Fachbereich Informatik und Mathematik
Institut für Informatik

Diplomarbeit

Grammatikalische Prüfung von deutschen Texten und Fehleranalyse auf Basis von Attributgrammatiken

Sebastian Behr
Rosenborn 4
65719 Hofheim
Mat.-Nr.: 2663953

Juli 2009

eingereicht bei
Prof. Dr. Manfred Schmidt-Schauß
Künstliche Intelligenz / Softwaretechnologie

Danksagung

Hiermit möchte ich mich bei Allen bedanken, die mich bei der Entstehung dieser Diplomarbeit begleitet und unterstützt haben. Mein Dank gilt Prof. Dr. Schmidt-Schauß und Dr. David Sabel für die hervorragende Betreuung. Desweiteren gilt mein besonderer Dank Frau Professor Dr. Leuninger und Dr. Eric Fuß, die mir alle linguistischen Fragen freundlich und ausführlich beantwortet haben.

Ausserdem möchte ich mich bei meinen Freunden, meinem Vater Willi und meinem Bruder Martin für Anregungen und Motivation bedanken, sowie bei meiner Mutter Doris (1946 - 2008) für die stete Unterstützung beim Studium.

Sebastian Behr

Erklärung gemäß Diplomprüfungsordnung §11 Abs. 11

Hiermit bestätige ich, dass ich die vorliegende Arbeit selbstständig verfasst habe und keine anderen Quellen oder Hilfsmittel als die in dieser Arbeit angegebenen verwendet habe.

Hofheim-Langenhain am Taunus, im Juli 2009

S e b a s t i a n B e h r

Inhaltsverzeichnis

1 Motivation

Die Erkennung natürlicher Sprache mittels des Computers ist ein wachsendes Feld. Ziel dieser Diplomarbeit ist es, die Entwicklung und Implementierung eines Parsers zu dokumentieren, welcher für einen gegebenen Satz entscheiden kann, ob es sich um einen syntaktisch korrekt konstruierten Satz der deutschen Sprache handelt. Es wird bei korrekten (Teil)sätzen auch ein sogenannter Phrasenstrukturbaum ausgegeben, welcher die hierarchische Struktur des Satzes im Sinne linguistischer Theorien wiederspiegelt. Darüber hinaus ist auch eine nähere Analyse erwünscht, insbesondere soll bei grammatisch nicht korrekten Sätzen eine automatisierte Analyse Anhaltspunkte geben, was genau am vorliegenden Satz ungrammatisch ist.

Natürliche Sprache ist in ihrem Umfang, ihren Regeln und Variationen ungeheuer komplex. Auch Standardwerke wie [DUDG06] oder [DUDR06] können unmöglich alle Regeln und Worte der Sprache umfassen. Der von mir implementierte Parser bietet die Möglichkeit, mit einem verhältnismäßig sehr schlanken Regelwerk eine große Anzahl grammatischer Konstruktionen zu erfassen. Gleichzeitig sollen in dieser Diplomarbeit auch die Grenzen der Implementierung aufgezeigt werden.

Das Lexikon umfasst nur wenige Worte, da die Einträge eine genau festgelegte Form haben müssen. Ein Python-Konsolen-Interface erleichtert es, größere Folgen neuer Worte (und deren Abwandlungen, siehe Abschnitt 2) einzugeben. Der Wortschatz der deutschen Alltagssprache wird auf über 500.000 Worte geschätzt, der Wortschatz mancher Fachgebiete kann ein Vielfaches davon betragen. Selbst wenn man “nur“ den zentralen Wortschatz von 70.000 Worten heranzieht, so wird die Unmöglichkeit deutlich, bei manueller Pflege des Lexikons die deutsche Sprache umfassend erkennen zu können. Durch ein Interface, das automatisiert Daten in der richtigen Form aus einem anderen elektronisch gespeichertes Lexikon überträgt, ist jedoch eine Befüllung und Pflege des Lexikons gut denkbar.

In der Implementierung wurde auch die Skalierbarkeit des Lexikons berücksichtigt, sodass die praktische Anwendbarkeit auch bei einem großen Lexikon erhalten bleibt.

Da ich mich bei der Implementierung recht eng an sprachwissenschaftliche Theorien gehalten habe, gibt es einen weiteren Vorteil: Man nimmt an, dass die linguistischen Regeln generell für *alle natürlichen Sprachen* gültig sind und lediglich in Details parametrisiert sind (z.B. bei der Stellung des Verbs). Hätte ich einfach die Auflistung aller Regeln aus [DUDG06] implementiert, so wäre zwar die deutsche Sprache erfasst worden, für alle anderen Sprachen hätten diese Regeln jedoch versagt! In [DUDG06] werden zwar Schemata für einzelne Satzteile aufgelistet, wie diese Schemata aber durch das Zusammenspiel komplexer linguistischer Theorien und des Lexikons zustande kommt, wird dort nicht aufgeführt.

Ein adäquates Lexikon vorausgesetzt, reicht die Anpassung der weniger als 50 gramma-

tischen Regeln und die Modifikation einiger Hilfsfunktionen bereits aus, um den Parser auch für andere Sprachen "umzurüsten".

Der Hauptgrund, zur Syntaxanalyse natürlicher Sprache Prolog zu verwenden, liegt in der gegebenen Unterstützung von *Definite Clause Grammars*. Prologs Suchstrategie (Tiefensuche mit Backtracking) und eingebaute Restlistenverarbeitung reduzieren die zu bewältigende Aufgabe darauf, grammatische Regeln zu definieren. Der große Vorteil gegenüber einer imperativen Implementierung ist, dass syntaktische Ambiguitäten automatisch ermittelt werden können.

Der Nachteil von Prolog besteht darin, dass der Syntaxbaum immer nur lokal betrachtet werden kann. Dies entspricht nicht der linguistischen Realität, dass es auch zwischen "entfernten" Satzgliedern Beziehungen gibt.

Verwendete Materialien: Das Hauptthema ist die grammatikalische Überprüfung von Texten. Die Fehleranalyse schließt sich an die Implementierung an. Die grammatikalische Überprüfung von Texten entspricht dem Parsing eines Satzes. Alle modernen linguistischen Theorien basieren auf *generativen Transformationsgrammatiken*, die ähnlich wie *kontextfreie Grammatiken* mit einem Produktionssystem arbeiten. Als "Ausgangspunkt" für die Implementierung diente die "Mini-DCG" aus [SSSK06] für die deutsche Sprache. Sowohl die Mini-DCG als auch jede andere dokumentierte Prolog-Implementierung (z.B.: [KARW08], [DOUG94], [BRAT90, S. 437 ff.]), zu denen ich Informationen finden konnte, basiert darauf, für für die verarbeitete Sprache Satzbaupläne anzugeben. Sie legen die Positionen von Satzgliedern im hierarchischen Strukturgefüge des Satzes von vornerein fest ([BDS06, S. 20]). Somit ist auch indirekt diejenige Ordnung der Wörter vorgegeben, die der Sprecher hört oder liest. Dies ist problematisch: es gibt im Deutschen nicht nur eine Vielzahl unterschiedlicher Satzbaupläne (siehe [DUDG06, S. 932 ff.]), es ist zusätzlich möglich, Satzglieder in eine andere Reihenfolge zu bringen. Die Anzahl von Satzbauplänen, die man für eine umfangreiche Erfassung deutscher Sprache bräuchte, wäre sehr groß. Diese Modelle lassen sich daran erkennen, dass in ihnen die Produktionsregel **S** $\rightarrow$ **NP, VP** zu finden ist. Auch alternative Ansätze wie die *Controlled Partition Grammars* (vgl. [BLAC08]) basieren darauf, Satzbaupläne anzugeben.

Der *S-NP-VP*-Ansatz wurde 1965 von Noam Chomsky vorgestellt, ist linguistisch jedoch mittlerweile veraltet. Das Modell wurde bereits Mitte der 1970er Jahre stark modifiziert und seitdem fortlaufend weiterentwickelt. Der neuere Ansatz lässt sich daran erkennen, dass in Strukturdarstellungen die Bezeichner **CP** und **IP** auftreten. Statt alle möglichen Satzstrukturen aufzulisten, wird nun angenommen, dass es *genau eine* Grundwortstellung gibt, und sich alle anderen Strukturen anhand gegebener grammatischer Regeln ableiten lassen. In der Lehre und jüngerer sprachwissenschaftlicher Literatur findet sich praktisch nur noch dieser neuere Ansatz.

Als Leitfaden und Rahmen bei der Implementierung diente mir [LEUN04], ein Einführungsbuch für Studenten der Kognitiven Linguistik. Für Ergänzungen des Modells, insbesondere was komplexere syntaktische Strukturen (mit Nebensätzen) angeht, verwendete ich [STEI05] und [BDS06]. Einige Anmerkungen zum Modell und seiner historischen Entwicklung wurden [GREW89] entnommen.

Anhand der linguistischen Literatur habe ich eine Implementierung des CP-IP-Modells von Grund auf neu entwickelt, da keine Vorlagen aus dem Bereich der Informatik oder Computerlinguistik auffindbar waren.

Für die lexikalische Klassifikation von Worten und ihren Eigenschaften wurde [DUDG06] verwendet.

Die Implementierung des Grammatik-Parsers erfolgte mit SWI-Prolog. Um die Grundlagen von Prolog in dieser Arbeit zu beschreiben, habe ich [SSSK06], [BRAT90] und [MANL08] verwendet.

Das Python-Interface dient der Benutzerinteraktion und Input/Output-Operationen. Einige Python-Grundlagen sind [PYTH03] entnommen.

Fokus der Implementierung: Weitestgehend ausgeklammert aus der Implementierung werden alle Gebiete der Linguistik, die dem syntaktischen Satzbau vor- oder nachgeordnet sind. Zu ersteren zählt die **Morphologie**, die sich mit der Flexion von Worten, ihren Abwandlungen und der Zusammensetzung von Worten befasst. Zu letzteren zählt die **logische Form**, welche Ambiguitäten in Sätzen durch eine Umordnung der von der Syntaxtheorie "gefundenen" hierarchischen Strukturen herausstellt, sowie die **phonetische Form**, die eine Vorstufe zur menschlichen Lautproduktion ist.

Die Schnittstelle zur **Semantik**, die fester Teil der der syntaktischen Theorie ist ([LEUN04], [BDS06]), wird gar nicht behandelt, weil zu viele Aspekte einer formellen Anwendung ungeklärt sind.

$$\text{Morphologie} \rightarrow \overbrace{\text{D-Struktur} \rightarrow \text{S-Struktur}}^{\text{Fokus der Implementierung}} \rightarrow \text{PF/LF} \cdots \rightarrow \text{Semantik}$$

Doch auch in Bezug auf die Syntax beherrscht das in Prolog implementierte Modell nur einen Teil der sprachlichen Konstruktionen, welche durch die Syntaxtheorie bereits erklärt werden können. Der Abschnitt 5.4.1 dieser Diplomarbeit erläutert, welche sprachlichen Konstrukte sich durch die Prolog-Regeln bausteinartig zusammensetzen lassen.

2 Morphologische Grundlagen

2.1 Merkmale

Die Morphologie ist die Lehre von der Wortgestalt. Sie umfasst sowohl Komposita als auch die Flexionen oder Deklinationen von Worten. In diesem Abschnitt sollen Wortarten und ihre *morphologischen* Eigenschaften vorgestellt werden. Eigenschaften, die die Syntax betreffen, werden im folgenden Kapitel 3 behandelt. Sofern nicht anders erwähnt, stammen alle Grundlagen dieses Kapitels aus [DUDG06].

Als **Lexem** bezeichnet man zusammenfassend ein Wort (z.B. *"Haus"*) und alle seine Abwandlungen (*"Haus", "Häuser", "Häusern"* usw.). Jede Abwandlung wird als **Flexionsform** bezeichnet. Flexionsformen, die im Satz erscheinen, heissen auch **Wortform** oder **syntaktisches Wort**.

Die Flexionsformen können folgende grammatische Merkmale haben. Diese Merkmale bezeichnet man auch als **morphosyntaktische Merkmale**:

Merkmal	**Merkmalsausprägungen**
Person	1. Person, 2. Person, 3. Person
Numerus	Singular, Plural
Genus	Maskulinum, Femininum, Neutrum
Kasus	Nominativ, Genitiv, Dativ, Akkusativ
Komparation	Positiv, Komparativ, Superlativ
Modus	Indikativ, Imperativ, Konjunktiv
Tempus	Präsens, Präteritum

Es hängt von der zugehörigen Wortart ab, nach welchem dieser Merkmale die Lexeme flektiert werden, und mit welchem linguistischen Fachausdruck man die Flexion dann bezeichnet:

Wortart	**relevante morphosynt. Merkmale**	**Bezeichnung der Flexion**
Nomen	Numerus, Genus[1], Kasus	Deklination
Adjektiv	Numerus, Genus, Kasus, Komparation[2]	Deklination
Verb	Person, Numerus, Tempus, Modus	Konjugation
Pronomen	Person[3], Numerus, Genus, Kasus	Deklination
Nichtflektierbare	*nicht möglich*	–

[1] lexikalisch festgelegt
[2] sofern semantisch möglich
[3] teilweise

Man nimmt an, dass die für ein Wort "relevanten" Merkmale **morphologisch repräsentiert** sind. Das heisst, dass im Wortstamm, in den Präfixen, Suffixen oder Infixen einer Flexionsform alle morphosyntaktischen Merkmale der zugehörigen Wortart enthalten sind. Um die Flexionsform einer Wortart (Verb, Nomen, ...) eindeutig definieren zu können, muss man für alle Merkmale ihre Ausprägung angeben. Nicht alle Merkmale einer Wortart lassen sich bei jedem Lexem variieren, so ist z.B. der *Genus* bei Nomen lexikalisch festgelegt. Eine vollständige Auflistung *aller* Flexionsformen eines Lexems erreicht man davon abgesehen durch die Kombination aller Ausprägungen der Merkmale.

Die Möglichkeiten, morphologische Abwandlungen eines Lexems aufgrund seiner Klassifizierung algorithmisch zu bestimmen, sind begrenzt ([DUDG06, S. 154]).

In [SSSK06] wurde ein durch eine Micro-DCG implementierter Parser vorgeschlagen. Dort wurde die Erzeugung unterschiedlicher Flexionsformen durch morphologische Regeln nicht weiter berücksichtigt, da in der englischen Sprache die Varietät von Flexionsformen unterschiedlicher morphologischer Eigenschaften (z.B. viele verschiedene Suffixe bei Verbkonjugationen) erheblich weniger ausgeprägt ist als im Deutschen.

2.2 Komposita

Die Theorien zur **Derivation** bzw. **Komposition** beschreibt, wie sich aus mehreren Worten/Wortstämmen und Affixen Worte zusammensetzen lassen (siehe [LEUN04]). Beide sind im Programm nicht berücksichtigt, d.h. jedes Wort muss so im Lexikon stehen, wie es im Eingabesatz für den Parser verwendet werden soll.

2.3 Mehrteilige Prädikate

Die allermeisten Verben sind Vollverben. Es gibt jedoch auch Verben mit Spezialfunktionen, die sich mit anderen Konstituenten zu mehrteiligen Prädikaten verbinden können [DUDG06, S. 421]. Im Deutschen wird dies z.B. für bestimmte *Perfekt*-Formen gebraucht (z.B. "Fritz *hat gelogen*"). In der Implementierung verwenden wir insbesondere die **infinitregierenden Verben**. Ihnen ist ein Verb in Infinitivform strukturell untergeordnet (man sagt auch, das infinitive Verb wird *regiert*). Die infinitregierenden Verben lassen sich wiederum danach gruppieren, welche *Art* von Infinitiv sie vom regierten Verb verlangen:

Bezeichnung der inf.reg. Verben	**Verlangter Infinitiv**	**Beispiele**
(Hilfsverben)	Partizip II	haben, sein
Modalverben	Infinitiv	können, dürfen
Modalitätsverben	Infinitiv mit *zu*	brauchen, scheinen, pflegen
Verben der Wahrnehmung	A.c.I.[1]	sehen, hören

[1] *accusativus cum infinitivo*. Eingebetteter Satzteil, in dem das Verb im Infinitiv steht. Das Subjekt wird vom A.c.I.-Verb regiert und bekommt *Akkusativ*-Kasus zugewiesen. Bsp.: *"Maria [sieht]$_{AcI}$ [[den Redner]$_{Akk}$ [sprechen]$_{inf}$]"*

Insbesondere werden Verben mit Spezialfunktionen verwendet, um komplexe, mehrteilige Tempusformen zusammenzusetzen (für eine Auflistung derselben, siehe [DUDG06, S. 437]). Für ein aus den Verben $\lambda_1 \dots \lambda_i \lambda_j \dots \lambda_n$ bestehendes Prädikat (Beispiel: *"einzukaufen gedacht hatte"*) muss die Infinitrektion immer nur paarweise für λ_i, λ_j erfüllt werden:

- λ_i und λ_j sind adjazent
- λ_j ist ein infinitregierendes Verb
- λ_j regiert λ_i
- λ_i ist in derjenigen Infinitivform, die von λ_j gefordert wird.
- Wenn λ_i ein Partizip II ist, muss λ_j das "passende" Perfekthilfsverb sein. Bsp.: **"versucht ist"* ist ungrammatisch, das Partizip *"versucht"* kann nur mit einer Wortform des Perfekthilfsverbs *"haben"* kombiniert werden: *"versucht hat"*, *"versucht hatte"* ...
- λ_j kann von λ_i eine bestimmte Infinitivform "verlangen". Welche Form jedoch λ_j hat, wird wiederum von λ_{j+1} bestimmt.
- λ_n wird von keinem weiteren Verb regiert.

Dies ist eine enormer Vorteil für die Implementierung! Wir müssen uns wenig Gedanken um komplexe Tempusformen machen. Es genügt, immer nur die morphosyntaktischen Eigenschaften je zweier Verben paarweise zu betrachten.

3 Syntaktische Grundlagen

3.1 Linguistische Konventionen

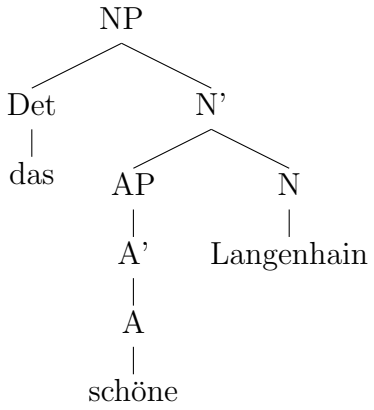
Zur Dokumentation der linguistischen Grundlagen habe ich im wesentlichen Bücher verwendet, die Studenten der kognitiven Linguistik einen Einstieg in die Methodologie der Linguistik und die Grundlagen syntaktischer Analyse vermitteln sollen.

Die Diplomarbeit soll in stärkerem Maße als die linguistische Literatur die beschriebenen Datenstrukturen mittels der in der Informatik gebräuchlichen Terminologie erfassen.

Grammatisch falsche Sätze werden durch das Sternsymbol * zu Beginn des Satzes gekennzeichnet.

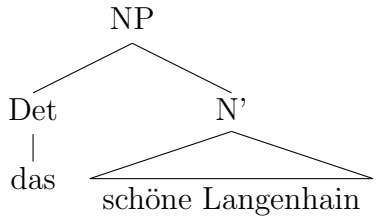
Als **Konstituente** bezeichnen wir abkürzend Satzglieder/Teilsätze, welche aus einem oder mehreren Worten bestehen. Sofern nicht anders angegeben, sind hierbei **Satzkonstituenten** gemeint (im Gegensatz zu Wortkonstituenten, welche die morphologischen Bestandteile eines einzelnen Wortes beschreiben) [DUDR06]. Hauptsätze werden auch als **Matrixsätze** bezeichnet.

Sätze haben eine hierarchische Struktur, welche am übersichtlichsten durch **Bäume**[1] dargestellt werden kann:



Ist die Struktur einer Satzteils unbekannt oder nicht von Interesse, so kann man sie durch ein "dachförmiges" Dreieck abkürzen. Die Spitze des **Dachs** ist mit der Wurzel des entsprechenden Teilbaums markiert, unter seiner Unterseite stehen alle Worte, die zur Konstituente gehören.

[1] Definition z.B. gemäß [CLR94]



Es gibt noch eine kompaktere Darstellung mittels **indizierter Klammern**, die zu Bäumen äquivalent ist. Hierbei werden alle Worte, die zu einer Konstituente gehören, durch eckige Klammern [ ] eingefasst. Ist ein Knoten α die Wurzel des zugehörigen Teilbaums, so wird die linke Klammer mit α indiziert: $[_\alpha$]. Wir können die Klammerung "vollständig" durchführen, so dass jeder innere Knoten des Baums als Index repräsentiert ist:

$$[_{NP}\ [_{Det}\ \text{das}\ ]\ ['_N\ [_{AP}\ [_{A'}\ [_A\ \text{schöne}\ ]\ ]\ ]\ [_N\ \text{Langenhain}\ ]\ ]\ ]$$

... oder man stellt nur bestimmte Konstituenten heraus:

$$[_{NP}\ \text{das schöne}\ [_N\ \text{Langenhain}\ ]\ ]$$

Die vollständige Klammerung hat den Nachteil, dass sie für komplexe Sätze unübersichtlicher ist als Bäume. [STEI05, S. 15 ff.]

3.2 Lexikalische Kategorien

Jedes Wort einer Sprache lässt sich einer passenden **lexikalischen Kategorie** zuordnen. Man nimmt an, dass eine Kategorie durch ein **Bündel** sogenannter **distinktiver syntaktischer Merkmale** definiert ist. Ein *Bündel* könnte man durch eine Menge beschreiben. Ein *distinktives syntaktisches Merkmal* $[\pm M]$ hat die Ausprägungen $[+M]$ und $[-M]$. Die Menge möglicher unterschiedlicher Kategorien ergibt sich aus dem Produkt der Anzahl der Merkmalsausprägungen. Jede natürliche Sprache besitzt Nomen und Verben. Dementsprechend wurden die Merkmale $[\pm N]$ und $[\pm V]$ gewählt, um die Kategorien zu definieren. Folgende Tabelle ist aus [LEUN04, S. 109] übernommen:

	+N	**-N**
+V	A (Adjektiv)	V (Verb)
-V	N (Nomen)	P (Präposition)

Im Folgenden können wir lexikalische Kategorien einerseits als die Menge aller Worte verstehen, die dieselben Merkmalsausprägungen wie die Kategoriendefinition tragen. Wir können N, A, V, P andererseits auch als Symbole betrachten, aus denen sich ein Wort der entsprechenden Kategorie ableiten lässt. Zusätzlich treten jedoch noch mehr Worte auf, die in der linguistischen Literatur nicht eindeutig einer dieser Kategorien zugeordnet sind:

- Adv (Adverb)

- Koord (Koordinierende Konjunktion)
- Det (Determinatoren)
- W-Worte (*"Was"*, *"Wie"*, ...)
- Ana (Anaphern)
- Pron (Pronomen)
- PosPron (Possessivpronomen)
- Part (Partikel, *"zu"*, *"sehr"*, ...)

Sie alle fassen wir in der Menge[2] $\mathcal{L}$ der lexikalischen Kategorien zusammen.

3.2.1 zu V

Die Zuweisung des Kasus an die Objekte des Verbs erfolgt mittels Rektion (siehe Abschnitt 3.12.8). Wir benötigen den Kasus, um Grammatikalitätsunterschiede erfassen zu können. Der Kasus ist auch immer mit einer bestimmten strukturellen Position im Satz verbunden.

Folgende funktionale/lexikalische Kategorien weisen den Konstituenten innerhalb ihrer Projektion einen bestimmten Kasus zu:

N weist *Genitiv* an COMP zu.

P weist *Dativ* an COMP zu.

I weist *Nominativ* an SPEC zu.

Für V, bei der das Komplementsystem aufgrund multipler Objekte und Θ-Rollen komplexer ist, muss weiter differenziert werden.

3.3 Government-Binding-Theory

Natürlichsprachliche Sätze zeichnen sich durch eine hierarchische Struktur aus. Beispiel: *"Das rote Feuerwehrauto fährt in die Stadt"*. Dass in diesem Fall *"rote"* und *"Feuerwehrauto"* enger miteinander verknüpft sind als *"rote"* und *"Stadt"*, ist sogar Menschen ohne linguistische Vorkenntnisse intuitiv begreiflich. Das Hauptziel jeder sprachwissenschaftlichen Theorie ist es, Regeln zu beschreiben, welche die menschliche Fähigkeit zur Sprachverarbeitung bzw. -produktion approximieren. Dieser Prozess kann bei Weitem noch nicht als abgeschlossen betrachtet werden. Jedes Modell soll einen möglichst großen Teil aller wohlgeformten Sätze erklären können und "einfach" sein. Letzteres bezieht sich darauf, dass zu einer propagierten grammatischen Regel keine Ausnahmefälle unterschieden werden müssen. Ich habe mich bei der Implementierung eng an die **Government**

[2]Definition gemäß [CLR94, S. 77]

Binding Theory (dt.: **Rektions-Bindungs-Theorie**) gehalten, die zwar nicht das einzige oder jüngste linguistische Modell ist. Sie ist jedoch weit entwickelt, ausgereift und mächtig.

Zudem ist, was für die Recherche zur Diplomarbeit eine erhebliche Rolle spielte, weitreichend Literatur vorhanden – im Gegensatz zu neueren Ansätzen wie z.B. dem **Minimalistischen Programm** nach Chomsky (siehe [CHOM95]).

Die *GBT* besteht aus einzelnen, nicht sehr eng miteinander verknüpften Theoriekomponenten ([BDS06, S. 22]), die auch **Module** genannt werden. Folgende Module werden in [LEUN04] beschrieben:

- X-Bar-Theorie
- Kasustheorie
- Lexikon und Θ-Theorie
- Bewege-α
- Rektionstheorie
- Bindungtheorie
- Kontrolltheorie

Teilweise bauen die Module aufeinander auf. Sofern nicht anders angegeben, stammen die Grundlagen dieses Kapitels aus [LEUN04].

3.4 Modul: X-Bar-Theorie

Die X-Bar-Theorie geht auf Noam Chomsky zurück und wurde bereits 1970 entwickelt [GREW89]. Bisher wurden nur die lexikalischen Kategorien definiert. Jedem Menschen ist intuitiv begreiflich, dass manche Satzteile enger zusammen gehören als andere. Eine Erklärung dafür liefert die X-Bar-Theorie, welche hierarchische Strukturen eines Satzes in Form eines Baums beschreibt.

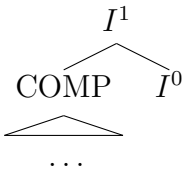
Die Bäume können aus lexikalischen Elementen mittels des **X-Bar-Schemas** aufgebaut werden, und werden auch als **Projektionen** bezeichnet. Das **X-Bar-Schema** ist nichts anderes als die rekursive Definition eines Baums, dessen Knoten einen Grad $i \in \{0, 1, 2\}$ haben. Die Blätter sind mit lexikalischen oder funktionalen[3] Kategorien X markiert. An *jedem* Knoten des Baums notiert man in Potenzschreibweise die **Ordnung**/den **Typ**/die **Komplexität** der Projektion. Lexikalische Kategorien/Blätter haben hierbei immer den Typ 0 und werden als **Kopf** (engl.: *"head"*) der Projektion bezeichnet.

Die trivialen Typ-0-Projektionen können mit sogenannten **Komplementen** (engl.: *"complementizer"* oder **COMP**) zu **Projektionen 1. Ordnung** verbunden werden.

[3] siehe folgenden Abschnitt 3.6; das Symbol $\mathcal{F}$ steht für die Menge der funktionalen Kategorien

Wir nehmen hierzu X^0 und COMP als die Wurzeln zweier verschiedener Teilbäume an. Wir definieren nun eine neue gemeinsame Wurzel X^1 für die beiden Teilbäume, und verbinden sie durch zwei Kanten mit X^0 und COMP. Ob X^0 hierbei als linker oder als rechter Teilbaum zugeordnet wird, ist von der betrachteten Sprache, der lexikalischen/ funktionalen Kategorie von X *und* der Kategorie von COMP abhängig. X^1 gehört derselben Kategorie X an wie der Kopf, hebt sich jedoch durch die größere Komplexität 1 vom Kopf ab.

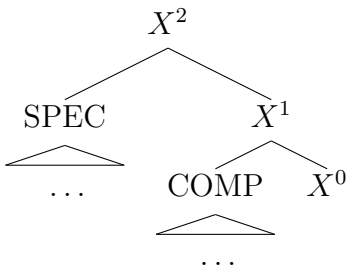
Bsp.: ist das Blatt mit I^0 markiert, so ist die Wurzel der Typ-1-Projektion mit I^1 markiert:



Typ-1-Projektionen können mit Spezifikatoren (engl.: *"specificator"* oder **SPEC**) zu **Projektionen 2. Ordnung** verbunden werden. Auch hier wird ein Knoten X^2 eingeführt, der mittels zweier Kanten zur gemeinsamen neuen Wurzel von X^1 und $SPEC$ wird. Die Neuerung gegenüber älteren Modellen besteht genau darin, dass maximale Projektionen **endozentrisch** sind, d.h. dass sie die gleichen Merkmale tragen wie ihr Kopf. Ein solcher Baum 2. Ordnung wird in der Linguistik auch als **Phrase** oder **maximale Projektion** bezeichnet. Letzteres Synonym begründet sich in der Beschränkung der maximalen Komplexität einer Projektion. Die Tiefe einer Projektion kann zwar durch Adjunktionen (siehe Abschnitt 3.5) noch erhöht werden, Projektionen einer Ordnung > 2 sind aber nicht möglich.

COMP und SPEC sind genau wie X^2 immer maximale Projektionen, jedoch sind die lexikalisch/funktionalen Kategorien von COMP und SPEC von X notwendig verschieden!

Bsp.:



Den Pfad vom Kopf X^0 zur maximalen Projektion X^2 bezeichnet man auch als **Projektionslinie**. [GREW89]

Modifikationen: Da die rekursive Definition in oben ausgeführter Variante keinen Basisfall kennt und somit unendlich wäre, ist es erlaubt, dass X^2 und/oder X^1 nur eine unäre Verzweigung zu X^1 bzw. X^0 besitzen. Somit wird auf SPEC bzw. COMP verzichtet. Man beachte jedoch, dass diese Auslassung(en) gegebenenfalls weiteren grammatischen Regeln genügen müssen, die zu anderen Modulen gehören!

Obwohl das Konzept der Projektion von Eigenschaften hinreichend ist, kann man auch von Kongruenz der lexikalischen Eigenschaften zwischen den Knoten einer Projektion sprechen. Es ist also nicht zwingend eine "Richtung" bei der Übertragung kategorialer

Merkmale vorgegeben. [GREW89]

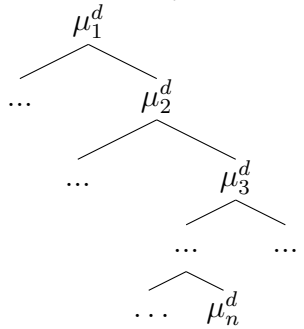
Wesentlich gebräuchlicher als die Kennzeichnung der Symbole mit hochgestellten Ziffern ist es , X^0 mit X, X^1 mit X' (gesprochen "X Bar") sowie X^2 mit XP (z.B. NP für *Nominalphrase*) zu bezeichnen. Aus diesen gängigeren Bezeichnungen leitet sich auch der Name der Theorie ab. Keine der Schreibweisen hat jedoch einen Vor- oder Nachteil bezüglich der Mächtigkeit der Darstellung.

Mittels dieser wenigen Regeln kann jeder Satz der deutschen Sprache erzeugt werden. Natürlich ist dies zur Erkennung gültiger Sätze noch nicht ausreichend. Jede Phrasenstruktur für einen Satz ist ein Baum. Damit die Struktur gültig ist, muss es Pfade mit bestimmten Eigenschaften geben, bzw. die Anzahl bestimmter Knoten auf einem Pfad ist beschränkt (siehe z.B. Abschnitt 3.12.4). Die Überprüfung von Pfaden und Knoten begründet auch, warum eine kontextfreie Grammatik für einen Parser natürlicher Sprachen unzureichend ist. Nur durch die attributierten Symbole einer *Definite Clause Grammar*[4] können Pfade, Zähler uvm. implementiert werden.

Wie man sich leicht klar macht, gibt es viele hundert verschiedene Projektionen. Dies ist durch die Anzahl der lexikalischen und funktionalen Kategorien, Permutationen bezüglich Links-/Rechtsköpfigkeit und den möglichen Kategorien von COMP oder SPEC bedingt. Nur ein Bruchteil all dieser Projektionen ist in der Praxis relevant oder überhaupt grammatisch. Welche dies sind, hängt von der betrachteten Sprache ab. Bsp.: Im Englischen ist die Projektion $[_{I^1} \quad I^0 \quad V^2\ ]$ grammatisch, im Deutschen ist I dagegen rechtsköpfig: $[_{I^1} \quad V^2 \quad I^0\ ]$.

3.5 Adjunktion

Eine Projektion μ^d, $d \in \{0, 1, 2\}$, und $\mu \in (\mathcal{L} \cup \mathcal{F})$ (eine lexikalische oder funktionale Kategorie), kann auf mehrere **Segmente** $\mu_1^d \ldots \mu_i^d \ldots \mu_n^d$ verteilt sein. Jedes Segment besteht aus genau einem Knoten. μ_i^d ist dabei immer die Wurzel eines Teilbaums im Phrasenstrukturbaum. Das in der Indizierung darauf folgende Segment μ_{i+1}^d ist entweder der linke oder der rechte Tochterknoten von μ_i^d.



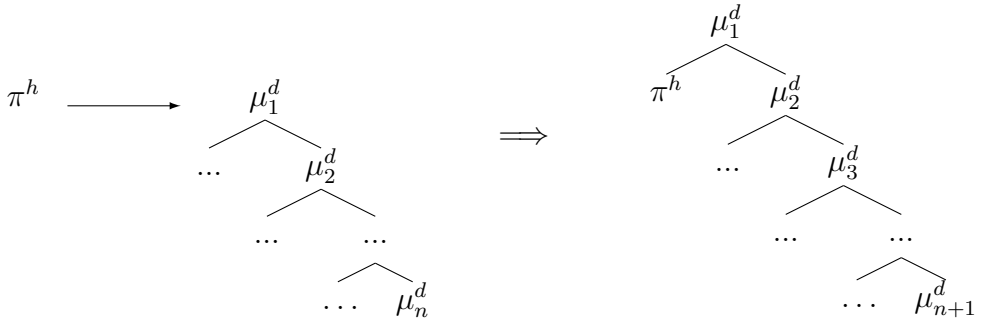
Bei Projektionen μ^d, die nicht auf mehrere Segmente verteilt sind, nimmt man implizit an, dass der mit μ^d markierte Knoten das *einzige* Segment der Projektion ist! Ausgelöst wird eine *Segmentierung* immer durch **Adjunktion**.

[4] siehe Abschnitt 4.2.5

Die Adjunktion ist die Basis sowohl von Adjunktionstransformationen als auch von transformationsunabhängigen Adjunktionen. Das Ergebnis einer Adjunktion wird auch als **verteilte Kategorie** bezeichnet. Die Komplexität von Projektionen bleibt auch nach Adjunktion auf Grad 2 beschränkt. Adjunktionen verändern auch nicht die lexikalisch-funktionale Kategorie der Projektion.

Die Adjunktion ist eine rekursiv definierte Vorschrift, die sich prinzipiell beliebig oft hintereinander anwenden lässt:

1. Sei μ_i^d Segment einer Projektion mit Grad $d \in \{0, 1, 2\}$, Segmentnummer $i = 1$, und Projektionstyp $\mu \in (\mathcal{L} \cup \mathcal{F})$.
2. Sei nun π^h eine weitere Projektion mit $h \in \{0, 2\}$ und $(d = 0 \wedge h = 0) \vee (d = 1 \wedge h = 2) \vee (d = 2 \wedge h \in \{0, 2\}))$, die adjungiert werden soll.
3. Betrachte μ_i^d und π^h als Wurzeln von Teilbäumen im Phrasenstrukturbaum.
4. Erstelle einen Knoten, der durch zwei neue Kanten zur gemeinsamen Wurzel von μ_i^d und π^h wird. Markiere den neuen Knoten mit dem Projektionstyp μ und dem Komplexitätsgrad d.
5. Versehe den neuen Knoten μ^d mit Index 1, um ihn als neues, erstes Segment der Projektion zu kennzeichnen. Erhöhe bei allen strukturell tiefer gelegenen Segmenten, die auch noch *zur Projektion gehören*, den Segmentindex i um +1.



Aus den Vorschriften ergeben sich folgende interessante Eigenschaften:

- Es können lexikalische Kategorien π^0 an andere lexikalische Kategorien μ^0 oder an maximale Projektionen μ^2 adjungiert werden, und maximale Projektionen π^2 an andere Projektionen μ vom Typ 1 oder 2.
- Adjunktionen können bei bereits segmentierten Projektionen immer nur an das oberste Glied stattfinden.
- Adjunktionen verändern den Komplexitätsgrad einer Projektion nicht.

- Das strukturell am höchsten gelegene Segment einer Projektion trägt immer den Index 1.
- Das strukturell am tiefsten gelegene Segment einer Projektion trägt immer den höchsten Index.

Es können z.B. beliebig viele Präpositionalphrasen, die nicht Teil des Subkategorisierungsrahmens des Verbs sind, an Verbalphrasen adjungiert werden:

$[_{VP_1}\ [_{PP}$ *in Hamburg* $]\ [_{VP_2}\ [_{PP}$ *neben der Biertheke* $]\ [_{VP_3}$ *den Verbrecher verhaftet* $]\ ]\ ]\ ]$

Da ihre Existenz nicht von anderen Konstituenten überprüft wird, bezeichnet man sie auch als *freie Adjunkte* (siehe auch Abschnitt 3.7.1).

3.6 Funktionale Kategorien

Funktionale Kategorien entsprechen nicht notwendig einer Wortart. Vielmehr nimmt man an, dass sie bestimmte Eigenschaften des Satzes beinhalten. Sie können genau wie lexikalische Kategorien Phrasen projizieren. Die Tatsache, dass die Köpfe z.T. nur funktionale grammatische Eigenschaften beinhalten und **phonetisch nicht spezifiziert**[5] sind, darf nicht damit verwechselt werden, dass die Köpfe "leer" sind. [STEI05, S. 28]

3.6.1 C

Steht für Complementizer. Man nimmt an, dass diese funktionale Kategorie Informationsträger bezüglich der Eigenschaften eines Satzes ist. Die Satzart ergibt sich u.a. aus der Besetzung der SPEC- und C^0-Positionen. Entweder können am C^0-Head Konjunktionen (*"dass"*, *"und"*, *"oder"*, *"weil"*, *"ob"*) basisgeneriert sein, oder es können I^0+V^0-Kategorien adjungiert werden. Eine CP schließt einen Satz nach oben hin ab. Die Merkmale einer CP, welche durch die Besetzungen besagter Positionen determiniert werden, bestimmen, um welche Satzart es sich handelt. Folgende Satzarten finden sich im Deutschen (aus [LEUN04, S. 146]):

[5] die Konstituente an dieser Stelle wird nicht vom Sprecher ausgesprochen

Satzart	**phonetisch spezifizierte Konstituente in SPEC-C^1**	**phonetisch spezifizierte Konstituente in C^0**	**Merkmale der Kategorie C**
deklarativer Wurzelsatz	XP $\overset{\text{Move-}\alpha}{\longrightarrow}$ SPEC-C^1	I^0 $\overset{\text{Move-}\alpha}{\longrightarrow}$ C^0	[-w, -imp]
deklarativer Konstituentensatz	$\varnothing$	*dass*	[-w, -imp]
interrogativer Wurzelsatz **A**	W-Pronomen $\overset{\text{Move-}\alpha}{\longrightarrow}$ SPEC-C^1	I^0 $\overset{\text{Move-}\alpha}{\longrightarrow}$ C^0	[+w, -imp]
interrogativer Wurzelsatz **B**	$\varnothing$	I^0 $\overset{\text{Move-}\alpha}{\longrightarrow}$ C^0	[+w, -imp]
interrogativer Konstituentensatz **A**	W-Pronomen $\overset{\text{Move-}\alpha}{\longrightarrow}$ SPEC-C^1	$\varnothing$	[+w, -imp]
interrogativer Konstituentensatz **B**	$\varnothing$	*ob*	[+w, -imp]
imperativer Satz	$\varnothing$	I^0 $\overset{\text{Move-}\alpha}{\longrightarrow}$ C^0	[-w, +imp]

Im Anhang A finden sich einige Beispiele für verschiedene Satzarten inklusive ihrer syntaktischen Analyse.

3.6.2 I

Man nimmt an, dass die Flexion des Verbs, also seine Tempus- und Kongruenzmerkmale, nicht in der VP entsteht, sondern eigenständige Merkmale sind. Die Flexionsmerkmale sind in der funktionalen Kategorie I^0 enthalten. Damit das Verb, also der Kopf einer Verbalphrase, seine Flexionsmerkmale *erhalten* (oder auch: *überprüfen* [STEI05]) kann, muss V^0 an die Position I^0 bewegt werden. Der Kopf der funktionalen Kategorie I ist Träger der Flexionsmerkmale des Verbs. Welche Modalitäten bei diesem Bewegungsmechanismus zu beachten sind, siehe Abschnitte 3.5 und 3.10. Das *I* steht für **Inflection**, dt. **Flexion**.

Der Kopf I^0 projiziert eine Phrase IP. In der COMP-Position steht eine VP, in der SPEC-Position eine Subjekt-NP. Wenn die Merkmale aus I^0 an ein Verb übertragen wurden, dann weist I^0 dem Subjekt den *Nominativ*-Kasus zu. In der IP wird auch die Kongruenz zwischen Subjekt und Verb bezüglich der morphosyntaktischen Merkmale *Person* und *Numerus* überprüft. Bsp.:

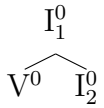
$$[_{IP}\ \textit{die Studenten} \overset{\text{Kongruenz}}{\longleftrightarrow} \textit{lernen}\ ]$$

3.6.3 Zusammenfassung

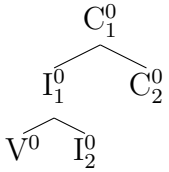
Die Symbole I und C fassen wir in der Menge $\mathcal{F}$ der funktionalen Kategorien zusammen.

3.6.4 Mehrfache Typ-0-Adjunktion

Die Bewegung einer Typ-0-Kategorie erzwingt an der "Landeposition" die Adjunktion an eine weitere lexikalische Kategorie. Am häufigsten tritt die Adjunktion von V^0 an I^0 auf:



Für diese Adjunktionsstruktur gibt es die abkürzende Schreibweise $\mathbf{I^0{+}V^0}$. Es ist möglich, dass auch eine $X^0{+}Y^0$-Kategorie erneut bewegt wird. Dies macht eine weitere Typ-0-Adjunktion erforderlich:



Diese Konfiguration tritt bei Verbzweitsätzen auf und wird dann als $\mathbf{C^0{+}I^0{+}V^0}$ notiert.

3.7 Modul: Lexikon und Θ-Theorie

3.7.1 Subkategorisierung

Transitive, ditransitive und tritransitive Verben benötigen Objekte in Form von Phrasen als Ergänzungen. Meistens handelt es sich dabei um NPs, jedoch sind PPs oder CPs ebenfalls möglich. Die Anzahl und die Art dieser Komplemente, welche ein Verb benötigt, sind im Lexikon gespeichert. Die entsprechenden Repräsentationen im Lexikon bezeichnet man als **Subkategorisierungen**. Der Begriff kann sich allgemein auf lexikalische Elemente beziehen, im Rahmen dieser Implementierung reicht es aus, nur Verben zu betrachten. Neben dem Begriff *Subkategorisierung* gibt es in [STEI05, S. 41] noch die synonyme Bezeichnung **c-Selektion** (c → *"categorial"*, engl. *"kategorial"*).

Subkategorisierungen kann man sich als Listen von Tupeln vorstellen. Das erste Element des Tupels enthält die geforderte kategoriale Form des Komplements, z.B. *NP*. Das zweite Element enthält den Kasus, welcher dem Objekt durch das Verb zugewiesen wird. Das Objekt erhält durch die Zuweisung seine grammatische Funktion und ist für den Sprecher identifizierbar. Das dritte Element enthält die Information, ob die Ergänzung zwingend erforderlich ist, oder ob sie ausgelassen werden kann. Die Kennzeichnung von Ergänzungen als obligatorisch oder optional ist im **Uniformitätsprinzip** begründet, nach dem das Lexikon organisiert ist. Dies bedeutet, dass das Lexikon[6] für ein Lexem keine redundanten Subkategorisierungsinformationen für unterschiedliche Satzstrukturen enthält. Bsp.:

[6]Bezogen auf das menschliche Gehirn

Verb: *schenken*
Subkategorisierung:[(NP,Dativ,obligatorisch),(NP,Akkusativ,obligatorisch)]

Verb: *kochen*
Subkategorisierung: [(NP,Akkusativ,optional)]

Zusätzlich zu den subkategorisierten Objekten gibt es auch noch **freie Ergänzungen** (auch: **freie Adjunkte**), welche in beliebiger Anzahl an die VP adjungiert werden können. Diese sind nicht im Lexikoneintrag vermerkt. Eine gebräuchliche Form sind z.B. Präpositionalphrasen mit Ortsbezug:

$$\textit{Er hat } \underline{[_{PP} \textit{ auf dem Herd }]} \ \underline{[_{PP} \textit{ in der Küche }]} \ldots [_{NP} \textit{ den Kaffee }] \textit{ gekocht.}$$

3.7.2 Θ-Theorie

Die Θ-Theorie (das Θ steht hierbei für **"Thema"**) ist eine Schnittstelle zur Semantik. Für jedes Wort ist im Lexikon eine Menge von semantischen Merkmalen gespeichert. Bsp.:

Kuh: [+Tier,+belebt,-flugfähig, ...]

Eine Θ-Rolle ist eine Art semantische Beschreibung, z.B. AGENS, PATIENS, THEMA, ORT, MITTEL etc. Verben können Θ-Rollen an Konstituenten des Satzes zuweisen, so dass man sie auch als **Θ-Zuweiser** bezeichnet. Die Empfänger der Θ-Rollen werden schlicht als **Argumente** bezeichnet. Gebräuchlich für den Zuweisungsvorgang ist auch der Fachbegriff **Θ-Markierung**. Argumente sind die Objekte des Verbs und ggf. das Subjekt. Welche und wieviele Θ-Rollen ein Verb vergibt, ist lexikalisch festgelegt und wird im Lexikon im sogenannten **Θ-Gitter** gespeichert. Bsp. (aus [LEUN04, S. 135]):

schenken: [AGENS, ZIEL, THEMA]
wohnen: [THEMA, ORT(Lokativ)]

Um die grammatische Korrektheit der Θ-Zuweisung sicherzustellen, gibt es drei Regeln:

Projektionsprinzip: thematische Rollen müssen phrasenstrukturell realisiert sein. Dies bedeutet, dass an der Stelle in der X-Bar-Struktur des Satzes, die die Θ-Rolle erhält, auch eine passende Konstituente vorhanden sein muss. Dabei kann es sich z.B. um eine NP handeln. Die Regel betrifft somit die Existenz der *Argumente* des Θ-Zuweisers.

Θ-Kriterium: Jede Θ-Rolle ist mit genau einem Argument und jedes Argument mit genau einer Θ-Rolle verknüpft. Zwischen Θ-Rollen und Argumenten besteht somit eine bijektive Beziehung.

Sichtbarkeitsbedingung: nur kasusmarkierte XPs sind für die Θ-Zuweisung sichtbar. Das Gesetz regelt die Bedingung, unter denen Θ-Zuweisung stattfinden kann. Man kann sagen, dass Θ-Rollen "mächtiger" sind als Subkategorisierung, weil sie nicht auf Rektion angewiesen sind.[7]

Die Θ-Theorie hat insgesamt zum Ziel, die semantische Kombinierbarkeit von Konstituenten bereits auf der syntaktischen Ebene einzugrenzen.

Verbindung zur Subkategorisierung

Auch zwischen den Subkategorisierungen des Lexikons und den thematischen Rollen, die im Θ-Gitter gespeichert sind, besteht eine bijektive Beziehung. Gibt es dieses 1:1-Verhältnis nicht, so ist immer mindestens eine der drei o.g. Regeln verletzt.[8] Freie Adjunkte erhalten jedoch generell keine Θ-Rolle und sind nicht subkategorisiert. Zwischen der Θ-Rolle und der von der Subkategorisierung geforderten Kategorie/Kasus gibt es eine gewisse Parallele: das Subjekt z.B. erhält die Rolle AGENS, zugleich muss gefordert werden, dass sie eine *Nominativ*-NP ist. Umgekehrt werden PPs nur Θ-Rollen mit Orts- oder Zeitbezug erhalten, z.B. ORT(DIREKTIONAL) oder ZEIT.

Probleme

In [BDS06, S. 270 ff.] wird beispielhaft erläutert, wie ein Θ-Gitter im Lexikon aufgebaut sein könnte. In keiner der verwendeten Quellen wird jedoch ein genauer Mechanismus angegeben, mit dem systematisch eine Übereinstimmung zwischen der Menge semantischer Merkmale S eines Wortes und einer vergebenen Θ-Rolle R überprüft werden kann.[9] Problematisch sind die noch komplexeren Fälle. Beispiel: im Satz

"Peter schenkt Maria $[_{NP}$ *einen schönen Blumenstrauß* $]$*."*

vergibt das Verb *schenkt* die thematische Rolle THEMA an die NP. Es bleibt in der verwendeten Literatur offen, wie sich die Mengen von semantischen Merkmalen S_1, S_2, S_3 der Worte *einen*, *schönen*, *Blumenstrauß* auf die Wurzel der NP übertragen.

Aus diesem Grund sind Θ-Rollen nicht implementiert. Wären diese theoretischen Probleme gelöst, so fiele es recht leicht, thematische Rollen in die Grammatik der Implementierung zu integrieren. Die Repräsentation im Lexikon und die Überprüfung in den grammatischen Regeln hätten analog und parallel zu den Subkategorisierungen zu erfolgen.

---

[7] Ob die Zuweisung unter Rektion erfolgt, ist in der linguistischen Theorie nicht endgültig geklärt.

[8] Beispiele: gibt es weniger Subkategorisierungen als Θ-Rollen, so ist zumindest die Sichtbarkeitsbedingung verletzt. Gibt es weniger Subkategorisierungen *und* weniger Objekte des Verbs, so sind Projektionsprinzip und Θ-Kriterium verletzt. Bei mehr Subkategorisierungen als Θ-Rollen gibt es Probleme mit dem Θ-Kriterium, weil nicht eindeutig klärbar ist, *welche* Konstituente *welche* Rolle erhalten muss.

[9] Beispiele für die Probleme: Wieviele semantische Merkmale gibt es überhaupt? Wie sind Θ-Rollen formal definiert?

3.8 Phrasenstrukturbaum

Durch die rekursiv definierten Regeln der X-Bar-Struktur, welche Projektionen der funktionalen und lexikalischen Kategorien beschreiben, lassen sich sukzessive immer komplexere Strukturen aufbauen. Solche Bäume, die mehrere maximale Projektionen enthalten können, werden auch als **Phrasenstrukturbäume** bezeichnet. Mit $\mathcal{P}$ bezeichnen wir den Phrasenstrukturbaum, welcher die X-Bar-Struktur für einen *ganzen Satz* umfasst. $\mathcal{T}_\alpha$ ist ein Teilbaum von $\mathcal{P}$, dessen Wurzel der Knoten α ist. Um diese Teilbaumbeziehung auszudrücken, verwenden wir ab sofort den Operator $\in_\Delta$. Wir können ihn aber auch dazu verwenden um auszudrücken, dass *einzelne Knoten* in $\mathcal{P}$ enthalten sind. α steht dabei für eine lexikalische oder funktionale Kategorie. Ebenso führen wir die Funktion *tiefe*(α) ein:

Definition: *tiefe*: $\alpha \in_\Delta \mathcal{P} \longmapsto \mathbb{N}$. Anzahl der Kanten des Pfads zwischen dem Knoten $\alpha \in_\Delta \mathcal{P}$ und der Wurzel des Phrasenstrukturbaums $\mathcal{P}$.

Das Aufbauen einer X-Bar-Struktur wird auch als **generieren**[10] bezeichnet. Ist ein Teilbaum von $\mathcal{P}$ nicht an eine andere Position bewegt worden, sondern befindet er sich vielmehr "immer noch" an der spezifischen Position, an der er mittels der X-Bar-Regeln aufgebaut wurde, so bezeichnet man ihn als **basisgeneriert**.

Satzaufbau: Zunächst werden in einer VP das Verb des Satzes und seine Objekt-XPs basisgeneriert. Das Verb steht an der Kopfposition V^0 der VP, das 1. Objekt wird an der COMP-Position generiert. Alle weiteren Objekte werden an V^1 adjungiert ([STEI05]). Für die Implementierung wird folgende Abwandlung angenommen: die COMP-Position wird nie besetzt, stattdessen werden *alle* Objekte an V^1 adjungiert. Statt der Regeln

```
v1 --> obj_{i>1}, v1
v1 --> obj_1, v0
  v1 --> v0
```

die für Verben mit n Objekten bzw. *keinem* Objekt unterscheiden, gibt es nur noch:

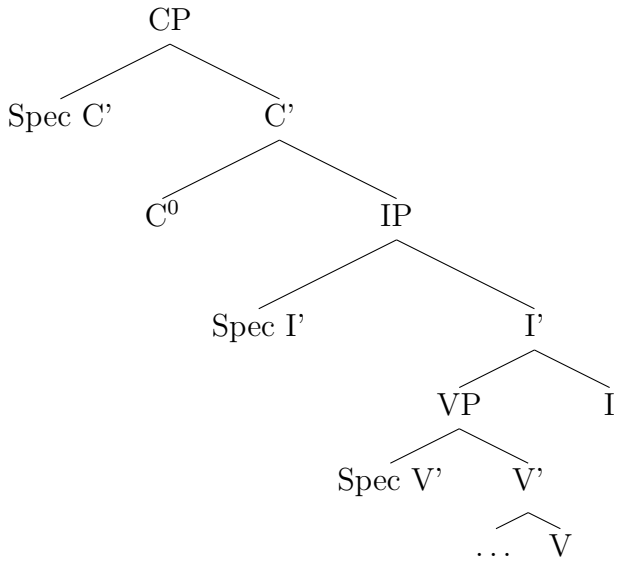
```
v1 --> obj_i, v1
  v1 --> v0
```

Die Tiefe von $\mathcal{P}$ wird hierdurch um +1 erhöht, Nachteile gibt es jedoch nicht. Gemäß bestimmter Modelle ([LEUN04], [STEI05]) ist das Subjekt in der SPEC-V^1-Position der VP basisgeneriert. Wir bevorzugen eine abweichende Theorie ([STEI05, S. 48]), nach der das Subjekt auch direkt in der SPEC-I^1-Position basisgeneriert werden kann, weil dies im Programm das Regelwerk einfacher macht. Welcher Ansatz zu bevorzugen ist, ist von linguistischer Bedeutung, ändert jedoch nichts an der Mächtigkeit des Implementierungsansatzes.

[10] Dies bezieht sich auch auf die Sprachproduktion des menschlichen Gehirns

Die VP steht wiederum in der COMP-Position von I^0. Die SPEC-I^1-Position ist in den meisten Fällen mit dem Subjekt[11] besetzt. Die IP wiederum ist das Komplement des C^0-Heads. Die SPEC-C^1-Position muss nicht besetzt sein. Strukturell gesehen steht die CP in einem Satz somit immer am höchsten und "kennzeichnet" einen Satz als solchen. Eine CP kann aber auch in eine andere CP eingebettet sein (z.B. als Nebensatz), so dass sie die Wurzel eines Teilbaums von $\mathcal{P}$ ist. Es ergibt sich eine rekursive Vorschrift zur Generierung beliebig komplexer, verschachtelter Sätze.

Die hier vorgenommene Beschreibung hat einen Vorteil: es lässt sich eine Art CP-IP-VP-**Rückgrat**[12] von $\mathcal{P}$ erkennen:



Verbindung zwischen S- und D-Struktur: In der Implementierung wird die S-Struktur anhand der vorhandenen Wortordnung ermittelt. Unter Kenntnis und Anwendung des Strukturerhaltungsprinzips, der Module Bewege-α, Lexikontheorie, Rektions- und Bindungstheorie kann ermittelt werden, wie die zugrundeliegende D-Struktur auszusehen hat. Da einige Zuweisungen in der Syntax (z.B. Θ-Rollen) bereits in der D-Struktur des Satzes stattfindet, werden auch Möglichkeiten zur Übertragung von Merkmalen zwischen entfernten Konstituenten eingesetzt.

3.9 Kongruenz

Wenn eine maximale Projektion mit Wurzel $\mu^2 \in_\Delta \mathcal{P}$ von einer anderen Konstituente α regiert wird, so erhält α das "Recht", μ^2 einen Kasus κ zuzuweisen. Möglicherweise wird kein Nachfolger von μ^2 ebenfalls von α regiert. Dann ist μ^2 gefordert, den Kasus an alle seine Nachfolger innerhalb der Projektion zu "verteilen". In Abschnitt 2 wurde dargelegt, dass Worte Träger *morphosyntaktischer Merkmale* sind. Wir nehmen nun an, dass auch lexikalische Kategorien und ihre Projektionen morphosyntaktische Merkmale

[11] Sonderfälle: von A.c.I.-Verben eingebettete IPs oder Konstruktionen aus der *Kontrolltheorie*
[12] Die Idee zu dieser Bezeichnung stammt aus [LENE01]

tragen können. Dann gilt: zwischen μ^2 und allen seinen Nachfolgern muss **Kongruenz** bezüglich κ herrschen, d.h. alle Nachfolgerknoten von μ^2 müssen den entsprechenden Kasus als morphologisches Merkmal tragen. Die Projektionen selbst werden durch die Merkmale nicht beeinflusst – auf diese Weise wird jedoch der **Merkmalstransfer** ermöglicht.

3.10 Modul: Bewege-α

Es wurde bereits beschrieben, wie mit den X-Bar-Regeln ein Grundgerüst eines Satzes aufgebaut werden kann. Konstituenten werden immer an einer spezifischen Stelle basisgeneriert. Man bezeichnet eine solche X-Bar-Struktur auch als **D-Struktur** (**D** für engl. *"deep"*) oder in der deutschen Terminologie auch als **Tiefenstruktur**. Um unterschiedliche Wortordnungen zu ermöglichen, wird im *GBT*-Modell die Operation Bewege-α (auch engl.: Move-α) spezifiziert. α^d ist dabei eine Variable für X-Bar-Kategorien mit Komplexitätsgrad $d \in \{0, 2\}$. Sie steht also für einen einzelnen Knoten (lexikalische Kategorie) oder die Wurzel eines Teilbaums (eine maximale Projektion).

Funktionsprinzip: bewege α in eine in $\mathcal{P}$ *strukturell höher gelegene* Position. Entferne dabei α^d von seiner jetzigen Position, und ersetze es durch eine **Spur**. Letzteres sind phonetisch nicht spezifizierte Konstituenten, die die Bezeichner **e** (für *"empty"*) oder **t** (für *"trace"*) haben. Hänge α^d anschließend an einen Knoten an, welcher in $\mathcal{P}$ eine *geringere* Tiefe hat als seine D-Struktur-Position. Stelle nun eine Verbindung zwischen bewegter Konstituente und Spur durch Koindizierung her, indem α^d und e mit einem identischen Index i versehen werden: $\alpha_i^d \Longleftrightarrow e_i$.

Dieser Vorgang der Bewegung wird auch **Transformation** genannt. Je nachdem, wie die "Landeposition" von α^d beschaffen ist, unterscheidet man **Substitutionstransformationen** und **Adjunktionstransformationen**. Erstere finden dann Anwendung, wenn die Zielposition nur mit einer dort basisgenerierten Spur e^h besetzt ist. Dann wird e^h durch α^d ersetzt, wobei $d = h$. e^h ist ein Platzhalter, um die Zielposition als solche auszuweisen, und den Komplexitätsgrad der substituierenden Konstituente α festzulegen.

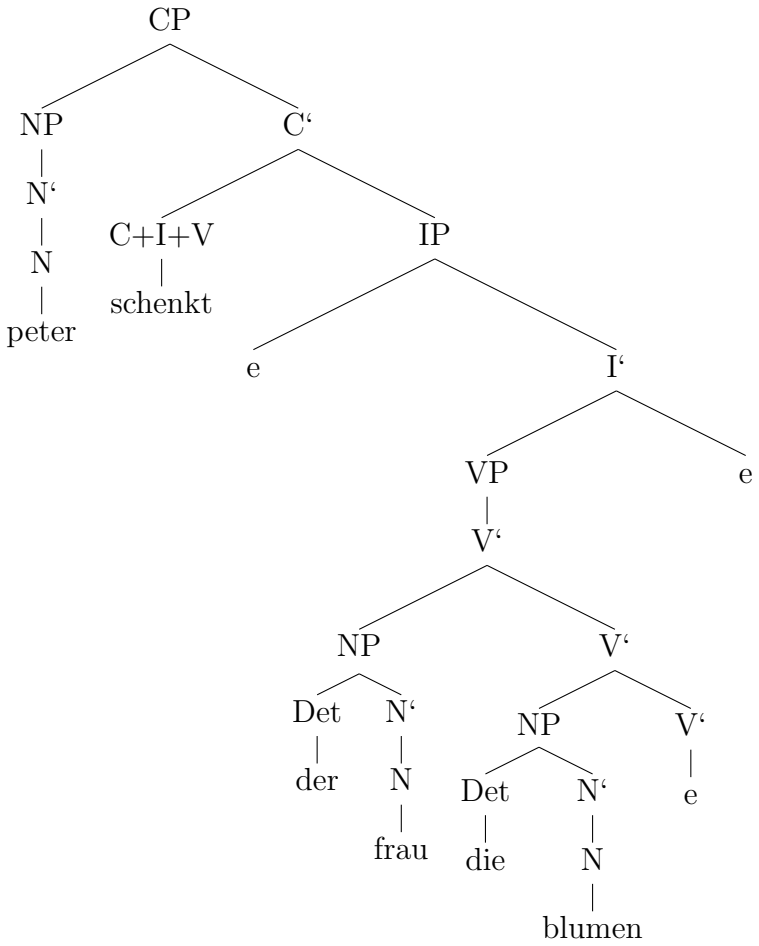
Adjunktionstransformationen finden dagegen dann statt, wenn es keine benötigte freie Zielposition mehr gibt, eine Transformation aus unterschiedlichen Gründen aber trotzdem durchgeführt werden muss. Das Anhängen von α^d findet dann mittels *Adjunktion* statt. Bei Typ-0-Adjunktionen wird z.B. V^0 an I^0 adjungiert, damit V^0 seine Flexionsmerkmale überprüfen kann. Ist $d = 2$, so werden die Transformationen im Rahmen von **Scrambling**[13] eingesetzt. Darunter versteht man eine Veränderung der Reihenfolge von Konstituenten, speziell der Objekte des Verbs. Bsp.:

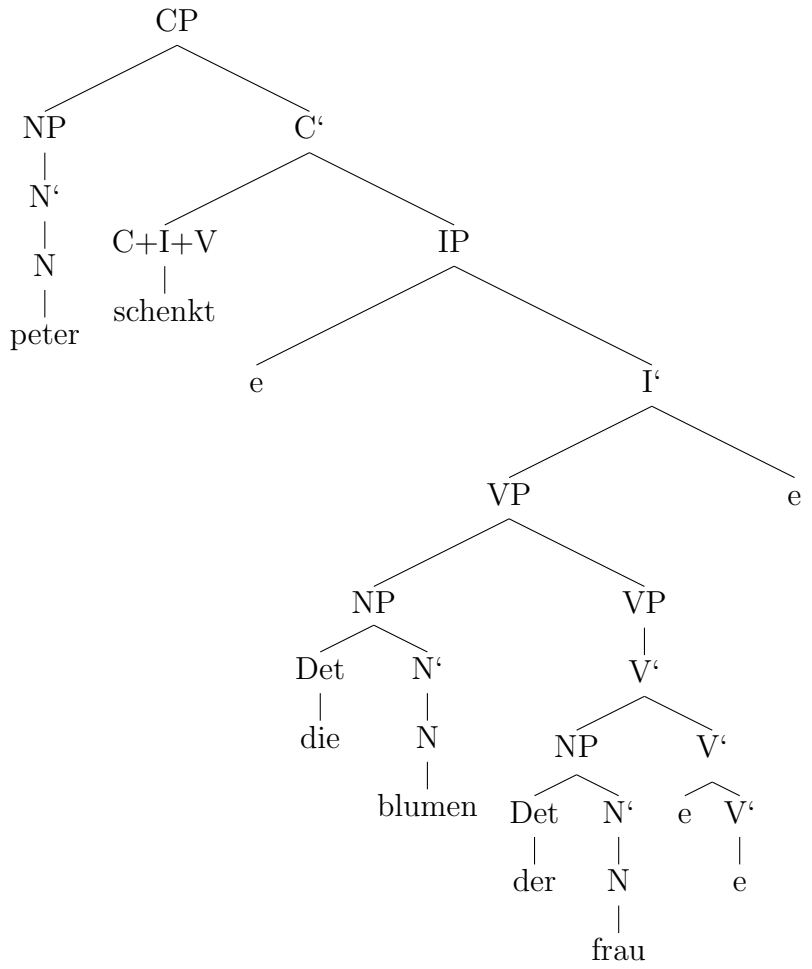
- Weil Peter [$_{VP}$ [in Hamburg] [dem Oberhaupt] [einen Wagen]] schenkte.
- Weil Peter [$_{VP}$ [dem Oberhaupt] [in Hamburg] [einen Wagen]] schenkte.
- Weil Peter [$_{VP}$ [in Hamburg] [einen Wagen] [dem Oberhaupt]] schenkte.

[13]Die Arten von Scrambling, die das Programm unterstützt, sind in Abschnitt 5.4.1 aufgeführt.

- Weil Peter [$_{VP}$ [einen Wagen] [in Hamburg] [dem Oberhaupt]] schenkte.

Exemplarischer Phrasenstrukturbaum für Scrambling von Objekten, wie vom Programm ausgegeben:





Wichtig: Nicht jede Transformation verändert auch die Oberflächenreihenfolge der Worte!

3.10.1 Strukturerhaltungsprinzip

Es werden meist mehrere Transformationen durchgeführt, bis eine Linearisierung abgeleitet ist, die zur Aussprache in die phonetische Form umgewandelt werden kann. Diese Wortstellung/"Vorstufe zur phonetischen Form" wird dann als **S-Struktur** (**S** für *"surface"*) oder **Oberflächenstruktur** bezeichnet. Das Strukturerhaltungsprinzip stellt sicher, dass sich immer eine Beziehung zwischen D-Struktur und allen Repräsentationsebenen der X-Bar-Struktur herstellen lässt: die D-Struktur-Informationen müssen immer erhalten bleiben. Dies geschieht dadurch, dass jede Transformation eine Spur zurücklässt, und die Relation $\alpha \leftrightarrow e$ immer durch Koindizierung markiert wird.

Bewegte Konstituenten können ausschließlich in typenidentischen Positionen landen (**Wiederauffindbarkeitsprinzip**). Auch nach Abschluss einer Transformation ist keine der in Abschnitt 3.4 genannten Regeln verletzt!

Jeder Satz der geschriebenen Sprache (der als Eingabe für das Programm dient) liegt in Form der S-Struktur vor. Wir müssen uns also damit zufrieden geben, dass Prolog

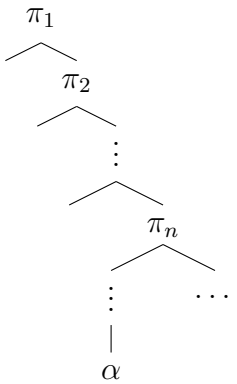
als Eingabe nur die Oberflächen-Wortordnung der S-Struktur zur Verfügung steht. Dies stellt aber aufgrund des Strukturerhaltungsprinzips überhaupt kein Problem dar. Wird eine Konstituente bewegt, so bleibt an ihrer ursprünglichen Position eine Spur ihrer Existenz zurück. Wir können prinzipiell in Prolog überprüfen, ob diese Spur vorhanden ist, und die Beziehung zwischen Satzglied und Spur im Sinne der linguistischen Theorie korrekt ist.

Es gibt weitere **syntaktische Ebenen** ([BDS06, S. 21]), welche dadurch entstehen, dass ausgehend von der S-Struktur mittels *Bewege-α* weitere Konstituenten-Umordnungen vorgenommen werden. Manche Sätze weisen Ambiguitäten auf. Zur Disambiguierung trägt die sogenannte **Logische Form** (*LF*) bei, die eine Schnittstelle zur Semantik bildet. Die Satzglieder werden in eine strukturell prominentere Position bewegt, wobei die "Zielposition" mittels Adjunktion geschaffen wird. Semantische Bedeutungsunterschiede in ambigen Sätzen spiegeln sich in unterschiedlichen logischen Formen desselben Satzes wieder. Die Konstituente mit dem geringeren Abstand zur Wurzel von $\mathcal{P}$ hat bei der Bewertung der Satzbedeutung dann das größere Gewicht. Die Logische Form ist in der Implementierung nicht berücksichtigt, weil sie nicht an der Oberfläche "sichtbar" ist, und auf eine semantische Betrachtung von Eingaben verzichtet wird. Auf *syntaktische* Ambiguitäten wirden aber durch Ausgabe mehrerer unterschiedlicher Phrasenstrukturbäume hingewiesen!

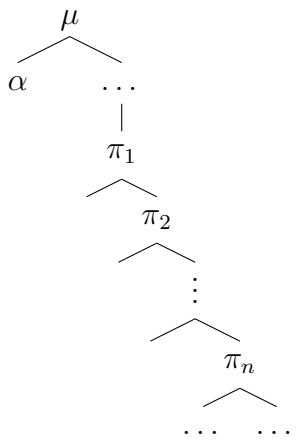
Die als Vorstufe zur menschlichen Lautproduktion dienende **Phonetische Form** wird auch ausgeklammert, da sie nicht im Fokus der Aufgabenstellung ist.

3.11 Inklusion und Exklusion

Inklusion funktioniert wie folgt: Sei π eine verteilte maximale Projektion mit n Segmenten $\pi_1 \dots \pi_i \dots \pi_n$. Dann wird eine Kategorie α genau dann von π inkludiert, wenn gilt: jedes π_i dominiert α. Eine Folge dieser Bedingung ist, dass *tiefe(*α*) > tiefe(*π_i*)*.



Exklusion heisst das genaue Gegenteil: nicht alle, sondern *überhaupt keines* der Segmente π_i dominiert α. O.B.d.A. ist also *tiefe(*α*) < tiefe(*π_i*)*. Folgendes Diagramm veranschaulicht Exklusion:



Merke: es ist gemäß dieser Definition natürlich möglich, dass eine Kategorie *weder* inkludiert noch exkludiert wird! Im Programm werden diese Kategorien als *Non-IE*-Kategorien (Non-Inklusion/Exklusion) bezeichnet.

3.12 Modul: Rektionstheorie

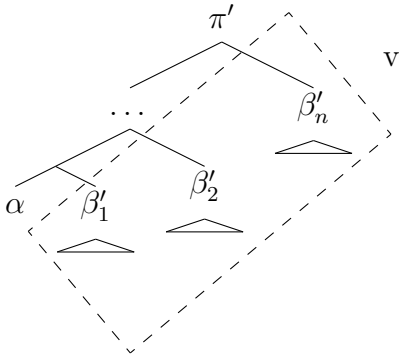
Wie bereits erläutert, ist ein Satz nur dann gültig, wenn er mit den (relativ simplen) Regeln der X-Bar-Struktur generiert werden kann, *und* wenn die X-Bar-Struktur bestimmte Eigenschaften erfüllt. Da die X-Bar-Struktur der Definition eines Baums gleicht, lassen sich auch die folgenden Definitionen auch mittels graphentheoretischer Terminologie erfassen. Die folgenden Begriffe bauen stark aufeinander auf.

3.12.1 c-Kommando

c-Kommando ist wie folgt definiert: ein Knoten α c-kommandiert einen Knoten β (beide aus $\mathcal{P}$) genau dann, wenn gilt:

1. α ist kein Vorgänger von β.
2. β ist kein Vorgänger von α.
3. Jeder Knoten π der Kategorie Typ 2 (maximale Projektion), der Vorgänger von α ist, ist auch Vorgänger von β.

Das c-Kommando ist somit eine asymmetrische Beziehung. α kann einen anderen Knoten nur dann c-kommandieren, wenn dieser sich in einem Teilbaum befindet, dessen jeweilige Wurzel β' ein *Schwesterknoten* oder der Schwesterknoten eines *Vorgängers* von α ist. Zudem muss gelten, dass $\textit{tiefe}(\pi') < \textit{tiefe}(\beta')$, wenn π' die maximale Projektion mit dem kürzesten Abstand zu α ist. Somit wird gewährleistet, dass ein strukturell prominenteres Element (im Sinne maximaler Projektionen) nicht c-kommandiert werden kann. Es gilt zu beachten, dass sich zwei Knoten auch gegenseitig c-kommandieren können.



von α c-kommandierte Elemente

3.12.2 Bindung

Mittels Bewege-α lässt sich bereits die Reihenfolge von Konstituenten verändern. Es gibt jedoch das Problem, dass diese Transformation in keiner Weise restringiert wird, wenn man mal davon absieht, dass das Element in eine strukturell höher gelegene Position bewegt wird. Zu weite Bewegung im Sinne der X-Bar-Struktur produziert jedoch ungrammatische Sätze.

Def.: α bindet β genau dann, wenn gilt:

1. α c-kommandiert β
2. α und β sind koindiziert

3.12.3 Ketten

Eine Folge von Kategorien $\langle \alpha_1, \ldots, \alpha_i, \ldots, \alpha_n \rangle$ nennt man **Kette**, wenn folgende Bedingungen gelten:

1. α_i bindet α_{i+1}.
2. α_n ist eine Θ-Position (*unbeschränkt durch Strukturen*)
3. Ein α_i ist in einer Kasusposition

Satzglieder können also über mehrere Zwischenpositionen $\alpha_{n-1}, \ldots, \alpha_2$ an ihre endgültige S-Struktur-Position α_1 bewegt werden. Jede dieser Zwischenpositionen auf der S-Struktur ist genau wie die Ausgangsposition α_n eine Spur. Dass diese Spuren existieren, folgt aus dem Strukturerhaltungsprinzip. Offensichtlich sind sie nicht mit anderen phonetisch spezifizierten Konstituenten besetzt, weil sich alle nicht bewegten phonetisch realisierten Konstituenten weiterhin an ihrer D-Struktur-Position befinden.

Ketten sind für uns wegen ihrer zentralen Eigenschaft wichtig: die Elemente einer Kette werden *nicht* als einzelne Kategorien verstanden, stattdessen wird die Kette als *eine einzige* auf mehrere Positionen verteile Kategorie angesehen. Jede Information, die

irgendein Element bedingt durch seine Position im Syntaxbaum erhält (Θ-Markierung, Kasus, ...), steht dann in jedem Element der Kette zur Verfügung.

Es folgen weitere Eigenschaften:

- Für jede Transformation einer Konstituente von D-Struktur-Position α_2 zu S-Struktur-Position α_1 gilt: *tiefe(α_1) < tiefe(α_2)*. Dies lässt sich auch auf Ketten von Bewegungen $\langle \alpha_1 \dots \alpha_n \rangle$ übertragen: Für jedes α_i, $i = 1 \dots n-1$ gilt: *tiefe(α_i) < tiefe(α_{i+1})*.
- Wir nutzen diese Eigenschaft für unsere Implementierung: Jedes Element der Kette erhält Attribute, die die über die Kette "verteilten" Informationen repräsentieren. Durch Unifikation der entsprechenden Variablen findet genau diese Verteilung statt.

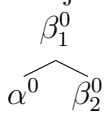
Kettenglieder: Eine Kette $\langle \alpha_1, \dots, \alpha_n \rangle$ kann aus mehr als zwei Elementen bestehen. Dann sind die Kettenglieder $\alpha_2 \dots \alpha_{n-1}$ Spuren, die zur "Zwischenlandung" des bewegten Elements dienen. Subjazenz- und Rektionsbarrieren werden immer paarweise zwischen den Gliedern α_i und α_{i-1} gezählt. Dadurch kann die bewegte Konstituente α_1 auf dem Pfad $\alpha_n \rightsquigarrow \alpha_1$ insgesamt mehr als eine Subjazenzbarriere überqueren.

3.12.4 Subjazenz

Wir nehmen für die deutsche Sprache an, dass IP und NP **Subjazenzbarrieren** darstellen[14]. In einer Kette $\langle \alpha_1, \dots, \alpha_i, \alpha_{i+1}, \dots, \alpha_n \rangle$ darf zwischen α_i und α_{i+1} (also *pro* Bewegung einer Konstituenten) maximal eine Subjazenzbarriere stehen, die α_i exkludiert.

3.12.5 Distinktheit

Ein Element α, welches einer Typ-0-Kategorie angehört, ist **distinkt** von einem Typ-0-Element β, wenn kein Teil von β in einer Kette enthalten ist, die α enthält. Wird ein α^0 nicht bewegt, so ist es auf triviale Weise distinkt von jedem anderen Head γ^0, für das $tiefe(\alpha^0) > tiefe(\gamma^0)$ gilt. Wird α bewegt, aber nicht an β adjungiert, so ist α ebenfalls distinkt von β. Nicht-Distinktheit kann ausschließlich durch Bewegung von α^0 an die Position von β^0 mit anschließender Typ-0-Adjunktion geschehen:



Letzteres tritt ein, weil β^0 nicht leer ist, und wir nichtleere Konstituenten nicht einfach durch bewegte Satzglieder ersetzen können. Es ist zwar denkbar, dass ein Satzglied nach der Adjunktion nochmals bewegt wird – aber nicht, dass zuerst eine Bewegung $t_i \longrightarrow \beta_i^0$

[14] In neueren Theorien (z.B. aus [BDS06]), die nicht berücksichtigt werden konnten, sind es IP und DP. Gemäß der Definition in [LUTZ04] werden Barrieren für Subjazenz genau wie bei der Rektion sogar als relational begriffen. Bei Rektion ist der Begriff der Barriere dann weiter gefasst als bei Subjazenz.

stattfindet, und dann α^0 an die Position von t_i bewegt wird. Distinktheit wird bei der Definition von Rektionsbarrieren in Abschnitt 3.12.7 wieder aufgegriffen.

3.12.6 Selektion

Es gibt drei Konditionen für Selektion, es gilt jeweils genau eine:

1. α selegiert β, wenn α β Θ-markiert.
2. I^0 selegiert VP.
3. C^0 selegiert IP.

Unter Θ-Markierung versteht man, dass der Konstituenten in der COMP-Position vom lexikalischen Head der Projektion ein Θ-Index zugewiesen wird. Dies geschieht bereits auf der D-Struktur. Wir berücksichtigen die semantischen Informationen von Θ-Rollen nicht, jedoch ist eine Erweiterung prinzipiell möglich: Aufgrund der bijektiven Beziehung des Θ-Kriteriums zu den Objekten des Verbs findet die Überprüfung von Subkategorisierungsinformationen und die Θ-Markierung an der gleichen strukturellen Position statt!

3.12.7 Rektions-Barriere

Barrieren für Rektion sind relational definiert. Das heisst, dass kein Knoten *per se* eine Barriere ist, sondern immer nur in Bezug auf eine Rektionsbeziehung zwischen zwei spezifischen Konstituenten. Rektion ist stärker beschränkt als Transformationen. Im folgenden die vollständige Definition. Seien

α **:** ein potentieller Regent

β **:** ein Element, welches von α regiert werden soll

σ **:** eine Projektion, die sich auf dem Pfad von α nach β in $\mathcal{P}$ befindet. Eine potentielle Barriere.

σ^0 **:** der Typ-0-Head von σ.

δ **:** die nächste maximale Projektion, die α dominiert/Vorgänger der Kategorie Typ 2 mit dem geringsten Abstand zu α

δ^0 **:** der Typ-0-Head von δ.

Es ergibt sich folglich die Konfiguration $[_\delta \ldots \alpha \ldots [_\sigma \ldots \beta \ldots]]$. Dann ist σ genau dann eine Barriere zwischen α und β, wenn die folgenden Punkte 1, 2, 3 alle gelten und *entweder* die Bedingung 4.a) *oder* 4.b).

1. σ ist eine maximale Projektion.

2. σ exkludiert α.

3. σ inkludiert β.

4. a) σ ist von keinem Element in $\mathcal{P}$ selegiert.
   b) σ^0 ist distinkt von δ^0, und σ^0 selegiert eine Projektion γ^2, welche β inkludiert oder selber β ist (**Minimalitätsbarriere**).

Diskussion zur Minimalitätsbarriere: Eine Minimalitätsbarriere schützt die Domäne eines Regenten vor potentiellen anderen Regenten. Es gibt jedoch verschiedene Beschreibungen von Minimalitätsbarrieren. In manchen von diesen wird die Möglichkeit eingeräumt, dass auch nichtmaximale Projektionen σ^1 zu Minimalitätsbarrieren werden können. In diesem Fall könnte der potentielle Regent α immer noch die Spezifikatorposition Spec-σ^1 regieren. Dies ist zum Beispiel die Interpretation der ursprünglichen Beschreibung durch Chomsky [MUEL95]. In der Fachliteratur herrscht keinesfalls Einigkeit darüber, welche der beiden Versionen zu bevorzugen ist, es gibt sogar Stimmen, die explizit auf diese Unklarheit hinweisen [UUNL01]. Ich habe mich für die weniger strenge Auslegung nach [LEUN04] entschieden, weil diese Quelle als Leitfaden für die Implementierung dienen soll.

3.12.8 Rektion

Rektion ist ein zentrales Konzept in der hier vorgestellten linguistischen Theorie. Wichtige Schritte wie das Zuweisen eines Kasus vom Verb an seine Argumente wird über Rektion geregelt. Desweiteren wird die Rektion in der Definition des Empty-Category-Principle (siehe 3.12.10) verwendet, welches hilft, Bewegung einzuschränken. Ein Knoten $\alpha \in_\Delta \mathcal{P}$ regiert eine weitere Konstituente, deren Wurzel $\beta \in_\Delta \mathcal{P}$ ist, genau dann, wenn gilt:

1. α c-kommandiert β

2. Es gibt keine Rektionsbarriere auf dem Pfad von α nach β

3.12.9 Strikte Rektion/Antezedensrektion

α regiert β strikt, wenn gilt:

1. α regiert β

2. α und β sind koindiziert. Die Koindizierung kann durch Bewege-α oder durch Θ-Markierung entstanden sein.

Als Antezedentien α kommen in Frage:

- lexikalische Kategorien aus $\mathcal{L}$
- Spuren
- transformationell eingeführte Antezedentien

3.12.10 Empty Category Principle

Jede Spur muss strikt regiert sein. Hierbei ist nicht gefordert, *wer* die Spur strikt regiert. Strikte Regenten können Teil derselben Kette sein, zu der die zu regierende Spur gehört – dann ist die strikte Rektion dank der Koindizierung erfüllt. Oder die Spur kann in einer Position stehen, die von einem lexikalischen Head (z.B. V^0) aufgrund der strukturellen Position zur Θ-Markierung vorgesehen ist. Dann ist die Spur ebenfalls strikt regiert.

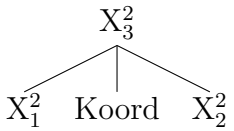
3.13 Koordinierende Konjunktionen

Unter koordinierenden Konjunktionen verstehen wir die Verbindung von zwei oder mehr Konstituenten gleicher Kategorie mittels der Worte **und** und **oder**. Sie gehören der lexikalischen Kategorie **Koord** an. Konjunktionen können alle maximalen Projektionen der Kategorien ($\mathcal{L} \cup \mathcal{F}$) miteinander verknüpfen. Konjunktionen zweier Konstituenten XP und YP mit $X \neq Y$ sind nicht möglich. Die Anzahl der Konjunktionsoperationen ist im Prinzip nicht beschränkt, wie man an folgendem Beispiel erkennen kann:

"Peter oder Maria oder Franz oder Xaver ..."

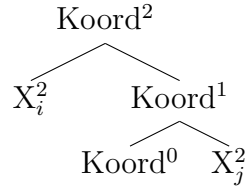
Wie genau Konjunktionen durch die X-Bar-Struktur repräsentiert werden müssen, ist in der linguistischen Forschung noch umstritten. Es gibt zwei Ansätze:

1. **Trinäre Verzweigung.** Laut dieser Darstellung darf man für Konjunktionen von der Regel der binären Verzweigung von X-Bar-Strukturen abweichen, sodass trinäre Verzweigungen erlaubt sind:

   Zwischen X_1, X_2 und X_3 muss Kongruenz bezüglich der morphosyntaktischen Merkmale und der Kategorie herrschen, $X \in (\mathcal{L} \cup \mathcal{F})$. Dieser Ansatz ist deutlich älter ([GREW89]) als der folgende.

2. **Projektion erlaubt.** *Koord* darf Phrasen projizieren ([BDS06, S. 101 ff.]). Die Konstituenten in SPEC- und COMP-Position müssen der gleichen Kategorie angehören und kongruent sein bezüglich ihrer morphosyntaktischen Merkmale.

   Der Junktor hat auch noch grammatischen Einfluss auf die konjugierten Elemente, abhängig davon, mit welchen Kategorien zusammen er verwendet wird. Bei *und* muss die Ausprägung PLURAL des *Numerus*-Merkmals an die maximale Projektion der Konjunktionsphrase übertragen werden (je nach Ansatz also an NP oder $Koord^2$).

3.14 Modul: Bindungstheorie

Die Bindungstheorie befasst sich mit dem Bezug von Pronomen und Anaphern zu ihren jeweiligen Antezedentien. Syntaktisch stehen erstere in der Position von Nominalphrasen und werden deshalb auch als **pronomiale Nominalphrasen** bezeichnet. Für Anaphern sprechen wir analog von **anaphorischen Nominalphrasen**.

A-Bindung: Pronomiale und anaphorische NPs sind an ihre Antezedentien **A-gebunden**. Die Antezedentien sind NPs in Subjekt- oder Komplementpositionen. Letztere werden auch zusammenfassend **Argumentposition** (daher stammt die Abkürzung **A-**) genannt, weil ihnen in der D-Struktur eine Θ-Rolle zugewiesen werden kann.

anaphorisch: *"Peter bemerkt, dass Maria$_i$ sich$_i$ im Spiegel betrachtet."*

pronomial: *"Maria$_i$ weiss, dass Peter sie$_i$ bewundert."*

Der Begriff *"A-Bindung"* soll die referentielle Bindung von derjenigen Bindung unterscheiden, die bei *Bewege-α* auftritt. Die A-Bindung muss bestimmten Bedingungen genügen, sonst ist sie ungrammatisch. Ist eine NP nicht A-gebunden, so wird sie komplementär als **A-frei** bezeichnet.

Rektionsdomänen können relational zu einem beliebigen Knoten α des Phrasenstrukturbaums angegeben werden. Ein Knoten δ ist genau dann eine Rektionsdomäne für ein Element α, wenn gilt:

1. δ dominiert α
2. δ dominiert einen Knoten ρ, welcher α *regiert*.
3. δ dominiert einen Opazitätsfaktor ω, wobei ω ein Subjekt ist.

Das Subjekt wird hier strukturell verstanden, gemeint ist also nicht zwangsläufig das Subjekt des Satzes, sondern eine Nominalphrase in der SPEC-Position. Beispiel aus [LEUN04, S. 188] :

$$[_{NP}\ Ferdis_j\ Buch\ von\ sich_j]$$

Man formuliert üblicherweise: "δ ist eine Rektionsdomäne", präziser wäre es jedoch zu sagen: "Die Rektionsdomäne ist ein Teilbaum $\mathcal{T}_\delta$ des Phrasenstrukturbaums $\mathcal{P}$ mit Wurzel δ". Beachte: nach dieser Definition gilt o.B.d.A.: *tiefe(α) > tiefe(δ).*

Bindungsprinzipien geben an, wie sich eine NP, seine Rektionsdomäne und ein mögliches Antezedens zueinander zu verhalten haben:

Prinzip A Anaphorische NPs sind in ihrer Rektionsdomäne A-gebunden.

Prinzip B Pronomiale NPs sind in ihrer Rektionsdomäne A-frei.

Prinzip C Alle anderen NPs sind überall A-frei.

Weitere Eigenschaften ergeben sich aus den oben aufgeführten Regeln:

- Gemäß der Definition der Rektionsdomänen kann eine anaphorische/pronomiale NP immer nur versuchen, eine A-Bindung zu ihrem Antezedens "nach oben hin" zu suchen.
- Eine Anapher darf ihr Antezedens nicht c-kommandieren.
- Ein Pronomen muss von seinem Antezedens c-kommandiert *werden*.
- Aus Prinzip A folgt, dass eine anaphorische NP zwingend A-gebunden ist.
- Aus Prinzip B folgt, dass eine pronomiale NP nur optional A-gebunden ist. Bsp.:

  "Maria$_i$ weiss, dass Peter sie bewundert." (keine A-Bindung des Pronomens)

- Aus Prinzip C folgt, dass andere NPs in keinem Fall A-gebunden sein können.
- Ist α anaphorisch, so erfolgt die A-Bindung zwischen α und ω: $\omega \overset{\text{A-Bindung}}{\longleftrightarrow} \alpha$

3.15 Modul: Kontrolltheorie

Die Kontrolltheorie ist in [LEUN04] nur kurz vorgestellt und ist sogar in der linguistischen Forschung nur fragmentarisch beschrieben. Aus diesem Grund ist sie in der Implementierung nicht vorhanden.

4 Grundlagen von Prolog

4.1 Prädikatenlogik

Die theoretische Basis von Prolog ist die **Prädikatenlogik erster Stufe**[1]. Die Syntax besteht aus der **Signatur** $\Sigma = (F, P)$ und der abzählbar unendlich großen Menge der **Variablensymbole** V. F ist die Menge der **Funktionssymbole**, und P die Menge der **Prädikatensymbole**. F, P und V sind paarweise disjunkt. Jedem Funktions- und jedem Prädikatensymbol ist eine **Stelligkeit** ≥ 0 zugeordnet. Funktionssymbole mit Stelligkeit 0 werden **Konstantensymbole** genannt; es muss mindestens ein Konstantensymbol in F geben.

Die Menge der **Terme** T ist induktiv über Σ und P definiert. Der "Basisfall"/einfachste Term ist, wenn Konstanten- oder Variablensymbole (entsprechend der Stelligkeit) als Argumente eines Funktionssymbols f eingesetzt werden, z.B. $f(a)$. Variablensymbole werden dabei durch den **Allquantor** $\forall$ oder den **Existenzquantor** $\exists$ quantifiziert. Bei komplexen Termen/im Rekursionsfall werden Terme als Argumente eines Funktionssymbols eingesetzt, z.B. $\forall x : f(x, g(a))$.

Die Menge der **Formeln** wird ebenfalls induktiv über Σ und P definiert. Im Basisfall ist eine Formel ein **Atom**. Ein Atom ist ein Prädikatensymbol, als dessen Argumente Terme eingesetzt wurden, z.B. $\forall x : Q(f(x), g(a))$. Ein **Literal** ist ein negiertes oder nicht negiertes Atom.

Der Rekursionsfall der *Formel*-Definition liegt vor, wenn Literale oder Formeln durch **Junktoren** zueinander in Beziehung gesetzt wurden. Die Syntax jeder Logik enthält zumindest die drei "Basis"-Junktoren $\neg$ (Not), $\wedge$ (And), $\vee$ (Or).

In der Prädikatenlogik gibt es auch noch die sogenannte **erweiterte Syntax**. Diese besteht aus weiteren Konjunktionsoperatoren, mit denen man prädikatenlogische Formeln zueinander in Beziehung setzen kann. Sie machen Formeln leichter lesbar und wesentlich kompakter notierbar. Jede Formel, die die erweiterte Syntax verwendet, kann umgeschrieben werden, sodass nur noch ausschließlich die drei "Basis"-Junktoren gebraucht werden. An der Mächtigkeit der Prädikatenlogik ändert sich durch die erweiterte Syntax nichts, jedoch können Formeln ohne sie exponentiell länger sein ([SSSK06, Kap. 1]).

Aus der erweiterten Syntax benötigen wir die **Implikation** "$\Leftarrow$", welche wie folgt definiert ist:

[1] Formale Definition der Prädikatenlogik: [SSSK06, Kap. 4]

$$A \Leftarrow B$$
$$=$$
$$A \ \vee \ \neg B$$

4.2 Funktionsweise von Prolog

4.2.1 Klauseln

Eine **Hornklausel** ist eine Klausel mit maximal einem positiven Literal. Das Programmieren besteht in der Definition von Regeln. Jede Regel hat die Form einer Hornklausel, wobei es unterschiedliche Arten von Hornklauseln gibt:

Definite Klausel Klausel mit genau einem positiven Literal:

$$A \vee \neg B_1 \vee \cdots \vee \neg B_m$$
$$=$$
$$A \Leftarrow B_1 \wedge \cdots \wedge B_m$$

Hierbei ist A der **Kopf**, und $B_1 \ldots B_m$ der **Rumpf** der Klausel. Der Prolog-Code ähnelt der zweiten, der erweiterten Syntax entstammenden Schreibweise ($A \Leftarrow \ldots$). Da der ASCII-Zeichensatz natürlich prädikatenlogische Symbole nicht unterstützt, wird der Implikationspfeil $\Leftarrow$ als `:-` notiert, und der Konjunktionsjunktor $\wedge$ durch `,` (Komma):

`A :-` B_1`, ...,` B_m

Disjunktionen werden durch `;` (Semikolon) ausgedrückt.

`A :-` B_1`; ...;` B_m

Definites Ziel: auch **Anfrage**, **Query** oder **Goal** genannt. Eine Klausel, die kein positives Literal enthält:

$$\Leftarrow B_1 \wedge \cdots \wedge B_m$$

$B_1 \ldots B_m$ werden **Unterziele** oder **Subgoals** genannt. Der Prolog-Code wird etwas anders notiert:

B_1`, ...,` B_m

Unit-Klausel: auch **Fakt**. Eine Klausel mit Kopf, aber ohne Rumpf:

$$A \Leftarrow$$

Die Notation in Prolog weicht etwas ab:

```
a.
```

Definition von Q: Die Menge aller Klauseln, deren Kopfliteral das Prädikat Q hat. Eine Definition muss nicht nur aus definiten Klauseln bestehen, sondern kann auch Fakten enthalten.

Definites Programm: Menge von definiten Klauseln.

Nachdem ein Regelsatz vom Entwickler festgelegt wurde, kann das Programm ausgewertet werden. Hierzu wird die **SLD-Resolution** eingesetzt.

4.2.2 SLD-Resolution

SLD ist ein Akronym für **Selektions-Funktion, Lineare Resolution, Definite Klauseln**. Sie bildet den theoretischen Unterbau für die Funktionsweise des Prolog-Compilers. Der Benutzer hat eine Menge von Regeln vorgegeben, nämlich das Programm P. Durch das Angeben eines nichtdefiniten Ziels $G = \Leftarrow A_1, \ldots, A_m, \ldots, A_k$ kann der Benutzer eine Abfrage ausführen. Dabei muss $P \cup \{G\}$ durch SLD-Resolution ausgewertet werden. Dies basiert auf folgendem Algorithmus:

SLD-Schritt(G)

1. Selektiere das Atom A_k aus dem Rumpf von G.
2. Wähle eine definite Klausel (also eine Regel) $C = A \Leftarrow B_1, \ldots, B_q$ aus der *Definition* von A_m aus.
3. Ermittle einen allgemeinsten Unifikator θ von A_m und A.
4. Das neue Ziel ist $G' = \theta(\Leftarrow A_1, \ldots, B_1, \ldots, B_q, \ldots, A_k)$
5. Falls möglich: führe ***SLD-Schritt(G')*** aus.

Eine **SLD-Ableitung** von $P \cup \{G\}$ ist eine Folge von *SLD-Schritten*, und wird als

$$G \longrightarrow_{C_1,m_1} G_1 \longrightarrow_{C_2,m_2} G_2 \ldots$$

notiert. Die C_i sind jeweils Varianten einer Klausel C aus Programm P, jedoch mit neuen Variablen, und werden auch als **Eingabeklauseln** im Sinne der linearen Resolution bezeichnet. m_i ist der Index des selektierten Atoms.

Für eine *SLD-Ableitung* kann es verschiedene Ergebnisse geben:

Erfolgreiche Ableitung. Hierbei endet die SLD-Ableitung mit einer leeren Klausel. Dies bezeichnet man auch als **SLD-Widerlegung**. Warum die leere Klausel Erfolg repräsentiert, ist auch intuitiv verständlich. In jedem ***SLD-Schritt*** wird der Kopf einer definiten Klausel in G durch seinen Rumpf ersetzt (mit anschließender Unifikation durch θ). Irgendwann wählt die Suche bei Schritt 2 von ***SLD-Schritt***

aus der Definition des Kopfes A eine Regel aus, die eine Unit-Klausel ist. In einem solchen Fakt ist der Rumpf leer. Somit wird anstelle des Kopfes ein leerer Rumpf eingesetzt. Wenn alle Atome von G "abgearbeitet" sind, kann nur noch die leere Klausel übrig bleiben.

Fehlgeschlagene Ableitungen treten dann auf, wenn eine Ableitung nicht fortsetzbar ist. Verursacht werden sie durch ein Scheitern von Schritt 2 oder 3 des ***SLD-Schritt***-Algorithmus.

Unendliche Ableitungen gibt es, wenn ***SLD-Schritt*** ab seiner k-ten Iteration in seinem 2. Schritt kein A mehr auswählt, für das gilt: A ist ein Fakt.

Korrektheit/Vollständigkeit: Die Korrektheit und Vollständigkeit der SLD-Resolution konnte gezeigt werden (Wiedergabe des Beweises z.B. in [SSSK06]). Für die Implementierung der SLD-Resolution in Prolog ist die Vollständigkeit nicht mehr erfüllt. Wird kein *Occurs-Check* durchgeführt, ist sie i.a. auch nicht mehr korrekt. Meist verzichtet man trotzdem auf seine Anwendung, weil durch ihn mit Performanceeinbußen zu rechnen ist. Es liegt dann am Programmierer, rekursive Termdefinitionen, deren Unifikation zu einer Endlosschleife führen würde, zu vermeiden. In SWI-Prolog kann der *occurs*-Check bei Bedarf eingeschaltet werden [MANL08, S. 40].

Auswertungsstrategie: Das theoretische Modell ist nichtdeterministisch, was zwei Sachverhalte impliziert:

- Es kann in jedem Resolutionsschritt jedes der Atome $A_1 \dots A_k$ gewählt werden.
- Für ein selektiertes Atom A kann jede Regel $A \Leftarrow \dots$ aus der Definition von A ausgewählt werden, um den Resolutionsschritt zu vollziehen.

Eine nichtdeterministische Variante kann man sich so vorstellen, dass alle möglichen SLD-Ableitungen parallel ausprobiert werden. Das impliziert wiederum, dass die explizite Angabe einer SLD-Ableitung ($G \longrightarrow_{C_1,m_1} \dots$) nur *eine einzige* mögliche Ableitung darstellt. Prolog soll aber eine **erschöpfende Suche** (engl.: *"exhaustive search"*, [DOUG94]) durchführen, damit alle möglichen Lösungen gefunden werden können. Backtracking wird benötigt, weil die Suche auf real existierenden Rechnern nur deterministisch sein kann. Bei der SLD-Ableitung wird Backtracking durch zwei Ereignisse ausgelöst (siehe auch [DOUG94, S.121]):

- Der Benutzer kann das Backtracking durch Eingabe eines Semikolons oder eines Leerzeichens auslösen, nachdem bereits ein Ergebnis gefunden wurde. Voraussetzung: die SLD-Ableitung endet mit einer leeren Klausel G_n, und es gibt in der Ableitung einen SLD-Schritt $G_{n-1} \longrightarrow_{C_n,m_1} G_n$, für den gilt: es wurde für das selektierte Atom A_1 (des definiten Ziels G_{n-1}) eine Klausel C ausgewählt und der Resolutionsschritt damit durchgeführt. Jedoch gibt es noch mindestens eine

weitere Klausel C' aus der Definition von A_1, die noch nicht in einem n-ten Resolutionsschritt ausprobiert wurde. Ein solcher Backtracking-Schritt lässt sich im Debug-Modus von Prolog durch das Schlüsselwort `Redo` erkennen.

- Die SLD-Ableitung scheitert im n-ten Schritt, so dass $G_{n-1} \longrightarrow_{C_n,m_1}$ FAIL. Dies erkennt man im Debug-Modus an der Kennzeichnung `Fail`.

In beiden Fällen führen wir dann Backtracking zur Klausel G_{n-1} durch, und setzen die SLD-Ableitung mit der alternativen Eingabeklausel $C'_n \neq C_n$ bis zu ihrem Ende fort. Sind schon alle C' aus der Definition von A_1 ausprobiert worden, so setze zu G_{n-2} zurück, und führe die SLD-Ableitung entsprechend fort. Dieses Vorgehen entspricht genau der Tiefensuche.

Die Implementierung der SLD-Resolution hat folgende Eigenschaften und Einschränkungen (Auswahl, siehe [SSSK06]):

- In einer Klausel selektiert ***SLD-Schritt*** immer das *erste* Unterziel/Atom.
- Die Klauseln aus der *Definition* eines Atoms A im Prolog-Programm werden immer in der Reihenfolge abgesucht, in der sie im Quelltext stehen.
- Wird im 4. Schritt von ***SLD-Schritt*** der Rumpf einer Klausel anstelle des Unterziels in G eingesetzt, so werden die Literale, die den Rumpf bilden, in unveränderter Reihenfolge übernommen.
- Es wird kein **occurs-Check** durchgeführt.
- Es gibt Assert/Retract-Funktionalität.
- Ein Cut-Operator kann Backtracking verhindern. Dies kann die Suche deutlich verkürzen.
- Negation wird als Fehlschlag der Suche interpretiert.
- Erweiterung um Typinformationen.
- Ein Prädikat kann als Funktion definiert werden. Die ersten $k-1$ Argumente sind dabei die Eingabe, das k. Argument ist dagegen die Ausgabe. Durch einen Cut-Operator wird die Klausel deterministisch ausgewertet.

Für die Implementierung mussten die Klauseln so entworfen werden, dass die erzwungene Links-Rechts-Ordnung bei der Selektion der Atome nicht zu Endlosschleifen führt. Ich habe mir diese Ordnung sogar zunutze gemacht, in dem jedem Atom, dessen Auswertung zu einer Endlosschleife führen würde, eine "Kontrollklausel" vorangeht. Nur wenn diese erfüllbar ist, lässt sich die "kritische Regel" auswerten.

SLD-Bäume

SLD-Bäume sind ein formales Werkzeug zur Beschreibung von Suchbäumen in der SLD-Resolution. Insbesondere werden wir sie uns in Abschnitt 5.8 zunutze machen, um eine Verbesserung der Laufzeit zu beschreiben. Die Konstruktion eines SLD-Baums entspricht genau der Suche.

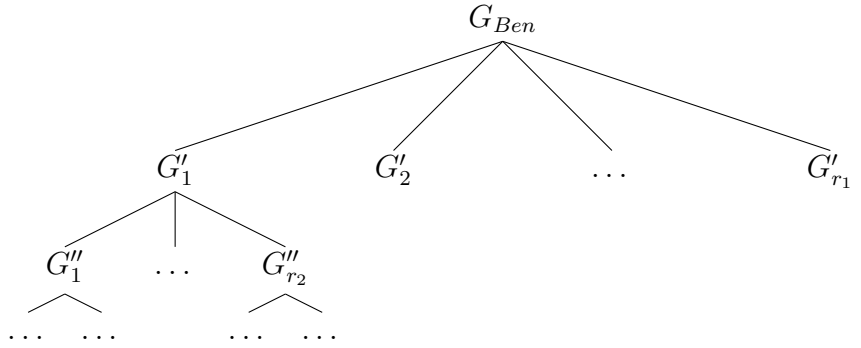
Jeder Knoten des Baums ist mit einem definiten Ziel G markiert. Die Wurzel ist mit G_{Ben} markiert, dem Query, das der Benutzer stellt. Jeder Pfad von der Wurzel zu einem Blatt entspricht einer SLD-Ableitung. Ist ein Blatt mit einer leeren Klausel markiert, so entspricht dies dem erfolgreichen Ende einer SLD-Ableitung. Ist ein Blatt dagegen mit einer nichtleeren Klausel markiert, so entspricht dies einer fehlgeschlagenen SLD-Ableitung. Pfade unendlicher Länge entsprechen unendlichen Ableitungen.

Die Kante zwischen einem mit $G = A_1, \ldots, A_m, \ldots A_k$ markierten Knoten und seinem Nachfolger G' entspricht einem erfolgreichen SLD-Schritt $G \longrightarrow_{C,m} G'$. m ist hierbei der Index des selektierten Atoms, C ist eine Regel aus der Definition von A_m. Der Grad des Knotens G wird durch die Anzahl der Regeln in der Definition von A_m bestimmt.

Ist eine Klausel nichtleer, so wird versucht, die SLD-Ableitung fortzusetzen. Misslingt dies, so gibt es zwei Gründe:

- Für das selektierte Atom A_m gibt es keine Definition, d.h. keine Klausel $\in P$, deren Kopf A_m ist. In Prolog führt dies zu einem Programmabbruch mit Fehler.
- Für das selektierte Atom A_m gibt es für keine Regel $C_i = A_i \Leftarrow \ldots$ einen allgemeinsten Unifikator von A_m und A_i. Dies kann z.B. auftreten, wenn A_m und A_i zwar gleiche Prädikate sind, Unifikation aber bezüglich der Terme in ihren Argumentpositionen unmöglich ist.

Das Aussehen eines solchen Ableitungsbaums hängt also von G_{Ben}, dem Programm P (sozusagen einem "Regelwerk") und der Strategie ab, mit welcher Atome selektiert werden. Sind diese drei Faktoren festgelegt, zeigt der SLD-Baum auf, welche unterschiedlichen Ableitungen es geben kann, wenn man die Auswahl der Regeln für einen Ableitungsschritt variiert. Für die Implementierung wurde die Grammatik P festgelegt. Die Benutzeranfrage G_{Ben} ist fest vorgegeben ("Ist folgende Liste von Worten ein grammatisch korrekter vollständiger Satz/CP/NP/...?"). Die Auswahl der Atome ist von Prolog vorgegeben: in jeder Klausel wird immer das erste selektiert.



4.2.3 Cut-Operator

Der Cut-Operator dient dazu, Backtracking zu verhindern. Er wird im Prolog-Code als Ausrufezeichen (!) im Rumpf einer Klausel geschrieben. Für Literale, die vor dem Cut-Operator stehen, wird kein Backtracking mehr durchgeführt. Effektiv werden dadurch Teilbäume des SLD-Baums abgeschnitten, was man als **Cut** bezeichnet. Diese werden wiederum in **rote** und **grüne** Cuts unterschieden ([BRAT90, S. 142]). Rote Cuts müssen vermieden werden, denn sie schneiden Teile des Suchbaums ab, die erwünschte Ergebnisse enthalten. Grüne Cuts verkürzen die Suche sinnvoll, da in dem entfernten Teilbaum sowieso keine Ergebnisse mehr zu erwarten sind. In der Implementierung wurden grüne Cuts bei einigen Prolog-Klauseln verwendet.

Manche Klauseln sollen deterministisch wie eine mathematische Funktion verwendet werden, d.h. es gibt exakt eine richtige Lösung. Prolog geht die Klauseln in der Menge der Definitionen in der Reihenfolge durch, in der sie im Quelltext stehen. Sobald das Pattern einer Regel deren Auswertung möglich macht, können wir durch den Cut-Operator bestimmen, dass keine weitere Regel aus der Definition mehr ausprobiert werden soll.

4.2.4 Syntaktische Konventionen in Prolog

Bezeichner

Namen können in Prolog aus Großbuchstaben A-Z, Kleinbuchstaben a-z, Ziffern 0-9, dem Unterstrich _ oder sogar nur aus einigen Sonderzeichen bestehen.

Variablen. Ein Variablenname muss immer mit einem Großbuchstaben oder einem Unterstrich (_) beginnen.

Konstanten. Bei diesen müssen die Bezeichner mit einem Kleinbuchstaben beginnen.

Anonyme Variablen (auch Wildcards, Joker) bestehen nur aus dem Unterstrich. Wir setzen eine anonyme Variable immer dann ein, wenn uns der Wert der Variablen an dieser Stelle nicht interessiert, bzw. wir keinen Wert vorgeben wollen. Die Unifikation einer Konstanten oder beliebigen Variablen mit einer Wildcard gelingt immer (Achtung: vergleiche hierzu den folgenden Abschnitt zu Vergleichsoperatoren).

Komplexe Terme (auch: **zusammengesetzte Terme**):

$$\textit{funktor}(\textit{komponente}_1, \ldots, \textit{komponente}_n)$$

Sie haben eine feste Stelligkeit n. Ein **Funktor** (engl.: *"functor"*) mit gleichem Namen, aber *verschiedener* Stelligkeit $m \neq n$ wird als *anderer* Funktor interpretiert. Die **Komponenten** (engl.: *"component"*) an den Argumentpositionen $1 \ldots n$ können Konstanten, Variablen oder wiederum komplexe Terme mit Funktoren sein.

Prädikate sind *komplexe Terme* auf oberster Ebene.

Listen

Listen zählen in Prolog zu den komplexen Termen. Prolog erlaubt es, Listen durch die abkürzende Syntax auszudrücken: `[element`$_1$`, element`$_2$`]`, oder durch das Pattern `[Head|Tail]`. Die leere Liste wird als `[]` notiert. Zahlreiche Operationen auf Listen werden in [MANL08, S. 145 ff.] beschrieben.

Vergleichsoperatoren

Zu unterscheiden sind bei den Vergleichsoperatoren = und ==, bzw. deren Negationen \= und \==. Dabei verwendet man `=` und `\=` man um festzustellen, ob sich zwei Variablen unifizieren lassen, insbesondere wenn eine der Variablen noch nicht instanziiert ist. Für `\=` kann man umgangssprachlich sagen: der Inhalt einer der Variablen ist noch nicht bekannt, man möchte jedoch für die Zukunft unbedingt vermeiden, dass die beiden zu vergleichenden Elemente jemals denselben Wert annehmen. Dies ist z.B. dann wichtig, wenn wir schon vor Beginn einer Suche bestimmte Argumentwerte eines Funktors ausschließen möchten.

Der ==-Operator überprüft, ob der *Inhalt* der beiden verglichenen Elemente zum Zeitpunkt des Vergleichs exakt identisch ist. Der Vergleich scheitert z.B. schon, wenn eine Konstante mit einer nicht instanziierten Variablen verglichen wird.

Die beiden Operatorarten sind für unsere Zwecke nicht austauschbar!

4.2.5 Definite Clause Grammars

Statt dem `:-` Operator des "normalen" Prologs wird der `-->` Operator verwendet. In einer definiten Klausel, die den DCG-Operator verwendet, hat jeder komplexe Term zwei zusätzliche, aber implizite Argumente. Dies sind die sogenannten **Differenzlisten**. Die erste dieser beiden Listen ist die **Eingabeliste**, die zweite ist die **Restliste**. Eine Klausel, die wie folgendes Beispiel definiert ist ...:

```
root(a,b, ...,z) --> lnode(a,b, ...,z), rnode(a,b, ...,z)
```

... läse sich in herkömmlichem Prolog wie folgt:

```
root(a,b, ...,z,Eingabe1,Rest2) :- lnode(a,b, ...,z,Eingabe1,Rest1),
                                   rnode(a,b, ...,z,Rest1,Rest2)
```

Die Differenzlisten werden automatisch vom Compiler hinzugefügt. In *Definite Clause Grammars* sind die Produktionen also fast wie für kontextfreie Grammatiken definiert. Der Unterschied ist jedoch, dass alle Nichtterminale beliebig viele Attribute haben können. "Terminalregeln" sind wie folgt definiert:

```
terminal --> [wort].
```

Sie sind auch in normalem Prolog notierbar:

```
terminal([wort|Rest], Rest).
```

Es ist auch möglich, zwischen den "DCG-Literalen" (die implizite Argumente haben), normale Prologaufrufe zu notieren. Letztere muss man dazu in geschweifte Klammern einfassen:

```
a --> {normal}, b.
```

Klauseln, die den `-->`-Operator verwenden, werden wir ab jetzt als *DCG-Regeln* bezeichnen. Sie werden in der Implementierung ausschließlich dafür verwendet, um die Wortordnung/grammatischen Strukturen eines gegebenen Satzes widerzuspiegeln.

4.2.6 Quelltext

Hier einige nicht zusammenhängende Fakten zur Implementierung:

- Im eigentlichen Programmcode können innerhalb einer Klausel beliebig viele Zeilenumbrüche und Leerzeichen enthalten sein, sogar zwischen den Komponenten eines komplexen Terms/Funktors. Da die DCG-Klauseln recht kompliziert sind, erhöht sich durch Zeilenumbrüche die Lesbarkeit enorm! Jede Klausel wird durch einen Punkt (`.`) abgeschlossen.

- statt der *consult*-Klausel habe ich zum Laden von Prolog-Dateien aus Prolog heraus die abkürzende Listenschreibweise verwendet. Bsp.: wir verwenden `[grammatik]` statt `consult(grammatik)`. Die Dateiendungen `.pl` werden immer weggelassen ([MANL08], S. 21).

- Es ist möglich, aus einem Prologprogramm heraus Unterprogramme zu laden, die sich in anderen Dateien befinden. Dies erhöht die Übersichtlichkeit und Modularität. Einschränkend ist jedoch zu beachten, dass die Menge der Funktoren, die sich in einer zu ladenden Datei befindet, von allen anderen Dateien disjunkt sein muss. Gibt es zwei Dateien, die beide den gleichen Funktor enthalten, so werden nur die Definitionen aus der *zuletzt geladenen* Datei verwendet. Hier sei nochmals darauf verwiesen, dass Funktoren nur über Bezeichner *und* Stelligkeit eindeutig identifiziert werden können!

5 Implementierung in Prolog

5.1 Entwurf der Grammatik für Prolog

In diesem Abschnitt wird erklärt, wie zu einem Eingabesatz ein $\mathcal{P}$ ermittelt werden kann, und wie die notwendigen Eigenschaften von $\mathcal{P}$ überprüft werden können. Der Vorteil von Prolog ist, dass man sich bei der Implementierung weder um eine Suchstrategie noch um die Verarbeitung von Differenzlisten kümmern musste. Der Nachteil gegenüber z.B. objektorientierten Programmiersprachen wie C++/Java ist, dass in jeder Regel immer nur ein kleiner Teil des gesamten $\mathcal{P}$-Graphen betrachtet werden kann – problematisch, weil es viele Beziehungen zwischen nicht-adjazenten Satzgliedern gibt. Auf komplexe Prolog-Funktionen konnte verzichtet werden, es wurden meist nur fundamentale Operationen wie z.B. die Verarbeitung von Listen verwendet. Mit der Bezeichnung *"grammatisch"* sind gültige sprachliche Strukturen gemeint, im Gegensatz zu *"ungrammatisch"*.

5.2 Aufbau des Programms

Der Prolog-Parser besteht aus folgenden Teilen:

- Lexikon
- Grammatik
- Fehleranalyse

5.3 Lexikon

Das Lexikon ist zuerst nach lexikalischen Kategorien, und im zweiten Kriterium alphabetisch nach Wortformen sortiert. Dies spielt für den Prolog-Parser, abgesehen von der Effizienz, keine Rolle – in der Literatur gibt es auch Hinweise darauf, dass das menschliche Gehirn keine alphabetische Ordnung von Lexikoneinträgen vornimmt (z.B. [DOUG94, S. 142/143]). Es soll jedoch eine Betrachtung des Lexikons in einem Texteditor vereinfachen.

Im Lexikon stehen ausschließlich DCG-Regeln, deren Rumpf ein Terminal enthält (in Prolog-Code: Terminal = Liste, deren einziges Element eine Konstante ist). Dies entspricht der Zuordnung eines Wortes zu einer lexikalischen Kategorie und einem Tupel morphologischer Merkmale. Bsp.:

```
n0(_,sg,neu,nom,_,_,_,_,_,_) --> [schiff].
```

Im Abschnitt 2 wurde beschrieben, nach welchen morphosyntaktischen Merkmalen sich die Flexionsformen unterscheiden lassen. In unterschiedlichen lexikalischen Kategorien werden unterschiedliche Attributkombinationen herangezogen, um eine Flexionsform eindeutig definieren zu können. Ich habe einen 10-stelligen Funktor `mp(...)` definiert. *mp* steht dabei für **morphologisches Paket**. Diese Namensschöpfung habe ich gewählt, um auszudrücken, dass die morphologischen Merkmale in einem Tupel zusammengefasst werden. Ein Attribut wird hierbei durch die i-te Komponente von `mp` ausgedrückt. Die Ausprägungen eines Merkmals werden durch verschiedene Konstanten repräsentiert, hier die Auflistung für die ersten sieben Komponenten von `mp`:

Merkmal	**Konstanten**
Person	`1`, `2`, `3`
Numerus	`sg`, `pl`
Genus	`mas`, `fem`, `neu`
Kasus	`nom`, `gen`, `dat`, `akk`
Komparation	`pos`, `kom`, `sup`
Modus	`ind`, `imp`, `kj1`, `kj2`
Tempus	`praes`, `praet`

Die Anzahl der Lexikoneinträge für das Lexem entspricht der Anzahl möglicher Kombinationen einiger *Komponenten von* `mp`/Merkmale. `mp` hat eine nicht variable Stelligkeit, und die Position eines bestimmten Merkmals ist immer fest. Ist uns wortartbedingt die Ausprägung eines Merkmals egal, so wird an der entsprechenden Argumentposition von `mp` statt einer Konstanten die anonyme Variable _ (Unterstrich) notiert. Für keine Wortart sind *alle* morphosyntaktischen Merkmale relevant. Dies würde auch das momentan verwendete System, alle Flexionsformen dezidiert im Lexikon zu speichern, ernsthaft in Frage stellen und unpraktikabel machen. Es ergäben sich pro Lexem mindestens 1728 Lexikoneinträge!

Die restlichen drei Merkmale `Finitheit`, `Verbart`, `TempKombi` haben in stärkerem Maße Einfluss auf syntaktische Strukturen, z.B. darauf wie mehrere Verben zu komplexen, mehrelementigen Tempusformen zusammengesetzt werden können. Sie sind jedoch durch das Lexem eindeutig festgelegt und tragen nicht zur Diversifikation der Flexionsformen bei. Beispiel: das Verb *"schlagen"* bildet seinen Präteritumperfekt (*"hat geschlagen"*) immer mit dem Perfekt-Auxiliar *"haben"*.

Die Zusammenfassung aller Attribute mittels eines komplexen Prolog-Terms hat als weiteren Vorteil, dass man die Attribute eines Wortes durch eine einzige Variable in der Grammatik repräsentieren kann. Dies erhöht die Lesbarkeit des Programms enorm; nur an wenigen spezifischen Stellen muss man ein komplexes Pattern des Funktors notieren, um auf die einzelnen morphologischen Attribute zuzugreifen.

Jeder Lexikoneintrag ist eine Regel im Sinne der *Definite Clause Grammar*. Doppelte Lexikoneinträge, also identische Zeilen in der Lexikondatei, müssen in jedem Fall vermieden werden. Nach Backtracking probiert Prolog alle Klauseln aus einer betrachteten Definition aus. Führt die Verwendung einer Lexikon-Klausel C dazu, dass eine erfolgreiche SLD-Ableitung gefunden werden kann, so wird Prolog, wenn C doppelt vorhanden

ist, exakt dieselbe Teilableitung ein zweites Mal finden. Dann werden zwei identische Ergebnisse ausgegeben, ohne dass aus linguistischer Sicht ein Grund für diese Ambiguität vorhanden ist. Diese Problematik ist jedoch nicht damit zu verwechseln, dass lexikalische oder semantische Ambiguiten die Ausgabe mehrerer identischer Phrasenstrukturbäume verursachen können.

Subkategorisierungen der Verben werden analog zur Definition in Abschnitt 3.7.1 implementiert. Eine Liste der subkategorisierten Objekte ist immer die zweite Komponente des `v0`-Funktors. Bei intransitiven Verben ist die Liste leer:

```
v0(mp(3,sg,_,_,_,ind,praes,fini,voll,is),[]) --> [laeuft].
```

Bei transitiven Verben ist jedes Objekt durch den Funktor `obj` beschrieben, welcher das Tupel repräsentiert. Seine Komponenten sind `Kategorie`, `Kasus` und `Pflicht`:

```
v0(mp(3,sg,_,_,_,ind,praes,fini,voll,ha),[obj(np,dat,m),obj(np,akk,m)])
                                --> [schenkt].
```

Jedes Listenelement entspricht einem vom Verb geforderten Objekt. `obj` hat folgende Komponenten:

ART Die lexikalische Kategorie des geforderten Objekts. Die Ausprägungen dieser Variablen sind `np`, `pp`.

KASUS Der Kasus, welcher dem Objekt vom Verb zugewiesen werden soll.

KONDITIONALITAET Kann die Werte `m` (für *"muss"* oder engl. *"mandatory"*) oder `o` ("optional") annehmen, je nachdem, ob ein Objekt obligatorisch ist oder nicht.

Ansätze, bestimmte strukturelle Positionen in der VP mit einem bestimmten Objekttypen/Kasus zu assoziieren [1], sind für die deutsche Sprache unzulänglich. In der deutschen Sprache gibt es 34 verschiedene, verbspezifische Argumentstrukturen im Bezug auf Art (NP,PP...) und Kasus der Objekte des Verbs [DUDG06, S. 939 ff.]. Listen zu verwenden, ist somit die einfachste Lösung.

5.4 Grammatik

Im Gegensatz zum Lexikon ist die Grammatik statisch. Wenn die Erkennung bestimmter sprachlicher Strukturen fertig implementiert ist, muss die Grammatik nicht mehr modifiziert werden. Dass die Grammatik mit recht wenigen Regeln auskommt, ist ein Ausdruck des Stands linguistischer Forschung.

Die Grammatik unterteilt sich wiederum in zwei Teile:

[1] Bsp.: "COMP-V^0 ist immer mit einer Akkusativ-NP besetzt"

DCG-Regeln definieren die möglichen syntaktischen Strukturen eines Satzes, insbesondere die Wortstellung. Bei seiner Suche operiert Prolog auf ihnen, um zu versuchen, einen Phrasenstrukturbaum zu finden.

Prolog-Regeln dienen zur Kontrolle sprachlicher Strukturen, die mittels der DCG-Regeln gefunden werden können. Sie können auch die Suchtiefe beschränken, besonders um Endlosschleifen zu verhindern.

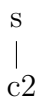
Eine große Schwierigkeit bei der Implementierung war die Suchstrategie von Prolog. Die Auswahl der Regeln erfolgt mittels Tiefensuche mit Backtracking. Manche der Regeln sind linksrekursiv, um das linguistische Modell angemessen zu repräsentieren. Dies führt jedoch dazu, dass die Suche in Prolog in eine Endlosschleife läuft. Problematisch sind auch rechtsrekursive Regeln, wenn diese Regeln eine Spur im Rumpf enthalten. In der X-Bar-Struktur $\mathcal{P}$ sollen Spuren repräsentiert werden, obwohl die Position dieser Spuren in der geschriebenen Sprache nicht repräsentiert ist.

5.4.1 DCG-Regeln zur Wortstellung

In diesem Abschnitt wird das "Herzstück" der Prolog-Grammatik beschrieben: die *DCG-Regeln*, aus denen sich ein $\mathcal{P}$ konstruieren lässt. Eine DCG-Regel bezeichnet eine Prolog-Klausel, bei der Kopf und Rumpf durch den DCG-Operator "`-->`" separiert werden. Sie besteht aus einem *Kopfliteral* und ein oder zwei *Rumpfliteralen*. Der Kopf entspricht der Wurzel eines Teilbaums, die beiden DCG-Literale im Rumpf den Wurzeln von dessen linkem und rechtem Teilbaum. Die entsprechenden Funktoren besitzen die impliziten Differenzlisten-Komponenten. Die Definition einer jeden Klausel könnten zwar aus beliebig vielen DCG-Funktoren bestehen. Durch die Beschränkung der Regeln auf maximal drei Knoten lässt sich die Größe der Definitionen minimieren. Kleinere "Bausteine" lassen sich auf vielfältigere Weise zu grammatischen Konstruktionen zusammensetzen.

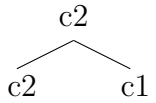
Die Regeln bestimmen, welche Wortstellungen als grammatisch richtig erkannt werden sollen, und in welchen Positionen Transformationen von Konstituenten möglich sind. Auch die Anwesenheit von Spuren wird durch sie repräsentiert.

Sätze

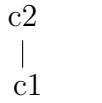


Oberste Regel, die die Wurzel des Matrixsatzes darstellt.

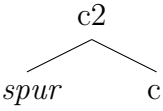
Complementizer-Phrasen



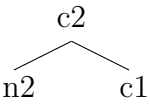
CP, die ins Vorfeld an die SPEC-C^1-Position bewegt wurde.



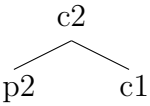
Die SPEC-C[1]-Position ist nicht besetzt. Tritt z.B. bei *deklarativen Konstituentensätzen*[2] auf.



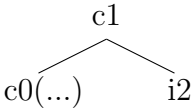
Escape-Hatch für eine Konstituente, welche über die CP hinaus bewegt wird.[3] Ohne eine solche "Zwischenlandeposition" würde bei der Bewegung das Subjazenzprinzip und/oder das ECP verletzt.



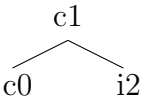
Durch Transformation ins Vorfeld bewegte NP. Die NP kann Subjekt oder Objekt sein.



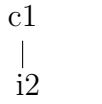
Durch Transformation ins Vorfeld bewegte Objekt-PP. Mit freien Adjunkten ist dies in der Implementierung nicht möglich.



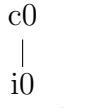
Tritt bei Verbzweitsätzen auf, `c0(...)` entspricht der Kategorie C^0+I^0+V^0.



CP-Kopf-Position ist mit satzeinleitender Konjunktion besetzt; tritt bei *deklarativen Konstituentensätzen* auf.



Sonderfall, bei dem die C^0 nicht mit einer phonetisch spezifizierten Konstituente markiert ist – I^0+V^0 wurde *nicht* an C^0 adjungiert. C^0 trägt jedoch die Merkmale eines *interrogativen Konstituentensatzes*[4].



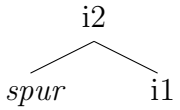
Typ-0-Adjunktion der Kategorie I^0+V^0 an C^0.

[2] $[_{CP}$ *dass ...* $]$

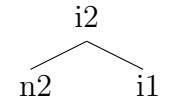
[3] $[_{OBJ}$ *Den Kaffee* $]_{i_1}$, *berichtet Maria,* $[_{CP}$ $\underline{e_{i_2}}$ *habe Peter* e_{i_3} *gekocht* $]$

[4] $[_{CP}$ *Was* $[_{IP}$ *Maria gesagt hat* $]$ $]$

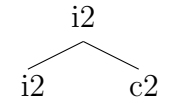
Inflection-Phrasen



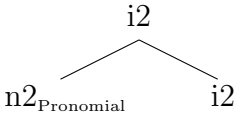
Spur des Subjektes, das an eine höhere Position bewegt wurde.



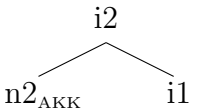
Subjektposition, an der eine NP basisgeneriert wurde.



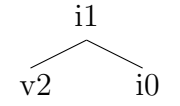
Nebensatz-CP wird aus der von V' dominierten Objektposition an die IP adjungiert. Man sagt auch, die CP steht im *Nachfeld* des Satzes. Wird z.B. bei zusammengesetzten *Perfekt*-Formen erzwungen, um die direkt beobachtbare Oberflächen-Wortordnung zu ermöglichen.[5]



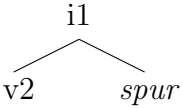
Umgestelltes/vor das Subjekt gezogenes pronomiales Objekt wird an die IP adjungiert.
[6]



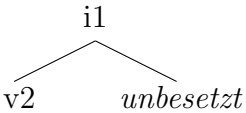
Regent des Subjekts steht oberhalb der IP und weist ihm *Akkusativ* zu, wenn IP eine AcI-Ergänzung ist.



V^0 wurde an I^0 adjungiert, so dass die Kopf-Position der IP phonetisch spezifiziert ist.



Spur der Kategorie I^0+V^0, die an eine strukturell höhergelegene Position bewegt wurde.



Sonderfall: die Kopf-Position der IP ist unbesetzt, sodass die IP kein flektiertes Verb enthält. Tritt auf, wenn IP eine AcI-Ergänzung ist.

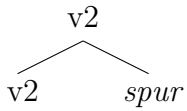
[5] *Peter hat* $[_{IP}$ $[_{IP}$ *... e_i ... gesagt,*] $[_{CP}$ *dass ...* $]_i$]

[6] [*Dass* $[_{IP}$ [*ihn* $]_i$ $[_{IP}$ Peter e_i verrät]], hätte Hans nie gedacht]

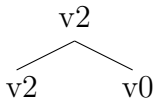
Typ-0-Adjunktion der Kategorie V^0 an I^0.

Verbalphrasen

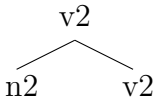
SPEC-V^1 ist hier unbesetzt, da laut [STEI05] alle Objekte unterhalb von V^1 basisgeneriert werden und das Subjekt an der SPEC-I^1-Position.



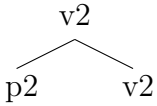
Spur eines flektierten Verbs V^0, das an die VP adjungiert wurde. V^0 wurde zwar hier basisgeneriert, muss aber im Folgenden an I^0 adjungiert werden, um seine Flexionsmerkmale überprüfen zu können.



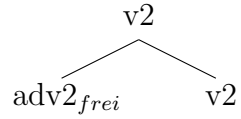
Unflektiertes Verb, das an die VP adjungiert wurde.[7]



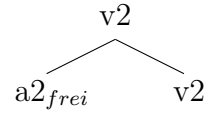
NP-Objekt, welches aufgrund von Scrambling bewegt und an die VP adjungiert wurde.



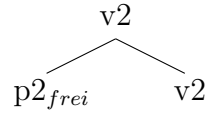
PP-Objekt, welches aufgrund von Scrambling bewegt und an die VP adjungiert wurde.



Freies Adverbial-Adjunkt, an die VP adjungiert.

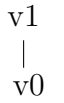


Freies Adjektiv-Adjunkt, an die VP adjungiert.

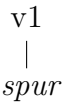


Freies Präpositional-Adjunkt, an die VP adjungiert.

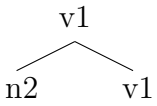
[7] $[_{VP}$ ... gewollt $[_{V^0}$ haben] würde]



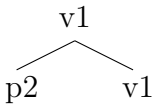
Basisgenerierter Kopf der VP



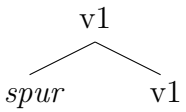
Spur des Verbs V^0, welches bewegt wurde.



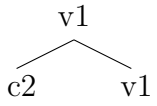
NP an Objektposition des Verbs.



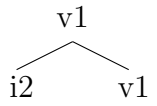
PP an Objektposition des Verbs.



Spur einer bewegten Konstituente an Objektposition.

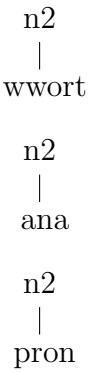


Nebensatz-CP an Objektposition.

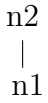


IP an Objektposition des Verbs. Bei AcI-Verben wird eine solche IP zwingend als Objekt gefordert.

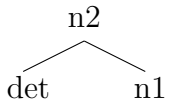
Nominalphrasen



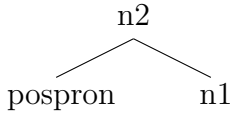
W-Worte, Anaphern und Pronomen werden ebenfalls zu den NP gezählt. Ihre genaue interne Struktur (falls sie eine besitzen), betrachten wir jedoch nicht.



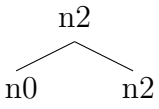
SPEC-N^1-Position unbesetzt



SPEC-N^1-Position mit *Determinator* besetzt.

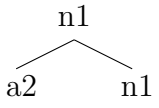


SPEC-N^1-Position mit *Possessivpronomen* besetzt.

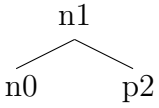


Adjunktion eines Nomens an die NP.[8]

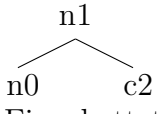
Kopf der NP mit unbesetzter COMP-N^0-Position.



APs werden im Regelwerk immer an N' adjungiert. Beliebig viele Adjunktionen sind möglich[9]. Theoretisch könnten sie auch direkt die COMP-N^0-Position besetzen.

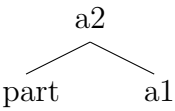


COMP-N^0-Position mit PP besetzt.



Relativsatz-CP in COMP-Position. Eingebettete Relativsätze erhöhen die Laufzeit sehr stark, von wenigen Minuten auf z.T. mehrere Stunden.

Adjektivphrasen



SPEC-A^1-Position mit *Partikel* besetzt (*"zu"*, *"sehr"*, ...).

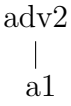
[8] $[_{NP}$ *Karl* $[_{NP}$ *der Große*]]

[9] $[_{NP}$ *der* $[_{N'}$ *schöne, heisse, leckere ...* $[_{N}$ *Kaffee*]]]

SPEC-A^1-Position unbesetzt.

Kopf der AP.

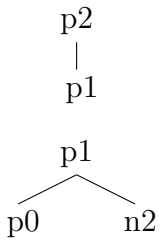
Adverbialphrasen



SPEC-Adv^1-Position unbesetzt.

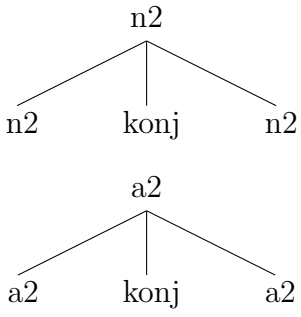
Kopf der AdvP.

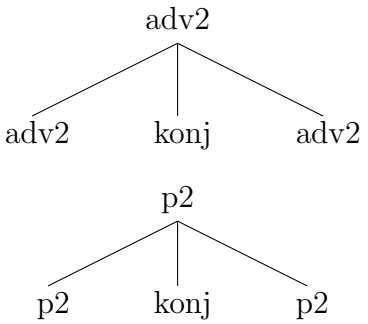
Präpositionalphrasen



COMP-P^0-Position mit NP besetzt. Bsp.: [$_{PP}$ *in* [$_{NP}$ *Hamburg*]]

Konjunktionen von NP, AP, AdvP, PP





Normale, durch geschweifte Klammern {} eingefasste Prologaufrufe, können vor, zwischen oder nach den DCG-Literalen des Rumpfes stehen. Bsp.:

```
dcg1(X1) --> {prol1(X1)}, dcg2(X1), {prol2(X1,X2)}, dcg3(X2), {prol3(X2)}
```

Sie dienen der Beschränkung der Suche und der Einschränkung der Akzeptanz gefundener Strukturen. Das System, wie Werte von Variablen weitergegeben werden, können wir vielfältig nutzen: zum einen können damit Werte, an die die Variable `X1` im Kopf gebunden ist, "an den Rumpf heruntergereicht" werden. Die Variablen `X1` im Rumpf sind zunächst alle ungebunden. Wenn jedoch der Rumpf von ***SLD-Schritt*** in das *Query* eingesetzt wird und Unifikation durchgeführt wird, nehmen sie aber denselben Wert an wie `X1` im Kopf. Zum Anderen können wir Funktoren wie `prol2` wie eine mathematische Funktion verwenden. Die ersten $k-1$ Komponenten werden an Werte gebunden, die eine Eingabe sein sollen (in diesem Schema: `X1`). Die k-te Komponente steht für die Ausgabe (hier: `X2`). Entsprechend der Werte der Eingabe-Komponenten wird aus der Definition von `prol2` eine Regel ausgewählt. Dazu muss der Kopf der Regel mit dem selektierten Atom `prol2` unifizierbar sein. Im Folgenden wird daher die abkürzende Formulierung "Argumente in Funktion einsetzen" verwendet. Falls gewünscht, können Hilfsfunktionen wie `prol2` durch Verwendung des Cut-Operators und disjunkte Patterns sogar deterministisch gemacht werden.

5.4.2 Adjunktion

Adjunktionsstrukturen werden durch die im Abschnitt 5.4.1 genannten Regeln erfasst. Bei Funktoren, die maximale Projektionen repräsentieren, muss auch die Segmentindizierung repräsentiert sein. Dies geschieht durch Einsetzen einer Zahl in die die erste Komponente des entsprechenden Funktors. Das erste Segment trägt immer den Index 1, bei allen weiteren muss der Index um +1 erhöht werden. Dies geschieht durch folgende einfache Funktion:

```
n2(LO) --> {HI is LO + 1}, x, n2(HI).
```

Es ist wichtig, Segmente nach ihrem Index zu unterscheiden, damit nur ein einziges Element der verteilten Kategorie die Erhöhung eines Zählers auslöst, z.B. des Subjazenzbarrierenzählers.

5.4.3 Typ-0-Adjunktion

Bei mehrfachen Adjunktionen $X^0_n + \ldots + X^0_1$ steht immer nur X^0_1 für eine lexikalische Kategorie mit einem phonetisch realisierten Wort. Somit ist nur X^0_1 Teil der Eingabe. Die funktionalen Kategorien $X^0_n \ldots X^0_2$ sind aber nicht *"nichts"*, sondern tragen bestimmte grammatische Eigenschaften, die mit der adjungierten Konstituente interagieren. Wenn wir Klauseln mit Kopf I^0 und C^0 definieren, so können wir mittels eigener Regeln in deren Rumpf die Eigenschaften der adjungierten Kategorie überprüfen. Beispiel: Bei der Adjunktion von V^0 an I^0 ist gewiss, dass V^0 die von I^0 getragenen Flexionsmerkmale aufnimmt/überprüft. Es dürfen aus dem Lexikon also nur Regeln der Form `v0 --> [`α`]` ausgewählt werden, für die gilt: α ist ein flektiertes Verb.

In der Implementierung findet sich diese komplexe Adjunktion auch in den Regeln wieder. Ist der Satz in Verbzweitstellung, so wählt die Suche nacheinander die Regeln `c0 --> i0` und `i0 --> v0` aus. In Verbletztstellung wird die Regel `i0 --> v0` verwendet. In der Regel `i0 --> v0` setzen wir das *morphologische Paket* von `v0` in die Regel `flexion` ein. Diese ist nur dann erfüllbar, wenn das *MP* nicht das Merkmal *Infinitiv* trägt. Auf diese Weise überprüfen wir die von I^0 geforderte Flexion.

Man könnte sich auch vorstellen, `v0` direkt in den Rumpf der Regeln `c1 --> ...` oder `i1 --> ...` zu schreiben, gefolgt von den Regeln zur Überprüfung von Flexion/etc., und die Regeln `c0 --> i0` und `i0 --> v0` gar nicht zu verwenden. Schließlich können wir bei Verbzweitsätzen für die Kopfposition der CP daher immer die Adjunktions-Kategorie $C^0+I^0+V^0$ annehmen, und bei Verbletztsätzen liegt immer die Adjunktions-Kategorie I^0+V^0 vor. Die vorliegende Struktur ist also schon bekannt und auch eindeutig. Diese Eingebung, dass an dieser Stelle Regeln sinnvoll eingespart werden können, trügt. Tatsächlich würde entgegen der Intuition die *gesamte* Suche sogar deutlich länger dauern. Hinzu kommt, dass die tatsächlich gewählte Implementierung linguistisch sogar deutlich sauberer ist, weil sie die Typ-0-Adjunktionsstrukturen nicht ausspart.

5.4.4 Implementierung von koordinierenden Konjunktionen

Da die Frage in der Konjunktionen in der sprachwissenschaftlichen Forschung noch nicht abschließend geklärt ist (siehe Abschnitt 3.13), habe ich mich bei der Implementierung für die erste Variante mit der Dreiverzweigung entschieden. Die zweite ist zwar konstistenter mit der X-Bar-Theorie, macht jedoch das Regelwerk sehr unübersichtlich und ist bezüglich der Laufzeit etwas ungünstiger. Wenn *KOORD* eine Phrase projizieren könnte um zwei XPs zu konjugieren, müssten wir die Fälle Nicht-Konjunktion / Konjunktion aber trotzdem an jeder möglichen Position einer XP unterscheiden. Beispiel für NPs:

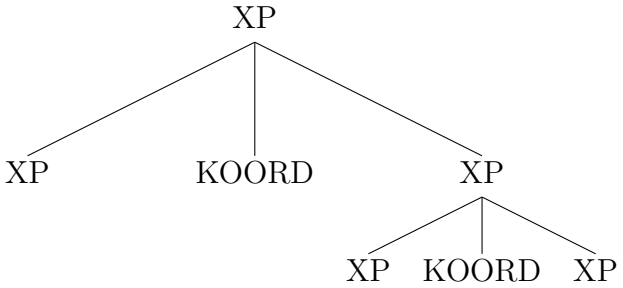
```
c2 --> n2, c1.
c2 --> KOORD_n2, c1.
v1 --> n2, v1.
v1 --> KOORD_n2, v1.
```

Bei trinärer Verzweigung kann diese Unterscheidung jedoch entfallen:

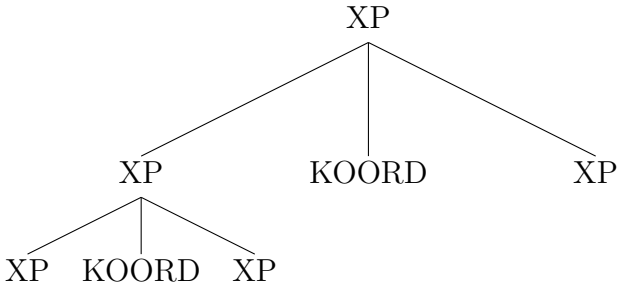
```
c2 --> n2, c1.
v1 --> n2, v1.
```

Momentan sind auch nur Konjunktionen für die Kategorien N, A, Adv und P möglich. Die Konjunktion von C, I oder V würden die Laufzeit sehr stark erhöhen, und es gibt viele Probleme bezüglich der zugrundeliegenden Theorie, die mit der verfügbaren Literatur nicht zu klären waren[10]. Aus diesen Gründen sind sie ausgespart. Für jedes DCG-Literal `x2` im Rumpf einer DCG-Regel, `x` eine lexikalische oder funktionale Kategorie, wird von Prolog eine weitere Regel aus der Definition von `x2` ausgewählt, um die Suche fortzusetzen. Ob an der Position von `x2` eine "einfache Phrase" steht, oder die Projektion *XP* das Ergebnis einer Konjunktion ist, spielt keine Rolle bezüglich seiner grammatischen Umgebung. Ob beispielsweise eine "einfache" NP in der Objektposition von V^0 steht oder eine $[_{NP}$ NP_1 KOORD $NP_2]$, ist für die Zuweisung von Kasus und Θ-Rollen irrelevant. Zwar sind theoretisch beliebig viele Konjunktionsoperationen möglich, die Implementierung bleibt jedoch auf eine einzige beschränkt. Der `x2`-Funktor benötigt lediglich eine Komponente `IsConjunction`, um $\mathcal{T}_{\texttt{x2}}$ als Konjunktionsstruktur zu kennzeichnen. Die Variable `IsConjunction` kann die Werte `conjunction` oder `noConjunction` annehmen:

Es wäre zwar möglich, auch Rechtskonjunktion zu erlauben:



Sobald wir dagegen eine Regel definieren würden, die Linkskonjunktion erkennt, würde Prolog bei der Suche in eine Endlosschleife laufen.



[10]Bsp.: An eine VP kann ein Verb adjungiert werden. Wenn man zwei VPs konjugiert – wie genau ist dann die Konjunktion durchzuführen?

Für einen Satz wie z.B. *"Peter oder Hans und Xaver"* gibt es mehrere Möglichkeiten der Klammerung bezüglich Konjunktion. Prolog könnte dann aber nur *"Peter oder [Hans und Xaver]"* erkennen. Ist man mit Prolog nicht vertraut, ist es nicht ersichtlich, warum die ebenfalls grammatische Klammerung *"[Peter oder Hans] und Xaver"* nicht ebenfalls als Lösung gefunden wird. Wir bevorzugen letztlich grammatische Präzision gegenüber einer größeren Vielfalt erkennbarer grammatischer Strukturen.

5.4.5 Prolog-Regeln

Diese Regeln sind in "normalem" Prolog definiert, weil die impliziten Differenzlisten-Argumente unerwünscht sind. Sie unterteilen sich wiederum in zwei unterschiedliche Funktionalitäten:

- Regeln, die die Prolog-*Suche* beschränken. Sie stehen immer ganz am Anfang des Rumpfes.
- Regeln, die für eine *gefundene* Phrasenstruktur kontrollieren, ob sie grammatikalisch korrekt ist. Meist sind dies die Literale am Ende des Rumpfes.

5.4.6 Suche beschränken

Bei nichtrekursiven DCG-Regeln `y --> x, z` ohne Spuren gibt es bei der Prolog-Suche keine Endlosschleifen[11]. Sei hierbei der Funktor `x` ein Symbol, welches für eine lexikalische Kategorie steht, und $[\alpha_1, \ldots, \alpha_n]$ die Eingabeliste. Dann überprüft Prolog, ob im Lexikon ein Eintrag der Form `x --> [`α_1`]` vorhanden ist. Als "unsichtbares" Argument erhält `x` die Eingabeliste und gibt eine Restliste zurück: `x(`$[\alpha_1, \alpha_2, \ldots, \alpha_n]$`,`$[\alpha_2, \ldots, \alpha_n]$`) --> [`$\alpha_1$`]`. Es existiert genau dann keine passende Regel, wenn jede Regel in der Definition von `x` die Form `x --> [`β`]` mit $\alpha_1 \neq \beta$ hat.

Dann ist auch das Unterziel `x` nicht erfüllbar, und die Regel `y --> x, z` kann nicht ausgewählt werden. Im Anschluss findet Backtracking statt, und es wird eine andere Regel mit *Kopf* `y` ausgewählt.

Sobald die zurückgegebene Restliste leer ist, wird vor dem Backtracking keine weitere DCG-Regel mehr ausgewählt, die Suche ist zumindest im Hinblick auf DCG-Klauseln beendet:

`x([`α_n`],[]) --> [`α_n`]`

Wegen der Adjunktion sind auch rechtsrekursive Regeln der Form `y --> x, y` erlaubt und problemlos anwendbar. Erst wenn `x` als erfüllbar erkannt wurde, wird die Suche beim `y` im Rumpf fortgesetzt. Da x nur so oft erfüllbar sein kann, wie es passende Elemente in der Eingabeliste gibt, und Differenzlisten nicht unendlich lang sind, kann es keine Endlosschleifen geben.

Endlosschleifen treten jedoch bei folgenden zwei Konstellationen auf:

[11] Weitere Informationen zur Entstehung von Endlosschleifen in Prolog: [WARR99]

Regeln mit Spuren Es gibt eine rekursive DCG-Regel `y --> spur, y` und eine Regel `spur --> []`. Letztere Regel kann unabhängig von der Eingabeliste ausgewählt werden, die Restliste ist mit der Eingabeliste identisch: `spur(`$[\alpha_1, \ldots, \alpha_n]$`,`$[\alpha_1, \ldots, \alpha_n]$`) --> []`. Das Prädikat `spur` ist auf diese Weise immer trivial erfüllbar. Die Suchstrategie sieht vor, dass sukzessive alle Regeln aus der Definition von `y` ausgewählt werden. Früher oder später wird somit auch wieder die Regel `y --> spur, y` ausgewählt werden, was einer Rekursion entspricht. Die Suche endet also nie!

linksrekursive Regeln haben die Form `y --> y, x`. Selbst wenn `x --> [`β`]`, Eingabeliste $[\alpha_1, \ldots, \alpha_n]$ und $\alpha_1 \neq \beta$, so wird die Regel `y --> y, x` trotzdem immer wieder ausgewählt. Daran ist die Tiefensuche in links-rechts-Ordnung "schuld". Die problematische Regel gehört zur *Definition von* `y`. Wenn das `y` im Rumpf ausgewertet werden soll, wird auch immer wieder die problematische Regel ausprobiert. Die Suche steigt immer weiter ab, ohne dass es ein Abbruchkriterium gibt.

Beide Arten dieser "problematischen" Regeln sind durch die linguistische Theorie vorgegeben. Durch das Verbot linksrekursiver Regeln oder von "alleine stehenden" Spuren im Rumpf wäre zwar das Problem von Endlosschleifen gelöst. Wir würden dann allerdings wesentlich mehr Regeln benötigen als jetzt, und diese wären auch nicht so einheitlich wie das in Abschnitt 5.4.1 beschriebene "Baukastensystem". Das Regelwerk wäre insgesamt nur schwer zu verwalten und zu modifizieren. Für die Implementierung habe ich ein Verfahren entwickelt, mit dem auch linksrekursive Regeln und "Spur-Regeln" beibehalten werden können. Es verhindert Endlosschleifen, stellt einen Bezug zwischen D- und S-Struktur her, und repräsentiert Transformationsketten.

Nicht basisgenerierte Konstituenten können Nominalphrasen, Präpositionalphrasen, W-Pronomen/W-Worte (*"Wer"*, *"Was"*, ...), Pronomen oder Anaphern sein. Die morphologischen Attribute der Konstituente sind durch ein Tupel repräsentiert, welches wir mit der Variable `MP` bezeichnen.

Aus dem *Strukturerhaltungsprinzip* und *Move-*α folgt, dass es zwischen einer bewegten Konstituente und einer Spur eine bijektive Beziehung gibt. Das bedeutet:

$\Longrightarrow$ Wir können aufgrund des X-Bar-Schemas für jede Konstituente bestimmen, ob sie an ihrer Position basisgeneriert ist, oder "ortsfremd". Ist sie nicht basisgeneriert, so muss sie durch Transformation an diese Stelle gelangt sein, und es muss an strukturell tiefer gelegener Stelle eine Spur geben.

$\Longleftarrow$ Eine Spur ist im Allgemeinen [12] immer die Folge einer Bewegung. Sie darf nur existieren, wenn es auch eine zugehörige Konstituente gibt.

Die Grundidee ist: Sei das Subgoal x ein Funktor einer DCG-Regel, welcher eine Typ-1- oder Typ-2-Projektion π darstellt, also z.B. `v1`, `v2`, `i2`, etc. Eine Komponente dieses Funktor ist eine 5-elementige Liste. Eine Kette $\langle \alpha_1, \ldots, \alpha_n \rangle$ kann man gemäß der

[12] Es kann auch Spuren auf generell nicht besetzten Positionen geben. Die Auswahl einer entsprechenden Regel bei solchen Spezialfällen verursacht jedoch keine Endlosschleifen.

Bindungstheorie als eine Reihe von Pfaden von Element α_i zu α_{i+1} sehen. Wenn π auf dem Pfad von α_i nach α_{i+1} liegt, dann kann in der Liste von x vermerkt werden, dass π Teil der Kette ist. Die Liste hat 5 Elemente, weil es mehrere voneinander unabhängige Ketten in $\mathcal{P}$ geben kann. Dementsprechend wird sie durch die Variable `ChainBundle` (engl. *"bundle" = "Bündel"*) bezeichnet.

Für die Regel `initChainBundle(X)` wird `X` nach Auswertung an die Liste der leeren Positionen gebunden:

```
X = [empty,empty,empty,empty,empty]
```

Wird eine DCG-Regel ausgewählt, in der eine bewegte Konstituente auftritt, so wird mittels `openChain` eine Listenposition mit Informationen über eine neue Kette eröffnet. Statt mit `empty` wird die Position im Listenterm mit dem 8-stelligen Funktor `c(...)` (*c* für engl. *"chain"*) besetzt. `c` soll ein 8-Tupel von Ketten-Eigenschaften repräsentieren. Die Komponenten von `c` werden mit folgenden Variablen bezeichnet:

`ART :` Kategorie der bewegten Konstituente

`MP :` Das *Morphologische Paket* = das Tupel morphosyntaktischer Merkmale, die alle Worte des bewegten Satzteils tragen.

`IEstatReg :` Wird der potentieller Regent (im Sinne der *strikten Rektion*) von der letzten maximalen Projektion exkludiert?

`IEstatBeta :` Wird die regierte Konstituente von der zuletzt betrachteten maximalen Projektion inkludiert?

`Selektion :` Ist die letzte maximale Projektion σ^2 selegiert?

`Distinktheit :` Ist σ^0 (= Kopf der letzten maximalen Projektion) distinkt von δ^0 (= Kopf der nächsten maximalen Projektion, die den Regenten dominiert)?

`SubjazenzBknoten :` Anzahl der überschrittenen Subjazenzbarrieren. Muss ≤ 1 sein.

`SigmasCount :` Anzahl der überschrittenen Rektionsbarrieren. Muss bei einem grammatisch korrekten Satz 0 sein.

Zu beachten gilt es, dass `ART` und `MP` "statische" Eigenschaften der Kette sind. Die Variablen werden durch `openChain` sofort an Werte gebunden und werden auch nicht mehr modifiziert. Alle anderen Komponenten beschreiben relationale Eigenschaften der Kette, die immer nur für eine bestimmte Position in $\mathcal{P}$ gelten. Wir benötigen sie, um *Bindung* und *strikte Rektion* implementieren zu können. Die strikte Rektion ist das Instrument der linguistischen Theorie, um das *ECP* erfüllen zu können. Die DCG-Regel, die das Verb an seiner S-Struktur-Position repräsentiert, "sieht" die Objekte des Verbs [13] nicht. Aus diesem Grund müssen sowohl das *MP* mittels `MP` als auch die Kateogorie

[13] genauer: die Funktoren, die die Wurzeln der Objektteilbäume darstellen

mittels der `ART`-Eigenschaft von `c` "übertragen" werden. In Prolog gibt es keine Unterscheidung zwischen Zuweisung und Kongruenz, so dass man auch sagen kann: an der D-Struktur-Position α_n überprüfen wir, ob die die Kongruenz zwischen `MP` und `ART` aus dem `c`-Term und dem vom Verb geforderten Kasus und Kategorie vorhanden ist. Dies entspricht trotzdem der Definition der Ketten aus Abschnitt 3.12.3 als verteilter Kategorie: `MP` und `ART` stehen allen Elementen $\alpha_1, \ldots, \alpha_i, \ldots, \alpha_n$ der Kette zur Verfügung.

`openChain` steht immer in S-Struktur-Positionen von $\mathcal{P}$. Nachdem `openChain` oder eine andere Funktion einen modifizierten Kettenlistenterm zurückgeben, kann dieser einem Funktor `x` des Rumpfes einer DCG-Regel zugeordnet werden: Die Variable `CB_Ausgabe`, die die letzte Komponente von `openChain`, `closeChain` usw. bezeichnet, wird in `x` eingesetzt: `x(..., CB_Ausgabe, ...)`. Auf der Suche, die immer tiefere Strukturen von $\mathcal{P}$ ausprobiert, müssen beim Abstieg Unterterme der Liste an neue Werte gebunden werden, weil sich die zu speichernden Informationen ändern. Es ist insgesamt einfacher sich vorzustellen, dass die Liste im Phrasenstrukturbaum von Ebene zu Ebene "heruntergereicht" und an bestimmten Punkten modifiziert wird.

Jede DCG-Regel, die eine Spur enthält, enthält auch die Regel `closeChain`. Die Spur muss ein Überbleibsel von *Bewege-α* sein. `closeChain` überprüft, ob die bisher gesammelten Informationen in einem spezifischen `c`-Unterterm der Kettenliste plausibel sind. Der Rückgabewert ist die Liste, in der die Kettenposition freigegeben wurde: statt des `c`-Funktors steht dort wieder die Konstante `empty` als Platzhalter. Ist eine Kettenposition in der Liste bereits leer, so kann `closeChain` nicht erfüllt werden – die ganze DCG-Klausel ist damit nicht erfüllbar. Dies ist exakt der Mechanismus, welcher die Entstehung von Endlosschleifen unterbindet. Wurde z.B. eine kritische Regel `v1 --> spur, v1` ausgewählt, so wird die Kette geschlossen. In der folgenden Suche kann `v1 --> ...` nicht nochmal verwendet werden; es gibt keine passende offene Kette mehr.

Abschluss der Ketten: Das *Rückgrat* des Phrasenstrukturbaums ist ein Pfad von der Wurzel CP bis zur *Kopf*-Position der eingebetteten VP : [CP .... [IP .... [VP ..... V/e] ..] ..]. Die Kopfposition der VP ist grundsätzlich in jeder Satzstruktur der Knoten mit der größten Tiefe. Jede Kette kann nur endliche Länge haben. An dieser strukturell tiefsten Position muss gesichert sein, dass alle Ketten, die im Verlauf in höhergelegenen Positionen eröffnet wurden, auch wieder geschlossen wurden. Das Prädikat `terminateChainBundle` wird ausgewertet und ist nur dann erfüllbar, wenn als sein Argument die Liste der leeren Ketten-Positionen eingesetzt wird.

Zusätzlich zu diesen vier Funktionen gibt es noch *weitere Regeln*, welche die Ketteninformationen "unterwegs" verändern können. Die im Folgenden vorgestellten Funktionen beeinflussen ausschließlich die Komponenten der Funktoren `c` in der Kettenliste; die Konstanten `empty` bleiben unverändert. Dies bringt zum Ausdruck, dass Barrieren in starkem Maße relational sind. Sie alle werden nach folgendem Prinzip angewandt, wobei *CB* ein Bezeichner für das "Kettenbündel" ist, *dcg* ein DCG-Funktor und p_i eine Prolog-Regel zur Veränderung der Ketteninformationen:

$$dcg_1(CB_1) \longrightarrow \{p_1(CB_1,CB_2),\ p_2(CB_2,CB_3),\ \ldots,\ p_k(CB_{k-1},CB_k)\},\ dcg_2(CB_k),\ \ldots$$

Durch die feste Auswertungsreihenfolge werden die Variablen $CB_1 \ldots CB_k$ sukzessive an Listen gebunden, bevor die Auswertung mit *dcg₂(CB_k)* fortfährt.

Exklusion

Sei $\langle \alpha_1, \ldots, \alpha_i, \alpha_{i+1}, \ldots, \alpha_n \rangle$ eine Kette. Für jeden Knoten μ auf dem Pfad von α_i nach α_{i+1} lässt sich bestimmen, ob α_i von der maximalen Projektion μ exkludiert wird. Dies ist wichtig, um Subjazenz und strikte strikte Rektion zu überprüfen. In der Liste der Ketten sind Informationen über offene Ketten enthalten. Die "Eröffnung" einer Kette K_i erfolgt durch eines seiner Elemente α_{i_j}. Wenn wir eine maximale Projektion μ^2 bei der Suche überschreiten, sind alle Konstituenten, die die Ketten $K_1 \ldots K_n$ in der Kettenliste eröffnet haben, automatisch von μ^2 exkludiert. Die Funktion `exklusionsHorizont` durchläuft die Kettenliste, und setzt für jeden Funktor `c` die Komponente `IEstatReg` auf den Wert `exkludiert`. Die Klausel `exklusionsHorizont` muss ausgewertet werden, bevor wir ein DCG-Literal des Rumpfes auswählen können.

Inklusion

Sei $\langle \alpha_1, \ldots, \alpha_i, \alpha_{i+1}, \ldots, \alpha_n \rangle$ eine Kette. Für jeden Knoten μ auf dem Pfad von α_i nach α_{i+1} lässt sich bestimmen, ob α_i von der maximalen Projektion μ^2 inkludiert wird. Dies ist wichtig, um *strikte Rektion* zu überprüfen. Das Prinzip ist vergleichbar mit dem von `exklusionsHorizont`. Bei Auswertung der Regel wird in der Liste der Kettenelemente in die Kompenente `IEstatBeta` die Konstante `inkludiert` eingesetzt.

Selektion

Sei α ein potentieller Regent, und β eine potentiell regierte Konstituente. Wenn Inklusion von β und Exklusion von α gegeben sind, so kann eine maximale Projektion σ^2 durch fehlende Selektion zur Barriere werden. Eine Konstituente ist in jedem Fall selegiert, wenn sie z.B. vom Verb eine Θ-Rolle erhält. Selektion kann jedoch auch strukturell verstanden werden. Das heisst, dass bestimmte Konstituenten allein aufgrund ihrer Position in der X-Bar-Struktur selegiert sind. Evtl. müssen diese beiden Arten von Selektion unterschieden werden, strukturelle Selektion ist dann als "schwächer" anzusehen. Für die Implementierung nehmen wir an, dass jede IP durch C^0, und jede VP durch I^0 strukturell selegiert ist. Für die Segmente $\mu_2^2 \ldots \mu_n^2$ einer maximalen Projektion μ^2 gilt, dass sie den Selektionsstatus von Segment μ_1^2 erben. Um Selektion als relationale Eigenschaft auszudrücken, wird vor dem "Überqueren" eines selegierten Knotens die Regel `selegiert` auf die Kettenliste angewandt. Die Komponente `Selektion` erhält als Wert die Konstante `selegiert`[14].

Einen Knoten μ^2 mittels der Regel `selegiert` als *selegiert* zu vermerken, ist im Bezug auf Bewegung nur dann nötig, wenn eine Konstituente α aus $\mathcal{T}_\mu$ herausbewegt wurde,

[14] Eine Verwechslung zwischen dem Funktor `selegiert` und der Konstante mit dem gleichen Bezeichner ist seitens Prolog ausgeschlossen!

sodass gilt: *tiefe*($\alpha_{\text{S-Struktur}}$) < *tiefe*($\mu$). Im Deutschen ist es meist ungrammatisch, Konstituenten aus einer NP heraus zu bewegen. Beispiel [LEUN04, S. 157]:

$$*\text{“}\textit{Was}_j \textit{ ist Ferdi } [_{NP} \textit{ der Meinung } [_{CP} \textit{ hat Olli } \text{e}_j \textit{ getrunken }]]\text{”}$$

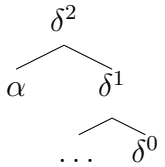
Zudem gilt, dass eine NP_i immer selegiert ist, sofern sie nicht von einer NP_j mit *tiefe*(NP_j) < *tiefe*(NP_i) dominiert wird. Dies liegt daran, dass NP_i sonst immer als Subjekt oder Objekt basisgeneriert wird. Somit ist NP_i sogar durch Erhalt einer Θ-Rolle selegiert.

Auch wenn es in der Implementierung nicht genutzt wird, gibt es die Möglichkeit, eine Projektion μ^2 als *nicht selegiert* zu markieren, indem man die Regel `nichtSelegiert` auf die Kettenliste anwendet.

Distinktheit

Sei α ein potentieller Regent, β eine potentiell regierte Konstituente, und δ die maximale Projektion, welche α unmittelbar dominiert. Wenn nicht fehlende Selektion einen Knoten σ^2 zur Barriere werden lässt, so kann σ^2 immer noch durch Distinktheit zur Barriere werden. Es stehen uns drei Regeln zur Verfügung: `distinkt`, `nichtDistinkt` und `miniBarrSchutzAufheben`. Alle drei beeinflussen die Komponente `Distinktheit`.

Die `Distinktheit`-Komponente wird zunächst mit dem Platzhalter `pre` besetzt, wenn `openChain` eine Kette eröffnet. Steht α in der SPEC-δ^1-Position, dann sind die Literale α und δ^0 nur in verschiedenen DCG-Regeln zu finden:



In einer DCG-Regel mit Kopf δ^1 wird entweder die Klausel `distinkt` oder die Klausel `nichtDistinkt` verwendet. Sobald Prolog eine Regel ausprobiert, in deren Rumpf das Literal δ^0 steht, können wir entscheiden, ob Distinktheit überhaupt möglich ist, und folgende Fälle unterscheiden:

An δ^0 wurde mindestens eine weitere lexikalische Kategorie adjungiert. Dann ist δ^0 der *Kopf einer Kette*. Wegen der Regeln für die Typ-0-Bewegung/Typ-0-Adjunktion gilt: es muss ein *X-Bar-Head* σ^0 geben, welches an δ^0 adjungiert wurde: δ^0+σ^0. Dann gilt: σ^0 ist nicht distinkt von δ^0. Gemäß Definition kann σ^2 dann nicht zu einer Rektionsbarriere werden. In der Klausel, in der δ^0 steht, wird dann die Regel `nichtDistinkt` auf die Kettenliste angewandt. Trägt die Komponente `Distinktheit` den Wert `pre`, so erhält sie als neuen Wert die Konstante `miniBarrSchutz`. Mit diesem Konstantennamen soll ausgedrückt werden, dass die Rektionsbeziehung zwischen α und β durch die Nicht-Distinktheit davor geschützt wird, durch eine Minimalitätsbarriere “zerstört” zu werden, zumindest wenn β sich innerhalb der σ-Projektion befindet.

Ende des Minimalitätsbarrierenschutzes Sucht Prolog eine Regel aus, in der die Bewegungskette eines Verbs geschlossen wird, so endet auch der Schutz der Rektionsbeziehung vor Minimalitätsbarrieren. Es wird die Funktion `miniBarrSchutzAufheben` auf die Kettenliste angewandt, die Komponente `Distinktheit` erhält den Wert `keinMiniBarrSchutz`.

Angenommen, σ^0 wurde an δ^0 adjungiert, sodass tatsächlich ein lokaler Minimalitätsbarrierenschutz bestand. β könnte nun jedoch in einer tiefer gelegenen Projektion enthalten sein, so dass es von diesem Schutz gar nicht mehr erfasst wird: $\beta \in_\Delta \mathcal{T}_{\kappa^2}$, *tiefe*$(\kappa^0) >$ *tiefe*(σ^0), und der Head κ^0 ist von σ^0 distinkt. Dann kann κ^0 seine maximale Projektion κ^2 als Minimalitätsbarriere errichten, die bei der Rektion von β durch α interveniert. Je nach Struktur des Satzes wirkt der Schutz vor einer Minimalitätsbarriere also nur begrenzt.

An δ^0 wurde keine weitere Kategorie adjungiert. Dies ist der Trivialfall, bei dem δ^0 von allen anderen Heads $\in_\Delta \mathcal{P}$ distinkt ist. Jeder Head σ^0, für den *tiefe*$(\sigma^0) >$ *tiefe*(δ^0) gilt, macht seine entsprechende maximale Projektion σ^2 damit zur Minimalitätsbarriere für Rektion durch α. In der Klausel, in der δ^0 steht, wird dann die Regel `distinkt` auf die Kettenliste angewandt. Trägt die Komponente `Distinktheit` den Wert `pre`, so erhält sie als neuen Wert die Konstante `keinMiniBarrSchutz`.

Der Konstantenname drückt aus, dass die Rektionsbeziehung nicht mehr automatisch davor geschützt wird, durch σ^2 zerstört zu werden. *Ob* eine Störung in Form einer Minimalitätsbarriere eintritt, hängt von weiteren Faktoren ab:

- *tiefe*$(\sigma^0) >$ *tiefe*(δ^0)
- α exkludiert
- β inkludiert
- β oder $\mathcal{T}_\gamma$ (mit $\beta \in_\Delta \mathcal{T}_\gamma$) steht in COMP-$\sigma^0$-Position der X-Bar-Struktur. Dies liegt daran, dass Selektion strukturell verstanden wird, und ein Head seine COMP-Position selegiert. Umgekehrt wird Rektion von Konstituenten in SPEC-Position nie durch Minimalitätsbarrieren verhindert, da Heads die SPEC-Position ihrer Projektion nicht selegieren.

Sind alle o.g. Bedingungen erfüllt, ist es unerheblich, ob σ^2 auch selegiert ist (siehe Def. *Rektionsbarriere*). Wenn δ^0 und α Geschwister sind, sind sowohl `openChain` als auch `distinkt`/`nichtDistinkt` Literale des Rumpfes.

`nextBinding`

Diese Funktion wird genau dann in einer DCG-Regel angewandt, wenn sie eine Spur enthält, und wenn in der Position der Spur keine Konstituente basisgeneriert werden kann. Sobald man `nextBinding` anwenden kann, ist definitiv gesichert, dass man die tatsächliche Relation zwischen α und β gefunden hat. Es wird geprüft, ob der Zähler `SubjazenzBgrenzen` ≤ 1 und `Rektionsbarrieren` $= 0$ ist. Wenn dem so ist, werden alle relationalen Eigenschaften der Kette (die Komponenten des Funktors `c`) auf die Werte einer "neuen" Kette zurückgesetzt. Ansonsten ist die Klausel `nextBinding` unerfüllbar.

Empty Category Principle

Das *ECP* verlangt, dass jede Spur strikt regiert ist. Die Rektion der Spur durch das Verb ist eine Möglichkeit, und kann als Trivialfall betrachtet werden. Die andere Möglichkeit ist die Antezedensrektion von α_{i+1} durch sein Vorgängerelement α_i. Hierbei könnten die relational definierten Rektionsbarrieren intervenieren. Keine maximale Projektion ist *per se* eine Barriere, sondern immer nur im strukturellen Bezug zum potentiellen Regent α und regierter Konstituente β. An dieser Stelle wird die Definition der DCG-Klauseln zu einem wirklichen Problem: wir können immer nur drei Knoten gleichzeitig betrachten, es gibt keine Zugriffsmöglichkeiten auf strukturell höhergelegene Elemente des Syntaxbaums.

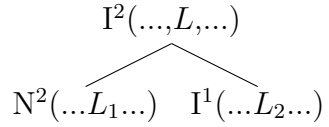
Die Implementierung beruht auf folgendem Prinzip: Binde die Komponenten des `c(...)`-Terms aus der Kettenliste an Werte, die bestimmte Eigenschaften von α, seiner Umgebung und potenziellen Barrieren σ^2 repräsentieren – es müssen alle Markierungen vorhanden sein, die in der Definition von Rektionsbarrieren aufgeführt sind. Entscheide *nachträglich*, wenn alle Informationen vorliegen, und vor der Betrachtung von β, ob es sich bei σ^2 um eine Rektionsbarriere handelte. Durch die Anwendung der Regel `ecpAdjust` auf die Kettenliste wird der Zähler `Rektionsbarrieren` um +1 erhöht, *wenn* für die relationalen Kettenattribute `Inklusion`, `Exklusion`, `Selektion` und `MiniBarrSchutz` die entsprechende Kombination von Werten vorliegt. Merke hierzu, dass die Klausel `closeChain` *unerfüllbar* ist, wenn `Rektionsbarrieren` ≥ 1. Dies entspricht exakt der Definition, dass keine Rektionsbarriere zwischen zwei Gliedern α_i und α_{i+1} der Kette vorkommen darf! Ausnahme: steht β in der SPEC-Position, so müssen wir σ^0 nicht betrachten – die Erfüllbarkeit des Inklusions-/Exklusions-Kriteriums und der Selektion von σ^2 lässt sich unmittelbar nach "Überquerung" von σ^2 entscheiden.

Subjazenzbedingung

Wir können diese Regel so verstehen, dass die Anzahl der Subjazenzbarrieren in einer Bindungsbeziehung nicht zu groß werden darf. Wir geben von einem Element der Kette α_i aus einen Zähler an das Element α_{i+1} weiter. Dieser Zähler wird immer dann inkrementiert, wenn wir tatsächlich eine Subjazenzbarriere σ^2 überschritten haben, die α_i exkludiert. An der Spur α_{i+1} verlangen wir, dass der Zähler ≤ 1 sein muss. In jeder Regel, deren Kopf eine maximale Projektion darstellt, wenden wir die Funktion `isSubjacencyBarrier` auf die Kettenliste an. Wenn die lexikalische Kategorie N oder I ist, und das gerade überschrittene Segment den Index 1 trägt, durchläuft `isSubjacencyBarrier` die Liste und inkrementiert die Variable `SubjazenzBKnoten` jedes `c`-Terms um +1. Der Segmentindex muss beachtet werden, weil eine verteilte Projektion als eine *einzige* Kategorie angesehen wird, und eine einzige Projektion ein Element nicht *mehrmals* exkludieren darf. Dass nicht zu viele Subjazenzbarrieren überschritten wurden, wird durch den Test `SubjazenzBKnoten` $\stackrel{?}{\leq} 1$ bei Auswertung der Klauseln `closeChain` bzw. `nextBinding` überprüft.

Verzweigung 1

Wenn wir auf eine Kettenliste L zwei unterschiedliche Funktionen anwenden, so erhalten wir die Ergebnisse L_1 und L_2, $L_1 \backslash== L_2$. Wenn wir die beiden Variablen als Komponenten in unterschiedliche DCG-Literale einsetzen, "verzweigt" sich die Kettenliste. Beispiel:



Hier wird deutlich, warum L_1 und L_2 verschieden sein müssen. Wenn α ein potentieller Regent ist, dann wird es von N^2 in jedem Fall exkludiert, von I^1 aber nicht. Aus diesem Grund müssen zwei unterschiedliche Kettenlisten verwendet werden, die die unterschiedlichen Relationen zwischen α und seinen möglichen Rektionspartnern β widerspiegeln.

Verzweigung 2

Folgende Satzkonfiguration ist möglich:

$$[_{CP_1}\ [_{NP}\ \textit{Den Kaffee}\ ]_{i_1}\ \textit{berichtet Peter}\ [_{CP_2}\ e_{i_2}\ \textit{hat Hans}\ [_{CP}\ e_{i_3}\ \textit{gekocht}\ ]\ ]\ ]$$

In einer CP_1 ist eine weitere CP_2 als Nebensatz eingebettet. Das Objekt $[_{NP}$ *Den Kaffee* $]_{i_1}$ wurde innerhalb der Nebensatz-CP CP_2 basisgeneriert. Anschließend wurde es jedoch über CP_2 hinausbewegt und steht nun in der SPEC-C^1-Position der Matrixsatz-CP CP_1. Um eine solche grammatisch korrekte Bewegung berücksichtigen zu können, können die Informationen der Kette $\langle [\textit{Den Kaffee}]_{i_1}, e_{i_2}, e_{i_3} \rangle$ mittels der Regel `branchChainBundleCP` in die Nebensatz-CP "abgezweigt" werden. Analog funktioniert `branchChainBundleIP`, wenn eine IP das Objekt eines A.c.I.-Verbs ist (siehe S. 52).

D-Struktur und S-Struktur

Die Überprüfung der Subkategorisierung findet nach unserer Implementierung nicht an der S-Struktur-Position statt, sondern an der D-Struktur-Position. Aus diesem Grund muss das *morphologische Paket* auch bei den Ketteninformationen gespeichert werden. Durch die Unifikation von Prolog sind die morphologischen Informationen dann sowohl an der D-Struktur- als auch an der S-Struktur-Position verfügbar. Dies wird der linguistischen Definition von Ketten als *verteilter Kategorie* gerecht (siehe Abschnitt 3.12.3).

Kettenlistenpositionen raten

In der Praxis ist die Anzahl der "gleichzeitig offenen" Ketten durch die Anzahl der beweglichen Satzteile stark beschränkt. In einer CP können das Verb, das Subjekt und (wenn vorhanden) ein oder zwei Objekte des Verbs bewegt werden. Zur Überprüfung der Subkategorisierung wird genau einer der Einträge der Kettenliste herangezogen. Da es, ohne den gesamten Phrasenstrukturbaum gesehen zu haben, schwerfällt, eine Konstituente in der S-Struktur seiner D-Struktur-Position zuzuordnen, müssen verschiedene

Kettenlistenpositionen ausprobiert werden. Hierbei macht man sich das automatische Backtracking von Prolog zunutze.

... für Objekte: Wir nehmen an, dass die Reihenfolge der Objekte di-/tritransitiver Verben durch die Subkategorisierung des Verbs fest vorgegeben ist. In dieser Reihenfolge werden die Objekte dann auch in der D-Struktur basisgeneriert. Davon abweichende Objektstellungen in der S-Struktur sind in Wirklichkeit das Ergebnis von *Scrambling*. Zudem kann sowohl das erste als auch das zweite Objekt in die SPEC-C[1]-Position bewegt werden:

"$[_{O_1}$ *Das Buch*] *überreichte Peter* $[_{O_2}$ *dem Kunden*]"
"$[_{O_2}$ *dem Kunden*] *überreichte Peter* $[_{O_1}$ *das Buch*]"

Das Problem hierbei ist, dass die Objektposition in der Kettenliste schon dann besetzt werden muss, wenn Prolog bei der Regel angelangt ist, die z.B. die SPEC-C[1]-Position beschreibt, obwohl die Suche das Verb und seine Subkategorisierung noch gar nicht "gesehen" hat. Zum Eröffnen der Position setzt man als erstes Argument der `openChain`-Funktion die Konstante `obj` ein. Die Definition von `openChain(obj,...)` umfasst zwei Klauseln. Bei der einen wird die erste Objektposition der Liste besetzt, bei der anderen die zweite Objektposition. Es wird also "geraten", ob das momentan "betrachtete" Objekt das erste oder das zweite ist. Nachdem die Belegung der Kettenliste auf diese Weise fixiert ist, geht die Suche weiter. Ist die erste Position besetzt, und die betrachtete XP auch tatsächlich das erste Objekt, dann kann die Suche nach einer Lösung erfolgreich sein. Ansonsten wird das von Prolog gefundene $\mathcal{P}$ nicht akzeptiert, weil der Kasus von O_1 nicht mit dem vom Verb geforderten Kasus übereinstimmt. Wenn alle möglichen Regeln durchprobiert wurden, setzt das Backtracking zur `openChain(obj,...)`-Klausel zurück. Dann wird die zweite Objektposition der Kettenliste mit den Ketteninformationen besetzt. Das darauffolgende Verfahren funktioniert analog zum ersten Objekt.

`closeChain(obj,...)`, mit dem eine Kettenlistenposition geschlossen werden soll, wird immer in einer D-Struktur-Objektposition $[_{V^1}\quad O_i/e\quad V^1]$ angewandt. `closeChain` ist so definiert, dass zuerst die erste Objektkette geschlossen wird. Erst dann kann die zweite Position geschlossen werden. Dadurch wird sichergestellt, dass O_1 von der Subkategorisierungsüberprüfung an der syntaktisch ersten Position gefunden wird.

Hat das Verb nur ein einziges Objekt, so werden durch die Definition von `openChain` zwar beide Objektpositionen in der Kettenliste ausprobiert, jedoch kann die $\langle O_1, \ldots, e\rangle$-Kette nur dann geschlossen werden, wenn sie die erste Objektposition der Liste belegt.

Repräsentation der Satzarten

Sowohl `c2` als auch `c1` haben eine Komponente, die durch die Variable `CPbesetzung` bezeichnet wird. Schon bevor eine Regel mit Kopf `c2` oder `c1` ausgewertet wird, ist `CPbesetzung` bereits an den Funktor `cpbesetzung(_,_,`*`Satzart`*`)` gebunden. Jede CP hat von außen eine bestimmte Satzart vorgegeben: bei der obersten CP gilt immer

[14] das 3. oder 4. Element der Liste

Satzart == matrixsatz, bei eingebetteten CPs ist *Satzart* == nebensatz und bei Relativsatz-CPs in COMP-N^0-Position *Satzart* == relativsatz. Diese Vorgaben sind nötig, weil alle Satzarten zwar unterschiedliche Wortstellungen erlauben, jedoch nicht alle, die *überhaupt* möglich sind (siehe Tabelle in Abschnitt 3.6.1).

Jede DCG-Regel mit Kopf c2 oder c1 "weiss", mit welcher Art von Konstituente die betrachtete SPEC-C^1- oder C^0-Position besetzt ist. Im Rumpf von c2 --> ... steht die Klausel setSpecCP, die der ersten Komponente von cpbesetzung(...) eine Konstante zuweist, die die Wortart wiederspiegelt. Analog wird die zweite Komponente mittels setC0 in der Regel

```
c1 --> ...,{setC0(...)},{erfuelltSatzartVorgabe(...)}, i2
```

eingesetzt. Bevor jedoch i2 ausgewertet werden darf, muss erst die Klausel erfuelltSatzartVorgabe(... erfüllbar sein. Ihr Argument ist der cpbesetzung-Term. Wenn die Konstituenten in SPEC-C^1 und C^0 nicht zur vorgegebenen Satzart passen, ist der Satz ungrammatisch.

Rekursionsfilter

Es gibt eine besondere Einschränkung der Implementierung, was die Anzahl und Art der eingebetteten Nebensätze angeht: ein Nebensatz, dessen Phrasenstrukturbaum die Wurzel CP_{Ne} hat, kann nur im *Matrixsatz* in der Vorfeldposition stehen. Beispiel:

"[$_{\mathrm{CP}_{Ne}}$ *Dass der Bus zu spät kommt*]*, ahnte ich sofort."*

Die Regel, die einen Nebensatz in der SPEC-C^1-Position erfasst, lautet c2 --> c2, c1. Ohne weitere Beschränkung läuft Prolog jedoch in eine Endlosschleife, da die Suche immer die Regel c2 --> ... zuerst ausprobiert und in dieser immer zuerst bei Rumpf-Literal c2 ansetzt. Eine Struktur [$_{CP_i}$ CP_j C^1] ist dennoch analysierbar: die Regel rekursionsFilter ist nur dann erfüllbar, wenn die Komponente CPBesetzung von CP_i den Wert matrixsatz hat und von CP_j den Wert nebensatz. CP_i muss also ein Hauptsatz/Matrixsatz sein, und CP_j ein Nebensatz mit beliebiger Wortstellung.

Verbindung zu kontrollierenden Prologregeln:

Das Ergebnis einer Regel kann durch eine Variable in der letzten Komponente eines Funktors ausgedrückt werden. Eine solche Variable kann natürlich auch als Argument einer "kontrollierenden" Prolog-Regel eingesetzt werden.

5.5 Kontrollierende Prologregeln

Hierbei handelt es sich um reguläre Prolog-Regeln, die im Rumpf den DCG-Literalen nachgeordnet sind, also "ganz rechts" im Rumpf stehen. Innerhalb einer Regel werden sie deshalb erst ganz zum Schluss ausgewertet. Sie lassen sich von den suchbeschränkenden

Regeln dadurch abgrenzen, dass sie keinen Einfluss darauf haben, welche Phrasenstrukturbäume $\mathcal{P}_1 \dots \mathcal{P}_n$ von Prolog überhaupt gefunden werden können. Ebenso tragen sie nicht zur Beschränkung der Komplexität der Bäume bei.

Vielmehr kann man durch sie für ein $\mathcal{P}$ entscheiden, ob es grammatisch ist oder nicht. Die Überprüfung der Anzahl und Art der Objekte eines Verbs, die Implementierung der Bindungstheorie, und die Ausgabe eines $\mathcal{P}$ in Form *indizierter Klammern* erfolgen z.B. über diese kontrollierenden Regeln.

Diese Regeln sind oftmals darauf angewiesen, dass bestimmte Variablen, die ein Argument eines Funktors sind, instanziiert sind. Sei $\mu \in_\Delta \mathcal{P}$ die Wurzel eines Teilbaums, mit dem Rand $\lambda_1 \dots \lambda_n$. Sobald Prolog im Lexikon erfolgreich eine DCG-Regel der Form $X \rightarrow [\lambda_n]$ gefunden hat, werden in denjenigen Regeln, die Prolog zur Kontruktion des Teilbaums $\mathcal{T}_\mu$ verwendet hat, die Kontrollregeln ausgewertet. Dies geschieht sukzessive von unten nach oben. Die Suche zur Auswertung der Kontrollregeln hat eine sehr geringe Tiefe, denn die Regeln sind fast alle so geschrieben, dass nur eine einzige Klausel aus der entsprechenden Definition ausgewählt werden kann.

5.5.1 Liste der Subkategorisierungen

Die Beschreibung der Subkategorisierungen als Tupel in Abschnitt 3.7.1 wurde bewusst so gewählt, weil sich Tupel sehr leicht durch Prolog-Funktoren darstellen lassen. In der Implementierung wird an der D-Struktur-Position (also an einer Spur) überprüft, ob die morphologischen Attribute, die vom Head α_1 der Kette $\langle \alpha_1, \dots, \alpha_n \rangle$ bis zur D-Struktur-Position α_n übertragen wurden, mit den vom Verb geforderten morphosyntaktischen Merkmalen der Konstituente kongruent sind. Die Kasuszuweisung an die Kette kann an jedes beliebige α_i erfolgen – dies ermöglichte es mir, die Kongruenz bei α_n abzugleichen.

Im Prolog-Lexikon ist für jede Flexionsform eines Verbs die zugehörige Subkategorisierung vermerkt. Die Informationen sind streng genommen redundant, jedoch lässt sich mit dieser Herangehensweise das Lexikon einfacher verwalten.

Subkategorisierung ist theoretisch nicht in ihrem Umfang beschränkt. Aus diesem Grund wird sie im Lexikon als eine Liste repräsentiert (gemäß der Listendefinition von Prolog, siehe [SSSK06], [MANL08]). Die Elemente der Liste sind dreistellige Funktoren mit Bezeichner `obj`.

Die Subkategorisierungslisten des Lexikons können nicht ohne Weiteres übernommen werden. Zwar vergibt ein flektiertes Verb eine Θ-Rolle an das Subjekt, der Kasus wird dem Subjekt jedoch von I^0 zugewiesen. Auch die Behandlung *optionaler Objekte* – also die Frage, ob die Konstituente im betrachteten Satz vom Verb subkategorisiert wird – können nur mit der speziellen Regel `generateSubkat` abgedeckt werden, wenn man zusätzliche Lexikoneinträge vermeiden möchte. `generateSubkat` legt als erstes Element einer "anwendbaren" Subkategorisierungsliste den zweistelligen Funktor `subj(np,nom)` fest, falls das Verb Flexionsmerkmale trägt.

Nun können die Lexikoninformationen aufgenommen werden: für jedes Verb werden die als *obligatorisch* gekennzeichneten Objekte aus der Subkategorisierung in jedem Fall in die fertige Subkategorisierungsliste übernommen. Für jedes *optionale* Objekt gibt es zwei Listen: eine, die den entsprechenden `obj`-Term enthält, und eine, die ihn nicht

enthält. Die Definition von `generateSubkat` umfasst mehrere ambige Regeln, sodass die Auswertung nach Backtracking mehrere unterschiedliche Subkategorisierungslisten zurückgibt. Um zu wissen, welche Objektstruktur genau gemeint ist, müssen wir alle möglichen Subkategorisierungslisten mit $\mathcal{P}$ abgleichen.

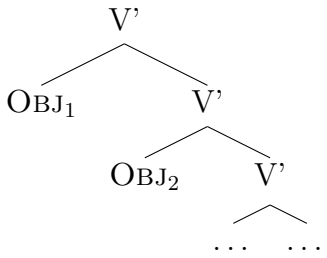
Eine Liste wenden wir dann mittels der `kongruenz`-Regel sukzessive von unten nach oben in $\mathcal{P}$ auf die Objektpositionen an. Dies heisst es wird überprüft, ob die Kategorie des Objekts dem lexikalischen Typ der geforderten Kategorie entspricht, und ob der geforderte Kasus mit dem tatsächlichen übereinstimmt. Anschließend wird durch Backtracking – unabhängig davon, ob die Anwendung erfolgreich war oder nicht – mittels `generateSubkat` eine davon verschiedene Liste generiert, die in gleicher Art auf $\mathcal{P}$ angewandt wird.

Merke: der Funktor `obj` in einer "anwendbaren" Subkategorisierungsliste ist nur noch zweistellig, die Komponente `Konditionalität` wird nicht übernommen – ihre Behandlung wurde ja bereits abgedeckt.

Die "fertige" Subkategorisierungsliste wird durch den im folgenden Abschnitt beschriebenen Mechanismus an die D-Struktur-Position des VP-Heads gebracht, beginnend mit der Regel, die diesen Head für die S-Struktur darstellt.

5.5.2 D-Struktur-Schlinge

Die Anzahl der Regeln kann reduziert werden, wenn die Überprüfung der morphologischen Kongruenz (geforderte Verb-Argumente $\leftrightarrow$ Phrasen an Argumentposition) immer an der D-Struktur-Position seiner Objekte stattfindet:



Das Problem ist jedoch, dass sowohl Argumente als auch Verb durch Move-α ihre D-Struktur-Position verlassen können. Um den Entwurf der Grammatik zu vereinfachen und die Anzahl von Regeln gering zu halten, wurde für die Implementierung eine Datenstruktur mit der Bezeichnung **D-Struktur-Schlinge** entwickelt. Die Subkategorisierungsinformationen, die dem Lexikon entstammen, verraten uns, welche und wieviele Satzteil-Ergänzungen das Verb benötigt. Das könnte z.B. eine Akkusativ-Nominalphrase sein (weitere Details siehe Abschnitt 5.6). Die Subkategorisierungsinformationen sind in einer Liste zusammengefasst. Wir machen uns zunutze, dass die Reihenfolge der Argumente eines Verbs eindeutig im Lexikon festgelegt ist. Alle Variationen der Objektreihenfolge (vergleiche *"Peter schenkte der Freundin das Buch"* $\leftrightarrow$ *"Peter schenkte das Buch der Freundin"*) sind in Wirklichkeit das Ergebnis von Konstituentenbewegung (*Scrambling*).

Der Funktor `v0` kommt nur in zwei Regeln der Grammatik vor: $V^1 \longrightarrow V^0$ und $I^0 \longrightarrow$

V^0. Eine seiner Komponenten enthält die Subkategorisierungen aus dem Lexikon. Mittels `generateSubkat` wird dann die endgültige Liste der Subkategorisierungen und der Subjektmerkmale erzeugt, die zur Überprüfung verwendet werden soll. Dabei stehen die Merkmale in der Reihenfolge, wie sie in der D-Struktur des Satzes vorkommen müssen. Bsp.:

```
[subj(np,nom), obj(np,dat), obj(np,akk)]
```

Diese Liste `L` und die morphosyntaktischen Merkmale `MP` des Verbs werden im Funktor `subkatChan` als `subkatChan(MP,L)` zusammengefasst. Der Funktor wird dann als eine Komponente von `i0` eingesetzt. Nachdem die Suche `v0` "erreicht" hat, stehen durch anschließende Unifikation die Subkategorisierungsinformationen an einer Position von `i0` zur Verfügung, nämlich in folgenden beiden Regeln:

```
c1(...) --> c0(..., SubkatChan[15], ...), i2(...)
i1(...) --> v2(...), i0(..., SubkatChan, ...)
```

Für beide Positionen gilt: V^0 ist adjungiert und steht somit an einer strukturell höhergelegenen Position als die D-Struktur-Positionen der Verb-Objekte. V^0 kann ausschließlich durch *Move-α* an die beschriebene Position gelangt sein. Die X-Bar-Struktur, die tiefer als `i2` bzw. `v2` gelegen ist, muss erst noch ermittelt werden – somit ist auch die D-Struktur-Position der Argumente noch nicht erreicht. Die Suche wird nun aber mit dem "Hintergrundwissen" fortgesetzt, dass wir die x-te Stelle der Funktoren `v1`, `v2`, `i1`, `i2`, `c1` und `c2` ab sofort für diejenigen Subkategorisierungsinformationen reservieren, die von `c0`/`i0` aus zur D-Struktur-Position des Verbs gelangen sollen:

```
c1(...,empty_x,UP_{x+1},...)
-->
c0(..., SubkatChan, ...),
i2(..., SubkatChan_x, UP_{x+1}, ...)

i1(...,empty_x,UP_{x+1},...)
-->
v2(..., SubkatChan_x, UP_{x+1}, ...),
i0(..., SubkatChan, ...)
```

Bei den beiden Regeln, die `c0` bzw. `i0` "sehen", wird die x-te Komponente der Köpfe `c1` bzw. `i1` mit der Konstanten `empty` besetzt, die x-te Komponente von `i2` bzw. `v2` wird dagegen mit der Variablen `SubkatChan` als Argument besetzt. Das heisst, dass bei der folgenden Suche sichergestellt ist, dass die Subkategorisierungsinformationen immer an der x-ten Komponente zur Verfügung stehen. Die $(x+1)$-te Komponente von `c1` und

[15] Im Programm wird für die leichtere Lesbarkeit der Funktor mittels der Variablen `SubkatChan` repräsentiert. Die Variable ist dann bereits an den Funktor `subkatChan(...)` gebunden

`i2` bzw. von `i1` sind mit der Variablen `UP` belegt, die wir zum jetzigen Zeitpunkt noch nicht instanziieren können.

Durch die Unifikation in der ***SLD-Schritt***-Prozedur werden die Subkategorisierungsinformationen bei der Suche in strukturell immer tiefer gelegene Positionen "transportiert". Irgendwann ist die Suche natürlich am tiefsten Punkt des Phrasenstrukturbaums angelangt. Nun werden die Regeln ausprobiert, die die Head-Position der VP betreffen. Ist die x-te Position von `v1` mit Subkategorisierungsinformationen belegt (und das Verb somit nicht mehr in der D-Struktur-Position), so wird die Regel

$$\texttt{v1(..., SubkatChan}_x\texttt{, SubkatChan}_{x+1}\texttt{, ...) --> e}$$

ausgewählt. Die $(x+1)$-te Position, die uns schon einmal in Form der Variablen `UP` begegnet ist, ist nun mit den Subkategorisierungsinformationen belegt. Durch vorausgegangene Unifikation wird jede Variable gleichen Namens an der $(x+1)$-ten Position ebenfalls an die Informationen gebunden. Wir erwähnten bereits, dass für alle Knoten, die strukturell höher stehen als V^1, die $(x+1)$-te Position noch nicht bestimmt werden konnte. Die Liste wird sozusagen von "unten nach oben weitergereicht" im Phrasenstrukturbaum. Erst nachdem die Suche bei dieser "strukturell tiefsten" Regel angekommen ist, kann sie an Positionen gelangen, an denen Kongruenzüberprüfungen vorgenommen werden. Sie gelangen sogar an Positionen, die überhalb des eigentlichen Verbs liegen. Daher kommt die Bezeichnung *"Schlinge"*. Die x-te Komponente wird auch als *Down*-Kanal, die $(x+1)$-te Komponente als *Up*-Kanal bezeichnet. Bei an VP adjungierten Verben V^0 können durch `mergeSubkats` die beiden verschiedene `subkatChan`-Terme zu einem einzigen Term vereinigt werden.

Befindet sich V^0 dagegen weiterhin an seiner ursprünglichen D-Struktur-Position (weil es sich um einen Infinitiv handelt), so gab es auf der bisherigen Suche keine Regel, die die x-te Position mit Subkategorisierungsinformationen besetzen konnte. Stattdessen ist die `x`-te Komponente von `v1` dann die Konstante `empty`. Dann kommt nur folgende Regel in Frage, bei der die $(x+1)$-te Komponente von `v1` direkt mit den Subkategorisierungsinformationen besetzt wird:

$$\texttt{v1(..., empty, SubkatChan}_{x+1}\texttt{, ...) --> v0, \{generateSubkat(...)\}, ...}$$

Mittels Unifikation können die Subkategorisierungsinformationen an eine von V' dominierte, höhergelegene Position weitergegeben werden, an der eine Objekt-XP steht. An jeder Objektposition wird die Liste um das letzte Element gekürzt. Es wird die Kasus-Kongruenz und die Übereinstimmung der lexikalischen Kategorie zwischen dem Listeneintrag und der tatsächlich vorhandenen Konstituente überprüft (mehr dazu im folgenden Abschnitt zur Überprüfung der Subkategorisierung). Die Restliste wird im Anschluss dann nach oben weitergereicht. Die Liste repräsentiert also an jedem Punkt genau diejenigen Satzelemente, deren geforderte Existenz noch nicht verifiziert werden konnte. Ist die Subkategorisierungsliste an der Wurzel des Phrasenstrukturbaumes (meist eine maximale Projektion CP) leer, so konnte die Existenz aller geforderten Argumente erfolgreich sichergestellt werden. Umgekehrt wird ein Satz als ungrammatisch erkannt,

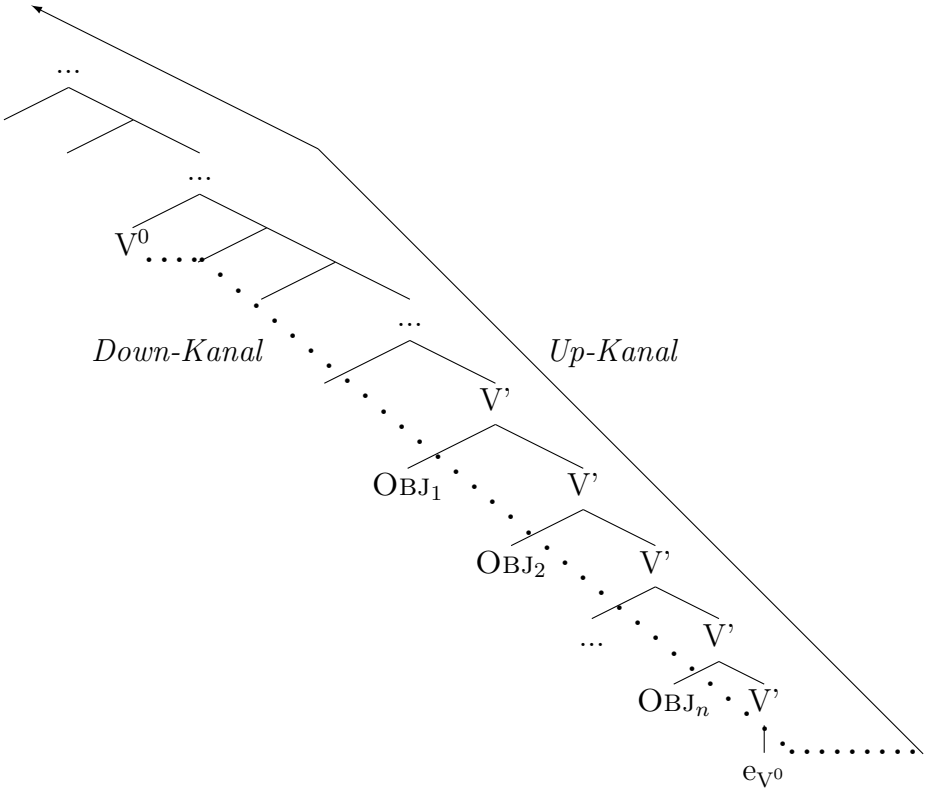
wenn ein Objekt (z.B. NP) an einer Argumentposition gefunden wird, obwohl die Subkategorisierungsliste gar kein weiteres Argument mehr vorsieht (die Liste also bereits leer ist).

Die Subjektinformationen stehen ebenfalls in der Subkategorisierungsliste, damit man sich denselben Mechanismus zunutze machen kann, und keine zusätzlichen Strukturen implementieren muss. Zudem ist das Subjekt vom Verb nicht gänzlich unabhängig. Zwar vergibt nicht V^0 den Kasus an das Subjekt, sondern I^0. Wenn jedoch kein V^0 an I^0 adjungiert wird, weil das Verb im Infinitiv vorliegt, so ist der Kopf der IP leer. Das Subjekt wird dann nicht von innerhalb der IP aus regiert, und somit auch kein Kasus gefordert. Zudem lässt dieser kombinierte Ansatz Raum für eine Erweiterung in Form von Θ-Rollen – auch wenn das infinitive Verb dem Subjekt keinen Kasus zuweist, so kann es das Subjekt trotzdem Θ-markieren.

Das Prinzip der D-Struktur-Schlinge, hat den Vorteil, dass zur Überprüfung der Kongruenz nicht mehr unterscheiden muss, ob V^0 an einer basisgenerierten Position steht, oder nicht!

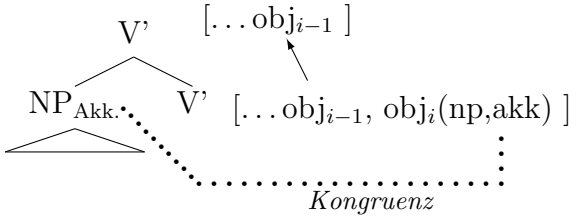
Die Liste sich auf die D-Struktur-Positionen beziehen zu lassen und sukzessive von hinten zu kürzen, hat den Vorteil, dass wir die Überprüfung der Subkategorisierung nicht mit unterschiedlichen S-Strukturen variieren müssen. Da die deutsche Sprache *Scrambling* erlaubt, könnte man nicht mehr die letzte Position selbstverständlich als Träger der Objekt-Informationen annehmen, sondern müsste erst die Liste durchsuchen. Wenn man hinzunimmt, dass das Verb in der S-Struktur strukturell oberhalb oder unterhalb des Subjektes und oberhalb, unterhalb oder zwischen seinen Objekten stehen kann, müsste auf komplizierte Weise ermittelt werden, in welche Richtung die Informationen weiterzugeben sind, und wie sie miteinander abzugleichen wären. Das Vermeiden von Fallunterscheidungen durch die D-Struktur-Schlinge vereinfacht das Regelwerk.

Folgendes Diagramm verdeutlicht die Weitergabe der Subkategorisierungsinformationen in sowohl zeitlicher als auch logischer Reihenfolge:



5.5.3 Überprüfung der Objekte

Jede D-Struktur-Position wird durch eine DCG-Regel ausgedrückt. Da wir in jeder DCG-Regel die Wurzel μ eines Objekts $\mathcal{T}_\mu$ sehen, können wir überprüfen, ob die lexikalische Kategorie von μ mit den Subkategorisierungsinformationen übereinstimmt. Die `kongruenz`-Klausel in derselben Regel erhält Typ_μ und $Kasus_\mu$ dann als Argumente. Diese beiden Informationen werden mit den Untertermen `ART`, `KASUS` des letzten Elements der Subkategorisierungsliste abgeglichen. Herrscht Kongruenz, so wird das letzte Element aus der Liste entfernt, die gekürzte Liste zurückgegeben und anschließend in $\mathcal{P}$ nach oben weitergereicht. Beispiel für μ = NP:



Sollte die NP bewegt worden sein und nur eine Spur e zurückgelassen haben, dann erhält `kongruenz` Typ_μ und $Kasus_\mu$ von der Funktion `closeChain`. In anderen Fällen kann die Klausel `kongruenz` nicht erfüllt werden, wofür es unterschiedliche Gründe gibt:

- Der Eingabesatz ist ungrammatisch.
- Prolog hat die "falschen" DCG-Regeln ausprobiert, und der gefundene Phrasenstrukturbaum $\mathcal{P}^*$ entspricht nicht dem tatsächlichen Phrasenstrukturbaum $\mathcal{P}$ des Eingabesatzes.
- Das Verb hat optionale Argumente, und Prolog hat eine Subkategorisierungsliste falscher Länge ausprobiert.

Wenn die Liste zum Knoten VP hochgereicht wurde, hat die Überprüfung aller Objektpositionen stattgefunden – dementsprechend sind auch alle `obj(...)`-Einträge aus der Liste herausgekürzt worden. Trägt das Verb, auf dessen Basis die finale Subkategorisierungsliste mittels `generateSubkat` erzeugt worden ist, Flexionsmerkmale, so enthält sie noch die grammatischen Merkmale, die das Subjekt besitzen muss: `[subj(np,nom)]`. Sind Flexionsmerkmale nicht vorhanden, so ist nur noch die leere Liste `[]` übrig. Ist das Subjekt vorhanden, so findet der Abgleich nach ähnlichem Prinzip statt, spätestens nach dieser Operation ist die Liste leer.

War dagegen eines der benötigten Objekte gar nicht im Satz vorhanden (z.B. bei * *"Peter schenkt dem Postboten."*), so wurde die Liste nicht *genug* gekürzt. An der Subjektposition wird dann ein Abgleich mit `obj(...)` versucht, der nicht funktionieren kann. Als Konsequenz wird ein Subkategorisierungsfehler erkannt.

Nachdem die Liste wird bis zur CP "hinaufgereicht" wurde, wird mittels der Klausel `dStrukturSchlingeLeer` überprüft, ob sie tatsächlich leer ist und somit alle Subkategorisierungen erfolgreich waren.

Kompromiss:

Obwohl mit dem beschriebenen System prinzipiell beliebig viele Objekt-Subkategorisierungen behandelt werden können, habe ich es auf 3 Objekte beschränkt. Die Funktion `allbutlast` entfernt das letzte Element der finalen Liste. Die Definition der Regel lautet wie folgt:

```
allbutlast([A,B,C,_],[A,B,C]).
allbutlast([A,B,_],[A,B]).
```

```
allbutlast([A,_],[A]).
allbutlast([_],[]).
```

Die richtige Klausel wird mittels Patternmatching ausgewählt. Eine allgemeinere Definition, die für Listen mit einer Länge > 4 funktioniert, lautet wie folgt:

```
allbutlast(A,B) :- last(A,Last), append(B,[Last],A).
```

Die Verwendung der eingeschränkten Definition von `allbutlast` ist zwingend notwendig. Bei der allgemeineren Definition läuft die Suche von Prolog in eine Endlosschleife. Schuld an einem solchen Aufruf ist folgende Regel:

$$\texttt{i1(...) --> v2(...,SubkatChan}_x\texttt{,...), i0(...)}$$

Sie ist die einzige linksrekursive Regel, bei der das $\texttt{SubkatChan}_x$-Argument des ersten Funktors `v2` im Rumpf noch nicht instanziiert ist (weil die Informationen erst bei V^0 zu finden sind, das an I^0 adjungiert wurde). `allbutlast` wird dann mit einer Liste als Argument aufgerufen, deren Elemente nicht instanziiert sind. Prolog probiert dann, ein Ergebnis `allbutlast` für *jede* Listenlänge auszuwerten. Es werden also alle Listen sukzessiv steigender Länge durchprobiert. Da es keine Beschränkung gibt, könnten diese Listen sogar unendlich lang sein. Prolog läuft also in die Endlosschleife.

Linksrekursive Regeln sollten bei Prolog nach Möglichkeit zwar vermieden werden, die Vorgabe liefert uns jedoch die X-Bar-Struktur des Deutschen[16]. Glücklicherweise ist die tatsächlich gewählte Implementierung für unsere Zwecke völlig ausreichend, weil in der deutschen Sprache kein Verb denkbar ist, welches mehr als drei Objekte als Argumente hat. Das einfache Patternmatching ermöglicht sogar einen Performancegewinn.

5.5.4 Θ-Markierung der Argumente

Θ-Rollen sind nicht implementiert, weil in der Syntax-bezogenen Literatur kein Formalismus für ihre Überprüfung angegeben ist. Laut der linguistischen Theorie findet Θ-Markierung immer in der D-Struktur-Position = dem letzten Element α_n der Kette $\langle \alpha_1, \ldots, \alpha_n \rangle$ statt. Somit ist die Erweiterbarkeit des Programms um Θ-Rollen möglich.

5.5.5 Mehrteilige Prädikate

In Abschnitt 2.3 wurde beschrieben, welche Bedingungen bei der Kombination von Verben zu mehrteiligen Prädikaten erfüllt werden müssen. Zur Umsetzung wird die Regel `verbKombination` verwendet.

5.5.6 `tree`-Funktion

Die Ausgabe eines Phrasenstrukturbaums $\mathcal{T}_\alpha$ erfolgt in Form der linguistischen Klammernotation und ist vom Typ *String*[17]. Die `tree`-Funktion setzt dementsprechend $\mathcal{T}_\alpha$ aus

[16] In einer Grammatik für die englische Sprache gäbe es dieses Problem nicht!

[17] Es wurde der Typ *String* anstatt *Listen von Listen* gewählt, weil Prolog bei Listen mit großer Schachtelungstiefe das Ergebnis nur abgekürzt (`...`) ausgibt: `T = [cp,[np ...],[c' ...]]`

den "Baumstrings" des linken und rechten Teilbaums sowie aus derem gemeinsamen Vaterknoten α zusammen. Das Ergebnis wird an die letzte Komponente des `tree`-Funktors gebunden.

5.5.7 Bindungstheorie

Die Implementierung der Bindungstheorie beschränkt sich auf Anaphern, die innerhalb einer Rektionsdomäne gebunden sein müssen. *A-Bindung* wird durch Koindizierung ausgedrückt. A-Bindung bei nicht-anaphorischen/-pronomialen NPs ist generell nicht möglich. Um diese muss man sich daher in der Implementierung nicht kümmern:

$$*\text{“}Trudi_i \textit{ bewundert } Trudi_i\text{.”}$$

Pronomiale NPs *können* in ihrer Domäne A-gebunden sein:

$$\text{“}Maria_i \textit{ weiss, dass Trudi sie}_i \textit{ bewundert”}$$
$$\text{“}Maria_i \textit{ weiss, dass Trudi sie}_j \textit{ bewundert”}$$

Ob diese A-Bindung besteht oder nicht, lässt sich zwar syntaktisch durch Indizes ausdrücken, schlägt sich aber nicht in der in den phonetisch spezifizierten Worten des Satzes nieder. Mit einem reinen Syntax-Parser gibt es keine Möglichkeit zu entscheiden, ob eine A-Bindung eines Pronomens zu einem Antezedens *innerhalb* des Satzes besteht oder nicht, sofern diese syntaktisch möglich ist. Ob eine solche A-Bindung besteht, wird vom Programm nicht berücksichtigt.

Bei der Bindungstheorie liegt liegt die Schwierigkeit für die Implementierung ähnlich wie bei den Rektionsbarrieren: Die Konstituenten, die in den Definitionen gefordert werden, können nicht alle in derselben DCG-Regel stehen. Die zur Bindungstheorie gehörigen Klauseln gehören zu den kontrollierenden Prolog-Regeln. Letztere haben die Eigenschaft, dass die Auswertung seitens Prolog in umgekehrter Reihenfolge zur Auswertung der DCG-Regeln erfolgt. Durch die Tiefensuche werden die Klauseln in einer Reihenfolge ausgewertet, als würde man $\mathcal{P}$ “von unten nach oben durchlaufen”. Auf die Konstituenten α, ρ, ω muss geachtet werden.

Die Funktoren `v1`, `v2`, `i1`, `i2`, `c1` und `c2` haben eine Komponente, welche für die Behandlung der Bindungstheorie reserviert ist. In den DCG-Regeln wird sie im Allgemeinen mit der Variablen `RectDom` bezeichnet. In DCG-Regeln, die das Literal `v0` enthalten (das untere Ende des Rückgrats), wird `RectDom` mit dem dreistelligen Funktor `rectDom` initialisiert. Die Komponenten sind dann zunächst mit den Konstanten `noAlpha`, `noRho`, `noOmega` besetzt. Das Verhältnis von α,ρ und ω ist eindeutig, aber erst, wenn wir auf dem Weg von “unten nach oben” alle Konstituenten $\in_\Delta \mathcal{P}$ betrachten konnten. `rectDom` beinhaltet relationale Informationen, die angeben, welche Konstituenten einer Rektionsdomäne wir bereits “gesehen” haben.

Potentielle Regenten ρ sind Heads. In jeder Regel, in der sich im Rumpf ein entsprechender Head `v0`[18] oder `p0` befindet, wenden wir innerhalb derselben DCG-Regel

[18]Sowohl lexikalische Heads V^0 als auch Adjunktionen an funktionale Heads I^0+V^0, C^0+I^0+V^0

auch die Klausel `newRho` auf den `rectDom`-Term an. Die Konstante `noRho` wird hierbei durch die Konstante `rho` ersetzt. Ein Head ρ kann sowohl die COMP-ρ^0 als auch die SPEC-ρ^1-Position seiner Projektion regieren. Somit wird auch jede Anapher α regiert, die sich in COMP- oder SPEC-Position befindet. Umgekehrt ist es nicht möglich, dass ρ ein α regiert, für das gilt: *tiefe*$(\alpha) \leq$ *tiefe*(ρ^2) – α kann dann nämlich nicht von ρ^0 c-kommandiert werden.

Der strukturell verstandene Opazitätsfaktor steht in SPEC-Position. In jeder DCG-Regel, welche eine besetzte SPEC-Position behandelt, wenden wir daher die Regel `newOmega` auf `rectDom` an. Die Konstante `noOmega` wird durch `omega(MP)` ersetzt. `MP` bezeichnet das morphologische Paket des Subjekts ω.

In jeder DCG-Regel, die die D-Struktur-Position einer Objekt-NP erfasst, wird die Regel `newAlpha` auf `rectDom` angewandt. Ein entsprechendes Pattern in der Definition von `newAlpha` bewirkt, dass `noAlpha` nur dann durch `alpha(MP)` ersetzt wird, wenn die NP anaphorisch ist. `MP` bezeichnet das Tupel morphosyntaktischer Merkmale der NP.

Die Regel `closeRectDom` wird in jeder DCG-Regel angewendet, deren Kopfliteral eine maximale Projektion ist. Dies entspricht der Verfolgung eines Pfads entlang des Rückgrats in $\mathcal{P}$. `closeRectDom` wird immer dann angewandt, bevor man eine eine maximale Projektion überschreitet. Das erste Argument ist der `RectDom`-Term aus einem Funktor des Rumpfes. Wenn die Rektionsdomäne ein Teilbaum $\mathcal{T}_\delta$ ist, dann wird mit `closeRectDom` versucht, einen Knoten, der im Kopf einer DCG-Klausel steht, als δ zu identifizieren. Damit wäre ein Abschluss der Rektionsdomäne nach oben gefunden. Wegen $\omega \overset{\text{A-Bindung}}{\longleftrightarrow} \alpha$ gilt, dass zwischen α und ω Kongruenz bezüglich *Numerus* herrschen muss, sonst ist der Satz ungrammatisch. Ist eine solche Kongruenz nicht gegeben, so gibt `closeRectDom` einen Fehlerstatus zurück (siehe folgenden Abschnitt).

Anders liegt der Fall, wenn bei Anwendung von `closeRectDom` noch gar nicht alle Bestandteile α, ρ, ω einer Domäne "gesehen" worden sind. Hier lassen sich zwei Fälle unterscheiden:

`Alpha == noAlpha.` Dann werden `Rho` und `Omega` auf `noRho` bzw. `noOmega` zurückgesetzt, was durch zwei mögliche Konstellationen zu begründen ist:

1. $\{\alpha_{\text{anaph}} \in_\Delta \mathcal{P}\} = \emptyset$. Es lässt sich in keiner Regel sagen, ob in einer anderen ausgewerteten Regel eine anaphorische NP vorkommt. Aus diesem Grund werden Eigenschaften von potentiellen ρ und ω in Termen gespeichert. Wenn jedoch in keiner Regel die `Alpha`-Komponente besetzt wird, so gibt es keinen Grund, die gesammelten Informationen zu "behalten".

2. $|\{\alpha_{\text{anaph}} \in_\Delta \mathcal{P}\}| \geq 1$. Wenn ρ^0 mittels `newRho` als potentieller Regent von α ausgewählt wird, und bei "Überschreiten" von ρ^2 `Alpha` noch nicht besetzt ist, dann muss *tiefe*$(\rho^2) \geq$ *tiefe*(α) sein. Das bedeutet aber, dass ρ^0 α nicht c-kommandieren kann, weil eine maximale Projektion ρ^2 interveniert. Die Komponente `Omega` wird ebenfalls zurückgesetzt, weil laut Definition der Bindungstheorie α sein Antezedens ω nicht c-kommandieren darf.

`Alpha \== noAlpha` **und entweder** `Rho == noRho` ***oder*** `Omega == noOmega`**.** Alle drei Komponenten werden unverändert übernommen.

5.5.8 Fehlerstatus

Die normalen Prologregeln werden aus den DCG-Regeln heraus aufgerufen. Sie sind erfüllbar, wenn als ihre Argumente genau solche grammatischen Attribute eingesetzt werden, die korrekte grammatikalische Strukturen repräsentieren (z.B. Anzahl der Subjazenzbarrieren). Das Problem mit der Fehleranalyse ist, dass, sobald eine Regel `{r(...,Err)}` korrekterweise unerfüllbar ist, die gesamte DCG-Regel `y --> ..., {r(...,Err)}, ...` ebenfalls nicht erfüllbar ist. Die Suche probiert dann einen anderen Zweig des Suchbaums aus, und findet eine andere Lösung (falls dies möglich ist).

Dabei geht jedoch die Information verloren, dass der Aufruf von `{r(...,Err)}` fehlerhaft war. Die Lösung ist, dass einige Klauseln aus der Definition von `r` sehr wohl für Argumente mit ungrammatischen Werten[19] erfüllbar sind, im Gegenzug jedoch die Variable `Err` aus `{r(...,Err)}` zur Rückgabe eines Fehlerstatus verwendet wird. Bei Argumenten mit grammatischen Werten ist der Status `[ok]`. Ansonsten wird bei der Regel `kongruenz(...,Err)` dann `Err` an `[numErr]` gebunden oder an `[missingObj]`, bei `closeRectDom(...,Err)` an `[anapherKongruenzFail]`.

Diese Informationen müssen beim ursprünglichen Benutzerquery verfügbar sein, damit sie als Argumente in eine Funktion eingesetzt werden können, die sie aus Prolog ausgibt (z.B. `writeln`). In den Funktoren `v1`, `v2`, `i1`, `i2`, `c1`, `c2`, die Projektionen entlang des *Satzrückgrats* repräsentieren, ist deshalb die letzte Komponente `Err` für diesen Fehlerstatus reserviert. Die "Weitergabe der Werte" erfolgt mittels SLD-Resolution, bis zuletzt die Variable `Err` aus der obersten DCG-Regel `s(Err,T)` an die Fehlerstatusliste gebunden wird.

Durch die Funktion `mergeErrorStats` können die Fehlerstati aus den verschiedenen Teilbäumen zu einer Liste zusammengefügt werden, wobei sich "echte" Fehler immer gegen `[ok]` durchsetzen.

5.6 Triviale Implementierungen

Kongruenz: Die Implementierung der Kongruenz zum Zweck des Merkmalstransfers wird auf recht triviale Weise durch die Unifikation von Prolog gelöst. Eine komplexere Variante der Kongruenzüberprüfung zwischen Verben und Objekten findet sich im Abschnitt 5.5.2.

c-Kommando: Das c-Kommando konnte recht einfach implementiert werden. In Prolog ist nicht zu unterscheiden, von wo das c-Kommando ausgeht, also ob

$$\alpha \text{ c-kommandiert } \beta \overset{?}{\longleftrightarrow} \beta \text{ wird von } \alpha \text{ c-kommandiert}$$

[19] z.B. `SubjazenzBknoten > 1`

Es muss jedoch Kongruenz zwischen α und β herrschen. Gleiche Variablennamen werden an den gleichen Wert gebunden. Durch entsprechende Benennung der Variablen in den Argumentpositionen der Funktoren kann sichergestellt werden, dass Prolog mittels Unifikation die Beziehung zwischen α und β herstellt, und dass keine Bedingung des c-Kommandos verletzt wurde.

5.7 Fehleranalyse

Für einen nicht grammatischen Satz soll eine Fehleranalyse möglich sein, um dem Anwender Hinweise auf die Art des syntaktischen Fehlers zu liefern. Folgende Fehler kann das Programm direkt erkennen:

- Verb und Subjekt stimmen im Numerus nicht überein. Beispiel:

  * *"Peter kochen den Kaffee."* (Singular $\leftrightarrow$ Plural)

- Es wurde kein Objekt angegeben, oder das falsche. Beispiele:

  * *"Maria schenkt dem Kind."* (Akkusativ-Objekt fehlt)
  * *"Peter kocht* $[_{\mathrm{NP_{Dat}}}$ *dem Kaffee* $]$*."* (Objekt trägt nicht den Akkusativ-Kasus)

- Zu einer Anapher (*"sich"*, *"einander"*) besteht keine Kongruenz zum Bezugswort bezüglich Numerus:

  * *"Peter*$_i$ *hat einander*$_i$ *gekratzt."*

  Sätze, in denen Anaphern keine A-Bindung zu einem Antezedens eingehen können, werden als ungrammatisch abgelehnt. Beispiel:

  * *"Einander haben die Streitenden angeschrien"*

Die Fehlermeldung wird anhand des Fehlerstatus generiert. Wird ein Satz vollständig abgelehnt, so soll aber immer noch eine Teilanalyse von Satzgliedern möglich sein. Oft sind nämlich einzelne Phrasen für sich genommen grammatisch, jedoch in Kombination mit anderen Konstituenten nicht (fehlende Kongruenz, falsche Subkategorisierung etc.).

Wenn für einen Satz ein oder mehr mögliche Phrasenstrukturbäume gefunden werden, so wird der Satz als grammatisch angenommen. Falls nicht, werden folgende Teilanalysen ausgeführt:

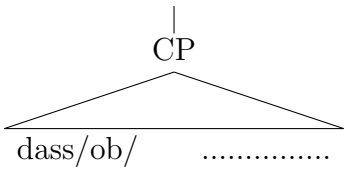
1. Auflistung, wie oft ein Wort des Eingabesatzes im Lexikon steht. Bei mehr als einem Eintrag gibt es mehrere Flexionsformen im Lexikon, die alle mit dem Eingabewort identisch sind (ihre morphosyntaktischen Merkmale sind natürlich trotzdem verschieden!). Dies hilft zu erklären, warum für manche Sätze mehrere identische Phrasenstrukturbäume gefunden werden. Worte, die nicht im Lexikon stehen, sind entweder *tatsächlich* nicht im Lexikon, oder im Eingabetext nur falsch geschrieben. Ob man für diese Überprüfung das dynamische Lexikon[20] oder das Gesamtlexikon

[20] siehe Abschnitt 5.8

verwendet, spielt keine Rolle, weil in das dynamische Lexikon sowieso alle Einträge aufgenommen werden, die mit dem Eingabewort identisch sind.

2. Alle Nominalphrasen $\mathcal{T}_{\text{NP}} \subset \mathcal{P}$ werden ausgegeben. Die erkannten Konstituenten sind maximal bezüglich der X-Bar-Struktur für NPs, d.h. $\mathcal{T}_{\text{NP}}$ soll so groß wie möglich sein. Die NPs sind dabei nicht zusammenhängend bezüglich Kongruenz o.ä. Zusätzlich werden die Ausprägungen der morphosyntaktischen Merkmale *Numerus*, *Kasus*, *Genus* der NPs aufgelistet. Die Merkmale wurden mittels Kongruenz innerhalb der NP verteilt.

3. Enthält der fehlerhafte Eingabesatz eine Nebensatz-einleitende Konjunktion (*dass*, *ob*, *weil*, ...), so wird eine Teilsatzanalyse versucht. Die Konjunktionen können ausschließlich an der Position C^0 stehen und sind somit ein eindeutiges Identifikationsmerkmal für deklarative Konstituentensätze (diese sind den Nebensätzen zuzurechnen). Dabei spielt es keine Rolle, ob die Nebensatz-CP im Vorfeld, Mittelfeld oder Nachfeld steht. Kann für den Nebensatz eine X-Bar-Struktur



gefunden werden, so wird die Phrasenstruktur in Klammernotation in die Ausgabedatei geschrieben.

5.7.1 Aufbau der Fehleranalyse

Die Regeln zur Fehleranalyse sind in der Datei `prog.pl` enthalten. Zunächst wird versucht, mittels der Prolog-eigenen `findall`-Klausel alle möglichen Lösungen für einen Satz zu finden, die den Fehlerstatus `[ok]` haben. Nur wenn dies nicht gelingt, wird die o.g. Fehleranalyse ausgeführt. Zur lexikalischen Auflistung, der NP-Analyse und der separaten Nebensatzanalyse werden folgende Klauseln verwendet:

`lexError` geht die Satzliste elementweise durch. Für jede Konstante wird für jeweils alle Kategorien $\mathcal{L} \cup \mathcal{F}$ getestet, ob eine Regel im Lexikon vorhanden ist, welche die Konstante im Rumpf enthält. Zum Schluss wird die Anzahl solcher Regeln für jedes Wort ausgegeben.

`npKongruenz` soll dabei helfen, alle NPs mit möglichst vielen Konstituenten zu finden. Hierzu wird für jedes vorkommende N^0 eine Liste generiert. Jede dieser Listen enthält alle Worte, die dem N^0 im Satz vorausgehen. Im Anschluss wird ein Query `n2(...)` an die Grammatik gestellt, ob der vorliegende Teilsatz eine grammatische NP ist. Ist dies nicht der Fall, so wird das erste Element der Teilliste gestrichen, und das Query nochmals für den Rest durchgeführt. Wiederholt wird dies rekursiv, bis es ein Ergebnis gibt. Dann fährt man mit dem nächsten N^0 fort. Für diese Regel macht man sich zunutze, dass N^i-Projektionen immer rechtsverzweigend bezüglich der *Projektionslinie* sind.

`partAnalyse` durchläuft die Satzliste, bis sie eine satzeinleitende Konjunktion C^0 findet. Diese ist immer das erste Wort im Nebensatz, so dass wir die Satzliste *ab dieser Stelle* für die Nebensatzanalyse verwenden können.

5.8 Optimierung des Lexikons/der Laufzeit

Es wurde darauf verzichtet, Flexionsformen algorithmisch aus dem Wortstamm des Lexems generieren zu lassen. Der Grund liegt darin, dass die Implementierung der benötigten morphologischen Regeln und Gesetze den Rahmen der Diplomarbeit gesprengt hätte. Stattdessen sind alle Flexionsformen des Lexems gesondert im Lexikon gespeichert. Abhängig von der lexikalischen Kategorie hat ein Lexem zwischen 8 (Nomen) und 72 (z.B. Adjektive) Flexionsformen. Für eine Abschätzung eines realistisch großen Lexikons sollen grobe Anhaltspunkte genügen:

- 200.000 Lexeme
- 72 Flexionsformen pro Lexem (*worst case*, wahrscheinlich sind es im Durchschnitt weniger)
- pro Flexionsform ein Lexikoneintrag
- pro Lexikoneintrag eine Zeile
- pro Zeile 90 Byte (ungefähre maximale Länge)

$\Longrightarrow 200.000 \cdot 70 \cdot 90 = 1.260.000.000$ Byte = 1,173 Gigabyte [21]

Das Lexikon müsste zum Parsen eines Satzes durchsucht werden. 1,17 Gb sind heutzutage zwar keine prohibitive Größe, selbst wenn man "nur" lineare Suchalgorithmen verwendet, die nicht besser als $\mathcal{O}(n)$ [22] sind. Das Problem ist auch nicht der erhöhte Speicherverbrauch, sondern dass die Laufzeit des Prolog-Programms bei großen Lexika extrem stark ansteigt.

Optimierungsbedarf: Bei einem wenige Kilobyte großen Lexikon liegt die Laufzeit typischerweise bei einigen Sekunden/Minuten. Doch schon bei einem 4 Mbyte großen Lexikon steigt die Laufzeit schlagartig auf mehrere Stunden an, selbst wenn Grammatik und Eingabesatz noch die gleichen sind. Die oben ausgeführte Abschätzung macht deutlich, dass ein umfangreiches, praxistaugliches Lexikon sogar noch sehr viel größer ausfallen würde.

Dies ist durch die Suchstrategie von Prolog bedingt. Da Prolog auf der SLD-Resolution fußt, verwenden wir *SLD-Bäume*[23] zur genaueren Analyse. Die Tiefensuche mit Backtracking können wir auch als die Konstruktion eines SLD-Baumes verstehen. Die allgemeine Tiefensuche hat eine Laufzeitkomplexität von $\mathcal{O}(|V_{ges}| + |E_{ges}|)$, wobei $|V_{ges}|$ die

[21] Rechnung mit 1 Gb = 1024 Mb
[22] Definition der $\mathcal{O}$-Notation gemäß [CLR94]
[23] Definition siehe Abschnitt 4.2.2

Gesamtanzahl der Knoten und $|E_{ges}|$ die Gesamtanzahl der Kanten im SLD-Baum sind. Die Knoten des Suchbaums sind mit definiten Zielen G markiert. Jede Kante entspricht einem SLD-Resolutionsschritt.

Der SLD-Baum bei einem kleinen Lexikon ist exakt identisch mit dem eines großen Lexikons – sofern wir voraussetzen, dass die Menge derjenigen Wortformen in den beiden Lexika, die auch im Eingabesatz vorkommen, identisch ist. Der Unterschied ist jedoch, dass die Konstruktion mancher Kanten bei einem großen Lexikon sehr viel teurer ist, nämlich entsprechend der Größe des Lexikons. Sei L das Lexikon, welches von Prolog bei der Tiefensuche verwendet wird, $|L|$ dementsprechend die Größe des Lexikons. Für die Teilmenge $V_{lex} \subset V_{ges}$ gilt: die Knoten $\in V_{lex}$ sind mit $v_{lex,1}, \ldots, v_{lex,i}, \ldots, v_{lex,n}$ aufzählbar und jeweils mit einer Klausel G_i markiert. Für jedes G_i gilt: das erste Atom der Klausel ist ein Funktor, dessen Definiton[24] in der Lexikondatei steht, und somit eine lexikalische Kategorie repräsentiert. Also `v0`, `p0`, `a0`, `n0`, `c0`, usw. Durch je einen erfolgreichen Resolutionsschritte lassen sich die Klauseln $G_{i,1}, \ldots, G_{i,j}, \ldots, G_{i,m_i}$ aus G_i ableiten. Die Knoten des SLD-Baums, die mit den $G_{i,j}$ markiert sind, bezeichnen wir analog mit $v_{i,1}, \ldots, v_{i,j}, \ldots, v_{i,m_i}$.

Jede Kante entspricht einem erfolgreichen SLD-Resolutionsschritt. $E_{lex} \subset E_{ges}$ ist die Menge all derjenigen Kanten e, die einen Knoten $v_{lex,i} \in V_{lex}$ mit einem Nachfolger $v_{i,j}$ verbinden. $E_{rest} = E_{ges} \setminus E_{lex}$ sind alle "übrigen" Kanten.

Die Anzahl der Nachfolger m_i eines $v_{lex,i}$ ist klein, weil es bedingt durch die Eingabeliste $[\alpha_1, \alpha_2, \ldots]$ nur wenige Regeln `x0(...) --> [`α_1`]` im Lexikon gibt, die zusammen mit der Klausel G_i einen SLD-Schritt *erfolgreich* durchführen können. Ist $m_i > 1$, dann gibt es mehrere Flexionsformen im Lexikon, die oberflächlich identisch sind.

Hohe Kosten: eine Kante $e \in E_{lex}$ nimmt die Zeit $\mathcal{O}(|L|)$ in Anspruch. Bei den meisten Regeln `x0(...) --> [...]` scheitert, wie oben bereits erwähnt, durch die Eingabelisten bereits der Versuch, einen allgemeinsten Unifikator zu finden. Trotzdem versucht Prolog *genau dies* für alle Regeln. Prolog verfolgt nämlich die festgelegte Strategie, alle Regeln des Programms in derjenigen Reihenfolge auszuprobieren, in der sie im Quelltext vorkommen. Die Anzahl dieser Regeln korrespondiert in der Praxis auch mit der Größe des Lexikons. Insgesamt ergibt sich eine Laufzeitkomplexität von

$$\mathcal{O}(|V_{ges}| + |E_{rest}| + |E_{lex}| \cdot |L|)$$

Bei Tests ergab sich, dass die Kosten einer Kante $e \in E_{lex}$ auch für sehr große Lexika nur wenige Sekunden betragen. Da $|E_{lex}|$ aber nicht vernachlässigbar klein ist, summiert sich der Zeitverbrauch selbst für einfache Eingabesätze auf mehrere Stunden auf. Die Verbesserung beruht auf folgender Idee:

Optimierung Seien $\sigma_1 \ldots \sigma_n$ die Bezeichnungen für die Worte des Eingabesatzes. Aus dieser Auflistung entfernen wir alle Duplikate, so dass nur noch die Worte $\sigma_1 \ldots \sigma_m$ übrig bleiben, $m \leq n$. $|\sigma|$ bezeichnet im Folgenden die Kardinalität dieser *gekürzten* Liste.

[24] "Definition" im Sinne einer *definiten Klausel*

Generiere vor dem Start des eigentlichen Parse-Algorithmus dynamisch, und für jeden Eingabesatz gesondert, ein kleines Teillexikon L_{dyn}, welches nur die Worte $\sigma_1 \dots \sigma_m$ enthält. Mehrere Flexionsformen (die meist zu demselben Lexem gehören) können identisch sein. Daher suchen wir einfach alle DCG-Regeln `x0(...) --> [`σ_i`]` aus dem Lexikon L heraus, bei denen die Konstante σ_i im Rumpf mit dem zu parsenden Wort σ_i identisch ist. Für jede Flexionsform ist die Zahl lexikalischer Ambiguitäten in der Praxis gering. Die Kardinalität des dynamischen Lexikons ist demnach $|L_{dyn}| = c \cdot |\sigma|$, wobei $c \in \mathbb{R}$ und in der Praxis $c \ll 100$. c ist der Faktor, um den sich L_{dyn} durch oberflächlich identische, aber morphologisch voneinander verschiedene Flexionsformen vergrößert. Da wir ohne syntaktische Analyse nicht bestimmen können, welche der Flexionsformen gemeint ist, müssen wir zunächst alle aufnehmen. Lexeme, die in ihren Flexionsformen eine geringe Varietät aufweisen, tragen demzufolge stärker zur Vergrößerung von L_{dyn} bei. In geringerem Maße als lexikalische fließen semantische Ambiguitäten in den Faktor c ein; zwei semantisch verschiedene Worte haben möglicherweise nur wenige Flexionsformen gemeinsam. Bsp.: die beiden Lexeme *"Bank"* (Sitzgelegenheit, bzw. Geldinstitut) haben einen Singular gemeinsam, nämlich *"die Bank"*. Die Flexionsformen des Plurals differieren dagegen: *"die Bänke"* $\leftrightarrow$ *"die Banken"*.

Trotz der Ambiguitäten ist $|L_{dyn}| \ll |L|$ für große Lexika. Die Optimierung macht sich in der Laufzeit bemerkbar. Dies ist auch durch eine erneute Komplexitätsanalyse sofort einzusehen:

$$\mathcal{O}(|\sigma| \cdot |L| + |V_{ges}| + |E_{rest}| + |E_{lex}| \cdot |L_{dyn}|)$$

Das vollständige Lexikon muss also nur ein einziges Mal durchlaufen werden. Alle Tests mit Lexika, die nur geringfügig größer waren als $|\sigma|$, ergaben für den eigentlichen Syntax-Parser eine Laufzeit, die sich typischerweise im Bereich weniger Sekunden bewegt. Das Preprocessing benötigt nicht mehr als Linearzeit $\mathcal{O}(|\sigma| \cdot |L|)$ und nimmt auch bei Millionen von Einträgen nur wenige Minuten in Anspruch. Würde die Generierung von L_{dyn} anders implementiert, können sogar noch weitere Optimierungen erzielt werden: auch in C++ findet die Generierung in Linearzeit statt, jedoch ist C++ in der Ausführung um ein Vielfaches schneller als Python. Wäre das Lexikon dagegen in einer SQL-Datenbank oder einer passenden Datenstruktur wie etwa binären Suchbäumen gespeichert, so würde die Effizienz dank sublinearer Suchalgorithmen gesteigert werden.

6 Python-Interface

Prolog erfüllt zwar die Aufgabe der Satzverarbeitung und Fehleranalyse problemlos, jedoch werden weitere Schnittstellen benötigt:

- Der Eingabestring muss in eine Form gebracht werden, die Prolog überhaupt verarbeiten kann.
- Neue Lexikoneinträge müssen einfach hinzugefügt werden können.
- Die Ergebnisse des Parsers sollen in eine optisch ansprechende Form gebracht werden.

Für die Formen, die eine Eingabe der Grammatik haben darf, sind sehr enge Grenzen gesteckt. Jede Eingabe für den Parser muss eine Liste von Konstanten sein. Es handelt sich dabei um Worte in Kleinschreibung, die durch Kommas abgetrennt und durch eckige Klammern `[ ]` eingefasst sind. Beispiel: `[peter,kocht,den,kaffee]`. Dies entspricht jedoch nicht der schriftlichen Standardnotation natürlicher Sprache; dort würde das eben genannte Beispiel vielmehr so aussehen: *"Peter kocht den Kaffee."*. Sonderzeichen z.B. aus dem Unicode sind zwar prinzipiell erlaubt, können jedoch bei der Portierung zwischen verschiedenen Plattformen (Linux ↔ Windows !!) zu Problemen führen.

Vor dem Aufruf des eigentlichen Parsers muss also der Eingabestring durch passende Ersetzungen entsprechend verändert werden. Dies gelingt sehr leicht mittels regulärer Ausdrücke, die Prolog für die Stringverarbeitung jedoch nicht von Haus aus unterstützt [MANL08]. Hinzu kommt, dass selbst simple Stringkonkatenationen in Python wesentlich kompakter zu schreiben sind als in Prolog.

Während der Satzverarbeitung muss der Benutzer die Möglichkeit haben, manuell neue Lexikoneinträge hinzuzufügen. Der Benutzer muss mittels eigenem sprachlichen Wissen ein Wort, welches nicht im Lexikon steht, einer lexikalischen Kategorie und den passenden morphologischen Attributen zuordnen. Um den Benutzer so gut es geht zu unterstützen, wird ein interaktives Menü verwendet. Prolog ist auf Benutzerinteraktion schlechter eingerichtet als Python.

Stringverarbeitung wird ebenfalls wieder benötigt, wenn Prolog die Ergebnisse ausgegeben hat. Eine Ausgabe auf der Konsole gestattet lediglich die Darstellung der Satzstruktur mittels Klammernotation. Dies ist sehr unübersichtlich für komplexe X-Bar-Strukturen. Eleganter ist da die Darstellung von Baumdiagrammen mittels LaTeX. Python wandelt die Prolog-Ausgaben in LaTeX-Code um, schreibt diesen in eine `.tex`-Datei und kompiliert ein übersichtliches PDF. Voraussetzung ist, dass eine LaTeX-Distribution auf dem Rechner installiert ist, die die **Qtree**-Bibliothek[1] eingebunden hat. Das Qtree-Paket wurde eigens für Linguisten entwickelt.

[1] http://www.ling.upenn.edu/advice/latex/qtree/

Fazit: Das Python-Interface übernimmt hauptsächlich einfache Aufgaben der Stringverarbeitung oder bietet dem Nutzer Interaktionsmöglichkeiten. Dadurch können die Ein-/Ausgabeformate von Prolog und LaTeX, die ohne Vorwissen nicht intuitiv verständlich sind, vom Benutzer ferngehalten werden. Beispielsweise könnte auch Linguisten mit geringen PC-Kenntnissen der Einstieg in die computergestützte Sprachanalyse vereinfacht werden.

6.1 Python-Grundlagen

Python ist eine Interpreter-Sprache. Sie besitzt eine einfache, kompakte Syntax, so dass sie schnell erlernbar ist. Python wurde als Schnittstelle zum Prolog-Parser gewählt, weil sie plattformunabhängig ist. Es werden mehrere Programmierparadigmen unterstützt, z.B. funktionale und aspektorientierte Programmierung, für die Diplomarbeit wurden jedoch nur objektorientierte und strukturierte Programmierung benötigt.

Das Layout hat in Python eine feste Funktion: Anweisungsblöcke werden durch Einrückung gekennzeichnet, und Zeilenumbrüche entsprechen dem Anweisungsende. Eine explizite Ausweisung von Anweisungsblöcken/Methoden/Klassen/Anweisungsende durch Klammerung wie z.B. in Java entfällt somit. Die Syntax ist sehr übersichtlich.

Python ist schwach dynamisch getypt, d.h. wir müssen nicht explizit Variablen deklarieren. Ihre Datentypen werden vom Python-Interpreter aus dem Kontext heraus ermittelt. Python ist in dieser Hinsicht auch sehr flexibel: im Gegensatz zu Java ist es z.B. möglich, dass die Elemente eines Arrays voneinander verschiedene Typen haben. Insgesamt kommt der Programmierer also wenig mit Typisierung in Berührung. Die automatische Garbage Collection vereinfacht den Entwurf von Programmen zusätzlich.

Python hat den großen Vorteil, dass es eine Vielzahl von Modulen "von Haus aus" bereitstellt, die Module müssen lediglich ins Programm importiert werden. Wir benötigten grundlegende Operationen wie for-Schleifen, if-elif-Konstrukte, I/O-Operationen auf Dateien und Handling von (mehrdimensionalen) Arrays. Auch speziellere Funktionen wie z.B. zur Bearbeitung großer Dateien stehen uns sofort zur Verfügung.

Die mächtige Stringverarbeitung trug ebenfalls zur Entscheidung bei, Python zu verwenden: ein String kann wie ein Array behandelt werden, dessen Elemente die einzelnen *Characters* sind. Neben einer Vielzahl von Methoden (`find`, `delete`, `lower`/`upper`, etc.) werden auch reguläre Ausdrücke durch das `re`-Modul unterstützt.

6.2 Bestandteile

Die Implementierung umfasst drei Ordner, die alle in einem Verzeichnis stehen.

`IO/` : enthält die Dateien zur Benutzereingabe. In diesen Ordner werden auch die Ausgabedaten geschrieben.

`Grammatik/` : enthält das Lexikon, die Grammatikregeln und das Programm zur Fehleranalyse.

`Python-Interface/ :` hier sind das dynamische Lexikon sowie alle Python- und LaTeX-Quelltexte zu finden.

Für die Implementierung wurden folgende Python-Module entworfen:

`entry_2_lex` schreibt sortiert eine Klausel in das Prolog-Lexikon ein.

`generate_dynlex` Aus dem eigentlichen Lexikon wird ein kleines Lexikon generiert, in welchem ausschließlich Worte stehen, welche auch im momentan betrachten Eingabesatz vorkommen (siehe Abschnitt 5.8).

`nat_2_pr` wandelt den natürlichsprachlichen Eingabetext in Listen von Worten um. Die Wort-Strings werden zudem so angepasst, dass sie Prolog-Konstanten entsprechen (Kleinschreibung, alle Kommas, mehrfache Leerzeichen, Zeilenumbrüche, doppelte Satzzeichen gelöscht, alle Sonderzeichen/Umlaute ersetzt ...).

`MAIN.py :` Hauptklasse, die das Python-Interface steuert und alle anderen Module einbindet.

`performance_test` diente zur Ermittlung der durchschnittlichen Zugriffszeiten zeitkritischer Python-Operationen (`generate_dynlex`, `entry_2_lex`, `probe_lex`) auf das Lexikon. Sie ist jedoch nicht Teil der eigentlichen Implementierung.

`params` enthält alle veränderlichen Parameter, die auf unterschiedlichen Rechnern angepasst werden müssen. Wird von `MAIN` und `PFLEGE` importiert.

`PFLEGE.py` ermöglicht es, unabhängig von dem Parser, neue Lexikoneinträge zu erstellen.

`pr_2_pdf` wandelt die Textdatei, die der Parser ausgibt, in LaTeX-Quellcode um[2] und kompiliert mittels einer LaTeX-Distribution ein PDF als Ausgabe.

`probe_lex` bietet verschiedene Funktionalitäten, um die Existenz von Worten/Zeileneinträgen zu verifizieren, bzw. alle Lexikoneinträge eines Wortes zurückzugeben.

`word_2_entry` lässt den Benutzer über ein interaktives Kommandozeilen-Menü einem Wort morphosyntaktische Eigenschaften zuordnen. Linguistische/grammatische Grundkenntnisse sind hierbei Voraussetzung. Ausgegeben wird eine Prolog-Klausel, die genau einem Lexikoneintrag entspricht.

Jede Klasse ist in einer eigenen `.py`-Datei gespeichert. Sie alle werden in `MAIN.py` mittels `import` importiert, um ihre Methoden nutzen zu können. Der Lexikonzugriff folgt einfachen Prinzipien:

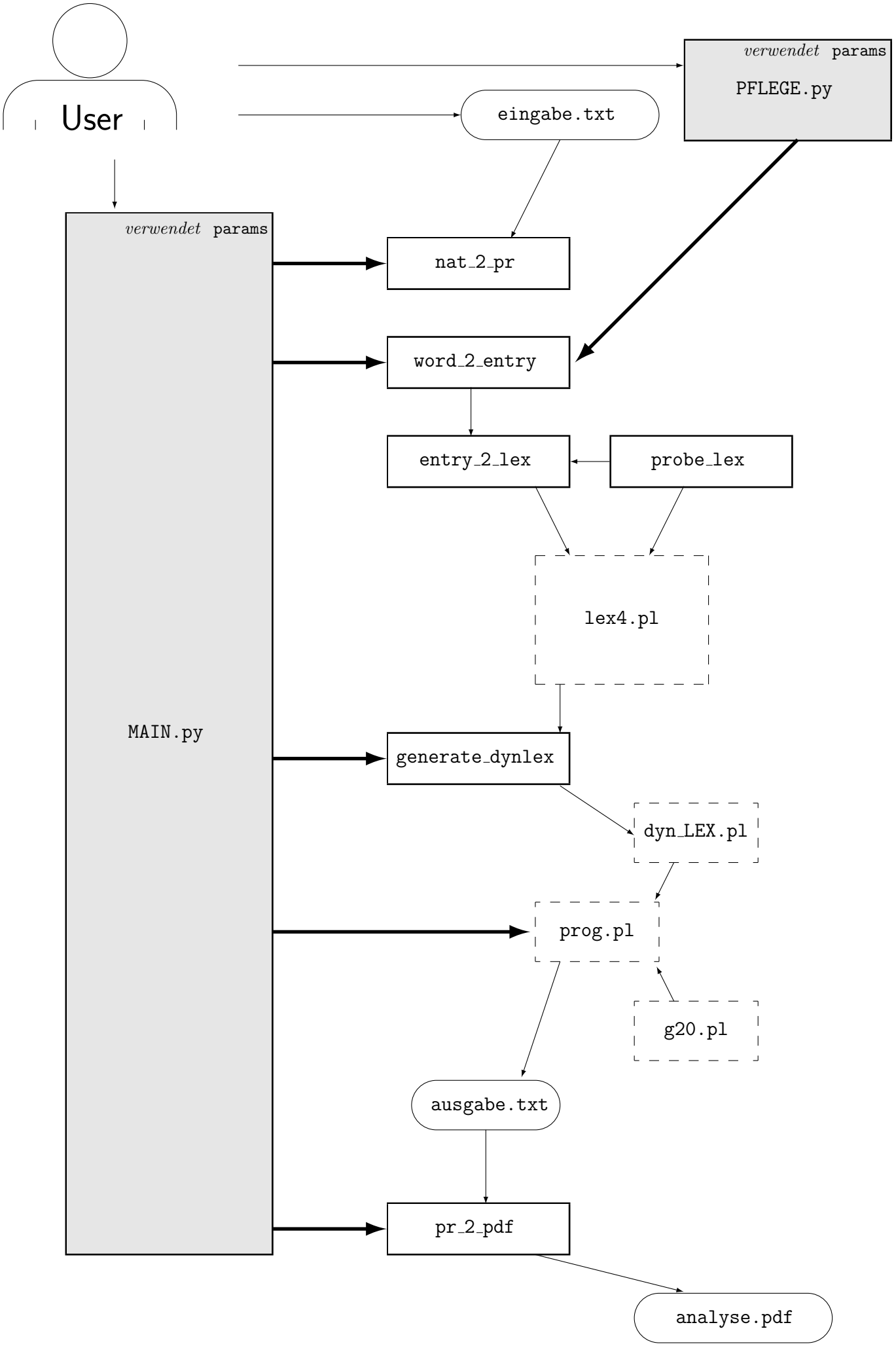
1. Jeder Lexikoneintrag entspricht einer Zeile in der Lexikondatei.

[2] insbesondere werden die als "Listen von Listen" repräsentierten Phrasenstrukturbäume in Code für das QTree-Paket umgewandelt

2. Die Lexikondatei wird sukzessive zeilenweise in einen String ausgelesen.

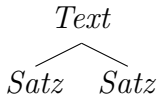
3. Auf dem String werden Operationen ausgeführt (Vergleiche, Ermittlung bestimmter Teilstrings usw.)

In folgendem Flussdiagramm ist die Funktionsweise des Programms dargestellt:


User
verwendet params
PFLEGE.py
eingabe.txt
verwendet params
nat_2_pr
word_2_entry
entry_2_lex
probe_lex
lex4.pl
MAIN.py
generate_dynlex
dyn_LEX.pl
prog.pl
g20.pl
ausgabe.txt
pr_2_pdf
analyse.pdf

6.3 Ausgabe der Ergebnisse

Die Analyse eines gegebenen Textes erfolgt satzweise und im Bezug auf die syntaktische Korrektheit. Die Sätze – insbesondere in sich abgeschlossene – werden syntaktisch nicht zueinander in Beziehung gesetzt. Strukturen wie



sind ausgeschlossen. Dazu heisst es in [DUDG06, S. 775]:

> *"Außerhalb des Gegenstandsbereichs der Satzlehre oder Syntax liegt alles, was kleiner als ein Wort und größer als ein Satz ist. Zum einen betrifft es [...] Morphologie, zum anderen betrifft es Text, den Gegenstand der Textlinguistik."*

Das Programm lässt sich auch ohne installierte LaTeX-Distribution/PDF-Reader betreiben, da alle Informationen, die ins PDF gelangen, vorher vom Programm in die Datei `IO/ausgabe.txt` geschrieben werden.

6.4 Einrichtung

Sowohl Prolog als auch Python sind Interpretersprachen. Um die Bedienung und die Programmierung möglichst einfach zu halten, ist das Programm für die Kommandozeile ausgelegt. Es müssen nur wenige Variablen angepasst werden, um es auf unterschiedlichen Rechnern lauffähig zu machen. Besteht hierfür dennoch Bedarf, so kann eine Anpassung durchgeführt werden:

params.py: Python unterstützt relative Pfade. Aus diesem Grund muss in dieser Datei nur etwas geändert werden, wenn man eine grundlegende Änderung der Interface-Struktur wünscht; z.B. folgende Variablen:

`IO_PATH` : Der Pfad, unter dem die Ein-/Ausgabedaten zu finden sind.

`GRAMMAR_PATH` : Pfad, unter dem das Lexikon, Grammatik und Auswertungssteuerung zu finden sind.

prog.pl: In dieser Datei zur Steuerung der Fehleranalyse/Auswertung müssen die Pfade in der Klausel `loadFiles` angepasst werden. Prolog setzt voraus, dass entweder alle geladenen Dateien im selben Arbeitsverzeichnis liegen oder – falls das wie hier nicht möglich ist – *absolute* Pfade angegeben werden.

6.5 Bedienung

Der Benutzer muss in die Datei `IO/eingabe.txt` einen oder mehrere Eingabesätze eingeben. Jeder Satz muss durch einen Punkt, ein Frage- oder Ausrufezeichen abgeschlossen sein, und die Rechtschreibung muss korrekt sein. Kommas, mehrfache Leerzeichen,

Zeilenumbrüche, doppelte Satzzeichen werden automatisch entfernt. Sonderzeichen und Umlaute werden ersetzt. Anschließend kann das Programm auf der Kommandozeile gestartet werden:

```
...Programm/Python-Interface> python MAIN.py
```

Befinden sich Worte, die noch nicht im Lexikon stehen in einem der Eingabesätze, so wird der Benutzer für jedes neue Wort aufgefordert, diesem lexikalisch-morphologische Eigenschaften zuzuordnen. Hierzu wird ein interaktiv geführtes Multiple-Choice-Menü eingesetzt, in dem jeweils mehrere Optionen nummeriert aufgelistet werden. Durch Eingabe der entsprechenden Zahl wird die Option im Menü ausgewählt. Das Programm speichert die Wahl des Users und springt dann automatisch zum nächsten Untermenü. Als erstes muss eine lexikalische Kategorie gewählt werden. Abhängig von dieser Kategorie wird der Benutzer gefragt, welche Ausprägungen die relevaten morphosyntaktischen Merkmale haben. Bei Verben wird zudem der Subkategorisierungsrahmen erfragt. Für die Bedienung dieses Menüs sind grammatikalische Grundkenntnisse (z.B. aus [DUDG06, S. 129 ff.]) unerlässlich. Sind alle Merkmale abgefragt, wird ein Lexikoneintrag generiert und das Menü für das nächste Wort erneut aufgerufen. Zum Abschluss werden alle Einträge sukzessive in das "Hauptlexikon" hineingeschrieben.

Im Anschluss sind keine weiteren Benutzer-Aktivitäten nötig. Die Ausgabe des Prolog-Parsers wird in die Textdatei `IO/ausgabe.txt` umgeleitet. Zusätzlich findet sich eine grafisch ansprechendere Präsentation der Ergebnisse (mit Baumdiagrammen) in `IO/analyse.pdf`.

Man kann mit dem Modul `PFLEGE.py` auch unabhängig vom Parser Lexikoneinträge erstellen. Zunächst ruft man das Modul mittels

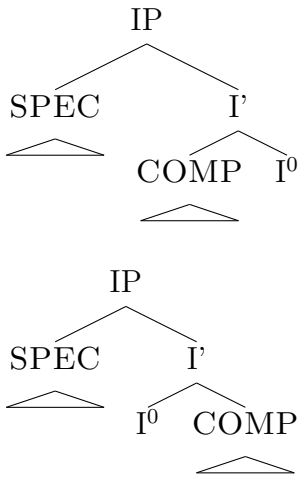
```
...Programm/Python-Interface> python PFLEGE.py
```

auf der Kommandozeile auf. Der Benutzer wird nun aufgefordert, ein Wort einzugeben. Der String wird, ähnlich wie es im Modul `nat_2_pr` geschieht, in eine Prolog-Konstante umgewandelt – Großschreibung, Sonderzeichen, Umlaute, Leerzeichen etc. werden ersetzt bzw. getilgt. Im Anschluss wird das bereits oben beschriebene Multiple-Choice-Menü aufgerufen, damit dem Wort seine grammatischen Merkmale zugeordnet werden können. Zum Abschluss wird aus den Benutzereingaben eine Prologklausel generiert und in das Lexikon eingetragen.

7 Anwendungsgebiete, Ausbau und Einschränkungen

7.1 Anpassung an andere Sprachen

Das GBT-Modell, insbesondere die CP-IP-Struktur, ist für alle natürlichen Sprachen gültig. Die wichtigste Anpassung des Programms an andere Sprachen wäre, die DCG-Regeln abzuändern. Durch die Modifikation der X-Bar-Regeln für lexikalische/funktionale Kategorien bezüglich Linksverzweigung, Rechtsverzweigung oder Mischformen bestimmt sich die mögliche Wortstellung einer Sprache. Bsp.: im Deutschen ist die IP rechtsverzweigend. Im Englischen dagegen steht I^0 links, die IP ist insgesamt eine Mischform bezüglich der Verzweigung. [STEI05]



Das Kasussystem ist nicht in allen Sprachen gleich: im Deutschen gibt es vier unterschiedliche Kasusfälle. Andere Sprachen unterscheiden wesentlich mehr, eines der reichhaltigsten Kasussysteme findet sich z.B. in der finnischen Sprache mit 15 unterschiedlichen Fällen.

Die Flexionsmerkmale, die die funktionale Kategorie I trägt, unterscheiden sich in unterschiedlichen Sprachen. Während I im Deutschen Tempus- und Kongruenzmerkmale tragen kann, besitzt z.B. I im Japanischen nur Tempusmerkmale.

In einigen Sprachen müssen zusätzliche Bedingungen berücksichtigt werden. Bsp.: im Englischen gilt die **Adjazenzbedingung**: zwischen Verb und Objekt darf keine weitere Konstituente stehen.

Manche Regeln müssen etwas abgeändert werden: im Italienischen sind NP und CP

Subjazenzbarrieren, nicht NP und IP wie im Deutschen und Englischen.

7.2 Unzulänglichkeiten und mögliche Erweiterungen

Die natürliche Sprache vollständig mittels eines Parsers zu erfassen, ist im Rahmen einer Diplomarbeit nicht möglich. Hier eine nicht vollständige Liste mit grammatischen Konstruktionen, die das Programm nicht erfassen kann, die jedoch als Erweiterung des Programms dienen könnten:

- Passiv wird nicht unterstützt. Insbesondere das Passivhilfsverb *"werden"* kann somit nicht verwendet werden.
- Verwendung von *"sein"* als Kopulaverb (siehe [DUDG06, S. 800]).
- Imperativ
- Verben mit Partikeln (*zugeben* $\longleftrightarrow$ *gab ... zu*)
- Prädikativ-/Funktionsverben ([DUDG06, S. 421])
- Koordinierende Konjunktionen sind für die Kategorien C, I und V noch nicht möglich.
- Erweiterung der Ausgabe-Bäume um Spur-Indizes, Pfeile etc.
- Mehrere Präpositionalphrasen im Vorfeld (*"*$[_{PP}$ *Auf der Mauer* $]$ $[_{PP}$ *auf der Lauer* $]$ *... liegt eine kleine Wanze"*)
- Weitere funktionale Kategorien aus der jüngeren linguistischen Theorie (siehe [STEI05, S. 54 ff.], [BDS06, S. 30 ff.]): **D** (*Determinator*), **Deg** (*Grad*), **Q** (*Quantifikation*) .... Die Implementierung hat ohne die Determinatorphrasen weitreichende Probleme, was *bestimmte Artikel* angeht. So kann in PPs nicht bestimmt werden, ob in der eingebetten NP ein Determinator vorkommen muss oder nicht[1]. Besitzanzeigende NPs[2] und Adjunktionen von Nomen an NPs[3] lassen sich lassen sich ohne **Determinatorphrasen** schlecht realisieren.
- Unterscheidung: Auffächerung der Selektion (siehe 3.12.6) in "starke" (durch Θ-Rollen) und "schwache" (strukturelle) Selektion.
- Nur eine Art von Relativsätzen kann erkannt werden: *"*[*das Auto* $[_{CP}$ *das Peter so gut gefällt* $]$ *"*. Relativsätze, die in COMP-N^0 basisgeneriert werden, können nicht ins Nachfeld bewegt werden (Rechtsadjunktion an IP)

[1] $[_{PP}$ *in* $[_{NP}$ *den Tunnel* $]$ $]$ $\leftrightarrow$ $[_{PP}$ *im* $[_{NP}$ *Tunnel* $]$ $]$
[2] $Neros_{Gen}$ *Zerstörung von Rom*
[3] $[_{NP}$ *Karl* $[_{NP}$ *der Große* $]$ $]$

- Schnittstelle zur Semantik (Θ-Rollen, Logische Form, wörtliche Bedeutung mittels Typenlogik, ...). Da Θ-Rollen nicht implementiert werden konnten, werden viele semantisch unsinnige Satzstrukturen als richtig erkannt.
- Linksrekursive grammatische Regeln, deren erzeugbare Strukturen nicht bezüglich der Tiefe beschränkt sind. Nötig z.B. für *Determinatorphrasen*, CPs im Vorfeld oder *koordinierende Konjunktionen*. Z.B. durch iteratives Vertiefen?
- Rektion einer beliebigen Konstituenten β durch α. Bisher ist nur die strikte Rektion im Rahmen des ECP im vollen Umfang implementiert. Wird eine Projektion π^i durch ein Verb α regiert, so führen wir keine Überprüfung auf Rektionsbarrieren durch. Das ist linguistisch gesehen unsauber, in unseren Regeln ist jedoch zumindest kein Fall denkbar, in denen eine solche Barriere eine Rolle spielt.
- Kontrolltheorie
- Implementierung von Modellen, die auf der mittels *GBT* ermittelten syntaktischen Struktur aufbauen (Logische Form, Phonetische Form, [LEUN04, S. 151])
- Noch mehr "Bausteine" (vgl. Abschnitt 5.4.1) für eine größere Vielfalt erkennbarer sprachlicher Konstrukte.
- Algorithmen/Regeln, die Theorien der Morphologie modellieren, insbesondere *Derivation/Komposition*. Dies würde die Größe des Lexikons stark reduzieren.
- Die Transformation freier Adjunkte (z.B. nach SPEC-C[1] (Vorfeld), Linksadjunktion an IP ...) wird noch nicht unterstützt.
- Aufzählungen (*"Peter, Paul und Maria"*)
- Subjekte können nicht aus dem Nebensatz in die SPEC-C'-Position des Matrixsatzes angehoben werden.
- Umfassendere Mechanismen zur Verwaltung des Lexikons (Löschfunktion, Sortierung etc.)
- Eine Nebensatz-CP sollte nur dann an die IP adjungiert werden (*Nachfeldposition*), auch dies durch die Wortordnung erzwungen wird.
- Die Größe der durch QTree erzeugten Phrasenstrukturbäume sollte berechnet werden können, damit sie im Ausgabe-PDF, wenn notwendig, herunterskaliert werden kann. Phrasenstrukturbäume für sehr lange/komplex geschachtelte Sätze verursachen nichtgewollte Seitenumbrüche und ragen häufig über den Seitenrand hinaus. Um letzteres zu begrenzen, ist das Seitenformat der PDFs auf DIN A3 gesetzt.

Ideengeber zur Erweiterung des Regelwerks könnten darüber hinaus z.B. [BDS06] sein, oder Veröffentlichungen, die sich mit Spezialfällen der deutschen Syntax befassen (z.B. [TRIS00]).

7.3 Performance-Test zur Skalierbarkeit

Eine hundertprozentig präzise Aussage über die Skalierbarkeit des Programms zu treffen, ist nicht möglich. Dennoch war es von großem Interesse, mittels einer groben Abschätzung zu erfahren, ob das Programm für den Einsatz mit einem großen Lexikon geeignet ist. Aus diesem Grund wurde ein Performance-Test durchgeführt.

7.3.1 Probleme

Folgende Unzulänglichkeiten waren bei einer Performance-Messung zu beachten:

- Wie groß ein "praxistaugliches" Lexikon tatsächlich ist, hängt stark von der intendierten Anwendung ab, eine repräsentative Wahl des Lexikons ist somit nur schwer zu treffen. Der deutsche Alltagswortschatz ist weniger umfangreich als von Spezialgebieten wie Medizin oder Rechtswissenschaft. Einen der Spitzenplätze nimmt dabei vermutlich die Fachsprache der Chemie ein, die weit über 20 Millionen verschiedene Stoffnamen kennt ([CHEM86]).

- Die Verteilung der lexikalischen Kategorien hängt von der Wahl des Lexikons ab. Es war nicht praktisch möglich, eine Verteilung zu ermitteln. Zu beachten gilt auch, dass der Wortschatz einem ständigen Wandel unterworfen ist: alte Lexeme und Flexionsformen werden ungebräuchlich, neue Wortformen und Neologismen der Sprache hinzugefügt.

Zum Testen wurde ein normaler Desktop-PC verwendet.

7.3.2 Funktionsweise

Trotz des Python-Menüs muss für jedes neue, ins Lexikon einzutragende Wort Sprecherwissen durch den Benutzer eingebracht werden. Ein großes Testlexikon kann auf diese Weise nicht befüllt werden. Es war auch kein bestehendes, elektronisch gespeichertes Lexikon verfügbar. Auch wenn man eines gefunden hätte, wäre aufgrund der "exotischen" Form, die das Prolog-Lexikon haben muss, eine Übertragung unverhältnismäßig aufwändig gewesen.

Aus diesem Grund wurden die Tests mit einem zufälligen Lexikon durchgeführt, auf welchem die beiden zeitkritischsten Operationen des Python-Interface – das Einfügen eines neuen Lexikoneintrags, und die dynamische Generierung eines Teillexikons – für fast beliebige Lexikongrößen praktisch getestet werden konnten. Statt realer Worte werden randomisierte Buchstabenfolgen zufälliger Länge verwendet; die Kombination morphosyntaktischer Merkmale ist ebenfalls randomisiert.

Das Testlexikon enthält nur die lexikalischen Kategorien N, A, V, P. Für jede dieser Kategorien wurde eine Größenvorgabe $l \in \mathbb{N}$ gemacht. Für die Erzeugung eines dynamischen Lexikons mit `generate_dynlex` ist eine eventuelle Sortierung des Lexikons völlig unerheblich – das gesamte Lexikon muss bis zum Ende durchsucht werden, weil wir uns

nicht sicher sein können, dass eine Flexionsform nur einer einzigen lexikalischen Kategorie angehört. Das Modul `entry_2_lex` soll eine neue Klausel in das Lexikon eintragen. Die Auswahl der Zeile erfolgt zuerst anhand der Kategorie und dann nach lexikographischer Ordnung der einzutragenden Wortform. Bei einem völlig zufälligen, unsortierten Lexikon wäre die Suche möglicherweise im Durchschnitt zu kurz. Da ein randomisiertes Lexikon nicht sinnvoll sortierbar war[4], kommt zumindest in jeder Kategorie jeder Anfangsbuchstabe in etwa gleich häufig vor[5]. Die Suche ist somit auch nicht zu kurz.

Beispiel. Ein kleiner Ausschnitt aus einem Zufallslexikon:

```
n0(mp(3,sg,neu,gen,pos,ind,praes,fini,inferg)) --> [zjrithztr].
n0(mp(1,pl,fem,dat,sup,kj1,praes,part1,voll)) --> [znxmljvvyzwxhi].
n0(mp(3,sg,neu,dat,pos,kj1,praes,inf,zuinferg)) --> [zlfx].
n0(mp(2,sg,mas,akk,com,kj2,praet,fini,voll)) --> [zykuhqtgi].

a0(mp(1,pl,mas,gen,com,kj1,praes,inf,zuinferg)) --> [arojdqtoicjj].
a0(mp(3,pl,mas,gen,com,kj1,praet,part2,inferg)) --> [aufehkyyb].
```

7.3.3 Überprüfung `generate_dynlex`

Um eine bessere Aussage zur Skalierbarkeit treffen zu können, wurden verschiedene Größenvorgaben l verwendet: 500, 5000, 50000, 500000, 2000000 und 4000000. Ebenfalls sollte der Einfluss der Anzahl von Worten s des Satzes auf die Performance begutachtet werden; s = 5, 10, 20, 40, 80.

Für jedes l wird einmalig ein Lexikon L_l erzeugt, was mehrere Minuten in Anspruch nehmen kann. Nachdem wir uns auf ein s festgelegt haben, wird das Lexikon durchsucht. Jeder Eintrag aus dem Rumpf einer Zeile wird dabei mit einer Wahrscheinlichkeit $s/|L_l|$ in eine Liste w von Worten aufgenommen. Wir stellen sozusagen einen zufälligen Satz aus unserem vorhandenen Wortschatz zusammen. Anschließend wird w an `generate_dynlex` übergeben, welches dynamisch ein zum Satz passendes Teillexikon generiert. Für jedes s wird dieser Vorgang 100 Mal mit unterschiedlichen Zufallssätzen wiederholt. Es wurden insgesamt also 3000 Testläufe mit 30 unterschiedlichen Konfigurationen durchgeführt.

Ergebnisse:

Spaltenbeschriftung: Vorgegebene Länge s der randomisiert gewählten Sätze.

[4] Die Sortierung hätte erstens ausschließlich anhand des Rumpfs der Klauseln, nicht der ganzen Zeilen, zu erfolgen. Zweitens hält Python keine Methode zum Sortieren sehr großer Dateien bereit. Ein solches Sortierverfahren zu implementieren wäre zu zeitaufwändig gewesen.

[5] $l/26 \cdot f$, wobei f eine Zufallszahl $\in [0.95, 1.05]$. Die Größe jeder Kategorie und damit des Lexikons ist also ebenfalls zufällig, jedoch ist die Abweichung prozentual gesehen nicht groß. Mit einer Wahrscheinlichkeit von 10% kann ein generiertes Wort ein weiteres Mal mit einem anderen Tupel morphosyntaktischer Merkmale im Lexikon auftauchen. Dies trägt der Tatsache Rechnung, dass manche Wortformen oberflächlich identisch sind, ihre morphologischen Eigenschaften jedoch voneinander abweichen.

Zeilenbeschriftung: Ungefähre Anzahl $|L_l|$ der Einträge des randomisiert generierten Lexikons, wobei $|L_l| \approx l \cdot 4$

Felder: Durchschnittliche Zeit für `generate_dynlex` bei 100 Test-Wiederholungen (Sekunden):

	5	**10**	**20**	**40**	**80**
2000	< 1	< 1	< 1	< 1	< 1
20000	< 1	< 1	< 1	< 1	< 1
200000	< 1	< 1	1,16	1,77	3,09
2000000	7,30	8,89	12,19	18,56	31,36
8000000	30,25	37,19	51,52	76,34	125,98
16000000	74,56	86,20	113,04	164,00	269,27

7.3.4 Überprüfung `entry_2_lex`

Wieder gab es mehrere Größenvorgaben l: 500, 5000, 50000, 500000, 2000000, 4000000. Im Anschluss wurden 100 zufällige Einträge in Form von Prolog-Klauseln generiert. Mittels `entry_2_lex` wurden diese im Anschluss sukzessive ins Lexikon eingefügt. Dabei wurde die durchschnittliche Zeit gemessen, die `entry_2_lex` zum Einfügen einer Zeile benötigt.

Ergebnisse:

Spaltenbeschriftung: Durchschnittliche Dauer, um neuen Lexikoneintrag in Datei zu schreiben, bei 100 Versuchen.

Zeilenbeschriftung: Ungefähre Anzahl $|L_l|$ der Einträge des randomisiert generierten Lexikons, wobei $|L_l| \approx l \cdot 4$

Felder: Durchschnittliche Zeit für `entry_2_lex` bei 100 Test-Wiederholungen (Sekunden):

Lexikongröße	**durchschnittliche Zeit**
2000	< 1
20000	< 1
200000	< 1
2000000	4,60
8000000	18,39
16000000	39,38

7.3.5 Fazit

Die Messergebnisse können höchstens eine Approximation an die Leistung bei einem tatsächlichen, umfangreichen Lexikon vermitteln. Trotzdem kann man das Resultat als vielversprechend einstufen. Der Zeitaufwand beträgt für jede Satzlänge $\mathcal{O}(|L|)$. Bei steigender Satzlänge wächst er sogar mit $o(\log \log|L|)$ sublogarithmisch. Insbesondere ersteres Ergebnis war zwar aus theoretischer Sicht zu erwarten. Es war jedoch gänzlich unklar, welche zeitliche Größenordnung die Implementierung in Python benötigt. Da sich diese im Bereich weniger Minuten selbst für Millionen von Einträgen bewegt, ist die Lexikonoptimierung mittels Python anwendungstauglich.

7.4 Praktische Verwendbarkeit: Parsen eines Wikipedia-Artikels

Obwohl das beschriebene System schon jetzt recht komplex ist, zeigt sich erst bei einer praktischen Anwendung, wie wenig grammatische Konstruktionen in Wirklichkeit erst abgedeckt sind. Als Beispiel hierfür wurden mehrere Sätze aus dem deutschen Wikipedia-Artikel über Wale geparset:

http://de.wikipedia.org/wiki/Wale $\longrightarrow$ *Merkmale* $\longrightarrow$ *Äußere Anatomie*

Jeder Satz wird gesondert aufgelistet. Zu Sätzen, die sich mit dem Parser auf seinem jetzigen Entwicklungsstand nicht parsen lassen, wird kurz erläutert, wo die Schwierigkeiten liegen. Die meisten Sätze können durch kleine Modifikationen dennoch als Beispiel verwendet werden, indem problematische Konstituenten ausgelassen oder durch einfachere Konstruktionen ersetzt werden. Aus Platzgründen stehen die Ergebnisse des Praxistests in Anhang B. Ob ein Text durch den Parser zu verarbeiten ist, hängt ausschließlich von den verwendeten syntaktischen Strukturen ab – es gibt z.B. signifikant einfachere Grundschultexte, die aufgrund ihrer Satzstruktur deutlich größere Probleme bereiten. Bei den meisten Sätzen werden vom Parser mehrere Lösungen gefunden. Sie alle verstoßen nicht gegen die Prinzipien der X-Bar-Struktur, viele sind jedoch ungrammatisch im Bezug auf die Θ-Theorie. Aus Platzgründen konnten nur diejenigen Ergebnisse aufgelistet werden, die im Zusammenhang mit dem Eingabesatz am sinnvollsten sind. Ist in einem Satz ein Relativsatz eingebettet, so dauert das Parsen oft mehrere Stunden.

Problematisch ist, dass das Verb *"sein"* in vielen unterschiedlichen Funktionen gebraucht werden kann. Als Perfekthilfsverb[6] kann es zwar vom Parser erkannt werden. Häufiger wird jedoch wird es für Zustandsbeschreibungen eines Sachverhaltes als **Kopulaverb** (siehe [DUDG06, S. 800]) gebraucht, welche der Bezugsnominalphrase eine Eigenschaft zuweist: [$_{NP}$ *Die Hauskatze*] *ist* [*ein Säugetier*]. Zumindest in der verwendeten sprachwissenschaftlichen Einführungsliteratur ist nicht geklärt, wie Kopulaverben angewendet werden können, und welche Eigenschaften die entsprechenden grammatischen Konstruktionen haben.

[6] Bsp.: "*das Zungenbein* [$_V$ *ist*] *verknöchert*"

7.5 Ausblick

Das Programm kann schon eine Vielzahl von Sätzen verifizieren, die in der linguistischen Einführungsliteratur oft verwendet werden. Jedoch schon mit der Alltagssprache zeigen sich erhebliche Probleme. Jede Erweiterung ist sehr aufwendig, da nicht nur die DCG-Regeln, sondern auch zahlreiche "normale" Prologklauseln angepasst werden müssen. Noch gar nicht berücksichtigte Regeln zu integrieren (z.B. aus [BDS06]), ist sogar noch komplexer, da sie mit dem bestehenden System abgeglichen werden müssen, das im Programm Rektion und Bindung überwacht. Trotzdem ist das System so geschrieben, dass eine Erweiterung nahtlos integrierbar ist und nicht "das System zum Einsturz bringt".

Ein guter Teilerfolg ist, dass Phrasenstrukturbäume (auch in der Fehleranalyse) ausgegeben werden können, und dass dabei erstmals auch neuere Entwicklungen in der *Rektions-Bindungs-Theorie* integriert werden konnten, die von älteren Implementierungen noch nicht berücksichtigt wurden (z.B. [DOUG94], [BRAT90, S. 437 ff.]).

Für eine praktische Anwendung, die über das Experimentieren hinausgeht, wäre ein wesentlich größerer zeitlicher Rahmen zu veranschlagen. Das Aufzeigen syntaktischer Ambiguitäten könnte helfen, "verschachtelte" Texte aus den Rechtswissenschaften zu analysieren. Auch ein Anfragesystem in der Medizin oder Chemie, das wenig oder keine Umgangssprache verwendet, wäre denkbar. Bsp.: *"Zeige alle chemischen Verbindungen, die Kohlenstoff enthalten."*.

Würde ich das Programm nochmal schreiben, so würde ich die Grammatik modularer gestalten und auf mehrere Dateien verteilen. Es ist ausserdem fraglich, ob man mittels einer objektorientierten Programmiersprache wie Java die unzähligen Beziehungen innerhalb eines Phrasenstrukturbaums nicht besser repräsentieren könnte.

8 Zusammenfassung

In Kapitel 1 wird der Bedarf für Syntax-Parser für natürliche Sprachen aufgezeigt, und bereits darauf hingewiesen, dass der Parser unmöglich die vielfältigen syntaktischen Konstruktionen und die lexikalische Reichhaltigkeit der deutschen Sprache abdecken kann. Anschließend werden kurz die wichtigsten verwendeten Quellen vorgestellt. Es wird auf den Unterschied der hier durchgeführten Implementierung zu den in den Quellen beschriebenen Implementierungen hingewiesen. Zuletzt wird beschrieben, wie das Syntax-Parsing eines Satzes unter den sprachwissenschaftlichen Theorien einzuordnen ist.

Kapitel 2 ist eine kurze Einführung in die Morphologie, die die grammatischen Merkmale einzelner Worte beschreibt. Dies ist wichtig, da die Syntaxtheorie diese Merkmale von Worten oder größeren Satzgliedern zueinander in Beziehung setzt. Daraus ergibt sich bereits eine Vorgabe für das Speicherformat des Lexikons.

In Kapitel 3 wird die *Rektions-Bindungs-Theorie* vorgestellt. Sie ist ein verbreitetes Modell der Sprachwissenschaft und enthält die linguistischen Formalismen, mit denen syntaktische Satzstrukturen aufgebaut und beschrieben werden können. Die Gliederung des Kapitels orientiert sich an den Modulen der Theorie.

Kapitel 4 beschreibt den theoretischen "Unterbau" der deklarativen Programmiersprache Prolog. Neben einigen syntaktischen Konventionen und Anmerkungen zur Umsetzung liegt das Hauptaugenmerk auf der Beschreibung der *SLD-Resolution*, welche die operationelle Semantik für Prolog bildet.

In Kapitel 5 wird dargelegt, wie die theoretischen Grundlagen aus Kapitel 2, 3 und 4 in einen Prolog-Syntax-Parser umgesetzt worden sind. Insbesondere wird auch beschrieben, wie die Anforderungen der linguistischen Theorie z.T. mit den Beschränkungen von Prolog kollidieren, und welche Lösungen dafür gefunden wurden.

In Kapitel 6 wird eine in der prozeduralen Programmiersprache Python geschriebene Benutzerschnittstelle vorgestellt. Diese ermöglicht es, das Programm auch ohne spezifische Kenntnis der Programmmechanismen und sogar ohne Programmierkenntnisse zu bedienen. Anschließend wird eine notwendige Laufzeitoptimierung des bestehenden Programms behandelt. Eine Begründung für den Schritt, dem eigentlichen Prolog-Kern ein Python-Interface vorzuschalten, ist auch enthalten.

Kapitel 7 befasst sich mit den Grenzen und dem Potential der Implementierung. Zunächst wird aufgezeigt, wie man den Parser sogar für andere natürliche Sprachen nutzbar machen kann. Nach einer anschließenden Liste von Fehlern, Schwierigkeiten und vielen Möglichkeiten zur Verbesserung des Codes werden ein Test zur Skalierbarkeit und zu einer praktischen Anwendung dokumentiert. Ein Fazit und Vorschläge zu möglichen Anwendungsgebieten runden das Kapitel ab.

Im Anhang A werden keine neuen Erkenntnisse mehr vermittelt. Stattdessen werden an dieser Stelle einige Ausgabeergebnisse des Parsers und der Fehleranalyse präsentiert.

Im Anhang B werden die Ergebnisse eines praktischen Tests besprochen, bei dem der Parser, soweit möglich, auf Teile eines Wikipedia-Artikels angewandt wurde.

Das Programm selbst wird auf auf einer CD-R bereitgestellt.

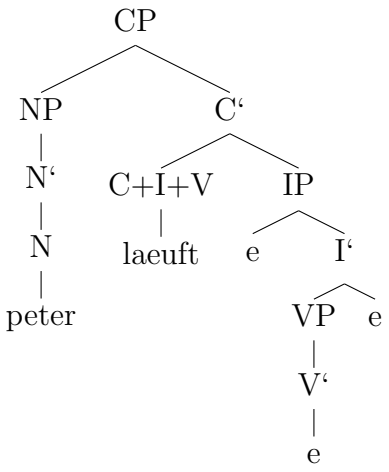
Eine ggf. aktualisierte Version findet sich auch unter folgender Adresse:

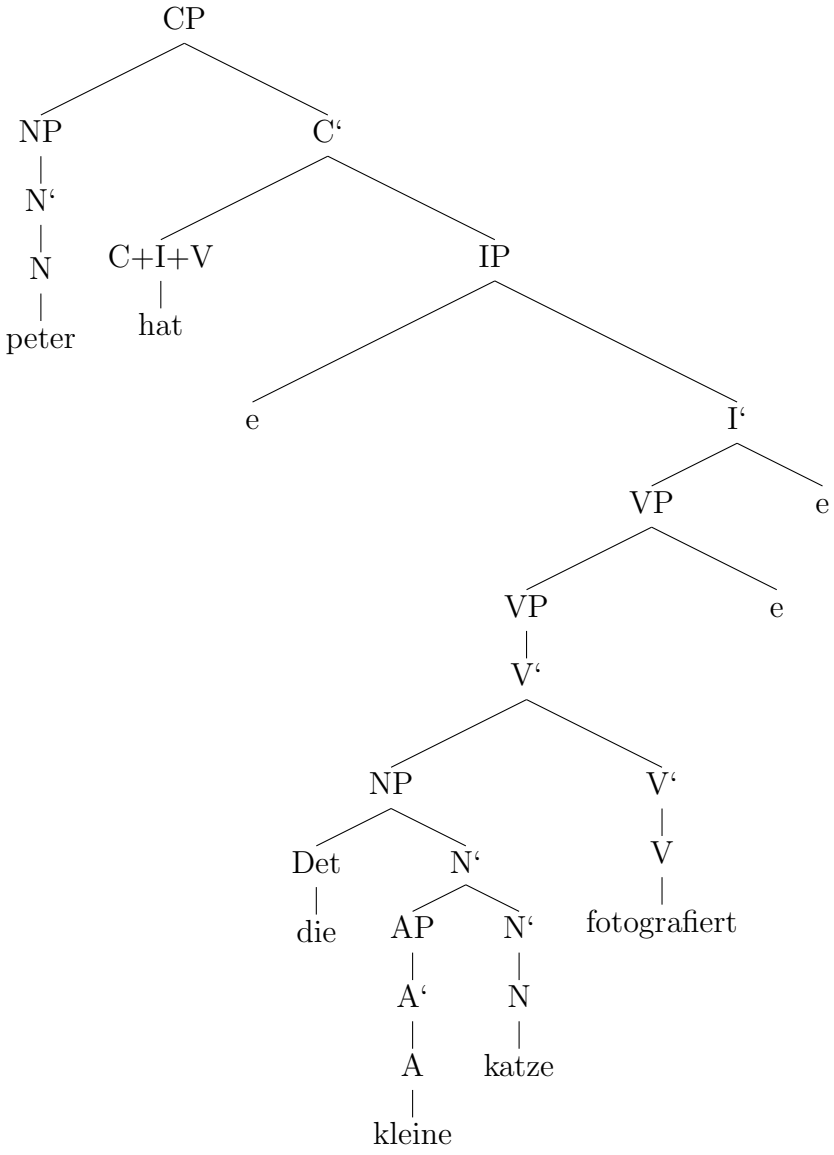
`http://www.informatik.uni-frankfurt.de/~sbehr/`

A Beispiele

In diesem Abschnitt werden einige Phrasenstrukturbäume präsentiert, die vom Prolog-Parser ermittelt wurden. Zum Teil wurde die Skalierung im QTree-Code etwas auf die Seitengröße angepasst, die Struktur der Bäume selbst wurde jedoch nicht modifiziert. Der LaTeX-Code, den der implementierte Parser ausgibt, konnte direkt übernommen werden.

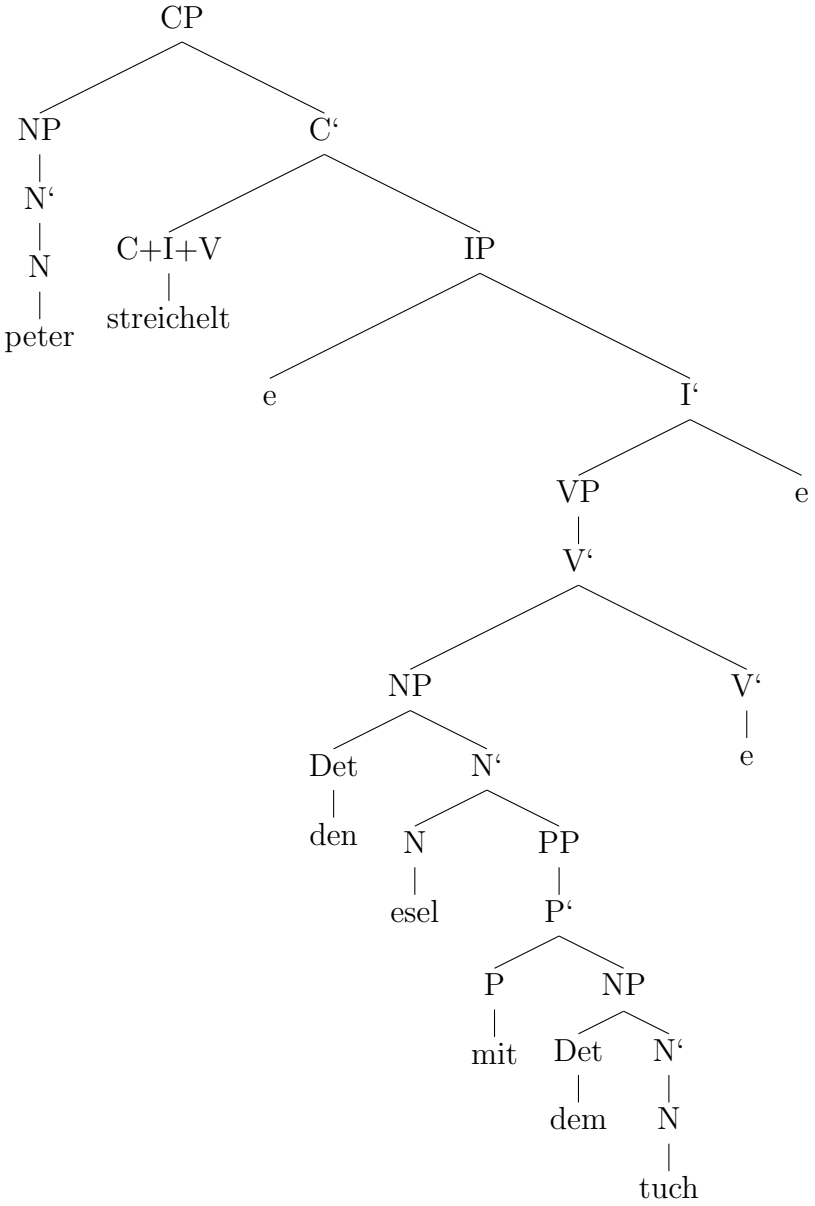
Deklarativsätze



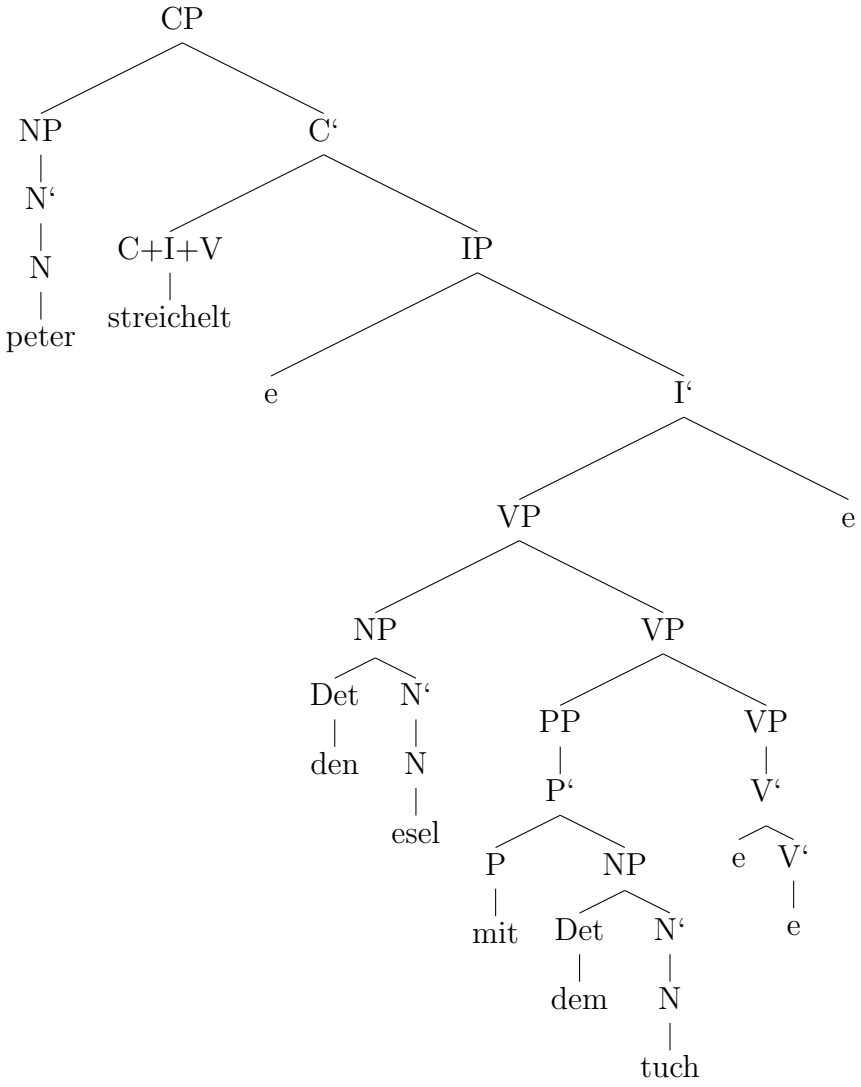

CP
NP
C‘
N‘
C+I+V
IP
N
hat
peter
e
I‘
VP
e
VP
e
V‘
NP
V‘
Det
N‘
V
die
AP
N‘
fotografiert
A‘
N
A
katze
kleine

Ambiguität

Lösung 1 :

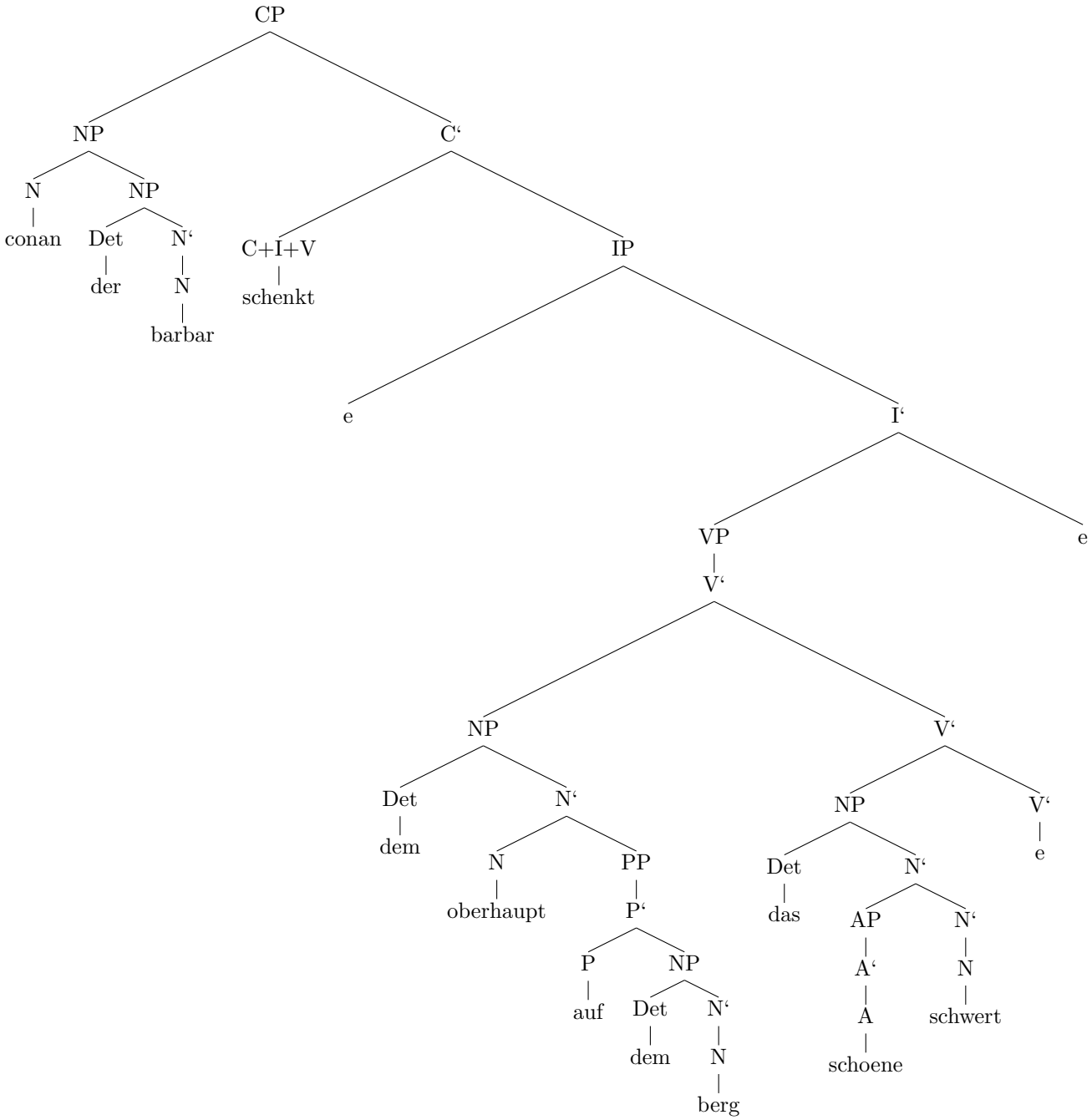


Lösung 2 :

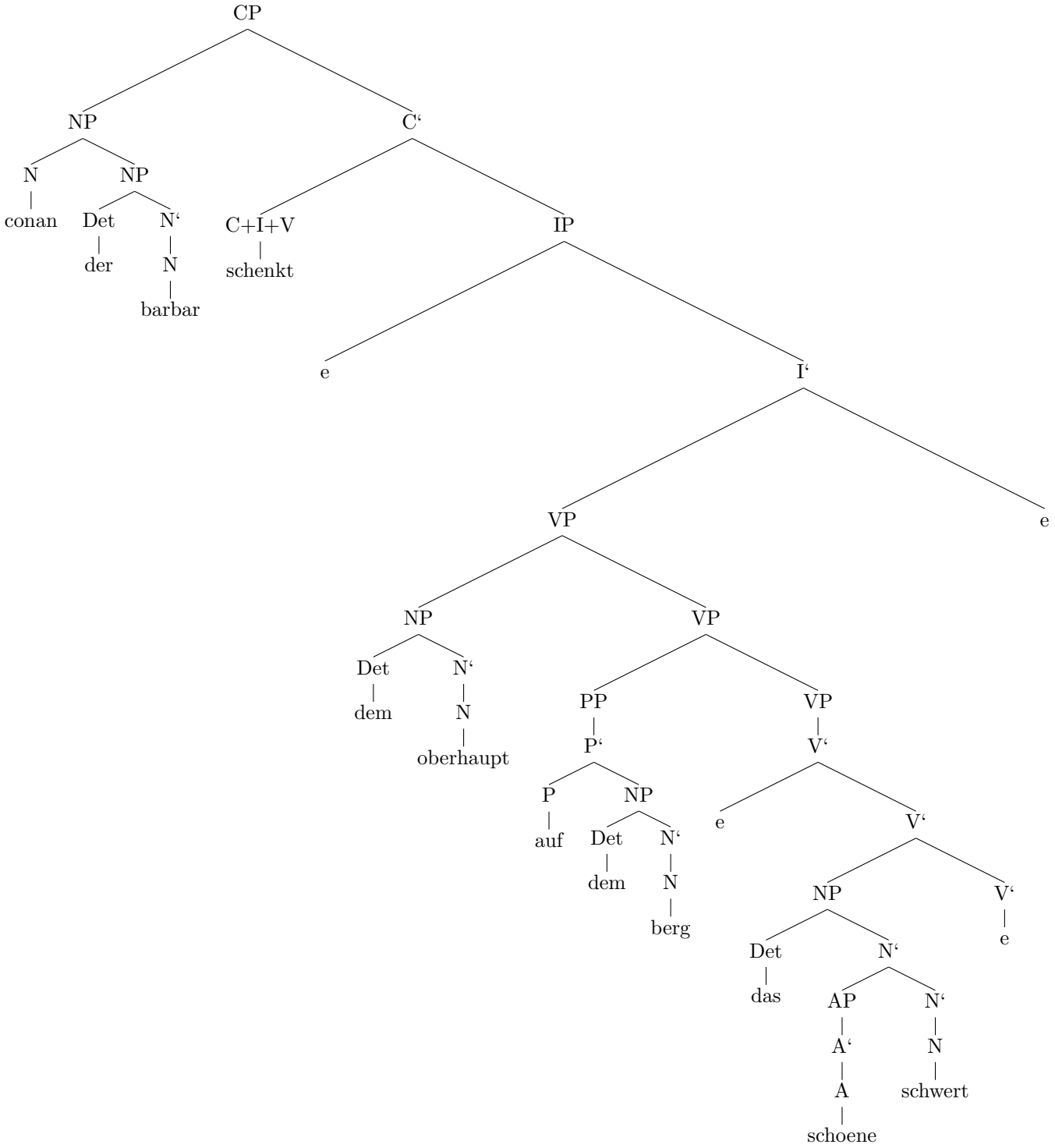

CP
NP
C‘
N‘
C+I+V
IP
N
streichelt
peter
e
I‘
VP
e
NP
VP
Det
N‘
PP
VP
den
N
P‘
V‘
esel
P
NP
e
V‘
mit
Det
N‘
e
dem
N
tuch

Ambiguität mit Scrambling

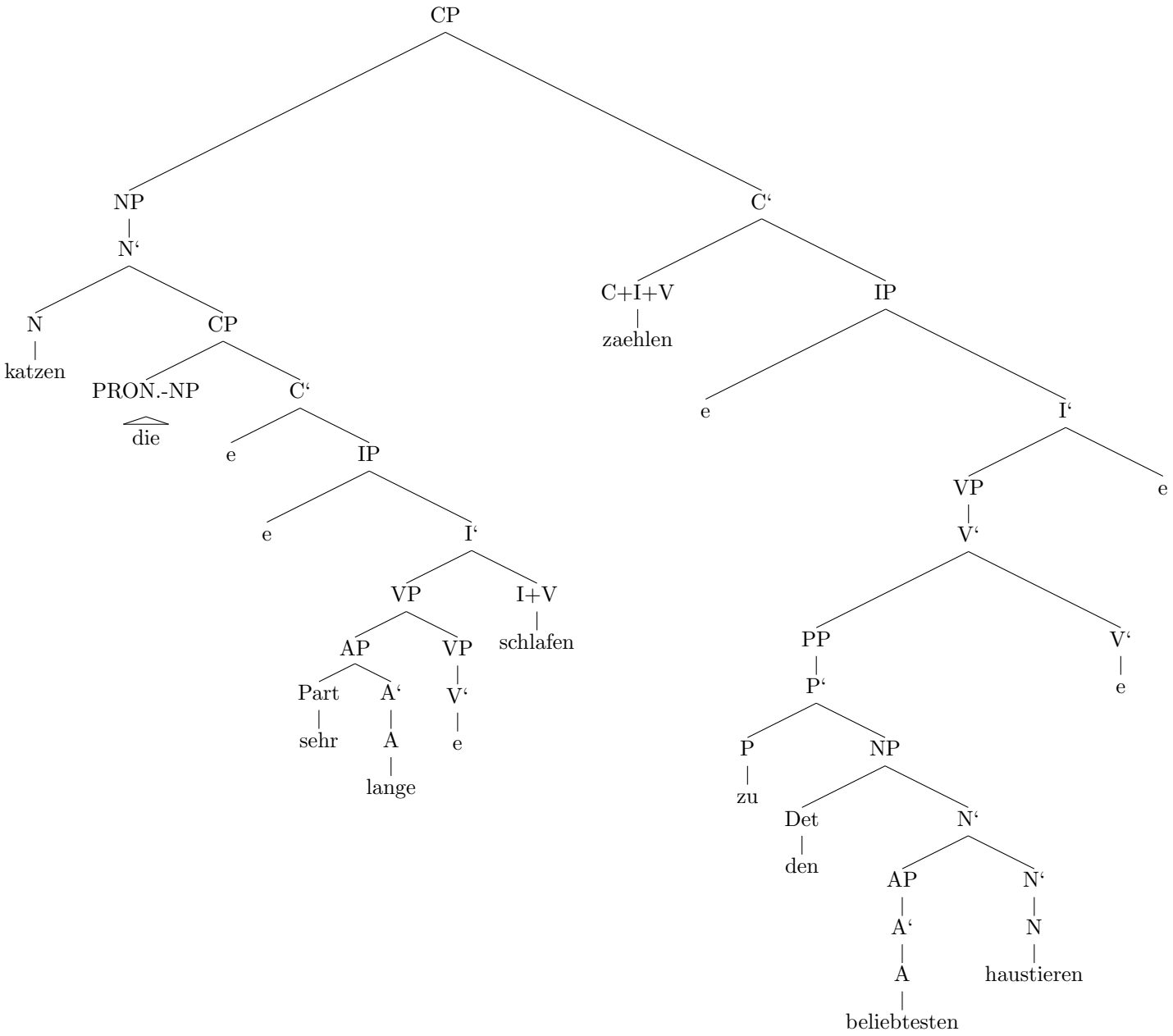
Lösung 1 :



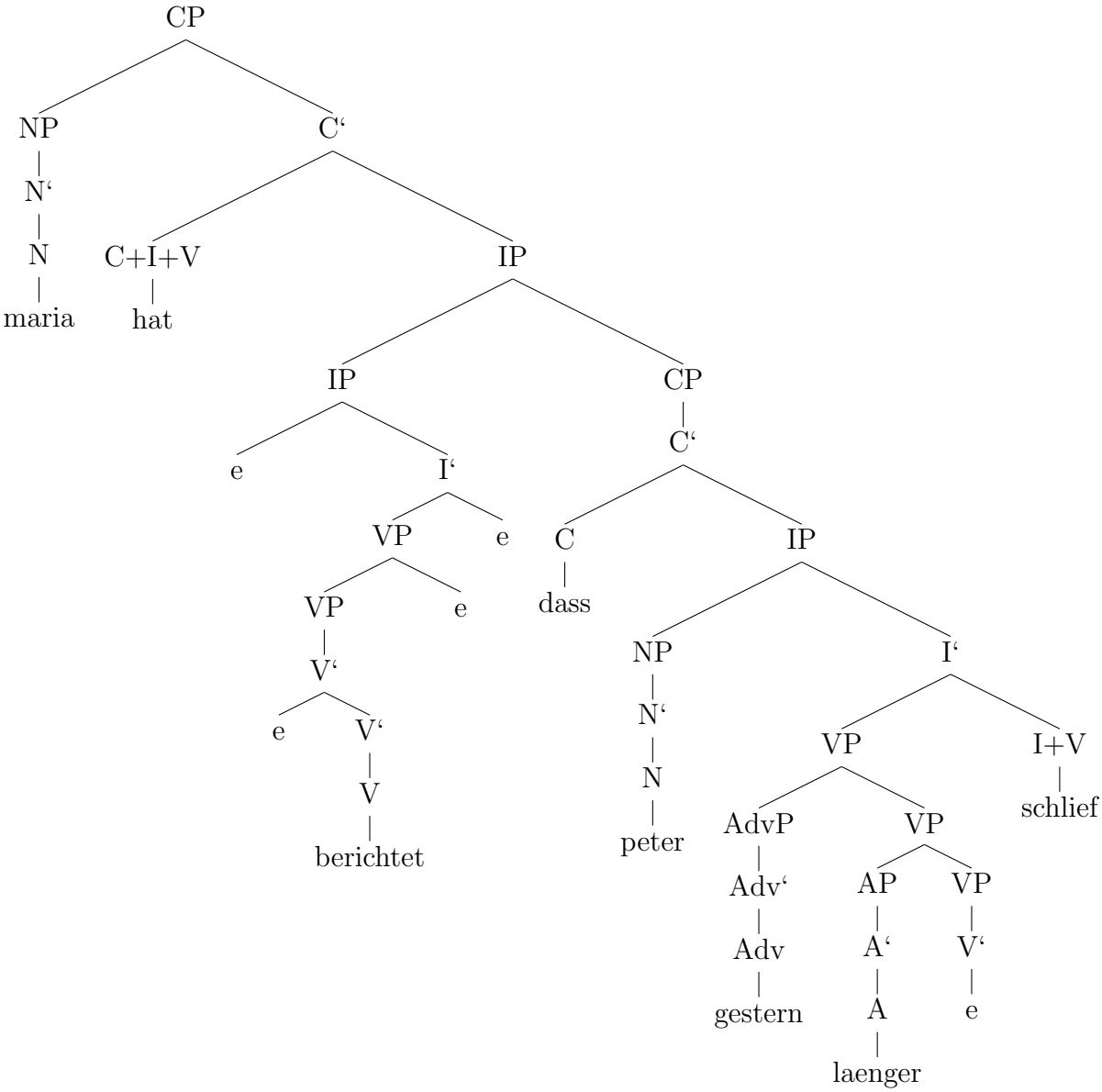
Lösung 2 :

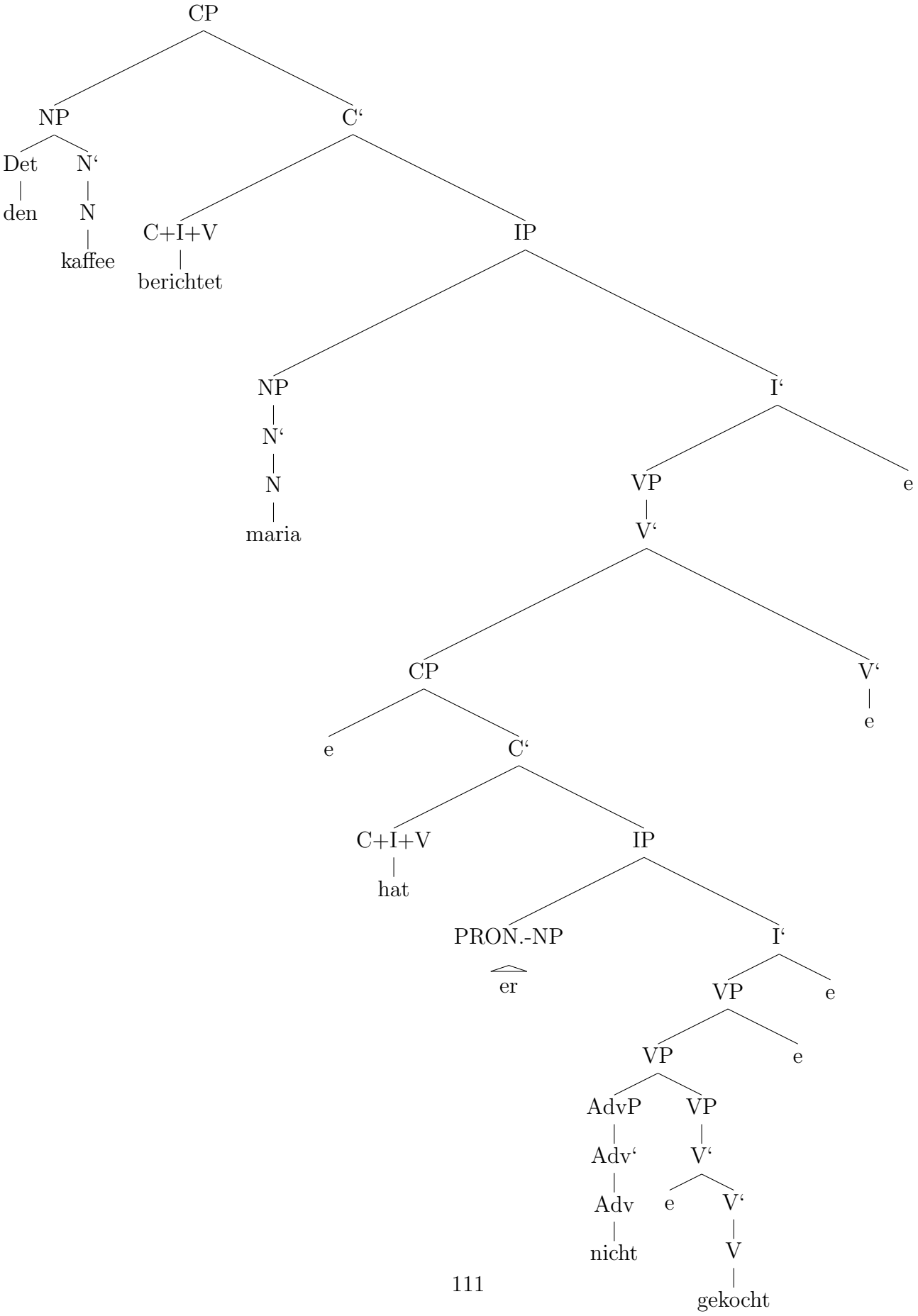

CP
NP
C‘
N
NP
conan
Det
N‘
C+I+V
IP
der
N
schenkt
barbar
e
I‘
VP
e
NP
VP
Det
N‘
PP
VP
dem
N
P‘
V‘
oberhaupt
P
NP
e
V‘
auf
Det
N‘
dem
N
NP
V‘
berg
e
Det
N‘
das
AP
N‘
A‘
N
A
schwert
schoene

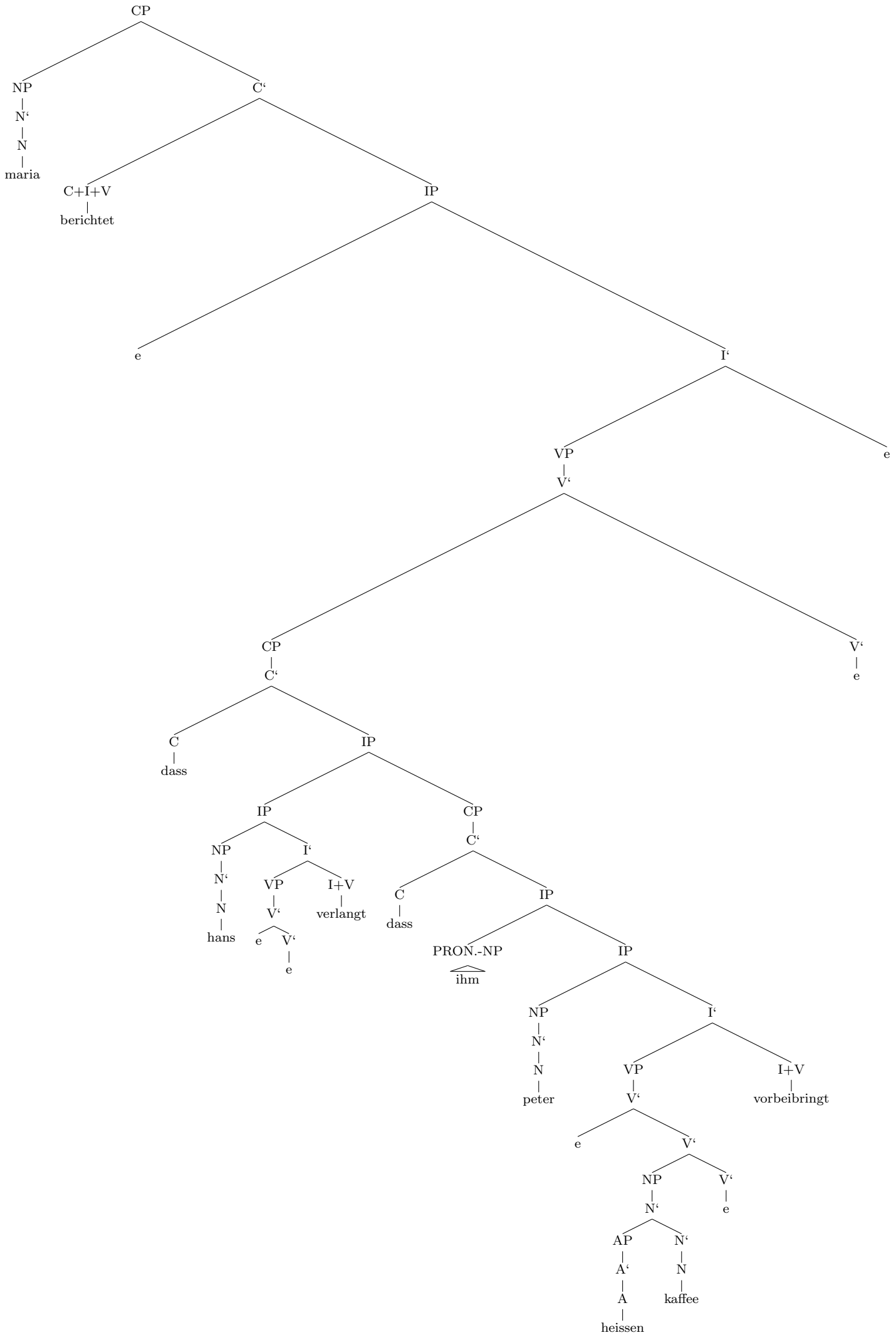
Subjekt mit Relativsatz



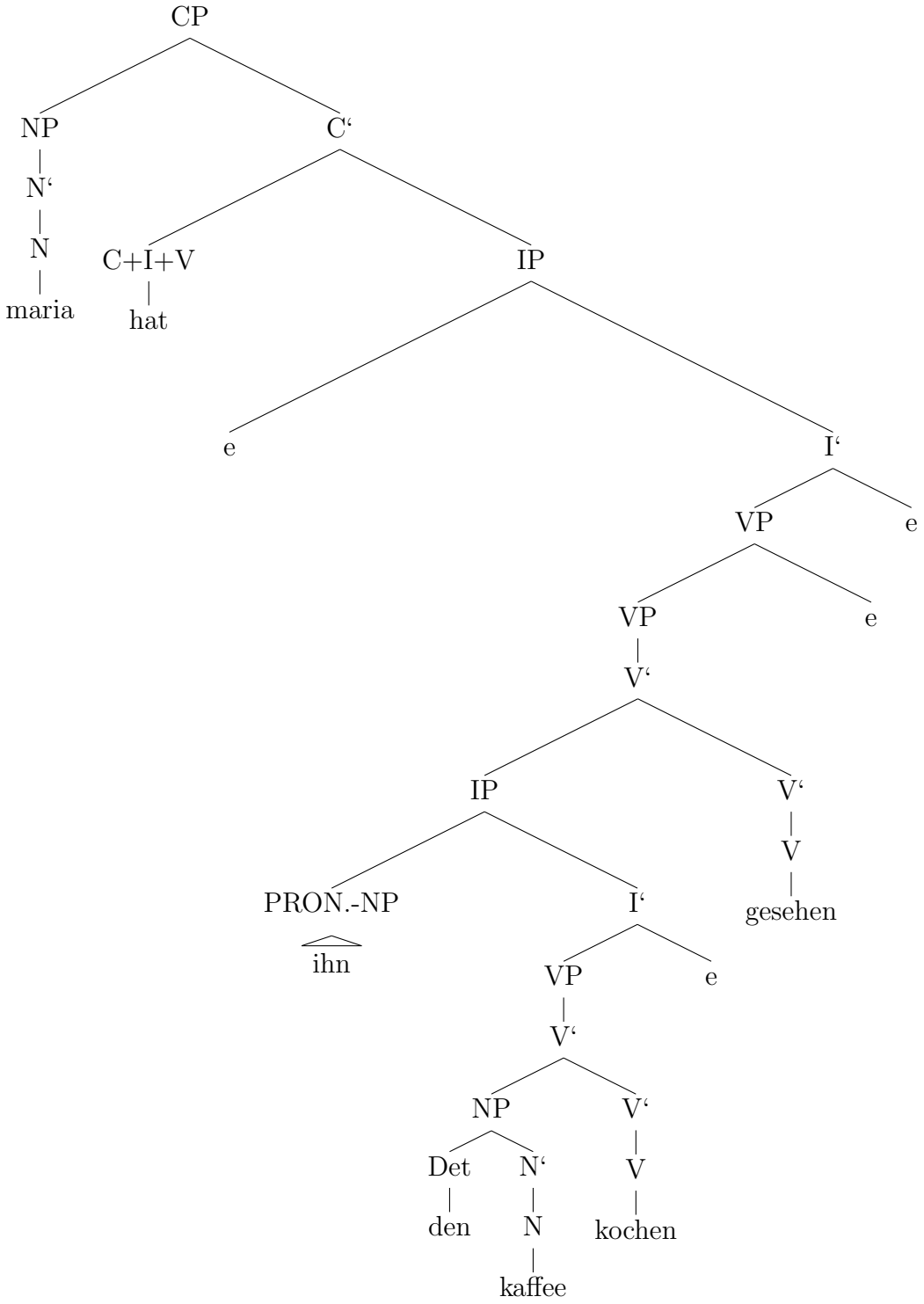
Nebensätze


CP
NP
C‘
N‘
N
C+I+V
IP
maria
hat
IP
CP
e
I‘
C‘
VP
e
C
IP
VP
e
dass
V‘
NP
I‘
e
V‘
N‘
VP
I+V
V
N
schlief
berichtet
peter
AdvP
VP
Adv‘
AP
VP
Adv
A‘
V‘
gestern
A
e
laenger

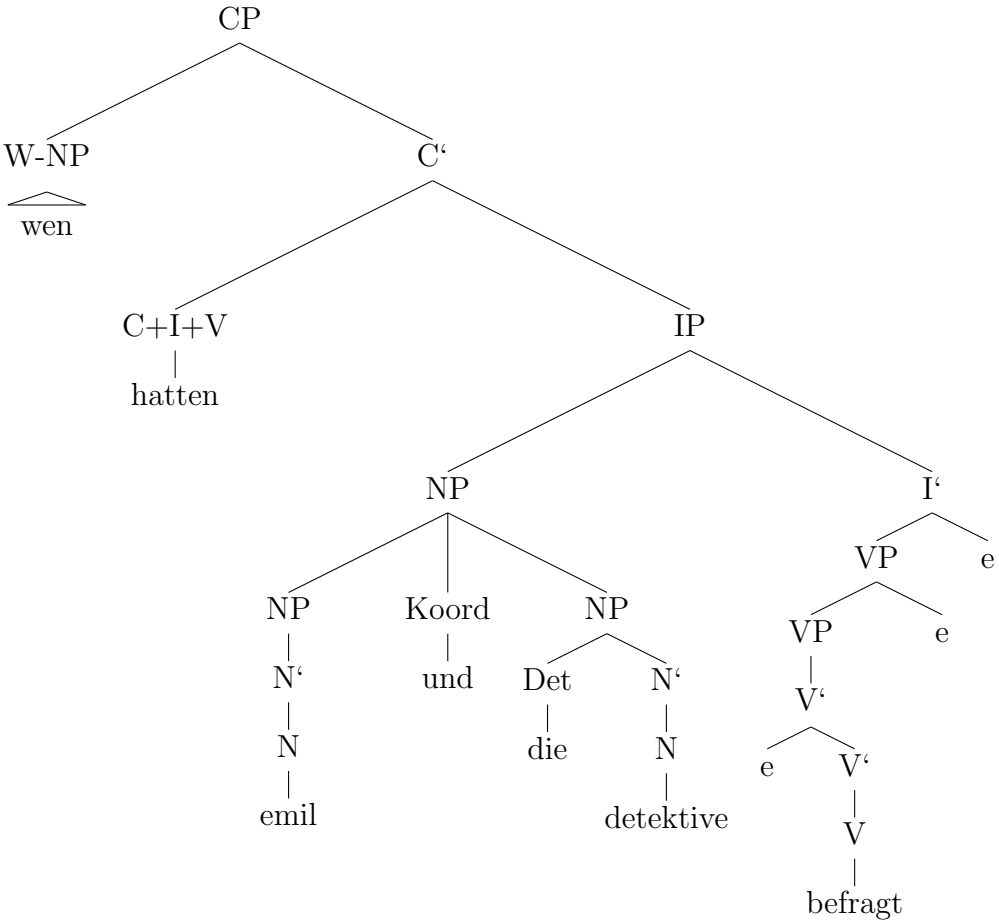

CP
NP
C‘
Det
N‘
den
N
kaffee
C+I+V
berichtet
IP
NP
I‘
N‘
N
maria
VP
e
V‘
CP
V‘
e
C‘
C+I+V
hat
IP
PRON.-NP
I‘
er
VP
e
VP
e
AdvP
VP
Adv‘
V‘
Adv
e
V‘
nicht
V
gekocht

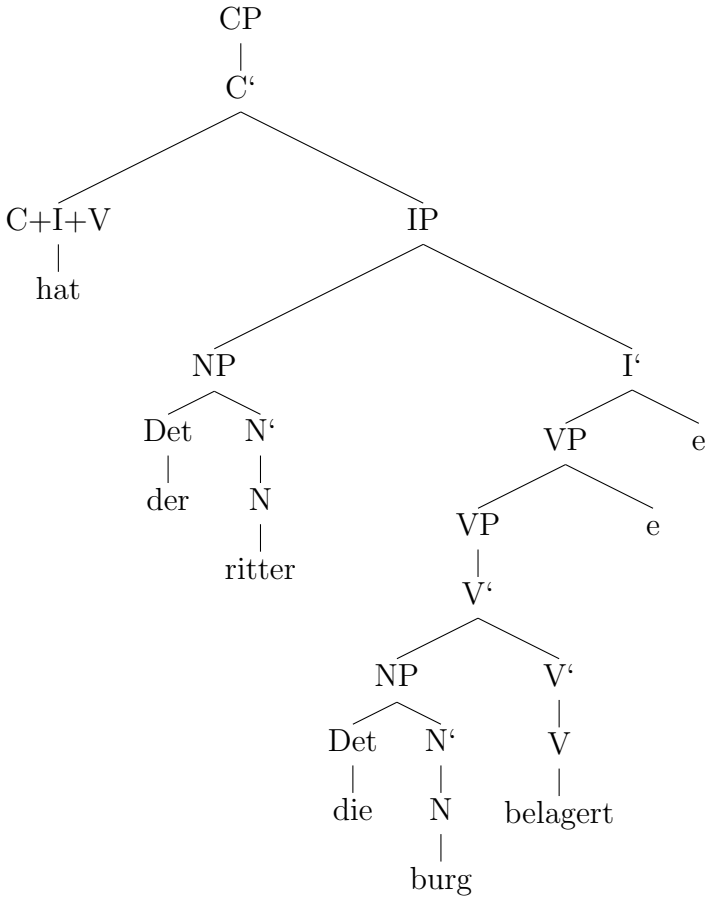

CP
NP
N‘
N
maria
C‘
C+I+V
berichtet
IP
e
I‘
VP
V‘
e
CP
C‘
V‘
e
C
dass
IP
IP
CP
NP
N‘
N
hans
I‘
VP
V‘
e
V‘
e
I+V
verlangt
C‘
C
dass
IP
PRON.-NP
ihm
IP
NP
N‘
N
peter
I‘
VP
V‘
I+V
vorbeibringt
e
V‘
NP
N‘
V‘
e
AP
A‘
A
heissen
N‘
N
kaffee

Satz mit A.c.I.-Verb



Interrogativsätze




CP
C‘
C+I+V
hat
IP
NP
Det
der
N‘
N
ritter
I‘
VP
e
VP
e
V‘
NP
Det
die
N‘
N
burg
V‘
V
belagert

Fehleranalyse

[maria,berichten,dass,peter,den,schoenen,heissen,kaffee,gekocht,hat]

Der Eingabesatz ist ungrammatisch.

Subjekt $\longleftrightarrow$ Verb : *Numerus* stimmt nicht überein.

Lexikoneinträge:

Wort	Anzahl Vorkommen im Lexikon
maria	1
berichten	1
dass	1
peter	1
den	3
schoenen	4
heissen	4
kaffee	1
gekocht	1
hat	1

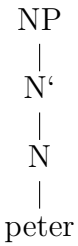
Einzelne Nominalphrasen analysieren:

[**maria**]

$$\begin{bmatrix} \textit{Numerus:}\ \textsc{Singular} \\ \textit{Kasus:}\ \textsc{Femininum} \\ \textit{Genus:}\ \textsc{Nominativ} \end{bmatrix}$$

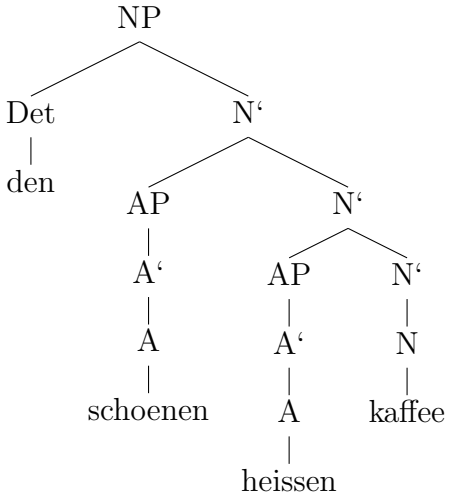
[peter]

$$\begin{bmatrix} \textit{Numerus:} \text{ Singular} \\ \textit{Kasus:} \text{ Maskulinum} \\ \textit{Genus:} \text{ Nominativ} \end{bmatrix}$$



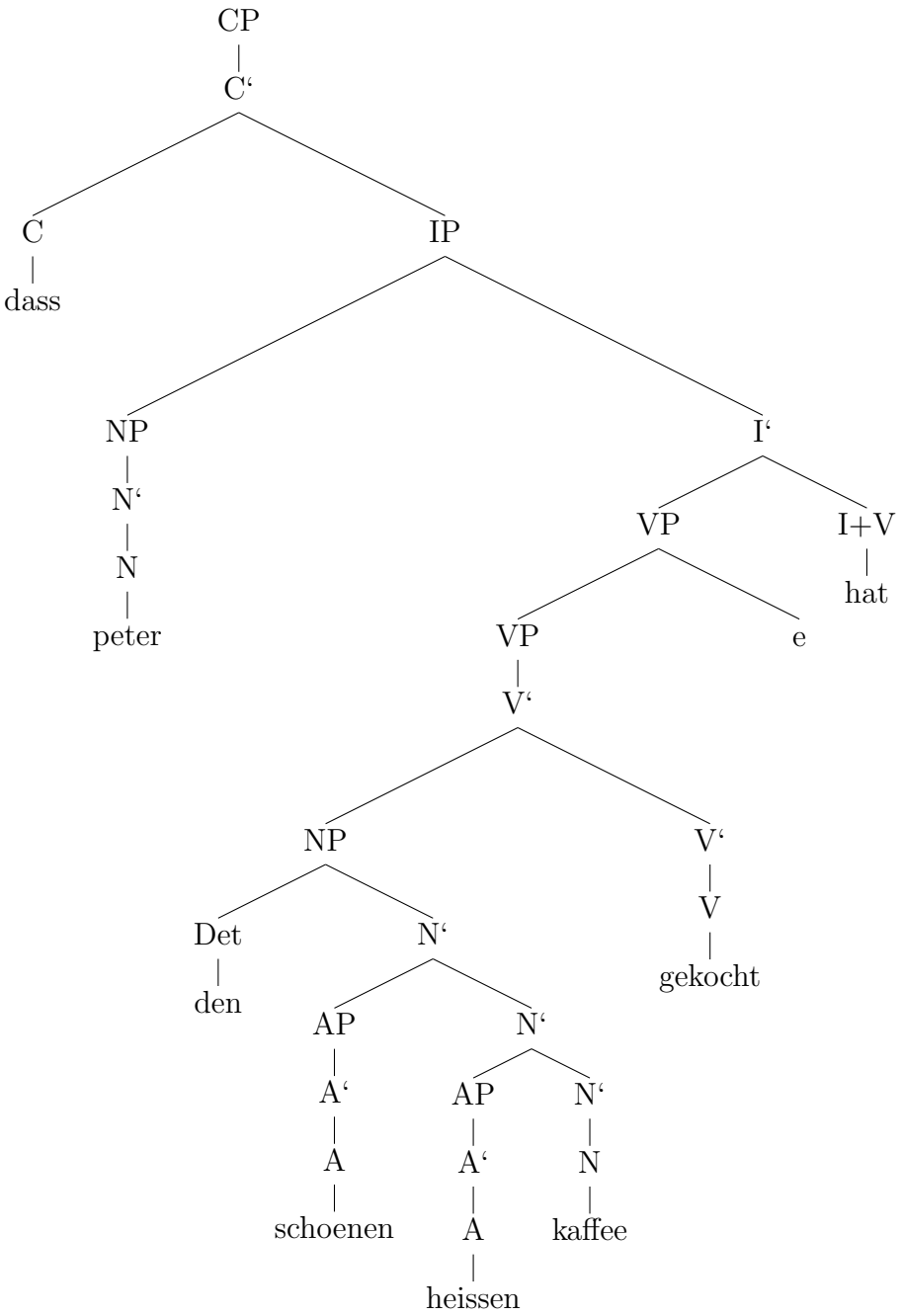
[den, schoenen, heissen, kaffee]

$$\begin{bmatrix} \textit{Numerus:} \text{ Singular} \\ \textit{Kasus:} \text{ Maskulinum} \\ \textit{Genus:} \text{ Akkusativ} \end{bmatrix}$$



Versuche separate Nebensatzanalyse ...

1. Lösung für Teilsatz



B Ergebnis Praxistest

In diesem Abschnitt werden die Ergebnisse des Praxistests aufgelistet, in dessen Rahmen Teile eines Wikipedia-Artikels geparset wurden. Der Aufbau des Tests wurde in Abschnitt 7.4 erläutert. Der LaTeX-Code, den der implementierte Parser ausgibt, konnte fast 1:1 übernommen werden. Zum Teil wurde die Skalierung im QTree-Code etwas auf die Seitengröße angepasst, die Struktur der Bäume selbst wurde jedoch nicht modifiziert.

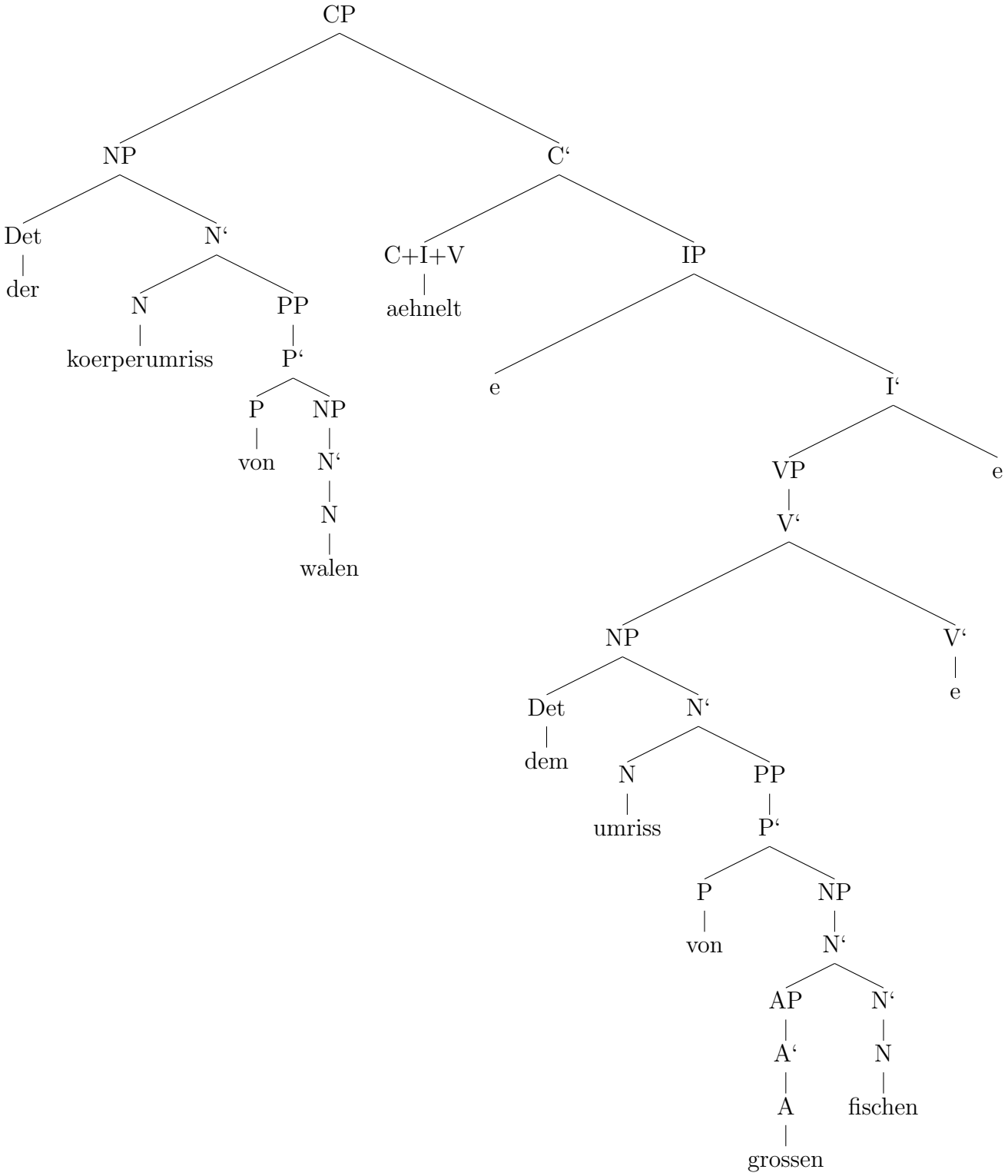
Satz 1

Der Körperumriss der Wale ähnelt dem von großen Fischen, was sich auf die Lebensweise und die besonderen Bedingungen des Lebensraums zurückführen lässt.

Problem $\Longrightarrow$

[*dem von großen Fischen*] kann nicht geparset werden. Der Satzteil [*was ... lässt*] ist zudem eine Aufzählung, und enthält einen Passiv, der nicht vom Parser unterstützt wird. [$_{NP}$ *Körperumriss der Wale*] drückt ein Besitzverhältnis aus, diese Struktur kann ebenfalls nicht erkannt werden.

Abwandlung $\Longleftrightarrow$ *Der Körperumriss von Walen ähnelt dem Umriss von großen Fischen.*



Satz 2

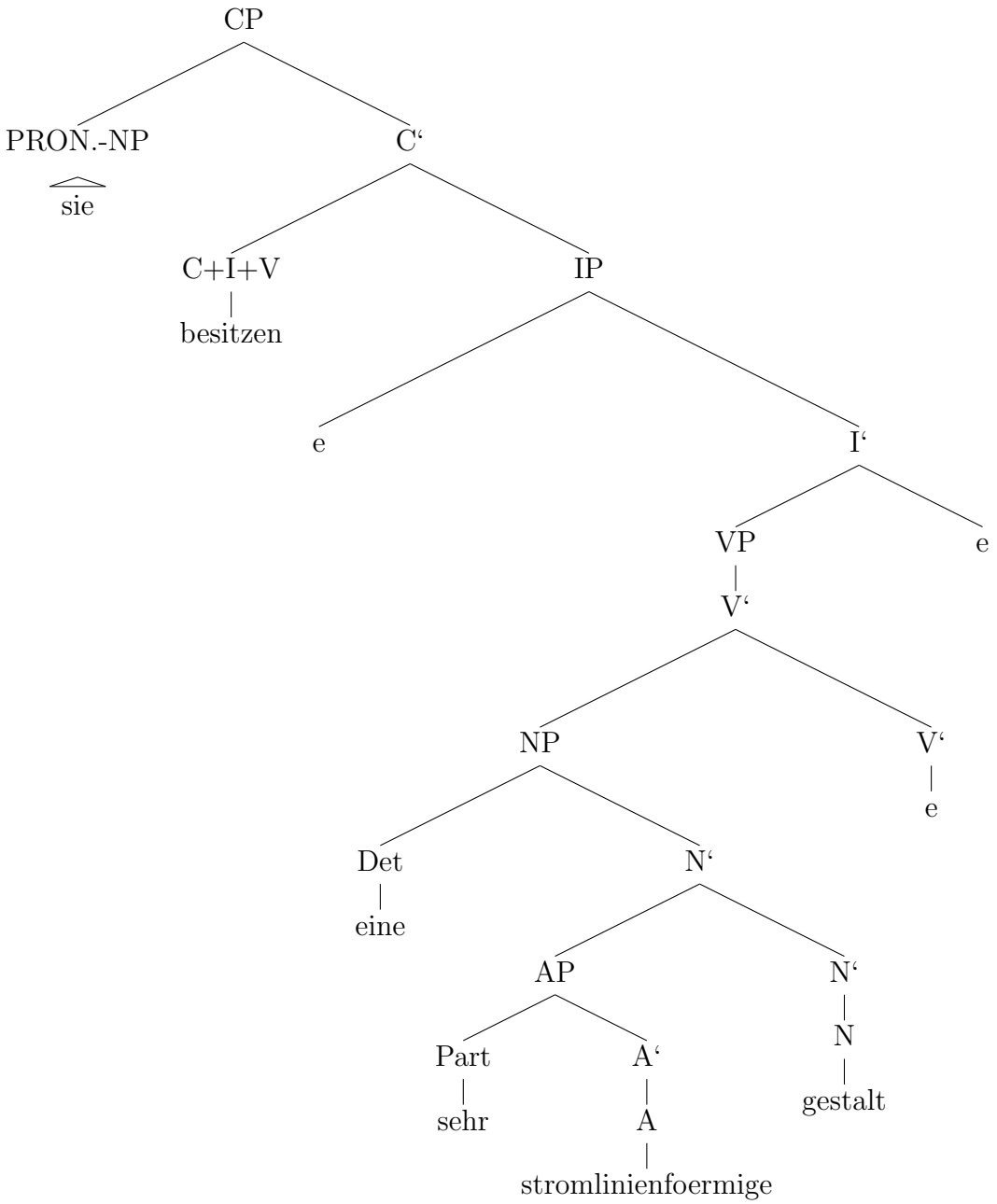
So besitzen sie eine stromlinienförmige Gestalt, und ihre Vorderextremitäten sind zu

Flossen umgestaltet.

Problem
$\Longrightarrow$

In SPEC-C[1] steht das Adverb *so*. Da dieses kein Objekt des Verbs ist, gehört seine Bewegung ins Vorfeld nicht zu den erkennbaren syntaktischen Operationen. Der ganze Satz enthält eine Aufzählung, die nicht unterstützt wird. Der [*und...*]-Satzteil enthält einen Passiv und lässt sich überhaupt nicht parsen.

Abwandlung
$\Longleftrightarrow$ *Sie besitzen eine sehr stromlinienförmige Gestalt.*



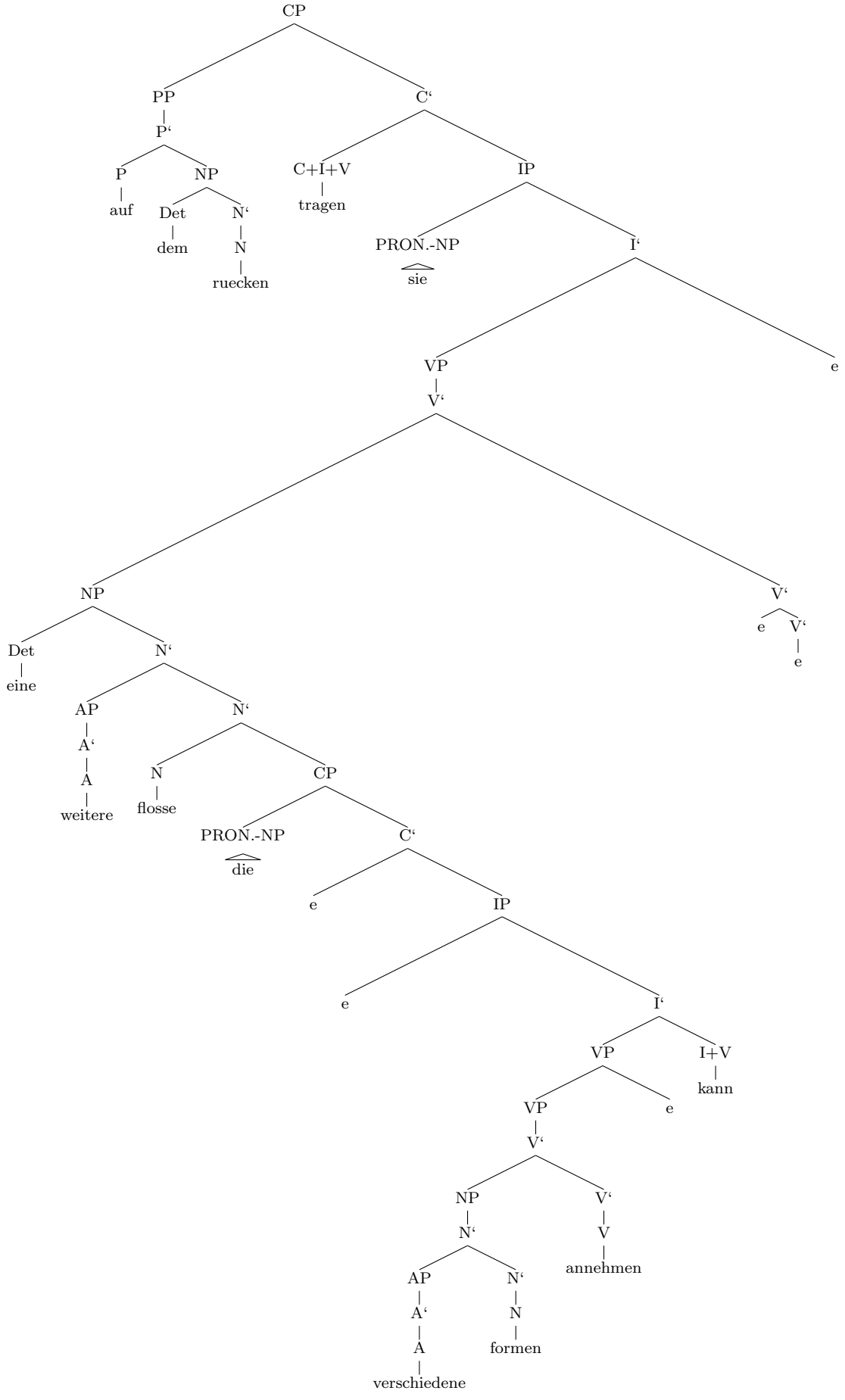
Satz 3

Auf dem Rücken tragen sie eine weitere Flosse, die als Finne bezeichnet wird und je nach Art verschiedene Formen annimmt.

Problem
$\Longrightarrow$

Der Relativsatz [*die als Finne . . . annimmt.*] enthält eine IP-Konjunktion und einen Passiv, die beide nicht erkannt werden können. Es war nicht klar, wie [*je nach*] syntaktisch dargestellt werden sollte.

Abwandlung $\Longleftrightarrow$ *Auf dem Rücken tragen sie eine weitere Flosse, die verschiedene Formen annehmen kann.*



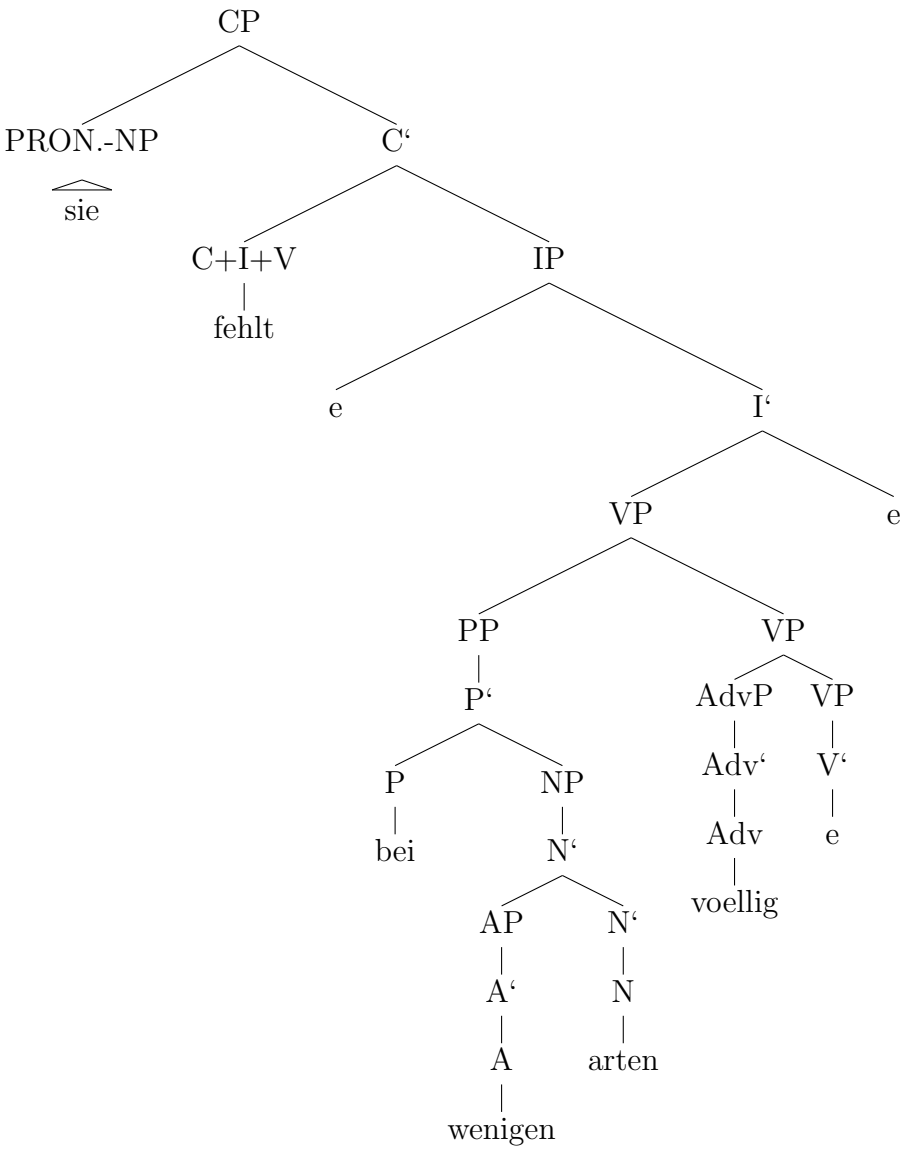
Satz 4

Bei wenigen Arten fehlt sie völlig.

Problem
$\Longrightarrow$

Ein freies PP-Adjunkt steht im Vorfeld.

Abwandlung
$\Longleftrightarrow$ *Sie fehlt bei wenigen Arten völlig.*



Satz 5

Sowohl die Flipper als auch die Finne dienen ausschließlich der Stabilisierung der Wale

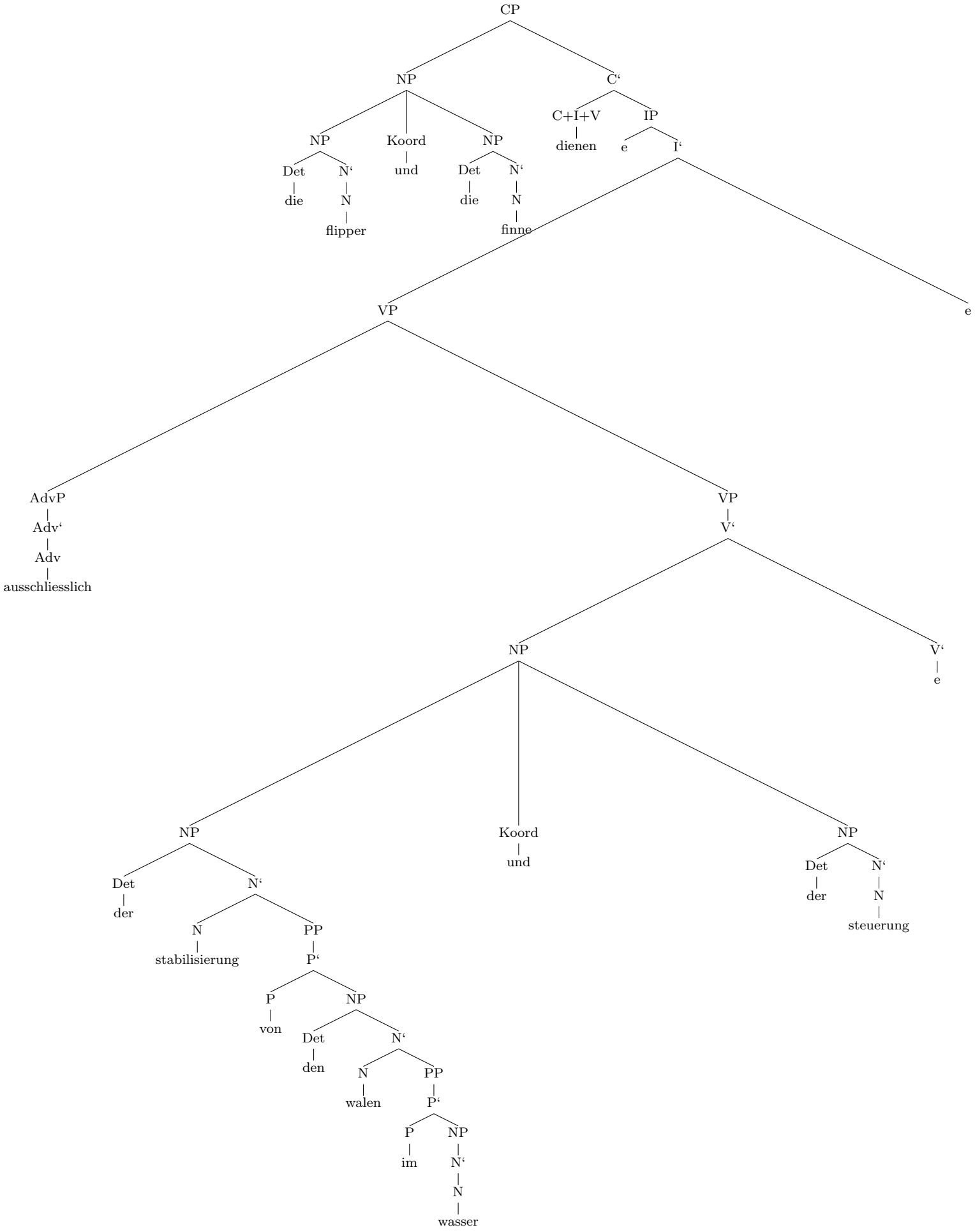
im Wasser und der Steuerung.

Problem
$\Longrightarrow$

[*sowohl ... als auch ...*] ist nicht zu parsen. Die Possessiv-NP [*Stabilisierung der Wale*] kann nicht geparset werden, wir können sie jedoch durch eine PP ersetzen.

Abwandlung
$\Longleftrightarrow$ *Die Flipper und die Finne dienen ausschließlich der Stabilisierung von den Walen im Wasser und der Steuerung.*

Der Parser findet 7 verschiedene Lösungen, hier die sinnvollste Zuordnung:


CP
NP
C‘
NP
Koord
NP
C+I+V
IP
Det
N‘
Det
N‘
dienen
e
I‘
die
und
die
N
N
flipper
finne
VP
e
AdvP
VP
Adv‘
V‘
Adv
ausschliesslich
NP
V‘
e
NP
Koord
NP
und
Det
N‘
Det
N‘
der
der
N
PP
N
steuerung
stabilisierung
P‘
P
NP
von
Det
N‘
den
N
PP
walen
P‘
P
NP
im
N‘
N
wasser

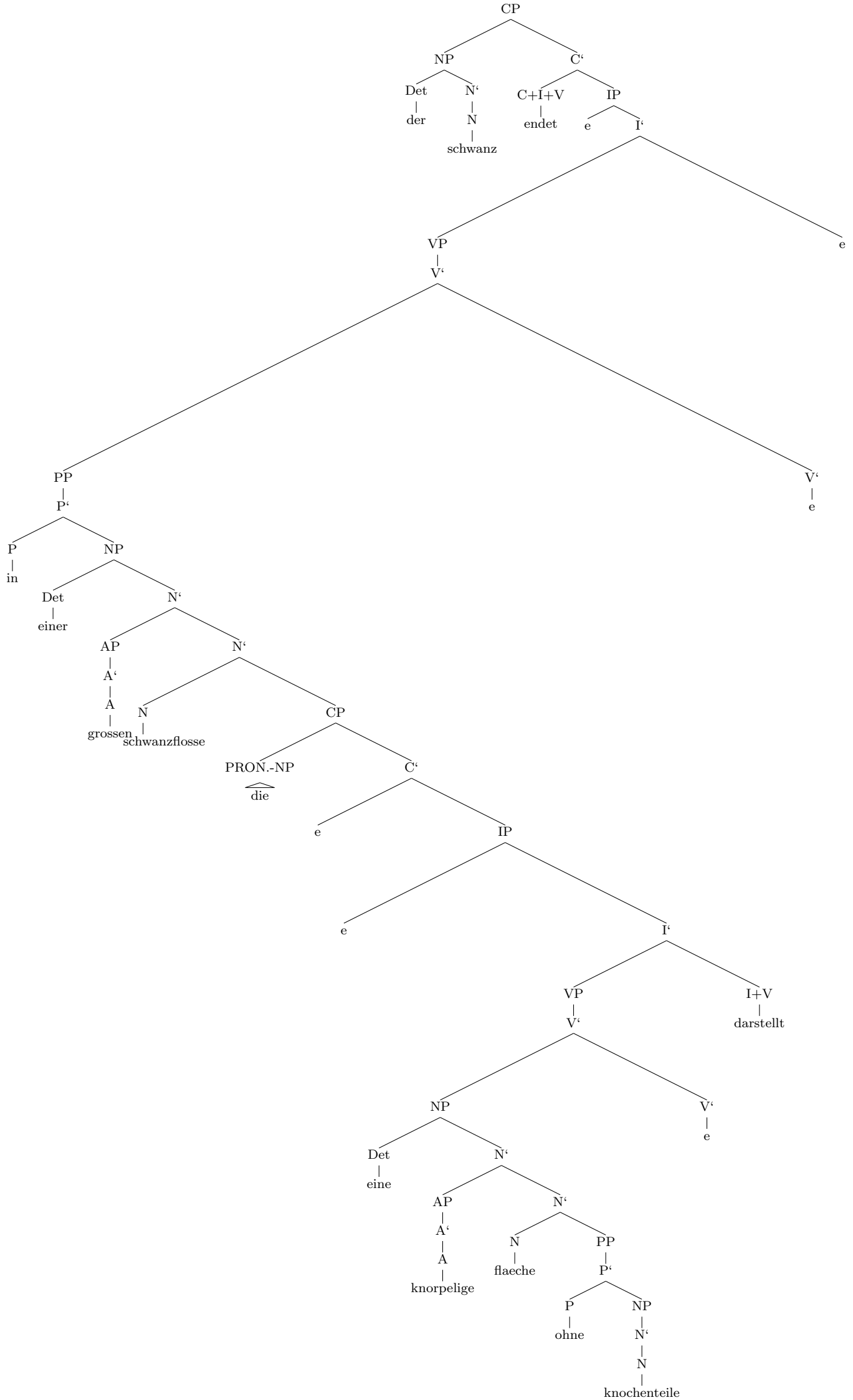
Satz 6

Der Schwanz endet in einer großen Schwanzflosse, die Fluke heißt und wie die Finne eine knorpelige Fläche ohne Knochenteile darstellt.

Problem
$\Longrightarrow$

Einordnung des Verbs *heisst* war nicht möglich. Die strukturellen Eigenschaften von [*wie die Finne*] waren nicht in Erfahrung zu bringen.

Abwandlung
$\Longleftrightarrow$ *Der Schwanz endet in einer großen Schwanzflosse, die eine knorpelige Fläche ohne Knochenteile darstellt.*


CP
NP
C‘
Det
N‘
C+I+V
IP
der
N
endet
e
I‘
schwanz
VP
e
V‘
PP
V‘
P‘
e
P
NP
in
Det
N‘
einer
AP
N‘
A‘
A
N
CP
grossen
schwanzflosse
PRON.-NP
C‘
die
e
IP
e
I‘
VP
I+V
V‘
darstellt
NP
V‘
e
Det
N‘
eine
AP
N‘
A‘
N
PP
A
flaeche
P‘
knorpelige
P
NP
ohne
N‘
N
knochenteile

Satz 7

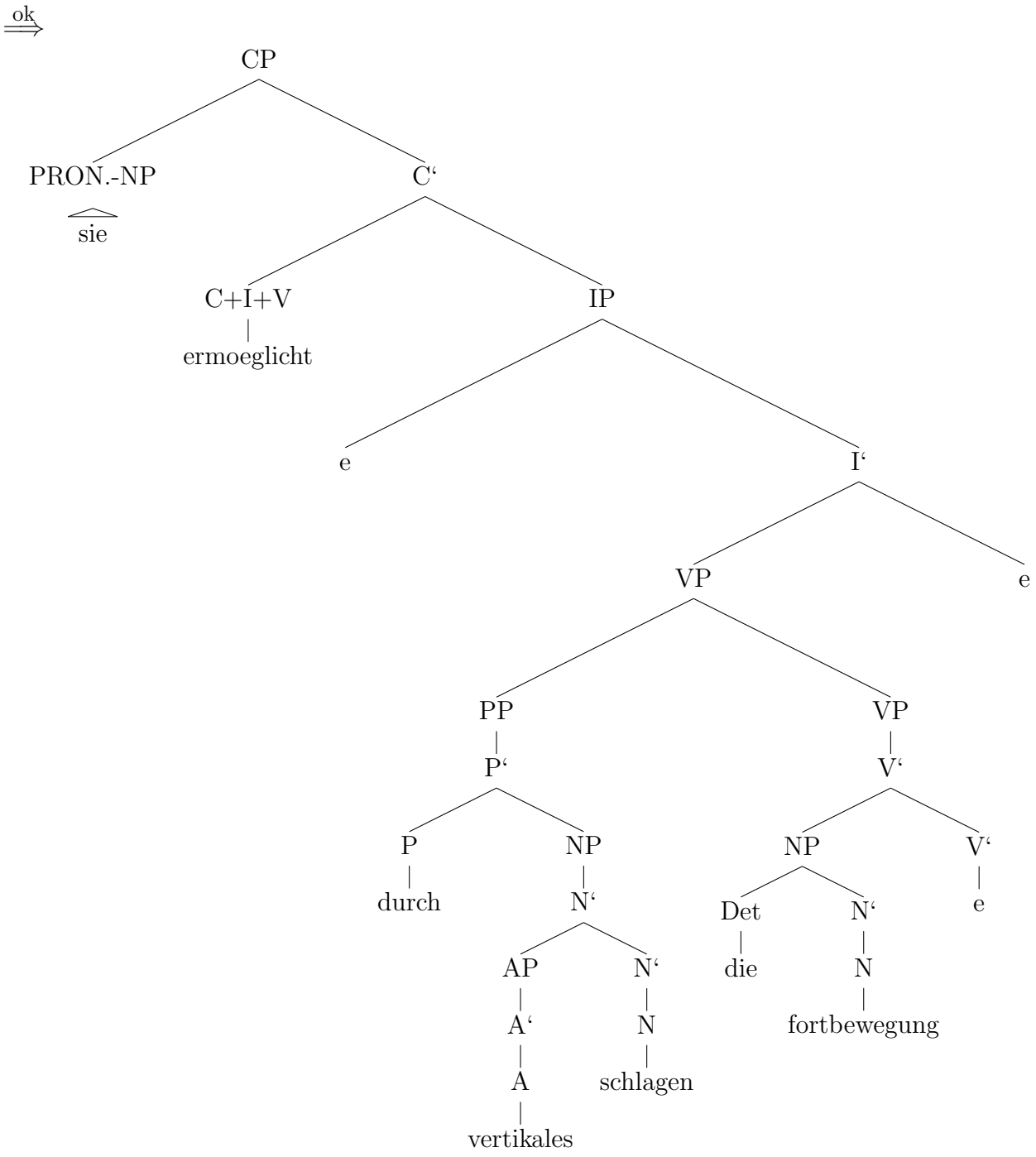
Die Fluke setzt waagerecht statt senkrecht am Körper an, ein von außen sehr gut erkennbares Unterscheidungsmerkmal zu den Fischen.

Problem
$\Longrightarrow$

Der erste Satzteil kann nicht geparset werden, weil er ein Verb mit Partikel enthält ([*setzt ... an*]). Der zweite Satzteil kann, stünde er für sich, auch nicht verarbeitet werden, weil man dann *ist* als Kopulaverb heranziehen müsste.

Satz 8

Sie ermöglicht durch vertikales Schlagen die Fortbewegung.



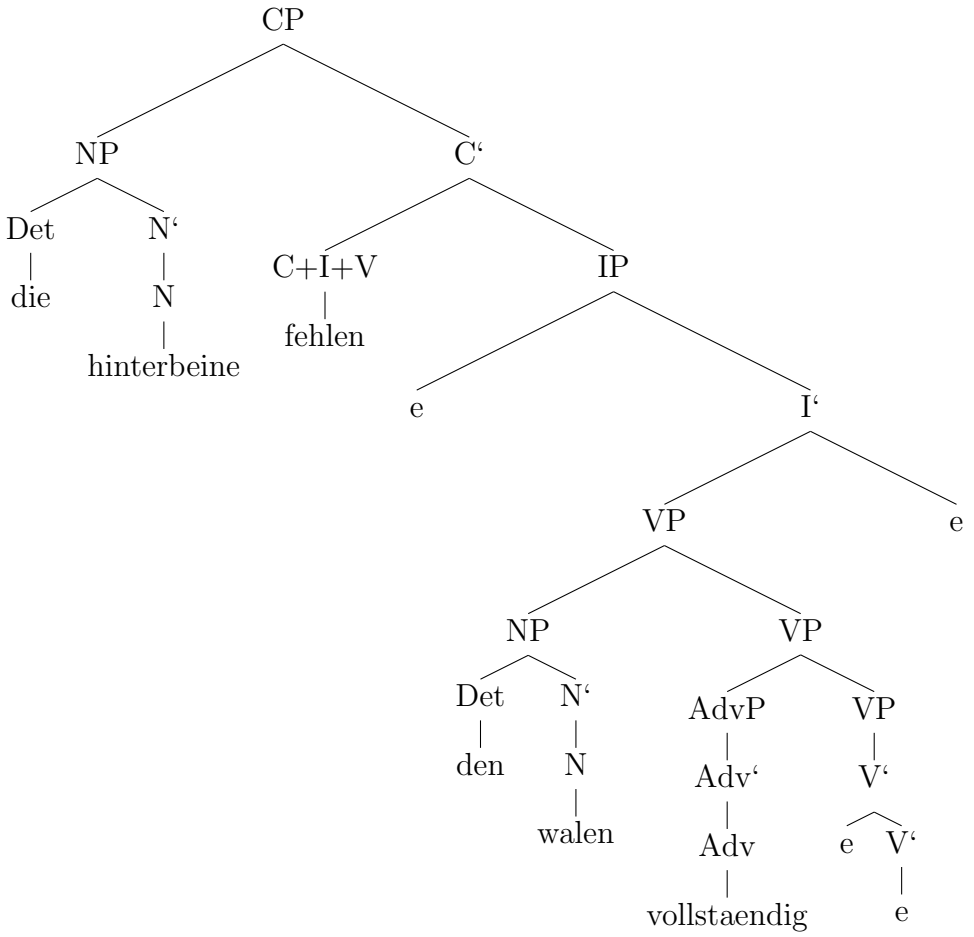
Satz 9

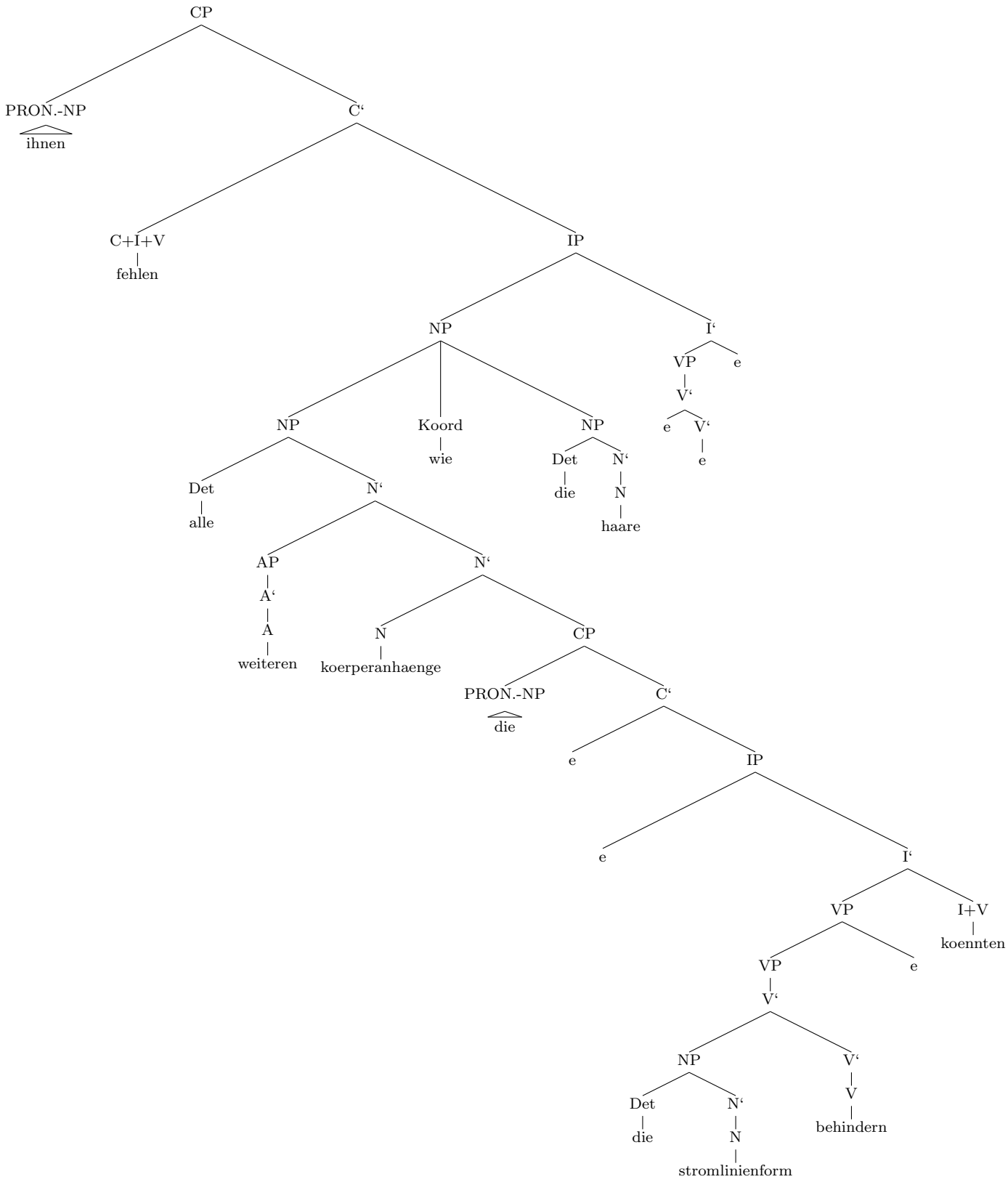
Die Hinterbeine fehlen den Walen vollständig, ebenso alle weiteren Körperanhänge, welche die Stromlinienform behindern könnten, wie die Ohren und auch die Haare.

Problem
$\Longrightarrow$

Die Satzstruktur lässt sich so nicht erfassen. *alle* wird als Artikelwort/Determinator klassifiziert. Das Wort *wie* ist streng genommen eine Subjunktion, wird aber im Lexikon als koordinierende Konjunktion geführt. Der Satzteil [*Körperanhänge . . . wie . . .* [*Ohren und ... Haare*]] enthält dann zwei ineinander "verschachtelte" Konjunktionen. Aus den in Abschnitt 5.4.4 aufgeführten Gründen werden solche Konjunktionsstrukturen nicht unterstützt. Das Adverb *ebenso* wird an die IP adjungiert, vor dem Subjekt [*alle weiteren Körperanhänge ...*]. Dies kann jedoch nicht erkannt werden, weil es sich beim Adverb um ein freies Adjunkt handelt, deren Transformation vom Programm nicht unterstützt wird. Vergleiche dagegen die Adjunktion von pronomialen Objekten an die IP (S. 50).

Abwandlung
$\Longleftrightarrow$ *Die Hinterbeine fehlen den Walen vollständig. Ihnen fehlen alle weiteren Körperanhänge, die die Stromlinienform behindern könnten, wie die Haare.*




CP
PRON.-NP
ihnen
C‘
C+I+V
fehlen
IP
NP
I‘
VP
e
V‘
NP
Koord
NP
e
V‘
wie
Det
N‘
e
Det
N‘
die
N
haare
alle
AP
N‘
A‘
A
N
CP
weiteren
koerperanhaenge
PRON.-NP
C‘
die
e
IP
e
I‘
VP
I+V
koennten
VP
e
V‘
NP
V‘
Det
N‘
V
die
N
behindern
stromlinienform

Satz 10

Die männlichen Genitalien und die Brustdrüsen der Weibchen sind in den Körper versenkt.

$\overset{\text{Problem}}{\Longrightarrow}$

Der Satz enthält einen Passiv, so dass er nicht geparset werden kann.

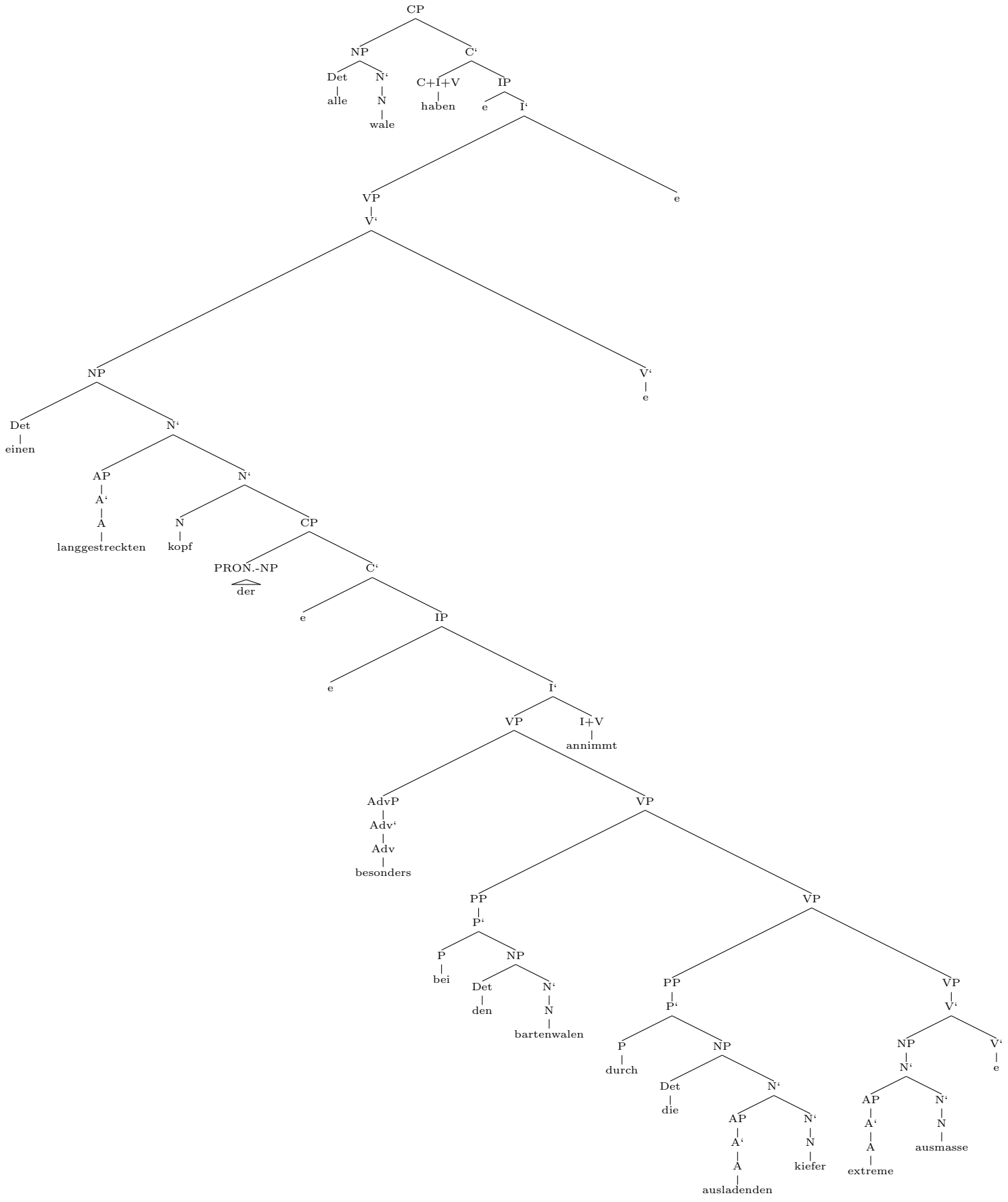
Satz 11

Alle Wale haben einen langgestreckten Kopf, der besonders bei den Bartenwalen durch die weit ausladenden Kiefer extreme Ausmaße annimmt.

$\overset{\text{Problem}}{\Longrightarrow}$

haben wird hier als transitives Vollverb gebraucht. [*weit ausladenden*] sind als zwei Adjektive zu betrachten. Ließe man *ausladenden* weg, würde eine syntaktische Struktur erkannt ([*die weit kiefer*]) werden, die ungrammatisch ist. Man könnte folgende Abwandlung nehmen:

$\overset{\text{Abwandlung}}{\Longleftrightarrow}$ *Alle Wale haben einen langgestreckten Kopf, der besonders bei den Bartenwalen durch die ausladenden Kiefer extreme Ausmaße annimmt.*


CP
NP
C‘
Det
N‘
C+I+V
IP
alle
N
haben
e
I‘
wale
VP
e
V‘
NP
V‘
Det
N‘
e
AP
N‘
A‘
N
CP
A
kopf
PRON.-NP
C‘
langgestreckten
der
e
IP
e
I‘
VP
I+V
annimmt
AdvP
VP
Adv‘
Adv
besonders
PP
VP
P‘
P
NP
bei
Det
N‘
den
N
bartenwalen
PP
VP
P‘
V‘
P
NP
NP
V‘
durch
Det
N‘
N‘
e
die
AP
N‘
AP
N‘
A‘
N
A‘
N
kiefer
A
ausmasse
ausladenden
extreme

Literaturverzeichnis

[LEUN04] Helen Leuninger: *Grammatische Strukturen und Kognitive Prozesse*. Gunter Narr Verlag Tübingen, 2. Auflage, 2004

[STEI05] Markus Steinbach: *Deskriptive Grammatik*. Reader, Universität Mainz, 2005
www.staff.uni-mainz.de/steinbac/Lehre/thempsreader.pdf
abgerufen am 12.07.2009

[BDS06] Patrick Brandt, Rolf-Albert Dietrich, Georg Schön: *Sprachwissenschaft*. Böhlau Verlag Köln, 2. Auflage, 2006

[DUDG06] *Duden - Die Grammatik*. Bibliographisches Institut & F.A. Brockhaus AG, Mannheim, 7. Auflage, 2006

[DUDR06] *Duden - Die Rechtschreibung*. Bibliographisches Institut & F.A. Brockhaus AG, Mannheim, 24. Auflage, 2006

[SSSK06] Prof. Dr. Manfred Schmidt-Schauss: *Einführung in die Methoden der künstlichen Intelligenz*. Skript, Universität Frankfurt, 2006

[GREW89] Grewendorf, Hamm, Sternefeld: *Sprachliches Wissen*. Suhrkamp Verlag Frankfurt am Main, 3. Auflage, 1989

[CLR94] Cormen, Leiserson, Rivest: *Introduction to Algorithms*. The MIT Press, Cambridge (MA), 14. Auflage, 1994

[MANL08] *SWI-Prolog 5.6 Reference Manual*. 2008
http://www.swi-prolog.org/download/stable

[BRAT90] Ivan Bratko: *Prolog - Programming For Artificial Intelligence*. Addison-Wesley Publishing Company, Reading (MA), 2. Auflage, 1990

[PYTH03] Michael Weigend: *Python Ge-Packt*. mitp-Verlag Bonn, 1. Auflage, 2003

[CHEM86] Horst Winter: *Benennungsmotive für chemische Stoffnamen*. Special Language/Fachsprache 8, S. 155-162, 1986

[LUTZ04] Ulrich Lutz: *Studien zu Extraktion und Projektion im Deutschen*, S.63. Universität Tübingen, 2004
http://tobias-lib.ub.uni-tuebingen.de/volltexte/2004/1161/
abgerufen am 21.06.2009

[MUEL95] Gereon Müller: *A-Bar Syntax - A Study in Movement Types*, S. 26. Mouton de Gruyter, 1995, *via Google Books*

[UUNL01] Johan Kerstens, Eddy Ruys, Joost Zwarts: *Lexicon of Linguistics*. Utrecht Institute of Linguistics OTS, Utrecht University, 2001 http://www2.let.uu.nl/uil-ots/lexicon/zoek.pl?lemma=Minimality abgerufen am 03.06.2009

[DOUG94] Ray C. Dougherty: *Natural Language Computing - An English Generative Grammar in Prolog*. Lawrence Erlbaum Associates, Philadelphia (PA), 1994

[KARW08] Leschek Karworth: *Erkennung deutscher Sprache mithilfe von Definite Clause Grammars in Prolog*. Bachelorarbeit, Universität Frankfurt, 2008

[BLAC08] William B. Blacoe: *Controlled Partition Grammars*. Bachelorarbeit, Universität Frankfurt, 2008

[TRIS00] Susanne Trissler: *Syntaktische Bedingungen für w-Merkmale: Zur Bildung interrogativer w-Phrasen im Deutschen*. überarbeitete Dissertation, Universtität Tübingen, 2000 http://w210.ub.uni-tuebingen.de/dbt/volltexte/2001/216/pdf/diss_trissler.pdf abgerufen am 20.07.2009

[WARR99] David S. Warren: *Programming in Tabled Prolog*. Department of Computer Science, Stony Brook University, New York, 1999 http://www.cs.sunysb.edu/∼warren/xsbbook/node12.html abgerufen am 12.07.2009

[LENE01] Prof. Dr. Jürgen Lenerz: *Einführungsseminar Sprachwissenschaft*, S. 4. Handout, Universität Köln, 2001 http://www.uni-koeln.de/phil-fak/idsl/dozenten/lenerz/ss01/einf_sem/6sitzung.pdf abgerufen am 12.07.2009

[CHOM95] Noam Chomsky: *The Minimalist Program*. The MIT Press, 1995, *via Google Books*