Parsing und Generierung in uniformen Architekturen

Magisterarbeit

zur Erlangung des Grades 'Magister Artium'
der Philosphischen Fakultät
der Heinrich-Heine-Universität Düsseldorf

vorgelegt von:	Sebastian Varges Remscheider Str.3 40215 Düsseldorf 0211–330425
Prüfer im Hauptfach:	Prof. Dr. James Kilbury Seminar für Allgemeine Sprachwissenschaft Heinrich-Heine-Universität Düsseldorf

April 1997
Revision: 4.6.1997

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Ziel der Arbeit

Die vorliegende Arbeit beschäftigt sich mit einem bestimmten Aspekt der Verarbeitung natürlicher Sprache *(Natural Language Processing, NLP)*: die Modellierung von Parsing und Generierung in uniformen Architekturen. In diesem Kapitel soll ein Überblick über das Thema und seine Behandlung im Rahmen dieser Arbeit gegeben werden.

In der Computerlinguistik wird zwischen der Grammatik, die eine natürliche Sprache – oder jedenfalls Teile davon – beschreibt, und den Mechanismen, die zu ihrer Verarbeitung dienen, unterschieden. Der Grundgedanke ist, daß die Grammatik sprachliches Wissen repräsentiert, unabhängig davon, ob es zur Analyse oder zur Erzeugung einer gegebenen sprachlichen Äußerung eingesetzt wird. Eine solche Grammatik wird als *deklarativ* bezeichnet: die Reihenfolge, in der das linguistische Wissen verwendet wird, ändert nichts an den Aussagen über die sprachliche Kompetenz, die in der Grammatik repräsentiert werden.

Im Gegensatz dazu sind die Mechanismen oder Algorithmen, die dieses Wissen verwenden, reihenfolgeabhängig *(prozedural)*. Sie ziehen z.B. bestimmte Syntaxregeln oder Lexikoneinträge vor anderen in betracht und kommen ihrem Ziel in unterschiedlich geschickter Weise näher. Die Analyse von sprachlichen Äußerungen wird als *Parsing* im weiteren Sinn bezeichnet, ihre Erzeugung wird *Generierung* genannt. Es wird hier in beiden Fällen von einzelnen Sätzen in verschrifteter Form ausgegangen, d.h. daß z.B. diskursive Aspekte nicht berücksichtigt werden können.

Historisch gesehen ist die strikte Trennung von Repräsentation und Verarbeitung mit der Einführung von Unifikationsgrammatiken zu Beginn der 80er Jahre aufgekommen (Kay 1979). Die vorher vor allem verwendeten ATNs *(Augmented Transition Networks)* vermischen prozedurale Operationen wie die Bearbeitung von Registern während der Traversierung des Netzwerks mit der repräsentierten linguistischen Information: "An ATN is a description of a *process* for recognizing a language, rather than a description of the language itself" (Pereira und Warren 1980, S.120). Eine Motivation für die Verwendung von

ATNs war die generative Grammatik, die selbst nicht als Verarbeitungsmodell geeignet ist. Die Verwendung von Unifikationsgrammatiken ermöglicht dagegen eine Verbindung von theoretischer Linguistik mit der Verarbeitung der Beschreibungen.

Wenn Grammatiken als deklarativ angesehen werden, müßten sie *reversibel* sein in dem Sinne, daß sie sowohl für Parsing als auch für Generierung gleichermaßen geeignet sind. In der Computerlinguistik wurde Parsing traditionellerweise stärker berücksichtigt als Generierung. Ein möglicher Grund könnte darin liegen, daß zunächst vor allem der Wunsch bestand, komplexe Maschinen möglichst direkt bedienen zu können, ohne eine Programmiersprache erlernen zu müssen. Dagegen können Menschen sich auch auf einen sehr primitiven Output von Programmen einstellen, so daß die Generierung nur den absoluten Minimalanforderungen gerecht werden mußte. Viele Grammatiken wurden aus diesem Grund nur zum Parsing verwendet und es hat sich gezeigt, daß sie nicht in jedem Fall auch für die Generierung geeignet sind. Eine Verwendung in beiden Richtungen der Sprachverarbeitung stellt sicher, daß eine Grammatik reversibel ist und als deklarativ bezeichnet werden kann.

Diese Arbeit bezieht sich auf ein Kerngebiet der Computerlinguistik, betrachtet dieses aber unter einem Aspekt, der zumindest bisher nicht so große Beachtung gefunden hat: Parsing und Generierung werden *nicht* mit *verschiedenen* Algorithmen ausgeführt, sondern in *einheitlicher Art und Weise*. Neben der Eleganz, die ein solcher Ansatz aufweist, scheint es zumindest intuitiv plausibel zu sein, daß auch die menschliche Sprachverarbeitung nicht völlig verschiedene Mechanismen für die beiden Aufgaben einsetzt. Während der Mensch spricht, kann er sich selbst sprechen hören und so seine Äußerung gegebenfalls noch korrigieren.

Bisher gibt es nur wenige Vorschläge, die das Konzept einer uniformen Architektur verwirklichen. Diese Arbeit untersucht deshalb auch andere Ansätze, die sich auf reversible Grammatiken beziehen, in bezug auf ihre mögliche Anwendung in uniformen Architekturen.

1.1.1 Überblick

Die folgenden Abschnitte dieses Kapitels geben eine Einführung in das Thema. In Abschnitt 1.2 werden Gründe genannt, die für reversible Grammatiken im allgemeinen sprechen. Abschnitt 1.3 ordnet uniforme Architekturen in ein Klassifikationsschema für Parsing- und Generierungskonzeptionen ein. Darauf wird in Abschnitt 1.4 eine besondere Anwendungsmöglichkeit von uniformen Architekturen erläutert.

Kapitel 2 stellt einige formale Grundlagen und Definitionen, die für das Thema von Bedeutung sind, vor. Grammatikformalismen dienen der Repräsentation sprachlichen Wissens in formal klar definierter Weise. In Abschnitt 2.1 wird ein bestimmter Typ von Grammatikformalismen, der für reversible Grammatiken besonders geeignet erscheint, erläutert. Weiterhin werden grundlegende Operationen von Grammatikformalismen wie die Unifikation von einfachen und getypten Merkmalsstrukturen definiert. In Abschnitt 2.3 wird ein

Überblick über die bekanntesten Analyse-Strategien für Parsing und Generierung gegeben. Dabei wird besonders auf Terminations-Probleme und Linksrekursion eingegangen, da dies für die grundlegenden Weichenstellungen der im folgenden vorgestellten Vorschläge von besonderer Bedeutung ist. Abschnitt 2.2.2 behandelt definitorische und formale Fragen, die besonders in bezug auf reversible Grammatiken auftreten.

In Kapitel 3 und 4 werden einige Vorschläge zu uniformen Architekturen dargestellt und erörtert. Nach einem Überblick über die verschiedenen Dimensionen, in denen sich die Vorschläge unterscheiden (Abschnitt 3.1), werden in Abschnitt 3.2 vom semantischen bzw. syntaktischen Kopf gesteuerte Algorithmen für Parsing und Generierung erläutert. Kapitel 4 stellt daraufhin auf verallgemeinerter Earley-Deduktion beruhende Architekturen vor, die die bis dahin eingeführten Analyse-Strategien verwenden. Nach einem Überblick über Chart-Parsing und Definite Klauseln werden drei Ansätze zur top-down-Earley-Deduktion (Shieber (1988), Gerdemann (1991) und Neumann (1994)) sowie ein Ansatz zur bottom-up-Earley-Deduktion (Erbach (1995a)) behandelt. Am Ende von Kapitel 4 wird eine Zusammenfassung und Einordnung der Vorschläge gegeben.

Die Darstellung der Algorithmen in Kapitel 3 und 4 abstrahiert von der konkreten Implementierung in einer Programmiersprache. In Kapitel 5 werden Aspekte der Implementierung eines vom semantischen Kopf gesteuerten Algorithmus und eines Verfahrens, das auf Earley-Deduktion beruht, in der Logik-basierten Programmiersprache Prolog gegeben (Abschnitt 5.3 und 5.4).

Kapitel 6 gibt einen Überblick über die Ergebnisse dieser Arbeit. Dazu werden offene Fragen angesprochen und ein Ausblick auf weitere Themen gegeben. Insbesondere wird die Möglichkeit eines uniformen left-corner-Algorithmus angesprochen und einige (vorläufige) Ideen hierzu präsentiert. Im Anhang wird eine Beispielgrammatik sowie der Quell-Code der Implementierungen einschließlich einiger ausgewählter Analyseergebnisse dargestellt.

Verwendete Formalismen und Programmiersprachen Die Implementierung der Prolog-Programme erfolgte in SICStus 3.2 für UNIX-Rechner. Da keine speziellen Konstrukte dieser Prolog-Version verwendet werden, liegt Kompatibilität zu älteren SICStus-Versionen und zu QUINTUS-Prolog vor. Da mit einer Ausnahme (dem Library-Prädikat `subsumes_chk/3`; s. Abschnitt 5.3.1) nur Standard-built-in-Prädikate verwendet werden, ist die Implementierung leicht auf andere Prolog-Dialekte portierbar.

Die vorliegende Arbeit ist als komprimierte Postscript-Datei im Internet verfügbar (`http://web.phil-fak.uni-duesseldorf.de/varges/master.html`). Unter dieser Adresse sind die Prolog-Dateien auch einzeln einsehbar. Zudem wird eine Zusammenfassung der Arbeit auf Englisch gegeben.

1.2 Motivation für reversible Grammatiken

Für die Verwendung von reversiblen Grammatiken sprechen linguistische und sprachtechnologische Überlegungen, die in diesem Abschnitt genauer erläutert werden sollen (vgl. van Noord (1993), Neumann und van Noord (1992) und Neumann (1994)).

Linguistische Gründe Wenn es ein Ziel der Sprachwissenschaft ist, Aussagen über natürliche Sprachen im allgemeinen zu machen, muß von der Verwendung dieser Aussagen für bestimmte Aufgaben, z.B. Sätze zu äußern oder zu verstehen, abstrahiert werden. Das Wissen über die Eigenschaften einer Sprache darf nicht an einen bestimmten Verarbeitungsprozeß gekoppelt sein. Es ist aus linguistischer Perspektive nicht akzeptabel, auf bestimmte Verarbeitungsmechanismen und ihre besonderen Erfordernisse Rücksicht nehmen zu müssen.

Reversible Grammatiken müssen die Beziehung zwischen Zeichen und Bezeichnetem genau definieren. Sie erfordern gewissermaßen ein ausgeglichenes Verhältnis zwischen Form und Inhalt, wodurch die von de Saussure eingeführte Unterscheidung zwischen *Signifiant* und *Signifié* wieder aufgenommen wird (wie dies z.B. in dem Grammatikformalismus HPSG (Pollard und Sag 1994) geschieht).

Es ist möglich, die syntaktische Korrektheit von Phrasen zu testen, ohne eine semantische Repräsentation aufzubauen (ein sog. *Erkenner* im Gegensatz zum Parser, der mit fortschreitender Analyse des Satzes eine – wenn auch nicht notwendigerweise semantische – Struktur aufbaut). Eine für diesen Zweck geeignete Grammatik kann völlig auf semantische Angaben verzichten, d.h. allein der Formteil wird in der Grammatik repräsentiert. Wenn dagegen die gleiche Grammatik auch für die Generierung verwendet werden soll, ist eine semantische Repräsentation zwingend notwendig, allein schon aus dem Grund, daß sie als Ausgangspunkt der Generierung dient.

Sprachtechnologische Gründe Die Verwendung einer reversiblen Grammatik in natürlich-sprachlichen Systemen hat deutliche Vorteile gegenüber dem Einsatz zweier verschiedener Grammatiken für Parsing und Generierung. Bei Entwicklung und Einsatz eines solchen Systems ist es einfacher, nur eine Grammatik zu warten und zu erweitern als für jede Änderung in der einen Grammatik eine korrespondierende Modifizierung der anderen in Erwägung ziehen zu müssen. Auf diese Weise werden Redundanzen zwischen den beiden beteiligten Grammatiken vermieden. Diese sind sowohl theoretisch unbefriedigend wie auch vom praktischen Standpunkt aus gesehen kostspielig in Hinblick auf den benötigten Speicherplatz.

Aus methodologischer Sicht hilft der Einsatz einer Grammatik in einem reversiblen System, Über- bzw. Untergenerierung zu vermeiden. Grammatiken, die ausschließlich für das Parsing entwickelt worden sind, generieren häufig Sätze, die ungrammatisch sind (vgl. van Noord (1993, S.11)). Umgekehrt erzeugen Grammatiken, die zur Generierung entwickelt wurden, oft nur Sätze in einer bestimmten kanonischen Standardform. Wird diese Grammatik für das Parsing eingesetzt, kann sie nur einen sehr kleinen Ausschnitt der wohlgeformten Sätze analysieren.

Reversible Grammatiken sind *konsistent* in bezug auf Parsing und Generierung: jeder Satz, den ein natürlich-sprachliches System generiert, muß auch analysiert werden können (1993, S.13). Für den Benutzer eines solches Systems bedeutet dies, daß er sich an den Sätzen orientieren kann, die das System selbst produziert hat. Wenn er feststellt, daß diese Sätze nicht analysiert werden können, wird er Probleme haben, die sprachlichen Fähigkeiten des Systems überhaupt angemessen einschätzen zu können (etwa in dem Sinne: *'kann ich den Äußerungen des Systems trauen, wenn es seine eigenen Sätze nicht versteht?'*). Parsing und Generierung sind in diesem Fall inkonsistent. Eine reversible Grammatik erlaubt es, die Konsistenz der Parsing- und Generierungskomponente gewissermaßen automatisch sicherzustellen, weil sie das linguistische Wissen für beide Prozesse bereitstellt. Die Forderung nach Konsistenz spiegelt die Tatsache wieder, daß der Mensch auch Sätze verstehen kann, die er selbst nicht hätte erzeugen können. Andererseits wird aber erwartet, daß ein Mensch jeden Satz, den er äußert, auch verstanden hat. Dies scheint mir der Grund dafür zu sein, daß die Forderung nach Konsistenz nicht auch andersherum formuliert wird: daß jeder Satz, der verstanden wird, auch generiert werden kann. Konsistenz liegt also vor, wenn die Menge der generierbaren Sätze eine Teilmenge der analysierbaren Sätze ist.

Ein weiterer Vorteil reversibler Grammatiken zeigt sich im Hinblick auf maschinelles Lernen und den Umgang mit unbekannten Wörtern. Wird bei der Analyse eines Satzes ein dem System bisher unbekanntes Wort festgestellt und hat das System die Fähigkeit, dieses Wort unter Verwendung der bis dahin vorliegenden Informationen in einen (vorläufigen) Lexikoneintrag zu verwandeln, dann steht der neue Eintrag automatisch auch für die Generierung zur Verfügung. Es ist z.B. vorstellbar, daß das System den vorläufigen Lexikoneintrag in der Generierung verwendet, um die Hypothesen über das Wort aktiv zu überprüfen. Allgemein gesehen erlaubt eine reversible Grammatik, das in einer Verarbeitungsrichtung erworbene Wissen für die andere zu verwenden. Obwohl es sich nicht um eine direkte Parallele handelt, könnte der umgekehrte Fall darin gesehen werden, daß schon generierte Phrasen bei der Analyse sofort zur Verfügung stehen.

Ein anderes Anwendungsbeispiel ist die Anpassung an den Sprachgebrauch eines Gesprächspartners (bzw. allgemein an verschiedene Varietäten): wählt dieser beispielsweise ein bestimmtes von mehreren synonymen Wörtern in seinen Äußerungen, so kann das System dieses ebenfalls benutzen. Die Analyse der Sätze des Gesprächspartners dient so als Entscheidungshilfe bei der Auswahl zwischen den alternativen Wortformen (vgl. den Vorschlag zur inkrementellen Annäherung an die sprachliche Varietät eines Benutzers in Kilbury (1986)).

Darüberhinaus ist die Verwendung reversibler Grammatiken im Bereich der maschinellen Übersetzung *(Machine Translation, MT)* naheliegend, wenn jeweils nur eine Grammatik für jede der beteiligten Sprachen in beiden Übersetzungsrichtungen benutzt werden soll.

Ein meines Wissens in der Literatur nicht beachteter Punkt ist die mögliche Vermischung von Parsing und Generierung: wenn z.B. nur Teile eines Satzes gehört werden, andererseits aber kontextuelle Informationen vorliegen, die die möglichen Bedeutungen beschränken, kann die Aufgabe, die sich dem Hörer bzw. dem NLP-System stellt, nicht mehr eindeutig einer der beiden Verarbeitungsrichtungen zugeordnet werden. In diesem Fall ist in einem System, das zwei verschiedene Grammatiken benutzt, nicht mehr klar zu entscheiden, welche verwendet werden soll.

In diesen Argumenten spielten auch psycholinguistische Überlegungen eine Rolle, ohne eine zu enge Parallele zwischen menschlicher und maschineller Sprachverarbeitung ziehen zu wollen. Auf einer so allgemeinen Ebene scheint mir dies jedoch (noch) nicht gefährlich zu sein – es soll hier nicht unterstellt werden, daß mentale Prozesse in identischer Weise ablaufen.

1.3 Ein Klassifikationsschema für NLP-Systeme

NLP-Systeme lassen sich in vier verschiedene Klassen einteilen, je nachdem, ob eine einzige Grammatik für Parsing und Generierung verwendet wird oder nicht und ob diese Prozesse uniform oder unterschiedlich ablaufen (Neumann 1994, S.25ff). In allen Fällen wird davon ausgegangen, daß der Grammatikschreiber nur eine einzige Grammatik spezifiziert.[1]

Reversible Architekturen		*Grammatik* zwei	*Grammatik* eine einzige
Prozessor	zwei	**Typ A**	**Typ B**
	ein einziger	**Typ C**	**Typ D**

Abbildung 1.1: Klassifikationsschema für reversible Systeme

Typ A Die Grammatik kann zur Compile-Zeit in zwei spezialisierte Grammatiken verwandelt werden, die eine für einen Parsing-Algorithmus, die andere für einen sich davon unterscheidenden Generierungs-Algorithmus. Der Vorteil einer solchen Architektur ist die Effizienz zur Laufzeit, weil sowohl die Grammatiken wie auch die Prozesse optimal auf ihre Aufgabe zugeschnitten werden können. Der Nachteil ist, daß die beiden erzeugten Grammatiken ein hohes Maß an Redundanz aufweisen und daß bei der Grammatikentwicklung jedes Mal neu kompiliert werden muß, bevor die Grammatik getestet werden kann.

Typ B Es wird dieselbe Grammatik für Parsing und Generierung benutzt, wobei die Prozesse unterschiedlich sind. Dadurch entfallen die Nachteile von Typ A. In dem Fall, daß der Parser oder der Generator *unvollständig* ist, kann nicht mehr garantiert werden,

[1] Die Fälle, in denen die Grammatik in zwei verschiedene kompiliert wird, schließen die Möglichkeit ein, daß der Grammatikschreiber zwei Grammatiken von Hand geschrieben hat. Die Probleme der Feststellung der Äquivalenz beider Grammatiken treten dann sogar noch verschärft auf.

daß beide Prozessoren alle Sätze bzw. semantischen Repräsentationen verarbeiten können, die die Grammatik definiert.

Wenn der Generator Sätze erzeugt, die der Parser nicht analysieren kann, treten Inkonsistenzen auf.

Typ C Die Ausgangs-Grammatik wird in zwei spezialisierte Grammatiken kompiliert, die von einem uniformen Mechanismus verwendet werden. Der Sinn einer solchen Architektur liegt darin, daß der Grammatik weitere Information hinzugefügt wird, die den Prozessor effizienter steuert. Hier stellt sich wieder das Problem der Redundanz linguistischer Information und der Umständlichkeit der Grammatikentwicklung.

Typ D Schließlich kann eine einzige Grammatik mit einem uniformen Prozessor verwendet werden. Dieser Ansatz ist der konsequenteste der vier möglichen Kombinationen. Er hat wenigstens zwei Vorteile:

– Es können keine Inkonsistenzen auftreten, weil es eben der gleiche Prozessor ist, der für beide Verarbeitungsrichtungen verwendet wird.

– Parsing und Generierung können auf verschiedene Weise interagieren und miteinander verzahnt werden. So ist es z.B. möglich, Teilergebnisse der einen Verarbeitungsrichtung für die andere zu verwenden. Weiterhin kann die eine Verarbeitungsrichtung zur Kontrolle der anderen eingesetzt werden (s. den folgenden Abschnitt).

Eine uniforme Architektur des letzten Typs birgt andererseits aber die Gefahr, daß Parsing und Generierung ineffizient werden. Neuman (1994) hat deshalb an einer Implementierung gearbeitet, die die Effizienz uniformer Architekturen unter Beweis stellen soll.

1.4 Monitoring

Die Kontrolle der einen Verarbeitungsrichtung durch die andere wird als *Monitoring* bezeichnet. Dieses Thema wird bisher vor allem in der Kognitionswissenschaft und Psycholingustik behandelt (vgl. z.B. (Levelt 1989)). Zumeist geht es dabei um die Kontrolle von Generierung durch zugleich mitlaufendes Parsing der schon generierten phonolgischen Kette. Das Ziel ist die Generierung eines nicht-ambigen Satzes, d.h. dem Hörer soll das Verständnis der Äußerung erleichtert und so die Kommunikation entlastet werden.

Psycholinguistische Motivation findet sich weiter in der Tatsache, daß der Mensch sich selbst beim Sprechen verbessert und sogar Sätze abbricht, wenn er merkt, daß der intendierte Inhalt in Fortsetzung des schon Geäußerten nicht mehr realisiert werden kann.

Eine uniforme Architektur ermöglicht es, dieses Verhalten zu modellieren. Der Parser kann jeden Zustand des Generators zum Ausgangspunkt für eine Analyse nehmen. Da beide

Komponenten gewissermaßen 'die gleiche Sprache sprechen', können zudem Teilergebnisse wiederverwendet werden.

Wenn das Monitoring, wie oben beschrieben, zeitgleich mit der Generierung stattfindet, handelt es sich um eine *inkrementelle* Vorgehensweise. Sie kann lokale Ambiguitäten sofort entdecken, hat aber den Nachteil, daß sie nicht wissen kann, ob diese lokale Ambiguität durch erst später erzeugte Satzteile wieder aufgelöst wird oder nicht. Die Alternative ist, zu warten, bis der Satz beendet ist und diesen dann auf die noch verbleibenden Ambiguitäten zu untersuchen. In beiden Fällen ist das Ziel des Monitorings die *Revision* der erzeugten Satzteile.

Wenn bei der Analyse eines vom Gesprächspartner geäußerten Satzes Ambiguitäten festgestellt werden, kann die Generierungskomponente Paraphrasen für die verschiedenen Lesarten erzeugen, so daß es möglich ist, im Dialog zu klären, welche die intendierte Bedeutung ist.

Neumann (1994) und Neumann und van Noord (1992) haben ein Performanzmodell implementiert, daß sowohl zu inkrementellem Monitoring und Revision wie auch zur Erzeugung von Paraphrasen fähig ist. Diese allgemeinen Mechanismen sind grundsätzlich nicht von einem bestimmten Algorithmus abhängig.

Die Verwendung eines uniformen Algorithmus für Parsing und Generierung ist bei inkrementellem Monitoring zwingend erforderlich. Wenn jedoch gewartet wird, bis der ganze Satz erzeugt wurde, ist meiner Ansicht nach nur eine reversible Grammatik notwendig. Dies gilt auch für die Erzeugung von Paraphrasen, die sich darin vom (Self-)Monitoring unterscheidet, daß der gegebene Satz nicht vom Hörer selbst stammt.

Während die Kontolle von Generierung durch das Parsing die naheliegenste Anwendung für Monitoring ist, wird der umgekehrte Fall z.B. in (Levelt 1989) nicht behandelt. Die Frage ist, ob es Fälle gibt, in denen beim Parsing eines Satzes auftretende Ambiguitäten durch Generierung von den schon vorhandenen alternativen semantischen Repräsentationen aufgelöst werden könnnen (z.B. in dem Sinne, daß unter der Annahme der gleichen reversiblen Grammatik bei beiden Gesprächspartnern vom Hörer überlegt werden kann, daß der Sprecher eher eine andere Realisierung der semantischen Repräsentation gewählt hätte, so daß bestimmte Lesarten ausgeschlossen werden können).

Eine in diesem Zusammenhang relevante Beobachtung ist die Tatsache, daß Hörer in der Lage sind, von einem Sprecher noch nicht beendete Sätze weiterzuführen. Auch dies spricht für die enge Verzahnung von Parsing und Generierung. Die vom Sprecher schon geäußerten Satzteile stellen sozusagen Beschränkungen für die nun noch möglichen Weiterführungen des Satzes dar. Die Generierungskomponente verzichtet beim 'normalen' Verstehen gewissermaßen auf die artikulatorische Realisierung, um dem Sprecher nicht ins Wort zu fallen.

Die Möglichkeiten von Monitoring sind meiner Ansicht nach in bezug auf eine computerlingistische Behandlung bisher noch nicht ausgeschöpft worden.

Kapitel 2

Linguistische und formale Grundlagen

2.1 Constraint-basierte Grammatikformalismen

Die Grundlage für reversible Systeme sind deklarative Grammatikformalismen, die dazu dienen, Informationen über sprachliche Objekte zu repräsentieren. Diese Grammatikformalismen werden oft als *constraint-basiert* bezeichnet. Damit wird zum Ausdruck gebracht, daß das prinzipiell unendlich große Universum an Möglichkeiten, sprachliche Zeichen zu bilden, durch die Grammatik beschränkt wird auf diejenigen Zeichen und Zeichenkombinationen, die in der jeweiligen natürlichen Sprache erlaubt sind. Mit dieser sehr allgemeinen Interpretation des Begriffs *Constraint* (Beschränkung) werden Formalismen wie z.B. PATR-II, Lexical Functional Grammar (LFG) und Head-driven Phrase Structure Grammar (HPSG) erfaßt, die auch als *unifikationsbasiert* und *komplex-merkmalsbasiert* bezeichnet werden. Beide zuletzt genannten Bezeichnungen zielen auf Eigenschaften dieser Grammatikformalismen ab, die im Rahmen dieses Kapitels erläutert werden sollen. Aufgrund der Breite der Themen, die in dieser Arbeit angesprochen werden, erhebt die Darstellung keinen Anspruch auf Vollständigkeit, sondern dient eher dazu, einen Überblick zu geben.

Grammatikformalismen repräsentieren Eigenschaften natürlicher Sprachen in formal klarer Weise. Sie können deshalb als formale Metasprachen über natürlichen Objektsprachen angesehen werden. Darüber hinaus gibt es *Constraint-Sprachen*, die bestimmte Beschränkungen auf einer Datenstruktur (meistens Merkmalsstrukturen) formulieren und den erlaubten Ausdrücken eine Interpretation zuweisen. Constraint-Sprachen erreichen einen größeren Grad an Allgemeinheit als Grammatikformalismen, indem sie deren spezifische Beschränkungen zu vermeiden versuchen. Sprachliches Wissen soll so direkt wie möglich kodiert werden. Constraint-Sprachen stehen in einer Art Konkurrenzverhältnis zu Grammatikformalismen, wie Johnson (1995, S.6) zeigt. Andererseits können sie aber auch als formale Grundlegung für die Implementierung von Grammatikformalismen dienen. ALE kann z.B. als Implementierung des Formalismus in (Carpenter 1992) angesehen werden. Es lassen sich also drei Sprachtypen unterscheiden:

- Natürliche Objektsprachen.
- Grammatikformalismen: sie können neutral in bezug auf bestimmte linguistische Theorien sein (*werkzeug-orientierte* Formalismen wie PATR-II) oder schon bestimmte linguistische Theorien voraussetzen (*theorie-orientierte* Formalismen wie HPSG).
- Constraint-Sprachen, die die erlaubten Datenstrukturen und ihre Interpretation definieren. Als Beispiel kann die Logik von Kasper und Rounds (1986, s. auch die Darstellung in (Rumpf 1995)) angesehen werden oder die Constraint-Sprache L basierend auf (Höhfeld und Smolka 1988). Letztere dient als Grundlage für den Grammatikformalismus *R(L)* in (van Noord 1993) und (Neumann 1994). Van Noord verwendet die Constraint-Sprache *R(L)* aber nicht nur, um Aussagen über natürliche Objektsprachen zu machen (gewissermaßen als Grammatikformalismus), sondern auch zur Definition von Metainterpretern für das in dieser Sprache kodierte Wissen. Dennoch können Constraint-Sprachen nicht generell als Metasprachen über Grammatikformalismen betrachtet werden. Das Verhältnis zwischen beiden scheint mir noch nicht geklärt zu sein.

2.1.1 Merkmalsstrukturen

Merkmalsstrukturen dienen der Repräsentation von Bedingungen, die an wohlgeformte Ausdrücke einer natürlichen Sprache gestellt werden.

Eine Merkmalsstruktur (oder auch 'Merkmal-Wert-Struktur') besteht entweder aus einem Atom (atomare Merkmalsstruktur) oder aus einer Menge von Paaren aus atomarem Merkmal und Wert (komplexe Merkmalsstruktur), wobei der Wert wiederum eine atomare oder komplexe Merkmalsstruktur ist.

Diese rekursive Definition erlaubt Strukturen mit beliebiger Einbettungstiefe. Es handelt sich um eine partielle Funktion, die Merkmale auf ihre Werte abbildet. Die Funktion ist partiell, weil nicht alle möglichen Merkmale in einer Merkmalssstruktur definiert werden müssen.

Ein Pfad in einer Merkmalsstruktur besteht aus einer nicht-leeren Anzahl von Merkmalen, die unmittelbar aufeinander folgen. Sein Wert ist die Merkmalsstruktur, die am Ende dieses Pfades steht. Wenn der Pfad am Wurzelknoten beginnt und eine atomare Merkmalsstruktur als Wert hat, wird er als *vollständig* bezeichnet (Müller 1994, S.123).

Zwei Pfade einer Merkmalsstruktur können einen gemeinsamen Wert haben *(Koindizierung, reentrancy, structure sharing)*. Wenn sich der Wert des einen Merkmals ändert, ändert sich auch der Wert für das andere Merkmal, d.h. es besteht Token-Identität zwischen den Werten. Im Gegensatz dazu liegt Typ-Identität vor, wenn die Werte der beiden Merkmale nicht in dieser Weise voneinander abhängig sind. In diesem Fall müssen die Werte nicht auf dasselbe Objekt referieren. Es handelt sich um zwei Token.

Merkmalsstrukturen lassen sich in verschiedener Weise darstellen: als Attribut-Wert-Matrizen (*attribute-value matrix*, AVM), als gerichtete azyklische Graphen (*directed acyclic graph*, DAG) oder in Form von Pfadgleichungen. Die Darstellung als DAG macht den Unterschied zwischen Typ- und Token-Identität besonders deutlich: bei Token-Identität führen mehrere Kanten zu demselben Knoten, bei Typ-Identität führen sie zu verschiedenen Knoten. In der Darstellung als Attribut-Wert-Matrize wird Token-Identität durch Koindizierung ausgedrückt (s. Abb. 2.1). In den in dieser Arbeit verwendeten Repräsentationen lasse ich spezielle Bezeichner für die Merkmalsstrukturen weg, da aus dem Kontext hervorgeht, auf welche jeweils Bezug genommen wird.

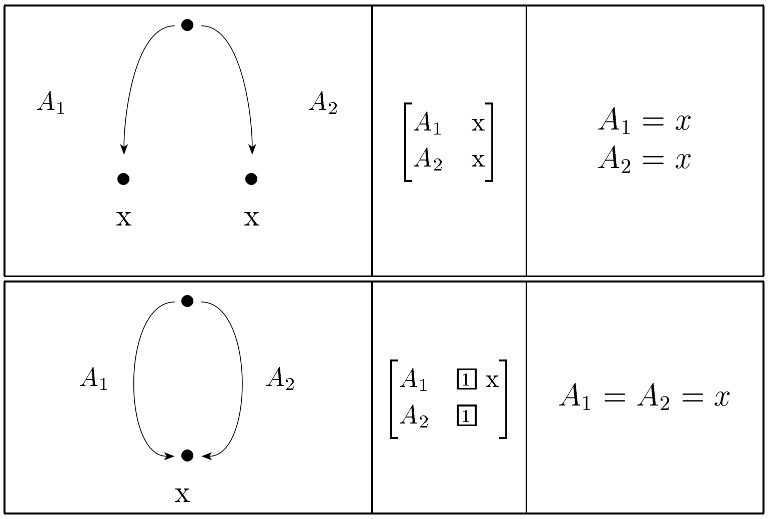


Abbildung 2.1: Typ- und Tokenidentität in DAGs, AVMs und Pfadgleichungen

Die Reihenfolge der Merkmale in einer Merkmalsstruktur ist unerheblich. Dies unterscheidet Merkmalsstrukturen als Datenstruktur von Termen erster Ordnung, wie sie in Prolog verwendet werden. In letzteren kann z.B. jedes Merkmal einer bestimmten Argumentposition zugeordnet werden. Da die Anzahl der Argumente festgelegt ist, können nicht ohne weiteres beliebige neue Merkmale hinzugefügt werden. Trotzdem ist es möglich, Merkmalsstrukturen in Termen erster Ordnung zu kodieren (vgl. Abschnitt 5.1). Terme sind im Vergleich zu Merkmalsstrukturen jedoch weniger intuitiv zu handhaben.

2.1.2 Relationen und Operationen auf Merkmalsstrukturen

Merkmalsstrukturen enthalten partielle Informationen über sprachliche Objekte. Sie sind insofern partiell, als sie nicht alle Beschränkungen enthalten, die für die entsprechenden sprachlichen Objekte gelten.

Die Information verschiedener Merkmalsstrukturen muß zusammengebracht werden können, um größere Informationseinheiten zu bilden. Die Verwendung in reversiblen Grammatiken erfordert, daß die Reihenfolge, in der dies geschieht, keine Rolle spielt. Es kann nicht erwartet werden, daß die Reihenfolge der Kombination von Merkmalsstrukturen bei Parsing und Generierung die gleiche ist – ganz abgesehen davon, daß dies schon beim Parsing verschiedener Sätze nicht vorausgesetzt werden kann. Es wird also eine monotone Operation benötigt: es kann zwar Information zu einer Merkmalsstruktur hinzugefügt, aber nicht gelöscht oder verändert werden. Damit ist ausgeschlossen, daß Informationen, die eine Kombination verhindern, bei einer bestimmten Reihenfolge noch vorhanden sind und bei einer anderen nicht, so daß verschiedene Merkmalsstrukturen das Ergebnis wären.

Eine Operation, die diese wünschenswerte Eigenschaft aufweist, ist die Unifikation. Zu ihrer Definition ist die Festlegung der Subsumptionsrelation zwischen Merkmalsstrukturen notwendig, die zunächst informell definiert werden soll.

Subsumption

Subsumption ist eine zweistellige Relation auf Merkmalsstrukturen: eine Merkmalsstruktur M1 subsumiert eine Merkmalsstruktur M2, wenn sie genereller ist als M2 ($M1 \sqsubseteq M2$). Dies bedeutet, daß in M2 mindestens die Information von M1 enthalten ist. Genauer gesagt: jeder vollständige Pfad in M1 ist auch in M2 enthalten und hat dort denselben Wert. M1 ist eine (unechte) Teilmenge von M2.

Die Subsumptionsrelation ist nach Shieber (1986, S.15) nicht für atomare Merkmalsstrukturen (Werte) definiert, z.B. stehen *sing* und *plur* als atomare Werte eines Merkmals *num* nicht in einer Subsumptionsbeziehung. Nach Müller (1994, S.126) gilt dagegen, daß zwei atomare Merkmalsstrukturen in einer Subsumptionsbeziehung stehen, wenn sie Typidentisch sind. Diese Auffassung ist insofern wünschenswerter, als nun alle Arten von Merkmalsstrukturen in einer Subsumptionsrelation stehen können.

Eine leere Merkmalsstruktur subsumiert alle anderen Merkmalsstrukturen. Ordnet man Merkmalsstrukturen bezüglich ihres Informationsgehalts, so ergibt sich eine hierarchische Struktur mit der leeren Merkmalsstruktur als maximalem Element. Es handelt sich um eine Halbordnung, weil nicht für alle Paare von Merkmalsstrukturen gilt, daß sie in einer Subsumptionsbeziehung stehen.

Koindizierte Werte werden als spezifischer angesehen als zwei Typ-identische Werte: in ihnen ist gewissermaßen zusätzlich die Bedingung (die Information) enthalten, daß der Merkmalswert auch bei Änderungen für beide Pfade gleich bleibt.

Die Subsumptionsrelation hat folgende Eigenschaften:

- **Reflexivität** Jede Merkmalsstruktur subsumiert sich selbst: $M \sqsubseteq M$.

- **Transitivität** Wenn $M1 \sqsubseteq M2$ und $M2 \sqsubseteq M3$, dann gilt auch $M1 \sqsubseteq M3$.

- **Antisymmetrie** Wenn $M1 \sqsubseteq M2$ und $M2 \sqsubseteq M1$, dann ist $M1 = M2$. Anders herum gesagt: wenn eine Merkmalsstruktur M3 eine Merkmalsstruktur M4 subsumiert und weniger Information als M4 enthält (also der reflexive Fall ausgeschlossen

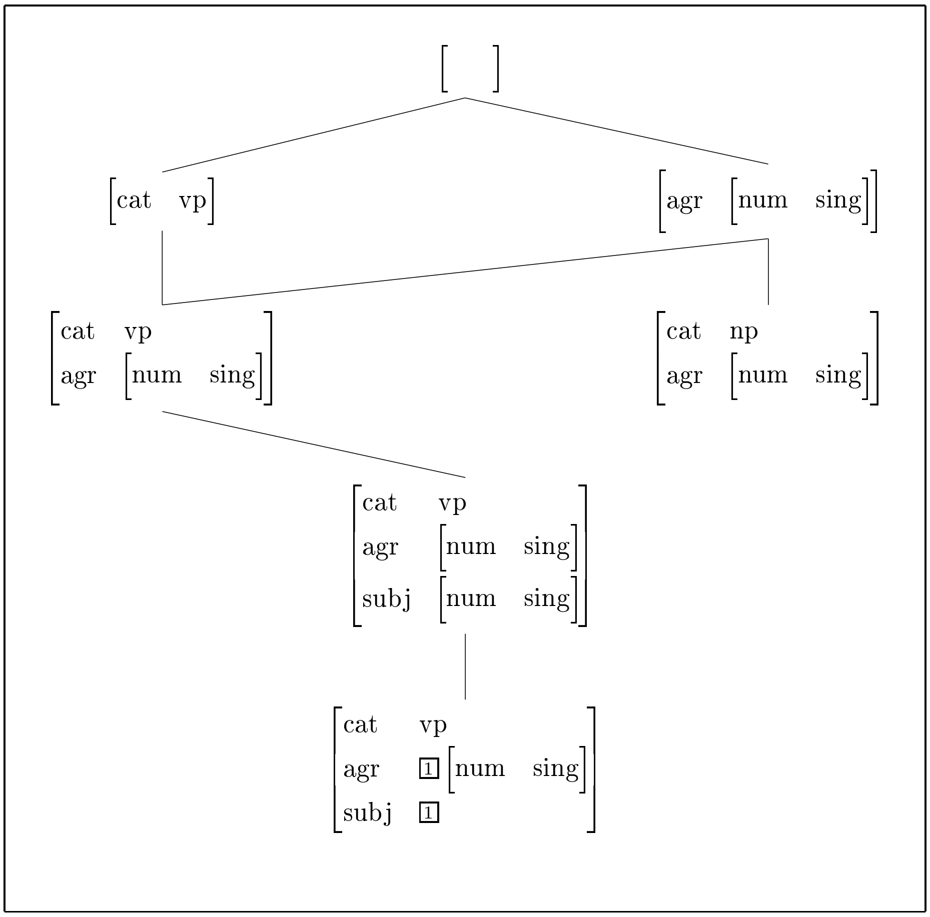


Abbildung 2.2: Eine Subsumptions-Halbordnung

ist), dann gilt die Subsumptionsrelation nur in einer Richtung, d.h. $M3 \sqsubseteq M4$ und $M4 \not\sqsubseteq M3$.

Unifikation

Da die Subsumptionsrelation jetzt zur Verfügung steht, kann mit ihrer Hilfe die Unifikation als die Kombination der Informationen zweier Merkmalsstrukturen definiert werden. Unifikation ist eine Operation auf zwei Merkmalsstrukturen M1 und M2, die als Ergebnis eine Merkmalsstruktur M3 hat ($M1 \sqcup M2 = M3$). Die Bedingung für das Gelingen der Unifikation ist die Kompatibilität der Information in M1 und M2. Es dürfen keine Widersprüche auftreten: für jeden vollständigen Pfad, der sowohl in M1 wie auch in M2 vorkommt, gilt, daß er in beiden den gleichen Wert hat.

Die Ergebnis-Merkmalsstruktur M3 wird sowohl von M1 wie auch von M2 subsumiert. Koindizierungen bleiben erhalten, da anderenfalls Information verlorengeht. Zusätzlich wird verlangt, daß keine weitere Information (außer der aus M1 und M2) hinzugefügt wird. M3 ist deshalb die generellste Merkmalsstruktur, die von M1 und M2 subsumiert wird. Alle anderen Merkmalsstrukturen, die von M1 und M2 subsumiert werden, werden auch von M3 subsumiert. Wenn die Unifikation gelingt, hat sie immer ein einziges Resultat.

Auch die Unifikation bildet eine Halbordnung: sie gelingt nicht für jedes Paar von Merkmalsstrukturen. Das maximale Element ist die leere Merkmalsstruktur (mit *Top* oder mit $\top$ bezeichnet). Es unifiziert mit allen anderen Merkmalsstrukturen, wobei es keine neue Information hinzufügt. Für den Fall des Scheiterns der Unifikation wird als weiteres Element *Bottom* (oder auch $\perp$) eingeführt, das das minimale Element der Halbordnung darstellt (in manchen Darstellungen werden *Top* und *Bottom* umgekehrt verwendet).

Die Unifikationsoperation hat folgende Eigenschaften (Müller 1994, S. 130):

- **Idempotenz** Zwei identische Merkmalsstrukturen unifizieren zu einer mit ihnen identischen Merkmalsstruktur: $M \sqcup M = M$.

- **Kommutativität** $M1 \sqcup M2 = M2 \sqcup M1$.

- **Assoziativität** Die Reihenfolge der Unifikation zweier Merkmalsstrukturen M1 und M2 hat keinen Einfluß auf das Ergebnis: $(M1 \sqcup M2) \sqcup M3 = M1 \sqcup (M2 \sqcup M3)$.

- **Neutrales Element** Die leere Merkmalsstruktur (*Top*, $\top$) hat als Ergebnis der Unifikation mit einer Merkmalsstruktur M wiederum M: $\top \sqcup M = M$.

- Die Unifikation von *Bottom* mit einer Merkmalsstruktur M scheitert, d.h. dem Ergebnis wird wiederum das Element *Bottom* zugewiesen: $\perp \sqcup M = \perp$.

- Unifikation und Subsumption können wechselseitig auseinander definiert werden: $M1 \sqsubseteq M2 \Leftrightarrow M1 \sqcup M2 = M2$.

Für reversible Grammatiken ist die Reihenfolgeunabhängigkeit, die sich aus der Assoziativität und der Kommutativität der Unifikation ergibt, besonders wichtig.

Zyklische Merkmalsstrukturen Die Unifikation zweier nicht-zyklischer Merkmalsstrukturen kann zu einer zyklischen Merkmalsstruktur führen. Das Ergebnis der Unifikation laßt sich nicht mehr als azyklischer Graph darstellen (s. Abb. 2.3).

Gerdemann (1991, S.12) gibt im Anschluß an Kasper und Rounds (1986) eine formale Definition von Merkmalsstrukturen auf der Basis von DAGs. Eine zyklische Merkmalsstruktur wird von einer solchen Definition naturgemäß ausgeschlossen, d.h. es handelt sich um eine ungültige Struktur. (Carpenter 1992) erlaubt dagegen auch zyklische Merkmalsstrukturen. Zum einen werden sie zur Repräsentation linguistischen Wissens tatsächlich benötigt (Carpenter 1992, S. 35). Zum anderen hat sich vom Standpunkt der Implementierung her gezeigt, daß Operationen auf zyklischen Strukturen zwar schwer handhabbar sind, aber der Aufwand für einen Test auf Zyklizität noch größer ist.

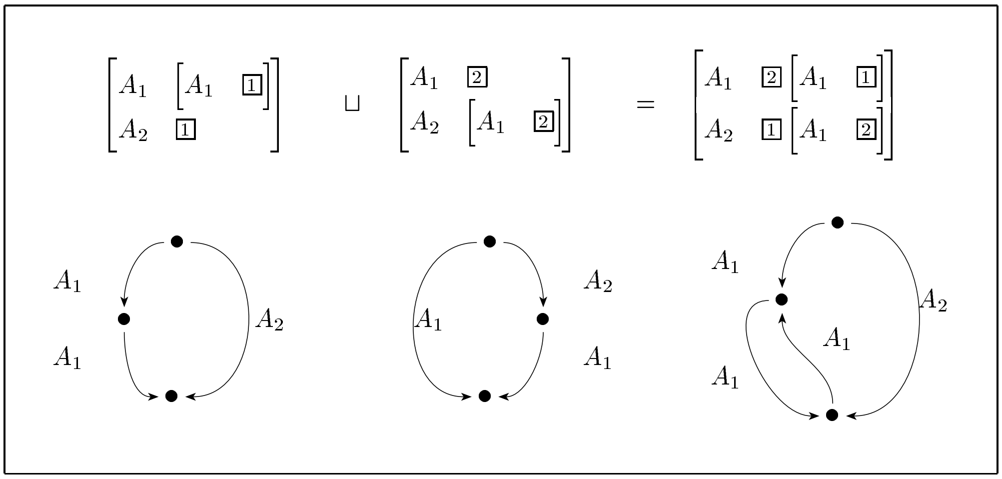


Abbildung 2.3: Bildung einer zyklischen Merkmalsstruktur durch Unifikation. Darstellung als AVMs und als gerichtete Graphen.

2.1.3 Getypte Merkmalsstrukturen

Gemäß der bisher gegebenen Beschreibung von Merkmalsstrukturen ist es möglich, durch Unifikation jede Art von Information hinzuzufügen, solange sie nicht im Widerspruch zu der schon vorhandenen steht. Dadurch kann eine Merkmalsstruktur Informationen enthalten, die semantisch gesehen in keiner Weise zusammenpassen.

Diese Überlegung ist eine wesentliche Motivation für die Festlegung einer bestimmten Menge von Merkmalen, die für eine Merkmalsstruktur als angemessen gelten sollen *(appropriateness)*. Auch für die Merkmale kann festgelegt werden, wie ihre Werte beschaffen sein sollen.

Weitere Motivation für diese Festlegung ist die nun mögliche Unterscheidung zwischen Informationen, die zwar angemessen, aber noch nicht bekannt sind, und solchen, die für die Merkmalsstruktur überhaupt nicht relevant sind.

Zudem kann die Negation von Merkmalsstrukturen erst jetzt sinnvoll definiert werden. Wenn z.B. für das Merkmal PERSON gesagt werden soll, daß der Wert nicht *2* sein darf, dann möchte man damit ausdrücken, daß es sich um die erste oder dritte Person handelt. Dieser Schluß kann aber nur gezogen werden, wenn das Universum der möglichen Werte an dieser Stelle auf {*1,2,3*} beschränkt wird.

Die Menge von Merkmalen, die für eine Merkmalsstruktur als angemessen gelten soll, wird mit einem Namen belegt, dem *Typ*. Man sagt, daß eine Merkmalsstruktur von einem bestimmten Typ ist. Dieser Typ wird in der Attribut-Wert-Matrix als Index an einer bestimmten Stelle, z.B. am unteren Ende der linken Klammer, angegeben. In der Darstellung als DAG wird zu jedem Knoten der Name des Typs hinzugefügt.

Typverbände

Für getypte Merkmalsstrukturen gilt, daß sie einem Typ zugeordnet werden. Die Typen können in einer Hierarchie, genauer: einem Verband, angeordnet werden, der die Subsumptionsbeziehung zwischen den Typen widerspiegelt.[1]

Ein Typ T_1 subsumiert einen Typ T_2, wenn alle Merkmale, die in Merkmalsstrukturen vom Typ T_1 vorkommen, auch in Merkmalsstrukturen vom Typ T_2 vorkommen. Für Merkmale, die in beiden Merkmalsstrukturen vorkommen, muß wiederum eine Subsumptionsbeziehung zwischen ihren Typen bestehen: der Typ des Merkmals in (Merkmalsstrukturen vom Typ) T_1 subsumiert den Typ in (Merkmalsstrukturen vom Typ) T_2. Der Typ T_1 wird als der *Supertyp* von T_2 bezeichnet; T_2 ist ein *Subtyp* von T_1. Zwei Typen werden als *disjunkt* bezeichnet, wenn sie nicht in einer Subsumptionsbeziehung stehen.

Der Typverband, der durch die Subsumptionsrelation gebildet wird, läßt sich als Vererbungsnetzwerk interpretieren. Der spezifischere Subtyp erbt alle Merkmale des generelleren Supertyps. Wenn der Typverband bekannt ist, läßt sich für jede Merkmalsstruktur, die von einem Typ aus diesem Verband ist, bestimmen, welche Merkmale für sie angemessen sind. Da die Subsumptionsrelation transitiv ist, erbt ein Typ von allen seinen Supertypen. Die Berechnung der angemessenen Merkmale für einen unterspezifizierten Typ wird als *Typ-Inferenz* bezeichnet.

Eine Merkmalsstruktur, die alle Merkmale besitzt, die für ihren Typ definiert sind, ist *total wohlgetypt (totally well-typed)*.[2] Wenn alle ihre Teilmerkmalsstrukturen von Typen sind, die keine Subtypen mehr haben, wird sie als *Typen-aufgelöst (sort-resolved)* bezeichnet (Pollard und Sag 1994, S. 18).

An der Spitze des Typverbandes steht der generellste Typ $\top$, der alle anderen Typen subsumiert. Der spezifischste Typ $\perp$ repräsentiert wieder inkonsistente Information.

Zyklische Strukturen Eine wichtige Einschränkung der Vererbungshierarchie ist, daß keine zyklischen Strukturen erlaubt sind. Es darf nicht möglich sein, von einem Typ aus die Kanten zu den Subtypen zu verfolgen und wieder bei diesem Typ anzukommen (außer im reflexiven Fall). Die Relation zwischen zwei Typen ist also anti-symmetrisch. Dies ist, wie bereits erwähnt, eine Eigenschaft der die Hierarchie begründenden Subsumptionsrelation.

Unifikation von getypten Merkmalsstrukturen

Auch die Information zweier getypter Merkmalsstrukturen kann durch eine Unifikations-Operation vereint werden. Zu den Bedingungen für die Unifikation ungetypter Merkmals-

[1]Es sind jedoch auch schwächere Anforderungen an die Typ-Hierarchie möglich: TFS und CUF verlangen eine partielle Ordnung und keinen Verband wie ALE (Manandhar 1993). Verbände können als partielle Ordnungen angesehen werden, die speziellen Anforderungen genügen: für jedes Paar von Elementen eines Verbandes ist eine einzige tiefste obere Schranke und eine einzige höchste untere Schranke definiert (Partee et al. 1990, Kap.11).

[2]Müller (1994, S. 136) verwendet den Ausdruck 'total typisiert'.

strukturen kommt die Anforderung, daß auch die Typen der beiden Merkmalsstrukturen unifiziert werden müssen. Durch das Ergebnis dieser Unifikation wird festgelegt, welcher Typ der Ergebnis-Merkmalsstruktur zugewiesen wird.

Unifikation von Typen Da die Subsumptionsrelation auf Typen definiert wurde, kann die Unifikation von Typen nun analog zur Unifikation ungetypter Merkmalsstrukturen definiert werden. Ein Typ T_3 ist das Ergebnis der Unifikation von T_1 und T_2, wenn er von T_1 und T_2 subsumiert wird und er alle anderen Typen subsumiert, die auch von T_1 und T_2 subsumiert werden. Die letzte Bedingung stellt sicher, daß T_3 der am wenigsten spezifische Typ ist, der von T_1 und T_2 subsumiert wird.

In Begriffen der Verbandstheorie stellt der Ergebnis-Typ die gemeinsame größte untere Schranke der beiden unifizierten Typen dar. Er ist also mindestens so spezifisch wie die beiden unifizierten Typen. Damit ist – wie bei der Unifikation von Merkmalsstrukturen bereits angesprochen – sichergestellt, daß keine Information verlorengeht. Andererseits ist durch die Bedingung, daß das Ergebnis der generellste der von beiden subsumierten Typen ist, dafür gesorgt, daß keine Information außer der in den beiden Ausgangs-Typen enthaltenen im Ergebnis vorhanden ist.

2.2 Reversible Grammatiken

Unter Verwendung von Merkmalsstrukturen als Datenstruktur ist es möglich, die Aufgabe, die im Rahmen dieser Arbeit an reversible Architekturen gestellt wird, in allgemeiner Form zu formulieren. Der Ausgangspunkt ist die Annahme, daß ein sprachliches Zeichen zumindest durch eine phonologische und eine semantische Repräsentation gekennzeichnet ist. Parsing und Generierung können dann als Vervollständigung der Werte einer in einer dieser beiden Merkmale unterspezifizierten Merkmalsstruktur angesehen werden. Diese generelle Aufgabenstellung erlaubt es, auch unkonventionelle Ansätze ohne Phrasenstrukturregeln (s. Abschnitt 6.2) zu erfassen. Aus jeweils einer der beiden unterspezifizierten Merkmalsstrukturen[3]

$$\text{(1a)} \begin{bmatrix} \text{phon} & \left\langle \text{peter, sieht, maria} \right\rangle - \left\langle \ \right\rangle \end{bmatrix} \quad \text{und} \quad \text{(1b)} \begin{bmatrix} \text{sem} & \begin{bmatrix} \text{pred} & \text{sehen} \\ \text{arg1} & \text{peter} \\ \text{arg2} & \text{maria} \end{bmatrix} \end{bmatrix}$$

wird folgende Merkmalsstruktur erzeugt:

[3]Für die phonolgischen Repräsentationen werden hier Differenzlisten benutzt, die in Abschnitt 2.2.1 näher erläutert werden.

(2) $\begin{bmatrix} \text{phon} & \left\langle \text{peter, sieht, maria} \right\rangle - \left\langle\ \right\rangle \\ \text{sem} & \begin{bmatrix} \text{pred} & \text{sehen} \\ \text{arg1} & \text{peter} \\ \text{arg2} & \text{maria} \end{bmatrix} \end{bmatrix}$

Der erste Fall kann als Parsing, der zweite als Generierung aufgefaßt werden.

Die Generierung von sprachlichen Äußerungen kann unterteilt werden in zwei verschiedene Schritte. Der *strategische Teil* entscheidet, was inhaltlich gesagt werden soll; der *taktische Teil* bestimmt, wie der Inhalt realisiert wird. Während der erste Teil also die semantische Struktur festlegt, von der aus generiert werden soll, dient der zweite Teil dazu, die Repräsentation einer phonologischen Realisierung zu einer gegebenen semantischen Repräsenation zu finden.[4] Insofern ist mit 'Generierung' im Rahmen reversibler Grammatiken meistens der taktische Teil gemeint.

Unifikationsbasierte Semantik Semantische Information wird wie alle anderen Informationen in der Form von Merkmalsstrukturen repräsentiert. Die Verknüpfung der Information erfolgt durch Unifikation. Dies ist nach den vorgestellten Überlegungen naheliegend, es gibt jedoch auch alternative Datenstrukturen und Operationen auf ihnen, z.B. Lambda-Ausdrücke und Lambda-Reduktion.

Die Vor- und Nachteile der verschiedenen Ansätze zur semantischen Repräsentation können im Rahmen dieser Arbeit nicht diskutiert werden. Für die unifikationsbasierte Semantik spricht, daß die Schnittstelle zwischen Syntax und Semantik in sehr einfacher Weise definiert werden kann. Auch phonologische oder pragmatische Information beispielsweise kann zur Semantik beitragen, indem die entsprechenden Constraints oder Pfadgleichungen formuliert werden.

Weiterhin kann durch die Verwendung von Unifikation auch zur Kombination von semantischer Information diese parallel zu Syntax und phonologischer Repräsentation aufgebaut werden. Dies ist gerade für reversible Grammatiken von Vorteil. Für alle Repräsentationen von Phrasen auf dem Weg zu Parsing oder Generierung eines vollständigen Satzes kann angenommen werden, daß sie zumindest die vorhandene phonologische und semantische Information wiedergeben. Da der Aufbau der Repräsentationen der verschiedenen Abbildungsebenen genau gleichzeitig erfolgt, kann es in dieser Hinsicht keinen Unterschied zwischen den Verarbeitungsrichtungen geben.

[4]Tatsächlich legt der strategische Teil mehr fest, nämlich vor allem die Struktur eines ganzen Textes, in den der zu erzeugende Satz sinnvoll eingepaßt werden muß *(Text Planning)*.
Diese und andere Fragen in Zusammenhang mit der Generierung von Äußerungen können hier nicht behandelt werden. Dazu gehört z.B. auch das Problem der logischen Äquivalenz von Formeln, die zur semantischen Repräsentation dienen (vgl. Shieber (1988, S. 618) und van Noord (1993, S. 15)).

2.2.1 Beispiel-Repräsentationen

Semantische Strukturen Um einen Eindruck von den semantischen Strukturen zu geben, wie sie in (van Noord 1993) und (Neumann 1994) verwendet werden, sollen hier einige Beispiele angeführt werden. Es handelt sich um Prädikat-Argument-Strukturen, die mit Information über die Stelligkeit versehen werden. Als Beispiel wird die Repräsentation von *schlafen* und *Peter ißt Äpfel/einen Apfel* angegeben.

$$\text{(3a)} \quad \begin{bmatrix} \text{sort} & \text{unary} \\ \text{pred} & \text{schlafen} \\ \text{arg1} & \text{X} \end{bmatrix} \qquad\qquad \text{(3b)} \quad \begin{bmatrix} \text{sort} & \text{binary} \\ \text{pred} & \text{essen} \\ \text{arg1} & \begin{bmatrix} \text{sort} & \text{nullary} \\ \text{pred} & \text{peter} \end{bmatrix} \\ \text{arg2} & \begin{bmatrix} \text{sort} & \text{nullary} \\ \text{pred} & \text{apfel} \end{bmatrix} \end{bmatrix}$$

Referentielle Objekte werden als nullstellige Prädikate aufgefaßt. Die Merkmalsstrukturen sind in bezug auf Quantifizierung hier der Einfachheit halber nicht spezifiziert.

Die von Neumann und von van Noord verwendete Constraint-Sprache verfügt nur über ungetypte Merkmalsstrukturen. Durch die Verwendung von Typen kann das Merkmal SORT als Typname benutzt werden. Die beiden Merkmalsstrukturen sehen nun folgendermaßen aus:

$$\text{(4a)} \quad {}_{\textit{unary}}\begin{bmatrix} \text{pred} & \text{schlafen} \\ \text{arg1} & \text{X} \end{bmatrix} \qquad\qquad \text{(4b)} \quad {}_{\textit{binary}}\begin{bmatrix} \text{pred} & \text{essen} \\ \text{arg1} & {}_{\textit{nullary}}\begin{bmatrix} \text{pred} & \text{peter} \end{bmatrix} \\ \text{arg2} & {}_{\textit{nullary}}\begin{bmatrix} \text{pred} & \text{apfel} \end{bmatrix} \end{bmatrix}$$

Modifier-Konstruktionen werden durch die Merkmale MOD und ARG1 repräsentiert. Die modifizierte Struktur ist das Argument des Modifiers. Als Beispiel wird die semantische Repräsentation von *Heute trinkt Peter irischen Whisky* angegben:

$$\text{(5)} \quad {}_{\textit{modifier}}\begin{bmatrix} \text{mod} & {}_{\textit{nullary}}\begin{bmatrix} \text{pred} & \text{heute} \end{bmatrix} \\ \text{arg1} & {}_{\textit{binary}}\begin{bmatrix} \text{pred} & \text{trinken} \\ \text{arg1} & {}_{\textit{nullary}}\begin{bmatrix} \text{pred} & \text{peter} \end{bmatrix} \\ \text{arg2} & {}_{\textit{modifier}}\begin{bmatrix} \text{mod} & \text{irischen} \\ \text{arg1} & {}_{\textit{nullary}}\begin{bmatrix} \text{pred} & \text{whisky} \end{bmatrix} \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

Phonologische Ketten Eine Kette von phonologischen Zeichen kann durch eine Liste repräsentiert werden. Die einzige hier berücksichtigte Methode, eine phonologische Kette aufzubauen, ist die Konkatenation von (Teil-)Listen. Darüber hinausgehende Operationen auf phonologischen Ketten werden im Rahmen dieser Arbeit nicht behandelt (vgl. (van Noord 1993, Kapitel 4.2.: *Beyond concatenation*) und (Erbach 1995a, Kapitel 3.2.1–3.2.3)). Die Konkatenation von Listen ist mithilfe von Unifikation leicht zu modellieren, weil Information nicht gelöscht, sondern nur hinzugefügt wird.

Listen können als Merkmalsstrukturen mit den Merkmalen FIRST und REST rekursiv definiert werden. Sie bestehen aus einem ersten Element (der Wert von FIRST) und einem Rest (der Wert von REST), der leer sein kann oder wiederum eine Liste.

Phonologische Ketten werden hier darüber hinaus als Differenzlisten repräsentiert. Es werden zwei Listen zueinander in Beziehung gesetzt: die zu repräsentierende Liste ist das Ergebnis der Subtraktion der zweiten von der ersten Liste. Die Verwendung von Differenzlisten z.B. in Prolog-Interpretern für Definite Clause Grammars (s. Abschnitt 4.1.1) hat den praktischen Vorteil, daß die Restliste gewissermaßen automatisch berechnet wird, wenn die zu repräsentierende Kette und die erste (Input-)Liste bekannt sind. Darüberhinaus hat dies den theoretischen Vorteil, daß keine zusätzliche Konkatenations-Funktion benötigt wird. Die Restliste wird als Variable (in Großbuchstaben) dargestellt.

Ein Beispiel für eine getypte Merkmalsstruktur zur Repräsentation einer Liste:

(6)
$$
{}_{\mathit{ne\_list}}\begin{bmatrix}
\text{first} & \text{heute} \\
\text{rest} & {}_{\mathit{ne\_list}}\begin{bmatrix}
\text{first} & \text{trinkt} \\
\text{rest} & {}_{\mathit{ne\_list}}\begin{bmatrix}
\text{first} & \text{peter} \\
\text{rest} & {}_{\mathit{ne\_list}}\begin{bmatrix}
\text{first} & \text{irischen} \\
\text{rest} & {}_{\mathit{ne\_list}}\begin{bmatrix}
\text{first} & \text{whisky} \\
\text{rest} & \text{REST}
\end{bmatrix}
\end{bmatrix}
\end{bmatrix}
\end{bmatrix}
\end{bmatrix}
$$

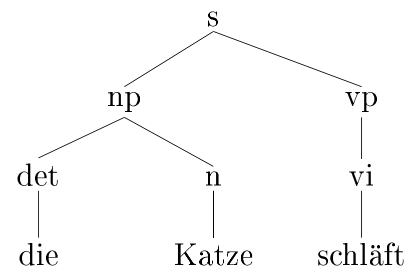
Die Typenbezeichnung *ne_list* steht fur 'non-empty list'. Zur besseren Lesbarkeit kann eine abkürzende Schreibweise verwendet werden. Die Restliste wird explizit wiederaufgenommen, so daß der Differenzlistencharakter deutlich wird:

(7)
$$
\begin{bmatrix}\text{phon} & \left\langle \text{heute, trinkt, peter, irischen, whisky} \,|\, \text{Rest} \right\rangle - \text{Rest}\end{bmatrix}
$$

Konstituentenstruktur Bisher wurden nur PHON und SEM als *essentielle Merkmale* (Neumann 1994, S.80) erwähnt. Auch die (syntaktische) Konstituentenstruktur ist jedoch von Interesse, um den Aufbau einer Konstituente aus anderen nachvollziehen zu können. Diese Information kann in der Form von Merkmalsstrukturen repräsentiert werden. Sie wird während der Verarbeitung gleichzeitig mit der Semantik bzw. der phonologischen Kette aufgebaut.

Eine Ableitungsstruktur läßt sich – wie Listen auch – in einer rekursiven Struktur repräsentieren. Um die Informationsmenge nicht zu groß werden zu lassen, wird für jede Regelanwendung der Name der Regel angegeben (mit dem Merkmal LABEL) und die Liste der Regelanwendungen für die Elemente der rechten Regelseite (mit dem Merkmal DTRS). Der rekursive Fall ist gegeben, wenn die Liste der Elemente der rechten Regelseite mindestens ein Element enthält. Es wird dann für jedes Element wiederum der Regelname und die Liste der Elemente der rechten Seite der betreffenden Regel angegeben. Die Abbruchbedingung stellen lexikalische Einträge dar: die Liste der Tochterknoten ist leer.

Die Ableitungsstruktur repräsentiert also Regelanwendungen, indem jede Regel durch einen eindeutigen Bezeichner angegeben wird. Es grundsätzlich auch möglich, statt einem Regelnamen die syntaktische Kategorie des Elements der linken Regelseite anzugeben. Dann kann aber bei lexikalischen Einträgen keine eindeutige Zuordnung zu einem bestimmten lexikalischen Eintrag mehr vorgenommen werden, da es normalerweise mehrere Einträge der betreffenden syntaktische Kategorie gibt.



$$\begin{bmatrix} \text{label} & \text{s\_np\_vp} \\ \text{dtrs} & \left\langle \begin{bmatrix} \text{label} & \text{np\_det\_n} \\ \text{dtrs} & \left\langle \begin{bmatrix}\text{label} & \text{det\_die}\\ \text{dtrs} & \langle\,\rangle\end{bmatrix}, \begin{bmatrix}\text{label} & \text{n\_katze}\\ \text{dtrs} & \langle\,\rangle\end{bmatrix}\right\rangle \end{bmatrix}, \begin{bmatrix}\text{label} & \text{vp\_vi}\\ \text{dtrs} & \left\langle \begin{bmatrix}\text{label} & \text{vi\_schläft}\\ \text{dtrs} & \langle\,\rangle\end{bmatrix}\right\rangle\end{bmatrix}\right\rangle \end{bmatrix}$$

Abbildung 2.4: Ableitungsstruktur als Baum und als Merkmalsstruktur

Die in Abb. 2.4 kodierte Konstituentenstruktur nimmt eine top-down-Sicht auf den repräsentierten Baum ein. Obwohl dies zunächst als die natürlichste Betrachtungsweise erscheinen mag, sind jedoch auch andere Sichtweisen möglich (z.B. bottom-up), die grundsätzlich gleichberechtigt sind. Kilbury (unveröffentlicht) zeigt, daß die Repräsentation von Konstituentenstrukturbäumen von einer Perspektive in andere transformiert werden kann.

2.2.2 Eigenschaften reversibler Grammatiken

Kohärenz und Vollständigkeit

Parsing und Generierung wurden oben beschrieben als die Vervollständigung von unterspezifizierten Merkmalsstrukturen. Das *unbeschränkte Parsing-Problem* besteht darin, zu einer beliebigen gegebenen phonologischen Kette alle passenden semantischen Repräsentationen aufzuzählen *(p-enumeration)*. Entsprechend besteht das *unbeschränkte Generierungs-Problem* darin, zu einer gegebenen semantischen Repräsentation alle passenden phonologischen Ketten aufzuzählen *(g-enumeration)*.

Im Rahmen von Unifikationsgrammatiken kann das Wort ‘passend’ im Sinne von ‘unifiziert’ aufgefaßt werden. Die gegebene Information, d.h. die phonologische Kette oder semantische Struktur, stellt einen Constraint dar, der die Menge der möglichen Lösungen einschränkt. Die Ergebnis-Merkmalsstrukturen müssen kompatibel mit der Input-Merkmalsstruktur sein. Input- und Ergebnis-Merkmalsstrukturen müssen unifiziert werden können.

Es sind zwei Fälle möglich, in denen die Ergebnis-Merkmalsstruktur zwar kompatible Information enthält, aber trotzdem nicht das gewünschte Resultat darstellt. Zum einen kann nur ein Teil der schon gegebenen Information vorhanden sein. Zum anderen kann der gegebenen Information weitere hinzugefügt werden, ohne die Bedingung der Kompatibilität zu verletzten.

Als Beispiel für das Generierungsproblem soll der Satz *der mann trinkt* aus folgender semantischer Repräsentation generiert werden:

$$(8)\quad \begin{bmatrix} \text{sem} & \begin{bmatrix} \text{pred} & \text{trinken} \\ \text{arg1} & \text{mann} \end{bmatrix} \end{bmatrix}$$

Die Grammatik lizensiert (neben anderen) diese beiden Merkmalsstrukturen:

$$(9a)\quad \begin{bmatrix} \text{phon} & \left\langle \text{trinkt} \mid \text{R} \right\rangle - \text{R} \\ \text{sem} & \begin{bmatrix} \text{pred} & \text{trinken} \\ \text{arg1} & \text{X} \end{bmatrix} \end{bmatrix} \quad \text{und} \quad (9b)\quad \begin{bmatrix} \text{phon} & \left\langle \text{der, mann, trinkt, whisky} \mid \text{R} \right\rangle - \text{R} \\ \text{sem} & \begin{bmatrix} \text{pred} & \text{trinken} \\ \text{arg1} & \text{mann} \\ \text{arg2} & \text{whisky} \end{bmatrix} \end{bmatrix}$$

Beide könnten als gültige Antworten angesehen werden, wenn nur Kompatibilität zur Input-Merkmalsstruktur gefordert ist.[5]

Für das Parsing-Problem können analoge Beispiele gefunden werden. Zu der Input-Merkmalsstruktur

$$(10)\quad \begin{bmatrix} \text{phon} & \left\langle \text{der, mann, trinkt} \mid \text{R} \right\rangle - \text{R} \end{bmatrix}$$

sind die folgenden beiden Merkmalsstrukturen kompatibel:

[5] In (8) und (9a) wird *trinken* intransitiv verwendet und in (9b) transitiv. Der hier wichtige Punkt ist, daß, wenn keine getypten Merkmalsstrukturen benutzt werden, die beiden Merkmalsstrukturen in (9) zu der in (8) kompatibel sind.
Im nächsten Abschnitt werden sowohl die Verwendung von getypten Merkmalsstrukturen wie auch von leeren Restlisten zur Implementierung von Kohärenz und Vollständigkeit diskutiert.

$$
\text{(11a)} \begin{bmatrix} \text{phon} & \left\langle \text{der, mann} \,|\, \text{R} \right\rangle - \text{R} \\ \text{sem} & \begin{bmatrix} \text{pred} & \text{mann} \end{bmatrix} \end{bmatrix}
\quad
\text{(11b)} \begin{bmatrix} \text{phon} & \left\langle \text{der,mann,trinkt,mehrere,flaschen,whisky} \,|\, \text{R} \right\rangle - \text{R} \\ \text{sem} & \begin{bmatrix} \text{pred} & \text{trinken} \\ \text{arg1} & \text{mann} \\ \text{arg2} & \begin{bmatrix} \text{...} & \text{...} \end{bmatrix} \end{bmatrix} \end{bmatrix}
$$

Die Information in den linken Merkmalsstrukturen ist in bezug auf die vorher gegebene nicht *vollständig*, in den rechten ist sie nicht *kohärent*.

Wedekind (1988, S.733) definiert Kohärenz *(coherence)* und Vollständigkeit *(completeness)* in Bezug auf Generierung im Rahmen der LFG. Seine Definitionen sind jedoch allgemein genug, um auch auf andere Formalismen angewendet zu werden. Vor allem können sie sich in völlig paralleler Weise auf das Parsing-Problem beziehen. Eine abgeleitete Merkmalsstruktur M_{out} kann als akzeptable Antwort auf ein Parsing- bzw. Generierungs-Problem in Form einer Merkmalsstruktur M_{in} angesehen werden, wenn folgende beiden Bedingungen erfüllt sind:

- **Vollständigkeit** Die abgeleitete Merkmalsstruktur darf nicht weniger Informationen enthalten als die Input-Merkmalsstruktur, d.h. es darf zu keinem Informationsverlust kommen. Alle Constraints, die für M_{in} gelten, müssen auch für M_{out} gelten. Dies bedeutet, daß auch Koindexierungen erhalten bleiben müssen:
  $M_{in} \sqsubseteq M_{out}$.

- **Kohärenz** Es darf in der abgeleiteten Merkmalsstruktur keine zusätzliche Information hinzugefügt sein:
  $M_{out} \sqsubseteq M_{in}$.

Diese Bedingungen sind für die in diesem Abschnitt gezeigten Beispiele jedoch noch nicht angemessen. Wedekind bezieht sich nur auf die semantische Struktur im Falle der Generierung. Die hier vorgestellten Merkmalsstrukturen umfassen aber auch die phonologische Kette. Diese soll ja gerade der Merkmalsstruktur hinzugefügt werden. In jeder realistischen Anwendung wird zudem mindestens noch syntaktische Information hinzukommen. Auch dies soll durch die Forderung nach Kohärenz nicht verhindert werden.

Umgekehrt wird beim Parsing die gegebene phonologische Kette durch die semantische Struktur ergänzt. Vollständigkeit und vor allem Kohärenz kann also nur von bestimmten Merkmalswerten in der Merkmalsstruktur verlangt werden: beim Parsing von dem Wert des Merkmals für die phonologische Information; bei der Generierung von dem Merkmalswert für die semantische Struktur. Es wird also eine Definition benötigt, die sich einen bestimmten Pfad aus der Input-Merkmalsstruktur und der abgeleiteten Merkmalsstruktur herausgreift und überprüft, ob Vollständigkeit und Kohärenz für diese beiden Pfade gegeben sind.

p-Parsing Problem Van Noord (1993, S.56) führt den Begriff *p-Parsing Problem* ein, der Parsing und Generierung umfaßt. Vollständigkeit und Kohärenz werden sichergestellt, indem von einem Pfad p, der in der Input-Merkmalsstruktur und in der abgeleiteten Merkmalsstruktur vorkommt, verlangt wird, daß exakt der gleiche Constraint gilt. Mit 'Constraint' ist hier eine Pfadgleichung gemeint, die einem Merkmal am Ende eines Pfades einen Wert zuweist. Über alle anderen Pfade außer dem explizit genannten wird keine Aussage gemacht. Diese Forderung besagt letztendlich nichts anderes, als daß die Merkmalswerte des Pfades in beiden Merkmalsstrukturen identisch sind.

Da in dieser Arbeit keine Constraint-Sprache eingeführt wurde, die zur formalen Definition des p-Parsing Problems verwendet werden kann, greife ich auf eine vereinfachte Notation zurück. Auf den Wert des letzten Merkmals eines Pfades p in der Merkmalsstruktur M ausgehend vom Wurzelknoten R wird mit $val(p, R, M)$ referiert.

Definition p-Parsing Problem Das p-Parsing Problem besteht aus einer Input-Merkmalsstruktur M_{in} mit einem instantiierten Wert für das essentielle Merkmal EF als Ziel und einer Grammatik G.

Die Lösung des p-Parsing Problems ist die Menge der Merkmalsstrukturen $S_{G,ef}$, so daß gilt:

- $S_{G,ef} \subseteq S_G$. Die Lösungsmenge $S_{G,ef}$ ist eine Teilmenge der Menge der durch die Grammatik G lizensierten Merkmalsstrukturen S_G. Dies ist eine Vorbedingung für die nun folgenden Bedingungen.

- $\{M_{out} \in S_{G,ef} | M_{out} \sqcup M_{in}\}$. Für alle Merkmalsstrukturen M_{out} der Menge $S_{G,ef}$ gilt, daß sie kompatibel sind mit der Input-Merkmalsstruktur M_{in}. Kompatibilität wird im Rahmen von unifikationsbasierten Formalismen durch Unifizierbarkeit ausgedrückt.

- $\{M_{out} \in S_{G,ef} | val(ef, R_{out}, M_{out}) = val(ef, R_{in}, M_{in})\}$. Für alle Merkmalsstrukturen M_{out} der Menge $S_{G,ef}$ gilt, daß der Wert des Merkmals ef (Typ)-identisch ist mit dem Wert des Merkmals ef in M_{in}.

Das p-Parsing Problem ist lösbar, wenn die Menge $S_{G,ef}$ mindestens ein Element enthält.

Implementierung von Kohärenz und Vollständigkeit Die übliche Verwendung von leeren Restlisten beim Parsing von Sätzen ist ein Beispiel für die Implementierung von Kohärenz, z.B. im Rahmen von DCGs (s. Abschnitt 4.1.1). Der entscheidende Punkt ist, daß die Variable für die Restliste instantiiert ist, so daß keine weitere Instantiierung stattfinden kann. Ist dies nicht der Fall, besteht zudem die Gefahr, daß Terminationsprobleme auftreten, da immer längere Ketten hineinunifiziert werden können.

Kohärenz bei der Generierung kann in ähnlicher Weise gewährleistet werden: alle Variablen, die in der semantischen Repräsentation vorkommen, werden durch Atome ersetzt.

Dadurch wird die Instantiierung mit zusätzlicher Information verhindert. In Prolog kann dies z.B. durch Verwendung des Prädikats `numbervars/3` geschehen, das Variablen durch Atome ersetzt.

Das Problem liegt grundsätzlich darin, daß Variablen des Formalismus (der Programmiersprache) und Variablen der beschriebenen linguistischen Objekte nicht unterschieden werden. Durch eine Instantiierung der Variablen in der semantischen Struktur bzw. in der Restliste wird eine Vermischung der beiden Ebenen verhindert (Shieber et al. 1990, S.39).

Vollständigkeit kann überprüft werden, indem die in der Input-Merkmalsstruktur gegebene Information mit der der abgeleiteten Merkmalsstruktur verglichen wird. Da der abgeleiteten Merkmalsstruktur nicht mehr anzusehen ist, welche Information schon gegeben war und welche durch Regelanwendung hinzukommt, schlagen Shieber et al (1990, S.39) vor, eine Kopie der ursprünglichen Information zu erstellen. Diese Kopie muß die durch die Grammatik hinzugekommene Information subsumieren.[6]

Ein Test auf Vollständigkeit wird durch die Verwendung von getypten Merkmalsstrukturen erleichtert. In den Beispielen (9) und (11) wurden zu Demonstrationszwecken bewußt ungetypte Merkmalsstrukuren verwendet. Wenn jedoch bekannt ist, daß z.B. die semantische Repräsentation für *trinken* vom Typ *binary* ist, läßt sich leicht feststellen, wenn ein Merkmal fehlt oder der Wert nicht instantiiert ist. Die Merkmalsstruktur ist dann nicht Typen-aufgelöst.

Weiterhin wird Kohärenz dadurch sichergestellt, daß zu einer Typen-aufgelösten Merkmalsstruktur keine zusätzliche Information mehr hinzugefügt werden kann. Die Verwendung getypter Merkmalsstrukturen kann Kohärenz und Vollständigkeit jedoch nicht garantieren sondern nur erleichtern, da

- die Input-Merkmalsstruktur nicht notwenigerweise Typen-aufgelöst ist (die semantische Repräsentation kann z.B. eine Variable enthalten) und

- die Typen-Signatur von Merkmalsstrukturen für Listen, die z.B. die phonologische Kette repräsentieren, rekursiv definiert ist, d.h. daß es nicht für Listen verschiedener Länge jeweils einen eigenen Typ gibt.

Reversibilität Die Bezeichnung ‘Reversibilität’ wurde bisher eher intuitiv in dem Sinne gebraucht, daß eine einzige Grammatik sowohl für Parsing als auch für Generierung verwendet werden kann. In diesem Abschnitt soll eine präzisere Definition gegeben werden.

Eine (reversible) Grammatik G definiert die Beziehung zwischen sprachlicher Form, der phonologischen Kette $s_i \in S$, und ihrer Bedeutung, der semantischen Repräsentation

[6] Zur Erläuterung des Ansatzes in (Shieber et al. 1990) s. Abschnitt 3.2.1. Da durch die head-Relation die ursprüngliche Semantik top-down weitergegeben wird, ist dieser Vorschlag nicht so einfach zu realisieren wie es zunächst klingt: die Information der abgeleiteten Merkmalsstruktur stammt nicht nur aus der Grammatik. Deshalb wird eine Technik vorgeschlagen, zusätzlich zu der abgeleiteten Merkmalsstrukur festzuhalten, welche Information tatsächlich aus der Grammatik stammt.

$lf_i \in LF$ ('lf' steht für *logical form*). Es handelt sich um eine Relation R zwischen der Domäne der Strings und der der semantischen Repräsentationen: $R \subseteq S \times LF$.

Ein Parser ist ein Programm P_p, das die Relation R für einen gegebenen String berechnet: $\langle s_i, ?LF\rangle$. Entsprechend ist ein Generator ein Programm P_g, das die Relation R für eine gegebene logische Form berechnet: $\langle ?S, lf_i\rangle$. Ein Programm P ist *reversibel*, wenn es die Relation R in beide Richtungen berechnen kann.

Für einen gegebenen String kann es mehrere Analysen geben *(Ambiguität)* und für eine gegebene semantische Repräsentation kann es mehrere phonologische Realisierungen geben *(Paraphrase)*.[7] Da eine Grammatik mit endlichen rekursiven Beschreibungsmitteln eine potentiell unendliche Menge an sprachlichen Zeichen definiert, muß zwischen zwei Fällen unterschieden werden:

1. Es gibt (möglicherweise) aufzählbar unendlich viele Lösungen für die Berechnung der Relation P. Parsing bzw. Generierung ist *unendlich aufzählbar (infinitely enumerable)*. Ein Programm P, das die Relation R berechnet, terminiert nicht in jedem Fall.

2. Es gibt endlich viele Lösungen für die Berechnung der Relation *(endlich aufzählbar, finitely enumerable)*. Eine Relation ist *effektiv reversibel* (van Noord 1993, S.22), wenn sie in beiden Richtungen endlich aufzählbar ist. Dies bedeutet, daß
   - Parsing endlich aufzählbar ist und
   - Generierung endlich aufzählbar ist.

Eine Grammatik, die eine effektiv reversible Relation definiert, wird *inhärent reversibel* genannt (Dymetman 1994). Ein Programm, das diese Relation berechnet, zählt alle Lösungen auf und hält.

Uniformität der Implementierung Ein reversibles Programm ist *uniform* in dem Sinne, daß es eben nur ein einziges Programm ist, das die Relation in beide Richtungen berechnet. Für den Fall der Relation zwischen phonologischer Kette und semantischer Repräsentation, in welchem die Berechnung in die eine Richtung Parsing (mit dem Programm P_p) und in die andere Generierung (mit dem Programm P_g) genannt wird, ist also $P_p = P_g$.

Ist durch die Verwendung einer (inhärent) reversiblen Grammatik eine uniforme Implementierung garantiert? Der Interpreter für den Grammatikformalismus muß vollständig

[7]Bloomfield geht davon aus, daß es keine Synonyme gibt: *"If the forms are phonemically different, we suppose that their meanings are also different [...]"* (1933, S. 145). Zu der Diskussion der sprachwissenschaftlichen Frage, ob unterschiedliche phonologische Formen immer mit unterschiedlichen Bedeutungen verbunden sind, soll hier nicht Stellung genommen werden. Wenn man der Ansicht Bloomfields folgt, kann man annehmen, daß Paraphrasen deswegen möglich sind, weil die Merkmalsstrukturen (und die sie lizensierende Grammatik) nicht differenziert genug sind, um die Bedeutungsunterschiede der Formen zu erfassen.

sein, damit die Relation in jedem Fall in beide Richtungen berechnet werden kann (Dymetman 1994). Der Begriff 'Vollständigkeit eines Interpreters' ist nicht mit der oben eingeführten Definition von Vollständigkeit (von Parsing und Generierung) identisch. Gemeint ist ein Interpreter, der in der Lage ist, *alle* Lösungen zu einer gestellten Aufgabe zu finden. Der Standard-(Prolog)-Interpreter für Definite Klauseln ist beispielsweise nicht vollständig. Er terminiert unter bestimmten Umständen nicht, auch wenn die Grammatik in Form von Definiten Klauseln eine endlich aufzählbare Menge an Zeichen definiert.

Um unter Verwendung eines unvollständigen Interpreters doch zu den erwünschten Terminierungseigenschaften zu kommen, kann eine Kompilierung in spezialisierte Grammatiken erforderlich sein. Dadurch geht die Uniformität der Grammatik verloren. Bisher wurde vorausgesetzt, daß nur eine einzige Grammatik verwendet wird, so daß die Menge der durch die Grammatik definierten Zeichen für das Parsing gleich der für die Generierung ist.

Weiterhin ist nicht automatisch garantiert, daß durch den Kompilierungsschritt nicht einer spezialisierten Grammatik Information hinzugefügt wird oder verloren geht und in der anderen nicht. In diesem Fall besteht die Gefahr, daß die Menge der durch die beiden Grammatiken definierten Zeichen nicht mehr gleich ist.

Neumann (1994, S.70) bezeichnet Grammatiken, die erst kompiliert werden müssen, als *schwach reversibel* und solche, die direkt zur Laufzeit interpretiert werden, als *stark reversibel*.

Eine *uniforme Architektur* soll gekennzeichnet sein durch einen einzigen Interpreter und eine einzige Grammatik (Typ D nach dem in Kapitel 1.3 eingeführten Klassifikationsschema). Ihre Grammatik ist also stark reversibel.

Aus den hier vorgestellten Überlegungen geht hervor, daß

- der Interpreter vollständig sein muß und
- die Grammatik inhärent reversibel,

damit ein Programm, das ohne einen Kompilierungsschritt auskommt, in jedem Fall terminiert. Damit ist noch nicht gesichert, daß die Lösungssuche besonders effizient ist. Insofern stellt die hier verwendete Definition von Reversibilität nur eine notwendige Bedingung dar. Die besondere Schwierigkeit für den Interpreter besteht darin, sich den unterschiedlichen Aufgaben so anzupassen, daß die Lösungssuche zielgerichtet abläuft. Dies wird in Kapitel 3 behandelt. Zunächst sollen kurz die bekanntesten Analysestrategien vorgestellt werden.

2.3 Analysestrategien

Um eine potentiell unendliche Menge an sprachlichen Zeichen mit endlichen Mitteln zu analysieren oder zu generieren, werden rekursiv definierte Ableitungs- bzw. Produktionsregeln sowie lexikalische Einträge verwendet. Die von vielen Grammatikformalismen verwendeten Phrasenstrukturregeln beruhen auf einer kontextfreien Basis. Ihre Form entspricht Typ 2 der Chomsky-Hierarchie für formale Sprachen. Die in der Definition solcher Regeln verwendeten atomaren Symbole werden jedoch durch Merkmalsstrukturen ersetzt, was *Complex-Feature-Based Formalisms* ergibt.

Kontextfreie Phrasenstrukturregeln definieren *lokale Bäume*. Sie umfassen den Mutterknoten (linke Regelseite) und die von ihm unmittelbar dominierten Tochterknoten (rechte Regelseite). Anforderungen an die Merkmalsstrukturen der Knoten können als Pfadgleichungen angegeben werden (z.B. in PATR-II (Shieber 1986)). Diese Annotationen lassen sich als lokale Constraints interpretieren, die partielle Bedingungen an wohlgeformte Merkmalsstrukuren stellen.

Die erfolgreiche Analyse einer Phrase kann in Form eines Baumes dargestellt werden, dessen Knoten mit Merkmalsstrukturen beschriftet sind. Ein solcher Baum ist gewissermaßen zusammengesetzt aus lokalen Bäumen, wobei die Blätter lexikalischen Einträgen entspechen. Die unterschiedlichen Analysestrategien können als verschiedene Wege angesehen werden, diesen Baum abzuschreiten: *Parsing as tree traversal* (Gerdemann 1994).

Da sowohl syntaktische wie auch semantische Information in Merkmalsstrukturen kodiert wird und ihre Verknüpfung in denselben Regeln mit derselben Operation (Unifikation) stattfindet, entspricht die Menge der Analysebäume beim Parsing einer Phrase der der erzeugten Bäume bei der Generierung dieser Phrase. In bezug auf den Baum gibt es grundsätzlich keinen Unterschied zwischen Parsing und Generierung.

Ein Analyse- oder Produktionsbaum (im folgenden wird ‘Analysebaum’ für beide Begriffe verwendet) gibt alle Regelanwendungen an, die zu seiner erfolgreichen Konstruktion notwendig sind. Er gibt jedoch nicht die erfolglosen Versuche an, diesen Baum zu entdecken.[8] Sie können durch einen Suchbaum dargestellt werden, wobei eine geglückte Analyse durch einen Teilbaum des Suchbaumes repräsentiert wird. Nur bei einer optimalen Suchstrategie würde der Suchbaum alleine die Analyse umfassen. In der Praxis gibt es jedoch fast immer eine Differenz zwischen beiden. Wenn man das ganze Suchverfahren betrachtet, bestehen Parsing und Generierung also in der *Entdeckung eines Analysebaumes.*

Während die Analysebäume bei Parsing und Generierung derselben Phrase zusammenfallen, ist dies für die Suchbäume nicht notwendigerweise der Fall. Aufgrund der unterschiedlichen Ausgangsinformation wird die Suche verschiedene Wege einschlagen. Weil jedoch früher oder später inkonsistente Information auftritt, scheiden alle nicht zur korrekten Analyse gehörenden Teilbäume schließlich aus.

2.3.1 Traversierung von Bäumen

Drei der bekanntesten Analysestrategien lassen sich jeweils einer bestimmten Reihenfolge, einen Baum zu traversieren, zuordnen ((Johnson 1995), (Gerdemann 1994)). Der aktuelle

[8]Dies gilt entsprechend auch für die Repräsentation des Analysebaumes in einer Merkmalsstruktur (s. Abschnitt 2.2.1).

Baum wird durch die Traversierung eines virtuellen Baumes konstruiert (Shieber et al. 1990, S.34). Hier sollen keine exakten Algorithmen angegeben werden, sondern der Bezug zur Baumtraversierung herausgestellt und einige Eigenschaften der Strategien genannt werden. Die Erläuterungen beziehen sich auf das Parsing, da dies die verbreitetste Anwendung der hier genannten Analysestrategien ist.

	Top-down	**Left-corner**	**Bottom-up**
Traversierung	Pre-order	In-order	Post-order
Reihenfolge	Mutterknoten erste Tochter zweite Tochter	erste Tochter Mutterknoten zweite Tochter	erste Tochter zweite Tochter Mutterknoten

Abbildung 2.5: Drei Arten der Traversierung binärer Bäume nach Johnson (1995)

Top-down-Analyse Zuerst wird der Mutterknoten konstruiert und anschließend die Tochterknoten, die zumeist in der Reihenfolge von links nach rechts abgearbeitet werden. Dies entspricht der Abarbeitung des Eingabestrings in der gleichen Richtung. Es handelt sich um eine *pre-order* Traversierung, weil die linke Regelseite vor der Folge von Elementen auf der rechten Regelseite untersucht wird. Wenn die rechte Seite kein lexikalischer Eintrag ist, müssen ihre Elemente als Mutterknoten in weiteren Regelanwendungen dienen.

Die top-down-Analyse ist sehr effizient, weil die nächste Regel sehr schnell gefunden werden kann. Die linke Seite der gesuchten Regel muß mit dem gerade aktuellen Element (auf rechten Seite der Regel, die bearbeitet wird) kompatibel sein. Diese Zielgerichtetheit wird dadurch verstärkt, daß Information top-down weitergegeben wird, was durch Koindizierung von Information auf auf der linken und auf der rechten Regelseite erreicht wird. Um die Zielgerichtetheit der top-down-Analyse von Anfang an auszunutzen, ist es sinnvoll, die Zielkategorie zu Beginn anzugeben.[9]

Bottom-up-Analyse Zuerst wird die rechte Regelseite untersucht und die Tochterknoten konstruiert. Wenn alle Tochterknoten einer Regel vorhanden sind, wird zur linken Regelseite übergegangen *(post-order traversal)*. Der Mutterknoten kann ein Element auf der rechten Seite einer Regel sein, so daß der Prozeß rekursiv weitergeht. Wenn beim Parsing die Eingabekette unter Verwendung aller Tochterknoten der betroffenen Regeln abgearbeitet ist, ist das Ergebnis der Analyse der 'am höchsten stehende' Mutterknoten.

Ein Vorteil der bottom-up-Analyse ist, daß das Ziel der Analyse bei Beginn nicht schon feststehen muß. Sie ist Daten-gesteuert in dem Sinne, daß (beim Parsing) das nächste Element der phonologischen Kette bestimmt, welcher Schritt als nächster erfolgt. Ande-

[9]Kategoriale Information ist zwar nur noch ein Merkmal unter anderen, aber sie stellt gewissermaßen die 'kontextfreie Basis' von Grammatikregeln da. Wenn z.B. *s* als Zielkategorie zu Beginn der Analyse angegeben wird, gibt es zumeist nur noch sehr wenige Regeln, die einen passenden Mutterknoten aufweisen. Wenn die Zielkategorie dagegen unbekannt ist, müssen zunächst nichtdeterministisch alle Regeln getestet werden.

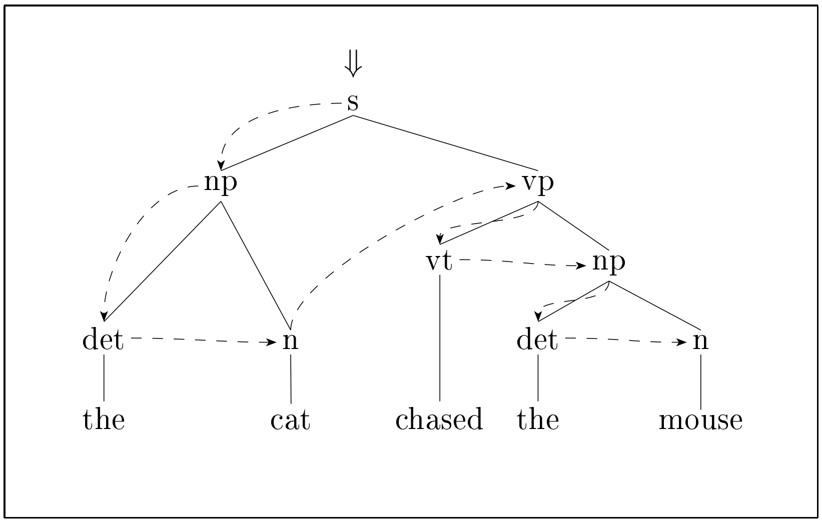


Abbildung 2.6: top-down-Analyse

rerseits ist die Suche nach passenden Regeln ungerichtet, weil die konstruierten Teilbäume solange unverbunden nebeneinander stehen, bis sie zusammen auf einer rechten Regelseite verwendet werden. Erst dann können Inkonsistenzen entdeckt werden, z.B. wenn Kongruenz-Bedingungen verletzt sind.

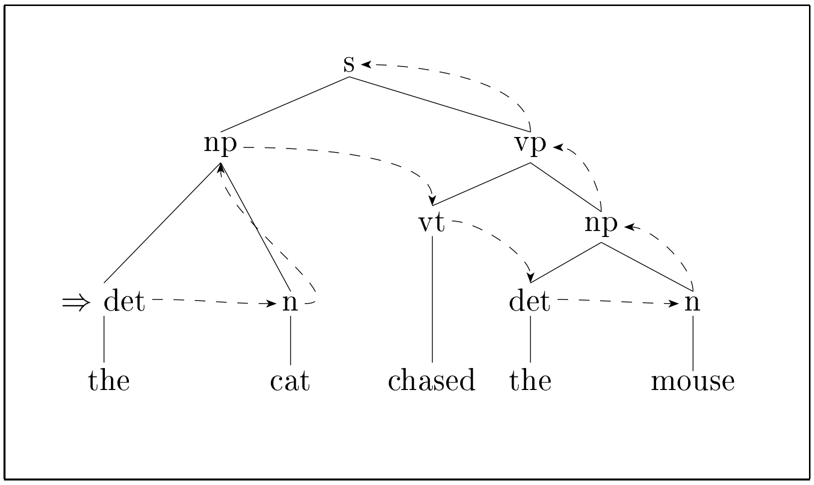


Abbildung 2.7: bottom-up-Analyse

Left-corner-Analyse Eine Kombination der beiden erstgenannten Analyseformen stellt die left-corner-Analyse dar, die meines Wissens nur beim Parsing verwendet wird (in Abschnitt 6.1.1 wird eine mögliche Anwendung auf die Generierung angesprochen). Sie beginnt bottom-up mit der Analyse des ersten Wortes der Eingabekette. Nun wird eine Regel gesucht, die dieses Element als seine *linke Ecke* aufweist, d.h. es muß das erste Element der rechten Regelseite sein. Auf diese Weise kann die passende Regel sehr effizient gefunden werden. Die verbleibenden Tochterknoten rechts der linken Ecke werden nun top-down abgearbeitet. Ist dies erfolgreich, kann zur linken Regelseite übergegangen werden. Sie wird wiederum als linke Ecke in einer Regel gesucht.

Es können verschiedene Left-corner-Parsing-Strategien unterschieden werden, je nachdem an welchem Zeitpunkt die Anbindung einer linken Regelseite als linke Ecke an den 'nächsthöheren' lokalen Baum erfolgt (Abney und Johnson (1991)). Die bekannteste Strategie ist das *Arc Standard Left Corner*-Parsing *(ASLC)*. Ein mit dieser Strategie arbeitender Parser wartet ab, bis alle Elemente der rechten Regelseite vorhanden sind und sucht dann eine linke Ecke in einer Grammatikregel, die mit der linken Seite der bearbeiteten Regel unifiziert. Ein *Arc Eager Left Corner*-Parser *(AELC)* dagegen sucht die 'nächsthöhere' Grammatikregel sofort, wenn durch die linke Ecke eine Grammatikregel feststeht. Es ist offensichtlich, daß sich diese Anbindung als unnötig herausstellen kann, wenn es nicht gelingt, die fehlenden Elemente der rechten Seite der schon angebundenen Regel zu konstruieren. Auch für die ASLC-Strategie gilt aber im Gegensatz zur bottom-up-Verarbeitung, daß Grammatikregeln anhand ihrer linken Ecke ausgewählt werden, ohne daß feststeht, ob alle fehlenden Elemente bearbeitet werden können (Gerdemann und Hinrichs (1995, S. 174)), d.h. left-corner-Strategien können allgemein als 'voreilig' bezeichnet werden. Zwischen den Extrempunkten ASLC und AELC lassen sich darüberhinaus Strategien definieren, die eine Anbindung von der Existenz eines bestimmten Elements der rechten Regelseite erlauben (Mahesh 1994).

Die Effizienz der left-corner-Analyse kann durch eine Tabelle gesteigert werden, die feststellt, welche linke Ecke zu welchem Mutterknoten führt *(Linking-Relation)*. Da der Mutterknoten eine linke Ecke in anderen Regeln sein kann, ist diese Relation transitiv. Durch Berechnung der transitiven Hülle der Linking-Relation kann vor einer Regelanwendung schon getestet werden, ob das gesuchte Ziel mit ihr zu erreichen ist *(reachability)*. Neben ihrer Funktion als Test kann die Linking-Relation zum Informationstransport dienen (Kilbury 1996) und den Suchraum einschränken. Das erste Ziel der Analyse muß wie in der top-down-Analyse bekannt sein, damit die Berechnung der Linking-Relation einen Sinn macht.

Bei Betrachtung binärer lokaler Bäume ist die Traversierung *in-order*. Nachdem zuerst die linke Ecke gefunden ist, wird der Mutterknoten und dann die rechte Ecke konstruiert. Diese Sichtweise kann zunächst etwas befremdlich erscheinen, da doch alle Knoten der rechten Seite zuerst konstruiert werden: die left-corner-Analyse hat starke bottom-up-Aspekte.[10] Man kann aber sagen, daß der Mutterknoten vor der rechten Ecke entdeckt wird, weil er durch die Wahl der Regel, die die gesuchte linke Ecke aufweist, bekannt ist und mit ihr in der Linking-Relation steht.

Bisher wurde immer von binären Bäumen ausgegangen (entsprechend der Chomsky-Normalform für kontextfreie Grammatiken mit Nicht-Terminalsymbolen auf der rechten Regelseite). Bei top-down- und bottom-up-Analyse gibt es, auch wenn mehr Elemente auf

[10] Pereira und Shieber (1987) behandeln die left-corner-Analyse als eine Realisierung von bottom-up-Parsing; Gerdemann (1994) zeigt, daß beide Strategien durch bestimmte Transformationen zusammenfallen.

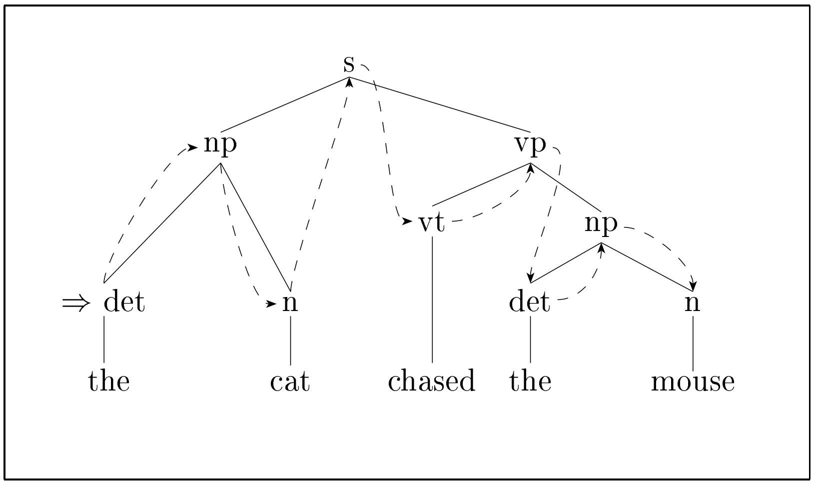


Abbildung 2.8: left-corner-Analyse

der rechten Seite auftreten, offensichtlich keine Schwierigkeiten: der Mutterknoten wird entweder vor oder nach *allen* Tochterknoten konstruiert. Bei der left-corner-Analyse gibt es dagegen mehrere Positionen, die in diesem Fall ‘in-order’ sind. Auch hier läßt sich jedoch ein Verfahren angeben: wenn die rechte Seite mehr als zwei Elemente enthält, wird rekursiv der verbleibende Rest in-order untersucht, d.h. das erste Element des Rests dient als linke Ecke usw. (Gerdemann 1994).

2.3.2 Termination

Ein wesentliches Kriterium zur Beurteilung von Analysestrategien sind ihre Terminationseigenschaften. Wie bereits erwähnt ist der Standard-Prolog-Interpreter z.B. unvollständig, weil er in dem Fall, daß die Analyse in eine Endlosschleife gerät, nicht alle Ergebnisse liefern kann. Die im folgenden genannten Terminationsprobleme der verschiedenen Analysearten sind jedoch nicht an eine bestimmte Suchstrategie gebunden.

Top-down-Analyse Problematisch für die top-down-Analyse sind linksrekursive Regeln (d.h. Regeln der Form $X \rightarrow X\ Y$) bzw. indirekte Linksrekursion (z.B. $X \rightarrow Y\ V; Y \rightarrow X\ W$). Nachdem der Mutterknoten einer linksrekursiven Regel gefunden ist, wird das am weitesten links stehende Element der rechten Seite gewählt. Zu diesem Element wird ein passender Mutterknoten gesucht: die linke Regelseite. Dieser Prozeß kann unendlich oft wiederholt werden. Entsprechend enthält der Suchbaum Äste unendlicher Länge.

Das Problem kann nicht dadurch gelöst werden, daß ein anderes Element der rechten Seite gewählt wird, z.B. das am weitesten rechts stehende. In diesem Fall tritt Rechtrekursion bei Regeln der Form $X \rightarrow YX$ auf.

Es ist jedoch möglich, das Problem dadurch zu vermeiden, daß nicht in jedem Fall das erste Element der rechten Seite sofort gewählt wird. Dies kann durch eine geeignete

Umstellung der Regel *(node reordering)* oder durch eine Verzögerung der Bearbeitung des 'gefährlichen' Elements *(goal freezing)* geschehen. Eine Umstellung der Reihenfolge darf natürlich die Grammatik nicht verändern. Es muß eine geeignete Transformation der ganzen Grammatik geschehen, damit die Menge der von der Grammatik definierten Zeichen gleich bleibt. Dies bedeutet eine Veränderung der Grammatik zur Compile-Zeit (Dymetman et al. 1990), die einer Architektur des Typs D, wie sie in dieser Arbeit vor allem berücksichtigt wird, widerspricht.

Ein Problem, das durch Linksrekursion bei der top-down-Analyse entsteht, läßt sich jedoch durch diese Techniken nicht lösen. Es entsteht durch die Verwendung von Merkmalsstrukturen anstelle von atomaren Symbolen in den Syntaxregeln. Es sei hier auf die ausführlichere Diskussion der Problematik in Abschnitt 2.3.3 verwiesen.

Bottom-up-Analyse Linksrekursion stellt für die bottom-up-Analyse kein Problem dar. Eine Regel wird nur dann gewählt, wenn die jeweils zu verarbeitende Informationsmenge es erlaubt, d.h. wenn z.B. beim Parsing das aktuelle Wort die Wahl eines lexikalischen Eintrags rechtfertigt oder, 'höher im Baum', wenn alle Tocherknoten einer Regel konstruiert sind, so daß diese zur linken Seite reduziert werden kann. Es findet keine Vorhersage *(prediction)* der nächsten Regel statt, wie bei der top-down-Analyse, die den gegebenen Input zunächst nicht berücksichtigen kann.

Dennoch gibt es ein Problem für die bottom-up-Analyse: leere rechte Seiten *(Nullproduktionen)* der Form $X \rightarrow \epsilon$ (vgl. z.B. Covington (1994)). Leere Konstituenten können in beliebiger Menge an jeder Position im String angenommen werden. Eine Analyse, die – z.B. aufgrund der Eingabe eines inkorrekten Satzes – eigentlich scheitern müßte, kann ad infinitum weitergeführt werden, ohne daß alle Analyse-Möglichkeiten erschöpft sind.

Dieses Problem kann durch Umwandlung der Grammatik in eine gleichen Sprachumfangs ohne leere rechte Seiten gelöst werden. Allerdings sind die Analysebäume nicht notwendigerweise identisch (Gazdar und Mellish 1989). Auch hier ist also ein Kompilierungsschritt notwendig.

Für die top-down-Analyse sind Nullproduktionen dagegen kein Problem, weil eine solche Regel nur einmal angewendet wird, wenn der Mutterknoten vorher auf der rechten Seite einer anderen Regel aufgetreten ist. Die leere Produktion wird ausdrücklich aufgerufen (dies bewirkt die Zielgerichtetheit der top-down-Analyse) und dann nur einmal (erfolgreich) abgearbeitet. Sie bietet keine Möglichkeit, unendlich oft wiederholt zu werden, da sie nicht an jeder Stelle ausprobiert werden kann.

Left-corner-Analyse Als eine Mischung aus den beiden vorher genannten Strategien besteht die left-corner-Analyse wie oben beschrieben aus einem bottom-up-Teil, der die linke Ecke einer Regel 'von unten' entdeckt, und einem top-down-Teil für den Rest der Tochterknoten. Da Linksrekursion nur für die top-down-Analyse ein Problem darstellt und gerade die linke Ecke bottom-up entdeckt wird, gibt es in dieser Hinsicht kein Problem.

Anders verhält es sich dagegen mit Nullproduktionen und dem bottom-up-Teil der left-corner-Analyse. Wenn die linke Ecke die Mutterkategorie einer Nullproduktion ist (z.B.: $NP \rightarrow Det\ N; Det \rightarrow \epsilon$) kann eine Anwendung unendlich oft versucht werden – und zwar wiederum in dem Fall, in dem die Analyse eigentlich scheitern sollte.

Die schon erwähnte Linking-Relation löst dieses Problem teilweise. Ein leeres Element kann nur angenommen werden, wenn es zur Erreichnung des Ziels beitragen kann. Um im Beispiel zu bleiben: es ist nun nicht mehr möglich, immer weitere 'leere' Determiner anzunehmen, wenn z.B. gerade eine VP analysiert wird und dies scheitert. Der Grund liegt darin, daß der Determiner – eine entsprechende Grammatik vorausgesetzt – keine (transitive) linke Ecke der VP ist (Covington 1994).

Durch die Link-Tabelle wird das Problem der Linksrekursion in left-corner-Parsern jedoch nicht grundsätzlich gelöst. Wenn von einer leeren linken Ecke aus bottom-up geparst wird und das folgende Parse-Ziel wiederum eine leere linke Ecke ist, wird dies als *versteckte Linksrekursion (hidden left-recursion)* bezeichnet: "[...] it may be possible that an empty category is predicted as the head, after which trying to construct a larger projection of the head gives rise to a parse goal for which a similar empty category is a possible candidate head ..." (van Noord 1997, Abschnitt 2.3).[11] Obwohl diese Formulierung den Eindruck erweckt, daß die Grammatik selbst eine unendliche Menge an Ableitungsschritten definiert, kann das Terminationsproblem auch dann auftreten, wenn die Grammatik nur eine endliche Menge von Wortketten lizensiert (vgl. ebda; in Anhang A.5 wird ein Beispiel für ein Terminations-Problem im Zusammenhang mit left-corner-Parsing gegeben).

2.3.3 Linksrekursion und Restriktion

In vorhergehenden Abschnitten wurden die grundlegenden Formen der Baum-Traversierung vorgestellt, so daß diese nun in bezug auf das Linksrekursions-Problem genauer untersucht werden können. Da alle Vorschläge zu Parsing bzw. Generierung sich mit möglichen Terminationsproblemen auseinandersetzen müssen, sind die unterschiedlichen Antworten auf die Problematik ein guter Vergleichspunkt für die in den nächtsten beiden Kapiteln dargestellten Ansätze.

Die Bezeichnung 'Linksrekursion' wird hier als Oberbegriff für verschiedene Formen von Rekursion verwendet, bei denen das Element der linken Regelseite auf der rechten Regelseite 'wiederholt' wird. Das Problem ist nicht auf das am weitesten links stehende Element der rechten Regelseite beschränkt – es kann mit jedem Element auftreten, das von der Selektionsfunktion gewählt wird und ist deshalb relativ zu dem verwendeten Interpreter.

Die VP-Komplement-Regel Das Problem der Linksrekursion wird zumeist auf die top-down-Analyse bezogen. Grundsätzlich kann es aber auch bei einer bottom-up-Analyse auftreten, wenn es keine Struktur wie eine instantiierte Subkategorisierungs-Liste gibt, die

[11] Dies bezieht sich sowohl auf left-corner- wie auch auf head-corner-Parser (s. Abschnitt 3.2.5). Im Kontext von left-corner-Parsing muß der Begriff 'Kopf' gegen 'linke Ecke' getauscht werden.

schrittweise verringert wird, so daß schließlich eine Abbruchbedingung erreicht ist und die Wahl einer anderen Regel erzwungen wird. Anhand einer linkrekursiven VP-Komplement-Regel ähnlich der in Shieber (1986, S. 32) soll dies im folgenden erläutert werden (zur Erläuterung der Notation für die VP-Regel und die dazugehörige S-Regel s. Abschnitt 4.1.1):

$$(12)\quad \left[\text{syn}\ \begin{bmatrix}\text{cat} & \text{s}\\ \text{subcat} & \boxed{1}\langle\ \rangle\end{bmatrix}\right] \longleftarrow \boxed{2} \wedge \left[\text{syn}\ \begin{bmatrix}\text{cat} & \text{vp}\\ \text{subcat} & \langle\boxed{2} \mid \boxed{1}\rangle\end{bmatrix}\right]$$

$$(13)\quad \left[\text{syn}\ \begin{bmatrix}\text{cat} & \text{vp}\\ \text{subcat} & \boxed{1}\end{bmatrix}\right] \longleftarrow \left[\text{syn}\ \begin{bmatrix}\text{cat} & \text{vp}\\ \text{subcat} & \langle\boxed{2} \mid \boxed{1}\rangle\end{bmatrix}\right] \wedge \boxed{2}$$

Die VP-Regel in (13) legt fest, daß eine VP zu einer VP und dem ersten von ihr in der Subkategorisierungs-Liste vorgesehenen Element expandieren kann. Wenn die Subkategorisierungs-Information im Lexikoneintrag des Verbs repräsentiert wird, sind die Elemente der Liste bei der top-down-Analyse zunächst nicht instantiiert. Die VP-Regel kann wiederholt angewendet werden, so daß eine immer längere Subkategorisierungs-Liste entsteht. Bei jedem Schritt wird der schon vorhandenen Liste ein weiteres (nicht instantiiertes) Element vorangestellt.

Das resultierende Terminationsproblem bei der top-down-Analyse ist durch Techniken wie die Verzögerung der Bearbeitung eines bestimmten Elements nicht zu lösen. Eine solche Verzögerung hat das Ziel abzuwarten, bis die 'gefährliche' Variable instantiiert ist. Da die Instantiierung der Merkmale für das Komplement von der Information im Verbeintrag abhängt, wird jede Verzögerungs- oder Neuordnungs-Methode die VP der rechten Regelseite zuerst wählen (vgl. Shieber *et al.* 1990, S. 32). Aber auch wenn das Komplement vor dem Verb bearbeitet wird, kann eine Rekursions-Schleife nicht verhindert werden. Es sind stets weitere top-down-Vorhersagen mit Regel (13) möglich, auch wenn das Element, das der Subkategorisierungs-Liste zusätzlich vorangestellt wird, instantiiert ist. Dies bedeutet also, daß unabhängig von der Reihenfolge, in der die Elemente der rechten Regelseite von (13) bearbeitet werden, Terminationsprobleme auftreten.

Die beiden Regeln zeigen, daß die Subkategorisierungs-Liste auf der linken Regelseite kürzer ist als auf der rechten. Wenn bottom-up vorgegangen wird *und* die Subkategorisierungs-Information im Lexikoneintrag des Verbs abgelegt wird, ist diese Liste instantiiert und wird mit jeder Anwendung der VP-Regel um ein Element kleiner. Dadurch wird schließlich als Abbruchbedingung der rekursiven Bearbeitung der Subkategorisierungs-Liste die S-Regel (12) erreicht. Bei der top-down-Analyse ist die Subkategorisierungs-Liste dagegen nicht instantiiert. Zudem wird sie mit jedem Schritt größer, woraus sich weitere Probleme ergeben (s.u.). Im folgenden werden zwei Vorschläge dargestellt, die trotz dieses Problems eine Verwendung der VP-Komplement-Regel bei der top-down-Analyse ermöglichen.

Restriktion Eine besondere Schwierigkeit für die top-down-Analyse liegt darin, daß bei der Verwendung der VP-Komplement-Regel auch die Speicherung schon bearbeiteter Teilaufgaben (s. Abschnitt 4.2) den Prozessor nicht davor bewahren kann, in eine Endlos-Schleife zu geraten. Eine solche Technik hat das Ziel, schon bearbeitete Aufgaben (und bei Erfolg auch deren Lösung) zu speichern, um sie nicht mehrfach ausführen zu müssen. Die Verwendung von Merkmalsstrukturen anstelle von atomaren Bezeichnern zur Beschriftung der Knoten in Phrasen-Strukturregeln macht es erforderlich, eine genauere Definition des Falls vorzunehmen, in dem eine Aufgabe nicht noch einmal bearbeitet werden soll (es sei darauf hingewiesen, daß das in diesem Abschnitt erläuterte Problem aus dem gleichen Grund entsteht). Eine Aufgabe wird dann nicht noch einmal bearbeitet, wenn sie von einer schon gespeicherten *subsumiert* wird, d.h. daß die früher bearbeitete Aufgabe eine Generalisierung der jeweils aktuellen darstellt. Der entscheidende Punkt ist nun, daß bei der rekusiven Verwendung der VP-Regel in (13) die VP auf der rechten Regelseite, die es als nächstes zu bearbeiten gilt, nie von den vorhergehenden VPs subsumiert wird, da ihre Subkategorisierungs-Liste stets um mindestens ein Element größer ist.

Zur Lösung dieses Problems schlägt Shieber (1985) vor, durch *Restriktion* zu verhindern, daß unendlich viele verschiedene Merkmalsstrukuren durch die Verwendung der Grammatikregeln aufgebaut werden. Ein *Restriktor* beschränkt eine Merkmalsstruktur M zu einer Merkmalsstruktur M_{restr}, indem nur noch die Pfade auf nicht-atomare Werte abgebildet werden, die ausdrücklich erlaubt sind (Shieber 1985, S. 148) – bei atomaren oder undefinierten Werten können ja keine unendlichen Strukturen entstehen. Die potentiell unendlich große Menge an Merkmalsstrukturen wird so auf eine endlich große Menge von *Äquivalenzklassen* beschränkt.

Mit dieser Methode kann beispielsweise der Pfad SYN SUBCAT eliminiert werden. Der Restriktor wird angewendet, *bevor* die top-down-Vorhersage stattfindet. Nun kann schon bei der zweiten Verwendung der VP-Komplement-Regel festgestellt werden, daß eine Subsumptions-Beziehung vorliegt. Dadurch werden weitere Vorhersagen blockiert.

Auf diese Weise geht Information verloren, wodurch es zu zusätzlichen unnötigen Vorhersagen kommen kann. Die Zielgerichtetheit der top-down-Analyse wird also eingeschränkt, allerdings mit dem Gewinn, daß das Terminationsproblem gelöst ist. Die Korrektheit des Algorithmus bleibt bei den verschiedenen Lösungsvorschlägen trotz möglicher falscher Vorhersagen erhalten, weil die Unifikation dann an einem späteren Zeitpunkt aufgrund inkonsistenter Information scheitert.

Statt der vollständigen Eliminierung des Pfades SYN SUBCAT ist auch eine Beschränkung der Listenlänge auf eine Länge von maximal drei Elementen denkbar. Wenn nun die VP-Regel rekursiv angewendet wird, tritt auch bei dieser Lösung sehr schnell der Fall auf, daß eine VP mit einer Subkategorisierungs-Liste der Länge drei bearbeitet werden soll, die schon gespeichert ist, wodurch auch in diesem Fall die Bearbeitung blockiert wird.

Dieses Verfahren macht zunächst einen etwas unbefriedigenden Eindruck, denn schließlich wird de facto die Rekursionstiefe von Grammatikregeln beschränkt. Shieber *et al.* (1990,

S. 32) halten es deshalb für eine 'ad-hoc-Lösung'.[12] Sie wenden vor allem ein, daß das Problem durch die Beschränkung der Listen-Länge nicht grundsätzlich gelöst ist, weil es z.B. für das Niederländische Analysen gibt, in denen die Subkategorisierungs-Listen mehrerer Verben verkettet werden. Eine Grammatik mit feststehender Listen-Länge ist für solche Analysen ungeeignet. Es scheint also günstiger zu sein, den entsprechenden Pfad vollständig zu beseitigen (in Abschnitt 4.2.3 wird im Kontext von Chart-Parsing erläutert, wie eine Analyse unter der Verwendung von Restriktion arbeitet).

'Restriktion' bezeichnet also eine allgemeine Methode zur Beschränkung von Merkmalsstrukturen, die im Laufe einer Analyse erzeugt werden. Es gibt eine Reihe weiterer Anwendungen, die hier nicht genannt wurden, da sie das Terminationsproblem von top-down-Strategien nicht direkt betreffen (s. Abschnitte 4.2.3 und 4.3.2). Grundsätzlich muß aber bemerkt werden, daß es sich um eine nicht-monotone Operation der Löschung von Information handelt, die in den 'monotonen Lauf der Unifikation' eingreift. Sie sollte also sehr vorsichtig eingesetzt werden.

Guides Dymetman *et al.* (1990, S. 93) führen den Begriff 'Guide' ein, der eine generellere Sicht auf das Problem der Linksrekursion ermöglicht. Ein *Guide* ist die Struktur, die die Verarbeitung steuert, also beim Parsing die Wortkette und bei der Generierung die semantische Repräsentation. Es wird verlangt, daß der Guide im Laufe der Verarbeitung schrittweise verbraucht wird: bei jedem lexikalischen Zugriff wird ein Teil des Guides gelöscht *(guide-consumption condition)*. Wenn ein 'normaler' Parser die Wortkette von links nach rechts abarbeitet, verwendet er sie also als Guide im Sinne von Dymetman *et al.*

Diese Überlegung kann zur Beantwortung der Frage beitragen, warum ein Rekursionsproblem nur dann auftritt, wenn das *erste* bearbeitete Element der rechten Regelseite gleich dem Element der linken Regelseite ist. Bei einer Regel $A \rightarrow$ B A entsteht dagegen kein Terminationsproblem, wenn das am weitesten links stehende Element gewählt wird. Der Grund dafür liegt (beim Parsing) in der Reduzierung des Strings bei der Bearbeitung des Elements B. Dies bedeutet, daß das Element A auf der linken Seite normalerweise nicht genau gleich dem Element A auf der rechten Seite ist. Wenn dagegen B eine leere Konstituente analysiert und die Wortkette unverändert läßt, tritt ein Terminationsproblem auf.

Um dennoch den Standard-Prolog-Interpreter, also eine top-down- und left-to-right-Strategie verwenden zu können, schlagen Dymetman *et al.* (1990) eine Kompilation der Grammatik vor. Durch eine Transformation der Grammatikregeln wird für Parsing und Generierung je eine spezialisierte nicht-linksrekursive Grammatik erzeugt.[13] Die Wortkette

[12] Es gibt dort erstaunlicherweise keinen Hinweis auf 'Restriktion' im Sinne von Shieber (1985), obwohl er einer der Co-Autoren des Artikels ist.

[13] Die nun folgenden Erläuterungen nehmen eine entsprechende Anwendung ihrer Prinzipien zur Laufzeit an, wie sie beispielsweise der Analyse-Baum in Dymetman *et al.* (1990, S. 94) nahelegt. Dymetman *et al.* geben jedoch sehr abstrakte Erläuterungen der Transformationen der Grammatik an, so daß diese Ausführungen nur eine Annäherung darstellen.

wird beispielsweise zur Bearbeitung der VP-Komplement-Regel auf die beiden Elemente der rechten Regelseite aufgeteilt. Dies kann nur nicht-deterministisch geschehen, da im voraus nicht bekannt ist, welcher Teil der Wortkette durch welches Element der rechten Regelseite 'verbraucht' wird.

Wenn beispielsweise die VP *gibt maria den ball* analysiert werden soll, dann ist die gesuchte Aufteilung *gibt maria* für die VP der rechten Regelseite und *den ball* für das Komplement. Die VP kann wiederum (nun deterministisch) in *gibt* und *maria* aufgeteilt werden, so daß eine zweite Anwendung der VP-Komplement-Regel gerechtfertigt ist. Eine weitere Anwendung dieser Regel ist jedoch nun nicht mehr möglich. Es bleibt nur die Wahl einer anderen Grammatik-Regel mit einem einzigen Element auf der rechten Regelseite bzw. gleich eines lexikalischen Eintrags, der den Guide *maria* 'verbraucht'.

Es ist offensichtlich, daß die 'guide-consumption condition' grundsätzlich keine Grammatikregeln mit leeren rechten Regelseiten erlaubt (Dymetman *et al.* (1990, S. 95) diskutieren allerdings die Möglichkeit entsprechender Erweiterungen). Die von Dymetman *et al.* vorgeschlagene Methode erlaubt es also, effiziente top-down-Interpreter für linksrekursive Grammatiken ohne Restriktion zu verwenden. Die Abarbeitung der Wortkette erfolgt von links nach rechts. Trotz dieser Vorteile soll der Vorschlag hier jedoch nicht weiter diskutiert werden, da er eine substantielle Veränderung der Grammatik zur Compile-Zeit bedeutet. Dies widerspricht der Definition von einheitlichen Architekturen des Typs D (s. Abschnitt 1.3).

2.3.4 Abstrakter Suchraum

Die Suche nach einem Analysebaum findet in einem Suchraum statt. Johnson (1995) definiert einen abstrakten Suchraum, der die *Informationszustände* enthält, die bei der Suche nach einer erfolgreichen Analyse eingenommen werden. In dieser Darstellung stelle ich den von Johnson definierten Suchbaum innerhalb des Suchraums 'auf den Kopf', so daß die Wurzel des Suchbaums wie die von Analysebäumen oben ist.

Ein *Suchraum* ist eine partielle Ordnung $\Im = \langle I, < \rangle$, wobei I die Menge der Informationszustände ist und $<$ die Relation zwischen Objekten dieser Menge bezeichnet. $O \subseteq I$ bezeichnet die Menge von Objekten, die gesucht werden soll. Eine *Suchstrategie* ist eine Funktion s mit der Domäne O, so daß es für jedes gesuchte Objekt $o \in O$ eine Sequenz von Informationszuständen gibt, die von dem Ausgangszustand $\top$ zu diesem Objekt führt.

Die Menge der Informationszustände, die von dem Ausgangszustand aus erreicht werden können, soll als C bezeichnet werden. Zwei Informationszustände $\imath$ und $\imath'$ aus C stehen dann in der Relation $<$, wenn gilt: $\imath < \imath' \rightarrow (\imath' \in C \rightarrow \imath \in C)$.

Wenn ein Zustand $\imath'$ Element von C ist, dann ist es auch ein Zustand $\imath$, der auf dem Weg zu diesem Zustand liegt: $\imath'$ ist in diesem Sinne eine Extension von $\imath$. Von einem Zustand in C kann man immer aufwärts zu dem Ausgangszustand gelangen. Die Relation ist aufwärts monoton.

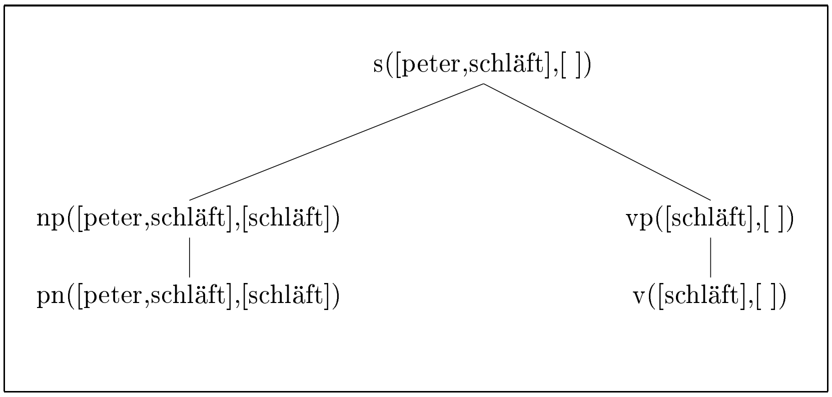


Abbildung 2.9: Analysebaum für *peter schläft* in DCG-Notation

Ein *Suchbaum* ist ein gerichteter Graph, der die Vorgehensweise einer Suchstrategie dokumentiert. Seine Knoten sind die Informationszustände, die die Suchstrategie im Laufe der Suche hervorgebracht hat. Der Ausgangszustand ist wiederum $\top$. Es gibt eine Kante von $\imath$ zu $\imath^{'}$, wenn $\imath < \imath^{'}$ und diese beiden Zustände von der Suchstrategie nacheinander eingenommen werden d.h. $s(o) = ..., \imath, \imath^{'}, ...$.

Ein Informationszustand kann z.B. durch die Konstituenten, die noch abgearbeitet werden müssen, charakterisiert werden. In einer Suche, die von der linken zur rechten Regelseite übergeht, werden bei Erreichen eines Mutterknotens die noch zu findenden Tochterknoten angegeben. Die Abbildungen 2.9 und 2.10 zeigen Analyse- und Suchbaum für das Parsing des Satzes *peter schläft* in DCG-Notation. Ein Knoten in einem Analysebaum gibt eine abgeleitete Konstituente an, ein Knoten im Suchbaum dagegen die noch abzuleitenden.

Der Suchbaum in Abbildung 2.10 zeigt, daß die Suchstrategie zuerst eine Regel $S \rightarrow NP\ VP$ anwendet und dann zwei Wege verfolgt: die Anwendung einer Regel $NP \rightarrow \mathit{Det}\ N$ scheitert sofort (durch *failure* deutlich gemacht), die Anwendung einer Regel $NP \rightarrow PN$ ist dagegen erfolgreich, d.h. *peter* wird geparst. Daraufhin wird die Verbalphrase richtig analysiert (mit der Regel $VP \rightarrow Vi$), wodurch es keine noch zu analysierende Konstituente mehr gibt. Der erfolgreiche Zweig ist mit '$\oplus$' gekennzeichnet.

Die Suchstrategie geht also top-down vor: der Mutterknoten wird ersetzt durch die von ihm unmittelbar dominierten Töchter. Weiterhin ist zu erkennen, daß die Wortkette von links nach rechts geparst wird.

Ein Suchbaum dieser Art sagt nur aus, welche Schritte zur Analyse des Satzes unternommen werden, nicht jedoch, in welcher Reihenfolge dies geschieht. Insofern ist nicht jeder Aspekt der Suchstrategie erfaßt. Ich habe deshalb durch zusätzliche gestrichelte Pfeile die Abarbeitungsreihenfolge deutlich gemacht. Da Verzweigungen im Suchbaum durch die Grammatik bedingt sind, unterscheiden sich Breitensuche und (erschöpfende) Tiefensuche in bezug auf die Struktur des Suchbaumes grundsätzlich nicht – es sei denn, eine Endlosschleife bei der Tiefensuche verhindert die Verfolgung aller weiteren Alternativen. In diesem Fall wird die Unvollständigkeit der Tiefensuche auch graphisch deutlich.

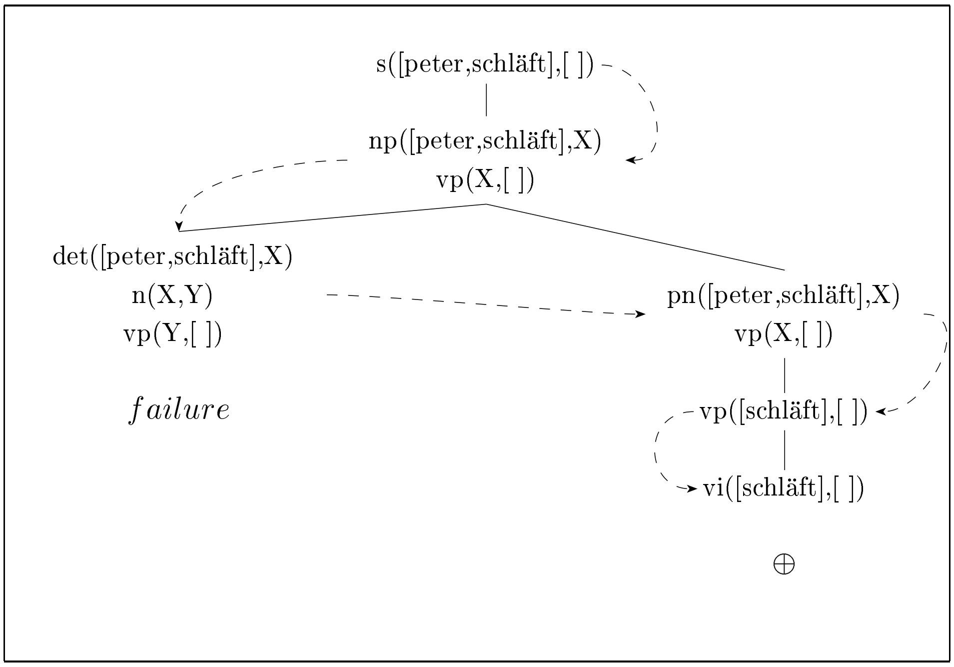


Abbildung 2.10: Suchbaum für *peter schläft* in DCG-Notation: top-down, depth-first und left-to-right Suchstrategie.

Deduktion und Analyse Ein entscheidender Aspekt in bezug auf die Verwendung von Analysestrategien in der Computerlinguistik ist bisher noch nicht genannt worden. Es gibt eine enge Beziehung zwischen Analysestrategien und Logik-Programmierung. Eine Grammatik kann in Prädikaten-Logik erster Stufe formuliert werden. Die Suche nach der Konstituentenstruktur ist dann eine Suche nach einer Ableitung. Ein Interpreter wendet Inferenzregeln an, um einen gegebenen Term unter Verwendung der Axiome zu beweisen: *Parsing as Deduction* (Pereira und Warren 1983).

Der enge Zusammenhang zwischen Analyse und Deduktion wird durch den Vergleich der Baumstrukturen für den Beweis und für die Konstituenten deutlich: sie sind homomorph. Der Analysebaum in Abbildung 2.9 kann als Beweisbaum für die entsprechende DCG angesehen werden. Auch der Beweisbaum zeigt nur die Schritte, die zur erfolgreichen Ableitung nötig sind.

Weiterhin dokumentiert der Suchbaum die Suche nach einem Beweis. Seine Knoten geben die noch nicht bewiesenen Ziele an. Das Thema dieses Abschnitts wird in Kapitel 4.1.1 eingehender behandelt.

Kapitel 3

Parsing und Generierung mit Kopf-gesteuerten Algorithmen

3.1 Überblick

In diesem Kapitel sollen einige Vorschläge zu uniformen Architekturen erläutert werden. Um sie in systematischer Weise darzustellen, werden folgende Dimensionen unterschieden, zu denen jeder Vorschlag direkt oder indirekt Stellung nimmt:

- **Analysestrategie** In dem vorhergehenden Kapitel wurden top-down-, bottom-up- und left-corner-Analyse dargestellt. Hier wird nun eine Strategie präsentiert, die den Kopf einer Phrase als Ausgangspunkt nimmt. Diese Strategie kann sowohl für das Parsing (van Noord (1993, 1997) und Kay (1989)) als auch für die Generierung verwendet werden (Shieber *et al.* (1990), van Noord (1989, 1990a, 1990b, 1991 und 1993)). Ihre grundsätzlichen Vor- und Nachteile werden möglichst unabhängig von Festlegungen in den anderen hier genannten Dimensionen erörtert.

  Die schon vorher behandelten Strategien wurden oben in bezug auf das Parsing erläutert. Sie sollen deshalb zusätzlich in Hinblick auf Generierung betrachtet werden, wobei vor allem die left-corner-Analyse berücksichtigt wird.

  Ein Schritt in der Traversierung eines lokalen Baumes bzw. eine Bewegung in einer Produktionsregel mit kontextfreier Basis läßt sich durch Einordnung in zwei weitere Dimensionen untergliedern:[1]

  – **Verarbeitungsrichtung** Wird von der linken zur rechten Seite übergegangen (top-down) oder umgekehrt (bottom-up)?

  – **Selektionsfunktion** Welches Element der rechten Seite wird gewählt?[2]

[1] Statt dem Begriff ‘linke Regelseite’ verwende ich im folgenden auch den Begriff ‘Mutterknoten’ und statt ‘Element der rechten Regelseite’ den Begriff ‘Tochterknoten’. In Hinblick auf einen lokalen Baum, der durch eine Produktionsregel mit kontextfreier Basis erzeugt wird, scheint mir beides zutreffend zu sein.

[2] Für die linke Regelseite stellt sich diese Frage natürlich nicht, weil sie aus nur einem Element besteht.

Die top-down-Analyse geht, wie oben beschrieben, von der linken zur rechten Regelseite über und wählt zumeist das am weitesten links stehende Element. Eine Analysestrategie kann aber auch verschiedene Festlegungen in diesen beiden Dimensionen integrieren. Die left-corner-Analyse beispielsweise geht im bottom-up-Teil von dem ersten Element der rechten Seite zur linken Seite über und im top-down-Teil von der linken zur rechten Seite, wobei das nächste noch nicht bearbeitete Element rechts der linken Ecke gewählt wird.

- **Kontrolle** Eine Suchstrategie kann, wenn sich Alternativen ergeben, entweder eine Analyse bis zum Ende verfolgen oder alle gleichzeitig (Tiefensuche versus Breitensuche). Dieses Verhalten kann durch die Verwendung einer *Agenda* modelliert werden. Tiefen- und Breitensuche stellen sich dann als die beiden Extrempunkte in einem Kontinuum von Kontrollstrategien dar.

- **Speicherung von Teilergebnissen** *(Tabulation* und *Memoization/Memoing)* Schon berechnete Teilergebnisse können gespeichert werden, um später auf sie zurückzugreifen. Dies führt zu einer Effizienzsteigerung, wenn die bei der Wiederverwendung der Teilergebnisse erzielte Ersparnis vor allem an Rechenzeit den Aufwand für die Speicherung rechtfertigt. Vor allem aber verhindert Memoization Endlosschleifen in der Programmausführung, wenn erkannt wird, daß ein Ziel schon einmal bearbeitet wurde.

  In diesem Zusammenhang gibt es Vorschläge zum möglichst effizienten Auffinden von gespeicherten Teilergebnissen (*Indexing*[3]), d.h. von schon analysierten Teilstrings bzw. schon generierten Phrasen.

3.2 Kopf-gesteuerte Algorithmen

3.2.1 Semantic-head-driven-Generation

Ein fundamentaler Unterschied zwischen Parsing und Generierung liegt in der Struktur des Inputs. Die beim Parsing gegebene phonologische Kette kann durch eine 'flache Liste' von Elementen auf gleicher Ebene repräsentiert werden. Die Generierung dagegen hat als Input eine hierarchische Struktur, die die Semantik repräsentiert (s. z.B. die Beispiele in Abschnitt 2.2.1). Zu jedem Wort einer zu parsenden Wortkette muß es einen lexikalischen Eintrag geben, der einen entsprechenden Merkmalswert für das Merkmal PHON aufweist.[4] Wenn beim Parsing die Ebene der Terminalsymbole erreicht ist, kann sehr schnell festgestellt werden, ob es einen (oder auch mehrere) Lexikoneinträge für das gesuchte Wort gibt. Der Input ist beim Parsing schon so gegliedert, daß den einzelnen Elemente der Kette

[3]Diese Form von Indexing ist nicht identisch mit dem Indexing von Prolog-Implementierungen, das zum schnellen Auffinden von Klauseln zumeist das erste Argument mitberücksichtigt.

[4]Diese Überlegungen gelten grundsätzlich nicht nur für Vollformen-Lexika, auch wenn dies hier der Einfachheit halber unterstellt wird.

bestimmte Lexikoneinträge entsprechen können. Da alle Elemente der Kette 'gleichwertig' sind, kann diese von links nach rechts abgearbeitet werden. Das Parsing folgt so der Abfolge, in der die Worte von einem Sprecher erzeugt werden.

Bei der Generierung dagegen ist es nicht klar, welcher Teil der semantischen Repräsentation mit einem lexikalischen Eintrag assoziiert werden kann. Die semantische Struktur kann durch Komposition aus einer Reihe von lexikalischen Enträgen entstanden sein. Ein zusätzliches Problem, das unten eingehender behandelt wird, tritt auf, wenn ein Wort nichts zur semantischen Repräsentation beiträgt oder aber die Semantik einer Phrase auf Satzebene nicht mehr vorhanden ist.

Wenn eine Wortkette für einen Teil der semantischen Repräsentation generiert wird, ergibt sich im allgemeinen bei der Generierung ein stärkerer Nicht-Determinismus als beim Parsing (auch hier sei auf die unten angesprochene Problematik der Existenz von Synoymen verwiesen). Beispielsweise kann eine NP in allen Numera und Fällen generiert werden, solange die Kongruenzbedingungen nicht bekannt sind. Es liegt also nahe, gerade bei der Generierung mit dem Element zu beginnen, das syntaktische Informationen über die anderen Elemente beitragen kann. Dadurch kann der Suchraum eingeschränkt bzw. die Generierung von letztendlich nicht benötigten Phrasen verhindert werden. Eine erfolgversprechende Strategie sollte also beispielsweise ein Verb vor seinen Komplementen generieren und ein Nomen vor dem Determiner. Im ersten Fall sind dann die Kasus- und Kongruenz-Informationen der Subkategorisierungs-Liste bekannt; im zweiten kann festgestellt werden, welche Kongruenz-Information ein Nomen mit der gesuchten Semantik enthält. Es ist andererseits z.B. wenig sinnvoll, zuerst einen Determiner zu erzeugen, um dann festzustellen, daß es kein Nomen mit dem nun feststehenden Genus und der gewünschten Semantik gibt.

Der bottom-up-Generator von van Noord

Van Noord (zuerst in van Noord (1989)) schlägt vor, die Generierung mit dem *Pivot* zu beginnen. Dies ist ein lexikalischer Eintrag, der mit der Input-Semantik unifiziert. Die gegebene Semantik wird nicht durch die Grammatikregeln dekomponiert, sondern von dem Zielknoten aus durch den lokalen Baum unverändert weitergegeben. Die geschieht durch die Verwendung von *Chain-Rules*. Sie sind durch einen *semantischen Kopf* auf der rechten Regelseite gekennzeichnet, dessen Wert für die Semantik mit dem semantischen Wert des Mutterknotens koindiziert ist. Wenn die Semantik des Mutterknotens instantiiert ist, dann ist auch die Semantik des Kopfes instantiiert. Van Noord (1989, S. 7) bezeichnet dies als *connectedness of the semantic head*. Es handelt sich also um Token-Identität, die sich in der Grammatik durch Koindizierung ausdrücken läßt. Die Semantik des Kopfes ist vorhersagbar aus der Semantik des Mutterknotens.

Zu Beginn der Analyse ist eine Zielkategorie gegeben. Aufgrund der im vorhergehenden Kapitel erläuterten Terminations-Problemen von top-down-Strategien soll jedoch bottom-up generiert werden. Die Semantik wird zu diesem Zweck unverändert durch die Grammatikregeln 'nach unten' weitergegeben, bis der Pivot gefunden ist. Es handelt sich also um eine top-down-Vorhersage. Nun kann eine zielgerichtete bottom-up-Generierung stattfinden. Es wird eine Regel gesucht, deren semantischer Kopf mit dem Pivot unifiziert, d.h. die Selektionsfunktion wählt zuerst den semantischen Kopf. Es handelt sich um einen bottom-up-Schritt, der den Analysebaum aufbaut und versucht, den Pivot mit dem gegebenen Ziel zu verbinden.

Nun werden die anderen Tochterknoten links und rechts von dem semantischen Kopf in derselben Weise generiert, d.h. erst top-down-Vorhersage eines lexikalischen Eintrags und dann bottom-up-Konstruktion des Analysebaums. Auf diese Weise vergrößert sich der generierte String rechts und links des semantischen Kopfes. Wenn alle Elemente der rechten Seite einer Grammatikregel abgearbeitet worden sind, kann für das Element der linken Regelseite wiederum ein passender semantischer Kopf in einer Regel gesucht werden. Diese Schritte werden rekursiv wiederholt, bis der Zielknoten mit der linken Seite einer vollständig bearbeiteten Regel unifiziert (wobei Kohärenz und Vollständigkeit beachtet werden müssen).

Wenn nach dem semantischen Kopf die anderen Elemente einer rechten Regelseite generiert werden, wird die Semantik des Kopfes entsprechend den Koindizierungen in den Grammatikregeln dekomponiert und auf die anderen Elemente verteilt. Der Pivot, der nun für diese Elemente gesucht wird, muß wieder vollständig mit der neuen Ziel-Semantik unifizieren. Es ist wichtig zu bemerken, daß die vom semantischen Kopf gesteuerte Generierung der Struktur des Inputs angepaßt ist. Wenn die Semantik, wie im vorhergehenden Kapitel beschrieben, mit Funktor-Argument-Strukturen repräsentiert wird, dann wird immer zuerst der Funktor generiert und dann die Argumente.

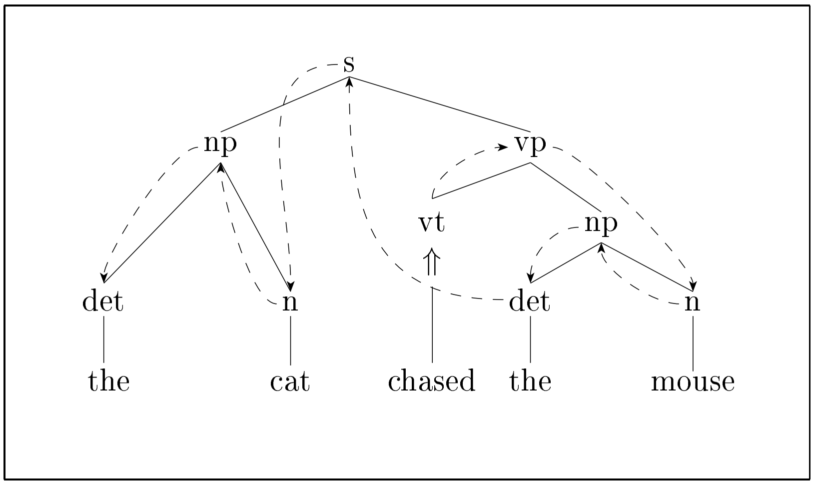


Abbildung 3.1: Semantic-head-driven-Analyse

Der Analysebaum in Abbildung 3.1 zeigt die deutliche Parallele zur left-corner-Analyse. Die Traversierung des Baumes ist in-order, nur daß nicht von links nach rechts vorgegangen wird, sondern vom semantischen Kopf aus. Dies ist durch den unterschiedlichen Input bei Parsing und Generierung motiviert. Statt Funktor-Argument-Strukturen werden beim Parsing Differenz-Listen verwendet, die eine Abarbeitung des Strings von links nach rechts ermöglichen. Bei beiden Analyse-Strategien werden die Grammatikregeln bottom-up ausgewählt und die anderen Elemente der rechten Regelseite rekursiv bearbeitet. Genau wie die Link-Relation in left-corner-Parsern kann die reflexive und transitive Hülle der Kopf-Relation zwischen semantischem Kopf und Mutterknoten zur Compile-Zeit berechnet werden. Es wird vorausgesetzt, daß jede Grammatikregel genau einen semantischen Kopf besitzt, wenn sie kein Lexikoneintrag ist.

In left-corner-Parsern stellt sich die Wahl einer linken Ecke erst dann als gerechtfertigt heraus, wenn alle anderen Elemente der rechten Seite bearbeitet werden konnten (Gerdemann und Hinrichs 1995, S. 174). Daraus ergibt sich ein Indeterminismus, denn die Wahl der Regel muß korrigiert werden können. Das gleiche gilt für Kopf-gesteuerte Ansätze. Erst wenn alle anderen Elemente neben dem semantischen Kopf erfolgreich bearbeitet wurden, kann ein weiterer bottom-up-Schritt erfolgen. Auch in diesem Punkt besteht also eine Parallele zum left-corner-Parsing.

Erläuterung des Pseudocodes Van Noord (zuerst in van Noord (1989)) definiert einen einfachen bottom-up-Generator (*BUG* genannt), der die grundlegenden Eigenschaften des Algorithmus demonstriert. In Abbildung 3.2 wird er in Pseudocode dargestellt. Der Code wurde modular formuliert, damit die einzelnen Prozeduren besser mit denen für den nächsten vorgestellten Algorithmus verglichen werden können. Gegeben ist eine Grammatik, die aus Lexikoneinträgen und Regeln mit einem semantischen Kopf auf der rechten Seite besteht. Zudem wird eine Merkmalsstruktur FS_{in} angegeben, mit der die Variable *ROOT* instantiiert wird. Da es sich um einen Generierungs-Algorithmus handelt, muß FS_{in} eine semantische Repräsentation enthalten. Zudem wird die Zielkategorie angegeben.

Nun wird die Prozedur `bottom-up-generation/1` mit *ROOT* als Argument aufgerufen. Es muß ein Lexikoneintrag gefunden werden `(1)`, der einen Link[5] zu *ROOT* aufweist und dessen Semantik mit der von *ROOT* unifizierbar ist `(2)`. Die Link-Relation dient dabei als *top-down oracle* (van Noord 1989, S. 5). Die Variable *PIVOT* wird mit dem gefundenen lexikalischen Eintrag instantiiert `(3)`. Mit *PIVOT* und *ROOT* als Argumenten wird nun die Prozedur `up-bug/2` aufgerufen `(4)`. Wenn es keinen geeigneten Pivot gibt, scheitert der Algorithmus `(5)`.

Die Prozedur `up-bug/2` wird solange wiederholt, bis der Pivot mit der gegebenen Ausgangsmerkmalsstruktur unifizierbar ist `(6)`: es wird eine Grammatikregel gesucht, deren semantischer Kopf mit dem Pivot unifiziert `(7)` und deren linke Seite einen Link zu *ROOT* aufweist `(8)`. Nun wird die Prozedur `generate-rhs-bug/2` aufgerufen, die die rechte Regelseite generiert `(9)`. Ist dies erfolgreich, wird die linke Seite der bearbeiteten Regel als

[5]Ich beziehe mich hier auf die Darstellung in van Noord (1989), wo die Link-Relation ausdrücklich verwendet wird. In van Noord (1993, 1990a) gehört sie nicht mehr zur Basis-Definition des Algorithmus, wird aber in Erweiterungen hinzugefügt.

Gegeben sei:

- Grammatik G aus Lexikoneinträgen und Regeln mit semantischem Kopf
- eine Input-Merkmalsstruktur FS_{in}

$ROOT := FS_{in}$

begin PROCEDURE bottom-up-generation($ROOT$)
1) **if** lex_entry $Lex \in G$ mit
2) Link zwischen Lex und $ROOT$, $SEM_{Lex} \sqcup SEM_{ROOT}$
3) **then** $PIVOT := Lex$;
4) **do** up-bug($PIVOT, ROOT$)
5) **else** fail
end

begin PROCEDURE up-bug($PIVOT$,$ROOT$)
6) **while** $PIVOT \not\trianglelefteq ROOT$
7) **do** suche Rule R mit $HEAD_R \sqcup PIVOT$ und
8) Link zwischen LHS_R und $ROOT$;
9) **do** generate-rhs-bug(RHS_R,$HEAD_R$);
10) $PIVOT := LHS_R$
end

begin PROCEDURE generate-rhs-bug(RHS,$HEAD$)
11) **for** alle Elemente $E \in RHS$ außer $HEAD$ **do**
12) bottom-up-generation(E)
end

Abbildung 3.2: Bottom-up Generation *(BUG)*

neuer Pivot definiert (10). Nun wird getestet, ob $PIVOT$ und $ROOT$ schon unifizierbar sind. Wenn dies nicht der Fall ist, muß beginnend mit (7) ein weiterer bottom-up-Schritt erfolgen.

Die Prozedur `generate-rhs-bug/2` mit den Argumenten RHS und $HEAD$ ruft für jedes Element der rechten Seite außer dem Kopf die Prozedur `bottom-up-generation/1` auf (11,12).

Der Algorithmus von Shieber *et al.* (1990)

Bisher wurde davon ausgegangen, daß alle Regeln mit der Ausnahme von lexikalischen Einträgen einen semantischen Kopf besitzen *(Chain-Rules)*. Der zuerst definierte Pivot ist ein lexikalischer Eintrag, von dem aus bottom-up generiert wird.

Es mag jedoch manchmal erwünscht sein, eine Regel ohne semantischen Kopf zu verwenden. Eine Grammatikregel, die eine Modifierkonstruktion wie in (5) einführt, kann beispielsweise so aussehen:[6]

$$(14)\quad \begin{bmatrix} \text{phon} & P_0 - P \\ \text{syn} & \begin{bmatrix} \text{cat} & \text{vp} \end{bmatrix} \\ \text{sem} & \begin{bmatrix} \text{mod} & \text{Mod} \\ \text{arg1} & \text{Pred} \end{bmatrix} \end{bmatrix} \longleftarrow \begin{bmatrix} \text{phon} & P_0 - P_1 \\ \text{syn} & \begin{bmatrix} \text{cat} & \text{ap} \end{bmatrix} \\ \text{sem} & \text{Mod} \end{bmatrix} \wedge \begin{bmatrix} \text{phon} & P_1 - P \\ \text{syn} & \begin{bmatrix} \text{cat} & \text{vp} \end{bmatrix} \\ \text{sem} & \text{Pred} \end{bmatrix}$$

Eine solche Regel ist eine *Nonchain-Rule*, weil es kein Element auf der rechten Seite gibt, dessen Semantik mit der des Elements der linken Seite identisch ist (nach dieser Definition sind also auch lexikalische Einträge Nonchain-Rules). Shieber *et al.* (1990) stellen einen *semantic-head-driven Generation*-Algorithmus vor *(SHDGA)*, der wie *BUG* arbeitet, aber auch mit Nonchain-Rules umgehen kann. Wenn der Pivot kein lexikalischer Eintrag ist, sondern das Element der linken Regelseite einer Nonchain-Rule (handelte es sich um eine Chain-Rule, könnte die linke Regelseite nicht als Pivot dienen, da es noch 'tiefer liegende' Knoten mit identischer Semantik gäbe), werden alle Elemente der rechten Seite top-down abgearbeitet. Bei jedem top-down-Schritt wird wieder versucht, einen Pivot zu finden.

Der Algorithmus von Shieber *et al.* (1990) arbeitet wie folgt: zunächst wird eine Nonchain-Rule gesucht, deren linke Seite als Pivot dienen kann, d.h. daß ein Link zur Zielkategorie besteht und daß die Semantik der linken Regelseite mit der der gegebenen Input-Merkmalsstruktur unifiziert. Nun werden alle Tochterknoten generiert – bei lexikalischen Einträgen ist die Abbruchbedingung für die rekursive Generierung der Elemente der rechten Regelseite sofort erreicht. Handelt es sich nicht um einen lexikalischen Eintrag, wird zu jedem Element ein Pivot gesucht, d.h. eine Nonchain-Rule, deren rechte Seiten wiederum rekursiv generiert werden.

Sobald alle Elemente einer rechten Regelseite generiert sind, kann der bottom-up-Schritt erfolgen. Der Pivot wird mit dem semantischen Kopf der 'nächsthöheren' Chain-Rule unifiziert und alle verbleibenden Tocherknoten in dieser neuen Regel werden generiert. Der bottom-up-Schritt wird solange wiederholt, bis Pivot und Wurzelknoten unifizierbar sind. Wie bei dem *BUG*-Algorithmus stellt die Link-Relation sicher, daß der Pivot gefunden werden kann, oder umgekehrt gesagt, daß es möglich ist, den Pivot mit dem Ausgangsknoten zu verbinden.

Die top-down-Generierung von Nonchain-Rules entsteht dadurch, daß der Pivot für das Element der rechten Regelseite, das gerade bearbeitet wird, der Mutterknoten der 'nächsttieferen' Regel ist, d.h. daß die verwendete Kopf-Relation nicht transitiv war bzw. daß die gefundene 'Kette' von Elementen, die die gleiche Semantik besitzen, nur aus zwei Elementen besteht. Kopf und Mutterknoten lassen sich sofort verbinden. Wenn nun auch

[6] Zur Erläuterung der Form der Grammatikregeln s. Abschnitt 4.1.1

für den Kopf der neuen Regel nur die unmittelbar als nächtes folgende Regel gefunden wird, entsteht so schrittweise eine top-down-Generierung. Dadurch, daß Nonchain-Rules sowohl Lexikoneinträge wie auch Grammatikregel ohne semantischen Kopf bezeichnen, braucht der Algorithmus keine spezielle Definition des Vorgehens für den allgemeinen top-down-Fall.

Erläuterung des Pseudocodes In Abbildung 3.3 wird der Algorithmus von Shieber *et al.* (1990) in Pseudocode dargestellt. Gegeben ist eine Grammatik aus Chain-Rules und Nonchain-Rules, wobei letztere auch Lexikoneinträge miteinschließen. Darüberhinaus ist eine Input-Merkmalsstruktur als Wert der Variablen $ROOT$ gegeben. Wie bei BUG muß sie die semantische Repräsentation und die Zielkategorie enthalten.

Zunächst wird eine Nonchain-Rule gesucht `(1)`, die einen Link zu $ROOT$ besitzt und deren linke Seite eine Semantik aufweist, die mit $ROOT$ unifiziert `(2)`. Diese linke Seite dient als Pivot `(3)`. Zur Generierung der rechten Seite wird die Prozedur `generate-rhs-shdga/2` mit der rechten Regelseite aufgerufen `(4)`. Da es sich um eine Nonchain-Rule handelt, gibt es keinen Kopf, mit dem das zweite Argument von `generate-rhs-shdga/2` instantiiert werden kann. Aus diesem Grund wird in `(4)` *nil* verwendet. Nach der Generierung der rechten Seite wird versucht, Pivot und Wurzelknoten zu verbinden. Dazu wird die Prozedur `up-shdga/2` aufgerufen `(5)`. Der Algorithmus scheitert, wenn es keinen geeigneten Pivot gibt `(6)`.

Die Prozedur `generate-rhs-shdga/2` mit den Argumenten RHS und $HEAD$ ruft für jedes Element E der rechten Seite außer dem Kopf die Prozedur `semantic-head-driven-generation/1` mit E als Argument auf `(12,13)`. Da lexikalische Einträge als Regeln mit leerer rechter Seite aufgefaßt werden, ist die Prozedur `generate-rhs-shdga/2` sofort erfolgreich.

Die Prozedur `up-shdga/2` wird mit $PIVOT$ und $ROOT$ als Argumente aufgerufen. Sie wird solange wiederholt, bis $PIVOT$ und $ROOT$ unifiziert werden können `(7)`. Für jeden bottom-up-Schritt wird eine Chain-Rule gesucht, deren semantischer Kopf mit dem aktuellen Pivot unifiziert `(8)` und deren linke Seite einen Link zu $ROOT$ besitzt `(9)`. Alle Elemente der rechten Seite außer dem Kopf werden mit der Prozedur `generate-rhs-shdga/2` generiert `(10)`. Nun wird die linke Seite der bearbeiteten Regel als neuer Pivot definiert, so daß überprüft werden kann, ob die Abbruchbedingung der while-Schleife erreicht ist.

Ein Vergleich von $SHDGA$ mit BUG anhand des Pseudocodes zeigt die strukturelle Ähnlichkeit beider Algorithmen: der $SHDGA$ kann als eine Generalisierung von BUG angesehen werden, weil als (Ausgangs-)Pivot nicht nur Lexikoneinträge, sondern linke Seiten von Nonchain-Rules im allgemeinen dienen können. Die Prozedur `semantic-head-driven-generation/1` unterscheidet sich also von `bottom-up-generation/1` darin, daß Chain-Rules statt Lexikoneinträge gesucht werden (Zeile `(1)` in beiden Abbildungen). Darüberhinaus muß in der Prozedur `semantic-head-driven-generation/1` die rechte Seite dieser Regel bearbeitet werden, so daß ein zusätzlicher Aufruf von `generate-rhs-shdga/2` nötig ist.

Gegeben sei:

- Grammatik G aus chain rules und nonchain rules;
- eine Input-Merkmalsstruktur FS_{in} mit semantischer Repräsentation und Zielkategorie

$ROOT := FS_{in}$

begin PROCEDURE semantic-head-driven-generation($ROOT$)
1) **if** nonchain rule $R_{nc} \in G$ mit
2) Link zwischen $LHS_{R_{nc}}$ und $ROOT$, $SEM_{LHS_{R_{nc}}} \sqcup SEM_{ROOT}$
3) **then** $PIVOT := LHS_{R_{nc}}$;
4) **do** generate-rhs-shdga($RHS_{R_{nc}}$,nil);
5) **do** up-shdga($PIVOT$,$ROOT$)
6) **else** fail
end

begin PROCEDURE up-shdga($PIVOT$,$ROOT$)
7) **while** $PIVOT \not\trianglelefteq ROOT$
8) **do** suche chain rule R_c mit $HEAD_{R_c} \sqcup PIVOT$ und
9) Link zwischen LHS_{R_c} und $ROOT$;
10) **do** generate-rhs-shdga(RHS_{R_c},$HEAD_{R_c}$);
11) $PIVOT := LHS_{R_c}$
end

begin PROCEDURE generate-rhs-shdga(RHS,$HEAD$)
12) **for** alle Elemente $E \in RHS$ außer $HEAD$ **do**
13) semantic-head-driven-generation(E)
end

Abbildung 3.3: Semantic-head-driven Generation Algorithmus *(SHDGA)*

Die Prozeduren `up-shdga/2` und `up-bug/2` unterscheiden sich nur darin, daß in `up-shdga/2` ausdrücklich eine Chain-Rule gesucht wird, während in *BUG* keine Differenzierung zwischen Chain-Rules und Nonchain-Rules nötig ist. Auch die Prozeduren `generate-rhs-shdga/2` und `generate-rhs-bug/2` sind weitgehend identisch, nur daß sie jeweils ihre 'Hauptprozedur' für jedes Element abrufen.

Die Besonderheit des *SHDGA* ist die geschickte Integration von top-down-Generierung, ohne daß zusätzliche Prozeduren eingeführt werden müssen. Im Falle der Bearbeitung von Nonchain-Rules ist der Pivot für ein Element der rechten Seite sofort mit der linken Seite des nächsten Regel unifizierbar. Es muß nur die rechte Seite der neuen Regel bearbeitet werden. Ein bottom-up-Schritt ist nicht notwendig, da Pivot und Root sofort unifizierbar sind. Wenn eine Grammatik überhaupt keine Chain-Rules besitzt, arbeitet der *SHDGA* wie ein top-down-Generator. Insofern kann er sich flexibel an verschiedene Grammatiken anpassen und die Kopf-Relation ausnutzen, wenn sie vorhanden ist.

3.2.2 Vorteile von Semantic-head-driven-Generation

Die beiden vorgestellten Algorithmen haben zumindest zwei wichtige Vorteile:

Kombination von top-down- und bottom-up-Strategie Das Links-Rekursions-Problem tritt nicht auf, wenn die Abbruchbedingung von Information aus dem Lexikon abhängig ist, also wenn z.B. die Lexikoneinträge für die Verben Subkategorisierungs-Information enthalten. Dies ist eine Eigenschaft von bottom-up-Algorithmen im allgemeinen und nicht speziell von Kopf-gesteuerten Ansätzen (s. Abschnitt 2.3.2). Andererseits haben bottom-up-Algorithmen generell aber das Problem, daß sie nicht zielorientiert arbeiten. Wie sich in Abschnitt 4.4.1 zeigen wird, tritt gerade bei der Generierung ein Effizienzproblem auf, weil eine große Anzahl von Phrasen erzeugt werden kann, ehe die 'richtige' gefunden ist. Die Steuerung der Verarbeitung durch die Kopf-Relation stellt eine geschickte Kombination von top-down-Vorhersage und bottom-up-Generierung dar.

Semantische Monotonie Das Problem der *semantischen Monotonie* ist in Zusammenhang mit dem Ansatz von Shieber (1988) aufgetreten (dazu s. Abschnitt 4.3.1). In diesem Abschnitt soll erläutert werden, was das Problem der 'semantischen Monotonie' ist und warum es semantic-head-driven-Generation nicht betrifft.

Eine Grammatik ist 'semantisch monoton', wenn die semantischen Repräsentationen aller Konstituenten einer Phrase einen (nicht leeren) Teil der semantischen Repräsentation der ganzen Phrase darstellen.[7] In bezug auf Grammatikregeln besagt dies, daß die semantische Repräsentation jedes Elements der rechten Seite einen Teil der semantischen Repräsentation der linken Seite subsumiert.

Die Forderung nach 'semantischer Monotonie' scheint zunächst nur zu besagen, daß die Semantik kompositionell aufgebaut wird. In Abschnitt 2.2 wurde das Konzept einer unifikationsbasierten Semantik erläutert, bei der die Semantik der ganzen Phrase schrittweise

[7] Am Ende dieses Abschnitt wird darauf eingegangen, ob diese Bezeichnung günstig gewählt ist.

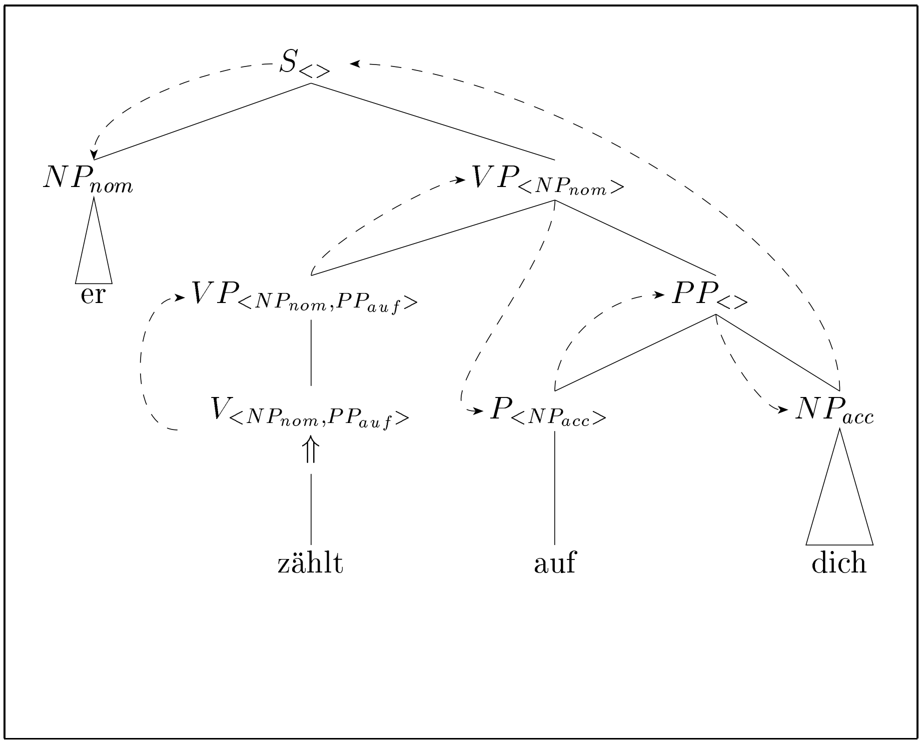


Abbildung 3.4: Semantic-head-driven Generierung von *er zählt auf dich*

durch Hinzufügen von Information zusammengesetzt wird. Dies ist z.B. bei der Grammatikregel in (14) der Fall.

Es gibt aber Analysen von syntaktischen Phänomenen, bei denen bestimmte Konstituenten nichts zur Semantik der ganzen Phrase beitragen. Dies gefährdet der Konzept der unifikationsbasierten Semantik nicht, weil keine Information gelöscht wird. In Abbildung 3.4 wird als Beispiel der Analysebaum für den Satz *er zählt auf dich* gegeben. Es wird angenommen, daß es – neben einem Lexikoneintrag für *zählen* in der 'normalen' Bedeutung – einen Eintrag für *zählen* mit der Semantik *zählen_auf* gibt, der für eine PP mit der Präposition *auf* subkategorisiert. Der lexikalische Eintrag für *auf* legt fest, daß die NP *dich* im Akkusativ stehen muß.

Abbildung 3.4 beschränkt sich auf die Angabe der Subkategorisierungs-Listen. Die gestrichelten Pfeile geben wie in den vorhergehenden Abbildungen von Analysebäumen die Abarbeitungs-Reihenfolge an. Es gibt zwei verschiedene 'semantische Ketten' in dieser Analyse: die erste führt von dem Lexikoneintrag für das Verb bis zum Wurzelknoten. Die zweite 'semantische Kette' führt von der Präposition *auf* zur PP. Auch für die Generierung dieser PP ist es wichtig, daß die Präposition zuerst generiert wird, damit die Subkategorisierungs-Information vorliegt und die NP sofort in dem richtigen Kasus erzeugt wird. Offensichtlich unterscheidet sich diese Analyse in bezug auf die Behandlung von Subkategorisierungs-Information nicht von den vorhergehenden Beispielen zur vom semantischen Kopf gesteuerten Generierung.

Um zu erläutern, worin die 'semantische Nicht-Monotonie' dieser Analyse besteht, werden im folgenden einige der Merkmalsstrukturen angegeben, mit denen die Knoten des Analysebaums in Abbildung 3.4 in einer Beispielgrammatik beschriftet sein können (und für die in Abb. 3.4 kein Platz ist).

In dem lexikalischen Eintrag für das Verb sind die Werte für die Pfade SEM ARG1 und SEM ARG2 koindiziert mit der Semantik der Komplemente in der Subkategorisierungsliste (15). Wenn ein Komplement des Verbs generiert wird, gelangt seine nun instantiierte Semantik dadurch in die Semantik des Verbs und wird so entlang der 'semantischen Kette' hochgereicht.

$$(15)\quad \left[\begin{array}{ll}
\text{phon} & \left\langle \text{zählt} \mid \text{R} \right\rangle - \text{R} \\
\text{syn} & \left[\begin{array}{ll}
\text{cat} & \text{v} \\
\text{sc} & \left\langle \left[\begin{array}{ll}
\text{syn} & \left[\begin{array}{ll}
\text{cat} & \text{np} \\
\text{agr} & \left[\begin{array}{ll}
\text{case} & \text{nom} \\
\text{num} & \text{sing} \\
\text{per} & \text{third}
\end{array}\right]
\end{array}\right] \\
\text{sem} & \left[\begin{array}{ll}\text{pred} & \boxed{1}\end{array}\right]
\end{array}\right],
\left[\begin{array}{ll}
\text{phon} & \left\langle \text{auf} \mid \text{R} \right\rangle - \text{R} \\
\text{syn} & \left[\begin{array}{ll}\text{cat} & \text{pp}\end{array}\right] \\
\text{sem} & \left[\begin{array}{ll}\text{arg} & \boxed{2}\end{array}\right]
\end{array}\right] \right\rangle
\end{array}\right] \\
\text{sem} & \left[\begin{array}{ll}
\text{pred} & \text{count_on} \\
\text{arg1} & \boxed{1} \\
\text{arg2} & \boxed{2}
\end{array}\right]
\end{array}\right]$$

Das gleiche Prinzip liegt bei dem lexikalischen Eintrag für *auf* vor: die Semantik des Komplements ist koindiziert mit dem Argument der Präposition.[8]

$$(16)\quad \left[\begin{array}{ll}
\text{phon} & \left\langle \text{auf} \mid \text{R} \right\rangle - \text{R} \\
\text{syn} & \left[\begin{array}{ll}
\text{cat} & \text{p} \\
\text{sc} & \left\langle \left[\begin{array}{ll}
\text{sem} & \left[\begin{array}{ll}\text{arg} & \boxed{3}\end{array}\right] \\
\text{syn} & \left[\begin{array}{ll}
\text{cat} & \text{np} \\
\text{case} & \text{acc}
\end{array}\right]
\end{array}\right] \right\rangle
\end{array}\right] \\
\text{sem} & \left[\begin{array}{ll}
\text{pred} & \text{on} \\
\text{arg} & \boxed{3}
\end{array}\right]
\end{array}\right]$$

Der entscheidende Punkt ist, daß in diesem Lexikoneintrag für *zählen* festgelegt ist, daß nur der semantische Wert des *Arguments* der PP mit dem Wert für SEM ARG2 koindiziert ist, nicht aber die Semantik der ganzen PP. Dadurch fällt die 'eigentliche' Semantik der

[8] Gerade die Semantik der Präposition mag linguistisch nicht angemessen repräsentiert sein, soll hier aber zu Demonstrationszwecken ausreichen.

Präposition in der VP und in S weg. Eine Grammatik, die einen solchen Analysebaum lizensiert, wird deshalb als 'semantisch nicht monoton' bezeichnet. Die Merkmalsstruktur für die VP nach der Generierung der PP sieht folgendermaßen aus (es muß nur noch das Subjekt generiert werden):

$$(17)\quad \left[\begin{array}{ll}
\text{phon} & \left\langle \text{zählt, auf, dich} \mid \text{R} \right\rangle - \text{R} \\
\text{syn} & \left[\begin{array}{ll}
\text{cat} & \text{vp} \\
\text{sc} & \left\langle \left[\begin{array}{ll}
\text{syn} & \left[\begin{array}{ll}
\text{cat} & \text{np} \\
\text{agr} & \left[\begin{array}{ll}
\text{case} & \text{nom} \\
\text{num} & \text{sing} \\
\text{per} & \text{third}
\end{array}\right]
\end{array}\right] \\
\text{sem} & \left[\begin{array}{ll} \text{pred} & \boxed{1} \end{array}\right]
\end{array}\right] \right\rangle
\end{array}\right] \\
\text{sem} & \left[\begin{array}{ll}
\text{pred} & \text{count_on} \\
\text{arg1} & \boxed{1} \\
\text{arg2} & \text{you}
\end{array}\right]
\end{array}\right]$$

Die semantische Information des lexikalischen Eintrags für die Präposition *auf* bleibt in dieser Analyse also auf die PP beschränkt. Die Semantik des Subjekts dagegen wird vollständig verwendet.

Eine ähnliche Analyse findet sich in Shieber *et al.* (1990, S. 36) für den Satz *John calls friends up*. Es handelt sich bei *up* nicht um eine PP mit interner Struktur, sondern um einen Partikel. Deswegen subkategorisiert *calls* in ihrer Analyse direkt für den Artikel *up*. Auch hier wird semantische Information, die ein Lexikoneintrag (in diesem Fall der für *up*) zur Verfügung stellt, auf höherer Ebene nicht wieder aufgenommen.

Es gibt weitere spachliche Phänomene, die 'semantisch nicht monoton' sind. Dazu gehören semantisch leere Wörter wie *es* in *es regnet* und Phraseo-Lexeme wie *den Löffel abgeben* bzw. engl. *to kick the bucket* als Bezeichnung für *sterben*. In Anlehnung an die in diesem Abschnitt dargestellten Analysen können Phraseo-Lexeme durch einen eigenen lexikalischen Eintrag für das Verb mit einer entsprechenden Subkategorisierungs-Liste behandelt werden. Ähnlich wie *auf* in der Analyse für *auf jemanden zählen* subsumiert *Löffel* keinen Teil der Semantik von *sterben*. Wie die Analyse in Abb. 3.4 zeigt, ist die semantic-head-driven-Generation-Strategie in der Lage, mit 'semantisch nicht monotonen' Grammatiken, die diese Phänomene zu erfassen versuchen, umzugehen.

Mir scheint die Bezeichnung 'semantisch monoton' jedoch nicht sehr glücklich zu sein. Die sprachlichen Phänomene, die in diesem Abschnitt angesprochen wurden, können im Rahmen von Unifikationsgrammatiken behandelt werden. Da die Unifikation eine monotone Operation ist, scheint es fraglich, wie mit ihr Nicht-Monotonie modelliert werden sollte. Es ist zutreffend, daß auf der höchsten phrasalen Ebene nicht die gesamte semantische Information aller Teilphrasen aufgenommen wird. Dies liegt daran, daß bei der Unifikation nicht immer die ganze semantische Information durch Koindizierung auf die nächsthöheren Ebene kopiert wird – wie dies auch mit anderer Information geschieht.

Darüberhinaus besteht die Gefahr, daß durch die Bezeichnung '*semantisch* nicht monoton' dies als eine Eigenschaft einer Objektsprache angesehen wird und nicht des Formalismus, der sie beschreibt. Ich halte es deshalb für besser, von 'Verletzung des semantischen Kompositionalitäts-Prinzips' zu sprechen.

3.2.3 Probleme von Semantic-head-driven-Generation

Die Verwendung von Kopf-gesteuerten Algorithmen setzt voraus, daß es ein Element auf den rechten Seiten der Grammatikregeln gibt, dessen Semantik mit der des Elements der linken Regelseite identisch ist. Wenn kein semantischer Kopf vorhanden ist, kann der Algorithmus nur top-down Vorgehen *(SHDGA)* bzw. er scheitert *(BUG)*. Der Grammatikentwickler ist gezwungen, Chain-Rules zu verwenden, wann immer dies möglich ist. Die Grammatikregel in (14) kann beispielsweise als folgende Chain-Rule geschrieben werden:

$$(18)\quad \begin{bmatrix} \text{phon} & P_0 - P \\ \text{syn} & \begin{bmatrix} \text{cat} & \text{vp} \end{bmatrix} \\ \text{sem} & \begin{bmatrix} \text{mod} & \text{Mod} \\ \text{arg1} & \text{Pred} \end{bmatrix} \end{bmatrix} \longleftarrow \begin{bmatrix} \text{phon} & P_0 - P_1 \\ \text{syn} & \begin{bmatrix} \text{cat} & \text{ap} \end{bmatrix} \\ \text{sem} & \begin{bmatrix} \text{mod} & \text{Mod} \\ \text{arg1} & \text{Pred} \end{bmatrix} \end{bmatrix} \wedge \begin{bmatrix} \text{phon} & P_1 - P \\ \text{syn} & \begin{bmatrix} \text{cat} & \text{vp} \end{bmatrix} \\ \text{sem} & \text{Pred} \end{bmatrix}$$

Da in vielen Grammatiktheorien ein 'Kopf-Prinzip' verwirklicht wird, sind die Voraussetzungen, die Kopf-gesteuerte Algorithmen machen, nicht so groß, wie es zunächst erscheinen mag. Sowohl die X'-Theorie in der *Government and Binding*-Theorie wie auch die GPSG (mit der *Head Feature Convention*), die HPSG und die *Head Grammar* als einer ihrer Vorläufer verwenden ein Konzept von 'Kopf'. "A fundamental and central concept in all contemporary syntax is the concept of *head"* (Sells 1985, S. 27; Hervorhebung im Original).

Es stellt sich einerseits die Frage, ob linguistisch motivierte Theorien tatsächlich immer das gleiche Kopf-Konzept verwenden. Andererseits ist es aber grundsätzlich vorstellbar, daß jede formalisierbare Relation zwischen linker Regelseite und einem bestimmten Element der rechten Seite als 'Kopf-Relation' definiert werden kann, um sie für den Algorithmus zu verwenden – solange es zumindest in den meisten Fällen genau ein ausgezeichnetes Element gibt, das der Definition genügt. Bei der Festlegung der Kriterien, die ein bestimmte Kopf-Definition fordert, spielt eine entscheidende Rolle, daß Kopf und linke Regelseite 'wesentliche' Information teilen.

Aufgrund der starken bottom-up-Komponente sind Kopf-gesteuerte Algorithmen gut für lexikalisierte Grammatiken geeignet. Auch dies deutet wieder auf die HPSG, aber auch auf die Categorial Unification Grammar (CUG).[9]

[9]Gerdemann und Hinrichs (1995, 186ff) zeigen, daß Kopf-gesteuerte Algorithmen für die Verarbeitung von Kategorialgrammatiken geeignet sind. Die Grundidee ist, daß der Funktor als 'Kopf' dienen kann, weil er wesentliche Information mit dem Mutterknoten teilt.

Schwierigkeiten treten in Kopf-gesteuerten Algorithmen naturgemäß dann auf, wenn die Bedingung, daß es genau einen semantischen Kopf geben muß, nicht eingehalten wird. Es können grundsätzlich folgende Fälle unterschieden werden: entweder es ist kein Kopf vorhanden oder es gibt mehrere oder das Kopf-Element ist leer bzw. unterspezifiziert. Der erste Fall wird durch die allgemeine Definition des Algorithmus von Shieber *et al.* abgedeckt. Die beiden letzten noch nicht behandelten Fälle werden nun näher untersucht.

Regeln mit mehreren möglichen semantischen Köpfen

Wenn es auf der rechten Seite einer Grammatikregel mehrere Elemente gibt, deren Semantik mit der des Elements der linken Seite identisch ist, kann nicht-deterministisch eines dieser Elemente als semantischer Kopf dienen. Die entscheidende Frage ist, ob es Fälle gibt, bei denen Terminations-Probleme auftreten können.

Die Generierung der Elemente der rechten Seite einer Regel außer dem semantischen Kopf erfolgt zunächst top-down, indem versucht wird, einen Pivot mit der entsprechenden Semantik zu finden. Führt diese Generierung zu der linken Seite der Regel zurück, kann sie ad infinitum angewendet werden. Als Beispiel wird eine an Shieber *et al.* (1990, S. 34) angelehnte Regel angegeben, die eine NP zu einer NP und einem Relativsatz expandiert:

$$(19)\quad \begin{bmatrix} \text{phon} & P_0 - P \\ \text{syn} & \begin{bmatrix} \text{cat} & \text{np} \end{bmatrix} \\ \text{sem} & \text{Sem} \end{bmatrix} \longleftarrow \begin{bmatrix} \text{phon} & P_0 - P_1 \\ \text{syn} & \begin{bmatrix} \text{cat} & \text{np} \end{bmatrix} \\ \text{sem} & \text{Sem} \end{bmatrix} \bigwedge \begin{bmatrix} \text{phon} & P_1 - P \\ \text{syn} & \begin{bmatrix} \text{cat} & \text{rel\_clause} \end{bmatrix} \\ \text{sem} & \text{Sem} \end{bmatrix}$$

Wenn das zweite Element der rechten Regelseite als semantischer Kopf benutzt wird, muß die Regel nach der Generierung des Relativsatzes von neuem angewendet werden, weil die noch zu bearbeitende NP der rechten Seite mit der der linken Seite identisch ist. Auch sie führt wiederum zu der Generierung eines Relativsatzes mit anschließender Bearbeitung der NP. Es entsteht also eine unendliche Kette von Relativsätzen.

Wie Shieber *et al.* (ebda) betonen, ist es deshalb wichtig, daß in einer Regel wie (19) das erste Element als semantischer Kopf dient. Nachdem die NP der rechten Seite bottom-up generiert wurde, kann das zweite Element (der Relativsatz) erzeugt werden. Grundsätzlich gibt es auch hier die – in diesem Fall linguistisch wenig plausible – Möglichkeit, daß auch die Bearbeitung des Relativsatzes wiederum zu einer NP *mit gleicher Semantik* führt, die kein Kopf-Element ist. In diesem Fall gäbe es keinen Ausweg aus der Endlosschleife.

Falls eines von mehreren in Frage kommenden Elementen als Kopf gewählt werden muß, um die Terminierung sicherzustellen, kann die Kompilation der Kopf-Relation nicht mehr automatisch geschehen (es sei denn, es wird ein zusätzliches Verfahren zum Aufspüren von Endlosschleifen eingebaut). Es müßten dann Annotationen eingeführt werden, die die Wahl des semantischen Kopfes steuern.

'Bewegung' von Köpfen

Ein relevantes Problem für Kopf-gesteuerte Algorithmen stellen Analysen der Wortstellung beispielsweise im Deutschen und im Niederländischen dar, bei denen das Verb aus der Satz-finalen Stellung an die zweite Position 'bewegt' wird.

Ohne diese Bewegung des Verbs im wörtlichen Sinn durchzuführen, kann eine solche Analyse im Rahmen von Unifikationsgrammatiken durch ein nicht realisiertes Element an der Verbletztposition, das mit dem Verb an der zweiten Position unifiziert, durchgeführt werden (Shieber et al. 1990, S. 40).[10] Dies erlaubt die Verwendung der meisten Grammatikregeln sowohl für den Hauptsatz (Verbzweit) wie auch für den Nebensatz (Verbletzt).

Das Problem für einen vom semantischen Kopf gesteuerten Algorithmus liegt darin, daß ein nicht realisiertes Verb (für die Verbletzt-Position) als Pivot gewählt werden muß *bevor* das finite Verb in Verbzweit-Position generiert wurde. Für ein nicht realisiertes Verb (oft auch als 'leeres Verb' bezeichnet) findet aber kein lexikalischer Zugriff statt, so daß die Subkategorisierungs-Liste nicht instantiiert ist und eine linksrekursive VP-Komplement-Regel unendlich oft angewendet werden kann:[11]

Der Analysebaum in (3.5) verdeutlicht die Probleme, die ein Kopf-gesteuerter Algorithmus bei der Verarbeitung von 'leeren Köpfen' hat. Nach der Generierung des Adverbs *heute* wird ein Pivot für das Element `s1` gesucht. Als Pivot dient das nicht realisierte Verb e_i, das nicht weiter instantiiert wird, weil kein lexikalischer Zugriff erfolgt. Dadurch ist bei der nun folgenden Bearbeitung der VP-Komplement-Regel die Subkategorisierungs-Liste nicht instantiiert.

Dieses Problem tritt unter Annahme auf, daß das Verb in der rechten Satzklammer als semantischer Kopf definiert wird, z.B. motiviert durch die Überlegung, daß die Wortstellung im Nebensatz die 'zugrunde liegende' ist.[12] Wenn umgekehrt das Verb in der linken Satzklammer als Kopf angesehen wird, tritt das Problem bei der Analyse von Nebensätzen auf.

Ein Lösungsvorschlag ist die Verzögerung der morphologischen Realisierung der Worte

---

[10] Nach Gerdemann und Hinrichs (1995, S. 178-186) lassen sich grundsätzlich zwei Arten von Analysen zum Problem der Verbzweitstellung unterscheiden: GB-orientierte Analysen, die eine Bewegung des Verbs aus einer zugrunde liegenden Verbletztstellung heraus annehmen, und monostratale Analysen, z.B. in der HPSG, die Regeln für die lineare Abfolge der Konstituenten annimmt. Dieses Thema kann hier nicht ausführlich diskutiert werden. Es soll aber darauf hingewiesen werden, daß das Ergebnis der Modellierung von GB-orientierten Analysen in Unifikationsgrammatiken monostratal ist. Darüberhinaus gibt es auch die Möglichkeit, verschiedene Regeln für Haupt- und Nebensatz zu verwenden (wie z.B. in der für das Projekt B3 entwickelten Grammatik), so daß sich die hier diskutierten Probleme gar nicht erst stellen.

[11] Beispielgrammatiken zur Behandlung von Verbzweitstellung finden sich in Gerdemann und Hinrichs (1995), Shieber *et al.* (1990, S. 40) und van Noord (1990a, S. 155). Ich verzichte hier darauf, eine eigene Beispielgrammatik zu entwickeln, sondern beschränke mich in Abbildung (3.5) auf einen möglichen Analysebaum für einen Satz mit Verbzweitstellung, wie er von der Grammatik in Shieber *et al.* (ebda) lizensiert wird (mit Ausnahme der Lexikoneinträge). Die Knotenbeschriftungen entsprechen den Kategoriesymbolen dort.

[12] Es soll noch einmal darauf hingewiesen werden, daß es hier nicht das Ziel ist, die Vor- und Nachteile bestimmter Analysen aus linguistischer Perspektive zu betrachten.

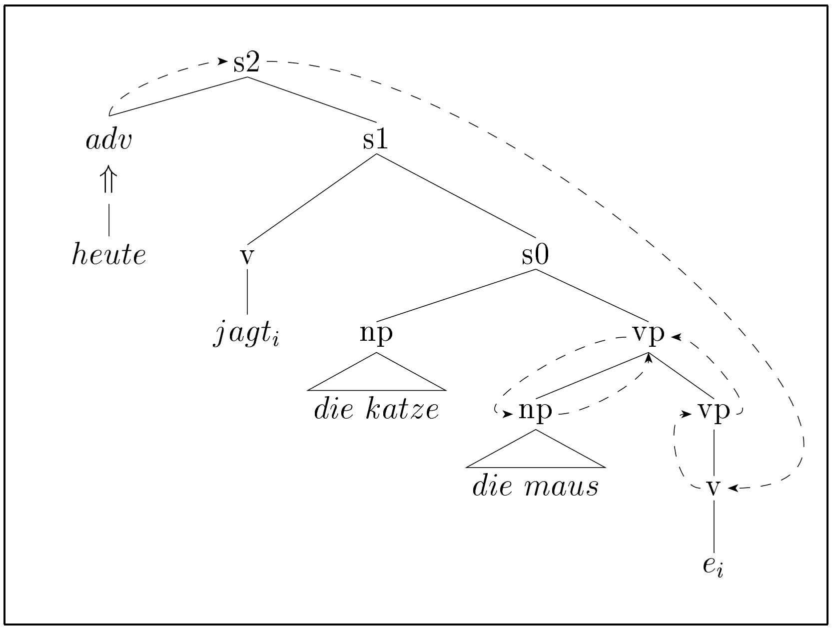


Abbildung 3.5: Loop bei vom semantischen Kopf gesteuerter Analyse von Grammatiken mit 'bewegten' Köpfen

(van Noord 1989, S. 8). Das leere Verb wird dann als eine Variante des morphologisch realisierten aufgefaßt. Dies bedeutet, daß während der Analyse in der Verbletzt-Position ein durch den lexikalischen Zugriff voll instantiiertes Verb verwendet wird. Nach Abschluß der Analyse stellt sich heraus, daß nicht es selbst morphologisch realisiert werden soll, sondern sein *Antezedens* in der Verbzweit-Position (das Element in der 'ursprünglichen' Position wird auch als *'Anapher'* bezeichnet).

Eine solche Verzögerung hat darüberhinaus den Vorteil, daß nach Abschluß der Baum-Traversierung alle Merkmale soweit wie möglich instantiiert sind und die Wortformen deshalb zielgerichteter erzeugt werden können. Wenn z.B. die Kongruenz-Merkmale des Subjekts nicht vollständig bekannt sind, gibt es verschiedene infrage kommende lexikalischer Einträge für das Verb (für die verschiedenen Flektionsvarianten), da diese zunächst nur nach ihrer Semantik ausgewählt werden.[13] Der daraus resultierende Nicht-Determinismus bei der Wahl des lexikalischen Eintrags kann durch die maximale Verzögerung der morphologischen Realisierung vermieden werden. Diese Methode ist grundsätzlich natürlich nicht auf eine bestimmte Analysestrategie beschränkt. Wilcock und Matsumoto (1996) zeigen durch die Anwendung auf robustes Parsing, daß sie darüberhinaus auch nicht nur auf eine Verarbeitungs-Richtung beschränkt ist.

Die Verzögerung der morphologischen Realisierung einzelner Verben ist keine allgemeingültige Lösung des Problems der Verbzweitstellung, weil es z.B. im Niederländischen oder

[13]Wenn man der Auffassung Bloomfields folgt, daß es keine Synonyme gibt (s.o.), muß man generell davon ausgehen, daß die zur Verfügung stehende semantische Repräsentation nicht vollständig spezifiziert ist, da sonst nur eine Wortform infrage käme.

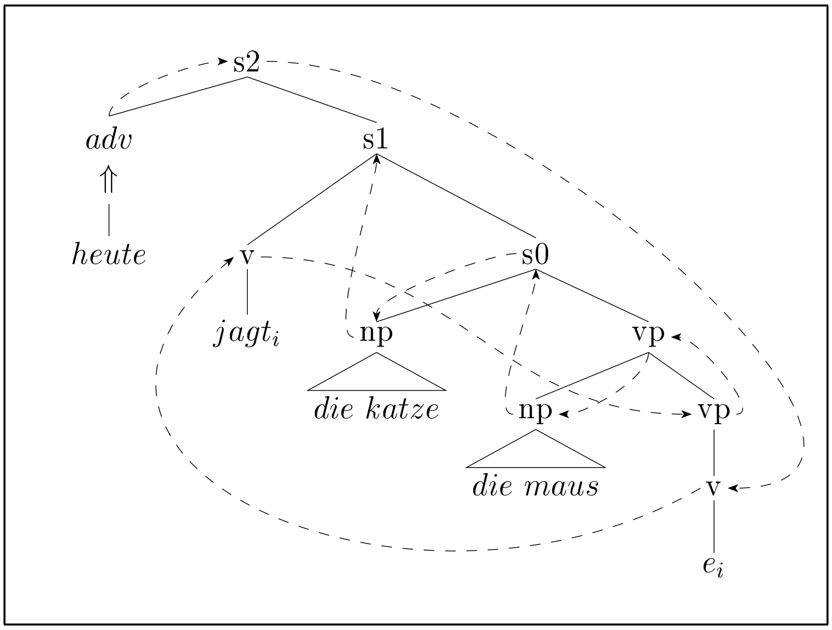


Abbildung 3.6: Vom semantischen Kopf gesteuerte Analyse mit vorgezogener Bearbeitung des Verbs in zweiter Position

im Spanischen sehr komplexe VPs gibt, die 'verschoben' werden. Das Problem reduziert sich dann nicht mehr auf die Frage, in welcher Position die realisierte Wortform steht, d.h. es werden nicht nur lexikalische Einträge 'bewegt'. Aus diesem Grund schlagen Shieber *et al.* (1990, S. 41) ein allgemeineres Verfahren vor, um das Antezedens (also die in zweiter Position stehende Konstruktion) vor der Anapher zu erzeugen. Es ist dabei nicht notwendig, daß es der semantische Kopf einer Phrase ist, der 'bewegt' wird. Dies kann als eine zeitweilige Aufgabe der Kopf-gesteuerten Strategie angesehen werden (Gerdemann und Hinrichs 1995, S. 184). Man könnte aber auch argumentieren, daß die Verarbeitungstrategie nur der 'Spur' *(trace)* des 'bewegten' Elements folgt und so letztendlich nichts anderes macht, als die grammatische Analyse annimmt.

Der Analysebaum in Abbildung (3.6) zeigt in schematischer Weise die Traversierung des Baumes aus Abbildung (3.5) mit der modifizierten Strategie. Wenn der leere Kopf erreicht ist, wird sofort versucht, das Antezedens zu generieren, d.h. den Lexikon-Eintrag für *jagt.* Da Antezedens und Anapher koindiziert sind, steht die Subkategorisierungs-Information nun auch in der Verbletzt-Position zur Verfügung, so daß die Generierung in Kopf-gesteuerter Weise weitergehen kann und terminiert. Zu dem Zeitpunkt der Generierung des 'bewegten' Elements ist unbekannt, an welcher Stelle im Baum es später 'eingehängt' wird.

Diese von Shieber *et al.* (1990) vorgeschlagene Lösung sieht also vor, daß die Relation zwischen Antezedens und Anapher explizit in der Grammatik bzw. der korrespondierenden Menge von Definiten Klauseln ausgedrückt wird, z.B. durch eine Prolog-Klausel, die aussagt, daß eine leere Konstruktion angenommen werden darf, wenn die korrespondierende zu realisierende Konstruktion generiert werden kann. Gerdemann und Hinrichs (1995, S. 185) interpretieren diese Lösung so, daß bestimmte Regeln (solche, die eine Beziehung zwischen nicht realisiertem und realisiertem Element einführen) besonders gekennzeichnet sind, so daß die Kopf-gesteuerte Analysestrategie zeitweilig unterbrochen wird. Sie sehen darin zwei Schwierigkeiten: zunächst einmal bedeutet diese Lösung, daß der Grammatikschreiber sich mit Fragen der Analysestrategie beschäftigen muß, d.h. daß die strikte Trennung zwischen Grammatik und Prozessor aufgehoben wird. Vor allem aber läßt sich das Verfahren von Shieber *et al.* nicht auf Phänomene wie Topikalisierung anwenden, bei denen nicht von vornherein feststeht, ob es ein Kopf ist, der 'bewegt' wurde, oder ein 'Nicht-Kopf'. Im folgenden sind die beiden Beispielsätze aus Gerdemann und Hinrichs aufgeführt (1995, S. 185):

(20) a. *Das Buch lesen wird Peter.*

b. *Lesen wird Peter das Buch.*

In (20a) wird die NP *das Buch* topikalisiert, in (20b) dagegen das Verb *lesen*, das als Kopf angesehen werden kann, weil es die Subkategorisierungs-Liste enthält. Obwohl der Vorschlag von Shieber *et al.* zunächst nicht für die Behandlung dieses Phänomens gedacht wurde, ist dies eine relevante Frage, weil so ein einheitliches Verfahren zur Behandlung von Konstituenten, die 'bewegt' werden, zur Verfügung stände. Der entscheidende Punkt ist, daß die Analyse von (20a) in der 'normalen' Weise durchgeführt werden kann, für (20b) aber zuerst das Antezedens generiert werden muß. Deswegen kann die Grammtik nicht mehr von vornherein festlegen, was geschehen soll, wenn eine nicht genauer spezifizierte Konstituente bearbeitet wird.

Meiner Ansicht nach ist dieses Problem aber nicht unlösbar: die Analysestrategie muß 'intelligent genug' sein, zu erkennen, ob die fehlende Information zu Terminations-Problemen führen kann (wie im Falle der Subkategorisierungs-Liste von *lesen*) oder nicht, d.h. es muß während der Laufzeit entschieden werden, wie die Analyse weitergehen soll. Zudem würden dadurch Grammatik und Prozessor wieder entkoppelt. Dies läßt sich aber nur durch eingehendere Untersuchungen und Experimente überprüfen.

Terminationsprobleme

Es lassen sich auch Grammatikregeln mit genau einem semantischen Kopf konstruieren, die Terminationsprobleme für den vom semantischen Kopf gesteuerten Generierungs-Algorithmus verursachen. Esther König (1994) zeigt einen solchen Fall, der aber in 'realen' Grammatiken kaum auftreten wird. Terminationsprobleme entstehen, wenn rekursiv Elemente generiert werden können, die keinen Teil der Input-Semantik subsumieren.

Angenommen, ein Verb subkategorisiert für einen Partikel *fou*, dessen Semantik nichts zu der Semantik auf Satz-Ebene beiträgt (also wie in dem oben beschriebenen Fall). Darüberhinaus soll ein Lexikoneintrag für diesen Partikel mit der Semantik *nil* angenommen werden (21) und eine Grammatikregel, die eine beliebige Anzahl dieses Partikels produziert (22).

$$(21)\quad \begin{bmatrix} \text{phon} & \text{fou} \\ \text{sem} & \text{nil} \\ \text{syn} & \begin{bmatrix}\text{cat} & \text{particle}\end{bmatrix} \end{bmatrix}$$

$$(22)\quad \begin{bmatrix} \text{sem} & \text{Sem} \\ \text{syn} & \begin{bmatrix}\text{cat} & \text{particle}\end{bmatrix} \end{bmatrix} \longleftarrow \begin{bmatrix} \text{sem} & \text{Sem} \\ \text{syn} & \begin{bmatrix}\text{cat} & \text{particle}\end{bmatrix} \end{bmatrix} \wedge \begin{bmatrix} \text{sem} & \text{Sem1} \\ \text{syn} & \begin{bmatrix}\text{cat} & \text{particle}\end{bmatrix} \end{bmatrix}$$

Um das Partikel *fou* zu generieren, wird als Pivot der Lexikoneintrag aus (21) gewählt. Wenn bei dem nun folgenden bottom-up-Schritt die Regel aus (22) verwendet wird anstatt den Pivot direkt mit dem Ziel zu verbinden, dann muß ein zweiter Partikel (das zweite Element der rechten Regelseite) generiert werden. Auch bei dessen Generierung kann ein weiterer lokaler Baum zwischen Pivot und Ziel 'eingehängt' werden, der wiederum ad infinitum zu weiteren Generierungen führt.

Das Problem bei Verarbeitung einer solchen Grammatik ist, daß es keine Abbruchbedingung wie z.B. eine leere Subkategorisierungs-Liste gibt. Es ist deshalb auch nicht die mangelnde Verfügbarkeit von Information, die das Problem verursacht, sondern die Grammatik selbst – indem sie nämlich eine unendliche Anzahl von Partikeln zuläßt. Dieses Beispiel ist deshalb kein ernsthafter Einwand gegen Kopf-gesteuerte Analysestrategien (und auch nicht gegen andere), da die Suchstrategie von der Grammatik tatsächlich definierte unendliche Zweige des Analysebaums traversiert.

Versteckte 'Kopf-Rekursion' Das in Abschnitt 2.3.2 beschriebene Terminationsproblem für left-corner-Parser trifft jedoch auch für Kopf-gesteuerte Algorithmen zu, da sie die gleiche in-order-Analyse-Strategie verfolgen, d.h. es entsteht auch in Verbindung mit Grammatiken, die eine endliche Menge von Ableitungsschritten definieren. In van Noord (1997) wird explizit auf das head-corner-Parsing bezug genommen (dazu s.u.). Es gibt jedoch keinen Grund, die Generierungs-Variante von diesem Problem auszuschließen. Wie beim Parsing mit 'leeren Konstituenten', die keinen Teil der Input-Wortkette verbrauchen, kann das Terminationsproblem auch bei der Generierung von semantisch leeren Elementen auftreten. Da die Semantik eines möglichen Pivots mit der Zielsemantik unifizieren muß, kann das Problem nur entstehen, wenn das zu generierende Element eine Variable als semantische Repräsentation enthält (s. z.B. den Lexikoneintrag (23)).

3.2.4 Erweiterungen

Syntaktische Information in der top-down-Vorhersage

Die Link-Relation in der bisher dargestellten Form enthält für jede Chain-Rule ein geordnetes Paar aus der Kategorie des Mutterknotens und der des semantischen Kopfes (zusätzlich wird die transitive Hülle dieser Relation berechnet). Dadurch kann beispielsweise die rekursive Anwendung von Regeln für Elemente mit leerer Semantik verhindert werden. Van Noord (1993, S. 89) nennt als Beispiel einen Komplementizer mit leerer Semantik, der für eine VP subkategorisiert (hier in etwas veränderter und vereinfachter Form dargestellt):

(23) $$\begin{bmatrix} \text{phon} & \textit{daß} \\ \text{sem} & \text{Sem} \\ \text{syn} & \begin{bmatrix} \text{cat} & \text{comp} \\ \text{sc} & \left\langle \begin{bmatrix} \text{sem} & \text{Sem} \\ \text{syn} & \begin{bmatrix}\text{cat} & \text{vp}\end{bmatrix}\end{bmatrix}\right\rangle \end{bmatrix}\end{bmatrix}$$

Die Semantik dieses lexikalischen Eintrags kann als 'leer' bezeichnet werden, weil sie einfach an die VP weitergegeben wird, ohne Information zu konsumieren. Bei jeder top-down-Vorhersage ohne Berücksichtigung der Link-Relation kann ein solcher Lexikoneintrag als Pivot dienen, da seine Semantik mit der gesuchten unifiziert. Durch die Verwendung von Links dagegen wird festgelegt, wann ein Komplementizer von seiner syntaktischen Kategorie her überhaupt nur als möglicher Pivot in Frage kommt, d.h. ob es eine Kombination von Regeln gibt, die es erlaubt, ihn ausgehend von dem jeweiligen Zielknoten zu erreichen.

Grundsätzlich kann in der Link-Relation auch syntaktische (oder auch jede andere) Information verwendet werden. Van Noord (1993, S. 90) macht einen deutlichen Unterschied bei der Behandlung von syntaktischer und semantischer Information. Während semantische Information bei der top-down-Vorhersage zur weiteren Instantiierung der semantischen Köpfe dient, wird syntaktische Information nur zur Einschränkung des Suchraums verwendet. Der Grund dafür sind möglicherweise auftretende falsche Ambiguitäten. Es kann unter Umständen zwei oder mehr Links zwischen zwei Elementen geben, obwohl sie aus den gleichen Regelkombinationen resultieren. Hier wird ein an van Noord (1993, S. 90) angelehntes Beispiel angegeben:

(24) $$\begin{array}{l} \left\langle \begin{bmatrix} \text{syn} & \begin{bmatrix} \text{cat} & \text{s} \\ \text{main} & \text{yes} \end{bmatrix} \end{bmatrix}, \begin{bmatrix} \text{syn} & \begin{bmatrix} \text{cat} & \text{v} \\ \text{main} & \text{yes} \end{bmatrix} \end{bmatrix} \right\rangle \\[2ex] \left\langle \begin{bmatrix} \text{syn} & \begin{bmatrix} \text{cat} & \text{s} \\ \text{mood} & \text{decl} \end{bmatrix} \end{bmatrix}, \begin{bmatrix} \text{syn} & \begin{bmatrix} \text{cat} & \text{v} \\ \text{mood} & \text{decl} \end{bmatrix} \end{bmatrix} \right\rangle \end{array}$$

Die Link-Relation wird als geordnetes Paar aus zwei Merkmalsstrukturen aufgefaßt, die jeweils eine Teilstruktur der ganzen den betreffenden Knoten beschriftenden Merkmalsstruktur darstellen. Van Noord schlägt vor, daß diese Information nur als Test verwendet wird und kein top-down-Informations-Transport stattfindet, damit – eine entsprechende Implementierung vorausgesetzt – kein unerwünschtes Backtracking erfolgt. In Prolog läßt sich ein solcher Test durch die Verwendung von doppelter Negation (implementiert als *negation as failure*) erreichen, wodurch Variablenbindungen vermieden werden.

Ein möglicher Ausweg ist die Zusammenfassung der Links durch Generalisierung (dies ist die Verbesserung, die in van Noord (1997, Abschnitt 2.2.3) vorgeschlagen wird). Aus dem Beispiel in (24) folgt dann jedoch, daß die in den ursprünglichen Links vorhandene Information nicht voll ausgenutzt wird – die Generalisierung der beiden Links in (24) enthält nur noch Angaben über die syntaktische Kategorie. Wenn die beiden Links tatsächlich aus der gleichen Regel bzw. Regelkombination stammen, sollte ihre syntaktische Information dagegen vollständig verwendet werden (25). Die in van Noord (1993, S. 90) als Beispiel gegebenen Links enthalten offensichtlich nicht die ganze Information der Grammatikregeln. Wenn diese jedoch berücksichtigt wird, besteht die Aufgabe in der Tat darin, für jedes Paar von syntaktischen Kategorien durch Generalisierung der verschiedenen Links einen einzigen maximal spezifischen Link zu finden.

(25) $$\left\langle \begin{bmatrix} \text{syn} & \begin{bmatrix} \text{cat} & \text{s} \\ \text{main} & \text{yes} \\ \text{mood} & \text{decl} \end{bmatrix} \end{bmatrix}, \begin{bmatrix} \text{syn} & \begin{bmatrix} \text{cat} & \text{v} \\ \text{main} & \text{yes} \\ \text{mood} & \text{decl} \end{bmatrix} \end{bmatrix} \right\rangle$$

Wenn nur noch ein Link verwendet wird, gibt es keinen Grund mehr, syntaktische Information anders als semantische zu behandeln. Grundsätzlich ist ein möglichst großer Informations-Transport wünschenswert, da dadurch zielgerichteter vorgegangen werden kann. Bei nur einer Regelanwendung ist ein Test genauso wirksam in bezug auf die Einschränkung des Suchraums wie die Instantiierung. Bei jedem weiteren Schritt dagegen ist die vorher zur Verfügung stehende Information nicht mehr vorhanden. Dadurch ist die Suche ungerichteter als bei der Instantiierung.[14]

Syntactic-head-driven-Generation

Wenn es in einer Grammatik keine Regeln mit einem semantischen Kopf gibt, muß – wie oben bereits erwähnt – eine andere Relation zwischen linker Regelseite und ausgezeichnetem Element der rechten Seite verwendet werden. Beispielsweise haben bei der Modellierung von *Underspecified Discourse Representation Structures (UDRSs)* die linken Regelseiten im Normalfall eine größere semantische Struktur als die Elemente der rechten Seite (König 1994).[15]

Um dennoch Kopf-gesteuerte Analysestrategien benutzen zu können, schlägt König (1994) vor, den *syntaktischen* Kopf zu verwenden, wobei dieser vom Grammatikschreiber festgelegt wird. Eine Link-Relation zwischen zwei Knoten ist dann gegeben, wenn die syntaktischen Kategorien in einer Link-Relation stehen und die Semantik des 'tiefer stehenden' Knotens eine (möglicherweise leere) Substruktur des 'höher stehenden' Knotens ist. Die zweite Bedingung hat das Ziel, die Generierung auf die gegebene Struktur auszurichten

[14] Die Vorteile von Informations-Transport in der Link-Relation beim left-corner-Parsing werden in Kilbury (1996) in bezug auf Flektionsmorphologie und die Verarbeitung unbekannter Wörter erläutert.

[15] Die Einzelheiten dieser Theorie sollen hier nicht besprochen werden. Wichtig sind die Konsequenzen für die Analysestrategie.

und die Input-Semantik zu berücksichtigen.

Die syntactic-head-driven-Generation-Strategie verhält sich wie die vom semantischen Kopf gesteuerten, wenn das gleiche Element einer rechten Regelseite sowohl als syntaktischer wie auch als semantischer Kopf definiert werden kann. Die Semantik des Kopfes ist dann eine unechte Teilstruktur der Semantik des Mutterknotens. Wenn sich syntaktischer und semantischer Kopf unterscheiden, wird der syntaktische Kopf zuerst bearbeitet. Ein Beispiel ist die Modifier-Regel in (18), in der das erste Element der rechten Seite (die AP) der semantische Kopf und das zweite Element der syntaktische Kopf ist. Wenn die Semantik des syntaktischen Kopfes nicht instantiiert ist, kann es wie bei der vom semantischen Kopf gesteuerten Generierung zu Effizienz-Problemen kommen. In einem solchen Fall ist der Teilstruktur-Test sinnlos.

'Bewegung' von Köpfen Wenn ein Kopf zu einer Position 'bewegt' wird, die den syntaktischen Kopf einer übergeordneten Phrase darstellt, kann die Analysestrategie diesen Kopf sofort finden. Beispielsweise kann in bezug auf die Analysebäume in (3.5) und (3.6) das Verb als der syntaktische Kopf der Regel $s1 \rightarrow v\ s0$ angesehen werden. Wenn der Lexikoneintrag für das Verb gefunden wurde, ist die Subkategorisierungs-Liste instantiiert und die Analyse kann problemlos weitergeführt werden. Der Analysebaum könnte dann wie in Abbildung 3.7 aussehen.

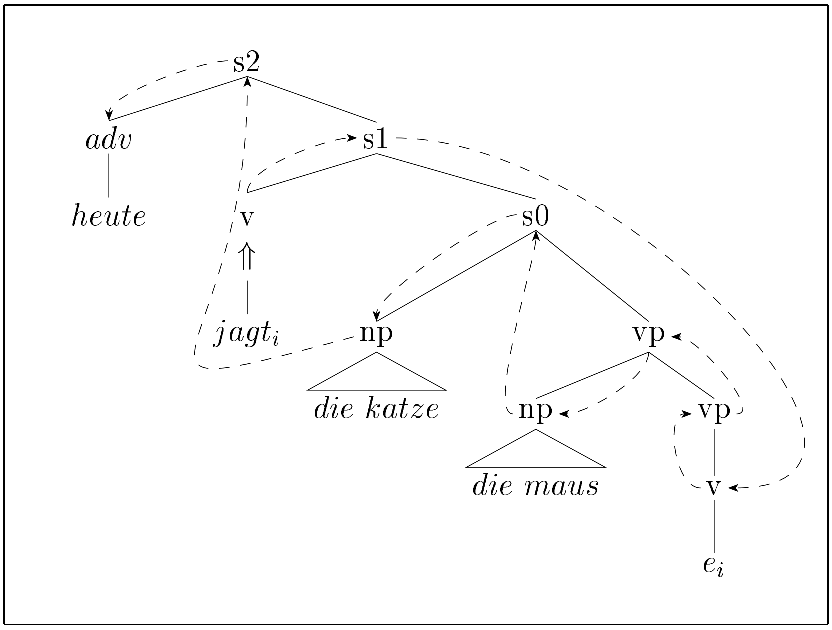


Abbildung 3.7: Vom syntaktischen Kopf gesteuerte Analyse von Grammatiken mit 'bewegten' Köpfen

Die Analyse beginnt mit dem Verb an der zweiten Position des Satzes als syntaktischer Kopf von $s2$. Daraufhin wird ein syntaktischer Kopf als Pivot für die Generierung von $s0$ gesucht. Dies ist das nicht realisierte Verb in der letzten Position, dessen

Subkategorisierungs-Liste aufgrund der Koindizierung mit dem realisierten Verb instantiiert ist. Nun werden die Verbkomplemente und schließlich das Adverb generiert.

Eine Analyse von Nebensätzen im Deutschen, die ein nicht realisiertes Element an der zweiten Position annimmt, ist mit dieser Strategie auch möglich. Wenn sich die Reihenfolge der Generierung nicht ändert, d.h. daß das 'leere' Element als syntaktischer Kopf des Satzes als erstes generiert wird und danach das zu realisierende Verb in der letzten Position, ist dies immer noch ausreichend, um bei Generierung der Verb-Komplemente über eine instantiierte Subkategorisierungs-Liste zu verfügen.

Die Darstellung in König (1994) ist sehr kurz gehalten, so daß einige Punkte offen bleiben. Die Link-Relation in am semantischen Kopf orientierten Algorithmen dient dem top-down-Informations-Transport, indem die Semantik des Pivots instantiiert wird. Wie oben beschrieben, dient eine Ausdehnung der Vorhersage auf syntaktische Information der Verbesserung der Effizienz. Damit die Subkategorisierungs-Liste des leeren Verbs in der in Abbildung 3.7 gezeigten Analyse instantiiert wird, ist es von entscheidender Bedeutung, daß auch syntaktische Information durch die Link-Relation instantiiert wird. Bei der Vorhersage des leeren Verbs als Pivot für die Generierung von $s0$ müssen die syntaktischen Informationen des Verbs in zweiter Position weitergereicht werden, damit die Anwendung der linksrekursiven VP-Komplement-Regel terminiert.

3.2.5 Head-corner-Parsing

Von einer syntactic-head-driven-Generation-Strategie ist es nur ein kleiner Schritt zur Anwendung auf das Parsing. Im Anschluß an Kay (1989) definiert van Noord (1993) einen head-corner-Parser, dessen Prolog-Implementierung sich in van Noord (1997) findet. Implizit geht van Noord von dem *syntaktischen* Kopf aus, der die Analyse leitet. Grundsätzlich arbeitet der Algorithmus wie seine analoge Generierungs-Variante. Hier sollen die wesentlichen Differenzen erläutert werden, die sich letztendlich aus der unterschiedlichen Struktur des Inputs ergeben.

Während bei der Generierung eine semantische Repräsentation gegeben ist, zu der ein Pivot mit kompatibler Semantik gesucht wird, ist es beim Parsing eine Wortkette. Im Gegensatz zum left-corner-Parser, bei dem das jeweils erste Wort der Kette als nächstes verarbeitet wird, muß beim head-corner-Parsing etwas mehr Aufwand getrieben werden, um das nächste Wort zu finden. Die erste Bedingung für einen möglichen Pivot ist die Existenz eines Links zur gesuchten Kategorie, wobei diese Links allein aufgrund der syntaktischen Köpfe berechnet werden. In diesem Punkt entspricht das head-corner-Parsing also der syntactic-head-driven-Generation-Strategie.

Als zweite Bedingung muß ein möglicher Pivot einen Teil des gegebenen Input-Strings 'abdecken' (dazu s.u.), d.h. daß der phonologische String eines lexikalischen Eintrags in der gegebenen Wortkette enthalten sein muß[16] – wobei eben nicht bekannt ist, an welcher

[16] Van Noord (1993) erlaubt weitergehende Operationen auf der phonologischen Kette. Sie dürfen aber keine vorhandenen Teile löschen (d.h. sie müssen *non-erasing* sein). Wenn das Löschen von Wörtern

Stelle ein syntaktischer Kopf zu finden ist. Die zu Beginn der Verarbeitung gegebene Wortkette kann als ein Guide im Sinne von Dymetman *et al.* (1990) betrachtet werden (s. Abschnitt 2.3.3). Dieser steuert durch seine Struktur die Verarbeitung, wobei er mit jedem lexikalischen Zugriff schrittweise 'verbraucht' wird, d.h. wenn ein Wort im Lexikon gefunden wurde, wird dieses aus dem Guide gelöscht. Jeder 'normale' left-to-right-Parser verwendet die Input-Wortkette also in diesem Sinne als Guide. Das Parsing ist erfolgreich, wenn der Guide vollständig verbraucht und die geforderte Zielkategorie erreicht ist (dies schließt ein, daß es keine offenen Ziele mehr gibt, d.h. Tochterknoten von syntaktischen Köpfen, die noch nicht bearbeitet wurden).

Ein head-corner-Parser verarbeitet die gegebene Wortkette nicht von links nach rechts, sondern von dem jeweiligen Kopf aus nach links und nach rechts. Diese Verarbeitungsweise wird auch als *bidirektional* bezeichnet, was im Kontext reversibler Grammatiken mißverständlich sein kann. Beim left-to-right-Parsing steht das nächste zu bearbeitende Element am Anfang der verbleibenden Wortkette. Beim head-corner-Parsing kann der erste syntaktische Kopf, der als Pivot benutzt wird, zwischen der ersten und der letzten Position der Wortliste liegen. Bei der Verarbeitung der Schwesterknoten links bzw. rechts von einem Kopf ist bekannt, daß nur der Teil der Wortkette berücksichtigt werden muß, der auch links bzw. rechts von dem vom Kopf abgedeckten Teil liegt, d.h. die Köpfe unterteilen die Kette immer weiter in jeweils einen linken, einen rechten und einen vom Kopf selbst abgedeckten Teil. Ein head-corner-Parser muß also die Teilkette berechnen, in der ein gesuchtes Wort liegen kann. Dabei gibt es grundsätzlich die Möglichkeit, mit String-Positionen zu arbeiten, oder aber die Wortkette selbst zu verwenden.

3.2.6 Uniformität von Parsing und Generierung

Das Thema einer uniformen Implementierung für Parsing und Generierung wird von van Noord nur einmal erwähnt (van Noord 1990a, S. 15). Dort geht es um die Kombination eines left-corner-Parsers mit einem semantic-head-driven-Generator, wobei in Shieber *et al.* (1990, S.41) zugestanden wird, daß dies eine sehr breite Parametrisierung erfordert. Dort wird auch die Möglichkeit der Verbindung mit einem syntactic-head-driven-Parser erwähnt.

Eine solche Kombination von (syntactic-head-driven-)Parser und (semantic-head-driven-) Generator ist meines Wissens nach noch nicht durchgeführt worden. Da unterschiedliche Konzepte von Kopf verwendet werden, gibt es verschiedene Links für Parsing und Generierung, wenn syntaktischer und semantischer Kopf nicht übereinstimmen. Dies ist für eine uniforme Architektur unbefriedigend, da zur Compile-Zeit zwei verschiedene Tabellen von Links angelegt werden müßten. Ein weiterer Nachteil besteht darin, daß nicht mehr garantiert ist, daß Parser und Generator die gleiche Menge an Phrasen verarbeiten können,

erlaubt wäre, könnte ein Pivot nur noch anhand der syntaktischen Kategorie ausgewählt werden. Jedes Wort mit der gesuchten Kategorie könnte vorhanden gewesen sein, auch wenn es später gelöscht wurde (hier zeigt sich schon der nicht-monotone Charakter solcher String-Operationen). Dadurch würde die Suche wenig zielgerichtet und sehr ineffizient verlaufen.

z.B. wenn die in bezug auf die Analyse von Verbzweitstellung auftretenden Probleme nur eine Verarbeitungs-Richtung betreffen.

Eine Verbindung von syntactic-head-driven-Parsing mit der syntactic-head-driven-Generierung von König (1994) ist meiner Ansicht nach eine geeignete Kombination für eine uniforme Architektur. Dies wird von Esther König auch angedeutet: *"By using the same notion of head both for parsing and for generation, both techniques come even closer. In effect, the abstract specifications of the generation algorithm we gave above, could be read as parsing algorithms, modulo a few changes (of the success condition and the link relation)"* (König 1994, S. 481).

In bezug auf die beiden von König genannten Veränderungen, die gewissermaßen bei einem 'Umschalten' von Generierung auf Parsing vorgenommen werden müßten, stellt sich aber die Frage, ob sie überhaupt notwendig sind. Die Link-Relation wird aufgrund des syntaktischen Kopfes berechnet.[17] Dazu kommt der Substruktur-Test für einen möglichen Pivot, der auf einer abstrakten Ebene für Parsing und Generierung der gleiche ist: entweder die Semantik des Pivots ist eine Teilstruktur der Zielsemantik oder die Wortkette des Pivots ist eine Teilkette der Zielkette. Da komplexe Merkmals-basierte Formalismen verwendet werden, ist im allgemeinen Fall ein Test auf Identität von Atomen unzureichend. Es muß vielmehr verlangt werden, daß der entsprechende Wert des infrage kommenden Pivots mit einem Teil der gegebenen Ziel-Struktur kompatibel ist.[18]

Auch die Kriterien für eine erfolgreiche Bearbeitung sind bei Parsing und Generierung gleich. Die Zielkategorie muß erreicht und die gegebene Merkmalsstruktur zu der im Laufe der Verarbeitung erzeugten kompatibel sein, wobei auf Kohärenz und Vollständigkeit geachtet werden muß (s. Abschnitt 2.2.2).

Wenn die Input-Wortkette beim Parsing als 'Guide' bezeichnet wird, scheint es im Sinne einer uniformen Architektur sinnvoll zu sein, auch bei der Generierung die Input-Struktur, also die semantische Repräsentation, als Guide anzusehen. Zum einen wird dadurch verhindert, daß eine Phrase mehrmals generiert wird – wie auch beim Parsing verhindert wird, daß ein Wort mehrfach abgearbeitet wird. Zum anderen wird dadurch bei beiden Verarbeitungs-Richtungen automatisch das Kriterium der Vollständigkeit erfüllt, da anderenfalls der Guide am Ende der Verarbeitung nicht vollständig verbraucht ist. Es ist jedoch fraglich, ob Guides in jedem Fall auch bei der Generierung explizit implementiert werden müssen, oder ob der Ausdruck 'Guide' eher zum tieferen Verständnis dient (zu einem Vorschlag zum sukzessiven Verbrauch der semantischen Repräsentation s. Abschnitt 6.1.1). Zumindest ist zu beobachten, daß die Kopf-gesteuerte Generierung keine

[17] Darüberhinaus muß sie den top-down-Informations-Transport ermöglichen und zumindest die Semantik weiterreichen – es sei auf oben angesprochene Erweiterung der Vorhersage verwiesen.

[18] Unter der Annahme, daß die Input-Merkmalsstruktur für PHON bzw. SEM voll instantiiert ist, kann verlangt werden, daß der Wert des betreffenden Merkmals des Pivots eine Teilstruktur des entsprechenden Merkmalswertes der Ziel-Merkmalsstruktur subsumiert, d.h. der Pivot ist genereller (dies gilt natürlich vor dem top-down-Informations-Transport). Beispielsweise ist der Wert für die Semantik der Verbargumente im Lexikoneintrag eines Verbs nicht instantiiert (s. z.B. (15)), in einer voll spezifizierten Input-Semantik dagegen schon. Bei der Einbeziehung unterspezifizierter Input-Merkmalsstrukturen kann diese Voraussetzung jedoch nicht mehr gemacht werden.

explizite Weitergabe einer Reststruktur benötigt. Die Frage, ob eine Phrase die Input-Semantik vollständig abdeckt, oder ob (durch Backtracking) weitere Lösungversuche notwendig sind, kann durch einen Subsumptions-Test beantwortet werden. Wie in Abschnitt 2.2.2 erwähnt, muß bei der Kopf-gesteuerten Generierung darauf geachtet werden, daß nur die Information aus der Grammatik, nicht aber die durch die top-down-Vorhersage hinzugefügte berücksichtigt wird.

Die Verwendung des syntaktischen Kopfes für Parsing *und* Generierung bewirkt, daß die Behandlung von 'bewegten' Köpfen bei Parsing und Generierung gleich ist. Wenn bei der Generierung vom semantischen Kopf ausgegangen wird, muß entweder ein Verfahren wie die vorzeitige Generierung des 'bewegten' Kopfes eingeführt werden (s.o.), wodurch die Uniformität beeinträchtigt wird, oder aber es besteht die Gefahr, daß die Menge der generierbaren Sätze nicht mehr gleich der der analysierbaren ist.

Inkrementelle Verarbeitung

Eine Verarbeitungs-Strategie ist *inkrementell*, wenn sie jede Teil-Information sofort verarbeiten kann, sobald diese vorhanden ist, d.h. daß z.B. beim Parsing nicht gewartet werden muß, bis der Satz beendet worden ist. Die möglichen Schlußfolgerungen aus einer Teil-Information werden also gezogen, sobald diese 'wahrgenommen' wird. Darüberhinaus gibt es den Begriff der *'vollen Inkrementalität'*, der zusätzlich das Löschen schon verarbeiteter Information erlaubt, d.h. daß gezogene Schlußfolgerungen u.U. rückgängig gemacht werden müssen (Erbach 1995a, S. 95). Dieser letzte Fall soll hier aber nicht berücksichtigt werden.

Beim Parsing gibt es eindeutig zu unterscheidende Informationsteile (die Wörter), die zudem in einer eindeutigen linearen Abfolge stehen, so daß eine sukzessive Eingabe von links nach rechts (aber im Prinzip auch umgekehrt) möglich ist. Die Fähigkeit zu inkrementeller Verarbeitung steht also in engem Zusammenhang mit der Abarbeitung der Wortkette von links nach rechts (deshalb auch *left-right incrementality* genannt). Angenommen, beim Parsing wurde das zuletzt eingegebene Wort als ein Determiner analysiert. Eine top-down- oder left-corner-Analysestrategie erwartet nun als nächstes voraussichtlich ein Nomen und kann diese Vorhersage überprüfen, wenn das nächste Wort eingegeben wurde. Die bottom-up-Analyse hat keine Erwartungen an das nächste Wort und zieht aus jedem gegebenen die möglichen Inferenzen.[19]

Ein head-corner-Parser kann insofern nicht inkrementell arbeiten, als er die Wortkette nicht von links nach rechts verarbeitet. Er ist darauf angewiesen, daß der syntaktische Kopf vorhanden ist, um ihn als Pivot verwenden zu können.

Allgemein gesehen entsteht die Fähigkeit zu inkrementeller Verarbeitung aus der Parallelität zwischen Eingabe-Reihenfolge und Verarbeitungs-Reihenfolge. Bei left-to-right-

[19] Wenn alle Inferenzen aus dem zuletzt eingegebenen Wort gezogen werden können, bevor das nächste vollständig eingegeben ist, entsteht für einen menschlichen Benutzer der Eindruck, das Programm arbeite 'in Echtzeit'.

Parsern entspricht die Abarbeitungs-Reihenfolge der Wortkette auch der Reihenfolge, in der die Wortkette eingegeben wird. Wenn also einem head-corner-Parser die Worte in genau der 'richtigen' Reihenfolge eingegeben werden (also zuerst der syntaktische Kopf des Satzes, dann die Köpfe der Schwesterknoten des 'untersten' lokalen Baumes usw.), dann kann auch hier von inkrementeller Verabeitung gesprochen werden.

In bezug auf inkrementelle Generierung stellt sich die Frage, in welcher Reihenfolge die semantische Repräsentation der taktischen Komponente bekannt wird (s.a. Abschnitt 6.1.1). Wenn z.B. Funktoren vor ihren Argumenten zur Verfügung stehen, scheint eine semantic-head-driven-Generation-Strategie geeignet zu sein. Die Frage nach der gewünschten Eingabe-Reihenfolge von Teilstrukturen ist für die Generierung wesentlich schwerer zu beantworten als für das Parsing, da es keine vergleichbare Evidenz zu der offensichtlichen Beobachtung gibt, daß Sätze (bzw. Texte im allgemeinen) linear organisiert sind.

3.3 Zusammenfassung und Einordnung

Kopf-gesteuerte Analysestrategien stellen eine Kombination aus top-down-Vorhersage und bottom-up-Verarbeitung dar. Sie können als stärker bottom-up-orientiert bezeichnet werden, da der Informationfluß zwischen den Regeln bei den bottom-up-Schritten voll ausgenutzt wird, während bei der top-down-Vorhersage nur ein sehr kleiner Teil der in den Grammatikregeln kodierten Information verwendet wird (auch wenn der Informationsfluß ausgedehnt werden kann). Bei der semantic-head-driven-Generierung wird für die Vorhersage zumindest Information über die Koindizierung der Semantik von Mutterknoten und (zumeist) einem Tochterknoten benötigt. Beim head-corner-Parsing und der syntactic-head-driven-Generierung ist nur die Information über die syntaktische Kategorie des Kopfes einer Grammatikregel notwendig.

Das in Abschnitt 1.3 vorgestellte Klassifikations-Schema sieht für die in dieser Arbeit bevorzugten Architekturen vom Typ D vor, daß kein Kompilierungs-Schritt notwendig sein darf. Die gesuchte Kopf-Relation kann grundsätzlich auch zur Laufzeit berechnet werden (auch wenn dies ineffizienter ist). Aber auch die Berechnung der Link-Relation zur Compile-Zeit verändert die Grammatik selbst nicht, sondern dient der Gewinnung von Information über die Grammatik. Für die Bestimmung des syntaktischen Kopfes sind dabei u.U. zusätzliche Annotationen der Regeln erforderlich. Das gleiche gilt für den semantischen Kopf, wenn es mehrere infrage kommende Elemente gibt.

Eine uniforme Implementierung der Kopf-gesteuerten Analysestrategie ist meiner Ansicht nach möglich, auch wenn dies bisher offenbar noch nicht geschehen ist. Ein besonders hoher Grad an Uniformität wird erreicht, wenn bei Parsing und Generierung die gleiche am syntaktischen Kopf orientierte Link-Relation verwendet wird. Sie erlaubt auch eine einheitliche und gegenüber der semantic-head-driven-Generation-Strategie überlegene Behandlung von 'bewegten' Köpfen, sofern eine solche Analyse unter linguistischen Gesichtspunkten angestrebt wird. Wenn die Generierung am syntaktischen Kopf orientiert ist, hat dies jedoch den Nachteil, daß ein möglicher Pivot nicht in jedem Fall die ganze

Zielsemantik ‘abdeckt’, sondern nur eine Teilstruktur. Dadurch wird die Suche nach einem Pivot aufwendiger.

Sowohl Parsing wie auch Generierung mit Kopf-gesteuerten Algorithmen sind sehr effizient, da z.B. die Subkategorisierungs-Information des Verbs für die Bearbeitung der Komplemente zur Verfügung steht. Es ist zu erwarten, daß ein head-corner-Parser einen Effizienz-Vorteil gegenüber einem left-corner-Parser hat, weil beispielsweise eine Subjekt-NP *peter* am Satzanfang nur in dem einen, zutreffenden Kasus analysiert wird, und nicht in allen morphologisch möglichen. Zudem ist anzumerken, daß durch die Verwendung von Kopf-gesteuerter Generierung das Verhältnis zwischen Parsing und Generierung in bezug auf die Effienz tatsächlich ausgeglichen erscheint, was bei der traditionell stärkeren Berücksichtigung des Parsings nicht selbstverständlich ist.

Die Einschränkungen, die sich aus der Verwendung von Kopf-gesteuerten Analysestrategien für die möglichen Grammatiken ergeben, sind nicht sehr stark, da in vielen Grammatiktheorien ein ‘Kopf-Prinzip’ angenommen wird. So können z.B. auch semantisch nicht kompositionelle Grammatiken verarbeitet werden.

Die Baum-Traversierung ist wie beim left-corner-Parsing in-order, wobei aber die Wortkette nicht von links nach rechts verarbeitet wird. Daraus ergeben sich Probleme für eine inkrementelle Verarbeitung. Die Selektionsfunktion wählt den semantischen bzw. den syntaktischen Kopf; die Verarbeitungsrichtung innerhalb der Regeln ist bottom-up.

Die Implementierungen der vorgestellten Analyse-Strategien von van Noord (1993, 1997) gehen von einer Tiefensuche-Kontrollstrategie mit Backtracking aus. Daraus entsteht die Gefahr, daß durch Backtracking schon gewonnene Ergebnisse wieder verloren werden. Dies kann durch die zusätzliche Speicherung von bearbeiteten Teilzielen verhindert werden (s. z.B. van Noord 1996, S. 12).

Eine alternative Kontrollstrategie ist die Breitensuche, die in Verbindung mit der Speicherung von Teilergebnissen für das Earley-Deduktions-Verfahren kennzeichenend ist. Die Grundprinzipien von Earley-Deduktion sowie darauf aufbauende Algorithmen für Parsing und Generierung werden im folgenden Kapitel vorgestellt.

Kapitel 4

Parsing und Generierung durch Earley-Deduktion

Die Grundlage für die Verbindung von Analysestrategien und Logik-Programmierung ist die Beschreibung von Grammatiken durch *Definite Klauseln*. Eine Analyse ist dann erfolgreich, wenn der gegebene Input aus der Grammatik ableitbar ist, wobei die Struktur des Beweisbaumes der des Analysebaumes entspricht. Historisch gesehen ist es sogar so, daß das Prolog-Beweisverfahren nach einem Parsing-Algorithmus modelliert wurde (Pereira und Warren 1983, S.138).

Der theoretische Vorteil einer solchen Sichtweise liegt in der klaren Trennung zwischen der Grammatik (oder allgemein dem Programm) und dem Interpreter, der diese Grammatik zur Durchführung des Beweises verwendet. Die Arbeitsweise von Interpretern läßt sich durch *Inferenzregeln* beschreiben. Daraus folgt, daß Analysestrategien durch Inferenzregeln für Definite Klauseln charakterisiert werden können (zusätzlich muß auch eine Kontrollstrategie angegeben werden). Zuvor soll jedoch kurz erläutert werden, was Definite Klauseln sind.

4.1 Definite Klauseln

Die allgemeine Form Definiter Klauseln, die hier behandelt werden sollen, ist

$$A \leftarrow \; B_1 \wedge ... \wedge B_n,$$

wobei A als *positives Literal* bezeichnet wird und B_1 bis B_n als *negative Literale*. Eine solche Definite Klausel sagt aus, daß A behauptet werden darf, wenn B_1 bis B_n bewiesen werden können.[1] Es handelt sich um eine (implizit allquantifizierte) logische Implikation.

[1]Die Bezeichnung 'negatives' bzw. 'positives' Literal bezieht sich darauf, daß eine Implikation in eine Disjunktion umgewandelt werden kann, wobei das Vorderglied negiert wird: $A \vee \neg B_1 \vee ... \vee \neg B_n$. Der Beweis einer solchen Formel wird dadurch geführt, daß der Versuch ihrer Widerlegung *(refutation)* scheitert. Zum Zusammenhang zwischen Prolog als logischer Programmiersprache, die Definite Klauseln verwendet, und Logik s. Clocksin und Mellish (1984, Kapitel 10).

Die linke Seite einer Definiten Klausel wird auch als *Kopf (head)* bezeichnet, die rechte Seite als *Rumpf (body)*. Damit keine Verwechselungen mit dem Ausdruck 'semantischer' bzw. 'syntaktischer Kopf' in der in den vorhergehenden Kapiteln verwendeten Bedeutung entstehen, wird hier der Ausdruck 'Klauselkopf' für die linke Seite verwendet, wenn dies der Kontext erfordert. Eine Klausel besteht immer aus einem Klauselkopf und einer beliebigen Anzahl von Elementen im Rumpf. Durch die Einschränkung des positiven Literals auf ein Element wird die Menge der hier in betracht gezogenen Definiten Klauseln auf eine Teilmenge der Prädikatenlogik erster Stufe, die sog. *Horn Klauseln* beschränkt (Clocksin und Mellish 1984, S. 249). Wenn es mehr als ein Element im Rumpf gibt, handelt es sich um eine *Nonunit*-Klausel; wenn der Rumpf kein Element enthält, handelt es sich um eine *Unit*-Klausel. Letztere ist gewissermaßen ein Fakt oder eine Proposition, die von keiner weiteren Bedingung abhängig ist. Ein Programm besteht aus einer endlichen Menge von Unit- und Nonunit-Klauseln.

Die einzelnen Literale bestehen aus Prädikaten mit einer endlichen Anzahl von Argumenten: $p(a_1...a_k)$. Die Anzahl der Argumente wird als die *Stelligkeit* des Prädikats bezeichnet. Ein Argument ist ein *Term*, der eine Konstante, eine Variable oder ein zusammengesetzter Term sein kann. Letzterer besteht wiederum aus einem Funktor mit einer endlichen Anzahl von Termen als Argumenten.

4.1.1 Kontextfreie Grammatiken und Deduktion

Eine kontextfreie Grammatik kann durch eine Menge von Definiten Klauseln aus Prädikaten der Stelligkeit null ausgedrückt werden. Beispielsweise wird durch $s \leftarrow \ np \wedge vp$ festgelegt, daß s angenommen werden darf, wenn np und vp bewiesen sind. Dies entspricht einer kontextfreien Grammatikregel der folgenden Form: $S \rightarrow NP\ VP$. Eine NP darf beispielsweise behauptet werden, wenn Determiner und Nomen bewiesen sind, und eine VP darf angenommen werden, wenn ein V bewiesen ist: $np \leftarrow \ det \wedge n$ bzw. $vp \leftarrow \ v$.

In dieser Darstellung wurde schon die Definition von Nonunit-Klauseln als Implikationen zur Operationalisierung der Beweisfindung verwendet. Um ein Ziel G zu beweisen, muß gezeigt werden, daß G eine logische Konsequenz aus dem Programm ist. Dazu wird ein Klauselkopf gesucht, der mit dem Ziel *identisch* ist. Nun muß gezeigt werden, daß auch die Elemente im Rumpf der Klausel logische Konsequenzen des Programms sind. Dieses Verfahren wird rekursiv angewendet, wobei ein Klauselkopf stets durch die Elemente im Rumpf ersetzt werden kann. Dies kann als ein ständiges Neuschreiben bzw. Ersetzen des Antezedens aufgefaßt werden *(rewriting)*. Der Beweis ist beendet, wenn alle Elemente durch Unit-Klauseln ersetzt wurden, d.h. es gibt keine Bedingungen mehr, die bewiesen werden müssen, um das Konsequens behaupten zu dürfen.

In kontextfreien Grammatiken sind die Unit-Klauseln, die als Abbruchbedingung für die rekursive Ersetzung der Elemente im Klauselrumpf dienen, Lexikoneinträge (z.B. *die, katze* und *schläft*). Die Ersetzungsregel *(resolution rule)* für Definite Klauseln ist die fundamentale Inferenzregel, die zum Beweis eines Zieles benötigt wird (Pereira und Warren 1983,

S. 138). Sie soll nun in bezug auf die oben angegebene kontextfreie Grammatik erläutert werden (wobei zusätzlich die folgenden Regeln verwendet werden: $det \leftarrow die$, $n \leftarrow katze$ und $v \leftarrow schläft$).

Gegeben ist ein Ziel *(goal statement)* $\leftarrow\ s$, für das nun die folgenden Ersetztungen stattfinden, wobei pro Zeile ein Schritt angegeben wird:

(26)
$$\begin{array}{l} \leftarrow\ s \\ s \leftarrow np \wedge vp \\ s \leftarrow det \wedge n \wedge vp \\ s \leftarrow \texttt{die} \wedge n \wedge vp \\ s \leftarrow \texttt{die} \wedge \texttt{katze} \wedge vp \\ s \leftarrow \texttt{die} \wedge \texttt{katze} \wedge v \\ s \leftarrow \texttt{die} \wedge \texttt{katze} \wedge \texttt{schläft} \end{array}$$

Die letzte Zeile besagt, daß, wenn *die*, *katze* und *schläft* gegeben ist, ein s abgeleitet werden kann (das Antezedens steht auf der rechten Seite). Die Ersetzungsregel arbeitet jedoch in umgekehrter Richtung (gewissermaßen entgegen der Richtung des Implikationspfeils): von dem Konsequens wird zum Antezedens übergegangen, d.h. das gegebene Ziel wird 'rückwärts' verfolgt bis es keine Nicht-Terminalsymbole (d.h. Köpfe von Nonunit-Klauseln) mehr gibt *(backward chaining)*.

Der Ableitungsbaum entspricht dem Analysebaum für die gegebene Wortkette, wobei die Baumtraversierung top-down ist. Darüberhinaus finden in (26) die Ersetzungen 'auf einer Ebene' nicht gleichzeitig statt, sondern gehen zuerst in die Tiefe, wobei immer das am weitesten links stehende noch nicht durch ein Terminalsymbol (also eine Unit-Klausel) ersetzte Element bearbeitet wird. Während eine top-down-Analyse also durch backward chaining durchgeführt wird, kann analog dazu eine bottom-up-Analyse durch *forward chaining* vollzogen werden, d.h. Verarbeitung in Richtung des Implikationspfeils (dazu s. Abschnitt 4.4).

Generalisierung der Inferenzregel

Bisher wurden nur Prädikate der Stelligkeit null verwendet, so daß ein Test auf Identität ausreicht, um einen passenden neuen Klauselkopf für ein Element im Klauselrumpf zu finden. Wenn dagegen Argumente, die Terme enthalten, verwendet werden, müssen die Literale unifiziert werden können. Die Inferenzregel kann dann in allgemeiner Form folgendermaßen formuliert werden:

(27)
$$\begin{array}{l} A \leftarrow\ B_1 \wedge ... \wedge B_n \\ C \leftarrow\ D_1 \wedge ... \wedge D_n \end{array}$$

Wenn B_i und C unifizierbar sind, kann eine Substitution erfolgen, so daß sich

(28)
$$\phi[A \leftarrow\ B_1 \wedge ... \wedge B_{i-1} \wedge D_1 \wedge ... \wedge D_n \wedge B_{i+1} \wedge ... \wedge B_n]$$

ergibt. Die *abgeleitete Klausel* (28) ist der Resolvent der beiden Klauseln in (27). Mit 'ϕ' wird der *most general unifier (mgu)* bezeichnet. Dadurch werden die durch die Unifikation von B_i und C erfolgten Informationsveränderungen erfaßt, beispielsweise Instantiierungen durch koindizierte Variablen auch in anderen Elementen. Es wird jedoch keine weitere Information außer der der beteiligten Elemente hinzugefügt (deswegen handelt es sich um den *generellsten* Ausdruck).

Bei der Verwendung nullstelliger Prädikate besteht keine Möglichkeit, Information innerhalb der Grammatikregeln weiterzureichen. *Definite Clause Grammars (DCGs)* verwenden zur Weitergabe der Wortkette zwei Argumente, wobei das erste Argument den Input-String und das zweite den Output-String enthält. Der durch den jeweiligen Knoten abgedeckte Teil der Wortkette ergibt sich also aus der Differenz der beiden Ketten (Differenzliste). Eine kontextfreie Grammatikregel $S \rightarrow NP\ VP$ wird beispielweise in folgende Definite Klausel übersetzt: $s(S_0, S_2) \leftarrow\ np(S_0, S_1) \wedge vp(S_1, S_2)$. Darüberhinaus können in DCGs weitere Argumente angegeben werden, die bei der automatischen Übersetzung, die in vielen Prolog-Implementierungen erfolgt, meistens vor die String-Argumente gesetzt werden.

Statt verschiedene Informationen an unterschiedlichen Argumentpositionen weiterzureichen, kann die ganze Information auch in einer einzigen Merkmalsstruktur, die als Liste kodiert an der ersten (und einzigen) Argumentposition steht, erfaßt werden. Neumann (1994) und van Noord (1993) verwenden ein Prädikat `sign/1`, das als Argument eine Merkmalsstruktur besitzt, die die linguistische Information enthält. Eine Grammatikregel kann dann beispielsweise in einer Definiten Klausel der folgenden Form angegeben werden, wobei zur besseren Lesbarkeit keine Listen, sondern Merkmalsstrukturen verwendet werden (zur Kodierung von Merkmalsstrukturen in Listen s. Abschnitt 5.1):

$$(29)\ \ \text{sign}(\begin{bmatrix}\text{phon} & P_0 - P \\ \text{syn} & \begin{bmatrix}\text{cat} & \text{vp}\end{bmatrix} \\ \text{sem} & \begin{bmatrix}\text{mod} & \text{Mod} \\ \text{arg1} & \text{Pred}\end{bmatrix}\end{bmatrix}) \longleftarrow \text{sign}(\begin{bmatrix}\text{phon} & P_0 - P_1 \\ \text{syn} & \begin{bmatrix}\text{cat} & \text{ap}\end{bmatrix} \\ \text{sem} & \text{Mod}\end{bmatrix}) \bigwedge \text{sign}(\begin{bmatrix}\text{phon} & P_1 - P \\ \text{syn} & \begin{bmatrix}\text{cat} & \text{vp}\end{bmatrix} \\ \text{sem} & \text{Pred}\end{bmatrix}).$$

Die Information über die syntaktische Kategorie (AP steht beispielsweise für 'Adverbialphrase') ist ein Merkmal innerhalb der Merkmalsstrukur und nicht wie in einer DCG im Prädikatsnamen kodiert. Wenn aber nur das Prädikat `sign/1` in einer Grammatik vorkommt, ist es nicht nötig, dieses ausdrücklich zu erwähnen, da es keine zusätzlichen Constraints beiträgt (vgl. Erbach 1995, S. 13). Auf die Angabe eines Prädikats als 'Träger' einer Merkmalsstruktur wird deshalb hier verzichtet, so daß Grammtikregeln wie bisher z.B. in (14) und (18) geschrieben werden können. Dies hat – wie auch die Verwendung logischer Konnektive – den Vorteil, daß von der Implementierung in einer konkreten Programmiersprache abstrahiert wird.

4.2 Memoing

Eine Suchstrategie mit Backtracking, wie sie beispielsweise von *BUG*, *SHDGA* und dem Standard-Prolog-Interpreter verwendet wird, verfolgt einen Zweig des Beweisbaumes zur Zeit. Wenn die Analyse an einem Punkt nicht weitergeführt werden kann, erfolgt Backtracking zum zuletzt gesetzten *Choice Point* und die noch nicht untersuchte Alternative wird verfolgt. Der Nachteil dieser Methode ist, daß richtige Teilergebnisse verloren werden können, so daß sie im Laufe der Analyse noch einmal erarbeitet werden müssen. Diese mehrfache Arbeit kann durch Techniken der Speicherung von Zwischenergebnissen verhindert werden, die als *Memoing*, *Memoization* oder *Tabulation* bezeichnet werden.

Eine Möglichkeit zur Realisierung dieser Techniken ist Ergänzung eines Backtracking-Interpreters mit der Fähigkeit, Teilergebnisse zu speichern. Vor der Bearbeitung eines neuen Zieles wird geprüft, ob dieses Ziel schon bearbeitet wurde. Van Noord (1997, S.12ff) verwendet bei der Implementierung eines head-corner-Parsers zwei Tabellen, deren eine erfolgreiche Teilziele mit ihren Ergebnissen speichert, während die andere bearbeitete, aber erfolglose Ziele speichert – denn auch Fehlversuche sollen nicht wiederholt werden (vgl. auch Covington 1994, S. 171f).

Obwohl diese Realisierung von Memoing in der Praxis erfolgreich ist, wird weiterhin eine Tiefensuche-Kontrollstrategie mit Backtracking verwendet. Es gibt jedoch ein Deduktionsverfahren für Definite Klauseln, das Memoing verwendet und sowohl Breitensuche wie auch Tiefensuche ohne Backtracking modelliert. Teilergebnisse aus einem Zweig des Beweisbaumes können in anderen Zweigen verwendet werden. Verschiedene Kontroll- und Analysestrategien ergeben sich dabei aus der Festlegung der Reihenfolge für die Anwendung einiger weniger elementarer Inferenzregeln. Dieses Deduktionsverfahren läßt sich auf Definite Klauseln im allgemeinen ausweiten, so daß eine systematische Beschreibung von Verarbeitungsstrategien im Rahmen der Logik-Programmierung möglich ist.

Zunächst wird im folgenden das Chart-Parsing als allgemeines Schema für Parsing-Strategien dargestellt. Darauf wird Earley-Deduktion als allgemeines Verfahren zur Verarbeitung Definiter Klauseln erläutert.

4.2.1 Chart-Parsing

Der Zustand eines Chart-Parsers wird durch eine *Tabelle (Chart)* aus *Kanten* repräsentiert. Jede Kante ist ein gerichteter azyklischer Graph von einer Position in der Wortkette zu einer anderen (ein weiter unten behandelter Grenzfall tritt auf, wenn beide Positionen identisch sind). Der zwischen diesen beiden Positionen liegende Teil der Wortkette ist derjenige, der schon erfolgreich analysiert worden ist. Bei der Bearbeitung einer Grammatikregel können zwei Fälle unterschieden werden: entweder alle Elemente der rechten Seite wurden erfolgreich abgearbeitet *(passive Kante)* oder es verbleiben noch Elemente, die nicht 'innerhalb der Kante' liegen *(aktive Kante)*. Neben einer graphischen Darstellung durch Kanten kann der Zustand des Parsers auch durch *Items* repräsentiert werden, die angeben, inwieweit die Grammatikregeln schon erfolgreich abgearbeitet wurden. Je nachdem, ob die Items aktive oder passive Kanten beschreiben, wird auch von *aktiven Items* bzw. *passiven Items* gesprochen. Zur Kennzeichnung der Position, bis zu der die Kante reicht, wird ein Punkt auf der rechten Regelseite verwendet (s. z.B. (30)).

Wenn nur erfolgreich abgearbeitete Items in die Tabelle aufgenommen werden, handelt es sich um einen *passiven Chart-Parser*. Dessen Chart enthält keine Information über widerlegte Hyphothesen, d.h. erfolglose Versuche, eine Phrase in einer bestimmten Weise zu analysieren. Fehlversuche können also im ungünstigesten Fall beliebig oft wiederholt werden. Die Interpretation der Tabelle wäre in einem solchen Fall nicht eindeutig möglich, da ein nicht vorhandenes Ergebnis entweder bedeuten kann, daß es noch gar nicht erzielt wurde, oder aber, daß der entsprechende Analyseversuch gescheitert ist. Darüberhinaus findet kein Informationsaustausch zwischen gleichzeitig durchgeführten Analysen statt, so daß auch in einem solchen Fall nach wie vor Arbeit mehrfach geleistet werden muß. Deswegen sollen im weiteren *aktive Chart-Parser* berücksichtigt werden, die sowohl passive wie auch aktive Items speichern, d.h. daß zusätzlich zu den erfolgreichen Analysen auch alle Analyseversuche gespeichert werden.

Die Verwaltung von Alternativen im Beweisbaum geschieht nicht durch das Setzen eines Choice Points wie beim Backtracking, sondern durch die explizite Aufnahme der Alternativen in die Tabelle. Die alternativen Beweispfade kommunizieren gewissermaßen über die Chart. Dabei ist es wichtig, daß tatsächlich *alle* Alternativen ausgeschöpft werden, denn ansonsten ist der Parser nicht *vollständig*. Der Begriff der 'Vollständigkeit' (der Tabelle) darf jedoch nicht mit dem der 'Vollständigkeit von Parsing und Generierung' verwechselt werden.

Die Ergänzung eines 'normalen' Interpreters mit Tiefensuche und Backtracking um eine zusätzliche Tabelle ist dagegen nur für feststehende Teilergebnisse sinnvoll. Es kann gespeichert werden, ob und welche Lösungen ein Ziel hat oder daß es nicht erfolgreich bearbeitet werden kann. Auch hier muß die Vollständigkeit der Tabelle verlangt werden. Diese Methode kann in bezug auf die Menge der gespeicherten Information also zwischen einem (sehr simplen) passiven Chart-Parser und einem aktiven Chart-Parser eingeordnet werden.

Das Grundprinzip beim Chart-Parsing besteht darin, daß ein passives Item dazu verwendet werden kann, den Punkt in einem aktiven Item um ein Element vorzurücken. Dies kann nur unter der Bedingung geschehen, daß das rechts des Punktes stehende Element mit der linken Regelseite des passiven Items unifiziert. Aus dieser Vorgehensweise ergibt sich, daß die Wortkette von links nach rechts abgearbeitet wird.

Im folgenden werden einige Beispiele für Items gegeben, die im Laufe einer Analyse von $_0\ die\ _1\ Katze\ _2\ schläft\ _3$ auftreten könnten (die Ziffern stehen für String-Positionen, wobei in einem Item jeweils die Anfangs- und Endposition des schon analysierten Teils der Phrase angegeben werden):

(30) a. $NP \rightarrow\ Det\ N\ \bullet\ \ < 0\ 2 >$

b. $S \rightarrow NP \bullet VP \; < 0\ 2 >$

c. $VP \rightarrow \; \bullet V \; < 0\ 0 >$

Das passive Item (30a) repräsentiert eine NP, die vollständig analysiert worden ist. Deswegen kann die linke Seite dieses Items beispielsweise dazu verwendet werden, in dem aktiven Item (30b) den Punkt bis hinter die NP vorzurücken. Item (30b) sagt aus, daß eine NP analysiert wurde und daß ein Satz gefunden wäre, wenn nun eine VP geparst werden kann. Da die VP noch nicht bearbeitet wurde, steht der Punkt in Item (30c) noch links des Verbs. Ein solches noch nicht verwendetes aktives Item wird auch als *leeres Item* bezeichnet. Der gerichtete Graph beginnt und endet an der ersten Position (in diesem Sonderfall ist der Graph nicht mehr azyklisch).

Die Kombination eines passiven Items und eines aktiven Items wird in der *fundamentalen Regel* des Chart-Parsings in Form einer Inferenzregel explizit gemacht, wobei die griechischen Buchstaben Sequenzen von Elementen bezeichnen:

(31)

$$\frac{\begin{array}{c} N_1 \rightarrow \; \alpha \; \bullet \; N_2 \; \beta \; < S_1 \; S_2 > \\ N_3 \rightarrow \; \gamma \; \bullet \; < S_2 \; S_3 > \end{array}}{\phi[N_1 \rightarrow \; \alpha \; (N_2 \sqcup N_3) \; \bullet \; \beta \;] \; < S_1 \; S_3 >}$$

Die wesentliche Bedingung für die Anwendung der fundamentalen Inferenzregel in (31) ist die Unifikation von N_2 und N_3 (der Ausdruck '$(N_2 \sqcup N_3)$' in der letzten Zeile referiert auf das Objekt als Ergebnis dieser Unifikation). Durch 'ϕ' werden die Informationsveränderungen auch in den anderen Elementen der Ergebnis-Klausel erfaßt (s. 28). Durch die Anwendung der fundamentalen Regel kann das beteiligte aktive Item zu einem passiven Item werden, wenn β leer ist, d.h. daß das letzte Element einer rechten Regelseite erfolgreich bearbeitet wurde.

Da ein Chart-Parser nur die Items zur Verfügung hat, die in der Tabelle stehen (und nicht automatisch die ganze Grammatik) müssen neben der fundamentalen Regel weitere (Pseudo-)Inferenzregeln eingeführt werden, die lexikalische Einträge und Grammatikregeln auswählen, um ihre Verwendung zu ermöglichen. Die *Vorhersage*-Regel *(prediction rule)* wählt eine Grammatikregel top-down aus:

(32)

$$\frac{\begin{array}{c} N_1 \rightarrow \; \alpha \; \bullet \; N_2 \; \beta \quad < S_1 \; S_2 > \\ N_3 \rightarrow \; \gamma \end{array}}{\phi[(N_2 \sqcup N_3) \rightarrow \; \bullet \; \gamma \;] \quad < S_2 \; S_2 >}$$

Wenn das zu bearbeitende Element N_2 mit dem Kopf einer Grammatikregel N_3 unifiziert, dann darf diese Grammatikregel als leeres Item in die Tabelle aufgenommen werden. Es beginnt (und endet) an der Position, bis zu der der von dem aktiven Item abgedeckte Teil reicht. Lexikalische Einträge werden durch die *Scanning*-Regel ausgewählt:

(33)

$$\frac{\begin{array}{c} N_1 \rightarrow \ \alpha \ \bullet \ N_2 \ \beta \ < S_1 \ S_2 > \\ N_3 \rightarrow \ T \end{array}}{\phi[(N_2 \sqcup N_3) \rightarrow \ T \ \bullet \ ] \ < S_2 \ S_{2+1} >}$$

Unter der Bedingung, daß N_2 und N_3 unifizieren, wobei N_3 ein Präterminalsymbol ist, und wenn darüberhinaus gilt, daß das Terminalsymbol T das erste Element in der Wortliste von N_2 ist, darf der lexikalische Eintrag als passives Item in die Tabelle eingefügt werden.

4.2.2 Modellierung von Parsingstrategien

Durch eine Veränderung des Zeitpunktes, an dem Grammatikregeln und lexikalische Einträge bearbeitet und in die Tabelle eingefügt werden, können verschiedene Analysestrategien modelliert werden. Daraus ergibt sich, daß Chart-Parsing und das daraus abgeleitete Deduktionsverfahren (s.u.) einen geeigneten Rahmen zur Modellierung verschiedener Analysesstrategien bilden. Hier soll zunächst ein Überblick gegeben werden. In Abschnitt 6.2 wird darüberhinaus die Möglichkeit eines allgemeinen Rahmens für die verschiedenen Analyse-Strategien angesprochen.

Ein **top-down-Parser** wählt für das zu bearbeitende Element der rechten Seite einen Lexikoneintrag bzw. eine Grammatikregel aus und bearbeitet im zweiten Fall deren rechts des Punktes stehendes Element. Zu Beginn des Parsings muß sich im *Startzustand* nur ein (triviales) Start-Item befinden, das Informationen über die gesuchte syntaktische Kategorie und die Wortkette enthält. Hier wird als Beispiel ein Start-Item in DCG-Notation angegeben, wobei die ersten beiden Argumente für die String-Positionen stehen:

$$ans \rightarrow \ \bullet \ s(0, 3, [die, katze, schläft], [\,])$$

Wenn es gelingt, den Satz zu parsen, ist *ans* das Ergebnis der Analyse. Ein Start-Item muß deshalb verwendet werden, weil nur ein aktives Item die Vorhersage z.B. der S-Regel auslösen kann. Dazu wird ein Element rechts von einem Punkt in einem aktiven Item benötigt. Es finden solange top-down-Vorhersagen statt, bis ein Lexikoneintrag gefunden wird, der als Unit-Item zur Reduktion eines aktiven Items verwendet werden kann. Wenn dieses aktive Item schließlich zu einem passiven reduziert wurde, kann es selbst wiederum zur Reduktion eines aktiven Items dienen. Die Anwendung der fundamentalen Regel kann als ein bottom-up-Schritt aufgefaßt werden, der die durch die Vorhersage aufgestellte Hypothese bestätigt. Wenn schließlich ein passives Item für einen Satz gefunden wurde, kann als letzter Schritt der Punkt in dem Start-Item nach rechts verschoben werden, so daß dieses zu einem passiven Item wird.

Ein **bottom-up-Parser** beginnt mit dem passivem Item für das am weitesten links stehende Wort der Kette. Unter Anwendung der fundamentalen Regel wird dies zur Reduktion eines aktiven Items verwendet. Nun wird – falls nötig – das passive Item für das zweite Wort zur Reduktion des aktiven Items verwendet. Erst wenn auf diese Weise das aktive Item in ein passives verwandelt wurde, kann dieses zur Reduktion eines anderen aktiven Items verwendet werden. Die fehlende Zielgerichtetheit der bottom-up-Analyse drückt sich darin aus, daß es keine Vorhersage gibt. Zu Beginn der Analyse werden alle Grammatikregeln in den Startzustand aufgenommen, damit sie falls nötig zur Verfügung stehen. Zusätzlich werden auch die passiven Items für jedes Wort der zu analysierenden Kette in die Tabelle eingefügt, so daß während der Analyse auch kein Scanning mehr nötig ist.

Bis jetzt wurde nur das Parsing betrachtet, für das die Menge der lexikalischen Einträge im Startzustand nicht übermäßig groß ist. Wie sich später herausstellen wird, treten im Falle der bottom-up-Generierung aber erhebliche Probleme bei der Eingrenzung des Item-Menge im Startzustand auf.

Auch eine **left-corner-Analyse** kann im Rahmen von Chart-Parsing modelliert werden. Nachdem die Unit-Items für die Wörter der gegebenen Kette in die Tabelle aufgenommen wurden, kann nun gezielt eine Grammatikregel gesucht werden, die unter Verwendung des lexikalischen Eintrags als passives Item reduziert wird. Wenn das 'richtige' aktive Item ausgewählt wurde (und dieses noch nicht zu einem passiven Item wurde), kann nun das passive Item für das nächste Wort zu seiner weiteren Reduktion verwendet werden. Die Zielgerichtetheit der left-corner-Analyse zeigt sich also auch im Rahmen von Chart-Parsing darin, daß Regeln anhand ihrer linken Ecken bottom-up ausgewählt werden. Durch das Parsing der Elemente rechts der linken Ecke wird diese Auswahl daraufhin überprüft. Dies bedeutet, daß es nicht nötig ist, alle Grammatikregeln zu Beginn in den Startzustand aufzunehmen. Die Effizienz der left-corner-Analyse kann auch im Rahmen von Chart-Parsing durch Verwendung einer Linking-Relation gesteigert werden.

Gazdar und Mellish (1989, S. 197) zeigen den engen Zusammenhang zwischen bottom-up- und left-corner-Analyse im Rahmen von Chart-Parsing. Wenn die Grammatikregeln nicht alle zu Beginn der Analyse in die Tabelle aufgenommen werden, sondern erst anhand ihrer linken Ecke ausgewählt werden, ergibt sich 'automatisch' eine gewisse Zielgerichtetheit.

Schon Kilbury (1985) hat eine formale Definition eines left-corner-Chart-Parsers gegeben, der sich aus einer Modifikation des Earley-Parsing-Algorithmus ergibt (s. den nächsten Abschnitt). Auch diese Chart-basierte Variante der left-corner-Analyse zeigt den generellen Vorteil von in-order-Algorithmen, der darin besteht, daß die Vorhersage Daten-gesteuert ist und nur die *nächsthöhere* Konstitente umfaßt, nicht jedoch alle höheren (vgl. Kilbury 1985, S. 84). Top-down-Algorithmen sagen dagegen alle 'tieferen' Konstitenten vorher, bis ein lexikalischer Zugriff stattfinden kann.

Earley-Parsing Der von Earley (1970) entwickelte Parsing-Algorithmus kann als eine spezielle Instanz des Chart-Parsings angesehen werden. Die Baum-Traversierung ist top-down und es werden alle Alternativen gleichzeitig verfolgt (Breitensuche). Beginnend mit einem Start-Item wie bei der oben beschriebenen top-down-Analyse werden die Inferenzregeln in der Reihenfolge Prediction-Scanning-Completion rekursiv angewendet.[2] Unter *Completion* bzw. *Vervollständigung* wird die Reduktion eines aktiven Items durch ein passives verstanden. Es handelt sich also um die Anwendung der fundamentalen Regel. Der Scanning-Schritt meint im Kontext von Earley-Parsing die Wahl eines lexikalischen Eintrags wie von der Pseudo-Inferenzregel in (33) vorgesehen und dazu die sofortige Reduzierung aller aktiven Items mit dem neuen passiven Item. Dies faßt also zwei Schritte zusammen. Da aber der Completor beim Earley-Parsing immer nach dem Scanning aufgerufen wird, kann Earley-Parsing auch als rekursive Anwendung von Vorhersage-, Scanning- und fundamentaler Inferenzregel in der zuvor eingeführten Definition aufgefaßt werden.

Der Earley-Parsing-Algorithmus wendet rekursiv jede Inferenzregel solange an, wie dies möglich ist. Wenn beispielsweise der Completor ein aktives Item in ein passives Item verwandelt hat, dann versucht er im folgenden, auch dieses neue passive Item zu einer Reduktion zu verwenden. Erst wenn es keine Anwendungsmöglichkeiten für eine Inferenzregel auch unter Berücksichtigung der unmittelbar hinzugekommenen Items mehr gibt, kommt die nächste Inferenzregel an die Reihe. Daraus ergibt sich die parallele Verfolgung aller Alternativen. Die Vorhersage nimmt alle Grammatikregeln, deren linke Seiten mit dem zu bearbeitenden Element kompatibel sind, als leere Items in die Tabelle auf. Der Completor ermöglicht durch seine erschöpfende Anwendung, daß alle Reduktionen – auch die, die verschiedenen Analysebäumen entsprechen – gleichzeitig durchgeführt werden.

Die Schleife aus Prediction, Scanning und Completion wird solange ausgeführt, bis keine Anwendungen mehr möglich sind. Wenn z.B. alle Wörter des Input-Strings abgearbeitet sind, können keine passiven Items für die lexikalischen Einträge mehr hinzugefügt werden, so daß bald alle Bearbeitungsmöglichkeiten der Tabelle erschöpft sind. Dann kann aus der Tabelle entnommen werden, ob und wieviele Items es gibt, die als Lösung angesehen werden können.

Der Earley-Parsing-Algorithmus ist unter anderem deswegen bekannt geworden, weil er einen Satz mit n Worten in einer Zeit proportional zu n^3 parsen kann, d.h. seine Komplexität beträgt $O(n^3)$.[3]

4.2.3 Earley-Deduktion

Das Chart-Parsing kann zu einem allgemeinen Deduktionsverfahren ausgedehnt werden, das eng an Earleys Parsing-Algorithmus angelehnt ist (deshalb auch *Earley-Deduktion* genannt) und wie dieses top-down vorgeht (Pereira und Warren 1983). Bisher wurde von

[2] Die Erläuterungen in diesem Abschnitt beziehen sich auf die Darstellung in Covington (1994, Kapitel 6.6).

[3] Dies ist eines der besten überhaupt möglichen Resultate. Allerdings beschreibt es nur die Komplexität im ungünstigsten Fall. Da alle Analysen parallel verfolgt werden, ist ein Earley-Parser im Durchschnitt oft langsamer als beispielsweise ein top-down-depth-first-Parser mit Backtracking, da diese ungünstigsten Fälle nicht so häufig auftreten (vgl. Covington 1994, S. 191).

Für den speziellen Fall der Verarbeitung natürlicher Sprache hat sich jedoch gezeigt, daß eine zielgerichtete Variante des von Kilbury (1985) vorgeschlagenen left-corner-Chart-Parsers effizienter ist als der Earley-Parsing-Algorithmus, an den er eng angelehnt ist (Wirén 1987).

kontextfreien Grammatikregeln ausgegangen, die in den Items verwendet wurden (gekennzeichnet durch den einfachen Rechtspfeil '$\rightarrow$'). Das Earley-Deduktionsverfahren arbeitet dagegen mit Definiten Klauseln im allgemeinen, unabhängig davon, ob sie eine Grammatik repräsentieren oder nicht.

Bei der Darstellung von Chart-Parsing wurde unterschieden zwischen der Tabelle und der Grammatik, die aus Lexikoneinträgen und Grammatikregeln besteht. Die fundamentale Inferenzregel arbeitet nur auf den Items, die sich in der Tabelle befinden. Die anderen beiden Inferenzregeln (Scanning und Vorhersage) dienen dazu, neue Items in die Tabelle einzufügen. Der Grammatik entspricht in einem Logik-Programm das Programm selbst, das aus Unit-Klauseln (Fakten, die z.B. lexikalische Einträge repräsentieren können) und Nonunit-Klauseln (Regeln) zusammengesetzt ist. Die Tabelle wird als der *Zustand (state)* des Algorithmus bezeichnet. Wenn eine Nonunit-Klausel aus dem Programm in den Zustand aufgenommen wird, dann wird ein negatives Literal (also ein Element des Antezedens) ausgewählt, d.h. besonders markiert. Dies entspricht beim Chart-Parsing dem Element rechts des Punktes – bei gerade in die Tabelle aufgenommenen Items also dem ersten Element der rechten Regelseite.

Durch das Vorrücken des Punktes von links nach rechts wird beim Chart-Parsing eine left-to-right-Reihenfolge erzwungen. Dies bedeutet, daß Kopf-gesteuerte Ansätze im ursprünglichen Rahmen von Chart-Parsing nicht erfaßt werden können, da der semantische Kopf und nicht das am weitesten links stehende noch nicht bearbeitete Element gewählt wird. Diese Beschränkung wird im Earley-Deduktionsverfahren aufgehoben. Jedes beliebige negative Literal kann durch die Selektionsfunktion gewählt werden. Tatsächlich erwähnen schon Pereira und Warren (1983, S. 141) eine 'head word parsing'-Strategie. Dadurch wird eine andere Datenstruktur für die Items erforderlich, denn der Punkt teilt die Liste der negativen Literale nur in einen unbearbeiteten und ein bearbeitenden Teil, wobei das gewählte Element dasjenige rechts des Punktes ist. Wenn dagegen *irgendein* Element bearbeitet werden kann, entsteht eine Mischung aus bearbeiteten und unbearbeiteten Elementen, wobei *jedes* der unbearbeiteten Elemente gewählt werden kann.

Die einfachste Form, dies zu repräsentieren, ist die Löschung der schon bearbeiteten Elemente, so daß nur noch das gewählte Element markiert werden muß. Der Vorteil einer solchen Repräsentation liegt darin, daß das Antezedens einer Nonunit-Klausel im Laufe der Verarbeitung immer weiter reduziert wird, bis schließlich eine Unit-Klausel übrig bleibt. Als Beispiel werden die Chart-Items aus (30) als State-Items angegeben, wobei eine Grammatik in DCG-Notation angenommen wird (zur Repräsentation der String-Positionen s.u.):

(34) a. $np([die, katze, schläft], [schläft])$

b. $s([die, katze, schläft], [\,]) \leftarrow \mathbf{vp}([schläft], [\,])$

c. $vp([schläft], [\,]) \leftarrow \mathbf{v}([schläft], [\,])$

Die Klausel in (34a) ist durch Reduktion zu einem passiven Item geworden, wobei in dieser Notation nicht mehr zu erkennen ist, welche negativen Literale bearbeitet wurden. Diese Unit-Klausel ist als die Proposition '*die katze* ist eine NP' zu interpretieren. Die Implikation in (34b) sagt aus, daß ein Satz analysiert wäre, wenn eine VP geparst werden kann. Das ausgewählte Element des Antezedens ist durch Fettdruck markiert – da es nur eines gibt, ist dies hier allerdings nicht informativ. Die noch nicht bearbeitete Klausel in (34c) ist von ihrer Struktur her nicht von einer abgeleiteten Klausel wie in (34b) zu unterscheiden.

Inferenzregeln in der Earley-Deduktion Die Klausel in (34c) zeigt, daß ihre String-Argumente instantiiert worden sind, als sie in den Zustand aufgenommen wurde. Deswegen nennen Pereira und Warren (1983, S. 139) diese Operation auch *instantiation*. Sie entspricht der top-down-Vorhersage in Chart-Parsing-Algorithmen. Die zweite Inferenzregel der Earley-Deduktion ist die *reduction*-Regel. Sie entspricht der Vervollständigung durch die fundamentale Regel. Um eine einheitliche Terminologie zu wahren, werden im folgenden weiterhin die Begriffe 'Vorhersage/prediction' statt 'Instantiierung' und 'Vervollständigung/completion' statt 'Reduktion' für die Inferenzregeln verwendet.

Die top-down-Vorhersage führt nach Pereira und Warren (1983) nur *Nonunit*-Klauseln in den Zustand ein, wenn das gewählte Element des gerade bearbeiteten Nonunit-Items mit dem positiven Literal der Nonunit-Klausel im Programm unifiziert. Es können Unit-Klauseln aus dem Zustand *und* dem Programm auswählt werden, wenn sie zur Reduktion des ausgewählten Elements des aktuellen Nonunit-Items benötigt werden. Nur das gewählte Element einer Nonunit-Klausel darf jeweils bearbeitet werden – die anderen Elemente werden zu einem späteren Zeitpunkt gewählt. Das Scannen wird also von dem Completor mitübernommen, wobei genauso wie beim Scanning des Earley-Parsing-Algorithmus nicht nur eine Unit-Klausel in den Zustand eingefügt wird, sondern diese auch sofort reduziert wird. Dadurch, daß der Completor auf alle Unit-Klauseln des Programms zugreifen kann, wird die Trennung zwischen Programm und Zustand in gewisser Weise aufgehoben – oder anders herum formuliert: alle Unit-Klauseln befinden sich gewissermaßen von Anfang an im Programm.

Der **Completor** erlaubt also die Kombination eines aktiven Items mit einem passiven und hat folgende allgemeine Form:

(35)

$$\frac{\begin{array}{c} N_1 \leftarrow \ \alpha \ \wedge \ \mathbf{N_2} \ \wedge \ \beta \\ N_3 \leftarrow \end{array}}{\phi[N_1 \leftarrow \ \alpha \ \wedge \ \beta]}$$

N_2 und N_3 müssen unifizieren, wobei noch einmal betont werden soll, daß in einer Grammatikregel nur das gewählte Element N_2 reduziert werden kann. Alle anderen negativen Literale können erst dann reduziert werden, wenn sie selbst das gewählte Element sind. Während in der fundamentalen Regel für das Chart-Parsing in (31) Unit-Klauseln durch kontextfreie Regeln mit einem Punkt rechts des letzten Elements der rechten Seite gekennzeichnet waren, gibt es in der allgemeinen Form für Unit-*Klauseln* keine rechte Seite (mehr).[4]

Die **Vorhersage-Regel** der Earley-Deduktion sieht folgendermaßen aus, wobei die gleichen Bedingungen wie bei der Reduktions-Regel gelten:

(36)

$$\frac{\begin{array}{c} N_1 \leftarrow \ \alpha \ \wedge \ \mathbf{N_2} \ \wedge \ \beta \\ N_3 \leftarrow \ \gamma \end{array}}{\phi[(N_2 \sqcup N_3) \leftarrow \ \gamma]}$$

Wie in (32) zeigt sich auch in (36), daß keine echte Inferenz stattfindet. Außer der Instantiierung der gewählten Nonunit-Klausel wird diese nicht weiter verändert.

Diese beiden Inferenzregeln enthalten keine Angaben über String-Positionen. Dies ist auch nicht möglich, da sie für Programme aus Definiten Klauseln im allgemeinen und nicht nur für Grammatiken gelten sollen. Sie können aber natürlich als 'normale Argumente' verwendet werden. Die Verwendung von String-Positionen beim Parsing dient dazu, einen effizienten Zugang zur Tabelle zu ermöglichen. Wenn bespielsweise ein Unit-Item beginnend an Position 3 zur Reduktion verwendet werden soll, müssen nur die Vorhersagen bzw. Nonunit-Items berücksichtigt werden, die an dieser Position beginnen. Bei der Generierung muß dagegen eine andere Form der Indizierung, die dieses leistet, gefunden werden. Auf diese Fragestellung geben die im folgenden vorgestellten Ansätze unterschiedliche Antworten, die jeweils mit den Vorschlägen selbst erläutert werden.

Zeitpunkt der Reduktionen Eine vereinfachende Annahme, die im Rahmen von Earley-Parsing gemacht werden kann, gilt nicht für Earley-Deduktion im allgemeinen: Es ist nicht garantiert, daß alle Unit-Klauseln in jedem Fall nach den Nonunit-Klauseln auftreten, zu deren Reduktion sie verwendet werden können. Daraus ergibt sich beim Earley-Parsing, daß der Completor nur verwendet werden muß, wenn ein Unit-Item verarbeitet wird. Pereira und Shieber (1987, S. 200) und im Anschluß Neumann (1994, S.77) begründen die beim Earley-Parsing mögliche Vereinfachung folgendermaßen: da die negativen Literale in Nonunit-Klauseln immer von links nach rechts abgearbeitet werden, muß für jede Unit-Klausel schon die Nonunit-Klausel in der Tabelle vorhanden sein, die zu ihrer Bildung geführt hat. Aus dieser Argumentation ergibt sich, daß z.B. bei einer head-first-Strategie auch bei der Bearbeitung von aktiven Items eine Reduktion versucht werden muß.

Diese Argumentation ist meiner Ansicht nach jedoch nicht überzeugend: auch im Falle eines anderen als dem am weitesten links stehenden Literals wird dieses ja zur Vorhersage verwendet. Es ist also gewählt, so daß es bei der Reduktion durch das später auftretende

[4]In bezug auf kontextfreie Grammatiken ist eine solche Schreibweise nötig, da Lexikoneinträge die allgemeine Form *Präterminalsymbol* $\rightarrow$ *Terminalsymbol* haben. Es muß sowohl eine rechte wie auch eine linke Regelseite für einen lexikalischen Eintrag geben, da die Information über den Zusammenhang zwischen Kategorie und Wortform sonst verloren würde. In Definiten Klauseln kann diese Information leicht in einem Literal gespeichert werden, z.B. $det([der|R], R)$.

passive Item zur Verfügung steht. Bei der top-down-Earley-Deduktion wird in den meisten Fällen das Nonunit-Item vor dem Unit-Item vorhanden sein, denn die Vorhersage (durch die Grammatikregeln) erfolgt, *bevor* die Lexikoneinträge erreicht sind – unabhängig von der Reihenfolge, mit der die negativen Literale bearbeitet werden.

Der Fall, daß ein passives Item schon vorhanden ist, wenn ein aktives bearbeitet wird, tritt dagegen dann auf, wenn eine Phrase aus einem Zweig des Beweisbaums in einem *parallelen* Zweig wiederverwendet werden kann. Das passive Item aus dem ersten Zweig befindet sich schon in der Tabelle und kann wiederverwendet werden. Der Grund dafür, daß der Earley-Parsing-Algorithmus vollständig ist, obwohl er nur bei passiven Items den Completor aufruft, liegt meiner Ansicht nach darin, daß er nur mit Breitensuche arbeitet. Wenn dagegen Tiefensuche möglich ist (s. den nächsten Abschnitt), muß auch bei aktiven Items eine Reduktion mit Unit-Items der Tabelle versucht werden. Diese Einschätzung wird zumindest durch die Implementierung von top-down-Earley-Deduktion (s. Kapitel 5) bestätigt: die Reduktion von gegebenen aktiven Items findet sowohl beim Parsing wie bei der Generierung nur bei Tiefensuche statt. Der Grund dafür liegt darin, daß bei der Tiefensuche ein Zweig des Beweisbaums soweit wie möglich verfolgt wird, ehe alternative Zweige berücksichtigt werden. Dies bewirkt, daß abgeleitete Unit-Items aus früheren Zweigen des Beweisbaumes für spätere zur Verfügung stehen. Der Grund für die nötige Erweiterung des Earley-Deduktionsverfahrens liegt also nicht darin, daß die negativen Literale von Nonunit-Klauseln nicht mehr in jedem Fall von links nach rechts abgearbeitet werden, sondern in der Verwendung einer anderen Kontrollstrategie.

Da es logisch keinen Unterschied macht, ob ein passives Item mit einem aktivem kombiniert wird oder umgekehrt, muß die Inferenzregel nicht verändert werden. Für eine Implementierung dagegen ist der Unterschied relevant, da entschieden werden muß, mit welchem Input der Completor aufgerufen wird (s.a. Abschnitt 4.3.3).

Agenda

Die Reihenfolge, in der neue Items bearbeitet werden, läßt sich durch eine *Agenda* steuern (Kay 1980). Bevor ein durch die Anwendung einer Inferenzregel gebildetes Item bearbeitet und in die Tabelle eingefügt wird, wird es in der Agenda zur Bearbeitung aufbewahrt. Diese kann als eine Menge von Items, die jeweils mit einer Priorität versehen sind, aufgefaßt werden. Das nächste zu bearbeitende Item ist dasjenige mit der höchsten Priorität. Als Abbruchbedingung für die Earley-Deduktion kann dann eine leere Agenda definiert werden, d.h. es gibt keine weiteren Aufgaben mehr auszuführen. Die einfachste Realisierung einer Agenda ist die Verwendung einer Liste, wobei sich die Priorität aus der Reihenfolge der Items in der Liste ergibt (Pereira und Shieber 1987, S. 200). Wenn die Agenda als *Keller (last-in-first-out-stack)* verwendet wird, ergibt sich eine Tiefensuche-Kontrollstrategie. Wenn dagegen immer das 'älteste' Item der Agenda verarbeitet wird, ergibt sich eine Breitensuche-Kontrollstrategie. Dies läßt sich dadurch erreichen, daß neue Items an das Ende der Liste angehängt werden.

Breitensuche und Tiefensuche stellen die beiden Extreme in einem Kontinuum von möglichen Kontrollstrategien dar. Grundsätzlich ist jede Reihenfolge möglich, so daß linguistische Präferenzen in der Wahl der Items ausgedrückt werden können. Erbach (1995a, Kap. 4) stellt eine probabilistische Konzeption von Priorität dar, die zu einer *best-first-Verarbeitung* beitragen soll. Die grundsätzliche Schwierigkeit liegt darin, daß unklar ist, nach welchen Kriterien die Prioritäten zugeordnet werden sollen. Dieses Thema kann im Rahmen der vorliegenden Arbeit nicht weiter verfolgt werden. Es wird jedoch deutlich, daß Earley-Deduktion durch die Verwendung einer Agenda einen geeigneten Rahmen zur Modellierung unterschiedlicher Kontrollstrategien darstellt.

Subsumptions-Check

Die wesentliche Motivation zur Verwendung einer Tabelle ist die Vermeidung von mehrfacher Arbeit. Dazu wird vor der Bearbeitung eines Items überprüft, ob es sich schon in der Tabelle befindet. Da nur bearbeitete Items in die Tabelle eingefügt werden, ist es in einem solchen Fall nicht mehr notwendig, das betreffende Item noch einmal zu bearbeiten: es wird *blockiert*. Bei der ausschließlichen Verwendung von atomaren Symbolen an den Argumentpositionen der Literale *(ground arguments)* ist es ausreichend, nach *identischen* Klauseln in der Tabelle zu suchen. Wenn dagegen komplexe Terme, die auch Variablen enthalten, auftreten – dies wird bei jeder Menge von Definiten Klauseln, die eine Merkmals-basierte Grammatik repräsentiert, der Fall sein – ist dies nicht mehr ausreichend. Stattdessen muß geprüft werden, ob ein schon bearbeitetes Item das aktuelle *subsumiert*, d.h. daß das aktuelle Item eine Instanz des schon bearbeiteten ist.

Die Notwendigkeit eines Subsumptions-Tests ist ein großer Nachteil für Chart-basierte Algorithmen, da er sehr aufwendig ist und den durch die Tabelle erzielten Effizienzgewinn zumindest vermindert. Wenn die beiden zu vergleichenden Strukturen sehr ähnlich sind, muß eine Subsumptions-Prozedur große Teile der Strukturen vergleichen, ehe eine Entscheidung über das Vorliegen einer Subsumptions-Beziehung möglich ist (Gerdemann 1991, S. 79).

Es gibt jedoch Fälle, in denen der Subsumptions-Test nicht greift. Wie in Abschnitt 2.3.3 erläutert, wird bei jedem top-down-Vorhersage-Schritt mit der linksrekursiven VP-Komplement-Regel die Subkategorisierungs-Liste um ein Element größer. Deswegen liegt nie eine Subsumptions-Beziehung zwischen einem Item der Tabelle und dem aktuellen Item vor. Dies macht die Verwendung der Restriktor-Technik erforderlich.

Um den Subsumptions-Test bei top-down-Vorhersagen zu vereinfachen, schlagen Pereira und Warren (1983, S. 142) vor, nur bestimmte *Literale*, d.h. nicht ganze Items, zu berücksichtigen. Wenn die Literale, die schon (erschöpfend) für Vorhersagen verwendet wurden, gespeichert werden, kann der Subsumptions-Check, der vor der Vorhersage mit einem neuen Item stattfinden muß, nur auf der Menge der gespeicherten Literale ausgeführt werden. Der Subsumptions-Test auf die ganze Tabelle muß dann nur noch auf abgeleitete Unit-Items angewendet werden. Diese Idee wird von Gerdemann (1991, S. 79ff) mit der

Restriktor-Technik verbunden (s. Abschnitt 4.3.2).

In bezug auf durch den Completion-Schritt abgeleitete Items gilt, daß sie sich bei Grammatiken, die die Ableitungsstruktur in der Merkmalsstruktur kodieren, immer in diesem Merkmalsswert unterscheiden – sie müssen aus verschiedenen Vorhersagen stammen, da gleiche Vorhersagen blockiert werden. Erbach (1995a, S. 24) zieht aus dieser Beobachtung den Schluß, daß der Subsumptions-Check für solche Grammatiken ausgelassen werden darf, da er nie erfolgreich sein kann (dies gilt nicht für Vorhersagen).

Es soll bemerkt werden, daß die Kodierung der Ableitungsstruktur im Falle der Generierung nicht die Verwendung einer einmal abgeleiteten Unit-Klausel an verschiedenen Positionen im Satz verhindert.[5] Da immer nur die Ableitungsstruktur von dem jeweiligen Knoten an abwärts repräsentiert wird, ist sie beispielsweise für eine NP aus Det und N immer gleich, unabhängig davon, wo dieser Teilbaum in den Baum für den ganzen Satz 'eingehängt' wird. Wenn andererseits eine Phrase tatsächlich zwei verschiedene Analysen hat, sollte sich dies auch in der Konstituentenstruktur niederschlagen.

Linksrekursive Regeln in top-down-Earley-Deduktion

Durch die Verwendung der Restriktor-Technik (Shieber 1985) können die Merkmalsstrukturen der Elemente in der VP-Komplement-Regel so beschränkt werden, daß eine Subsumptions-Beziehung schon vor der zweiten Vorhersage besteht. Dies soll für das Parsing des Satzes *peter gibt petra den ball* genauer erläutert werden. Es wird dabei von top-down-Earley-Deduktion unter Verwendung einer Agenda, die Tiefensuche modelliert, ausgegangen.

Durch das Start-Item wird eine S-Regel $s \ \leftarrow \ np \wedge vp$ vorhergesagt. Nachdem die NP für das Subjekt analysiert wurde, kann eine Reduktion des Items für die S-Regel stattfinden (hier wird eine DCG-Notation verwendet, wobei die Worte der Kette abgekürzt werden):

(37)

$$\frac{\begin{array}{c} s([peter, gibt, petra, den, ball], [\,]) \leftarrow \ \mathbf{np}([peter, gibt, ...], [gibt, ...]) \ \wedge \ vp([gibt, ...], [\,]) \\ np([peter|R], R) \leftarrow \end{array}}{s([peter, gibt, petra, den, ball], [\,]) \leftarrow \ \mathbf{vp}([gibt, petra, den, ball], [\,])}$$

Die VP wird nun zur Vorhersage verwendet. Da stets so viele Regelanwendungen wie möglich versucht werden, kann nun sowohl $vp \ \leftarrow \ vp \wedge np$ als auch $vp \ \leftarrow \ v$ vorhergesagt und in die Agenda aufgenommen werden. Wenn das zweite Item bearbeitet wird, führt es zu keinem lexikalischen Zugriff, da die Restliste mit der leeren Liste instantiiert ist – es wird eine VP gesucht, die die ganze Kette abdeckt. Das erste Item dagegen wird erst durch Restriktion beschränkt (d.h. die Subkategorisierungs-Liste wird eliminiert) und führt dann zu zwei weiteren Vorhersagen der beiden genannten Regeln.

[5]Dieses Problem tritt bei der Verwendung von String-Positionen in der Generierung auf. Ein Lösungsvorschlag von Gerdemann (1991)wird in Abschnitt 4.3.2 vorgestellt.

Da bei dieser zweiten Vorhersage der beiden Regeln für das Verb die Restliste für den String nicht mehr instantiiert ist, kann ein lexikalischer Zugriff auf das Verb mit gleichzeitiger Reduktion erfolgen:[6]

(38)

$$\frac{\begin{array}{c} vp([gibt, petra, den, ball], R) \leftarrow\ \mathbf{v}([gibt, petra, den, ball], R) \\ v([gibt|R1], R1) \leftarrow \end{array}}{vp([gibt, petra, dem, ball], [petra, den, ball]) \leftarrow}$$

Es entsteht eine Unit-Klausel für eine VP, die den String *gibt* abdeckt und die (hier nicht zu sehen) eine instantiierte dreielementige Subkategorisierungs-Liste besitzt. Wenn etwas später die vorhergesagte Nonunit-Klausel für die VP-Komplement-Regel aus der Agenda bearbeitet wird, kann sie mit der Unit-Klausel für *gibt* kombiniert werden:

(39)

$$\frac{\begin{array}{c} vp([gibt, petra, den, ball], R) \leftarrow\ \mathbf{vp}([gibt, petra, den, ball], R1)\ \wedge\ np(R1, R) \\ vp([gibt, petra, den, ball], [petra, den, ball]) \leftarrow \end{array}}{vp([gibt, petra, den, ball], R) \leftarrow\ \mathbf{np}([petra, den, ball], R)}$$

Das Ergebnis ist eine Nonunit-Klausel, die eine VP repräsentiert, wenn eine NP geparst werden kann. Die Kongruenz-Merkmale der NP sind instantiiert worden (hier wiederum nicht zu sehen). Die VP-Komplement-Regel kann zur Vorhersage einer weiteren VP-Komplement-Regel verwendet werden. Diese wird zuerst in die Agenda aufgenommen. Wenn sie bearbeitet wird, befindet sich die (identische) VP-Komplement-Regel, die sie 'verursacht' hat, in der Tabelle. Weitere Vorhersagen können blockiert werden, weil der Subsumptions-Check greift.

Im folgenden wird *petra* zielgerichtet als NP im Dativ analysiert. Diese kann die Nonunit-Klausel aus (39) zu einer Unit-Klausel reduzieren. Sie besagt nun, daß *gibt petra* eine VP ist. Ihre Subkategorisierungs-Liste enthält noch zwei Elemente (für eine NP im Nominativ und eine im Akkusativ).

Das Unit-Item für diese VP kann nun mit der VP-Komplement-Regel *aus der Tabelle* kombinieren. Es findet also keine weitere Vorhersage statt. Stattdessen wird die VP-Regel aus der Tabelle 'wiederverwendet'. Sie steht also für eine beliebige Anzahl von Reduktionen zur Verfügung, ohne daß 'neue' VP-Regeln vorhergesagt werden müßten. Die Reduktion verläuft analog zu der in (38):

(40)

$$\frac{\begin{array}{c} vp([gibt, petra, den, ball], [den, ball]) \leftarrow \\ vp([gibt, petra, den, ball], R) \leftarrow\ \mathbf{vp}([gibt, petra, den, ball], R1)\ \wedge\ np(R1, R) \end{array}}{vp([gibt, petra, den, ball], R) \leftarrow\ \mathbf{np}([den, ball], R)}$$

[6]Dies gibt einen kleinen Eindruck von der Vielzahl an möglichen Instantiierungen und Kombinationen von Items. Die Darstellung ist zur besseren Übersichtlichkeit im folgenden etwas vereinfacht worden.

Das Ergebnis der Reduktion besagt, daß *den ball* als NP (im Akkusativ) analysiert werden muß, damit eine VP bewiesen ist. Sie hat inzwischen nur noch ein Element in der Subkategorisierungs-Liste. Wenn die fehlende NP erfolgreich analysiert wurde, kann ein Unit-Item für die ganze VP *gibt petra den ball* gebildet werden. Diese VP reduziert nun das in (37) gebildete Nonunit-Item:

(41)

$$\frac{\begin{array}{c} vp([gibt, petra, den, ball], [\,]) \leftarrow \\ s([peter, gibt, petra, den, ball], [\,]) \leftarrow \ \mathbf{vp}([gibt, petra, den, ball], [\,]) \end{array}}{s([peter, gibt, petra, den, ball], [\,]) \leftarrow}$$

Als letztes erfolgt die Reduktion des Start-Items, das bei der top-down-Earley-Deduktion verwendet werden muß, um die Verarbeitung zu beginnen.

Diese Darstellung zeigt, daß eine unerwünschte rekursive Vorhersage der VP-Komplement-Regel durch den Subsumptions-Test verhindert wird. Sie zeigt aber auch, daß die Earley-Deduktion starke bottom-up-Aspekte hat: die VP wird bottom-up mit instantiierter Subkategorisierungs-Liste analysiert. Die beiden VP-Regeln $vp \leftarrow vp \wedge np$ und $vp \leftarrow v$ werden gleichzeitig bearbeitet. Das Ergebnis des lexikalischen Zugriffs der zweiten Regel kann in der ersten Regel verwendet werden. Die mehrmalige Verwendung der VP-Komplement-Regel aus der Tabelle macht es erforderlich, Instantiierungen von Items in der Tabelle zu verhindern. Darüberhinaus wäre der Subsumptions-Check sinnlos, wenn Items, die sich schon in der Tabelle befinden, spezifischer werden könnten. Es ist deshalb notwendig, Fragen des *Kopierens* von Information zu behandeln (s. Abschnitt 5.4).

Neumann (1994) zeigt eine weitere Möglichkeit, unnötige Vorhersagen zu verhindern: wenn ein aktives Items durch ein passives Item reduziert werden kann, sind keine weiteren Vorhersagen mit dem jeweiligen gewählten negativen Literal in diesem aktiven Item nötig (s. Abschnitt 4.3.3). Dadurch werden einige (redundante) Vorhersagen, die ansonsten später durch den Subsumptions-Test blockiert werden müßten, von vornherein vermieden.

4.3 Top-down-Earley-Deduktion

4.3.1 Die uniforme Architektur von Shieber (1988)

Der erste und bekannteste Vorschlag zu einer uniformen Architektur für Parsing *und* Generierung wurde von Shieber (1988) gemacht. Er ist eng an Earley-Deduktion im Sinne von Pereira und Warren (1983) angelehnt. Die einzigen beiden Inferenzregeln sind Vorhersage und Vervollständigung. Neben der Tabelle wird eine Agenda verwendet, die es erlaubt, verschiedene Kontrollstrategien zu modellieren. Eine auffällige Eigenschaft des Ansatzes ist die Tatsache, daß es einerseits keinen expliziten Scanner gibt, andererseits aber der Completor nicht auf die Unit-Items der Grammatik zugreifen kann. Daß letzteres nicht vorgesehen ist, zeigt die Tatsache, daß *alle infrage kommenden Unit-Items* (dazu s.u.) vor Beginn der Verarbeitung in die Agenda aufgenommen werden müssen, d.h. sie befinden sich im Startzustand. Auf diese Weise ist also letztlich doch ein Scanner vorhanden, der allerdings nicht während der Analyse eingesetzt werden kann.

Parsing und Generierung sowie jeweils verschiedene Kontrollstrategien können durch Parametrisierung in drei Punkten erreicht werden:

- Initialisierung der Agenda, d.h. Festlegung des Startzustands,
- Festlegung einer Prioritätsfunktion für die Items in der Agenda,
- Festlegung eines Erfolgskriteriums für Unit-Items, die als Lösung für das 'gegebene Ziel' angesehen werden sollen.

Dabei ist zu berücksichtigen, daß es nicht ein einziges gegebenes Ziel bei der Deduktion gibt, auch wenn es die Intention des Benutzers ist, einen bestimmten Satz zu parsen bzw. zu generieren. Vielmehr bearbeitet ein auf Earley-Deduktion basierender Algorithmus alle Start-Items in der Agenda sowie die aus ihnen abgeleiteten Klauseln bis schließlich die Agenda leer ist. Die beiden Inferenzregeln werden stets so oft wie möglich eingesetzt, um die Vollständigkeit des Interpreters zu garantieren (Shieber 1988, S. 615).

Parsing Für alle Parsing-Instanzen des Algorithmus werden die Unit-Items für die Wörter der Input-Wortkette in die Agenda aufgenommen. Sie enthalten einen Index für die String-Position wie im Earley-Parsing-Algorithmus. Weiterhin wird eine initiale Vorhersage mit dem Startsymbol der Grammatik als linke Regelseite in die Agenda eingefügt.[7] Das Erfolgskriterium besteht darin, daß ein abgeleitetes Unit-Item die volle Länge der Kette abdeckt – zu erkennen an der Start- und Endposition des Items. Zudem wird verlangt, *"[...] that the nonterminal be the start nonterminal"* (Shieber 1988, S. 615). Dieses zweite Kriterium meint in bezug auf komplexe Merkmals-basierte Formalismen die Kompatibilität des abgeleiteten Unit-Items mit den *linken Seiten* der im Start-Zustand befindlichen Regeln, auch wenn dies etwas unklar formuliert ist. Das Problem von Kohärenz und Vollständigkeit, das sich hier stellt, wird von Shieber nicht behandelt, aber als Aufgabe für weitere Forschungen gesehen (Shieber 1988, S. 618).

Es werden drei verschiedene Prioritätsfunktionen vorgestellt:

1. Die Items in der Agenda bekommen eine umso niedrigere Priorität, je höher die Zahl für die Endposition ist, wobei bei gleicher Endposition das 'neuere' Item vorgezogen wird. Dies bewirkt eine Verarbeitung von links nach rechts mit Breitensuche und modelliert den Earley-Parsing-Algorithmus.

2. Tiefensuche mit Backtracking wird simuliert durch die Verwendung der Agenda als last-in-first-out-stack. Jedes Item bekommt gewissermaßen einen 'Zeitstempel', wobei ein Item umso niedriger bewertet wird, je älter es ist.

[7] Die Bezeichnung 'Startsymbol' ist auf kontextfreie Grammatiken bezogen. Für komplexe Merkmals-basierte Formalismen gilt sinngemäß, daß der entsprechende Wert für die syntaktische Kategorie dem Startsymbol einer kontextfreien Grammatik entspricht.

3. Es lassen sich auch komplexere Prioritätsfunktionen definieren, die psycholinguistische Modelle realisieren sollen. Wenn Nonunit-Items immer vor Unit-Items bearbeitet werden, ansonsten weiter rechts stehende Items vorgezogen werden und die Reduktion von 'längeren' Items vor der Reduktion von 'kürzeren' stattfindet, wird ein *Satz-Desambiguierer* modelliert.

Generierung In Hinblick auf eine uniforme Architektur stellt sich die Frage, wie die in bezug auf das Chart-*Parsing* entwickelten Methoden für die Generierung verwendet werden können. Da die String-Positionen der zu generierenden Wörter zu Beginn unbekannt sind, schlägt Shieber vor, sie zu ignorieren. Dies hat den Vorteil, daß eine generierte Phrase an verschiedenen Positionen im Satz eingesetzt werden kann. Vor allem aber ist dies vorteilhaft für die Initialisierung der Agenda unter den speziellen Voraussetzungen von Shiebers Architektur. Wenn String-Positionen verwendet werden, müssen *alle* Unit-Items der Grammatik für *alle* Positionen der möglichen Kette in die Agenda aufgenommen werden. Abgesehen davon, daß dies zu einer nicht akzeptablen Redundanz und Ineffizienz führt, ist bei der Generierung gar nicht bekannt, wie lang der zu generierende Satz sein wird.

Wenn dagegen die String-Positionen ignoriert werden, müssen die lexikalischen Einträge nur (!) einmal in die Agenda aufgenommen werden. Da auch dies noch nicht akzeptabel ist, stellt Shieber die in Abschnitt 3.2.2 diskutierte Bedingung der 'semantischen Monotonie' auf. Es werden nur die Unit-Items in die Agenda aufgenommen, die einen Teil der Input-Semantik subsumieren. Shieber weist darauf hin, daß durch die Bedingung der 'semantischen Monotonie' Übergenerierung verhindert wird, d.h. daß die Kohärenz der Generierung gesichert ist (1988, S. 617, Fußnote 10). Andererseits wird durch die Erfolgsbedingung, daß das mit der Input-Semantik versehene Start-Item das abgeleitete Unit-Item subsumieren muß, Vollständigkeit sichergestellt. Shieber behandelt diese Fragen also, aber nicht in einem einheitlichen Rahmen für Parsing und Generierung.

Die Selektionsfunktion wählt wie beim Earley-Parsing immer das am weitesten links stehende noch nicht bearbeitete negative Literal der Nonunit-Items. Dies ist für die Generierung nicht angemessen. Das Subjekt wird dann beispielsweise im Normalfall vor dem Verb generiert. Dadurch sind nicht nur die speziellen Constraints bezüglich der Kongruenz-Information noch nicht bekannt. In einer lexikalisierten Grammatik – Shieber bezieht sich explizit auf die HPSG (1988, S. 617) – wird die Semantik der Argumente sogar erst im Verbeintrag instantiert (s. z.B. Lexikoneintrag (15)). In der Tat können dann beispielsweise alle NPs mit allen verschiedenen Kongruenzmerkmalen generiert werden. Vor diesem Hintergrund macht der Hinweis von Shieber Sinn, daß die Bedingung der 'semantischen Monotonie' eine gewisse Zielgerichtetheit der Generierung sicherstellen soll (ebda). Es werden nur Phrasen generiert, die etwas mit der Zielsemantik 'zu tun haben'.

Shieber sieht die Ursache für die mangelnde Zielgerichtetheit bei der Generierung jedoch darin, daß die String-Positionen ignoriert werden (1988, S. 617). Die Forderung nach 'semantischer Monotonie' soll eine Form von Indexing für die semantische Information darstellen. Dem liegt jedoch die falsche Auffassung zugrunde, daß die Zielgerichtetheit bei der top-down-Earley-Deduktion aus der Tabelle stammt. Tatsächlich aber ist es die (top-down-)Vorhersage, die die Zielgerichtetheit bewirkt (Neumann 1994, S. 37).

Es ist also fraglich, ob die Bedingung der 'semantischen Monotonie' überhaupt sinnvoll ist. Dies wird auch von Gerdemann bezweifelt: *"The reason for this requirement is rather obscure"* (1991, S. 121). Es zieht allerdings nicht die Verbindung zu der für die Generierung ungeeigneten Selektionsfunktion, die hier als mögliche Ursache dieser Bedingung rekonstruiert wird.

In Abschnitt 3.2.2 wurde die Generierung von *er zählt auf dich* mit einem vom semantischen Kopf gesteuerten bottom-up-Generator diskutiert. Auch ein auf Earley-Deduktion beruhender Algorithmus hat mit semantisch nicht kompositionellen Grammatiken jedoch keine Probleme. Genauso wie die VP-Komplement-Regel zum Parsing von NPs verwendet werden kann, ist sie auch zur Generierung von Komplementen geeignet – es müssen nur die entsprechenden Instantiierungen der Merkmale der Komplemente gegeben sein. Dies ist aber eine Frage der Selektionsfunktion, die in den im folgenden behandelten Ansätzen zur top-down-Earley-Deduktion am semantischen Kopf ausgerichtet ist.

Einordnung des Ansatzes Shiebers Architektur wird in der Literatur häufig als bottom-up-Earley-Deduktion bezeichnet.[8] Da jedoch neben der Vervollständigung nur die top-down-Vorhersage von Nonunit-Items verwendet werden kann, gibt es gar keine Möglichkeit, Regeln bottom-up auszuwählen und in die Agenda zu aufnehmen. Wie oben gezeigt wurde (Abschnitt 4.2.2), unterscheiden sich Chart-basierte bottom-up- und top-down-Ansätze nur in der Vorhersageregel, nicht aber in der stets gleichen fundamentalen Inferenzregel. Darüberhinaus gibt Shieber selbst Analysebeispiele an, die eine Einordnung als top-down-Earley-Deduktion rechtfertigen (1988, S. 616/617). Im Start-Zustand befindet sich als erste Vorhersage die S-Regel. Alle weiteren Vorhersagen finden top-down statt. Allerdings hat top-down-Earley-Deduktion durch den Completor auch ein starken bottom-up-Aspekt.

Der Ansatz von Shieber hat als erster das Thema einer uniformen Architektur aufgebracht und ist insofern verdienstvoll. Es werden Probleme wie das Indexing bei der Generierung oder Kohärenz und Vollständigkeit von Parsing und Generierung überhaupt erst aufgeworfen. Shiebers Ansatz ist jedoch durch die strikte Abarbeitung der Elemente von links nach rechts viel besser für das Parsing geeignet als für die Generierung. Durch die Eliminierung des Scanners aus dem Earley-Parsing-Algorithmus bei gleichzeitiger Beschränkung des Completors auf die Tabelle ist er gezwungen, alle möglicherweise benötigten Unit-Items vor der eigentlichen Analyse in die Agenda aufzunehmen. Darüberhinaus wird zur Reduzierung dieser bei der Generierung sehr großen Menge an Unit-Items eine 'semantische Monotonie-Bedingung' aufgestellt, die – ganz abgesehen von der ungünstig gewählten Bezeichnung – offensichtlich überhaupt nicht benötigt wird.

[8] Vgl. z.B. Shieber *et al.* (1990, S. 33), van Noord (1989, S. 1) und König (1994).

4.3.2 Der Ansatz von Gerdemann (1991)

Gerdemann (1991) bezieht sich auf den Ansatz von Shieber (1988) und verbessert ihn in mehreren Punkten:

- die Selektionsfunktion wählt bei der Generierung den semantischen Kopf,
- erweiterter Einsatz von Restriktion,
- Vermeidung mehrfacher Generierung der gleichen Phrase durch Modifikation der Tabelle.

Neben der Behandlung weiterer Themen wird darüberhinaus ein Unifikationsalgorithmus für Merkmalsstrukturen mit Negation und voller Disjunktion vorgestellt. Gerdemann verfolgt jedoch nicht explizit das Ziel, eine uniforme Architektur zu entwickeln.

Parsing Die Parsing-Komponente ist eng an Earleys Parsing-Algorithmus angelehnt. Es werden Scanner, Predictor und Completor eingesetzt, wobei der Scanner sofort eine Reduktion durchführt. Gerdemann verwendet keine Agenda zur Modellierung unterschiedlicher Kontrollstrategien. Es werden immer die Items mit der niedrigesten Position zuerst bearbeitet. Bei mehreren an der gleichen Position beginnenden Items wird das 'älteste' zuerst gewählt. Daraus ergibt sich eine Breitensuche-Kontrollstrategie und eine Abarbeitung des Strings von links nach rechts wie im Earley-Parsing-Algorithmus.

Gerdemann teilt die Chart in Untertabellen ein, die jeweils alle Items, die an einer bestimmten String-Position beginnen, enthalten (s. 1991, S. 30ff).[9] Aufgrund dieser Einteilung reicht zur Bestimmung des nächsten zu bearbeitenden Items ein einfacher Zeiger, der in der Untertabelle, die noch nicht vollständig bearbeitet wurde, jeweils um ein Element nach rechts weitergerückt wird (neue Items werden rechts angehängt).

Die Verwendung von String-Positionen, die die Untertabellen definieren, steigert wie in Shiebers Ansatz die Effizienz des Completors, weil für ein abgeleitetes Unit-Item nur die aktiven Items aus einer einzigen Untertabelle verwendet werden müssen. Es werden nur diejenigen aktiven Items berücksichtigt, die die String-Position aufweisen, von der aus die ursprüngliche Vorhersage gemacht wurde. Um zu dieser ursprünglichen String-Position zurückzufinden, schlägt Gerdemann vor, jedes Item mit einem entsprechenden Index zu versehen (1991, S. 29). Dieser bleibt erhalten, auch wenn sich die Endposition von daraus abgeleiteten Items im weiteren Verlauf der Analyse ändert.

Generierung Wie oben beschrieben, zeigt Gerdemann, daß die Forderung nach 'semantischer Monotonie' für einen auf dem Earley-Algorithmus beruhenden Generator weder sinnvoll noch notwendig ist. Um aus dem Earley-Parser einen Generator zu entwickeln, muß zum einem der Scanner verändert werden. Dazu wird dem Scanner erlaubt, jedes

[9] Dies entspricht der ursprünglichen Beschreibung in (Earley 1970).

lexikalische Item auszuwählen, das mit dem zu bearbeitenden Elements unifiziert (1991, S. 117). Diese erste Version des Scanners wird später (1991, Abschnitt 3.7.1) durch ein effizientes Modell für den lexikalischen Zugriff bei der Generierung ersetzt. Der Grundgedanke ist dabei, lexikalische Einträge aufgrund eines semantischen Index auszuwählen. Jeder Hauptkategorie (NP, VP etc.) wird ein eigener Pfad für die Semantik zugewiesen. Die einzelnen Einträge lassen sich innerhalb einer zu einer Hauptkategorie gehörenden Menge durch einen atomaren Merkmalswert unterscheiden.

Neben dem Scanning muß das Erfolgskriterium an die Generierung angepaßt werden. Ein Item soll als erfolgreich gelten, wenn es ein vollständig abgearbeitetes Start-Item repräsentiert. Das erste so erzeugte Item wird dann den kürzesten möglichen Satz enthalten. Wenn weitere Lösungen erwünscht sind, werden immer längere die Zielbedingung erfüllende Sätze erzeugt. Die Frage nach Kohärenz und Vollständigkeit wird in diesem Zusammenhang jedoch nicht diskutiert (1991, S. 118), sondern später als noch zu lösendes Problem präsentiert (1991, S. 164).

Um auch bei der Generierung eine dem Parsing entsprechende Zielgerichtetheit der top-down-Vorhersage zu erreichen, muß die semantische Information verwendet werden. Unter der Annahme von Chain-Rules in der Grammatik kann die gleiche Form der Vorhersage wie in den bottom-up-Generatoren von van Noord und Shieber *et al.* verwendet werden. Gerdemann schlägt eine Kompilation zur Neuordnung der Elemente auf den rechten Regelseiten vor, um den semantischen Kopf im Falle der Generierung zuerst bearbeiten zu können (1991, S. 118). Durch die Kompilierung kann an der Wahl des jeweils am weitesten links stehenden noch nicht bearbeiteten Elements auch im Falle der Generierung festgehalten werden. Dies entspricht letztlich aber einer am semantischen Kopf orientierten Selektionsfunktion ohne Kompilierung der Grammatik.

Restriktion Gerdemann erweitert die Anwendungsmöglichkeiten für das von Shieber (1985) eingeführte Konzept der Beschränkung von Merkmalsstrukturen (vgl. Abschnitt 2.3.3). Er interpretiert Shiebers Vorschlag in dem Sinn, daß jeder Pfad aus der erlaubten endlichen Menge von Pfaden nur *atomare Werte* besitzen darf (1991, S. 76/77). Shieber erläutert dagegen: "[...] every path in the restricted dag is either undefined, atomic, or *specifically allowed* by the restrictor" (1985, S. 148; Hervorhebung im Original).

In bezug auf die Generierung stellt Gerdemann zunächst fest, daß eine wesentlich größere Menge an Information für die Vorhersage benötigt wird als beim Parsing. Der Wert für die Semantik darf nicht durch Restriktion eleminiert werden. Es lassen sich jedoch Beispiele konstruieren, bei denen die Verwendung einer vollkommen unbeschränkten semantischen Repräsentation zu einer Vergrößerung des Suchraumes führt. Gerdemann läßt aus diesem Grund die Frage offen, ob die semantische Struktur tatsächlich vollkommen unverändert gelassen werden darf (1991, S. 121).

Generell ermöglicht es die Verwendung von beschränkten Merkmalsstrukturen *(restricted DAGs*, kurz *RDs)* erst, daß beispielsweise die VP-Komplement-Regel nicht mehrfach vorhergesagt wird. Darüberhinaus wird die Effizienz des Subsumptions-Tests gesteigert, da die zu vergleichenden Strukturen kleiner sind. Bevor eine Vorhersage gemacht wird, muß eine Beschränkung der entsprechenden Merkmalsstruktur stattfinden. Diese beschränkten Merkmalsstrukturen werden in eine Liste aufgenommen. Wenn es in der Liste schon eine (beschränkte) Merkmalsstruktur aus einer vorhergehenden Vorhersage gibt, findet keine weitere Vorhersage statt. Beim Parsing wird für jede Untertabelle, d.h. für jede Endposition in der Wortkette, jeweils eine anfangs leere Liste angelegt. Dadurch, daß die *RDs* in einer eigenen Liste festgehalten werden, ist es nicht nötig, eine potentiell sehr große Untertabelle zu durchsuchen. Dieses Verfahren hat aber den Nachteil, daß eine neue Datenstruktur mit sehr speziellen Aufgaben in den Algorithmus miteinbezogen werden muß. Darüberhinaus wird bei der Generierung eine andere Liste verwendet (s.u.), so daß die Uniformität beeinträchtigt wird.

Eine weitere Einsatzmöglichkeit von Restriktion betrifft den Completor. Das Ziel ist eine größere Effizienz der Suche nach einem Nonunit-Item für ein gegebenes Unit-Item. In der bisherigen Version des Algorithmus muß der Completor die ganze Untertabelle, die die ursprüngliche Vorhersage, die schließlich zu dem Unit-Item geführt hat, durchsuchen. Gerdemann erörtert verschiedene Möglichkeiten, einen *backpointer*[10] auf die geeigneten Nonunit-Items zu implementieren (1991, S. 81ff). Dies soll hier jedoch nicht im Detail erläutert werden. Das gleiche gilt für eine letzte Einsatzmöglichkeit von Restriktion, die den im originalen Earley-Algorithmus verwendeten *lookahead*-Schritt betrifft. Sie wird von Gerdemann allerdings als 'am spekulativsten' bezeichnet (1991, S. 86).

Indexing Beim Parsing erlaubt die Verwendung von String-Positionen das schnelle Auffinden von Items. Bei der Generierung dagegen verhindern String-Positionen die Möglichkeit, eine bereits erzeugte Phrase an anderen Stellen im Satz einzusetzen, da die String-Position bei einer weiteren Verwendung eine andere ist. Shieber (1988, S. 616) ignoriert deshalb bei der Generierung die String-Positionen. Dann ist aber kein effizienter Zugriff auf die Items der Tabelle mehr möglich. Aus diesem Grund schlägt Gerdemann (1991, Kapitel 3.6.3) vor, weiterhin mit String-Positionen zu arbeiten, aber eine Modifikation der Tabelle vorzunehmen. Wenn zu erkennen ist, daß eine Phrase doppelt generiert wird, wird die Tabelle so verändert, daß 'der Eindruck entsteht', daß die neue Phrase an der gleichen Position wie die alte generiert wird. Durch diesen 'Trick' wird erreicht, daß die gesuchte Phrase sofort gefunden wird. Sie kann dann mit neuen String-Positionen versehen wiederverwendet werden.

Um dies zu erreichen, führt Gerdemann eine globale Liste *GRD (global RD list)* ein, die alle schon verwendeten Restriktoren enthält. Im Gegensatz zum Parsing, das Listen für jeweils eine Untertabelle bzw. String-Position verwendet, muß diese Liste bei der Generierung global sein, um mehrfache Generierungs-Versuche zu erkennen. Zu jedem *RD* dieser Liste muß die Position angegeben werden, an der er zur Vorhersage diente, sowie die Unit-Items, die schließlich aus dem *RD* abgeleitet wurden. Die abgeleiteten

[10] Die Verwendung von Zeigern zum effizienten Auffinden von Chart-Items wird in Minnen *et al.* (1995) weitergeführt und vereinfacht.

Items werden eingefügt, nachdem sie vom Completor gebildet wurden.

Wenn nun für die Vorhersage ein *RD* gebildet wird, der von einem schon in der globalen Liste befindlichen subsumiert wird, kann diese Vorhersage blockiert werden. Dies ist also das gleiche Prinzip wie beim Parsing, nur daß dort eine lokale Liste verwendet wird. Bei der Generierung folgt nun noch ein weiterer Schritt: das aktuelle Nonunit-Item, das also nicht zur Vorhersage dient, wird in die Untertabelle des subsumierenden *RDSs* aus der globalen Liste bewegt (ebda, S. 126). Dadurch erscheint es so, als ob die neu zu generierende Phrase an der alten Position beginnt (da ja die Untertabellen jeweils einer Endposition entsprechen). Die schon erfolgten Ableitungen für die frühere Vorhersage können nun wiederverwendet werden, da sie in der globalen Liste immer in bezug auf die ganze Untertabelle, aus der sie stammen, bezogen werden. Sie gelten dadurch für mehr als nur eine Vorhersage.

Einordnung des Ansatzes Diese Darstellung zeigt, daß für Parsing und Generierung unterschiedliche Listen verwendet werden. Sie haben zum einen einen verschiedenen 'Geltungsbereich' (lokal oder global), zum anderen sind ihre Elemente unterschiedlich strukturiert. Dies beeinträchtigt die Uniformität der Architektur. Das gleiche gilt für die Kompilation der für das Parsing geeigneten Ausgangs-Grammatik in eine weitere für die Generierung geeignete mit dem semantischen Kopf als linke Ecke. Bei dieser Einschätzung muß allerdings eingeräumt werden, daß Uniformität von Parsing und Generierung trotz des expliziten Bezugs auf Shieber (1988) nicht das Ziel Gerdemanns ist.

Der Generator wird als direkte Modifikation des Earley-Parsing-Algorithmus entwickelt. Daraus ergibt sich, daß Parser und Generator nicht als Instanzen eines allgemeinen Deduktionsverfahrens betrachtet werden. Die Modifikationen am Parsing-Algorithmus – insbesondere verursacht durch die Einführung der globalen Liste der Restriktoren – haben zu substantiellen Veränderungen geführt, die sich mit allgemeiner Earley-Deduktion nur schwer vereinbaren lassen.

In bezug auf das Thema dieser Arbeit sind die Ergebnisse zum Problem der 'semantischen Monotonie' und die erweiterten Einsatzmöglichkeiten von Restriktion von Bedeutung. Vor allem aber wendet Gerdemann das Konzept der vom semantischen Kopf gesteuerten Generierung auf den Earley-Algorithmus an. Das Ergebnis ist also eine vom semantischen Kopf gesteuerte top-down-Generierung mit Breitensuche. Darüberhinaus wird die Notwendigkeit zur Wiederverwendung schon generierter Phrasen in Verbindung mit einem effizienten Indexing bei der Generierung deutlich – auch wenn der hier gewählte Weg die Uniformität beeinträchtigt. Im nächsten Abschnitt wird mit dem Ansatz von Neumann (1994) eine konsequente Fortführung des Deduktionverfahrens im Sinne von Pereira und Warren (1983) vorgestellt.

4.3.3 Der Uniforme Tabulare Algorithmus von Neumann (1994)

Neumann (1994) stellt einen *uniformen tabularen Algorithmus (UTA)* vor, der keine vorhergehende Kompilierung der Grammtik notwendig macht, d.h. es handelt sich um einen Interpreter für reversible Grammatiken. Parsing und Generierung werden als Instanzen des gleichen allgemeinen Deduktionsverfahrens gesehen. Sie unterscheiden sich nur in der als Prämisse gegebenen Information. Die grundlegenden ‘Komponenten’ des Algorithmus sind nebem top-down-Vorhersage und Completion der Scanner und eine Agenda. Durch die Verwendung des Scanners ist es nicht wie bei Shieber (1988) notwendig, alle infrage kommenden Unit-Items der Grammatik im voraus auszuwählen. Der Uniforme Tabulare Algorithmus kann deshalb als *on-line*-Algorithmus bezeichnet werden (Neumann 1994, S. 119). Die Verwendung einer Agenda ermöglicht eine flexible Kontrollstrategie. Neumann sieht zunächst nur Breiten- und Tiefensuche vor, doch ist der Algorithmus durch die Agenda auch für komplexere Priortätsfunktionen offen. Die wesentlichen Besonderheiten des Algorithmus werden im folgenden zunächst kurz dargestellt. Im Anschluß wird eine detaillierte Definition des Algorithmus in Pseudo-Code gegeben.

Selektionsfunktion Der Uniforme Tabulare Algorithmus verwendet eine *Daten-gesteuerte Selektionsfunktion*, die sich dynamisch an die jeweils vorhandene Information anpaßt. Zur Bestimmung des nächsten zu bearbeitenden negativen Literals *(selected element)* einer Nonunit-Klausel[11] wird untersucht, ob das erste Element der Wortkette instantiiert ist oder nicht. Ist dies der Fall, kann das am weitesten links stehende Element gewählt werden, da genug Information zu dessen zielgerichteter Bearbeitung vorhanden ist – es handelt sich also um Parsing. Ist dagegen der Wert für die semantische Repräsentation instantiiert, wird der semantische Kopf gewählt. Der Unterschied zwischen Parsing und Generierung liegt also im wesentlichen in der unterschiedlichen Selektionsfunktion.

Die von der Selektionsfunktion berücksichtigten Pfade der Merkmalsstrukturen sind also die Semantik SEM und der Pfad PHON.[12] Diese beiden Merkmale werden als *essential features (Ea)*[13] bezeichnet (Neumann 1994, S. 80). Es wird das am weitesten links stehende Element, dessen Ea instantiiert ist, gewählt. Dies ermöglicht auch eine eindeutige Entscheidung bei zwei (instantiierten) Elementen, deren Semantik mit der des positiven Literals koindiziert ist. Wenn dagegen keines der negativen Literale diese Bedingung erfüllt, wird (gewissermaßen als Default) das überhaupt am weistesten links stehende gewählt. Dies gilt auch für den Fall, daß es gar keinen semantischen Kopf gibt.

Der Vorteil dieses Ansatzes liegt in seiner Flexibilität. So ist es etwa denkbar, andere Präferenzen in der Selektionsfunktion zu berücksichtigen – beispielsweise indem die

[11] Der Bezug zu Definiten Klauseln ist hier beabsichtigt, während in dem Ansatz von Gerdemann (1991) ausdrücklich ‘Chart-Items’ im allgemeinen Sinne verwendet wurde.

[12] Da es beim Parsing von links nach rechts auf der erste Element der Wortliste ankommt, ist der für die Selektionsfunktion relevante Pfad PHON FI.

[13] Die Abkürzung ‘Ea’ steht eigentlich für ‘essential argument’ und bezeichnet dasjenige Argument eines Literals, das zur dessen erfolgreicher, d.h. insbesondere terminierender Bearbeitung notwendig ist (Strzalkowski 1994, S. 182).

Grammatikregeln selbst mit Präferenz-Information versehen werden. Ein von Neumann nicht erwähnter Punkt ist die Möglichkeit, auch einen 'gemischten' Input aus teilweise instantiierter Semantik und teilweise instantiiertem phonologischen String optimal bearbeiten zu können. Es muß nicht zu Beginn der Analyse global entschieden werden, ob es sich um Parsing oder Generierung handelt.

Definition des Algorithmus

Der Uniforme Tabulare Algorithmus arbeitet wie die von Pereira und Warren definierte Earley-Deduktion auf einem feststehenden Programm (den Grammatikregeln und lexikalischen Einträgen) und einem veränderlichen Zustand (der Tabelle). Ob in der Praxis eine oder mehrere Tabellen angelegt werden, ist dabei letztendlich nicht entscheidend. Der Interpreter zerlegt die Menge der Items zur Laufzeit anhand ihrer Indizes in unterschiedliche interne Item-Mengen.

Der Completor wird in drei verschiedene Komponenten unterteilt. *Passive completion* liegt vor, wenn ein passives Item gegeben ist und durch ein aktives Item aus dem Zustand reduziert wird. Umgekehrt liegt *active completion* vor, wenn ein zu bearbeitendes aktives Item durch ein passives Item *aus dem Zustand* reduziert wird. Der Scanner sucht dagegen ein Unit-Item *aus dem Programm* und verwendet es zur Reduktion des gegebenen aktiven Items.

Scanner und Active-Completion unterscheiden sich also nur in ihrem Zugriff auf eine bestimmte Menge von Definiten Klauseln. In Pareira und Warren (1983) wurde dem Reduktions-Schritt dagegen erlaubt, auf Unit-Klauseln aus dem Programm oder dem Zustand zuzugreifen, d.h. er umfaßt Scanner und Active-Completion in Neumann (1994). Dagegen gibt es bei Pereira und Warren keine Entsprechung zur Passive-Completion.[14] Für alle hier erwähnten Reduktions-Schritte gilt, daß nur das jeweils gewählte Element der beteiligten aktiven Items reduziert werden kann. Die anderen negativen Literale werden erst reduziert, wenn sie selbst gewählt sind.

Die top-down-Vorhersage sucht eine Nonunit-Klausel des Programms, deren positives Literal mit dem gewählten Element der zu bearbeitenden Nonunit-Klausel unifiziert. Um Terminations-Probleme mithilfe des Subsumptions-Checks zu vermeiden, wird Restriktion im Sinne von Shieber (1985) in einer sehr zurückhaltenden Form eingesetzt: erlaubt sind alle Pfade, die nicht ausdrücklich eleminiert werden – dies betrifft letztlich nur das Merkmal für die Subkategorisierungs-Liste.

Alle durch Scanner, Completor und Vorhersage gebildeten Items werden zuerst in der Agenda gespeichert. Zuvor wird ihnen durch die Prioritätsfunktion eine Priorität zugeordnet. Das Agenda-Item mit der höchsten Priorität wird jeweils als nächstes bearbeitet. Wenn es kein subsumierendes Item in der Tabelle gibt, wird es in diese eingefügt. Weiterhin werden die Inferenzregeln auf dieses nicht blockierte Item angewendet.

[14] Zur Diskussion s.o. (Abschnitt 4.2.3).

Ein Subsumptions-Test für das Einfügen neuer Items in die Agenda ist nicht notwendig. Er wird erst in bezug auf die Tabelle ausgeführt, wenn ein Item der Agenda bearbeitet werden soll. Dies kann dazu führen, daß als Ergebnis einer Vorhersage mehrere aktive Items in der Agenda vorhanden sind, zwischen denen eine Subsumptions-Beziehung besteht. Diese Redundanz ist jedoch nicht 'gefährlich', da die Bearbeitung der Items letztlich blockiert wird. Der Vorteil der Beschränkung des Subsumptions-Tests auf die Tabelle ist dagegen, daß bei einem Stop des Interpreters nach der ersten gefundenen Lösung der Subsumptions-Test nur auf die tatsächlich verwendeten Items angewendet werden muß.

```
Gegeben sei:

  • Programm P aus unit-Klauseln und
    nonunit-Klauseln

  • Prioritätsfunktion PRIO

  • eine Input-Merkmalsstruktur FS_in

begin PROCEDURE uta(FS_in, PRIO)
1)    init-Agenda(FS_in);
2)    while  not empty − agenda
3)           do exhaust-agenda(PRIO)
4)    find-solutions(FS_in)
end

begin PROCEDURE exhaust-agenda(PRIO)
5)    get-highest-priority-task(TASK);
6)    if blocked(TASK)
7)      then true
8)    else do add-task-to-table(TASK);
9)            apply-task(TASK, PRIO)
end

begin PROCEDURE apply-task(TASK, PRIO)
10)   if passive-item(TASK, PRIO)
11)     then do passive-completion(TASK, PRIO)
12)   else do active-completion(TASK, PRIO, N)
13)     if N = 0
14)       then do prediction(TASK, PRIO);
15)            do scanning(TASK, PRIO)
16)       else true
end
```

Abbildung 4.1: Hauptaufrufe des Uniformen Tabularen Algorithmus *(UTA)*

Erläuterung der Hauptaufrufe als Pseudocodes In Abbildung 4.1 werden die Hauptaufrufe des Uniformen Tabularen Algorithmus in Pseudocode dargestellt. Gegeben ist ein Programm P mit Unit-Klauseln für die Lexikoneinträge und Nonunit-Klauseln für die Grammatikregeln sowie eine Prioritätsfunktion und eine Input-Merkmalsstruktur. Die Prozedur `uta/2` initialisiert zunächst die Agenda mit dem für die top-down-Earley-Deduktion nötigen trivialen Start-Item `(1)`. Nun werden die Elemente der Agenda solange bearbeitet, bis die Agenda leer ist. Die `while`-Schleife ruft die Prozedur `exhaust-agenda/1` mit der Prioritätsfunktion als Argument auf `(3)`. Die Agenda (und später die Tabelle) werden hier nicht als Argumente weitergereicht, sondern es wird davon ausgegangen, daß die verschiedenen Prozeduren Zugriff auf die entsprechenden Datenstrukturen im Speicher haben. Die Prozedur `empty-agenda` überprüft nach der Bearbeitung eines Items durch `exhaust-agenda/1`, ob die Agenda leer ist `(2)`. Ist dies der Fall, kann `find-solutions/1` alle Unit-Klauseln der Tabelle heraussuchen, die als Lösung angesehen werden können (zu Kohärenz und Vollständigkeit s.u.).

Die Prozedur `exhaust-agenda/1` nimmt das Item mit der höchsten Priorität aus der Agenda `(5)` und überprüft, ob es ein subsumierendes Item in der Tabelle gibt, so daß das Agenda-Item blockiert wird. In diesem Fall ist die Bearbeitung des Agenda-Items abgeschlossen `(7)`. Im anderen Fall muß das Item in die Tabelle eingefügt und mit `apply-task/2` bearbeitet werden `(8,9)`.

Die Prozedur `apply-task/2` ruft die Prozedur `passive-completion` auf `(11)`, wenn es sich um ein passives Item handelt, das bearbeitet wird `(10)`. Handelt es sich um ein aktives Item, wird zunächst die Prozedur `active-completion/3` versucht. Wenn es *mindestens eine* Anwendung von `active-completion/3` gibt, ist die Prozedur `apply-tak/2` beendet `(16)`. Nur wenn `active-completion/3` kein einziges Mal erfolgreich war `(13)`, werden Scanner und Vorhersage aufgerufen `(14,15)`. Die Anzahl der Anwendungen von `active-completion/3` wird hier explizit durch die Variable N weitergegeben. Wenn es möglich ist, für das gegebene aktive Item ein passives Item zur Vervollständigung zu finden, bedeutet dies, daß eine weitere Suche nach passenden passiven Items, ausgelöst durch den Scanner oder die Vorhersage, redundant wäre. Es gibt also schon eine abgeleitete Phrase, die den entsprechenden Teil der Wortkette oder semantischen Repräsentation abdeckt.

Erläuterung der Inferenzregeln als Pseudocodes Abbildung 4.2 gibt die Inferenzregeln des Algorithmus sowie eine Hilfsprozedur in Pseudocode wieder. Alle vier Inferenzregeln werden mit einem Item und der Prioritätsfunktion aufgerufen. Es werden dann jeweils soviele Items aus der Tabelle bzw. Klauseln aus dem Programm wie möglich gesucht, die eine bestimmte Bedingung erfüllen. Die Inferenzregeln scheitern nie, d.h. auch dann nicht, wenn keine einzige Anwendung möglich war.

Die Vorhersage-Prozedur `prediction/2` hat als Input (neben $PRIO$) ein aktives Item. Nachdem sie dessen gewähltes Element durch Restriktion auf $RestrictedSE$ beschränkt hat `(1)`, sucht sie Nonunit-Klauseln NC aus dem Programm `(2)`, deren positives Literal

```
begin PROCEDURE prediction(NI_TASK, PRIO)
1)    restrict(NI_TASK, RestrictedSE);
2)    for alle Nonunit-Klauseln NC ∈ P
3)       mit positivem Literal PL_nc do
4)       if RestrictedSE ⊔ PL_nc := NI_inst
5)         then define-new-task(NI_inst, PRIO)
6)       else true
end

begin PROCEDURE passive-completion(UC_TASK, PRIO)
7)    for alle Nonunit-Items NI ∈ Tabelle T
8)       mit gewähltem Element SE_NI
8)       if UC_TASK ⊔  SE_NI = NI_inst
9)         then ReducedNI := NI_inst − UC_TASK
10)             define-new-task(ReducedNI, PRIO)
11)      else true
end

begin PROCEDURE active-completion(NI_TASK, PRIO, N)
12)   for alle Unit-Items UI ∈ Tabelle T
13)      if SE_NI_TASK ⊔  UI = NI_inst
14)        then ReducedTASK = NI_TASK_inst − UI
15)             define-new-task(ReducedTASK, PRIO);
15)             N := N + 1
16)      else true
end

begin PROCEDURE scanning(NI_TASK, PRIO)
17)   for alle Unit-Klauseln UC ∈ P
18)      if SE_NI_TASK ⊔  UC = NI_TASK_inst
19)        then ReducedTASK := NI_TASK_inst − UC
20)             define-new-task(ReducedTASK, PRIO)
21)      else true
end

────────────────────────

begin PROCEDURE define-new-task(NI_inst, PRIO)
1)            assign-priority(NI_inst, PRIO, NI_PRIO);
2)            select-element(NI_inst, SE_NI_inst);
3)            add-task-to-agenda(NI_inst, SE_NI_inst, NI_PRIO)
end
```

Abbildung 4.2: Inferenzregeln des Uniformen Tabularen Algorithmus *(UTA)* mit dem beschränkten gewählten Element des aktiven Input-Items *RestrictedSE* unifiziert `(4)`. Ist die Unifikation erfolgreich, wird die nun *weiter instantiierte* Nonunit-Klausel NC_{inst} aus dem Programm durch `define-new-task/2` in die Agenda aufgenommen.

Die Prozedur `passive-completion/2` erhält als Input ein passives Item. Es werden aktive Items aus der Tabelle ausgewählt, deren gewähltes Element mit dem passiven Item unifiziert `(9)`. Das gefundene aktive Item wird nun zu *ReducedNI* reduziert. Dies ist das inzwischen weiter instantiierte aktive Item ohne sein gewähltes Element `(10)`. Das reduzierte Item wird schließlich in die Agenda aufgenommen `(11)`.

Die Prozedur `active-completion/3` besitzt als einzige Inferenzregel ein drittes Argument N, das als Zähler für die Anzahl der Anwendungen dient. Der Grund dafür ist, daß bei mindestens einer Anwendnung dieser Prozedur auf Vorhersage und Scanning für das betreffende Item verzichtet werden kann (s.o.). Der Zähler steht zu Beginn der Prozedur auf 0.[15] Der Input der Prozedur ist ein Nonunit-Item NI_{task}, für das Unit-Items *aus der Tabelle* gesucht werden `(13)`, die mit dem gewählten Element von NI_{task} unifizieren `(14)`. Das nun weiter instantiierte Nonunit-Item wird um das Unit-Item (bzw. das gewählte Element) zu *ReducedTASK* reduziert `(15)`. Jetzt wird das reduzierte Item in die Agenda eingefügt `(16)` und der Zähler N um 1 erhöht `(17)`.

Die Prozedur `scanning/2` erhält ebenfalls als Input ein aktives Item. Sie wählt Unit-Klauseln *UC aus dem Programm* `(19)`, wenn das gewählte Element des Nonunit-Items mit der Unit-Klausel unifiziert `(20)`. Das weiter instantiierte Nonunit-Item wird nun um das Unit-Item (bzw. das gewählte Element) vermindert `(21)` und in die Agenda aufgenommen `(22)`.

Alle gefundenen Items und Klauseln werden also durch die Prozedur `define-new-task/2` als neue Tasks in die Agenda aufgenommen. Dazu wird allen Klauseln C gemäß der zu Beginn der Verarbeitung festgelegten Prioritätsfunktion $PRIO$ eine Priorität C_{prio} zugewiesen `(24)`. Die Prozedur `select-element/2` bestimmt das gewählte Element SE_C für Klausel C `(25)`. Nun kann C zusammen mit der Information über die Priorität und das gewählte Element in die Agenda eingefügt werden `(26)`.

Uniformes Indexing Um dem Completor einen effizienten Zugang zu den relevanten Items zu ermöglichen, werden beim Parsing abgeleitete Items aufgrund der abgearbeiteten Wortkette indiziert; bei der Generierung erzeugte Items werden dagegen nach ihrer semantischen Information indiziert. Das Ergebnis sind zwei unterschiedliche Tabellen für Parsing und Generierung. Dem Indexing liegt ein einziger uniformer Mechanismus zugrunde, der für die jeweilige Aufgabe parametrisiert wird. Diese Parametrisierung geschieht wie bei der Selektionsfunktion durch das essentielle Merkmal Ea, das als Index verwendet wird. Bei Nonunit-Items bestimmt das essentielle Merkmal des gewählten Elements den Index. Das aus einem Scanning-Schritt resultierende Item wird einen anderen Index bekommen als das Input-Item, wenn bei der Reduktion ein nicht-leerer Teil von dessen

[15] Feinere technische Details für die Weitergabe von Variablen sollen im Pseudocode nicht berücksichtigt werden. So werden bespielsweise in Prolog durch ein Scheitern Variablenbindungen wieder aufgehoben.

Ea verbraucht wird. Bei passiver und aktiver Vervollständigung müssen die jeweils zu suchenden Items den gleichen Index haben (d.h. in der gleichen Untertabelle stehen) wie das Item, das gerade bearbeitet wird. Der Grund dafür ist, daß eine Vorhersage ab der gleichen Position bzw. für die gleiche Semantik gemacht worden sein muß wie das passive Item, damit es 'paßt'. Dadurch wird die Suche auf einen kleinen Teil der Items beschränkt.

Der Vorteil der Verwendung einer eigenen Tabelle für die Generierung ist die Möglichkeit, erzeugte Phrasen an anderen Positionen im Satz zu verwenden. An den durch das jeweilige essentielle Merkmal definierten Untertabellen läßt sich die Traversierung der Input-Struktur erkennen. Beispielsweise besteht beim Parsing der Index des Start-Sets (dies wird zunächst nur durch das Start-Item definiert) aus der ganzen Wortkette. Alle Indizes für später bearbeitete Phrasen, d.h. Teile des Satzes, sind durch die jeweils abgedeckte Teilkette – das jeweilige essentielle Merkmal – gekennzeichnet. Die Indizes bei der Generierung geben dagegen die Dekomposition der Funktor-Argument-Struktur durch die Grammatik wieder.

Neumann und van Noord (1992) sowie Neumann (1994, Kapitel 5) stellen eine Methode vor, Parsing und Generierung miteinander zu verzahnen. Durch während der Generierung mitlaufendes Parsing erfolgt eine Kontrolle des erzeugten Strings (Monitoring) mit der Möglichkeit zu Korrekturen. Um diesen Ansatz mit dem Uniformen Tabularen Algorithmus effizient umsetzten zu können, müssen Items aus der einen Verarbeitungs-Richtung für die andere verwendet werden. Dazu wird eine *item sharing*-Methode eingeführt. Sie hebt den potentiellen Nachteil der Aufteilung der Items in zwei verschiedenen Tabellen, daß nämlich die abgeleiteten Items nur für eine Verarbeitungs-Richtung zur Verfügung stehen, wieder auf. Es werden nur die *Unit*-Items, also vollständig bewiesene Klauseln, zwischen Parsing und Generierung geteilt. Der Grund dafür ist, daß bei Nonunit-Items das Ea nur für jeweils eine Verarbeitungs-Richtung instantiiert ist. Bei passiven Items sind dagegen die abgedeckte Wortkette und dessen Semantik bekannt, so daß beispielsweise ein aus dem Parsing stammendes Item anhand seiner Semantik sofort auf die entsprechende Untertabelle der Generierung bezogen werden kann.

Kohärenz und Vollständigkeit Es ist nicht nötig, am Ende der Verarbeitung einen expliziten Test auf Kohärenz und Vollständigkeit für alle Items durchzuführen, um mögliche Ergebnis-Items zu finden. Kohärenz wird durch den Scanner sichergestellt. Er darf lexikalische Einträge nur dann auswählen, wenn das erste Element des Strings bzw. die gegebene Semantik mit den korrespondierenden Werten des Lexikon-Eintrags identisch sind. Um zu verhindern, daß Lexikoneinträge mit 'leerer Semantik', beispielsweise repräsentiert durch eine Variable als semantischer Wert, bei der Generierung immer ausgewählt werden können, muß der semntische Wert durch ein Atom wie z.B. *nil* instantiiert sein. Die Grundidee ist also, daß nur die Unit-Klauseln verwendet werden können, die explizit einen Teil der Input-Struktur abdecken und 'verbrauchen'. Dadurch ergibt sich eine interessante Parallele zur oben erläuterten 'guide-consumption condition' von Dymetmann *et al.* (1990, S. 93).

Vollständigkeit kann durch die Verwendung des Ea als Index sehr einfach und effizient überprüft werden: wenn es ein Unit-Item gibt, das die Input-Struktur in bezug auf das Ea vollständig abdeckt, dann muß sich dieses auch in derselben Item-Menge wie das Start-Item befinden. Das gesuchte Unit-Item hat also immer den gleichen Index wie das Start-Item.

Einordnung des Ansatzes Der Uniforme Tabulare Algorithmus *(UTA)* von Neumann (1994) stellt eine konsequente Erweiterung des Earley-Deduktionsverfahrens von Pereira und Warren (1983) auf Parsing und Generierung dar. Beide Verarbeitungs-Richtungen sind im Gegensatz zu Shieber (1988) in gleicher Weise effizient. Darüberhinaus werden sie in Hinblick auf das Indexing der Items absolut symmetrisch behandelt, d.h. es handelt sich nicht um eine Adaption eines Parsing-Algorithmus für die Generierung (wie bei Gerdemann (1991)).

Durch die dynamische Selektionsfunktion kann sich der Algorithmus flexibel an den jeweils zur Verfügung stehenden Teil des Inputs anpassen. Alle in Neumann (1994) angeführten Beispiele gehen jedoch davon aus, daß zu Beginn der Analyse entweder die Wortkette oder die Semantik vollständig instantiiert sind, so daß die mögliche Flexibilität der Selektionsfunktion nicht völlig ausgeschöpft wird. Durch die Verwendung des 'essentiellen Merkmals' wird darüberhinaus die Unterscheidung von Parsing und Generierung wieder eingeführt. Die Entscheidung zwischen Parsing und Generierung muß jedoch nicht global erfolgen, sondern erfolgt lokal in bezug auf das jeweilige Item in der Selektionsfunktion und dem Indexing-Mechanismus. Dies ist aufgrund der größeren Flexibilität eine Verbesserung gegenüber vorhergehenden Ansätzen.

Zu den Vorteilen des Ansatzes gehört, daß die Frage nach Kohärenz und Vollständigkeit nicht nur explizit beantwortet wird, sondern daß diese Eigenschaften auch auf effiziente Weise sichergestellt werden. Die Verwendung des essentiellen Merkmals als Index ermöglicht die Wiederverwendung von passiven Items in der jeweils anderen Verarbeitungs-Richtung – abgesehen davon, daß die Wiederverwendung der gleichen Phrase bei der Generierung praktisch automatisch geschieht.

Der Algorithmus eignet sich als Grundlage für weitere Modifizierungen und Experimente im Rahmen von top-down-Earley-Deduktion, beispielsweise durch die Einführung von komplexeren Präferenzen in der Agenda. Die Funktionsweise des Algorithmus ist im Grunde sehr einfach. Wie oben erwähnt, haben bereits Pereira und Warren die Möglichkeit angesprochen, andere, z.B. am Kopf einer Phrase orientierte Selektions-Funktionen zu realisieren (1983, S. 141). Es ist deshalb erstaunlich, daß keiner der vorhergehenden Ansätze das Earley-Deduktionsverfahren in dieser Konsequenz umgesetzt hat.

4.4 Bottom-up-Earley-Deduktion

Allen bisher vorgestellten Ansätzen ist gemeinsam, daß sie ausgehend von dem zu beweisenden Ziel den Nonunit-Klauseln ‘entgegen den Implikationspfeil’ folgen, bis sie schließlich auf Unit-Klauseln, d.h. Fakten treffen, die die Ableitung rechtfertigen (backward chaining). Dies kann als ein sukzessives Rewriting des Antezedens beschrieben werden (s. Abschnitt 4.1.1). Da die Suche nach passenden Unit-Klauseln unter Verwendung der im Ziel gegebenen Information geschieht, handelt es sich um eine zielgerichtete Vorgehensweise. Die resultierende Baum-Traversierung ist top-down.

In Abschnitt 4.2.2 wurde erläutert, daß auch eine bottom-up-Analyse mit Chart-basierten Algorithmen möglich ist. Das korrespondierende allgemeine Beweisverfahren ist entsprechend *bottom-up-Earley-Deduktion*. Ausgehend von den Unit-Klauseln werden sukzessive die Elemente des Antezedens von Nonunit-Klauseln duch das jeweilige Konsequens ersetzt. Um eine Ersetzung durch das Konsequens zu rechtfertigen, müssen *alle* Elemente des Antezedens vorhanden sein. Auch dies kann als ein Rewriting beschrieben werden, allerdings in umgekehrter Richtung. Zur Erläuterung soll das Beispiel (26) wieder aufgenommen werden:

(42)
$$\begin{array}{l} ? \leftarrow \texttt{die} \wedge \texttt{katze} \wedge \texttt{schläft} \\ ? \leftarrow det \wedge \texttt{katze} \wedge \texttt{schläft} \\ ? \leftarrow det \wedge n \wedge \texttt{schläft} \\ ? \leftarrow np \wedge \texttt{schläft} \\ ? \leftarrow np \wedge v \\ ? \leftarrow np \wedge vp \\ s \leftarrow s \end{array}$$

Es ist kein Ziel gegeben: erst wenn alle Ersetzungen stattgefunden haben, steht das Ergebnis als der ‘höchste’ Knoten im Analysebaum fest. In dieser Darstellung mag die Vorgehensweise zielgerichetet erscheinen. Dies liegt jedoch daran, daß die Menge der zur Verfügung stehenden Nonunit-Klauseln auf die tatsächlich benötigten eingeschränkt ist.

Bei reiner bottom-up-Earley-Deduktion gibt es also keine Vorhersagen, die als Hypothesen interpretiert werden könnten. Dies wäre dann der Fall, wenn durch das Vorhandensein eines Elements des Antezedens eine zielgerichtete Vorhersage und Überprüfung der noch fehlenden Elemente ausgelöst würde. Wenn nach den ersten Elementen im Antezedens von Nonunit-Klauseln gesucht wird – also den ‘linken Ecken’ –, ergibt sich ein left-corner-Earley-Deduktionsverfahren (dies wird von dem left-corner-Chart-Parser in Kilbury (1985) realisiert). Wird ein anderes Element des Antezedens gewählt, bespielsweise ein als ‘Kopf’ definiertes, resultiert ein Kopf-gesteuertes Earley-Deduktionsverfahren. Letzteres wird allerdings schwieriger auf andere Nonunit-Klauseln als solche, die Grammatikregeln repräsentieren, anzuwenden sein, da eine geeignete Definition des Kopf-Elements im allgemeinen Fall problematisch erscheint. Earley-Deduktionsverfahren, die diese Mischung aus bottom-up- und top-down-Analyse vollziehen, sind meines Wissens bisher nicht im Rahmen von uniformen Architekturen für Parsing und Generierung eingesetzt worden. Eine Realisierung scheint grundsätzlich jedoch nicht schwierig zu sein – schließlich ist der einzige wesentliche Unterschied zu den korrespondierenden bekannten Parsing- und Generierungs-Strategien mit Backtracking die Verwaltung der Alternativen in einer Tabelle. Im folgenden soll der Ansatz von Erbach (1995a) erläutert werden.

4.4.1 Erbach (1995)

Erbach (1995a)[16] definiert ein reines bottom-up-Earley-Deduktionsverfahren, das auf Parsing und Generierung angewendet wird. Die von Erbach aufgeführten Vorteile eines solchen Ansatzes (1994, S. 796 und 1995, S. 69) sollen zunächst genannt und diskutiert werden.

Daten-gesteuerte Verarbeitung Wie alle bottom-up-Verfahren hat auch bottom-up-Earley-Deduktion unmittelbaren Zugang zu lexikalischer Information, also den Unit-Klauseln des Programms. Im Hinblick auf lexikalisierte Grammatiken mit entweder sehr allgemeinen Regelschemata (HPSG) oder sehr vielen Syntax-Regeln ist der Vorhersage-Schritt der top-down-Strategien problematisch. Im ersten Fall kann es nur sehr allgemeine Vorhersagen geben, im zweiten gibt es sehr viele Vorhersagen.

In beiden Fällen führen die Vorhersagen zu einer drastischen Erhöhung der Kombinations-Möglichkeiten der Items und damit zu Ineffizienz, wie mit der im nächsten Kapitel erläuterten Implementierung von top-down-Earley-Deduktion gezeigt werden kann: Wenn bespielsweise statt einer Regel mit der kontextfreien Basis $VP \leftarrow VP \wedge NP$ die Regel $VP \leftarrow VP \wedge XP$ verwendet wird (XP soll nicht auf eine bestimmte syntaktische Kategorie festgelegt sein), dann ergibt sich sofort eine wesentlich größere Anzahl an Kombinations-Möglichkeiten.

Neumann (1994, S. 117) argumentiert zugunsten des top-down-Ansatzes, daß es bei der Verarbeitung von HPSG-Schemata nur sehr wenige und allgemeine Vorhersagen gibt, ehe die lexikalischen Einträge erreicht sind. Obwohl dem zuzustimmen ist, weil die Earley-Deduktion durch die eingebaute Breitensuche immer auch versucht, den Scanner zu verwenden, werden jedoch von Neumann die durch die ersten Vorhersagen möglichen Kombinationen nicht berücksichtigt. Die ursprünglichen Vorhersagen, die sich daraus ergeben, daß top-down begonnen wird, 'stören' gewissermaßen die weitere Verarbeitung. Sie sind so allgemein, daß sie sehr häufig zur Kombination verwendet werden können. Da Earley-Deduktion immer alle Anwendungs-Möglichkeiten der Inferenzregeln ausschöpfen muß, um die Vollständigkeit (des Beweisverfahrens) zu garantieren, gibt es auch keine Möglichkeit, das Problem zu lösen, wenn die entsprechenden aktiven Items einmal in die Tabelle

[16] Dieser Erörterung liegt der **Entwurf** der Dissertation vom 31. August 1995 zugrunde. Dieser ist jedoch öffentlich zugänglich gemacht worden (s. Literaturverzeichnis). Darüberhinaus ist mit Erbach (1994) ein Aufsatz 'regulär' veröffentlicht worden, der sich inhaltlich stark mit den hier relevanten Teilen der Dissertation deckt.

aufgenommen wurden.

Subsumptions-Check Bei reiner bottom-up-Verarbeitung kann der Subsumptions-Check ausgelassen werden. Es besteht keine Gefahr einer Vorhersage-Schleife, da es keine Vorhersage gibt. Aufgrund des hohen Aufwands für den Subsumptions-Check ist dies ein großer Vorteil der bottom-up-Earley-Deduktion. Allerdings besteht dann die Möglichkeit, daß durch unechte Mehrdeutigkeiten in der Tabelle wiederum unerwünschte Kombinationen von Items stattfinden. Wenn der Subsumptions-Check also ausgelassen werden soll, muß sichergestellt sein, daß die Grammatik keine unechten Mehrdeutigkeiten erlaubt.

Inkrementalität Earley-Deduktion ist grundsätzlich gut geeignet für eine inkrementelle Verarbeitung, weil die logischen Konsequenzen aus einem neuen Unit-Item (also einem Fakt bzw. einer Proposition) in bezug auf alle Items der Tabelle sofort gezogen werden können. Dies gilt sowohl für top-down- wie auch für bottom-up-Earley-Deduktion. Erbach (1994, S. 796 und 1995, S. 69) stellt die Fähigkeit zu inkrementeller Verarbeitung als einen besonderen Vorteil der bottom-up-Verarbeitung heraus. Dieser kann darin gesehen werden, daß das Scanning eines Wortes sofort geschieht, ohne daß vorher eine Vorhersage stattgefunden haben muß, die das Scanning rechtfertigt.

Präferenzen Ein weiterer Vorteil von bottom-up-Earley-Deduktion liegt nach Erbach (ebda.) darin, daß eine Suchstrategie mit Präferenzen modelliert werden kann, wenn die lexikalischen Einträge mit der entsprechenden Information versehen werden. Obwohl bei bottom-up-Verfahren zusammen mit der lexikalischen Information auch die Präferenzwerte unmittelbar vorliegen, sind jedoch auch alle anderen Earley-Deduktionsverfahren bei Verwendung einer Agenda zur Präferenz-basierten Verarbeitung fähig. Wie oben angedeutet, scheint die Schwierigkeit grundsätzlich darin zu liegen, die Präferenz eines Items festzulegen. Wenn nur lexikalische Information mit Präferenzen versehen ist, wird die Vorhersage bei der top-down-Verarbeitung zunächst also ohne Präferenzen ausgeführt.

Dieses Problem ist meiner Ansicht nach entweder dadurch zu lösen, daß auch die Nonunit-Klauseln des Programms Präferenzwerte erhalten oder daß ein 'neutraler Mittelwert' im Falle des Fehlens von Präferenz-Information angenommen wird. Dieses Thema kann hier nicht weiter vertieft werden. Es scheint jedoch plausibel, daß Präferenzwerte für lexikalische Information relativ leicht festgelegt werden können. In diesem Fall ist bottom-up-Earley-Deduktion in der Tat gut geeignet, diese Information unmittelbar zu nutzen. Inbesondere kann für alle abgeleiteten Unit-Items eine neue Präferenz aufgrund der Präferenzen der zu seiner Bildung verwendeteten Unit-Items berechnet werden.

Algorithmus

Das bottom-up-Earley-Deduktionsverfahren verwendet die gleiche fundamentale Inferenzregel (den Completor) wie die top-down-Variante (s. (35)). Wie bereits erwähnt, gibt es keine Vorhersage. Um dem Completor den Zugriff auf die Unit-Klauseln des Programms zu ermöglichen, wird ein Scanner verwendet. Daraus ergibt sich, daß Erbach genauso wie die anderen hier vorgestellten auf Earley-Deduktion beruhenden Ansätze dem Completor selbst keinen Zugang zum Programm gibt – im Gegensatz zu Pereira und Warren (1983).

Um eine größere Zielgerichtetheit der bottom-up-Analyse zu erreichen, werden nur diejenigen Unit-Items in die Tabelle aufgenommen, die *zum Beweis des Zieles beitragen* (vgl. Erbach 1994, S. 797). Es liegt hier also wie bei Shieber (1988) die Konzeption zugrunde, daß alle Unit-Klauseln vor dem Beginn der Verarbeitung *off-line* ausgewählt werden. Die später zu klärende wichtige Frage ist, wie im voraus festgestellt werden soll, welche Unit-Klauseln für den Beweis benötigt werden.

Die grundlegenden Operationen des Algorithmus sind einfach. Die Agenda wird zunächst mit allen infrage kommenden Unit-Items initialisiert. Diese werden mit einem oder mehreren Indizes versehen, die jeweils bestimmte Teile der Information der Unit-Klausel repräsentieren. Die Verwendung von Indizes dient – wie auch die Beschränkung der Menge der Unit-Klauseln – dazu, die Verarbeitung stärker zielorientiert zu machen. Weiterhin werden alle Nonunit-Klauseln des Programms in die Agenda aufgenommen (zu einer Ausnahme s.u.). Aufgrund der relativ kleinen Anzahl an Regeln zumindest im Falle lexikalisierter Grammatiken erscheint dies weniger problematisch. Es ist jedoch anzumerken, daß die Vorhersage bei der top-down-Verarbeitung geschickter agiert, weil sie nur die tatsächlich benötigten Nonunit-Items unmittelbar zu ihrer Verwendung auswählt.

Solange die Agenda nicht leer ist, wird das Item mit der jeweils höchsten Priorität ausgewählt. Dieses wird in jedem Fall bearbeitet und in die Tabelle aufgenommen, da es keinen Subsumptions-Test gibt, der dies blockieren kann. Zur Bearbeitung muß zwischen Unit-Klauseln und Nonunit-Klauseln unterschieden werden. Im ersten Fall werden Nonunit-Items in der Tabelle zur Reduktion gesucht ('passive completion' nach Neumann (1994)); im zweiten Fall werden passende Unit-Items aus der Tabelle gesucht ('active completion'). Eine Reduktion findet statt, wenn die beiden betreffenden Literale unifizieren und ihre Indizes eine Kombination erlauben. Die zweite Bedingung ist von besonderer Bedeutung und wird weiter unten näher erläutert.

Scanning beim Parsing Der Scanning-Schritt, der zur Initialisierung der Agenda dient, richtet sich nach der Input-Struktur. Beim Parsing werden die Nonunit-Klauseln und alle lexikalischen Einträge für die Elemente der Wortkette gesucht. Dabei können die gewählten Unit-Klauseln weiter instantiiert werden: in einer Grammatik, die Differenz-Listen für die Strings konkateniert, wird die variable Restliste der lexikalischen Einträge mit dem jeweiligen Rest der Wortkette instantiiert, so daß die Kombination von nicht adjazenten Wörtern verhindert wird (dies gilt für alle Formen von Earley-Deduktion).

Neben den Elementen der Wortkette werden auch alle 'leeren Elemente' der Grammatik in die Agenda aufgenommen (1995a, S. 71). Beim reinen Parsing kann dies nur Wörter betreffen, die eine *leere phonologische Form* besitzen – beispielsweise leere Elemente für bestimmte syntaktische Analysen mit *gap-filler*-Konstruktionen. Bei der Generierung dagegen müssen *semantisch leere* Elemente berücksichtigt werden. In beiden Fällen ist die Motivation zur Aufnahme der betreffenden Unit-Klauseln, daß sie zwar zum Beweis benötigt werden könnten, aber in der gegebenen Input-Struktur nicht erkennbar sind.

Scanning bei der Generierung Grundsätzlich müssen bei der bottom-up-Verarbeitung alle Grammatikregeln zur Verfügung stehen, d.h. in die Agenda bzw. die Tabelle eingefügt werden. Bei einer Aufteilung der Grammatikregeln in Chain- und Nonchain-Rules kann jedoch eine Ausnahme gemacht werden. Es werden nur diejenigen Nonchain-Rules ausgewählt, deren positives Literal mit einer Teilstruktur der gegebenen Input-Semantik unifiziert werden kann (Erbach 1995a, S. 85/86). Dies führt nicht zur Unvollständigkeit des Interpreters, da eine Regel, die mit keinem Teil der Input-Semantik unifiziert, auch während der Verarbeitung nicht verwendet werden kann. Dagegen werden alle Chain-Rules ausgewählt, weil deren Semantik nur aus einer Variablen besteht und somit mit jeder semantischen Repräsentation unifiziert.

Es werden alle Unit-Klauseln des Programms ausgewählt, deren Semantik entweder vollständig der gegebenen Input-Semantik entspricht, oder die eine Teilstruktur der Input-Semantik enthalten. Bei der zweiten Möglichkeit muß die Input-Semantik in ihre Teile dekomponiert werden. Dazu werden entweder die Argumente eines Prädikats oder etwa der Wert für die semantische Rolle, beispielsweise AGENT oder PATIENT, isoliert. Der Scanner muß also auf die jeweilige Grammatik eingestellt werden, um eine angemessene Dekomposition vornehmen zu können. Enthält die Input-Semantik Quantoren, so müssen diese mitberücksichtigt werden. Dazu werden spezielle Dekompositions-Regeln verwendet (vgl. Erbach 1995, S.85).

Diese Definition des Scanning-Schrittes erlaubt nur 'semantisch monotone' Grammatiken. Es werden alle Elemente ausgewählt, deren Semantik eine Teilstruktur der Input-Semantik subsumiert. Wie in Abschnitt 3.2.2 gezeigt, ist dies beispielsweise bei bestimmten Analysen von *to kick the bucket* nicht der Fall. Es ist zu anzumerken, daß die Forderung nach 'semantischer Monotonie' bei der bottom-up-Earley-Deduktion wesentlich sinnvoller ist als bei top-down-Ansätzen. Werden alle Unit-Items des Programms in die Agenda aufgenommen, entsteht eine völlig ungerichtete Suche, da es keine Vorhersage gibt.

Damit auch 'semantisch nicht-monotone' Grammatiken verarbeitet werden können, muß der Scanner modifiziert werden. Erbach (1995a, S. 87) schlägt vor, schon während des Scannings die Grammatikregeln zu benutzen. Bei Nonchain-Rules wird das positive Literal mit einer Teilstruktur des Input-Semantik instantiiert und der Scanning-Schritt rekursiv für die negativen Literale, also die Tochterknoten im lokalen Baum, wiederholt. Bei Chain-Rules dagegen wird zusätzlich der semantische Kopf mit einem lexikalischen Eintrag instantiiert. Dann werden die anderen Tochterknoten ebenfalls rekursiv bearbeitet. Da dies zu einem sehr großen Suchraum und mehrfachen Items führen kann, schlägt Erbach einen Kompilierungs-Schritt vor, der eine Tabelle der Unit-Items anlegt, die aber erst zur Laufzeit ausgewählt werden. In Erbach (1995a, S. 87) werden jedoch keine genauen Erläuterungen gegeben, obwohl es sehr bemerkenswert ist, daß bei der bottom-up-Analyse das Scanning zur Laufzeit ausgeführt werden soll.

Dieses Vorgehen ist aus mehrfacher Sicht nicht überzeugend. Abgesehen davon, daß es sehr aufwendig ist, findet vor der eigentlichen bottom-up-Earley-Deduktion eine weitere Verarbeitung statt. Diese geht nicht bottom-up vor, sondern top-down und Kopf-gesteuert. Meiner Ansicht nach kann das Scanning auch für 'semantisch nicht-monotone' Grammatiken wesentlich einfacher erfolgen. Es müssen nur neben dem Wert für die eigentliche Semantik der Einträge (beispielweise mit dem Pfad SEM) auch alle in dem Eintrag selbst gegebenen Informationen berücksichtigt werden, z.B. die der Subkategorisierungs-Liste. Grundsätzlich kann man sagen, daß alle gefundenen lexikalischen Einträge rekursiv daraufhin untersucht werden müssen, ob sie instanttierte Werte für die *Merkmale* (nicht die vollständigen Pfade) PHON und SEM besitzen – dies entspricht den 'essentiellen Argumenten'. Wenn also bespielsweise ein lexikalischer Eintrag einen atomaren Wert für den Pfad SYN SC FI PHON besitzt, dann wird der gefundene Wert als Ausgangspunkt für einen weiteren Scanning-Schritt unternommen, etwa für *bucket*. Dies erfordert zwar eine eingehendere Untersuchung jedes gefundenen Unit-Items, aber es erlaubt eine uniforme Definition des Scanners für Parsing und Generierung ('suche rekursiv lexikalische Einträge für alle instantiierten Werte der essentiellen Merkmale'). Darüberhinaus muß keine globale Entscheidung für eine bestimmte Verarbeitungs-Richtung erfolgen.

Indexing Grundsätzlich dient die Verwendung von Indizes dazu, die Suche nach Items in der Tabelle, die zur Kombination mit einem gegebene Item geeignet sind, auf diejenigen Items einzuschränken, die zum Beweis beitragen können. Dies ist vor allem bei der bottom-up-Earley-Deduktion wichtig, da sie im Gegensatz zur top-down-Variante nicht zielgerichtet ist. Erbach verwendet deshalb als Indizes String-Positionen beim Parsing und die semantische Repräsentation bzw. einen eindeutigen Bezeichner für diese bei der Generierung.

Wie van Noord (1993) erweitert Erbach die Möglichkeiten zur Kombination von Konstituenten. Es werden verschiedene Operationen eingeführt, die beispielsweise die Kombination nicht-adjazenter Phrasen erlauben. Die hinter diesen Operationen stehende linguistische Motivation sowie ihre formalen Konsequenzen sollen hier nicht weiter ausgeführt werden. Für das Indexing ergeben sich verschiedene Schemata, die die Kombinierbarkeit von Items mit bestimmten Indizes bestimmen. So wird beispielsweise der 'normalen' String-Konkatenation das Schema *directional adjacent combination* zugewiesen (Erbach 1995a, S. 73).

Einordnung des Ansatzes Mit der Definition des Scanning-Schrittes vor dem eigentlichen Beginn der Analyse wird eine wesentliche Eigenschaft der bottom-up-Earley-Deduktion festgelegt. Die Kombination von Items selbst erfolgt ungerichtet und bedarf insofern keiner differenzierten Beschreibung: rekursiv werden die Nonunit-Items durch die vorhandenen Unit-Items reduziert. Über das Vorhandensein der Unit-Items bestimmt jedoch der Scanner. Eine Nonunit-Klausel muß vollständig reduziert sein, damit eine bottom-up-Verknüpfung zu dem 'nächsthöheren Knoten' möglich ist.

Parsing und Generierung werden vollkommen parallel behandelt. Es gibt keine möglicherweise unterschiedliche Selektionsfunktion. Unit-Items reduzieren diejenigen negativen Literale von Nonunit-Items, mit denen sie unifizieren. Die Reihenfolge, in der dies geschieht, hat keine Auswirkung auf die Effizienz. Diese ist bei Parsing und Generierung jedoch sehr unterschiedlich. Beim Parsing werden nur wenige nicht relevante Lexikoneinträge ausgewählt. Falls es mehrere Lexikoneinträge für eine gegebene Wortform gibt, wird ihre Anzahl vergleichsweise gering sein. Bei der Generierung dagegen müssen alle morphologischen Formen eines Wortes mit einer bestimmten Semantik berücksichtigt werden. Gerade bei morphologisch sehr reichen Sprachen führt dies zu einer nicht akzeptablen Anzahl von Kombinations-Möglichkeiten, von denen letztendlich nur ein sehr kleiner Anteil für den Beweis benötigt wird. Diese Überlegung setzt jedoch schon voraus, daß die Grammatik 'semantisch monoton' ist – eine bei der reinen bottom-up-Verarbeitung relevante Forderung, wie Erbach zeigt. Ist die Grammatik dagegen 'semantisch nicht-monoton', wird der Scanning-Schritt aufwendiger. In Hinsicht auf die Effizienz-Probleme bei der Generierung mit bottom-up-Earley-Deduktion kommt Erbach zu folgender pessimistischer Schlußfolgerung: *"[...] bottom-up Earley deduction in its current form does not appear as an efficient algorithm for generation unless further improvements are made"* (1995a, S. 88).

4.5 Zusammenfassung

In diesem und dem vorhergehenden Kapitel wurden verschiedene Ansätze zu Parsing und Generierung vorgestellt und auf ihre Uniformität hin untersucht. Die Algorithmen von van Noord (1989, 1990a, 1990b, 1991, 1993 und 1997) sowie von Shieber *et al.* (1990) verwenden Backtracking zur Behandlung von Alternativen. Dies muß jedoch nicht zum Verlust von erzielten Teilanalysen führen, wenn Zwischenergebnisse gespeichert werden. Die Ansätze von Shieber (1988), Gerdemann (1991), Neumann (1994) sowie Erbach (1995a) verwalten Alternativen und Zwischenergebnisse in einer Tabelle. Wegen der flexiblen Kontrolle, die Earley-Deduktionsverfahren ermöglichen, und der klaren Trennung von Inferenzregeln und Programm bzw. Grammatik handelt es sich um einen sehr attraktiven Rahmen zur Modellierung von Parsing- und Generierungs-Algorithmen. Es bleibt aber letztlich eine Frage der empirischen Überprüfung, ob Chart-basierte Algorithmen ähnlich effizient sein können wie Algorithmen mit Backtracking.

Die Diskussion des Ansatzes von Erbach (1995a) hat gezeigt, daß reine bottom-up-Algorithmen bei der Generierung ineffizient sind. Sie machen es notwendig, vor Beginn der Analyse alle infrage kommenden Unit-Klauseln auszuwählen. Der Grund liegt darin, daß es kein Item gibt, das die Information der Input-Merkmalsstruktur 'aufbewahrt'. Dies Aufgabe erfüllt bei der top-down-Verarbeitung zunächst das Start-Item. Durch die Grammatikregeln wird diese Information über Vorhersagen 'nach unten' weitergereicht, bis schließlich ein Scanning-Schritt erfolgen kann. Er erfolgt genau zu dem Zeitpunkt, an dem er notwendig ist, aber nicht früher. Dagegen ist die bottom-up-Analyse darauf angewiesen, daß alle relevanten Unit-Klauseln zur Kombination zur Verfügung stehen. Das Problem der 'semantischen Monotonie' stellt sich deshalb alleine im Hinblick auf die reine bottom-up-Verarbeitung. Auch dann ist diese Beschränkung jedoch nicht zwingend notwendig – auch wenn sie das Scanning vereinfacht. Die Kopf-gesteuerten Ansätze von van Noord und Shieber *et al.* verwenden dagegen auf sehr geschickte Weise top-down-Information zur Steuerung der bottom-up-Verarbeitung. Es ist ein naheliegender Gedanke, auch die Generierung mit bottom-up-Earley-Deduktion am semantischen Kopf auszurichten.

Die Kopf-gesteuerte Generierung richtet sich optimal an der gegebenen Input-Semantik aus, unabhängig davon, ob sie bei bottom-up- oder top-down-Verfahren eingesetzt wird. Die Reihenfolge, in der auf die lexikalischen Einträge bei der Generierung mit top-down-Earley-Deduktion in den Ansätzen von Gerdemann und Neumann zugegriffen wird, entspricht der bei den bottom-up-Algorithmen von van Noord und Shieber *et al.* Vom semantischen Kopf-gesteuerte Algorithmen ermöglichen eine Dekomposition der semantischen Struktur durch die Grammatikregeln selbst. Dies ist völlig analog zu der Dekomposition der Input-Wortkette beim Parsing von links nach rechts. Bei Differenzlisten-Kodierung des Strings wird die weiterzugebende Restliste 'automatisch' berechnet. Die resultierende Baum-Traversierung ist bei Parsing und Generierung nicht gleich, wird aber in optimaler Weise der gegebenen Input-Struktur gerecht.

Ein top-down-Earley-Deduktionsverfahren mit Wahl des linken Elements beim Parsing und des semantischen Kopfes bei der Generierung ermöglicht eine exakte Anpassung an die gegebene Input-Struktur bei großer Uniformität. Die Entsprechung zur Kopf-gesteuerten bottom-up-Generierung ist im Hinblick auf Differenzlisten-Kodierung nicht das head-corner-Parsing, sondern das left-corner-Parsing. Eine uniforme Architektur für eine solche Kombination erfordert jedoch eine sehr breite Parametrisierung (Shieber et al. 1990, S. 41).

Der head-corner-Parser teilt die Wortkette in drei Teile (den Head-String und die Wortlisten links und rechts des Kopfes). Ein uniformer am syntaktischen Kopf ausgerichteter Algorithmus muß dies berücksichtigen (s.a. Abschnitt 6). Eine solche Architektur kann als uniformer als die bisher vorgestellten angesehen werden, weil auch die Baum-Traversierung für Parsing und Generierung gleich ist. Zudem ist sie sehr effizient, da lexikalische Subkategorisierungs-Information vor der Bearbeitung der Elemente, für die subkategorisiert wird, vorliegt. Der Nachteil von Kopf-gesteuerten Analysestrategien ist jedoch, daß sie nicht links-rechts-inkrementell arbeiten können, da sie auf die Existenz des Kopf-Elements angewiesen sind.

Abbildung 4.3 gibt einen Überblick über die wichtigsten Eigenschaften der vorgestellten Ansätze. Eine Beurteilung der Uniformität (letzte Zeile der Tabelle) ist insofern schwierig und subjektiv, weil die Kriterien, an denen sie gemessen werden soll, nicht eindeutig definiert sind (zu einem Vorschlag s. Kapitel 6). In Abschnitt 2.2.2 wurde eine Minimalanforderung für uniforme Algorithmen angegeben: die Relation zwischen phonologischer und semantischer Repräsentation soll in beide Richtungen in endlicher Zeit berechnet werden

	van Noord (1993)	**Shieber (1988)**	**Gerdemann (1991)**	**Neumann (1994)**	**Erbach (1995)**
Verwaltung von Alternativen	Backtracking	Chart	Chart	Chart	Chart
Kontrolle	nur Tiefensuche	Tiefensuche – Breitensuche	nur Breitensuche	Tiefensuche – Breitensuche	Tiefensuche – Breitensuche
Verarbeitungs-Richtung	bottom-up	top-down	top-down	top-down	bottom-up
Selektionsfunktion: **Parsing** **Generierung**	 syn. Kopf sem. Kopf	 linkes Element linkes Element	 linkes Element sem. Kopf	 linkes Element sem. Kopf	 — —
Indexing: **Parsing** **Generierung**	 nicht notwendig	 String-Position —	 String-Position String-Position	 String Semantik	 String-Position Semantik
Uniformität	$\odot$	$\oplus$	$\ominus$	$\oplus\oplus\oplus$	$\oplus\oplus$

Abbildung 4.3: Übersicht über die vorgestellten Ansätze

können. Dies sagt jedoch nichts über die Effizienz der beiden Verarbeitungs-Richtungen aus. Wenn die Effizienz nicht berücksichtigt wird, könnte beispielsweise der Ansatz von Shieber (1988) als der uniformste gelten, da sogar seine Selektionsfunktion einheitlich ist. Sie ist bei der Generierung jedoch nicht der Input-Struktur angepaßt, so daß sich eine starke Asymmetrie in bezug auf die Effizienz von Parsing und Generierung ergibt. Der gleiche Einwand gilt für den sehr uniformen Ansatz von Erbach (1995a), der keine Selektionsfunktion besitzt.

Bei der Einschätzung der Ansätze muß jedoch auch die zeitliche Reihenfolge ihrer Entstehung berücksichtigt werden: die späteren konnten von den vorhergehenden lernen. Insofern ist eine klare Entwickung bei den Ansätzen zur top-down-Earley-Deduktion zu erkennen. Es wird jedoch deutlich, daß erst der Ansatz von Neumann (1994) die Grundideen von Pereira und Warren (1983) konsequent umsetzt.

Kapitel 5

Implementierungsaspekte

In diesem Kapitel werden einige Aspekte der Implementierung von zwei der vorgestellten Ansätze in der Logik-basierten Programmiersprache Prolog erläutert. Es wird eine Implementierung des semantic-head-driven-Generation-Algorithmus *(SHDGA)* sowie des Uniformen Tabularen Algorithmus *(UTA)* ohne Indexing vorgestellt. Für beide Implementierungen wird die gleiche Beispielgrammatik verwendet. Anschließend wird die Integration beider Interpreter in eine gemeinsame Test-Umgebung erläutert. Zunächst sollen jedoch die Beispielgrammatik sowie einige Aspekte der Verwendung von Merkmalsstrukturen und DCGs behandelt werden.

Programm und Interpreter Der Standard-Prolog-Interpreter arbeitet top-down, von links nach rechts mit Tiefensuche und Backtracking. 'Normale' Prolog-Programme als eine Menge von Fakten und Regeln werden von diesem eingebauten Interpreter verarbeitet, so daß sich der Benutzer keine Gedanken über die Implementierung des Interpreters selbst machen muß. Die in den Abschnitten 5.3 und 5.4 erläuterten Parser und Generatoren können dagegen als Interpreter in Prolog bezeichnet werden. Sie implementieren explizit eine alternative Abarbeitung der Programm-Klauseln. Da sie selbst in der Syntax von Prolog ausgedrückt werden, gibt es also zwei Mengen von Definiten Klauseln: die des Programms bzw. der Grammatik und die des Interpreters.

5.1 Reversibilität von Prolog-Programmen

Eine wesentliche Anforderung an eine Implementierung im Hinblick auf das Thema dieser Arbeit ist die reversible Berechenbarkeit der Relation *String* $\leftrightarrow$ *Semantik*. Da Prolog als (weitgehend) deklarative Programmiersprache bezeichnet wird, bietet es sich an, die Reversibilität von Prolog-Programmen direkt auszunutzen, indem die Ziel-Klausel der jeweiligen Aufgabe entsprechend instantiiert wird (dies gilt zumindest für die als *pure-Prolog* bezeichnete Teilmenge von Prolog, nicht aber für metalogische Prädikate sowie bestimmte build-in-Prädikate, die Seiteneffekte erzeugen).

Die reversible Verwendung von Prolog-Programmen kann zu zwei unerwünschten Ergebnissen führen. Zum einen kann sich herausstellen, daß das Ziel mit einer bestimmten Instantiierung überhaupt nicht berechnet werden kann (z.B. indem die Beweisfindung nicht terminiert). Ein zweites unerwünschtes Ergebnis einer reversiblen Verwendung von Prolog-Programmen kann die mangelnde Effizienz in einer Verarbeitungs-Richtung sein, d.h. daß zwar in endlicher Zeit ein Ergebnis berechnet werden kann, der Effizienz-Unterschied bei den verschiedenen Instantiierungen der Ziel-Klausel jedoch groß ist. Grundsätzlich sollte gelten, daß Parsing und Generierung zumindest annähernd gleich effizient sind. Im folgenden werden zwei einfache Beispiele für diese beiden Probleme gegeben. Es soll jedoch nicht behauptet werden, daß dieses Thema damit erschöpfend behandelt ist. Es ist auffällig, daß Prolog in der Literatur kaum unter dem Blickwinkel der Reversibilität betrachtet wird – eine besonders interessante Frage in diesem Zusammenhang ist die nach dem grundlegenden Verhältnis von Deklarativität und Reversibilität.

Terminations-Probleme bei mangelnder Instantiierung Es gibt Notations-Konventionen für die Beschreibung von Prolog-Prädikaten im Hinblick auf die Verwendung der Argumente als *Input-* oder *Output-Argumente*. Beispielsweise soll ein Prädikat `pred(+Arg1, -Arg2)` mit instantiiertem `Arg1` als Input-Argument und nicht instantiiertem `Arg2` als Out-Argument aufgerufen werden. Wenn beides möglich ist, wird dies oft durch `?Arg3` gekennzeichnet (vgl. z.B. die *Mode-Spezifikationen* im SICStus v3 User's Manual (1995)).

Bei der reversiblen Verwendung eines zweistelligen Prädikats müssen beide Argumente als Input und als Output dienen können. Das folgende Beispiel zeigt in vereinfachter Form, daß die (prozedurale) Reihenfolge bei der Verarbeitung durch den Prolog-Interpreter eine entscheidende Rolle bei der Evaluierung spielen kann:

```
(43.1)   a(In,Out) :-   b(In,A), c(A,Out).

(43.2)   b(In,A)   :-   In == in, A = atom.
(43.3)   b(In,A)   :-   A == atom, In = in.
(43.4)   c(A,Out)  :-   A == atom, Out = out.
(43.5)   c(A,Out)  :-   Out == out, A = atom.
```

Das Prädikat `a/2` soll mit `a(in,Out)` aufgerufen werden können und die Variable `Out` mit dem Atom `out` instantiieren. Wenn es dagegen mit `a(In,out)` aufgerufen wird, soll es für `In` die Instantiierung `in` liefern. Für die Prädikate `b/2` und `c/2` soll gelten, daß, wenn eine Variable instantiiert ist, sie auch die andere instantiieren. Wenn beispielsweise `b(In,A)` mit `In = in` aufgerufen wird, ist der Output von `A` das Atom `atom` und umgekehrt. Wenn dagegen keine der Variablen instantiiert ist, scheitert das Prädikat. Das gleiche gilt für `c(A,Out)` mit den Instantiierungen `atom` für `A` und `out` für `Out`. Dies soll ein realistischeres (pure-Prolog-)Programm simulieren, bei dem die mangelnde Instantiierung beispielsweise einer Liste zu Endlosschleifen führt (`'=='` in (43.2) bis (43.5) ist ein metalogisches Prädikat).

Das Programm in (43) kann jedoch nur für `a(in,Out)` das gewünschte Ergebnis liefern. Der Grund liegt in der Reihenfolge der Evaluierung der Prädikate im Klauselrumpf: Bei dem Aufruf `a(In,out)` scheitert `b/2`, weil keines seiner Argumente instantiiert ist. Der entscheidende Punkt ist, daß in diesem Fall zuerst `c(A,out)` aufgerufen werden müßte, damit die Argumente in `b/2` ausreichend instantiiert sind.

Die Behandlung der wechselseitigen Abhängigkeit der Instantiierung von Argumenten, indem die verschiedenen Ziele aufeinander warten, wird als *Coroutining* bezeichnet (vgl. Johnson (1995)). Grundsätzlich handelt es sich um eine Frage der Selektions-Funktion, die für die Bestimmung des nächsten zu bearbeitenden negativen Literals verantwortlich ist. Es gibt eine Reihe von Prolog-Konstrukten, die die Bearbeitung eines Prädikats verzögern, bis die Argumente ausreichend instantiiert sind. SICStus Prolog in der Version 3 bietet einige build-in-Prädikate, die dies bewirken. Das Prädikat `when(+Condition,+Goal)` blockiert die Ausführung von `Goal` bis `Condition` erfüllt ist, wobei `Condition` Angaben über die Instantiierung der Argumente in `Goal` enthält. (43.1) läßt sich dann in folgender Weise schreiben:

(44)
```
a(In,Out) :-
          when((ground(In);ground(A)),b(In,A)),
          when((ground(A);ground(Out)),c(A,Out)).
```

Bei dem Aufruf `a(In,out)` wird die Ausführung von `b/2` solange blockiert, bis eines der beiden Argumente instantiiert ist. Auch die Verwendung von Coroutining-Methoden hat jedoch ihre Grenzen. So kann die Anfrage `a(In,Out)` in dem gegebenen Beispiel nach wie vor nicht evaluiert werden. Keine der Variablen ist instantiiert, so daß sowohl `b/2` als auch `c/2` immer blockiert werden. In bezug auf die VP-Komplement-Regel (13) war zudem in Abschnitt 2.3.3 bemerkt worden, daß eine Verzögerung der Bearbeitung des Komplements sinnlos ist, weil dies nur weiter in die 'Linksrekursions-Falle' führt. Dies hatte die Einführung von Restriktion oder Guides motiviert.

Parsing und Generierung mit DCGs Das Parsing mit DCGs und dem Standard-Prolog-Interpreter ist sehr effizient (allerdings muß das Linksrekursions-Problem in Kauf genommen werden). Abgesehen von der Effizienz, die sich aus der Implementierung eines solchen Interpreters ergibt (und die hier nicht behandelt werden soll), kann über die Differenzlisten-Kodierung sehr gezielt auf den nächsten lexikalischen Eintrag zugegriffen werden. Hier soll ein Beispiel für eine sehr simple erweiterte DCG gegeben werden. Die Übersetzung in Prolog-Klauseln (s. Abschnitt 4.1.1) erfolgt in den meisten Prolog-Implementierungen automatisch.

(45)
```
np(Quant,Cont) --> det(Quant,Gen), n(Quant,Gen,Cont).
```

```
det(sg,masc)   --> [der].
det(sg,fem)    --> [die].
det(sg,neut)   --> [das].
n(sg,neut,car) --> [auto].
```

Die DCG in (45) enthält eine Grammatikregel, die eine NP analysiert. Die Variable `Quant` steht für die Quantifizierung, die einfach durch den Numerus ausgedrückt werden soll; die Variablen `Gen` und `Cont` stehen für das Genus bzw. den semantischen Inhalt, der in dem Lexikoneintrag für das Nomen festgelegt ist.

Ein Trace für den Beweis des Ziels `np(Q,C,[das,auto],[])` zeigt, daß erst der Determiner analysiert wird und daraufhin mit instantiierten Kongruenz-Merkmalen auf den Lexikoneintrag für das Nomen zugegriffen wird. Dabei werden die Variablen `Q` und `C`, die gewissermaßen die Semantik der NP repräsentieren sollen, mit `sg` bzw. `car` instantiiert. Wenn dagegen eine Generierung durch das Ziel `np(sg,car,F,R)` stattfindet, werden alle Determiner tatsächlich gewählt, um daraufhin festzustellen, daß erst der letzte mit dem gesuchten Nomen kongruiert.

Der Prolog-Interpreter testet bei der Generierung alle Determiner durch, weil der gesuchte in diesem Beispiel textuell am weitesten unten steht. Dieses Verhalten kann bei größeren Grammatiken zu starker Ineffizienz führen. Natürlich kann es auch beim Parsing zu Ineffizienzen kommen, weil beispielsweise der Determiner *die* mehrere Lexikoneinträge haben kann. Lexikalische Ambiguitäten stellen generell ein Problem für jede Verarbeitungs-Richtung dar. Eine Generierung von links nach rechts steht hier jedoch vor besonders großen Problemen. Die Verarbeitung ist grundsätzlich dann am effizientesten, wenn sie mit dem Element beginnt, das die Menge der weiteren Möglichkeiten am stärksten einschränkt. Dies spricht, wie in Kapitel 3.2 erläutert, für eine Ausrichtung der Verarbeitung am syntaktischen bzw. semantischen Kopf.

5.2 Beispielgrammatik

Kodierung von Merkmalsstrukturen

Grundsätzlich gibt es zwei Arten von Datenstrukturen zur Kodierung sprachlicher Information, die hier relevant sind: Prolog-Terme und DAGs, wobei sich bei letzteren wiederum die Frage der Repräsentation in Prolog stellt. Mit der Entscheidung für eine der beiden Datenstrukturen verbunden ist die Frage nach der Unifikations-Operation, die zur Vereinigung der Information zweier kompatibler Strukturen dient: Term-Unifikation und DAG-Unifikation. In die Implementierung von Prolog als Programmiersprache ist nur erstere eingebaut, so daß die Verwendung von Term-Unifikation in jedem Fall effizienter ist als die Verwendung von DAG-Unifikation, die einen zusätzlichen Unifikations-Algorithmus mit dem entsprechenden Overhead benötigt (ein Vorschlag zur Implementierung von DAG-

Unifikation in Prolog findet sich in Gerdemann (1991)).[1] Auf der anderen Seite sind Merkmalsstrukturen für den Grammatikentwickler von Vorteil, weil sie leichter zu handhaben sind. Die folgenden Beispiele sollen die verschiedenen Möglichkeiten der Repräsentation der linguistischen Information über eine VP illustrieren:

(46.1)
```
vp(syn(Num,pers3,sc(np(Num,nom))),sem(Sem),A,B).
```

(46.2)
```
vp(syn(num:Num,pers:3,sc(np(num:Num,case:nom))),sem(Sem),A,B).
```

(46.3)
```
node([syn:[cat:vp,num:Num,pers:3,sc:[cat:np,num:Num,case:nom|_]|_],
                    sem:Sem,phon:[fi:A,re:B|_]|_]).
```

(46.4)
```
node([phon:[fi:A,re:B|_],sem:Sem,syn:[sc:[cat:np,num:Num,case:nom|_],
                    cat:vp,pers:3,num:Num|_]|_]).
```

(46.5)
```
node([phon:[fi:A,re:B],syn:[cat:vp,num:Num,pers:3,
                    sc:[cat:np,num:Num,case:nom]],sem:Sem]).
```

In (46.1) wird die Term-Repräsentation einer VP in der dritten Person, die für eine NP im Nominativ mit dem gleichen Numerus wie die VP subkategorisiert, widergegeben. An den letzten beiden Argumentstellen befinden sich String-Variablen, die als Ergebnis einer Übersetzung aus DCG-Notation eingefügt worden sind. Da Terme rekursiv eingebettet sein können, besteht die Möglichkeit, Baumstrukturen zu kodieren. Koindizierung kann durch Variable, die in Prolog immer lokal zur jeweiligen Definiten Klausel gelten, realisiert werden (im Beispiel wird die Variable `Num` zur Koindizierung verwendet). Die Unifikation zweier solcher Terme kann durch die effiziente Prolog-Term-Unifikation realisiert werden. Für den Benutzer haben Terme jedoch den Nachteil, daß die Merkmale durch ihre Argument-Position gekennzeichnet sind. Bei größeren Grammatik-Fragmenten sind diese nur sehr schwer zu überblicken. In (46.2) wird der Operator : (in SICStus Prolog beispielsweise vordefiniert durch `op(550,xfy,:)`) verwendet, um Merkmale und *atomare* Werte zu unterscheiden und aufeinander zu beziehen. Dies ändert jedoch nichts an der Struktur des komplexen Terms.

In (46.3) wird die linguistische Information einschließlich des phonologischen Strings in einer Merkmalsstruktur, die durch eine rekursiv eingebettete Prolog-Liste mit variabler Rest-Liste repräsentiert wird, kodiert (vgl. Gazdar und Mellish (1989, S.228ff und 273ff)). Da die Reihenfolge der Pfade in einer Merkmalsstruktur irrelevant ist, muß diese Prolog-Liste als Menge interpretiert werden. DAG-Unifikation von (46.3) und (46.4) ist deshalb erfolgreich und ergibt eine Merkmalsstruktur, die die gleiche Information enthält wie (46.3) oder (46.4). Dies macht schon deutlich, daß ein Grund für die Ineffizienz von DAG-Unifikation mit Prolog-Listen in dem rekursiven Durchsuchen der Listen liegt, das zum

[1] Diese Überlegung hat beispielsweise auch die Entwicklung von ProFIT (Erbach 1995b) motiviert. ProFIT dient zur Kompilation von Repräsentationen von getypten Merkmalsstrukturen in Prolog-Terme, so daß die effizientere Term-Unifikation von Prolog genutzt werden kann.

Auffinden der Merkmale notwendig ist. Die Merkmale sind nicht auf eine bestimmte Position festgelegt.

In (46.3) und (46.4) wird ein weiterer Unterschied zwischen Termen und (ungetypten) Merkmalsstrukturen deutlich: Während Terme eine feste Stelligkeit haben, sind DAGs offen für (kompatible) neue Information (*sideways openness* (Gazdar und Mellish 1989, S.229)). Dies wird durch die variablen Restlisten erreicht.

Wenn dagegen die Anzahl der Merkmale für jedes sprachliche Zeichen in einer getypten Grammatik feststeht, sind die offenen Restlisten nicht mehr zwingend notwendig. Deswegen werden sie in der Repräsentation in (46.5), die ein Beispiel für die Repräsentation der in dieser Arbeit verwendeten Beispielgrammatik sein soll, nicht mehr verwendet. Da keine neuen Merkmale mehr hinzugefügt werden können, handelt es sich um eine (sehr bescheidene) Form der Verwendung von Typen. Die Reihenfolge der Merkmale in (46.5) soll festgelegt sein, so daß die Prolog-Term-Unifikation verwendet werden kann. Für alle Formen der Unifikation gilt jedoch grundsätzlich, daß ein *occurs check* notwendig ist, um zyklische Strukturen wie z.B. als Ergebnis von `X = a(X)`), zu vermeiden. Aus Effizienzgründen wird er jedoch zumeist ausgelassen.

Grammatikregeln

Die Repräsentationen in (46.3) bis (46.5) verwenden das Prädikat `node/1` als 'Träger' der Merkmalsstruktur, d.h. die syntaktische Kategorie ist ein Merkmal unter anderen. Dies hat den Vorteil, daß auch unterspezifizierte Grammatik-Regeln ohne Angabe der syntaktischen Kategorie formuliert werden können. Dagegen muß beispielsweise in der DCG-Notation die syntaktische Kategorie bekannt sein, da Prolog-Terme keine Variablen als Prädikatnamen besitzen können. Grammatikregeln sollen die folgende allgemeine Form haben:

(47) `node(N,Fs1) <--- [node(Fs2), node(Fs3)].`

Diese Grammatikregel definiert einen binären lokalen Baum, wobei die Tocherknoten zum besseren Zugriff während der Verarbeitung in einer Liste angegeben werden. Der Infix-Operator `<---` wird genauso wie der vordefinierte Operator `-->` für DCGs mit `op(1200,xfx, <---)` definiert. Alle Grammatikregeln und Lexikoneinträge werden mit einer Nummer als erstem Argument des Prädikats `node/2` versehen. So können zur Compile-Zeit gewonnene Aussagen über einzelne Regeln in bezug auf die Regelnummer getroffen werden (s. Abschnitt 5.3). Zudem haben sich Regelnummern als günstig für die Fehlersuche erwiesen.

Die in dieser Arbeit verwendete sehr einfache Beispielgrammatik besteht aus folgenden Grammatikregeln:

(48.1) `%` $S \rightarrow\ NP,\ VP$

```
node(1,[phon:[fi:S,re:S0],syn:[s],sem:Sem])   <---
```

```
  [node([phon:[fi:S,re:S1],syn:SynNp,sem:SemNp]),
   node([phon:[fi:S1,re:S0],syn:[vp,sc:[[syn:SynNp,sem:SemNp]]],sem:Sem])].
```

(48.2) % $VP \rightarrow\ VP,\ XP$

```
node(2,[phon:[fi:S,re:S0],syn:[vp,sc:[Subj|Cs]],sem:Sem])   <---
  [node([phon:[fi:S,re:S1],syn:[vp,sc:[Subj,[syn:SynC,sem:SemC]|Cs]],sem:Sem]),
   node([phon:[fi:S1,re:S0],syn:SynC,sem:SemC])].
```

(48.3) % $VP \rightarrow\ V$

```
node(3,[phon:[fi:S,re:S0],syn:[vp,sc:Cs],sem:Sem])   <---
  [node([phon:[fi:S,re:S0],syn:[v,sc:Cs],sem:Sem])].
```

(48.4) % $NP \rightarrow\ Det,\ N$

```
node(4,[phon:[fi:S,re:S0],syn:[np,Gen,Num,Cas,Pers],sem:[cont:Ct,quant:Q]]) <---
  [node([phon:[fi:S,re:S1],syn:[det,Gen,Num,Cas],sem:[quant:Q]]),
   node([phon:[fi:S1,re:S0],syn:[n,Gen,Num,Cas,Pers],sem:[cont:Ct,quant:Q]])].
```

(48.5) % $NP \rightarrow\ PN$

```
node(5,[phon:[fi:S,re:S0],syn:[np,Gen,Num,Cas,Pers],sem:Sem])   <---
  [node([phon:[fi:S,re:S0],syn:[pn,Gen,Num,Cas,Pers],sem:Sem])].
```

Regel (48.1) mit der kontextfreien Basis $S \rightarrow\ NP\ VP$ legt fest, daß ein Satz aus einer (Subjekt-)NP als einzigem Element der Subkategorisierungs-Liste und einer VP besteht. Die semantischen und syntaktischen Merkmale des Verb-Komplements sind mit denen des subkategoriserten Elements koindiziert. Da das Subjekt immer an der ersten Stelle der Subkategorisierungs-Liste stehen soll, greift die VP-Komplement-Regel (48.2) auf das jeweils zweite Element der Liste zu und realisiert es im Gegensatz zur S-Regel nach dem Verb. Die VP-Regel (48.3) mit der kontextfreien Basis $VP \rightarrow\ V$ läßt die Subkategorisierungs-Liste unverändert.

Die Regeln (48.4) und (48.5) lizensieren NP-Konstituenten, wenn ein Determiner und ein Nomen bzw. ein Eigenname gegeben ist. Die Kongruenz-Merkmale werden dabei wiederum durch Koindizierung weitergegeben. Alle Grammatikregeln sind Chain-Rules: die Semantik der Kopf-Tochter ist mit der des Mutterknotens koindiziert.

Lexikoneinträge

Es werden ein-, zwei- und dreistellige Verben verwendet. Wie in größeren lexikaliserten Grammatiken soll möglichst viel Information im Lexikon kodiert werden (die Frage nach redundanzfreien Lexikon-Repräsentationen kann hier jedoch nicht behandelt werden). In (49) werden Lexikoneinträge für Verben der verschiedenen Stelligkeiten angegeben:

(49.1)

```
% traeumt
node(28,[phon:[fi:[traeumt|R],re:R],
```

```
              syn:[v,sc:[[syn:[np,_,sg,nom,3],sem:ArgSem]]],
              sem:[pred:dream,arg:ArgSem]]).
```

(49.2)
```
% liebst
node(33,[phon:[fi:[liebst|R],re:R],
         syn:[v,sc:[[syn:[np,_,sg,nom,2],sem:ArgSem1],
                     [syn:[np,_,_,acc,_],sem:ArgSem2]]],
         sem:[pred:love,arg1:ArgSem1,arg2:ArgSem2]]).
```

(49.3)
```
% schenken
node(40,[phon:[fi:[schenken|R],re:R],
         syn:[v,sc:[[syn:[np,_,pl,nom,3],sem:ArgSem1],
                     [syn:[np,_,_,dat,_],sem:ArgSem3],
                     [syn:[np,_,_,acc,_],sem:ArgSem2]]],
         sem:[pred:donate,arg1:ArgSem1,arg2:ArgSem2,arg3:ArgSem3]]).
```

Die Verben subkategorisieren jeweils für eine bestimmte Anzahl von NP-Komplementen, wobei das Subjekt als erstes Element der Liste durch Kasus, Numerus, Genus und Person festgelegt ist. Die weiteren Komplemente benötigen nur die Angabe des Kasus und der syntaktischen Kategorie, da keine Kongruenz mit dem Verb bestehen muß. Die Semantik der Komplemente ist koindiziert mit den entsprechenden Merkmalswerten in der Semantik des Verbs. Dies bedeutet bei einer Kopf-gesteuerten Generierung, daß die Information der gegebenen Input-Semantik bei Erreichen des Verbeintrags auf die Komplemente aufgeteilt wird.

Um Numerus-Kongruenz zwischen Subjekt und Verb zu erreichen, werden die Lexikoneinträge für Nomen und Eigennamen mit einem entsprechenden Merkmalsswert versehen (das Pronomen *du* soll wie ein Eigenname behandelt werden (50.4)). In der Grammatikregel (48.4) wird der Wert für den Numerus aus dem Nomen in die NP übernommen. Nomen und Eigennamen wird lexikalisch immer die dritte Person zugewiesen (50.2 und 50.3).

(50.1)
```
% der
node(6,[phon:[fi:[der|R],re:R],syn:[det,masc,sg,nom],sem:[quant:sg]]).
```

(50.2)
```
% mann
node(14,[phon:[fi:[mann|R],re:R],syn:[n,masc,sg,nom,3],sem:[cont:man,quant:sg]]).
```

(50.3)
```
% peter
node(24,[phon:[fi:[peter|R],re:R],syn:[pn,masc,sg,dat,3],
         sem:[cont:peter,quant:sg]]).
```

(50.4)
```
% du
node(27,[phon:[fi:[du|R],re:R],syn:[pn,_,sg,nom,2],sem:[cont:you,quant:sg]]).
```

Die Lexikoneinträge zeigen, daß der String als Differenz-Liste kodiert wird. Der von dem Lexikoneintrag abgedeckte bzw. 'verbrauchte' Teil steht am Beginn der Liste; die Restliste wird weitergegeben. Die hier erläuterte Beispielgrammatik wird im Anhang A.3 vollständig angegeben. Sie lizensiert Sätze wie *peter schläft*, *du träumst*, *der mann wirft den ball* und *die frau schenkt petra den ball.*

5.3 Semantic-head-driven-Generation

Die Implementierung des in Abschnitt 3.2.1 erläuterten vom semantischen Kopf gesteuerten Algorithmus orientiert sich an Shieber *et al.* (1990). Es zeigt sich, daß relativ wenige Erweiterungen vorzunehmen sind, um eine lauffähige Implementierung zu erreichen. Dabei soll es hier vor allem um Klarheit des Prolog-Codes gehen und weniger um möglichst große Effizienz.

Zunächst müssen einige Informationen über die Grammatik gewonnen werden (s. den folgenden Abschnitt). Dies kann grundsätzlich zur Compile-Zeit geschehen, auch wenn hier manuell vorgegangen wurde. Anschließend wird ein Kopf-gesteuerter Interpreter, der diese Information verwendet, vorgestellt.

Information über die Grammatik Es wird in der Implementierung davon ausgegangen, daß es nur unäre oder binäre Grammatikregeln gibt. Dies erleichtert die Bestimmung des noch zu generierenden Rests der rechten Seite von Chain Rules, die durch das Prädikat `chain_rule(Cat,N,Nhd)` beschrieben werden (die Variable `Cat` steht für die syntaktische Kategorie, `N` für die Regelnummer und `Nhd` gibt die Position der *non head daughter* an). Die S-Regel wird beispielsweise durch `chain_rule(s,1,left)` beschrieben: Die Regel mit der Nummer 1 hat eine linke Regelseite mit der syntaktischen Kategorie `s` und der 'Nicht-Kopf' steht links des semantischen Kopfes.

Eine Nonchain Rule wird durch `nonchain_rule(Cat,N,rule)` erfaßt. Das dritte Argument drückt aus, daß es sich um eine Grammatikregel (im Gegensatz zu Lexikoneinträgen) handelt. Für alle Lexikoneinträge wirde entsprechend ein Prolog-Fakt `nonchain_rule(Cat, N,lex)` angenommen. Die transitive Link-Relation wird durch `head_link/2` ausgedrückt, wobei das erste Argument die Kategorie des 'höheren' Knoten angibt. Die aus den in (48) erläuterten Grammtikregeln sowie den lexikalischen Einträgen (manuell) gewonnene Information sieht folgendermaßen aus:

(52)
```
chain_rule(s,1,left).
chain_rule(vp,2,right).
chain_rule(vp,3,unary).
chain_rule(np,4,left).
chain_rule(np,5,unary).

nonchain_rule(det,6,lex).
```

```
nonchain_rule(det,7,lex).
nonchain_rule(det,8,lex).
... etc. ...

head_link(s,vp).
head_link(vp,v).
head_link(s,v).
head_link(np,pn).
head_link(np,n).
head_link(X,X).
```

Die letzte Angabe von `head_link/2` besagt, daß jede Kategorie auch einen Link zu sich selbst hat, d.h. daß beispielsweise die Kategorie `np` von `np` aus erreichbar ist. Dies ist dann von Bedeutung, wenn die Zielkategorie mit der Kategorie der linken Seite einer Regel identisch ist, also unmittelbar vor der letzten bottom-up-Verknüpfung.

Interpreter Das Prädikat `head_proc/1` in (53) sucht eine Nonchain Rule, die einen Link zur Zielkategorie aufweist (zur ausführlichen Beschreibung des Algorithmus selbst s. Abschnitt 3.2.1). Es wird zunächst eine syntaktische Kategorie bestimmt, die einen Link zur Zielkategorie aufweist (53.2). Nun kann die Nummer einer Nonchain Rule der gesuchten Kategorie bestimmt werden ((53.3); diese Vorgehensweise erscheint effizienter als umgekehrt alle Nonchain Rules durchzugehen und jeweils zu überprüfen, ob ein Link gegeben ist). Erst dann wird auf die Regel selbst zugegriffen (53.4). Anschließend wird die rechte Regelseite bearbeitet (53.5) und eine bottom-up-Verknüpfung zur Zielmerkmalsstruktur versucht (53.6).

```
(53)     % head_proc(+Root)
(53.1)   head_proc([Phon,syn:[Cat|Syn],Sem]) :-
(53.2)       head_link(Cat,Cat1),
(53.3)       nonchain_rule(Cat1,N,RuleType),
(53.4)       instantiate_nonchain_rule(RuleType,N,[PivPhon,PivSyn,Sem],RHS),
(53.5)       head_proc_rhs(RHS),
(53.6)       connect_up([PivPhon,PivSyn,Sem],[Phon,syn:[Cat|Syn],Sem]).
```

Die Variable `RuleType` kann entweder mit `lex` oder `rule` instantiiert sein. Sie erlaubt einen effizienten Zugriff auf eine der beiden Klauseln von `instantiate_nonchain_rule/4` durch das Indexing auf dem ersten Argument der meisten Prolog-Systeme. Die rechte Seite (`RHS`) von lexikalischen Einträgen ist leer (letztes Argument von `instantiate_nonchain_rule/4` in (54)). Die erste Klausel definiert den Zugriff auf lexikalische Einträge; die zweite Klausel sucht die jeweilige Grammatikregel anhand ihrer Nummer.

```
(54)      % instantiate_nonchain_rule(+RuleType,+RuleNumber,-Pivot,-RHS)
(54.1)    instantiate_nonchain_rule(lex,N,[PivPhon,PivSyn,Sem],[]) :-
(54.2)         node(N,[PivPhon,PivSyn,Sem]).

(54.3)    instantiate_nonchain_rule(rule,N,[PivPhon,PivSyn,Sem],RHS) :-
(54.4)         (node(N,[PivPhon,PivSyn,Sem]) <--- RHS).
```

Damit tatsächlich eine Nonchain Rule *mit passender Semantik* gefunden wird, muß die Zielsemantik mit der der Nonchain Rule unifizieren. Deshalb wird die Semantik `Sem` des Pivots in (53) mit der des Ziels koindiziert. Die Mode-Beschreibung `-Pivot` in (54) muß deshalb genauer mit `[-PivPhon,-PivSyn,+Sem]` angegeben werden.

Die Abarbeitung der rechten Regelseite kann durch rekursive Bearbeitung der Elemente der Liste erfolgen:

```
(55)      % head_proc_rhs(+RHS).
(55.1)    head_proc_rhs([]).

(55.2)    head_proc_rhs([node(F)|R]) :-
(55.3)         head_proc(F),
(55.4)         head_proc_rhs(R).
```

Während in `head_proc/1` top-down eine Nonchain Rule gesucht und instantiiert wird, sucht `connect_up/2` Chain Rules, um diese Nonchain Rule bottom-up mit dem Ziel zu verbinden. Es werden bottom-up-Schritte durchgeführt, bis Pivot und Ziel unifizieren (56.1). Solange dies nicht der Fall ist, wird rekursiv eine Chain Rule mit einem Link gesucht ((56.3) und (56.4)) und `connect_up/2` aufgerufen (56.7).

```
(56)      % connect_up(+Pivot,+Root).
(56.1)    connect_up(Pivot,Pivot).
(56.2)    connect_up([Phon1,Syn1,Sem],[Phon,syn:[Cat|Syn],Sem]) :-
(56.3)         head_link(Cat,Cat1),
(56.4)         chain_rule(Cat1,N,RuleType),
(56.5)         instantiate_chain_rule(RuleType,N,[Phon1,Syn1,Sem],NewPivot,RedRHS),
(56.6)         head_proc_rhs(RedRHS),
(56.7)         connect_up(NewPivot,[Phon,syn:[Cat|Syn],Sem]).
```

Das einzige noch nicht definierte Prädikat ist `instantiate_chain_rule/5`. Es hat als erstes Argument eine Angabe über die Position der *non head daughter*, die eine eindeutige Unterscheidung der drei Klauseln erlaubt. Generell liegt der Vorteil einer solchen Festlegung des Regeltyps darin, daß Information, die leicht zur Compile-Zeit gewonnen werden kann, nicht zur Laufzeit berechnet werden muß. In diesem Fall betrifft dies Information über die Position des noch nicht bearbeiteten Elements.

Der aktuelle Pivot muß jeweils mit dem semantischen Kopf unifizieren (`Pivot` in (57)). Bei unären Regeln ist die restliche RHS leer (57.1). Bei binären Regeln wird die Variable `RedRHS` durch das noch nicht bearbeitete Element instantiiert.

```
(57)    % instantiate_chain_rule(+RuleType,+RuleNumber,+Pivot,-NewPivot,-RedRHS)
(57.1)  instantiate_chain_rule(unary,N,Pivot,NewPivot,[]) :-
(57.2)        (node(N,NewPivot) <--- [node(Pivot)]).

(57.3)  instantiate_chain_rule(left,N,Pivot,NewPivot,[node(Left)]) :-
(57.4)        (node(N,NewPivot) <--- [node(Left),node(Pivot)]).

(57.5)  instantiate_chain_rule(right,N,Pivot,NewPivot,[node(Right)]) :-
(57.6)        (node(N,NewPivot) <--- [node(Pivot),node(Right)]).
```

Top-down-Generierung

In der oben angegebenen Grammatik wird die NP-Regel (48.4) als Chain Rule definiert. Dies führt zu einer bottom-up-Generierung der NP beginnend mit dem Nomen. Wenn dagegen die NP-Regel als Nonchain Rule geschrieben wird, beginnt die Generierung top-down mit dem Determiner. Die NP-Regel hat dann die folgende leicht veränderte Form (die semantische Repräsentation des Nomens enthält keine Angabe der Quantifizierung mehr):

(58.1)

```
node([phon:[fi:S,re:S0],syn:[np,Gen,Num,Cas,Pers],sem:[cont:Ct,quant:Q]]) <---
  [node([phon:[fi:S,re:S1],syn:[det,Gen,Num,Cas],sem:[quant:Q]]),
   node([phon:[fi:S1,re:S0],syn:[n,Gen,Num,Cas,Pers],sem:[cont:Ct]])].
```

Dazu wird die Angabe `nonchain_rule(np,4,rule)` benötigt. Es können grundsätzlich fast alle Grammatikregeln auch als Nonchain Rules verwendet werden. Dies führt zu einer sehr ineffizienten Generierung von links nach rechts. Nur die VP-Komplement-Regel muß wegen der top-down-Vorhersage-Schleife als Chain Rule bottom-up bearbeitet werden. Es werden die folgenden Prolog-Fakten benötigt, um eine maximale top-down-Generierung zu erreichen:

```
(59)  head_link(X,X).
      chain_rule(vp,2,right).
      nonchain_rule(s,1,rule).
      nonchain_rule(vp,3,rule).
      nonchain_rule(np,4,rule).
      nonchain_rule(np,5,rule).
```

Die im Anhang angegebene Beispielgrammatik definiert alle Grammatikregeln als Chain Rules. Um eine top-down-Generierung zu erreichen, genügt es, die Links und Regel-Deklarationen zu vertauschen. Wenn das Prädikat `head_link/2` für zwei unterschiedliche Kategorie-Symbole scheitert, müssen die Chain Rules als Nonchain Rules behandelt werden.

5.3.1 Unerwünschtes Backtracking

Auch wenn die ganze Grammatik aus Chain Rules besteht, kann es zu Ineffizienzen in Form von unerwünschtem Backtracking kommen. Bei der Generierung des Satzes *du siehst den ball* aus der Input-Merkmalsstruktur `[Phon,syn:[s],sem:[pred:see,arg1:[cont:you,quant:sg],arg2:[cont:ball,quant:sg]]]` wird – bedingt durch Anordnung der Lexikoneinträge als Prolog-Fakten – zunächst das Verb *sieht* ausgewählt. Daraufhin wird die VP *sieht den ball* generiert. Da das Verb keine Kongruenz zu dem geforderten Subjekt aufweist, findet Backtracking statt. Dadurch wird die Information über die erfolgreich generierte NP *den ball* wieder verloren. Da die Input-Semantik für eine solche Grammatik keine deterministische Auswahl des Verbs erlaubt, kann Backtracking nicht grundsätzlich vermieden werden. Damit aber keine schon generierte Phrase ein zweites Mal erzeugt wird, sind Techniken des *selektiven Memoings* notwendig. Diese werden im übernächsten Abschnitt an einer einfachen Implementierung erläutert.

Syntaktische Information in der Link-Relation

Es gibt jedoch eine zweite Quelle für unerwünschtes Backtracking. Bei der Generierung der NP *den ball* mit einer Chain Rule wird zunächst ein Nomen mit der Semantik *ball* gesucht. Auch in diesem Fall wird aufgrund der Reihenfolge der Lexikoneinträge zunächst eine falsche Wahl getroffen, nämlich *ball* im Nominativ. Im Gegensatz zur Auswahl des Verbs ist dieser Indeterminismus jedoch vermeidbar: die syntaktische Information über den Kasus ist aus der Subkategorisierungs-Liste des Verbs schon bekannt. Sie wird jedoch nicht bei der top-down-Instantiierung weitergegeben.

Wie Kilbury (1996) zeigt, kann die Link-Relation zum Informations-Transport verwendet werden. In der bisher vorgestellten Implementierung findet die Weitergabe der semantischen Information jedoch nicht in der Link-Relation statt. Es sind deshalb einige kleinere Änderungen notwendig, die, wie sich zeigen wird, zu einer Vereinfachung des Prolog-Codes führen, weil nicht mehr auf einzelne Teile der Merkmalsstrukturen zugegriffen werden muß. Die Angabe der Link-Relation sowie der weiteren Information über die Grammatikregeln hat bei bezug auf die ganze Merkmalsstruktur einer Konstituente folgende Form (wobei Merkmalsstrukturen weiterhin die Struktur `[Phon,Syn,Sem]` besitzen):

(60)
```
chain_rule([_,syn:[s|_],_],1,left).
chain_rule([_,syn:[vp|_],_],2,right).
chain_rule([_,syn:[vp|_],_],3,unary).
```

```
chain_rule([_,syn:[np|_],_],4,left).
chain_rule([_,syn:[np|_],_],5,unary).

head_link([_,syn:[s|_],Sem],[_,syn:[vp|_],Sem]).
head_link([_,syn:[s|_],Sem],[_,syn:[v|_],Sem]).
head_link([_,syn:[vp|_],Sem],[_,syn:[v|_],Sem]).
head_link([_,syn:[np|_],Sem],[_,syn:[pn|_],Sem]).
head_link([_,syn:[np|R],Sem],[_,syn:[n|R],Sem]).
head_link(X,X).
```

Durch die koindizierte Variable `Sem` in `head_link/2` wird die Semantik weitergegeben. Der vorletzte Link für die Chain Rule $NP \rightarrow\ Det\ N$ gibt zusätzlich durch die Variable `R` die gesamte syntaktische Information weiter. Dies löst das Problem der ineffizienten Auswahl von Verbkomplementen: Es wird sofort der Lexikoneintrag mit der gesuchten Semantik *und* den gesuchten syntatischen Merkmalen gefunden. Um dieses Link-Prädikat verwenden zu können, müssen folgende Änderungen an dem Prolog-Code vorgenommen werden:

```
(61.1)  head_proc(Root) :-
(61.2)          head_link(Root,Pivot),
(61.3)          nonchain_rule(Pivot,N,RuleType),
(61.4)          instantiate_nonchain_rule(RuleType,N,Pivot,RHS),
(61.5)          head_proc_rhs(RHS),
(61.6)          connect_up(Pivot,Root).
```

Ein Vergleich zwischen (53) und (61) zeigt die nun möglich gewordenen Vereinfachungen: die interne Struktur der Merkmalsstruktur muß nicht mehr berücksichtigt werden. Die Konstituenten werden mit einer einzigen Variablen erfaßt (`Root` und `Pivot` in (61)). Auch die Definition von `instantiate_chain_rule/4` und `connect_up/2` kann nun vereinfacht werden:

```
(62.1)  instantiate_nonchain_rule(lex,N,Pivot,[]) :-
(62.2)          node(N,Pivot).
(62.3)  instantiate_nonchain_rule(rule,N,Pivot,RHS) :-
(62.4)          (node(N,Pivot) <--- RHS).

(63.1)  connect_up(Pivot,Pivot).
(63.2)  connect_up(Pivot,Root) :-
(63.3)          head_link(Root,Pivot),
(63.4)          chain_rule(NewPivot,N,RuleType),
(63.5)          instantiate_chain_rule(RuleType,N,Pivot,NewPivot,RedRHS),
(63.6)          head_proc_rhs(RedRHS),
(63.7)          connect_up(NewPivot,Root).
```

Die Definitionen von `instantiate_chain_rule/5` und `head_proc_rhs/1` müssen nicht geändert werden. Der Vorteil des Verwendung der Link-Relation zur Informations-Weitergabe liegt unter anderem in der Flexibilität, die sich daraus ergibt. Wenn beispielsweise in `head_link/2` die Variable `Sem` durch die anonyme Prolog-Variable '`_`' ersetzt wird, findet keine top-down-Instantiierung der Semantik statt, so daß ein *generate and test*-Verhalten entsteht.

Die Weitergabe von Information muß grundsätzlich jedoch vorsichtig eingesetzt werden. Wenn die in (59) für eine maximale top-down-Generierung definierten Links auf Merkmalsstrukturen wie in (60) erweitert werden, besteht die Gefahr, daß durch `head_link(X,X)` die einelementige Subkategorisierungs-Liste der VP in der S-Regel bis zum Verbeintrag weitergegeben wird. Wenn allein die VP-Komplement-Regel eine Chain Rule ist, dann ist die einzige nach der Generierung des Subjekts infrage kommende Nonchain Rule mit einem Link zu einer VP die Regel $VP \rightarrow V$. Bei Weitergabe der Subkategorisierungs-Liste in der Link-Relation ist aber kein Zugriff auf den Eintrag für ein mehrstelliges Verb möglich. Es ist also eine Form von Restriktion notwendig, die den Wert für die Subkategorisierungs-Liste auf eine anonyme Variable beschränkt:

```
(64.1) head_link([_,syn:[vp|_],Sem],[_,syn:[vp|_],Sem]).
(64.2) head_link(X,X) :-   \+ X = [_,syn:[vp|_],_].
```

Der Link in (64.1) ermöglicht nur die Weitergabe der Semantik. Der Zugriff auf Lexikoneinträge für mehrstellige Verben ist nun möglich, weil er nicht mehr durch die instantiierte einelementige Subkategorisierungs-Liste verhindert wird. Wenn dieser beschränkte Link eingesetzt wird, ergibt sich eine Suchstrategie, die der in Abschnitt 4.2.3 beschriebenen Analyse mit top-down-Earley-Deduktion sehr ähnlich ist: erst nach dem Zugriff auf das Verb durch die Regel $VP \rightarrow V$ wird die VP-Komplement-Regel bottom-up verwendet, bis eine Verbindung zur VP in der S-Regel möglich ist.

Um identische durch Backtracking erzeugte Lösungen für Sätze mit einstelligen Verben zu vermeiden, muß bei der hier vorgeschlagenen Veränderung der Links verhindert werden, daß zwei Definitionen von `head_link/2` für dieselbe Regelanwendung verwendet werden können (64.2). In diesem Fall bestrifft dies den reflexiven Link der VP, der schon durch (64.1) abgedeckt ist.

Selektives Memoing

Im vorhergehenden Abschnitt wurde gezeigt, daß durch die Verbesserung des Informations-Transports in der Link-Relation zielgerichteter auf das gesuchte Nomen zugegriffen werden kann. Dies löst aber noch nicht das Problem, daß bereits generierte Phrasen durch Backtracking wieder 'verloren' werden. Van Noord (1997) beschreibt eine Prolog-Implementierung, die bestimmte Ziele und Ergebnisse speichert, aber die allgemeine Backtracking-Suchstrategie nicht ändert (im Gegensatz zu der im Anschluß erläuterten Implementierung von Earley-Deduktion). Die wesentliche Motivation für *selektives* Memoing ist, daß die Kosten, eine

Tabelle für *alle* Beweisziele zu verwalten, so hoch sein können, daß sie die Vorteile des Memoings wieder aufheben. Deswegen werden nur dem Prädikat `head_proc/1` entsprechende Ziele gespeichert. Es handelt sich also um die Aufrufe für das Start-Ziel sowie für alle Nonhead-Elemente von rechten Regelseiten. Die Implementierung in van Noord (1997) bezieht sich auf den head-corner-Parser. Sie kann jedoch ohne Probleme auf die vom semantischen Kopf gesteuerte Generierung angewendet werden. Der Unterschied der hier angegebenen Implementierung besteht in einigen Vereinfachungen und der Vermeidung von `if-then-else`-Konstrukten (`->`).

Es werden zwei verschiedene Ziele gespeichert: das Prolog-Fakt `memo_goal/1` wird für jeden Aufruf von `head_proc/1` in der Datenbasis abgelegt. Darüberhinaus wird `memo_result/1` für jede erfolgreiche Lösung gespeichert. Der Unterschied zwischen den beiden Prädikaten liegt darin, daß im ersten Fall noch keine weitere Instantiierung durch die Evaluierung stattgefunden hat. Ein fehlgeschlagenes Ziel hat also einen Eintrag als `memo_goal/1`, ohne daß es zu einem daraus folgenden `memo_result/1` gekommen ist (der Zusammenhang zwischen beiden wird jedoch nicht repräsentiert). Das Prädikat `head_proc/1` wird nun wie folgt definiert (alle anderen Prädikate bleiben unverändert).[2]

```
(65.1)  head_proc(Root) :-   memo_goal(Goal), subsumes_chk(Goal,Root), !,
(65.2)         memo_result(Root).
(65.3)  head_proc(Root) :-
(65.4)         head_link(Root,Pivot),
(65.5)         nonchain_rule(Pivot,N,RuleType),
(65.6)         instantiate_nonchain_rule(RuleType,N,Pivot,RHS),
(65.7)         head_proc_rhs(RHS),
(65.8)         connect_up(Pivot,Root),
(65.9)         assertz(memo_result(Root)),
(65.10)        fail.
(65.11) head_proc(Root) :-   assertz(memo_goal(Root)), fail.
(65.12) head_proc(Root) :-   memo_result(Root).
```

Wenn ein Ziel schon einmal bearbeitet wurde ((65.1); zum Subsumptions-Test s.u.), müssen seine Lösungen in der Tabelle stehen, falls es welche gegeben hat (65.2). Durch den (roten) Cut kann es kein Backtracking zu einer anderen Klausel von `head_proc/1` geben, wohl aber zu den verschiedenen Lösungen von `memo_result/1`. Die zweite Klausel führt wie in (61) die Kopf-gesteuerte Generierung durch. Da, wie in Abschnitt 4.2.1 erläutert, beim Memoing die Tabelle vollständig sein muß, werden durch ein Fail-Loop alle Lösungen gesucht. Bevor durch `fail` (65.10) die Variablen-Bindungen wieder aufgehoben werden, wird das jeweilige Ergebnis in der Datenbasis gespeichert (65.9). Wenn alle Lösungen gefunden sind, wird das ursprüngliche Ziel gespeichert (65.11). Auch dies

[2] In der in Anhang A.1.2 aufgeführten Datei `memo_head.pl` werden einige Prädikatsnamen so verändert, daß der Kopf-gesteuerte Interpreter mit Backtracking sowie der mit selektivem Memoing gleichzeitig verwendet werden können.

ist wiederum deswegen möglich, weil das Fail-Loop die Instantiierungen durch die schon gefundenen Lösungen aufgehoben hat. Schließlich muß eine der abgelegten Lösungen aus der Tabelle geholt werden (65.12).

Bei einer Generierung des Satzes *du siehst den ball* wird wie beschrieben zunächst die VP *sieht den ball* erzeugt, wobei aber die NP in der Tabelle abgelegt wird. Da eine bottom-up-Verbindung an die S-Regel mit `connect_up/2` scheitert, erfolgt wie bisher Backtracking. Für das Verb *siehst* wird nun wiederum eine NP im Akkusativ gesucht. Sie kann durch (65.1) sofort aus der Tabelle entnommen werden. Duch die Reihenfolge von `memo_goal/1` und `memo_result/1` in der Datenbasis können die Verarbeitungs-Schritte in bezug auf `head_proc/1` nachvollzogen werden (`assertz/1` fügt den neuesten Eintrag immer an das Ende der Datenbasis ein).

Für jedes neue Ziel wird in (65.1) getestet, ob es ein subsumierendes Ziel in der Goal-Tabelle gibt. In diesem Fall müssen sich mögliche Lösungen in der zweiten Tabelle befinden, da sie von diesem generelleren Ziel mit erfaßt werden. In van Noord (1997) wird daüberhinaus ein neues Resultat nur dann in die Lösungs-Tabelle aufgenommen, wenn kein subsumierendes vorhanden ist (dies betrifft also den Vergleich der Resultate untereinander; es ist klar, daß die Ziele immer genereller sind als die korrespondierenden Lösungen). Wenn spezifischere Resultate bei dem Einfügen eines generelleren gelöscht werden, enthält die Lösungs-Tabelle keinerlei Redundanzen und ist somit *minimal*. Dann kann allerdings nicht mehr festgestellt werden, ob der Interpreter Lösungen mehrfach geliefert hat.

Das Prädikat `subsumes_chk(+General,+Specific)` (diese Mode-Deklaration soll nicht ausschließen, daß auch Variablen getestet werden) wird in Quintus Prolog ab Version 2.5 als built-in-Prädikat zur Verfügung gestellt. In SICStus Prolog muß die entsprechende Library geladen werden. Der Grundgedanke einer Implementierung des Subsumptions-Checks ist, daß eine Unifikation von `General` und `Specific` möglich sein muß, wenn alle Variablen in `Specific` gebunden sind (durch `numbervars/3`) – in diesem Fall ist `General` tatsächlich genereller. Damit keine Instantiierungen in `General` und `Specific` stattfinden, muß doppelte Negation verwendet werden (vgl. Pereira und Shieber (1987, S.193)):

```
(66.1)   subsumes_chk(General,Specific) :-
(66.2)          \+ \+ ( numbervars(Specific,0,_),
(66.3)                  General = Specific ).
```

Wie Pereira und Shieber anmerken (1987, S. 192), dürfen bei dieser Implementierung Variablennamen, die in `Specific` verwendet werden, nicht in `General` vorkommen, da ansonsten auch in `General` Variablen gebunden werden. Eine Umbenennung von Variablen mit dem Ziel, zwei disjunkte Mengen von Variablennamen zu erhalten, macht den Subsumptions-Test jedoch wesentlich aufwendiger. Aus diesem Grund wird hier von den built-in-Prädikaten ausgegangen.

5.4 Top-down-Earley-Deduktion

Die Implementierung eines auf top-down-Earley-Deduktion beruhenden Algorithmus orientiert sich an dem Pseudo-Code des Uniformen Tabularen Algorithmus *(UTA)* in Neumann (1994) (s.a. Abschnitt 4.3.3). Da bei Algorithmen, die Alternativen in einer Tabelle verwalten, kein Backtracking notwendig ist, wird diese grundlegende Eigenschaft von Prolog nicht zwingend benötigt. So wurde der Algorithmus von Neumann in LISP implementiert, während hier eine Prolog-Implementierung vorgestellt wird. Auch wenn die Backtracking-Fähigkeit nur wenig genutzt wird, hat Prolog gegenüber LISP jedoch den Vorteil, daß eine Unifikations-Operation zur Verfügung steht und Definite Klauseln als Datenstrukturen vorhanden sind. Demgegenüber müssen Definite Klauseln in LISP als Listen kodiert werden (vgl. Neumann (1994, S. 120ff)). Der im folgenden dargestellte Prolog-Code hat im Gegensatz zu dem Kopf-gesteuerten Interpreter mit Backtracking eine sehr ‘prozedurale Lesart’.

Datenstruktur Die Implementierung realisiert eine einfache Version des Interpreters in Neumann (1994). Sie verwendet kein Indexing mithilfe des *essentiellen Merkmals*. Es wird jedoch das Indexing auf dem ersten Argument der Prolog-Systeme für das Durchsuchen der Tabelle ausgenutzt (in SICStus Prolog findet Indexing auf dem Funktor des ersten Arguments im Klauselkopf statt; vgl. SICStus v3 User’s Manual (1995, S.178)). Zunächst muß eine Datenstruktur, die das Indexing ausnutzt, für die Items der Tabelle gewählt werden. Da die gleiche Grammatik wie zuvor benutzt wird, stehen zudem Regelnummern zur Verfügung. Sie ermöglichen eine Repäsentation der Kombinationen von Grammatikregeln und Lexikoneinträgen, aus denen ein Item hervorgegangen ist (ein Beispiel wird weiter unten gegeben).

(67)
```
node(N,Head) <--- [Left,Right].   ⇒   table_item(Right,node(Head),[Left],N).
node(N,Head) <--- [Left].         ⇒   table_item(Left,node(Head),[],N).
node(N,Head).                     ⇒   table_item(nil,node(Head),[],N).
```

Die erste Zeile in (67) zeigt die ‘Übersetzung’ einer binären Grammatikregel in ein Item der Tabelle (als Ergebnis einer Vorhersage). Das (fettgedruckte) gewählte Element `Right` steht an der ersten Argumentposition von `table_item/4`. Durch das Indexing soll ein möglichst schneller Zugriff auf das gewählte Element möglich sein, da dieses das einzige ist, das jeweils reduziert werden kann. An der zweiten Argumentposition steht der Klauselkopf. Darauf folgt der *Rest* der rechten Regelseite, d.h. das gewählte Element wird nicht zweimal repräsentiert (zum einen als gewähltes Element und zum anderen als Element des Klauselrumpfs). Dies ist insofern vorteilhaft, als in der Selektionsfunktion schon eine Reduktion vorweggenommen werden kann (s.u.) – die Untersuchung des Klauselrumpfs muß dann nicht zweimal erfolgen. An der vierten Argumentposition wird die Information über die Regelnummer angegeben (tatsächlich wird an dieser Position ein stärker strukturierter Term weitergegeben; s.u.).

In der zweiten Zeile wird eine unäre Grammatikregel in ein Item der Tabelle verwandelt. Da es nur ein Element auf der rechten Regelseite gibt, muß dieses gewählt werden. Der verbleibende Rest der rechten Regelseite ist die leere Liste. Ein Lexikoneintrag (dritte Zeile) hat kein gewähltes Element. Unit-Klauseln sind sofort an dem Prolog-Atom `nil` auf der ersten Argumentposition von `table_item/4` zu erkennen.

(67) zeigt die Datenstruktur für Items der Tabelle. Agenda-Items sind in der gleichen Weise strukturiert, nur daß alle Argumente durch runde Klammern zusammengefaßt werden, so daß das Prädikat `agenda_item` einstellig ist. Dies ermöglicht die Weitergabe des Items mit nur einer Variable. Ein Indexing in der Agenda ist nicht nötig, weil immer das erste Item als nächstes bearbeitet wird.

Hauptaufruf In (68) wird der Hauptaufruf für den Interpreter wiedergegeben. Das Prädikat `earley_proc/5` wird mit den folgenden Argumenten aufgerufen: die unterspezifizierte Input-Merkmalsstruktur `Fs` wird durch die Verarbeitung vervollständigt; die Variable `Search` kann mit `first` oder `all` instantiiert sein, je nachdem, ob eine erschöpfende Lösungssuche erwünscht ist oder das erste Ergebnis ausreicht; `Control` wird mit `bf` für Breitensuche oder `df` für Tiefensuche instantiiert. Die beiden Variablen `Trace` und `Display` (jeweils instantiiert mit `on` oder `off`) stehen für die Tracing-Option und verschiedene Möglichkeiten der Lösungs-Ausgabe. Ein eigener Interpreter-Trace hat sich als sehr nützlich für die Fehlersuche gerade bei einem tabularen Algorithmus erwiesen. Eine Beispiel-Ausgabe wird in Anhang A.4.4 gegeben; Details der Implementierung des Trace sollen hier jedoch nicht erläutert werden.

```
(68)     % earley_proc(+Fs,+Search,+Control,+Trace,+Display)
(68.1)   earley_proc(Fs,Search,Control,Trace,Display) :-
(68.2)           clear_tables,
(68.3)           set_switch(trace,Trace),
(68.4)           init_agenda(Fs),
(68.5)           earley_proc(Search,Fs,Control), !,
(68.6)           display_solutions(Display,Search,Fs).
```

Zunächst wird die Tabelle aufgeräumt und der Trace eingestellt ((68.2) und (68.3)). Nach der Initialisierung der Agenda (68.4) wird die eigentliche Verarbeitung begonnen (68.5). Da immer alle Alternativen in die Tabelle aufgenommen werden, kann nun ein Cut verwendet werden. Darauf findet in (68.6) Backtracking zu den verschiedenen Lösungen statt, wobei die Input-Merkmalsstruktur `Fs` jeweils neu vervollständigt wird. Die einzigen beiden interessanten Prädikate sind also `init_agenda/1` und `earley_proc/3`.

```
(69)     init_agenda(Fs) :-   assertz(agenda_item((node(Fs),ans(Fs),[],(_,0)))).
```

Die Initialisierung der Agenda erfolgt durch das Einfügen eines aktiven Items, um eine Vorhersage auszulösen und so die Verarbeitung ‘anzustoßen’ (das Agenda-Item entspricht der Klausel `ans(Fs) <--- node(Fs)`).

```
(70.1)  earley_proc(_,_,_) :-  empty_agenda.
(70.2)  earley_proc(Search,Fs,Control) :-
(70.3)          get_highest_priority_task(CurrentTask),
(70.4)          trace(newtask,CurrentTask,_),
(70.5)          add_item(CurrentTask,Control),
(70.6)          !,
(70.7)          continuation(Search,Fs,Control).

(71.1)          continuation(first,_,_) :- table_item(nil,ans(_),[],_).
(71.2)          continuation(Search,Fs,Control) :- earley_proc(Search,Fs,Control).
```

Die Verarbeitung ist (spätestens) beendet, wenn die Agenda leer ist (70.1). Ist dies nicht der Fall, wird das Agenda-Item mit der höchsten Priorität gesucht ((70.3); es handelt sich hier immer um das erste Agenda-Item in der Datenbasis) und bearbeitet (70.5). Nun wird durch `continuation/3` entschieden, ob die Verarbeitung fortgesetzt werden soll (70.7). Zuvor kann durch den Cut in (70.6) explizit *tail recursion optimization* (oder auch *last call optimization*) ausgelöst werden, da in keinem Fall Backtracking zu den zuvor bearbeiteten Zielen notwendig ist. Wenn nur das erste Ergebnis gefordert ist, kann bei Existenz eines passiven Lösungs-Items die weitere Verarbeitung abgebrochen werden (71.1). Im anderen Fall wird rekursiv `earley_proc/3` aufgerufen (71.2).

Ein Item wird in die Tabelle eingefügt (72.5) und durch `apply_task/5` bearbeitet (72.6), wenn es nicht blockiert wird ((72.2); dazu s.u.).

```
(72.1)  add_item(CurrentTask,_) :-
(72.2)          blocked(CurrentTask),
(72.3)          trace(blocking,CurrentTask,_).
(72.4)  add_item((Selected,Head,Body,N),Control) :-
(72.5)          add_item_to_table(Selected,Head,Body,N),
(72.6)          apply_task(Selected,Head,Body,N,Control).
```

Inferenzregeln Die Anwendung der Inferenzregeln wird durch `apply_task/5` gesteuert, das neben der Information über das zu bearbeitende Item mit der Angabe über die Programmkontrolle aufgerufen wird. Auf ein passives Item kann nur `passive_completion/3` angewendet werden (73.1). Für aktive Items gilt, daß entweder `active_completion/5` mindestens ein mal gelingen muß (73.3) oder aber `prediction/2` und `scanning/5` angewendet werden (73.4 bis .6). Die erste Bedingung wird durch das built-in-Prädikat `bagof/3` erreicht (73.3), das genau dann gelingt, wenn das gegebene Ziel mindestens einmal bewiesen werden kann.

```
(73.1)  apply_task(nil,Head,[],N,Control) :- passive_completion(Head,N,Control).

(73.2)  apply_task(node(S),Head,Body,N,Control) :-
```

```
(73.3)          bagof(_,active_completion(node(S),Head,Body,N,Control),_).

(73.4)  apply_task(node(S),_,_,_,Control) :- prediction(node(S),Control).

(73.5)  apply_task(node(S),Head,Body,N,Control) :-
(73.6)          scanning(node(S),Head,Body,N,Control).

(73.6)  apply_task(prolog(PrologGoal),Head,Body,N,Control) :-
(73.7)          prolog_goal(PrologGoal,Head,Body,N,Control).

(73.8)  apply_task(_,_,_,_,_).
```

Da immer eine erschöpfende Anwendung der Inferenzregeln notwendig ist, werden diese durch `fail` zum Backtracking gezwungen (s.u.). Dies ist nur bei `active_completion/5` nicht notwendig, da `bagof/3` die Anwendung steuert. Wegen der `fail`-Schleifen und für den Fall, daß überhaupt keine Anwendung der Inferenzregeln möglich ist, stellt (73.8) sicher, daß `apply_task/5` in jedem Fall gelingt. Zusätzlich wird hier die experimentelle Möglichkeit vorgesehen, Prolog-Ziele aufzurufen, d.h. Teile des Beweises von einem anderen Interpreter durchführen zu lassen oder die Seiteneffekte bestimmter built-in-Prädikate zu nutzen.

```
(74.1)  passive_completion(Selected,(_,Index),Control) :-
(74.2)          table_item(Selected,Head,Body,(N1,Index1)),
(74.3)          priority(Body,Head,N1,[Index,Index1],Control,'passive completion'),
(74.4)          fail.
```

Das Prädikat `passive_completion/3` sucht ein aktives Item in der Tabelle, dessen gewähltes Element mit dem passiven Item unifiziert (die Variablen `N` und `Index` in den Inferenzregeln beziehen sich auf die Regelnummern; dazu s.u.). Darauf wird durch `priority/6` das gewählte Element des reduzierten Items bestimmt und dieses in die Agenda aufgenommen. `active_completion/5` sucht ein passives Item in der Tabelle, um es zur Reduktion zu verwenden (75.2).

```
(75.1)  active_completion(Selected,Head,Body,(N,Index),Control) :-
(75.2)          table_item(nil,Selected,[],(_,Index1)),
(75.3)          priority(Body,Head,N,[Index,Index1],Control,'active completion').
```

Der Scanner sucht Unit-Klauseln im Programm, die mit dem gewählten Element des gegebenen aktiven Items unifizieren, so daß dieses sofort reduziert werden kann. Da hier die gleiche Grammatik wie für den Kopf-gesteuerten Interpreter verwendet wird, kann über `nonchain_rule/3` die Suche auf Lexikoneinträge der gewünschten syntaktischen Kategorie beschränkt werden, falls diese bekannt ist (76.2).

```
(76.1)  scanning(node(SelFs),Head,Body,(N,Index),Control) :-
(76.2)          nonchain_rule(SelFs,N1,lex),
(76.3)          node(N1,SelFs),
(76.4)          priority(Body,Head,N,[Index,N1],Control,scanning),
(76.5)          fail.
```

Die Vorhersage darf nur die beschränkte Merkmalsstruktur des gewählen Elements verwenden ((77.2), s.a. Abschnitt 2.3.3). Zur Bestimmung der Regelnummer kann wieder auf die zur Verfügung stehende Information über die Grammatik zugegriffen werden ((77.3); bei der geringen Anzahl von Regeln in einer lexikalisierten Grammatik ist dies allerdings weniger dringend erforderlich als ein effizienter Zugriff auf die Lexikoneinträge). Sowohl (76.2) als auch (77.3) können jedoch auch einfach weggelassen werden.

```
(77.1)  prediction(node(SelFs),Control) :-
(77.2)          restriction(SelFs,RestrSelFs),
(77.3)          (chain_rule(SelFs,N,_); nonchain_rule(SelFs,N,rule)),
(77.4)          (node(N,RestrSelFs) <--- Body),
(77.5)          priority(Body,node(RestrSelFs),N,N,Control,prediction),
(77.6)          fail.
```

Prolog-Ziele schließlich werden direkt aufgerufen (78.2) und das entsprechende Item reduziert (78.3).

```
(78.1)  prolog_goal(PrologGoal,Head,Body,N,Control) :-
(78.2)          call(PrologGoal),
(78.3)          priority(Body,Head,N,N,Control,'prolog goal').
```

Restriktion kann sehr einfach realisiert werden. Alle Merkmalsstrukturen werden unverändert gelassen (79.3), es sei denn, sie besitzen die syntaktische Kategorie VP (79.1 und .2). Mit `Cat == vp` wird sichergestellt, daß die Variable `Cat` auch tatsächlich mit `vp` instantiiert ist – eine nicht instantiierte Variable `Cat` darf durch `restriction/2` nicht instantiiert werden. Der Cut in (79.2) bewirkt, daß kein Backtracking zu (79.3) mit einer nicht beschränkten Merkmalsstruktur als Folge möglich ist.

```
(79.1)  restriction([Phon,syn:[Cat|_],Sem],[Phon,syn:[Cat|_],Sem]) :-
(79.2)          Cat == vp, !.
(79.3)  restriction(X,X).
```

Selektionsfunktion Die Bestimmung des nächsten gewählten Elements wird von `task_priority/6` übernommen, das von `priority/6` aufgerufen wird. Der Grund für die Auswahl des Elements kann (als Wert der Variablen `SelectionType`) im Trace angezeigt werden (80.3). Schließlich wird das neu bestimmte Item in die Agenda aufgenommen. Bei Tiefensuche wird es an den Anfang der Datenbasis gestellt (81.1); bei Breitensuche an das Ende (81.2).

```
(80.1)  priority(Body,Head,N,Index,Control,InfRule) :-
(80.2)          task_priority(Body,Head,N,Selected,RedBody,SelectionType),
(80.3)          trace(InfRule,(Selected,Head,RedBody,(N,Index)),SelectionType),
(80.4)          add_task_to_agenda(Control,(Selected,Head,RedBody,(N,Index))), !.

(81.1)  add_task_to_agenda(df,Task) :-   asserta(agenda_item(Task)).
(81.2)  add_task_to_agenda(bf,Task) :-   assertz(agenda_item(Task)).
```

Das Prädikat `task_priority/5` muß die interne Struktur von Unit-Items (82.1) und aktiven Items mit nur einem Element im Klauselrumpf (82.2) nicht weiter berücksichtigen, da es keinen Entscheidungsbedarf gibt. Grundsätzlich müssen nur Klauseln mit mindestens zwei negativen Literalen näher untersucht werden. Da die Grammatik nur aus unären und binären Regeln besteht, wird in (82.3) von zwei Elementen im Klauselrumpf ausgegangen (`Left` und `Right`). Wenn die Semantik des Kopfes nicht 'ausreichend instantiiert' ist (oder es überhaupt kein negatives Literal gibt, dessen Semantik mit dem positiven Literal koindiziert ist) wird als Default das am weitesten links stehende Element im Klauselrumpf gewählt (82.6).

```
(82)    % task_priority(+Body,+Head,+N,-Selected,-RedBody)
(82.1)  task_priority([],_,_,nil,[],'nothing to select').
(82.2)  task_priority([Selected],_,_,Selected,[],'the only element').
(82.3)  task_priority([Left,Right],Head,N,Selected,RedBody,'semantic head') :-
(82.4)          chain_rule(_,N,Nhd),
(82.5)          sem_inst(Head,Nhd,[Left,Right],Selected,RedBody).
(82.6)  task_priority([Left|Rest],_,_,Left,Rest,'leftmost by default').

(83.1)  sem_inst(node([_,_,sem:Sem]),Nhd,[Left,Right],Selected,RedBody) :-
(83.2)          \+ \+ ( (Sem = [pred:Pred|_], ground(Pred));
(83.3)                  (Sem = [cont:Cont|_], ground(Cont)) ),
(83.4)          semantic_head(Nhd,[Left,Right],Selected,RedBody).

(84.1)  semantic_head(left,[Left,Right],Right,[Left]).
(84.2)  semantic_head(right,[Left,Right],Left,[Right]).
```

Die einzige verbleibende Frage ist also, ob es einen semantischen Kopf gibt und ob dieser gewählt werden soll. Da die Nummer der Grammatikregel, aus der ein aktives Item stammt, vorhanden ist, kann mit `chain_rule/3` in (82.4) überprüft werden, ob ein semantischer Kopf vorhanden ist und welcher dieses ist. Die Variable `Nhd` gibt, wie oben erläutert, die Position der *non head daughter* an, so daß der semantische Kopf das andere Element des Klauselrumpfs sein muß (das Prädikat `semantic_head/4` ((84.1) und (84.2)) bestimmt den semantischen Kopf aufgrund dieser Angabe).

Da es mit dieser Implementierung möglich sein soll, auch von einer teilweise instantiierten Semantik aus zu generieren, darf nicht verlangt werden, daß die Repräsentation der Semantik *ground* ist. Andererseits muß eine gewisse Instantiierung vorhanden sein, um eine zielgerichtete Generierung zu ermöglichen. Vor allem jedoch soll beim Parsing das am weitesten links stehende Element durch (82.6) gewählt werden, ohne die Instantiierung der String-Variablen näher zu untersuchen. Als *ausreichende Instantiierung* soll hier die Instantiierung des semantischen Prädikats (83.2) oder der Variablen `Cont` für NPs angesehen werden (83.3). Da durch diesen Test keine Information hinzugefügt werden soll, wird doppelte Negation verwendet.

Kopieren von Items Wenn ein Item durch `add_item_to_table/4` in der Tabelle (72.5) bzw. durch `add_task_to_agenda/2` in der Agenda abgelegt wird (81), müssen Koindizierungen von Variablen mit dem durch die anschließende Anwendung von Inferenzregeln gebildeten neuen Items vermieden werden. Bei Koindizierungen zwischen Items besteht die Gefahr, daß ein Item der Tabelle ohne Bearbeitung spezifischer wird. Eine Lösung dieses Problems kann nur darin liegen, daß jeweils mit Kopien der Items gearbeitet wird, wobei diese Kopien 'frische Variablen' enthalten. Das built-in-Prädikat `copy_term/2` erzeugt beispielsweise eine solche Kopie. Seine explizite Verwendung ist jedoch nicht notwendig, weil jede Anwendung von `assert/1` bzw. `assert/2` automatisch eine Kopie der betreffenden Klausel erzeugt und diese in die Datenbasis einfügt (vgl. SICStus v3 User's Manual (1995, S. 124); tatsächlich kann `copy_term/2` durch `assert/1` und `retract/1` definiert werden (ebda, S. 123)).

Repräsentation der Item-Kombinationen Da aus der Grammatik Nummern für die Grammatikregeln und Lexikoneinträge zur Verfügung stehen, ist es naheliegend, sie in den Items selbst zu repräsentieren. Ein gegebenes Item enthält nur in einem Sonderfall – als Ergebnis einer Vorhersage – genau die gleichen Elemente wie die korrespondierende Grammatikregel (dazu kann eine Grammatikregel durch die Vorhersage weiter instantiiert werden). Alle anderen Items sind das Ergebnis von Kombinationen von Items. Um dies zu repräsentieren, wird bei jeder Reduktion durch `active_completion/5`, `passive_completion/3` und `scanning/5` ein geordnetes Paar aus den Indizes der beiden beteiligten Items gebildet. Vorhersagen werden nicht berücksichtigt, weil sie keine Reduktion durchführen. Das geordnete Index-Paar wird als eine zwei-elementige Prolog-Liste repräsentiert, wobei das erste Element den Index des gegebenen Items und das zweite den des gefundenen Items darstellt.

Wenn beispielsweise mit der VP-Regel `3` der lexikalische Eintrag `29` für *schläft* gescannt wird, lautet der Index der entstandenen reduzierten Grammatikregel `[3,29]`. Im Laufe einer Ableitung nimmt die Komplexität des Index immer mehr zu. So hat beispielsweise das Antwort-Item für das Parsing von *der mann schläft* den Index `[[[3,29],[[[4,6],14], 1]],0]`. Bei jeder Lösung muß am Ende der Liste das zum Schluß reduzierte Start-Item `0` stehen. Die Nummern zeigen alle beteiligten Einträge der Grammatik an. Durch die interne Struktur des Index werden zudem Unterschiede in der Analysestrategie für Parsing und Generierung deutlich (es lassen sich auch einige weniger deutliche Differenzen zwischen Tiefen- und Breitensuche finden). Der Index für die Generierung des obenstehenden Satzes lautet `[[[[4,14],6],[[3,29],1]],0]`. Die S-Regel `1` wird beim Parsing von links nach rechts erst durch die NP reduziert (`[[[4,6],14],1]`); bei der Kopf-gesteuerten Generierung wird sie zuerst durch die VP vervollständigt (`[[3,29],1]`). Auch bei der Bearbeitung der NP mit Regel `4` zeigt sich der Unterschied: `[[4,14],6]` bei der Generierung beginnend mit dem Nomen `14`; `[[4,6],14]` für das Parsing beginnend mit dem Determiner `6`.

Die Kodierung der Item-Kombinationen in den Items selbst wirft eine Frage auf, die in Abschnitt 4.2.3 im Zusammenhang mit der Repräsentation der Konstituentenstruktur in den Merkmalsstrukturen angesprochen wurde: Items bzw. Phrasen, die sich nur in ihrer Ableitungsstruktur unterscheiden, subsumieren sich nicht. Der Subsumptions-Test im Hinblick auf die Tabelle ist in einem solchen Fall erfolglos. Tatsächlich kann mit dieser Implementierung gezeigt werden, daß redundante Lösungen auftreten, wenn der Subsumptions-Test dieses Problem nicht berücksichtigt: Wenn die VP *liebt die frauen* mit erschöpfender Tiefensuche geparst wird, gibt es zwei Lösungs-Items, die sich nur in ihrer Kombinationsstruktur unterscheiden, d.h. es handelt sich um eine unechte Mehrdeutigkeit. Der Subsumptions-Check darf also die dem Kombinations-Index entsprechende vierte Argumentstelle von `table_item/4` nicht berücksichtigen. Die noch nachzutragende Definition von `blocked/1` sieht dann folgendermaßen aus:

```
(85.1)  blocked((Selected,Head,Body,N)) :-
(85.2)          table_item(Selected1,Head1,Body1,N1),
(85.3)          subsumes_chk((Selected1,Head1,Body1,_),(Selected,Head,Body,_)).
```

Ein Item wird blockiert, wenn sich (mindestens) ein subsumierendes Item in der Tabelle befindet. Der Subsumptions-Check nimmt die Variablen `N` und `N1` nicht wieder auf (85.3).

Es gibt jedoch einen entscheidenden Unterschied zwischen der Kodierung der Item-Kombinationen und der Repräsentation der Konstituentenstruktur in Merkmalsstrukturen: Die Konstituentenstruktur enthält keine prozeduralen Aspekte. Sie definiert den Analysebaum, jedoch nicht die Art der Baum-Traversierung. Demgegenüber ist der einzige Unterschied der oben aufgeführten unterschiedlichen Indizes die Reihenfolge der Kombinationen. Der korrespondierende Konstituentenstrukturbaum ist dagegen für Parsing und Generierung gleich.

Generelle Vervollständigung von Merkmalsstrukturen Obwohl es sich bei dem Uniformen Tabularen Algorithmus *(UTA)* um ein allgemeines Deduktionsverfahren handelt, geht Neumann immer von einer voll instantiierten Semantik bei der Generierung oder einem vollständig gegebenen String beim Parsing aus. Zudem ist in den Beispielen in Neumann (1994) die Zielkategorie gegeben. Wie oben erwähnt, kann die Anzahl der Kombinations-Möglichkeiten bei z.B. in bezug auf die Kategorie des Verbkomplements unterspezifizierten Grammatikregeln stark steigen (s. die alternative Definition der

Grammatikregeln 1 und 2 in Anhang A.3). Ein wichtiger Unterschied der hier dargestellten Implementierung in Hinblick auf den *UTA* ist das Ziel, auch unterspezifizierten Input – beispielsweise eine semantische Repräsentation, die unter anderem Variablen beeinhaltet – und vor allem 'gemischte' Merkmalsstrukturen vervollständigen zu können. Dies hat mehrere Konsequenzen, die auch von theoretischer Bedeutung sind.

Der Scanner wird in Neumann (1994) so definiert, daß er nur dann auf einen lexikalischen Eintrag zugreifen kann, wenn er einen Teil des Inputs verbraucht. Dieser Teil ist bei reinem Parsing bzw. reiner Generierung immer instantiiert. Um Kohärenz sicherzustellen, darf beispielsweise bei der Generierung der Semantik keine Information hinzugefügt werden. Wie in Abschnitt 4.3.3 dargelegt wurde, darf es deswegen keine Lexikoneinträge mit einer Variablen zur Repäsentation der 'leeren Semantik' geben. Bei der hier angestrebten allgemeinen Vervollständigung von Merkmalsstrukturen soll dagegen sowohl der semantischen wie der phonologischen Repräsentation Information hinzugefügt werden können. Der Scanner in (76) verlangt deshalb nur die Unifikation von Lexikoneintrag und gewähltem Element des aktiven Items. Auch für eine in der Tabelle zu findende gültige Antwort am Ende der Bearbeitung kann deshalb anstelle der unten in (87) genannten speziellen Bedingungen für Parsing und Generierung die Unifikation mit der Input-Merkmalsstruktur verlangt werden:

```
(86)    earley_solution(Fs) :-   table_item(nil,ans(Fs),[],_).
```

Da sich nur das reduzierte Start-Item durch den Prädikatsnamen `ans` auszeichnet, kann dieses sehr leicht in der Tabelle gefunden werden. Darüberhinaus ist durch diesen ausgezeichneten Namen sichergestellt, daß das als Lösung zu betrachtende passive Item nicht einfach irgendein während der Ableitung entstandenes kompatibles Item darstellt.

Wenn das essentielle Merkmal der Input-Merkmalsstruktur voll instantiiert ist, können Kohärenz und Vollständigkeit explizit gefordert werden. Bei 'gemischtem Input' kann dann kein Ergebnis mehr gefunden werden.

```
(87.1)  earley_solution([Phon,Syn,Sem]) :-
(87.2)          ground(Phon),
(87.3)          table_item(nil,ans([PhonAns,Syn,Sem]),[],_),
(87.4)          Phon == PhonAns.

(87.5)  earley_solution([Phon,Syn,Sem]) :-
(87.6)          ground(Sem),
(87.7)          table_item(nil,ans([Phon,Syn,SemAns]),[],_),
(87.8)          Sem == SemAns.
```

Probleme bei genereller Vervollständigung Wenn eine Input-Merkmalsstruktur eine voll instantiierte Semantik und dazu eine instantiierte (leere) Restliste besitzt, kann es bei mehrstelligen Verben zu Problemen kommen – es wird keine Lösung gefunden. Abgesehen von der leeren Restliste handelt es sich um eine reine Generierungs-Aufgabe. Wegen der instantiierten Semantik wählt die Selektionsfunktion immer den semantischen Kopf. Mit der VP der S-Regel werden sowohl $VP \rightarrow\ VP\ NP$ als auch $VP \rightarrow\ V$ vorhergesagt. Dem jeweils letzten Element der rechten Regelseite wird die leere Restliste 'vererbt'. Beim Scanning eines Verbeintrags mit der zweiten Regel entsteht deshalb ein passives Item für eine VP, die wiederum eine leere Restliste besitzt (z.B. mit `phon:[fi:[sieht],re:[]]`). Nach Kombination mit der ersten VP-Regel wird nun eine dem Verb folgende NP mit `phon:[fi:[],re:[]]` gesucht, da die leere Restliste des passiven VP-Items mit der ersten Wortliste der NP koindiziert ist.

Dieses Problem scheint mit einer speziellen Annahme für den Uniformen Tabularen Algorithmus zusammenzuhängen: Wenn `active_completion/5` erfolgreich ist, d.h. mindestens einmal angewendet werden kann, findet kein Scanning und keine weitere Vorhersage statt. Dies war in Abschnitt 4.3.3 damit begründet worden, daß eine weitere Suche sinnlos ist, wenn es schon ein einen Teil des Inputs verbrauchendes passives Item gibt. Dies ist unter der Annahme von voll instantiierten essentiellen Merkmalen als Input zutreffend. Beim Parsing einer VP mit der phonologischen Repräsentation `phon:[fi:[sieht,petra],re:[]]` ist nicht sofort ein lexikalischer Zugriff möglich, weil das Verb nur einen Teil der ersten Kette verbraucht. Aus diesem Grund muß eine weitere Vorhersage der beiden VP-Regeln stattfinden. Wenn dieser Algorithmus also als allgemeines Deduktionsverfahren dienen soll, muß `apply_task/5` deshalb geändert werden. Vorhersage und Scanning sind immer möglich, unabhängig von `active_completion/5`. Bei ansonsten gleichbleibenden Klauseln müssen (73.2) und (73.3) deshalb wie folgt definiert werden:

```
(88.1)  apply_task(node(S),Head,Body,N,Control) :-
(88.2)          active_completion(node(S),Head,Body,N,Control),
(88.3)          fail.
```

Zusammenfassung Obwohl sich die dargestellte Implementierung in einigen Punkten von dem als Ausgangspunkt dienenden Algorithmus unterscheidet, macht sie deutlich, welche Flexibilität das Earley-Deduktionsverfahren bietet. Da Earley-Deduktion ein allgemeines Deduktionsverfahren darstellt, kann beispielsweise das Standard-Prädikat `append/3` neu formuliert werden (s. File `grammar.pl` in Anhang A.3). Es können leicht alternative Selektionsfunktionen getestet werden. Wenn (83.2) und (83.3) auskommentiert werden, wird in jeder Chain Rule zuerst der Kopf gewählt. Werden dagegen (82.3 bis .5) weggelassen, findet immer eine Verarbeitung von links nach rechts statt. Dadurch werden Effizienzunterschiede für die verschiedenen Verarbeitungsrichtungen in Abhängigkeit von der Selektionsfunktion deutlich. So ist z.B. Kopf-gesteuertes Parsing zwar möglich, ohne eine geschickte Aufteilung der Wortkette jedoch sehr ineffizient (dazu s. Abschnitt 6).

5.5 Test-Umgebung für reversible Interpreter

Für die Entwicklung und den Test von Interpretern ist es sinnvoll, eine Menge von korrekten und falschen Testsätzen zu bilden. Wenn eine Änderung an einem Interpreter vorgenommen wird, kann so schnell festgestellt werden, ob tatsächlich noch alle und nur alle gewünschten Ergebnisse berechnet werden. Eine grundsätzliche Frage ist die Abdeckung aller relevanten Konstruktionen der Grammatik – dies kann hier nicht formal garantiert werden.

Eine mögliche Realisierung eines Interpreter-Tests ist die Repräsentation der Relation zwischen Input- und Output-Merkmalsstruktur durch ein Prolog-Fakt. Das Test-Verfahren nimmt sich den Input und vergleicht das Ergebnis mit dem angegebenen Output. Für reversible Interpreter (dies betrifft hier die Implementierung des Earley-Deduktions-Algorithmus) stellt sich die Frage, inwieweit sich die Reversibilität der Relation String-Semantik in dem Verhältnis zwischen Input und Output niederschlägt. Ein zunächst naheliegender Gedanke ist die Angabe nur einer einzigen voll spezifizierten Merkmalsstruktur. Das Testverfahren nimmt beim Parsing die phonlogische Repräsentation (und die syntaktische Kategorie) und bei der Generierung die semantische Repräsentation (sowie die syntaktische Kategorie) als Input. Das Ergebnis muß dann die voll spezifizierte Merkmalsstruktur sein.

Ein solches Test-Verfahren berücksichtigt jedoch nicht die Möglichkeit von Ambiguitäten und Paraphrasen (s. Abschnitt 2.2.2): Zu einer unterspezifizierten Input-Merkmalsstruktur kann es mehrere Output-Merkmalsstrukturen geben. Im Kontext der generellen Vervollständigung von Merkmalsstrukturen kann das oben skizzierte Test-Verfahren für Parsing und Generalisierung mit einem reversiblen Interpreter als Generalisierung einer voll spezifizierten Merkmalsstruktur aufgefaßt werden, wobei die Aufgabe des Interpreters darin besteht, die ursprüngliche Merkmalsstruktur zurückzugeben. In diesem Fall stellt sich das Problem mehrfacher Lösungen besonders scharf. Wenn bespielsweise die Merkmalsstruktur für den Satz *peter schläft* zu `[_,syn:[s],sem:[pred:sleep|_]]` generalisiert wird, sind alle Sätze mit der Verbsemantik von *schlafen* gültige Lösungen.

Auch für reversible Interpreter müssen Test-Sätze deshalb durch die explizite Angabe von Input- und Output-Merkmalsstruktur charakteriert werden. Als Beispiel wird hier der Test 21b aus dem File `tests.pl` in Anhang A.4.2 angegeben. Das Prädikat `fs_gen/3` enthält die Test-Nummer als erstes Argument; darauf folgen die Input-Merkmalsstruktur und die Liste der Output-Merkmalsstrukturen. In (89) wird festgestellt, daß eine NP mit der Semantik *woman* im Singular sowohl im Nominativ wie auch im Akkusativ als *die frau* realisiert werden kann.

(89)
```
fs_gen('21b',[_,syn:[np|_],sem:[cont:woman,quant:sg]],
        [ [phon:[fi:[die,frau|_684],re:_684],
            syn:[np,fem,sg,nom,3],sem:[cont:woman,quant:sg]],
          [phon:[fi:[die,frau|_684],re:_684],
            syn:[np,fem,sg,acc,3],sem:[cont:woman,quant:sg]] ]).
```

Der Aufruf von Testsätzen erfolgt durch eine `fail`-Schleife. Als Beispiel soll hier der Test für die Generierung angegeben werden:

```
(90.1)  test(Int,Mode,gen) :-
(90.2)        test_output(gen_correct),
(90.3)        fs_gen(N,GenFS,Wanted),
(90.4)        test_output(Mode,N),
(90.5)        findall(GenFS,proc(Int,GenFS),Solutions),
(90.6)        test_output(Mode,GenFS,Wanted,Solutions),
(90.7)        fail.
```

Das Prädikat `test/3` wird mit drei Argumenten aufgerufen: `Int` gibt den gewünschten Interpreter an; `Mode` kann mit `v` für `verbose` oder `s` für `silent` instantiiert werden und bestimmt, ob die getesteten Sätze immer ausgegeben werden oder nur bei einem Fehler (im `silent`-Modus wird bei korrekten Ergebnissen nur die Test-Nummer angezeigt). Das dritte Argument bestimmt die Verarbeitungs-Richtung (`gen`, `parse` oder `mixed` bei den verschiedenen Klauseln von `test/3`). Die Prädikate `test_output/*` mit verschiedener Stelligkeit steuern die Bildschirmausgabe.

Mit `fs_gen/3` wird eine Testeinheit gesucht (90.3). Der jeweilige Interpreter wird mit der Input-Merkmalsstruktur aufgerufen (90.5). `findall/3` erzeugt durch Backtracking alle Lösungen mit `proc/2`. Wenn keine gefunden wird, ist die Ergebnis-Liste `Solutions` leer. Durch `fail` (90.7) findet Backtracking zur nächsten Testeinheit statt.

Die Prolog-Liste mit den gewünschten Lösungen (`Wanted` in (90.3)) kann als eine Menge aufgefaßt werden, wenn die Grammatik keine unechten Ambiguitäten erlaubt, d.h. es kommen keine Elemente mehrfach vor. Darüberhinaus ist die Reihenfolge der Lösungen nicht festgelegt. Sie kann je nach Interpreter und in Abhängigkeit von Tiefen- oder Breitensuche unterschiedlich sein. Die Prolog-Liste der tatsächlichen Lösungen (`Solutions` in (90.5)) kann dagegen Elemente mehrfach enthalten, wenn der Interpreter beispielsweise unerwünschte Choice-Points setzt. Es handelt sich deshalb um ein *multiset*. Um festzustellen, ob die gewüschten Lösungen den tatsächlichen entsprechen, wird in `test_output/4` ein Prädikat `equal_elements/2` aufgerufen, das sukzessive das erste Element der einen Prolog-Liste entfernt und das korrespondierende Element der anderen Prolog-Liste löscht (s. File `test_suite.pl` in Anhang A.4.1). Hat der Interpreter alle gewünschten Lösungen geliefert (jedoch keine mehrfach), sind schließlich beide Listen leer.

Alle zum Testen der in diesem Kapitel beschriebenen Interpreter benötigten Files sind im Anhang aufgelistet. Die Interpreter sind mit SICStus Prolog 3 #3 und 2 #9 sowie mit Quintus Prolog 3.1.4 verwendet worden. Da `append/3` in Quintus Prolog vordefiniert ist, sollte die entsprechende Klausel in File `aux.pl` bei Bedarf auskommentiert werden. Im Anhang (A.4.4) werden einige Beispielaufrufe angegeben.

Kapitel 6

Ausblick

6.1 Weitergehende Überlegungen

Wie sich gezeigt hat, ist das Thema dieser Arbeit in der Literatur noch nicht erschöpfend bearbeitet worden. Es wurde ein Überblick über einige neuere Ansätze gegen, von dem aus weitere Fragestellungen möglich sind. In dem folgenden Abschnitt wird eine präzisere Definition des Begriffs 'Uniformität' vorgeschlagen. Darauf wird eine Idee für die Implementierung eines uniformen Kopf-gesteuerten Interpreters in Prolog vorgestellt, die im Rahmen dieser Arbeit nicht mehr ausgeführt werden konnte. In Abschnitt 6.1.1 wird ein Ausblick auf eine uniforme left-corner-Architektur gegeben. Daraufhin werden in Abschnitt 6.2 weitere offene Fragen erläutert.

Definition von Uniformität

Wie sich herausgestellt hat, ist der zentrale Begriff eines 'uniformen Algorithmus für Parsing und Generierung' bisher offenbar nicht präzise definiert worden. In Abschnitt 2.2.2 wurde deutlich, daß jeder parametrisierbare (Deduktions-)Algorithmus, der die Relation *String* ↔ *Semantik* in beide Richtungen berechnet, als uniform bezeichnet werden kann. Über die Breite der Parametrisierung wird dagegen keine Aussage gemacht.

Als Alternative soll hier vorgeschlagen werden, 'Uniformität' durch eine einheitliche Analysestrategie zu definieren. Dies ist eine wesentlich strengere und leicht zu überprüfende Definition, der keiner der in dieser Arbeit diskutierten Vorschläge gerecht wird. Eine einheitliche Analysestrategie ist insbesondere dann sinnvoll, wenn 'gemischte' Input-Merkmalsstrukturen vervollständigt werden sollen. Eine Anwendung ist beispielsweise die Modellierung einer Gesprächssituation, in der nur ein Teil des Satzes gehört wurde, aber zusätzlich semantische und pragmatische Constraints vorhanden sind. Eine eindeutige Zuordnung zu Parsing oder Generierung ist in dann nicht möglich – entsprechend ist eine Entscheidung zwischen zwei unterschiedlichen Analysestrategien problematisch.

Eine uniforme Analysstrategie liegt beipielsweise vor, wenn Parsing und Generierung am syntaktischen Kopf ausgerichtet werden. Dies könnte zum einen in einem in-order-

Algorithmus durch Modifikation des *SHDGA* realisiert werden (s.u.). Zum anderen könnte die Selektionsfunktion des Uniformen Tabularen Algorithmus *(UTA)* in entsprechender Weise verändert werden. Der Aufbau dieses Algorithmus ist für eine solche Modifikation offen. Das Ergebnis ist ein auf top-down-Earley-Deduktion beruhender Algorithmus mit uniformer Analysesstrategie.

Diese beiden Vorschläge richten sich am syntaktischen Kopf aus, weil dies die Verfügbarkeit von Information zum frühstmöglichen Zeitpunkt erlaubt. Wenn dagegen inkrementell von links nach rechts generiert und geparst werden soll, muß der Zugriff auf Information aus dem Lexikon in anderer Weise organisiert werden. Im Hinblick auf die verschiedenen möglichen Analysestrategien gibt es auch bei links-rechts-inkrementeller Verarbeitung wieder zwei Möglicheiten: eine top-down-Strategie mit Memoing zur Vermeidung von Terminations-Problemen oder einen left-corner-Algorithmus. Reine bottom-up-Verfahren in der in Abschnitt 4.4.1 definierten Form scheiden meiner Ansicht nach aus, weil sie für die Generierung zu ineffizient sind. Da die left-corner-Strategie nicht nur aus computerlinguistischer Sicht eine geschickte Kombination von top-down- und bottom-up-Verfahren darstellt, sondern auch in der Psycholinguistik eine große Bedeutung hat (s. z.B. Hemforth (1993, S.177)), scheint diese Alternative interessanter zu sein. In Abschnitt 6.1.1 werden einige vorläufige Überlegungen hierzu präsentiert.

Implementierung eines uniformen Kopf-gesteuerten Algorithmus

Die Implementierung eines uniformen Kopf-gesteuerten Algorithmus scheint leichter zu sein als die eines uniformen left-corner-Algorithmus, da es für beide Verarbeitungs-Richtungen existierende Algorithmen gibt. Die in van Noord (1997) vorgestellte Implementierung des head-corner-Parsers unterscheidet sich in einigen Punkten von der Kopf-gesteuerten Generierung in Shieber *at al.* (1990).

Um überhaupt eine Link-Relation feststellen zu können, findet vor dem eigentlichen Parsing ein Vorverarbeitungsschritt statt, der durch lexikalischen Zugriff für jedes Wort der gegebenen Kette die syntaktische Kategorie festlegt. Zudem muß die String-Position berücksichtigt werden: eine Konstituente muß immer innerhalb eines bestimmten Intervalls liegen. Durch das bidirektionale Parsing ist die Bestimmung des jeweils infrage kommenden Intervalls von Bedeutung, da ansonsten der Suchraum unnötig groß ist. Wenn beispielsweise $_0$ *die* $_1$ *katze* $_2$ *jagt* $_3$ *die* $_4$ *maus* $_5$ gesparst wird, ist nach der Analyse von *jagt* bekannt, daß die rechte NP zwischen Position 3 und 5 liegen muß. Nach dem Parsing von *maus* kann der fehlende Determiner nur noch zwischen Position 3 und 4 liegen.

Eine solche Verwaltung der String-Positionen ist, wie die Implementierung in Abschnitt 5.3 zeigt, bei der Generierung nicht nötig. Wenn sie in einem uniformen Algorithmus dennoch durchgeführt wird, muß durch eine Verzögerung dafür gesorgt werden, daß die benötigten arithmetischen Funktionen in Prolog erst ausgeführt werden, wenn die betreffenden Variablen ausreichend instantiiert sind. Bei der Generierung läßt sich nur festlegen, daß das erste Wort an Position 0 beginnt. Alle weiteren Positionen lassen sich dann 'rückwärts' berechnen.

Eine sinnvoller Ansatz für die Implementierung eines uniformen Kopf-gesteuerten Algorithmus scheint zu sein, beim Parsing den gegebenen String zu 're-generieren', d.h. unter bezug auf die durch den Vorverarbeitungsschritt gegebene Information den String – lizensiert durch die Grammatik – erneut zu erzeugen. Wenn dies gelingt, ist der Input-String grammatisch (also gewissermaßen *Parsing as Re-Generation* – daraus ergibt sich eine interessante Perspektive auf das Konzept von 'Monitoring', das bisher vor allem als Kontrolle des Parsings durch die Generierung verstanden wird; vgl. Abschnitt 1.4). Für die Generierung ist der Vorverarbeitungsschritt trivial: es darf jedes Wort an jeder Position angenommen werden – die nötigen Einschränkungen bei der Auswahl der lexikalischen Einträge ergeben sich während der Generierung aus der Input-Semantik. Dazu muß angemerkt werden, daß auch das Parsing grundsätzlich ohne Vorverarbeitungsschritt durchgeführt werden kann, jedoch die Bestimmung der Link-Relation dadurch ineffizienter würde (die synatktische Kategorie der infrage kommenden Worte müßte on-line bestimmt werden).

Ein weiterer in Abschnitt 3.2 erwähnter Unterschied zwischen head-corner-Parser und vom semantischen Kopf gesteuerter Generierung liegt darin, daß nur der zweite Algorithmus Nonchain Rules vorsieht. Es gibt jedoch keinen prinzipiellen Grund, der gegen die Verwendung von Nonchain Rules auch in einem head-corner-Parser spricht.

6.1.1 Uniforme left-corner-Verarbeitung

Einige Überlegungen zur left-corner-Generierung

Die Abarbeitungs-Reihenfolge beim Parsing von links nach rechts entspricht der Reihenfolge, in der auch der Mensch – insbesondere in der mündlichen Kommunikation – die Information wahrnimmt. Da die Worte nicht alle zur gleichen Zeit zur Verfügung stehen, findet eine inkrementelle Verarbeitung statt. Dies gilt aber auch für die Generierung: der Mensch erzeugt die Worte inkrementell und zeitlich linear. Es ist wünschenswert, daß ein NLP-System genauso vorgeht, um eine inkrementelle Ausgabe zu ermöglichen.

Es ergeben sich jedoch grundsätzliche Schwierigkeiten bei einer Generierung von links nach rechts. Wenn z.B. eine NP erzeugt werden soll, müssen bei der Auswahl des Determiners schon die Kongruenzmerkmale des Nomens bekannt sein. Bei der vom semantischen Kopf gesteuerten Generierung wird zuerst auf das Nomen zugegriffen, um z.B. das Genus festzulegen. Unter dem Blickwinkel der Generierung von links nach rechts macht es wenig Sinn, erst einen Determiner zu generieren, um dann festzustellen, daß es kein dazu passendes Nomen mit der gewünschten Semantik gibt.

Noch schärfer stellt sich das Problem in bezug auf die Verb-Komplemente. Die vor dem Verb stehenden Komplemente müssen durch die Subkategorisierungs-Liste des Verbs bestätigt werden. Für die nach dem Verb stehenden Komplemente ist dieser Zusammenhang umgekehrt: die Subkategorisierungs-Information ist vorhanden, wobei die Komplemente ihr entsprechen müssen. Aufgrund der vorhandenen Information können die nach dem Verb stehenden Komplemente sehr zielgerichtet generiert werden. Bei der Erzeugung der Komplemente vor dem Verb dagegen (im deutschen Nebensatz etwa betrifft dies alle Komplemente) ist die Frage, woher die Information bespielsweise über die Anzahl der Komplemente kommen soll, d.h. beispielsweise wie oft eine VP-Komplement-Regel angewendet werden darf.

Offensichtlich ist die zu bestimmten Zeitpunkten der Generierung vorhandene Information in den bisherigen Modellen nicht ausreichend. Im Rahmen dieser Arbeit wurde davon ausgegangen, daß Information z.B. über die Verbkomplemente erst vorhanden ist, wenn auf den Lexikoneintrag des Verbs zugegriffen wurde. Die oben genannten Überlegungen deuten jedoch darauf hin, daß Information, die Auswirkungen auf andere Worte hat (d.h. Kongruenz- oder Kasus-Information), in vielen Fällen früher vorhanden sein muß als die konkrete phonologische Realisierung des Wortes.

Allein schon aufgrund der Bezugnahme auf die menschliche Sprachproduktion reicht diese Problemstellung in den Bereich der Modellierung der menschlichen Sprachfähigkeit. Mit den Grundannahmen, die in dieser Arbeit gemacht werden, ist ein solches Modell aber meiner Ansicht nach nicht zu erreichen. Im Zusammenhang mit hierarchischen Lexikonmodellen ist zu überlegen, ob die bei der Generierung sehr früh benötigte Information auch diejenige ist, die auf höheren Ebenen in der Hierarchie abgelegt wird (oder sinnvoll dort abgelegt werden kann). Dafür spricht, daß konkrete Verbformen nicht zu Beginn der Generierung bekannt sein müssen, dafür aber die sehr viel allgemeinere semantische Information, wieviele Argumente es hat. Dies legt die Idee nahe, daß der Generator sehr früh auf höhere Knoten in der Lexikon-Hierarchie zugreifen kann, während er die Einträge für bestimmte Worte dagegen erst verwendet, wenn diese unmittelbar generiert oder analysiert werden sollen.

Jedes bereits generierte Wort legt den möglichen Fortsetzungen des Satzes Beschränkungen auf. Beim Parsing weckt das Vorhandensein einer linken Ecke bestimmte Erwartungen hinsichtlich der möglichen Forsetzungen, die nun noch folgen können. Bei der Generierung dagegen kann die Erzeugung einer linken Ecke als die Verpflichtung interpretiert werden, die fehlenden Elemente der rechten Regelseite einer der Regeln, in denen die geäußerte Phrase als linke Ecke dienen kann, zu generieren. In einem größeren Kontext deutet dies beispielsweise auf die Planungstheorie innerhalb der KI, da sich die Frage stellt, in welcher Art und Weise Sätze mit einem solchen Algorithmus geplant werden können.

In bezug auf die Verwendung eines hierarchischen Lexikons in der Computerlinguistik ist zu fragen, wie der Zugriff auf lexikalische Information erfolgen soll. Es ist auffällig, daß hierarchische Lexika von den in dieser Arbeit dargestellten Algorithmen nicht berücksichtigt werden. Wenn sie vorhanden sind, dann dienen sie der redundanzfreieren Speicherung von Information. Ihre interne Struktur wird jedoch von den Verarbeitungs-Algorithmen nicht verwendet. In bezug auf eine Generierung von links nach rechts ist es dagegen wünschenswert, daß beispielsweise nach der Erzeugung von zwei Verb-Komplementen festgestellt werden kann, daß intransitive Verben nun nicht mehr verwendet werden können.

Dies bedeutet, daß durch immer weitergehende Festlegungen während des Generierungs-Prozesses die Menge der infrage kommenden lexikalischen Einträge für das Verb immer kleiner wird. Daraus ergibt sich eine interessante Parallele zu der Konzeption von *incremental description refinement (IDR)* in Mellish (1988).

Van der Linden (1992) bezeichnet die inkrementelle Einschänkung der Menge der lexikalischen Items als *windowing*. In seiner Konzeption eines inkrementellen Parsings findet ein Backtracking durch das Lexikon statt. Wenn neue Information, die beispielsweise gegen einen bestimmten gewählten Subkategorisierungs-Rahmen spricht, hinzukommt, wird durch Backtracking ein anderer gewählt.[1] Nach der Grundidee von Windowing und IDR scheint mir jedoch eine andere Implementierung näherzuliegen: die verschiedenen Pfade in der lexikalischen Hierarchie müssen parallel verfolgt werden. Wenn weitere Information vorhanden ist, können bestimmte Pfade ausgeschlossen werden. In bezug auf eine Typ-Hierarchie kann es zu einer Typ-Verschärfung kommen, wenn die neue Information dies erlaubt.

Mit dieser Methode läßt sich das Lexikon für die weitere Planung des Satzes bei der Generierung berücksichtigen. Alle noch möglichen Verben und Verbklassen müssen Subtypen der noch erlaubten Typen sein. Im Rahmen eines uniformen Algorithmus kann IDR vorteilhafte Rückwirkungen auf das Parsing haben:

(91) *weil* $[_{VP}$ *dem mann* $^{*}[_{VP}$ *dem mann ...* .

In dem Beispiel geht es um eine Analyse der NPs als Verb-Komplemente. Ein 'normaler' Parser kann die mangelnde Übereinstimmung mit der Subkategorisierungs-Information des Verbs erst feststellen, wenn das Verb selbst analysiert wird. Dies ist in einem deutschen Nebensatz wie in (91) also erst am Ende des Satzes der Fall. Inkrementell gesehen kann dem Parser eine unendliche Anzahl an NPs im Dativ eingegeben werden, die als Verb-Komplemente analysiert werden, ohne daß dieser Analyseversuch scheitert. Dies ist insofern ineffizient, als der betreffende Pfad des Beweisbaumes länger verfolgt wird als nötig. Wenn der Zugriff auf die Lexikon-Information dagegen beim Parsing in der gleichen Art und Weise wie bei der Generierung erfolgt, kann der Fehler sofort bemerkt werden: es gibt kein Verb mit mehr als einem Dativ-Objekt im Lexikon.

Ein uniformer Algorithmus mit IDR erlaubt darüberhinaus eine sehr enge Verzahnung von Parsing und Generierung. Beispielsweise ist es möglich, beim inkrementellen Parsing jederzeit den Satz weiterzuführen, wenn dies mit den Constraints, die sich aus der schon durchgeführten (Teil-)Analyse ergeben, noch möglich ist. Auch dies kommt der Modellierung des menschlichen Sprachverhaltens ein Stück näher. In Hinblick auf eine Implementierung im Rahmen von Earley-Deduktion kann Monitoring sehr einfach realisiert werden: nach der Äußerung eines Wortes wird es sofort geparst, so daß alle alternativen Lesarten bis zu der betreffenden Position im Satz zur Verfügung stehen. Die Menge der State-Items ist nach diesem Monitoring bei Parsing und Generierung genau gleich.

[1] Sogar wenn das Verb selbst schon geparst wurde, steht der Subkategorisierungs-Rahmen noch nicht fest (van der Linden 1992, S.228).

Diese Überlegungen sind natürlich sehr vorläufig. Sie zeigen jedoch, daß ein uniformer left-corner-Algorithmus mit Berücksichtigung eines hierarchischen Lexikons eine sehr interessante Perspektive bietet. Als Ausgangspunkt für die Implementierung eines uniformen left-corner-Algorithmus im Rahmen von Earley-Deduktion könnte der Chart-Parser in Kilbury (1985) dienen. Er hat den besonderen Vorteil, daß die Eigenschaft des reinen bottom-up-Verfahrens, daß kein Subsumptions-Check notwendig ist, erhalten bleibt (Wirén 1987, S. 228).

Grundsätzlich gesehen handelt bei den oben vorgestellten Vorschlägen zur inkrementellen Verfeinerung von Beschreibungen um eine engere Verzahnung von hierarchischem Lexikon und Verarbeitungs-Algorithmus: Lexikalische Constraints werden sofort, d.h. bei jeder Regelanwendung, überprüft, und nicht erst, wenn die Ebene der Terminalsymbole erreicht ist. Die Terminations-Probleme, die sich aus linksrekursiven Grammatikregeln ergeben können, sind aus dieser Sicht die Folge eines zu späten Constraint-Checkings, das beispielsweise im Falle der VP-Komplement-Regel zur Vorhersage von VPs für Verben führt, die im Lexikon gar nicht vorhanden sind.

Semantische Repräsentationen

Bei der top-down-Analyse wird im Gegensatz zur reinen bottom-up-Analyse sofort ein Zielknoten, der die Input-Information aufnehmen kann, konstruiert. Die Kopf-gesteuerte Generierung und das Parsing von links nach rechts (mit einer Grammatik mit Differenzlisten-Kodierung) folgen dem Informationsfluß, der durch die Grammatik-Regeln festgelegt ist. Da nun auch die Generierung top-down und von links nach rechts erfolgen soll, muß die semantische Information anders als bisher aufgeteilt werden. Wenn wie in den vorgestellten Ansätzen Chain-Rules verwendet werden, müßte bei einer Generierung von links nach rechts eine Dekomposition der semantischen Struktur erfolgen, die den Grammatikregeln nicht entspricht. Beispielsweise muß die Semantik für ein Subjekt am Satzanfang aus der Input-Semantik extrahiert werden. Dies legt die Verwendung grundlegender semantischer (De-)Kompositions-Regeln nahe.

Beim left-corner-Parsing wird der String in die linke Ecke des ganzen Analysebaums hineingegeben und verteilt sich durch die Weitergabe der Restlisten vollständig über diesen Analysebaum. Ein analoges Verhalten bezüglich der semantischen Repräsentation bei der Generierung läßt sich mit den in dieser Arbeit verwendeten Grammatiken nicht erreichen. Da die semantische Repräsentation im Gegensatz zur Wortkette nicht festlegt, welches Element als nächstes bearbeitet werden soll, kann es sich nicht um eine Liste, sondern nur um eine Menge handeln. Es scheint mir eine naheliegende Idee zu sein, die Semantik als eine Differenz-Merkmalsstruktur weiterzugeben.

Wenn inkrementelle Generierung ermöglicht werden soll, ist analog zur Dekomposition einer gegebenen voll spezifizierten semantischen Struktur während der Analyse eine Komposition notwendig. Um bei dem Vorschlag einer Differenz-Merkmalsstruktur zu bleiben: Spätestens wenn die Rest-Semantik leer ist (und die Verpflichtungen beispielsweise in

Hinsicht auf die Verbkomplemente noch nicht erfüllt sind), besteht der Bedarf, weitere semantische Information zu inkorporieren. Dies zeigt die Anforderungen, die an die semantische Repräsentation gestellt werden.

Eine Alternative zu hierarchischen Funktor-Argument-Strukturen ist die Verwendung einer flachen Struktur für die semantische Repräsentation (Kay 1996). Copestake *et al.* (1995) schlagen beispielweise einen merkmalsbasierten Formalismus zur Vereinfachung von Generierung und Maschineller Übersetzung vor. Nicolov *et al.* (1996) verwenden flache *konzeptuelle Graphen*. Hinsichtlich der Weitergabe der Rest-Semantik haben flache Strukturen meiner Ansicht nach generell den Vorteil, daß die Differenz aus zwei Strukturen leicht zu berechnen und zu repräsentieren ist.

In bezug auf die Weitergabe der Reststruktur ergibt sich eine weitere Fragestellung: während beim Parsing die Restliste am Ende des Satzes leer sein muß, ist es nicht selbstverständlich, daß die gegebene Semantik *in einem Satz* vollständig realisiert werden muß. Es müßte möglich sein, eine Rest-Semantik, die in einem Satz nicht mehr realisiert werden kann, an den folgenden Satz weiterzureichen. Allgemein stellt sich also die Frage der Aufteilung des Inhalts auf verschiedene Sätze.

Eine ausführlichere Diskussion dieser Fragen kann im Rahmen dieser Arbeit natürlich nicht erfolgen. Die hier aufgezeigten Probleme machen aber deutlich, vor welche Schwierigkeiten die Generierung im Gegensatz zum Parsing gestellt ist. Es gibt keine empirisch beobachtbaren semantischen Repräsentationen, so daß es keinen feststehenden Ausgangspunkt für die (taktische) Generierung gibt.

6.2 Offene Fragen

Aus dem in dieser Arbeit behandelten Thema ergeben sich eine ganze Reihe offener Fragen. Im folgenden werden einige von ihnen genannt. Sie zeigen, daß es oft sehr grundsätzliche Punkte sind, die noch nicht geklärt sind.

- **Ein einheitlicher Rahmen für Earley-Deduktions-Verfahren** Vom Standpunkt der Implementierung her ist es wünschenswert, die verschiedenen Earley-Deduktions-Verfahren in einem einheitlichen, parametrisierten Rahmen zu behandeln. Die fundamentale Inferenzregel ist für alle Formen von Earley-Deduktion gleich. Sie unterscheiden sich dagegen in dem Startzustand sowie dem Zugriff auf das Programm während der Laufzeit, d.h. dem Einsatz von Scanner und Predictor. Shieber *et al.* (1996) machen beispielsweise einen Vorschlag, verschiedene Parsing-Algorithmen als Inferenzregeln in einem einheitlichen Deduktions-Verfahren zu behandeln. In bezug auf die in Abschnitt 6.1.1 vorgestellten Überlegungen ist zudem zu fragen, inwiefern das Earley-Deduktions-Verfahren mit left-corner-Verarbeitung zu verbinden ist.

- **Ausdehnung auf Morphologie** Ein naheliegender Gedanke ist die Ausdehnung der reversiblen Verarbeitung auf andere linguistische Beschreibungs-Ebenen. Da immer sowohl sprachliche Form und Inhalt vorliegen müssen, ist die kleinste zu verwendende sprachliche Einheit deshalb das Morphem. Die Generierung muß eine semantische Repräsentation als Ausgangspunkt nehmen können; das Parsing muß eine semantische Repräsentation als Ergebnis der Analyse liefern. Im Hinblick auf die verschiedenen Formen der Baumtraversierung ergibt sich daraus die Fragestellung, inwieweit sich Satz- und Wortebene unterscheiden. Neumann (1994, S.25) weist in diesem Zusammenhang darauf hin, daß die in der Computermorphologie verwendeten zwei-Ebenen-Modelle insofern einer uniformen Architektur entsprechen, als sie für Generierung und Analyse die gleiche Wissensbasis und die gleiche Verarbeitungs-Komponente verwenden.

- **Nicht-monotone Lexikon-Formalismen** Die Anforderung an die Lexikon-Komponente im Rahmen von uniformen Architekturen läßt sich in analoger Weise definieren: Es muß sowohl von der Form- wie auch von Inhaltsseite her ein lexikalischer Zugriff erfolgen können. In Hinblick auf nicht-monotone Lexikon-Formalismen ist dabei zu fragen, inwieweit sich aus der Nicht-Monotonie Probleme für den Zugriff auf die gespeicherte Information ergeben. Wenn für die Verarbeitung selbst verlangt wird, daß Parsing und Generierung (annährend) gleich effizient sein sollen, dann muß dies auch für den lexikalischen Zugriff gelten. In bezug auf den oben genannten Vorschlag zur Berücksichtigung des hierarchischen Lexikons während der Verarbeitung ergibt sich zudem die Frage, ob dies auch mit einem nicht-monotonen Lexikon möglich ist.

- **Typ-Inferenz** Ein in dieser Arbeit nicht behandelter Ansatz zu Parsing und Generierung geht von Typ-Inferenz als grundlegendem Mechanismus aus ((Zajac 1994) und (Emele und Zajac 1990)). Als grundlegende Datenstruktur werden getypte Merkmalsstrukturen verwendet. Parsing und Generierung werden (wie in dieser Arbeit) als Vervollständigung von unterspezifizierten Merkmalsstrukturen definiert. Aufgrund des Einsatzes einer Typ-Hierarchie kann Typ-Inferenz als wesentlicher Mechanismus zur Erreichung dieses Ziels verwendet werden *(Type checking as deduction)*.

  Die grundlegenden Fragen nach der Analyse-Strategie stellen sich meiner Ansicht nach aber auch in bezug auf diesen Ansatz. Beispielsweise hat ein `append` zur Listen-Konkatenation mit getypten Merkmalsstrukturen (Emele und Zajac 1990, S.294) immer noch die Form einer Implikation, so daß der Interpreter nach wie vor eine der verschiedenen Formen des Zugriffs auf das Konsequens oder die Elemente des Antezedens wählen muß. Dennoch bietet sich in Verbindung mit Typ-Inferenz eine interessante Perspektive, die bisher offenbar noch nicht ausgeschöpft worden ist.

Literatur

Abney, S. P. und M. Johnson. 1991. Memory Requirements and Local Ambiguities of Parsing Strategies. *Journal of Psychological Research* 20(3):S.233–250.

Bloomfield, L. 1933. *Language*. New York (Die Verweise beziehen sich auf die britische Ausgabe [Allan & Unwin Ltd, London, 1935]).

Carpenter, B. 1992. *The Logic of Typed Feature Structures. With Applications to Unification Grammars, Logic Programs and Constraint Resolution*. Cambridge, England: Cambridge University Press.

Clocksin, W. F. und C. S. Mellish. 1984. *Programming in Prolog*. Berlin: Springer.

Copestake, A., D. Flickinger, R. Malouf, S. Riehemann und I. Sag. 1995. Translation using Minimal Recursion Semantics.

Covington, M. A. 1994. *Natural Language Processing for Prolog Programmers*. Englewood Cliffs, NJ: Prentice Hall.

Dymetman, M. 1994. Inherently Reversible Grammars. In T. Strzalkowski (Hrsg.), *Reversible Grammar in Natural Language Processing*, S.33–57. Kluwer Academic Publishers. Dordrecht, The Netherlands.

Dymetman, M., P. Isabelle und F. Perrault. 1990. A Symmetrical Approach to Parsing and Generation. In *Proceedings of the 13th International Conference on Computational Linguistics (COLING)*, S.90–96.

Earley, J. 1970. An Efficient Context-Free Parsing-Algorithm. In *Communications of the ACM*, Vol. 13, S.94–102 [Wiederabdruck in Grosz *et al.* (1986, S.25–33)].

Emele, M. und R. Zajac. 1990. Typed Uunification Grammars. In *Proceedings of the 13th International Conference on Computational Linguistics (COLING)*, Vol. 3, S.293–298.

Erbach, G. 1994. Bottom-Up Earley-Deduction. In *Proceedings of the 15th International Conference on Computational Linguistics (COLING)*, S.796–802.

Erbach, G. 1995a. *Bottom-Up Earley Deduction for Preference-Driven Natural Language Processing*. PhD thesis, Universität des Saarlandes, Saarbrücken. Entwurf vom 31.8.1995.

Erbach, G. 1995b. ProFIT: Prolog with Features, Inheritance and Templates. In *Proceedings of the 7th Conference of the European Chapter of the Association of Computational Linguistics (EACL)*, Dublin.

Gazdar, G. und C. Mellish. 1989. *Natural Language Processing in PROLOG. An Introduction to Computational Linguistics*. Wokingham: Addison-Wesley.

Gerdemann, D. 1991. *Parsing and Generation of Unification Grammars.* PhD thesis, University of Illinois.

Gerdemann, D. 1994. Parsing as Tree Traversal. In *Proceedings of the 15th International Conference on Computational Linguistics (COLING).*

Gerdemann, D. und E. Hinrichs. 1995. Some open Problems in head-driven-Generation. In *Linguistics & Computation*, S.165–197, Stanford. CSLI.

Grosz, B. J., K. S. Jones und B. L. W. (Hrsg)). 1986. *Readings in Natural Language Processing.* Los Altos: Morgan Kaufmann.

Hemforth, B. 1993. *Kognitives Parsing: Repräsentation und Verarbeitung sprachlichen Wissens.* Sankt Augustin: Infix.

Höhfeld, M. und G. Smolka. 1988. Definite Relations over Constraint Languages. Technical Report 53, LILOG IBM, Stuttgart.

Johnson, M. 1995. *Constraint-based Natural Language Parsing.* Entwurf vom 6.8.1995.

Kasper, R. T. und W. C. Rounds. 1986. A Logical Semantics for Feature Structures. In *Proceedings of the 24th Annual Meeting of the Association for Computational Linguistics (ACL)*, S.257–266.

Kay, M. 1979. Functional grammar. In *Proceedings of the Fifth Annual Meeting of the Berkeley Society of Linguistics*, S.142–158, Berkeley.

Kay, M. 1980. Algorithm Schemata and Data Structures in syntactic Processing. Technical report, Xerox PARC, Palo Alto [Wiederabdruck in Grosz *et al.* (1986, S.35-70)].

Kay, M. 1989. Head-driven Parsing. In *Proceedings of the International Workshop on Parsing Technologies*, S.52–62, Pittsburgh. Carnegie Mellon University.

Kay, M. 1996. Chart Generation. In *Proceedings of the 34th Annual Meeting of the Association for Computational Linguistics (ACL)*, S.200–204.

Kilbury, J. 1985. Chart Parsing and the Earley Algorithm. In U. Klenk (Hrsg.), *Kontextfreie Syntaxen und verwandte Systeme. Vorträge eines Kolloquiums in Grand Ventron im Oktober 1984*, S.76–89. Niemeyer, Tübingen.

Kilbury, J. 1986. Language Variation, Parsing, and the Modelling of Users' Language Varieties. In *Proceedings of ECAI*, Vol. 2, Brighton.

Kilbury, J. 1996. Top-Down Predictive Linking and Complex-Feature-Based Formalisms. In *Proceedings of the 16th International Conference on Computational Linguistics (COLING).*

Kilbury, J. (unveröffentlicht). Encoding Constituent Structure in Feature Structures.

König, E. 1994. Syntactic-Head-Driven Generation. In *Proceedings of the 15th International Conference on Computational Linguistics (COLING)*, S.475–481, Kyoto.

Levelt, W. J. M. 1989. *Speaking. From Intention to Articulation.* Cambridge, MA: MIT Press.

Mahesh, K. 1994. Reaping the Benefits of Interactive Syntax and Semantics. In *Proceedings of the 32nd Annual Meeting of the Association for Computational Linguistics (ACL)*, S.310–312.

Manandhar, S. 1993. CUF in Context. In *DYANA-2. Deliverable R1.2.A.*

Mellish, C. 1988. Implementing Systemic Classification by Unification. *Computational Linguistics* 14(1):S.40–51.

Minnen, G., D. Gerdemann und T. Götz. 1995. Off-line Optimalization for Earley-style HPSG Processing. In *Proceedings of the 7th Conference of the European Chapter of the Association of Computational Linguistics (EACL).*

Müller, S. 1994. *Prolog und Computerlinguistik.* WWW: http://www.compling.hu-berlin.de/~stefan/: Humbold-Universität zu Berlin.

Neumann, G. 1994. *A Uniform Computational Model for Natural Language Parsing and Generation.* PhD thesis, Universität des Saarlandes, Saarbrücken.

Neumann, G. und G. van Noord. 1992. Self-Monitoring with Reversible Grammars. In *Proceedings of the 14th International Conference on Computational Linguistics (COLING)*, S.700–706.

Nicolov, N., C. Mellish und G. Richie. 1996. Approximate Generation from Non-Hierarchical Representations. In *8th International Workshop on Natural Language Generation*, Herstmonceux Castle, UK.

Partee, B. H., A. ter Meulen und R. E. Wall. 1990. *Mathematical Methods in Linguistics.* Dordrecht: Kluwer Academic Publishers.

Pereira, F. C. und S. M. Shieber. 1987. *Prolog and Natural-Language Analysis.* CSLI lecture notes, No.10. Center for the Study of Language and Information, Stanford.

Pereira, F. C. und D. Warren. 1980. Definite Clause Grammars for Language Analysis – a Survey of the Formalism and a Comparison with Augmented Transition Networks. *Artificial Intelligence* 13:S.231–278 [Wiederabdruck in Grosz *et al.* (1986, S.101–124)].

Pereira, F. C. und D. Warren. 1983. Parsing as Deduction. In *Proceedings of the 21st Annual Meeting of the Association for Computational Linguistics (ACL)*, S.137–144.

Pollard, C. und I. A. Sag. 1994. *Head-Driven Phrase Structure Grammar*. The University of Chicago Press, Chicago.

Rumpf, C. 1995. Modell einer aspektuellen Klassifikation japanischer Verben und seine Implementierung. Magisterarbeit, Heinrich-Heine-Universität Düsseldorf.

Sells, P. 1985. *Lectures on Contemporary Syntactic Theories.* Chicago: Chicago University Press.

Shieber, S. M. 1985. Using Restriction to extend Parsing Algorithms for Complex-Feature-Based Formalisms. In *Proceedings of the 23rd Annual Meeting of the Association for Computational Linguistics (ACL)*, S.145–52.

Shieber, S. M. 1986. *An Introduction to Unification-based Approaches to Grammar.* CSLI, Stanford.

Shieber, S. M. 1988. A Uniform Architecture for Parsing and Generation. In *Proceedings of the 12th International Conference on Computational Linguistics*, S.614–619.

Shieber, S. M., Y. Schabes und F. C. Pereira. 1996. Principles and implementation of deductive parsing. Technical Report TR-11-94, Center for Research in Computing Technology, Division of Applied Sciences, Harvard University.

Shieber, S. M., G. van Noord, F. C. Pereira und R. Moore. 1990. Semantic-head-driven Generation. *Computational Linguistics* 16(1):S.30–42.

Strzalkowski, T. 1994. A General Computational Method for Grammar Inversion. In T. Strzalkowski (Hrsg.), *Reversible Grammar in Natural Language Processing*, S.175–196.

Swedish Institute of Computer Science, Kista, Sweden. 1995. *SICStus v3 User's Manual (Release 3 ♯ 0).*

van der Linden, E.-J. 1992. Incremental Processing and the Hierarchical Lexicon. *Computational Linguistics* 18(2):S.219–238.

van Noord, G. 1989. *BUG: A directed bottom-up Generator for Unification-based Formalisms.* Vol. 4 of *Working Papers in Natural Language Processing.* Stichting Taaltechnologie Utrecht: Katholieke Universiteit Leuven.

van Noord, G. 1990a. An Overview of head-driven bottom-up Generation. In R. Dale, C. Mellish, und M. Zock (Hrsg.), *Current Research in Natural Language Generation*, London. Academic Press.

van Noord, G. 1990b. Reversible Unification-based Machine Translation. In *Proceedings of the 13th International Conference on Computational Linguistics (COLING)*, Vol. 2, S.299–304.

van Noord, G. 1991. Head Corner Parsing for Discontinous Constituency. In *Proceedings of the 29th Annual Meeting of the Association for Computational Linguistics (ACL)*, S.114–121.

van Noord, G. 1993. *Reversibility in Natural Language Processing.* PhD thesis, University of Utrecht, 1993.

van Noord, G. 1997. An efficient Implementation of the head-corner-Parser. Erscheint in *Computational Linguistics.*

Wedekind, J. 1988. Generation as structure driven Derivation. In *Proceedings of the 12th International Conference on Computational Linguistics (COLING)*, S.732–737.

Wilcock, G. und Y. Matsumoto. 1996. Reversible delayed lexical choice in a bidirectional Framework. In *Proceedings of the 16th International Conference on Computational Linguistics (COLING)*, S.758–763.

Wirén, M. 1987. A Comparison of Rule-Invocation strategies in Context-Free Cart Parsing. In *Proceedings of the third Conference of the European Chapter of the Association for Computational Linguistics (EACL)*, S.226–233.

Zajac, R. 1994. A Uniform Architecture for Parsing, Generation and Transfer. In T. Strzalkowski (Hrsg.), *Reversible Grammar in Natural Language Processing*, S.97–112.

Anhang A

Quell-Kode der Implementierungen

A.1 Semantic-head-driven Generation

A.1.1 Backtracking-Interpreter

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                             %
%       file:           head.pl                                               %
%       purpose:        semantic-head-driven generation                       %
%       author:         Sebastian Varges                                      %
%       date:           Tue Feb  4 12:35:37 MET 1997                          %
%       related files:  memo_head.pl, top.pl, grammar.pl                      %
%                                                                             %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

head_proc(Root) :-
        head_link(Root,Pivot),
        nonchain_rule(Pivot,N,RuleType),
        instantiate_nonchain_rule(RuleType,N,Pivot,RHS),
        head_proc_rhs(RHS),
        connect_up(Pivot,Root).

instantiate_nonchain_rule(lex,N,Pivot,[]) :-    %  lexical entry
        node(N,Pivot).

instantiate_nonchain_rule(rule,N,Pivot,RHS) :-  % grammar rule
        (node(N,Pivot) <--- RHS).

head_proc_rhs([]).
head_proc_rhs([node(F)|R]) :-
        head_proc(F),
        head_proc_rhs(R).

connect_up(Pivot,Pivot).
connect_up(Pivot,Root) :-
        head_link(Root,Pivot),
        chain_rule(NewPivot,N,RuleType),
```

```
        instantiate_chain_rule(RuleType,N,Pivot,NewPivot,RedRHS),
        head_proc_rhs(RedRHS),
        connect_up(NewPivot,Root).

instantiate_chain_rule(unary,N,Pivot,NewPivot,[]) :-
        (node(N,NewPivot) <--- [node(Pivot)]).

instantiate_chain_rule(left,N,Pivot,NewPivot,[node(Left)]) :-
        (node(N,NewPivot) <--- [node(Left),node(Pivot)]).

instantiate_chain_rule(right,N,Pivot,NewPivot,[node(Right)]) :-
        (node(N,NewPivot) <--- [node(Pivot),node(Right)]).
```

A.1.2 Selektives Memoing

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                           %
%       file:           memo_head.pl                                        %
%       purpose:        semantic-head-driven generation with selective      %
%                       memoization                                         %
%       author:         Sebastian Varges                                    %
%       date:           Tue Feb  4 12:35:37 MET 1997                        %
%       related files:  head.pl, top.pl, grammar.pl                         %
%                                                                           %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
:- dynamic memo_goal /1.
:- dynamic memo_result /1.

% call from test_suite.pl
memo_hd_proc(Root) :- memo_clear, memo_head_proc(Root).
%------------------------

memo_head_proc(Root) :-
        memo_goal(Goal), subsumes_chk(Goal,Root), !,
        memo_result(Root).

memo_head_proc(Root) :-
        head_link(Root,Pivot),
        nonchain_rule(Pivot,N,RuleType),
        instantiate_nonchain_rule(RuleType,N,Pivot,RHS),
        memo_head_proc_rhs(RHS),
        memo_connect_up(Pivot,Root),
        assertz(memo_result(Root)),
        fail.

memo_head_proc(Root) :- assertz(memo_goal(Root)),fail.

memo_head_proc(Root) :- memo_result(Root).

memo_head_proc_rhs([]).
```

```
memo_head_proc_rhs([node(F)|R]) :-
        memo_head_proc(F),
        memo_head_proc_rhs(R).

memo_connect_up(Pivot,Pivot).
memo_connect_up(Pivot,Root) :-
        head_link(Root,Pivot),
        chain_rule(NewPivot,N,RuleType),
        instantiate_chain_rule(RuleType,N,Pivot,NewPivot,RedRHS),
        memo_head_proc_rhs(RedRHS),
        memo_connect_up(NewPivot,Root).

%-------------------------------------------------------------
memo_clear :- retractall(memo_goal(_)),
              retractall(memo_result(_)).
```

A.2 Top-down-Earley-Deduktion

A.2.1 Interpreter

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                           %
%       file:           chart.pl                                            %
%       purpose:        Earley Deduction for Parsing and Generation         %
%       author:         Sebastian Varges                                    %
%       date:           Sat Feb 15 13:31:04 MET 1997                        %
%       related refs:   [Neumann, 1994]                                     %
%                                                                           %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
:- dynamic table_item /4.
:- dynamic agenda_item /1.
:- dynamic switch /2.
%--------------------------------
earley_proc(Fs,Search,Control,Trace,Display) :-
        clear_tables,
        set_switch(trace,Trace),
        init_agenda(Fs),
        earley_proc(Search,Fs,Control),
        !,
        display_solutions(Display,Search,Fs).
/*
% explicit completeness and coherence for ground semantics/phonetics
earley_solution([Phon,Syn,Sem]) :-
        ground(Phon),
        table_item(nil,ans([PhonAns,Syn,Sem]),[],_),
        Phon == PhonAns.

earley_solution([Phon,Syn,Sem]) :-
        ground(Sem),
        table_item(nil,ans([Phon,Syn,SemAns]),[],_),
        Sem == SemAns.
*/
earley_solution(Fs) :-  table_item(nil,ans(Fs),[],_). % unification

clear_tables :-
        retractall(table_item(_,_,_,_)),
        retractall(agenda_item(_)),
        retractall(switch(_,_)).

init_agenda(Fs) :- assertz(agenda_item((node(Fs),ans(Fs),[],(_,0)))).

% loop for processing of agenda items
earley_proc(_,_,_) :- empty_agenda.
earley_proc(Search,Fs,Control) :-
        get_highest_priority_task(CurrentTask),
        trace(newtask,CurrentTask,_),
```

```
        add_item(CurrentTask,Control),
        !,                              % tail recursion optimalization
        continuation(Search,Fs,Control).

continuation(first,_,_) :-
        table_item(nil,ans(_),[],_). % option: first solution
continuation(Search,Fs,Control) :- earley_proc(Search,Fs,Control).

empty_agenda :- \+ agenda_item(_).

add_task_to_agenda(df,Task) :-    asserta(agenda_item(Task)). % depth-first
add_task_to_agenda(bf,Task) :-    assertz(agenda_item(Task)). % breath-first

get_highest_priority_task(CurrentTask) :-  retract(agenda_item(CurrentTask)).

add_item(CurrentTask,_) :-
        blocked(CurrentTask),
        trace(blocking,CurrentTask,_).

add_item((Selected,Head,Body,N),Control) :-        % for non-blocked items:
        add_item_to_table(Selected,Head,Body,N),   %  add them to table
        apply_task(Selected,Head,Body,N,Control).  %  apply inference rules

blocked((Selected,Head,Body,_)) :-
        table_item(Selected1,Head1,Body1,_),
        subsumes_chk((Selected1,Head1,Body1,_),(Selected,Head,Body,_)).

% add_item_to_table(CurrentTask) :- assertz(table_item(CurrentTask)).
add_item_to_table(Selected,Head,Body,N) :-
        assertz(table_item(Selected,Head,Body,N)).

% call for inference rules
apply_task(nil,Head,[],N,Control) :- passive_completion(Head,N,Control).

apply_task(node(S),Head,Body,N,Control) :-
        %% setof(_,active_completion(node(S),Head,Body,N,Control),_). % [Neumann 1994]
        % to make test sentences 42l and 42m work:
        active_completion(node(S),Head,Body,N,Control),fail.

apply_task(node(S),_,_,_,Control) :-  prediction(node(S),Control).

apply_task(node(S),Head,Body,N,Control) :-
        scanning(node(S),Head,Body,N,Control).

apply_task(prolog(PrologGoal),Head,Body,N,Control) :-
        prolog_goal(PrologGoal,Head,Body,N,Control).

apply_task(_,_,_,_,_).
```

```
% inference rules
passive_completion(Selected,(_,Index),Control) :-
        table_item(Selected,Head,Body,(N1,Index1)),
        priority(Body,Head,N1,[Index,Index1],Control,'passive completion'),
        fail.

active_completion(Selected,Head,Body,(N,Index),Control) :-
        table_item(nil,Selected,[],(_,Index1)),
        priority(Body,Head,N,[Index,Index1],Control,'active completion').

scanning(node(SelFs),Head,Body,(N,Index),Control) :-
        nonchain_rule(SelFs,N1,lex),
        node(N1,SelFs),
        priority(Body,Head,N,[Index,N1],Control,scanning),
        fail.

prediction(node(SelFs),Control) :-
        restriction(SelFs,RestrSelFs),
        (chain_rule(SelFs,N,_);nonchain_rule(SelFs,N,rule)),
        (node(N,RestrSelFs) <--- Body),
        priority(Body,node(RestrSelFs),N,N,Control,prediction),
        fail.

prolog_goal(PrologGoal,Head,Body,N,Control) :-
        call(PrologGoal),
        priority(Body,Head,N,N,Control,'prolog goal').

% restriction of feature structure with instantiated category vp
restriction([Phon,syn:[Cat|_],Sem],[Phon,syn:[Cat|_],Sem]) :- Cat == vp,!.
restriction(X,X).

% determine new item and add it to agenda
priority(Body,Head,N,Index,Control,InfRule) :-
        task_priority(Body,Head,N,Selected,RedBody,SelectionType),
        trace(InfRule,(Selected,Head,RedBody,(N,Index)),SelectionType),
        add_task_to_agenda(Control,(Selected,Head,RedBody,(N,Index))),!.

% task_priority(+Body,+Head,+N,-Selected,-RedBody),
task_priority([],_,_,nil,[],'nothing to select').
task_priority([Selected],_,_,Selected,[],'the only element').
task_priority([Left,Right],Head,N,Selected,RedBody,'semantic head') :-
        chain_rule(_,N,Nhd),
        sem_inst(Head,Nhd,[Left,Right],Selected,RedBody).
task_priority([Left|Rest],_,_,Left,Rest,'leftmost by default').

sem_inst(node([_,_,sem:Sem]),Nhd,[Left,Right],Selected,RedBody) :-
        \+ \+ (    (Sem = [pred:Pred|_], ground(Pred));
                   (Sem = [cont:Cont|_], ground(Cont))  ),
        semantic_head(Nhd,[Left,Right],Selected,RedBody).
```

```
semantic_head(left,[Left,Right],Right,[Left]).
semantic_head(right,[Left,Right],Left,[Right]).
```

A.2.2 Hilfsprädikate

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                            %
%       file:           aux.pl                                               %
%       purpose:        auxiliary predicates for chart.pl                    %
%       author:         Sebastian Varges                                     %
%       date:           Wed Feb 19 16:42:24 MET 1997                         %
%       related files:  chart.pl                                             %
%                                                                            %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%***********   user output: trace, display solutions and tables   ***********

trace(_,_,_) :- get_switch(trace,off).
trace(InfRule,Item,SelectionType) :-
        get_switch(trace,on),
        show_trace(InfRule,Item,SelectionType),!.

%---  show_trace(+InfRule,(+Selected,+Head,+Body,+RuleIndex))
show_trace(newtask,CurrentTask,_) :-
        write('**************************  new task **************************'),
        nl, write_item(CurrentTask),step.

show_trace(blocking,_,_) :- write('%%%%  ... blocked'),nl,step.
show_trace(InfRule,Item,SelectionType) :-
        write('% ----------------------------------------------------'),nl,
        write('%  inference rule:       '), write(InfRule),nl,
        write('%  reduced new item:'), nl, write_item(Item),
        tab(3),write('selection type:  '), write(SelectionType),nl,nl,
        step.

step :- write('[Options: a = abort, n = nonstop, <rtn> = creep]'),nl,
        write('||  '), read_line(ASCII),name(Option,ASCII),
        process_option(Option).

process_option(a) :- abort.
process_option(n) :- set_switch(trace,off).
process_option(_).

%--- display_solutions(Mode,Goal)
display_solutions(off,_,Fs) :- earley_solution(Fs).
display_solutions(on,all,Fs) :-
        nl,
        earley_solution(Fs),
        write(Fs),nl,
        fail.
display_solutions(on,first,Fs) :-
```

```
        earley_solution(Fs),
        nl, write('% first solution:    '), write(Fs),nl.
display_solutions(on,_,_) :- write('% no (more) solutions'),nl.

%--- tables/0
tables :-
        nl, write('********************  agenda items  ********************'),nl,nl,
        write_agenda,
        nl, write('********************  table items   ********************'),nl,nl,
        write_table.

write_agenda :-
        agenda_item(Item),
        nl,write('%-----------------------------------------------'),nl,
        write_item(Item),nl,
        fail.
write_agenda.

write_table :-
        table_item(Selected,Head,Body,N),
        nl,write('%-----------------------------------------------'),nl,
        write_item((Selected,Head,Body,N)),nl,
        fail.
write_table.

write_item((nil,Head,[],(_,Index))) :-
        write('%         Passive Item:'),nl,nl,
        write(Head),write('.'),nl,
        tab(3),write('Grammar Rule Index:  '),write(Index),nl,nl,!.

write_item((Selected,Head,Body,(_,Index))) :-
        write('%        Active Item:'),nl,nl,
        tab(3),write(Head),tab(3),write('<---'),nl,
        write_body(Body),nl,
        tab(3),write('Selected:'),nl,nl,
        tab(8),write(Selected),nl,nl,
        tab(3),write('Grammar Rule Index:  '),write(Index),nl,nl,!.

write_body([]).
write_body([F|R]) :-
        tab(8),write(F),nl,
        write_body(R).

get_switch(Switch,Mode) :- switch(Switch,Mode).
set_switch(Switch,Mode) :- assertz(switch(Switch,Mode)).

% read_line/1 reads full line from standard input, terminates with return
read_line(Line) :- read_line([],Line),!.
read_line(Akku,Line) :-
         get0(X),
```

```
         ln_continuation(X,Akku,Line).

ln_continuation(X,Line,Line) :-   X = 10,!.
ln_continuation(X,Akku,Line) :-
         append(Akku,[X],Akku1),!,
         read_line(Akku1,Line).

% not necessary for Quintus Prolog
append([],L,L).
append([F|R],L,[F|R1]) :-
        append(R,L,R1).
```

A.3 Beispielgrammatik

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                              %
%       file:           grammar.pl                                            %
%       purpose:        example grammar with rule numbers                     %
%       author:         Sebastian Varges                                      %
%       date:           Mon Jan 27 16:10:41 MET 1997                          %
%       related files:  top.pl (links and declarations)                       %
%                                                                              %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% grammar rules
node(1,[phon:[fi:Fi,re:Re],syn:[s],sem:Sem]) <---
        [node([phon:[fi:Fi,re:Re1],syn:[np|SynNp],sem:SemNp]),
         node([phon:[fi:Re1,re:Re],syn:[vp,sc:[[syn:[np|SynNp],sem:SemNp]]],sem:Sem])].

node(2,[phon:[fi:Fi,re:Re],syn:[vp,sc:[Subj|Cs]],Sem]) <---
        [node([phon:[fi:Fi,re:Re1],
               syn:[vp,sc:[Subj,[syn:[np,Gen,Num,Cas,Pers],SemC]|Cs]],Sem]),
         node([phon:[fi:Re1,re:Re],syn:[np,Gen,Num,Cas,Pers],SemC])].
/*
% category of complement from subcat list
% -> more item combinations with earley deduction
node(1,[phon:[fi:Fi,re:Re],syn:[s],sem:Sem]) <---
        [node([phon:[fi:Fi,re:Re1],SynC,SemC]),
         node([phon:[fi:Re1,re:Re],syn:[vp,sc:[[SynC,SemC]]],sem:Sem])].

node(2,[phon:[fi:Fi,re:Re],syn:[vp,sc:[Subj|Cs]],Sem]) <---
        [node([phon:[fi:Fi,re:Re1],syn:[vp,sc:[Subj,[SynC,SemC]|Cs]],Sem]),
         node([phon:[fi:Re1,re:Re],SynC,SemC])].
*/
node(3,[phon:[fi:Fi,re:Re],syn:[vp,sc:Cs],Sem]) <---
        [node([phon:[fi:Fi,re:Re],syn:[v,sc:Cs],Sem])].

node(4,[phon:[fi:Fi,re:Re],syn:[np,Gen,Num,Cas,Pers],sem:[cont:Ct,quant:Q]]) <---
        [node([phon:[fi:Fi,re:Re1],syn:[det,Gen,Num,Cas],sem:[quant:Q]]),
         node([phon:[fi:Re1,re:Re],syn:[n,Gen,Num,Cas,Pers],sem:[cont:Ct,quant:Q]])].

node(5,[phon:[fi:Fi,re:Re],syn:[np,Gen,Num,Cas,Pers],sem:Sem]) <---
        [node([phon:[fi:Fi,re:Re],syn:[pn,Gen,Num,Cas,Pers],sem:Sem])].

%% lexical entries
% determiners
node(6,[phon:[fi:[der|R],re:R],syn:[det,masc,sg,nom],sem:[quant:sg]]).
node(7,[phon:[fi:[den|R],re:R],syn:[det,masc,sg,acc],sem:[quant:sg]]).
node(8,[phon:[fi:[die|R],re:R],syn:[det,masc,pl,nom],sem:[quant:pl]]).
node(9,[phon:[fi:[die|R],re:R],syn:[det,masc,pl,acc],sem:[quant:pl]]).

node(10,[phon:[fi:[die|R],re:R],syn:[det,fem,sg,nom],sem:[quant:sg]]).
node(11,[phon:[fi:[die|R],re:R],syn:[det,fem,sg,acc],sem:[quant:sg]]).
```

```
node(12,[phon:[fi:[die|R],re:R],syn:[det,fem,pl,nom],sem:[quant:pl]]).
node(13,[phon:[fi:[die|R],re:R],syn:[det,fem,pl,acc],sem:[quant:pl]]).

% nouns
node(14,[phon:[fi:[mann|R],re:R],syn:[n,masc,sg,nom,3],sem:[cont:man,quant:sg]]).
node(15,[phon:[fi:[mann|R],re:R],syn:[n,masc,sg,acc,3],sem:[cont:man,quant:sg]]).
node(16,[phon:[fi:[ball|R],re:R],syn:[n,masc,sg,nom,3],sem:[cont:ball,quant:sg]]).
node(17,[phon:[fi:[ball|R],re:R],syn:[n,masc,sg,acc,3],sem:[cont:ball,quant:sg]]).
node(18,[phon:[fi:[maenner|R],re:R],syn:[n,masc,pl,nom,3],sem:[cont:man,quant:pl]]).
node(19,[phon:[fi:[maenner|R],re:R],syn:[n,masc,pl,acc,3],sem:[cont:man,quant:pl]]).

node(20,[phon:[fi:[frau|R],re:R],syn:[n,fem,sg,nom,3],sem:[cont:woman,quant:sg]]).
node(21,[phon:[fi:[frau|R],re:R],syn:[n,fem,sg,acc,3],sem:[cont:woman,quant:sg]]).
node(22,[phon:[fi:[frauen|R],re:R],syn:[n,fem,pl,nom,3],sem:[cont:woman,quant:pl]]).
node(23,[phon:[fi:[frauen|R],re:R],syn:[n,fem,pl,acc,3],sem:[cont:woman,quant:pl]]).

% proper nouns
node(24,[phon:[fi:[peter|R],re:R],syn:[pn,masc,sg,dat,3],sem:[cont:peter,quant:sg]]).
node(25,[phon:[fi:[peter|R],re:R],syn:[pn,masc,sg,nom,3],sem:[cont:peter,quant:sg]]).
node(26,[phon:[fi:[petra|R],re:R],syn:[pn,fem,sg,_,3],sem:[cont:petra,quant:sg]]).
node(27,[phon:[fi:[du|R],re:R],syn:[pn,_,sg,nom,2],sem:[cont:you,quant:sg]]).

%% verbs
%   with one conplement
node(28,[phon:[fi:[traeumt|R],re:R],
      syn:[v,sc:[[syn:[np,_,sg,nom,3],sem:ArgSem]]],
      sem:[pred:dream,arg1:ArgSem]]).

node(29,[phon:[fi:[schlaeft|R],re:R],
      syn:[v,sc:[[syn:[np,_,sg,nom,3],sem:ArgSem]]],
      sem:[pred:sleep,arg1:ArgSem]]).

node(30,[phon:[fi:[schlafen|R],re:R],
      syn:[v,sc:[[syn:[np,_,pl,nom,3],sem:ArgSem]]],
      sem:[pred:sleep,arg1:ArgSem]]).

node(31,[phon:[fi:[schlaefst|R],re:R],
      syn:[v,sc:[[syn:[np,_,sg,nom,2],sem:ArgSem]]],
      sem:[pred:sleep,arg1:ArgSem]]).

% verbs with two complements
node(32,[phon:[fi:[liebt|R],re:R],
      syn:[v,sc:[[syn:[np,_,sg,nom,3],sem:ArgSem1],
                 [syn:[np,_,_,acc,_],sem:ArgSem2]]],
      sem:[pred:love,arg1:ArgSem1,arg2:ArgSem2]]).

node(33,[phon:[fi:[liebst|R],re:R],
      syn:[v,sc:[[syn:[np,_,sg,nom,2],sem:ArgSem1],
                 [syn:[np,_,_,acc,_],sem:ArgSem2]]],
      sem:[pred:love,arg1:ArgSem1,arg2:ArgSem2]]).
```

```
node(34,[phon:[fi:[sieht|R],re:R],
      syn:[v,sc:[[syn:[np,_,sg,nom,3],sem:ArgSem1],
                 [syn:[np,_,_,acc,_],sem:ArgSem2]]],
      sem:[pred:see,arg1:ArgSem1,arg2:ArgSem2]]).

node(35,[phon:[fi:[siehst|R],re:R],
      syn:[v,sc:[[syn:[np,_,sg,nom,2],sem:ArgSem1],
                 [syn:[np,_,_,acc,_],sem:ArgSem2]]],
      sem:[pred:see,arg1:ArgSem1,arg2:ArgSem2]]).

% verbs with three complements
node(36,[phon:[fi:[gibst|R],re:R],
      syn:[v,sc:[[syn:[np,_,sg,nom,2],sem:ArgSem1],
                 [syn:[np,_,_,dat,_],sem:ArgSem3],
                 [syn:[np,_,_,acc,_],sem:ArgSem2]]],
      sem:[pred:give,arg1:ArgSem1,arg2:ArgSem2,arg3:ArgSem3]]).

node(37,[phon:[fi:[gibt|R],re:R],
      syn:[v,sc:[[syn:[np,_,sg,nom,3],sem:ArgSem1],
                 [syn:[np,_,_,dat,_],sem:ArgSem3],
                 [syn:[np,_,_,acc,_],sem:ArgSem2]]],
      sem:[pred:give,arg1:ArgSem1,arg2:ArgSem2,arg3:ArgSem3]]).


node(38,[phon:[fi:[schenkst|R],re:R],
      syn:[v,sc:[[syn:[np,_,sg,nom,2],sem:ArgSem1],
                 [syn:[np,_,_,dat,_],sem:ArgSem3],
                 [syn:[np,_,_,acc,_],sem:ArgSem2]]],
      sem:[pred:donate,arg1:ArgSem1,arg2:ArgSem2,arg3:ArgSem3]]).

node(39,[phon:[fi:[schenkt|R],re:R],
      syn:[v,sc:[[syn:[np,_,sg,nom,3],sem:ArgSem1],
                 [syn:[np,_,_,dat,_],sem:ArgSem3],
                 [syn:[np,_,_,acc,_],sem:ArgSem2]]],
      sem:[pred:donate,arg1:ArgSem1,arg2:ArgSem2,arg3:ArgSem3]]).


node(40,[phon:[fi:[schenken|R],re:R],
      syn:[v,sc:[[syn:[np,_,pl,nom,3],sem:ArgSem1],
                 [syn:[np,_,_,dat,_],sem:ArgSem3],
                 [syn:[np,_,_,acc,_],sem:ArgSem2]]],
      sem:[pred:donate,arg1:ArgSem1,arg2:ArgSem2,arg3:ArgSem3]]).

% append
node(41,([],L,L)).
node(42,([F|R],L,[F|R1])) <---
        [node((R,L,R1))].
```

A.4 Test-Umgebung

A.4.1 Test-Aufruf

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                               %
%       file:           test_suite.pl                                           %
%       purpose:        interpreter tests                                       %
%       author:         Sebastian Varges                                        %
%       date:           Wed Feb 19 14:35:13 MET 1997                            %
%       related files:  tests.pl, chart.pl, head.pl, memo_head.pl,              %
%                                                                               %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

test(Mode) :-
        interpreter(Interpreter,Int),
        nl,write('******  Testing '),write(Interpreter),write(' interpreter in '),
        mode(Mode,ModeName),write(ModeName),write(' mode  ******'),nl,
        test(Int,Mode,_),nl,nl,fail.

test(ed,Mode,parse) :-
        test_output(parse_correct),
        fs_parse(N,ParseFS,Wanted),
        test_output(Mode,N),
        findall(ParseFS,proc(ed,ParseFS),Solutions),
        test_output(Mode,ParseFS,Wanted,Solutions),
        fail.

test(ed,Mode,parse) :-
        test_output(parse_false),
        fs_error_parse(N,ParseFS,Wanted),
        test_output(Mode,N),
        findall(ParseFS,proc(ed,ParseFS),Solutions),
        test_output(Mode,ParseFS,Wanted,Solutions),
        fail.

test(Int,Mode,gen) :-
        test_output(gen_correct),
        fs_gen(N,GenFS,Wanted),
        test_output(Mode,N),
        findall(GenFS,proc(Int,GenFS),Solutions),
        test_output(Mode,GenFS,Wanted,Solutions),
        fail.

test(Int,Mode,gen) :-
        test_output(gen_false),
        fs_error_gen(N,GenFS,Wanted),
        test_output(Mode,N),
        findall(GenFS,proc(Int,GenFS),Solutions),
        test_output(Mode,GenFS,Wanted,Solutions),
```

```
        fail.

test(ed,Mode,mixed) :-
        test_output(mixed_correct),
        fs_cons_mixed(N,ConsFS,Wanted),
        test_output(Mode,N),
        findall(ConsFS,proc(ed,ConsFS),Solutions),
        test_output(Mode,ConsFS,Wanted,Solutions),
        fail.

test(ed,Mode,mixed) :-
        test_output(mixed_false),
        fs_incons_mixed(N,InconsFS,Wanted),
        test_output(Mode,N),
        findall(InconsFS,proc(ed,InconsFS),Solutions),
        test_output(Mode,InconsFS,Wanted,Solutions),
        fail.

test(_,_,_).
test(_).

%--------
test_output(TestType) :- test_msg(TestType,Msg),nl,nl,write(Msg),nl,nl.

test_output(v,N) :-  nl,write('Nr. '),write(N),write(':'),tab(4),nl.
test_output(v,FS,Wanted,Solutions) :-
        equal_elements(Wanted,Solutions),verbose_output(FS,Solutions),nl,!.

test_output(s,N) :- write('['), write(N).
test_output(s,_,Wanted,Solutions) :-
        equal_elements(Wanted,Solutions),write(']'),tab(2),!.

test_output(_,FS,_,Solutions) :-
        nl,write('*** error ***'),nl,verbose_output(FS,Solutions),nl.

verbose_output(FS,Solutions) :-
        write('Input:   '),write(FS),nl,
        write('Output:  '),write_solutions(Solutions).

write_solutions([]).
write_solutions([F|R]) :- write(F),nl,tab(9),write_solutions(R).

%---------
mode(v,verbose).
mode(s,silent).

interpreter('head-driven',hd).
interpreter('memoized head-driven',mhd).
interpreter('Earley deduction',ed).
```

```
test_msg(parse_correct,'%%% Parsing correct phrases:').
test_msg(parse_false,'%%% Parsing incorrect phrases:').
test_msg(gen_correct,'%%% Generating with consistent feature structures:').
test_msg(gen_false,'%%% Generating with inconsistent feature structures:').
test_msg(mixed_correct,'%%% Completing consistent feature structures:').
test_msg(mixed_false,'%%% Trying to complete inconsistent feature structures:').

%---------
equal_elements([],[]).
equal_elements([F|R],List) :-
        delete_one(F,List,List1),
        equal_elements(R,List1).

delete_one(F,[F|R],R).
delete_one(F,[E|R],[E|R1]) :-
        delete_one(F,R,R1).
```

A.4.2 Testsätze

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                            %
%       file:           tests.pl                                             %
%       purpose:        interpreter tests                                    %
%       author:         Sebastian Varges                                     %
%       date:           Wed Feb 19 14:35:13 MET 1997                         %
%       related files:  test_suite.pl                                        %
%                                                                            %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%--- correct parsing examples

fs_parse('0a',[phon:[fi:[petra],re:[]],_,_],
        [[phon:[fi:[petra],re:[]],syn:[np,fem,sg,_B,3],sem:[cont:petra,quant:sg]],
         [phon:[fi:[petra],re:[]],syn:[pn,fem,sg,_A,3],sem:[cont:petra,quant:sg]]]).

fs_parse('0b',[phon:[fi:[gibst],re:[]],_,_],
        [[phon:[fi:[gibst],re:[]],syn:[vp,sc:[[syn:[np,_P,sg,nom,2],sem:_O],
                [syn:[np,_N,_M,dat,3],sem:_L],[syn:[np,_K,_J,acc,3],sem:_I]]],
          sem:[pred:give,arg1:_O,arg2:_I,arg3:_L]],
         [phon:[fi:[gibst],re:[]],syn:[v,sc:[[syn:[np,_H,sg,nom,2],sem:_G],
                [syn:[np,_F,_E,dat,3],sem:_D],[syn:[np,_C,_B,acc,3],sem:_A]]],
          sem:[pred:give,arg1:_G,arg2:_A,arg3:_D]]]).

fs_parse('1a',[phon:[fi:[die,frau],re:[]],syn:[np|_],_],
        [[phon:[fi:[die,frau],re:[]],syn:[np,fem,sg,nom,3],sem:[cont:woman,quant:sg]],
         [phon:[fi:[die,frau],re:[]],syn:[np,fem,sg,acc,3],sem:[cont:woman,quant:sg]]]).

fs_parse('1b',[phon:[fi:[die,frauen],re:[]],syn:[np|_],_],
        [[phon:[fi:[die,frauen],re:[]],syn:[np,fem,pl,nom,3],sem:[cont:woman,quant:pl]],
         [phon:[fi:[die,frauen],re:[]],syn:[np,fem,pl,acc,3],sem:[cont:woman,quant:pl]]]).

fs_parse('1c',[phon:[fi:[liebt,die,frauen],re:[]],syn:[vp|_],_],
        [[phon:[fi:[liebt,die,frauen],re:[]],syn:[vp,sc:[[syn:[np,_B,sg,nom,3],sem:_A]]],
                sem:[pred:love,arg1:_A,arg2:[cont:woman,quant:pl]]]]).


fs_parse('2a',[phon:[fi:[der,mann,schlaeft],re:[]],syn:[s],_],
        [[phon:[fi:[der,mann,schlaeft],re:[]],syn:[s],
          sem:[pred:sleep,arg1:[cont:man,quant:sg]]]]).

fs_parse('2b',[phon:[fi:[du,schlaefst],re:[]],syn:[s],_],
        [[phon:[fi:[du,schlaefst],re:[]],syn:[s],
          sem:[pred:sleep,arg1:[cont:you,quant:sg]]]]).


fs_parse('3a',[phon:[fi:[peter,sieht,petra],re:[]],syn:[s],_],
        [[phon:[fi:[peter,sieht,petra],re:[]],syn:[s],
          sem:[pred:see,arg1:[cont:peter,quant:sg],arg2:[cont:petra,quant:sg]]]]).
```

```
fs_parse('3b',[phon:[fi:[du,siehst,die,frau],re:[]],syn:[s],_],
        [[phon:[fi:[du,siehst,die,frau],re:[]],syn:[s],
          sem:[pred:see,arg1:[cont:you,quant:sg],arg2:[cont:woman,quant:sg]]]]).

fs_parse('3c',[phon:[fi:[der,mann,liebt,die,frauen],re:[]],syn:[s],_],
        [[phon:[fi:[der,mann,liebt,die,frauen],re:[]],syn:[s],
          sem:[pred:love,arg1:[cont:man,quant:sg],arg2:[cont:woman,quant:pl]]]]).

fs_parse('3d',[phon:[fi:[der,mann,sieht,petra],re:[]],syn:[s],_],
        [[phon:[fi:[der,mann,sieht,petra],re:[]],syn:[s],
          sem:[pred:see,arg1:[cont:man,quant:sg],arg2:[cont:petra,quant:sg]]]]).


fs_parse('4a',[phon:[fi:[du,schenkst,peter,den,ball],re:[]],syn:[s],_],
        [[phon:[fi:[du,schenkst,peter,den,ball],re:[]],syn:[s],
          sem:[pred:donate,arg1:[cont:you,quant:sg],
               arg2:[cont:ball,quant:sg],arg3:[cont:peter,quant:sg]]]]).

fs_parse('4b',[phon:[fi:[die,frauen,schenken,peter,den,ball],re:[]],syn:[s],_],
        [[phon:[fi:[die,frauen,schenken,peter,den,ball],re:[]],syn:[s],
          sem:[pred:donate,arg1:[cont:woman,quant:pl],
               arg2:[cont:ball,quant:sg],arg3:[cont:peter,quant:sg]]]]).

fs_parse('4c',[phon:[fi:[du,gibst,peter,den,ball],re:[]],syn:[s],_],
        [[phon:[fi:[du,gibst,peter,den,ball],re:[]],syn:[s],
          sem:[pred:give,arg1:[cont:you,quant:sg],
               arg2:[cont:ball,quant:sg],arg3:[cont:peter,quant:sg]]]]).

fs_parse('4d',[phon:[fi:[schenkt,peter,den,ball],re:[]],syn:[vp|_],_],
        [[phon:[fi:[schenkt,peter,den,ball],re:[]],
          syn:[vp,sc:[[syn:[np,_B,sg,nom,3],sem:_A]]],
          sem:[pred:donate,arg1:_A,arg2:[cont:ball,quant:sg],
               arg3:[cont:peter,quant:sg]]]]).

fs_parse('4e',[phon:[fi:[schenken,peter,den,ball],re:[]],syn:[vp|_],_],
        [[phon:[fi:[schenken,peter,den,ball],re:[]],
          syn:[vp,sc:[[syn:[np,_B,pl,nom,3],sem:_A]]],
          sem:[pred:donate,arg1:_A,arg2:[cont:ball,quant:sg],
               arg3:[cont:peter,quant:sg]]]]).

fs_parse('4f',[phon:[fi:[peter,gibt,petra,petra],re:[]],syn:[s],_],
        [[phon:[fi:[peter,gibt,petra,petra],re:[]],syn:[s],
          sem:[pred:give,arg1:[cont:peter,quant:sg],arg2:[cont:petra,quant:sg],
               arg3:[cont:petra,quant:sg]]]]).


%--- incorrect parsing examples
fs_error_parse('10a',[phon:[fi:[petra],re:[]],syn:[v|_],_],[]).
fs_error_parse('10b',[phon:[fi:[der],re:[der]],_,_],[]).
```

```
fs_error_parse('11a',[phon:[fi:[die,mann],re:[]],syn:[np],_],[]).

fs_error_parse('12a',[phon:[fi:[der,mann,schlaefst],re:[]],syn:[s],_],[]).
fs_error_parse('12b',[phon:[fi:[du,schlaeft],re:[]],syn:[s],_],[]).
fs_error_parse('12c',[phon:[fi:[der,frau,schlaeft],re:[]],syn:[s],_],[]).

fs_error_parse('13a',[phon:[fi:[der,mann,traeumt,den,ball],re:[]],syn:[s],_],[]).
fs_error_parse('13b',[phon:[fi:[der,mann,sieht,der,ball],re:[]],syn:[s],_],[]).
fs_error_parse('13c',[phon:[fi:[du,sieht,den,ball],re:[]],syn:[s],_],[]).
fs_error_parse('13d',[phon:[fi:[die,frau,sieht],re:[]],syn:[s],_],[]).

fs_error_parse('14a',[phon:[fi:[du,schenkt,peter,den,ball],re:[]],syn:[s],_],[]).
fs_error_parse('14b',[phon:[fi:[du,schenkst,den,ball],re:[]],_,_],[]).
fs_error_parse('14c',[phon:[fi:[du,schenkst,peter],re:[]],_,_],[]).
fs_error_parse('14d',[phon:[fi:[der,frau,schenkt,peter,den,ball],re:[]],syn:[s],_],[]).


%--- correct generation examples
fs_gen('20a',[_,syn:[pn,fem,sg,_A,3],sem:[cont:petra,quant:sg]],
        [[phon:[fi:[petra|_],re:_],syn:[pn,fem,sg,_A,3],sem:[cont:petra,quant:sg]]]).

%  more syntactic contraints than category
fs_gen('21a',[_,syn:[np,fem,_,_,3],sem:[cont:petra,quant:sg]],
        [[phon:[fi:[petra|_],re:_],syn:[np,fem,sg,_B,3],sem:[cont:petra,quant:sg]]]).

fs_gen('21b',[_,syn:[np|_],sem:[cont:woman,quant:sg]],
        [[phon:[fi:[die,frau|_684],re:_684],syn:[np,fem,sg,nom,3],
          sem:[cont:woman,quant:sg]],
         [phon:[fi:[die,frau|_684],re:_684],syn:[np,fem,sg,acc,3],
          sem:[cont:woman,quant:sg]]]).


fs_gen('22a',[_,syn:[s],sem:[pred:sleep,arg1:[cont:man,quant:sg]]],
        [[phon:[fi:[der,mann,schlaeft|_],re:_],syn:[s],
          sem:[pred:sleep,arg1:[cont:man,quant:sg]]]]).

fs_gen('22b',[_,syn:[s],sem:[pred:sleep,arg1:[cont:man,quant:pl]]],
        [[phon:[fi:[die,maenner,schlafen|_],re:_],syn:[s],
          sem:[pred:sleep,arg1:[cont:man,quant:pl]]]]).

fs_gen('22c',[_,syn:[s],sem:[pred:sleep,arg1:[cont:you,quant:sg]]],
        [[phon:[fi:[du,schlaefst|_],re:_],syn:[s],
          sem:[pred:sleep,arg1:[cont:you,quant:sg]]]]).

fs_gen('22d',[phon:[_,re:[]],syn:[s],sem:[pred:dream,arg1:[cont:peter,quant:sg]]],
        [[phon:[fi:[peter,traeumt|_],re:[]],syn:[s],
          sem:[pred:dream,arg1:[cont:peter,quant:sg]]]]).
```

```
fs_gen('23a',[_,syn:[s],sem:[pred:see,arg1:[cont:peter,quant:sg],
                             arg2:[cont:petra,quant:sg]]],
        [[phon:[fi:[peter,sieht,petra|_],re:_],syn:[s],
          sem:[pred:see,arg1:[cont:peter,quant:sg],
               arg2:[cont:petra,quant:sg]]]]).

fs_gen('23b',[_,syn:[s],sem:[pred:see,arg1:[cont:you,quant:sg],
                             arg2:[cont:woman,quant:sg]]],
        [[phon:[fi:[du,siehst,die,frau],_],syn:[s],
          sem:[pred:see,arg1:[cont:you,quant:sg],
               arg2:[cont:woman,quant:sg]]]]).

fs_gen('23c',[_,syn:[s], sem:[pred:love,arg1:[cont:man,quant:sg],
                              arg2:[cont:woman,quant:pl]]],
        [[phon:[fi:[der,mann,liebt,die,frauen],_],syn:[s],
          sem:[pred:love,arg1:[cont:man,quant:sg],
               arg2:[cont:woman,quant:pl]]]]).

fs_gen('23d',[_,syn:[s],sem:[pred:see,arg1:[cont:man,quant:sg],
                             arg2:[cont:petra,quant:sg]]],
        [[phon:[fi:[der,mann,sieht,petra|_],re:_],syn:[s],
          sem:[pred:see,arg1:[cont:man,quant:sg],
               arg2:[cont:petra,quant:sg]]]]).

fs_gen('24a',[_,syn:[s],sem:[pred:donate,arg1:[cont:you,quant:sg],
                  arg2:[cont:ball,quant:sg],arg3:[cont:peter,quant:sg]]],
        [[phon:[fi:[du,schenkst,peter,den,ball|_2162],re:_2162],syn:[s],
          sem:[pred:donate,arg1:[cont:you,quant:sg],arg2:[cont:ball,quant:sg],
               arg3:[cont:peter,quant:sg]]]]).

fs_gen('24b',[_,syn:[s],sem:[pred:donate,arg1:[cont:woman,quant:pl],
                             arg2:[cont:ball,quant:sg],arg3:[cont:peter,quant:sg]]],
        [[phon:[fi:[die,frauen,schenken,peter,den,ball|_2162],re:_2162],syn:[s],
          sem:[pred:donate,arg1:[cont:woman,quant:pl],arg2:[cont:ball,quant:sg],
               arg3:[cont:peter,quant:sg]]]]).

fs_gen('24c',[_,syn:[s],sem:[pred:give,arg1:[cont:you,quant:sg],
                    arg2:[cont:ball,quant:sg],arg3:[cont:peter,quant:sg]]],
        [[phon:[fi:[du,gibst,peter,den,ball|_2162],re:_2162],syn:[s],
                 sem:[pred:give,arg1:[cont:you,quant:sg],
                      arg2:[cont:ball,quant:sg],arg3:[cont:peter,quant:sg]]]]).

fs_gen('24d',[_,syn:[s],sem:[pred:give,arg1:[cont:peter,quant:sg],
                             arg2:[cont:petra,quant:sg],
                             arg3:[cont:petra,quant:sg]]],
        [[phon:[fi:[peter,gibt,petra,petra|_2162],re:_2162],syn:[s],
          sem:[pred:give,arg1:[cont:peter,quant:sg],
               arg2:[cont:petra,quant:sg],arg3:[cont:petra,quant:sg]]]]).
```

```
%--- incorrect generation examples (fully instantiated semantics)
% numerus incompatible
fs_error_gen('30a',[_,syn:[pn,fem,pl,_B,3],sem:[cont:petra,quant:sg]],[]).

% to sleep with two comps
fs_error_gen('30b',[_,syn:[v|_],sem:[pred:sleep,arg1:[cont:peter,quant:sg],
                                     arg2:[cont:peter,quant:sg]]],[]).

% genus incompatible
fs_error_gen('31a',[_,syn:[np,masc,sg,_,3],sem:[cont:petra,quant:sg]],[]).

% petra in plural
fs_error_gen('31b',[_,syn:[np,fem,pl,_,3],sem:[cont:petra,quant:pl]],[]).

% to sleep with two comps
fs_error_gen('31c',[_,syn:[vp|_],sem:[pred:sleep,arg1:[cont:peter,quant:sg],
                                      arg2:[cont:peter,quant:sg]]],[]).

% wrong category
fs_error_gen('32a',[_,syn:[np|_],sem:[pred:sleep,arg1:[cont:man,quant:sg]]],[]).

% missing argument
fs_error_gen('32b',[_,syn:[s],sem:[pred:love,arg1:[cont:man,quant:sg]]],[]).

% vp with empty subcat list
fs_error_gen('32c',[_,syn:[vp,sc:[]|_],sem:[pred:sleep,arg1:[cont:man,quant:sg]]],[]).

% petra in plural
fs_error_gen('33a',[_,syn:[s],sem:[pred:see,arg1:[cont:peter,quant:sg],
                                   arg2:[cont:petra,quant:pl]]],[]).

% peter in plural
fs_error_gen('33b',[_,syn:[s],sem:[pred:see,arg1:[cont:peter,quant:pl],
                                   arg2:[cont:petra,quant:sg]]],[]).

% to sleep with two comps
fs_error_gen('33c',[_,syn:[s],sem:[pred:sleep,arg1:[cont:peter,quant:sg],
                                   arg2:[cont:petra,quant:sg]]],[]).

% ball in dative not in grammar.pl
fs_error_gen('33d',[_,syn:[s],sem:[pred:donate,arg1:[cont:you,quant:sg],
                                   arg2:[cont:ball,quant:sg],
                                   arg3:[cont:ball,quant:sg]]],[]).

% to see with tree arguments
fs_error_gen('33e',[_,syn:[s],sem:[pred:see,arg1:[cont:you,quant:sg],
                                   arg2:[cont:ball,quant:sg],
                                   arg3:[cont:petra,quant:sg]]],[]).

% to sleep with tree arguments
```

```
fs_error_gen('33f',[_,syn:[s],sem:[pred:sleep,arg1:[cont:you,quant:sg],
                                   arg2:[cont:ball,quant:sg],
                                   arg3:[cont:petra,quant:sg]]],[]).


%--- consistent mixed input feature structures
% only syntax underspecified
fs_cons_mixed('40a',[phon:[fi:[petra],re:[]],syn:[pn|_],sem:[cont:petra,quant:sg]],
        [[phon:[fi:[petra],re:[]],syn:[pn,fem,sg,_,3],sem:[cont:petra,quant:sg]]]).

% petra as pn and np
fs_cons_mixed('41a',[phon:[fi:[petra|_348],re:_344],syn:_339,
                     sem:[cont:_329,quant:sg]],
        [[phon:[fi:[petra|_420],re:_420],syn:[np,fem,sg,_401,3],
          sem:[cont:petra,quant:sg]],
         [phon:[fi:[petra|_478],re:_478],syn:[pn,fem,sg,_459,3],
          sem:[cont:petra,quant:sg]]]).

% phon and sem underspecified
fs_cons_mixed('42a',[phon:[fi:[der,_,schlaeft],re:[]],syn:[s],
                     sem:[pred:_,arg1:[cont:man,quant:_]]],
        [[phon:[fi:[der,mann,schlaeft],re:[]],syn:[s],
          sem:[pred:sleep,arg1:[cont:man,quant:sg]]]]).

fs_cons_mixed('42b',[_,syn:[s],sem:[_,arg1:[cont:man,quant:pl]]],
        [[phon:[fi:[die,maenner,schlafen|_],_],syn:[s],
          sem:[pred:sleep,arg1:[cont:man,quant:pl]]]]).

fs_cons_mixed('42c',[_,syn:[s],sem:[_,arg1:[cont:man,quant:sg]]],
        [[phon:[fi:[der,mann,traeumt|_],_],syn:[s],
          sem:[pred:dream,arg1:[cont:man,quant:sg]]],
         [phon:[fi:[der,mann,schlaeft|_],_],syn:[s],
          sem:[pred:sleep,arg1:[cont:man,quant:sg]]]]).

% phon and sem underspecified
fs_cons_mixed('42d',[phon:[fi:[die,_,schlafen|_],re:_],syn:[s],sem:[_:sleep|_]],
        [[phon:[fi:[die,maenner,schlafen|_],re:_],syn:[s],
          sem:[pred:sleep,arg1:[cont:man,quant:pl]]],
         [phon:[fi:[die,frauen,schlafen|_],re:_],syn:[s],
          sem:[pred:sleep,arg1:[cont:woman,quant:pl]]]]).

fs_cons_mixed('42e',[phon:[fi:[du,_,die,frau],_],syn:_,
          sem:[pred:see,arg1:[cont:you,quant:_],arg2:[cont:_,quant:sg]]],
        [[phon:[fi:[du,siehst,die,frau|_],_],syn:[s],
          sem:[pred:see,arg1:[cont:you,quant:sg],arg2:[cont:woman,quant:sg]]]]).

fs_cons_mixed('42f',[phon:[fi:[der,mann|_],_],syn:[s],
                     sem:[pred:love,_,arg2:[cont:woman,quant:pl]]],
        [[phon:[fi:[der,mann,liebt,die,frauen],re:[]],syn:[s],
          sem:[pred:love,arg1:[cont:man,quant:sg],
```

```
              arg2:[cont:woman,quant:pl]]]]).

fs_cons_mixed('42g',[_,syn:[s],sem:[_,arg1:[cont:peter,quant:sg],
                                      arg2:[cont:petra,quant:sg]]],
        [[phon:[fi:[peter,liebt,petra|_1302],re:_1302],syn:[s],
          sem:[pred:love,arg1:[cont:peter,quant:sg],
                arg2:[cont:petra,quant:sg]]],
         [phon:[fi:[peter,sieht,petra|_1302],re:_1302],syn:[s],
          sem:[pred:see,arg1:[cont:peter,quant:sg],
                arg2:[cont:petra,quant:sg]]]]).

% als sentences for to sleep
fs_cons_mixed('42h',[_,syn:[s],sem:[pred:sleep|_]],
        [[phon:[fi:[der,mann,schlaeft|_],re:_],syn:[s],
          sem:[pred:sleep,arg1:[cont:man,quant:sg]]],
         [phon:[fi:[der,ball,schlaeft|_],re:_],syn:[s],
          sem:[pred:sleep,arg1:[cont:ball,quant:sg]]],
         [phon:[fi:[die,frau,schlaeft|_],re:_],syn:[s],
          sem:[pred:sleep,arg1:[cont:woman,quant:sg]]],
         [phon:[fi:[peter,schlaeft|_],re:_],syn:[s],
          sem:[pred:sleep,arg1:[cont:peter,quant:sg]]],
         [phon:[fi:[petra,schlaeft|_],re:_],syn:[s],
          sem:[pred:sleep,arg1:[cont:petra,quant:sg]]],
         [phon:[fi:[die,maenner,schlafen|_],_],syn:[s],
          sem:[pred:sleep,arg1:[cont:man,quant:pl]]],
         [phon:[fi:[die,frauen,schlafen|_],_],syn:[s],
          sem:[pred:sleep,arg1:[cont:woman,quant:pl]]],
         [phon:[fi:[du,schlaefst|_],_],syn:[s],
          sem:[pred:sleep,arg1:[cont:you,quant:sg]]]]).

fs_cons_mixed('42i',[phon:[fi:[du,gibst,_,_,ball],_],_,
                    sem:[_,arg1:[cont:you,quant:sg],
                           arg2:[_:ball,quant:sg],arg3:[cont:peter,_]]],
        [[phon:[fi:[du,gibst,peter,den,ball],_],syn:[s],
          sem:[pred:give,arg1:[cont:you,quant:sg],
              arg2:[cont:ball,quant:sg],arg3:[cont:peter,quant:sg]]]]).

fs_cons_mixed('42j',[_,syn:[s],sem:[_,arg1:[cont:peter,quant:sg],
                                      arg2:[cont:ball,_],arg3:[cont:petra,_]]],
        [[phon:[fi:[peter,gibt,petra,den,ball|_2111],re:_2111],syn:[s],
          sem:[pred:give,arg1:[cont:peter,quant:sg],arg2:[cont:ball,quant:sg],
              arg3:[cont:petra,quant:sg]]],
         [phon:[fi:[peter,schenkt,petra,den,ball|_2111],re:_2111],syn:[s],
          sem:[pred:donate,arg1:[cont:peter,quant:sg],arg2:[cont:ball,quant:sg],
              arg3:[cont:petra,quant:sg]]]]).

fs_cons_mixed('42k',[phon:[fi:[peter,_,petra,petra|_],re:_],syn:[s],sem:[pred:donate|_]],
        [[phon:[fi:[peter,schenkt,petra,petra|_],re:_],syn:[s],
          sem:[pred:donate,arg1:[cont:peter,quant:sg],
              arg2:[cont:petra,quant:sg],arg3:[cont:petra,quant:sg]]]]).
```

```
/*
% 42l + 42m: phon rest list instantiated: only works if prediction in earley deduction
% is always called (even if active_completion/5 succeeds); see file "charts.pl".
fs_cons_mixed('42l',[phon:[fi:[du,_,die,frau,heute],re:[heute]],syn:_,
          sem:[pred:see,arg1:[cont:you,quant:_],arg2:[cont:_,quant:sg]]],
        [[phon:[fi:[du,siehst,die,frau,heute],re:[heute]],syn:[s],
          sem:[pred:see,arg1:[cont:you,quant:sg],arg2:[cont:woman,quant:sg]]]]).

fs_cons_mixed('42m',[phon:[fi:[der,mann|_],re:[]],syn:[s],
                     sem:[pred:love,_,arg2:[cont:woman,quant:pl]]],
        [[phon:[fi:[der,mann,liebt,die,frauen],re:[]],syn:[s],
          sem:[pred:love,arg1:[cont:man,quant:sg],arg2:[cont:woman,quant:pl]]]]).
*/

%--- inconsistent mixed input feature structures

fs_incons_mixed('51a',[phon:[fi:[die,_,schlaeft],re:[]],syn:[s],
                       sem:[pred:_,arg1:[cont:man,quant:_]]],[]).

fs_incons_mixed('51b',[phon:[fi:[der,traeumt|_],_],syn:[s],
                       sem:[_,arg1:[cont:man,quant:sg]]],[]).

fs_incons_mixed('52a',[phon:[fi:[der,mann,_,die,frauen],re:[]],
                       syn:[s],sem:[pred:love,arg1:[cont:man,quant:pl],
                                    arg2:[cont:woman,quant:pl]]],[]).

fs_incons_mixed('52b',[phon:[fi:[peter,sieht,petra|_],re:_],syn:[s],
                       sem:[pred:see,arg1:[cont:peter,quant:sg],
                            arg2:[cont:petra,quant:pl]]],[]).

fs_incons_mixed('52c',[phon:[fi:[peter,sieht,petra|_],re:_],syn:[s],
                       sem:[pred:give,arg1:[cont:peter,quant:sg],
                            arg2:[cont:petra,quant:sg]]],[]).

fs_incons_mixed('52d',[phon:[fi:[peter,sieht,petra|_],re:_],syn:[s],
                       sem:[pred:sleep,arg1:[cont:peter,quant:sg],
                            arg2:[cont:petra,quant:sg]]],[]).

fs_incons_mixed('53a',[phon:[fi:[_,schenkt,petra,den,ball],re:[]],_,
                       sem:[pred:donate,arg1:[cont:man,quant:sg],
                            arg2:[cont:ball,quant:sg],_]],[]).

fs_incons_mixed('53b',[phon:[fi:[schenken,peter,den,ball],re:[]],
                       syn:[vp,sc:[[syn:[np,_,_,acc,3],_]]],
                       sem:[pred:donate,arg1:_A,arg2:[cont:ball,quant:sg],_]],[]).
```

A.4.3 Load-File

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                            %
%       file:           top.pl                                               %
%       purpose:        load file, Prolog and grammar declarations           %
%       author:         Sebastian Varges                                     %
%       date:           Wed Feb 12 14:44:24 MET 1997                         %
%       related files:  chart.pl, aux.pl, head.pl, memo_head.pl, grammar.pl %
%                       test_suite.pl, tests.pl                              %
%                                                                            %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% missing chain/2, non_chain/1 and table_item/4 predicates are treated as
% simple failures:
:- prolog_flag(unknown,_,fail).

:- op(1200,xfx,<---).
:- ensure_loaded(head).
:- ensure_loaded(memo_head).
:- ensure_loaded(chart).
:- ensure_loaded(aux).
:- ensure_loaded(grammar).
:- ensure_loaded(test_suite).
:- ensure_loaded(tests).
:- use_module(library(terms)).
%------------ interpreter calls -------------
proc(hd,FS) :- head_proc(FS).
proc(mhd,FS) :- memo_hd_proc(FS).
proc(ed,FS) :- earley_proc(FS,all,df,off,off).

%------ (compilable) grammar information ------
% all grammar rules as chain rules
chain_rule([_,syn:[s|_],_],1,left).
chain_rule([_,syn:[vp|_],_],2,right).
chain_rule([_,syn:[vp|_],_],3,unary).
chain_rule([_,syn:[np|_],_],4,left).
chain_rule([_,syn:[np|_],_],5,unary).

head_link([_,syn:[s|_],Sem],[_,syn:[vp|_],Sem]).
head_link([_,syn:[s|_],Sem],[_,syn:[v|_],Sem]).
head_link([_,syn:[vp|_],Sem],[_,syn:[v|_],Sem]).
head_link([_,syn:[np|_],Sem],[_,syn:[pn|_],Sem]).
head_link([_,syn:[np|R],Sem],[_,syn:[n|R],Sem]). % coindexing syntaktic features
head_link(X,X).
/*
% only vp-complement rule 2 as chain rule
chain_rule([_,syn:[vp|_],_],2,right).

nonchain_rule([_,syn:[s|_],_],1,rule).
nonchain_rule([_,syn:[vp|_],_],3,rule).
```

```
nonchain_rule([_,syn:[np|_],_],4,rule).
nonchain_rule([_,syn:[np|_],_],5,rule).

head_link([_,syn:[vp|_],Sem],[_,syn:[vp|_],Sem]).
head_link(X,X) :- \+ X = [_,syn:[vp|_],_].
*/

% declarations for lexical entries
nonchain_rule([_,syn:[det|_],_],6,lex).
nonchain_rule([_,syn:[det|_],_],7,lex).
nonchain_rule([_,syn:[det|_],_],8,lex).
nonchain_rule([_,syn:[det|_],_],9,lex).
nonchain_rule([_,syn:[det|_],_],10,lex).
nonchain_rule([_,syn:[det|_],_],11,lex).
nonchain_rule([_,syn:[det|_],_],12,lex).
nonchain_rule([_,syn:[det|_],_],13,lex).

nonchain_rule([_,syn:[n|_],_],14,lex).
nonchain_rule([_,syn:[n|_],_],15,lex).
nonchain_rule([_,syn:[n|_],_],16,lex).
nonchain_rule([_,syn:[n|_],_],17,lex).
nonchain_rule([_,syn:[n|_],_],18,lex).
nonchain_rule([_,syn:[n|_],_],19,lex).
nonchain_rule([_,syn:[n|_],_],20,lex).
nonchain_rule([_,syn:[n|_],_],21,lex).
nonchain_rule([_,syn:[n|_],_],22,lex).
nonchain_rule([_,syn:[n|_],_],23,lex).

nonchain_rule([_,syn:[pn|_],_],24,lex).
nonchain_rule([_,syn:[pn|_],_],25,lex).
nonchain_rule([_,syn:[pn|_],_],26,lex).
nonchain_rule([_,syn:[pn|_],_],27,lex).

nonchain_rule([_,syn:[v|_],_],28,lex).
nonchain_rule([_,syn:[v|_],_],29,lex).
nonchain_rule([_,syn:[v|_],_],30,lex).
nonchain_rule([_,syn:[v|_],_],31,lex).
nonchain_rule([_,syn:[v|_],_],32,lex).
nonchain_rule([_,syn:[v|_],_],33,lex).
nonchain_rule([_,syn:[v|_],_],34,lex).
nonchain_rule([_,syn:[v|_],_],35,lex).
nonchain_rule([_,syn:[v|_],_],36,lex).
nonchain_rule([_,syn:[v|_],_],37,lex).
nonchain_rule([_,syn:[v|_],_],38,lex).
nonchain_rule([_,syn:[v|_],_],39,lex).
nonchain_rule([_,syn:[v|_],_],40,lex).

% for append in grammar.pl
nonchain_rule((_,_,_),41,lex).
nonchain_rule((_,_,_),42,rule).
```

A.4.4 Beispiel-Aufrufe

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                           %
%       file:           examples.pl                                         %
%       purpose:        example interpreter calls                           %
%       author:         Sebastian Varges                                    %
%       date:           Thu Feb 20 12:36:18 MET 1997                        %
%       related files:  top.pl, head.pl, memo_head.pl, chart.pl, aux.pl,    %
%                       grammar.pl, test_suite.pl, tests.pl                 %
%                                                                           %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
/*
% loading the system
| ?- ensure_loaded(top).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Generation with head driven interpreter
| ?- proc(hd,[Phon,syn:[s],sem:[pred:sleep,arg1:[cont:man,quant:sg]]]).
Phon = phon:[fi:[der,mann,schlaeft|_A],re:_A] ? ;
no

% "Parsing": correct but inefficient
proc(hd,[phon:[fi:[der,mann,sieht,petra],re:[]],syn:[s],Sem]).
Sem = sem:[pred:see,arg1:[cont:man,quant:sg],arg2:[cont:petra,quant:sg]] ? ;
no

% mixed input
| ?- proc(hd,[phon:[fi:[die,A,schlafen|B],re:C],syn:[s],sem:[D:sleep|E]]).

A = maenner,
C = B,
D = pred,
E = [arg1:[cont:man,quant:pl]] ? ;

A = frauen,
C = B,
D = pred,
E = [arg1:[cont:woman,quant:pl]] ? ;
no

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Generation with memoized head driven interpreter
| ?- proc(mhd,[Phon,syn:[s],sem:[pred:donate,arg1:[cont:woman,quant:pl],
               arg2:[cont:ball,quant:sg],arg3:[cont:peter,quant:sg]]]).
Phon = phon:[fi:[die,frauen,schenken,peter,den,ball|_A],re:_A] ? ;
no
% getting memoized results from database
| ?- memo_result(Result).
Result = [phon:[fi:[peter|_A],re:_A],syn:[np,masc,sg,dat,3],
          sem:[cont:peter,quant:sg]] ? ;
```

```
Result = [phon:[fi:[den,ball|_A],re:[ball|_A]],syn:[det,masc,sg,acc],sem:[quant:sg]] ? ;
Result = [phon:[fi:[den,ball|_A],re:_A],syn:[np,masc,sg,acc,3],
          sem:[cont:ball,quant:sg]] ? ;
Result = [phon:[fi:[die,frauen|_A],re:[frauen|_A]],syn:[det,fem,pl,nom],
          sem:[quant:pl]] ? ;
Result = [phon:[fi:[die,frauen,schenken,peter,den,ball|_A],re:[schenken,peter,den,
                    ball|_A]],syn:[np,fem,pl,nom,3],sem:[cont:woman,quant:pl]] ? ;
Result = [phon:[fi:[die,frauen,schenken,peter,den,ball|_A],re:_A],syn:[s],
          sem:[pred:donate,arg1:[cont:woman,quant:pl],arg2:[cont:ball,quant:sg],
               arg3:[cont:peter,quant:sg]]] ? ;
no

% "Parsing": stored results demonstrate inefficiancy of search
%            (but algorithm is correct)
| ?- proc(mhd,[phon:[fi:[der,mann,schlaeft],re:[]],syn:[s],Sem]).
Sem = sem:[pred:sleep,arg1:[cont:man,quant:sg]] ? ;
no
| ?- memo_result(Result).
Result = [phon:[fi:[peter,traeumt|_A],re:[traeumt|_A]],syn:[np,masc,sg,nom,3],
          sem:[cont:peter,quant:sg]] ? ;
Result = [phon:[fi:[petra,traeumt|_A],re:[traeumt|_A]],syn:[np,fem,sg,nom,3],
          sem:[cont:petra,quant:sg]] ? ;
Result = [phon:[fi:[der,mann|_A],re:[mann|_A]],syn:[det,masc,sg,nom],
          sem:[quant:sg]] ? ;
Result = [phon:[fi:[der,mann,traeumt|_A],re:[traeumt|_A]],syn:[np,masc,sg,nom,3],
          sem:[cont:man,quant:sg]] ? ;
Result = [phon:[fi:[der,ball|_A],re:[ball|_A]],syn:[det,masc,sg,nom],sem:[quant:sg]] ? ;
Result = [phon:[fi:[der,ball,traeumt|_A],re:[traeumt|_A]],syn:[np,masc,sg,nom,3],
          sem:[cont:ball,quant:sg]] ? ;
Result = [phon:[fi:[die,frau|_A],re:[frau|_A]],syn:[det,fem,sg,nom],sem:[quant:sg]] ? ;
% ....... all feature structures licensed by grammar

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% top-down Earley deduction
%  "proc(ed,FS)" corresponds to "earley_proc(FS,all,df,off,off)."
%   all: all solutions to be found
%   df:  depth-first search
%   off: trace off
%   off: no display of solutions
%%%%%%%%%%%% Parsing example:
| ?- proc(ed,[phon:[fi:[schenken,peter,den,ball],re:[]],syn:[vp|Syn],Sem]).
Sem = sem:[pred:donate,arg1:_A,arg2:[cont:ball,quant:sg],arg3:[cont:peter,quant:sg]],
Syn = [sc:[[syn:[np,_B,pl,nom,3],sem:_A]]] ? ;
no

% tables/0 shows all table items
% agenda is empty because of exhaustive search
| ?- tables.
********************  agenda items  ********************
```

```
*******************  table items   *******************
%---------------------------------------------
%       Active Item:
   ans([phon:[fi:[schenken,peter,den,ball],re:[]],syn:[vp|_94],_91])   <---
   Selected:
        node([phon:[fi:[schenken,peter,den,ball],re:[]],syn:[vp|_94],_91])
   Grammar Rule Index:  0
%---------------------------------------------
%       Active Item:
   node([phon:[fi:[schenken,peter,den,ball],re:[]],syn:[vp,sc:_124],_120])   <---
   Selected:
        node([phon:[fi:[schenken,peter,den,ball],re:[]],syn:[v,sc:_124],_120])
   Grammar Rule Index:  3

.......... many more items .........

%---------------------------------------------
%        Passive Item:
ans([phon:[fi:[schenken,peter,den,ball],re:[]],syn:[vp,sc:[
     [syn:[np,_181,pl,nom,3],sem:_172]]],
      sem:[pred:donate,arg1:_172,arg2:[cont:ball,quant:sg],arg3:[cont:peter,quant:sg]]]).
   Grammar Rule Index:  [[[[4,7],17],[[[5,24],[2,[3,40]]],2]],0]

% this passive answer item is the reduced start item 0

%%%%%%%%%%% Generation example with underspecified semantics -> two solutions
% bf: breath-first search
| ?- earley_proc([_,syn:[s],sem:[_,arg1:[cont:man,quant:sg]]],all,bf,off,on).

[phon:[fi:[der,mann,traeumt|_15913],re:_15913],syn:[s],
 sem:[pred:dream,arg1:[cont:man,quant:sg]]]
[phon:[fi:[der,mann,schlaeft|_15913],re:_15913],syn:[s],
 sem:[pred:sleep,arg1:[cont:man,quant:sg]]]
% no (more) solutions
yes

%%%%%%%%%%% Parsing with trace
% first: stop after first solution is found
% df:    depth-first search
% on:    trace on
% on:    display of solutions
| ?- earley_proc([phon:[fi:[der,mann,schlaeft],re:[]],syn:[s],_],first,df,on,on).
*************************  new task *************************
%       Active Item:
   ans([phon:[fi:[der,mann,schlaeft],re:[]],syn:[s],_390])   <---
   Selected:
        node([phon:[fi:[der,mann,schlaeft],re:[]],syn:[s],_390])
   Grammar Rule Index:  0

[Options: a = abort, n = nonstop, <rtn> = creep]
```

```
||
% -----------------------------------------------------
% inference rule:      prediction
% reduced new item:
%       Active Item:
   node([phon:[fi:[der,mann,schlaeft],re:[]],syn:[s],sem:_643])   <---
        node([phon:[fi:_662,re:[]],syn:[vp,sc:[[syn:[np|_669],sem:_674]]],sem:_643])
   Selected:
        node([phon:[fi:[der,mann,schlaeft],re:_662],syn:[np|_669],sem:_674])
   Grammar Rule Index:  1
   selection type:  leftmost by default

[Options: a = abort, n = nonstop, <rtn> = creep]
||
***************************  new task ***************************
%       Active Item:
   node([phon:[fi:[der,mann,schlaeft],re:[]],syn:[s],sem:_702])   <---
        node([phon:[fi:_746,re:[]],syn:[vp,sc:[[syn:[np|_738],sem:_734]]],sem:_702])
   Selected:
        node([phon:[fi:[der,mann,schlaeft],re:_746],syn:[np|_738],sem:_734])
   Grammar Rule Index:  1

[Options: a = abort, n = nonstop, <rtn> = creep]
||
% -----------------------------------------------------
% inference rule:      prediction
% reduced new item:
%       Active Item:
   node([phon:[fi:[der,mann,schlaeft],re:_746],syn:[np,_1181,_1183,_1185,_1187],
        sem:[cont:_1193,quant:_1198]])   <---
        node([phon:[fi:_1217,re:_746],syn:[n,_1181,_1183,_1185,_1187],
              sem:[cont:_1193,quant:_1198]])
   Selected:
        node([phon:[fi:[der,mann,schlaeft],re:_1217],syn:[det,_1181,_1183,_1185],
              sem:[quant:_1198]])
   Grammar Rule Index:  4
   selection type:  leftmost by default

[Options: a = abort, n = nonstop, <rtn> = creep]
||
% -----------------------------------------------------
% inference rule:      prediction
% reduced new item:
%       Active Item:
   node([phon:[fi:[der,mann,schlaeft],re:_746],
        syn:[np,_1181,_1183,_1185,_1187],sem:_734])   <---
   Selected:
        node([phon:[fi:[der,mann,schlaeft],re:_746],
              syn:[pn,_1181,_1183,_1185,_1187],sem:_734])
   Grammar Rule Index:  5
```

```
   selection type:  the only element

[Options: a = abort, n = nonstop, <rtn> = creep]
||
***************************  new task ***************************
%        Active Item:
   node([phon:[fi:[der,mann,schlaeft],re:_1252],
         syn:[np,_1241,_1239,_1237,_1235],sem:_1232])   <---
   Selected:
        node([phon:[fi:[der,mann,schlaeft],re:_1252],
              syn:[pn,_1241,_1239,_1237,_1235],sem:_1232])
   Grammar Rule Index:  5
%%%%%% => no application of inference rules
[Options: a = abort, n = nonstop, <rtn> = creep]
||
***************************  new task ***************************
%        Active Item:
   node([phon:[fi:[der,mann,schlaeft],re:_1624],syn:[np,_1656,_1654,_1652,_1607],
         sem:[cont:_1599,quant:_1644]])   <---
        node([phon:[fi:_1667,re:_1624],syn:[n,_1656,_1654,_1652,_1607],
              sem:[cont:_1599,quant:_1644]])
   Selected:
        node([phon:[fi:[der,mann,schlaeft],re:_1667],
              syn:[det,_1656,_1654,_1652],sem:[quant:_1644]])
   Grammar Rule Index:  4

[Options: a = abort, n = nonstop, <rtn> = creep]
||
% ------------------------------------------------------
%  inference rule:       scanning
%  reduced new item:
%        Active Item:
   node([phon:[fi:[der,mann,schlaeft],re:_1624],syn:[np,masc,sg,nom,_1607],
         sem:[cont:_1599,quant:sg]])   <---
   Selected:
        node([phon:[fi:[mann,schlaeft],re:_1624],syn:[n,masc,sg,nom,_1607],
              sem:[cont:_1599,quant:sg]])
   Grammar Rule Index:  [4,6]
   selection type:  the only element
%%%%%% => grammar rule 4 reduced with lex entry 6
[Options: a = abort, n = nonstop, <rtn> = creep]
||
***************************  new task ***************************
%        Active Item:
   node([phon:[fi:[der,mann,schlaeft],re:_2244],syn:[np,masc,sg,nom,_2227],
         sem:[cont:_2219,quant:sg]])   <---
   Selected:
        node([phon:[fi:[mann,schlaeft],re:_2244],syn:[n,masc,sg,nom,_2227],
              sem:[cont:_2219,quant:sg]])
   Grammar Rule Index:  [4,6]
```

```
[Options: a = abort, n = nonstop, <rtn> = creep]
||
% -----------------------------------------------------
%  inference rule:       scanning
%  reduced new item:
%         Passive Item:
node([phon:[fi:[der,mann,schlaeft],re:[schlaeft]],syn:[np,masc,sg,nom,3],
      sem:[cont:man,quant:sg]]).
   Grammar Rule Index:  [[4,6],14]
   selection type:  nothing to select
%%%%%% => scanning of "mann"; result: passive item for np
[Options: a = abort, n = nonstop, <rtn> = creep]
||
***************************  new task ***************************
%         Passive Item:
node([phon:[fi:[der,mann,schlaeft],re:[schlaeft]],syn:[np,masc,sg,nom,3],
      sem:[cont:man,quant:sg]]).
   Grammar Rule Index:  [[4,6],14]

[Options: a = abort, n = nonstop, <rtn> = creep]
||  n   %%%%%%% => nonstop

% first solution:    [phon:[fi:[der,mann,schlaeft],re:[]],syn:[s],
                      sem:[pred:sleep,arg1:[cont:man,quant:sg]]]
yes

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%  test suite
%   testing earley deduction engine in verbose mode with mixed input
| ?- test(ed,v,mixed).
%%%  Completing consistent feature structures:
Nr. 40a:
Input:   [phon:[fi:[petra],re:[]],syn:[pn|_253],sem:[cont:petra,quant:sg]]
Output:  [phon:[fi:[petra],re:[]],syn:[pn,fem,sg,_205,3],sem:[cont:petra,quant:sg]]

Nr. 41a:
Input:   [phon:[fi:[petra|_312],re:_308],syn:_303,sem:[cont:_293,quant:sg]]
Output:  [phon:[fi:[petra|_224],re:_224],syn:[pn,fem,sg,_205,3],
          sem:[cont:petra,quant:sg]]
         [phon:[fi:[petra|_273],re:_273],syn:[np,fem,sg,_254,3],
          sem:[cont:petra,quant:sg]]
........ etc .....

%%%  Trying to complete inconsistent feature structures:
Nr. 51a:
Input:   [phon:[fi:[die,_227,schlaeft],re:[]],syn:[s],
          sem:[pred:_205,arg1:[cont:man,quant:_190]]]
Output:
```

```
Nr. 51b:
Input:   [phon:[fi:[der,traeumt|_220],_217],syn:[s],
          sem:[_203,arg1:[cont:man,quant:sg]]]
Output:
........ etc .....
% => inconsistent input: no solution

% testing interpreters with all test sentences in silent mode
| ?- test(s).
******  Testing head-driven interpreter in silent mode  ******
%%%  Generating with consistent feature structures:
[20a]  [21a]  [21b]  [22a]  [22b]  [22c]  [22d]  [23a]  [23b]  [23c]  [23d]  ...
%%%  Generating with inconsistent feature structures:
[30a]  [30b]  [31a]  [31b]  [31c]  [32a]  [32b]  [32c]  [33a]  [33b]  [33c]  ...

******  Testing memoized head-driven interpreter in silent mode  ******
%%%  Generating with consistent feature structures:
[20a]  [21a]  [21b]  [22a]  [22b]  [22c]  [22d]  [23a]  [23b]  [23c]  [23d]  ...
%%%  Generating with inconsistent feature structures:
[30a]  [30b]  [31a]  [31b]  [31c]  [32a]  [32b]  [32c]  [33a]  [33b]  [33c]  ...

******  Testing Earley deduction interpreter in silent mode  ******
%%%  Parsing correct phrases:
[0a]  [0b]  [1a]  [1b]  [1c]  [2a]  [2b]  [3a]  [3b]  [3c]  [3d]  [4a]  [4b] ...
%%%  Parsing incorrect phrases:
[10a]  [10b]  [11a]  [12a]  [12b]  [12c]  [13a]  [13b]  [13c]  [13d]  [14a]  ...
%%%  Generating with consistent feature structures:
[20a]  [21a]  [21b]  [22a]  [22b]  [22c]  [22d]  [23a]  [23b]  [23c]  [23d]  ...
%%%  Generating with inconsistent feature structures:
[30a]  [30b]  [31a]  [31b]  [31c]  [32a]  [32b]  [32c]  [33a]  [33b]  [33c]  ...
%%%  Completing consistent feature structures:
[40a]  [41a]  [42a]  [42b]  [42c]  [42d]  [42e]  [42f]  [42g]  [42h]  [42i]  ...
%%%  Trying to complete inconsistent feature structures:
[51a]  [51b]  [52a]  [52b]  [52c]  [52d]  [53a]  [53b]
yes
*/
```

A.5 Terminationsprobleme in left-corner-Parsern

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                           %
%       file:           lc.pl                                               %
%       purpose:        to illustrate termination problems of lc parsers    %
%       author:         Sebastian Varges                                    %
%       date:           Tue Sep 10 11:01:05 MET DST 1996                    %
%       related refs:   [Covington 94]                                      %
%                                                                           %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% left-corner parser from [Covington 94]:

parse(C,[Word|S2],S) :-
word(W,Word),
link(W,C),
complete(W,C,S2,S).

parse(C,S2,S) :-
rule(W,[]),
link(W,C),
complete(W,C,S2,S).

parse_list([C|Cs],S1,S) :-
parse(C,S1,S2),
parse_list(Cs,S2,S).

parse_list([],S,S).

complete(C,C,S,S).
complete(W,C,S1,S) :-
rule(P,[W|Rest]),
parse_list(Rest,S1,S2),
complete(P,C,S2,S).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% In [NLP for Prolog programmers(1994,p.162-165)] Covington describes how
% links inferred from a grammar help a left-corner parser to
% handle null constituents if it is not the grammar itself which
% defines an infinite number of parses.

% This simple grammar causes the lc parser to loop
% (the rules are given in the format of [Covington 94]):

rule(s,[x,y]).     % s --> x,y.
rule(y,[x]).       % y --> x.   %% this causes the problem
rule(x,[]).        % x --> 0.

word(x,w1).
word(y,w2).
```

```
% links inferred the grammar:
link(x,s).
link(x,y).          %% this causes the problem
link(X,X).

% This grammar defines five strings: [],[w1],[w2],[w1,w1] and [w1,w2].
% The parser is only able to handle [w2] and [w1,w2]. Parsing
% the other strings (and all ungrammatical ones) causes the parser
% to loop.

% For example, given the goal parse(s,[],[]). the parser makes
% the following steps and does not fail finitely. No backtracking
% is involved:
/*
% example trace:
{trace}
| ?- parse(s,[],[]).
   1  1  Call: parse(s,[],[]) ?   % parse s as empty element
   2  2  Call: rule(_337,[]) ?    % find rule which defines empty element
   2  2  Exit: rule(x,[]) ?       % rule ‘x --> 0’ found
   3  2  Call: link(x,s) ?
   3  2  Exit: link(x,s) ?        % link relation holds
   4  2  Call: complete(x,s,[],[]) ?
   5  3  Call: rule(_1261,[x|_1259]) ? % find rule x is a left corner of
   5  3  Exit: rule(s,[x,y]) ?         % rule ‘s --> x,y’ found
   6  3  Call: parse_list([y],[],_1254) ? % remaining dtr: y
   7  4  Call: parse(y,[],_1830) ?     % parse y as empty element
   8  5  Call: rule(_2041,[]) ?
   8  5  Exit: rule(x,[]) ?       % rule ‘x --> 0’ defines empty element ...
   9  5  Call: link(x,y) ?
   9  5  Exit: link(x,y) ?        % ... and is linked to y
   10  5  Call: complete(x,y,[],_1830) ?
   11  6  Call: rule(_2965,[x|_2963]) ? % find rule x is a left corner of
   11  6  Exit: rule(s,[x,y]) ?         % rule ‘s --> x,y’ found (= 5)
   12  6  Call: parse_list([y],[],_2958) ? % remaining dtr: y (= 6)
   13  7  Call: parse(y,[],_3534) ?     % parse y as empty element (= 7)
   14  8  Call: rule(_3745,[]) ?
   14  8  Exit: rule(x,[]) ?      % rule ‘x --> 0’ defines empty element ... (= 8)
   15  8  Call: link(x,y) ?
   15  8  Exit: link(x,y) ?       % ... and is linked to y (= 9)
   16  8  Call: complete(x,y,[],_3534) ?
   17  9  Call: rule(_4669,[x|_4667]) ?     % find rule x is a left corner of
   17  9  Exit: rule(s,[x,y]) ?             % rule ‘s --> x,y’ found (= 5, = 11)
   18  9  Call: parse_list([y],[],_4662) ?  % remaining dtr: y (= 6, = 12)
   19  10  Call: parse(y,[],_5238) ? n
Prolog interruption (h for help)? a
{Execution aborted}
*/
% A parsed constituent ‘x’ can be used to complete two grammar rules:
```

```
% 's --> x,y' and 'y --> x'. Here the parser always tries to complete
% the first one, so there's always a 'y' left to be parsed. If the ordering of
% these grammar rules is changed, the problem occurs when backtracking is used
% ('?- parse(s,[],[]),fail.' does not fail finitely).
```