Sven Naumann
Hagen Langer

Parsing

Eine Einführung
in die
maschinelle Analyse
natürlicher Sprache

Vorwort

Als wir Ende der 80er Jahre zum erstenmal Lehrveranstaltungen vorbereiteten, die einen Überblick über die aus Sicht der Computerlinguistik wichtigsten Parsingalgorithmen geben sollten, mußten wir feststellen, daß zwar einerseits die Zahl der Aufsätze, die sich direkt oder indirekt mit Fragen des Parsing beschäftigen, ständig wächst und schon fast nicht mehr überschaubar ist, es anderseits aber kaum Lehrbücher gibt, die sich als Grundlage für einführende Veranstaltungen eignen. Natürlich gibt es ganze Reihe von Büchern, die algorithmische Verfahren der Syntaxanalyse unter dem Gesichtspunkt der Entwicklung von Compilern (für Programmiersprachen) diskutieren. Einige fundamentale Unterschiede zwischen natürlichen und formalen Sprachen und den Anforderungen an Parser, die als Teil eines Compilers oder eines natürlichsprachlichen Systems konzipiert werden, lassen diese Bücher aus Sicht der Computerlinguistik nur eingeschränkt geeignet erscheinen. Aus diesem Grund begannen wir, ein Skript zu schreiben, aus dem sich im Verlauf mehrerer Jahre dieses Buch entwickelte. In dieser Zeit ist es durch viele Hände gegangen: Für viele wertvolle Anregungen möchten wir besonders P.Barg, F.Guenthner, R.Posner, I.Renz und J.Schrepp danken. Ohne die Geduld und die Sorgfalt von S.Bauer, J.Falkenberg, B.Grote, H.Kranzdorf, B.Krier-Brandt und C.Schulz wären viele sachliche und orthographische Fehler unentdeckt geblieben.

Alle die in diesem Buch abgedruckten Programme und weitere, die aus Platzgründen nicht aufgenommen werden konnten, sind über den ftp–Server der Universität Trier erhältlich (Adresse: *ftp.uni-trier.de* / User-Name: *anonymous*). Wem dieser Weg verstellt ist, kann sie gegen Einsenden einer formatierten Diskette (3 1/2 Zoll), eines frankierten Rückumschlags und einer Bearbeitungsgebühr von 10 .- DM an folgende Adresse erhalten:

Dr. Sven Naumann
LDV/CL
Universität Trier
D-54286 Trier

Wir möchten das über den ftp–Server verfügbare Angebot an für Forschung und Lehre relevanter Software möglichst umfassend und attraktiv gestalten. Aus diesem Grund fordern wir alle Entwickler und Entwicklerinnen von Parsern, Recognizern und Grammatiken, die bereit sind, ihre Ergebnisse zur Verfügung zu stellen, dazu auf, ihre Programme an folgende e-mail Addresse zu schicken:

naumann@ldv01.uni-trier.de

Trier, im April 1994
S.N. und H.L.

Einleitung

Ein kurzer Blick auf die Titel der in Zeitschriften und Tagungsbänden veröffentlichten computerlinguistischen Arbeiten genügt, um zu erkennen, daß sich eine nicht unerhebliche Zahl von ihnen mit Fragen des Parsings (wie z.B. der Entwicklung neuer bzw. der Optimierung bekannter Parsingalgorithmen, der Entwicklung von Parsern für bestimmte Grammatikformalismen, etc.) auseinandersetzt. Um so erstaunlicher ist es, daß es kaum aktuelle (Lehr-)Bücher gibt, die eine Einführung in diesen für die Computerlinguistik so zentralen Bereich bieten[1]. Das vorliegende Buch bemüht sich, diese Lücke zu schließen: Es enthält eine ausführliche Beschreibung der wichtigsten im Bereich der Computerlinguistik verwendeten Parsingalgorithmen. Alle Algorithmen werden nicht nur informell erläutert, sondern detailliert und in einem einheitlichen Format spezifiziert.

Das Buch kann, aufgrund unserer eigenen Erfahrung, als Textgrundlage für Parsingseminare, als Mittel zum Selbststudium bzw. als Nachschlagewerk verwendet werden. Da es sich außerdem ausführlich mit der Frage auseinandersetzt, wie sich auf Grundlage der beschriebenen Algorithmen Recognizer und Parser in Lisp bzw. in Prolog implementieren lassen, kann es darüber hinaus auch im Rahmen von Programmierkursen in diesen beiden Sprachen genutzt werden.

Wir möchten allerdings betonen, daß es sich bei dem Buch nicht um eine Einführung in die Computerlinguistik im allgemeinen handelt. Seine Lektüre setzt Grundkenntnisse der Mengenlehre und der Theorie der formalen Sprachen voraus. Programmierkenntnisse in Lisp oder Prolog dagegen sind wünschenswert, aber nicht unbedingt erforderlich.

Grenzen

Fast alle computerlinguistischen Methoden und Gegenstände haben Bezüge zum Parsing: Viele implementierte Systeme, gleichgültig, ob es sich dabei um Übersetzungsprogramme, Frage-Antwort-Systeme oder linguistische Werkzeuge („Tools") zur Text- und Sprachanalyse handelt, enthalten Parsingkomponenten. Auch für

[1] Am Ende dieses Vorwortes findet sich eine kommentierte Kurzbibliographie, die u.a. die uns bekannten Einführungen aufführt.

die Beurteilung von Repräsentationssprachen und Grammatikformalismen bildet die Verfügbarkeit geeigneter Parsingverfahren ein wichtiges Kriterium. Insofern gibt es nur sehr wenige computerlinguistische Arbeiten, die überhaupt nichts mit Parsing zu tun haben.

Da sich dieses Buch darauf beschränkt, einen Überblick über die wichtigsten in der Computerlinguistik verwendeten Parsingverfahren zu vermitteln, werden bestimmte Themenkomplexe, deren Behandlung eine LeserIn von einer Einführung in diesen Bereich erwarten würde, nur sehr knapp oder aber gar nicht berücksichtigt. Dazu zählen insbesondere die folgenden Punkte:

1. Wir haben darauf verzichtet, bestimmte existierende natürlichsprachliche Systeme bzw. Parsingkomponenten solcher Systeme detailliert zu beschreiben.

2. Auf Probleme, die sich bei der Entwicklung von Parsern für spezielle Grammatikformalismen (z.B. Parser für *tree adjoining grammars*, *Prinzipien-&-Parameter*-Parser etc.) ergeben, wird nur am Rande eingegangen.

3. Stochastische und konnektionistische Verfahren (*statistisches Parsen* bzw. *Parsen mit neuronalen Netzwerken*) werden nicht behandelt.

4. Die „*Spielzeug*"-Grammatiken, die sich in den diversen Beispielen des Buchs finden, spezifizieren entweder eine einfache formale Sprache, oder sie erlauben die Analyse bzw. Generierung einer kleinen Zahl deutscher (Deklarativ)Sätze. In jedem Fall wurden sie so formuliert, daß sich anhand ihrer die Charakteristika der verschiedenen Verarbeitungsstrategien illustrieren lassen. Wir erheben nicht den Anspruch, mit ihnen bestimmte sprachliche Phänomene des Deutschen in linguistisch adäquater Weise zu beschreiben.

5. Da die in diesem Buch verwendeten Grammatiken, die keine formale Sprache beschreiben, reine Satzgrammatiken sind, könnte der Eindruck entstehen, daß Parsing in irgendeinem direkten Zusammenhang mit Satzsyntax stünde. Dieser Eindruck ist jedoch falsch; die Algorithmen sind gegenüber der linguistischen Interpretation der Grammatiken neutral und eignen sich grundsätzlich auch für Grammatiken, die z.B. die Struktur von Silben, Morphemen, Wörtern, Texten oder sogar nicht-sprachlichen Zeichenkomplexen (z.B. Bildern oder Musikstücken) beschreiben. Fragen, die die Adäquatheit bestimmter Grammatiken oder Grammatiktypen für die Beschreibung gegebener linguistischer Phänomene betreffen, werden in diesem Buch nicht näher diskutiert. Es steht aber außer Frage, daß die Leistungsfähigkeit eines tatsächlich realisierten Parsingsystems in erheblichem Maße von dem Umfang und der Qualität der verwendeten Grammatik abhängt.

6. Verfahren aus der Signalphonetik und Mustererkennung werden wir nur am Rande behandeln, obwohl man die Erkennung gesprochener Sprache, also z.B.

die Abbildung eines kontinuierlichen Sprachsignals auf eine Kette von diskreten Symbolen eines phonologischen Alphabets, durchaus als ein Parsingproblem im weiteren Sinne auffassen kann. Die bei solchen Fragestellungen verwendeten Algorithmen sind jedoch grundsätzlich verschieden von den hier dargestellten Verfahren. Alle in diesem Buch beschriebenen Algorithmen setzen nämlich als Eingabe bereits eine Folge von Symbolen aus einem endlichen, vordefinierten Alphabet voraus, und alle Operationen, die in den Algorithmen Verwendung finden, sind Manipulationen dieser diskreten Symbole.

7. Die Verarbeitung natürlicher Sprache ist ein hochgradig komplexes Problem, was dazu führt, daß bereits bei der Entwicklung von Parsern für Sprachfragmente mittlerer Größenordnung Effizienzüberlegungen berücksichtigt werden müssen. Wir werden in diesem Buch nur gelegentlich auf effiziente Parserprogrammierung in den beiden verwendeten Programmiersprachen eingehen und uns im allgemeinen auf die Diskussion der Effizienz der zugrundeliegenden *Algorithmen* beschränken.
   Bei den Beispielprogrammen wurde dem Kriterium der Übersichtlichkeit des Kodes eine höhere Priorität eingeräumt als dem Kriterium der Verarbeitungsgeschwindigkeit (so finden sich z.B. aus Gründen der besseren Lesbarkeit Prologprogramme mit dem Listenkonstruktor *append*, obwohl die semantisch äquivalente Differenzlistenrepräsentation immer effizienter ist). Diese Bevorzugung des Übersichtlichkeitskriteriums gegenüber der Effizienz hat den folgenden Grund: Wir gehen davon aus, daß die Programmiersprachen Lisp und Prolog in der Regel für *rapid prototyping* eingesetzt werden. Die in diesen Sprachen entwickelten linguistischen Programme haben zumeist nur wenige Anwender und dienen eher dem Zweck, die grundsätzliche Algorithmisierbarkeit bestimmter Probleme zu zeigen. Soll ein in Lisp oder Prolog entwickeltes Programm einer größeren Zahl von Anwendern zur Verfügung gestellt werden, empfiehlt sich wohl ohnehin eine Reimplementierung in einer dafür geeigneten Programmiersprache wie z.B. C.

8. Für viele relevante mathematische Eigenschaften der in diesem Buch diskutierten Parsingalgorithmen liegen Beweise vor. Wir beschränken uns darauf, die wichtigsten Resultate wiederzugeben. Auf die ihnen zugrundeliegenden Beweise wird an den entsprechenden Stellen verwiesen.

9. Eine wichtige Motivation für die Beschäftigung mit Parsingverfahren ist die Modellierung der Sprachverarbeitung durch den Menschen. Da aber eine angemessene Berücksichtigung dieses interessanten Aspekts den Umfang dieses Buches sprengen würde, haben wir auf eine ausführliche Diskussion der psycholinguistischen Adäquatheit von Parsingverfahren verzichtet.

Aufbau

Im ersten Teil des Buchs (Kapitel 1 und 2) beschreiben wir zunächst kurz, welche Bedeutung Verfahren der maschinellen Sprachanalyse in der Informatik (Compilerbau), der kognitiven Psychologie und der Computerlinguistik haben. Außderdem führen wir die grundlegenden Begriffe wie *Parser*, *Recognizer*, *Parsingalgorithmus* usw. ein und erläutern unsere Darstellungsweise von Algorithmen.

Teil 2 (Kapitel 3 bis 5) stellt eine Reihe einfacher Parsingalgorithmen für kontextfreie Syntaxen vor, die entweder mit einer *Depth-first-Suchstrategie mit Backtracking* arbeiten oder eine *Breadth-first-Suchstrategie* verwenden.

Der dritte Teil (Kapitel 6 bis 9) gibt einen Überblick über *Chart-parsing*-Algorithmen. Charakteristisch für diese Parsingverfahren ist, daß sie zusätzliche Mechanismen für die Verwaltung bereits erreichter Teilergebnisse verwenden und so redundante Arbeitsschritte vermeiden, die bei den in Teil 2 besprochenen Algorithmen nicht ausgeschlossen werden können.

Anschließend (Kapitel 10 und 11) wenden wir uns dem deterministischen Parsing zu. Eine besonders wichtige Rolle spielen dabei die LL- und LR-Algorithmen, die vor allem im Compilerbau sehr häufig verwendet werden und die Grundlage für den in den letzten Jahren in vielen natürlichsprachlichen Systemen verwendeten Algorithmus von M.Tomita bilden.

In der Computerlinguistik werden seit Anfang der 80er Jahre zunehmend unifikationsbasierte Grammatikformalismen verwendet. Teil 5 (Kapitel 12 und 13) demonstriert, wie sich die zuvor behandelten Chart-Parsing-Algorithmen so modifizieren lassen, daß sie auch für diese Grammatikformalismen verwendet werden können.

Am Ende des Buchs finden sich eine Reihe von Appendices. Sie enthalten u.a.:

- eine kurze Darstellung der grundlegenden Begriffe aus dem Bereich der Theorie der formalen Sprachen und der Graphentheorie und
- erläutern die wichtigsten der verwendeten Datenstrukturen.

Allen Kapiteln liegt die folgende Struktur zugrunde: Den Anfang bildet die informelle Darstellung eines Algorithmus, die anschließend schrittweise präzisiert wird. In den meisten Fällen wird zunächst ein Erkennungsalgorithmus entwickelt, der dann zu einem Parsingalgorithmus erweitert wird. Gegebenenfalls werden auch interessante Varianten des Grundalgorithmus behandelt. Es folgen Implementierungen der im ersten Teil des Kapitels vorgestellten Algorithmen. Allerdings wird in der Regel nicht jeder Algorithmus in LISP **und** PROLOG dargestellt. Am Schluß eines Kapitels finden sich Übungsaufgaben, die der Vertiefung des behandelten Stoffs dienen sollen.

In dem folgenden Literaturüberblick führen wir einige der wichtigsten Werke auf, die geeignet sind, die Lektüre dieses Buches zu ergänzen und zu vertiefen.

Literatur

Aho, A.V., Ullman, J.D. (1972). The Theory of Parsing, Translation, and Compiling. Prentice-Hall: Englewood Cliffs, New York.
Dieses Buch ist ein Klassiker, in dem die wichtigsten Algorithmen und Beweise knapp und präzise dargestellt werden. Es führt auch in die Grundlagen der Mengentheorie, der Theorie der formalen Sprachen und der Graphentheorie ein. Die Darstellungsweise ist an informatischen Standards und dem imperativen Programmierparadigma orientiert. Das Buch enthält allerdings keine linguistischen Bezüge.

Bátori, I.S., Lenders, W., Putschke, W. (Hrsg.) (1989) Computational Linguistics. An International Handbook on Computer Oriented Language Research and Applications. Berlin: Walter de Gruyter.
Ein sehr umfangreiches Nachschlagewerk. Die beiden Artikel von P.Hellwig über Parsing geben einen knappen Überblick über Parsingalgorithmen.

Charniak, E., Riesbeck, C.K., McDermott, D.V., Meehan, J.R. (1987) Artificial Intelligence Programming. New Jersey: Lawrence Erlbaum Assoc., 2. Auflage.
Ein ausgezeichnetes LISP-Buch für Fortgeschrittene.

Gazdar, G., Mellish, C. (1989) Natural Language Processing in PROLOG/LISP. Addison-Wesley: Reading, M.A.
Diese beiden Bücher (oder beiden Varianten eines Buches) sind Einführungen in die Computerlinguistik. Sie decken alle wichtigen Teilgebiete von der Phonologie bis hin zur Pragmatik ab. Die Darstellungsweise hat zwei Komponenten: Zum einen werden die Inhalte informell und didaktisch angerissen, zum anderen werden sie durch Implementierungen in der jeweiligen Programmiersprache exemplifiziert. Auf eine systematische, formale und programmiersprachenunabhängige Darstellung der Verfahren wird allerdings weitgehend verzichtet.

Görz, G. [Hrsg.] (1993) Einführung in die künstliche Intelligenz. Bonn: Addisson-Wesley.
Dieses umfangreiche Buch enthält u.a. Abschnitte über KI-Programmiertechniken in LISP und PROLOG, linguistische Repräsentationsformalismen und kognitionswissenschaftliche Aspekte der Sprachverarbeitung.

Hopcraft, J.E., Ullman, J.D. (1979) Introduction to Automata Theory, Languages, and Computation. Massachusetts: Addison-Wesley.
Dieses Buch darf z.Zt. als DAS Standardwerk zur Theorie der formalen Sprachen gelten. Es deckt alle wichtigen Teilbereiche (Automatentheorie, Komplexitätstheorie usw.) ab und enthält alle relevanten Beweise. Die Darstellungsweise ist sehr redundanzarm und präzise.

König, E., Seiffert, R. (1989) Grundkurs PROLOG für Linguisten. Tübingen: Francke Verlag, UTB 1525.
Ein Prologlehrbuch mit linguistischen Anwendungen.

Partee, B.H., ter Meulen, A., Wall, R.E. (1990) Mathematical Methods in Linguistics. Dordrecht: Kluwer Academic Publishers.
Ein ausgezeichnetes Buch, das eine ausführliche und präzise Darstellung der für die (Computer-)Linguistik relevanten mathematischen Grundlagen und Verfahren bietet.

Pereira, F.C.N., Shieber, S.M. (1987) Prolog and Natural-Language Analysis. Stanford: CSLI Lecture Notes 10.
Eine fundierte, empfehlenswerte Einführung in das Parsing mit Prolog.

Sterling, L., Shapiro, E. (1986) The Art of Prolog. Cambridge: MIT Press.
Dieses Buch ist u.E. derzeit die beste Einführung in die Programmiersprache PROLOG. Es steht zwar in dem Ruf, deutlich anspruchsvoller zu sein als vergleichbare Werke, es ist aber auch für den engagierten Anfänger empfehlenswert.

Winston, P.H., Horn, B.K.P. (1989^3) LISP. Massachusetts: Addison-Wesley.
Trotz einiger konzeptioneller Schwächen immer noch ein besonders für Anfänger geeignetes Buch, das sich durch eine Vielzahl von Beispielen und Anwendungen auszeichnet.

Winograd, T. (1983) Language as a Cognitive Process: I. Syntax. Addison-Wesley.
Ein weitgefächerter, sehr systematischer Überblick über computerlinguistische Methoden und Resultate. Nach wie vor sehr empfehlenswert.

Viele wichtige Arbeiten im Bereich Parsing kursieren bereits Jahre vor ihrer regulären Veröffentlichung als *technical reports* oder als *graue Papiere*. Diese Situation macht es gerade für AnfängerInnen schwierig, sich einen Überblick über den aktuellen Stand der Diskussion in diesem Bereich zu verschaffen. Hilfreich sind in diesem Zusammenhang besonders die folgenden Quellen:

- die Proceedings der einschlägigen internationalen Konferenzen COLING, ACL und EACL;
- spezielle Tagungen und Workshops zum Thema Parsing (z.B. der TWLT-Workshop „Parsing Natural Language“, 1993);
- Fachzeitschriften wie *Computational Linguistics*, *Artificial Intelligence* etc.;
- themenspezifische Fachgruppen in Vereinen wie der *Deutschen Gesellschaft für Sprachwissenschaft* (DGfS), der *Gesellschaft für Informatik* (GI) oder der *Gesellschaft für linguistische Datenverarbeitung* (GLDV);
- die usenet-newsgroup *comp.ai.nat-lang* (die *elektronische Zeitung* im Internet) und andere elektronische Informationsquellen wie die *newsletter* bestimmter Interessengruppen und die ftp-server relevanter Projekte und Institute.

Inhaltsverzeichnis

Symbolverzeichnis

Symbol	*Bedeutung*
$<$	kleiner als; bei ID/LP-Syntaxen: lineare Präzedenz
$>$	größer als
x $\in$ A	x ist Element von A
x $\notin$ A	x ist kein Element von A
{x| ...}	die Menge aller x, für die gilt: ...
A $\subseteq$ B	A ist eine Teilmenge von B
A $\subset$ B	A ist eine echte Teilmenge von B
A $\cup$ B	Vereinigung der Menge A und der Menge B
$\bigcup A_1, \ldots, A_n$	Vereinigung aller A_i
A $\cap$ B	Schnitt der Menge A mit der Menge B
$\bigcap A_1, \ldots, A_n$	Schnitt aller A_i
A – B	Differenz der Mengen A und B
$\bar{A}$	Komplement der Menge A
POT(A)	Potenzmenge der Menge A
CARD(A)	Kardinalität der Menge A
A $\times$ B	kartesisches Produkt der Mengen A und B
f:A $\rightarrow$ B	f ist eine Funktion bzw. Abbildung von A nach B
$\emptyset$ bzw. {}	leere Menge
$\wedge$	Konjunktion
$\vee$	Disjunktion
$\neg$	Negation
$\exists$	Existenzquantor
$\forall$	Allquantor
a^n	die Kette, die aus n Vorkommen des Symbols a besteht
|k|	die Länge der Kette k
|G|	die Länge der Grammatik G
x $\rightarrow \alpha$	ersetze das Symbol x durch die Kette α
$\alpha \Rightarrow \beta$	die Kette β ist aus der Kette α direkt ableitbar
$\alpha \ {}^* \!\Rightarrow \beta$	die Kette β ist aus der Kette α ableitbar
V^*	Abschluß eines Alphabets bzw. einer Sprache (Kleene-Stern)
e	leeres Wort
$\Leftarrow$	Wertzuweisung
$\sqcup$	Unifikation
$\preceq$	Subsumtion
[ ]	die leere Merkmalsstruktur (*top*)
$\perp$	die inkonsistente Merkmalsstruktur (*bottom*)
Φ	Restriktor einer Merkmalsstruktur
FIRST	FIRST-Relation

Teil I

Grundlagen

Kapitel 1

Was ist Parsing?

Der Ausdruck „Parsing“ leitet sich aus den „partes orationis“, d.h. den Wortarten, ab. Entsprechend den Vorstellung der Grammatiktradition des 19. Jahrhunderts besteht die grammatische Analyse im wesentlichen aus der Bestimmung der Wortarten.

Die Vorstellungen von der Form von Grammatiken und dem, was sie leisten sollen, haben sich in der Zwischenzeit erheblich gewandelt. In der modernen Linguistik werden Grammatiken in der Regel als formale - oder zumindest doch im Prinzip formalisierbare - komplexe Objekte betrachtet, die in systematischer Form phonologische, morphologische, syntaktische, semantische und auch pragmatische Informationen über eine natürliche Sprache präsentieren. So betrachtet, gibt es für Grammatiken einerseits *formale* Adäquatheitskriterien (wie z.B. Konsistenz, Entscheidbarkeit, ...), wie anderseits bestimmte *empirische* Adäquatheitskriterien (z.B. die in [**?**] formulierten Kriterien der *Beobachtungs-*, *Beschreibungs-* und *Erklärungsadäquatheit*).

Für fast alle der in den letzten Jahrzehnten entwickelten Grammatikformalismen wurden geeignete Parsingalgorithmen entwickelt und implementiert. Dazu gehören u.a.:

- Verschiedene Typen von Unifikationsgrammatiken (PATR-II, GPSG, LFG, HPSG, ...),
- Tree Adjoining Grammars (TAGs),
- Dependenzgrammatiken,
- Kategorialgrammatiken und
- Systeme, die sich an der *Government-&-Binding*-Theorie orientieren („*parameter-based parsing*“).

Der Begriff „Parsing“ wird heutzutage in vielen Disziplinen wie dem Compilerbau, der kognitiven Psychologie, der Computerlinguistik, sowie der sprachorientierten KI

und der Sprachtechnologie als selbstverständlicher Grundbegriff verwendet, „der keiner weiteren Explikation bedarf". Vielleicht mag darin die Ursache dafür liegen, daß es nach wie vor keine allgemein akzeptierte und präzise Definition für einen Parser gibt.

Zuweilen wird unter einem Parser ein lauffähiges und tatsächlich auf einem konkreten Rechner implementiertes Programm verstanden, das in der Lage ist, formalsprachliche oder natürlichsprachliche Eingaben auf Korrektheit (Grammatikalität) zu überprüfen und ihnen dabei eine Interpretation, gewöhnlich in Form von einer oder mehreren Strukturbeschreibungen, zuordnet. Manchmal wird jedoch mit „Parser" auch der (abstrakte) Algorithmus bezeichnet, der einer solchen Implementierung zugrunde liegt (etwa, wenn man von einem „Earley-Parser" spricht).

Auch die Form der Eingabe sowie die Art des bei der Analyse herangezogenen Wissens variieren stark: Während insbesondere in der Computerlinguistik Parsing häufig mit automatischer (syntaktischer) Satzanalyse gleichgesetzt wird, gibt es gerade im Bereich der kognitiven Psychologie eine Reihe (meistens aus den 70er Jahren stammender) Systeme, deren *Parser* sich um eine direkte semantische Interpretation von Eingaben bemüht. Neben *syntaktischen* und *semantischen* Parsern finden sich aber auch *phonologische* und *morphologische* Parser sowie *Textparser*, d.h. Systeme die Text- bzw. Diskursstrukturen analysieren. Aus diesem Grund führt die Beschränkung auf die syntaktische Analyse von Sätzen zu einer zu engen Definition von Parsing[1]. Der Versuch, eine Definition des Begriffs Parser zu entwickeln, die allen seinen unterschiedlichen Verwendungsweisen gerecht wird, kann andererseits leicht zu einer zu weiten Definition dieses Begriffs führen: Wenn man einen Parser einfach als Algorithmus bzw. Programm definiert, das einem natürlichsprachlichen bzw. einem formalsprachlichen Ausdruck einen anderen formalsprachlichen Ausdruck zuordnet, dann fallen im Prinzip auch beliebige Computerprogramme bzw. Algorithmen unter diese Definition. Aufgrund dieser konzeptuellen Probleme werden wir Parser deshalb zunächst informell als Klasse von Algorithmen auffassen, die für die Analyse (natürlich-)sprachlicher Ausdrücke entwickelt und verwendet wurden, und zwar im wesentlichen - aber nicht nur - im Bereich der Satzsyntax.

Wenn man Parsing als einen Prozeß betrachtet, in dem einem sprachlichen Ausdruck eine Strukturbeschreibung zugewiesen wird, dann lassen sich die folgenden Faktoren unterscheiden:

- Eingabe:
  Als Eingabe kommen natürlichsprachliche Ausdrücke oder Ausdrücke einer formalen Sprache, wie z.B. einer Programmiersprache, in Frage. Natürlichsprachliche Ausdrücke können entweder in geschriebener Form oder als akustisches Signal vorliegen.

[1] Die starke Orientierung an der syntaktischen Analyse hat vor allem historische Gründe, die mit dem Primat der Syntax in Linguistik und Computerlinguistik in den letzten Jahrzehnten zusammenhängen.

- Ausgabe:
  Es gibt eine Vielzahl von Möglichkeiten, syntaktische Strukturbeschreibungen zu repräsentieren; zu den wichtigsten gehören Baumdiagramme und Attribut-Wert-Matrizen (Merkmalsstrukturen).

- Parser:
  Parsing kann entweder als Teil des menschlichen Sprachverstehensprozesses aufgefaßt werden oder als ein maschinell realisierter Prozeß.

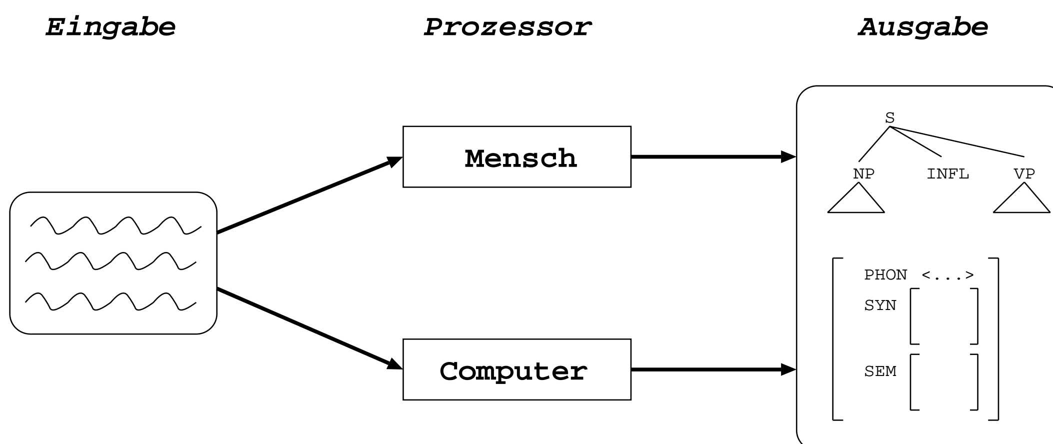


Abbildung (1-1)

Mit dieser intuitiven Vorstellung von Parsing wollen wir uns vorläufig begnügen[2]. In den folgenden Abschnitten skizzieren wir kurz, welche Rolle Parsing in den Disziplinen Compilerbau, kognitive Psychologie und Computerlinguistik spielt. Dabei zeigt sich, daß es ein breites Spektrum unterschiedlicher Fragestellungen und Erkenntnisinteressen gibt, die eine Auseinandersetzung mit Sprachanalysemethoden motivieren.

1.1 Parsing und Compilerbau

Anweisungen, die von einem Rechner direkt ausgeführt werden können, müssen in der *Maschinensprache* geschrieben sein, die der Prozessor dieses Rechners „versteht“. Für die Entwicklung von Programmen allerdings werden in der Regel sogenannte *höhere Programmiersprachen* wie z.B. C, COBOL, FORTRAN, LISP, PASCAL, PROLOG etc. verwendet. Neben der Portabilität dieser Programme liegt ein entscheidender Vorteil bei der Verwendung höherer Programmiersprachen darin, daß

[2] In Kapitel 2 dieses Buches werden wir dann präziser definieren, wie wir die Begriffe „Parser“, „Parsingalgorithmus“ usw. verwenden, und auch auf das Verhältnis von Parsingalgorithmen zu anderen Algorithmen (insbesondere Suchalgorithmen) eingehen.

sie – anders als eine Maschinensprache bzw. maschinennahe Sprache wie Assembler – eine große Zahl leistungsfähiger Funktionen und Prozeduren zur Verfügung stellen, die es ermöglichen, für bestimmte Aufgabenstellungen in kurzer Zeit leistungsfähige Programme zu entwickeln.

Die in einer höheren Programmiersprache geschriebenen Programme (*Quellkode*) können allerdings nicht direkt ausgeführt werden, sondern müssen dazu in eine Folge von Maschinensprachanweisungen (*Objektkode*) übersetzt werden. Diese Übersetzung des Quellkodes in den Objektkode wird als *Compilierung* bezeichnet[3].

P sei ein beliebiges in einer höheren Programmiersprache geschriebenes Programm und M der durch die Compilierung von P generierte Objektkode. Die Compilierung von P kann als ein zweistufiger Prozeß betrachtet werden:

1. *Syntaktische Analyse:*
   Die syntaktische Struktur von P wird analysiert; d.h. P wird auf eine Strukturbeschreibung SB(P) abgebildet.

2. *Semantische Analyse:*
   Die Strukturbeschreibung des Programms wird verwendet, um ein Maschinensprachprogramm zu generieren, dessen Semantik mit der von P übereinstimmt; d.h. SB(P) wird auf M abgebildet.

Die syntaktische und semantische Analyse lassen sich in eine Reihe einzelner Schritte zerlegen, wie die folgende Abbildung zeigt:

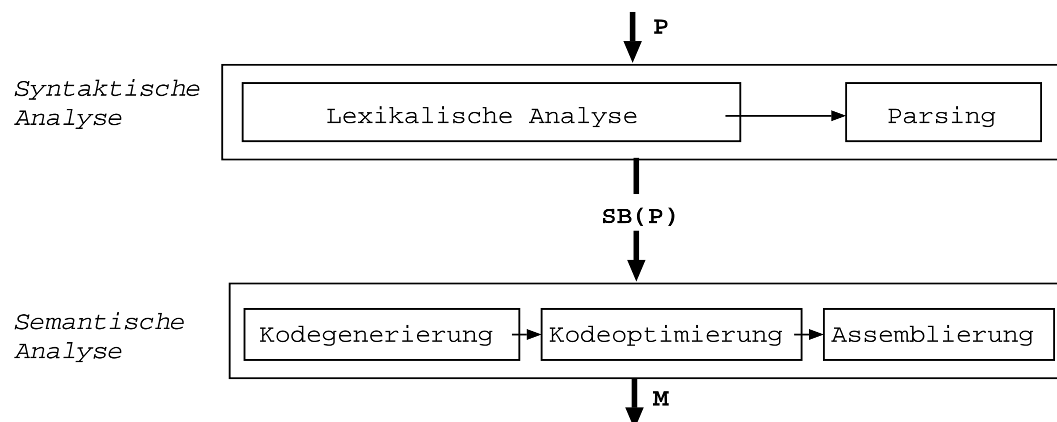


Abbildung (1-2)

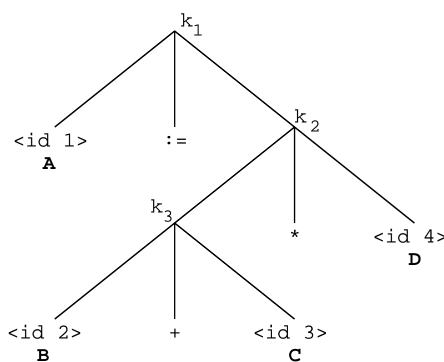
Die Aufgabe der lexikalischen Analyse besteht darin, die Zeichenfolgen des Programmtextes zu identifizieren, die eine einzelne syntaktische Einheit repräsentieren

[3] Die folgende Darstellung, die sich an [?, S.53-75] orientiert, beschränkt sich darauf, die Bedeutung der syntakischen Analyse bei der Compilierung eines Programms zu skizzieren und erhebt keinen Anspruch auf Vollständigkeit oder etwa darauf, den neuesten Stand der Compilertechnik zu beschreiben.

und ihnen eine syntaktische Kategorie (wie z.B. *Variable* oder *Konstante*) zuzuordnen. Diese Einheiten werden als *Token* bezeichnet. Der Parser operiert auf einer von der lexikalischen Analyse generierten Kette von Token und ordnet ihr eine Strukurbeschreibung (meistens in Form eines Strukturbaums) zu. Diese Strukturbeschreibung bildet dann den Ausgangspunkt für die Kodegenerierung: Häufig wird zunächst ein maschinensprachnaher Zwischenkode (z.B. *Assembler*) generiert, der dann optimiert und schließlich in den eigentlichen Maschinenkode übersetzt wird (*Assemblierung*).

Wie auch in vielen natürlichsprachlichen Systemen hat auch hier die syntaktische Analyse sprachlicher Einheiten primär die Funktion, bestimmte semantische Operationen zu unterstützen. Wir werden abschließend an einem kleinen, stark vereinfachten Beispiel demonstrieren, wie die syntaxgestützte Generierung von Assemblerkode realisiert werden kann.

Beispiel (1-1)
Zu den in einer Assemblersprache verfügbaren Operationen gehören typischerweise bestimmte Registeroperationen wie z.B. das Übertragen des Inhalts einer bestimmten Speicheradresse in ein Register (LDA m[4]), das Speichern des Registerinhalts an einer bestimmten Speicheraddresse (STA m) und das Addieren und Multiplizieren des Registerinhalts mit dem Inhalt einer bestimmten Speicheradresse (ADC m / MPY m).



Angenommen, bei dem Quellkode handelt es sich um ein PASCAL-Programm, das unter anderem auch die folgende Anweisung enthält:

A := (B + C) * D.

Wenn die lexikalische Analyse für diese Anweisung die Tokenkette

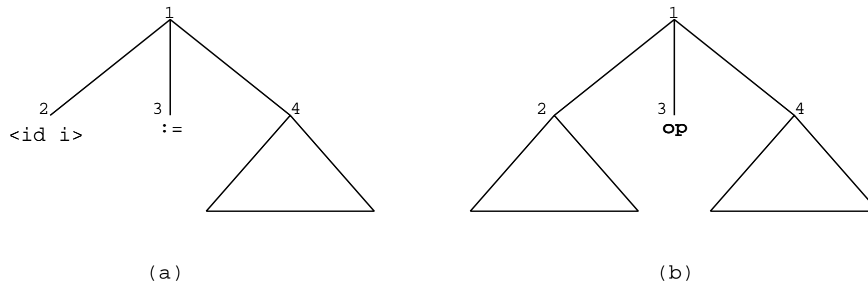
<id 1> := (<id 2> + <id 3>) * <id 4>

---

[4]Die in diesem Abschnitt verwendeten Befehle stammen aus dem Befehlssatz des 6502-Prozessors von Motorola, der in so legendären Kleincomputern wie dem Apple-II und dem C-64 verwendet wurde.

und der Parser für diese Kette die durch den Strukturbaum (s.o.) repräsentierte Strukturbeschreibung generiert, dann könnte die Kodegenerierung wie folgt realisiert werden:

1. Der Strukturbaum setzt sich aus zwei Typen von Teilbäumen zusammen:



   wobei Δ einen beliebigen Teilbaum repräsentiert, der natürlich auch nur aus einem einzelnen Knoten bestehen kann, und **op** $\in \{+, *\}$.

2. Jedem Teilbaumtyp kann eine Folge (schematischer) Anweisungen zugeordnet werden, die die intendierte Semantik des durch ihn repräsentierten Ausdrucks abbilden:

   (a) Berechne den Wert von 4.
   Speichere diesen Wert unter 2. (STA <id *i*>)

   (b) Berechne den Wert von 2.
   Speichere diesen Wert unter \$1, (STA <id \$1>)
   wobei \$1 eine nicht bereits verwendete Speicheradresse bezeichnet.
   Berechne den Wert von 4.
   Wende **op** auf diesen Wert und \$1 an. (MPY \$1 bzw. ADC \$1)

3. Besteht ein Teilbaum aus einem einzelnen Knoten, der mit einer Variablen *var* etikettiert ist, dann muß nur der Wert dieser Variablen in das Register übertragen werden:
   LDA *var*.

4. Der Kode für den gesamten Strukturbaum kann *bottom-up* generiert werden, indem man mit dem Teilbaum beginnt, dessen Wurzelknoten ausschließlich Blätter dominiert und sich dann schrittweise nach oben arbeitet. Der Kode, der dem Wurzelknoten des Strukturbaums (k_1) zugeordnet wird, ist der Assemblerkode für die vollständige PASCAL-Anweisung.

   In unserem Beispiel beginnen wir mit der Kodegenerierung bei dem Teilbaum mit der Wurzel k_3:

   (a) Dem Knoten k_3 wird folgende Anweisungsfolge zugeordnet:
   LDA B
   STA \$1
   LDA C
   ADC \$1

Nach Ausführung dieser Sequenz enthält das Register den Wert der Addition von A und B. Damit ist auch die Berechnung des ersten Arguments der Multiplikation abgeschlossen.

(b) Die Anweisungen, die k_2 zugeordnet werden, bestehen aus den k_3 zugeordneten Anweisungen, gefolgt von:

```
⋮
STA $2
LDA D
MPY $1
```

Nach Ausführung dieser Anweisungen enthält das Register den Wert des gesamten arithmetischen Ausdrucks, der nur noch der Variablen A zugewiesen werden muß.

(c) Der Kode für k_1 enthält als nur eine weiter Anweisung:

```
⋮
STA A
```

Als vollständigen Kode erhalten wir also:

```
LDA B
STA $1
LDA C
ADC $1
STA $2
LDA D
MPY $1
STA A
```

Ein wichtiger Unterschied zwischen formalen Sprachen (wie z.B. Programmiersprachen) und natürlichen Sprachen besteht darin, daß formalen Sprachen „*Kunstprodukte*“ sind, die für einen bestimmten Aufgabenbereich konzipiert werden. Sie unterscheiden sich von natürlichen Sprachen unter anderem dadurch, daß sie eine genau definierte Semantik besitzen und - wenn überhaupt - ein erheblich geringeres Maß an Ambiguität und Vagheit aufweisen.

Um die Entwicklung effizienter Compiler zu unterstützen, wird beim Entwurf einer Programmiersprache in der Regel darauf geachtet, daß sich die syntaktische Struktur ihrer Sätze durch einen Typ von Syntax beschreiben läßt, der die Verwendung von Parsingalgorithmen erlaubt, die sich durch günstige Zeit- und Speicheranforderungen auszeichnen. Häufig werden bestimmte Subtypen kontextfreier Syntaxen verwendet wie z.B. LL- und LR-Syntaxen (vgl. Kapitel 10).

1.2 Parsing und Kognitive Psychologie

Die sprachorientierte kognitive Psychologie und verwandte Disziplinen wie die Psycholinguistik, die Kognitive Linguistik, die Neurolinguistik etc. beschäftigen sich mit Fragestellungen wie den folgenden:

- Wie verarbeitet der Mensch Sprache?
- Welche kognitiven Prozesse und mentalen Zustände spielen dabei eine Rolle?
- Wie hängt die sprachliche Kompetenz und Performanz mit anderen kognitiven Fähigkeiten und Verhaltensweisen zusammen?

Gegenstände wie „mentale Zustände“ und „kognitive Prozesse“ sind - das liegt in der Natur der Sache - nicht direkt beobachtbar und können experimentell nur indirekt durch Korrelate wie Verarbeitungszeiten (beobachtbar durch Augenbewegungsmessungen und probantengesteuerte inkrementelle wortweise Darbietung von Sätzen am Bildschirm) erschlossen werden. Neben dem grundsätzlich problematischen Verfahren der Introspektion und den genannten indirekten empirischen Beobachtungsmethoden bedient sich die Psycholinguistik auch der Simulation von kognitiven Prozessen als einer Methode, die die Korrektheit von Hypothesen über die menschliche Sprachverarbeitung bis zu einem gewissen Grade überprüfbar und falsifizierbar machen kann. Zudem können Simulationen natürlich auch einen nicht unbedeutenden heuristischen Wert für die weitere Hypothesenbildung haben. Umgekehrt können psycholinguistische Ergebnisse über die menschliche Sprachverarbeitung (das nach wie vor beste Sprachverarbeitungssystem) die Entwicklung von verbesserten Parserarchitekturen unterstützen.
Wichtige psycholinguistische Untersuchungen mit Bezügen zur Parsingtheorie betreffen vor allem die Untersuchung der Verarbeitung von Konstruktionen wie:

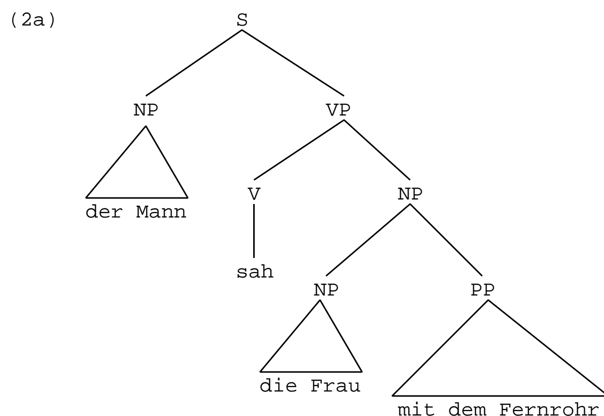
1. *PP-Attachment*
2. Anaphernresolution
3. *Garden-path*-Sätze

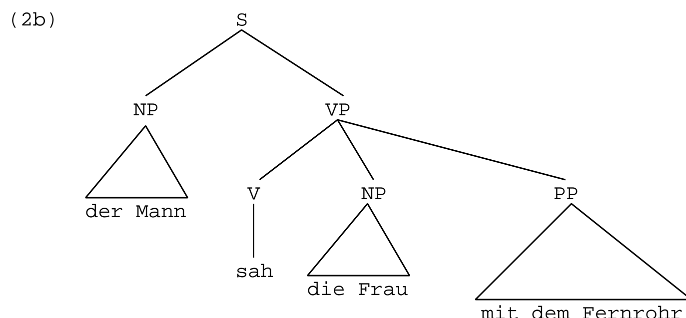
Die ersten beiden Phänomene unterstreichen die Notwendigkeit einer engen Interaktion von syntaktischer und semantischer Analyse, während die Verarbeitung von *Garden-path*-Sätzen wichtige Aufschlüsse über die Organisation des menschlichen Kurzzeitgedächtnisses geben kann.

Beispiel (1-2)
PP-Attachment
Im Deutschen wie im Englischen können Präpositionalphrasen sowohl attributiv als

auch adverbiell verwendet werden. Für den Satz (1) lassen sich deshalb die syntaktischen Strukturbeschreibungen (2a) und (2b) angeben, die mit diesen unterschiedlichen Lesarten korrespondieren. Bei einer direkten semantischen Interpretation von Sätzen, die derartige Konstruktionen enthalten, besteht die Gefahr, daß nur eine der beiden Lesarten erkannt wird.

(1) *Der Mann sah die Frau mit dem Fernrohr.*





Anaphernresolution
Als Anaphern bezeichnet man u.a. Reflexiv- und Reziprokpronomina (im Deutschen *sich*, *einander*; im Englischen *herself, himself, ..., each other*). Auf den ersten Blick scheint es, daß die Referenz solcher Ausdrücke allein durch den Äußerungskontext determiniert wird. Es läßt sich aber leicht zeigen, daß auch syntaktische Faktoren eine wichtige Rolle zu spielen scheinen. So gibt es z.B. Anaphern, die nur als korreferentiell mit solchen Nominalphrasen interpretiert werden können, die sich in ihrer unmittelbaren Umgebung befinden:

(3a) *Bill believes that* $John_i$ *hurt* $himself_i$[5].
(3b) **Bill*$_i$ *believes that John hurt* $himself_i$.
(4a) $Bill_i$ *believes that John hurt* him_i.
(4b) **Bill believes that* $John_i$ *hurt* him_i.

Es gibt verschiedene Versuche, diese syntaktischen Faktoren genau zu charakterisie-

ren: In der *Government-&-Binding*-Theorie z.B. wird versucht, eine rein strukturelle Analyse zu entwickeln; die *Lexical Functional Grammar* dagegen führt sie auf funktionale Relationen zurück.

Garden-path-Sätze
Als „*Garden-path*-Sätze“ werden solche Konstruktionen bezeichnet, bei denen lokale, zunächst nicht offensichtliche syntaktische Ambiguitäten auftreten und erst spät aufgelöst werden. Das folgende Beispiel illustriert eine typische *Garden-path*-Konstruktion des Englischen:

(5a) ... *the horse raced past the barn* ...

Auf den ersten Blick scheint es sich bei *raced* um das finite Verb eines einfachen Matrixsatzes zu handeln. Da es im Englischen jedoch möglich ist, Relativsätze ohne einleitende Relativpronomina wie *that* oder *who* zu bilden, kann *raced past the barn* auch als Relativsatz fungieren. Diese Lesart wird aber überhaupt erst in Betracht gezogen, wenn das Auftreten einer zweiten finiten Verbform die zunächst naheliegendere Lesart ausschließt:

(5b) ... *the horse raced past the barn fell* ...

Bei mehrfacher Verschachtelung solcher *Garden-path*-Konstruktionen (z.B. weitere eingebettete Relativsätze) treten beim menschlichen Rezipienten erhebliche Verarbeitungsprobleme auf, die aus kognitiver Sicht zurückgeführt werden auf die begrenzte Kapazität des menschlichen Kurzzeitgedächtnisses. Für die Simulation solcher Annahmen sind die Arbeiten von Mitchell Marcus grundlegend. Der von ihm entwickelte deterministische Parser verwendet einen begrenzten Stack, der nur die Verarbeitung solcher Sätze erlaubt, die auch menschlichen Hörern/Lesern keine Verständnisprobleme bereiten.

Charakteristisch für frühere Arbeiten im Bereich der Kognitiven Modellierung der menschlichen Sprachverarbeitung ist der Versuch einer unmittelbaren semantischen Interpretation sprachlicher Ausdrücke ohne vorherige syntaktische Analyse. Zu den einflußreichsten Ansätzen dieser Art zählen die „conceptual dependency theory“ [**?**] und die Arbeiten von Anderson (u.a. [**?**] und [**?**]). Probleme dieses, aus linguistischer Sicht „naiven“ Ansatzes führten Mitte der siebziger Jahre zur Entwicklung kognitiv motivierter Grammatikformalismen wie den ATNs („Augmented transition networks“), vgl. [**?**].

Einen ausführlichen Überblick über die kognitive Plausibilität von Parsingarchitekturen gibt [**?**]. In neuester Zeit sind insbesondere auch konnektionistische Ansätze in der Sprachverarbeitung von besonderem Interesse in der kognitiven Psychologie (vgl. [**?**], [**?**] und [**?**]).

[5]Die Indizes in diesen Beispielen werden als „referentielle Indizes“ bezeichnet und sollen zum Ausdruck bringen, daß Ausdrücke mit gleichen Indizes sich auf dasselbe Objekt beziehen.

1.3 Parsing in der Computerlinguistik

Die Computerlinguistik beschäftigt sich mit der automatischen Verarbeitung natürlicher Sprache. Dazu zählen Problembereiche wie natürlichsprachliche Benutzerschnittstellen zu Computersystemen, automatische und maschinengestützte Übersetzung, Hilfsmittel zur Textverarbeitung (automatische Rechtschreib- und Grammatikkorrektur), Text-to-speech-Systeme etc.

Die Zielsetzungen der Computerlinguistik sind im Unterschied zur kognitiven Psychologie nicht notwendigerweise darauf ausgerichtet, Parsingsysteme als (partielle) Modelle der menschlichen Sprachverarbeitungsstrategien zu entwickeln. Unterschiede zur Herangehensweise an Parsingprobleme im Bereich Compilerbau ergeben sich daraus, daß sich die Computerlinguistik in erster Linie mit der Verarbeitung natürlicher Sprache auseinandersetzt. Die Unterschiede zu den Nachbardisziplinen „sprachverarbeitende KI“ und „Sprachtechnologie“ liegen im wesentlichen in der deutlicheren Orientierung an linguistischen Methoden und der insgesamt stärkeren Betonung von Grundlagenproblemen, die nicht direkt auf spezifische Anwendungen bezogen sind. Die Überlappungen zwischen allen genannten Disziplinen sind jedoch relativ groß, und Konvergenztendenzen in vielen Einzelfragen sind zunehmend zu beobachten.

1.3.1 Die Rolle von Parsern in natürlichsprachlichen Systemen

Wesentliche Teile der Computerlinguistik beschäftigen sich mit theoretischen und praxisbezogenen Fragestellungen, die bei der Entwicklung von natürlichsprachlichen Systemen berücksichtigt werden müssen. Zu den wichtigsten Typen von natürlichsprachlichen Systemen zählen maschinelle Übersetzungssysteme und natürlichsprachliche Frage/Antwort-Systeme. In diesen Systemen spielen Parsingkomponenten eine wichtige Rolle. Abhängig von der Architektur des Gesamtsystems können auch mehrere verschiedene Parser eingesetzt werden (z.B. ein Parser für die morphologische Analyse von Wörtern und ein weiterer für die satzsyntaktische Analyse).

In solchen Systemen können Parsingkomponenten verschiedene Aufgaben erfüllen:

- Prüfung der Eingabe auf Wohlgeformtheit
- Aufbau von Strukturbeschreibungen, die die Weiterverarbeitung z.B. durch semantische Interpretationskomponenten oder Übersetzungsmodule unterstützen
- Disambiguierung der Eingabe (vgl. z.B. die Rolle der syntaktischen Analyse bei der Anaphernresolution, die wir im vorangegangenem Abschnitt bereits kurz diskutiert haben)
- Identifikation der Eingabe (z.B. bei stark verrauschten Sprachsignalen, bei denen zunächst mehrere Interpretationen des Wortlauts möglich erscheinen und

erst durch eine syntaktische Filterung eine Reduktion des Suchraums erzielt werden kann)

- Korrektur der Eingabe („robustes Parsing“)

Aber auch außerhalb des Bereichs der natürlichsprachlichen Systeme spielt Parsing eine wesentliche Rolle innerhalb der Computerlinguistik, so z.B. als unentbehrliches Werkzeug für die Entwicklung und Überprüfung von größeren Grammatiken.

1.3.2 Parsing natürlicher Sprache

Bei der Verarbeitung natürlicher Sprache ergeben sich Probleme, die beim Parsing im Bereich Compilerbau keine oder nur eine untergeordnete Rolle spielen. Diese Probleme resultieren direkt aus den besonderen Eigenschaften, die natürliche Sprachen im Unterschied zu künstlichen Sprachen wie z.B. Programmiersprachen aufweisen. Zu diesen Eigenschaften zählen u.a.:

- erheblich höhere syntaktische Komplexität und größere Strukturvielfalt
- zahlreiche Ambiguitäten
- Vagheit
- Sub- und Irregularitäten

Als Folge dieser Eigenschaften sind die Grammatiken für Fragmente natürlicher Sprachen deutlich umfangreicher und komplexer als Syntaxen für die Beschreibung von Programmiersprachen. Bereits bei der Wahl des Formalismus, in dem die Grammatik definiert werden soll, muß in der Regel über die im Compilerbau üblichen LL- und LR-Grammatiken (s. Kapitel 10) hinausgegangen werden. Während die Definition einer künstlichen Sprache bereits eine korrekte und vollständige Syntax voraussetzt, ist man bei der Verarbeitung von natürlichen Sprachen mit der Situation konfrontiert, daß selbst für ein eingeschränktes Sprachfragment keine *per definitionem* eindeutige und korrekte Grammatik vorliegt, sondern eine Rekonstruktion vorgefundener Strukturen, die lediglich nachträglich mit geeigneten Argumenten empirisch oder auch theoretisch gerechtfertigt werden kann.

Anhand des Beispiels (1-3) läßt sich exemplarisch zeigen, worin sich Parsing bei natürlichsprachlichen Ausdrücken von der syntaktischen Analyse eines Ausdrucks einer Programmiersprache unterscheidet:

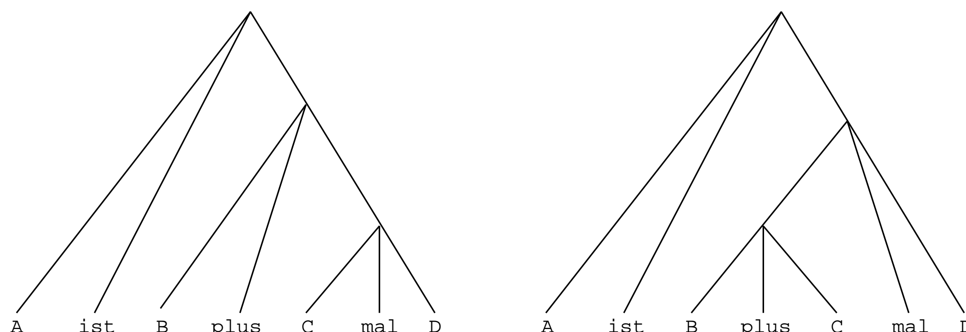
Beispiel (1-3)
In Beispiel (1-1) hatten wir anhand des folgenden PASCAL-Ausdrucks Grundzüge syntaxorientierter Kodegenierung illustriert:

A := (B + C) * D.

Das natürlichsprachliche Analogon zu diesem Ausdruck ist:

A ist B plus C mal D.

Es fällt auf, daß alle Sprachelemente des PASCAL-Ausdrucks genau einem Wort im natürlichsprachlichen Pendant entsprechen - bis auf eine Ausnahme: die Klammern. Durch das Fehlen der Klammern entsteht eine Mehrdeutigkeit, die sich durch die folgenden beiden Strukturbeschreibungen veranschaulichen läßt:



In dem Falle, daß dieser Satz als gesprochene Äußerung vorliegt, ist es vorstellbar, daß die Ambiguität durch eine entsprechende Intonationskurve oder durch Pausen aufgelöst wird. Solche prosodischen Markierungen sind jedoch in den seltensten Fällen wirklich eindeutig und stellen zudem keine Voraussetzung für die Grammatikalität der Äußerung dar, d.h. sie können auch fehlen (in der Schriftsprache wird eine solche prosodische Markierung ohnehin nicht sichtbar und auch eine analoge Markierung durch Interpunktion gibt es in diesem Fall nicht).
Derartige und andere Ambiguitäten, die bei der Verarbeitung natürlicher Sprachen aufgefunden und - sofern möglich - aufgelöst werden müssen, machen Algorithmen erforderlich, die entweder Ambiguitäten vollständig eliminieren oder *nondeterministisch* arbeiten. In den ersten Jahrzehnten der Computerlinguistik wurden zu diesem Zweck im wesentlichen sogenannte Backtracking-Parser verwendet, die wir im ersten Teil dieses Buches vorstellen werden. Einen entscheidenden Fortschritt stellte Ende der 60er Jahre die Entwicklung von Chart-Parsern dar[6], die im zweiten Teil des Buchs ausführlich dargestellt werden.

1.3.3 Grammatikformalismen

Eine maschinelle Analyse natürlicher Sprache ist nur möglich, wenn das erforderliche linguistische Wissen in einer zu diesem Zweck geeigneten Form vorliegt; d.h., es ist eine Repräsentation zu wählen, die es erlaubt, dieses Wissen präzise und eindeutig darzustellen. Die Annahme, Sprache sei ein wissensbasiertes, durch eine endliche Menge von Regeln zu modellierendes System, die lange Zeit die Entwicklung der modernen Linguistik prägte, kommt dieser Forderung entgegen. Sie führte dazu,

[6] Zu den *klassischen* Arbeiten in diesem Bereich gehören u.a.: [?], [?], [?] und [?].

daß in den vergangenen viereinhalb Jahrzehnten diverse Regeltypen formuliert wurden, die in einer Reihe verschiedener Grammatikformalismen Verwendung fanden[7].

Grundlegend sind die Arbeiten von N. Chomsky Ende der 50er Jahre, in denen er eine Hierarchie von formalen Grammatiken bzw. Syntaxen (*Chomsky-Hierarchie*) definierte. Es handelt sich dabei um einfache Ersetzungssysteme, die aus den folgenden Komponenten bestehen:

1. einer endlichen Menge V_N von nicht-terminalen Symbolen (grammatischen Kategorien);

2. einer endlichen Menge V_T von terminalen Symbolen (lexikalischen Einheiten);

3. einem ausgezeichneten Element S aus dem nicht-terminalen Vokabular, dem Startsymbol;

4. und einer endlichen Menge R von Ersetzungsregeln.

Eine Ersetzungsregel $\alpha \rightarrow \beta$ (zu lesen als: *ersetze α durch β*) besteht aus einer linken und einer rechten Seite. α und β sind *Ketten*, die aus Symbolen des terminalen und nicht-terminalen Vokabulars aufgebaut sind. Auf der linken Regelseite muß mindestens ein nicht-terminales Symbol vorkommen. Durch Anwendung dieser Regeln können aus dem Startsymbol neue Ketten abgeleitet werden.
Eine *Ableitung* besteht aus einer Folge von Ketten, die mit dem Startsymbol beginnt und bei der jede weitere Kette sich durch Anwendung einer Regel aus der vorangehenden Kette ergibt. Eine Regel ist auf eine Kette anwendbar, wenn sie eine Teilkette enthält, die mit der linken Seite der Regel identisch ist. Bei Anwendung der Regel wird diese Teilkette durch die rechte Seite der Regel ersetzt.

Eine aus dem Startsymbol einer Syntax G ableitbare Kette, die nur aus terminalen Symbolen besteht, wird als *Satz* bezeichnet; die Menge aller Sätze als die durch G festgelegte Sprache (notiert als: L(G)).

Beispiel (1-4)
Gegeben sei folgende einfache Grammatik $G_1 = <V_N, V_T, S, R>$:

$V_N =$ {S, NP, VP, DET, N, V}
$V_T =$ {der, Affe, schläft}
R = {S $\rightarrow$ NP VP, NP $\rightarrow$ DET N, VP $\rightarrow$ V, DET $\rightarrow$ der, N $\rightarrow$ Affe,
V $\rightarrow$ schläft}

Aus dieser Grammatik läßt sich der Satz *Der Affe schläft* wie folgt ableiten:

[7] Einen guten Überblick über verschiedene aktuelle Modelle gibt das Kapitel *Beschreibungsformalismen für sprachliches Wissen* in [?].

Ableitung:	*Aktion*:
S	Startsymbol
NP VP	Regel: S → NP VP
Det N VP	Regel: NP → DET N
der N VP	Regel: DET → der
der Affe VP	Regel: N → Affe
der Affe V	Regel: VP → V
der Affe schläft	Regel: V → schläft

Charakteristisch für die Regeln des voranstehenden Beispiels ist es, daß die linke Seite der Regeln aus einem einzelnen (nicht-terminalen) Symbol besteht. Regeln dieses Typs werden als *kontextfreie Regeln* bezeichnet und eine Syntax, die nur solche Regeln enthält, entsprechend als *kontextreie Syntax*[8]. Bei den Parsingalgorithmen, die wir in diesem Buch vorstellen werden, handelt es sich fast ausschließlich um Parsingalgorithmen für kontextfreie Syntaxen. Für das Interesse an Parsingalgorithmen für kontextfreie Syntaxen gibt es verschiedene Gründe:

- Kontextfreie Syntaxen sind mächtig genug, um die syntaktische Strukur natürlicher Sprachen fast vollständig zu beschreiben.
- Sie sind einfach zu entwickeln und zu modifizieren.
- Die Frage, ob ein Satz zu der von einer Grammatik festgelegten Sprache gehört (*Wortproblem*), ist entscheidbar.
- Sie erlauben die Entwicklung *effizienter* Parsingalgorithmen.

Aus diesen Gründen gibt es zahlreiche Grammatikformalismen, in denen kontextfreie Regeln verwendet werden: In der *Generativen Transformationsgrammatik* waren es ursprünglich kontextfreie Regeln, die dazu dienten, die *Tiefenstruktur* von Sätzen zu generieren, aus der dann durch Anwendung von *Transformationsregeln* ihre *Oberflächenstruktur* abgeleitet wurde (vgl. [**?**] und [**?**]). In neueren Versionen

[8]Die Chomsky-Hierarchie besteht aus vier Typen von Ersetzungssyntaxen, die sich durch die Form der Ersetzungsregeln unterscheiden:

- Typ-0-Syntaxen (*unbeschränkte* Syntaxen):
  Es gibt keine weiteren Beschränkungen für die linke bzw. rechte Regelseite.
- Typ-1-Syntaxen (*beschränkte* Syntaxen):
  Die rechte Regelseite darf nicht weniger Symbole als die linke Regelseite enthalten.
- Typ-2-Syntaxen (*kontextfreie* Syntaxen): s.o.
- Typ-3-Syntaxen (*reguläre* Syntaxen):
  Die linke Seite besteht aus einem nicht-terminalen Symbol. Die rechte Seite enthält maximal ein nicht-terminales Symbol, das in allen Regeln an erster Position (links-reguläre Syntaxen) oder an letzter Position (rechts-reguäre Syntaxen) steht.

der Transformationsgrammatik (*Government-&-Binding-Theorie* bzw. *Prinzipien-&-Parameter-Ansatz*) wird die Bildung von Tiefenstrukturen durch das Zusammenspiel verschiedener Wohlgeformtheitsbedingungen wie der $\overline{\text{X}}$-Konvention, des θ-Kriteriums und der im Lexikon für Wortformen verzeichneten Subkategorisierungsrahmen gesteuert (vgl. [**?**] und [**?**]).

Auch die meisten der seit Ende der 70er Jahre entwickelten unifikationsbasierten Grammatikformalismen verwenden entweder kontextfreie Regeln oder andere formale Mechanismen, die sich ähnlich wie die Wohlgeformtheitsbedingungen des *Prinzipien-&-Parameter-Ansatzes* als metasprachliche Beschreibung einer (endlichen) Menge von kontextfreien Regeln interpretieren lassen.

Kapitel 2

Begriffe & Repräsentationen

Im folgenden werden eine Reihe fundamentaler Begriffe eingeführt, verschiedene Kriterien zur Klassifikation von Parsingalgorithmen vorgestellt, Möglichkeiten der Repräsentation linguistischen Wissens diskutiert und die zur Beschreibung von Prozeduren und Algorithmen verwendeten Konventionen erläutert.

2.1 Einige grundlegende Begriffe

Was ist ein Parser? Ein Parser ist ein spezieller Typ von *abstrakter Maschine*. Diese Antwort ist zu knapp, um als befriedigend gelten zu können. Im Gegenteil: Sie mag empfindsame Gemüter mit ontologischen Skrupeln veranlassen zu fragen, ob wir nicht so unser Universum unnötig mit Objekten von recht fragwürdigem Status bevölkern. Ist es notwendig, neben uns so vertrauten Dingen wie Algorithmen und Programmen etwas Drittes anzunehmen, das wir als „abstrakte Maschine“ bezeichnen? Können wir uns nicht einfach mit einem „tertium non datur“ begnügen, indem wir „Parser“ als Bezeichnung für eine bestimmte Klasse von Algorithmen oder Programmen auffassen? Wenn nicht, wie ist dann die Beziehung zwischen den Begriffen **Algorithmus**, **abstrakte Maschine** und **Programm** zu charakterisieren?

Algorithmen. Ein Algorithmus ist ein durch eine präzise endliche Beschreibung gegebenes, grundsätzlich maschinell ausführbares Verfahren zur Lösung einer Klasse von Problemen. Uns interessieren Algorithmen zur syntaktischen Analyse natürlicher und auch formaler Sprachen, kurz *Analysealgorithmen* genannt. Zwei Typen von Analysealgorithmen sind zu unterscheiden: Erkennungsalgorithmen und Parsingalgorithmen.
Ein *Erkennungsalgorithmus* ist ein Verfahren, um für eine Kette w von Symbolen zu entscheiden, ob sie zu der durch eine Syntax G festgelegten Sprache L(G) gehört oder nicht, wobei G eine beliebige Syntax eines zuvor spezifizierten Typs sein kann.
Ein *Parsingalgorithmus* unterscheidet sich von einem Erkennungsalgorithmus darin, daß für den Fall, daß $w \in L(G)$ ist, nicht einfach ein boolescher Wert zurückgegeben

wird, sondern eine oder mehrere Strukturbeschreibungen für w. Anders formuliert: Wenn G eine beliebige Syntax, L(G) die durch G festgelegte Sprache und L die Menge aller über dem terminalen Vokabular von G formbaren Ketten ist, dann ist ein Recognizer R_G eine charakteristische Funktion für L(G). Ein Parser P_G kann als eine Funktion von L in die Potenzmenge von SB aufgefaßt werden, mit SB als Menge aller Strukturbeschreibungen, die G den Sätzen aus L(G) implizit zuordnet:

$$(1) \quad R_G(s_i) = \left\{ \begin{array}{l} 1 \ gdw. \ s_i \in L(G); \\ 0 \ sonst. \end{array} \right\}$$

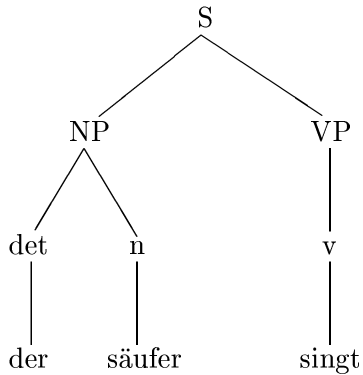
$$(2) \quad P_G(s_i) = \left\{ \begin{array}{l} p \in POT^+(SB) \ gdw. \ s_i \in L(G); \\ \emptyset \ sonst. \end{array} \right\}$$

Welches Repräsentationsformat für die Strukturbeschreibungen gewählt wird (indizierter Klammerausdruck, Strukturbaum, Links-/Rechtsparse, etc.), ist in diesem Zusammenhang irrelevant.

Beispiel (2-1)
Wenn man eine kontextfreie Syntax G betrachtet, die die folgenden indizierten Regeln enthält:

$$\begin{array}{lll} 1: S \rightarrow NP\ VP & 2: NP \rightarrow det\ n & 3: VP \rightarrow v \\ 4: det \rightarrow der & 5: n \rightarrow s\ddot{a}ufer & 6: v \rightarrow singt \end{array}$$

dann kann man für den Satz „*Der Säufer singt*“ folgende äquivalente Strukturbeschreibungen erhalten:



(a)

Strukturbaum

$[_{\mathrm{S}}[_{\mathrm{NP}}[_{\mathrm{det}}\ \text{der}][_{\mathrm{n}}\ \text{säufer}]][_{\mathrm{VP}}[_{\mathrm{v}}\ \text{singt}]]]$

(b)

indizierter Klammerausdruck

1 2 4 5 3 6

(c)

Linksparse

1 3 6 2 5 4

(d)

Rechtsparse[1]

[1]Intuitiv betrachtet scheinen die ersten beiden Strukturbeschreibungen mehr Informationen zu enthalten als die letzten beiden. Tatsächlich kann jedem Strukturbaum (jedem indizierten Klammerausdruck) genau ein Linksparse bzw. Rechtsparse zugeordnet werden und umgekehrt.

Ein Analysealgorithmus kann für alle Syntaxen eines Typs verwendet werden, die bestimmte formale Eigenschaften aufweisen: So gibt es z.B. Analysealgorithmen für deterministische kontextfreie Syntaxen, für kontextfreie Syntaxen in Chomsky-Normalform, für diverse Arten von kategorialen Syntaxen etc.

Abstrakte Maschinen. Erkennungs- und Parsingalgorithmen bilden die Grundlage für die Klassen von abstrakten Maschinen, die als „Recognizer“ bzw. „Parser“ bezeichnet werden. Recognizer/Parser resultieren aus der Verbindung eines Analysealgorithmus mit sprachspezifischen Daten; d.h. einer Syntax oder einer Syntax und einem Lexikon. Jeder Analysealgorithmus bildet so betrachtet die Grundlage für eine Vielzahl verschiedener abstrakter Maschinen. Die abstrakten Maschinen, die auf demselben Algorithmus basieren, werden als Maschinen eines Typs betrachtet.

Die Realisierung eines Parsers[2] durch Verbindung von Algorithmus und Syntax ist keine triviale Angelegenheit: In vielen Fällen werden z.B. die lexikalischen und syntaktischen Informationen aus ihrer durch den Formalismus der Syntax determinierten *externen Repräsentation* in eine für die Realisierung eines effizienten Parsers geeignetere *interne Repräsentation* transformiert[3]. In vielen Fällen werden Parser nicht direkt, sondern durch andere abstrakte Maschinen erzeugt, die als „Parser-Compiler“ oder „Parser-Generatoren“ bezeichnet werden. Ein Parser-Generator ist eine abstrakte Maschine 2. Stufe, die als Eingabe eine Syntax nimmt und aus ihr zusammen mit einem gegebenen Parsingalgorithmus einen Parser generiert (vgl. Abbildung (1-1)).

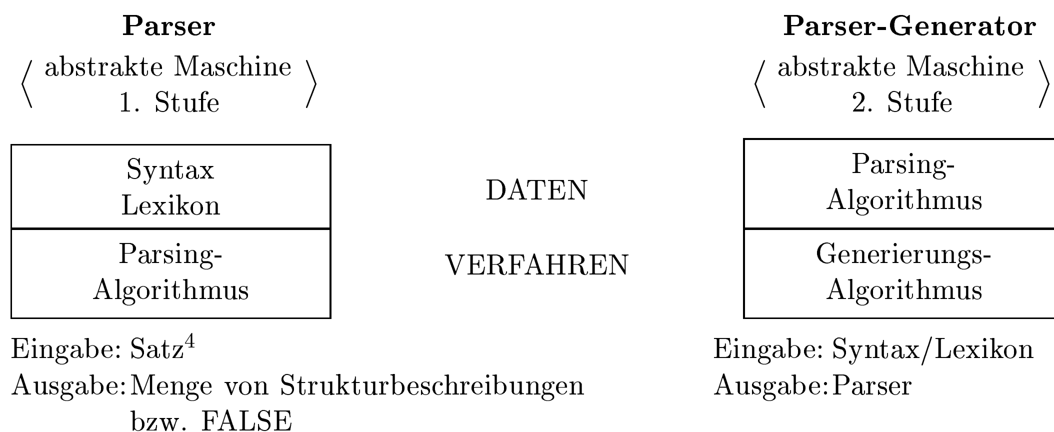


Abbildung (2-1)

[2] Auch wenn wir im folgenden nur von Parsern/Parsingalgorithmen reden, so gelten die Ausführungen ebenso für Recognizer/Erkennungsalgorithmen.

[3] Zu den möglichen Operationen bei dieser Transformation gehören unter anderem die Expansion von Makros, die Ordnung/Indizierung der Regelmenge, die Erstellung einer Parsingtabelle und die Compilierung der Regeln in Graphen.

[4] Nicht nur Sätze können Gegenstand syntaktischer Analysen sein: Texte, Textabschnitte, Teilsätze und einzelne Konstituenten bilden weitere Kandidaten.

Programme. Programme sind in der von uns verwendeten Terminologie **Realisierungen** von abstrakten Maschinen. Ähnlich wie ein Parsingalgorithmus die Grundlage für diverse Parser bilden kann, gibt es wiederum für einen Parser zahlreiche Realisierungsmöglichkeiten. Wir betrachten alle Programme, in denen ein Parsingalgorithmus mit demselben Satz Daten (Syntax/Lexikon) so verknüpft wird, daß sich die Programme gleich verhalten - dieselben Sätze werden akzeptiert bzw. zurückgewiesen, und es werden ihnen äquivalente Strukturbeschreibungen zugeordnet - als verschiedene Realisierungen *eines Parsers* und nicht als verschiedene Parser.

2.2 Zur Klassifikation von Parsingalgorithmen

Zur Klassifikation von Parsingalgorithmen bieten sich vor allem die folgenden drei Kriterien an:

1. *Verarbeitungsrichtung* - In welcher Reihenfolge werden die Worte eines Satzes bei seiner Analyse verarbeitet?

2. *Analyserichtung* - Welche Objekte bilden Ausgangspunkt bzw. Ziel der Analyse? Wie wird der Ablauf der Analyse gesteuert[5]?

3. *Suchstrategie* - Wie wird verfahren, wenn es verschiedene Analysemöglichkeiten gibt?

2.2.1 Verarbeitungsrichtung

Unidirektionale Verarbeitung. Für eine unidirektionale Verarbeitung von Sätzen gibt es zwei Möglichkeiten: Bei einer *Links-rechts-Verarbeitung* beginnt die Analyse mit dem ersten Wort eines Satzes und arbeitet ihn dann Wort für Wort von links nach rechts ab. Diese Verarbeitungsrichtung wird für die meisten unidirektionalen Parser gewählt und eignet sich unter anderem für die Entwicklung von *On-line*-Parsern; d.h. für Parser, die mit der Analyse eines Satzes beginnen, während er eingelesen wird. Bei einer *Rechts-links-Verarbeitung* wird der Satz vom Satzende ausgehend in umgekehrter Reihenfolge verarbeitet.

Bidirektionale Verarbeitung. Bei einem bidirektionalen Verfahren gibt es entweder mit Satzanfang und Satzende zwei feste Startpunkte für die Verarbeitung des Satzes, oder es werden zunächst nach bestimmten Kriterien ein/mehrere Worte markiert (sogenannte *Inseln*), von denen aus dann die Analyse uni- oder bidirektional fortschreitet, bis schließlich der vollständige Satz verarbeitet ist. Die Entwicklung bidirektionaler Algorithmen wurde vor allem durch Probleme motiviert, die bei der Verarbeitung gesprochener Sprache auftreten können (vgl. Kapitel 8).

[5] Statt „Analyserichtung“ wird auch der Begriff „rule invocation strategy“ [?] verwendet.

2.2.2 Analyserichtung

Unidirektionale Analyse. Die meisten Algorithmen arbeiten nicht nur bei der Verarbeitung des Satzes, sondern auch bei der Analyse selbst unidirektional:

Top-down arbeitende Algorithmen beginnen bei der Analyse eines Satzes w mit dem Startsymbol der Syntax und suchen nach einer **Ableitung** für w. Sie sind *zielgesteuert* und terminieren, sobald der abgeleitete Ausdruck mit w übereinstimmt[6]. *Bottom-up* arbeitende Algorithmen dagegen sind *datengesteuert*: Ausgehend von w suchen sie nach einer **Reduktion** für w; d.h., sie terminieren, sobald der reduzierte Ausdruck nur noch aus dem Startsymbol besteht. Eine Schwäche von Top-down-Algorithmen liegt darin, daß Kategorien vorgeschlagen werden können, die nicht terminierbar sind; bei Bottom-up-Algorithmen besteht dagegen die Gefahr, daß Konstituenten gebildet werden, die nicht weiter reduzierbar sind.

Bidirektionale Analyse. Neben den unidirektionalen finden sich auch bidirektionale Algorithmen, die bei der Analyse die beiden oben beschriebenen Verfahren miteinander kombinieren, um so die gerade beschriebenen Nachteile von reinen Top-down- bzw. Bottom-up-Algorithmen zu vermeiden.

Die Unterschiede zwischen den verschiedenen Verarbeitungs- und Analyserichtungen werden deutlich, wenn man betrachtet, in welcher Weise bei der Analyse eines Satzes $w \in L(G)$ ein Strukturbaum für w generiert wird.

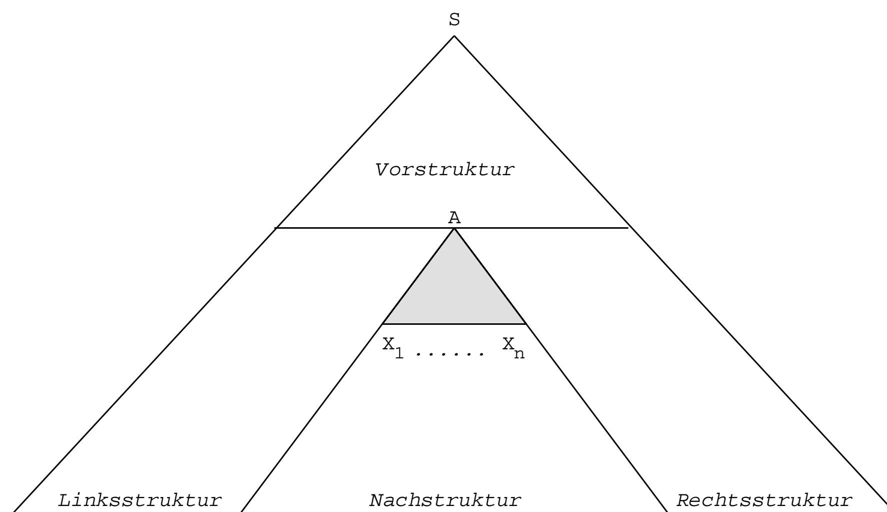


Abbildung (2-2)

Jeder bei der Ableitung von w verwendeten Regel entspricht ein **lokaler Baum**, d.h., der Strukturbaum kann als eine Menge von lokalen Bäumen aufgefaßt werden, die in bestimmter Weise miteinander verknüpft sind. Greift man einen dieser lokalen

[6] Wir spezifizieren hier nur die Terminierungsbedingung für den Fall, daß $w \in L(G)$.

Bäume heraus, z.B. den, der der Anwendung einer Regel $A \rightarrow X_1 \ldots X_n$ entspricht, so läßt sich jetzt der Strukturbaum in Teilstrukturen zerlegen (vgl. Abbildung 1-2). Die unidirektionalen Verarbeitungs- und Analysevarianten lassen sich nun wie folgt unterscheiden[7]:

1. Wird eine Regel nur dann angewendet, wenn die Linksstruktur vollständig, die Rechtsstruktur dagegen noch gar nicht bekannt ist, wird eine links-rechts Verarbeitung realisiert (im umgekehrten Fall eine rechts-links Verarbeitung).

2. Wird eine Regel nur dann angewendet, wenn die Vorstruktur vollständig, die Nachstruktur dagegen noch gar nicht erkannt ist, wird der Satz top-down analysiert (im umgekehrten Fall bottom-up).

2.2.3 Suchstrategien

Deterministische & nicht-deterministische Algorithmen. Für bestimmte Klassen von kontextfreien Syntaxen ist es möglich, deterministische Parsingalgorithmen zu verwenden, die sich durch ein hohes Maß an Effizienz auszeichnen (z.B. für LL- & LR-Sprachen). Deterministisches Parsen ist aber nur dann möglich, wenn in jeder Situation die anzuwendende Regel eindeutig identifizierbar ist, d.h. nur eine Analysemöglichkeit betrachtet werden muß. Für viele kontextfreie Sprachen gibt es allerdings keine Syntax, die die Verwendung solcher deterministischen Parsingalgorithmen erlaubt, und gerade natürliche Sprachen gehören, sofern man sie überhaupt als kontextfreie Sprachen betrachtet, sicher in diese Klasse.

Nicht-deterministische Parsingalgorithmen müssen über eine Kontrollstruktur verfügen, die es erlaubt, in den Situationen, in denen mehrere Regeln angewendet werden können, diese verschiedenen Analysemöglichkeiten zu verfolgen. Nicht-deterministisches Parsen kann in dieser Hinsicht als ein spezielles Suchproblem aufgefaßt werden.

Parsing als Suchproblem. Es gibt eine Klasse von Problemen, die sich als Suchprobleme der folgenden Art charakterisieren lassen:

Es gibt eine (eventuell nicht-endliche) Zahl von Zuständen. Diese Zustände bilden den sogenannten *Zustandsraum.* Einer dieser Zustände ist als *Startzustand* ausgezeichnet, andere Zustände als *Zielzustände*. Der Übergang von einem Zustand Z in einen Zustand Z' ist durch bestimmte, problemspezifische Operationen möglich.

Der Zustandsraum läßt sich als ein (gerichteter) Graph repräsentieren, in dem die Knoten die Zustände des Zustandsraums abbilden und dessen Kanten zwei Knoten Z, Z' miteinander verbinden, wenn Z' von Z aus erreichbar ist.

Diese Repräsentation erlaubt es, die Probleme als Suchprobleme zu betrachten: Die Suche nach einer Lösung wird so, abhängig von der Aufgabenstellung und dem Typ des verwendeten Suchalgorithmus, zu der Suche nach *einem* Weg oder dem

[7] Vgl. [?, S.81-83].

kürzesten bzw. *günstigsten* Weg oder nach *allen* Wegen, die vom Startzustand zu einem Zielzustand führen.

Auch das Problem, einem Satz die Strukturbeschreibungen zuzuordnen, die sich relativ zu einer gegebenen Syntax für ihn bilden lassen, kann als ein derartiges Suchproblem aufgefaßt werden: Der Satz und das Startsymbol der Syntax bilden Start- und Zielzustand, die Regeln der Syntax die problemspezifischen Operationen und die durch sie generierbaren Ausdrücke die übrigen Zustände des Zustandsraums.

Für Top-down-Algorithmen z.B. besteht der Zustandsraum aus allen Links-/Rechtssatzformen, die innerhalb der zugrundeliegenden Syntax gebildet werden können. Der Startzustand ist das Startsymbol dieser Syntax. Die Menge der potentiellen Zielzustände ist die Menge aller Ketten, die sich über dem terminalen Vokabular bilden lassen; aktueller Zielzustand ist die Satzform, die mit dem zu analysierenden Satz übereinstimmt (vgl. Beispiel (3-5)).

Für Bottom-up-Algorithmen dagegen enthält der Zustandsraum alle möglichen Reduktionen, die relativ zur Syntax gebildet werden können. Die Menge der potentiellen Startzustände besteht aus allen über dem terminalen Vokabular generierbaren Ketten. Den aktuellen Startzustand bildet der zu analysierende Satz, und der Zielzustand ist das Startsymbol der Syntax. Das folgende Schema skizziert einen prozeduralen Rahmen, in den sich verschiedene Suchalgorithmen einbetten lassen:

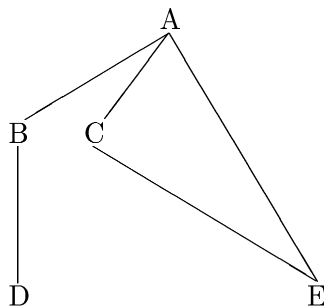
PROZEDUR GRAPH_SUCHE
Schema

DATEN: Ein Graph G.
EINGABE: Ein Knoten K_s aus G.
AUSGABE: *True/False.*
ARBEITSSTRUKTUREN:
KNOTEN - Eine Menge von noch zu bearbeitenden Knoten aus G.
Anfangswert: $\{K_s\}$.
NEUK - Eine Menge von Knoten aus G.
Anfangswert: $\emptyset$.

METHODE:
Wiederhole:
Wenn <KNOTEN = $\emptyset$>
Dann <RETURN(*False*)>
Wähle einen beliebigen Knoten k $\in$ KNOTEN aus.
Wenn <k repräsentiert einen Zielzustand>
Dann <RETURN(*True*)>
Sonst <NEUK $\Leftarrow$ {k' | k' ist von k aus direkt erreichbar $\wedge$
k' ist kein Vorgänger von k }>
KNOTEN $\Leftarrow$ (KNOTEN - {k}) $\cup$ NEUK.

Die Bedingung *k' ist kein Vorgänger von k* stellt sicher, daß die Suche auch bei zyklischen Graphen terminiert. Um aus der Prozedur **GRAPH_SUCHE** einen Algorithmus zu erhalten, ist es notwendig zu spezifizieren, nach welchem Kriterium der zu expandierende Knoten aus KNOTEN auszuwählen ist.
Zwei wichtige Suchalgorithmen bilden die *Depth-first-* und die *Breadth-first-*Suche:

Beispiel (2-2)



depth-first (links-rechts) : A B D C E
breadth-first (links-rechts) : A B C E D

Bei einer Depth-first-Suche wird ein begonnener Pfad immer weiter verfolgt, und andere Pfade werden erst dann berücksichtigt, wenn der untersuchte Pfad nicht weiter expandiert werden kann: Es wird also ein Knoten k $\in$ KNOTEN gewählt, der Nachfolger des zuletzt untersuchten Knotens k' ist. Bei einer Breadth-first-Suche dagegen werden alle bereits untersuchten Pfade ihrer Länge nach geordnet und immer der kürzeste erweitert. Es wird also in jedem Schritt ein Knoten k gewählt, der Nachfolger des Knotens k' ist, der den Endknoten des kürzesten Pfades bildet. Die Depth-first- und die Breadth-first-Suche gehören zu der Klasse der *uninformierten* („uninformed") Suchalgorithmen, die den zu expandierenden Knoten 'blind' auswählen. *Heuristische* Suchalgorithmen dagegen verwenden problemspezifische Informationen, um eine Bewertungsfunktion zu formulieren, die es ihnen erlaubt, aus KNOTEN in jedem Schritt den vielversprechendsten Knoten auszuwählen. Zu den heuristischen Suchverfahren gehören z.B. sogenannte *Best-first-* und *Beam-search-*Algorithmen.

Parallele versus sequentielle Suche. Es gibt einen weiteren Aspekt, der im Zusammenhang mit der Klassifikation von Algorithmen relativ zu der verwendeten Suchstrategie zu beachten ist: Werden andere verfügbare Analysemöglichkeiten sequentiell oder parallel verarbeitet?

Bei sequentiellen Algorithmen ist immer nur ein 'Prozeß' aktiv; d.h. es wird zu einem Zeitpunkt immer nur eine Analysemöglichkeit verfolgt. Dafür müssen die gerade nicht verfolgten Analysemöglichkeiten gespeichert werden, damit sie bei der

Terminierung des aktuellen Prozesses zur Verfügung stehen; bei parallelen Algorithmen dagegen sind verschiedene Prozesse zur gleichen Zeit aktiv, die überwacht und eventuell synchronisiert werden müssen. Allerdings eignet sich auf den ersten Blick nicht jede Suchstrategie gleichermaßen für eine parallele Realisierung: Es ist offensichtlich, daß die Breadth-first-Suche sowohl sequentiell als auch parallel ausgeführt werden kann. Sie eignet sich - sofern geeignete Soft- und Hardware zur Verfügung stehen - als Ausgangspunkt für die Realisierung echter paralleler Maschinen. Die Formulierung eines parallelen, depth-first arbeitenden Algorithmus dagegen ist aus konzeptuellen Gründen weniger naheliegend.

2.3 Komplexität und Effizienz

Man kann die Auffassung vertreten, daß sich als einzige anwendungsunabhängige Adäquatheitskriterien für Parsingalgorithmen und deren Implementierungen (neben solchen selbstverständlichen Voraussetzungen wie Korrektheit bzw. Lauffähigkeit) die Faktoren Komplexität bzw. Effizienz anführen lassen. Den Begriff Komplexität beziehen wir grundsätzlich auf algorithmische Probleme, während wir von Effizienz im Zusammenhang mit tatsächlich implementierten Systemen sprechen.

Die Komplexität von Parsingalgorithmen wird in der Regel durch ihr Worst-case-Verhalten definiert, d.h. durch den Raum- und Zeitbedarf des Algorithmus im ungünstigsten Fall (z.B. bei der Verwendung einer maximal mehrdeutigen Grammatik oder bzw. und einer maximal mehrdeutigen Eingabekette). Bei derartigen Komplexitätsüberlegungen wird über die Größe und Struktur der zugrundeliegenden Grammatik sowie über die Eigenschaften der Eingaben (z.B. ihre Länge) abstrahiert, d.h. diese Faktoren gehen als unbekannte *Konstanten* in die Komplexitätsberechnungen ein, während die Länge der Eingabekette typischerweise als *Variable* behandelt wird [8].

Die Bestimmung der Zeit- und Raumkomplexität von Algorithmen durch Worst-case-Untersuchungen ist von großem theoretischen Interesse, für die konkrete Aufwandsabschätzung bei implementierten Parsingsystemen dagegen besitzt sie nur eingeschränkte Aussagekraft, da sie Faktoren wie die Komplexität des Lexikonzugriffs, Merkmale der verwendeten Grammatiken und strukturelle Eigenschaften der zu verarbeitenden Satzmengen nicht hinreichend berücksichtigt. Diese Aspekte werden von einer Reihe empirischer Untersuchungen über die Effizienz und den De-facto-Speicherbedarf bei der Verarbeitung realistischer Grammatiken und Eingabeketten berücksichtigt.

Im Zusammenhang mit der Beschreibung der einzelnen Parsingalgorithmen haben wir in den entsprechenden Kapiteln auch Angaben über die Worst-case-Komplexität des jeweiligen Algorithmus gegeben. Ergänzend wollen wir deshalb an dieser Stelle

[8] Ein gut lesbarer Überblick über das Thema Komplexität von Sprachverarbeitungsmodellen findet sich in: [?] und [?, Kap. 12].

einige neuere Resultate von Tests mit tatsächlich implementierten Parsingsystemen unter (zum Teil) annähernd realistischen Bedingungen diskutieren.

2.3.1 Zur Bewertung von empirischen Tests von Parsingsystemen

Wir haben oben erläutert, warum Komplexitätsberechnungen allein keine hinreichenden Kriterien für die Abschätzung der Effizienz von Parsingsystemen liefern können und deshalb um empirische Tests unter realistischen oder prototypischen Bedingungen erweitert werden sollten. Es scheint uns aber erforderlich zu sein, zunächst auf einige Probleme hinzuweisen, die sich bei der Beurteilung von derartigen empirischen Resultaten ergeben. Wir wollen keinesfalls in Zweifel ziehen, daß es sich bei den meisten Arbeiten über die Performanz von Parsingalgorithmen und -systemen um ausgesprochen nützliche Resultate handelt, aber - um es vorwegzuschicken - es stehen wohl zur Zeit noch keine unumstrittenen und standardisierten Verfahren zur Verfügung, mit denen die Effizienz von Parsingsystemen verläßlich abzuschätzen ist. Da aber solche klar festgelegten Kriterien für Parsingsysteme bislang nicht vorlagen und deshalb auch noch keine Verwendung gefunden haben, sind die Ergebnisse der vorliegenden empirischen Untersuchungen nur sehr bedingt miteinander vergleichbar und können nur mit größter Vorsicht verallgemeinert werden. Die meisten vorliegenden Arbeiten zum Thema Effizienz von Parsingsystemen berücksichtigen jeweils nur genau einen Grammatikformalismus, genau eine linguistische *Ebene* (zumeist Syntax) und/oder nur genau eine Sprache.

Ein weiteres Problem stellen absolute Zeitangaben dar, die natürlich abhängig von der verwendeten Hardware sind und deshalb - obwohl scheinbar objektiv vergleichbar - doch nur von sehr beschränktem Wert sind. Besser geeignet für Zwecke des Vergleichs können gegebenenfalls Maße wie die Anzahl der Anwendungen der fundamentalen Regel in einem Chartparser oder die Anzahl der logischen Operationen eines Prolog-Interpreters sein (vgl. [**?**]). Aber auch in diesem Falle können implementationsspezifische Faktoren, die die Ergebnisse verfälschen, nicht völlig ausgeschlossen werden.

Schließlich werden in den meisten Arbeiten zur Effizienz von Systemen nur Resultate über die Performanz des Parsers bei *wohlgeformten* Eingaben berichtet. Unter Umständen kann jedoch der Raum- und Zeitbedarf eines Systems bei ungrammatischen Eingaben mindestens ebensoviel zu einer Performanzbeurteilung beitragen wie das Verhalten bei wohlgeformten Sätzen.

In den meisten Arbeiten zur empirischen Überprüfung der Effizienz von Parsingalgorithmen wird übereinstimmend festgestellt, daß die Leistungsfähigkeit eines Systems massiv von der Grammatik und der zu verarbeitenden Satzmenge abhängen. Einige Zitate mögen als Illustration genügen:

> *Our experiments confirm the results of Shann and Wiren that parsing efficiency depends heavily on the grammar, the language, the grammar formalism, and the sentence set.* [**?**]

> *The main conclusion to be drawn from the experiments discussed above is that the influence of the grammar can hardly be underestimated.* [?]
>
> *The recognition of sentences is highly influenced by the choice of sentences.* [?]

Angesichts der Tatsache, daß die meisten Untersuchungen auf unterschiedlichen Grammatikformalismen und Satzmengen beruhen, kann es nicht verwundern, daß die Resultate insgesamt äußerst hetegoren ausfallen und einander zum Teil sogar klar widersprechen:

Russi kommt einerseits zu dem Ergebnis, daß einfache Top-down-Algorithmen (ohne Look-ahead usw.) einfachen Bottom-up-Algorithmen überlegen sind, während andererseits Top-down-Parser mit Look-ahead in etwa gleich effizient ausfallen wie Left-Corner-Parser mit Look-ahead und Top-down-Filterung. Bouma und van Noord vergleichen CUG- und DCG-basierte Grammatiken[9] hinsichtlich der Frage, in welchem Verhältnis die Effizienz von head-corner-basierten Algorithmen und left-corner-basierten Algorithmen zueinander stehen. Die von ihnen berichteten Resultate besagen, daß Head-corner-Algorithmen bei bestimmten Grammatiken und Eingabeketten zu einer besseren Performanz führen, in manchen Fällen jedoch auch nicht. Außerdem bemerken Bouma und van Noord, daß Erreichbarkeitstafeln unter bestimmten Umständen sogar zu einer Verschlechterung der Effizienz führen, z.B. weil bei nicht oder nur geringfügig mehrdeutigen Eingaben der Zugriff auf die Erreichbarkeitstafeln mehr Aufwand erfordert als die direkte Überprüfung mit dem normalen Parsingsalgorithmus. Zu einem ähnlichen Resultat kommt Verlinden, die eine etwas bessere Effizienz eines Left-corner-Parsers gegenüber einem Head-corner-Parser experimentell feststellt. Haas[10], schließlich, berichtet, daß ein von ihm entwickelter Bottom-up-Parser für einen unifikationbasierten Formalismus ohne zusätzliche Prediktionen schneller arbeitet als mit.

Angesichts solcher Ergebnisse muß wohl zusammenfassend konstatiert werden, daß die Auswahl eine passenden Parsingalgorithmus für eine bestimmte Grammatik und eine festgelegte Satzmenge zwar durchaus zu deutlichen Optimierungen führen kann, daß aber unabhängig von diesen Voraussetzungen keine präzisen und verallgemeinerbaren Aussagen über die praktische Effizienz von Parsingmodellen gemacht werden können. Die oben kurz referierten Untersuchungsergebnisse gehen also in dieser Hinsicht wohl nicht substantiell über die Resultate hinaus, die bereits seit den frühen 80er Jahren bekannt sind (vgl. z.B. [?]): Experimentelle Untersuchungen der Effizienz von Parsingsystemen, die die Eigenschaften der Testsatzmenge und der verwendeten Grammatik unberücksichtigt lassen, sind unbrauchbar.

---

[9] CUG steht für *Categorial Unification Grammar*, einem auf der Kategoralgrammatik basierenden Grammatikformalismus und DCG für *Definite-Clause-Grammar*. DCGs werden in Kapitel 12 behandelt.

[10] Vgl. [?] und [?].

2.4 Repräsentation des linguistischen Wissens

In der traditionellen Definition von Phrasenstrukturgrammatiken, wie man sie vor allem in der Literatur zur Theorie der formalen Sprachen findet, wird nicht zwischen dem Regelsystem einer Grammatik und dem Lexikon differenziert: Terminale Symbole werden durch (terminale) Regeln der Form $X \rightarrow a$, mit $X \in V_N$ und $a \in V_T$, eingeführt. Dasselbe gilt auch für zahlreiche linguistische Arbeiten älteren Datums. Es gibt allerdings gute Gründe für die inzwischen übliche Praxis, eine solche explizite Trennung vorzunehmen, die hier kurz skizziert werden sollen:

- Zum einen ist das Lexikon in den vergangenen Jahren immer mehr in das Zentrum linguistischer Forschung gerückt und wird längst nicht mehr nur als unstrukturierte Auflistung von Wortformen aufgefaßt. Man nimmt für das Lexikon spezifische, linguistisch motivierte Strukturprinzipien und Organisationsformen an, die sich von denen anderer grammatischer Komponenten unterscheiden.

- Weitere Gründe für die Trennung zwischen Regelapparat und Lexikon sind eher praktischer Natur. Insbesondere bei größeren natürlichsprachlichen Systemen sind häufig Lexika mit mehreren zehntausend Einträgen erforderlich, für deren Verwaltung effiziente Repräsentationsformate und Zugriffsstrategien (zum Beispiel Diskriminationsnetzwerke) erforderlich sind. Auch unter dem Gesichtspunkt der flexiblen Erweiterung von Grammatiken (Modifikation, Austausch von Komponenten) ist diese Modularisierung wünschenswert.

Es stellt sich natürlich die Frage, welche Konsequenzen mit dieser primär methodologisch motivierten Trennung von Syntax und Lexikon für die Beschreibung von Parsingalgorithmen verbunden sind. Zunächst lassen sich einige formale Konsequenzen konstatieren:

1. Da die Verbindung zwischen lexikalischen Einheiten und syntaktischer Beschreibung nicht mehr durch Regeln der Syntax, sondern durch Einträge im Lexikon erfolgt, können nun die lexikalischen Kategorien als die terminalen Symbole der Syntax betrachtet werden; d.h. eine Syntax G besteht aus einer Menge syntaktischer und einer Menge lexikalischer Kategorien, dem Startsymbol und einer endlichen Menge von Ersetzungsregeln.

2. Die Trennung von Syntax und Lexikon erfordert eine Modifikation der für kontextfreie Syntaxen üblichen Sprachdefinition:

   Wenn $G = <V_N, V_T, S, R>$ eine kontextfreie Syntax und L ein Lexikon ist, dann besteht die durch G und L festgelegte Sprache aus der Menge aller Ketten $w = w_1 \ldots w_n$ $(n \geq 0)$ für die gilt:

   (a) $S \overset{*}{\Rightarrow} x_1 \ldots x_n$, $x_i \in V_T$ und

(b) $w_i \in x_i$, für $1 \leq i \leq n$;

d.h. aus dem Startsymbol der Syntax muß eine Kette von lexikalischen Kategorien ableitbar sein, so daß das i-te Wort Element der i-ten lexikalischen Kategorie ist.

Die durch G festgelegte Sprache besteht also aus der Menge aller Sätze, in die sich eine aus dem Startsymbol von G ableitbare Kette von lexikalischen Kategorien überführen läßt, indem jede lexikalische Kategorie durch eine der Wortformen ersetzt wird, die ihr das Lexikon zuordnet.

Auch wenn in den letzten Jahren das Interesse an effizienten und linguistisch motivierten Lexikonrepräsentationsformalismen in der Computerlinguistik stark zugenommen hat, bleibt festzustellen, daß für die in den folgenden Kapiteln beschriebenen Algorithmen die konkrete Organisationsform des Lexikons eine stark vernachlässigbare Größe bildet. Natürlich ist die Realisierung der innerhalb eines Recognizers oder Parsers erforderlichen Zugriffsoperationen stark abhängig von der für das Lexikon gewählten Repräsentation. Für die Formulierung eines Parsingalgorithmus dagegen ist nur entscheidend, daß ein derartiger Zugriff möglich ist, nicht aber, wie er im Detail zu realisieren ist. Aus diesem Grund verzichten wir bei der Beschreibung von Prozeduren und Algorithmen auf eine explizite funktionale Differenzierung zwischen der Anwendung syntaktischer Regeln und Lexikonzugriffen; d.h. der Zugriff auf einen Lexikoneintrag wird als Anwendung einer (lexikalischen) Regel betrachtet. Der Vorteil dieses Verfahrens liegt darin, daß es eine redundanzfreie Darstellung ermöglicht. Anderenfalls wären alle Formulierung der Art '*Wenn es eine Regel gibt, die die Bedingungen ... erfüllt, dann ...*', die bei der Beschreibung von Analysealgorithmen relativ häufig anzutreffen sind, mit einer Ergänzung zu versehen wie '*... oder einen Eintrag im Lexikon L gibt, der die Bedingungen ... erfüllt, ...*'.

Bei der Präsentation von Beispielen und für die Implementierung der Algorithmen setzen wir folgende sehr einfache Lexikonorganisation voraus:

$$\begin{aligned}\text{Lexikon} = {}& <<\text{Wortform}_1\ \text{Eintrag}_1^1\ \text{Eintrag}_1^2 \ldots >\\ & <\text{Wortform}_2\ \text{Eintrag}_2^1\ \text{Eintrag}_2^2 \ldots >\\ & \ldots >\end{aligned}$$

$$\text{Wortform} = \text{Symbol}$$

$$\begin{aligned}\text{Eintrag} = {}& \text{Kategoriensymbol} \mid\\ & \{\text{Merkmalsspezifikation}_1, \text{Merkmalsspezifikation}_2, \ldots\}\end{aligned}$$

Merkmalsspezifikation = (Merkmal Wert)[11]

[11] Es gibt natürlich verschiedene Möglichkeiten, das Lexikon zu organisieren. Adäquate Zugriffsoperationen vorausgesetzt, kann für die Beschreibung von Parsingalgorithmen die aktuelle Organisation des Lexikons vernachlässigt werden.

Jeder Wortform ist also eine Folge von *Einträgen* zugeordnet, die die verschiedenen Lesarten für diese Wortform repräsentieren. Ein Eintrag besteht in den „Spielzeuglexika", die in den ersten drei Abschnitten verwendet werden, nur aus einem Kategoriensymbol; Merkmalsspezifikationen werden erst bei der Behandlung von unifikationsbasierten Syntaxen im letzten Abschnitt eingeführt.

Grundsätzlich sollte beachtet werden, daß es weder erforderlich noch im Einzelfall besonders sinnvoll ist, innerhalb eines Parsers linguistisches Wissen exakt in der Form zu repräsentieren, die in den externen Wissensquellen Lexikon und Syntax verwendet wird. Verschiedene Typen von Algorithmen können verschiedene Zugriffsformen auf Syntax und Lexikon erfordern und damit auch unterschiedliche Repräsentationsprinzipien für dieses Wissen nahelegen:

- Syntax:
  Für Top-down-Algorithmen z.B. liegt es nahe, Regeln mit identischer linker Seite zusammenzufassen; bei von links nach rechts arbeitenden Bottom-up-Algorithmen andererseits könnten Regeln mit identischer linker Ecke (erstes Symbol der rechten Seite) in Bäume kompiliert werden.

- Lexikon:
  Bei Top-down-Algorithmen ist zu entscheiden, ob es in einer (lexikalischen) Kategorie K einen Eintrag für ein Wort w gibt; bei Bottom-up-Algorithmen dagegen sind für ein Wort w alle lexikalischen Kategorien des Lexikons zu finden, in denen es einen Eintrag für w gibt.

Beispiel (2-3)

Wenn G eine kontextfreie Syntax ist, die die folgenden Regeln enthält:

S → NP VP	VP → v NP	VP → v
NP → det n	NP → det adj n	NP → n

dann erhalten wir folgende interne Repräsentation für die Syntax:

(a) Top-down-Repräsentation	(b) Bottom-up-Repräsentation
<S <NP VP>>	<NP <VP S>>
<NP <det n> <det adj n> <n>>	<det <n NP> <adj n NP>>
<VP <v NP> <v>>	<n <NP>>
	<v <NP VP> <VP>>

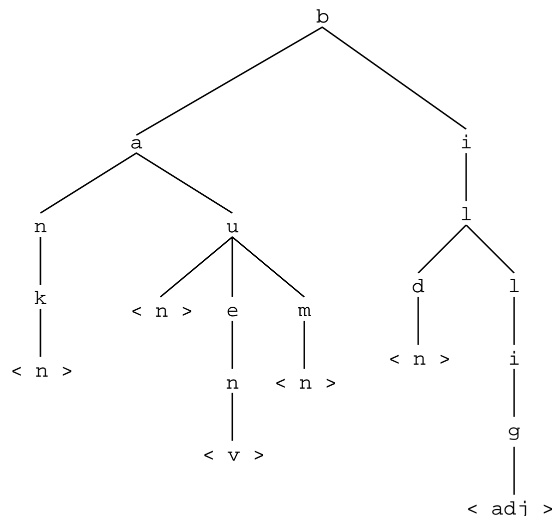
Eine effiziente Repräsentationsform für lexikalische Informationen bilden *Tries* (vgl. Appendix D.3), die in (computer)linguistischen Arbeiten auch häufig als *Buchstabenbäume* („letter trees") bezeichnet werden:

Beispiel (2-4)

Wenn L ein Lexikon ist, das u.a. die Einträge

<bauen v>	<bau n>	<baum n>
<bank n>	<bild n>	<billig adj>

enthält, dann läßt sich dieser Ausschnitt von L durch folgenden *Trie* repräsentieren:



2.5 Die Beschreibung von Algorithmen und Prozeduren

Zur Beschreibung von Algorithmen und Prozeduren verwenden wir, wie schon die Prozedur **GRAPH_SUCHE** aus Abschnitt 1.2.3 illustriert, eine Notation, die sich an der von T.Winograd entwickelten Sprache DL („description language“) orientiert[12]. Wir erläutern im folgenden die wichtigsten der von uns verwendeten Notationskonventionen. Jede Beschreibung einer Prozedur bzw. eines Algorithmus besteht aus folgenden drei Teilen:

- einem Beschreibungskopf,
- einem Spezifikationsteil und
- einem Operationsteil.

2.5.1 Der Beschreibungskopf

Der Beschreibungskopf spezifiziert den Typ und Namen des beschriebenen Objekts, gefolgt von einer (möglicherweise leeren) Folge von Attributen, die die wichtigsten Eigenschaften des Objekts spezifizieren:

PROZEDUR *Name*
$attribut_1 \ldots attribut_n$

ALGORITHMUS *Name*
$attribut_1 \ldots attribut_n$

[12]Vgl. [?, S.416-63].

2.5.2 Der Spezifikationsteil

In diesem Teil der Beschreibung wird festgelegt, welche Eingaben das beschriebene Objekt erwartet (**EINGABE**) und welche Resultate es generieren kann (**AUSGABE**). Der Spezifikationsteil endet mit der Deklaration der im Operationsteil verwendeten lokalen Variablen (**ARBEITSSTRUKTUREN**). Parameter- und Variablennamen werden durch Großschreibung hervorgehoben. Für lokale Variablen kann ein Anfangswert spezifiziert werden.

EINGABE: $PARAMETER_1$ - $Objektbereich_1$
⋮ ⋮
$PARAMETER_n$ - $Objektbereich_n$
AUSGABE: $Wert_1 \dots Wert_n$
ARBEITSSTRUKTUREN:
$VARIABLE_1$ - $Objektbereich_1$
Anfangswert: $Wert_1$
⋮ ⋮
$VARIABLE_m$ - $Objektbereich_m$
Anfangswert: $Wert_m$

2.5.3 Der Operationsteil

Der Operationsteil beginnt mit dem Schlüsselwort **METHODE**. Was folgt, ist eine (endliche) Sequenz von Anweisungen. Es gibt einfache und komplexe Anweisungen. Komplexe Anweisungen werden aus einfachen Anweisungen mit Hilfe der folgenden Kontrollstrukturen gebildet:

1. Iteration
   (a) **Wiederhole**: <*Anweisung*>$^+$
   (b) **Wiederhole bis** <*Bedingung*>: <*Anweisung*>$^+$
   (c) **Für** *Variable* = *Anfangswert*, ..., *Endwert*: <*Anweisung*>$^+$
   (d) **Für jedes** *Objekt* aus *Menge*: <*Anweisung*>$^+$
2. Bedingte Anweisungen
   (a) **Wenn** <*Bedingung*> **Dann** <*Anweisung*>$^+$
   (b) **Wenn** <*Bedingung*> **Dann** <*Anweisung*>$^+$ **Sonst** <*Anweisung*>$^+$

Bedingungen sind Anweisungen, die einen booleschen Wert erzeugen. Durch Negation, Konjunktion oder Disjunktion von einfachen Bedingungen können komplexe Bedingungen gebildet werden. Innerhalb komplexer Anweisungen werden einfache

Anweisungen und Bedingungen durch spitze Klammern eingeschlossen. Wir verzichten darauf, innerhalb komplexer Anweisungen Anfang und Ende eines Anweisungsblocks explizit zu markieren (BEGIN ... END), und begnügen uns damit, durch Einrückung mögliche Ambiguitäten aufzulösen. Einen wichtigen Anweisungstyp bilden Wertzuweisungen. Als Zuweisungsoperator verwenden wir das „$\Leftarrow$"-Zeichen; d.h. sie haben die Form:

$$\text{VARIABLE} \Leftarrow \textit{Wertbezeichner}$$

Enthält eine Anweisung einen Prozeduraufruf, wird der Name der Prozedur kursiv gesetzt und unterstrichen:

$$\ldots \underline{\textit{Prozedur}}\,(arg_1,\ \ldots,\ arg_n)\ \ldots$$

Die beschriebenen Prozeduren und Algorithmen operieren häufig auf *strukturierten Objekten*, die aus einer Sequenz von anderen Objekten (Komponenten genannt) bestehen. Die Beschreibung von Operationen auf Objekten dieses Typs läßt sich relativ einfach gestalten, wenn man geeignete *Selektoren* und *Konstruktoren* definiert, die den gezielten Bezug auf einzelne Komponenten der strukturierten Objekte ermöglichen und spezifizieren, wie aus bestehenden Objekten neue gebildet werden können. Wir verwenden die folgenden Selektoren und Konstruktoren:

1. *Selektoren*

   Wenn ein Parameter bzw. eine Variable VAR an ein strukturiertes Objekt OBJ gebunden ist, das aus einer Sequenz von anderen Objekten $<obj_1, \ldots, obj_n>$ ($n \geq 0$) besteht, dann bezeichnet:

   (a) VAR_i obj_i

   (b) *First*(VAR) obj_1

   (c) *Rest*(VAR) $<obj_2, \ldots, obj_n>$.

2. *Konstruktoren*

   Wenn VAR_1 und VAR_2 an zwei strukturierte Objekte $\text{OBJ}_1 = <obj_1, \ldots, obj_n>$, $\text{OBJ}_2 = <obj_1', \ldots, obj_m'>$ gebunden sind, dann bezeichnet:

   $$\text{VAR}_1 + \text{VAR}_2$$

   das Objekt $\text{OBJ}_{12} = <obj_1, \ldots, obj_n, obj_1', \ldots, obj_m'>$, das sich durch die Verknüpfung der Objekte OBJ_1 und OBJ_2 ergibt.

Aufgaben

2.1 Formulieren Sie eine Anzahl von Lexikoneinträgen (15-20) und repräsentieren Sie sie als *Trie*.

2.2 Gegeben sei ein als *Trie* repräsentiertes Lexikon. Definieren Sie eine Prozedur, die es ermöglicht

   (a) die einer Wortform im Lexikon zugeordneten Informationen und extrahieren

   (b) neue Einträge im Lexikon zu erzeugen.

2.3 Entwickeln Sie einen Lexikongenerator, der eine Folge von Lexikoneinträgen in einen *Trie* kompiliert.

2.4 Bei Sprachen, die über ein ausgeprägteres Flexionssystem verfügen als das Englische, liegt es nahe, nicht alle Formen eines Wortes innerhalb eines Lexikons aufzulisten (*Vollformlexikon*), sondern Wortstämme und Flexionsendungen separat zu verwalten (Trennung von *Grundform-* und *Endungslexikon*). Verwenden Sie dieses Verfahren, um ein kleines Lexikon für deutsche Verbformen zu erstellen.

2.5 Wo liegen die deskriptiven Grenzen des in der vorangegangenen Frage skizzierten Verfahrens?

Teil II

Elementare Analysealgorithmen

Die Algorithmen, die wir in den folgenden drei Kapiteln vorstellen werden, bezeichnen wir als „elementare Analysealgorithmen“, da es sich bei ihnen um die einfachsten Algorithmen handelt, die sich zur Analyse beliebiger kontextfreier Sprachen eignen[1]. Sie unterscheiden sich durch die bei der Verarbeitung eines Satzes realisierte Analyserichtung: Wir beginnen mit top-down arbeitenden Algorithmen (Kapitel 3), behandeln anschließend Bottom-up-Algorithmen (Kapitel 4) und beenden diesen Abschnitt mit *Left-corner*-Algorithmen (Kapitel 5), die eine Mischstrategie realisieren.

Speicherung von Teilergebnissen. Charakteristisch für alle in diesen drei Kapiteln behandelten Algorithmen ist, daß die bei der Analyse eines Satzes w gewonnenen Teilergebnisse nicht gespeichert werden. So ist es möglich, daß ein Satzabschnitt mehrfach auf dieselbe Art und Weise analysiert wird. Diese Situation kann z.B. dann eintreten, wenn die Syntax Regeln der Form $X \rightarrow \alpha_1$, $X \rightarrow \alpha_2$, ..., $X \rightarrow \alpha_n$ $(n > 1)$ enthält, deren rechte Seiten einen identischen Präfix aufweisen:

Beispiel (II-1)
Wenn G eine Syntax ist, die u.a. die folgenden Regeln enthält:

S → NP S
S → NP Aux VP
S → NP VP PP
S → NP VP

dann kann die Verarbeitung eines Satzes dazu führen, daß die Subjekt-NP mehrfach auf dieselbe Weise analysiert wird, bevor es möglich ist, den Satz vollständig zu verarbeiten.

Kontrollstruktur. Da es verschiedene Möglichkeiten (Regeln) geben kann, mit denen die Analyse von w fortgesetzt werden kann, müssen diese Algorithmen eine Kontrollstruktur verwenden, die sicherstellt, daß bei Bedarf alle diese Alternativen untersucht werden können. Wir formulieren jeweils einen depth-first arbeitenden Algorithmus mit Backtracking und einen breadth-first arbeitenden Algorithmus (ohne Backtracking).

Komplexität. Alle der vorgestellten Algorithmen besitzen einen polynomiellen Zeit- und Raumbedarf.

[1]Daraus folgt nicht, daß sie beliebige kontextfreie Syntaxen verarbeiten können. Im Gegenteil: Es sind für die Algorithmen aus allen drei Klassen weitreichende Restriktionen bzgl. der in der Syntax zulässigen Regeltypen zu beachten (z.B. keine links-/rechtsrekursive Regeln, keine Tilgungsregeln etc.).

Kapitel 3

Top-down-Parsing

Wir entwickeln in diesem Kapitel zunächst ein Schema zur Top-down-Analyse von Sätzen, das wir anschließend schrittweise zu einem Erkennungs- und einem Parsingalgorithmus erweitern. Dabei wird gezeigt, wie bei einer Top-down-Analyse entweder die Depth-first-Suche mit Backtracking oder die Breadth-first-Suche verwendet werden kann.

3.1 Grundzüge

Charakteristisch für das Top-down-Parsing ist der von oben nach unten gerichtete Aufbau syntaktischer Strukturbeschreibungen: Ausgehend von dem Startsymbol der Syntax wird nach einer Ableitung gesucht, die es erlaubt, Schritt für Schritt eine oder gegebenenfalls mehrere Strukturbeschreibungen für die Eingabekette zu generieren.

ALGORITHMUS RECOGNIZE$_{TD}$
top-down / links-rechts
deterministisch

DATEN: Ein Lexikon L und eine kontextfreie Syntax G = $<V_N, V_T, S, R>$.
EINGABE: Ein Satz $w = w_1 \ldots w_n$, mit $n \geq 0$.
AUSGABE: *True*/*False*.
ARBEITSSTRUKTUREN:

- WORT - Eine Zahl), die auf das nächste zu verarbeitende Wort des Satzes weist.
  Anfangswert : 1.
- ABLEITUNG - Eine Folge von syntaktischen und lexikalischen Kategorien.
  Anfangswert: [S].

METHODE:

Wiederhole:
- Wenn <ABLEITUNG = [ ]> $\wedge$ <WORT = n+1>
- Dann <RETURN(*True*)>
- Sonst
  - Wenn <ABLEITUNG = [ ]> $\vee$ <*First*(ABLEITUNG) $\in V_T$ >
  - Dann <RETURN(*False*)>
  - Sonst <ABLEITUNG $\Leftarrow$ *Reduziere*(*Expandiere*(ABLEITUNG))>

Wenn wir vernachlässigen, daß dieser Suchprozeß nicht-deterministisch ist, d.h., daß es häufig verschiedene Möglichkeiten gibt, eine begonnene Ableitung fortzusetzen, dann lassen sich die grundlegenden Operationen für Top-down-Parsing durch die Prozedur RECOGNIZE$_{TD}$ beschreiben. Diese Prozedur verarbeitet einen Satz w von links nach rechts. Die beiden Variablen ABLEITUNG und WORT dienen dazu, den aktuellen Stand der Ableitung und einen Verweis auf das nächste zu verarbeitende Wort zu speichern. Sie terminiert, wenn eine der folgenden Situationen eintritt:

1. der Satz ist komplett abgearbeitet und ABLEITUNG enthält keine Symbole[1];

2. ABLEITUNG enthält keine Symbole, aber der Satz ist noch nicht vollständig verarbeitet (WORT $<$ n+1);

3. das erste Symbol in ABLEITUNG ist eine *lexikalische* Kategorie, und das nächste zu verarbeitende Wort des Satzes ist kein Element dieser Kategorie.

[1] Wenn die Syntax Tilgungsregeln enthält, dann muß eine Ableitung nicht terminieren, sobald WORT = n + 1 und ABLEITUNG $\neq$ [ ].

Bis der Algorithmus terminiert, wird in jedem Verarbeitungszyklus zunächst das erste Symbol in ABLEITUNG durch die rechte Seite einer geeigneten Regel ersetzt[2] (EXPANDIERE). Anschließend wird versucht, alle lexikalischen Kategorien, die in ABLEITUNG links vor der ersten syntaktischen Kategorie stehen, zu tilgen (REDUZIERE). Wenn nicht alle führenden lexikalischen Kategorien entfernt werden können, terminiert der Algorithmus im nächsten Durchgang mit „*False*“ (Fall 3).

Die Wahl einer Regel zur Fortsetzung der begonnenen Ableitung in EXPANDIERE ist von entscheidender Bedeutung: Die Kontrollstruktur von $\mathrm{RECOGNIZE}_{\mathrm{TD}}$ erlaubt es nicht, verschiedene Analysemöglichkeiten zu verfolgen. Ein Satz nur dann erkannt, wenn EXPANDIERE immer die 'richtige' Regel wählt. Da in jedem Schritt immer die syntaktische Kategorie in ABLEITUNG ersetzt wird, die am weitesten links steht, generiert der Algorithmus sogenannte *Linksableitungen* und der Inhalt von ABLEITUNG wird entsprechend als *Linkssatzform* bezeichnet. Die Prozeduren EXPANDIERE und REDUZIERE lassen sich wie folgt beschreiben:

PROZEDUR EXPANDIERE

EINGABE: Eine Linkssatzform L_{SF}.
AUSGABE: Eine neue Linkssatzform.

METHODE:
Wähle eine Regel $\mathrm{r}_{\mathrm{i}} \in \mathrm{R}$, mit $\mathit{Left}(\mathrm{r}_{\mathrm{i}}) = \mathit{First}(\mathrm{L}_{\mathrm{SF}})$
$<\mathrm{RETURN}(\mathit{Right}(\mathrm{r}_{\mathrm{i}}) + \mathit{Rest}(\mathrm{L}_{\mathrm{SF}}))>$

[2] Wir setzen voraus, daß die Syntax keine überfüssigen nicht-terminale Symbole enthält (mit anderen Worten: $X \stackrel{*}{\Rightarrow} \alpha$, mit $X \in V_N$ und $\alpha \in V_T^*$, für alle $X \in V_N$). So ist sichergestellt, daß EXPANDIERE immer gelingt.

PROZEDUR REDUZIERE

EINGABE: Eine Linkssatzform L_{SF}.
AUSGABE: Eine neue Linkssatzform.
SEITENEFFEKT: Die Änderung der Variablen WORT.

METHODE:
Wiederhole:
Wenn $<\mathit{First}(L_{SF}) \in V_T> \ \wedge \ <w_{WORT} \in \mathit{First}(L_{SF})>$
Dann $<WORT \Leftarrow WORT + 1>$
$<\underline{\mathit{Reduziere}}(\mathit{Rest}(L_{SF}))>$
Sonst $<RETURN(L_{SF})>$

Beispiel (3-1)

Satz : Der alte Mann starb.
Syntax : $\{1{:}S \rightarrow NP\ VP,\ 2{:}NP \rightarrow det\ N1,\ 3{:}N1 \rightarrow AP\ n,\ 4{:}AP \rightarrow adj,\ 5{:}VP \rightarrow v\}$

	Satz[3]	ABLEITUNG	Aktion	Regel
(1)	[Der alte Mann starb]	[S]		
(2)	[Der alte Mann starb]	[NP VP]	EXPANDIERE	1
(3)	[Der alte Mann starb]	[det N1 VP]	EXPANDIERE	2
(4)	[alte Mann starb]	[N1 VP]	REDUZIERE	
(5)	[alte Mann starb]	[AP n VP]	EXPANDIERE	3
(6)	[alte Mann starb]	[adj n VP]	EXPANDIERE	4
(7)	[Mann starb]	[n VP]	REDUZIERE	
(8)	[starb]	[VP]	REDUZIERE	
(9)	[starb]	[v]	EXPANDIERE	5
(10)	[ ]	[ ]	REDUZIERE	

Es gibt verschiedene Möglichkeiten, auf Grundlage der Prozedur $RECOGNIZE_{TD}$ einen Parsingalgorithmus zu entwickeln: Die konzeptuell vielleicht naheliegendste Möglichkeit besteht darin, eine weitere Variable einzuführen, die dann für den Aufbau dieser Strukturbeschreibung verwendet wird[4]. Allerdings führt diese Lösung zu einer unübersehbaren Redundanz: Wenn man das oben gegebene Beispiel betrachtet, wird deutlich, daß diese Struktur zumindest *implizit* bereits in der Variablen ABLEITUNG durch EXPANDIERE aufgebaut und durch REDUZIERE allerdings auch wieder abgebaut wird.

[3] Aus Gründen der Anschaulichkeit geben wir hier nicht den Wert von WORT an, sondern den noch zu verarbeitenden Teil des Satzes.

[4] In diesem Fall hat man die Möglichkeit, entweder die Struktur direkt aufzubauen oder die Indizes der Regeln, die verwendet werden, zu akkumulieren und anschließend den Linksparse des Satzes auszugeben.

Eine andere Möglichkeit besteht darin, statt in einer neuen Variablen die Strukturbeschreibung aufzubauen, die Prozedur REDUZIERE so zu reformulieren, daß sie nicht mehr Symbole löscht, sondern einen *Zeiger* auf der in ABLEITUNG gebildeten Strukturbeschreibung verschiebt:

Beispiel (3-2)

$[\overset{1}{S}]$

$[S\ [\overset{2}{NP}\ \ VP]]$

$[S\ [[NP\ [\overset{3}{det}\ \ N1]]\ VP]]$

$[S\ [[NP\ [[det\ der]\ \ \overset{5}{N1}]]\ VP]]$

$[S\ [[NP\ [[det\ der]\ [N1\ [\overset{6}{AP}\ \ n]]]]\ VP]]$

$[S\ [[NP\ [[det\ der]\ [N1\ [[AP\ [\overset{7}{adj}]]\ n]]]]\ VP]]$

$[S\ [[NP\ [[det\ der]\ [N1\ [[AP\ [adj\ alte]]\ \ \overset{9}{n}]]]]\ VP]]$

$[S\ [[NP\ [[det\ der]\ [N1\ [[AP\ [adj\ alte]]\ [n\ mann]]]]]\ \ \overset{11}{VP}]]$

$[S\ [[NP\ [[det\ der]\ [N1\ [[AP\ [adj\ alte]]\ [n\ mann]]]]]\ [VP\ [\overset{12}{v}]]]]$

$[S\ [[NP\ [[det\ der]\ [N1\ [[AP\ [adj\ alte]]\ [n\ mann]]]]]\ [VP\ [v\ starb]]]]\ ^{14}$

Die beiden von uns betrachteten Alternativen zur Formulierung des Parsingalgorithmus illustrieren gut ein bekanntes Problem. Zeitverhalten und Speicherbedarf bilden die wichtigsten Kriterien zur Beurteilung von Problemlösungen. Häufig korrelieren diese Faktoren in der Weise miteinander, daß eine Lösung, die weniger Speicher beansprucht, mehr Zeit benötigt als eine andere, speicherintensivere Lösung und vice versa.

In unserem Fall hätte die Einführung einer weiteren Variablen zum Aufbau der Satzstruktur mehr Speicherbedarf erfordert als die von uns gewählte Methode. Wird dagegen die Strukturbeschreibung in der Variablen ABLEITUNG aufgebaut, sind komplexere Operationen erforderlich, die sich je nach Realisierung als relativ zeitintensiv erweisen können[5].

Die Prozedur $\mathrm{PARSE_{TD}}$ (s.u.) unterscheidet sich von der Prozedur $\mathrm{RECOGNIZE_{TD}}$ vor allem in folgenden Punkten:

1. Neben WORT und STRUKTUR (in $\mathrm{RECOGNIZE_{TD}}$: ABLEITUNG) gibt es eine weitere Variable POSITION, deren Wert eine Zahl j ist, die auf das j-te Symbol der in STRUKTUR gebildeten Strukturbeschreibung weist (Anfangswert: 1).

2. Wenn die Prozedur erfolgreich terminiert, wird nicht TRUE, sondern der Wert

[5]Ein anderer wichtiger Parameter zur Beurteilung konkurrierender Lösungen bildet ihre Transparenz: Auch unter diesem Aspekt dürfte die erste Lösung vorzuziehen sein, da sie als eine relativ einfache konservartive Erweiterung aufgefaßt werden kann.

von STRUKTUR (die Strukturbeschreibung des Satzes) als Wert zurückzugeben.

3. Die Prozeduren EXPANDIERE und REDUZIERE wurden ersetzt durch EXPANDIERE' und REDUZIERE'.

PROZEDUR EXPANDIERE'

Schema

EINGABE: Eine Strukturbeschreibung SB.
AUSGABE: Eine neue Strukturbeschreibung.
SEITENEFFEKT: Die Änderung der Variablen POSITION.
ARBEITSSTRUKTUREN:
NEUE-STRUKTUR - Die aktualisierte Strukturbeschreibung des Satzes w.
METHODE:
Wähle eine Regel $r_i \in R$, mit $\mathit{Left}(r_i) = SB_{POSITION}$
NEUE-STRUKTUR $\Leftarrow$ Ersetze $SB_{POSITION}$ durch: $[SB_{POSITION}\ [\mathit{Right}(r_i)]]$
POSITION $\Leftarrow$ POSITION + 1
RETURN(NEUE-STRUKTUR)

Die Prozedur EXPANDIERE' ersetzt ein Symbol aus ABLEITUNG durch eine Struktur, die aus dem Symbol und der rechten Seite einer Regel besteht, durch die das Symbol ersetzt werden kann.

PROZEDUR REDUZIERE'

EINGABE: Eine Strukturbeschreibung SB.
AUSGABE: Eine neue Strukturbeschreibung.
SEITENEFFEKT: Die Änderung der Variablen STRUKTUR und POSITION.
METHODE:
Wiederhole:
Wenn <$SB_{POSITION} \in V_T$> $\wedge$ <$w_{WORT} \in SB_{POSITION}$>
Dann <Ersetze $SB_{POSITION}$ durch $[SB_{POSITION}\ w_{WORT}]$>
<POSITION $\Leftarrow$ POSITION + 2>
<WORT $\Leftarrow$ WORT + 1>
Sonst <RETURN(SB)>

Die Verschiebung des Zeigers POSITION um zwei Symbole ist erforderlich, da, wenn eine Reduktion möglich ist, die führende lexikalische Kategorie durch eine aus zwei Symbolen bestehende Struktur ($[SB_{POSITION}\ w_{WORT}]$) ersetzt wird.

ALGORITHMUS PARSE$_{TD}$
top-down / links-rechts
deterministisch

DATEN: Ein Lexikon L und eine kontextfreie Syntax G = <V_N, V_T, S, R>.
EINGABE: Ein Satz w = $w_1 \ldots w_n$, mit $n \geq 0$.
AUSGABE: Eine Strukturbeschreibung für w / *False*.
ARBEITSSTRUKTUREN:

- WORT - Eine Zahl, die auf das nächste zu verarbeitende Wort des Satzes weist.
  Anfangswert : 1.
- STRUKTUR - Eine Strukturbeschreibung für w.
  Anfangswert: S.
- POSITION - Ein Zeiger auf ein Symbol in STRUKTUR.
  Anfangswert: 1.

METHODE:
Wiederhole:
Wenn <POSITION = |STRUKTUR| + 1>[6] $\wedge$ <WORT = n+1>
Dann <RETURN(STRUKTUR)>
Sonst
Wenn <POSITION = |STRUKTUR| + 1> $\vee$
<STRUKTUR$_{POSITION}$ $\in$ V_T >
Dann <RETURN(*False*)>.
Sonst <STRUKTUR $\Leftarrow$ *Reduziere'*(*Expandiere'*(STRUKTUR))>

3.2 Kontrollstrukturen

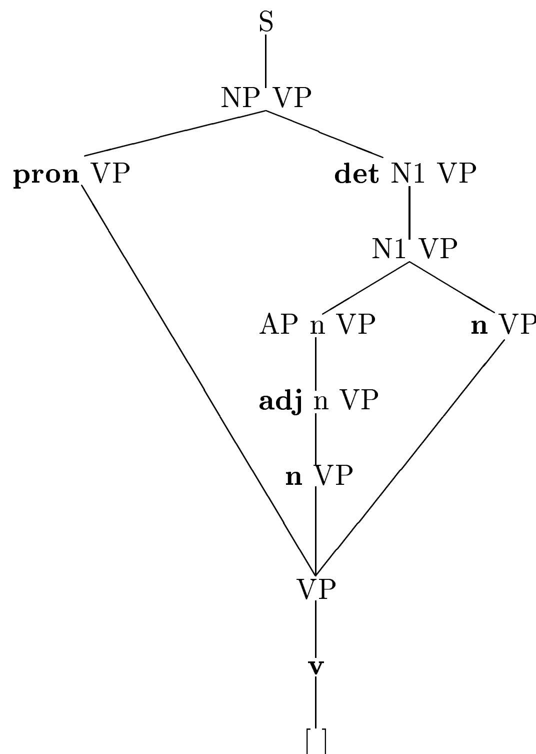
Bislang haben wir vernachlässigt, daß das Parsen eines Satzes in der Regel ein nichtdeterministischer Prozeß ist, bei dem es Situationen gibt, in denen mehrere alternative Regeln angewendet werden können. Jede Regel repräsentiert eine Möglichkeit, die begonnene Ableitung fortzusetzen. Die in einer Syntax möglichen Ableitungen lassen sich durch einen Baum bzw. Graphen repräsentieren.

[6]Die Länge von STRUKTUR wird in diesem Zusammenhang bestimmt durch die Zahl der terminalen bzw. nicht-terminalen Symbole, die STRUKTUR enthält.

Beispiel (3-3)
Für die Syntax

S → NP VP	VP → v
NP → pron	NP → det N1
N1 → AP n	N1 → n
AP → adj	

hat der Graph die Form[7]:



Analysealgorithmen verwenden Kontrollstrukturen, die es ihnen ermöglichen, die verschiedenen Ableitungsmöglichkeiten (nacheinander/ gleichzeitig) zu verfolgen, bis eine/alle Ableitung(en) für den Satz gefunden wurde(n) bzw. die Suche erfolglos terminiert (vgl. Kapitel 2.2.3). Wir werden uns im folgenden mit zwei solchen Kontrollstrukturen, der sequentiellen Depth-first-Suche mit Backtracking und der quasi-parallelen Breadth-first-Suche beschäftigen.

[7]Wir beschränken uns auf Linksableitungen.

3.2.1 Depth-first-Parsing mit Backtracking

Wenn es in einem Ableitungsschritt mehrere Regeln gibt, die zur Expansion des nächsten nicht-terminalen Symbols verwendet werden können, dann repräsentiert jede dieser Regeln eine andere mögliche Ableitung für den Satz. Allerdings muß sich keine dieser *möglichen Ableitungen* erfolgreich zu Ende führen lassen.

Der Algorithmus $\mathrm{RECOGNIZE}_{\mathrm{TD/DF}}$ basiert auf der Idee, immer nur eine Ableitung des Satzes zu verfolgen und andere mögliche Ableitungen erst zu untersuchen, **nachdem** die aktuelle Ableitung terminiert ist. Informationen über andere mögliche Ableitungen werden in einem Stack gespeichert: Wenn in einem Ableitungsschritt mehrere Regeln angewendet werden können, wird ein *Eintrag* erzeugt und auf den Stack geschoben, in dem die aktuelle Ableitung und alle Regeln außer der zunächst verwendeten vermerkt werden.

ALGORITHMUS $\mathbf{RECOGNIZE}_{\mathrm{TD/DF}}$
top-down / links-rechts
depth-first-Suche mit Backtracking

DATEN: Ein Lexikon L und eine kontextfreie Syntax $G = \langle V_N, V_T, S, R \rangle$.
EINGABE: Ein Satz $w = w_1 \ldots w_n$ $(n \geq 0)$.
AUSGABE: *True*/*False*.
ARBEITSSTRUKTUREN:

- WORT - Eine Zahl, die auf das nächste zu verarbeitende Wort des Satzes weist.
  Anfangswert: 1.
- ABLEITUNG - Eine Folge von syntaktischen und lexikalischen Kategorien.
  Anfangswert: [$Startsymbol_G$].
- STACK - Eine Folge von *Einträgen*.
  Anfangswert: [ ].

METHODE:
Wiederhole:

Wenn <WORT = n+1> $\wedge$ <ABLEITUNG = [ ]>
Dann <RETURN(*True*)>

Wenn <ABLEITUNG = [ ]> $\vee$ <*First*(ABLEITUNG) $\in V_T$>
Dann
  Wenn <STACK = [ ]>
  Dann <RETURN(*False*)>
  Sonst <STACK $\Leftarrow$ $\underline{\mathit{Backtrack}_{\mathrm{TD}}}$(STACK)>
Sonst <ABLEITUNG $\Leftarrow$ $\underline{\mathit{Reduziere}}$($\underline{\mathit{Expandiere}_{\mathrm{DF}}}$(ABLEITUNG, WORT))>

Ein *Eintrag* hat die Form: <*wort ableitung expansionen*>[8], wobei *wort* ein Zeiger auf das nächste zu verarbeitende Wort des Satzes ist und *expansion*en die rechten Seiten der Regeln aus G enthält, durch die die aktuelle Ableitung, deren Stand durch die Linkssatzform *ableitung* repräsentiert wird, fortgesetzt werden kann.

Solange der Stack nicht leer ist, werden, jedesmal wenn die aktuelle Ableitung nicht fortgesetzt werden kann, dem obersten Eintrag des Stacks die Informationen entnommen, die erforderlich sind, um eine andere Ableitung für den Satz verfolgen zu können, und der Stack wird entsprechend aktualisiert. Beim Backtracking wird der letzte Ableitungsschritt zurückgenommen und die Ableitung mit einer anderen als der zuvor gewählten Regel fortgesetzt. Dazu werden zunächst die Variablen WORT und ABLEITUNG mit den im obersten Eintrag von STACK enthaltenen Werten aktualisiert und anschließend das erste Element aus EXPANSIONEN entfernt. Wenn EXPANSIONEN keine weiteren Elemente mehr enthält, wird dieser Eintrag aus STACK entfernt.

Die Änderung der Prozedur EXPANDIERE ist erforderlich, um eine korrekte Stackverwaltung sicherzustellen.

PROZEDUR EXPANDIERE$_{DF}$

EINGABE: Eine Linkssatzform L_{SF} und eine Zahl j.
AUSGABE: Eine neue Linkssatzform.
SEITENEFFEKT: Die Änderung der Variablen STACK.
ARBEITSSTRUKTUREN:
EXPANSIONEN - Die Folge aller rechten Regelseiten der Regeln, durch die das erste Symbol von ABLEITUNG expandiert werden kann.
METHODE:

$\text{EXPANSIONEN} \Leftarrow <\alpha \mid X \rightarrow \alpha \in R \wedge X = \mathit{First}(L_{SF})>$

Wenn $<\mathit{Rest}(\text{EXPANSIONEN}) \neq \emptyset>$
Dann $<\text{STACK} \Leftarrow \mathit{Push}(<j\ L_{SF}\ \mathit{Rest}(\text{EXPANSIONEN})>, \text{STACK})>$

$<\text{RETURN}(\mathit{First}(\text{EXPANSIONEN}) + \mathit{Rest}(L_{SF}))>$

Wenn man von dem Aufwand absieht, den die Stackverwaltung erfordert, dann besteht einer der wichtigsten Vorzüge dieses Algorithmus darin, daß sich durch minimale Änderungen einzelner Operationen verschiedene Suchstrategien realisieren lassen (s.u.).

[8] Wir verwenden die Notation „*wort*(*eintrag*)“, „*ableitung*(*eintrag*)“ und „*expansionen*(*eintrag*)“, um auf die erste/zweite/dritte Komponente eines Eintrags Bezug zu nehmen.

PROZEDUR BACKTRACK$_{\mathrm{TD}}$

EINGABE: Ein Stack T.
AUSGABE: Der geänderte Stack.
SEITENEFFEKT: Die Änderung der Variablen ABLEITUNG und WORT.

METHODE:

WORT $\Leftarrow$ *wort*(*First*(STACK))
ABLEITUNG $\Leftarrow$ Ersetze *First*(*ableitung*(*First*(STACK)))
durch *First*(*expansionen*(*First*(STACK))).

Entferne *First*(*expansionen*(*First*(STACK))) aus *expansionen*(*First*(STACK)).

Wenn <*expansionen*(*First*(STACK)) = ∅>
Dann <RETURN(*Pop*(STACK))>
Sonst <RETURN(STACK)>

Wenn man von dem Aufwand absieht, den die Stackverwaltung erfordert, dann besteht einer der wichtigsten Vorzüge dieses Algorithmus darin, daß sich durch minimale Änderungen einzelner Operationen verschiedene Suchstrategien realisieren lassen (s.u.).

Das folgende Beispiel illustriert die Arbeitsweise des Algorithmus:

Beispiel (3-4)
Syntax $=\{\, S \rightarrow NP\ VP,\ NP \rightarrow det\ n,\ NP \rightarrow det\ adj\ n,\ VP \rightarrow V1,$
$VP \rightarrow V1\ PP,\ V1 \rightarrow v,\ V1 \rightarrow v\ NP,\ PP \rightarrow p\ NP\}$
Satz = Die Katze jagt die Maus in der Küche

WORT	ABLEITUNG	STACK	AKTION[9]
1	[S]	[ ]	EX : S → NP VP
1	[NP VP]	[ ]	EX : NP → det n
1	[det n VP]	[<1 [NP VP] [[det adj n]]>]	RE
3	[VP]	[<1 [NP VP] [[det adj n]]>]	EX : VP → V1
3	[V1]	[<3 [VP] [[V1 PP]]>	EX : V1 → v
		<1 [NP VP] [[det adj n]]>]	
3	[v]	[<3 [V1] [[v NP]]>	RE
		<3 [VP] [[V1 PP]]>	
		<1 [NP VP] [[det adj n]]>]	
4	[ ]	[<3 [V1] [[v NP]]>	BT : V1 → v NP
		<3 [VP] [[V1 PP]]>	
		<1 [NP VP] [[det adj n]]>]	
3	[v NP]	[<3 [VP] [[V1 PP]]>	RE
		<1 [NP VP] [[det adj n]]>]	

4	[NP]	[<3 [VP] [[V1 PP]]> <1 [NP VP] [[det adj n]]>]	EX : NP → det n
4	[det n]	[<4 [NP] [[det adj n]]> <3 [VP] [[V1 PP]]> <1 [NP VP] [[det adj n]]>]	RE
6	[ ]	[<4 [NP] [[det adj n]]> <3 [VP] [[V1 PP]]> <1 [NP VP] [[det adj n]]>]	BT : NP → det adj n
4	[det adj n]	[<3 [VP] [[V1 PP]]> <1 [NP VP] [[det adj n]]>]	RE
4	[adj n]	[<3 [VP] [[V1 PP]]> <1 [NP VP] [[det adj n]]>]	BT : VP → V1 PP
3	[V1 PP]	[<1 [NP VP] [[det adj n]]>]	EX : V1 → v
3	[v PP]	[<3 [V1 PP] [[v NP]]> <1 [NP VP] [[det adj n]]>]	RE
4	[PP]	[<3 [V1 PP] [[v NP]]> <1 [NP VP] [[det adj n]]>]	EX : PP → p NP
4	[p NP]	[<3 [V1 PP] [[v NP]]> <1 [NP VP] [[det adj n]]>]	BT : V1 → v NP
3	[v NP PP]	[<1 [NP VP] [[det adj n]]>]	RE
4	[NP PP]	[<1 [NP VP] [[det adj n]]>]	EX : NP → det n
4	[det n PP]	[<4 [NP PP] [[det adj n]]> <1 [NP VP] [[det adj n]]>]	RE
6	[PP]	[<4 [NP PP] [[det adj n]]> <1 [NP VP] [[det adj n]]>]	EX : PP → p NP
6	[p NP]	[<4 [NP PP] [[det adj n]]> <1 [NP VP] [[det adj n]]>]	RE
7	[NP]	[<4 [NP PP] [[det adj n]]> <1 [NP VP] [[det adj n]]>]	EX : NP → det n
7	[det n]	[<7 [NP] [[det adj n]]> <4 [NP PP] [[det adj n]]> <1 [NP VP] [[det adj n]]>]	RE
9	[ ]	[<7 [NP] [[det adj n]]> <4 [NP PP] [[det adj n]]> <1 [NP VP] [[det adj n]]>]	

In seiner vorliegenden Form arbeitet der Algorithmus, wie man anhand des Beispiels (3-4) sieht, nach der Depth-first-Methode. Andere Suchstrategien lassen sich realisieren, indem man die Einträge nicht in einen Stack speichert, der nach dem *Last-in-first-out*-Prinzip arbeitet, sondern eine andere Datenstruktur wählt: Verwendet man z.B. eine Queue (*First-in-first-out*-Prinzip), werden die durch neue

[9] In dieser Spalte geben wir die Aktionen an, die zu den in der nächsten Zeile angegebenen Variablenwerten führen: EX - $\text{EXPANDIERE}_{\text{DF}}$, RE - REDUZIERE und BT - $\text{BACKTRACK}_{\text{TD}}$.

Einträge repräsentierten Ableitungsalternativen erst dann berücksichtigt, nachdem die zu diesem Zeitpunkt bereits gefundenen Alternativen untersucht wurden; d.h., es wird eine Breadth-first-Suche realisiert. Eine weitere Möglichkeit besteht darin, die Einträge in einer Liste zu verwalten und diese Liste in jedem Ableitungsschritt nach einem bestimmten Kriterium zu sortieren (z.B. unter Nutzung heuristischer Informationen) bzw. neue Einträge immer an der 'korrekten' Position in die Liste einzufügen.

Um aus dem Erkennungsalgorithmus einen Parsingalgorithmus abzuleiten, kann eine weitere Variable STRUKTUR eingeführt werden, deren Wert die Liste der Regeln ist, die bei der aktuellen Ableitung verwendet wurden. Bei erfolgreicher Terminierung des Ableitungsprozesses wird aus dieser Regelfolge eine Strukturbeschreibung für den Satz generiert. Mit einer zweiten einfachen Änderung kann außerdem sichergestellt werden, daß nicht nur eine, sondern alle Strukturbeschreibungen eines Satzes generiert werden: Nach erfolgreicher Terminierung **einer** Ableitung wird zunächst die gefundene Strukturbeschreibung ausgegeben und anschließend - sofern der Stack nicht leer ist - Backtracking erzwungen (s. Aufgaben am Ende dieses Kapitels).

3.2.2 Breadth-first-Parsing ohne Backtracking

Die letzte Variante eines Top-down-Algorithmus, die wir betrachten werden, arbeitet ohne Backtracking: Jedesmal wenn es verschiedene Möglichkeiten gibt, eine begonnene Ableitung fortzusetzen, wird für jede dieser Möglichkeiten ein 'Prozeß' gestartet, der sie weiter verfolgt (breadth-first). Der Algorithmus terminiert erst, nachdem alle Analysemöglichkeiten für einen Satz untersucht wurden. Da die 'Prozesse' unabhängig voneinander ausgeführt werden können, bietet es sich an, auf Grundlage dieses Algorithmus einfache parallele Parser zu entwickeln und, sofern geeignete Soft- und Hardware zur Verfügung steht, auch zu implementieren.

Die rekursive Prozedur $\mathrm{RECOGNIZE}_{\mathrm{TD/BF}}$ wird mit zwei Argumenten aufgerufen: dem zu analysierenden Satz $\mathrm{w} = \mathrm{w}_1 \ldots \mathrm{w}_\mathrm{n}$ $(\mathrm{n} \geq 0)$ und S, dem Startsymbol der Syntax:

ALGORITHMUS $\mathbf{RECOGNIZE}_{\mathrm{TD/BF}}$
top-down / links-rechts
breadth-first

DATEN: Ein Lexikon L und eine kontextfreie Syntax $\mathrm{G} = \langle \mathrm{V_N}, \mathrm{V_T}, \mathrm{S}, \mathrm{R} \rangle$.
EINGABE: Eine Folge von Wörtern SATZ (Anfangswert: $[\mathrm{w}_1, \ldots \mathrm{w}_n]$) und eine Folge von terminalen und nicht-terminalen Symbolen ABLEITUNG (Anfangswert [S]).
AUSGABE: *True*/*False*.

ARBEITSSTRUKTUREN:

EXPANSIONEN - Die Folge aller rechten Regelseiten durch die das erste Symbol von ABLEITUNG ersetzt werden kann.

METHODE:

Wenn <SATZ = [ ]> $\wedge$ <ABLEITUNG = [ ]>
Dann <RETURN(*True*)>
Sonst
  Wenn <ABLEITUNG = [ ]> $\vee$ <*First*(ABLEITUNG) $\in V_T$>
  Dann <RETURN(*False*)>
  Sonst <EXPANSIONEN $\Leftarrow$ <x | $r_i \in R \wedge$
    Left(r_i) = *First*(ABLEITUNG) $\wedge$
    x = *Right*(r_i)>>
  Für alle x $\in$ EXPANSIONEN:
    <$\underline{RECOGNIZE_{TD/BF}}$
    ($\underline{Reduziere_{BF}}$(SATZ, (x+*Rest*(ABLEITUNG))))>

PROZEDUR REDUZIERE$_{BF}$

EINGABE: Eine Folge von Wörtern SATZ und eine Linkssatzform ABLEITUNG.

AUSGABE: Ein geordnetes Paar (SATZ' und ABLEITUNG'), das sich durch Reduktion aus SATZ und ABLEITUNG ergibt.

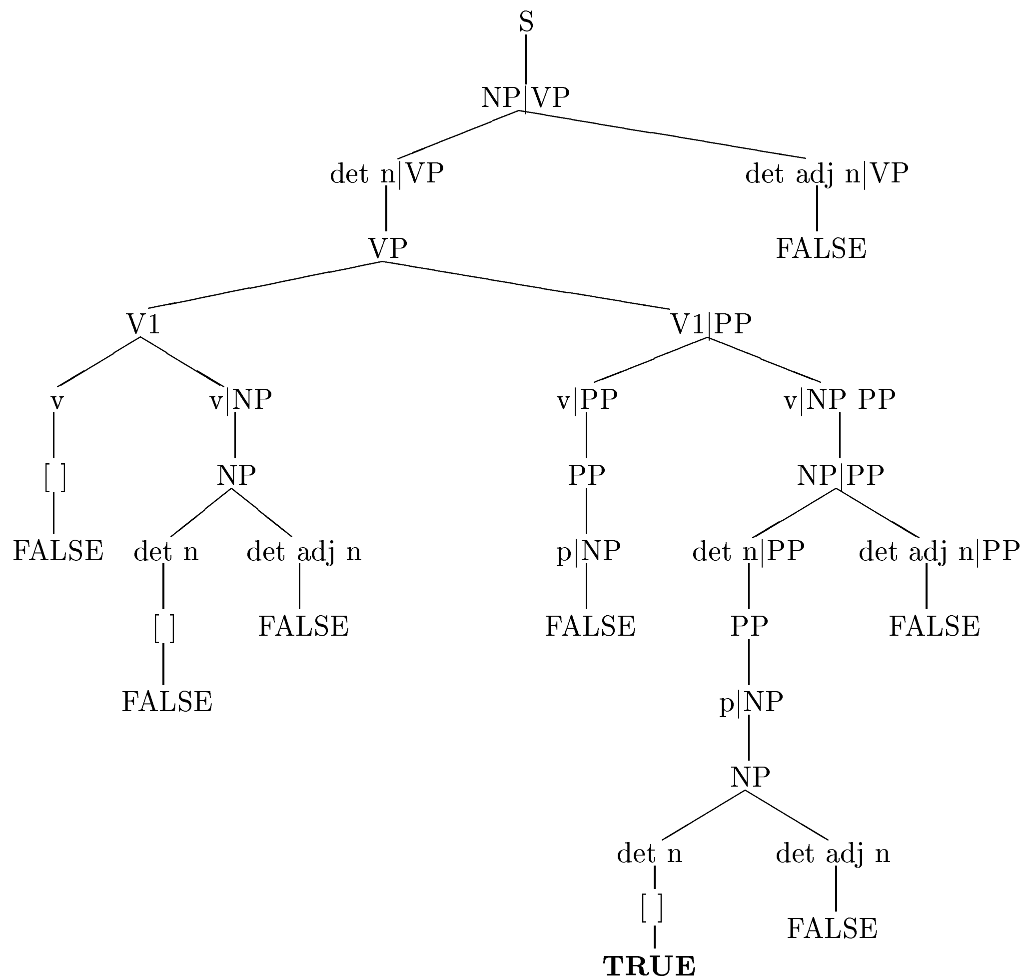
METHODE:

Wenn <*First*(ABLEITUNG) $\in V_T$> $\wedge$
  First(SATZ) $\in$ *First*(ABLEITUNG)>
Dann <$\underline{Reduziere_{BF}}$(*Rest*(SATZ), *Rest*(ABLEITUNG))>
Sonst <RETURN(SATZ, ABLEITUNG)>

Beispiel (3-5)

Der folgende Baum gibt einen Überblick über die Prozesse, die bei der Analyse des Satzes „Die Katze jagt die Maus in der Küche“ bei Verwendung der Syntax aus Beispiel (3-4) erzeugt werden: Jeder Pfad in diesem Baum, der vom Wurzelknoten zu einem der mit „TRUE“ etikettierten Blattknoten führt, repräsentiert eine mögliche Ableitung für den Satz.

Der Algorithmus ist ebenso wie der mit Backtracking arbeitende Algorithmus aus dem vorangegangenen Abschnitt relativ ineffizient, da Satzabschnitte von verschiedenen Prozessen mehrfach in gleicher Weise analysiert werden.



3.3 Grenzen des Top-down-Parsings

Top-down-Parser, die mit einem der in diesem Kapitel vorgestellten Algorithmen arbeiten, lassen sich schnell und ohne Probleme implementieren und sind insofern zur maschinellen Verarbeitung beliebiger kontextfreier *Sprachen* geeignet, als für jede kontextfreie Sprache eine Syntax angegeben werden kann, die für Top-down-Parsing geeignet ist. Allerdings können nicht beliebige kontextfreie *Syntaxen* mit diesen Algorithmen verarbeitet werden:

Abhängig von der Verarbeitungsrichtung des Satzes darf die Syntax nicht *linksrekursiv* (Verarbeitungsrichtung: links $\rightarrow$ rechts) bzw. *rechtsrekursiv* (Verarbeitungsrichtung: rechts $\rightarrow$ links) sein. Wird diese Restriktion nicht beachtet, ist nicht sichergestellt, daß der Parser bei der Analyse eines Satzes terminiert.

Beispiel (3-6)
Wenn wir die Syntax aus Beispiel (3-1) um die Regel $AP \rightarrow AP\ adj$ ergänzen und voraussetzen, daß diese Regel immer vor der Regel $AP \rightarrow adj$ angewendet wird, dann terminiert die Prozedur nicht, da mit jedem Schritt die Länge von ABLEITUNG kontinuierlich wächst:

	SATZ	ABLEITUNG
(1)	[Der alte Mann starb]	[S]
⋮	⋮	⋮
(5)	[alte Mann starb]	[AP n VP]
(6')	[alte Mann starb]	[AP adj n VP]
(6")	[alte Mann starb]	[AP adj adj n VP]
⋮	⋮	⋮

Enthält die Syntax Tilgungsregeln oder/und unäre Regeln[10], ist es häufig nicht so einfach wie in dem vorangegangenen Beispiel zu erkennen, daß die Syntax *links-/rechtsrekursiv* ist; denn in diesem Fall können es mehrere Regeln sein, deren Interaktion wie die Anwendung einer links-/rechtsrekursiven Regel wirkt:
Eine Syntax S', die man aus einer Syntax S erhält, indem man jede linksrekursive Regel der Form $X \rightarrow X\ \alpha$ in S z.B. durch Regeln der Form

(a) $X \rightarrow B\ \alpha$ bzw. (b) $X \rightarrow B\ X\ \alpha$ oder (c) $X \rightarrow B\ \alpha$
$B \rightarrow X$ $B \rightarrow e$ $B \rightarrow Y\ X$
$\mathrm{Y} \rightarrow e$

ersetzt, führt beim Parsen zu denselben Problemen wie die ursprüngliche Syntax. Man kann in diesem Zusammenhang verschiedene Lösungsansätze verfolgen:

1. Grundsätzlich ist es möglich, eine beliebige links-/rechtsrekursive kontextfreie Syntax in eine schwach äquivalente kontextfreie Syntax zu überführen, die nicht links-/rechtsrekursiv ist. Allerdings ist in vielen Fällen die Annahme links-/rechtsrekursiver Regeln und Konstruktionen linguistisch motiviert, so daß eine derartige Reformulierung der Syntax nicht akzeptabel ist (unmotivierte nicht-terminale Symbole und Regeln).

2. Eine Syntax ohne Tilgungsregeln erlaubt eine andere Lösung: Eine Ableitung für einen Satz muß nur dann weiterverfolgt werden, wenn die Zahl der Symbole

[10]Unäre Regel, auch *Kettenregeln* genannt, sind Regeln, deren rechte Seite aus einem einzelnen nicht-terminalen Symbol besteht.

in ABLEITUNG kleiner oder höchstens gleich der Zahl der noch zu verarbeitenden Worte ist. So kann etwa im vorangegangenen Beispiel ausgeschlossen werden, daß die Regel $AP \rightarrow AP\ adj$ überhaupt angewendet wird; denn schon in Zeile 6' ist die Längenbegrenzung verletzt:$|SATZ| = 3 \wedge |ABLEITUNG| = 4$.

Enthält die Syntax dagegen Tilgungsregeln, und kommt eine Umformung in eine Syntax ohne links-/rechtsrekursive Regeln aus oben genannten Gründen nicht in Frage, ist es allerdings kaum möglich, eine Terminierung des Parsers in allen Fällen sicherzustellen: Mit einer einfachen Längenbegrenzung von ABLEITUNG zu operieren, ist nicht möglich. Wenn die Syntax nur links- oder nur rechtsrekursiv ist, kann die Verarbeitungsrichtung so gewählt werden, daß der Parser immer terminiert. Aber in allen anderen Fällen ist mit einfachen Mitteln, wenn man von Adhoc-Lösungen absieht (z.B. die Einführung von Bedingungen wie: jede Regel darf nur x-mal hintereinander ausgeführt werden), kein Erfolg zu erzielen.

3.4 Implementierung

3.4.1 Lisp

Die folgenden vier Programme orientieren sich eng an den in diesem Kapitel entwickelten Algorithmen und demonstrieren, wie gering die 'semantische Lücke' zwischen klar formulierten Algorithmen und (Lisp-)Programmen sein kann. Das erste Programm 'übersetzt' das Top-down-Erkennungsschema in ein einfaches Lisp-Programm. Dieses Programm wird anschließend zu einem Recognizer mit Backtracking erweitert. Es folgen die Implementierungen des Breadth-first-Erkennungs- und Parsingalgorithmus ohne Backtracking.

Alle Programme setzen voraus, daß die zur Analyse erforderlichen Daten (Syntax und Lexikon) den globalen Variablen *syntax* und *lexikon* zugewiesen sind[11]. Mit Ausnahme des ersten Programms gestatten außerdem alle Programme nicht nur die Analyse vollständiger Sätze, sondern auch die einzelner Konstituenten. In diesem Fall ist beim Funktionsaufruf neben dem Satz (als Liste von Symbolen) als zweites Argument der Konstituententyp anzugeben (als Liste, deren einziges Element ein Symbol ist, das den Konstituententyp bezeichnet).

Beispiel (3-7)

```
> (parse_td/bf '(der mond scheint))
  (S (NP (DET DER) (N MOND)) (VP (V SCHEINT)))
  T
> (parse_td '(der mond) '(np))
```

[11] Die Beschreibung der Repräsentation von Syntax, Lexikon und einigen fundamentalen Grundfunktionen findet sich im Anhang D.1.

```
(NP (DET DER) (N MOND))
T
```

Top-down-Recognizer (Schema)

```
;;; RECOGNIZE-TD
;;; Diese Funktion bildet das Top-down-Erkennungsschema ab. Als Kontrollstruktur
;;; wird eine LOOP-Form verwendet.
(defun recognize-td (satz)
  (let ((wort 0) (ableitung `(,(startsymbol)))) ; Initialisieren der Variablen
    (declare (special wort))                    ; WORT und ABLEITUNG
    (loop
      (if (and (endp ableitung) (= wort (length satz)))
          (return t))                           ; Satz akzeptiert
          (if (or (endp ableitung) (lexkat-p (first ableitung)))
              (return nil)                      ; Satz nicht akzeptiert
              (setq ableitung                   ; Fortsetzen der Ableitung
                    (reduziere (expandiere ableitung) (nthcdr wort satz)))))))

;;; EXPANDIERE
;;; Die Ableitung wird durch Anwendung der ersten Regel, deren linke Seite mit dem
;;; ersten Symbol von ABLEITUNG identisch ist, fortgesetzt.
(defun expandiere (ableitung)
  (let ((regeln (expansionen (first ableitung))))
    (if regeln (append (first regeln) (rest ableitung)))))

;;; REDUZIERE
;;; Aus ABLEITUNG werden - soweit möglich - alle führenden lexikalischen Katego-
;;; rien getilgt.
(defun reduziere (ableitung satz)
  (declare (special wort))
  (cond ((element_von-p (first satz) (first ableitung))
         (setq wort (+ wort 1)) (reduziere (rest ableitung) (rest satz)))
        (t ableitung)))

;;; Hilfsfunktionen

;;; EXPANSIONEN
;;; Die Funktion generiert für eine syntaktische Kategorie die Menge aller rechten
;;; Regelseiten, durch die dieses Symbol expandiert werden kann.
(defun expansionen (symbol)
  (apply #'append
         (mapcar #'(lambda (x) (and (eq (first x) symbol) (list (rest x))))
```

```
                    (regeln))))

;;; ELEMENT_VON-P
;;; Die Funktion prüft, ob es in *LEXIKON* einen Eintrag für WORT gibt, durch
;;; den der Wortform die lexikalische Kategorie KATEGORIE zugeordnet wird.
(defun element_von-p (wort kategorie)
  (when (lexkat-p kategorie) (member kategorie (eintrag-kategorien wort))))

;;; LEXKAT-P
;;; Die Funktion prüft, ob KATEGORIE eine lexikalische Kategorie ist.
(defun lexkat-p (symbol)
  (member symbol (lexikalische-kategorien)))
```

Top-down-Recognizer mit Backtracking

```
;;; RECOGNIZE-TD/DF
;;; Top-down-Recognizer mit Backtracking und Depth-first-Suche. Die beiden Funk-
;;; tionen EXPANDIERE-REDUZIERE und BACKTRACK steuern die Fortsetzung
;;; der aktuellen Ableitung bzw. die Wahl einer Analysealternative.
(defun recognize-td/df (satz &optional (symbol `(,(startsymbol))))
  (let ((wort 0) (ableitung symbol) (stack ()))
    (declare (special wort))
    (loop
      (if (and (= wort (length satz)) (endp ableitung))
          (return t))                              ; Satz akzeptiert
          (if (or (endp ableitung) (lexkat-p (first ableitung)))
              (if (endp stack) (return nil)        ; Satz nicht akzeptiert
                  ;; Backtracking:
                  (let ((backtrack-werte (backtrack stack satz)))
                    (setq stack     (first backtrack-werte)
                          ableitung (second backtrack-werte))))
              ;; Fortsetzung der Ableitung:
              (let ((ex/re-werte (expandiere-reduziere ableitung stack satz)))
                  (setq stack       (first ex/re-werte)
                        ableitung   (second ex/re-werte)))))))

;;; EXPANDIERE-REDUZIERE
;;; Die Funktion berechnet die Menge aller Regeln, die sich zur Fortsetzung der ak-
;;; tuellen (Links-)Ableitung verwenden lassen. Die erste dieser Regeln wird ange-
;;; wendet, und die übrigen werden in einem neu erzeugten Stackeintrag gespeichert.
(defun expandiere-reduziere (ableitung stack satz)
  (declare (special wort))
```

```
  (let ((regelseiten (expansionen (first ableitung))))
    (when regelseiten                       ; mindestens eine Regel anwendbar
      (if (endp (rest regelseiten))         ; genau eine Regel anwendbar
          (list stack
                (reduziere
                  (append (first regelseiten) (rest ableitung))
                  (nthcdr wort satz)))
          (list (cons (list wort ableitung (rest regelseiten)) stack)
                (reduziere
                  (append (first regelseiten)
                  (rest ableitung)) (nthcdr wort satz)))))))

;;; BACKTRACK
;;; Beim Backtracking werden die Variablen WORT, ABLEITUNG und STACK auf
;;; der Grundlage des ersten Stackeintrags aktualisiert. Die Funktion BACKTRACK
;;; berechnet die neuen Werte für diese Variablen.
(defun backtrack (stack satz)
  (declare (special wort))
  (setq wort (first (first stack)))
  (list (if (endp (rest (third (first stack))))      ; keine weitere Alternative
            (rest stack)                             ; entferne obersten Eintrag
            (subst (rest (third (first stack))) (third (first stack)) stack))
        (reduziere
          (append (first (third (first stack))) (rest (second (first stack))))
          (nthcdr wort satz))))
```

Ein breadth-first arbeitender Top-down-Recognizer

Anders als die rein iterativen top-level Funktionen der letzten beiden Programme verbinden die Funktionen RECOGNIZE-TD/BF und PARSE-TD/BF iterative und rekursive Elemente.

```
;;; RECOGNIZE-TD/BF
;;; Diese Funktion basiert auf dem Breadth-first-Erkennungsalgorithmus (ohne Back-
;;; tracking) aus Kapitel 3.2.2.
(defun recognize-td/bf (satz &optional (ableitung `(,(startsymbol))))
  (cond ((and (endp satz) (endp ableitung)) t)
        ((or (endp ableitung) (lexkat-p (first ableitung))) nil)
        (t (dolist (x (expansionen (first ableitung)))
             (print (apply #'recognize-td/bf
                           (reduziere-bf satz (append x (rest ableitung)))))))))
```

```
;;; REDUZIERE-BF
;;; Die Aufgabe dieser Funktion ist es, möglichst alle führenden lexikalischen Kate-
;;; gorien aus ABLEITUNG zu entfernen. Der Unterschied zu REDUZIERE besteht
;;; darin, daß REDUZIERE-BF als Wert die Liste zurückgibt, die die Linkssatzform
;;; und den noch nicht analysierte Satzabschnitt enthält.
(defun reduziere-bf (satz ableitung)
  (if (element_von-p (first satz) (first ableitung))
      (reduziere-bf (rest satz) (rest ableitung))
      (list satz ableitung)))
```

Ein breadth-first arbeitender Top-down-Parser

```
;;; PARSE-TD/BF
;;; Zur Generierung von Strukturbeschreibungen wird ein weiterer Parameter
;;; STRUKTUR eingeführt.
(defun parse-td/bf (satz &optional (ableitung `(,(startsymbol))) struktur)
  (cond ((and (endp satz) (endp ableitung)) (transform struktur))
        ((or (endp ableitung) (lexkat-p (first ableitung))) nil)
        (t (dolist (x (expansionen (first ableitung)))
             (print (apply #'parse-td/bf
                           (reduziere-bf*
                             satz
                             (append x (rest ableitung))
                             (cons (cons (first ableitung) x) struktur))))))))
```

```
;;; REDUZIERE-BF*
;;; Diese Funktion entfernt - soweit möglich - alle führenden lexikalischen Kategorien
;;; aus ABLEITUNG. Als Wert liefert sie eine Liste, die die Linkssatzform, den noch
;;; nicht analysierten Satzabschnitt und die bislang generierte Strukturbeschreibung
;;; enthält.
(defun reduziere-bf* (satz ableitung struktur)
  (if (element_von-p (first satz) (first ableitung))
      (reduziere-bf* (rest satz) (rest ableitung)
                     (cons (list (first ableitung) (first satz)) struktur))
      (list satz ableitung struktur)))
```

Die folgenden drei Funktionen werden für die top-down Parser benötigt. Sie speichern die Informationen über die Ableitungsgeschichte in Form einer Liste der Regeln, die bei dieser Ableitung (erfolgreich) verwendet wurden. TRANSFORM generiert aus dieser Regelliste eine Strukturbeschreibung.

```
;;; TRANSFORM
;;; Diese Funktion erzeugt aus einer Folge von Regeln eine Strukturbeschreibung.
(defun transform (struktur)
  (do* ((listen struktur
                (replace-element
                  (replace-element (first listen)
                                   (length (member (caar liste) (reverse liste)))
                                   liste)
                  (- (length listen) (length (member liste listen :test #'equal)))
                  (rest listen)))
        (liste (find-element (caar listen) (rest listen))
               (find-element (caar listen) (rest listen))))
        ((endp (rest listen)) (first listen))))

;;; FIND-ELEMENT
;;; Die Funktion findet die Liste, in der die andere Liste einzufügen ist.
(defun find-element (symbol list-of-lists)
  (dolist (x list-of-lists)
    (when (member symbol (rest x)) (return x))))

;;; REPLACE-ELEMENT
;;; Ersetze ZAHL-tes Element in LISTE durch NEU-Element.
(defun replace-element (neu zahl liste)
  (if (< (length liste) zahl)
      liste
      (let ((resultat ())
            (zaehler 1))
        (dolist (x liste)
          (if (< zaehler zahl)
              (setf zaehler (+ zaehler 1)
                    resultat (append resultat (list x)))
              (return (append resultat (list neu) (nthcdr zaehler liste))))))))
```

3.4.2 Prolog

Da ein Prolog-Interpreter eine Top-down-depth-first-Beweisstrategie mit Backtracking verwendet, ist die Realisierung eines entsprechenden Parsers in dieser Programmiersprache besonders einfach: Eine explizite Definition der prozeduralen Elemente des Algorithmus ist nicht erforderlich, es müssen lediglich die Datenstrukturen und -formate sowie Vor- und Nachbedingungen der Prozedur definiert werden. Es läßt sich hier also in der Tat ein konsequent deklarativer Programmierstil realisieren.

In den meisten neueren Prolog-Systemen ist der DCG-Formalismus im allgemeinen, bereits vordefinierten Befehlssatz enthalten. DCGs[12] (Definite Clause Grammars) sind Grammatiken, die kontextfreie Syntaxen als echte Teilmenge enthalten, so daß die Implementierung eines Top-down-Parsers für kontextfreie Syntaxen in Prolog in der Praxis selten sinnvoll ist. Aber selbst in dem Fall, daß DCGs und ihre Verarbeitung nicht bereits vordefiniert sind, ist die Programmierung einer solchen Komponente mit einfachen Mitteln realisierbar. Um ein Regelformat zu erhalten, das den üblichen Notationskonventionen für kontextfreie Syntaxen nahe kommt, ist es sinnvoll, einen Infix-Operator '—>' zu definieren. Die Ebene innerhalb der Operatorenhierarchie (das erste Argument einer Operatorendefinition) wird in verschiedenen Prolog-Implementierungen durch unterschiedliche Zahlenwerte angegeben; der hier angegebene Wert von 1100 bezieht sich auf Arity-Prolog (TM) und muß gegebenenfalls an die jeweils verwendete Prolog-Umgebung angepaßt werden.

```
:- op(1100, xfy, '—>').
```

Die Regeln der Grammatik lassen sich nun in der folgenden Form angeben:

K_1 —> [K_2, K_3, ..., K_n].

Das Startsymbol wird durch ein entsprechendes Prädikat festgelegt, z.B.:

```
startsymbol(s).
```

Schließlich müssen noch die Vokabulare definiert werden. Es ist für unsere Zwecke hinreichend, nur das terminale Vokabular anzugeben. Wir verwenden hierzu das einstellige Prädikat **terminal**. Zunächst definieren wir einen einfachen Recognizer: Der Recognizer nimmt eine Kette über dem terminalen Vokabular als Eingabe und liefert als Ausgabe 'yes' oder 'no'. Für die Ausgabe wird keine Resultatsvariable benötigt, weil man die Meldung des Prolog-Interpreters für diesen Zweck verwenden kann. Das Prädikat **recognize** hat also nur ein Argument, nämlich die Eingabekette; der Kopf der Prädikatsdefinition ist also: recognize(X). Zunächst sollte geprüft werden, ob die Elemente der Eingabekette Symbole aus dem terminalen Alphabet sind:

```
terminale_kette([ ]).
terminale_kette([X|R]) :-
   terminal(X),
   terminale_kette(R).
```

Anschließend wird geprüft, ob die Eingabekette eine Expansion des Startsymbols ist. Die vollständige Definition lautet also:

```
recognize(E) :-
   terminale_kette(E),        % E ist eine terminale Kette
   startsymbol(S),            % S ist das Startsymbol
   expansion([S], E).         % E ist eine Expansion von S
```

[12] Vgl. Kapitel 12.

Das Prädikat **expansion** hat zwei Argumente, die Eingabe und Ausgabe der durch **expansion** definierten Prozedur darstellen. Die Argumente sind jeweils Listen von terminalen und/oder nicht-terminalen Symbolen, wobei das zweite Argument durch Anwendung der Regeln aus dem ersten Argument ableitbar ist. Im Initialzustand ist das erste Argument von **expansion** eine Liste, die allein das Startsymbol enthält, das zweite Argument ist die Eingabekette. Die Anwendung der Regeln auf die Elemente der ersten Liste erfolgt von links nach rechts (das erste Element der Liste wird auch zuerst expandiert). Weiterhin ist die Verarbeitungsstrategie depth-first: Bevor also andere Expansionsalternativen in Betracht gezogen werden, wird der Versuch unternommen, das Resultat der ersten vorgenommenen Expansion noch weiter zu expandieren. Dieser Verarbeitungsprozeß ist rekursiv und terminiert, wenn keine weiteren Expansionen mehr durchgeführt werden können.

```
expansion([K|R], E) :-
    (K '—>' Ts),              % Es gibt eine Regel, die K zu Ts expandiert.
    expansion(Ts, E2),        % Depth-first-Rekursion: Expandiere das
                              % Resultat der ersten Regelanwendung zu E2.
    expansion(R, R2),         % links-rechts Rekursion: Expandiere den Rest.
    append(E2, R2, E).        % Verbinde die Resultate mit append.
expansion(X, X).              % Terminierung
```

Die Erweiterung des Recognizers zu einem Parser erfordert ein zusätzliches Argument, das als Resultatsvariable für die Strukturbeschreibung dient. Parallel zur Regelanwendung wird ein Strukturbaum in Klammernotation aufgebaut.

```
parse(X) :-
    terminale_kette(X),       % Ist die Eingabekette wohlgeformt?
    startsymbol(S),
    parse([S], X, Struktur),  % Aufruf des Prädikats parse/3
    nl,write(Struktur).       % Ausgabe der Strukturbeschreibung
parse([K|R], E, [K, S2|SR]) :-
    (K '—>' Ts),
    parse(Ts, E2, S2),
    parse(R, R2, SR),
    append(E2, R2, E).
    parse(X, X, X).
```

Vergleicht man die in diesem Kapitel präsentierten Programme miteinander, so ist offensichtlich, daß Lisp-Implementierungen von Algorithmen mit einer Depth-first-Strategie mit Backtracking fast immer deutlich umfangreicher ausfallen als vergleichbare Prolog-Programme. Dieses Ergebnis kann kaum überraschen, da in diesem Fall die von Prolog als Beweisstrategie verwendete Depth-first-Suche mit Backtracking direkt genutzt werden kann, sie in Lisp dagegen explizit definiert werden muß. Anders sieht es bei Breadth-first-Algorithmen aus: Ihre Realisierung durch ein Prolog-Programm ist häufig mit einem gewissen Aufwand verbunden, da es zunächst gilt,

die zu Verfügung stehende Beweisstrategie zu 'überlisten'. Wir werden uns aus diesem Grund im folgenden weitgehend darauf beschränken, Algorithmen des ersten Typs in Prolog und Algorithmen des zweiten Typs in Lisp zu implementieren.

Aufgaben

3.1 Verwenden Sie beide in 3.1 beschriebenen Verfahren zur Verwaltung von Strukturbeschreibungen, um einen Top-down-Parser mit Backtracking in Lisp zu implementieren.

3.2 Implementieren sie einen breadth-first arbeitenden Top-down-Parser in Prolog.

3.3 Schreiben Sie eine Trace-Prozedur, die in die in diesem Kapitel vorgestellten Programme eingebunden werden kann und bei der Analyse von Sätzen Informationen über den Analyseverlauf ausgibt.

3.4 Entwickeln Sie ein Programm, das prüft, ob eine Syntax

(a) Tilgungsregeln,

(b) linksrekursive Regeln,

(c) rechtsrekursive Regeln,

(d) nicht-terminierbare Symbole und

(e) nicht S-erreichbare Symbole enthält

und anschließend einen *Analysebericht* ausgibt.

3.5 Entwickeln Sie ein Programm, das Syntaxen und Lexika in der hier verwendeten Lisp-(Prolog-)Notation in die entsprechende Prolog-(Lisp-)Notation übersetzt.

3.6 Entwickeln und implementieren Sie auf Grundlage der in diesem Kapitel vorgestellten Algorithmen einen Top-down-Generator, d.h. ein Programm, das auf der Grundlage eines Lexikons und einer Syntax Sätze bzw. Strukturbeschreibungen für Sätze generiert, die zu der von der Syntax festgelegten Sprache gehören.

Kapitel 4

Bottom-up-Parsing

Wir werden zunächst die Grundkonzepte des Bottom-up-Parsings anhand eines Algorithmus für einen einfachen deterministischen Shift-reduce-Recognizer erläutern. Auf der Grundlage dieses Algorithmus entwickeln wir anschließend einen Shift-reduce-Parser mit Depth-first-Suche und Backtracking und einen breadth-first operierenden Shift-reduce-Parser.

4.1 Grundzüge

Bottom-up-Recognizer und Parser arbeiten bei der Analyse eines Satzes w *datengesteuert*, d.h. sie versuchen w in einem oder mehreren Schritten auf das Startsymbol der Syntax zurückzuführen. Die Regeln der Syntax werden dabei von rechts nach links angewendet: Die Symbole einer rechten Regelseite werden durch das Symbol auf der linken Regelseite ersetzt. Diese Ersetzungsrichtung wird „Reduktion" genannt. Ein Satz wird akzeptiert, wenn es gelingt, eine Folge von Reduktionsschritten zu finden, die ihn auf das Startsymbol der Syntax reduziert.

Die wichtigste Klasse der Bottom-up-Parser bilden die Shift-reduce-Parser. Ihr Name resultiert aus der Bezeichnung der beiden Operationen, die sie verwenden: *shift* und *reduce*. Diese Operationen lassen sich als Manipulationen von Stacks auffassen: Ein Stack enthält den Satz w bzw. den noch nicht analysierten Satzabschnitt, und ein anderer Stack wird dazu benutzt, das (Zwischen)Ergebnis der Analyse zu speichern. Die Verarbeitung von w erfolgt zumeist von links nach rechts. Zunächst wird das erste Wort von w vom ersten in den zweiten Stack übertragen (*shift*). Dann wird versucht, den Stackinhalt des zweiten Stacks mithilfe einer der Regeln der Syntax zu reduzieren (*reduce*). Nur wenn keine weiteren Reduktionen mehr möglich sind, wird das jeweils nächste Wort von w vom ersten in den zweiten Stack übertragen und bei der Suche nach weiteren Reduktionsmöglichkeiten berücksichtigt.

Ähnlich wie beim Top-down-Parsing gibt es auch beim Bottom-up-Parsing bes-

timmte Restriktionen für die Syntaxen, die mit diesen Algorithmen verarbeitet werden können. Probleme bereiten in diesem Fall nicht links- oder rechtsrekursive Regeln, sondern Tilgungsregeln und solche unären Regeln[1], die zu Zyklen führen. Wir müssen aus diesem Grund für die in diesem Kapitel formulierten Algorithmen fordern, daß die verwendete kontextfreie Syntax $G = <V_N, V_T, S, R>$ zyklenfrei ist und keine Tilgungsregeln enthält.
Diese beiden Restriktionen sind notwendig, um sicherzustellen, daß die Analyse eines Satzes in jedem Fall terminiert; denn Zyklen und Tilgungsregeln führen dazu, daß die Analyse eines Satzes stagniert: Eine Tilgungsregel $X \rightarrow e$ kann beliebig oft angewendet werden, um den zweiten Stack zu reduzieren. Während die Reduktion mit anderen Regeln die Größe des Stacks monoton sinken läßt, führt dagegen jede Anwendung einer Tilgungsregel dazu, daß ein neues Symbol auf den zweiten Stack gepusht wird, ohne daß vorher Symbole entfernt werden. Zyklen können zu ähnlichen Problemen führen.

Beispiel (4-1)

Wenn es in der Syntax eine Regel $X \rightarrow e$ gibt und für den zweiten Stack STACK2 gilt: STACK2 = $[Y_1, \ldots, Y_n]$, mit $Y_i \in V$ und $n \geq 0$, dann enthält STACK2 nach m Anwendungen dieser Regel m Vorkommen von X; d.h. STACK2 hat anschließend die Form $[\underbrace{X, \ldots, X}_{m-mal}, Y_1, \ldots, Y_n]$.

4.2 Ein deterministischer Shift-reduce-Recognizer

Ein- & Ausgabe. Wie bei jedem Recognizer besteht auch beim deterministischen Shift-reduce-Recognizer die Eingabe aus einer Kette $w = w_1 \ldots w_n$ $(n \geq 1)$ von terminalen Symbolen. Die Ausgabe lautet *True* gdw. $\alpha \in L(G)$; sonst *False*.

Restriktionen. Für den Shift-reduce-Recognizer, den wir zuerst entwickeln, ist neben den oben formulierten Restriktionen noch zu fordern, daß die Regelmenge keine zwei Regeln enthält, deren rechte Regelseiten ein gemeinsames Präfix besitzen. Diese zusätzliche Restriktion erlaubt eine deterministische Verarbeitung.

Arbeitsstrukturen. Aus Gründen der Einheitlichkeit repräsentieren wir alle Strukturen, die von den Prozeduren des Shift-reduce-Recognizers manipuliert werden, als Stacks, obwohl z.B. anstelle des ersten Stacks auch ein Zähler oder Zeiger verwendet werden könnte. Es werden zwei Stacks benötigt:

- STACK1
  Der Stack enthält den noch nicht verarbeiteten rechten Teil des Satzes. Zunächst enthält STACK1 alle Wörter des Satzes w und hat die Form $[w_1, \ldots, w_n]$, für $w = w_1 \ldots w_n$.

[1] Als *unäre* Regeln bezeichnet man Regeln, deren rechte Seite aus genau einem Symbol besteht.

- STACK2
  Der Stack enthält die bereits gefundenen Ableitungen für Teilstücke der Eingabekette. Die Objekte in STACK2 sind Elemente aus $V_N \cup V_T$. Zunächst ist STACK2 leer.

Prozeduren. Da beide Operationen relativ einfach sind, beschreiben wir sie durch Angabe der Eingabe-/Ausgabe-Relation und der für sie geltenden Vorbedingungen. Die Vorbedingung für die Prozedur SHIFT ist, daß STACK1 nicht leer ist. Die Elemente von STACK1 sind die Wörter des zu analysierenden Satzes w, die noch nicht verarbeitet wurden. Die Prozedur SHIFT überträgt das Topelement von STACK1 auf STACK2.

PROZEDUR SHIFT

EINGABE: $\text{STACK1} = [w_1, w_2, \ldots, w_j]$
$\text{STACK2} = [O_1, O_2, \ldots, O_k]$

AUSGABE: $\text{STACK1} = [w_2, \ldots, w_j]$
$\text{STACK2} = [w_1, O_1, O_2, \ldots, O_k]$

VORBEDINGUNGEN:
$j > 0$

Die Prozedur REDUCE versucht die ersten Symbole von STACK2 durch Anwendung einer lexikalischen oder syntaktischen Regel zu reduzieren:

PROZEDUR REDUCE

DATEN: Eine kontextfreie Syntax $G = <V_N, V_T, S, R>$ und ein Lexikon L.
EINGABE: $\text{STACK2} = [O_1, O_2, \ldots, O_i, O_{i+1}, \ldots, O_k]$
AUSGABE: $\text{STACK2} = [K, O_{i+1}, \ldots, O_k]$

VORBEDINGUNGEN:

(a) $k > 0$;

(b) Es gibt eine Regel $r \in R$ der Form $K \rightarrow O_i\ O_{i-1} \ldots O_1$ bzw.
eine lexikalische Kategorie K und $O_1 \in K$ (in diesem Fall gilt: $i = 1$).

Beispiel (4-2)

DATEN : $R = \{\ldots, S \rightarrow NP\ VP, VP \rightarrow v, \ldots\}$
$L = \{\ldots, (\text{geht } v) \ldots\}$

(1) EINGABE : STACK2 = [geht, NP]
AUSGABE : STACK2 = [v, NP]

(2) EINGABE : STACK2 = [v, NP]
AUSGABE : STACK2 = [VP, NP]

(3) EINGABE : STACK2 = [VP, NP]
AUSGABE : STACK2 = [S]

Die Vorbedingung (a) fordert, daß STACK2 mindestens ein Element enthält. Diese invariante Vorbedingung ist nach der ersten Anwendung von SHIFT immer erfüllt, da es keine Prozeduren gibt, die nur Elemente von STACK2 entfernen, ohne zugleich auch neue Elemente in diesen Stack einzufügen.

Vorbedingung (b) sichert die Existenz einer anwendbaren Regel bzw. eines geeigneten Lexikoneintrags: Wenn es eine Regel r aus der Regelmenge R gibt, deren rechte Regelseite i Symbole enthält und die Inversion der ersten i Elemente von STACK2 mit der rechten Regelseite von r identisch ist, dann werden diese Elemente durch die linke Regelseite von r ersetzt. Daß die Symbole der rechten Regelseite einer anwendbaren Regel in STACK2 in genau umgekehrter Reihenfolge erscheinen, liegt daran, daß sich die Reihenfolge der Wörter der Eingabekette bereits bei der Übertragung von STACK1 auf STACK2 umkehrt. Ist das erste Symbol in STACK2 ein Wort w_i, für das es einen Lexikoneintrag gibt, der ihm eine (lexikalische) Kategorie K zuordnet, kann w_i durch K ersetzt werden.

ALGORITHMUS RECOGNIZE$_{BU}$
bottom-up / links-rechts
deterministisch

DATEN: Eine kontextfreie Syntax G = <V_N, V_T, S, R> und ein Lexikon L.
EINGABE: STACK1 - Anfangswert: [$w_1, \ldots, w_n$], $n \geq 1$
STACK2 - Anfangswert: [ ]
AUSGABE: *True*/*False*.
METHODE:

Wenn <Alle Vorbedingungen für *Reduce* sind erfüllt>
Dann <*Recognize*$_{BU}$(STACK1, *Reduce*(STACK2))>
Sonst
    Wenn <Alle Vorbedingungen für *Shift* sind erfüllt>
    Dann <*Recognize*$_{BU}$ (*Shift*(STACK1, STACK2))>
    Sonst
        Wenn <STACK2 = [Startsymbol$_G$]>
        Dann <RETURN(*True*)>
        Sonst <RETURN(*False*)>

RECOGNIZE$_{BU}$ steuert die Interaktionen zwischen den Prozeduren SHIFT und REDUCE: Solange die Vorbedingungen der Prozedur REDUCE erfüllt sind, wird REDUCE angewendet. Wenn keine weiteren Reduktionen mehr möglich sind, wird das nächste Wort der Eingabekette in STACK2 übertragen (SHIFT) und der Verarbeitungsprozeß durch den erneuten Aufruf von RECOGNIZE$_{BU}$ fortgesetzt. Wenn die Eingabekette abgearbeitet ist und SHIFT nicht mehr angewendet werden kann, wird überprüft, ob die Eingabekette vollständig auf das Startsymbol reduziert werden konnte. In diesem Fall wird die Erfolgsmeldung *True* ausgegeben, sonst *False*.

Beispiel (4-3)

Wenn G eine kontextfreie Syntax ist, die die Regeln
1 : S → NP VP   2 : NP → det n   3 : VP → v
enthält und L ein Lexikon ist, so daß gilt:
4 : der ∈ det   5 : Hund ∈ n   6 : bellt ∈ v
dann verläuft die Analyse des Satzes „Der Hund bellt“ wie folgt:

OPERATION	STACK1	STACK2
Anfangszustand	[der, Hund, bellt]	[ ]
Shift	[Hund, bellt]	[der]
Reduce	[Hund, bellt]	[det]
Shift	[bellt]	[Hund, det]
Reduce	[bellt]	[n, det]
Reduce	[bellt]	[NP]
Shift	[ ]	[bellt, NP]
Reduce	[ ]	[v, NP]
Reduce	[ ]	[VP, NP]
Reduce	[ ]	[S]

4.3 Ein Shift-reduce-Recognizer mit Backtracking

Der einfache deterministische Shift-reduce-Recognizer soll nun so erweitert werden, daß auch globale und lokale Mehrdeutigkeiten behandelt werden können. Zu diesen Typen von Ambiguität kommt es, wenn entweder

1. die verwendete Grammatik zwei Regeln enthält, für die gilt, daß die rechte Seite der einen Regel ein Präfix der rechten Seite der anderen Regel ist, oder

2. das zugrundegelegte Lexikon Einträge enthält, die einem Wort des Satzes verschiedene lexikalische Kategorien zuordnen.

Wenn die Syntax z.B. die Regeln VP → v und VP → v NP enthält, dann kann bei der Analyse eines Satzes eine Situation eintreten, in der STACK2 durch beide Regeln reduziert werden kann, aber nur eine von ihnen die erfolgreiche Beendigung der Analyse ermöglicht. Gleiches gilt für Fälle von lexikalischer Ambiguität.

Es gibt verschiedene Möglichkeiten, dieses Problem zu lösen: Es kann versucht werden, durch Vorausschau auf weitere Symbole der Eingabekette („Look-ahead“) die korrekte Regel bzw. lexikalische Kategorie zu identifizieren; oder man verfolgt alle möglichen Alternativen quasi-parallel (Breadth-first-Suche, vgl. 3.6). Die bei Shift-reduce-Algorithmen am häufigsten verwendete Lösung besteht darin, einen

Pfad im Suchraum (d.h. in diesem Falle: eine Folge von Reduktionen und Shifts) solange weiterzuverfolgen, bis er sich als definitiv falsch erweist, und die Verarbeitungsschritte dann soweit zurückzunehmen („Backtracking“), bis eine frühere Situation erreicht wird, in der eine bislang ungeprüfte Alternative zum erfolglos eingeschlagenen Pfad gefunden wird. In diesem Fall wird der alternative Pfad weiterverfolgt, bis er zum Erfolg führt oder ein erneutes Backtracking erforderlich wird.

4.3.1 Daten und Prozeduren

Syntax & Lexikon. Wie auch beim deterministischen Shift-reduce-Recognizer ist vorauszusetzen, daß die kontextfreie Grammatik G = $<V_N, V_T, S, R>$ zyklenfrei ist und keine Tilgungsregeln enthält. Außerdem wird eine (beliebige) Reihenfolge für die Regeln in R und Lexikoneinträge in L angenommen, so daß es eine Funktion f von der Menge der Syntaxregeln und Lexikoneinträge in die Menge der natürlichen Zahlen gibt. Jeder Regel und jedem Lexikoneintrag wird durch diese Funktion f genau eine Zahl aus der Wertemenge zugeordnet, und für jede Zahl gibt es genau eine Regel bzw. einen Eintrag. Diese Zahl wird als „Regelindex“ bzw. „Eintragsindex“ bezeichnet. Wir gehen im folgenden davon aus, daß die Regeln und das Lexikon in dieser partiellen Ordnung vorliegen und alle Prozeduren die Regeln bzw. Lexikoneinträge in dieser Reihenfolge auf ihre Anwendbarkeit überprüfen.

Arbeitsstrukturen. Neben STACK1 und STACK2 ist ein weiterer Stack erforderlich, in dem die für das Backtracking notwendigen Informationen gespeichert werden:

- STACK3: Der Stack repräsentiert die bisherige Verarbeitungsgeschichte in einem Verarbeitungszustand Z. Er enthält Regel- und Eintragindizes (Zahlen zwischen 1 und CARD(R ∪ L)) bzw. die Zahl 0 als Markierung für die *Shift*-Operation.

Prozeduren. Die Prozeduren SHIFT und REDUCE werden so modifiziert, daß sie die für das Backtracking erforderlichen Informationen in den STACK3 eintragen:

PROZEDUR SHIFT$_{DF}$	
EINGABE:	STACK1 = $[w_1, w_2, \ldots, w_j]$
	STACK2 = $[O_1, \ldots, O_k]$
	STACK3 = $[I_1, \ldots, I_m]$
AUSGABE:	STACK1 = $[w_2, \ldots, w_j]$
	STACK2 = $[w_1, O_1, \ldots, O_k]$
	STACK3 = $[0, I_1, \ldots, I_m]$
VORBEDINGUNGEN:	$j > 0$

Die Vorbedingung für die Prozedur SHIFT$_{DF}$ ist, daß es in STACK1 mindestens ein Element gibt. Die Elemente von STACK1 sind die Wörter des Satzes, die noch nicht verarbeitet worden sind. Die Prozedur SHIFT$_{DF}$ überträgt das Topelement

von STACK1 auf STACK2. Diese Operation wird vermerkt, indem 0 als neues Topelement auf STACK3 gepusht wird.

PROZEDUR REDUCE$_{DF}$

DATEN: Eine kontextfreie Syntax G = <V_N, V_T, S, R> und ein Lexikon L.
EINGABE: STACK2 = [O_1, O_2, ..., O_i, O_{i+1}, ..., O_k]
STACK3 = [I_1, I_2, ..., I_j]
AUSGABE: STACK2 = [K, O_{i+1}, ..., O_k]
STACK3 = [m, I_1, I_2, ..., I_j]

VORBEDINGUNGEN:

(a) $k > 0$.

(b) $j > 0$.

(c) Es gibt ein r, mit f(r) = m. Wenn $r \in R$, dann hat r die Form $K \rightarrow O_i\ O_{i-1} \ldots O_1$; sonst gilt: i = 1 und r ist ein Lexikoneintrag, der O_1 die lexikalische Kategorie K zuordnet.

(d) Es gibt kein weiteres r', das diese Bedingung erfüllt und einen Index f(r') = m' hat, so daß m' < m.

Für REDUCE$_{DF}$ gilt: Wenn es eine Regel r aus der Regelmenge R gibt, deren rechte Regelseite eine Inversion der ersten i Elemente von STACK2 ist, dann werden diese Elemente durch die linke Regelseite von r ersetzt, sofern diese Regel nicht bereits in derselben Situation angewendet wurde. Wenn das erste Objekt in STACK2 ein Symbol der Eingabekette ist und es für dieses Symbol einen Eintrag l in L gibt, der noch nicht verwendet wurde, wird es durch die im Eintrag enthaltene lexikalische Kategorie ersetzt. Diese Bedingung ist erfüllt, wenn das Topelement I_1 von STACK3 einen niedrigeren Wert hat als der Regelindex f(r)/Eintragsindex f(l). Die durchgeführte Reduktion wird in STACK3 vermerkt, indem dieser Index als neues Topelement in STACK3 eingefügt wird[2].

Beispiel (4-4)
Nehmen wir an, es soll der Satz „Der Hund jagt die Katze“ analysiert werden. Zu Beginn der Analyse ist der Recognizer in folgendem Zustand:

STACK1 = [der, Hund, jagt, die, Katze]
STACK2 = [ ]
STACK3 = [0]

Dann wird das erste Symbol von STACK1 in STACK2 übertragen:

[2] Wir werden in der Beschreibung der folgenden Prozeduren nicht weiter zwischen syntaktischen Regeln und lexikalischen Einträgen differenzieren (vgl. Kapitel 2.3); d.h. ein Lexikoneintrag, der einem Symbol w die Kategorie K zuordnet, wird als eine Regel der Form $K \rightarrow w$ interpretiert.

STACK1 = [Hund, jagt, die, Katze]
STACK2 = [der]
STACK3 = [0]

Nehmen wir an, es gibt einen Lexikoneintrag in L, der „der“ die Kategorie „det“ zuordnet, und der Index dieses Eintrags ist 5. Dann sind alle Vorbedingungen der Prozedur $\mathrm{REDUCE}_{\mathrm{DF}}$ erfüllt, und wir erhalten:

STACK1 = [Hund, jagt, die, Katze]
STACK2 = [det]
STACK3 = [5, 0].

4.3.2 Backtracking

Backtracking ist erforderlich, wenn in einer Situation weder eine Reduktion noch ein Shift möglich ist und auch kein Endzustand (s.u.) erreicht ist. Die Schritte, die zu dieser Situation geführt haben, müssen deshalb sukzessive zurückgenommen werden, bis eine frühere Situation erreicht wird, in der es eine Alternative zum tatsächlich eingeschlagenen Weg gibt. Die erste Aktion beim Backtracking ist deshalb die Rücknahme des letzten Verarbeitungsschrittes. Wenn der letzte Arbeitsschritt eine Reduktion war, gibt es nach der Rücknahme dieser Reduktion folgende Möglichkeiten:

1. Es gibt eine andere, noch nicht geprüfte Reduktionsmöglichkeit als die bislang vorgenommenen; d.h. es gibt eine Regel, deren Index größer ist als der Index, der zur Zeit Topelement von STACK3 ist.

2. Es gibt keine solche Reduktionsalternative mehr, es kann aber stattdessen ein neues Element geshiftet werden; d.h. STACK1 enthält mindestens ein Element.

3. Es gibt weder Reduktionsalternativen, noch die Möglichkeit ein neues Wort zu shiften.

Eine Shift-Aktion wird nur in einer Situation vorgenommen, in der keine Reduktion möglich ist. Wenn also die letzte Aktion eine Shift-Aktion war, gibt es keine verschiedenen Alternativen: Der Shift-Schritt muß zurückgenommen werden und mindestens ein weiterer Schritt.

Diese vier verschiedenen Möglichkeiten lassen sich durch die folgenden Prozeduren erfassen:

PROZEDUR Backtracking-1

EINGABE: STACK2 = $[K_0, K_{i+1}, \ldots, K_n]$
STACK3 = $[I_0, I_1, I_2, \ldots, I_m]$

AUSGABE: STACK2 = $[K_0', K_{j+1}, \ldots, K_n]$
STACK3 = $[I_0', I_1, I_2, \ldots, I_m]$

VORBEDINGUNGEN:

(a) I_0 ist der Index einer Regel r = $K_0 \rightarrow K_i\ K_{i-1} \ldots K_1$, mit $1 \leq i \leq n$.

(b) Es gibt eine Regel r' = $K_0' \rightarrow K_j\ K_{j-1} \ldots K_1$ $(1 \leq j \leq n)$, so daß f(r') = I_0' und $I_0' > I_0$.

Der letzte Schritt war eine Reduktion; denn das Topelement I_0 von STACK3 ist der Index einer Regel r. Es gibt eine andere Reduktionsmöglichkeit (eine Regel r'): Die letzte Reduktion wird zurückgenommen und stattdessen die Reduktion durch r' vorgenommen. Da die rechten Seiten der beiden Regeln nicht gleich lang sein müssen, kann die Reduktion mit r' die Größe von STACK2 verändern.

PROZEDUR Backtracking-2

EINGABE: STACK1 = $[w_1, w_2, \ldots, w_x]$
STACK2 = $[K_0, K_{i+1}, \ldots, K_n]$
STACK3 = $[I_0, I_1, \ldots, I_m]$

AUSGABE: STACK1 = $[w_2, \ldots, w_x]$
STACK2 = $[w_1, K_1, K_2, \ldots, K_i, K_{i+1}, \ldots, K_n]$
STACK3 = $[0, I_1, \ldots, I_m]$

VORBEDINGUNGEN:

(a) I_0 ist der Index einer Regel r = $K_0 \rightarrow K_i\ K_{i-1} \ldots K_1$, mit $1 \leq i \leq n$

(b) $x \geq 1$.

Der letzte Schritt war ebenfalls eine Reduktion. Es gibt jedoch keine alternative Reduktionsmöglichkeit. Es kann aber stattdessen ein neues Element geshiftet werden. Die letzte Reduktion wird zurückgenommen, und das nächste Element der Eingabekette wird geshiftet.

PROZEDUR Backtracking-3

EINGABE: STACK1 = [ ]
STACK2 = $[K_0, K_{i+1}, \ldots, K_n]$
STACK3 = $[I_0, I_1, \ldots, I_m]$

AUSGABE: STACK1 = [ ]
STACK2 = $[K_1, K_2, \ldots, K_i, K_{i+1}, \ldots, K_n]$
STACK3 = $[I_1, \ldots, I_m]$

VORBEDINGUNG:
I_0 ist der Index einer Regel $r = K_0 \rightarrow K_i\ K_{i-1} \ldots K_1$, mit $1 \leq i \leq n$.

Der letzte Schritt war eine Reduktion. Es gibt aber keine weitere, bislang ungeprüfte Reduktion und es kann auch keine Shift-Operation durchgeführt werden, da STACK1 leer ist. Die letzte Reduktion wird zurückgenommen.

PROZEDUR Backtracking-4

EINGABE: STACK1 = $[w_1, w_2, \ldots, w_x]$
STACK2 = $[w_0, K_0, K_{i+1}, \ldots, K_n]$
STACK3 = $[0, I_1, \ldots, I_m]$

AUSGABE: STACK1 = $[w_0, w_1, w_2, \ldots, w_x]$
STACK2 = $[K_0, K_{i+1}, \ldots, K_n]$
STACK3 = $[I_1, \ldots, I_m]$

Der letzte Schritt war eine Shift-Operation, denn das Topelement von STACK3 ist „0“. Die Shift-Operation wird zurückgenommen.

Die Prozeduren Backtracking-1 und Backtracking-2 eröffnen einen neuen Pfad im Suchraum (durch eine neue Reduktion oder ein Shifting anstelle einer Reduktion), der anschließend weiterverfolgt wird. Die Prozeduren Backtracking-3 und Backtracking-4 sind lediglich Rücksetzoperationen, die einen früheren Verarbeitungszustand wiederherstellen (nach einer Reduktion bzw. nach einer Shift-Operation). Nach Backtracking-3 und Backtracking-4 muß deshalb mindestens ein weiteres Backtracking stattfinden. Die ersten drei Backtracking-Prozeduren haben die gemeinsame Vorbedingung, daß der letzte Verarbeitungsschritt eine Reduktion war. Die Backtrack-Prozeduren faßt die Prozedur $\mathrm{BACKTRACK_{BU}}$ zusammen:

PROZEDUR $\mathbf{BACKTRACK_{BU}}$

DATEN: Eine kontextfreie Syntax $G = <V_N, V_T, S, R>$ und ein Lexikon L.
EINGABE: Die drei Stacks STACK1, STACK2, STACK3.
AUSGABE: Die durch Anwendung der Prozeduren Backtracking-1/-2/-3/-4 modifizierten Stacks.

METHODE:

Wenn <Die zuletzt ausgeführte Aktion war eine *Reduce*-Aktion>

Dann

  Wenn <Es gibt eine andere Reduktionsmöglichkeit>

  Dann <RETURN(STACK1, *Backtracking-1*(STACK2, STACK3))>

  Sonst

    Wenn <Es kann ein weiteres Symbol geshiftet werden>

    Dann <RETURN(*Backtracking2*(STACK1, STACK2, STACK3))>

    Sonst <*Backtrack*$_{\mathrm{BU}}$(*Backtracking-3*(STACK1, STACK2, STACK3))>

Sonst

  Wenn <Die letzte Aktion war eine *Shift*-Aktion>

  Dann <*Backtrack*$_{\mathrm{BU}}$(*Backtracking-4*(STACK1, STACK2, STACK3))>.

  Sonst <RETURN([ ], [ ], [ ])>

Diese Prozedur erlaubt eine knappe und übersichtliche Beschreibung des Algorithmus RECOGNIZE$_{\mathrm{BU/DF}}$:

ALGORITHMUS RECOGNIZE$_{\mathrm{BU/DF}}$

bottom-up / links-rechts

depth-first / Backtracking

DATEN: Eine kontextfreie Syntax $G = <V_N, V_T, S, R>$ und ein Lexikon L.

EINGABE: Ein Satz $w = w_1 \ldots w_n$, mit $n \geq 1$.

AUSGABE: *True*/*False*.

ARBEITSSTRUKTUREN:

STACK1 - Anfangswert: $[w_1, \ldots, w_n]$.

STACK2 - Anfangswert: [ ].

STACK3 - Anfangswert: [ ].

METHODE:

Wenn <STACK1 = [ ]> und <STACK2 = [S]>

Dann <RETURN(*True*)>

Wenn <REDUCE$_{\mathrm{DF}}$ anwendbar ist>

Dann <*Recognize*$_{\mathrm{BU/DF}}$(STACK1, *REDUCE*$_{\mathrm{DF}}$(STACK2, STACK3))>

Sonst

  Wenn <SHIFT$_{\mathrm{DF}}$ anwendbar ist>

  Dann <*Recognize*$_{\mathrm{BU/DF}}$ (*SHIFT*$_{\mathrm{DF}}$(STACK1, STACK2, STACK3))>

  Sonst

    Wenn <STACK2 $\neq$ [ ]>

    Dann <*Recognize*$_{\mathrm{BU/DF}}$ (*Backtrack*$_{\mathrm{BU}}$(STACK1, STACK2, STACK3))>

    Sonst <RETURN(*False*)>

4.4 Vom Recognizer zum Parser

Es gibt zwei einfache Möglichkeiten, den Recognizer zu einem Parser zu erweitern. Die erste Möglichkeit besteht darin, nach einer erfolgreichen Ableitung die Struktur der Eingabekette anhand der Regelindizes in STACK3 in einem eigenen Arbeitsgang nachträglich zu rekonstruieren. Die zweite Möglichkeit besteht darin, die Struktur bereits parallel zur Ableitung aufzubauen. STACK2, der bisher lediglich atomare Symbole aus $\mathrm{V_N} \cup \mathrm{V_T}$ (bzw. dem Lexikon) enthält, müßte dann (Teil-)Bäume als Elemente haben und würde dann nicht mehr nur eine Ableitungszeile repräsentieren, sondern eine Liste von Strukturbeschreibungen für Teilstücke der Eingabekette. Die Prozedur $\mathrm{REDUCE_{DF}}$ würde dann nicht nur einen Stackabschnitt durch ein Symbol ersetzen, sondern mehrere Teilbäume an einen Knoten adjungieren. Dieses Verfahren liegt der Prolog-Implementierung nach [**?**] zugrunde, die wir im folgenden diskutieren werden.

4.5 Ein Shift-reduce-Parser in Prolog

Es ist typisch für viele Prolog-Programme, insbesondere jedoch für Parser-Implementierungen, daß prozedurale Elemente wie z.B. höhere Kontrollstrukturen, die durch den Aufruf von Subroutinen den Programmablauf steuern, nicht explizit durch entsprechende Anweisungen im Programmkode definiert werden. Ein Prolog-Programm ist immer auf das Ableitungsverhalten des Interpreters zu beziehen und ergibt nur vor diesem Hintergrund eine Spezifikation für den Algorithmus.

Für die Implementierung eines Shift-reduce-Parsers ist dabei entscheidend, daß ein Prolog-Interpreter intern mit einer Depth-first-Suchstrategie mit Backtracking arbeitet und deshalb bereits über die entsprechenden Kontrollstrukturen und eine Buchführung für die Verarbeitungsgeschichte verfügt. Die explizite Definition einer Backtracking-Komponente mit einer Datenstruktur für die Buchführung (wie STACK3) ist deshalb in Prolog nicht erforderlich.

Für die Implementierung in Prolog müssen also vor allem die folgenden Fragen geklärt werden:

- Wie werden die Arbeitstrukturen STACK1 und STACK2 in Prolog repräsentiert?
- Wie sieht ein geeignetes Regelformat aus?
- Wie werden die Vor- und Terminierungsbedingungen definiert?

4.5.1 Die Repräsentation der Stacks und Regeln

Beide Stacks lassen sich einfach als Prolog-Listen repräsentieren:

- STACK1: Dieser Stack wird als Liste von Prolog-Atomen repräsentiert. Da der Parser möglicherweise auch eine Eingabekette akzeptiert, die aus nicht-terminalen Symbolen besteht oder die neben terminalen Symbolen auch nicht-terminale Symbole enthält, wäre es für eine korrekte Implementierung des Algorithmus erforderlich, zunächst zu prüfen, ob die Eingabekette ausschließlich aus nicht-terminalen Symbolen besteht. Da aber die Möglichkeit, in der Eingabekette auch nicht-terminale Symbole zu verwenden, das Testen von Grammatiken erleichtern kann, wollen wir auf einen solchen Test verzichten[3].

- STACK2: Dieser Stack läßt sich als Liste repräsentieren, die zunächst leer ist und bei einem erfolgreichen Analyseverlauf eine Strukturbeschreibung in der üblichen Klammernotation, mit dem Startsymbol der Grammatik als Kopfelement, enthält.

Die Repräsentation der Regeln sollte so gewählt werden, daß die Realisierung der Shift-Operation und besonders der Reduce-Operation einen minimalen Aufwand erfordert; d.h., sie sollte es ermöglichen, die rechten Regelseiten direkt mit den bereits aufgebauten Teilstrukturen in STACK2 zu unifizieren. Dieses Ziel erreichen wir, indem

- die rechte Regelseite invertiert wird und
- bereits in der Regelrepräsentation Variablen bereitgestellt werden, die mit den Strukturen unifizieren, die von den entsprechenden Kategorien in STACK2 dominiert werden.

Aus diesen Gründen wird eine kontextfreie Regel $\mathrm{m} \rightarrow \mathrm{t}_1\ \mathrm{t}_2 \ldots \mathrm{t}_\mathrm{n}$ repräsentiert als:

regel([[t_n | TN], ..., [t_2 | T2], [t_1 | T1] | RESTSTACK],
    [[m, [t_1 | T1], [t_2 | T2], ..., [t_n | TN]] | RESTSTACK]).

Die klein geschriebenen Ausdrücke sind Metavariablen für Prolog-Atome, die die Kategorien der Regel repräsentieren; die groß geschriebenen Ausdrücke sind Metavariablen für Prolog-Variablen, die mit den Strukturen unifizieren, die von den Kategorien in STACK2 dominiert werden.

Lexikoneinträge (bzw. terminale Regeln) der Form *Kat* $\rightarrow$ *Wort* werden im folgenden Format spezifiziert:

regel([Wort|R], [[Kat, Wort]|R]).

[3] Es muß aber betont werden, daß der Parser dadurch auch Ketten akzeptiert, die nicht der von der Grammatik definierten Sprache angehören. Für eine entsprechende Korrektur des Programms kann das in Abschnitt 3.4.2 definierte Prolog-Prädikat *terminale_kette*/1 verwendet werden.

Beispiel (4-5)
Die kontextfreie Regel S → NP VP wird überführt in die Prolog-Klausel

```
regel([[vp | VP], [np | NP] | RESTSTACK],
      [[s, [np | NP], [vp | VP]] | RESTSTACK]).
```

Die terminale Regel *nomen* → *haus* wird überführt in die Prolog-Klausel:

```
regel([haus|R], [[nomen, haus]|R]).
```

4.5.2 Programmstruktur

Initialisierung. Nach dem Einlesen der Eingabekette ist der erste Schritt immer eine Shift-Operation, die das erste Wort von STACK1 in STACK2 überträgt.

```
% parse/1
% Hauptprädikat
parse([W1 | Ws]) :-
    parse(Ws, [W1], Struktur),
    write(Struktur).
```

Das Prädikat parse/1 wird mit einer Eingabekette [W1|Ws] aufgerufen und ruft das Prädikat parse/3 auf. Das erste Argument von parse/3 repräsentiert STACK1, das zweite Argument STACK2 und das dritte Argument ist eine Resultatsvariable, die die Strukturbeschreibung bei einem erfolgreichen Parse an das Mutterprädikat zurückgibt, wo es dann mit write/1 ausgegeben wird.

Der Shift/Reduce-Zyklus. Das Prinzip der Depth-first-Suche ist bei diesem Parser dadurch realisiert, daß grundsätzlich zunächst alle in einem Verarbeitungszustand möglichen Reduktionen vorgenommen werden, bevor ein neues Wort der Eingabekette geshiftet wird. Nach jeder Reduktion durch eine Regel wird zunächst überprüft, ob es eine weitere Regel gibt, die das Resultat der ersten Reduktion noch weiter reduziert. Hierfür bietet sich ein rekursives Prädikat der folgenden Form an:

```
reduce(Stack2_alt, Stack2_nach_allen_Reduktionen) :-
    regel(Stack2_alt, Stack2_nach_Reduktion),
    reduce(Stack2_nach_Reduktion, Stack2_nach_allen_Reduktionen).
```

Diese Struktur kann als Prädikatsdefinition so angegeben werden:

```
% reduce/2
reduce([X | Y], [X2 | Y2]) :-
    regel([X | Y], [X1 | Y1]),
    reduce([X1 | Y1], [X2 | Y2]).
```

Damit das Prädikat, nachdem alle möglichen Reduktionen ausgeführt wurden, nicht mit *fail* terminiert, benötigen wir noch eine „catch-all“-Klausel:

```
reduce(X, X).
```

Wir haben jetzt ein „Reduziere-solange-es-geht-Prädikat“ und ein Hauptprädikat definiert, das die geforderte Eingabe-/Ausgabespezifikation erfüllt und die Stacks korrekt initialisiert. Es fehlt noch ein Prädikat, das reduce/2 aufruft, anschließend eine Shift-Operation durchführt (sofern STACK1 noch nicht leer ist) und schließlich die Strukturbeschreibung an parse/1 zurückgibt, wenn die Ableitung erfolgreich ist, bzw. anderenfalls den Verarbeitungsprozeß durch rekursiven Selbstaufruf mit dem erzielten Zwischenresultat fortsetzt. Dieses Prädikat definieren wir wie folgt:

```
parse([W1 | Ws], Stack2, Struktur) :-          % STACK1 ist nicht leer
    reduce(Stack2, Stack2_neu),                % zunächst wird reduziert
    parse(Ws, [W1 | Stack2_neu], Struktur).    % dann wird W1 geshiftet und
                                               % parse rekursiv aufgerufen

parse([ ], Stack2, Struktur) :-                % STACK1 ist leer
    reduce(Stack2, Struktur).                  % durch die zweite Reduce-Klausel
                                               % wird STACK2 auf die Resultats-
                                               % variable kopiert
```

4.5.3 Das Programm im Überblick

```
parse([W1 | Ws]) :-
      parse(Ws, [W1], Struktur),
      write(Struktur).
parse([W1 | Ws], Stack2, Struktur) :-
      reduce(Stack2, Stack2_neu),
      parse(Ws, [W1 | Stack2_neu], Struktur).   % W1 wird geshiftet
parse([ ], Stack2, Struktur) :-
      reduce(Stack2, Struktur).
reduce([X | Y], [X2 | Y2]) :-
      regel([X | Y], [X1 | Y1]),
      reduce([X1 | Y1], [X2 | Y2]).
reduce(X, X).                                   % Terminierung
```

4.6 Ein breadth-first arbeitender Shift-reduce-Recognizer

Es ist relativ einfach, auf Grundlage des Algorithmus RECOGNIZE$_{BU/DF}$ einen Erkennungsalgorithmus zu formulieren, der ohne Backtracking arbeitet: Statt jedesmal, wenn es verschiedene Möglichkeiten gibt, die Analyse fortzusetzen, nur die erste direkt zu verfolgen und die für ein Backtracking erforderlichen Informationen in STACK3 zu speichern, werden jetzt alle Analysemöglichkeiten sofort weiterverfolgt (quasi-parallel). Dazu sind folgende Änderungen des Algorithmus notwendig:

1. $\text{BACKTRACK}_{\text{BU}}$ und der dritte Stack werden überflüssig, da es ohne Backtracking nicht mehr notwendig ist, die Indizes alternativer Regeln zu verwalten. Auch die Indizierung der Regeln und Lexikoneinträge ist nicht mehr erforderlich.

2. $\text{RECOGNIZE}_{\text{BU/DF}}$ wird so modifiziert, daß in jedem Arbeitszyklus alle möglichen Reduce-Aktionen und eventuell eine Shift-Aktion ausgeführt werden.

3. REDUCE nimmt als zweites Argument die Regel, mit der STACK2 reduziert wird.

PROZEDUR $\textbf{REDUCE}_{\text{BF}}$

DATEN: Die Menge R der Regeln der Grammatik G und ein Lexikon L.
EINGABE: $\text{STACK2} = [\text{O}_1, \text{O}_2, \ldots, \text{O}_\text{i}, \text{O}_{\text{i}+1}, \ldots, \text{O}_\text{k}]$ und eine Regel $\text{r} \in \text{R}$.
AUSGABE: $\text{STACK2} = [\text{K}, \text{O}_{\text{i}+1}, \ldots, \text{O}_\text{k}]$
VORBEDINGUNGEN:

(a) $\text{k} > 0$;

(b) $\text{r} = \text{K} \rightarrow \text{O}_\text{i}\ \text{O}_{\text{i}-1} \ldots \text{O}_1$, mit $\text{i} \geq 1$.

$\mathrm{RECOGNIZE}_{\mathrm{BU/BF}}$ verwendet die ursprüngliche Version der Shift-Prozedur:

ALGORITHMUS $\mathbf{RECOGNIZE}_{\mathrm{BU/BF}}$
bottom-up / links-rechts
breadth-first

DATEN: Eine kontextfreie Syntax G = <V_N, V_T, S, R> und ein Lexikon L.
EINGABE: STACK1 - Anfangswert: [$w_1, \ldots, w_n$], mit $n \geq 1$.
(die zu analysierende Eingabekette)
STACK2 - Anfangswert: [ ].
AUSGABE: *True*/*False* .
METHODE:
Wenn <STACK1 = [ ]> und <STACK2 = [S]>
Dann <RETURN(*True*)>

Wenn <$\mathrm{REDUCE}_{\mathrm{BF}}$ und SHIFT sind nicht anwendbar>
Dann <RETURN(*False*)>
Sonst
Für alle Regeln $r \in R$:
Wenn <STACK2 durch r reduzierbar ist>
Dann <$\underline{\mathit{Recognize}_{\mathrm{BU/BF}}}$(STACK1, $\underline{\mathit{Reduce}_{\mathrm{BF}}}$(STACK2, r)>
Wenn <STACK1 ≠ [ ]> und <First(STACK2) $\neq w_i$, $1 \leq i \leq n$>
Dann <$\underline{\mathit{Recognize}_{\mathrm{BU/BF}}}$($\underline{\mathit{Shift}}$(STACK1, STACK2))>

Beispiel (4-6)
Wenn G = <V_N, V_T, S, R> eine kontextfreie Syntax ist, die die Regeln

S → NP VP	NP → det N1	N1 → n
N1 → adj N1	VP → v	VP → v NP

enthält, und L ein Lexikon, mit den Einträgen:

(die (det)), (ein (det)), (alte (adj)), (frau (n)), (lied (n)), (singt (v)), ...;

dann werden bei der Analyse des Satzes „Die alte Frau singt ein Lied" folgende Konfigurationen durchlaufen[4]:

```
[a f s e l]  [d]
[a f s e l]  [DET]
[f s e l]    [a DET]
[f s e l]    [ADJ DET]
[s e l]      [f ADJ DET]
```

[4]Die Symbole der Eingabekette werden durch ihre Anfangsbuchstaben vertreten. Nicht alle Analysemöglichkeiten werden vollständig angegeben.

```
[s e l]  [N ADJ DET]
[s e l]  [N1 ADJ DET]
[e l]  [s N ADJ DET]
[s e l]  [N1 DET]
[e l]  [s N1 ADJ DET]
[s e l]  [NP]
[e l]  [s N1 DET]
[e l]    [s NP]
[e l]    [V NP]
[e l]  [VP NP]
[l]  [e V NP]
[l]  [DET V NP]
[ ]  [l DET V NP]
[ ]  [N DET V NP]
[ ]  [N1 DET V NP]
[ ]  [NP V NP]
[ ]  [VP NP]
[ ]  [S]
  true
[e l]  [S]
[l]    [e S]
[l]    [DET S]
[ ]    [l DET S]
[ ]    [N DET S]
[ ]    [N1 DET S]
[ ]    [NP S]
  false
[l]  [e VP NP]
[l]  [DET VP NP]
[ ]  [l DET VP NP]
[ ]  [N DET VP NP]
[ ]  [N1 DET VP NP]
[ ]  [NP VP NP]
  false
```

4.7 Ein Shift-reduce-Recognizer in Lisp

Wie bei der Prolog-Implementierung des Shift-reduce-Parsers mit Backtracking werden auch in diesem Fall die beiden Stacks durch Listen repräsentiert. Die Hauptfunktion RECOGNIZE_BU/BF wird mit einer Liste, die den zu analysierenden Ausdruck enthält, aufgerufen. Optional können noch der Wert für den zweiten Stack und der Typ der zu analysierenden Konstituente (*startsymbol*) spezifiziert werden; anderenfalls wird STACK2 mit NIL initialisiert, und es wird versucht, den Ausdruck als vollständigen Satz zu analysieren. Es wird vorausgesetzt, daß die zur Analyse erforderlichen Daten (Syntax & Lexikon) den beiden globalen Variablen *syntax* und *lexikon* zugewiesen sind.

;;; RECOGNIZE_BU/BF
;;; Ein einfacher Bottom-up-Recognizer ohne Backtracking: Zuerst wird die Menge
;;; aller von der gegebenen Konfiguration aus erreichbaren Konfigurationen[5] berech-
;;; net (KONFIGURATIONEN). Wenn keine der beiden Abbruchbedingungen erfüllt
;;; ist, wird RECOGNIZE_BU/BF rekursiv mit allen erreichbaren Konfigurationen
;;; aufgerufen.

```
(defun recognize_bu/bf (stack1 &optional (stack2 nil) (symbol `(,(startsymbol))))
  (let ((konfigurationen (berechne-konfigurationen stack1 stack2)))
    (cond ((and (endp stack1) (equal stack2 symbol))
           (print t))                              ; Satz akzeptiert
          ((endp konfigurationen) (print nil))     ; Satz nicht akzeptiert
          (t (dolist (x konfigurationen 'stop)     ; Analyse fortsetzen
               (apply #'recognize_bu/bf (append x (list symbol))))))))

;;; BERECHNE-KONFIGURATIONEN
;;; Die Funktion berechnet die Menge aller von der aktuellen Konfiguration aus er-
;;; reichbaren Konfigurationen.
(defun berechne-konfigurationen (stack1 stack2)
  (append (mapcar #'(lambda (x) (reduce stack1 stack2 x))     ; reduziere mit jeder
                  (matching-regeln stack2))                   ; anwendbaren Regel
          (when (shift-p stack1 stack2)                       ; wenn möglich:
            (list (shift stack1 stack2)))))                   ; Shift-Aktion ausführen
```

Die Funktionen SHIFT und REDUCE liefern als Wert die Liste mit den beiden durch Shift- bzw. Reduce-Aktionen modifizierten Stacks STACK1 und STACK2:

```
;;; SHIFT
(defun shift (stack1 stack2)
  (list (rest stack1) (cons (first stack1) stack2)))

;;; REDUCE
(defun reduce (stack1 stack2 regel)
  (list stack1
        (cons (first regel) (nthcdr (length (rest regel)) stack2))))

;;; SHIFT-P
;;; Das Prädikat prüft, ob SHIFT angewendet werden kann oder nicht: Die Tren-
;;; nung von Syntax und Lexikon läßt die Anwendung von Shift nur dann sinnvoll
;;; erscheinen, wenn das erste Symbol von STACK2 kein Symbol der Eingabekette
;;; ist; d.h. nach jedem Shift wird zunächst reduziert.
(defun shift-p (stack1 stack2)
  (and stack1
       (or (endp stack2) (lexkat-p (first stack2)) (synkat-p (first stack2)))))
```

[5] Als „Konfiguration“ bezeichnen wir in diesem Kontext ein geordnetes Paar <Stack1, Stack2>, das den aktuellen Verarbeitungsstand repräsentiert.

```
;;; MATCHING-REGELN
;;; Die Funktion berechnet die Menge der für Reduce-Aktionen verwendbaren Regeln:
;;; Wenn das erste Symbol in STACK2 eine syntaktische oder lexikalische Kategorie
;;; ist, findet sie alle Regeln der Syntax, die anwendbar sind. Ist das erste Symbol ein
;;; Symbol der Eingabekette, werden auf Grundlage des Lexikoneintrags für dieses
;;; Symbol geeignete lexikalische Regeln formuliert.
(defun matching-regeln (stack2)
  (if (or (lexkat-p (first stack2)) (synkat-p (first stack2)))      ; suche Regeln
     (remove-if #'(lambda (x) (not (match x stack2))) (regeln))
     (let ((eintrag (lexikon-eintrag (first stack2))))              ; suche Kategorien
       (mapcar #'(lambda (x) (append x (list (first eintrag))))
               (rest eintrag)))))

;;; MATCH
;;; Das Prädikat testet, ob REGEL verwendet werden kann, um STACK zu reduzie-
;;; ren; d.h. ob die invertierte rechte Seite von REGEL mit den ersten Symbolen von
;;; STACK übereinstimmt.
(defun match (regel stack)
  (let ((vergleiche (length (rest regel))))
    (dotimes (x vergleiche t)
      (unless (eq (nth x (rest regel)) (nth (- vergleiche x 1) stack))
        (return nil)))))
```

Aufgaben

4.1 Erweitern Sie den Erkennungsalgorithmus RECOGNIZE$_{BU/BF}$ zu einem Parsingalgorithmus und implementieren Sie ihn in Lisp/Prolog.

4.2 Definieren Sie ein Prädikat, das überprüft, ob eine Syntax zyklenfrei ist und keine Tilgungsregeln enthält.

4.3 Entwickeln und implementieren Sie ein Verfahren, das eine begrenzte Verarbeitung von Tilgungsregeln mit den in diesem Kapitel vorgestellten Algorithmen erlaubt.

4.4 Schreiben Sie ein Prologprogramm, das eine kontextfreie Grammatik mit Regeln der Form $m \longrightarrow [t_1, t_2, \ldots, t_n]$. in die für Shift-reduce-Parsing in Prolog geeignete Form

regel([[t_n|TN], ..., [t_2|T2], [t_1|T1]|RESTSTACK],
[[m, [t_1|T1], ..., [t_n|TN]]|RESTSTACK]).

überführt und Lexikoneinträege der Form *lex(wort,cat).* in das Format

regel([word|W],[[cat,wort]|W]).

4.5 Schreiben Sie ein Programm, das eine Syntax G (mit $e \notin$ L(G)), die Tilgungsregeln enthält, in eine schwach äquivalente Syntax ohne Tilgungsregeln übersetzt.

4.6 Entwickeln Sie einen Bottom-up-Parser, der Sätze von rechts nach links verarbeitet.

Kapitel 5

Left-corner-Parsing

In diesem Kapitel beschreiben wir die Left-corner-Parsingstrategie. Nach Einführung der Begriffe „linke Ecke“ (einer Regel) und „Left-corner-Parse“ entwickeln wir einen nicht-deterministischen Left-corner-Erkennungsalgorithmus, den wir dann schrittweise zu einem Parsingalgorithmus (mit Look-ahead) erweitern. Das Kapitel endet mit der Beschreibung des in Prolog implementierten BUP-Systems von Matsumoto.

5.1 Grundzüge

Left-corner-Parsingalgorithmen verbinden Aspekte des Bottom-up-Parsing mit denen des Top-down-Parsing: Für reines Bottom-up-Parsing ist es charakteristisch, daß eine Regel $k_0 \rightarrow k_1 \dots k_n$ nur dann angewendet werden kann, wenn für jede Kategorie k_i ($1 \leq i \leq n$) bereits eine vollständige Teilstruktur gefunden wurde. Für Left-corner-Parsing dagegen ist es nur erforderlich, daß bereits eine von k_1 dominierte Struktur erkannt wurde[1] - die Regel wird dann verwendet, um sowohl Annahmen über die k_1 dominierende Kategorie als auch über die nachfolgenden Konstituenten zu formulieren. Insofern läßt sich sagen, daß die linke Ecke einer Regel bottom-up erkannt, der Rest der Regel aber *top-down* verarbeitet wird.

Wenn $G = <V_N, V_T, S, R>$ eine kontextfreie Syntax ist, dann ist ein Left-corner-Parse eines Satzes $w \in L(G)$ eine Folge von Regelindizes $\gamma = i_1, \dots, i_n$, die einer Ableitung von w in G korrespondieren und für die gilt:

1. β sei der durch γ implizit festgelegte Strukturbaum.

2. Für die Knoten in β gilt folgende Ordnungsvorschrift:

    (a) Wenn ein Knoten n die Knoten $n_1, \dots, n_m$ direkt dominiert, dann liegen alle Knoten des Teilbaums mit der Wurzel n_1 vor n;

[1] k_1 wird als *linke Ecke* („left corner“) der Regel bezeichnet.

(b) n liegt vor allen anderen von ihm dominierten Knoten und

(c) alle von n_i dominierten Knoten liegen vor den durch n_{i+1} dominierten Knoten $(1 < i < r)$.

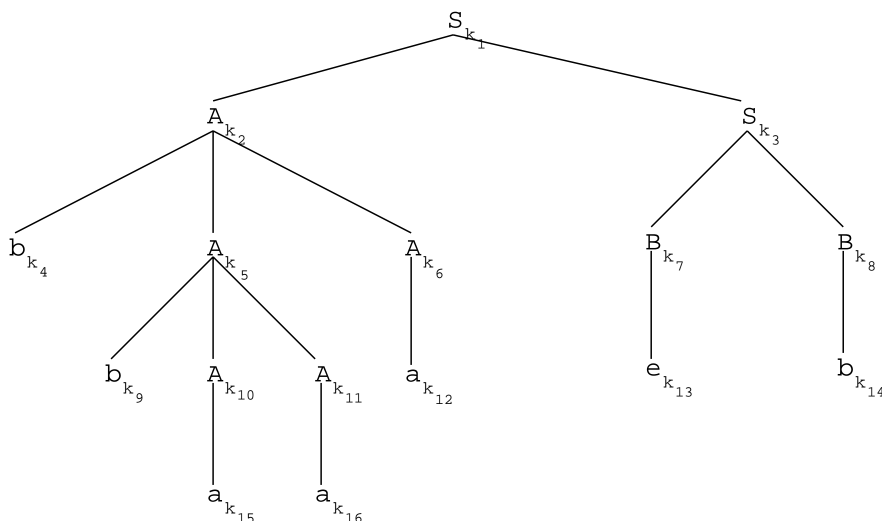
3. Die durch γ beschriebene Reihenfolge der Regelanwendung verletzt diese Ordnungsvorschrift nicht.

Beispiel (5-1)

Wenn G = <V_N, V_T, S, R> eine kontextfreie Syntax ist, die die folgenden Regeln enthält:

1: S → AS	2: S → BB,	3: A → bAA
4: A → a	5: B → b	6: B → e

dann erhalten wir für den Satz „bbaaab“ die Struktur:



Reihenfolge der Knoten:

k_4 k_2 k_9 k_5 k_{15} k_{10} k_{16} k_{11} k_{12} k_6 k_1 k_{13} k_7 k_3 k_{14} k_8

Linksparse	: 1 3 3 4 4 4 2 6 5	(Top-down-Parsing)
reverser Rechtsparse[2]	: 4 4 3 4 3 6 5 2 1	(Bottom-up-Parsing)
Left-corner-Parse	: 3 3 4 4 4 1 6 2 5	

Wir orientieren uns im folgenden an dem in [**?**] beschriebenen Algorithmus, der auf dem NBT-Algorithmus (**N**_onselective_ **B**_ottom to_ **T**_op_) von [**?**] basiert. Anders als Ross spezifizieren wir die erforderlichen Operationen nicht durch Turing-Maschinenanweisungen, sondern durch drei Prozeduren $\text{REDUCE}_{\text{LC}}$, MOVE_{LC} und $\text{REMOVE}_{\text{LC}}$, die auf Stacks operieren.

5.2 Ein Left-corner-Erkennungsalgorithmus

Wie beim Shift-reduce-Parsing können auch hier die benötigten Arbeitsstrukturen als Stacks repräsentiert werden. Für den Recognizer werden drei Stacks (STACK1, STACK2 und STACK3) benötigt:

[2]Eine Linksableitung (Rechtsableitung) für einen Satz ist eine Ableitung, bei der immer das am weitesten links (rechts) vorkommende nicht-terminale Symbool ersetzt wird. Ein Linksparse (Rechtsparse) ist eine Folge von Regelindizes, die einer Linksableitung (Rechtsableitung) korrespondiert.

STACK1 enthält den noch nicht verarbeiteten Teil des zu analysierenden Satzes, STACK2 die noch top-down zu erkennenden Kategorien ($k_2 \ldots k_n$) und STACK3 die gesuchten Konstituenten ($k_0 \ldots$)[3].

PROZEDUR REDUCE$_{LC}$

DATEN: Eine kontextfreie Syntax G = <V_N, V_T, S, R> und Lexikon L.
EINGABE: STACK1, mit *First*(STACK1) = k_1
STACK2
STACK3
AUSGABE: *Pop*(STACK1)
Push($k_2 \ldots k_n$ t, STACK2)
Push(k_0, STACK3)

VORBEDINGUNGEN:

1. Es gibt eine Regel $k_0 \rightarrow k_1 \ldots k_n \in R$ oder $k_1 \in k_0$, für eine beliebige lexikalische Kategorie k_0.
2. *First*(STACK2) $\in (V_N \cup V_T)$.

Durch Anwendung der Regel $k_0 \rightarrow k_1 \ldots k_n$ wird das erste Symbol aus STACK1, das mit der linken Ecke der Regel identisch ist, getilgt. Die übrigen Symbole der rechten Regelseite werden zusammen mit dem Symbol 't', das weder in V_N noch in V_T enthalten ist, auf den STACK2 geschoben; k_0 auf den STACK3. Das Symbol 't' markiert gewissermaßen das *Ende* der Regel und trennt die noch zu erkennenden Symbole dieser Regel von denen anderer Regeln.

PROZEDUR MOVE$_{LC}$

EINGABE: STACK1
STACK2
STACK3
AUSGABE: *Push*(*First*(STACK3), STACK1)
Pop(STACK2)
Pop(STACK3)

VORBEDINGUNGEN:

1. *First*(STACK2) = t.
2. *First*(STACK3) = A, mit $A \in (V_N \cup V_T)$.

[3] Zur Formulierung der drei Prozeduren verwenden wir die Operationen *Push(Objekt, Stack)* und *Pop(Stack)*: **Push** schiebt *Objekt* auf den Stack *Stack*; **Pop** entfernt das erste Objekt aus *Stack* (vgl. Appendix *Datenstrukturen*)

Da 't' das erste Symbol in STACK2 ist, wurde die rechte Seite einer Regel, deren linke Seite das Symbol A bildet, komplett abgearbeitet; d.h. es wurde eine Konstituente vom Typ A erkannt. A wird auf den STACK1 geschoben und aus STACK3 gelöscht, und das erste Symbol aus STACK2 wird ebenfalls entfernt.

PROZEDUR REMOVE$_{\text{LC}}$

EINGABE: STACK1
STACK2
STACK3
AUSGABE: *Pop*(STACK1)
Pop(STACK2)
STACK3
VORBEDINGUNG:
First(STACK1) = *First*(STACK2)

Das erste Symbol in STACK1 ist mit dem ersten Symbol in STACK2 identisch, und die Symbole werden aus STACK1 und STACK2 entfernt. Die Prozedur REMOVE$_{\text{LC}}$ ist genau dann anwendbar, wenn *First*(STACK2) eine Kategorie k_i ist, die die linke Ecke einer Regel bildet, und k_i (top-down) erkannt wurde.

Die drei Prozeduren erlauben eine einfache Formulierung einer nicht-deterministischen Erkennungsprozedur RECOGNIZE$_{\text{LC}}$, deren Arbeitsweise Beispiel (5-2) illustriert.

ALGORITHMUS RECOGNIZE$_{\text{LC}}$
nicht-deterministisch
left-corner

DATEN: Eine kontextfreie Syntax G = <V_N, V_T, S, R>, ein Lexikon L und ein Satz $w = w_1 \ldots w_n$, mit $n \geq 1$.
EINGABE: STACK1 = [$w_1 \ldots w_n$]
STACK2 = [S]
STACK3 = [ ]
AUSGABE: *True*/*False*.

METHODE:
Wiederhole:
Wenn <STACK1 = STACK2 = STACK3 = [ ]>
Dann <RETURN(*True*)>
Sonst
Wenn <Es gibt eine Prozedur P ∈ {REDUCE$_{\text{LC}}$, MOVE$_{\text{LC}}$, REMOVE$_{\text{LC}}$}, deren Vorbedingungen erfüllt sind>
Dann <*RECOGNIZE*$_{\text{LC}}$(*P*(STACK1, STACK2, STACK3))>
Sonst <RETURN(*False*)>

Beispiel (5-2)
Wenn G = <V_N, V_T, S, R> eine kontextfreie Syntax ist, die die Regeln

S → NP VP NP → det n VP → v NP

enthält, und L ein Lexikon mit den Einträgen (der (det)) (meister (n)) ...etc., dann erhalten wir für den Satz „Der Meister sucht einen Fehler“ folgenden Trace:

STACK1	STACK2	STACK3	Prozedur
[*der Meister sucht ...*]	[S]	[ ]	$REDUCE_{LC}$
[*Meister sucht ...*]	[t S]	[det]	$MOVE_{LC}$
[det *Meister sucht ...*]	[S]	[ ]	$REDUCE_{LC}$
[*Meister sucht ...*]	[n t S]	[NP]	$REDUCE_{LC}$
[*sucht einen Fehler*]	[t n t S]	[n NP]	$MOVE_{LC}$
[n *sucht einen Fehler*]	[n t S]	[NP]	$REMOVE_{LC}$
[*sucht einen Fehler*]	[t S]	[NP]	$MOVE_{LC}$
[NP *sucht einen Fehler*]	[S]	[ ]	$REDUCE_{LC}$
[*sucht einen Fehler*]	[VP t S]	[S]	$REDUCE_{LC}$
[*einen Fehler*]	[t VP t S]	[v S]	$MOVE_{LC}$
[v *einen Fehler*]	[VP t S]	[S]	$REDUCE_{LC}$
[*einen Fehler*]	[NP t VP t S]	[VP S]	$REDUCE_{LC}$
[*Fehler*]	[t NP t VP t S]	[det VP S]	$MOVE_{LC}$
[det *Fehler*]	[NP t VP t S]	[VP S]	$REDUCE_{LC}$
[*Fehler*]	[n t NP t VP t S]	[NP VP S]	$REDUCE_{LC}$
[ ]	[t n t NP t VP t S]	[n NP VP S]	$MOVE_{LC}$
[n]	[n t NP t VP t S]	[NP VP S]	$REMOVE_{LC}$
[ ]	[t NP t VP t S]	[NP VP S]	$MOVE_{LC}$
[NP]	[NP t VP t S]	[VP S]	$REMOVE_{LC}$
[ ]	[t VP t S]	[VP S]	$MOVE_{LC}$
[VP]	[VP t S]	[S]	$REMOVE_{LC}$
[ ]	[t S]	[S]	$MOVE_{LC}$
[S]	[S]	[ ]	$REMOVE_{LC}$
[ ]	[ ]	[ ]	*True*

Die Prozedur $RECOGNIZE_{LC}$ ist aus zwei Gründen nicht-deterministisch:

1. Es kann mehrere Regeln geben, deren linke Ecke mit dem ersten Symbol des STACK1 übereinstimmt, d.h., es kann verschiedene Möglichkeiten geben, $REDUCE_{LC}$ anzuwenden.

2. Es kann Konfigurationen geben, in denen sowohl die Vorbedingungen für $REDUCE_{LC}$ als auch die für $REMOVE_{LC}$ erfüllt sind. Dieser Fall tritt immer dann ein, wenn eine Struktur gebildet wurde, die entweder verwendet werden

kann, um die gerade bearbeitete, noch unvollständige Struktur zu ergänzen ($\text{REMOVE}_{\text{LC}}$), die aber auch den Anfang einer neuen Teilstruktur bilden kann ($\text{REDUCE}_{\text{LC}}$); vgl. Abbildung (5-1).

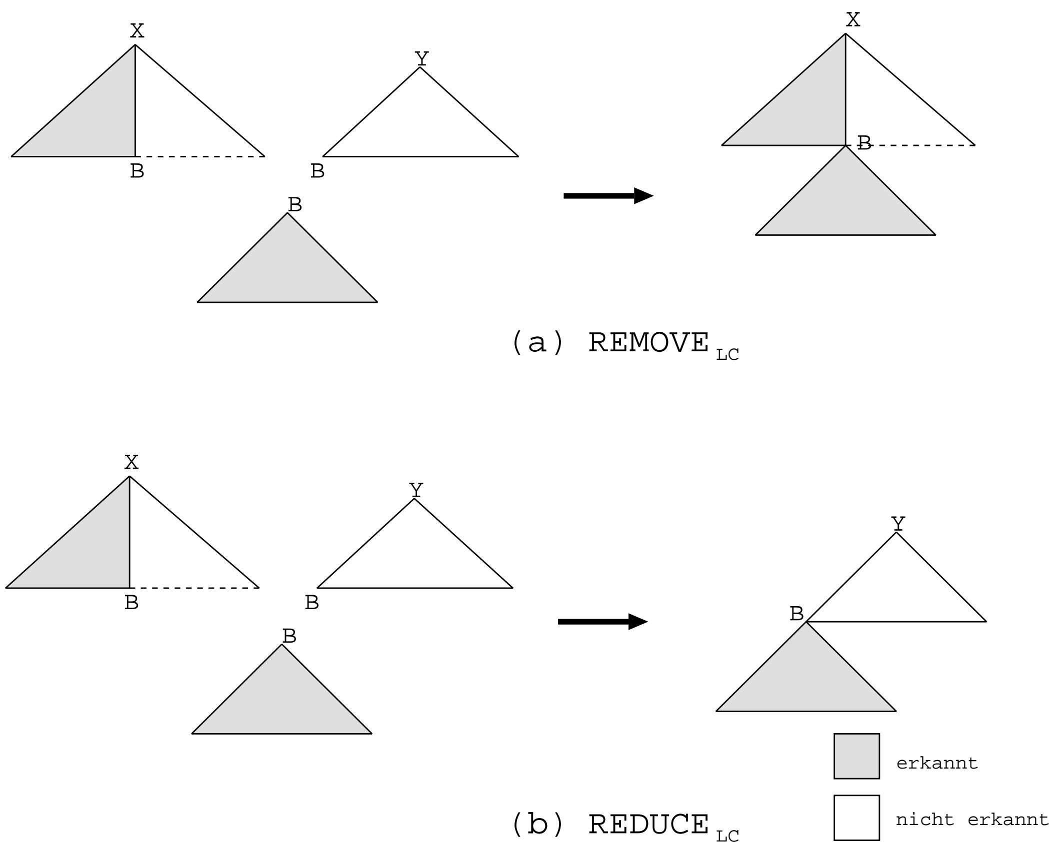


Abbildung (5-1)

Zwei einfache und uns schon vertraute Möglichkeiten, diesem Problem zu begegnen, bildet die Verwendung der Depth-first-Suche mit Backtracking bzw. der quasi-parallelen Breadth-first-Suche. Da das BUP-System, das wir in 5.6 beschreiben werden, die zweite dieser beiden Möglichkeiten nutzt, beschränken wir uns hier auf die Formulierung eines breadth-first arbeitenden Erkennungsalgorithmus: Der Algorithmus $\text{RECOGNIZE}_{\text{LC/BF}}$ berechnet alle Konfigurationen[4], die aus der aktuellen Konfiguration erreichbar sind. Jede dieser neuen Konfigurationen repräsentiert eine Analysemöglichkeit für den Satz, die anschließend weiterverfolgt wird.

[4]Unter einer *Konfiguration* $k = (s_1, s_2, s_3)$ verstehen wir in diesem Zusammenhang ein Tripel, das aus unseren drei Stacks STACK1, STACK2 und STACK3 gebildet wird.

ALGORITHMUS RECOGNIZE$_{LC/BF}$

DATEN: Eine kontextfreie Syntax G = <V_N, V_T, S, R>, ein Lexikon L und ein Satz $w = w_1 \ldots w_n$, mit $n \geq 1$.
EINGABE: STACK1 = [$w_1, \ldots, w_n$]
STACK2 = [S]
STACK3 = [ ]
AUSGABE: *True*/*False*.
ARBEITSSTRUKTUREN:
KONFIGURATIONEN - Eine Menge von Konfigurationen.
Anfangswert: $\emptyset$.
METHODE:
Wenn <STACK1 = STACK2 = STACK3 = [ ]>
Dann <RETURN(*True*)>
Sonst
KONFIGURATIONEN $\Leftarrow$ {die Menge aller Konfigurationen, die sich durch REMOVE$_{LC}$, REDUCE$_{LC}$ oder MOVE$_{LC}$ aus der aktuellen Konfiguration ableiten lassen}
Wenn <KONFIGURATIONEN = $\emptyset$>
Dann <RETURN(*False*)>
Sonst <Für alle <$s1_i$, $s2_i$, $s3_i$> $\in$ KONFIGURATIONEN:
<*RECOGNIZE*$_{LC/BF}$($s1_i$, $s2_i$, $s3_i$)>>

5.3 Ein Left-corner-Parsingalgorithmus

Um den Erkennungsalgorithmus RECOGNIZE$_{LC/BF}$ in einen Parsingalgorithmus PARSE$_{LC}$ zu überführen, führen wir einen weiteren Stack ein (STACK4), der zum Aufbau der Strukturbeschreibung verwendet wird. Dieser Stack ist zunächst leer. Wenn der Satz zu der durch die Syntax festgelegten Sprache gehört, wird nicht TRUE, sondern die in STACK4 aufgebaute Strukturbeschreibung ausgegeben. Die drei Prozeduren müssen modifiziert werden, um den korrekten Aufbau der Strukturbeschreibung sicherzustellen.

Die Modifikation der Prozedur MOVE$_{LC}$ beschränkt sich darauf, für den vierten Stack einen weiteren Parameter einzufügen. Strukturbildende Operationen sind anders als bei den anderen beiden Prozeduren nicht erforderlich.

PROZEDUR REDUCE$_{\mathrm{LC/BF}}$

DATEN: Eine kontextfreie Syntax $G = \langle V_N, V_T, S, R \rangle$ und ein Lexikon L.
EINGABE: STACK1, mit *First*(STACK1) = k_1
STACK2
STACK3
STACK4
AUSGABE: *Pop*(STACK1)
Push($k_2 \ldots k_n$ t, STACK2)
Push(k_0, STACK3)
STRUKTUR1(k_0, k_1, STACK4)
VORBEDINGUNGEN:

1. Es gibt eine Regel $k_0 \rightarrow k_1 \ldots k_n \in R$ oder
   $k_1 \in k_0$, für eine beliebige lexikalische Kategorie k_0.
2. *First*(STACK2) $\in (V_N \cup V_T)$.

Die für REDUCE$_{\mathrm{LC/BF}}$ notwendigen strukturbildenden Operationen faßt die Prozedur STRUKTUR1 zusammen:

PROZEDUR STRUKTUR1

EINGABE: Ein Stack STACK und zwei Symbole $k_0, k_1 \in (V_N \cup V_T)$.
AUSGABE: Der modifizierte Stack STACK'.
METHODE:
Wenn <STACK = [ ] $\vee$ *First*(STACK) = (k' α), mit k' $\neq k_1$ >
Dann <RETURN(*Push*((k_0 k_1), STACK)>
Sonst <RETURN(*Push*((k_0 *First*(STACK)), *Pop*(STACK)))>

Es ist zu unterscheiden, ob bereits eine von k_1 dominierte Strukturbeschreibung existiert oder nicht: Ist das nicht der Fall, wird mit dem Aufbau einer neuen Strukturbeschreibung begonnen; anderenfalls wird diese Strukturbeschreibung in die aufzubauende Strukturbeschreibung integriert.

Bei der dritten Prozedur sind die Fälle zu unterscheiden, in denen es noch nicht erkannte Kategorien in STACK3 gibt bzw. nicht gibt: Wenn STACK3 leer ist, sind auch in diesem Fall keine strukturbildenden Aktionen notwendig; anderenfalls werden die zuletzt gebildeten beiden partiellen Strukturbeschreibungen miteinander verknüpft.

PROZEDUR REMOVE$_{\mathrm{LC/BF}}$

EINGABE: STACK1
STACK2
STACK3
STACK4

AUSGABE: *Pop*(STACK1)
Pop(STACK2)
STACK3
STRUKTUR2(STACK3, STACK4)

VORBEDINGUNG:
First(STACK1) = *First*(STACK2)

Die Strukturbeschreibung wird durch die Prozedur STRUKTUR2 aufgebaut.

PROZEDUR STRUKTUR2

EINGABE: Die beiden Stacks STACK3, STACK4.
AUSGABE: Der modifizierte Stack STACK4'.

METHODE:
Wenn <STACK3 = [ ]>
Dann <RETURN(STACK4)>
Sonst <RETURN(*Push*((*Second*(STACK4) + *First*(STACK4)), *Pop*(*Pop*(STACK4))))>

Beispiel (5-3)
Wenn G = <V_N, V_T, S, R> eine kontextfreie Syntax ist, die die Regeln

S → NP VP	NP → n	VP → v NP PP
VP → v	PP → p NP	

enthält, und L ein Lexikon mit geeigneten Einträgen ist, dann ergibt sich für den Satz „Eva sah Adam am Morgen“ folgender Trace[5]:

STACK1	STACK2	STACK3	STACK4
[1 2 3 4 5]	[S]	[ ]	[ ]
[2 3 4 5]	[t S]	[N]	[(n 1)]
[n 2 3 4 5]	[S]	[ ]	[(n 1)]
[2 3 4 5]	[t S]	[NP]	[(NP (n 1))]
[NP 2 3 4 5]	[S]	[ ]	[(NP (n 1))]
[2 3 4 5]	[VP t S]	[S]	[(S (NP (n 1)))]
[3 4 5]	[t VP t S]	[v S]	[(v 2) (S (NP (n 1)))]
[v 3 4 5]	[VP t S]	[S]	[(v 2) (S (NP (n 1)))]
[3 4 5]	[NP PP t VP t S]	[VP S]	[(VP (v 2)) (S (NP ...))]
[4 5]	[t NP PP t VP t S]	[n VP S]	[(n 3) (VP (v 2)) (S (NP ...))]
[n 4 5]	[NP PP t VP t S]	[VP S]	[(n 3) (VP (v 2)) (S (NP ...))]
[4 5]	[t NP PP t VP t S]	[NP VP S]	[(NP (n 3)) (VP (v 2)) (S ...)]
[NP 4 5]	[NP PP t VP t S]	[VP S]	[(NP (n 3)) (VP (v 2)) (S ...)]
[4 5]	[VP t NP PP t VP t S]	[S VP S]	[(S (NP (n 3))) (VP (v 2)) ...]
[5]	[t VP t NP PP t VP t S]	[p S VP S]	[(p 4) (S (NP (n 3))) (VP ...)]
[p 5]	[VP t NP PP t VP t S]	[S VP S]	[(p 4) (S (NP (n 3))) (VP ...)]
[5]	[NP t VP t NP PP t ...]	[PP S VP S]	[(PP (p 4)) (S (NP (n 3))) ...]
[ ]	[t NP t VP t NP PP t ...]	[n PP S VP S]	[(n 5) (PP (p 4)) (S (NP ...))]
[n]	[NP t VP t NP PP t ...]	[PP S VP S]	[(n 5) (PP (p 4)) (S (NP ...))]
[ ]	[t NP t VP t NP PP t ...]	[NP PP S VP S]	[(NP (n 5)) (PP (p 4)) (S ...)]
[NP]	[NP t VP t NP PP t ...]	[PP S VP S]	[(NP (n 5)) (PP (p 4)) (S ...)]
[ ]	[VP t NP t VP t ...]	[S PP S VP S]	[(S (NP (n 5))) (PP (p 4)) ...]
[ ]	[t VP t NP PP t VP t S]	[PP S VP S]	[(PP (p 4) (NP (n 5))) (S ...)]
[PP]	[VP t NP PP t VP t S]	[S VP S]	[(PP (p 4) (NP (n 5))) (S ...)]
[4 5]	[PP t VP t S]	[VP S]	[(VP (v 2) (NP (n 3))) (S ...)]
[5]	[t PP t VP t S]	[p VP S]	[(p 4) (VP (v 2) (NP (n 3))) ...]
[p 5]	[PP t VP t S]	[VP S]	[(p 4) (VP (v 2) (NP (n 3))) ...]
[5]	[NP t PP t VP t S]	[PP VP S]	[(PP (p 4)) (VP (v 2) (NP ...))]
[ ]	[t NP t PP t VP t S]	[n PP VP S]	[(n 5) (PP (p 4)) (VP (v 2) ...)]
[n]	[NP t PP t VP t S]	[PP VP S]	[(n 5) (PP (p 4)) (VP (v 2) ...)]
[ ]	[t NP t PP t VP t S]	[NP PP VP S]	[(NP (n 5)) (PP (p 4)) (VP ...)]
[NP]	[NP t PP t VP t S]	[PP VP S]	[(NP (n 5)) (PP (p 4)) (VP ...)]
[ ]	[VP t NP t PP t VP t S]	[S PP VP S]	[(S (NP (n 5))) (PP (p 4)) ...]
[ ]	[t PP t VP t S]	[PP VP S]	[(PP (p 4) (NP (n 5))) (VP ...)]
[PP]	[PP t VP t S]	[VP S]	[(PP (p 4) (NP (n 5))) (VP ...)]
[ ]	[t VP t S]	[VP S]	[(VP (v 2) (NP (n 3)) ...) ...]
[VP]	[VP t S]	[S]	[(VP (v 2) (NP (n 3)) ...) ...]
[ ]	[t S]	[S]	[(S (NP (n 1)) (VP (v 2) (NP (n 3)) (PP (p 4) ...)))]
[S]	[S]	[ ]	[(S (NP (n 1)) (VP (v 2) (NP (n 3)) (PP (p 4) ...)))]
[ ]	[ ]	[ ]	[(S (NP (n 1)) (VP (v 2) (NP (n 3)) (PP (p 4) ...)))]
⋮	⋮	⋮	⋮

(S (NP (n 1)) (VP (v 2) (NP (n 3)) (PP (p 4) (NP (n 5)))))

[5]Im folgenden Trace repräsentieren wir die Worte des Satzes durch Zahlen.

5.4 Left-corner-Parsing mit Look-ahead

Da der von uns beschriebene Left-corner-Parsingalgorithmus partiell top-down arbeitet, läßt sich wie bei den Algorithmen aus Kapitel 3 ein deutlicher Effizienzgewinn erzielen, indem die Zahl der für eine Fortsetzung der Ableitung in Frage kommenden Regeln durch einen Look-ahead reduziert wird:

Für jedes nicht-terminale Symbol der Syntax wird die Menge aller Symbole berechnet, die als linke Ecken in von ihm aus *erreichbar*en Konstituenten vorkommen. Diese Relation zwischen einer linken Ecke und einer sie mittelbar dominierenden Kategorie wird als „*LINK*-Relation" bezeichnet. Die Relation ist reflexiv und transitiv.

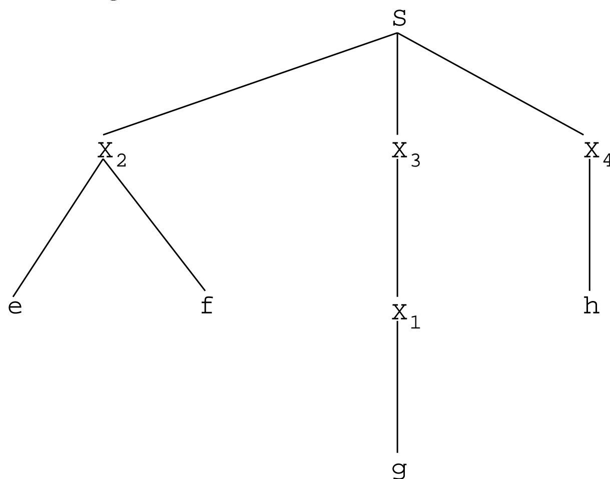
Definition (5-1) *Die LINK-Relation*

Wenn $G = <V_N, V_T, S, R>$ eine beliebige kontextfreie Syntax ist, dann ist *LINK(G)* definiert als die Menge aller geordneten Paare $<X, Y>$, mit $X \in V_N$ und $Y \in V$, die eine der folgenden drei Bedingungen erfüllen:

1. $X = Y$; (Reflexivität)
2. es gibt eine Regel $X \rightarrow Y\alpha \in R$ oder
3. $<X, X'> \in LINK(G)$ und $<X', Y> \in LINK(G)$, für ein beliebiges $X' \in V_N$. (Transitivität)

Beispiel (5-4)

Wenn $G = < \{S, X_1, X_2, X_3, X_4\}, \{e, f, g, h\}, S, R>$, mit $R = \{S \rightarrow X_2\ X_3\ X_4, X_2 \rightarrow e\ f, X_3, \rightarrow X_1, X_1 \rightarrow g, X_4 \rightarrow h\}$, dann sind im folgenden Baum e, X_2 und S selbst linke Ecken von S, g ist eine linke Ecke von X_1 ... etc.:



Für die Grammatik G gilt:

$$Link(G) = \{<S, S>, <X_1, X_1>, <X_2, X_2>, <X_3, X_3>, <X_4, X_4>, <S, X_2>, <S, e>, <X_2, e>, <X_1, g>, <X_3, X_1>, <X_3, g>, <X_4, h>\}$$

Für die Ablehnung von nicht zu L(G) gehörenden Ketten wie z.B. „fghe" oder „hefg" ist nun überhaupt kein Parsing-Aufwand mehr erforderlich; denn ein Blick in die Liste der *Link*-Paare zeigt, daß es keine S-Ableitung für Ketten geben kann, die mit den Symbolen „f", „g" und „h" beginnen.

Die einzige für den Erkennungs- bzw. Parsingalgorithmus notwendige Modifikation betrifft die Prozedur $\mathrm{REDUCE_{LC}}$, die um eine weitere Bedingung zu ergänzen ist:

Wenn X das oberste Symbol aus STACK2 ist, dann ist zu fordern, daß nur solche Regeln verwendet werden, deren linke Seite k_0 von X aus erreichbar ist ($<X, k_0> \in$ *Link*(G)). Die modifizierte Prozedur hat die Form:

PROZEDUR REDUCE$_{\mathrm{LC/LA}}$

DATEN: Eine kontextfreie Syntax $G = <V_N, V_T, S, R>$ und ein Lexikon L.
EINGABE: STACK1, mit *First*(STACK1) = k_1
STACK2
STACK3
AUSGABE: *Pop*(STACK1)
Push($k_2 \ldots k_n$ t, STACK2)
Push(k_0, STACK3)

VORBEDINGUNGEN:

1. Es gibt eine Regel $k_0 \rightarrow k_1 \ldots k_n \in R$ oder $k_1 \in k_0$, für eine beliebige lexikalische Kategorie k_0.
2. *First*(STACK2) $\in (V_N \cup V_T)$.
3. $<$*First*(STACK2), $k_0> \in$ *Link*(G).

Die Berechnung der LINK-Relation kann und sollte für eine Syntax ausgeführt werden, bevor sie zum Parsen verwendet wird, so daß der Parser direkt für ein Paar von Symbolen prüfen kann, ob sie die Erreichbarkeitsrelation erfüllen oder nicht. Die Alternative, erst beim Parsen die jeweils notwendigen Mengen zu berechnen, führt dazu, daß der beim Parsen erzielte Effizienzgewinn durch die für den Erreichbarkeitstest erforderlichen Berechnungen zumindest aufgehoben wird. In der Regel dürfte sich die Performanz deutlich verschlechtern.

5.5 Left-corner-Parsing in Lisp

Die in diesem Kapitel vorgestellten Algorithmen lassen sich mit geringem Aufwand in Lisp implementieren. Wie auch schon in den vorangegangenen Kapiteln stehen bei der Implementierung nicht Effizienzüberlegungen im Vordergrund, sondern es wird versucht, die Algorithmen möglichst direkt in lauffähige Programmme zu übersetzen.Es wird vorausgesetzt, daß auf die zur Analyse erforderlichen Daten (Syntax und Lexikon) über die globalen Variablen *syntax* und *lexikon* zugegriffen werden kann. Beide Programme erlauben auch die Analyse einzelner Konstituenten. In diesem Fall ist als zweites Argument die Liste mit der Zielkonstituente anzugeben.

Beispiel (5-5)

```
> (parse_lc '(der mann starb))
  (S (NP (DET DER) (N MANN)) (VP (V STARB)))

> (parse_lc '(der mann) '(np))
  (NP (DET DER) (N MANN))
```

5.5.1 Der Recognizer

Die drei Stacks werden durch Listen repräsentiert.

```
;;; RECOGNIZE-LC
;;; Die Funktion erwartet folgende Argumente:
;;;   STACK1  -  der zu analysierende Satz und
;;;   STACK2  -  das Startsymbol der Syntax
;;;   STACK3  -  leer
;;; Bis STACK1, STACK2 und STACK3 leer sind oder REDUCE-LC, MOVE-LC
;;; und REMOVE-LC als Wert NIL liefern, werden immer wieder zunächst alle drei
;;; Prozeduren angewandt und RECOGNIZE-LC rekursiv mit den von ihnen produ-
;;; zierten Werten aufgerufen. (Suchstrategie: Breadth-first-Suche)
(defun recognize-lc (stack1 &optional (stack2 `(,(startsymbol))) stack3)
  (if (and (endp stack1) (endp stack2) (endp stack3)) (print t)
      (dolist (x (append (reduce-lc stack1 stack2 stack3)
                         (move-lc stack1 stack2 stack3)
                         (remove-lc stack1 stack2 stack3)))
        (apply #'recognize-lc x))))
```

Die folgenden drei Funktionen realisieren die drei in 4.1 beschriebenen Prozeduren. Als Argumente nehmen sie alle die drei Listen, die die Stacks repräsentieren. Sind die für sie spezifizierten Vorbedingungen erfüllt, modifizieren sie diese Listen und liefern als Wert eine Liste, die eine Liste mit den neuen Listen enthält, sonst NIL. Anders als MOVE-LC und REMOVE-LC kann REDUCE-LC mehrere Resultatlisten als

Ergebnis liefern; d.h., der Wert der Funktionen läßt sich durch folgenden regulären Ausdruck beschreiben: ((Stack1' Stack2' Stack3')*).

```
;;; REDUCE-LC
(defun reduce-lc (stack1 stack2 stack3)
  (unless (eq (first stack2) t)
    (let ((resultat nil))
      (dolist (x (left-corner-regeln (first stack1)) resultat)
        (setf resultat
              (cons (list (rest stack1)
                          (append (nthcdr 2 x) (cons t stack2))
                          (cons (first x) stack3))
                    resultat))))))
;;; MOVE-LC
(defun move-lc (stack1 stack2 stack3)
  (when (and (eq (first stack2) t) (not (endp stack3)))
    (list (list (cons (first stack3) stack1) (rest stack2) (rest stack3)))))

;;; REMOVE-LC
(defun remove-lc (stack1 stack2 stack3)
  (when (eq (first stack1) (first stack2))
    (list (list (rest stack1) (rest stack2) stack3))))
;;; LEFT-CORNER-REGELN
;;; Wenn SYMBOL eine syntaktische oder lexikalische Kategorie bezeichnet, dann
;;; berechnet diese Funktion die Menge aller Regeln, deren linke Ecke SYMBOL ist;
;;; ist SYMBOL ein terminales Symbol, dann wird die Menge aller ihm zugeordneten
;;; lexikalischen Kategorien berechnet.
(defun left-corner-regeln (symbol)
  (or (rest (lexikon-eintrag[6]symbol))
      (apply #'append
             (mapcar #'(lambda (x) (when (eq symbol (second x)) (list x)))
                     (regeln)))))
```

5.5.2 Der Parser

Da die folgenden Funktionen weitgehend mit den vorangegangenen übereinstimmen, beschränken wir uns darauf, die wichtigsten Differenzen zu kommentieren. Alle Funktionen nehmen vier Argumente: der Parameter STACK4 wird in jedem Fall an eine Liste gebunden, die den vierten, zum Aufbau der Strukturbeschreibung benötigten Stack repräsentiert.

[6]Die Definition der Funktion LEXIKON-EINTRAG findet sich im Anhang E.1.1.

```
;;; PARSE-LC
(defun parse-lc (stack1 &optional (stack2 `(,(startsymbol))) stack3 stack4)
  (if (and (endp stack1) (endp stack2) (endp stack3))
      (print (first stack4))
      (dolist (x (append (reduce-lc/bf stack1 stack2 stack3 stack4)
                         (move-lc/bf stack1 stack2 stack3 stack4)
                         (remove-lc/bf stack1 stack2 stack3 stack4)))
        (apply #'parse-lc x))))
```

Die Definition von REDUCE-LC/BF fällt etwas komplexer aus, da zu unterscheiden ist, ob es in STACK4 bereits eine Strukturbeschreibung gibt, die mit dem ersten Symbol aus STACK1 beginnt oder nicht:

```
;;; REDUCE-LC/BF
(defun reduce-lc/bf (stack1 stack2 stack3 stack4)
  (unless (eq (first stack2) t)
    (let ((resultat nil))
      (dolist (x (left-corner-rules (first stack1)) resultat)
        (setf resultat
              (cons (list (rest stack1)
                          (append (nthcdr 2 x) (cons t stack2))
                          (cons (first x) stack3)
                          (if (eq (first stack1) (first (first stack4)))
                              (cons (list (first x) (first stack4)) (rest stack4))
                              (cons (list (first x) (first stack1)) stack4)))
                    resultat))))))
```

```
;;; MOVE-LC/BF
(defun move-lc/bf (stack1 stack2 stack3 stack4)
  (when (and (eq (first stack2) t) (not (endp stack3)))
    (list (list (cons (first stack3) stack1) (rest stack2) (rest stack3) stack4))))
```

Bei REMOVE-LC/BF ist zu unterscheiden, ob STACK3 leer ist oder nicht:

```
;;; REMOVE-LC/BF
(defun remove-lc/bf (stack1 stack2 stack3 stack4)
  (when (eq (first stack1) (first stack2))
    (list (list (rest stack1) (rest stack2) stack3
                (if stack3
                    (cons (append (second stack4) (list (first stack4)))
                          (nthcdr 2 stack4))
                    stack4)))))
```

5.6 Das BUP-System

Wir beenden dieses Kapitel mit der Beschreibung einer Prolog-Implementierung eines Left-corner-Parsers nach [?]. In dieser Implementierung werden Lexikon und Syntaxregeln getrennt repräsentiert. Der Zugriff auf das Lexikon erfolgt durch ein Selektorprädikat „dict/3". Die Eingabekette wird als Differenzlistenpaar repräsentiert. Das Programm verwendet das Backtracking des Prolog-Interpreters. Die Left-corner-Strategie schlägt sich bereits in der Regelnotation nieder: Im Gegensatz zu einer (top-down) DCG ist nicht die Mutterkategorie einer Regel der Kopf des entsprechenden Prolog-Prädikats, sondern die „linke Ecke", d.h. die erste Tochterkategorie.

5.6.1 Das Kernprogramm als Recognizer

Für linguistische Anwendungen werden häufig nur kontextfreie Regeln der Typen $X_0 \rightarrow X_1\ X_2 \ldots X_n$ und $X_0 \rightarrow w$ verwendet (mit $w \in V_T$ und $X_i \in V_N$). Diese Regeltypen lassen sich im DCG-Formalismus als Klauseln der folgenden Form repräsentieren:

x_0(Y, Z) :-
  x_1(Y, Y1),
  x_2(Y1, Y2),
  ...
  x_n(Y_{n-1}, Z).
x_0([w|Y], Y).

Y, Y1, ... etc. sind Differenzlistenvariablen, die einen Abschnitt des Satzes repräsentieren. DCG-Regeln dürfen nicht links-rekursiv sein, d.h. die Kategorien x_0 und x_1 dürfen nicht identisch sein, weil der als Parser benutzte Prolog-Interpreter top-down, links-rechts und depth-first ableitet und Linksrekursion aus diesem Grunde nicht terminiert. Linksrekursive Regeln sind allerdings in vielen Fällen aus linguistischen Gründen erwünscht (z.B.: $NP \rightarrow NP\ PP$, $S \rightarrow S\ S$, $S \rightarrow S\ VP$ usw.). Der folgende Left-corner-Parser kann solche linksrekursiven Regeln verarbeiten. Matsumoto et al. verwenden folgendes Regelformat:

x_1(G, Y1, Y) :-
  goal(x_2, Y1, Y2),
  ...
  goal(x_n, Yn-1, Yn),
  x_0(G, Yn, Y).

dict(x_0, [w|Y], Y).

Für die Terminierung ist als zusätzliche Klausel x_0(c_0, Y, Y) für jedes x_0 erforderlich. Das Prädikat 'goal' ist wie folgt definiert:

```
goal(G, X, Z) :-
    dict(C, X, Y),
    P =.. [C, G, Y, Z],
    call(P).
```

Die in diesen Klauseln verwendeten Variablen X, Y und Z sind Differenzlisten-Variablen. C und G sind Kategorien-Variablen. Das Prädikat 'goal' wird mit einem goal G und einem durch das Differenzpaar X,Z repräsentierten Satz w = [w_1, w_2, ..., w_n] aufgerufen. 'goal(G, X, Y)' ist erfolgreich, wenn w_1 einem Lexikoneintrag zufolge der Kategorie c angehört und es eine Regel mit dem Kopf x(G, Y, Z) (Y, Z repräsentiert dabei [w_2, ..., w_n]) gibt, so daß 'c(G, Y, Z)' bewiesen wird.

Beispiel (5-6)

Gegeben sei folgende Grammatik:

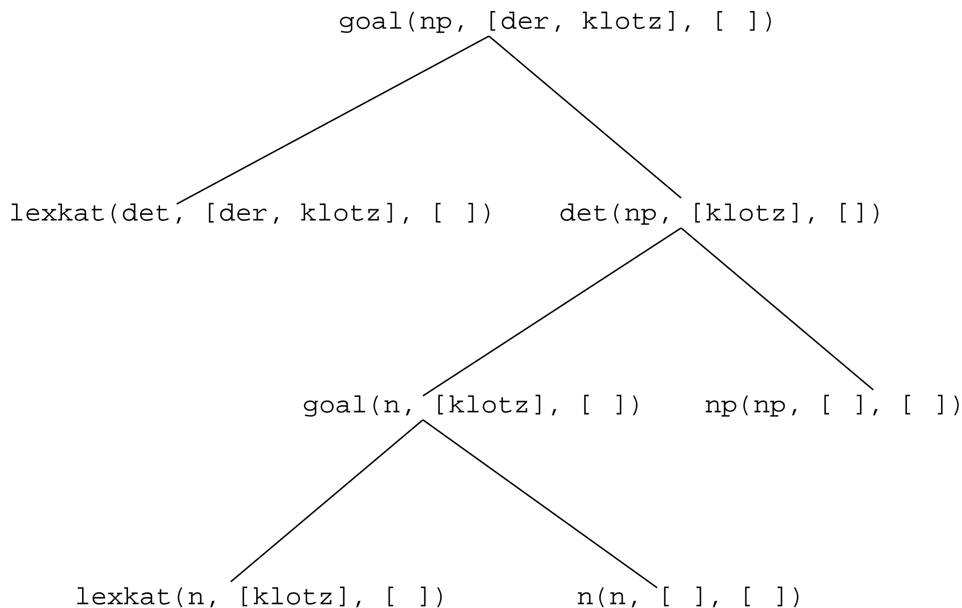
```
(1)     det(G, X, Z) :-
            goal(n, X, Y),
            np(G, Y, Z).

(2) a)  np(np, X, X).
    b)  det(det, X, X).
    c)  n(n, X, X).

(3) a)  dict(det, [der|X], X).
    b)  dict(n, [klotz|X], X).

(4)     goal(G, X, Z) :-
            dict(C, X, Y),
            P =.. [C, G, Y, Z],
            call(P).
```

Der Beweisbaum für *goal(np,[der,klotz],[ ])* hat die folgende Form:



Die erste Klausel der Definition des Prädikats GOAL unifiziert mit dem Fakt *dict(det,[der|X],X)*. Die Variable C erhält dadurch den Wert *det*. Die beiden letzten Klauseln der GOAL-Definition führen zu dem Aufruf *det(np,[klotz],[ ])*.

det(np,[klotz],[ ]) ist nach Regel (1) der Fall, wenn *goal(n,[klotz],Y)* und *np(np,Y,[ ])* erfolgreich sind. *np(np,Y,[ ])* ist nach (2)a nur dann der Fall, wenn Y = [ ]. Dann muß *goal(n,[klotz],Y)* also als *goal(n,[klotz],[ ])* beweisbar sein. *goal(n,[klotz],[ ])* führt zu den Aufrufen *dict(n,[klotz|X],[ ])* (unifiziert mit (3)b, wenn X = [ ]) und *n(n,[ ],[ ])* (unifiziert mit (2)c). Damit ist das Ziel *goal(np,[der,klotz],[ ])* bewiesen.

5.6.2 Optimierung: Die LINK-Relation

Die Verfolgung falscher Suchpfade (und damit der Backtracking-Aufwand) kann reduziert werden, indem für die jeweils benutzte Grammatik zuvor *LINK*(G) berechnet wird (vgl. 4.4). Um die Anwendung ungeeigneter Regeln zu verhindern, muß ein entsprechender Test in die Regeln integriert werden, der ihre Anwendung blockiert, wenn ihre Mutterkategorie nicht in der LINK-Relation zu der Mutterkategorie des zu konstruierenden (Teil-)Baums steht. Für die erste Regel der Grammatik aus Beispiel (5-6) sieht die Modifikation dann so aus:

det(G, X, Z) :-
    link(np, G),
    goal(n, X, Y),
    np(G, Y, Z).

Die Kategorien-Paare, für die die LINK-Relation gilt, werden als Fakten in der Wissensbasis abgelegt.

Aufgaben

5.1 Definieren Sie eine Lisp-Funktion bzw. ein Prolog-Prädikat zur Berechnung der LINK-Relation.

5.2 Implementieren Sie in Lisp einen Left-corner-Parser, der den Suchraum durch Verwendung der LINK-Relation einschränkt.

5.3 Entwickeln Sie ein Prolog-Programm, das eine kontextfreie Syntax, die ausschließlich Regeln der Typen $X_0 \rightarrow X_1\ X_2 \ldots X_n$ und $X_0 \rightarrow w$ (mit $w \in V_T$ und $X_i \in V_N$) enthält, in einen lauffähigen Left-corner-Parser mit Erreichbarkeitstafel (LINK-Relation) kompiliert.

Teil III

Chart-Parsing

Die im ersten Teil des Buches behandelten einfachen Parsingalgorithmen sind, anders als die meisten vor allem im Compilerbau verwendeten Algorithmen (z.B. die LL- oder LR-Parsingalgorithmen), insofern für computerlinguistische Aufgabenstellungen geeignet, als sie prinzipiell die Analyse beliebiger kontextfreier Sprachen ermöglichen. Allerdings weisen sie zwei erhebliche Nachteile auf:

1. Sie formulieren relativ starke Restriktionen bzgl. der Klasse der Syntaxen, die sie zu verarbeiten in der Lage sind: keine links-/ rechtsrekursiven Regeln bei Top-down-Parsing; keine Tilgungsregeln bei Bottom-up- und Left-corner-Parsing.

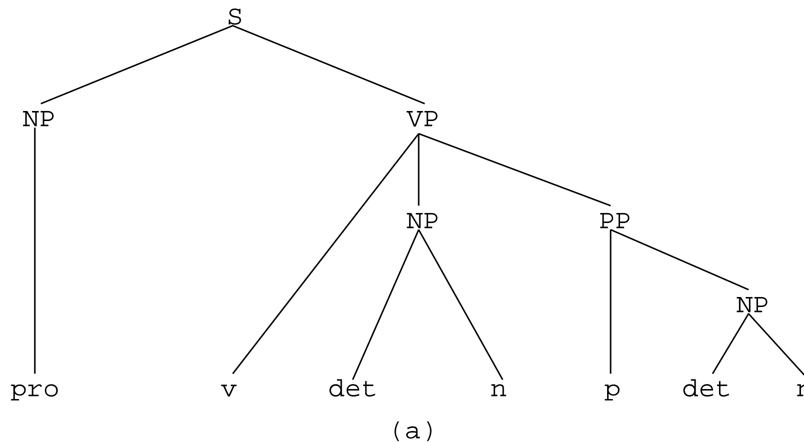
2. Sie sind *ineffizient*, da unabhängig von der gewählten Kontrollstrategie (depth-first mit Backtracking / quasi-parallele Breadth-first-Suche) nicht verhindert werden kann, daß in vielen Fällen ein Satzabschnitt mehrfach auf dieselbe Art und Weise analysiert wird.

Beispiel (III-1)

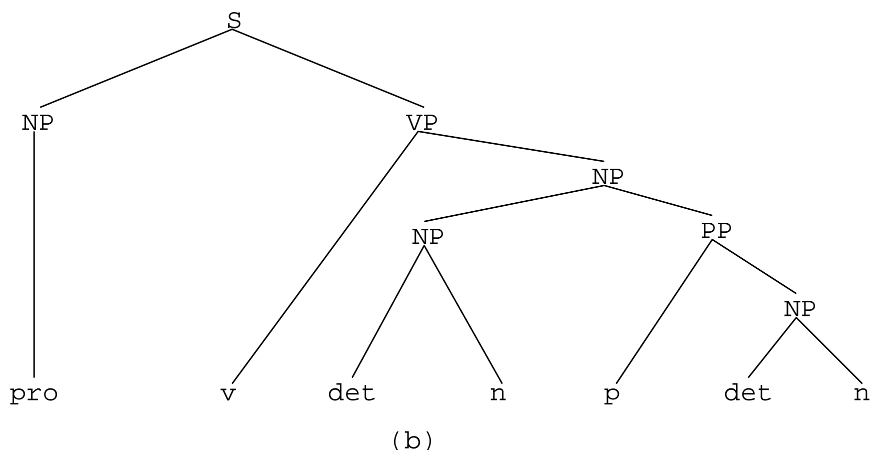
Syntax : $G_{R1} = \{ S \rightarrow NP\ VP,\ NP \rightarrow pro,\ NP \rightarrow det\ n,\ NP \rightarrow NP\ PP,$
$VP \rightarrow v\ NP,\ VP \rightarrow v\ NP\ PP,\ PP \rightarrow p\ NP \}$
$G_{R2} = G_{R1} \cup \{ VP \rightarrow v\ NP\ VP_{INF},\ VP_{INF} \rightarrow NP\ v \}$
$G_{R3} = G_{R1} \cup \{ S \rightarrow NP\ VP\ AP,\ AP \rightarrow adv\ adv \}$

Lexikon : { <er (pro) ...>, <mädchen (n) ...>, <sah (v) ...>,
<das (det) (relpro) ...>, <heute (adv)>, <auf (p) ...>, ...}

Sätze : (1) Er sah das Mädchen mit dem Skateboard.
(2) Er sah das Mädchen mit dem Skateboard die Straße hinunterjagen.
(3) Er sah das Mädchen mit dem Skateboard heute morgen.



(a)



(b)

Für Satz (1) gibt es relativ zur Syntax G_{R1} zwei Lesarten (s.o.): Einmal wird die PP als Komplement des Verbs, im anderen Fall als Attribut der Objekt-NP interpretiert. Die interne Struktur der als direktes Objekt fungierenden NP und der PP ist in beiden Fällen identisch. Trotzdem wird die Analyse dieser Satzabschnitte bei Verwendung der bislang betrachteten Algorithmen zweimal durchgeführt[1]. Noch deutlicher wird die Ineffizienz dieser Algorithmen, wenn man Satz (2) und Satz (3) und die Syntaxen G_{R2} und G_{R3} betrachtet: Die Analyse dieser Sätze kann wie zuvor zur Generierung der Strukturen II-1(a) bzw. II-1(b) führen. Sie terminiert in diesem Fall aber nicht, da bei beiden Sätzen das Satzende nicht erreicht ist. Bei der Analyse von Satz (2) erlaubt erst die neue VP-Regel eine vollständige Verarbeitung des Satzes und bei der Analyse von Satz (3) führt erst die zweite S-Regel zum Erfolg; d.h. die Analyse der Nominalphrasen und der Präpositionalphrase wird (im schlimmsten Fall) viermal ausgeführt.

Die Entwicklung und Realisierung effizienter Parsingalgorithmen setzt voraus, daß Verfahren gefunden werden, die es erlauben, Mehrfachanalysen dieser Art zu vermeiden. Ein Ansatz, der sich zur Lösung dieses Problems anbietet, besteht darin, die bei der voranschreitenden Analyse gewonnenen Teilergebnisse zu speichern, so daß auf sie zugegriffen werden kann, wenn für den Satz eine andere Analysemöglichkeit als die zunächst verfolgte geprüft wird. Zur Speicherung der Analyseergebnisse wird häufig eine *Chart* verwendet: Eine Chart ist ein *abstrakter Datentyp*, der charakterisiert ist durch die ihn konstituierenden Objekte und die auf diesen Objekten definierten Operationen. Da bei einem abstrakten Datentyp nur die wesentlichen Eigenschaften des zu charakterisierenden Datentyps genannt werden, gibt es in der Regel verschiedene Möglichkeiten, ihn zu realisieren. In den meisten Fällen wird eine Chart als *Vektor*, *Matrix* oder *azyklischer Graph* repräsentiert.

[1]Daß die Verwendung eines einfachen Top-down-Parsers für die Syntax aus dem vorangegangenen Beispiel wegen der linksrekursiven Regel NP $\rightarrow$ NP PP zu Problemen führen kann, soll hier vernachlässigt werden.

Die in der Chart gespeicherten Objekte werden als „Kanten" bezeichnet. Es werden zwei Typen von Kanten unterschieden: *aktive* und *passive* Kanten. Eine aktive Kante repräsentiert eine partiell erkannte Konstituente des Satzes; eine passive Kante eine vollständig erkannte Konstituente. Eine Kante muß mindestens die folgenden Informationen enthalten:

1. den kategorialen Typ der repräsentierten Konstituente,
2. die Angabe des Satzabschnitts, über den sie sich erstreckt, und
3. bei aktiven Kanten die Spezifikation des schon erkannten und des noch zu erkennenden Teils der Konstituente.

Um zu spezifizieren, welchen Abschnitt eines Satzes eine Konstituente abdeckt, werden die Positionen zwischen den Worten des Satzes numeriert, so daß der durch die Konstituente abgedeckte Satzabschnitt dann innerhalb der Kante durch Verweis auf die entsprechenden Indizes identifiziert werden kann. Ein Satz $\mathrm{w} = \mathrm{w}_1 \ldots \mathrm{w}_\mathrm{n}$ wird repräsentiert durch einen Ausdruck der Form:

$$0\ \mathrm{w}_1\ 1\ \mathrm{w}_2\ 2 \ldots \mathrm{n}\text{-}1\ \mathrm{w}_\mathrm{n}\ \mathrm{n}$$

Die auf einer Chart definierten Operationen fügen neue Kanten in die Chart ein. Es gibt zwei Typen von Operationen:

1. Operationen, die *passive* Kanten auf Grundlage des verfügbaren lexikalischen Wissens in die Chart eintragen und
2. Operationen, die abhängig von schon in der Chart enthaltenen Kanten und dem verfügbaren syntaktischen Wissen weitere *aktive/passive* Kanten erzeugen.

Unabhängig von den zur Repräsentation der Chart und Kanten gewählten Datenstrukturen verwenden wir folgende Notationskonventionen:

Wenn k eine beliebige Kante ist, dann bezeichnet:

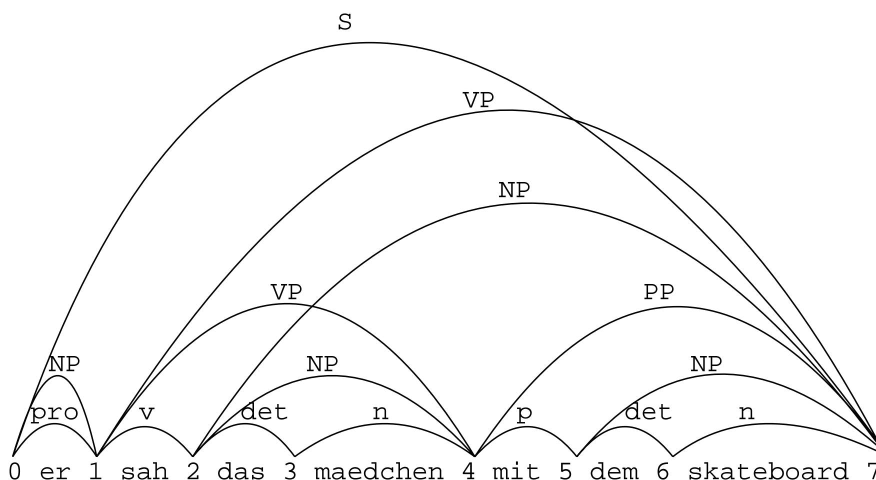
(i). '*Start*(k)' die Anfangsposition der Kante;
(ii). '*Ende*(k)' die Endposition der Kante;
(iii). '*Kopf*(k)' den Typ der durch k repräsentierten Konstituente[2];
(iv). '*geschlossener Abschnitt*(k)' den erkannten Teil und
(v). '*offener Abschnitt*(k)' den noch nicht erkannten Teil der durch k repräsentierten Konstituente.

Wenn k eine beliebige in einer Chart C gespeicherte Kante ist, mit *Start*(k) = i und *Ende*(k) = j, dann schreiben wir auch: 'k ∈ *Chart*(i, j)'.

[2]Wenn *Kopf*(k) = X, dann wird k auch einfach als eine *X-Kante* bezeichnet.

Beispiel (III-2)
Wenn wir ausgehend von der Syntax G_{R1} für den Satz (1) eine Chart berechnen, in die wir ausschließlich passive Kanten eintragen, erhalten wir folgende Ergebnisse:

(a) Azyklischer Graph



Die Repräsentation der *Chart* als azyklischer Graph ist sehr anschaulich, da sie direkten Aufschluß darüber liefert, aus welchen Konstituenten komplexe Konstituenten aufgebaut sind.

(b) Vektor

V_1	[pro, 0] [NP $\rightarrow$ pro]
V_2	[v, 1]
V_3	[det, 2]
V_4	[n, 3] [NP $\rightarrow$ det n, 2] [VP $\rightarrow$ v NP, 1]
V_5	[p, 4]

V_6	[det, 5]
V_7	[n, 6] [NP $\rightarrow$ det n, 5] [PP $\rightarrow$ p NP, 4] [NP $\rightarrow$ NP PP, 2] [VP $\rightarrow$ v NP PP, 2] [VP $\rightarrow$ v NP, 1] [**S $\rightarrow$ NP VP, 0**]

Für einen Satz der Länge n besteht der Vektor aus n Feldern. Wenn im Feld V_j des Vektors ein Eintrag der Form $<\alpha, i>$ enthalten ist, mit α als einer lexikalischen Kategorie oder syntaktischen Regel und $0 \leq i < n$, dann erstreckt sich die durch den Eintrag spezifizierte Konstituente vom i+1-ten bis zum j-ten Wort des Satzes. Der Satz ist akzeptiert, wenn es in V_n einen Eintrag der Form $<S \rightarrow \beta , 0>$ gibt.

(c) Matrix

	0	1	2	3	4	5	6	7
0		pro NP → pro						S → NP VP
1			v		VP → v NP			VP → v NP PP VP → v NP
2				det	NP → det n			NP → NP PP
3					n			
4						p		PP → p NP
5							det	NP → det n
6								n
7								

Ein Eintrag in Feld $M_{i,j}$ bezeichnet eine vom i+1-ten bis zum j-ten Wort des Satzes reichende Konstituente. Der Satz ist akzeptiert, wenn es in $M_{0,n}$ einen Eintrag der Form S $\rightarrow \beta$ gibt.

Natürlich sind für *Chart* und *Kanten* auch andere Repräsentationsformen möglich:

Beispiel (III-3)

Betrachtet man das vorangegangene Beispiel, dann könnte die *Chart* z.B. als eine **Menge** repräsentiert werden, die folgende passive Kanten enthält:

1 : [S, 3:NP 4:VP, *e*]
2 : [S, 3:NP 5:VP, *e*]
3 : [NP, pro, er]
4 : [VP, v 6:NP 7:PP, sah]
5 : [VP, v 8:NP, sah]
6 : [NP, det n, das mädchen]
7 : [PP, p 9:NP, mit]
8 : [NP, 6:NP 7:PP, *e*]
9 : [NP, det n, dem skateboard]

Jeder Eintrag ist indiziert und besteht aus der Angabe des Konstituententyps und der dominierten Kategorien. Nicht-lexikalischen Kategorien ist ein Index vorangestellt, der auf einen anderen Eintrag in der Chart verweist; für lexikalische Kategorien werden die ihnen zugeordneten Worte verzeichnet.

Kapitel 6

Bottom-up-Chart-Parsing

Wir werden in diesem Kapitel zwei Chart-Parsingalgorithmen entwickeln[1]. Beide Algorithmen arbeiten ausschließlich bottom-up und setzen voraus, daß die zu verarbeitende Syntax keine Tilgungsregeln und Zyklen enthält. Der erste dieser beiden Algorithmen stellt keine weiteren Anforderungen an die Syntax. Der zweite dagegen läßt sich nur auf Syntaxen in Chomsky-Normalform anwenden, d.h., die Syntax darf nur Regeln der Form $X \rightarrow Y\ Z$ und $X \rightarrow b$, mit $X, Y, Z \in V_N$ und $b \in V_T$, enthalten. Diese Restriktion ermöglicht eine sehr effiziente Berechnung der Chart.

6.1 Ein genereller Bottom-up-Chart-Parser

Der erste Algorithmus basiert auf folgender einfacher Idee: Zunächst werden für alle Wörter des Satzes auf Grundlage des Lexikons *lexikalische* Kanten gebildet, d.h. Kanten, die Konstituenten repräsentieren, deren Wurzel mit einer lexikalischen Kategorie etikettiert ist (z.B.: [1 N 1])[2]. Anschließend werden dann - gesteuert durch die Regeln der Syntax - solange bereits gefundene Kanten zu neuen (syntaktischen)

[1]Der zweite Algorithmus basiert auf dem von Cocke, Kasami und Younger unabhängig voneinander entwickelten Algorithmus (vgl. [?], [?] und [?]); der erste kann als Generalisierung dieses Algorithmus aufgefaßt werden (vgl. [?, S. 105-8]).

[2]Alle anderen Kanten bezeichnen wir als *syntaktische* Kanten.

Kanten kombiniert, bis sich schließlich keine weiteren Kanten mehr bilden lassen.

PROZEDUR LEXIKALISCHE-KANTEN

DATEN: Ein Lexikon L.
EINGABE: Eine Wort w und ein Index i ($n \geq 1$).
AUSGABE: Eine Menge von lexikalischen Kanten K.
ARBEITSSTRUKTUREN:
L-KANTEN - Eine Menge von lexikalischen Kanten.
Anfangswert = $\emptyset$.
METHODE:
Für jede lexikalische Kategorie K_L, mit $w \in K_L$:
<L-KANTEN $\Leftarrow$ L-KANTEN $\cup$ {[i-1, K_L, i]}>
<RETURN(L-KANTEN)>

Die Prozedur Lexikalische-Kanten bildet für jede Lesart eines Wortes eine lexikalische Kante, die zur Initialisierung der Chart verwendet werden. Um neue (syntaktische) Kanten zu bilden, führt NEUE-KANTEN für jede bereits gefundene Kante k = [i, Y, j] folgende Aktionen aus:

1. Es werden alle Regeln aus R ausgewählt, deren linke Ecke mit dem *Kopf*(k) identisch ist.

2. Für jede Regel $r{:}X \rightarrow Y_0\ Y_1 \ldots Y_n$ ($n \geq 0$), die Bedingung (1) erfüllt, wird überprüft, ob es Kanten $[i_1, Y_1, j_1] \ldots [i_n, Y_n, j_n]$ in der Chart gibt, die mit k kombiniert werden können. Ist das der Fall, wird die Kante $[i_0, X, j_n]$ in die Chart eingetragen.

PROZEDUR NEUE-KANTEN

DATEN: Eine kontextfreie Syntax G = $<V_N, V_T, S, R>$, die keine Tilgungsregeln enthält.
EINGABE: Eine Menge von Kanten K.
AUSGABE: Eine Menge von neugebildeten Kanten K'.
ARBEITSSTRUKTUREN:
S-KANTEN - Eine Menge von syntaktischen Kanten.
Anfangswert = $\emptyset$.
METHODE:
Für jedes k = $[i_0, Y_0, j_0] \in K$:
Für jede Regel r = $X \rightarrow Y_0\ Y_1 \ldots Y_n$ $(n \geq 0) \in R$:
Wenn < Es gibt eine Folge von Kanten $k_1, \ldots, k_n$,
mit $k_m = [i_m, Y_m, j_m]$ und $1 \leq m \leq n$,
so daß gilt: $i_m = j_{m-1}$ > $\wedge$
< $[i_0, X, j_n]$ ist nicht in K enthalten >
Dann <S-KANTEN $\Leftarrow$ S-KANTEN $\cup$ $\{[i_0, X, j_n]\}$ >

<RETURN(S-KANTEN)>

CHART-RECOGNIZE kombiniert beide Prozeduren zur Berechnung aller Konstituenten des Satzes:

ALGORITHMUS CHART-RECOGNIZE
bottom-up / links-rechts
breadth-first

DATEN: Ein Lexikon L und eine kontextfreie Syntax G = $<V_N, V_T, S, R>$, die zyklenfrei ist und keine Tilgungsregeln enthält.
EINGABE: Ein Satz w = $w_1 \ldots w_n$ $(n \geq 1)$.
AUSGABE: *True/False*.

ARBEITSSTRUKTUREN:

Chart - Eine Menge von *Kanten*. Eine Kante k besteht aus:

Anfang = eine Zahl,
Ende = eine Zahl und
Kopf = ein Kategoriensymbol

Anfangswert: ∅.

METHODE:

Für jedes Wort w_i:
Chart ⇐ Chart ∪ *Lexikalische-Kanten*(w_i, i).

Wiederhole:
Wenn <[0, S, n] ∈ CHART>
Dann <RETURN(*True*)>
Sonst
Wenn <*Neue-Kanten*(CHART) = ∅>
Dann <RETURN(*False*)>
Sonst <Chart ⇐ Chart ∪ *Neue-Kanten*(CHART)>

Beispiel (6-1)

Ausgehend von dem Lexikon und der Syntax G_{R1} aus Beispiel (III-1), erhalten wir für den Satz „Er sah das Mädchen mit dem Skateboard“ folgendes Ergebnis:

Die Prozedur LEXIKALISCHE-KANTEN generiert die Kanten:
{ [0, pro, 1], [1, v, 2], [2, det, 3], [2, relpro, 3], [3, n, 4], [4, p, 5],
[5, det, 6], [5, relpro, 6], [6, n, 7] }

Anschließend werden durch NEUE-KANTEN folgende weitere Kanten erzeugt:

(1)	[0, NP, 1]	aus: NP → pro	&	[0, pro, 1]
	[2, NP, 4]	aus: NP → det n	&	[2, det, 3], [3, n, 4]
	[5, NP, 7]	aus: NP → det n	&	[5, det, 6], [6, n, 7]
(2)	[1, VP, 4]	aus: VP → v NP	&	[1, v, 2], [2, NP, 4]
	[4, PP, 7]	aus: PP → p NP	&	[4, p, 5], [5, NP, 7]
(3)	[1, VP, 7]	aus: VP → v NP PP	&	[1, v, 2], [2, NP, 4], [4, PP, 7]
	[2, NP, 7]	aus: NP → NP PP	&	[2, NP, 4], [4, PP, 7]
(4)	**[0, S, 7]**	aus: S → NP VP	&	[0, NP, 1], [1, VP, 7]

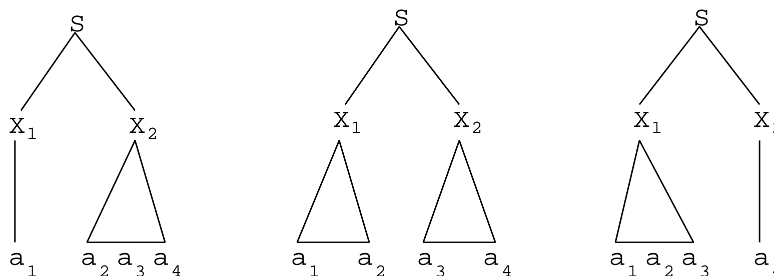
Ein zweiter Eintrag '[1, VP, 7]' aus: VP → v NP & [1, v, 2], [2, NP, 7] wird nicht gebildet, da es schon einen solchen Eintrag gibt und es für einen Recognizer unwesentlich ist, daß es verschiedene Möglichkeiten gibt, eine durch einen Eintrag bezeichnete Konstituente zu bilden.

Die konzeptuelle Einfachheit des Verfahrens steht in einem gewissen Kontrast zu seiner Effizienz: Die Berechnung der Chart ist ein relativ komplexer Prozeß, da die zur Bildung der syntaktischen Kanten benötigte Zahl der Vergleiche sehr hoch ist.

6.2 Der Cocke-Kasami-Younger-Algorithmus

Der Cocke-Kasami-Younger-Algorithmus kann als Spezialfall des im letzten Abschnitt beschriebenen Algorithmus betrachtet werden. Die Beschränkung auf Syntaxen in Chomsky-Normalform ermöglicht es, das Verfahren zur Berechnung der Chart erheblich zu vereinfachen. Bei allen kontextfreien Syntaxen, die keine Tilgungsregeln enthalten, wächst bei einer Ableitung die Länge des abgeleiteten Ausdrucks monoton. Darüber hinaus läßt sich für eine Syntax G in Chomsky-Normalform und einen Satz $w = w_1 \ldots w_n \in L(G)$ die Länge der Ableitungen von w in G bestimmen: In jedem Fall werden n lexikalische Regeln ($X \rightarrow b$) und n-1 syntaktische Regeln ($X \rightarrow Y\ Z$) angewendet; d.h. es gibt n-1 Möglichkeiten, den Satz in zwei Teile zu zerlegen.

Beispiel (6-2)
Satz : $a_1 a_2 a_3 a_4$



Der Cocke-Kasami-Younger-Algorithmus beschreibt ein effizientes Verfahren, um auf der Grundlage dieses Wissens für einen Satz alle Konstituenten zu berechnen.

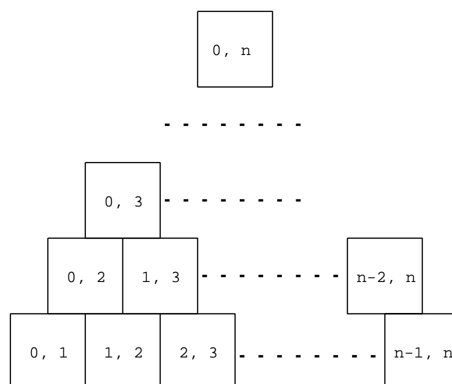


Abbildung (6-1)

Wir wählen zur Repräsentation der *Chart* in diesem Fall eine Matrix, die wir, da für einen Satz der Länge n nur die Felder (0, 1), (1, 2), ..., (n-1, n), (0, 2), ..., (n-2, n), ..., (0, n) benötigt werden, als eine *Pyramide* darstellen. Ein Eintrag in einem Feld(i, j) repräsentiert eine Konstituente, die vom i+1-ten Wort bis zum j-ten Wort reicht.

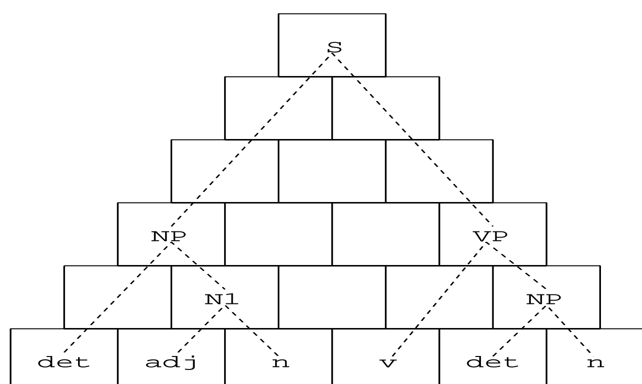
Wir werden zwei Varianten des Algorithmus beschreiben. Zunächst spezifizieren wir einen Algorithmus zur Berechnung einer Analysematrix, in der die Felder nur lexikalische/syntaktische Kategoriensymbole enthalten: Jede dieser Kategorien repräsentiert eine Konstituente, und die Indizes des Felds, in dem sie eingetragen wird, markieren den Satzabschnitt, über den sich diese Konstituente erstreckt. Bei der zweiten Variante des Algorithmus wird zusätzlich in allen Feldern(i,j), mit $j - i > 1$, für jedes Symbol ein Verweis auf die beiden Subkonstituenten gespeichert, die den Eintrag des Symbols in diesem Feld legitimieren. Für beide Varianten geben wir ein Verfahren zur Berechnung eines Leftparses bzw. einer Strukturbeschreibung an.

6.2.1 Berechnung einer einfachen Analysematrix

Beispiel (6-3)

Syntax : { S → NP VP, NP → det n, NP → det N1, N1 → adj n,
VP → v NP,VP → v PP, PP → p NP }[3]

Satz : Der faule Hund sucht einen Schlafplatz.



[3]Alle in den Regeln vorkommenden Symbole gelten hier als nicht-terminale Symbole; die lexikalischen Regeln werden nicht explizit angegeben.

ALGORITHMUS RECOGNIZE$_{CKY-1}$
einfache Analysematrix
Eintrag: Symbole

DATEN: Ein Lexikon L und eine kontextfreie Syntax $G = <V_N, V_T, S, R>$ in Chomsky-Normalform.
EINGABE: Ein Satz $w = w_1 \dots w_n$ $(n \geq 1)$.
AUSGABE: *True*/*False*.
ARBEITSSTRUKTUREN:
CHART[4]- Eine Analysematrix für S mit n Spalten und Zeilen.
Anfangswert: alle Felder sind leer.
METHODE:
Initialisierung der Analysematrix:
Für i = 0 bis n-1: $<CHART_{i,i+1} \Leftarrow \{ X \mid X \in V_T \text{ und } w_{i+1} \in X \}>$
Berechnung der übrigen Felder:
Für k = 2 bis n:
  Für i = 0 bis n-k:
    $j \Leftarrow i + k$
    $CHART_{i,j} \Leftarrow \{A \mid A \rightarrow X\ Y \in R \wedge \exists(m)\ (X \in CHART_{i,m} \wedge Y \in CHART_{m,j})$, mit $i < m < j \}$

Wenn $<S \in CHART_{0,n}>$
Dann <RETURN(*True*)>
Sonst <RETURN(*False*)>

Die Prozedur RECOGNIZE$_{CKY-1}$ berechnet die Felder von links nach rechts bottom-up.

PROZEDUR LINKSPARSE
Berechnung eines Linksparse
aus einer einfachen Analysematrix

DATEN: Eine kontextfreie Syntax $G = <V_N, V_T, S, R>$ in Chomsky-Normalform, deren Regeln indiziert sind, und eine einfache Analysematrix für einen Satz $w = w_1 \dots w_n$.
EINGABE: i $(0 \leq i < n)$
j $(1 \leq j \leq n)$
A $(A \in V_N)$
AUSGABE: Ein *Linksparse* für w.

[4]'$CHART_{i,j}$' bezeichnet das j-te Feld in der i-ten Zeile.

METHODE:
Wenn <i = j-1>
Dann <RETURN(w_j)>
Sonst <k sei die kleinste Zahl, für die es:
(i). ein $X \in CHART_{i,k}$,
(ii). ein Y in $CHART_{k,j}$ und
(iii). eine Regel $A \rightarrow X\ Y \in R$ gibt.>
<RETURN(*Index*(A → X Y))>
<*Linksparse*(i, k, X)>
<*Linksparse*(k, j, Y)>

Wenn $w \in L(G)$, dann extrahiert die Prozedur LINKSPARSE aus einer mit dem vorherigen Algorithmus generierten Analysematrix eine Linksableitung für w. Voraussetzung ist, daß die Regeln der Syntax indiziert sind. $RECOGNIZE_{CKY-1}$ kann zu einem Parsingalgorithmus erweitert werden, indem die Prozedur LINKSPARSE mit *Linksparse*(0, n, S) aufgerufen wird, wenn $S \in CHART_{0,n}$.

Beispiel (6-4)
Analysematrix : Beispiel (6-3)
Satz : Der faule Hund sucht einen Schlafplatz

Linksparse(0, 6, S)
  1
  Linksparse(0, 3, NP)
    3
    Linksparse(0, 1, det)
      der
      Linksparse(1, 3, N1)
        4
        Linksparse(1, 2, adj)
          faule
        Linksparse(2, 3, n)
          Hund
  Linksparse(3, 6, VP)
    5
    Linksparse(3, 4, v)
      sucht
      Linksparse(4, 6, NP)
        2
        Linksparse(4, 5 det)
          einen
        Linksparse(5, 6, n)
          Schlafplatz
= 1 3 der 4 faule Hund 5 sucht 2 einen Schlafplatz

6.2.2 Berechnung einer komplexen Analysematrix

Die andere Variante unseres Algorithmus ($RECOGNIZE_{CKY-2}$) zur Berechnung der Chart erhalten wir, indem wir die Felder(i, j), mit j-i > 1 nach folgendem Verfahren berechnen:

Berechnung der übrigen Felder:

Für k = 2 bis n:
Für i = 0 bis n-k:
$j \Leftarrow i+k$
$CHART_{i,j} = \{<A\ (X,\ i,\ m)\ (Y,\ m,\ j)> \mid A \rightarrow X\ Y \in R\ \wedge$
$\exists(m)\ (X \in CHART_{i,m} \wedge Y \in CHART_{m,j}),\ mit\ i < m < j\ \}$

Der Satz wird akzeptiert, wenn $[S, \alpha, \beta] \in CHART_{0,n}$. Eine nach diesem Verfahren berechnete Analysematrix bezeichnen wir als *komplexe* Analysematrix.

Beispiel (6-5)

Syntax : siehe Beispiel (6-3)
Satz : Der faule Hund sucht einen Schlafplatz.

$CHART_{0,1}$ = {det}
$CHART_{1,2}$ = {adj}
$CHART_{2,3}$ = {n}
$CHART_{3,4}$ = {v}
$CHART_{4,5}$ = {det}
$CHART_{5,6}$ = {n}
$CHART_{1,3}$ = {[N1, (adj, 1, 2), (n, 2, 3)]}
$CHART_{4,6}$ = {[NP, (det, 4, 5), (n, 5, 6)]}
$CHART_{0,3}$ = {[NP, (det, 0, 1), (N1, 1, 3)]}
$CHART_{3,6}$ = {[VP, (v, 3, 4), (NP, 4, 6)]}
$CHART_{0,6}$ = {[S, (NP, 0, 3), (VP, 3, 6)]}

PROZEDUR STRUKTUR
Berechnung einer Strukturbeschreibung
aus einer komplexen Analysematrix

DATEN: Eine komplexe Analysematrix für einen Satz $w = w_1 \ldots w_n$.
EINGABE: Ein Symbol $A \in V$ und zwei Indizes i, j ($1 \leq i, j \leq n$).
AUSGABE: Eine Strukturbeschreibung für w.

METHODE:
Wenn <i = j-1>
Dann <RETURN([A w_j])>
Sonst <Wähle ein [A, (X, i, m) (Y, m, j)] $\in CHART_{i,j}$>
<RETURN([A *Struktur*(X, i, m) *Struktur*(Y, m, j)>

Für eine Syntax $G = \langle V_N, V_T, S, R \rangle$, einen Satz $w = w_1 \ldots w_n$, mit $w \in L(G)$ und eine mit RECOGNIZE_{CKY-2} berechnete Analysematrix generiert die Prozedur STRUKTUR eine Strukturbeschreibung für w. Die Prozedur STRUKTUR muß aufgerufen werden mit: *Struktur*(Startsymbol_G, 0, n). Die Prozedur STRUKTUR liefert wie auch die Prozedur LINKSPARSE nur *eine* Strukturbeschreibung für jeden Satz. Beide Prozeduren lassen sich aber prinzipiell so generalisieren, daß sie alle Strukturbeschreibungen eines Satzes generieren.

Beispiel (6-6)

Analysematrix: siehe Beispiel (6-5)

```
Struktur(S, 0, 6) ⟹
[ S
  Struktur(NP, 0, 3)
     [ NP
       Struktur(det, 0, 1)
          [det der]
       Struktur(N1, 1, 3)]
          [ N1
            Struktur(adj, 1, 2)
               [adj faule]
            Struktur(n, 2, 3)]
               [n Hund]
          ]
     ]
  Struktur(VP, 3, 6)
     [ VP
       Struktur(v, 3, 4)
          [v sucht]
       Struktur(NP, 4, 6)
          [ NP
            Struktur(det, 4, 5)
               [det einen]
            Struktur(n, 5, 6)
               [n Schlafplatz]
          ]
     ]
]
= [S [NP [det der] [N1 [adj faule] [n Hund]]]
       [VP [v sucht] [NP [det einen] [n Schlafplatz]]]]
```

6.2.3 Zeit- und Speicherbedarf des Cocke-Kasami-Younger-Algorithmus

Theorem (1) - Zeitbedarf

Wenn G =<V_N, V_T, S, R> eine beliebige Syntax in Chomsky-Normalform ist, dann ist der Zeitaufwand zur Berechnung der Analysematrix (und damit um zu entscheiden, ob $w \in L(G)$) proportional zu n^3, mit $n = |w|$. Der Zeitbedarf des Algorithmus ist also $O(n^3)$.

Theorem (2) - Speicherbedarf

Wenn G =<V_N, V_T, S, R> eine beliebige Syntax in Chomsky-Normalform ist, dann ist der Speicherbedarf des Cocke-Kasami-Younger-Algorithmus $O(n^2)$[5].

6.3 Implementierung der Algorithmen

Auf der Grundlage des ersten der beiden in diesem Kapitel vorgestellten Algorithmen entwickeln wir einen Recognizer in Prolog (5.3.1). Die beiden Algorithmen $\text{RECOGNIZE}_{\text{CKY}-1}$ und $\text{RECOGNIZE}_{\text{CKY}-2}$ implementieren wir ebenso wie die Funktionen LEFTPARSE und STRUKTUR in Lisp.

6.3.1 Prolog-Implementierung des generellen Bottom-up-Chart-Parsers

Der Recognizer ist relativ ineffizient, da jedesmal, wenn eine neue Kante k generiert wird, die Chart sequentiell nach Kanten durchsucht wird, mit der k wiederum kombiniert werden kann, so daß keine wesentlichen Effizienzvorteile gegenüber einem Backtracking-Algorithmus erzielt werden. Die Formulierung des Algorithmus als Prolog-Programm ist sehr eng an der Formulierung des Algorithmus orientiert, so daß die konzeptuelle Einfachheit des Algorithmus erhalten bleibt.

Repräsentation des Lexikons und der Grammatik

Das Lexikon wird durch eine Menge von Fakten der Form *lex(wortform, lexikalische-Kategorie)* repräsentiert; syntaktische Regeln werden durch Fakten der Form *regel(linke-seite, rechte-seite)* abgebildet. *Wortform*, *lexikalische-Kategorie* und *linke-seite* sind Prolog-Atome und *rechte-seite* ist eine Liste von Prolog-Atomen.

Hauptprädikate

```
% recognize/2
% Der zu analysierende Satz wird durch eine Liste von Prolog-Atomen repräsentiert.
recognize(K, ja) :-
    chart_weg,                    % Entferne alte Chart aus der Wissensbasis.
```

[5]Der Beweis für beide Theoreme findet sich z.B. bei [?, S.317-20] bzw. [?, S.107-11].

```
    lex_kante(K, 1),              % Erzeuge lexikalische Kanten.
    neue_kanten,                  % Vervollständige die Chart.
    kante(0, S, N),               % Chart enthält eine S-Kante.
    startsymbol(S),               % Eingabekette vollständig abgearbeitet:
    length(K, N), !.              % Kein Backtracking ist erforderlich.
recognize(K, nein).               % Satz nicht akzeptiert.

% lex_kante/2
% Für jedes Wort des Satzes werden alle lexikalischen Kanten generiert.
lex_kante([ ], _).                % Der Satz wurde vollständig abgearbeitet.
lex_kante([W1 | Ws], I) :-
    eintraege(W1, I),
    J is I + 1,                   % Der Zeiger weist auf das nächste Wort.
    lex_kante(Ws, J).             % Rekursion mit dem Restsatz.

% eintraege/2
% Für ein Wort W mit Index I werden passende Lexikoneinträge gesucht und pro
% Eintrag eine (lexikalische) Kante in die Chart eingetragen.
eintraege(W, I) :-
    lex(W, Kat),                  % Suche Lexikoneintrag.
    I1 is I - 1,
    assertz(kante(I1, Kat, I)),   % Trage die Kante in die Chart ein.
    fail.                         % Suche weitere Einträge für W.
eintraege(_, _).                  % Terminierung.

% neue_kanten/0
% Vervollständigung der Chart durch (syntaktische) Kanten.
neue_kanten :-
    kante(I, Y, J),               % Es gibt eine Kante mit dem Kopf Y.
    regel(X, [Y | Ts]),           % Es gibt eine Regel, mit Y als linker Ecke.
    finde_kanten(Ts, J, _, I, X). % Suche Kanten für die übrigen Kategorien.
neue_kanten.

% finde_kanten/5
% Finde für eine Liste von Kategorien (Rechte Seite einer Regel minus erstem Sym-
% bol) eine Menge von Kanten, die miteinander verbunden werden können. Ist eine
% solche Kantenfolge gefunden, wird eine neue Kante [I, X, J] in die Chart eingetra-
% gen und der Suchprozeß nach weiteren Kanten durch den rekursiven Aufruf von
  neue_kanten/0 fortgesetzt.
finde_kanten([K | Ks], J, N, I, X) :-
    kante(J, K, J1),              % Kante für die erste Kategorie gefunden.
```

```
    finde_kanten(Ks, J1, N, I, X). % Suche Kanten für die restlichen Kategorien.
finde_kanten([ ], J, J, I, X) :-    % Alle Kategorien verarbeitet.
    not(kante(I, X, J)),            % Die Kante ist neu und
    asserta(kante(I, X, J)),        % wird in die Chart eingetragen.
    neue_kanten.                    % Rekursion.
```

Hilfsprädikate

```
% chart/0
% Darstellung der Chart.
chart :-
    kante(A, B, C),
    write([A, B, C]),
    get0(_), nl, fail.
chart.

% chart_weg/0
% Löschen der alten Chart.
chart_weg :-
    retractall(kante(_, _, _)).
retractall(X) :-
    retract(X), fail.
retractall(X).
```

6.3.2 Der Cocke-Kasami-Younger-Algorithmus

Es folgt eine Reihe von Lisp-Programmen zur Berechnung einfacher bzw. komplexer Analysematrizen und zur Generierung verschiedener Typen von Strukturbeschreibungen aus Analysematrizen.

Es wird vorausgesetzt, daß die zur Analyse erforderlichen Daten (Syntax und Lexikon) den globalen Variablen *syntax* und *lexikon* zugewiesen sind.

Die Programme erlauben auch die Analyse einzelner Konstituenten. In diesem Fall ist als zweites Argument die Liste mit der Zielkonstituente anzugeben.

Berechnung einer einfachen Analysematrix

Innerhalb des folgenden Programms wird die Chart durch eine Liste von *Feldern* dargestellt, wobei ein Feld selbst wieder durch eine Liste der Form

$$((i\ j)\ \text{Kategorie}_1 \ldots \text{Kategorie}_n)\ (n \geq 0)$$

repräsentiert wird; d.h. es beginnt mit einer Indexliste, auf die eine Anzahl von Einträgen (lexikalische/syntaktische Kategorien) folgt. So repräsentiert „((1 3) NP

N1 S)“ z.B. das Feld der Chart mit den Koordinaten 1 und 3. Es enthält drei Einträge: NP, N1 und S.

```
;;; RECOGNIZE-CKY
;;; Diese Funktion berechnet die Chart (MAKE-CHART) und überprüft, ob das
;;; Startsymbol der Syntax in Feld(0, n) enthalten ist.
(defun recognize-cky (satz &optional (symbol (startsymbol)))
  (let ((chart (make-chart satz)))
    (pprint chart)                          ; Ausgabe der berechneten Chart.
    (when (member symbol (first chart)) t)))

;;; MAKE-CHART
;;; Diese Funktion generiert für eine Syntax SYNTAX in Chomsky-Normalform die
;;; Chart. Sie wird von unten nach oben und von links nach rechts berechnet.
(defun make-chart (satz)
  (let ((chart (lexical-items satz))          ; Berechne die lexikalischen
        (n (length satz)))
    (when (> n 1)
      (compute-other-items n chart))))        ; und die syntaktischen Kanten.

;;; LEXICAL-ITEMS
;;; Generiert die 'unterste Zeile' der Chart; d.h die Felder(i, i+1), mit i = 0, 1,
;;; ..., |SATZ|-1. Für jede Wortform des Satzes werden alle passenden lexikalischen
;;; Kategorien in das ihr zugeordnete Feld eingetragen.
(defun lexical-items (satz)
  (let ((items nil) (index 0))
    (dolist (x satz items)
      (setf items (append items
                          (list `((,index ,(+ index 1)) ,@(eintrag-kategorien[6] x))))
            index (+ index 1)))))

;;; COMPUTE-OTHER-ITEMS
;;; Diese Funktion generiert die übrigen Felder der Chart, d.h. die Felder(i, j), mit
;;; j - i > 1. Die Einträge bestehen aus syntaktischen Kategorien der Syntax.
(defun compute-other-items (n chart)
  (do ((new-items chart)
       (k 2 (+ k 1)))
      ((> k n) new-items)
      (do* ((i 0 (+ i 1))
            (j (+ i k) (+ j 1))
```

[6]Diese Funktion liefert die Liste aller (lexikalischen) Kategorien, die WORT durch das aktuelle Lexikon zugeordnet werden. Ihre Definition findet sich im Anhang D.1.

```
          (symbols (search-chart i j new-items) (search-chart i j new-items)))
         ((> i (- n k)))
         (when symbols (setf new-items (cons `((,i ,j) ,@symbols) new-items))))))
;;; SEARCH-CHART
;;; Berechnet die Einträge für das Feld(i, j) der Chart. Es müssen für k (i < k < j) alle
;;; (schon berechneten) Paare von Feldern Feld(i, k), Feld(k, j) durchsucht werden.
(defun search-chart (i j chart)
  (let ((resultat nil))
    (dotimes (k (- j i 1) resultat)
      (setf resultat
            (append resultat
                    (match (assoc (list i (+ k i 1)) chart :test #'equal)
                           (assoc (list (+ k i 1) j) chart :test #'equal)
                           (regeln)))))))

;;; MATCH
;;; Berechnet für zwei Listen von Kategorien die Liste der linken Seiten der Re-
;;; geln, deren rechte Seite aus einer Kategorie aus FELD-1 und einer Kategorie aus
;;; FELD-2 besteht.
(defun match (feld-1 feld-2 regel-set)
  (let ((resultat nil) (regeln nil))
    (dolist (x (rest feld-1) resultat)
      (dolist (y (rest feld-2))
        (when (setf regeln(remove-if #'(lambda (z) (not (equal (rest z) (list x y)))) regel-set))
          (setf resultat (append resultat (mapcar #'first regeln))))))))
```

Berechnung einer komplexen Analysematrix

Um statt einer einfachen Analysematrix eine komplexe Analysematrix zu generieren, ist es nur erforderlich, die Funktion MATCH zu modifizieren, die die Einträge für alle Felder(i, j), mit j - i > 1, berechnet. Alle Felder(i, j), mit j-i > 1 haben die Form: $((i\ j)\ (kategorie_1\ i_1\ j_1)\ \ldots (kategorie_n\ i_n\ j_n))$

Ein Eintrag in diesen Feldern besteht aus einer syntaktischen Kategorie und den Indizes des Feldes, in dem es einen Eintrag für diese Kategorie gibt.

```
;;; MATCH
;;; Berechnet für zwei Listen von Einträgen die Liste der linken Seiten der Regeln,
;;; deren rechte Seite aus einer Kategorie besteht, für die es einen Eintrag in FELD-1
;;; gibt, gefolgt von einer Kategorie, für die es einen Eintrag in FELD-2 gibt.
(defun match (feld-1 feld-2 regeln)
  (let ((resultat nil) (regeln1 nil))
    (dolist (x (get-categories feld-1) resultat)
```