]) -->            % reduce
   [],
   {LHS ---> X,Y},
   recognise([LHS|Rest]).

recognise(Stack) -->                 % shift
   [Word],
   {Cat ---> [Word]},
   recognise([Cat|Stack]).

:- recognise([],String,[]).
```

9.4 Left-Corner-Erkenner

Der Shift-Reduce-Erkenner sucht eine Regel, deren gesamte rechte Regelseite matcht. Ein Left-Corner-Erkenner benutzt nur das erste Element einer rechten Seite (die linke Ecke *left corner*), um anwendbare Regeln zu finden. Wenn LC erkannt wurde und es eine Regel `M ---> LC,RC` gibt, dann ist `M` erkannt, wenn RC erkannt wird. Es wird eine Hilfsprozedur `lc` benutzt. Diese berechnet eine reflexive, transitive Hülle der Left-Corner-Relation. Zum Beispiel gilt: Wenn ein Determinator eine Left-Corner einer NP ist und eine NP eine Left-Corner eines Satzes, dann ist ein Determinator eine Left-Corner eines Satzes. Ein String kann als Kategorie C erkannt werden, wenn die Kategorie seines ersten Elements eine Left-Corner von C ist.

```
recognise(Phrase) -->
  [Word],
  {Cat ---> [Word]}
  lc(Cat,Phrase).

lc(Phrase,Phrase) --> [].

lc(SubPhrase,SuperPhrase) -->
  {Phrase ---> SubPhrase,Right},
  recognise(Right),
  lc(Phrase,SuperPhrase).

:- recognise(String,[],s).
```

Der Left-Corner-Erkenner hat gegenüber dem Shift-Reduce-Erkenner einige Vorteile. Der Shift-Reduce-Erkenner berechnet jede Konstituente, die aus dem entsprechenden Eingabestring gebildet werden kann, auch solche Konstituenten, die in der gesamten Erkennung keine Bedeutung haben, da ein Satz mit diesen Konstituenten nicht beginnen könnte. Die Left-Corner ermöglicht es, Voraussagen zu treffen und somit den Suchraum zu verkleinern. Es wird ein Top-Down-Element in die Strategie aufgenommen, die ansonsten Bottom-Up ist. Die lc-Relation kann bei linksrekursiven Grammatiken auf unendlich viele Weisen zustande kommen, aber es gibt nur endliche viele Kategoriepaare, die in der lc-Relation stehen. Diese können vor dem Parsen durch Auswertung der Grammatik ermittelt werden. Bevor wir `lc` für ein Kategoriepaar und einen Eingabestring zu beweisen versuchen, können wir überprüfen, ob das überhaupt möglich ist, indem wir ein *table lookup* durchführen. Das heißt, wir schauen in einer vor dem Parsen berechneten Tabelle nach, ob zwei Kategorien überhaupt in der Left-Corner-Relation stehen können. Somit werden von vornherein tote Zweige des Suchbaumes abgeschnitten.

Die Tabelle der möglichen Kategoriepaare wird auch Orakel genannt. Für die Grammatik

```
s ---> np, vp.
np ---> det, n.
det ---> np, gen.
vp ---> v, np.
```

ergibt sich das folgende Orakel:

```
link(s,s).      link(np,s).
link(np,np).    link(det,det).
link(n,n).      link(gen,gen).
link(det,s).    link(det,np).
link(vp,vp).    link(v,vp).
```

Der modifizierte Erkenner hat dann die folgende Form:

```
recognise(Phrase) -->
  [Word],
  {Cat ---> [Word]}
  {link(Cat,Phrase)},
  lc(Cat,Phrase).

lc(Phrase,Phrase) --> [].

lc(SubPhrase,SuperPhrase) -->
  {Phrase ---> SubPhrase,Right},
```

```
{link(Phrase,SuperPhrase)},
recognise(Right),
lc(Phrase,SuperPhrase).
```

So ein Orakel kann zu wesentlichen Verbesserungen in der Laufzeit des Erkenners führen. Left-Corner-Erkenner haben auch keine Probleme mit Linksrekursivität.

Lesetips

Reape (1991) beschäfigt sich mit verallgemeinerten Parsern, die die Bedingung, daß Kategorien auf der rechten Regelseite aneinander grenzen müssen aufheben. Solche Parser sind insbesondere für Sprachen mit relativ freier Wortstellung interessant. Reape gibt in seinem Aufsatz den Prolog-Code für verschiedene Parser. Die Komplexität der verallgemeinerten Algorithmen wird diskutiert. Sie ist exponential, was ein wesentlicher Grund dafür ist, die Worstellung mit anderen Mitteln zu beschreiben (siehe Müller (1999)).

Kontrollfragen

1. Was sind die Hauptunterschiede zwischen Compiler und Interpreter?
2. Was machen die Shift- und Reduce-Operationen in Shift-Reduce-Erkennern?
3. Warum ist es sinnvoll, bei Left-Corner-Erkennern ein Orakel zu benutzen?

Hausaufgabe

5. Das File `~smueller/Public/Prolog/Quellen/Recogniser/rec-count` enthält Versionen der Erkenner aus diesem Kapitel und eine Erweiterung, die jeden Zugriff auf die Grammatik zählt. Die verschiedenen Erkenner können wie folgt aufgerufen werden:

```
?- tdtest(String). % Top-Down-Erkenner
?- srtest(String). % Shift-Reduce-Erkenner
?- lctest(String). % Left-Corner-Erkenner ohne Orakel
?- olctest(String). % Left-Corner-Erkenner mit Orakel
```

wobei `String` eine Liste von Wörtern sein muß. Das System startet den entsprechenden Erkenner, sucht alle Lösungen, meldet eine erfolgreiche Erkennung und gibt auch die Anzahl der Zugriffe auf Regeln zurück.

- Schreiben Sie eine Grammatik, und geben Sie einen Satz an, für den `tdtest` weniger Versuche als `srtest` braucht.
- Schreiben Sie eine Grammatik, und geben Sie einen Satz an, für den `srtest` weniger Versuche als `tdtest` braucht.
- Schreiben Sie eine Grammatik, und geben Sie einen Satz an, für den `olctest` weniger Versuche als `lctest` braucht.

Die Grammatik muß keine Grammatik des Englischen sein. Sie kann sehr einfach sein und muß nicht mehr als 5 Regeln enthalten. Die Parser sollen bei der Analyse der Strings terminieren. Schreiben Sie zu jeder Grammatik ein bis zwei Sätze, die erklären, warum der eine oder andere Erkenner mit dieser Grammatik besser zurechtkommt.

Kapitel 10

Well Formed Substring Table

10.1 Probleme mit Backtracking bei Parsern

Alle Parser, die im vorigen Kapitel angegeben wurden, haben einen Nachteil. Sie benutzen das Prolog-Backtracking, um mit dem Nichtdeterminismus beim Parsen fertigzuwerden. Schlägt die Abarbeitung eines Suchbaumastes fehl, so werden auch bereits erkannte wohlgeformte Konstituenten verworfen, und es wird zum letzten Backtrackpunkt zurückgegangen. Danach muß diese Konstituente eventuell neu geparst werden. Anhand eines Beispiels wird das klarer werden:

$$\begin{array}{l} vp \rightarrow v[ditrans]\ np\ pp \\ vp \rightarrow v[ditrans]\ np\ np \end{array} \tag{10.1}$$

Angenommen wir wollen einen Satz wie (10.2) parsen.

(10.2) I sent the very pleasant double-glazing salesman that I met on holiday in Marbella last year a postcard.

Die erste Grammatikregel wird ausgewählt, und die sehr lange Objektnominalphrase wird geparst. Dann wird nach einer Präpositionalphrase gesucht, und natürlich wird keine gefunden. Alle Ergebnisse werden verworfen, und die zweite Regel wird gewählt. Die gesamte Arbeit des Parsens der Objektnominalphrase wird wiederholt, und erst danach wird *a postcard* als Nominalphrase geparst. Weil die Ergebnisse einer nicht erfolgreichen Suche nicht gespeichert werden, muß die Arbeit wiederholt werden. Obwohl Prolog sehr effizient ist, führt die Unterlassung des Speicherns von wohlgeformten Konstituenten zu einer Komplexität, die exponential mit der Länge des Eingabestrings wächst. Für lange Strings ist das unakzeptabel. In Kapitel 4.6 haben wir gesehen, daß die Komplexität des Parsens im Gegensatz zu der des Erkennens im ungünstigsten Fall für eine kontextfreie Grammatik exponential ist. Das heißt, daß es, falls es exponential viele Analysen

für einen String gibt, keinen Parser geben kann, der eine Komplexität unterhalb von exponential hat. In unserem Beispiel gibt es aber nur eine Lösung, und das Suchverhalten von Prolog ist trotzdem exponential. Außerdem hängt diese Exponentialität garnicht vom Problem des Parsens ab – selbst das Erkennen hat bei Erkennern mit Backtracking exponentiale Komplexität. Wir brauchen also bessere Erkenner als Grundlage für die Parser.

Die Lösung dieses Problems ist die Verschiebung der Komplexität von der Kontroll- in die Datenstruktur. Die Platz-Komplexität des Backtrackings eines Parsers ist linear. Indem wir eine Datenstruktur benutzen, deren Größe durch n^2 bei einer Stringlänge n begrenzt ist, können wir die Zeit-Komplexität auf polynomial reduzieren. Wir werden das am Beispiel eines Bottom-Up-Tabellen-Parsers demonstrieren, bei dem wir sicherstellen, daß keine Konstituente mehr als einmal geparst wird. Dabei können auch Konstituenten entstehen, die wir nicht brauchen. Wir werden den Algorithmus später so verändern, daß nur noch solche Konstituenten gebildet werden, die auch wirklich gebraucht werden.

Um die Wiederholung von Arbeit in verschiedenen Teilen des Suchraumes zu vermeiden, muß ein Parser erfolgreich geparste Konstituenten speichern und diese gespeicherten Resultate für die weitere Verarbeitung benutzen. Die abstrakte Datenstruktur, die benutzt wird, um die Resultate zu speichern, wird *Well-Formed Substring Table* oder *Chart* genannt. Da ein Parser mit polynomialer Komplexität eine Chart nutzen muß, gibt es sehr viel Literatur über das Chart-Parsing.

Für die Backtrack-Algorithmen haben wir reines Prolog benutzt. Der Prolog-Suchalgorithmus hat die für die Suche benötigten Datenstrukturen verwaltet. Es ist möglich, einen Chart-Parser in Prolog zu implementieren. Da man aber gezwungen ist, die Suche selbst zu verwalten, gibt es keine Vorteile für Prolog gegenüber anderen Programmiersprachen. Im folgenden werden die Algorithmen in einem ALGOL/PASCAL-ähnlichen Pseudo-Code angegeben.

Um die Erklärung einfach zu halten, betrachten wir zuerst Grammatiken in Chomsky-Normal-Form. Das erleichtert den Nachweis der Korrektheit und der Vollständigkeit des Parsers. Wir werden die CNF-Einschränkung dann wieder aufheben und einen Parser für allgemeine kontextfreie Grammatiken schreiben.

10.2 Der Cocke-Kasami-Younger-Algorithmus

Der im folgenden vorgestellte Algorithmus geht auf Arbeiten verschiedener Personen zurück und wird als CKY-Algorithmus bezeichnet. Wenn ein String aus n Wörtern besteht, wird die Chart als Matrix der Größe $(n+1)^2$ gespeichert. Die Indizes gehen von 0 bis n. Sie entsprechen Stringpositionen – Positionen zwischen den Wörtern und an jedem Ende des Strings. Die Einträge in der Chart entsprechen Konstituenten, die zwischen bestimmten Positionen im String gefunden wurden.

Die Einträge sind Mengen von Nichtterminalsymbolen. Wir werden im folgenden wieder nur einen Erkenner beschreiben. Wenn wir aus der Chart Parsebäume konstruieren wollen, müssen wir noch zusätzliche Information speichern.

Der Prozeß der Erkennung entspricht einem systematischen Ausfüllen der Matrix. Die Menge *chart(i,j)* ist die Menge aller Kategorien von Konstituenten, die an Position *i* beginnen und an Position *j* enden. Die Matrix ist dreieckig, da keine Konstituente vor ihrem Anfangspunkt endet. Die Erkennung ist erfolgreich, wenn die vollständig ausgefüllte Tabelle das Startsymbol S in der Menge *chart(0,n)* enthält.

Es ist wichtig, die Matrix systematisch auszufüllen. Wir müssen garantieren, daß ein Matrixeintrag vollständig ist, das heißt, daß er alle möglichen Konstituenten enthält, die im gerade bearbeiteten Teil des Strings enthalten sind. Nur wenn der Eintrag vollständig ist, können wir ihn zur Berechnung weiterer Einträge benutzen. Wäre es möglich, daß unvollständige Einträge in der Chart sind, so wüßte der Erkenner, wenn er eine bestimmte Teilkonstituente sucht, nicht, ob er in der Chart nachsehen kann, oder ob er diese Konstituente noch parsen muß. Somit wäre die Chart nutzlos.

Um die Vollständigkeit zu sichern, bauen wir alle Konstituenten, die an einem bestimmten Punkt enden, bevor wir irgendeine Konstituente bauen, die an einem späteren Punkt endet. An einem bestimmten Endpunkt bauen wir kleinere Konstituenten bevor wir größere bauen. Unser Erkenner arbeitet also mit Tiefe-zuerst-Suche und Bottom-Up. Es ist einfach, statt dessen eine Breite-zuerst-Suche zu verwenden. Darauf wird in Kapitel 11.4 näher eingegangen.

Ein Eintrag in die Chart wird wie folgt berechnet:

$$chart(i,j) = \bigcup_{i<k<j} chart(i,k) * chart(k,j) \tag{10.3}$$

Das Symbol $*$ ist eine Infix-Funktion, die zwei Symbolmengen $\{a_1, a_2, \ldots a_m\}$ und $\{b_1, b_2, \ldots b_p\}$ nimmt und eine Menge $\{g_1, g_2, \ldots g_s\}$ zurückgibt, wobei für alle g eine Regel $g \rightarrow \alpha\beta$ existiert, mit $\alpha \in \{a_1, a_2, \ldots a_m\}$ und $\beta \in \{b_1, b_2, \ldots b_p\}$. Mit einer entsprechenden Grammatik gilt:

$$\{np, n, verb, prep\} * \{np, s\} = \{vp, pp\} \tag{10.4}$$

Der CKY-Algorithmus kann in kubischer Zeit ausgeführt werden. Man muß alle Kombinationen von *i, j* und *k* berücksichtigen. Dabei gibt es maximal *n* verschiedene Werte für *i, j* und *k*. Die Komplexität der Aktion in der innersten Schleife ist konstant. Sie ist durch das Quadrat der Anzahl der Nichtterminale begrenzt. Da diese nur von der Größe der Grammatik abhängt, wird sie auch Grammatikkonstante genannt.

Der Algorithmus:

```
for j from 1 to n do
    set chart(j−1, j) to {A|A → word_j}
    for i from j−2 downto 0 do
        for k from i+1 to j−1 do
            set chart(i, j) to chart(i, j) ∪ (chart(i,k) ∗ chart(k, j))
        endfor
    endfor
endfor
if S ∈ chart(0,n) then akzeptiere else akzeptiere nicht
```

Dabei gilt:

- j ist die Stelle im String, bis zu der alle vollständigen Phrasen berechnet sind,
- i ist eine Anfangsstelle, die kleiner als j ist, und alle Phrasen zwischen i und j werden berechnet
- k ist eine Stelle zwischen i und j. Eine Phrase zwischen i und j wird gefunden, wenn eine Phrase zwischen i und k und eine zwischen k und j existiert und beide Phrasen der rechten Seite einer Regel entsprechen.

Die Reihenfolge der Abarbeitung kann auch geändert werden. Solange man nicht versucht, einen Chart-Eintrag zu benutzen, bevor er vollständig ist, ändert das aber nichts an der Effizienz des Algorithmus.

10.3 Mehrdeutigkeit

Ein Satz mit n Präpositionalphrasen nach dem Verb hat bei einer Grammatik wie der folgenden Catalan($n+2$) Parsebäume.

(10.5)

np → tigger	v → chases
n → dog	n → bone
n → garden	det → a
s → np vp	vp → v np
vp → vp pp	np → det n
np → np pp	pp → p np

Man mache sich das selbst klar indem man die Bäume für den Satz *Tigger chases a dog with a bone round the garden.* aufzeichnet. Obwohl eine Konstituente wie

chases a dog with a bone zwei Parsemöglichkeiten als VP hat, werden übergeordnete Konstituenten, die diese VP nutzen, nicht zweimal in die Chart eingetragen.

(10.6) $_0$ chases $_1$ a $_2$ dog $_3$ with $_4$ a $_5$ bone $_6$

Die folgende Tabelle zeigt ein Beispiel. In den Spalten stehen Phrasen mit gleichem Endpunkt und in den Zeilen Phrasen mit gleichem Anfangspunkt.

0	1	2	3	4	5	6
1	v		vp			vp
2		det	np			np
3			n			
4				p		pp
5					det	np
6						n

Die mir leichter verständliche Notation zeigt Abbildung 10.1. In dieser Abbildung sind Kanten vom Beginn zum Ende einer Konstituente eingezeichnet. Je eine Kante entspricht einem Eintrag in die Chart.

10.4 Vom Erkenner zum Parser

Wir haben bis jetzt einen Algorithmus für einen Erkenner geschrieben, der nicht genug Information verwaltet, um die Ableitungsgeschichte eines Satzes rekonstruieren zu können. Um diese zusätzliche Information zu speichern, ohne die Komplexität zu verschlechtern, müssen wir die möglichen Analysen in Form der einzelnen Teile repräsentieren. Wir tragen also statt einer Kategorie ein Tripel der Form

$$< Category, Rule, Pos > \tag{10.7}$$

in die Chart ein. Dabei ist *Rule* die Regel der Grammatik, die den Eintrag der Kategorie *Category* in die Chart rechtfertigt, und *Pos* ist die Position im String, an der sich die zwei Teilphrasen der rechten Regelseite treffen. Das und die Information darüber, wo *Category* beginnt und endet, ist dann genug Information für die Rekonstruktion des Parsebaumes. Man kann so rekursiv die Tabelleneinträge finden, die zu entsprechenden Teilphrasen gehören, und alle möglichen Analysen generieren.

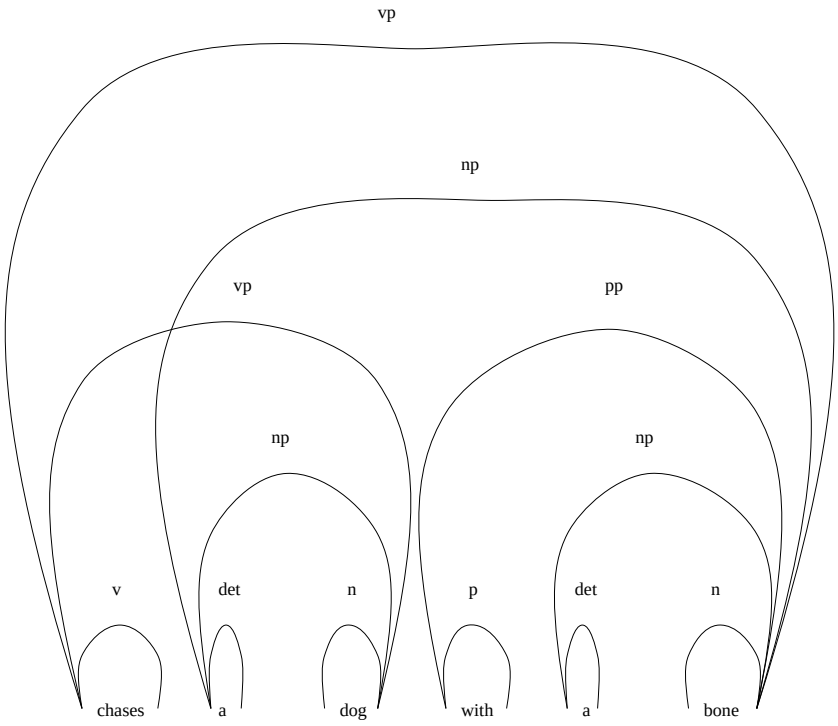

Abbildung 10.1: Konstituenten in *chases a dog with a bone*

Für ein solcherart modifiziertes System muß man die $*$-Operation so definieren, daß sie die zusätzliche Information bei den Eingabekategorien ignoriert, aber zusätzliche Informationen bei der Ausgabekategorie erzeugt.

$$
\begin{aligned}
&chart(i,k) * chart(k,j) = \\
&\{< LHS, R, k > \mid \\
&Regel\ R\ der\ Grammatik\ ist\ LHS \rightarrow RHS_1\ RHS_2 \\
&und\ es\ gibt\ ein\ Element\ < RHS_1, _, _ >\ in\ chart(i,k) \\
&und\ es\ gibt\ ein\ Element\ < RHS_2, _, _ >\ in\ chart(k,j)\ \}
\end{aligned}
\tag{10.8}
$$

Dabei ist zu beachten, daß, wenn es mehrere Analysen einer Teilphrase (z. B. RHS_1) gibt, diese nicht zu mehrfachen Einträgen von Phrasen führt, die sie enthalten. Bei der Konstruktion der Menge $< LHS, R, k >$ werden doppelte Elemente einfach nicht berücksichtigt.

Wir können so auch Grammatiken behandeln, die Regeln wie $S \rightarrow S$ beinhalten. Solche Grammatiken erzeugen für jeden String, der ein S ist, eine unendliche Anzahl von Parsebäumen. Die Aufzählung aller Parsebäume würde also niemals terminieren. Trotzdem terminiert aber unsere erweiterte Form des Erkenners, und man kann die ausgefüllte Chart dazu benutzen, die unendliche Menge von Parsebäumen aufzuzählen solange man will.

10.5 Nachteile des Bottom-Up Parsens

Ein Nachteil der Bottom-Up-Strategie ist, daß alle Konstituenten, die durch die Grammatik gerechtfertigt sind, auch gebaut werden, egal ob sie in einer vollständigen Phrase auftauchen könnten oder nicht. Ein Beispiel dafür, daß unsinnige Phrasen eingetragen werden ist der Satz *The search for Spock was successful.* Die Phrase *search for Spock* wird nicht nur als Teil der Nominalphrase gefunden, sondern auch als Verbphrase. Eine Verbphrase kann aber nicht direkt nach einem Artikel wie *the* stehen. Eine Top-Down-Vorhersage würde also den Suchraum verkleinern. Im folgenden Kapitel werden wir zeigen, wie man eine Chart mit Top-Down-Vorhersage benutzen kann.

Kontrollfragen

1. Fertigen Sie Tabellen wie die auf Seite 90 an, und tragen Sie die entsprechenden Kategorien so ein, wie das der CKY-Algorithmus für die folgenden Phrasen tun würde.

   sent a postcard
   sent a man a boat
   sent the man on a boat with the hat a postcard

Kapitel 11

Active Chart Parsing

11.1 Regeln mit Punkten

Wir verallgemeinern den CKY-Algorithmus so, daß er auch für Grammatiken mit mehr als zwei Nichtterminalen in rechten Regelseiten funktioniert. Dafür führen wir Regeln mit Punkten ein. Ein Punkt in einer Regel sagt etwas über die bereits gefundenen Konstituenten einer rechten Seite aus.

$$\begin{aligned} vp &\rightarrow .v\ np\ pp \\ vp &\rightarrow v.\ np\ pp \\ vp &\rightarrow v\ np.\ pp \\ vp &\rightarrow v\ np\ pp. \end{aligned} \tag{11.1}$$

Anstelle des Multiplikationsschrittes, der bereits vollständige Konstituenten verknüpft, wird eine zum Teil vollständige Konstituente mit einer vollständigen verknüpft. Der Punkt in den oben angegebenen Regeln mit Punkt steht immer vor den Konstituenten, die noch gefunden werden müssen und nach den Konstituenten, die schon im Eingabestring gefunden wurden. In der ersten Regel mit Punkt wurde noch garnichts gefunden, und die letzte entspricht einer vollständigen Verbphrase. Die anderen beiden Regeln entsprechen unterschiedlich vollständigen Verbphrasen.

Zur Repräsentation einer Kante in der Chart gehört eine Regel mit Punkt und deren Anfangs- und Endpunkte. Wenn die erkannten Konstituenten aus der Regel $vp \rightarrow v\ np.pp$ zwischen Position 3 und Position 8 liegen, schreibt man:

$$_3[vp \rightarrow v\ np\ _8\ pp] \tag{11.2}$$

Dabei entspricht die Zahl der Endposition dem Punkt in der Regel.

Die Multiplikationsregel für Mengen von Kanten ist unabhängig von der Grammatik und hat die Form:

$$S_1 * S_2 = \{ {}_i[L \rightarrow AX_jB] \mid {}_i[L \rightarrow A_k\ X\ B] \in S_1 \textit{ und } {}_k[X \rightarrow \ldots\ {}_j] \in S_2 \} \tag{11.3}$$

Weil wir öfter über zwei Kanten reden, die kombiniert werden sollen, als über die Kombination zweier Mengen, definieren wir das $*$-Symbol auch für Kanten:

$${}_i[L \rightarrow A_k\ X\ B] * {}_k[X \rightarrow \ldots\ {}_j] = {}_i[L \rightarrow AX_jB] \tag{11.4}$$

Dabei ist L ein Nichtterminal, X ein Terminal- oder Nichtterminalsymbol und A und B sind eventuell leere Strings aus Nichtterminalen. Die Gleichung in (11.4) wird *Fundamentalregel* des Chart-Parsens genannt. Beispiele sind:

$$\begin{aligned} {}_2[vp \rightarrow v\ {}_3\ np\ pp] * {}_3[np \rightarrow \ldots {}_5] &= {}_2[vp \rightarrow v\ np\ {}_5\ pp] \\ {}_2[vp \rightarrow v\ np\ {}_5\ pp] * {}_5[pp \rightarrow \ldots {}_8] &= {}_2[vp \rightarrow v\ np\ pp\ {}_8] \end{aligned} \tag{11.5}$$

Weil eine Kante mit einem Punkt, der nicht am Ende der Regel steht, etwas darüber aussagt, welche Kategorien noch nach diesem Punkt kommen müssen, und weil die Kante somit eine aktive Rolle im Parseprozeß spielt, wird eine solche Kante *aktive Kante* genannt. Eine Kante mit Punkt am Ende wird *vollständig* oder *inaktiv* genannt. Die entsprechenden Parser werden *Active-Chart-Parser* genannt.

11.2 Bottom-Up-Einbeziehung

Es stellt sich jetzt die Frage, wie die aktiven Kanten überhaupt in die Chart kommen. Wir werden im folgenden zwei mögliche Ansätze vorstellen.

Die Idee der Regeln mit Punkt kann in einen Bottom-Up-Parser integriert werden. Wie beim Left-Corner-Parser aus Kapitel 9.4 benutzen wir linke Ecken bzw. *left corners*, um auf Regeln zuzugreifen. Wenn ein Wort in die Chart eingetragen wird, werden auch Regeln mit Punkt für all die Grammatikregeln eingetragen, von denen das Wort eine linke Ecke ist. Der Punkt steht dann nach der Kategorie des Wortes, das zur Eintragung der Regel mit Punkt in die Chart führte. Wenn wir zum Beispiel ein Verb zwischen Position 4 und 5 finden und die Grammatik eine Regel der Form $vp \rightarrow v\ np\ pp$ enthält, dann wird die Regel mit Punkt ${}_4[vp \rightarrow v\ {}_5\ np\ pp]$ in die Chart eingetragen. Diese Regel mit Punkt entspricht der Hypothese, daß es im Satz eine vp gibt, die ein Verb enthält, das zwischen Position 4 und 5 liegt, und die noch eine np und eine pp braucht, um vollständig erkannt zu sein.

Der Gazdar-Mellish-Chart-Parser

Das Programm, das im folgenden beschrieben wird, ist eine Abwandlung des Chart-Parsers, der in (Gazdar und Mellish, 1989, Kapitel 6.7) vorgestellt wurde. Wesentliche Programmteile wurden übernommen.

Jedesmal, wenn eine inaktive Kante der Kategorie C in die Chart aufgenommen wird, werden alle Regeln, die auf der rechten Regelseite mit C beginnen, in die Chart eingetragen. Der Punkt steht bei den neuen Einträgen hinter C, da C bereits erkannt wurde. Etwas genauer formuliert sieht das wie folgt aus:

Bottom-Up-Regel
Wenn eine inaktive Kante der Form $C \rightarrow W1.$, die von i nach j geht, in die Chart eingetragen wird,
dann nimm für jede Regel der Form $B \rightarrow C\ W2$ eine Kante $B \rightarrow C.\ W2$ von i nach j in die Chart auf.

Dabei sind W1 und W2 eventuell leere Folgen von Terminal- und Nichtterminalsymbolen.

Bei der Prolog-Implementation eines Chart-Parsers benutzen wir die Prolog-Datenbank, um Kanten abzuspeichern. Dabei wird das folgende Format zugrunde gelegt:

```
edge(Start,Finish,Label,ToFind,Found).
```

Wir brauchen ein Prädikat, das die Chart initialisiert. Dieses Prädikat muß alle aktiven Kanten für jedes Wort im String in die Chart eintragen. Das Prädikat `start_chart` hat einen String als drittes Argument und eine Startposition als erstes Argument. Das Prädikat trägt die Kategorien der Wörter im String in die Chart ein und instantiiert das zweite Argument mit der letzten Position im String. Wenn der String leer ist, macht `start_chart` garnichts, und gibt die Position im String zurück:

```
start_chart(V0,V0,[]).
```

Ansonsten nimmt das Prädikat das erste Wort im String und fügt für jeden Lexikoneintrag dieses Wortes eine entsprechende Kategorie in die Chart ein.

```
start_chart(V0,Vn,[Word|Words]) :-
    V1 is V0+1,
    foreach(word(Category,Word),
        add_edge(V0,V1,Category,[],[Word,Category])),
    start_chart(V1,Vn,Words).
```

`foreach` ist wie folgt definiert:

```
foreach(X,Y) :-
   X, once(Y), fail.
foreach(_,_).

once(X) :- X, !.
```

Prolog sucht eine Lösung für `X`. Für diese Lösung von `X` wird einmal `Y` aufgerufen. Danach wird durch das `fail` ein Backtracking ausgelöst. Damit keine anderen Lösungen für `Y` generiert werden, ist das `Y` durch `once` gekapselt. So werden alle Möglichkeiten für `X` gefunden, und für jede wird `Y` genau einmal ausgeführt. Dabei muß `Y` nicht unbedingt für alle `X` gelingen. `foreach` ist nur für `Y`s sinnvoll, die die Datenbasis verändern oder nur Ausgabefunktionen haben, da ansonsten alle Lösungen durch das `fail` verlorengehen. Die zweite Klausel von `foreach` sorgt dafür, daß `foreach` immer erfolgreich ist.

In der zweiten Klausel von `start_chart` wird also für jede Kategorie-Wort-Kombination `add_edge` aufgerufen, die eine Lösung des Prolog-Rufes `word( Category, Word)` ist. Die Kategorie des Wortes wird in `Found` aufgenommen. Beim Ausfüllen der Chart werden die Parsebäume gleich mitkonstruiert.

Das Prädikat `add_edge` leistet noch mehr, als nur eine Kante in die Chart einzutragen. Es bestimmt außerdem, welche Kanten noch hinzugefügt werden müssen. Wenn eine Kante bereits in der Chart ist, passiert garnichts:

```
add_edge(V0,V1,Category,Categories,Parse) :-
    edge(V0,V1,Category,Categories,Parse),!.
```

Ansonsten wird die Kante der Datenbank hinzugefügt. Wenn eine inaktive Kante der Kategorie `Category` von `V1` nach `V2` der Datenbank hinzugefügt wird, das ToFind-Argument also eine leere Liste ist, dann werden auch für alle Regeln, deren erstes Symbol der rechten Regelseite `Category` ist, Kanten von `V1` nach `V2` hinzugefügt.

```
add_edge(V1,V2,Category1,[],Parse) :-
    asserta(edge(V1,V2,Category1,[],Parse)),

    foreach(rule(Category2,[Category1|Categories]),
        add_edge(V1,V2,Category2,Categories,[Parse,Category2])),
```

Der zweite Teil der Klausel sorgt dafür, daß für alle aktiven Kanten, die bis jetzt in der Chart sind und nach einer `Category1` suchen, eine neue Kante, die entweder aktiv oder inaktiv sein kann und bei `V2` endet, in die Chart eingetragen wird.

```
    foreach(edge(V0,V1,Category2,[Category1|Categories],Parses),
        add_edge(V0,V2,Category2,Categories,[Parse|Parses])).
```

Diese Klausel behandelt die Aufnahme inaktiver Kanten in die Chart. Wir brauchen aber noch eine Klausel, die für die Eintragung aktiver Kanten in die Chart verantwortlich ist. Beim Eintragen einer aktiven Kante wird einfach nach inaktiven Kanten gesucht, die sich mit der aktiven Kante kombinieren lassen.

```
add_edge(V0,V1,Category1,[Category2|Categories],Parses) :-
    asserta(edge(V0,V1,Category1,[Category2|Categories],Parses)),

    foreach(edge(V1,V2,Category2,[],Parse),
        add_edge(V0,V2,Category1,Categories,[Parse|Parses])).
```

Das ist im Prinzip schon der vollständige Parser. Das einzige, was noch benötigt wird, ist ein Prädikat `parse`, das die Chart initialisiert, d.h. eventuell noch vorhandene Kanten entfernt und `start_chart` aufruft.

```
parse(String) :-
    retractall(edge(_,_,_,_,_)),
    start_chart(1,Vn,String),
    edge(1,Vn,s,[],_),
    foreach(edge(1,Vn,s,[],Parse),
            ( write(Parse),
              nl )
           ).
```

11.3 Top-Down-Einbeziehung – der Earley Algorithmus

Im folgenden werden wir einen Top-Down-Active-Chart-Parser vorstellen. Der Algorithmus wurde von Jay Earley 1970 entwickelt. Er ist der effektivste Algorithmus zum Parsen von kontextfreien Grammatiken.

Wie wir in Kapitel 10.5 festgestellt haben, findet eine Bottom-Up-Strategie jede Phrase, sogar solche, die nicht zu vollständigen Phrasen innerhalb eines Satzes beitragen können.

Eine Top-Down-Vorhersage kann in den Chart-Parser wie folgt integriert werden: Ein Eintrag in der Diagonale einer Chart repräsentiert eine Phrase der Länge Null, die vom Scheitelpunkt i zum Scheitelpunkt i für ein beliebiges i geht. Ein solcher Eintrag auf der Diagonale entspricht der Hypothese, daß es eine Konstituente LHS gibt, die an diesem Punkt beginnt und von der noch keine Teilkonstituenten gefunden wurden. Die einzige sinnvolle Hypothese, die wir zu Beginn des

Parsens aufstellen können, ist, daß es einen Satz gibt, der bei Position 0 beginnt. Deshalb tragen wir für jede Regel mit einem *s* auf der linken Seite eine aktive Kante in die Chart ein. Zum Beispiel:

$$_0[s \rightarrow_0 np\ vp] \tag{11.6}$$

Da die aktiven Kanten den Prozeß des Parsens steuern, und zu Beginn die aktiven Kanten in der Chart genau die sind, von denen wir wissen, daß sie sinnvoll sind, ist die Anzahl der sinnlosen Konstituenten, die gebaut werden, gegenüber dem CKY-Algorithmus gering.

Earley definierte seinen Algorithmus durch drei Fälle, die auftreten können, wenn eine Kante in die Chart aufgenommen wird.

1. *Vorhersage* Die Kante, die eingetragen wird, ist aktiv, aber es gibt keine Kante in der Chart, mit der sie kombiniert werden könnte. In diesem Fall werden alle Hypothesen, also Regeln mit Punkt ganz am Anfang, für die benötigte Kante eingetragen.

2. *Scannen* Die Kante, die eingetragen wird, ist aktiv, und es gibt bereits Kanten in der Chart, die mit ihr kombiniert werden können. Für jede dieser Kanten wird die Multiplikationsregel angewendet. Dadurch entstehen weitere Kanten.

3. *Vervollständigung* Die Kante, die eingetragen wird, ist vollständig. Für alle Kanten, mit denen die eingetragene Kante kombiniert werden kann, wird die Multiplikationsregel angewendet. Dadurch entstehen weitere Kanten. Man beachte, daß es entsprechende aktive Kanten in der Chart geben muß. Diese aktiven Kanten sind durch Vorhersage entstanden.

Wenn mehrere Kanten durch eine dieser Operationen erzeugt werden, gibt es verschiedene Möglichkeiten, diese Kanten in die Chart einzutragen. Entweder wir fügen die Kanten, die bei der Anwendung einer Operation entstehen, alle auf einmal ein, oder wir fügen die erste Kante ein und dann alle Kanten, die durch dieses Einfügen entstehen, und dann die restlichen Kanten. Die erste Alternative entspricht einer Breite-Zuerst- und die zweite einer Tiefe-Zuerst-Suche. Wir werden darauf weiter unten noch genauer eingehen.

Eine andere Frage ist, wie man die Chart initialisiert. Sagen wir das Startsymbol voraus, oder werden die lexikalischen Kanten zuerst eingetragen? Mit der ersten Strategie wird eine vollständige Kante erst eingetragen, wenn alle Vorhersagen bis zum Anfang dieser Kante gemacht wurden. Das bedeutet, das die Bedingungen für das Scannen nie vorhanden sind. Wenn immer die Vorhersagen zuerst eingetragen werden, können nichtdiagonale Kanten nur als Resultat der Vervollständigung auftreten, da linke Kanten immer vor rechten Kanten, mit denen sie

kombiniert werden müssen, in die Chart aufgenommen werden. Wir werden diesen Ansatz weiterverfolgen.

Zur Erklärung nehmen wir eine dreidimensionale Matrix von Boolean-Werten an. Die zweite und die dritte Dimension sind Anfangs- und Endpunkte von Kanten, und die erste Dimension entspricht möglichen Zuständen. Für einen bestimmten Zustand s und Chart-Positionen i und j kann $chart(s,i,j)$ entweder *true* oder *false* sein. In einer Grammatik mit den Regeln

$$\begin{aligned} &s \rightarrow np\ vp \\ &np \rightarrow det\ n \\ &vp \rightarrow v\ np\ np \end{aligned} \tag{11.7}$$

sind die möglichen Zustände:

$$\begin{aligned} &s \rightarrow .np\ vp \quad s \rightarrow np.vp \quad s \rightarrow np\ vp. \\ &np \rightarrow .det\ n \quad np \rightarrow det.n \quad np \rightarrow det\ n \\ &vp \rightarrow .v\ np\ npvp \rightarrow v.np\ npvp \rightarrow v\ np.np \\ &vp \rightarrow v\ np\ np. \end{aligned} \tag{11.8}$$

Es folgt der Algorithmus fürs Chart-Parsen (bzw. für den darunterliegenden Erkenner):

```
procedure chartparse:

   for state in predictions(... --> .s) do
       enter_edge(state,0,0)
   for j from 1 to n do
       for state in {(A --> . | A --> wordj} do
           enter_edge(state,j-1,j)


procedure enter_edge(state1,i,j):

   if chart(state1,i,j) = false then
      set chart(state1,i,j) to true

      % predict
      for state2 in predictions(state1) do
          enter_edge(state2,j,j)

      % complete
      for each k, state2 such that chart(state2,k,i) is true do
          if left_sister(state2,state1) then
             enter_edge(state2*state1,k,j)

      % scan
      for each k, state2 such that chart(state2,j,k) is true do
```

```
if right_sister(state2,state1) then
   enter_edge(state1*state2,k,j)
```

Dabei wurde das Scannen der Vollständigkeit halber mit aufgeführt. Wenn wir die Kanten in der oben angegebenen Reihenfolge in die Chart eintragen, hat der Scan-Teil keine Aufgabe. Wir tragen zuerst die Regeln mit Punkt am Anfang, die Hypothesen für das Startsymbol entsprechen, und die Hypothese, die sich daraus ergeben, ein. Dann lesen wir nacheinander alle Wörter und tragen Regeln mit Punkt am Ende für die Kategorien dieser Wörter ein. Alle Operationen, die das Eintragen der Kategorie eines Wortes nach sich zieht, werden ausgeführt, bevor das nächste Wort verarbeitet wird.

Wenn eine Kante in die Chart aufgenommen wird, wird überprüft, ob es nicht schon genau so eine Kante in der Chart gibt. Ist das der Fall, so passiert nichts. Durch diesen Test werden die Probleme vermieden, die der Prolog-Interpreter mit linksrekursiven Grammatiken hat. Wenn eine Kante schon in der Chart ist, kehrt `enter_edge` ohne etwas in der Chart zu verändern zurück.

Der Algorithmus benutzt eine Funktion `predictions`, die aus der Menge der Zustände in Mengen von Zuständen und zwei Prädikate mit jeweils zwei Zuständen als Argument: `left_sister` und `right_sister`. `predictions` kann vor dem Parsen berechnet werden. Die Funktion berechnet zu einem Zustand eine Menge aller möglichen Zustände, die Grammatikregeln entsprechen, die man anwenden könnte, um die nächste gesuchte Phrase zu finden. Für die obige Grammatik:

$$\begin{aligned} &\mathit{predictions}(s \rightarrow np.vp) &&= \{vp \rightarrow .v\ np\ np\ \} \\ &\mathit{predictions}(vp \rightarrow v.np\ np) &&= \{np \rightarrow .det\ n\} \end{aligned} \tag{11.9}$$

Wenn das Symbol nach dem Punkt nur als lexikalische Kategorie exisitiert, ist das Ergebnis der Funktion `predictions` die leere Menge. In `enter_edge` hat eine aktive Kante mit dem Punkt direkt vor A in der Menge der `predictions` alle Zustände, deren linke Regelseite ein A ist. Der Punkt steht in diesen Zuständen am Anfang der rechten Regelseite. Diese Zustände werden auf der Diagonale der Chart durch einen rekursiven Aufruf von `enter_edge` an der entsprechenden Stelle eingetragen.

`left_sister(x,y)` gibt für zwei Zustände x und y *true* zurück, falls x nach links mit y kombiniert werden kann.

```
left_sister((s --> . np vp), (np --> det n .)) = true
left_sister((np --> det. n), (np --> det n .)) = false
left_sister((s --> .np vp),  (np --> det. n )) = false
```

Linke Schwesterzustände einer vollständigen Kante der Kategorie B sind genau die Zustände, die einen Punkt vor dem B in ihrer rechten Seite haben. Für jede

Kante mit linken Schwestern, die dort enden, wo die vollständige Kante anfängt, werden neue Kanten, die sowohl die Schwester als auch B umfassen, eingetragen. (Vervollständigung).

`right_sister(x,y)` funktioniert analog für die Kombination nach rechts:

```
right_sister((np --> det n .), (s --> . np vp)) = true
right_sister((np --> det n .), (np --> det. n)) = false
right_sister((np --> det. n ), (s --> .np vp )) = false
```

Rechte Schwesterzustände einer vollständigen Kante der Kategorie B sind genau die Zustände, die einen Punkt vor dem B in ihrer rechten Seite haben. Für jede Kante mit rechten Schwestern, die dort enden, wo die vollständige Kante anfängt, werden neue Kanten, die sowohl die Schwester als auch B umfassen, eingetragen. (Scannen).

`left_sister` und `right_sister` könnten auch schon vorher berechnet werden, aber in der Praxis wird das meist durch einfache Tests während der Laufzeit erledigt. Damit `left_sister` wahr ist, muß die erste Kategorie nach dem Punkt des ersten Zustandes der linken Seite des zweiten Zustandes entsprechen, und der zweite Zustand muß vollständig sein. Genauso muß für `right_sister` gelten, daß der erste Zustand vollständig sein muß und die linke Seite des ersten Zustands der Kategorie nach dem Punkt im zweiten Zustand gleicht.

Jeder Zustand hat entweder rechte oder linke Schwestern. Ein aktiver Zustand hat nur rechte Schwestern und ein inaktiver Zustand nur linke. Nur ein aktiver Zustand hat eine nichtleere `predictions`-Menge.

11.4 Die Tagesordnung (*Agenda*)

Der Earley-Algorithmus fügt Kanten in einer bestimmten von Links-nach-Rechts-Reihenfolge in die Chart ein. Obwohl das die Implementation etwas vereinfacht, weil z.B. kein Scanning benötigt wird, ist diese Reihenfolge nicht unbedingt vorgeschrieben. Das Einzige, was gewährleistet sein muß, ist, daß für jede Eintragung einer neuen Kante auch alle sich aus dieser Eintragung durch Vorhersage, Vervollständigung oder Scanning ergebenden Folgeeintragungen ebenfalls eingetragen werden, falls sie nicht schon in der Chart sind.

Wir können den Algorithmus etwas flexibler gestalten, indem wir eine Datenstruktur benutzen, die Tagesordnung oder *agenda* genannt wird. Anstatt eine Kante direkt in die Chart zu schreiben, fügen wir sie in die Tagesordnung ein. Der Algorithmus fängt mit einer Agenda an, die lexikalische Kanten und Vorhersagen für das Startsymbol enthält. Danach wird folgende Schleife ausgeführt:

```
while die Tagesordnung enthält Kanten do
    nimm eine Kante aus der Tagesordnung
    und trage sie in die Chart ein
```

Der Rest des Algorithmus bleibt gleich, nur die rekursiven Aufrufe von `enter_edge` werden durch Aufrufe einer Prozedur ersetzt, die die Kanten in die Tagesordnung einträgt.

Wir können die Reihenfolge, in der Kanten der Chart hinzugefügt werden, einfach dadurch ändern, daß wir verschiedene Strategien benutzen, um ein Element aus der Tagesordnung zu wählen. Wenn die Tagesordnung als Stack verwaltet wird, ist die letzte Kante, die in die Tagesordnung aufgenommen wurde, die erste Kante, die in die Chart eingetragen wird. Das Ergebnis ist ein Tiefe-Zuerst-Parser. Werden zwei verschiedene Kanten der Tagesordnung hinzugefügt – vielleicht Kanten für ein lexikalisch mehrdeutiges Wort –, dann werden in einem Stack-basierten System die erste Kante und alle sich daraus ergebenden Kanten eingetragen, bevor die zweite Kante berücksichtigt wird. Ist die Tagesordnung dagegen eine Schlange, so haben wir eine Breite-zuerst-Suche. Man kann sich auch noch kompliziertere Kriterien für die Auswahl einer Kante aus der Tagesordnung ausdenken. Zum Beispiel kann man die Kante wählen, die uns am weitesten im String voranbringt, d.h. die Kante, die die größte Endposition hat. Im allgemeinen brauchen solch ausgefallene Systeme sowohl die Vervollständigung als auch das Scannen.

11.5 Konstruktion eines Parsers

Aus dem Chart-Erkenner kann man auf dieselbe Weise einen Parser machen, wie wir das für den CKY-Algorithmus getan haben. Da wir nicht mehr mit binären Regeln arbeiten, müssen wir die Repräsentation, die für CKY-Kanten gewählt wurde, etwas verallgemeinern. Die Notation der Regeln mit Punkt bietet eine gute Grundlage dafür. Eine Möglichkeit, genug Information zu speichern, um später alle Analysen rekonstruieren zu können, ist, die Positionen im String mit in die Regel mit Punkt aufzunehmen. Die folgenden Zustände könnten zum Beispiel beim Parsen einer VP auftreten.

$$\begin{aligned}
vp &\rightarrow (2)\ .\ v\ np\ np \\
vp &\rightarrow (2)\ v\ (3)\ .\ np\ np \\
vp &\rightarrow (2)\ v\ (3)\ np\ (5)\ .\ np \\
vp &\rightarrow (2)\ v\ (3)\ np\ (5)\ np\ (7)\ .
\end{aligned} \tag{11.10}$$

In diesem Beispiel wurde ein Verb an Position 2, eine NP von Position 3 bis 5 und eine andere NP von 5 bis 7 gefunden. Zusammen mit der Information über

die Teilphrasen enthält die letzte Regel alle Information, die benötigt wird, um die Analyse zu rekonstruieren. Gibt es mehrere Analysen für NPs zwischen 5 und 7 dann gibt es trotzdem nur eine Repräsentation für die VP, die diese NPs benutzt.

Lesetips

In Gazdar und Mellish (1989) kann man eine detailierte Diskussion von Chart-Parsern finden. Dort findet man auch Prolog-Implementationen von Chart-Parsern mit Tagesordnung. Teile des Buches sind über WWW[1] verfügbar.

Kontrollfragen

1. Was ist eine Regel mit Punkt?
2. Welche Regeln mit Punkt entsprechen den Einträgen, die beim CKY-Algorithmus gemacht werden?
3. Was ist der Unterschied zwischen aktiven und inaktiven Kanten?
4. Wird Vorhersage auf aktive oder inaktive Kanten angewendet?
5. Wird Vervollständigung auf aktive oder inaktive Kanten angewendet?
6. Nach was für Kanten wird bei der Vervollständigung gesucht?
7. Wird Scannen auf aktive oder inaktive Kanten angewendet?
8. Nach was für Kanten wird beim Scannen gesucht?

Übungsaufgaben

1. * Erklären Sie, warum das Scannen im Earley Algorithmus nicht benötigt wird.
2. * Es ist nicht möglich, polynomiale Komplexität zu erreichen, wenn man Parsebäume in der Chart abspeichert. Warum ist das so?
3. Wie sollte sich ein Chart-Parser/Erkenner verhalten, wenn die Grammatik Regeln der Form $S \rightarrow S$ enthält?

[1] http://www.de.relator.research.ec.org/resources/gm

4. Unter dem Pfad `~smueller/Public/Prolog/Quellen/Chart` befindet sich ein Chart-Parser und eine Grammatik im entsprechenden Format.

   - Erreicht dieser Chart-Parser polynomiale Komplexität?
   - Die Parsebäume, die der Parser produziert, sind verkehrtherum. Warum ist das so?

5. * Der Gazdar-Mellish-Chart-Parser füllt die Chart von links nach rechts aus. Wieso kann der Teil der dritten Klausel von `add_edge`, der bereits vorhandene Kanten mit einer inaktiven Kante kombiniert, sinnvoll sein? Wie kann man phonologisch leere Kategorien mit minimalem Aufwand in die Chart eintragen?

Hausaufgabe

6. Verändern Sie den Gazdar/Mellish-Chart-Parser so, daß statt der Parsebäume Kategorien und Positionen in die Chart eingetragen werden (siehe Kapitel 11.5). Geben Sie eine Beispielgrammatik an, für die das modifizierte Programm eine kleinere Chart produziert als der Original-Parser. Für diesen Teil der Aufgabe kann man höchstens eine 2 bekommen.

   Eine Eins ist erreichbar, wenn die folgende Teilaufgabe ebenfalls gelöst wird: Erweitern Sie das Programm so, daß es alle möglichen Parsebäume anzeigt.

Kapitel 12

Kategorialgrammatik

12.1 Lexikongesteuerte Grammatiken

Während der Diskussion der Chart-Parser haben wir die Multiplikationsregel benutzt, die beschreibt, wie zwei Konstituenten zusammengefügt werden, um dann eine neue Konstituente zu bilden. Beim CKY-Algorithmus hängt diese Regel in einfacher Weise von der Grammatik ab: Zwei Konstituenten β und γ können kombiniert werden, wenn es in der Grammatik eine Regel $\alpha \rightarrow \beta\gamma$ gibt. Beim Algorithmus für *active chart parsing* bezieht sich die Multiplikationsregel nicht auf einzelne Grammatikregeln. Sie entspricht einem Regelschema. Bei der Vervollständigung werden Phrasen nur aufgrund formaler Eigenschaften zusammengefügt. In diesem Kapitel werden wir einen Grammatikformalismus kennenlernen, der eine solche Notation als Regelschema benutzt. Wir werden zeigen, daß dieser Formalismus einige Vorteile gegenüber anderen hat.

Zur Motivation betrachten wir die Top-Down-Einbeziehung wie beim Earley-Algorithmus. Nehmen wir an, daß Subkategorisierung wie im Kapitel 5 vorgeschlagen beschrieben wird, d.h. ein Merkmal der Lexikoneinträge ist dafür zuständig, welche Phrasenstrukturregel benutzt werden kann. Zum Beispiel:

$$vp \rightarrow v(ditrans)\ np\ np \tag{12.1}$$

Bei einem Top-Down-Parser ohne Vorausschau werden viele VP-Hypothesen mit dem Punkt am Anfang in die Chart eingetragen. Wenn dann im Lexikon nachgesehen wird und die Multiplikationsregel mit dem entsprechenden Verb angewendet wird, sind nur wenige dieser Hypothesen brauchbar. Das Ergebnis sind dann einige Kanten der Form:

$$\begin{array}{l} vp \rightarrow v(ditrans).np\ np \\ vp \rightarrow v(np_and_pp).np\ pp(to) \end{array} \tag{12.2}$$

Die Left-Corner-Strategie hat dieses Problem nicht. Ein Wort wird im Lexikon gesucht, und jede Regel, von der es eine linke Ecke ist, wird in die Chart eingetragen. Das funktioniert für Englisch ganz gut, weil Englisch eine nach rechts verzweigende Sprache ist. In einer nach rechts verzweigenden oder auch *head initial*-Sprache steht das Verb ganz links in der Verbphrase. Deshalb ist die linke Ecke einer Verbphrase das Verb. Es gibt aber auch *head final*-Sprachen, bei denen das Verb ganz rechts steht. Bei diesen Sprachen müßte man dann auch alle VP-Hypothesen in die Chart aufnehmen.

Ein gutes Beispiel für eine Sprache, in der Verben sowohl am Satzanfang als auch am Satzende stehen können, ist Deutsch. Im folgenden sind einige Regeln aufgeführt, die in einer Grammatik des Deutschen vorkommen können.

$$\begin{aligned} &s(fin, final) \rightarrow np\ v(fin, intr) \\ &s(fin, final) \rightarrow np\ np\ v(fin, trans) \\ &s(fin, final) \rightarrow np\ np\ np\ v(fin, ditrans) \end{aligned} \tag{12.3}$$

Diese Regeln sind nicht vollständig. Die letzte Kategorie könnte auch ein beliebiges nichtfinites Verb sein.

Will man keine Hilfsprozeduren verwenden, bekommt man eine kombinatorische Explosion, wenn man die obige Grammatik um Kasusmerkmale erweitert. würden von einem Bottom-Up-Chart-Parser ohne Vorhersage 10 unnütze Hypothesen in die Chart eingetragen, die alle verwaltet werden müssen.

(12.4) , weil Karl Maria das Buch gibt.

Bei jeder erkannten NP würden Hypothesen dafür in die Chart eingetragen, daß die NP den Anfang oder das Mittelstück einer Verbalprojektion bildet.

Eine Lösung dieses Problems ist, das Lexikon bestimmen zu lassen, welche aktiven Kanten in die Chart aufgenommen werden. Wenn man diese Information im Lexikon kodiert, werden die Phrasenstrukturregeln überflüssig.

12.2 Kategorialgrammatik

Ein Lexikon für eine kontextfreie Grammatik, das die Information der Phrasenstrukturregeln im Lexikon enthält, wird *Kategorialgrammatik* genannt (*Categorial Grammar* CG).

Lexikoneinträge der CG werden mit Hilfe des Slash beschrieben. Als Beispiel betrachten wir die beiden obigen aktiven Regeln:

(12.5)

Regel	Kategorie im Lexikon
vp → v(ditrans).np np	(vp/np)/np
vp → v(np_and_pp).np pp(to)	(vp/pp)/np

Diese komplexen Kategorien ersetzen die Subcat-Merkmale wie *ditrans* oder *np_and_pp* von Verben wie *give* und *send*. Die zweite Kategorie kann man sich genau wie die aktive Kante vorstellen. Es handelt sich bei (vp/pp)/np um etwas, das mit einer np zu etwas kombiniert werden kann, das dann mit einer pp zu einer vp kombiniert werden kann. Jetzt wird auch plausibel, warum man die Kombinationsregel der Chart-Parser *Multiplikationsregel* nennt. Benutzt man den Slash der Kategorialgrammatik, erhält man die folgende Regel:

$$Regel1 : X/Y * Y = X \tag{12.6}$$

wobei X und Y Kategorien sind. Wenn X und Y reelle Zahlen wären, dann wäre (12.6) eine wahre Aussage, wenn / der Division und $*$ der Multiplikation von reellen Zahlen entspräche. Man beachte, daß die obige Gleichung rekursiv angewendet werden kann. X und Y können komplexe Kategorien sein. Zur Vereinfachung nimmt man für / meist Linksassoziativität an. Man kann also Klammern weglassen und für (vp/pp)/np einfach vp/pp/np schreiben.

Die Kategorie v wird nicht mehr benötigt. Wir haben aber noch die vp. Man kann sogar ohne die vp auskommen, indem man etwas definiert, was noch eine np braucht – das Subjekt –, um einen Satz zu bilden. Im Gegensatz zu Objekten, die sich im Englischen rechts vom Verb befinden, steht das zu einer englischen Verbphrase gehörende Subjekt links von dieser. Deshalb werden der Backslash und eine zweite Multiplikationsregel eingeführt:

$$Regel2 : Y * X\backslash Y = X \tag{12.7}$$

Wir können die Menge der benötigten Kategorien noch weiter reduzieren, indem wir uns die Kategorien von optionalen Modifikatoren ansehen. Diese werden normalerweise durch Regeln der Form

$$\begin{aligned} vp &\rightarrow vp\ pp \\ noun &\rightarrow noun\ pp \end{aligned} \tag{12.8}$$

beschrieben. Diese Regeln erlauben beliebig viele Modifikatoren – in diesem Falle Präpositionalphrasen – nach einem Nomen bzw. einer Verbphrase. In der Kategorialgrammatik beschreibt man solche Modifikatoren durch Lexikoneinträge der Form $X\backslash X$. Im Falle eines vp-Modifikators wäre das X in $X\backslash X$ die Kategorie $s\backslash np$. Die Kategorie einer ein Verb modifizierenden pp wäre also $(s\backslash np)\backslash(s\backslash np)$. Ein Postmodifier für Nomina hat die Kategorie $n\backslash n$. Genauso beschreibt man den Effekt der Phrasenstrukturregel

$$noun \rightarrow adj\ noun \tag{12.9}$$

Adjektive bekommen die Kategorie n/n. Im Gegensatz zur Phrasenstrukturgrammatik gibt es bei der Kategorialgrammatik keinen expliziten Unterschied zwischen Phrasen und Wörtern. Ein intransitives Verb hat dieselben Eigenschaften

wie die entsprechende Verbphrase. Sowohl die Phrase als auch das Wort haben die Kategorie $(s\backslash np)$. Das gleiche gilt für Eigennamen und Nominalphrasen.

Eine typische Kategorialgrammatik benutzt nur s, np und n als Hauptkategorien. Manchmal werden noch pps als Komplementpräpositionalphrasen zugelassen.

Eine Ableitung in der CG ist im wesentlichen ein binär verzweigender Baum, wird aber meistens wie folgt repräsentiert: Ein Pfeil unter einem Paar von Kategorien zeigt an, daß diese mit einer Kombinationsregel kombiniert werden. Die Richtung des Pfeils gibt die Richtung der Kombination an. Das Ergebnis wird unter den Pfeil geschrieben. Ein Beispiel zeigt Abbildung 12.1. Eine Kategorial-

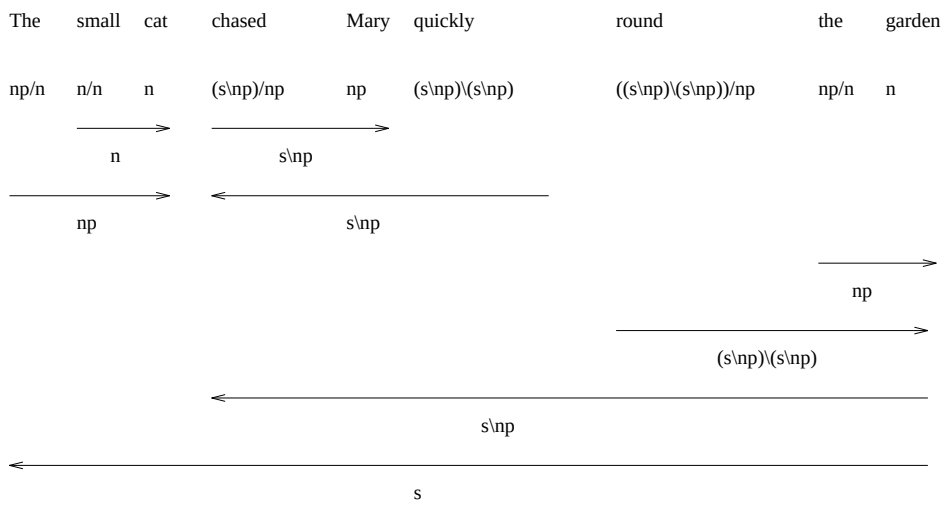

Abbildung 12.1: Eine Ableitung mit einer Kategorialgrammatik

grammatik mit den beiden Multiplikationsregeln, die oben angegeben wurden, ist schwach äquivalent zu einer kontextfreien Grammatik. Solche Grammatiken wurden zuerst von Adjukiewicz diskutiert. Die Äquivalenz zu den CFG wurde von Bar-Hillel bewiesen. Deshalb wird ein solches System auch AB genannt.

Obwohl AB schwach äquivalent zu kontextfreien Grammatiken ist, ermöglicht das System die Beschreibung verschiedener linguistischer Phänomene auf elegante Weise. In der CFG müßte man für entsprechende Beschreibungen Merkmale benutzen.

Zum Beispiel kann man englische Pronomina in AB darstellen, ohne ein Kasus-Merkmal zu verwenden. Man drückt in der Kategorialgrammatik die Verschiedenartigkeit von Nominativ- und Akkusativ-Pronomina (z.B. *he* bzw. *him*) durch unterschiedliche Kombinationsmöglichkeiten aus: *he* hat die Funktion eines Subjekts und muß nach rechts mit einer Verbphrase kombiniert werden. *him* hat die Funktion eines Objekts und muß nach links mit einem Verb kombiniert werden, um eine Verbphrase zu bilden.

```
he:  s/(s\np)
him: (s\np)\((s\np)/np)
```

Damit haben wir Ableitungen wie in Abbildung 12.2. Ungrammatische Sätze wie in (12.10) sind durch die Kategorien von *he* und *him* ausgeschlossen.

(12.10) a. * Him likes Mary.

b. * Mary likes he.

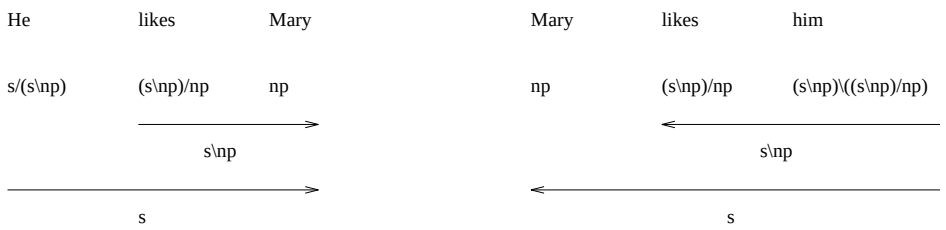

Abbildung 12.2: *he* und *him*

12.3 Unechte Mehrdeutigkeiten I

Die Kategorialgrammatik hat das Problem der unechten Mehrdeutigkeit, d.h. es werden mehrere syntaktische Ableitungen mit derselben Semantik für einen Satz gefunden. Die unechte Mehrdeutigkeit entsteht, weil nichts vorschreibt, in welcher Reihenfolge die Modifikation eines Kopfes erfolgt, wenn mehrere Modifikatoren einen Kopf modifizieren. So gibt es für den folgenden Satz die beiden Ableitungsmöglichkeiten die Abbildung 12.3 zeigt. In der Abbildung ist die Ableitung für den Relativsatz vernachlässigt worden. Relativsätze sind Postmodifier und haben demzufolge die Kategorie $n\backslash n$.

(12.11) The fat man who never moved died.

12.4 Kombinationsgrammatik

Die Kategorialgrammatik hat noch andere Vorteile. Man kann noch weitere Multiplikationsregeln festlegen, die einen sehr flexiblen Kategoriebegriff gestatten. Für die Sätze in (12.12), die die Konjunktion *and* enthalten benutzt man normalerweise Regeln der Form $X \rightarrow X \ and \ X$, wobei X eine beliebige Kategorie sein kann.

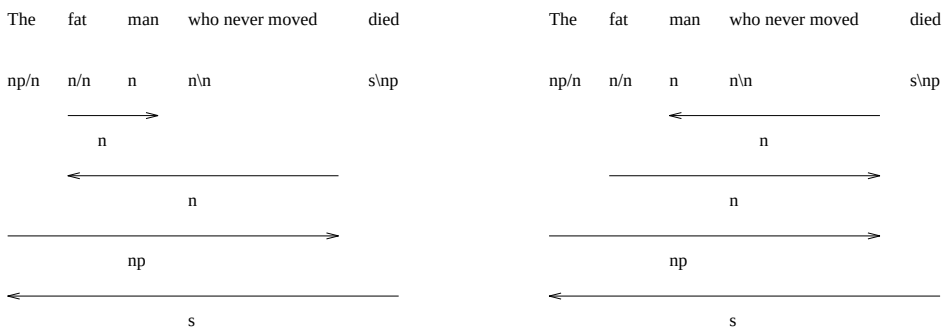

Abbildung 12.3: Modifikatoren und unechte Mehrdeutigkeiten

(12.12) a. Fred cooked, and Mary ate, the delicious spaghetti that they bought in Naples on holiday last week.

b. Bill drinks coffee for breakfast, and Mary tea.

c. I gave Jeanie a t-shirt on her birthday, and my dad a lawnmower for Xmas.

Die obigen Sätze sind problematisch für Grammatiken mit Phrasenstrukturregeln, da *Fred cooked*, *Mary tea* und *my dad a lawnmower for Xmas* keine Konstituenten im eigentlichen Sinne bilden. Auch in der Kategorialgrammatik sind sie nicht ohne weiteres beschreibbar. Man kann diese Phänomene aber beschreiben, indem man bestimmte mathematisch fundierte Operationen über Kategorien definiert.

Die beiden Multiplikationsregeln, die wir bis jetzt haben, werden auch Funktionsanwendungen (*functional application*) genannt, da man X/Y als Funktion von Dingen vom Typ Y in Dinge vom Typ X auffassen kann und $X/Y * Y \rightarrow X$ als Anwendung dieser Funktion auf ein Argument. Die beiden Regeln werden Vorwärtsfunktionsanwendung und Rückwärtsfunktionsanwendung genannt. Eine andere Kombinationsregel entspricht der Komposition ($\circ$) in der Theorie der rekursiven Funktionen:

$$(f \circ g)(a) = f(g(a)) \tag{12.13}$$

Das heißt, die Funktion, die man bekommt, wenn man die Funktion f mit der Funktion g kombiniert, gleicht der Funktion, die der Hintereinanderausführung von f und g entspricht. Hintereinanderausführung bedeutet im konkreten Fall die Anwendung der Funktion f auf das Ergebnis der Funktion g. Die entsprechenden Kombinationsregeln der Kategorialgrammatik haben die Form:

(12.14)	X/Y * Y/Z = X/Z	Vorwärtskomposition (fc)
	Y \ Z * X \ Y = X \ Z	Rückwärtskomposition (bc)

Es können auch Regeln definiert werden, die auf atomare Kategorien anwendbar sind. Eine solche Regel ist das *type raising*. Beim *type raising* wird ein Argument in eine Funktion umgewandelt, die eine bestimmte Sorte Argument verlangt, und zwar genau ein solches Argument, das nach Anwendung der erzeugten Funktion wieder die atomare Ausgangskategorie liefert.

Die Kategorie np kann durch *type raising* in die Kategorie $(s/(s\backslash np))$ umgewandelt werden. Kombiniert man diese Kategorie mit $(s\backslash np)$ erhält man dasselbe Ergebnis wie bei einer Kombination von np und $(s\backslash np)$ mit Regel 1.

$$\begin{aligned} &np * s\backslash np \rightarrow s \\ &s/(s\backslash np) * s\backslash np \rightarrow s \end{aligned} \tag{12.15}$$

Die allgemeinen Regeln für *type raising* sind:

(12.16) X = Y / (Y \ X) vorwärts *type raising* (ftr)
X = Y \ (Y/X) rückwärts *type raising* (btr)

Die Regeln in (12.14) entsprechen wahren Aussagen über echte Multiplikation und Division. Die Richtung der Anwendung ist aber wichtig. Ohne die Berücksichtigung der Richtung wäre die Beschreibung der Personalpronomina nicht korrekt. Die Kategorien der Personalpronomina sind durch *type raising* entstanden. Ein Gegenstück zum *type raising* – eine Reduktion also – läßt man nicht zu, da dann die Information in den Kategorien der Personalpronomina wieder zerstört werden würde.

Der Unterschied der linguistischen Multiplikation zur arithmetischen ist, daß die Multiplikation in der Linguistik nicht kommutativ ist:

$$X * Y \neq Y * X \tag{12.17}$$

Eine Grammatik, die Kompositionsregeln und Regeln für *type raising* enthält, wird Kombinationskategorialgrammatik (*Combinatory Categorial Grammar*) genannt, weil diese Regeln bestimmten Kombinationsarten der Kombinationslogik entsprechen. Eine andere Bezeichnung ist freie Kategorialgrammatik.

Mit den oben angegebenen algebraischen Manipulationen von Kategorien können wir die Sätze in (12.12) beschreiben. Die Abbildung 12.4 zeigt zum Beispiel den Satz (12.12a). Die Linien bzw. Pfeile sind mit den Regelnamen beschriftet.

12.5 Andere Vorteile der Kategorialgrammatik

Ein anderer Vorteil der Kategorialgrammtik ist, daß die Kombinationsregeln nicht sprachspezifisch sind. Es gibt nur eine kleine Menge von Kombinationsregeln; alle andere Information über einen Satz wird aus dem Lexikon abgeleitet. Man kann zum Beispiel ein automatisches Übersetzungssystem schreiben, das bestimmte

Fred	cooked	and	Mary	ate	spaghetti
np	(s\np)/np	(X\X)/X	np	(s\np)/np	np
—— (ftr)			—— (ftr)		
s/(s\np)			s/(s\np)		
——> (fc)			——> (fc)		
	s/np			s/np	
		——> (fa)			
		((s/np)\(s/np))			
<—— (ba)					
		s/np			
——> (fa)					
		s			

Abbildung 12.4: *Fred cooked and Mary ate spaghetti*

Wörter oder Phrasen übersetzt und diese dann mit denselben Regeln zu Sätzen zusammenfügt.

Psychologisch attraktive Parse-Theorien müssen den Umstand erklären, daß das menschliche Parsen inkrementell erfolgt: Eine Interpretation eines Fragments des Gelesenen oder Gehörten wird sequentiell aufgebaut. Die Freiheit, mit der in der CG Konstituenten gebildet werden können, liefert eine Erklärung für diese menschlichen Fähigkeiten.

Außerdem können die syntaktischen Kombinatoren mit semantischen verbunden werden. Dadurch erhält man eine einfache Beschreibung dafür, wie man die Bedeutung einer Phrase aus der Bedeutung ihrer Einzelteile konstruieren kann. Die Zusammensetzung der Bedeutung einer Phrase aus den Bedeutungen ihrer Teile wird *Kompositionalität* genannt. Zum Beispiel kann man die syntaktische Kategorie (X/Y) mit einer Funktion für deren Semantik verbinden. Diese Funktion hat die Form $\lambda y.x$. Hat Y die Semantik y, und wird Y mit X kombiniert, so erhält man als Semantik von X die Kombination von $\lambda y.x$ und y, also x. Ein intransitives Verb $(s\backslash np)$ hat als Semantik etwas, das noch die Bedeutung einer Nominalphrase benötigt und dann die Bedeutung des gesamten Satzes zurückgibt: $\lambda y.sleeps(y) * john = sleeps(john)$. So können semantische Werte parallel zur syntaktischen Ableitung erzeugt werden. Die folgende Tabelle zeigt den Aufbau der Semantik für die verschiedenen Kombinationsregeln:

Bezeichnung	Syntax	Semantik
Anwendung:		
Vorwärts	$X/Y * Y \Rightarrow X$	$(\lambda y.x) * z \Rightarrow x[z/y]$
Rückwärts	$Y * X\backslash Y \Rightarrow X$	$z * (\lambda y.x) \Rightarrow x[z/y]$
Komposition:		
Vorwärts	$X/Y * Y/Z \Rightarrow X/Z$	$\phi_1 * \phi_2 \Rightarrow \phi_1 \circ \phi_2$
Rückwärts	$Y\backslash Z * X\backslash Y \Rightarrow X\backslash Z$	$\phi_1 * \phi_2 \Rightarrow \phi_2 \circ \phi_1$
Type Raising:		
Vorwärts	$X \Rightarrow Y/(Y\backslash X)$	$x \Rightarrow (\lambda y.x(y))$
Rückwärts	$X \Rightarrow Y\backslash(Y/X)$	$x \Rightarrow (\lambda y.x(y))$

12.6 Unechte Mehrdeutigkeiten II

Die Erweiterung der Kategorialgrammatik zur freien Kategorialgrammatik hat viele interessante linguistische Konsequenzen. Sie hat auch aus dem Blickwinkel der Informatik interessante Konsequenzen, die jedoch eher unerwünscht sind: Ein einfacher Satz – auch wenn keine Modifikatoren in ihm vorkommen – kann mehrere semantisch äquivalente Ableitungen haben, wie Abbildung 12.5 zeigt. Es wird behauptet, daß die Mehrdeutigkeit nur für geschriebene Sprache unecht ist.

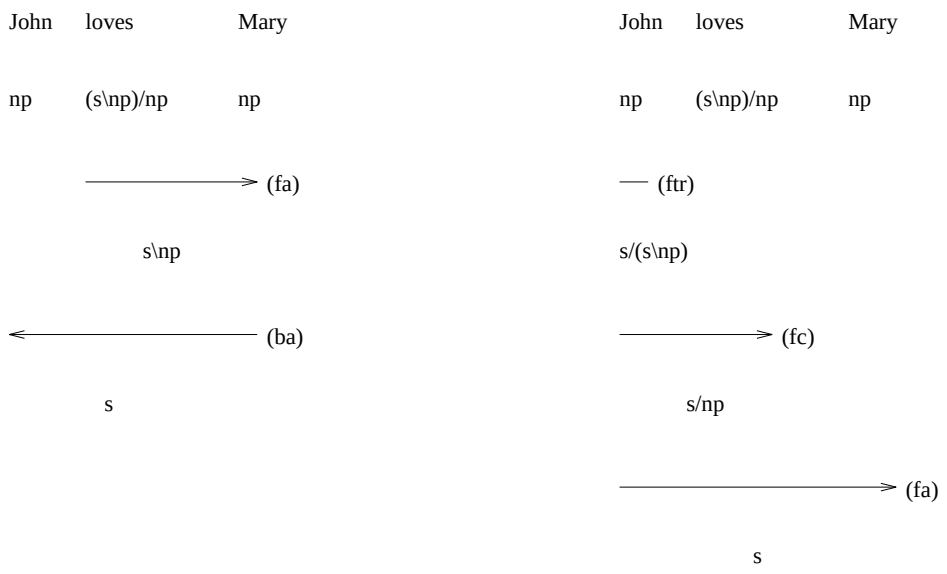

Abbildung 12.5: unechte Mehrdeutigkeit

Bei gesprochener Sprache soll jede Ableitung einer bestimmten Art der Intonation entsprechen, d.h. die Mehrdeutigkeit wäre nicht nur nicht unecht, sie wäre notwendig, um gesprochene Sprache zu parsen Steedman (1991).

Kontrollfragen

1. Mit welcher Kategorie kann X/Y in der Kategorialgrammatik kombiniert werden? Steht das Zeichen links oder rechts von X/Y?
2. Mit welcher Kategorie kann $X\backslash Y$ in der Kategorialgrammatik kombiniert werden? Steht das Zeichen links oder rechts von $X\backslash Y$?
3. Welche Kategorie hat ein transitives Verb normalerweise?
4. Geben Sie ein Beispiel für eine Konjunktion, die die Einführung der *type raising*- und Kompositionsregeln rechtfertigt.

Übungsaufgaben

1. * Schreiben Sie eine einfache AB-Grammatik mit den Kategorien s, np und n. Merkmale wie *vform* oder *num* müssen nicht berücksichtigt werden. Schreiben Sie die Kategorien der folgenden linguistischen Einheiten auf:

- Nomina im Singular
- Nomina im Plural (Man beachte, daß Plural-Nomina ohne Artikel vorkommen können. Man kann also annehmen, daß sie lexikalisch mehrdeutig sind.)
- Adjektive
- Determinatoren *a red house* und *a house* sind NPs, nicht aber *red house* oder *a a house*.
- intransitive Verben
- transitive Verben
- Hilfsverben Ein Hilfsverb ergibt nach Kombination mit einer VP eine VP.
- Präpositionen Eine Präposition wird mit einer NP kombiniert. Das Ergebnis kann mit einer NP nach links kombiniert werden, wobei wieder eine NP entsteht. Oder es wird mit einer VP links kombiniert, wobei wieder eine VP entsteht.

2. * Leiten Sie mit diesen Kategorien die folgenden Sätze ab:

   People slept.
   Every small red house sleeps.
   A person with a house disintegrated.
   People will buy houses under every pretence.

3. * Wie gut kann man ein Verb beschreiben, das eine PP verlangt? (*John argued with Mary.*)

4. * Wenn *und* die Kategorie $(X\backslash X)/X$ hat, wie kann man dann die folgenden Sätze mit Hilfe von *type raising* und Kombinationsregeln ableiten:

   A small and red house disintegrated.
   Every man with and every woman without a house sleeps.

5. * AB-Grammatiken sind einfach als DCGs zu kodieren. Es gibt nur zwei Regeln und viele Lexikoneinträge.

```
:- op(30,yfx,'/').
:- op(30,yfx,'\').

cat(X) --> cat(X/Y),cat(Y).
cat(X) --> cat(Y), cat(X\Y).
```

```
cat(np) --> [john].
cat(np) --> [mary].
cat((s\np)/np) --> [saw].

...

:- cat(s,[john,saw,mary],[]).
```

Welche Art von Erkenner ist für diese Grammatik geeignet?

Kapitel 13

Einführung in die Unifikation

13.1 Merkmale und Unifikation

In den Grammatikfragmenten, die wir bis jetzt behandelt haben, wurden oft Merkmale benutzt, um bestimmte Zusammenhänge darzustellen. Im folgenden wollen wir die Verwendung von Merkmalen etwas genauer betrachten.

Wir haben schon definiert, was das Numerus-Merkmal in einer DCG bedeutet (vergleiche S. 23). Von einer Regel der Form

```
s(Vform) --> n(2,Num), v(2,Vform,Num)
```

erwarten wir, daß sie dann und nur dann benutzt wird, wenn derselbe Wert an die Variable `Num` gebunden ist. Es ist egal, wo der Wert für `Num` instantiiert wird. Das kann in der np oder in der vp geschehen, wichtig ist nur, daß es derselbe Wert ist. Man kann DCG-Regeln als Abkürzung für die Menge der Regeln auffassen, die entstehen, wenn man alle Variablen durch ground-Werte ersetzt. Wenn eine Variable in einer Regel mehrfach auftaucht, steht in den entsprechenden Regeln mit ground-Instanzen an den Stellen, die den Variablen entsprechen, natürlich dieselbe ground-Instanz. Die oben angegebene Regel ist also eine Abkürzung für:

```
s(fin) --> n(2,sing), v(2,fin,sing).   % Num = sing, Vform = fin
s(fin) --> n(2,plur), v(2,fin,plur).   % Num = plur, Vform = fin
s(bse) --> n(2,sing), v(2,bse,sing).   % Num = sing, Vform = bse
s(bse) --> n(2,plur), v(2,bse,plur).   % Num = plur, Vform = bse
s(inf) --> n(2,sing), v(2,inf,sing).   % Num = sing, Vform = inf
s(inf) --> n(2,plur), v(2,inf,plur).   % Num = plur, Vform = inf
s(prp) --> n(2,sing), v(2,prp,sing).   % Num = sing, Vform = prp

...
```

Hier ist nur eine endliche Anzahl von ground-Regeln aufgeführt, aber im allgemeinen gibt es unendlich viele. Das erklärt, was eine Regel mit Variablen bedeutet. Wir müssen sicherstellen, daß unsere Operationen über den Regeln das

auch respektieren. Wenn wir ein Parse-Ziel haben, das Variablen enthält, ist das nur eine Abkürzung für die Menge der ground-Ziele, die durch eine Substitution der Variablen durch alle möglichen ground-Instanzen entstünden. Wenn wir das Ziel

```
?- s(Vform).
```

beweisen wollen,[1] haben wir eigentlich eine Menge simultaner Ziele:

```
?-
   {
      s(fin),     % Vform = fin
      s(bse),     % Vform = bse
      s(inf),     % Vform = inf
      s(prp),     % Vform = prp
      ...
    }
```

Wir verlangen dann von Prolog, daß es uns sagt, welche Ziele erfolgreich bewiesen werden können.

Wie muß nun die angegebene Regelmenge genutzt werden, um diese Ziele zu beweisen? Wenn wir nur ground-Regeln betrachten, ist natürlich klar, daß zum Beweis eines Ziels genau die Regeln benutzt werden müssen, bei denen die linke Seite identisch mit dem zu beweisenden Ziel ist. Wollen wir zum Beispiel `s(fin)` beweisen, so können wir die ersten beiden Regeln benutzen und haben also die folgenden Teilziele zu beweisen:

```
?-
   {
     n(2,sing), v(2,fin,sing)   % aus der ersten Regel
     n(2,plur), v(2,fin,plur)   % aus der zweiten Regel
   }
```

Diese Menge von Teilzielen ist äquivalent zu dem abstrakteren Teilziel

```
?- n(2,Num), v(2,fin,Num)
```

Allgemein gilt: Wenn wir eine Menge von ground-Zielen

$$\{Goal_1, Goal_2, \ldots Goal_n\} \tag{13.1}$$

[1] Die Differenzlistenargumente, die zu der Prolog-Anfrage gehören, sind der Einfachheit halber weggelassen worden.

und eine Menge von ground-Regeln

$$\{LHS_1 \rightarrow RHS_1., LHS_2 \rightarrow RHS_2., \ldots LHS_n \rightarrow RHS_n.\} \tag{13.2}$$

haben, dann wollen wir das in eine Menge von Anfragen der Form

$$?-\{\ RHS_i \mid \textit{für ein } j \textit{ gilt } LHS_i\ =\ Goal_j\ \} \tag{13.3}$$

umwandeln. Beim Parsen wollen wir natürlich nicht große Mengen von Regeln explizit bearbeiten und benutzen deshalb Variablen an Stellen, wo noch keine instantiierten Werte vorliegen. Wir müssen dabei sicherstellen, daß die Operationen, die wir mit Regeln durchführen, die Semantik der Ziele und Regeln als Mengen von ground-Instanzen korrekt wiedergeben. Wenn wir also ein Ziel *Goal* und eine Regel *Rule* mit der linken Seite *LHS* (beide enthalten eventuell Variablen) haben, dann ist die Menge aller Instanzen von *Rule*, die zum Beweis von *Goal* anwendbar sind, durch $Rule'$ repräsentiert, falls gilt:

$$\begin{aligned} & gi(Rule') = \\ & \{r \mid \\ & \qquad r \textit{ ist eine ground} - \textit{Instanz von Rule und} \\ & \qquad LHS(r)\ \in\ gi(Goal)\ \cap\ gi(LHS) \\ & \} \end{aligned} \tag{13.4}$$

Dabei ist $gi(X)$ die Menge der ground-Instanzen von X. $Rule'$ ist eine teilweise instantiierte Version von *Rule*, da $Rule'$ weniger oder genauso viele ground-Instanzen hat. Wir müssen eine teilweise Instantiierung von *Rule* finden, also $Rule'$, so daß die LHS der neuen Regel dieselben ground-Instanzen hat wie der Durchschnitt von *Goal* und LHS:

$$gi(LHS(Rule')) = gi(Goal) \cap gi(LHS) \tag{13.5}$$

Für das obige Beispiel sieht das wie folgt aus:

```
Goal = s(fin)
Rule = s(Vform) --> n(2,Num), v(2,Vform,Num)
LHS = s(Vform)
gi(Goal) = {s(fin)}
gi(LHS)  = {s(fin), s(bse), s(pas), ... }
```

und der Wert für $Rule'$ ist:

```
Rule' = s(fin) --> n(2,Num), v(2,fin,Num).
```

Aus dieser instantiierten Regel nehmen wir dann die nächsten Ziele, die bewiesen werden müssen. Diese können auch wieder Mengen von ground-Instanzen repräsentieren.

Wenn wir eine DCG-Regel top-down anwenden, können wir die Semantik der Regel korrekt modellieren, indem wir die Regel weiter instantiieren, und zwar so, daß die neue LHS genau die ground-Instanzen hat, die das Ziel hat und die, die schon in der Regel selbst instantiiert waren. Die RHS der neuen instantiierten Regel enthält dann die korrekten Teilziele. Ähnlich funktioniert die Instantiierung von Variablen beim Bottom-Up-Parsen. Die Operation, die bestimmt, wie Variablen in einer Regel zum Beweis eines bestimmten Ziels instantiiert werden, wird *Unifikation* genannt.

13.2 Term-Unifikation

Ein Term kann ein Atom, eine Variable oder ein zusammengesetzter Term sein. Ein zusammengesetzter Term besteht aus einem Funktor und einer festen Anzahl von Argumenten, die selbst Terme sind. Bei der Unifikation werden zwei Terme genommen, und es wird bestimmt, wie sie instantiiert werden müssen, so daß beide den gleichen Term darstellen. Wenn wir beide Terme sowenig wie möglich und so viel wie nötig instantiieren, erhalten wir als Resultat eine Term, dessen ground-Instanzen mögliche ground-Instanzen beider Ausgangsterme sind. Die Art und Weise, wie diese Instantiierung durchgeführt wird, ist dieselbe Art Instantiierung wie die, die man bei der Anwendung einer Regel zum Beweis eines bestimmten Ziels vornimmt.

Genauer: Zwei Terme unifizieren, wenn es eine Wertzuweisung für Variablen – auch Substitution genannt – gibt, deren Anwendung auf die Terme diese identisch macht. Eine Substitution kann man als Funktion verstehen, die auf einen Term angewendet wird und einen Term liefert, in dem alle Variablen durch Werte ersetzt wurden, die in der Substitution angegeben sind. Die beiden Terme

```
p(X,f(Y,b),c)      p(g(Z,c),f(a,W),V)
```

können durch die folgende Substitution identisch gemacht werden:

```
{ g(Z,c)/X, a/Y, b/W, c/V }
```

Da wir keine möglichen Lösungen ausschließen wollen, sind wir am allgemeinsten Unifikat (*most general unifier* MGU) von zwei Termen interessiert. Vereinfacht ausgedrückt heißt das, das allgemeinste Unifikat ist die Substitution, die kein Paar enthält, das nicht wirklich nötig ist, um die beiden Terme zu unifizieren. Der MGU eines Zieles und der linken Seite einer Regel bestimmt, wie die Regel instantiiert werden muß, um das Ziel zu beweisen.

`unify` ist eine Funktion, die zwei Terme in Listennotation nimmt, z.B.

```
p(a,f(b,c)) =.. [p,a,[f,b,c]]
```

und eine Substitution zurückgibt, wenn es eine gibt, oder fehlschlägt, falls eine solche Substitution nicht exisitiert. Mit der Prolog-Schreibweise für Variablen und Konstanten liefert die Funktion `unify` folgendes Ergebnis:

```
unify(a(X,b(c,Y),d(X)), a(e,b(Z,W),W)) = e/X, c/Z, Y/W, d(e)/Y
```

Es folgt ein Algorithmus für die Termunifikation:

```
function unify(T1,T2 : terms) -> substitution;

if either T1 or T2 is atomic( functor, constant or variable) then
   if T1 = T2
   then return the empty substitution
   else
       if T1 is a variable then
          if T1 occurs in T2 then return fail  % occurs check
          else return {T2/T1}
       else
          if T2 is a variable then
             if T2 occurs in T1 then return fail  % occurs check
             else return {T1/T2}
          else return fail                   % different atoms
else
   S1 <- unify(first(T1), first(T2) )        % unify heads
   if S1 = fail then return fail
   else
      G1 <- applay(S1,rest(T1))              % substitute in tails
      G2 <- applay(S1,rest(T2))
      S2 <- unify(G1,G2)                     % unify tails
      if S2 = fail then return fail
      else return compose(S1,S2)
```

Das, was dieser Algorithmus leistet, kann wie folgt einfach zusammengefaßt werden:

- Eine Variable unifiziert mit jedem Term, in dem sie nicht vorkommt.
- Ein Atom (Funktor oder Konstante) unifiziert nur mit einem identischen Atom.
- Zusammengesetze Terme unifizieren, wenn ihre Funktoren identisch sind und deren Argumente paarweise unifizierbar sind. Die Substitutionen, die bei diesen Unifikationen entstehen, müssen kompatibel sein.

Der Prolog-Interpreter arbeitet ähnlich. Aus Effektivitätsgründen werden die Enthaltenseins-Checks aber weggelassen. Das Enthaltensein einer Variable in einem Term, durch den sie ersetzt wird, wird nicht überprüft. Um zu sehen, wohin das führt, gebe man `X = f(X)` auf dem Prolog-Top-Level ein.

Außerdem berechnet ein Prolog-Interpreter nicht nur die Substitutionen, er wendet sie auch auf die entsprechenden Terme an, wobei diese verändert werden. Man nennt das auch zerstörende Unifikation.

13.3 Graph-Unifikation

Beim Schreiben von DCGs haben wir viele Merkmale benutzt:

```
number: {sing,plur}
subcat: {intrans,trans,ditrans,...}
vform:  {fin,bse,inf, ...}
gap:    {gap,nogap}
```

Wenn wir versuchen, die Menge der durch unsere Grammatik beschriebenen Sätze zu erweitern, werden wir bald an die Grenzen der Term-Unifikation stoßen. In einem Term ist die Anzahl der Argumente, die ein Funktor haben kann, fest. Die Argumente sind lediglich durch ihre Position im Term unterscheidbar. Daraus ergeben sich die folgenden Probleme. Man muß sich stets die Position der Merkmale merken. Wenn wir ein neues Merkmal hinzufügen wollen, so muß das an jeder Stelle in der Grammatik geschehen, an der das Merkmal relevant ist. Wenn wir auf einen Wert eines Merkmals in einem Term zugreifen wollen, müssen wir alle anderen Elemente im Term mitspezifizieren, eventuell als anonyme Variablen. Im folgenden wird das am Beispiel von Agreementmerkmalen gezeigt. Zum Anfang haben wir nur Numerus-Agreement, so daß der Term die Form

```
agr(Num)
```

hätte. Wenn man das dann um Person-Agreement erweitern wollte, müßte man überall in der Grammatik den Term in

```
agr(Num,Per)
```

umwandeln. Wenn wir Agreement-Merkmale im Lexikon spezifizieren, müssen wir darauf achten, daß diese immer in derselben Reihenfolge angegeben werden, da die Terme

```
agr(sing,3)          agr(3,sing)
```

nicht äquivalent sind. Wenn man einen Lexikoneintrag hat, bei dem man nur das Person- nicht aber das Numerus-Merkmal instantiieren will, wie zum Beispiel bei *fish*, dann muß man

```
agr(_,3)
```

schreiben. Da der Term `agr(3)` nicht mit den entsprechenden `agr`-Termen unifizieren würde. Diese Probleme werden alle auf einmal gelöst, indem man eine andere Repräsentation für Merkmale wählt. In dieser Repräsentation wird der Name des Merkmals explizit gemacht. Man kann sich dann mit Hilfe des Merkmalnamens auf das Merkmal beziehen, und die Position ist nicht länger wichtig. Eine Merkmalstruktur, die dieselben Merkmale wie der Term `agr(sing,3)` enthält, kann man wie folgt darstellen.

$$\begin{bmatrix} \text{NUM} & sg \\ \text{PER} & 3 \end{bmatrix} \tag{13.6}$$

Diese Struktur ist äquivalent zur Struktur

$$\begin{bmatrix} \text{PER} & 3 \\ \text{NUM} & sg \end{bmatrix} \tag{13.7}$$

Wenn die Struktur nur den Wert für das PER-Merkmal enthält, kann man

$$\begin{bmatrix} \text{PER} & 3 \end{bmatrix} \tag{13.8}$$

schreiben, ohne die Kompatibilität mit den voll spezifizierten Strukturen zu verlieren.

13.4 Der Formalismus der Merkmalstrukturen

13.4.1 Definition der Merkmalstrukturen

Für das Konzept *Merkmalstruktur* werden u.a. folgende Bezeichnungen gebraucht:

- Merkmal-Wert-Struktur
- Attribut-Wert-Struktur
- feature structure
- attribute-value matrix
- feature matrix

- functional description

Def. 5 (Merkmalstruktur – vorläufige Version) *Eine Merkmalstruktur ist eine Menge von Paaren der Form [attribut wert]. Dabei ist* attribut *ein Element einer Menge ATTR von Merkmalen. Die Komponente* wert *ist*

- *entweder atomar (Zeichenkette) (die Merkmalstruktur ist elementar)*
- *oder selbst auch Merkmalstruktur (die Merkmalstruktur ist komplex).*

Beispiele für Merkmalstrukturen sind:

$$\begin{bmatrix} \text{A1} & \textit{W1} \\ \text{A2} & \textit{W2} \\ \text{A3} & \textit{W3} \end{bmatrix} \tag{13.9}$$

Das Merkmal A1 hat den Wert W1, A2 hat den Wert W2 und das Merkmal A3 hat den Wert W3.

$$\begin{bmatrix} \text{A1} & \textit{W1} \\ \text{A2} & \begin{bmatrix} \text{A21} & \textit{W21} \\ \text{A22} & \begin{bmatrix} \text{A221} & \textit{W221} \\ \text{A222} & \textit{W222} \end{bmatrix} \end{bmatrix} \\ \text{A3} & \textit{W3} \end{bmatrix} \tag{13.10}$$

Das Merkmal A2 hat als Wert eine Merkmalstruktur mit den Merkmalen A21 und A22 , A21 hat den Wert W21 , A22 hat eine Merkmalstruktur mit den Merkmalen A221 und A222 als Wert, usw.

$$\begin{bmatrix} \ \end{bmatrix} \tag{13.11}$$

Die Struktur in (13.11) ist die leere Merkmalstruktur.

Eine Merkmalstruktur kann z.B. die Repräsentation einer Bedingung, die für deutsche Sätze gilt, sein. Will man die deutsche Sprache mit Merkmalstrukturen beschreiben, so müssen neben den Beschreibungen für Wortformen Bedingungen für Phrasen mit eingebracht werden. Sie haben aber das gleiche Format wie die Wortform-Beschreibungen. Lexikoneintrag und grammatische Regel sind formal ununterscheidbar. Merkmalstrukturen sind als Beschreibungen sprachlicher Objekte verwendbar. Die Objekte können eine Wortform, aber auch ein Morphem oder ein Satz sein. Das entscheidende ist die Homogenität der Repräsentation, die es erlaubt, ohne Änderung der Repräsentationsart die Verknüpfung von Morphemen zu Wörtern, zu Phrasen und zu Sätzen zu beschreiben. Dabei ist vorausgesetzt, daß die entsprechenden inhaltlichen Verknüpfungen formal mit diesen

Strukturen vollzogen werden können. Zusätzlich werden auch Regeln in der gleichen Weise behandelt.

Sowohl Lexikoneinträge als auch Regeln erscheinen letztlich als Bedingungen, d.h. es gilt eine Beziehung Information = Bedingung.

Das Universum aller Möglichkeiten wird eingeschränkt. Je mehr man weiß, umso weniger ist möglich. In diesem Sinne sind die Merkmalstrukturen und der Umgang mit ihnen eine monotone Angelegenheit.

Def. 6 *Ein* **Pfad** *in einer Merkmalstruktur ist eine Folge von Merkmalen, die in der Merkmalstruktur unmittelbar aufeinander folgen. Der* **Wert eines Pfades** *ist die Merkmalstruktur, die am Ende des Pfades beginnt.*

Der Wert von A2|A22|A221 in (13.10) ist W221 und der von A2|A22

$$\begin{bmatrix} \text{A221} & \textit{W221} \\ \text{A222} & \textit{W222} \end{bmatrix} \tag{13.12}$$

Def. 7 *Ein Pfad in einer Merkmalstruktur heißt* **vollständig**, *wenn er am Wurzelknoten beginnt und einen atomaren Wert hat.*

13.4.2 Darstellung als Graph

Man kann Merkmalstrukturen auch als Graphen darstellen: Merkmale werden zu Kanten, Werte vollständiger Pfade zu Endknoten. Knoten zwischen Kanten sowie Wurzelknoten haben keinen Namen.

Das Ergebnis ist ein gerichteter azyklischer Graph (*directed acyclic graph* DAG) mit einem Wurzelknoten. Die Abbildung 13.1 zeigt die Graphendarstellung der folgenden Merkmalstruktur:

$$\begin{bmatrix} \text{AGR} & \begin{bmatrix} \text{PER} & \textit{3} \\ \text{NUM} & \textit{sg} \end{bmatrix} \end{bmatrix} \tag{13.13}$$

13.4.3 *structure sharing* als wichtiges Beschreibungsmittel für linguistische Zwecke

Def. 8 *Zwei Merkmale innerhalb einer Merkmalstruktur* **teilen sich eine gemeinsame Struktur**, *wenn beide denselben Wert haben. Diese Gleichheit bleibt auch bei Veränderungen an der Merkmalstruktur bestehen. Der gemeinsame Wert wird nur einmal in die Merkmalstruktur eingetragen, die Übereinstimmung wird durch einen Index festgelegt.*

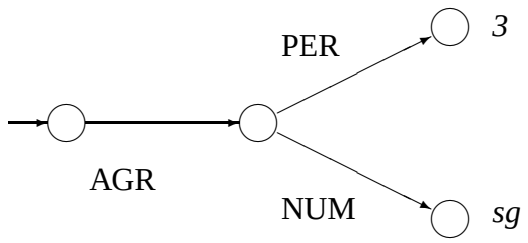

Abbildung 13.1: Merkmalstrukturen lassen sich auch als Graph darstellen

Statt *structure sharing* verwendet man auch die Begriffe Koreferenz bzw. *reentrancy*, wenn von Graphen die Rede ist. Beispiel:

$$\begin{bmatrix} \text{A1} & \boxed{1} \begin{bmatrix} \text{A3} & W3 \end{bmatrix} \\ \text{A2} & \boxed{1} \end{bmatrix} \tag{13.14}$$

A1 und A2 werden auch als token-identisch bezeichnet.

Im Gegensatz dazu steht die folgende Struktur:

$$\begin{bmatrix} \text{A1} & \begin{bmatrix} \text{A3} & W3 \end{bmatrix} \\ \text{A2} & \begin{bmatrix} \text{A3} & W3 \end{bmatrix} \end{bmatrix} \tag{13.15}$$

Als isolierte Strukturen drücken beide zunächst dasselbe aus. Der Unterschied kommt dann zum Tragen, wenn eine Verknüpfung stattfindet. Verbindet man jeweils beide mit einer Struktur, in der A1 : ... vorkommt, so wird A2 in der zweiten Struktur nicht betroffen.

Die folgende Merkmalstruktur zeigt, wie man Subjekt-Verb-Agreement mit *structure sharing* beschreiben kann.

$$\begin{bmatrix} \text{SUBJ} & \begin{bmatrix} \text{PHON} & \textit{hans} \\ \text{AGR} & \boxed{1} \begin{bmatrix} \text{NUM} & \textit{sg} \\ \text{PER} & 3 \end{bmatrix} \end{bmatrix} \\ \text{PRED} & \begin{bmatrix} \text{PHON} & \textit{schläft} \\ \text{AGR} & \boxed{1} \end{bmatrix} \end{bmatrix} \tag{13.16}$$

Abbildung 13.2 zeigt die Struktur (13.16) in Graphendarstellung.

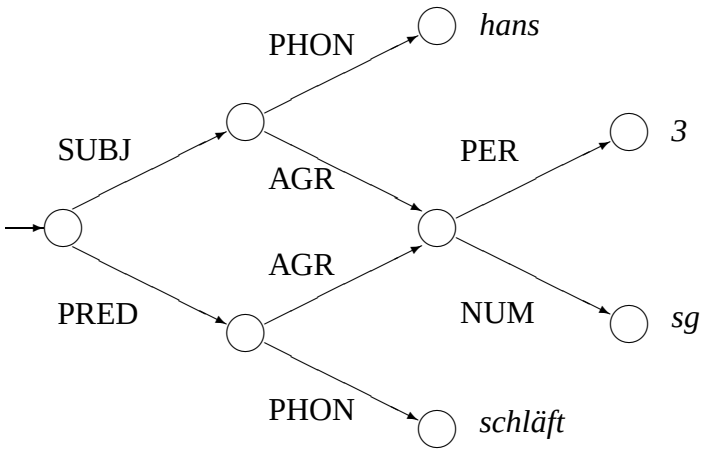

Abbildung 13.2: Die Struktur in Graphendarstellung

Mit Hilfe von *structure sharing* kann man zyklische Strukturen beschreiben.

$$\begin{bmatrix} \text{A1} & \boxed{1} & \begin{bmatrix} \text{A3} & \boxed{2} \end{bmatrix} \\ \text{A2} & \boxed{2} & \begin{bmatrix} \text{A3} & \boxed{1} \end{bmatrix} \end{bmatrix} \tag{13.17}$$

Zyklische Strukturen sind schwer handhabbar, weil Operationen und Relationen über zyklischen Strukturen schwerer zu implementieren sind, und werden auch nicht benötigt. In der endgültigen Version der Definition von Merkmalstrukturen werden Zyklen deshalb ausgeschlossen.[2]

Def. 9 (Merkmalstruktur – endgültige Version) *Eine Merkmalstruktur ist eine Menge von Paaren der Form [attribut wert]. Dabei ist* attribut *ein Element einer Menge ATTR von Merkmalen. Die Komponente* wert *ist*

- *entweder atomar (Zeichenkette) (die Merkmalstruktur ist elementar)*
- *oder selbst auch Merkmalstruktur (die Merkmalstruktur ist komplex),*

und es tritt kein Zyklus auf.

[2] In Johnson (1988) wird eine allgemeine theorieunabhängige Definition für Merkmalstrukturen gegeben, die Zyklen zuläßt. In manchen Theorien sind Zyklen erlaubt und werden verwendet.

13.4.4 Subsumption als Halbordnungsrelation auf der Menge der Merkmalstrukturen

Def. 10 *Eine Merkmalstruktur M1* **subsumiert** *eine Merkmalstruktur M2 (M1 $\succeq$ M2), wenn für jeden vollständigen Pfad P folgendes gilt:*

- *Jeder vollständige Pfad von M1 ist in M2 enthalten und hat dort denselben Wert wie in M1.*
- *Jedes Paar von Pfaden in M1, das denselben Wert hat (structure sharing), muß auch in M2 denselben Wert haben.*

Eigenschaften der Subsumption

- Wenn M1 $\succeq$ M2, dann enthält M1 weniger Informationen als M2, d.h. M2 enthält wenigstens so viele Informationen wie M1. Je weniger Information eine Merkmalstruktur enthält, desto mehr linguistische Objekte werden durch sie beschrieben.
- Jede Merkmalstruktur ist eine Teilspezifikation der Merkmalstrukturen, die sie subsumiert.
- $\succeq$ bildet eine Halbordnung in der Menge der Merkmalstrukturen, denn
  - $\succeq$ ist reflexiv, d.h. für jede Merkmalstruktur M gilt: M $\succeq$ M.
  - $\succeq$ ist transitiv, d.h. wenn gilt: M1 $\succeq$ M2 und M2 $\succeq$ M3, dann gilt auch: M1 $\succeq$ M3.
  - $\succeq$ ist antisymmetrisch, d.h. wenn gilt: M1 $\succeq$ M2 und M2 $\succeq$ M1, dann ist M1 = M2.
- Es gilt: [ ] $\succeq$ M für jede Merkmalstruktur M, d.h. [ ] ist das maximale Element in der Halbordnung.
- Für eine atomare Merkmalstruktur W1 gilt: Wenn W1 $\succeq$ W2, dann ist W1 = W2.

Für die folgenden Beispiele gelten die Relationen:

M1 $\succeq$ M2 $\succeq$ M7 $\succeq$ M8 $\succeq$ M9

M1 $\succeq$ M4 $\succeq$ M6 $\succeq$ M7 $\succeq$ M8 $\succeq$ M9

M1 $\succeq$ M3

M1 $\succeq$ M4 $\succeq$ M5

M1: $\begin{bmatrix} \\ \end{bmatrix}$

M2: $\begin{bmatrix} \text{CAT} & \textit{np} \end{bmatrix}$ M3: $\begin{bmatrix} \text{CAT} & \textit{vp} \end{bmatrix}$ M4: $\begin{bmatrix} \text{AGR}\vert\text{PER} & \textit{3} \end{bmatrix}$

M5: $\begin{bmatrix} \text{AGR} & \begin{bmatrix} \text{NUM} & \textit{pl} \\ \text{PER} & \textit{3} \end{bmatrix} \end{bmatrix}$ M6: $\begin{bmatrix} \text{AGR} & \begin{bmatrix} \text{NUM} & \textit{sg} \\ \text{PER} & \textit{3} \end{bmatrix} \end{bmatrix}$

M7: $\begin{bmatrix} \text{CAT} & \textit{np} \\ \text{AGR} & \begin{bmatrix} \text{NUM} & \textit{sg} \\ \text{PER} & \textit{3} \end{bmatrix} \end{bmatrix}$

M8: $\begin{bmatrix} \text{CAT} & \textit{np} \\ \text{AGR} & \begin{bmatrix} \text{NUM} & \textit{sg} \\ \text{PER} & \textit{3} \end{bmatrix} \\ \text{SUBJ} & \begin{bmatrix} \text{NUM} & \textit{sg} \\ \text{PER} & \textit{3} \end{bmatrix} \end{bmatrix}$

M9: $\begin{bmatrix} \text{CAT} & \textit{np} \\ \text{AGR} & \boxed{1} \begin{bmatrix} \text{NUM} & \textit{sg} \\ \text{PER} & \textit{3} \end{bmatrix} \\ \text{SUBJ} & \boxed{1} \end{bmatrix}$

13.4.5 Unifikation als zweistellige Operation auf der Menge der Merkmalstrukturen

Def. 11 *M1, M2 und M3 seien Merkmalstrukturen. M3 ist genau dann* **Unifikation** *von M1 und M2 (M3 = M1 ∧ M2) , wenn*

- *M3 von M1 und M2 subsumiert wird und*
- *M3 alle anderen Merkmalstrukturen M subsumiert, die ebenfalls von M1 und M2 subsumiert werden.*

Folgerungen:

- M3 ist die am wenigsten informative Merkmalstruktur, die M1 und M2 subsumiert.
- M3 beschreibt all die linguistischen Objekte, die von M1 und M2 beschrieben werden (im Sinne des mengentheoretischen Durchschnitts).
- M3 enthält alle Informationen aus M1 und M2, aber keine zusätzlichen.
- Es sei P ein vollständiger Pfad in M1. Dann ist P auch in M3 enthalten. Kommt P auch in M2 vor, dann hat er dort den gleichen Wert wie in M1.
- Subsumiert eine Merkmalstruktur M1 eine zweite Merkmalstruktur M2 (M1 $\succeq$ M2), dann gilt M1 ∧ M2 = M2.

Es gibt Merkmalstrukturen M1 und M2, für die die Unifikation als Merkmalstruktur nicht definiert ist. In diesem Falle schlägt die Unifikation fehl. Bei der Unifikation von

$$\begin{bmatrix} \text{A1} & \begin{bmatrix} \text{A1} & \boxed{1} \end{bmatrix} \\ \text{A2} & \boxed{1} \end{bmatrix} \tag{13.18}$$

mit

$$\begin{bmatrix} \text{A1} & \boxed{2} \\ \text{A2} & \begin{bmatrix} \text{A1} & \boxed{2} \end{bmatrix} \end{bmatrix} \tag{13.19}$$

würde die zyklische Struktur

$$\begin{bmatrix} \text{A1} & \boxed{2} & \begin{bmatrix} \text{A1} & \boxed{1} \end{bmatrix} \\ \text{A2} & \boxed{1} & \begin{bmatrix} \text{A1} & \boxed{2} \end{bmatrix} \end{bmatrix} \tag{13.20}$$

entstehen. (13.20) ist nach Definition aber keine Merkmalstruktur. Man kann auch sagen, daß die Unifikation in diesem Fall eine ungültige Struktur ergibt.

Die folgenden beiden Beispiele sollen zeigen, daß das Ergebnis der Unifikation von Merkmalstrukturen, in denen *structure sharing* vorkommt, mit einer anderen Merkmalstruktur, sich von der Unifikation zweier Merkmalstrukturen ohne *structure sharing* unterscheiden kann.

$$\begin{bmatrix} \text{AGR} & \boxed{1}\begin{bmatrix}\text{NUM} & sg\end{bmatrix} \\ \text{SUBJ} & \boxed{1} \end{bmatrix} \wedge \begin{bmatrix}\text{SUBJ} & \begin{bmatrix}\text{PER} & 3\end{bmatrix}\end{bmatrix} = \begin{bmatrix} \text{AGR} & \boxed{1}\begin{bmatrix}\text{NUM} & sg \\ \text{PER} & 3\end{bmatrix} \\ \text{SUBJ} & \boxed{1} \end{bmatrix} \tag{13.21}$$

$$\begin{bmatrix} \text{AGR} & \begin{bmatrix}\text{NUM} & sg\end{bmatrix} \\ \text{SUBJ} & \begin{bmatrix}\text{NUM} & sg\end{bmatrix} \end{bmatrix} \wedge \begin{bmatrix}\text{SUBJ} & \begin{bmatrix}\text{PER} & 3\end{bmatrix}\end{bmatrix} = \begin{bmatrix} \text{AGR} & \begin{bmatrix}\text{NUM} & sg\end{bmatrix} \\ \text{SUBJ} & \begin{bmatrix}\text{NUM} & sg \\ \text{PER} & 3\end{bmatrix} \end{bmatrix} \tag{13.22}$$

Def. 12 *In den Verband der Merkmalstrukturen wird zusätzlich das Element aufgenommen, das für widersprüchliche Merkmalstrukturen steht.* $\perp$ *ist das Ergebnis der Operation* $\wedge$ *im Falle der Nichtunifizierbarkeit.* $\perp$ *wird auch* **Bottom** *genannt.*

Folgerungen:

- $\perp$ ist das minimale Element in der Halbordnung, die durch $\preceq$ gebildet wird.
- $\perp$ beschreibt kein linguistisches Objekt.
- Demgegenüber beschreibt [ ] (auch mit $\top$ für **Top** bezeichnet) jedes beliebige linguistische Objekt.

Es sei für beliebige Merkmalstrukturen O(M) die Menge der von M beschriebenen linguistischen Objekte. Dann ist O(M1 $\wedge$ M2) = O(M1) $\cap$ O(M2).

Wenn zwei verschiedene Merkmalstrukturen M1 und M2 das gleiche linguistische Objekt beschreiben, ist es gerechtfertigt, eine neue Struktur M zu bilden, die die Informationen aus M1 und M2 in sich vereinigt. Im Normalfall enthält also M mehr Informationen als M1 und mehr Informationen als M2.

Eigenschaften der Unifikation

- Die Unifikation ist idempotent: A $\wedge$ A = A
- Die Unifikation ist kommutativ: A $\wedge$ B = B $\wedge$ A

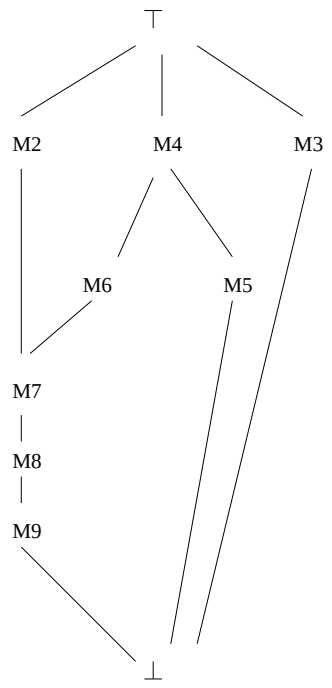

Abbildung 13.3: Halbordnung der Beispielstrukturen

- Die Unifikation ist assoziativ: $(A \wedge B) \wedge C = A \wedge (B \wedge C)$
- Top entspricht dem Einselement der Multiplikation von natürlichen Zahlen:
  $A \wedge \top = A$
- Bottom entspricht dem Nullelement der Multiplikation von natürlichen Zahlen:
  $A \wedge \bot = \bot$
- Unifikation und Subsumption sind auseinander definierbar:
  $A \preceq B \leftrightarrow A \wedge B = A$

Die Unifikation ist eine Operation, bei der sich nicht widersprechende, möglicherweise voneinander unabhängige Informationen vereinigt werden. Wenn in den Informationen ein Widerspruch vorliegt, d.h. einem Merkmal unvereinbare Werte zugeordnet werden müßten, ist das Ergebnis der Unifikation Bottom.

13.4.6 Negation, Disjunktionen und Implikationen

Gegeben sei ein linguistisches Objekt LO. Wenn zwei Merkmalstrukturen M1 und M2 dieses Objekt beschreiben, dann beschreibt auch die logische Konjunktion M1 $\wedge$ M2 dieses Objekt.

Umgekehrt gilt: Wenn eine Merkmalstruktur M ein linguistisches Objekt LO beschreibt und bekannt ist, daß M = M1 $\wedge$ M2 gilt, dann wird LO auch von M1 und M2 beschrieben.

Merkmalstrukturen sind im logischen Sinne als Konjunktionen ihrer Bestandteile aufzufassen. Die Konjunktion entspricht dabei der Unifikationsoperation. Eine Merkmalstruktur ist also als eine Aussage über ein linguistisches Objekt aufzufassen, die wahr bzw. falsch sein kann. Es gibt außer Konjunktionen noch andere Operationen über logischen Aussagen. Im folgenden werden die Analoga für Merkmalstrukturen definiert.

Disjunktion

Disjunktion von Merkmalstrukturen können eine Menge von atomaren Werten sein:

Person	1 $\vee$ 3
Numerus	sg $\vee$ pl

Disjunktion von Merkmalstrukturen können aber auch eine Menge von komplexen Merkmalstrukturen sein. Ein Beispiel ist die Merkmalstruktur für den Artikel *die*:

$$M_{die}: \begin{bmatrix} \text{CAS} & nom \vee acc \\ \text{AGR} & \begin{bmatrix} \text{GEN} & fem \\ \text{NUM} & sg \end{bmatrix} \vee \begin{bmatrix} \text{NUM} & pl \end{bmatrix} \end{bmatrix} \tag{13.23}$$

Merkmalstrukturen werden durch Disjunktion erweitert. Merkmalstrukturen ohne Disjunktion werden Basis-Merkmalstrukturen genannt. Jede Merkmalstruktur läßt sich als Disjunktion von Basis-Merkmalstrukturen darstellen.

Def. 13 *Eine Disjunktion von Basis-Merkmalstrukturen $A = A_1 \vee A_2 \vee ... \vee A_n$ wird von einer Disjunktion von Basis-Merkmalstrukturen $B = B_1 \vee B_2 \vee ... \vee B_m$* **subsumiert** *($A \preceq B$) gdw. jedes A_i von einem B_j subsumiert wird.*

Für nicht-Basis-Merkmalstrukturen gilt diese Definition nicht. Mann kann jedoch daraus, daß jedes A_i von einem B_j subsumiert wird folgern, daß B A subsumiert. Daß der Umkehrschluß für nicht-Basis-Merkmalstrukturen nicht möglich ist, zeigt das folgende Beispiel. Die beiden Strukturen subsumieren einander, aber für B_1 gibt es kein A_i, das B_1 subsumiert.

$$\begin{aligned} A &= \begin{bmatrix} \text{A} & a \end{bmatrix} \vee \begin{bmatrix} \text{A} & b \end{bmatrix} \\ B &= \begin{bmatrix} \text{A} & a \vee b \end{bmatrix} \end{aligned} \tag{13.24}$$

Def. 14 *Das Ergebnis der* **Unifikation einer Disjunktion** $A = A_1 \vee A_2 \vee ... \vee A_n$ *mit einer Disjunktion* $B = B_1 \vee B_2 \vee ... \vee B_m$ *ist eine Disjunktion* $C = C_1 \vee C_2 \vee ... \vee C_l$ *wobei die* C_i *die Menge der Unifikate jeweils eines Elements von A mit einem Element von B sind.*

Das folgende Beispiel zeigt die Unifikation der Struktur M_{die} (13.23) mit der Struktur $\mathrm{M}_{Menschen}$.

$$\mathrm{M}_{Menschen} : \begin{bmatrix} \mathrm{CAS} & gen \vee dat \vee acc \\ \mathrm{AGR} & \begin{bmatrix} \mathrm{GEN} & mas \end{bmatrix} \end{bmatrix} \vee \begin{bmatrix} \mathrm{CAS} & nom \\ \mathrm{AGR} & \begin{bmatrix} \mathrm{GEN} & mas \\ \mathrm{NUM} & pl \end{bmatrix} \end{bmatrix}$$

$$\mathrm{M}_{die} \wedge \mathrm{M}_{Menschen} = \begin{bmatrix} \mathrm{CAS} & nom \vee acc \\ \mathrm{AGR} & \begin{bmatrix} \mathrm{GEN} & mas \\ \mathrm{NUM} & pl \end{bmatrix} \end{bmatrix}$$

Ein anderes Beispiel:

$$([A\ u] \vee [A\ v]) \wedge ([B\ w] \vee [A\ w]) = \begin{bmatrix} \mathrm{A} & u \\ \mathrm{B} & w \end{bmatrix} \vee \begin{bmatrix} \mathrm{A} & v \\ \mathrm{B} & w \end{bmatrix}$$

Eigenschaften der Disjunktion

Die Disjunktion zweier Merkmalstrukturen A und B ist die informativste Merkmalstruktur, die beide subsumiert.

- Die Disjunktion ist idempotent: $\mathrm{A} \vee \mathrm{A} = \mathrm{A}$
- Die Disjunktion ist kommutativ: $\mathrm{A} \vee \mathrm{B} = \mathrm{B} \vee \mathrm{A}$
- Die Disjunktion ist assoziativ: $(\mathrm{A} \vee \mathrm{B}) \vee \mathrm{C} = \mathrm{A} \vee (\mathrm{B} \vee \mathrm{C})$
- Top $\top$: $\top \vee \mathrm{A} = \top$
- Bottom $\bot$: $\bot \vee \mathrm{A} = \mathrm{A}$
- Disjunktion und Subsumption sind auseinander definierbar:
  $\mathrm{A} \preceq \mathrm{B} \leftrightarrow \mathrm{A} \vee \mathrm{B} = \mathrm{B}$

Unifikation und Disjunktion interagieren ähnlich wie Addition und Multiplikation, d.h. es gilt auch die Distributivität:

- $A \wedge (B \vee C) = (A \wedge B) \vee (A \wedge C)$
  $A \vee (B \wedge C) = (A \vee B) \wedge (A \vee C)$

Implikation

Während wir die Disjunktion auf allen Ebenen der Merkmalstrukturen betrachtet haben, ist die Implikation $A \Rightarrow B$ nur auf der Ebene vollständiger Merkmalstrukturen sinnvoll. Mit der Implikation werden bestimmte Prinzipien ausgedrückt, die für Merkmalstrukturen innerhalb linguistischer Theorien gelten. Ihre Bedeutung wird bei der Behandlung der HPSG klarer werden.

Die Implikation zweier Merkmalstrukturen A und B ($A \Rightarrow B$) ist die am wenigsten informative Merkmalstruktur, deren Unifikation mit A von B subsumiert wird.

Folgerung: Wenn ein linguistisches Objekt X sowohl von einer Merkmalstruktur C als auch von $A \Rightarrow B$ beschrieben wird und C von A subsumiert wird, dann ist jede Merkmalstruktur C', C' = $C \wedge B$, auch eine Beschreibung für X.

Negation

Die Negation von Merkmalstrukturen kann anstelle einer umfangreichen Disjunktion eingesetzt werden, ist aber stets durch diese ersetzbar, wenn man voraussetzt, daß bekannt ist, welche Werte ein Merkmal annehmen kann und welche Merkmale zu einer Struktur einer bestimmten Art gehören. $\neg$ A entspricht außerdem der Implikation $A \Rightarrow \perp$.

Beispiel:

$$\begin{bmatrix}\text{NUM} & pl\end{bmatrix} \vee \begin{bmatrix}\text{PER} & 1 \\ \text{NUM} & sg\end{bmatrix} \vee \begin{bmatrix}\text{PER} & 2 \\ \text{NUM} & sg\end{bmatrix} = \neg \begin{bmatrix}\text{PER} & 3 \\ \text{NUM} & sg\end{bmatrix} \tag{13.25}$$

13.4.7 Listen

Listen sind geordnete Mengen von Merkmalstrukturen. Sie werden benötigt, um Sequenzen von Objekten zu beschreiben. Ein Beispiel für die Verwendung von Listen ist die Subcat-Liste, die zu jedem HPSG-Zeichen gehört. Sie wird im Kapitel 17.4 genauer beschrieben.

Die Subsumption und Unifikation von Listen ist wie folgt definiert:

Def. 15 *Eine Liste* $A = < A_1, \ldots, A_n >$ **subsumiert** *eine Liste* $B = < B_1, \ldots, B_n >$ *(*$A \succeq B$*) gdw. jedes* A_i *das dazugehörige* B_i *subsumiert (*$A_i \succeq B_i$*, i = 1, . . . , n).*

Daraus folgt: Eine Subsumptionsrelation kann nur zwischen zwei Listen mit derselben Anzahl von Elementen bestehen.

Def. 16 *Seien* $A = < A_1, \ldots, A_n >$, $B = < B_1, \ldots, B_n >$ *und* $C = < C_1, \ldots, C_n >$ *Listen von Merkmalstrukturen. C ist die* **Unifikation** *von A und B (*$C = A \wedge B$*) gdw. für jedes* C_i *gilt* $C_i = A_i \wedge B_i$.

Das heißt, die Unifikation von Listen wird unter Beachtung der Reihenfolge elementweise durchgeführt.

13.4.8 Funktionen

In bestimmten Fällen hängt der Wert eines Pfades von Werten anderer Pfade innerhalb einer Struktur ab. Diese Abhängigkeit kann man durch Funktionen ausdrücken. Ein Beispiel für eine Funktion, die innerhalb von Merkmalstrukturen verwendet wird, ist $append(X,Y)$. *append* liefert als Funktionswert eine Liste, die eine Verkettung von X und Y ist:

$$\begin{aligned} &append(< X_1, X_2, \ldots, X_n >, < Y_1, Y_2, \ldots, Y_m >) = \\ &\qquad < X_1, X_2, \ldots, X_n, Y_1, Y_2, \ldots, Y_m > \end{aligned}$$

Statt *append* wird auch das Operationssymbol $\oplus$ verwendet.

13.4.9 Typisierte Merkmalstrukturen als Erweiterung des Grundformalismus

Bei der bisherigen Behandlung von Merkmalstrukturen gibt es folgende Probleme:

- Es gibt keinerlei Einschränkungen für die Wahl von Merkmalen und ihren Werten beim Schreiben von Merkmalstrukturen. So ist z.B.

  $$\begin{bmatrix} \text{AGR} & \begin{bmatrix} \text{PER} & 3 \\ \text{NUM} & sg \end{bmatrix} \end{bmatrix} \tag{13.26}$$

  mit

  $$\begin{bmatrix} \text{IDIOT} & karl \end{bmatrix} \tag{13.27}$$

  unifizierbar. Das Unifikat wäre eine Merkmalstruktur, die Informationen völlig verschiedener Art enthält.

- Eine Negation läßt sich als Disjunktion darstellen. Bisher wurde aber nirgends festgelegt, welche Merkmale und welche Werte für diese Disjunktionen überhaupt in Frage kommen.

  Die Negation ¬[NUM pl] als Disjunktion der Form [NUM sg] ∨ [NUM 17] ∨ [IDIOT karl] zu schreiben, wäre sicher nicht sonderlich sinnvoll.

- Es ist oft wichtig zu unterscheiden, ob Informationen über ein bestimmtes Objekt zu einem Zeitpunkt unbekannt oder ob sie für das beschriebene Objekt irrelevant sind. Zum Beispiel sind bei Agreementmerkmalen Person und Numerus wichtig. In der Struktur AGR[NUM pl] ist die Information über die Person nicht enthalten. Eine Unifikation mit AGR[PER 3] ist sinnvoll. Für eine Agreement-Merkmalstruktur ist die Information, daß Karl ein Idiot ist, belanglos.

Die Probleme lassen sich durch die Zuordnung von Typen zu Merkmalstrukturen lösen.

Hat eine Merkmalstruktur M einen Typ t, dann wird durch t die Menge der Merkmale $a_1, ..., a_n$, die zu M gehören spezifiziert, und die möglichen Werte der Merkmale $a_1, ..., a_n$ sind durch Typen $t_1, ..., t_n$ festgelegt.

Typbezeichnungen werden in *kursiven* Kleinbuchstaben geschrieben.

Subsumption und Unifikation mit Typen

Die Definitionen sind analog zu denen der Merkmalstrukturen, d.h.

Def. 17 *Ein Typ t1* **subsumiert** *einen Typ t2 (t1 $\succeq$ t2), wenn gilt:*

- *Wenn t1 und t2 selbst keine Struktur haben, dann muß t1 allgemeiner als t2 sein.*

- *Wenn t1 und t2 eine Struktur haben, dann muß gelten: Jedes Merkmal ATTR, das in Merkmalstrukturen vom Typ t1 vorkommt, muß auch in Merkmalstrukturen vom Typ t2 vorkommen, und für die ATTR zugeordneten Typen $t1_{ATTR}$ und $t2_{ATTR}$ gilt: $t1_{ATTR} \succeq t2_{ATTR}$.*

Man sagt, *t1* ist **Supertyp** von *t2* bzw. *t2* ist **Subtyp** von *t1*.

Def. 18 *t1, t2 und t3 seien Typen. t3 ist dann die* **Unifikation** *von t1 und t2, wenn*

- *t3 von t1 und t2 subsumiert wird und*

- *t3 alle anderen Typen t subsumiert, die ebenfalls von t1 und t2 subsumiert werden.*

Aus den obigen Definitionen folgt:

- Ist t_1 Supertyp von t_2, dann ist t_2 das Unifikat von t_1 und t_2, d.h. der spezifischere Typ der zwei Typen ist das Unifikat dieser beiden Typen.
- Ein Merkmal, das in einem Typ vorkommt, kommt auch in jedem Subtyp dieses Typs vor.
- Wenn ein Typ *t* verlangt, daß die Werte eines seiner Merkmale einen gewissen Typ *t'* haben, dann gilt das auch für alle seine Subtypen von it t.

Durch die Subsumption der Typen wird eine Hierarchie gebildet, deren oberstes Element $\top$ ist. Dabei erbt ein Typ alle Eigenschaften seiner Supertypen und gibt selbst Eigenschaften hinzu . Hinzugefügt werden können weitere Merkmale sowie Einschränkungen der Typen der Werte seiner ererbten Merkmale. Es ist möglich, gleichzeitig von mehreren Supertypen zu erben, die jeweiligen Merkmalstrukturen werden unifiziert.

Ein nichtlinguistisches Beispiel zeigt Abbildung 13.4. Der *Kopierer* erbt vom

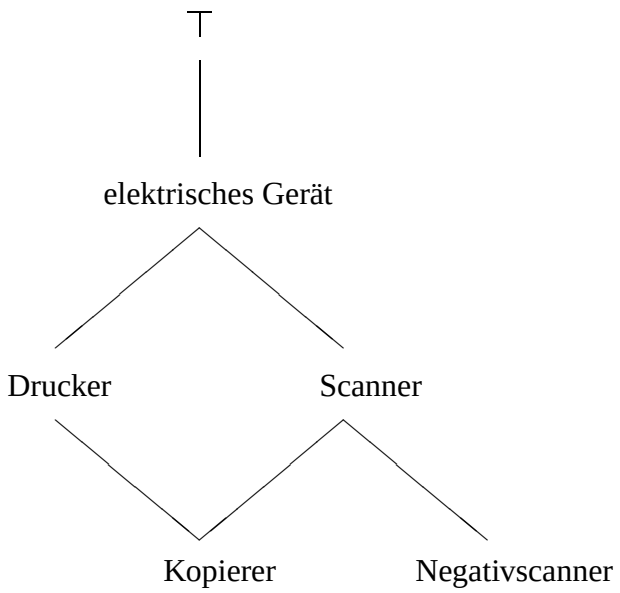

Abbildung 13.4: Ein nichtlinguistisches Beispiel für eine Typhierarchie

Drucker und vom *Scanner* und diese vom *elektris chen Gerät*. Ein elektrisches Gerät besitzt z.B. die Eigenschaft, eine Stromversorgung zu haben, diese Eigenschaft wird an *Drucker* und an *Scanner* vererbt. Ein *Drucker* kann Informationen

ausgeben, ein *Scanner* Informationen erfassen. Ein *Kopierer* kann beides. Ein *Negativscanner* kann nur bestimmte Informationen einlesen, nämlich die von Negativen. *Kopierer* ist die Unifikation von *Scanner* und *Drucker*. Die Unifikation von *Scanner* und *Negativscanner* ist *Negativscanner*.

Def. 19 *Ein Merkmalstruktur ist* **typisiert**, *wenn die möglichen Merkmale, die in der Merkmalstruktur auftauchen können, durch deren Typ festgelegt sind.*

Def. 20 *Ein Merkmalstruktur ist* **total typisiert (totally well-typed)**, *falls sie typisiert ist und außerdem jedes Merkmal, das zu dem Typ einer Teilmerkmalstruktur gehört, auch vorhanden ist.*

Def. 21 *Eine Merkmalstruktur ist* **Typen-aufgelöst (sort-resolved)**, *falls jede Teilmerkmalstruktur einen Typ zugewiesen bekommen hat, der maximal ist, d.h. keinen Subtyp besitzt.*

Man schreibt typisierte Merkmalstrukturen wie folgt:

$$\begin{bmatrix} \text{ATTR1} & [\textit{type1}] \\ \text{ATTR2} & [\textit{type2}] \\ \textit{type} & \end{bmatrix} \tag{13.28}$$

Die Schriftzüge in den kleinen *kursiven* Buchstaben sind jeweils die Typbezeichnungen.

Die folgende Struktur ist eine typisierte Merkmalstruktur, die Typen-aufgelöst ist. Ob sie auch total typisiert ist, hängt von den Festlegungen für die entsprechenden Typen ab, die beschreiben, welche Merkmale zu Strukturen dieses Typs gehören.

$$\begin{bmatrix} \text{SUBJ} & \begin{bmatrix} \text{PHON} & [\textit{hans}] \\ \text{AGR} & \boxed{1} \begin{bmatrix} \text{NUM} & [\textit{sg}] \\ \text{PER} & [\textit{3}] \\ \textit{agr} & \end{bmatrix} \\ \textit{subj} & \end{bmatrix} \\ \text{PRED} & \begin{bmatrix} \text{PHON} & [\textit{schläft}] \\ \text{AGR} & \boxed{1} \\ \textit{pred} & \end{bmatrix} \\ \textit{satz} & \end{bmatrix} \tag{13.29}$$

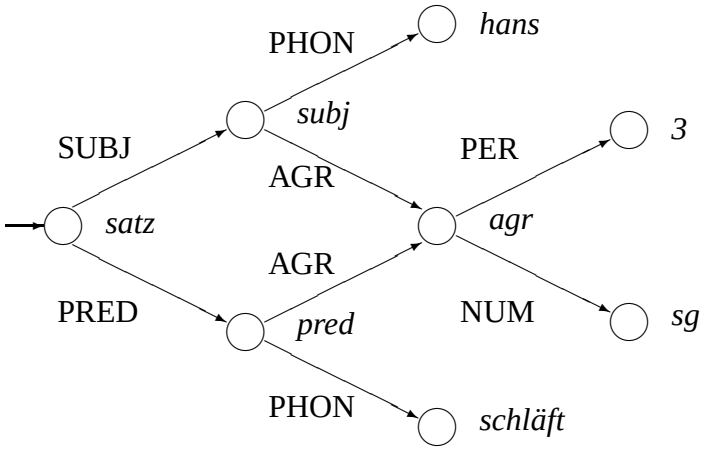

Abbildung 13.5: Die Struktur in Graphendarstellung

Die Struktur (13.16) unterscheidet sich von (13.29) nur dadurch, daß jede Teilmerkmalstruktur in (13.29) eine Typbezeichnung besitzt. Abbildung 13.5 unterscheidet sich nur dadurch von Abbildung 13.2, daß rechts neben jedem Knoten sein Typ steht.

Manche Typen haben keine Merkmale (im Beispiel *3* und *sg*). Sie werden Atome genannt. Eine Merkmalstruktur, die nur aus einem Typ besteht, wird auch Atom genannt, obwohl die korrekte Bezeichnung atomare Merkmalstruktur sein müßte.[3] Statt

$$\begin{bmatrix} 3 \end{bmatrix} \tag{13.30}$$

schreibt man nur *3*.

Das Merkmal *CASE* hat als Wert eine atomare Merkmalstruktur vom Typ *case*. *nom*, *gen*, *dat* und *acc* sind Subtypen von *case*. *case* subsumiert *nom*.

Man kann negierte Typen mit in eine Typhierarchie aufnehmen. Man spart sich damit die Berechnung der Disjunktionen, die den Negationen entsprechen würden. Ein Beispiel für den Supertyp *per* zeigt die Abbildung 13.6.

Subsumption und Unifikation auf typisierten Merkmalstrukturen sind analog zu Subsumption und Unifikation auf nicht typisierten Merkmalstrukturen definiert. Zu beachten sind dabei neben der Subsumption der einfachen Merkmal-

[3]Merkmalstrukturen vom Typ *hans* oder *schläft* gibt es in der HPSG nicht. Sie würden das Typkonzept sinnlos machen, da man für jede phonologische Variante eines Wortes einen Typ definieren müßte. Statt dessen ist der Wert des *PHON*-Merkmals vom Typ *list*.

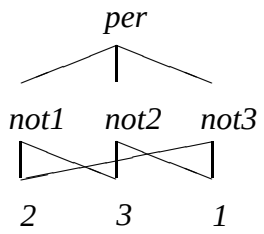

Abbildung 13.6: Typhierarchie, die auch negierte Subtypen des Typs *per* enthält

strukturen die zusätzlichen Bedingungen, die durch die Typhierarchie dazukommen.

Wie schon erwähnt, kann man [AGR [PER 3]] und [AGR [NUM sg]] als partielle Beschreibung oder Teilspezifikation eines linguistischen Objekts auffassen. Von Merkmalstrukturen, die ein linguistisches Objekt repräsentieren sollen, verlangt man, daß sie vollständig sind. Das heißt, sie müssen total typisiert und Typen-aufgelöst sein. Es darf keine Typen in der Struktur geben, die atomare Subtypen besitzen.[4]

Lesetips

Dieses Kapitel entspricht der Einführung in den Formalismus getypter Merkmalstrukturen in Müller (1999).

In Johnson (1988) wird der Formalismus ungetypter Merkmalstrukturen mathematisch exakt beschrieben. Es werden Kriterien angegeben, die eine Attribut-Wert-Grammatik erfüllen muß, damit das Halteproblem für Parser entscheidbar ist.

? setzt sich sehr mathematisch mit getypten Merkmalstrukturen auseinander.

Im Buch von Stuart Shieber (1986) findet man eine allgemeine Einführung in die Theorie der Unifikationsgrammatiken. Es wird ein ziemlich allgemeiner Überblick gegeben, und wichtige Grammatik-Typen (DCG, LFG, GPSG, HPSG, PATR-II) werden kurz vorgestellt.

[4]Eine Konsequenz wäre, daß man für den Lexikoneintrag für *Frau* den Wert des Kasus-Merkmals als atomaren Typ angeben muß. Das würde bedeuten, daß man vier verschiedene Einträge für *Frau* hat, die jeweils maximal spezifische Typen als Kasus-Merkmal haben. Es ist sicherlich sinnvoll, stattdessen eine Disjunktion der vier möglichen Kasuswerte zuzulassen bzw. statt dieser Disjunktion einfach den Supertyp aller vier Typen, nämlich *case* ins Lexikon zu schreiben.

Kontrollfragen

1. Was passiert bei der Term-Unifikation, wenn zwei Terme mit gleichem Funktor und einer unterschiedlichen Anzahl von Argumenten unifiziert werden sollen?

2. Was passiert bei der Prolog-Unifikation bei der Unifikation von X und f(X)? Was sollte passieren?

3. Was ist eine Merkmalstruktur?

4. Wie kann man eine komplexe Merkmalstruktur als DAG schreiben?

5. Was sind Reentrancys?

6. Welche Erweiterungen des Merkmalstrukturbegriffs kennen Sie?

7. Was ist Subsumption?

8. Wie hängt die Unifikation mit der Subsumption zusammen?

Übungsaufgaben

7. * Wie kann man Listen mit dem Formalismus der Merkmalstrukturen beschreiben, ohne die in Kapitel 13.4.7 beschriebene Erweiterung des Formalismus zu benutzen?

Kapitel 14

Eine PATR-II-Implementierung

14.1 Der PATR-II-Formalismus

PATR-II ist eine weitverbreitete Notation zur Beschreibung von Merkmalstrukturen und von Grammatiken, die diese kombinieren. PATR besteht aus einem Skelett von kontextfreien Regeln, in denen die Symbole nicht Atome oder Terme sondern Merkmalstrukturen sind. Zu jeder Regel gehört eine Menge von Gleichungen, die die Merkmalstrukturen beschreiben, die zur Regel gehören.

Die DCG-Regel `s --> np,vp` ist äquivalent zu der PATR-II Regel der Form:

```
X0 -> X1 X2
<X0 cat > = s
<X1 cat> = np
<X2 cat> = vp
```

In der PATR-Regel werden die Kategorien als Merkmale dargestellt. Die Ausdrücke in spitzen Klammern beschreiben Pfade in Merkmalstrukturen. In der obigen Regel gibt es nur Gleichungen zwischen Pfaden und Werten, man kann aber auch Gleichungen zwischen zwei Pfaden formulieren. Die Regel `s --> np(Agr), vp(Agr).` entspricht:

```
X0 -> X1 X2
<X0 cat> = s
<X1 cat> = np
<X2 cat> = vp
<X1 agr> = <X2 agr>
```

Ein Beispiel für Agreement mit den entsprechenden Lexikoneinträgen folgt:

```
X0 -> X1 X2
```

```
<X0 cat> = s
<X1 cat> = np
<X2 cat> = vp
<X0 head> = <X2 head>
<X2 head subj> = <X1>

X0 -> fred
<X0 cat> = np
<X0 head agr num> = sing
<X0 head agr per> = 3

X0 -> sleeps
<X0 cat> = vp
<X0 head vform> = fin
<X0 head subj head agr num> = sing
<X0 head subj head agr per> = 3
```

14.2 Subkategorisierung

Die Grammatiken, die wir bis jetzt betrachtet haben, haben die Beziehungen zwischen einem Verb und seinen Komplementen durch ein atomares Subcat-Merkmal beschrieben, das die Verbindung zwischen dem Lexikoneintrag des Verbs und einer Phrasenstrukturregel herstellt. Ein Trend in der modernen Linguistik geht dahin, die Komplemente von Köpfen stärker im Lexikon zu spezifizieren und dafür allgemeinere Phrasenstrukturregeln zu verwenden. Dieser Ansatz ist charakteristisch für die Head-Driven Phrase Structure Grammar (HPSG) und für die Kategorialgrammatik (CG).

In der HPSG ist der Wert des Subcat-Merkmals eine Liste. Die Graph-Repräsentation einer Liste benutzt die Merkmale *first* und *rest*. Ein Verb, das eine np und eine vp als Komplement verlangt, würde die folgende Beschreibung für den Subcat-Wert haben:

```
<X subcat first cat> = np
<X subcat rest first cat> = vp
<X subcat rest rest first> = nil
```

Jedes Element der Liste wird mit einer der Komplementkategorien unifiziert. Da die Merkmalstruktur in der Subcat-Liste Information über die Kategorie des Komplements enthält, braucht diese Information in der Regel, die die Kombination vornimmt, nicht mehr vorhanden zu sein. Wir können also allgemeine Regeln der Form

```
X0 -> X1 X2
<X0 cat> = vp
<X1 cat> = vp
<X0 head> = <X1 head>
<X1 subcat first> = <X2>
<X1 subcat rest> = <X0 subcat>
```

benutzen. In der obigen Regel wird das erste Element der Subcat-Liste der Kopftochter mit einer Komplementtochter und der Rest der Subcat-Liste mit der Subcat-Liste der Mutter unifiziert. Diese eine Regel, die dem Subkategorisierungsprinzip der HPSG entspricht, ersetzt alle Verbphrasenregeln in früheren Grammatiken. Als Grundlage für die obige Regel müssen wir nur die entsprechenden Subcat-Listen ins Lexikon schreiben.

Subjekte können entweder ein Wert eines spezifizierten Kopfmerkmals sein oder mit in die Subcat-Liste aufgenommen werden. Im letzteren Fall werden sie durch eine Regel der folgenden Art beschrieben.

```
X0 -> X1 X2
<X0 cat> = s
<X2 cat> = vp
<X0 head> = <X2 head>
<X2 subcat first> = <X1>
<X2 subcat rest first> = nil
```

Für das Verb *persuades* gibt es den folgenden Lexikoneintrag:

```
X0 -> persuades
<X0 cat> = vp
<X0 head vform> = fin
<X0 subcat first cat> = vp
<X0 subcat first head vform> = inf
<X0 subcat first subcat first cat> = np
<X0 subcat first subcat rest first> = nil
<X0 subcat rest first cat> = np
<X0 subcat rest first nform> = norm
<X0 subcat rest rest first cat> = np
<X0 subcat rest rest first agr per> = 3
<X0 subcat rest rest first agr num> = sg
<X0 subcat rest rest rest first> = nil
```

Diese Information würde man natürlich nicht in dieser Form aufschreiben. Da die oben angegebene Subcat-Liste für viele Verben charakteristisch ist (nämlich gerade für die *object control*-Verben) ist es sinnvoll, diese Information an anderer

Stelle zu kodieren und im Lexikon Abkürzungen zu verwenden. Diese Abkürzungen stehen dann für eine bestimmte Menge von Pfadgleichungen. Man nennt sie *Templates*. Templates können selbst wieder Templates enthalten. Die Prozedur, die die Abkürzungen expandiert muß also rekursiv sein. Wir kommen darauf in Kapitel 15 noch zurück.

14.3 PATR-II in Prolog

In Kapitel 14.1 haben wir gesehen, wie Regeln in einer Grammatik mit Merkmalen mit Hilfe der Unifikationsoperation interpretiert werden. Die Darstellung von Merkmalstrukturen als Term bringt einige Probleme mit sich und ist nicht sehr elegant. Obwohl die Terme einfacher in Prolog zu implementieren sind, haben wir uns für eine alternative Repräsentation der Merkmale – die Merkmalstrukturen – entschieden. Der Formalismus der Merkmalstrukturen bildete die Grundlage für den PATR-II-Formalismus, mit dem wir Grammatikregeln über Merkmalstrukturen schreiben können.

PATR-II-Regeln in Prolog auszudrücken, ist nicht schwer. Man muß nur die entsprechenden Operatoren definieren. Es folgt ein Beispiel für eine Prolog-Notation:

```
X0 ---> X1, X2 :-
   X0:cat <=> s,
   X1:cat <=> np,
   X2:cat <=> vp,
   X1:num <=> X2:num.

X0 ---> [john] :-
   X0:cat <=> np,
   X0:num <=> sg.
```

Dabei sind `--->` und `<=>` Prolog-Prädikate. Der Funktor `:` wird benutzt, um Pfade zu konstruieren.

Um einen vollständigen PATR-II-Interpreter zu bekommen, müssen wir das Prädikat `<=>` definieren. `<=>` bekommt zwei Argumente, die Merkmalstrukturen spezifizieren. Argumente können Prolog-Variablen, Atome oder Pfade sein. `<=>` muß

- beide Argumente „ausrechnen“. Das Ergebnis sind zwei Merkmalstrukturen oder Konstanten.
- beide Ergebnisse an ein Prädikat weitergeben, das sie „gleichmacht“, d.h. eine Graph-Unifikation durchführt.

Die Semantik von `<=>` wird in Prolog wie folgt beschrieben:

```
X <=> Y :-
  denotes(X,Z1),
  denotes(Y,Z2),
  graph_unify(Z1,Z2).
```

Dabei ist `denotes` ein Prädikat, das Pfade berechnet und `graph_unify` die Graph-Unifikation.

14.4 Die Repräsentation von Merkmalstrukturen

Die Merkmalstruktur

$$\begin{bmatrix} \text{NUM} & sg \\ \text{PER} & 3 \end{bmatrix} \tag{14.1}$$

kann man in Prolog in Form einer offenen Liste darstellen:

```
[num:sg,per:3|_]
```

Der nicht spezifizierte Rest ist wichtig. Er ermöglicht das Hinzufügen weiterer Merkmal-Wert-Paare, ohne daß eine komplett neue Struktur erzeugt werden muß. Im allgemeinen wird eine Merkmalstruktur als Liste von Merkmal-Wert-Paaren dargestellt, die mit einer Variable endet. Wenn ein Wert atomar ist, wird er durch ein Prolog-Atom repräsentiert. Ist der Wert komplex, so wird der Wert auch als Liste von Merkmal-Wert-Paaren mit einer Variable am Ende dargestellt.

Mit dieser Repräsentation kann man dann `denotes` implementieren. `denotes` liefert für den Pfad `X:cat` den Wert des Merkmals `cat`, der in der Prolog-Variablen `X` gespeichert ist. Ist kein Wert für `cat` in `X` enthalten, so wird `X` entsprechend instantiiert. `denotes` nimmt eine Pfadbeschreibung und gibt den Wert am Ende des Pfades zurück. Das Prädikat `denotes` muß dazu in der Lage sein, den Wert für Ausdrücke der folgenden Art zurückzugeben:

```
X
np
X:cat
X:subj:cat
```

Die Prolog-Implementation, die das leistet:

```
denotes(X,X) :- var(X),!.
denotes(X:F,V) :- !, denotes(X,V1), member(F:V,V1), !.
denotes(X,X).
```

Die erste Klausel behandelt den Fall der uninstantiierten Variable. Die zweite den Fall eines nichttrivialen Pfades. Das Programm findet heraus, was der erste Teil des Pfades (`X`) beschreibt (`V1`) und sucht dann einen Eintrag für das Merkmal `F` in der zurückgegebenen Liste. Der erste Cut in der Klausel sorgt dafür, daß weiter unten stehende Klauseln nicht mehr angewendet werden, falls das erste Argument von `denotes` die Form `X:F` hat. Der zweite Cut sorgt dafür, daß die erste Lösung von `member` genommen wird. Wenn `V1` zu Beginn nicht instantiiert ist, so wird es durch `member` soweit instantiiert, daß es einen Eintrag für `F` mit dem Wert `V` gibt. Die letzte Klausel behandelt alle anderen Fälle, z.B. atomare Werte oder etwas, das schon ein Graph ist.

14.5 Implementierung der Graph-Unifikation

Man erinnere sich daran, daß Prologs prozedurale Semantik zwei Strukturen, die Term-unifiziert wurden, durch deren Unifikat ersetzt. Genau dasselbe wollen wir bei der Graph-Unifikation tun, d.h. wenn `X <=> Y` erfüllt wurde, sollen `X` und `Y` für dieselbe Merkmalstruktur stehen. Eine Merkmalstruktur kann auf verschiedene Art und Weise aufgeschrieben werden. Die folgenden beiden Prolog-Beschreibungen sind äquivalent:

```
[num:sg,per:3|_]
[per:3,num:sg|_]
```

Das Unifikationssymbol von Prolog `=` können wir nicht umdefinieren. Unifikationen von offenen Listen, kann man deshalb nicht durch Gleichungen wie `[gen:mas|_] = [num:sg|_]` beschreiben. Wir müssen stattdessen ein Prädikat `graph_unify` definieren.

```
graph_unify(X,X) :- !.
graph_unify([A:V1|R1],F2) :-
  del(A:V2,F2,R2),
  graph_unify(V1,V2),
  graph_unify(R1,R2).

del(F,[F|X],X) :- !.
del(F,[E|X],[E|Y]) :- del(F,X,Y).
```

Die erste Klausel von `graph_unify` besagt, daß zwei Graphen unifizieren, wenn sie Term-unifizieren. Wenn z.B. ein DAG eine Variable ist, kann die erste Klausel angewendet werden. Sollen zwei atomare Merkmalstrukturen unifiziert werden, so findet auch die erste Klausel Anwendung.

Die zweite Klausel versucht zwei beliebige DAGs zu unifizieren und schlägt fehl, falls das nicht möglich ist. Das Prädikat `del(El,List1,List2)` ist wahr, wenn `List2` `List1` ohne das Element `El` ist. Dabei wird das am weitesten vorn stehende Vorkommen von `El` entfernt. Da wir Listen mit offenen Enden benutzen, ist `El` immer Element von `List1` nachdem `del` terminiert ist. Da `List1` der zweite DAG ist und `del` für jedes Merkmal des ersten DAGs aufgerufen wird, ist sichergestellt, daß der zweite DAG zumindest die Merkmale enthält, die im ersten DAG vorkommen. Gleichzeitig wird eine Liste konstruiert, die alle Merkmale enthält, die außerdem noch im zweiten DAG enthalten sind, (`R2`). Wenn alle Merkmale des ersten DAGs behandelt wurden, bleibt nur noch der Rest der Liste, der ja variabel ist. Die Variable wird mit den Merkmal-Wert-Paaren aus dem zweiten DAG unifiziert, die nicht im ersten sind (R2). Somit enthält der erste DAG dann auch die Merkmale, die im zweiten vorkommen. Beide DAGs enthalten also dieselben Merkmal-Wert-Paare.

`graph_unify` kann auch fehlschlagen. Das passiert, wenn eine Liste das Merkmal-Wert-Paar `Name:Value1` und die andere das Paar `Name:Value2` enthält und `Value1` und `Value2` verschiedene Atome oder nicht unifizierbare Graphen sind. Ein Beispiel:

```
X = [a1:v1, a2:v2, a3 : [a11:v11,a21:v21|_]|_],
Y = [a2:v2, a4:v4, a3 : [a21:v21,a31:v31|_]|_],
graph_unify(X,Y).

X=[a1:v1,a2:v2,a3:[a11:v11,a21:v21,a31:v31|_157],a4:v4|_105]
Y=[a2:v2,a4:v4,a3:[a21:v21,a31:v31,a11:v11|_157],a1:v1|_105]
```

14.6 Typskelette und *Partial Execution*

Im Kapitel 13 haben wir die Merkmalstrukturen eingeführt, da Terme bestimmte Nachteile haben. Terme sind, wenn hinreichend viele Argumente vorhanden sind, nicht mehr lesbar. Große Grammatiken mit vielen Merkmalen sind schlecht wartbar, da Änderungen in der Termstruktur an allen Stellen durchgeführt werden müssen.

Terme haben aber auch einen entscheidenden Vorteil: Die in Prolog implementierte Unifikation ist schneller als eine explizit durchgeführte Graphunifikation. Wie kann man nun die Vorteile der Graph-Darstellung mit den Vorteilen der Termunifikation verbinden?

Im allgemeinen ist klar, welche interne Struktur die Merkmalstrukturen bestimmter Zeichen haben können. Das heißt, es ist bekannt, welche Merkmale zu einer Merkmalstruktur gehören, die eine Nominalphrase beschreibt. Genauso weiß man, welche Merkmale zu einer Merkmalstruktur, die eine Verbphrase

beschreibt, gehören. Man kann eine kanonische Ordnung dieser Merkmale festlegen:

```
Sign isa sign_type :-
  Sign:phon     <=> _,
  Sign:head:cat <=> _.
```

Diese „Typbeschreibung“ enthält die Information, daß jede Beschreibung eines linguistischen Objekts vom Typ *sign_type* ein PHON-Merkmal und ein CAT-Merkmal haben muß. Das CAT-Merkmal steht unter dem Pfad HEAD|CAT. Außerdem gibt die obige „Typbeschreibung“ eine Standardreihenfolge der Merkmale vor, die zu diesem Typ gehören. Die Berechnung des Aufrufes `Sign isa sign_type.` ergibt einen Graphen der Form:

```
[phon:_,head:[cat:_|_]|_]
```

Diese Struktur ist nur ein leeres Gerüst. Sie enthält keine Information über Werte von Merkmalen sondern nur über die Reihenfolge in der Merkmale in Strukturen diesen Typs vorkommen müssen. Man beachte, daß die Listen, die bei der Berechnung der „Typbeschreibungen“ entstehen hinten offen sind, so daß beliebig viele Merkmale hinzugefügt werden können. Es ist somit möglich, „Typhierarchien“ aufzubauen. Ein Subtyp des Typs *sign_type* kann zum Beispiel der Typ *noun_type* sein. *noun_type* erbt alle Merkmale von *sign_type* und fügt noch bestimmte nomenspezifische Merkmale hinzu:

```
Sign isa noun_type :-
  Sign isa sign_type,
  Sign:head:cat <=> noun,
  Sign:head:cas <=> _,
  Sign:head:num <=> _.
```

Die Berechnung ergäbe:

```
[phon:_,head:[cat:noun,cas:_,num:_|_]|_]
```

Im Lexikon steht dann ein Eintrag der Form:

```
word(frau,Sign) :-
  Sign isa noun_type,
  Sign:phon <=> [frau],
  Sign:head:num <=> sg.
```

Das Ergebnis der Berechnung wäre:

```
[phon:[frau],head:[cat:noun,cas:_,num:sg|_]|_]
```

Dabei ist zu beachten, daß obwohl kein Wert für das Kasus-Merkmal angegeben wurde, eine entsprechende Stelle in der Struktur freigehalten wird. Man kann die Parsezeit eines Systems um den Faktor 10 reduzieren, wenn man statt der Graph-Unifikation die Prolog Termunifikation benutzt. Das wird möglich, wenn man an allen Stellen in der Grammatik, an denen auf irgendwelche Merkmale bezuggenommen wird, die entsprechende „Typ“-Information angibt und die Grammatik pre-compiliert. Das heißt, daß an jeder Stelle, wo ein Aufruf von `<=>` erfolgt, die entsprechende Berechnung vor der eigentlichen Benutzung der Grammatik erfolgen muß. Das erledigt das folgende Programm, das in ähnlicher Form in Cooper (1990) angegeben wurde:

```
pe((Head :- Body),(Head :- ExpandedBody)) :- !,
  partialy_ex(Body,ExpandedBody).

partialy_ex((Literal,Rest),Expansion) :- !,
  partialy_ex(Literal,ExpandedLiteral),
  partialy_ex(Rest,   ExpandedRest),
  conjoin(ExpandedLiteral,ExpandedRest,Expansion).

partialy_ex(A isa B, true) :- !,
  A isa B.
partialy_ex(A <=> B, true) :- !,
  A <=> B.

partialy_ex(Literal, Literal).

conjoin((A,B),C,ABC) :-
  conjoin(B,C,BC),
  conjoin(A,BC,ABC).
conjoin(true,A,A) :- !.
conjoin(A,true,A) :- !.
conjoin(A,C,(A,C)).
```

Die ersten vier Klauseln von `partial_ex` führen ein Prolog-Ziel der Form `Head :- Body` partiell aus. Es werden alle Aufrufe, die die Form `A isa B` oder `A <=> B` haben oder die eine Konjunktion sind ausgeführt. Für den Lexikoneintrag

```
word(frau,Sign) :-
 Sign isa noun_type,
 Sign:phon <=> [frau],
 Sign:head:num <=> sg.
```

ergibt sich nach der partiellen Ausführung der Lexikoneintrag

```
word(frau,[phon:[frau],head:[cat:noun,cas:_,num:sg|_]|_]) :-
```

Die Benutzung solcher Typskelette ist nicht mehr deklarativ im Sinne von Prolog. Die Reihenfolge der Typdefinition und der Pfadgleichungen ist nicht vertauschbar. Ein Prologprogramm, das solche Typskelette verwendet hat prozeduralen Charakter. Man kann sich diese Art von „Typ“ als Analogon zu Typen bzw. Records in Programmiersprachen wie Pascal oder C vorstellen. Im Prozedurkopf muß eine Typdeklaration enthalten sein, damit der Compiler weiß, wieviel Speicherplatz für die Manipulation von Daten beim Aufruf eine Prozedur zur Verfügung gestellt werden muß. Man beachte, daß jede „Typdeklaration“ für einen DAG vor der ersten Pfadgleichung, die diesen DAG verwendet, stehen muß:

```
word(frau,Sign) :-
 Sign:head:num <=> sg,
 Sign isa noun_type,
 Sign:phon <=> [frau].
```

Die Berechnung dieses Lexikoneintrages ergibt:

```
word(frau,[head:[num:sg,cat:noun,cas:_|_],phon:[frau]|_]) :-
```

Das Weiterarbeiten mit diesem Eintrag führt zu Unifikationsfehlern bei Unifikation mit korrekt „getypten“ DAGs.

Reduktion

Da die Merkmalsbezeichnungen nur eingeführt wurden, weil die Position von Werten in DAGs nicht fest ist, die Positionen der Werte in diesen „typisierten“ Merkmalstrukturen aber feststeht, kann man diesen Lexikoneintrag noch weiter reduzieren:

```
word(frau,[[frau],[noun,_,sg|_]|_]) :-
```

Diese Struktur ist nicht mehr ohne weiteres les- dafür aber um so effektiver verarbeitbar. Nach erfolgtem Parsen kann man diese Strukturen wieder expandieren. Die Arbeit, die zur Komprimierung aufgewendet werden muß, kann vor dem Parsen geleistet werden, so daß sie die Parsezeit nicht negativ beeinflußt. Man braucht weniger Zeit für das Parsen mit gepackten Kategorien und anschließender Entpackung als für das Parsen mit ungepackten Kategorien.

Kontrollfragen

1. Welche Prädikate müssen für einen PATR-II-Interpreter in Prolog geschrieben werden, und welche Argumente haben sie?

2. Warum werden Merkmalstrukturen durch offene Listen repräsentiert?

3. Welches Format haben Subcat-Listen von Wörtern in dem gezeigten Ansatz?

Übungsaufgaben

1. Zeichnen Sie die Graphen, die durch folgende Prolog-Terme repräsentiert werden. Wie sieht das Ergebnis der Unifikation und die entsprechende Prolog-Repräsentation aus?

```
[argument:[cat:np,num:_1|_2],cat:backward,
   result:[vform:inf,cat:s|_3] |_4]
[cat:backward, argument:[cat:np,num:sg, nform:norm|_5],
   result:[cat:s,vform:inf|_6]|_7]
```

2. Wenn die folgende PATR-II-Regel die einzige in der Datenbank ist, was liefert dann die Prolog-Anfrage `LHS ---> RHS`?

```
X0 ---> X1,X2 :-
X0:cat <=> s,
X1:cat <=> np,
X2:cat <=> vp,
X0:head <=> X2:head,
X2:head:subj <=> X1.
```

3. * Sehen sie sich den DCG-Interpreter aus Kapitel 9.2 an, und überlegen Sie, wo dieser Interpreter Term-Unifikation benutzt, um zu prüfen, ob Kategorien matchen. Wie muß man den Interpreter verändern, damit er für PATR-II-Regeln funktioniert?

Hausaufgabe

8. In PATR kann man Kategorien der Kategorialgrammatik wie folgt beschreiben:

```
<X cat> = np

<X cat> = forward
<X argument cat> = np
<X result cat> = s
```

Die erste Beschreibung entspricht einer NP, die zweite der komplexen Kategorie (s/np).

Schreiben Sie eine kleine Kategorialgrammatik in PATR-2-Notation. Die Funktionsanwendungsregeln und Lexikoneinträge für die folgenden Kategorien sollen enthalten sein.

- Nominalphrase
- Verbphrase
- singular Verbphrase (Nominalphrasen müssen ein Numerus-Merkmal haben)
- finite Verbphrase (ein S muß ein Vform-Merkmal haben)
- das Wort *to* aus Infinitvkonstruktionen wie *to go*

Benutzen Sie die Prolog-Notation für die PATR-Regeln.

Kapitel 15

Unifikationsgrammatiken und deren Lexikon

15.1 Die Komplexität von Lexikoneinträgen

Im letzten Kapitel haben wir Agreement dadurch behandelt, daß wir bei finiten Verben einen Pfad `<X subj agr>` in den Lexikoneintrag aufgenommen haben. Das heißt, jeder Eintrag eines finiten Verbs muß diese Information enthalten. Eine weitere Ursache für die enorme Komplexität von Lexikoneinträgen ist die Art der Kodierung der Subcat-Information. Im Gegensatz zu Grammatikregeln, von denen es bei einem lexikalischen Ansatz nur noch wenige gibt, gibt es tausende von Lexikoneinträgen. Wenn man eine Grammatik mit lexikalischer Orientierung schreibt, ist es wichtig, sich darüber Gedanken zu machen, wie man Lexikoneinträge kompakt, übersichtlich und wartbar aufschreiben kann.

15.2 Templates

Eine Möglichkeit, die PATR-II bietet, um das Lexikonschreiben zu erleichtern, sind Templates. Ein Template ist ein Makro – es gibt einer Menge von Gleichungen, die einen DAG definieren, einen Namen. Das ist besonders dann nützlich, wenn die gleiche Menge von Gleichungen in vielen verschiedenen Einträgen vorkommt. Man schreibt dann einfach den Templatenamen, anstatt alle Gleichungen nocheinmal hinzuschreiben. Einige Beispiele sind:

```
let verb be X such_that
  <X cat> = v

let sing3 be X such_that
```

```
  <X head subject head agr num> = sg
  <X head subject head agr per> = 3
  <X head vform> = fin

let transitive be X such_that
  <X subcat first cat> = np
  <X subcat rest first cat> = np
  <X subcat rest first> = <X head subject>
  <X subcat rest rest first> = nil
```

Der Lexikoneintrag für ein bestimmtes transitives Verb in der 3. Person Singular wäre dann:

```
X -> chases
  X = verb
  X = transitive
  X = sing3
```

Der Interpreter expandiert die Templates in die entsprechenden Gleichungen und berechnet den Graphen indem er die Gleichungen löst. Templates können überall in Merkmalstrukturen auftauchen, wo eine Merkmalstruktur definiert wird. Rekursive Templates sind allerdings verboten, da die Expansion terminieren muß. Ein Beispiel für ein eingebettetes Template folgt. Die beiden verwendeten `X` haben verschiedenen Skopus.

```
let dummy_NP be X such_that
  <X cat> = np
  <X nform> = it

X -> seems
  <X head subject> = dummy_NP
```

15.3 Redundanzregeln

Ein noch weiter ausgebautes System wurde von Graham Ritchie u.a. in Edinburgh und Cambridge entwickelt. Wie in PATR gibt es Templates, die Aliase genannt werden. Außerdem gibt es noch drei weitere Arten lexikalischer Regeln.

Es gibt Regeln, die sich nur auf eine Merkmalstruktur beziehen. Diese Regeln beschreiben den Sachverhalt, daß bestimmte Merkmal-Wert-Paare nur in Kombination mit anderen Merkmal-Wert-Paaren auftauchen können. Deshalb kann

man aus der Anwesenheit eines Paares auf die Anwesenheit eines anderen Paares schließen. Andererseits gilt, daß eine Merkmalstruktur, in der nur eines der beiden Paare vorhanden ist, keine wohlgeformte Merkmalstruktur ist, die ein linguistisches Objekt beschreibt. In Ritchies System wird zwischen *Vervollständigungsregeln* und *Konsistenzchecks* unterschieden. Man kann beide Komponenten unter dem Begriff *Redundanzregeln* zusammenfassen.

Eine Sprache, die zwischen Menschen, Tieren und unbewegten Einheiten unterscheidet, könnte die folgenden Redundanzregeln benutzen:

```
forall X if <X human> = yes then <X animate> = yes
forall X if <X animate> = no then <X human> = no
```

Wenn man weiß, daß die Merkmale `human` und `animate` vom Typ *boolean* sind, kann man den Wert des einen aus dem Wert des anderen Merkmals ableiten.

In Grammatikregeln, die sich auf Verben beziehen, wird oft verlangt, daß das Verb finit ist, ohne daß spezifiziert ist, ob das Verb in der present- oder in der past-Form ist. Andererseits gibt es verschiedene Endungen für die present- und past-Formen. Bei den Lexikoneinträgen dieser Morpheme muß die Finitheit nicht spezifiziert werden. Stattdessen hat man eine Regel der Form:

```
forall X if <X tense> = _ then <X finite> = yes
```

Damit diese Regel sinnvoll ist, muß es möglich sein, daß eine Merkmalstruktur kein tense-Merkmal hat. Das würde das Fehlschlagen der Pfadberechnung für `<X tense>` ermöglichen. Die einfachste Art, das zu erreichen, ist die Einführung eines Typ-Systems für Merkmalstrukturen.

In der HPSG gibt es solche Regeln auch. Sie werden mit der bereits eingeführten Implikation ($\Rightarrow$) ausgedrückt und Prinzipien genannt.

Außer den oben aufgeführten Regeln gibt es noch Regeln für Default-Spezifikationen. Wie bei den Redundanzregeln werden von den Default-Regeln gewisse Merkmal-Wert-Paare zu einer Merkmalstruktur hinzugefügt, falls nicht bereits Werte vorhanden sind. Im Gegensatz zu den Redundanzregeln gibt es aber keinen Fehler, wenn das Hinzufügen eines neuen Merkmal-Wert-Paares fehlschlägt. Das ist der Fall, wenn der Default-Wert überschrieben wurde.

Eine typische Default-Regel ist:

```
forall X if <X cat> = noun default <X nform> = norm
```

15.4 Multiplikationsregeln / Lexikalische Regeln

Ein anderer Regeltyp in Ritchies System wird *Multiplikationsregel* genannt. Diese Regeln beziehen sich nicht auf eine einzige Merkmalstruktur. Sie legen fest, wie

ein bestimmter Eintrag aus einem anderen Eintrag konstruiert werden kann. Diese Regeln entsprechen den lexikalischen Regeln der HPSG. In PATR-II sehen Regeln dieser Art wie folgt aus:

```
define diTrans as In => Out such_that
<Out cat> = <In cat>
<Out head> = <In head>
<Out subcat first sem> = <In subcat rest first sem>
<Out subcat first cat> = np
<Out subcat rest first sem> = <In subcat first sem>
<Out subcat rest first cat> = np
<Out subcat rest rest> = <In subcat rest rest>
```

`In` und `Out` bezeichnen dabei die Ein- bzw. Ausgabe-DAGs der lexikalischen Regel. Im Beispiel ist `In` ein ditransitives Verb, das ein direktes Objekt und ein indirektes Objekt (eine *to* Präpositionalphrase wie in *give a bone to a dog*) verlangt, und `Out` ist eine Form, die zwei Nominalphrasen als Komplement verlangt, wie in *gives a dog a bone*. Alternativ könnte man dieselbe Information repräsentieren, indem man `diTrans` als zwei Templates aufschreibt, die die entsprechenden Subcat-Listen erzeugen, und die disjunktiv interpretiert werden.

```
define passiv as In => Out such_that
<Out head subj> = <In subcat first>
<Out subcat> = <In subcat rest>
<Out head vform> pastp
```

Diese Regel für die Passivbildung setzt voraus, daß das Subjekt nicht in der Subcat-Liste ist. Die Regel nimmt das erste Element der Subcat-Liste (das direkte Objekt) und macht es zum Subjekt des Ausgabezeichens.

Man beachte, daß Systeme, die lexikalische Regel wie die oben aufgeführten benutzen, im Gegensatz zu PATR-II Systemen, die nur Templates benutzen, nicht deklarativ sind. Das Ergebnis der Anwendung einer Default-Regel auf einen lexikalischen Eintrag, das ein Merkmal instantiiert, hängt davon ab, ob sie vor oder nach einer Multiplikationsregel angewendet wird, die den betreffenden Wert spezifiziert. Multiplikationsregeln terminieren eventuell nicht, da die Ausgabe einer Multiplikationsregel Eingabe derselben oder einer anderen Multiplikationsregel sein kann. Zum Beispiel erzeugt die `diTrans`-Regel eine neue Merkmalstuktur, die die Eingabe für die `passiv`-Regel sein kann. Die Ausgabe der `passiv`-Regel darf nicht wieder Eingabe der `passiv`-Regel werden.

Kontrollfragen

1. Welche Erweiterung von PATR-II wurde durch den Wunsch, das Lexikon zu vereinfachen, motiviert?
2. Welche zwei Arten von Redundanzregeln wurden von Ritchie vorgeschlagen?
3. Geben Sie ein Beispiel dafür, wofür man Multiplikationsregeln benutzen kann.

Übungsaufgaben

1. * Wie würden Sie Templates in einer PATR-II-Implementation einer Kategorialgrammatik benutzen?
2. * Die PATR-II-Implementation, die Sie in der nächsten Hausaufgabe schreiben werden, wird mit Ihrem PATR-II-Kategorialgrammatikfragment nicht arbeiten. Warum ist das so? Was müßte man tun, um eine funktionierende Version zu erhalten?

Hausaufgabe

9. Stellen Sie eine vollständige PATR-II-Implementation zusammen. Folgendes ist zu tun:

- Stellen Sie den Code aus dem vorigen Kapiteln zusammen, und deklarieren Sie die entsprechenden Operatoren.
- Passen Sie einen Parser so an, daß er das neue Grammatikformat verarbeiten kann. Benutzen Sie den Top-Down-Parser. Es muß der erste Aufruf des Ziels `recognise` geändert werden.
- Probieren Sie den Interpreter an einer kleinen Grammatik aus. Schreiben Sie eine kleine Grammatik im PATR-II-Format, die die folgenden Sätze akzeptiert:

  (15.1) a. John sees the monkey.
  b. The monkey sings.

  aber die folgenden Sätze zurückweist:

(15.2) a. * The monkey see the table.

b. * The monkey sings the table.

Die Grammatik muß nichts Besonderes sein, sie soll nur zum Test des Interpreters dienen und Merkmale auf in sinnvoller Weise benutzen.

Für diesen Teil der Aufgabe kann man höchstens eine 2 bekommen.

Eine Eins ist erreichbar, wenn die folgende Teilaufgabe ebenfalls gelöst wird:

- Erweitern Sie Ihre PATR-II-Implementation um einen Template-Mechanismus zur Spezifikation von Lexikoneinträgen, so wie das in diesem Kapitel beschrieben wurde. Es folgt ein Vorschlag für die Schreibweise:

  ```
  template(vtr,X) :-
    X:cat <=> v,
    X:subcat:first <=> np,
    X:subcat:rest:first <=> nil.
  ```

  Ein Lexikoneintrag könnte dann die folgende Form haben:

  ```
  X ---> [chases] :-
   X <=> vtr,
   X:num <=> sing.
  ```

  Um die Templates korrekt interpretieren zu können, muß man noch eine Klausel zu den Klauseln des `denotes`-Prädikates hinzufügen.

Kapitel 16

Parsen mit Unifikationsgrammatiken

16.1 Möglichkeiten für das Parsen mit UGs

Wir haben Algorithmen für Chart-Parser vorgestellt, die kontextfreie Grammatiken verarbeiten können. Aber seit dem Kapitel 12 haben wir Grammatiken mit komplexeren Kategorien beschrieben, für die die Parse-Algorithmen in der vorgestellten Form nicht mehr anwendbar sind. Was muß man beachten, wenn man einen Parser für solche Grammatiken schreiben will? Es gibt für das Parsen von Grammatiken mit komplexen Kategorien drei Möglichkeiten:

1. *Umwandlung in eine kontextfreie Phrasenstrukturgrammatik*: Wie wir gesehen haben, kann man die Benutzung von Merkmalen in Regeln als Abkürzung für eine Menge von Regeln verstehen. Wir müssen also nur unsere Regelmenge expandieren und können dann mit den Standardtechniken parsen. Das funktioniert aber nur mit kleinen Spielzeuggrammatiken, weil die expandierten Grammatiken für realistische Ausgangsgrammatiken riesig werden würden. Allein die Größe der Regelmenge, die für die Koordination gebraucht wird, wird mindestens so groß sein, wie die Kategoriemenge. Die Größe einer Kategoriemenge, die n binäre Merkmale verwendet, ist 3^n, wenn man den Fall zuläßt, daß Merkmale uninstantiiert sein können. Hat man zum Beispiel eine Grammatik mit nur 24 binären Merkmalen, so bekommt man $3^{24} = 282.429.536.481$ Kategorien. Die Größe der Grammatik beinflußt direkt den benötigten Speicherplatz und die beim Parsen benötigte Zeit.

2. *Die Verwendung eines kontextfreien Rückgrates* (*CF-PSG backbone*): Wenn es ein Merkmal oder eine Menge von Merkmalen gibt, die in jeder Grammatikregel spezifiziert sind, kann man den Parser mit diesen Merkmalen

arbeiten lassen und dabei die anderen Merkmale vorerst unberücksichtigt lassen. Zum Beispiel spezifizieren manche Grammatiken immer einen Wert für das *CAT*-Merkmal, so daß man nur unter Berücksichtigung des *CAT*-Merkmals parsen könnte. Natürlich besteht die Gefahr, wenn man nur einige Merkmale berücksichtigt, daß der Parser Zeit damit verschwendet, Hypothesen und mögliche Analysen zu bilden, die die vollständige Grammatik ausschließen würde. Die falschen Analysen, die der Parser liefert, müssen durch einen Filterprozeß ausgeschlossen werden. Der Filterprozeß muß jede gelieferte Analyse auf Gültigkeit untersuchen und die vernachlässigten Merkmale instantiieren. Das Parsen mit einem kontextfreien Rückgrat hat den entscheidenen Nachteil, daß ein Filterprozeß gebraucht wird und daß der Suchraum erweitert wird, wenn man Merkmale wegläßt. Außerdem ist der Ansatz nicht anwendbar, wenn es kein Merkmal gibt, das immer einen vollständig instantiierten Wert hat.

3. *Verwendung besonderer Mechanismen im Parser*: Das Parse-Problem ist im wesentlichen unabhängig davon, ob man mit Merkmalstrukturen oder mit atomaren Kategorien parst. Das Einzige was man sicherstellen muß, ist, daß die grundlegenden Operationen des Parsers (das Testen auf Merkmalsgleichheit, die Suche nach Regeln, usw.) so definiert werden, daß sie die andere Datenstruktur der Kategorien akzeptieren.

16.2 Parsen mit komplexen Kategorien

Welche Unterschiede gibt es zwischen dem Parsen mit atomaren und dem Parsen mit komplexen Kategorien? Erstens haben die Regeln und die Lexikoneinträge ein anderes Format. Zweitens müssen im Parser an den entsprechenden Stellen Merkmalstrukturen verwendet werden. Statt Regeln mit Punkten mit atomaren Kategorien müssen wir Regeln mit Punkten mit Merkmalstrukturen in die Chart eintragen. Es ist auch nicht sinnvoll, die Relationen `left_sister` und `right_sister` im Voraus zu berechnen, da es sehr viele mögliche Regeln mit Punkt geben kann.

Die Kategorien in einer Kante sind im allgemeinen nicht vollständig spezifiziert, und wir müssen – wie bei den normalen Regeln auch – eine Kante als Abkürzung für alle möglichen Instantiierungen der Kante auffassen.

Wie sieht die Fundamentalregel für komplexe Kategorien aus? Das Ergebnis der Anwendung der Fundamentalregel ist eine neue Kante, die der alten aktiven Kante ähnelt, nur daß:

- die Kante weiter instantiiert ist. An der Stelle der ersten gesuchten Kategorie steht das Unifikat dieser Kategorie mit der gefundenen Kategorie, und

- der Punkt wurde eine Stelle weiter nach rechts bewegt.

Die Abbildung 16.1 zeigt das in Graph-Form. Die aktive Kante entspricht der folgenden Grammatikregel:

```
X0 -> X1 X2
<X0 cat> = s
<X1 cat> = np
<X2 cat> = vp
<X0 mood> = <X2 mood>
<X1 num> = <X2 num>
```

Dabei wurde die Nominalphrase schon gefunden. Das Numerus-Merkmal der gefundenen Nominalphrase war nicht instantiiert. Die aktive Kante repräsentiert den Fakt, daß eine Verbphrase gefunden werden muß, damit ein vollständiger Satz erkannt wird. Die inaktive Kante repräsentiert eine vollständige VP. Wenn sich die beiden Kanten an den richtigen Positionen im String befinden, können sie unter Anwendung der Fundamentalregel zu der neuen Kante kombiniert werden. Man beachte, daß die Kategorie der vollständigen Verbphrase mit der Kategorie der Verbphrase in der aktiven Kante unifiziert wird und dabei das Numerus-Merkmal der Nominalphrase und das Mood-Merkmal des Satzes instantiiert wird.

Wie funktioniert die Vorhersage? Die Abbildung 16.2 zeigt einen relativ einfachen Fall. Wir haben eine aktive Kante, die eine Singular-Nominalphrase braucht. Beim Top-Down-Parsen würde die Vorhersage angewendet. Für jede Regel der Grammatik, deren linke Seite mit der gesuchten Kategorie unifiziert, wird eine aktive Kante in die Chart eingetragen, die den Punkt ganz am Anfang hat, also noch die gesamte rechte Seite sucht. Die Grammatikregel, die der in der Abbildung entspricht, hat die Form:

```
X0 -> X1 X2
<X0 cat> = np
<X1 cat> = det
<X2 cat> = noun
<X0 num> = <X1 num>
<X1 num> = <X2 num>
```

Das Resultat der Vorhersage ist eine aktive Kante, die dieser Regel entspricht und den Punkt genau hinter dem `->` hat. Das Numerus-Merkmal wurde durch die Unifikation instantiiert.

16.3 Kopieren und *structure sharing*

Bei der Erklärung der Arbeitsweise der Fundamentalregel haben wir ein wichtiges Detail vernachlässigt. In einem Chart-Parser muß jede aktive und inaktive

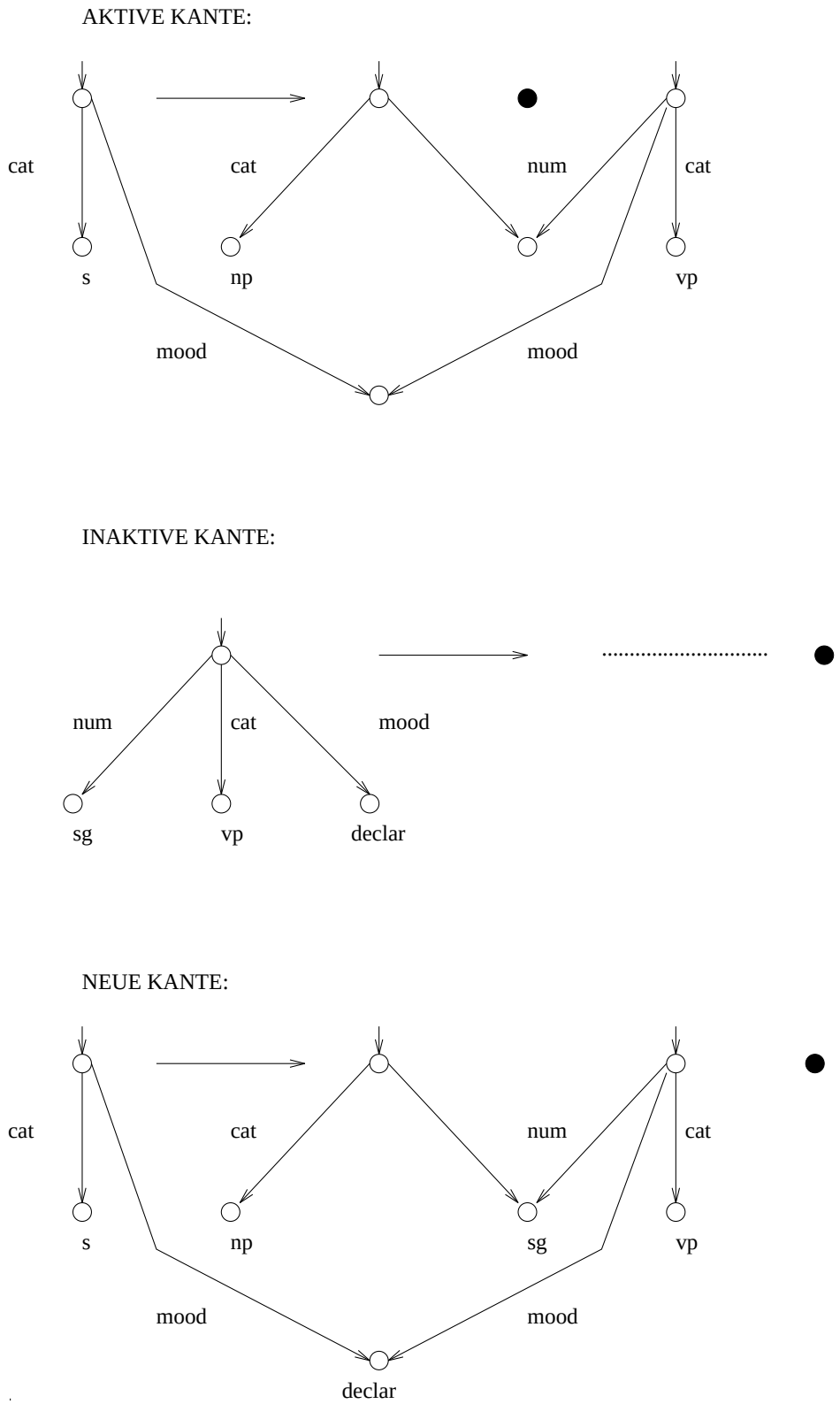

Abbildung 16.1: Parsen mit Unifikationsgrammatiken

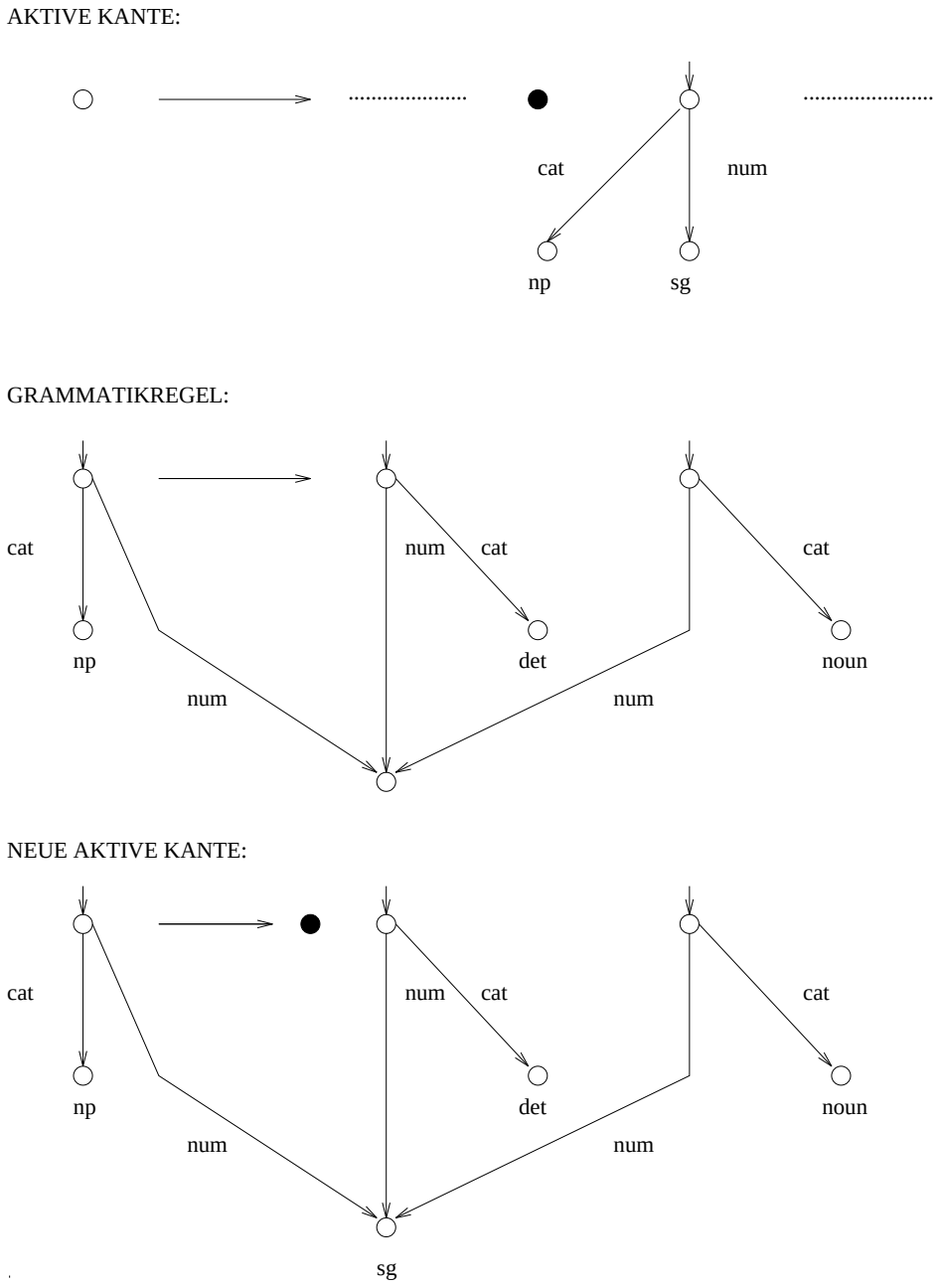

Abbildung 16.2: Die Vorhersage

Kante solange unverändert zur Kombination mit anderen Kanten zur Verfügung stehen bis der Parseprozeß beendet ist. Das soll anhand des folgenden Beispiels verdeutlicht werden:

(16.1) An employee with two supervisors can apply to the area head.

Wenn beim Parsen Numerus-Merkmale für Nominal- und Verbphrasen berücksichtigt werden, dann muß das Numerus-Merkmal der Verbphrase *can apply to the area head* unspezifiziert sein, da diese Verbphrase sowohl im Plural als auch im Singular auftreten kann. Ein Bottom-Up-Parser wird diese Verbphrase mit der Nominalphrase *two supervisors* zu einem vollständigen Satz kombinieren. Die entsprechende Anwendung der Fundamentalregel ergibt eine Kante, in der die NP und die VP den Numerus-Wert plural haben, da Subjekt und Verb im Numerus-Wert übereinstimmen müssen. Es wäre falsch, wenn die Information über den Numerus-Wert der Verbphrase irgendwo anders als in der neu entstandenen Verbphrase auftauchen würde, da die Kante für die VP auch mit der Nominalphrase *an employee with two supervisors* kombiniert werden muß. Bei dieser Kombination muß das Numerus-Merkmal der VP mit Singular instantiiert werden, d.h. daß bei der Kombination der inaktiven VP-Kante die Kante selbst nicht verändert werden darf. Da bei einer Unifikation mit den entsprechenden Kategorien in den aktiven Kanten aber eventuell eine Veränderung, d.h. eine weitere Instantiierung von Merkmalen stattfindet, muß man eine Kopie einer Kante zur Kombination benutzen.[1] In der Kopie haben wir immer frische Variablen, die nicht mit anderen Variablen, die an anderer Stelle benutzt werden, zusammenhängen. Das Anfertigen von Kopien ist jedoch sehr aufwendig. Deshalb wurden Vorgehensweisen entwickelt, die *structure sharing* verwenden und die den Aufwand beim Kopieren reduzieren. Die neue Kante, die durch Anwendung der Fundamentalregel entsteht, ist der aktiven Kante, die bei der Kombination verwendet wurde, meist sehr ähnlich. Man kann also Implementationen schreiben, die die aktive Kante nicht kopieren und mit der Kopie weiterarbeiten, sondern stattdessen nur die Veränderungen der aktiven Kante abspeichern. In einer Prolog-Implementation könnte man das tun, indem man Substitutionen benutzt, um Veränderungen zu repräsentieren. Eine Kante besteht dann aus einem Tripel: der Grammatikregel, einer Nummer für die Position des Punktes und einer Substitution. Solch eine Repräsentation macht die Kanten in der Chart unlesbar. Andererseits wird der Aufwand beim Erzeugen neuer Kanten verringert.

[1] Das ist auch bei Chart-Parsern, die normale DCGs parsen der Fall. Bei einem Prolog-Parser, der die Prolog-Datenbasis benutzt, wird automatisch beim Konsultieren der Datenbasis eine Kopie angefertigt. Die Datenbasis wird durch diesen Zugriff nicht verändert. Verändert wird sie nur beim Hinzufügen neuer Kanten durch `assert`.

16.4 Terminierung

Der wesentliche Vorteil des Chart-Parsens gegenüber anderen Parse-Techniken liegt in der Vermeidung von unnötiger Arbeit. Beim Parsen mit atomaren Kategorien ist der Test, ob Kanten bereits in der Chart sind, trivial, weil es entweder eine Kante mit den entsprechenden Kategorien in der Chart gibt oder nicht. Dadurch, daß man überprüft, ob eine Kante bereits in der Chart ist, bevor man mit ihr weiterarbeitet, stellt man die Terminierung des Parseprozesses sicher, da es für eine kontextfreie Grammatik nur endlich viele Kanten gibt, die zu einem bestimmten Eingabestring gehören. Wenn die Kategorien eine komplexe Struktur haben, gibt es mehrere mögliche Relationen zwischen einer Kante und einer Kante, die schon in der Chart ist. Der verwendete Duplikationstest ist deswegen etwas komplizierter. Wenn wir bereits eine aktive Kante in der Chart haben, die nach einer inaktiven Kante mit variablem Numerus-Merkmal sucht, ist es nicht sinnvoll, der Chart eine ähnliche aktive Kante hinzuzufügen, die sich nur dadurch von der anderen unterscheidet, daß sie nach einer Plural-Nominalphrase sucht. Man könnte annehmen, daß der entsprechende Test ein Unifikationstest ist, d.h. daß es ausreichen würde, wenn man eine Kante in die Chart eintragen will, daß man vorher testet, ob es bereits eine Kante in der Chart gibt, die mit der einzutragenden Kanten unifiziert. Würde man nur auf Unifikation testen, so wäre der Parser nicht vollständig, da es Fälle gibt, in denen nicht alle Analysen geliefert werden. Wenn wir zum Beispiel schon eine Plural-Nominalphrase in der Chart haben, so wäre es trotzdem sinnvoll, eine Nominalphrase mit variablem Numerus-Wert in die Chart aufzunehmen. Statt eines Unifikationstests muß man ein Subsumptionstest durchführen, bevor eine Kante in die Chart eingetragen wird. Das heißt, eine Kante wird nur dann in die Chart eingetragen, wenn es in der Chart keine Kante gibt, die allgemeiner als die einzutragende Kante ist.

Der Subsumptionstest ist eine Möglichkeit, unendliche Schleifen zu verhindern, die dadurch entstehen können, daß verschiedene unterschiedlich instantiierte Kanten in die Chart eingetragen werden. Es ist aber trotzdem immernoch möglich, daß unendlich viele Kanten in die Chart eingetragen werden, wobei keine von einer anderen subsumiert wird. Man kann sich das zum Beispiel am Arbeiten der Vorhersageregel für eine Grammatikregel der Kategorialgrammatik klarmachen. Die Regel

```
X -> X/Y Y
```

hat in PATR-II die Form:

```
X0 -> X1 X2
<X1 res> = <X0>
<X1 arg> = <X2>
```

Wenn eine Kategorie *s* gesucht wird, erzeugt die Vorhersageregel eine aktive Kante der Form *s/Y* für ein beliebiges Y. Die Vorhersageregel wird dann auf diese Kante angewendet und erzeugt eine Kante der Form *(s/Y)/Z* usw. usf. bis in alle Ewigkeit. Die Abbildung 16.3 zeigt, wie das mit den PATR-II-Regeln aussehen würde. Man beachte, daß keine der neu kreierten Kanten eine bereits vorhandene subsumiert.

Lesetips

Die Idee der *patial execution* stammt von Cooper (1990). Die weitere Reduktion der „typisierten“ DAGs wurde im Rahmen einer HPSG-Implementation Müller (1996) von mir entwickelt.

Kontrollfragen

1. Was sind die drei Möglichkeiten, einenParser für Unifikationsgrammatiken zu implementieren?
2. Wie sieht eine Regel mit Punkt für eine Unifikationsgrammatik aus?
3. Wo wird die Unifikation in der Fundamentalregel und wo in der Vorhersage benutzt?
4. Warum müssen Chart-Kanten kopiert werden, bevor sie kombiniert werden?
5. Welchen Test muß man durchführen, um herauszufinden, ob eine Kante „bereits in der Chart“ ist?
6. Ist der Chart-Parser aus Kapitel 11 für eine Grammatik, die Regeln der Form `s(Vform) --> n(2,Num), v(2,Vform,Num)` enthält, verwendbar?

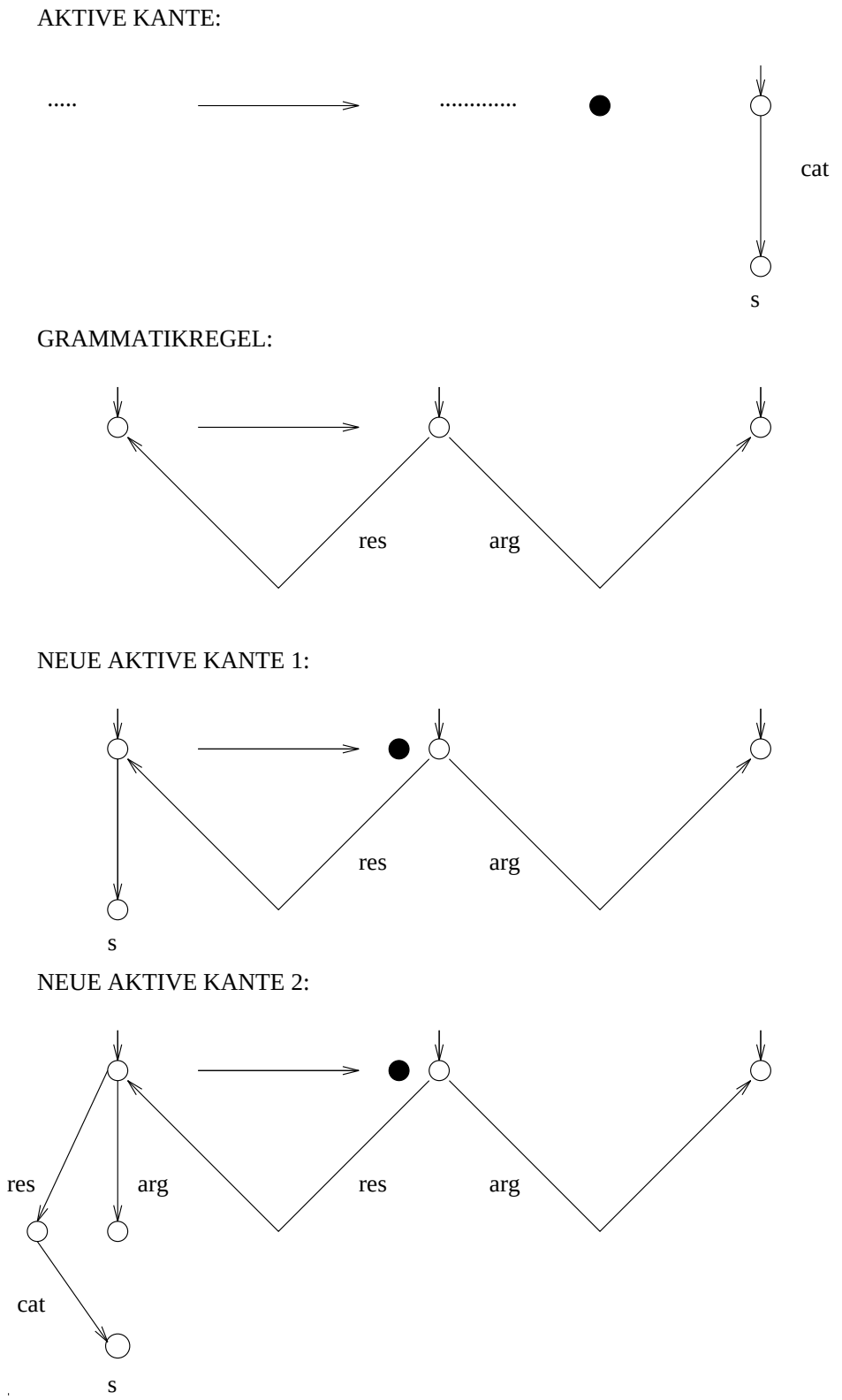

Abbildung 16.3: Nichtterminierende Vorhersage

Kapitel 17

Constraint-Based Grammars

17.1 Prinzipien und Beschränkungen

Alle Grammatikformalismen, die wir bis jetzt behandelt haben, benutzten Regeln (meist Phrasenstrukturregeln). In der Theorie der formalen Sprachen wird eine Grammatik durch eine Regelmenge definiert. Es ist aber sehr schwer natürliche Sprachen nur mit Hilfe solcher Regelsysteme zu beschreiben. Die vollständige Beschreibung von Sprachen durch Phrasenstrukturregeln würde einige tausend, wenn nicht zehntausende, solcher Regeln erfordern. Es wäre schwierig eine Grammatik dieser Größe und Komplexität zu warten und konsistent zu halten. Generalisierungen, die man in Bezug auf Regeln machen kann, gehen beim expliziten Aufschreiben der Regeln verloren. Es ist nicht möglich ein allgemeines Prinzip wie: „Der Wert des Numerus-Merkmals einer Phrase ist identisch mit dem Numerus-Wert ihrer Kopftochter.“ zu formulieren. Diese Information muß in jeder Grammatikregel durch Verwendung gleicher Variablen bzw. *structure sharing* neu kodiert werden.

Eine Phrasenstrukturregel sagt viele Dinge über die Kategorien, die in der Regel vorkommen, aus. Sie beschreibt, welche Kategorien von der linken Regelseite unmittelbar dominiert werden, und sie beschreibt die Reihenfolge der Konstituenten auf der rechten Regelseite.

17.2 *Generalized Phrase Structure Grammar*

Wie in Kapitel 5.7 schon erwähnt wurde, sind die Grammatiken aus der Chomsky-Hierarchie zur Beschreibung von Sprachen mit freier Wortstellung weniger gut geeignet, da man sehr viele Regeln brauchte, um alle Permutationen abzudecken. Statt dessen verwendet man Grammatiken, deren Regeln eine andere Semantik haben: Die Konstituenten einer rechten Regelseite müssen vorhanden sein, wenn

eine linke Seite erkannt werden soll. Über die Reihenfolge der Konstituenten auf der rechten Seite sagt die Regel nichts aus. Dafür gibt es extra Regeln, die unabhängig von den Ersetzungsregeln gelten. Ein solcher Grammatikformalismus ist die *Generalized Phrase Structure Grammar* (GPSG). GPSG entstand aus der Arbeit von Gerald Gazdar 1978 (siehe (**?**)). Im folgenden werde ich einige syntaktische Komponenten der GPSG vorstellen. Wie die Syntax mit der Semantik verbunden ist, kann man in Uszkoreit (1987) nachlesen.

17.2.1 Die Objektgrammatik

Die Regeln der GPSG sind Tripel der Form $< n, r, t >$. Dabei ist n eine natürliche Zahl, die dieselbe Funktion wie das Subcat-Merkmal hat. t ist der Semantikteil. Ich werde im folgenden den Semantikteil weglassen.

$$< 3, V^2 \rightarrow V\ N^2 > \tag{17.1}$$

(17.1) ist z.B. eine Regel für ein transitives Verb wie *kennen*.

17.2.2 Die Metagrammatik

Die Meta-Grammatik besteht aus vier verschiedenen Arten von Regeln, die von drei Komponenten genutzt werden, um Schritt für Schritt die Objektgrammatik zu generieren. Die Objektgrammatik entspricht unter bestimmten Voraussetzungen einer kontextfreien Phrasenstrukturgrammatik. Meist wird die Objektgrammatik jedoch nicht wirklich erzeugt. Stattdessen werden die drei Komponeneten in einen Parser integriert, der dann Expansionen nur dann vornimmt, wenn sie nötig sind.

Abbildung 17.1 zeigt das Zusammenwirken der Komponenten und Regeln. Eine Regelform sind die *immediate-dominance-rules* (IDR). Sie haben die Form von CF-PS-Regeln, aber eine andere Bedeutung. Die CF-PS-Regeln sagen sowohl über die unmittelbare Dominanz als auch über die lineare Abfolge der Konstituenten auf der rechten Seite der Regel etwas aus. Im Gegensatz dazu sagen *immediate-dominanz*-Regeln nur darüber etwas aus, daß die Konstituenten auf der rechten Seite der Regel in einem Ableitungsbaum von der Konstituente auf der linken Seite der Regeln dominiert werden. Die Ordnung der Elemente auf der rechten Seite von ID-Regeln ist somit bedeutungslos. Die beiden Regeln in (17.2) sind äquivalent.

$$\begin{array}{l} < 3, V^2 \rightarrow V\ N^2 > \\ < 3, V^2 \rightarrow N^2\ V > \end{array} \tag{17.2}$$

Eine andere Art Regeln sind die Metaregeln. Sie bilden ID-Regeln auf ID-Regeln ab. Metaregeln sind sowohl auf Basis-ID-Regeln als auch auf Regeln, die bereits

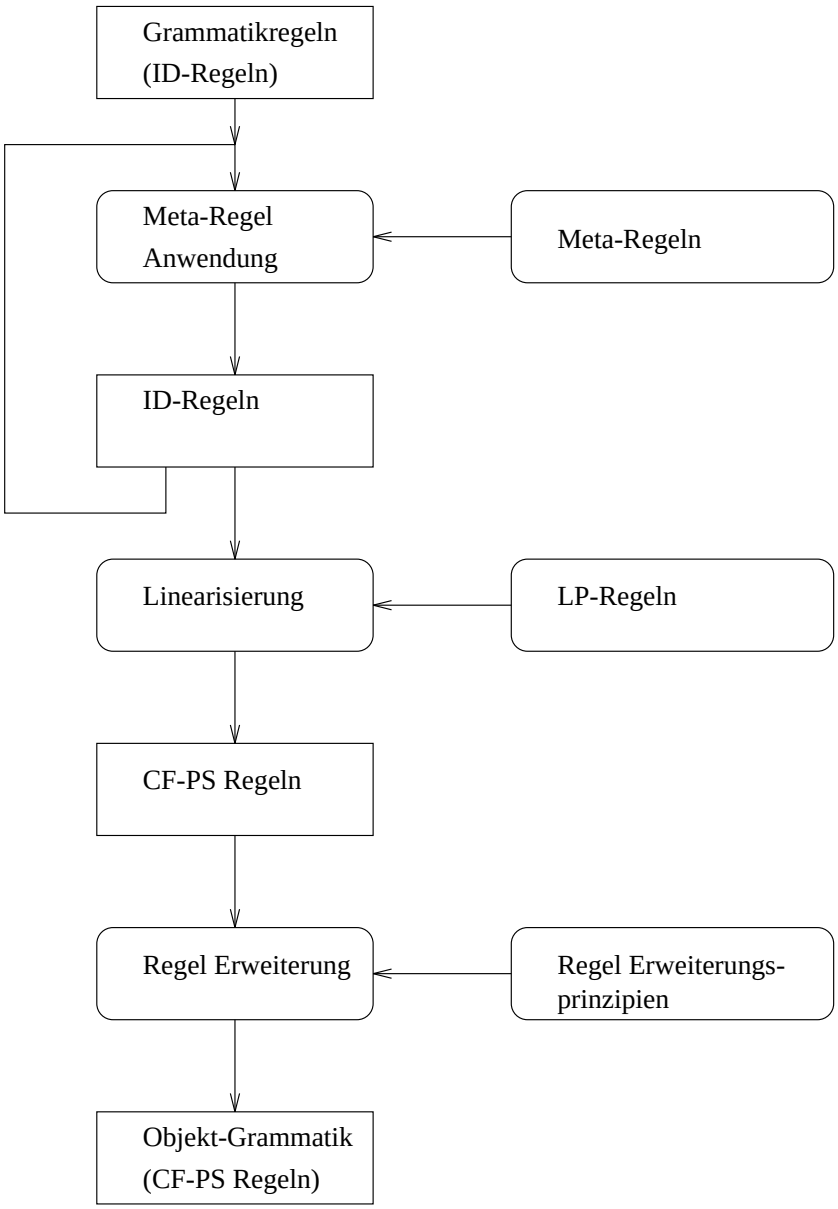

Abbildung 17.1: Das GPSG-System

durch die Anwendung von Metaregeln entstanden sind, anwendbar. Metaregeln kann man als Relation zwischen ID-Regeln verstehen. Man schreibt $A \Rightarrow B$. Das heißt, für jede ID-Regel in der Grammatik, die A entspricht, muß es eine Regel der Form B in der Grammatik geben. Die Nummer zu Beginn einer ID-Regel wird von der Input-Regel zur Output-Regel übertragen, d.h. das Subcat-Merkmal wird übernommen.

Die Metaregel in (17.2) ist eine vereinfachte Regel für die Bildung des Passivs im Englischen.

$$\begin{array}{l} V^2 \rightarrow V, N^2, X \Rightarrow V^2 \quad \rightarrow V, X, (P^2)) \\ \qquad\qquad\qquad\quad +PASS \qquad\quad +by \end{array} \tag{17.3}$$

Die Variable X kann dabei Terminale und Nichtterminale, die durch Kommas getrennt sind, repräsentieren. Variablen, deren Inhalt nicht durch die Grammatik festgelegt werden, werden essenzielle Variablen genannt.

Die Metaregel drückt aus, daß es für jede ID-Regel, die eine Verbphrase in ein Verb, eine Nominalphrase und eventuell andere Konstituenten expandiert, eine ID-Regel in der Grammatik geben muß, die eine Passivverbphrase in ein Verb, in das, was X war, und in eine optionale Präpositionalphrase expandiert.

Wendete man diese Metaregel auf (17.1) an, erhielte man:

$$\begin{array}{l} < 3, V^2 \quad \rightarrow V, (P^2)) > \\ \quad +PASS \quad +by \end{array} \tag{17.4}$$

In diesem Fall wäre X der leere String. Die Subcat-Merkmale sind auf beiden Regelseiten gleich. In der Ableitung einer ID-Regel können verschiedene Metaregeln zur Anwendung kommen. Das *Finite Closure Principle* von Thompson verlangt, daß jede Metaregel in der Ableitung einer ID-Regel höchstens einmal vorkommt. Dieses Prinzip verhindert die Ableitung von unendlichen ID-Regel-Mengen.

Eine andere Komponenete fügt die semantische Übersetzung zur Regel hinzu und instantiiert syntaktische Merkmale. ID-Regel-Paare der Form $< n, i >$ werden auf Tripel der Form $< n, i, t >$ abgebildet, wobei n die Subcat-Nummer, i die ID-Regel und t die semantische Übersetzung ist. Die enstehenden Tripel sind vollständig instantiiert. Sie entsprechen den Regeln der Objektgrammatik ohne Angaben über die Konstituentenreihenfolge. Die Abbildung der geordneten Paare auf die Tripel wird durch eine Menge Prinzipien kontrolliert:

Feature Cooccurrence Restrictions (FCR) speziefizieren das gleichzeitige Auftreten von Merkmal-Wert-Kombinationen. $+MASSN \rightarrow +SING$ bedeutet, wenn eine Kategorie den Wert + für MASSN hat, muß sie auch den Wert + für SING haben.

Default Value Assignment (DVA) Wenn es einen Default-Wert für ein Merkmal gibt, so bekommt es diesen Wert, falls kein anderer Wert durch eine ID-Regel zugewiesen wird.

Die *Head Feature Convention* (HFC) sorgt dafür, daß alle Merkmale, die als Kopfmerkmale angegeben wurden, vom Mutterknoten zum Kopf der Phrase kopiert werden. In der Regel (17.4) wurde nur die Mutter mit +PASS markiert, dieser Wert wird automatisch zum Kopf der Phrase kopiert.

Foot Feature Convention (FFC) Zu den Fußmerkmalen zählen, wie schon erwähnt, Gaps. Das Kopieren der Gap-Information von der Mutter zu den Töchterknoten sichert die FFC.

Das *Control Agreement Principle* (CAP) ist für das Kopieren von Agreementmerkmalen zuständig. Im Deutschen gibt es zum Beispiel Agreement zwischen Determinatoren und Nomina in Nominalphrasen und zwischen Subjekt und finitem Verb in Sätzen.

Die letzte Komponente der Metagrammatik bildet die IDR-Tripel auf Regeln der Objektgrammatik ab, deren CF-PS-Regeln den *linear precedence*-Regeln entsprechen. LP-Regeln sind Elemente der LP-Relation, einer partiellen Ordnung über den Terminalen und Nichtterminalen. LP-Regeln haben die Form $\alpha < \beta$. Das bedeutet, daß α immer vor β steht, wenn beide in der rechten Seite derselben Regel auftauchen.

17.2.3 Eine GPSG-Beispiel-Grammatik

Die folgende Grammatik ist von Uszkoreit (1987) übernommen.

ID-Regeln

1.	$< 1,$	V^3 +PERF	$\rightarrow V,$	$V^2 >$ +PSP		(*haben*, *sein*)
2.	$< 2,$	V^2 +AUX	$\rightarrow V,$	$V^2 >$ +BSE		(*müssen*, *können*, *dürfen*, ...)
3.	$< 3,$	V^2 +AUX	$\rightarrow V,$	$V^2 >$ +PAS		(*werden*)
4.	$< 4,$	V^2 +AUX +FIN	$\rightarrow V,$	$V^2 >$ +BSE		(*werden*)
5.	$< 5,$	V^2	$\rightarrow V >$			(*kommen*, *laufen*, *schlafen*)
6.	$< 6,$	V^2	$\rightarrow V,$	$N^2 >$ +ACC		(*kennen*, *suchen*, *nehmen*)
7.	$< 7,$	V^2	$\rightarrow V,$	$N^2 >$ +DAT		(*helfen*, *begegnen*, *vertrauen*)
8.	$< 7,$	V^2	$\rightarrow V,$	N^2 +ACC	$N^2 >$ +DAT	(*geben*, *schicken*, *zeigen*)
9.	$< 7,$	V^2	$\rightarrow V,$	$V^3 >$ +daß		(*wissen*, *glauben*, *bezweifeln*)

Metaregeln

A.	V^2	$\rightarrow X$	$\Rightarrow$	V^3	$\rightarrow$	N^2, +NOM	X
B.	V^2 +AUX	$\rightarrow V, V^2$	$\Rightarrow$	V^3	$\rightarrow$	V, V^3	
C.	V^3 -AUX	$\rightarrow X, B$	$\Rightarrow$	V^3/B -AUX	$\rightarrow$	X,t	
D.	V^2	$\rightarrow X$	$\Rightarrow$	V^2 $+\alpha$	$\rightarrow$	X,	SEPPREF $+\alpha$

Feature Cooccurrence Restrictions

I. +AC $\rightarrow$ +MC
II. +MC $\rightarrow$ +FIN

Linear Precedence Rules

a.	V +MC	$<$	X
b.	X	$<$	V -MC
c.	+TOP	$<$	X
d.	X^2	$<$	SEPPREF
e.	+NOM	$<$	+DAT
	+NOM	$<$	+ACC
	+DAT	$<$	+ACC
	-FOC	$<$	+FOC
	+PRN	$<$	-PRN
f.	daß	$<$	V^3

17.3 *Government and Binding*

Der momentan dominante Ansatz in der theoretischen Linguistik ist wahrscheinlich der Ansatz von **?**, *Government and Binding* GB genannt. In der GB wird ebenfalls auf komplexe Phrasenstrukturregeln verzichtet. Statt dessen gibt es verschiedene Repräsentationsebenen und Prinzipien, die etwas darüber aussagen, welche Art Informationen auf welcher Ebene auftauchen dürfen und in welcher Beziehung diese Information zu Information auf anderen Ebenen steht. Die Motivation, die dem zugrunde liegt, ist abstrakte Prinzipien, die für alle Sprachen gelten, und Parameter, die die Sprachen unterscheiden, zu finden. Die GB-Theoretiker befinden sich auf der Suche nach der „Universalgrammatik“. Abbildung 17.2 zeigt die Generierung eines Satzes. Das Wort Generierung wird hier im Sinne der Theorie der formalen Sprachen benutzt. Die X-bar-Regeln der Basiskompo-

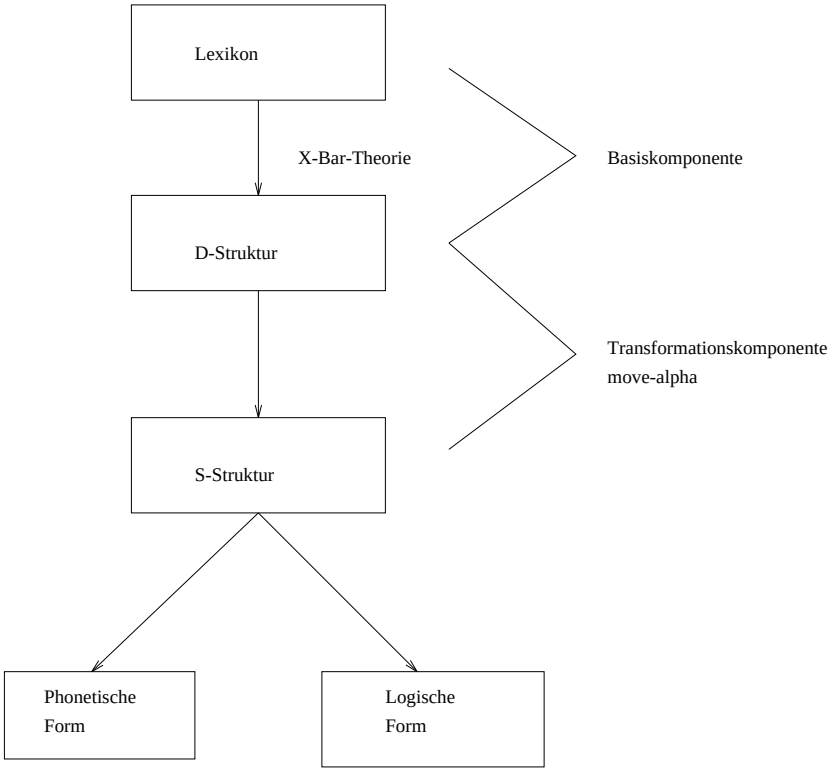

Abbildung 17.2: Das GB-System

nente werden zusammen mit dem Lexikon dazu benutzt, die „D-Structur“ (*deep structure*) zu generieren. Man kann sich die Tiefenstrukturen als Parsebäume vorstellen, die man beim Parsen mit X-bar-Grammatiken erhalten würde.Die Funktion „*move* – α“ wird dazu benutzt, Tiefenstrukturen auf Oberflächenstrukturen abzubilden. So kann z.B. aus der folgenden Tiefenstruktur die darunterstehende

Oberflächenstruktur erzeugt werden:

DS: Karl das Buch der Frau gab
SS: Das Buch gab Karl der Frau
SS: Der Frau gab Karl das Buch
...

Im allgemeinen darf eine Konstituente mehrmals bewegt werden. Die Menge der zwischenzeitlich durch die Konstituente besetzten Positionen wird Kette (*chain*) genannt. *move* – α darf alles überallhin bewegen. Unmögliche Strukturen werden jedoch von anderen GB-Prinzipien ausgeschlossen.

Die Idee der GB-Theorie ist, daß die logische Form (LF) (eine syntaktische Repräsentation, die sehr nah an der Semantik ist) und die phonetische Form (PF) (eine Repräsentation von Klangsequenzen, die man zur Äußerung eines Satzes braucht) von der Oberflächenstruktur abgeleitet werden können.

Die wesentlichen Prinzipien, die bestimmen, welche syntaktischen Strukturen zulässig sind, sind:

X-bar-Theorie: Die Ideen der X-bar-Theorie wurden in Kapitel 5 erklärt.

Theta-Theorie: Im Lexikon wird für jedes Wort eine Menge von Theta-Rollen, die Argumenten in der logischen Repräsentation entsprechen, spezifiziert. Theta-Rollen werden Komplementen von ihren Köpfen zugewiesen. Das wird Theta-Markierung genannt. Jede Theta-Rolle muß genau einmal zugewiesen werden, und jedes Komplement muß Theta-markiert sein. Im Lexikon ist auch spezifiziert, in welcher Form die Theta-Rollen realisiert werden müssen (z.B. als Nominalphrasenobjekt oder als Komplementsatz). Das ähnelt der Behandlung der Subkategoriesierung in vorangegangenen Kapiteln.

Kasustheorie: Nominalphrasen in bestimmten Positionen (z.B. als Komplement von Präpositionen oder als Objekt von transitiven Verben oder als Subjekt von finiten Sätzen) bekommen Kasus zugewiesen. Jede Nominalphrase muß genau einmal Kasus zugewiesen bekommen. Jede Kette muß genau einmal durch eine kasusmarkierende Position gehen. Also:

(17.5) a. It seems John likes Mary.
b. * It seems John to like Mary.
c. John$_i$ seems $__i$ to like Mary.

Die Tiefenstruktur dieses Satzes ist etwas wie

seems [John [likes Mary]]

seems hat eine Theta-Rolle für eine Aussage, nicht aber für einen AGENS. Die Subjektposition kann also durch eine dummy-Nominalphrase gefüllt werden, wie im ersten Fall. Im zweiten Satz ist die Kasustheorie verletzt, da *John* keinen Kasus bekommen hat. Im dritten Satz wurde *John* an die Subjektposition bewegt, wo *John* Kasus erhält. Die Theta-Markierung erhält *John* jedoch immernoch vom Verb *like*.

Bindungstheorie: Die Bindungstheorie beschäftigt sich mit der Bindung von Pronomina und Anaphora. Das heißt, es wird versucht, syntaktische Gesetzmäßigkeiten für die Bindung von Pronomina und Anaphora im Satz zu finden:

(17.6) a. They introduced each other to Bill.

b. * They expected me to introduce each other to Bill.

Eine Kritik der GB-Ansätze findet man in Dalrymple (1993).

Begrenzungstheorie: Die Begrenzungstheorie schränkt die möglichen Bewegungen ein:

(17.7) * Who$_i$ do [you like [the book that John gave to $__i$]]

Diese Bewegung ist nicht zulässig, da sie über einen Knoten in einer „Specifier“-Position gehen würde. Diese Beschränkung wird auch *Subjazensprinzip* genannt.

Kontrolltheorie: Die Kontrolltheorie beschäftigt sich mit Kontrollverben (siehe Übung 6.5).

Rektionstheorie: (*Government theory*) Die Relation zwischen einem lexikalischen Kopf und einem seiner Komplemente ist ein Beispiel für Rektion. Mit Hilfe des Rektionsbegriffes kann man zum Beispiel Bedingungen dafür formulieren, an welchen Stellen Traces auftreten können.

Das war eine sehr kurze Erwähnung einiger Prinzipien und Teiltheorien einer sehr komplexen Theorie.

Die GB ist nicht sehr gut implementierbar. Die Schwierigkeit, die Theorie in allen Belangen vollständig zu formalisieren und zu verstehen und die Schwierigkeit, entsprechende Parser zu entwickeln, die nicht auf Phrasenstrukturregeln basieren, dürften die Hauptgründe dafür sein, daß es wenige GB-Parser gibt.

17.4 *Head-Driven Phrase Structure Grammar*

Pollard und Sag (1987) haben die *Head-Driven Phrase Structure Grammar* (HPSG) entwickelt. Die HPSG ist ein Beispiel für die Kombination von Ideen aus den verschiedensten Theorien:

- Generalized Phrase Structure Grammar (GPSG)
- Categorial Grammar (CG)
- Government and Binding (GB)
- Dependency Grammar (DG)
- Situation theory

Gegenüber anderen Theorien zeichnet sich die HPSG dadurch aus, daß Syntax und Semantik mit demselben Formalismus – den Merkmalstrukturen – beschrieben werden.

Die HPSG setzt sich aus Prinzipien, Regeln, Sorten und Lexikoneinträgen zusammen. Es wird behauptet, daß es universelle, also für alle Sprachen gültige, Prinzipien gibt. Die Theorie der Universalgrammatik ist die Unifikation aller Prinzipien:

$$UG = P_1 \wedge P_2 \wedge \ldots \wedge P_n \tag{17.8}$$

wobei $P_1, P_2, \ldots P_n$ die vollständige Liste der universellen Prinzipien ist. Des weiteren gibt es sprachspezifische Prinzipien $P_{n+1}, \ldots, P_{n+m}$, die für Zeichen des Deutschen gelten müssen. Beispiele sind das Subkategorisierungsprinzip und das Kasusprinzip. Das Subkategorisierungsprinzip ist ein universelles Prinzip, das Kasusprinzip ist ein sprachspezifisches. Außerdem gehört zu einer HPSG eine endliche Menge lexikalischer Zeichen und eine endliche Menge von Dominanzschemata. Ein solches Schema ist ein teilweise spezifiziertes phrasales Zeichen, das Informationen darüber enthält, wie in Sprachen aus kleineren größere Zeichen gebildet werden können. Die Disjunktion all dieser Schemata bildet das *Immediate Dominance Principle* (IDP). Das IDP ist ebenfalls ein universelles Prinzip. Will man mit der HPSG eine bestimmte Sprache beschreiben, muß man die für diese Sprache zutreffenden Dominanzschemata aus der Disjunktion auswählen. Das heißt, das universelle Prinzip wird weiter spezifiziert. Seien $P'_1, \ldots P'_n$ die für Deutsch angepaßten universellen Prinzipien, $P_{n+1}, \ldots, P_{n+m}$ die sprachspezifischen Prinzipien des Deutschen und sei $L_1, \ldots, L_p$ die Menge aller deutschen lexikalischen Zeichen, dann ist

$$Deutsch = P'_1 \wedge \ldots \wedge P'_n \wedge P_{n+1} \wedge \ldots \wedge P_{n+m} \wedge (L_1 \vee \ldots \vee L_p) \tag{17.9}$$

die Theorie des Deutschen. Das heißt, ein Objekt ist ein deutsches Zeichen gdw. es alle universellen und alle deutschen Prinzipien erfüllt und entweder ein deutsches lexikalisches Zeichen oder ein deutsches phrasales Zeichen ist. Phrasale Zeichen müssen durch ein Dominanzschema gerechtfertigt sein. Dominanzschemata entsprechen den X-Bar-Regeln, die wir in Kapitel 5 bereits kennengelernt haben.

17.4.1 Prinzipien – zwei Beispiele

Prinzipien werden in der HPSG in Form von Implikationen (siehe Kapitel 13.4.6) ausgedrückt. Wenn eine Merkmalstruktur mit der linken Seite der Implikation unifiziert, so muß sie auch mit der rechten Seite unifizieren. Ansonsten ist ein Prinzip der HPSG verletzt, und die betreffende Merkmalstruktur ist kein Zeichen der Sprache, für die das Prinzip formuliert wurde. Ein Beispiel ist das folgenden Prinzip. Es besagt, daß die Kopfmerkmale, die unter dem Pfad SYNSEM|LOC|-CAT|HEAD zusammengefaßt sind, bei Kopftochter und Mutter gleich sind. Das Prinzip gilt nur für Zeichen eines bestimmten Types – Zeichen des Typs *headed-structure*, Zeichen also, die auch wirklich einen Kopf haben. Auf den genauen Aufbau der Merkalstrukturen soll hier nicht weiter eingegangen werden. Der interessierte Leser sei auf Müller (1999) verwiesen.

Prinzip 1 (Kopfmerkmalprinzip (*Head Feature Principle*))

$$\begin{bmatrix}\text{DTRS} & \begin{bmatrix}\textit{headed-structure}\end{bmatrix}\end{bmatrix} \Rightarrow$$

$$\begin{bmatrix}\text{SYNSEM|LOC|CAT|HEAD} & \boxed{1} \\ \\ \text{DTRS|HEAD-DTR|SYNSEM|LOC|CAT|HEAD} & \boxed{1}\end{bmatrix}$$

Wie schon erwähnt, werden die Komplemente eines Kopfes in der Subcat-Liste aufgezählt. Die Sättigung von Komplementen wird durch das Subcat-Prinzip beschrieben. Es entspricht in etwa den Regeln der Kategorialgrammatik:

$$\begin{aligned} X &= X/Y * Y \\ X &= Y * X\backslash Y \end{aligned} \tag{17.10}$$

Im Gegensatz zur Kategorialgrammatik kann die Sättigung von Komplementen eines Kopfes in beliebige Richtung erfolgen. Auch ist es möglich, alle Komplemente auf einmal zu sättigen.

Prinzip 2 (Subkategorisierungsprinzip (*Subcat Principle*))

$$\begin{bmatrix}\text{DTRS} & \begin{bmatrix}\textit{headed-structure}\end{bmatrix}\end{bmatrix} \Rightarrow$$

$$\begin{bmatrix}\text{SYNSEM} & \begin{bmatrix}\text{LOC|CAT|SUBCAT} & \boxed{1}\end{bmatrix} \\ \\ \text{DTRS} & \begin{bmatrix}\text{HEAD-DTR} & \begin{bmatrix}\text{SYNSEM|LOC|CAT|SUBCAT} & \boxed{1} \oplus \boxed{2}\end{bmatrix} \\ \\ \text{COMP-DTRS} & \boxed{2}\end{bmatrix}\end{bmatrix}$$

Natürlich versucht man mit der HPSG dieselben Phänomene wie mit der GB-Theorie zu beschreiben. Es gibt deshalb genauso Bindungs- und Kasusprinzipien, Kommandierungsverhältnisse und Beschreibungen von Kontrollstrukturen. Genaueres hierzu findet man in Müller (1999).

Lesetips

Die GPSG wird erstmals ausführlich in **?** vorgestellt. Wie bereits erwähnt, wurde die angegebene GPSG-Grammatik von Uszkoreit (1987) übernommen. In seinem Buch wird die GPSG ausfühlich erklärt, und es wird insbesondere auf Wortstellungsphänomene des Deutschen eingegangen.

Dem an GB interssierten Leser empfehle ich Grewendorf (1988).

Eine HPSG für das Deutsche wird in Müller (1999) beschrieben.

Kontrollfragen

1. Welche Vorteile hat das ID/LP-Regelformat der GPSG gegenüber den Ersetzungsregeln kontextfreier Grammatiken?

Anhang A

Abkürzungsverzeichnis

CFG	Context Free Grammar
CFL	Context Free Language
CNF	Chomsky Normal Form
DCFL	Deterministic Context Free Language
DCG	Definite Clause Grammar
FOPL	First Order Predicate Logic
FSA	Finite State Automaton
GPSG	Generalized Phrase Structure Grammar
HPSG	Head Driven Phrase Structure Grammar
LP	Linear Precedence
NL	natural language
NLP	natural language processing
NP	Nominalphrase
PDA	Push Down Automaton
PP	Präpositionalphrase
PRN	pronoun
UDC	unbounded dependency contstruction
VP	Verbphrase

Literaturverzeichnis

Bar-Hillel, Yehoshua, Perles, Micha A. und Shamir, Eliahu. 1961. On Formal Properties of Simple Phrase-Structure Grammars 14(2), 143–172.

Bratko, Ivan. 1987. *Programmierung für künstliche Intelligenz*. Bonn: Addison-Wesley Verlag (Deutschland) GmbH.

Clocksin, W. F. und Mellish, C. S. 1987. *Programmieren in Prolog*. Berlin, Heidelberg: Springer-Verlag.

Cooper, Richard. 1990. *Classification-Based Phrase Structure Grammar: An extended Revised Version of HPSG*. Dissertation, Edinburgh University, Center of Cognitiv Sience.

Dalrymple, Mary. 1993. *The Syntax of Anaphoric Binding*. CSLI Lecture Notes, Nr. 36, Stanford.

Engel, Ulrich. 1977. *Syntax der deutschen Gegenwartssprache*, Band 22 von *Grundlagen der Germanistik*. Berlin: Erich Schmidt Verlag.

Gazdar, Gerald und Mellish, Christopher. 1989. *Natural Language Processing in Prolog*. Addison-Wesley. http://www.informatics.susx.ac.uk/research/nlp/gazdar/nlp-in-prolog/, 17.10.2004.

Grewendorf, Günther. 1988. *Aspekte der deutschen Syntax. Eine Rektions-Bindungs-Analyse*. Studien zur deutschen Grammatik, Nr. 33, Tübingen.

Jackendoff, Ray S. 1977. $\overline{\mathrm{X}}$ *Syntax: A Study of Phrase Structure*. Cambridge: Massachusetts, London: England: The MIT Press.

Johnson, Mark. 1988. *Attribute-Value Logic and the Theory of Grammar*. CSLI Lecture Notes, Nr. 14, Stanford.

Lehner, Christoph. 1990. *Prolog und Linguistik*. München, Wien: R. Oldenburg Verlag.

Mellish, Christopher S. und Whitelock, Pete. 1992. *Computational Syntax*. AI3 – Lecture Notes, Edinburgh University.

Mellish, Christopher S., Whitelock, Pete und Ritchie, Graeme. 1994. *Techniques in Natural Language Processing 1: Module Workbook, Autumn Term 1994*. Department of Artificial Intelligence. University of Edinburgh. http://www.dai.ed.ac.uk/staff/personal_pages/chrism/distrib/csyn.ps, 14.03.2000.

Müller, Stefan. 1996. The Babel-System—An HPSG Prolog Implementation. In *Proceedings of the Fourth International Conference on the Practical Application of Prolog*, Seiten 263–277, London. http://hpsg.fu-berlin.de/~stefan/Pub/babel.html, 28.05.2008.

Müller, Stefan. 1999. *Deutsche Syntax deklarativ. Head-Driven Phrase Structure Grammar für das Deutsche*. Linguistische Arbeiten, Nr. 394, Tübingen: Max Niemeyer Verlag. http://hpsg.fu-berlin.de/~stefan/Pub/hpsg.html, 28.05.2008.

Pereira, Fernando C. N. und Shieber, Stuart M. 1987. *Prolog and Natural-Language Analysis*. CSLI Lecture Notes, Nr. 10, Stanford. http://www.mtome.com/Publications/PNLA/pnla.html, 23.02.2006.

Pollard, Carl J. und Sag, Ivan A. 1987. *Information-Based Syntax and Semantics*. CSLI Lecture Notes, Nr. 13, Stanford.

Pollard, Carl J. und Sag, Ivan A. 1994. *Head-Driven Phrase Structure Grammar*. Studies in Contemporary Linguistics, Chicago, London: University of Chicago Press.

Reape, Mike. 1991. Word Order Variation in Germanic and Parsing. DYANA Report Deliverable R1.1.C, University of Edinburgh.

Shieber, Stuart M. 1986. *An Introduction to Unification-Based Approaches to Grammar*. CSLI Lecture Notes, Nr. 4, Stanford.

Shieber, Stuart M., Uszkoreit, Hans, Pereira, Fernando, Robinson, Jane und Tyson, Mabry. 1983. The Formalism and Implementation of PATR-II. In *Research on Interactive Acquisition and Use of Knowledge*, Seiten 39–79, Menlo Park, CA: Artificial Intelligence Center, SRI International.

Steedman, Mark. 1991. Structure and Intonation. *Language* 67(2), 260–296.

Uszkoreit, Hans. 1987. *Word Order and Constituent Structure in German*. CSLI Lecture Notes, Nr. 8, Stanford.

Gesamtindex

$\oplus$, 136
_ , 44

Algorithmus, 27

Bambara, 24
berechenbar, 28
Bottom, 131
Bottom-Up, 75

Catalan-Folge, 22
Chomsky-Normal-Form, 80
Compiler, 79
Context-free backbone, 161
Control Agreement Principle, 174

data-driven-Strategie, 72
Default Value Assignment, 173
directed acyclic graph (DAG), 125
Disjunktion
    Eigenschaften, 134

Englisch, 19
equi-Verben, 59

Feature Cooccurrence Restrictions, 173
Foot Feature Convention, 174
Fundamentalregel, 94
Funktionen, 136
    $\oplus$, 136
    append, 136
Funktionsanwendung, 110

Gap Threading, *see* Lückenfädeln
GB, *see* Government and Binding
Generalized Phrase Structure Grammar, 170–176
goal-driven-Strategie, 72
Government and Binding, 177
GPSG, *see* Generalized Phrase Structure Grammar
Grammatik
    formale -, 10

Head Feature Convention, 173
Hilfsverb, 46

ID-Regeln, *see* Immediate-Dominanz-Rules
Immediate-Dominanz-Rules, 171
Implikation, 135
Interpreter, 79
intransitiv, 45

Kante
    aktiv, 94
    inaktiv, 94
Kapazität
    schwache-, 18
    starke -, 19
Kategorialgrammatik, 106–114
Komplement, 45
Komplexität, 29–36
    des Erkennens, 34
    des Parsens, 35
Komposition, 110
Kompositionalität, 113
Koordination, 48, 109–111
Koreferenz, 126

Lückenfädeln, 64
Left-Corner, 82
lexikalische Kategorie, 40–41
lexikalische Regel, 158
linear precedence rules, 174
Linksrekursivität, 74
Listen, 135
LP-Regeln, *see* linear precedence rules

Mehrdeutigkeit
    inhärente-, 20
Merkmalstruktur, 124
    Definition, 127

typisierte, 139
Metagrammatik, 171
Metaregeln, 171
Modifikator, 45
Morphologie, 46
multiple inheritance, 138
Multiplikationsregel, 157

Negation, 135
Typen, 140
Niederländisch, 23

Objektgrammatik, 171
Objektkontrolle, 59

Pfad, 125

raising-Verben, 59
rechtslinear, 13
Redundanzregel, 156
reentrancy, 126

Satzform
DCG, 54
Shift-Reduce-Erkenner, 75
Sprachen
deterministische kontextfreie, 20
indizierte, 24
kontextfreie, 23
reguläre, 19
structure sharing
Definition, 125
Subjektkontrolle, 59
Subkategorisierung, 144–146
Subsumption, 167
Definition, 128
Disjunktion, 133
Listen, 135
Typen, 137
Eigenschaften, 128
Subtyp, 137
Supertyp, 137

Template, 155–156
Templates, 146

Top, 131
Top-Down, 73
Trace, 61
transitiv, 45
Typ, 137–141
type raising, 111
Typhierarchie, 138
Typskelett, 149–152

unbounded dependencies, 48
Unifikation
Definition, 130
Disjunktion, 134
Listen, 136
Typen, 137
Eigenschaften, 131

Vererbung, 138

Wortstellung, 47

Zyklus, 127, 130