Seminarskript

Parsing I

Helmut Schmid

Achtung: Dieses Skript deckt den Inhalt des Seminares nur teilweise ab!

Inhaltsverzeichnis

1 Thema und Motivation

Natürliche Sprachen können durch *Grammatiken* beschrieben werden. Ziel des *Parsings* ist es, möglichst effiziente Verfahren und Algorithmen zu beschreiben, die es erlauben, für eine bestimmte Grammatik und einen Eingabesatz festzustellen, ob dieser Satz durch die Grammatik beschrieben wird und somit zu der Sprache gehört oder nicht (das sogenannte *Wortproblem*).

Ziel des Parsing-Seminars ist es, die bisher eher informell verwendeten Begriffe *Grammatik, Sprache, Syntaxbaum, Satz der Sprache* etc. formal zu definieren und insbesondere im Hinblick auf die Verarbeitung natürlicher Sprache verschiedene *Formalismen* anzugeben, die eine linguistisch orientierte Formulierung von Grammatiken und ein effizientes Parsing von Eingabesätzen erlauben.

Die wesentliche Grundlage bildet dabei die Theorie der formalen Sprachen, die in dem Seminar "Formale Sprachen" ausgiebig behandelt wird. Für unsere Zwecke werden aus dieser Vorlesung hier einige zentrale Begriffe wiederholt, soweit sie für das Parsing-Seminar wesentlich sind. Als weiterführende Literatur dazu werden insbesondere folgende Bücher empfohlen:

- Hopcroft/Ullman: *Introduction to Automata Theory, Languages and Computation* [Hopcroft/Ullman 79]
- Aho/Ullman: *The Theory of Parsing, Translation, and Compiling. Volume 1 (Parsing)* [Aho/Ullman 72]
- Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman: *Compilers. Principles, Techniques, and Tools* [Aho/Sethi/Ullman 86]

2 Grundlagen

2.1 Grundbegriffe

Alphabet: Ein *Alphabet* ist eine (endliche) Menge von Symbolen ("Zeichen") und wird meistens mit den Buchstaben V oder Σ abgekürzt (z.B. $\Sigma = \{a, b, c\}$).

Zeichenkette: *Zeichenketten* (Strings) über einem Alphabet sind Folgen von Symbolen aus dem Alphabet. Mit $\Sigma = \{a, b, c\}$ sind bspw. $aa, b, abbac, \ldots$ Zeichenketten über Σ. Wenn x eine Zeichenkette ist, so ist x_i das i-te Zeichen von x (z.B. $x = acb, x_3 = b$).

Konkatention: Die *Zusammensetzung (Konkatenation)* von Strings ergibt sich durch Aneinanderreihung der "Buchstaben" der Strings, z.B. ist die Konkatenaton der Strings $x = aa$ und $y = cba$ der String $xy = aacba$.

Leerer String: Der *leere String* ε ist das neutrale Element bezüglich der Konkatenation, es gilt also $x\varepsilon = \varepsilon x = x$.

Teilstring: Wenn w ein String der Form $w = a_1 \ldots a_n$ ist, so ist jeder String $a_i \ldots a_j$ mit $1 \leq i \leq j \leq n$ ein *Teilstring* von w.

Präfix/Suffix: Als *Präfix* (Anfangsteilwort) eines Strings $a_1 a_2 \ldots a_n$ wird jede Kette $a_1 \ldots a_i$ bezeichnet mit $i \leq n$ (ein *echtes* Präfix liegt vor, wenn $i < n$ ist). Ein *Suffix* (Postfix, Endteilwort) eines Strings ist eine Kette $a_i \ldots a_n$ mit $i \geq 1$ (für ein *echtes* Suffix ist $i > 1$).

Länge eines Strings: Die *Länge eines Strings* $|w|$ ist die Anzahl der Symbole von w. Für $w = abc$ ist $|w| = 3$ (Anmerkung: $|\varepsilon| = 0$).

Spiegelbild: Das *Spiegelbild* w^R eines Strings $w = w_1 w_2 \dots w_n$ ist der String $w^R = w_n w_{n-1} \dots w_1$ (Anmerkung: Strings mit $w = w^R$ heißen *Palindrome*, z.b. *abcba* oder *OTTO*).

Σ^n: Ist Σ ein Alphabet, so wird mit Σ^n die Menge der Strings w über Σ mit $|w| = n$ bezeichnet. Die *Sternhülle* Σ^* über einem Alphabet Σ ist $\cup_{i \geq 0} \Sigma^i$, die *Plushülle* Σ^+ ist $\cup_{i \geq 1} \Sigma^i$. Die Sternhülle ist somit die Menge aller über einem Alphabet erzeugbaren Strings inklusive dem leeren String ε, während der leere String nicht in der Plushülle enthalten ist.

Formale Sprache: Eine *(formale) Sprache* L über einem Alphabet Σ ist eine Teilmenge $L \subset \Sigma^*$ (also eine Teilmenge der Strings beliebiger Länge, die über dem Alphabet Σ gebildet werden können).

Wort: Falls $x \in L$, so ist x ein *Wort* der Sprache L.

Diese Verwendung des Ausdrucks *Wort* kann im Zusammenhang mit dem Parsen natürlicher Sprache zu Verwirrung führen, weil dort ein *Wort* gemäß der obigen Definition einem ganzen Satz entspricht. Ein Symbol in der obigen Definition entspricht einem Wort in einer natürlichen Sprache.

Grammatik: Eine Grammatik G ist ein Viertupel $G = (V, \Sigma, P, S)$ mit

V : die Menge der *Nichtterminalsymbole* ("Variablen" der Grammatik)

Σ : die Menge der *Terminalsymbole* ($V \cap \Sigma = \emptyset$)

P : die Menge der *Produktionen* und

S : das *Startsymbol* der Grammatik ($S \in V$)

bedeuten. Nichtterminalsymbole werden in der Regel mit Großbuchstaben (A, B, C) bezeichnet und Terminalsymbole meistens mit Kleinbuchstaben (a, b, c). Für Symbole, die sowohl Terminal- als auch Nichtterminalsymbole sein können, werden Großbuchstaben vom Ende des Alphabets (X, Y, Z) verwendet, während Wörter einer Sprache (also Ketten von Terminalsymbolen) meistens mit u, w, x bezeichnet werden. Für Ketten von terminalen und nichtterminalen Grammatiksymbolen werden dagegen Kleinbuchstaben aus dem griechischen Alphabet (α, β, γ) verwendet.

Die Produktionen einer Grammatik haben die Form $\alpha \to \beta$. Beide Seiten der Regel bestehen aus einer Folge von Terminal- und Nichtterminalsymbolen. Die linke Seite muss außerdem mindestens ein Nichtterminal enthalten. Es gilt also $\alpha \in (V \cup \Sigma)^* V (V \cup \Sigma)^*$ und $\beta \in (V \cup \Sigma)^*$. (Man schreibt auch $P \subset (V \cup \Sigma)^* V (V \cup \Sigma)^* \times (V \cup \Sigma)^*$). Bei der Angabe einer Grammatik ist es üblich, nur die Menge der Produktionen aufzulisten, da sich daraus V, Σ und S ergeben.

Ableitungsschritt: Ein *Ableitungsschritt* ist die Anwendung einer Produktionsregel auf eine Symbolkette. Seien $u, v \in (V \cup \Sigma)^*$. Dann heißt

- v direkt aus u ableitbar ($u \Rightarrow v$), wenn gilt
  $u = \gamma\alpha\delta, v = \gamma\beta\delta$ und $\alpha \to \beta \in P$
- v in n Schritten aus u ableitbar ($\overset{n}{\Rightarrow}$), wenn gilt
  $\exists u_0 \dots u_n : u = u_0 \Rightarrow u_1 \Rightarrow \dots \Rightarrow u_n = v$

- v aus u (in beliebig vielen Schritten) ableitbar ($\overset{*}{\Rightarrow}$), wenn $u \overset{n}{\Rightarrow} v$ für ein $n \geq 0$.

L(G): Die *von einer Grammatik erzeugte Sprache* $L(G)$ ist die Menge aller Symbolketten $w \in \Sigma^*$, die – ausgehend vom Startsymbol der Grammatik – ableitbar sind: $L(G) = \{w \in \Sigma^* \mid S \overset{*}{\Rightarrow} w\}$.

Satzform: Eine *Satzform* s ist eine Kette von Terminal- oder Nichtterminalsymbolen, die aus dem Startsymbol der Grammatik ableitbar ist ($S \overset{*}{\Rightarrow} s$ mit $s \in (V \cup \Sigma)^*$). Unter anderem sind auch alle Wörter in $L(G)$ in der Menge der Satzformen von G enthalten. Die von einer Grammatik erzeugte Sprache kann dann als die Menge der von der Grammatik erzeugten Satzformen definiert werden, die nur aus Terminalzeichen bestehen.

Äquivalenz von Grammatiken: Zwei Grammatiken sind *äquivalent*, wenn die von ihnen erzeugten Sprachen äquivalent sind: $G \equiv G' \iff L(G) = L(G')$.

Chomsky-Hierarchie: Noam Chomsky definierte 1956 die *Chomsky-Hierarchie* von Grammatiken, in der Grammatiken aufgrund der Gestalt ihrer Produktionsregeln klassifiziert werden:

Typ-0-Grammatiken: Die Regeln von Typ-0-Grammatiken sind nicht weiter eingeschränkt und haben die Form $\alpha \to \beta$ wobei $\alpha, \beta \in (V \cup \Sigma)^*$ gilt und α mindestens ein Nichtterminal enthält.

Typ-1-Grammatiken (kontextsensitiv): Hier gilt zusätzlich für alle Produktionen $\alpha \to \beta \in P : |\alpha| \leq |\beta|$
Außerdem darf die Produktion $S \to \varepsilon$ in P enthalten sein, wobei dann aber S in keiner Produktion auf der rechten Seite auftauchen darf.

Typ-2-Grammatiken (kontextfrei): alle Produktionsregeln haben die Form
$A \to \alpha$ mit $A \in V, \alpha \in (V \cup \Sigma)^*$

Typ-3-Grammatiken (regulär): alle Produktionsregeln haben die Form
$A \to uB$ oder $A \to u$ (rechtslineare Grammatiken) bzw.
$A \to Bu$ oder $A \to u$ (linkslineare Grammatiken)
wobei $u \in \Sigma^*$ (also eventuell auch ε).

Achtung: Oft wird bei der Definition von regulären Grammatiken $u \in \Sigma$ gefordert. Diese Definition weist jedoch dieselbe Mächtigkeit auf!

Aus den Begriffen *kontextsensitive, kontextfreie* bzw. *reguläre Grammatik* ergeben sich die Begriffe *kontextsensitive (kontextfreie, reguläre) Sprache*. Es gilt: Eine Sprache ist vom Typ i, genau dann wenn es eine Typ-i-Grammatik gibt, die diese Sprache erzeugt. Aus den Beschränkungen für den Aufbau der Produktionsregeln ergeben sich Konsequenzen für die "Mächtigkeit" der jeweiligen Sprachklasse. Es läßt sich zeigen, daß die Menge der Typ-i+1-Sprachen immer eine echte Teilmenge der Menge der Typ-i-Sprachen ist (für $i < 3$).

Parsing: Bei der Verarbeitung von (formalen) Sprachen, z.B. zur Analyse eines Quellprogramms oder – im Fall der Verarbeitung "natürlicher" Sprachen – von Eingabesätzen, ist einer der ersten Schritte die Überprüfung, ob das Eingabestring w (also das Quellprogramm bzw. der Satz) ein Wort der Sprache ist, ob also gilt $S \overset{*}{\Rightarrow} w$ *("Wortproblem")*. Dieser Vorgang wird *Parsing* genannt. Dazu ist es notwendig, Algorithmen und Techniken zu entwickeln, die diese Überprüfung leisten (*Parsing-Algorithmen*).

Automat: Grammatiken sind Anweisungen zur Generierung von Wörtern. Automaten *erkennen* Wörter, wobei Automaten im weitesten Sinne durch Grammatik und Eingabe gesteuerte Algorithmen sind, die aufgrund einer Grammatik und einer Eingabezeichenkette entscheiden, ob die Eingabe ein Wort der Sprache ist oder nicht. Dabei kann man beweisen, daß den verschiedenen Typen von Grammatiken auch unterschiedliche Klassen von Automaten zugeordnet werden können:

Grammatiktyp	erkennender Automat
reguläre Grammatiken (Typ 3)	Endliche Automaten
kontextfreie Grammatiken (Typ 2)	Kellerautomaten
kontextsensitive Grammatiken (Typ 1)	Linear-beschränkte Automaten
uneingeschränkte Grammatiken (Typ 0)	Turing-Maschinen

Die einzelnen Automaten sind für diese Einführung nicht weiter von Bedeutung und werden nur der Vollständigkeit halber erwähnt.

In der Informatik finden hauptsächlich reguläre Sprachen und kontextfreie Sprachen (bspw. als Programmiersprachen) Anwendung. In der Computerlinguistik werden (in diesem Fall *natürliche*) Sprachen häufig durch Grammatiken beschrieben, die ein kontextfreies Gerüst und einige darüber hinausgehende Erweiterungen haben. Da jedoch auch beim Parsing natürlicher Sprachen häufig Algorithmen eingesetzt werden, die auf denen zur kontextfreien Analyse beruhen, sind diese Analyseverfahren für die maschinelle Sprachverarbeitung besonders wichtig. Deshalb werden nun die kontextfreien Sprachen genauer betrachtet.

2.2 Eigenschaften von kontextfreien Grammatiken

Kontextfreie Grammatiken können einige wichtige Eigenschaften besitzen, die bei der Formulierung von Parsingalgorithmen wichtig sind:

- G heißt ε-**frei**, wenn es in P keine Produktionen der Form $A \to \varepsilon$ (sog. ε-*Produktionen*) gibt, oder aber die einzige ε-Produktion in P die Produktion $S \to \varepsilon$ ist, wobei aber S in keiner Produktionsregel auf der rechten Seite auftreten darf.
- G heißt **zyklenfrei**, falls es kein Nichtterminal gibt mit $A \overset{+}{\Rightarrow} A$. Es darf also kein Nichtterminal geben, das in einem oder mehreren Schritten (auch unter Anwendung von ε-Produktionen) zu sich selbst abgeleitet werden kann.
- Eine Produktion $A \to B$ mit $A, B \in V$ heißt **Kettenproduktion** oder **Variablenumbennennung**.
- Ein Nichtterminal A heißt **unerreichbar**, wenn es keine Satzform gibt, in der A vorkommt.
- Ein Nichtterminal A heißt **unproduktiv**, wenn es kein $w \in \Sigma^*$ gibt mit $A \overset{*}{\Rightarrow} w$. Es gibt also keine Terminalzeichenkette, die daraus ableitbar ist.
- ein Nichtterminal A heißt **linksrekursiv** bzw. **rechtsrekursiv**, wenn $A \overset{+}{\Rightarrow} A\alpha$ bzw. $A \overset{+}{\Rightarrow} \alpha A$.
- G heißt **linksrekursiv** bzw. **rechtsrekursiv**, wenn es (mindestens) ein links- bzw. rechtsrekursives Nichtterminal in G gibt.
- G ist **in Chomsky-Normalform**, wenn alle Produktionen die Form $A \to BC$ oder $A \to a$ haben. Die einzige erlaubte ε-Produktion ist $S \to \varepsilon$, wobei dann aber S nicht auf der rechten Seite einer Produktion auftauchen darf.

- G ist in **Greibach-Normalform**, wenn P nur Produktionen der folgenden Form enthält: $A \to a\alpha$ mit $a \in \Sigma$ und $\alpha \in (V \cup \Sigma)^*$ (mit der $S \to \varepsilon$-Einschränkung).

Es läßt sich zeigen, daß jede kontextfreie Grammatik G in eine kontextfreie Grammatik G' umgeformt werden kann, so daß gilt:

- $L(G) = L(G')$;
- G' ist ε-frei;
- G' enthält keine Kettenproduktionen;
- G' ist zyklenfrei;
- G' enthält keine unerreichbaren Nichtterminale;
- G' enthält keine unproduktiven Symbole.

Zusätzlich ist es möglich, linksrekursive Nichtterminale in rechtsrekursive Nichtterminale umzuwandeln (und umgekehrt). Außerdem kann jede kontextfreie Grammatik in Chomsky- bzw. Greibach-Normalform transformiert werden. Bei jeder dieser Grammatiktransformationen bleibt die von G erzeugte Sprache gleich (Beweise und Umformungsverfahren siehe einschlägige Literatur).

2.3 Ableitungen für kontextfreie Grammatiken

Ableitung: Eine endliche Folge $S \Rightarrow \delta_1 \Rightarrow \ldots \Rightarrow \delta_n$ von Ableitungsschritten mit $\delta_n \in \Sigma^*$ heißt *Ableitung*. n ist die Länge der Ableitung.

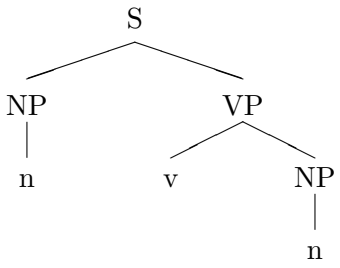
Als **Links- bzw. Rechtsableitungsschritt** wird ein Ableitungsschritt bezeichnet, in dem das am weitesten links (rechts) stehende Nichtterminalsymbol expandiert wird: $\delta A\gamma \Rightarrow \delta\alpha\gamma$ wobei beim Linksableitungsschritt $\delta \in \Sigma^*$, bzw. beim Rechtsableitungsschritt $\gamma \in \Sigma^*$. Eine **Linksableitung ("leftmost derivation",** $\alpha \overset{*}{\Rightarrow}_L \beta$**)** ist eine Ableitung, die nur aus Linksableitungsschritten besteht (entsprechend für Rechtsableitung $\alpha \overset{*}{\Rightarrow}_R \beta$). Eine **Rechtssatzform** ist eine Satzform, die nur aufgrund von Rechtsableitungen entsteht ($S \overset{*}{\Rightarrow}_R s$). Entsprechend ist die Linkssatzform definiert.

Parse, Kontrollwort: Bei der Ableitung einer Zeichenkette aus dem Startsymbol ist es oft wichtig, die Produktionen zu kennen, die bei den einzelnen Ableitungsschritten angewendet wurden. Dazu werden die Produktionen eindeutig markiert (z.B. durchnumeriert). Die Folge der zu den Ableitungsschritten gehörenden Produktionen wird dann *Kontrollwort (Parse)* genannt. Wie oben werden die Begiffe *Rechts-/Linkskontrollwort* (bzw. *Rechts-/Linksparse*) definiert.

Ableitungsbaum: Die Ableitungsschritte eines Wortes werden oft mit Hilfe einer Baumnotation dargestellt. Die Wurzel des Baumes ist dabei das Startsymbol der Grammatik, die Blätter des Baumes sind die Symbole des Wortes (also die Terminalsymbole), das abgeleitet wurde. Zu der Grammatik

S → NP VP
NP → a n
NP → n
VP → v NP

und dem "Wort" *nvn* gibt es also folgenden *Ableitungsbaum (Strukturbaum, Parsebaum)*, durch den die *syntaktische Struktur* des Wortes definiert wird:



Zu beachten ist, daß der Strukturbaum von der genauen Reihenfolge der Ableitungsschritte abstrahiert. So ist in dem Beispielbaum nicht ersichtlich, ob zuerst die linke NP oder die VP expandiert werden soll. Es ist ein Charakteristikum der kontextfreien Grammatiken, daß diese Art von Reihenfolgebeziehung für den Begriff der Ableitbarkeit keine Rolle spielt und damit das besondere Interesse an Ableitungsbäumen (und nicht an Ableitungen selbst) gerechtfertigt ist. Außerdem besteht eine 1-1-Beziehung zwischen Ableitungsbäumen und Linksableitungen (ebenso: Rechtsableitungen).

Eindeutigkeit: Eine Grammatik heißt *eindeutig*, wenn es für kein Wort mehr als eine Analyse (äquivalent: mehr als eine Linksableitung/Rechtsableitung/Parsebaum) gibt. Eine Grammatik, die nicht eindeutig ist, heißt *mehrdeutig* (oder ambig). Eine Sprache L ist *eindeutig*, wenn es eine eindeutige Grammatik G gibt, die L erzeugt (also falls $L(G) = L$).

Parser vs. Erkenner: Algorithmen, die bei der Ableitung die verwendeten Produktionen angeben und somit die Rekonstruktion des Ableitungsbaumes (Parsebaumes) des Wortes erlauben, nennt man *Parser* (im engeren Sinn). Algorithmen, die lediglich ausgeben, ob ein Wort in $L(G)$ ist und die bei der Ableitung verwendeten Produktionen *nicht* ausgeben, nennt man *Erkenner* (in der maschinellen Sprachverarbeitung werden fast ausschließlich Parser verwendet, da die Parsebäume für nachfolgende Analysen wichtig sind).

3 Analyseverfahren für kontextfreie Grammatiken

3.1 Einteilung

Ableitungsorientierte Analyseverfahren verfolgen nur eine mögliche Analyse auf einmal. Wenn die Analyse zu einem bestimmten Zeitpunkt auf mehrere Arten fortgesetzt werden könnte, wählt der Parser eine davon und merkt sich die anderen Möglichkeiten. Wenn er bei der Weitervorfolgung der ausgewählten Analyse dann scheitert, kehrt er zu einer der anderen, noch nicht untersuchten Alternativen zurück. Dieser Vorgang heißt *Backtracking.*

Tabellengesteuerte Analyseverfahren verwenden eine sogenannte Steuertabelle, die zu jedem Zeitpunkt die nächste Aktion des Parsers vorgibt. Diese Verfahren sind in der Informatik von großer Bedeutung, können aber in der Computerlinguistik nur in abgewandelter Form eingesetzt werden.

Chartorientierte Analyseverfahren speichern alle Zwischenergebnisse der Analyse in einer *Chart* genannten Tabelle, wodurch die mehrfache Berechnung identischer Teilergebnisse verhindert werden kann. Sie stellen die gebräuchlichsten Verfahren in der Computerlinguistik dar.

Man kann die Analyseverfahren auch in **Top-Down-** und **Bottom-Up**-Verfahren einteilen. Erstere bauen die syntaktischen Analysen von oben (dem Startsymbol) nach unten (zu den Terminal-

symbolen/Wörtern) auf. Letztere beginnen mit den Wörtern und bauen iterativ immer komplexere syntaktische Strukturen auf, bis eine vollständige Analyse gefunden wurde.

3.2 Ableitungsorientierte Analyse

3.2.1 Top-Down-Verfahren

Die Grundidee bei den Top-Down-Verfahren besteht in der Konstruktion einer Ableitung der Eingabekette durch fortgesetzte Expansion eines Nichtterminals der aktuellen Satzform. Ausgehend vom Startsymbol wird ein Nichtterminal A der aktuellen Satzform $\gamma A \delta$ durch die rechte Seite α einer Produktionsregel $A \rightarrow \alpha$ ersetzt ($\gamma A \delta \Rightarrow \gamma \alpha \delta$). Dabei muß sichergestellt werden, daß α mit dem aktuellen Eingabezeichen "verträglich" ist.

Dieser Algorithmus ist *nichtdeterministisch*, da der Parser zu jedem Zeitpunkt raten muß, welcher von mehreren möglichen Ableitungsschritten am Ende zum Ziel führen wird. Da ein Computer nur deterministische Algorithmen ausführen kann, wird das Raten des nächsten Ableitungsschrittes durch eine systematische Suche ersetzt. Der Parser merkt sich dazu an jedem Entscheidungspunkt die möglichen Alternativen. Wenn ein Ableitungsversuch fehlschlägt, kann der Parser zu dem letzten Entscheidungspunkt zurückkehren, an dem es noch ungeprüfte Alternativen gibt und mit der nächsten Alternative fortfahren (Backtracking).

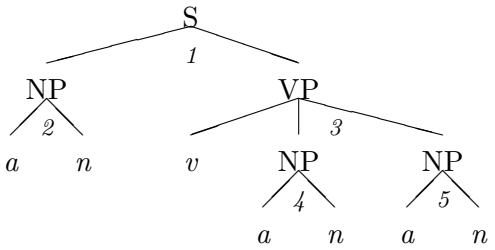
Wegen der 1:1-Beziehung zwischen Linksableitungen und Ableitungbäumen genügt es, wenn der Parser nur Linksableitungen untersucht, indem er jeweils das am weitesten links stehende Nichtterminal einer Satzform expandiert.

Beispiel: Für die Grammatik

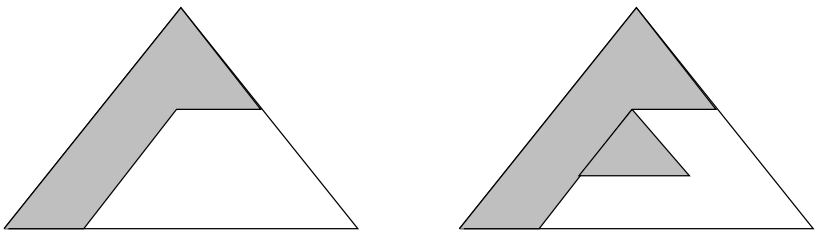
$$
\begin{aligned}
\text{S} &\rightarrow \text{NP VP} \\
\text{NP} &\rightarrow a\ n \\
\text{VP} &\rightarrow v\ \text{NP NP} \\
\text{VP} &\rightarrow v\ \text{NP} \\
\text{VP} &\rightarrow v
\end{aligned}
$$

und die Eingabe $a\ n\ v\ a\ n\ a\ n$ ergibt sich folgende Linksableitung und damit der folgende Strukturbaum:

$$
\begin{aligned}
\text{S} &\Rightarrow \text{NP VP} \\
&\Rightarrow a\ n\ \text{VP} \\
&\Rightarrow a\ n\ v\ \text{NP NP} \\
&\Rightarrow a\ n\ v\ a\ n\ \text{NP} \\
&\Rightarrow a\ n\ v\ a\ n\ a\ n
\end{aligned}
$$



Im Strukturbaum sind die Nummern der jeweiligen Ableitungsschritte durch kursive Ziffern dargestellt. Allgemein ergibt sich bei der Top-Down-Analyse (mit Linksableitung) folgendes Schema:



Formale Charakterisierung des Top-Down-Erkenners:

Eine **Konfiguration** ist ein Paar (α, r), mit $\alpha \in (V \cup \Sigma)^*, r \in \Sigma^*$, wobei α für die Liste der erwarteten Symbole steht, r für den Rest der Eingabe.

Die **Anfangskonfiguration** ist (S, w), wobei S das Startsymbol ist und w das Eingabewort. Mögliche **Konfigurationsübergänge** sind:

- Von $(a\alpha, ar)$ nach (α, r), d.h. ein erwartetes Terminalsymbol wird "konsumiert"; oder
- von $(A\beta, r)$ nach $(\alpha\beta, r)$, falls $A \to \alpha \in P$, d.h. ein erwartetes Nichtterminalsymbol wird durch die rechte Seite einer Produktion ersetzt (expandiert). Hier geht ein, daß immer das am weitesten links stehende Nichtterminal ersetzt wird.

Gelangt der Erkennungsalgorithmus in die Konfiguration $(\varepsilon, \varepsilon)$, so ist die gesamte Eingabe verarbeitet und es wird kein weiteres Symbol erwartet, also ist die Eingabe akzeptiert. Die zugehörige Ableitung läßt sich direkt aus den durchlaufenen Schritten rekonstruieren. Es ist daher sehr leicht, den Erkenner zu einem Parser zu erweitern.

Das angegebene Schema ist noch kein Algorithmus, denn für ein erwartetes Nichtterminalsymbol sind in der Regel mehrere Produktionen vorhanden, also auch mehrere Folgekonfigurationen möglich. Um einen "wirklichen" Algorithmus zu bekommen, muß der sich daraus ergebende Nichtdeterminismus in konkrete (deterministische) Verarbeitungsschritte aufgelöst werden, indem eine geeignete Suchstrategie angegeben wird. Beispielsweise kann der Suchraum via Backtracking (Depth-First-Suche) durchwandert werden.

Für das obige Beispiel ergibt sich die Konfigurationsfolge

(S ,	$a\ n\ v\ a\ n$)	
(NP VP ,	$a\ n\ v\ a\ n$)	
(a n VP ,	$a\ n\ v\ a\ n$)	
(n VP ,	$n\ v\ a\ n$)	
(VP ,	$v\ a\ n$)	(VP , $v\ a\ n$)
(v NP NP ,	$v\ a\ n$)	(v NP , $v\ a\ n$)
(NP NP ,	$a\ n$)	(NP , $a\ n$)
(a n NP ,	$a\ n$)	(a n , $a\ n$)
(n NP ,	n)	(n , n)
(NP ,	ε)	(ε , ε)

In diesem Beispiel wählt der Parser zunächst die VP-Regel für ditransitive Verben und gerät in eine Sackgasse. Mit Backtracking wird doch noch die korrekte Analyse gefunden.

Auf die Liste der erwarteten Symbole wird stets nur am linken Ende zugegriffen. Sie läßt sich deshalb durch einen Stapelspeicher (Stack) implementieren.

Probleme des Top-Down-Erkenners:

Linksrekursion: Wenn Ableitungen der Form $A \overset{+}{\Rightarrow} A\beta$ möglich sind, so kann der Algorithmus von $(A\alpha, r)$ via $(A\beta\alpha, r)$ zu beliebigen $(A\beta^n\alpha, r)$ gelangen, ohne jemals weitere Eingabezeichen zu konsumieren. Der Algorithmus terminiert in diesem Falle nicht (d.h. er läuft endlos).

Regelauswahl: Sind mehrere alternative Expansionen möglich, so muß eine Auswahl getroffen werden, ohne daß der relevante Teil der Eingabe zuvor betrachtet werden konnte.

Ineffizienz: Der Algorithmus hat schlimmstenfalls eine exponentielle Laufzeit, weil beim Backtracking identische Teilanalysen immer wieder aufs Neue berechnen müssen. Die Rechenzeit steigt also sehr schnell mit der Länge der Eingabe an.

3.2.2 Bottom-Up-Verfahren

Oben wurde bereits der Begriff der *Expansion* eines Nichtterminalsymbols A in eine Zeichenkette α aufgrund einer Produktion $A \to \alpha$ verwendet. Diese Expansion entspricht einem *Top-Down-Vorgehen* bei der Bestimmung der Ableitung eines Wortes aus $L(G)$. Das Gegenteil der Expansion ist die *Reduktion*: Ausgehend von einer Satzform wird durch die "umgekehrte" Anwendung von Produktionsregeln versucht, eine Satzform bis zum Startsymbol der Grammatik zu reduzieren *(Bottom-Up-Analyse)*. Die Bottom-up-Analyse ist nur auf zyklenfreie Grammatiken ohne ε-Produktionen anwendbar, da sie nur bei diesen immer terminiert.

Bei dem Versuch, einen Reduktionsschritt durchzuführen, muß klar sein, welche Teilkette einer Satzform reduziert werden kann. Generell kann jede Teilkette einer Satzform zur Reduktion herangezogen werden. In der Regel aber wird die Satzform als Rechtssatzform betrachtet und versucht, durch die Reduktionsschritte eine "umgekehrte Rechtsableitung" vorzunehmen. Dazu wird der *Henkel (Handle)* der Rechtssatzform $s = \delta\alpha u$ als Paar $(i, A \to \alpha)$ definiert, wobei $S \overset{*}{\Rightarrow} \delta A u \Rightarrow \delta\alpha u = s$ und $|\delta| = i$ ist (da s eine Rechtssatzform ist, ist $u \in \Sigma^*$). Wenn die Rechtsableitung von $S \overset{*}{\Rightarrow} s$ betrachtet wird, entspricht der Henkel offensichtlich der rechten Seite der zuletzt angewandten Produktionsregel. Wenn der Henkel einer Rechtssatzform aber bekannt ist, kann der Reduktionsschritt vorgenommen werden und der Henkel zur linken Seite der entsprechenden Produktionsregel reduziert werden. Die Bestimmung einer Linksreduktion für ein Wort $w \in L(G)$ kann also auf fortgesetzte Bestimmung eines Henkels und anschließende Reduktion zurückgeführt werden.

Die Idee bei der Bottom-Up-Analyse ist also, ausgehend vom Eingabewort *Henkel* zu bestimmen und mit Hilfe dieser Henkel Reduktionen durchzuführen. Der Parser sammelt dazu die gelesene Eingabe in einer Liste auf. Befindet sich am Ende der Liste die rechte Seite einer Produktion, so kann diese durch die linke Seite der Produktion ersetzt werden (Reduktionsschritt). Die Eingabe wird genau dann akzeptiert, wenn sich die gelesenen Symbole zum Startsymbol reduzieren lassen.

Formale Charakterisierung des Bottom-Up-Erkenners:

Eine **Konfiguration** ist ein Paar (α, r), mit $\alpha \in (V \cup \Sigma)^*, r \in \Sigma^*$, wobei α für die Liste der bereits gefundenen Symbole steht, r für den Rest der Eingabe.

Die **Anfangskonfiguration** ist (ε, w), wobei w die Eingabe ist.

Die möglichen Übergänge zu einer **Folgekonfiguration** sind:

- von (α, ar) nach $(\alpha a, r)$, d.h. ein Terminalsymbol wird aus der Eingabe auf die Liste der erkannten Symbole geschoben (**Shift**-Aktion); oder
- von $(\beta\alpha, r)$ nach $(\beta A, r)$, falls $A \to \alpha \in P$, d.h. die gefundene rechte Seite einer Produktion am Ende der Liste (Henkel) wird durch die linke Seite ersetzt (**Reduce**-Aktion).

Gelangt der Erkennungsalgorithmus in die Konfiguration (S, ε), wobei S das Startsymbol ist, so ist die gesamte Eingabe zum Startsymbol reduziert, kann also akzeptiert werden. Wegen der beiden möglichen Aktionen *Shift* und *Reduce* wird ein solcher Parser auch als **Shift-Reduce-Parser** bezeichnet.

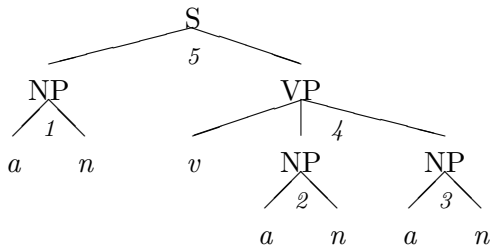
Für das obige Beispiel ergibt sich beispielsweise die Folge von Konfigurationen

$$
\begin{array}{rl}
(\varepsilon\ , & a\ n\ v\ a\ n\ a\ n) \\
(a\ , & n\ v\ a\ n\ a\ n) \\
(a\ n\ , & v\ a\ n\ a\ n) \\
(\mathrm{NP}\ , & v\ a\ n\ a\ n) \\
(\mathrm{NP}\ v\ , & a\ n\ a\ n) \\
(\mathrm{NP}\ v\ a\ , & n\ a\ n) \\
(\mathrm{NP}\ v\ a\ n\ , & a\ n) \\
(\mathrm{NP}\ v\ \mathrm{NP}\ , & a\ n) \\
(\mathrm{NP\ VP}\ , & a\ n) \\
(\mathrm{S}\ , & a\ n) \\
(\mathrm{S}\ a\ , & n) \\
(\mathrm{S}\ a\ n\ , & \varepsilon) \\
(\mathrm{S\ NP}\ , & \varepsilon)
\end{array}
\qquad
\begin{array}{rl}
(\mathrm{NP}\ v\ \mathrm{NP}\ , & a\ n) \\
(\mathrm{NP}\ v\ \mathrm{NP}\ a\ , & n) \\
(\mathrm{NP}\ v\ \mathrm{NP}\ a\ n\ , & \varepsilon) \\
(\mathrm{NP}\ v\ \mathrm{NP\ NP}\ , & \varepsilon) \\
(\mathrm{NP\ VP}\ , & \varepsilon) \\
(\mathrm{S}\ , & \varepsilon)
\end{array}
$$

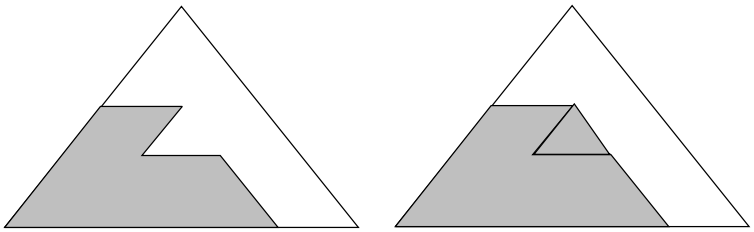
Wiederum wird Backtracking eingesetzt, um aus Sackgassen herauszufinden.

Die Bottom-Up-Analyse entspricht einer rückwärts durchlaufenen Rechtsableitung (Linksreduktion), wodurch sich zwar der gleiche Strukturbaum wie bei der Linksableitung ergibt, die Produktionen aber in einer anderen Reihenfolge angewendet werden:

$$
\begin{aligned}
a\ n\ v\ a\ n\ a\ n &\Leftarrow \mathrm{NP}\ v\ a\ n\ a\ n \\
&\Leftarrow \mathrm{NP}\ v\ \mathrm{NP}\ a\ n \\
&\Leftarrow \mathrm{NP}\ v\ \mathrm{NP\ NP} \\
&\Leftarrow \mathrm{NP\ VP} \\
&\Leftarrow \mathrm{S}
\end{aligned}
$$



Allgemeines Schema:



Auch hier läßt sich ggf. aus den durchlaufenen Schritten direkt die zugehörige Ableitung rekonstruieren, wodurch die Konstruktion eines Parse-Algorithmus aus dem Erkenner erleichtert wird.

Auf die Liste der gefundenen Symbole wird stets nur am rechten Ende (genauer: nahe des rechten Endes) zugegriffen. Auch sie entspricht somit einem Stack.

Das oben zum Thema Nichtdeterminismus Gesagte gilt hier sinngemäß, da (bei nichtleerer Eingabe) stets eine Shift-Aktion, oft aber gleichzeitig auch eine Reduktion durchgeführt werden kann (sog. *shift-reduce-Konflikt*). Unter Umständen sind auch mehrere Reduktionen gleichzeitig möglich (*reduce-reduce-Konflikt*). Auch hier kann man den Suchraum mit Backtracking durchwandern.

Probleme des Bottom-Up-Erkenners:

Zyklen und ε-Produktionen: Falls die Grammatik Zyklen oder ε-Produktionen enthält, wird der Suchraum unendlich groß, d.h. das Verfahren terminiert in diesem Fall nicht.

"Blind Shift": Der Parser kann immer eine Shift-Aktion ausführen, auch wenn sich auf dem Stack schon längst unbrauchbares Material angesammelt hat. Der Suchraum enthält deshalb sehr viele Irrwege, die beim Backtracking alle durchlaufen werden müssen.

Effizienz: Wie beim Top-Down-Parser führt das systematische Durchprobieren aller Möglichkeiten mittels Backtracking zu einer exponentiellen Laufzeit.

Diese Nachteile lassen sich teilweise dadurch beheben, daß der Inhalt des Stapels mit zur Entscheidung über die nächste durchzuführende Aktion herangezogen wird. Durch die Verwendung sog. Parser-Steuertabellen kann sichergestellt werden, daß die Symbole auf dem Stack tatsächlich ein gültiges Präfix einer Rechtssatzform sind. Mit einigen Grammatiken (den sog. LR-Grammatiken) ist dadurch eine deterministische Analyse mit linearem Zeitbedarf möglich (LR(k)-Parser). Diese Verfahren eignen sich vor allem für Programmiersprachen.

3.3 Tabellengesteuerte Analyseverfahren

3.3.1 Informelle Beschreibung

Die ableitungsorientierten Analyseverfahren lassen sich verbessern, indem vor jeder Aktion geprüft wird, ob sie überhaupt Aussicht auf Erfolg hat. Ein Top-Down-Parser für die Grammatik

S → if E then S else S
S → while E do S
S → s
E → e

beispielsweise muß nur das nächste noch nicht gelesene Symbol in der Konfiguration prüfen, um eindeutig entscheiden zu können, welche Regel als nächstes anzuwenden ist. Diese Entscheidungsregeln werden vorab berechnet und in einer Tabelle gespeichert, die jedem Paar bestehend aus einem Nichtterminal und einem Vorschauzeichen eine Regel bzw. die Aktion 'error' zuordnet. Tabelle 1 zeigt eine LL(1)-Steuertabelle für die obige Grammatik. Leere Fehler entsprechen einer 'error'-Aktion.

Nicht-terminal	Vorschausymbol						
	if	while	s	e	then	else	do
S	S → if E then S else S	S → while E do S	S → s				
E				E → e			

Tabelle 1: LL(1)-Steuertabelle

Ein solcher Top-Down-Parser heißt **LL(1)-Parser**, weil er die Eingabe von links nach rechts abarbeitet, dabei eine Linksableitung erzeugt und 1 weiteres Symbol der Eingabe prüft. Ein Parser, der k weitere Symbole (k=1,2,...) betrachtet, heißt entsprechend LL(k)-Parser. Grammatiken, die ein LL(k)-Parser deterministisch parsen kann, heißen **LL(k)-Grammatiken**. Ein LL(k)-Parser kann seine Eingabe in linearer Zeit abarbeiten.

Nicht alle kontextfreien Grammatiken sind jedoch LL(k)-Grammatiken. Die folgende Grammatik beispielsweise besitzt diese Eigenschaft nicht:

S → if E then (S) else (S)
S → if E then (S)
S → while E do (S)
S → s
E → e

Ein LL(k)-Parser müsste bis zu einem möglicherweise nachfolgenden *else* schauen können, um sich zwischen Regel 1 und Regel 2 entscheiden zu können. Wegen des vorhergehenden Nichtterminals S ist es jedoch möglich, daß das *else* beliebig weit entfernt und somit nicht in den nächsten k Symbolen enthalten ist. Kein LL(k)-Parser kann daher diese Grammatik deterministisch parsen.

Analog zu den deterministischen Top-Down-Parsern gibt es auch **deterministische Bottom-Up-Parser**. Ein Bottom-Up-Parser für die obige Grammatik kann eine Eingabe der Form *if e then (s) else (s)* deterministisch parsen, indem er allein den Stapelinhalt und das nächste Symbol prüft:

1	(ε , *if e then (s) else (s)*)
2	(*if* , *e then (s) else (s)*)
3	(*if e* , *then (s) else (s)*)
4	(*if E* , *then (s) else (s)*)
5	(*if E then* , *(s) else (s)*)
6	(*if E then (* , *s) else (s)*)
7	(*if E then (s* , *) else (s)*)
8	(*if E then (S* , *) else (s)*)
9	(*if E then (S)* , *else (s)*)
10	(*if E then (S) else* , *(s)*)
11	(*if E then (S) else (* , *s)*)
12	(*if E then (S) else (s* , *)*)
13	(*if E then (S) else (S* , *)*)
14	(*if E then (S) else (S)* , ε)
15	(*S* , ε)

Im 10. Schritt hätte der Parser auch die Möglichkeit, den Stapelinhalt mit Regel 1 zu S zu reduzieren. Das nachfolgende *else* signalisiert dem Parser jedoch, daß er eine Shift-Operation ausführen muß, um später mit Regel 2 reduzieren zu können.

Ein Bottom-Up-Parser, der k nachfolgende Symbole zur Entscheidung heranzieht, heißt **LR(k)-Parser**, weil er die Eingabe von links nach rechts abarbeitet und eine Rechtsableitung erzeugt.

Grammatiken, die von einem LR(k)-Parser deterministisch geparst werden können, heißen **LR(k)-Grammatiken**. Sie sind echte Obermengen der LL(k)-Grammatiken (für gleiches k). Wiederum sind nicht alle kontext-freien Grammatiken LR(k)-Grammatiken. Insbesondere ist keine mehrdeutige Grammatik eine LR(k)-Grammatik, weil ein deterministischer Parser immer nur eine Analyse liefern kann.

Die folgende Grammatik kann den String *if e then if e then s else s* auf zwei Arten generieren und ist somit mehrdeutig und keine LR(k)-Grammatik:

(1) S → if E then S else S
(2) S → if E then S
(3) S → s
(4) E → e

Damit ein LR(k)-Parser seine Eingabe in linearer Zeit abarbeiten kann, muß die Untersuchung des

Stapelinhaltes in konstanter Zeit erfolgen. Dies geschieht mit Hilfe von LR-Steuertabellen, deren Konstruktion in den folgenden Kapiteln erläutert wird.

3.3.2 LL(k)-Analyse

Ein ableitungsorientierter Top-Down-Parser hat genau dann eine Entscheidungsfreiheit, wenn zu einer Konfiguration $(A\beta, r)$ mehrere Grammatikregeln $A \to \alpha_1, \ldots, A \to \alpha_n$ existieren. Ein deterministischer Parser muß nun entscheiden, welche der möglichen Folgekonfigurationen $(\alpha_1\beta, r), \ldots, (\alpha_n\beta, r)$ zu einer Endkonfiguration führen wird, d.h. für welches i gilt: $\alpha_i\beta \stackrel{*}{\Rightarrow} r$.

Ableitungsklasse: Die Menge aller möglichen (d.h. von einer Anfangskonfiguration aus erreichbaren) Konfigurationen $(A\beta, r)$, bei denen die Ableitung $A \to \alpha$ zum Erfolg führt, wird als *Ableitungsklasse* $\mathcal{A}_{A\to\alpha}$ bezeichnet.

$$\mathcal{A}_{A\to\alpha} = \{(A\beta, r) \mid \beta \in (V \cup \Sigma)^*, r \in \Sigma^*, S \stackrel{*}{\Rightarrow}_L lA\beta \text{ und } \alpha\beta \stackrel{*}{\Rightarrow} r \text{ für ein } l \in \Sigma^*\}$$

Eine Grammatik ist genau dann eindeutig, wenn die zu verschiedenen Produktionen gehörigen Ableitungsklassen disjunkt sind. In diesem Falle kann ein Top-Down-Parser die Eingabe deterministisch abarbeiten, indem er für jede erreichte Konfiguration die Ableitungsklasse berechnet und die zugehörige Aktion ausführt. Die Bestimmung der Ableitungsklasse ist jedoch äquivalent zum Worterkennungsproblem und damit genauso aufwendig wie das Parsen:

$$w \in L(G) \Longleftrightarrow (S, w) \in \mathcal{A}_{S\to\alpha} \text{ mit } A \to \alpha \in P$$

Daher werden Obermengen der Ableitungsklassen verwendet, bei denen die Zugehörigkeit einer Konfiguration in konstanter Zeit berechnet werden kann. Falls diese Obermengen disjunkt sind, ist deterministisches Parsen möglich.

k-Präfix: Das *k-Präfix* ${}_k(x)$ eines Strings x der Länge $l > k$ besteht aus seinen ersten k Zeichen. Ist der String kürzer, so ist er mit seinem k-Präfix identisch.

LL(k)-Obermengen: Die *LL(k)-Obermengen* sind wie folgt definiert:

$$\Omega^k_{A\to\alpha} = \{(A\beta, r) \mid A \in V, \beta \in (V \cup \Sigma)^*, r \in \Sigma^*, \exists s \in \Sigma^* : (A\beta, s) \in \mathcal{A}_{A\to\alpha}, {}_k(r) = {}_k(s)\}$$

LL(k)-Grammatik: Eine *LL(k)-Grammatik* ist eine Grammatik bei der alle LL(k)-Obermengen paarweise disjunkt sind. Bei einer LL(k)-Grammatik genügt es, die nächsten k Symbole in der Eingabe zu betrachten, um die Ableitungsklasse einer Konfiguration eindeutig bestimmen zu können.

Es werden nun zwei Funktionen *First* und *Follow* definiert, die bei der Erstellung von Parser-Steuertabellen nützlich sind.

First-Mengen: $First(\alpha)$ ist die Menge aller Terminale, mit denen ein aus α abgeleiteter String beginnen kann. Falls der leere String aus α abgeleitet werden kann, so ist auch ε in $First(\alpha)$.

$First : (V \cup \Sigma)^* \longrightarrow \wp(\Sigma \cup \{\varepsilon\})$

$$First(\alpha) = \begin{cases} \{a \in \Sigma \mid \alpha \stackrel{*}{\Rightarrow} a\beta, \beta \in \Sigma^*\} \cup \{\varepsilon\}, & \text{falls } \alpha \stackrel{*}{\Rightarrow} \varepsilon \\ \{a \in \Sigma \mid \alpha \stackrel{*}{\Rightarrow} a\beta, \beta \in \Sigma^*\}, & \text{sonst} \end{cases}$$

Follow-Mengen: $Follow(A)$ ist die Menge aller Terminale, die in einer Satzform direkt neben dem Nichtterminal A stehen können. Wenn A am rechten Rand einer Satzform stehen kann, so ist auch \$ in $Follow(A)$. Das Symbol \$ ist ein neues Symbol, das nicht in Σ enthalten ist, und das an das Ende der Eingabe angehängt wird, um das Eingabeende eindeutig zu markieren.

$Follow : V \longrightarrow \wp(\Sigma \cup \{\$\})$

$$Follow(A) = \begin{cases} \{a \in \Sigma \mid S \stackrel{*}{\Rightarrow} \alpha A a \beta; \alpha, \beta \in (V \cup \Sigma)^*\} \cup \{\$\}, & \text{falls } S \stackrel{*}{\Rightarrow} \alpha A \\ \{a \in \Sigma \mid S \stackrel{*}{\Rightarrow} \alpha A a \beta; \alpha, \beta \in (V \cup \Sigma)^*\}, & \text{sonst} \end{cases}$$

Anhand der First- und Follow-Mengen einer Grammatik läßt sich eine LL(1)-Steuertabelle wie folgt konstruieren:

1. Wenn $A \to \alpha \in P$ und $a \in First(\alpha)$ mit $a \neq \varepsilon$, dann $A \to \alpha \in Tab[A, a]$
2. Wenn $\varepsilon \in First(\alpha)$ und $b \in Follow(A)$, dann $A \to \alpha \in Tab[A, b]$ (b kann \$ sein)
3. Falls $Tab[A, a]$ leer ist, so setze $Tab[A, a] = \{error\}$

Wenn die Tabelle keine Mehrfacheinträge enthält, so ist die Grammatik in LL(1).

3.3.3 LR(k)-Analyse

Analog zur Top-Down-Analyse lassen sich die möglichen Konfigurationen eines Bottom-Up-Parsers anhand der Aktionen, die zu einem Endzustand führen, in Klassen einteilen.

Reduktionsklasse: Die Menge aller möglichen Konfigurationen $(\beta\alpha, r)$, bei denen die Reduktion $\alpha \to A$ zum Erfolg führt, wird als *Reduktionsklasse* $\mathcal{R}_{A\to\alpha}$ bezeichnet.

$$\mathcal{R}_{A\to\alpha} = \{(\beta\alpha, r) \mid \beta \in (V \cup \Sigma)^*, r \in \Sigma^*, S \stackrel{*}{\Rightarrow}_R \beta A r \Rightarrow \beta\alpha r\}$$

Shiftklasse: Die Menge aller möglichen Konfigurationen $(\beta\alpha, r)$, bei denen eine Shift-Operation zum Erfolg führt, wird als *Shift-Klasse* $\mathcal{R}_{sh}$ bezeichnet.

$$\mathcal{R}_{sh} = \{(\beta, zr) \mid \beta \in (V \cup \Sigma)^*, z \in \Sigma^+, r \in \Sigma^*, \exists A \to \alpha \in P : (\beta z, r) \in \mathcal{R}_{A\to\alpha}\}$$

Wieder gilt, daß eine Grammatik genau dann eindeutig ist, wenn diese Klassen paarweise disjunkt sind. Doch auch hier ist die Berechnung der Klasse, der eine gegebene Konfiguration angehört, im allgemeinen Fall zu aufwendig. Also wird wieder versucht, disjunkte Obermengen der Reduktions- und Shiftklassen zu finden, für die die Zuordnung effizient (d.h. in konstanter Zeit) entscheidbar ist.

LR(k)-Obermenge: Die *LR(k)-Obermengen* sind wie folgt definiert:

$$\mathcal{O}_p^k = \{(\gamma, w) \mid \gamma \in (V \cup \Sigma)^*, w \in \Sigma^*, \exists (\gamma, r) \in \mathcal{R}_p,\ {}_k(w) = {}_k(r)\}$$

LR(k)-Grammatik: Eine Grammatik ist eine LR(k)-Grammatik, wenn es keine zyklische Ableitung des Startsymbols in sich selbst gibt und ihre LR(k)-Obermengen disjunkt sind.

Diese Definition der Obermengen ermöglicht ein effizientes Entscheidungsverfahren. Vom Rest der Eingabe sind hierbei nur k Vorschauzeichen relevant. Die möglichen Stapelinhalte können durch

einen finiten Automaten beschrieben werden. Daher ist nur ein konstanter Aufwand (Tabellenzugriff) nötig, um einer Konfiguration ihre Obermenge zuzuordnen. Die Zustände des o.g. finiten Automaten werden mit auf dem Stapel abgelegt. Tatsächlich würde es genügen, nur die Zustände auf den Stapel zu legen, da sich die Stapelsymbole daraus ableiten lassen.

LR(k)-Steuertabelle: Eine LR(k)-Steuertabelle ist ein Tripel $(Q, Action, Goto)$, wobei Q eine Menge von Zuständen und *Action* und *Goto* Funktionen sind.

$$\begin{aligned} Action: &\quad Q \times (\Sigma \cup \{\$\})^k \longrightarrow \{reduce\} \times P \cup \{shift\} \times Q \cup \{error, accept\} \\ Goto: &\quad Q \times V \longrightarrow Q \cup \{error\} \end{aligned}$$

Die o.g. Definition der LR(k)-Obermengen liefert Tabellen mit sehr vielen Zuständen. Um den Platzbedarf zu verringern, werden die zugrundeliegenden Obermengen weiter vergröbert, indem die Beziehung zwischen Zuständen und Vorschauzeichen weniger genau berücksichtigt wird. Es entsteht eine Hierarchie von Steuertabellen und Parse-Verfahren unterschiedlicher Genauigkeit: LR(1), LALR, SLR, LR(0)

Tabelle 2 zeigt eine LR(1)-Steuertabelle für die mehrdeutige Grammatik

(1) S → if E then S else S
(2) S → if E then S
(3) S → s
(4) E → e

Die Mehrdeutigkeit drückt sich in dem Mehrfacheintrag im Feld $Action[6, else]$ aus. Um Platz zu sparen, wurden die Aktionen 'shift', 'reduce' und 'accept' zu 's', 'r' und 'acc' abgekürzt. Die leeren Felder enthalten 'error'-Aktionen.

	Action						Goto	
Zustand	if	s	e	then	else	$	S	E
0	s1	s2					9	
1			s3					4
2					r3	r3		
3				r4				
4				s5				
5	s1	s2					6	
6					s7,r2	r2		
7	s1	s2					8	
8					r1	r1		
9						acc		

Tabelle 2: LR(1)-Steuertabelle

3.3.4 Arbeitsweise des xLR(1)-Erkenners

Gegeben sei die Anfangskonfiguration $(s_0, w\$)$, wobei s_0 der Startzustand und w das zu analysierende Wort ist.

Der Kern des Parsers besteht aus der folgenden Programmschleife:

repeat
  let $(s_0X_1s_1...X_ms_m, a_ia_{i+1}...a_n\$)$ be the current configuration
  $A := Action[s_m, a_i]$
  if A = 'shift s' then
    enter the configuration $(s_0X_1s_1...X_ms_ma_is, a_{i+1}...a_n\$)$
  else if A = 'reduce $A \to \alpha$' then
    enter the configuration $(s_0X_1s_1...X_{m-r}s_{m-r}As, a_ia_{i+1}...a_n\$)$
      where $r = |\alpha|$ and $s = Goto[s_{m-r}, A]$
until A = 'accept' or A = 'error'

Die Eingabe ist genau dann wohlgeformt, wenn die Schleife mit `A='accept'` verlassen wird. Da der Erkenner eine umgekehrte Rechtsableitung durchführt, genügt es für die Erweiterung zu einem Parser, die verwendeten Grammatikregeln mitzuprotokollieren.

3.3.5 Erstellung der Parser-Tabelle

Wir werden hier nur die Generierung von LR(0)- und SLR-Tabellen vorstellen. Für die Erstellung von LR(1)- und LALR-Tabellen wird auf die Literatur verwiesen [Aho/Ullman 72, Aho/Sethi/Ullman 86, Mayer 78].

Geteilte Produktion: Wenn $A \to \alpha\beta \in P$, so heißt $A \to \alpha \cdot \beta$ *geteilte Produktion* (Item). Die Menge aller geteilten Produktionen wird mit *Item* bezeichnet.

Lebensfähiges Präfix: Ein Präfix γ einer Rechtssatzform heißt *lebensfähiges Präfix* (lfP), falls es nicht nach rechts über den Henkel der Rechtssatzform hinausreicht, d.h. es existiert eine Ableitung $S \overset{*}{\Rightarrow}_R \alpha Ar \Rightarrow \alpha\beta r$, wobei γ Präfix von $\alpha\beta$ ist.

Eine Konfiguration (p, r) kann in einem Shift-Reduce-Parser nur dann zum Erfolg führen, wenn p ein lebensfähiges Präfix ist. Den lebensfähigen Präfixen lassen sich geteilte Produktionen zuordnen, die Auskunft darüber geben, welche Produktionen nach der Verarbeitung von p "aktiv" sind und welcher Teil der rechten Seite jeweils abgearbeitet wurde.

Gültige Items: Die *gültigen Items* eines lfPes p sind wie folgt definiert:

$$ValidItems(p) = \{A \to \alpha \cdot \beta \mid S \overset{*}{\Rightarrow}_R \gamma Ar \Rightarrow \gamma\alpha\beta r, p = \gamma\alpha\}$$

Zur Erstellung der Parser-Tabelle wird die Grammatik G zunächst um ein spezielles Startsymbol S' und um die Produktion $S' \to S$ zu G' erweitert.

Durch die nachfolgende Itemmengen-Konstruktion wird ein finiter Automat berechnet, der jedem lfP die Menge der gültigen Items zuordnet. Sie benutzt folgende Hilfsfunktionen auf Grammatiksymbolen und Itemmengen:

$Closure : \wp(Item) \longrightarrow \wp(Item)$
$Closure(I) =$ kleinste Menge mit
  1. $Closure(I) \supseteq I$
  2. Wenn $A \to \alpha \cdot B\beta \in Closure(I)$ und $B \to \gamma \in P$, so auch $B \to \cdot\gamma \in Closure(I)$

$Goto : \wp(Item) \times V \cup \Sigma \longrightarrow \wp(Item)$
$Goto(I, X) = Closure(\{A \to \alpha X \cdot \beta \mid A \to \alpha \cdot X\beta \in I\})$

Algorithmus zur Berechnung von LR(0)-Itemmengen:

```
procedure Items(G')
begin
  C := {Closure({S' → ·S})};
  repeat
    for each I ∈ C and each X ∈ V ∪ Σ such that Goto(I, X) ≠ {} do
      add Goto(I, X) to C
  until nothing else can be added to C
end
```

Die aufgefundenen Itemmengen stellen die möglichen Zustände des LR-Parsers dar. Die Parser-Tabelle wird nun wie folgt konstruiert:

Sei $C = \{I_0, I_1, ..., I_n\}$ das im vorherigen Schritt gefundene System von Itemmengen. Die Zustände des Parsers werden mit $0, 1, ..., n$ bezeichnet, wobei der Zustand i der Itemmenge I_i entspricht. Die Parser-Aktionen und die Sprungfunktion ergeben sich wie folgt:

1. Wenn $A \to \alpha \cdot a\beta \in I_i$ und $Goto(I_i, a) = I_j$ mit $a \in \Sigma$, dann $Action[i, a] =$*'shift j'*.
2. Wenn $A \to \alpha\cdot \in I_i$, dann $Action[i, a] =$*'reduce* $A \to \alpha$*'* für alle $a \in Follow(A)$ (SLR-Parser) bzw. $a \in \Sigma$ (LR(0)-Parser).
3. Wenn $S' \to \alpha\cdot \in I_i$, dann $Action[i, \$] =$*'accept'*.
4. Wenn $Goto(I_i, A) = I_j$, dann $Goto[i, A] = j$.
5. Alle Felder, die in Schritt 1. bis 4. nicht definiert werden, erhalten den Wert $'error'$.
6. Der Anfangszustand des Parsers ergibt sich aus der Itemmenge, die $S' \to \cdot S$ enthält.

Wenn dabei einem Feld der Action-Tabelle mehrere Werte zugewiesen werden, so liegt ein Konflikt vor und es handelt sich nicht um eine SLR- bzw. LR(0)-Grammatik.

Der LR(0)-Parser wendet eine Reduktion $A \to \alpha$ immer an, wenn der daraus resultierende Stapelinhalt noch ein lfP bleibt, der SLR-Parser nur dann, wenn das Vorschauzeichen ein möglicher Nachfolger von A ist.

3.3.6 Tomita-Parser

Die xLR-Parser eignen sich hervorragend für die Analyse fast aller gängigen Programmiersprachen. Natürliche Sprachen sind dagegen wegen ihrer Mehrdeutigkeit für eine deterministische Analyse ungeeignet. Mehrdeutigkeit resultiert immer in Mehrfacheinträgen in der Steuertabelle des Parsers.

Im Prinzip bestünde die Möglichkeit, diesen Nichtdeterminismus in der Steuertabelle wie beim ableitungsorientierten Bottom-up-Parser durch Backtracking zu simulieren. Dies würde jedoch auch hier zu einer (schlimmstenfalls) exponentiellen Laufzeit führen.

Im folgenden wird der Tomita-Parser vorgestellt, ein verallgemeinerter LR-Parser, der einen graphförmigen Stapel benutzt, um alle möglichen Analysen parallel verfolgen zu können.

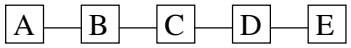
graphförmiger Stapel: Ein *graphförmiger Stapel* ist ein gerichteter, azyklischer Graph mit *einem* Ursprungsknoten, dessen Knoten den Zuständen auf den Stapeln der parallel simulierten LR-Parser entsprechen.

Auf graphförmige Stapel sind drei Operationen anwendbar:

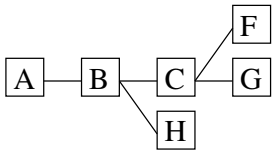
Teilung: Wenn ein Stapel auf mehrere Arten reduziert werden kann, wird der Stapel *geteilt*.

Beispiel: Gegeben sei die Teilgrammatik

F → D E
G → D E
H → C D E

und der Stapelinhalt 

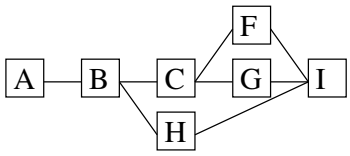
Nach einer nichtdeterministischen Reduzierung des Stapelinhaltes ergibt sich folgender Stapel:



Der neue Stapel besitzt nun drei oberste Knoten, nämlich F, G und H.

Zusammenfassung: Wenn ein Symbol auf einen Stapel mit mehr als einem obersten Knoten geschoben wird, so wird nur ein neuer Knoten erzeugt, der mit den (bisherigen) obersten Knoten verbunden wird.

Wenn im vorhergehenden Beispiel, das Symbol I auf den Stapel geschoben wird, ergibt sich der neue Stapelinhalt:

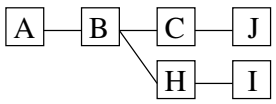


Zusammenpacken lokaler Mehrdeutigkeit: Wenn mehrere Zweige des Stapels identisch sind, so repräsentieren sie eine lokale Mehrdeutigkeit; d.h. derselbe Zustand wurde auf mehreren Wegen erreicht. Diese Zweige können zusammengepackt und wie eine Einheit behandelt werden.

Wenn im obigen Beispiel der Stapel mit den beiden Regeln

J → F I
J → G I

reduziert wird, so ergibt sich der folgende Stapelinhalt:



Der Zweig A—B—C—J ergibt sich auf zwei Arten, wird aber nur einmal im Stapel repräsentiert.

Tomita-Erkenner:

$x_1 x_2 \ldots x_n$ ist der Eingabestring.
$|p|$ bezeichnet die Länge der rechten Seite der Produktion p.

$[a, b, c]$ bezeichnet eine Liste mit den Elementen a,b und c.
$\circ$ ist der Listenverkettungsoperator.
$\{a, b, c\}$ bezeichnet eine Menge mit den Elementen a,b und c.
D_p ist das Nichtterminal auf der linken Seite der Produktion p.
$\langle i, s, l\rangle$ bezeichnet einen Knoten des graphförmigen Stapels, wobei $0 \leq i \leq n$ gilt, s ein Zustand ist und l eine Liste von Elternzuständen ist.
$\langle 0, S_0, \{\}\rangle$ ist die Wurzel des Stapels.
S_0 ist der Startzustand, der nie auf der rechten Seite einer Produktion erscheint.
U_i ist die Menge der obersten Stapelknoten nach der Verarbeitung von i Eingabesymbolen.
$Actions(s, x)$ ist die Menge der Aktionen im Zustand s bei Vorschauzeichen x.
$Goto(s, D)$ ist der Folgezustand von s, wenn das Symbol D konsumiert wird.

```
   Recognize(x_1 ... x_n)
(1)  x_{n+1} := $                          % füge ein Endesymbol an
     U_0 := [⟨0, S_0, {}⟩]
     for i := 1 to n + 1                   % für alle Eingabesymbole
(2)    U_i := [ ]
       P := [ ]
(3)    for all v = ⟨i − 1, s, l⟩ such that v ∈ U_{i−1}   % für alle Knoten in U_i inklusive der neu eingefügten
         P := P ∘ [v]                      % Merke die bereits verarbeiteten Knoten
(4)      for all a ∈ Actions(s, x_i) do    % für alle möglichen Aktionen dieses Knotens
(5)        if a = shift s' then
             Shift(v, s', i)
           else if a = reduce p then
             Reduce(v, p, i)
           else if a = accept then
             return true
(6)    if U_i = [ ] then
         return false                      % alle Analysen sind gescheitert
     return true

   Shift(v, s, i)
(7)  if ∃v' = ⟨i, s, l⟩ such that v' ∈ U_i then
       l := l ∪ {v}                        % Zusammenfassung von Knoten
     else
       U_i := U_i ∘ [⟨i, s, {v}⟩]          % erzeuge neuen Knoten

   Reduce(v, p, i)
(8)  for all v'_1 = ⟨j', s', l'_1⟩ such that v'_1 ∈ Ancestors(v, |p|) do
       s'' := Goto(s', D_p)
(9)    if ∃v'' = ⟨i − 1, s'', l''⟩ such that v'' ∈ U_{i−1} then
(10)     if v'_1 ∈ l'' then
           do nothing                      % lokale Mehrdeutigkeit
(11)     else if ∃v'_2 = ⟨j', s', l'_2⟩ such that v'_2 ∈ l'' then % v'_1 ist eine Kopie von v'_2
           v''_c := ⟨i − 1, s'', {v'_1}⟩   % erzeuge Kopie
           for all 'reduce p' ∈ Actions(s'', x_i) do   % verpaßte Reduktionen nachholen
             Reduce(v''_c, p, i)
         else
```

```
(12)        l'' := l'' ∪ {v'_1}
(13)        if v'' ∈ P then                          % Wurde v'' bereits verarbeitet?
              v''_c := ⟨i − 1, s'', {v'_1}⟩          % erzeuge Kopie
              for all 'reduce p' ∈ Actions(s'', x_i) do % verpaßte Reduktionen nachholen
                Reduce(v''_c, p, i)
            else
(14)          U_{i−1} := U_{i−1} ∘ [⟨i − 1, s'', {v'_1}⟩]

  Ancestors(v = ⟨j, s, l⟩, k)
(15)  if k = 0 then
        return {v}
      else
        return ∪_{v'∈l} Ancestors(v', k − 1)
```

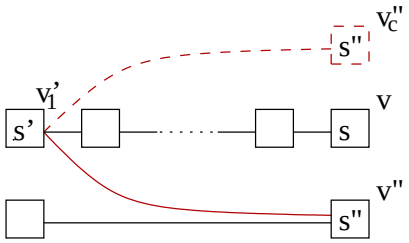
Der Erkenner *Recognize* bekommt den Eingabestring $x_1 \ldots x_n$ übergeben und liefert *true* zurück, falls der String erkannt wird und andernfalls $false$. Er ruft die Funktionen $Shift$ und *Reduce* auf. $Shift(v, s, i)$ fügt entweder einen neuen Knoten mit Zustand s und Vaterknoten v zu U_i hinzu, oder es fügt v als neuen Vaterknoten zu einem bereits existierenden Knoten hinzu. *Reduce* führt eine Reduzieraktion mit Knoten v und Produktion p durch. *Reduce* ruft die Funktion $Ancestors(v, |p|)$ auf, welche die Menge aller Vorgängerknoten, die $|p|$ Knoten weit von v entfernt sind, liefert.

In der Funktion *Recognize* wird zunächst in (1) das Symbol \$ an die Zeichenkette angehängt. (2) erzeugt den Wurzelknoten des Stapels. (3) iteriert über alle Eingabesymbole. Für jedes Symbol x_i verarbeitet (4) alle Knoten in U_{i-1}. Jeder Knoten v wird zunächst zur Liste P hinzugefügt, um ihn als bearbeitet zu markieren. Dann werden auf diesem Knoten alle Shift-, Reduce- und Accept-Aktionen der Steuertabelle für den Zustand s und das Vorschau-Symbol x_i ausgeführt. Reduzieraktionen können dazu führen, daß weitere Knoten in U_{i-1} eingefügt werden, die im selben Durchlauf in (4) abgearbeitet werden. Nachdem alle Knoten in U_{i-1} abgearbeitet sind, wird in (6) geprüft, ob das nächste Symbol erfolgreich auf den Stapel geschoben werden konnte.

In der Funktion $Shift$ wird in (7) ein Knoten mit Zustand s zu U_i hinzugefügt. Wenn bereits ein Knoten mit demselben Zustand in U_i vorhanden ist, so wird lediglich der Knoten v zur Liste der Elternknoten hinzugefügt.

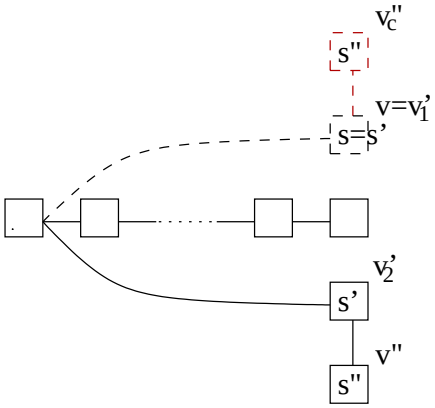
In der Funktion *Reduce* wird in (8) über alle Vorgängerknoten von v, die $|p|$ Knoten entfernt sind, iteriert. Für jeden Vorgänger wird s'' gleich dem Wert der Goto-Tabelle für den Vorgängerzustand s' und das Nichtterminal D_p gesetzt. Nun muß ein Knoten mit Zustand s'' zu U_{i-1} hinzugefügt werden. In (9) wird geprüft, ob ein solcher Knoten v'' bereits existiert. Falls nicht, wird in (14) ein neuer Knoten erzeugt und zu U_{i-1} hinzugefügt. Falls jedoch v'' existiert, so wird in (10) geprüft, ob bereits ein Übergang vom Vorgängerknoten v' erfolgt ist. Wenn dies der Fall ist, so wurde ein Segment des Eingabestrings auf zwei verschiedene Arten zu dem Nichtterminal D_p reduziert. Die neue Analyse repräsentiert somit eine lokale Mehrdeutigkeit und kann ignoriert werden. Andernfalls wird v zu den Elternknoten von v'' hinzugefügt. Vor dem Hinzufügen wird jedoch in (11) geprüft, ob v'_1 eine "Knotenkopie" ist, die in (13) in einem früheren Aufruf von *Reduce* erzeugt wurde (wie unten beschrieben). Falls v'_1 keine Kopie ist, so wird er in (12) zu den Elternknoten von v'' hinzugefügt. In (13) wird geprüft, ob v'' schon verarbeitet wurde. Falls ja, dann wurden eventuelle Reduktionen mit v' noch nicht durchgefuehrt. Um dies zu nachzuholen, wird v'' nach v''_c "kopiert", wobei die Elternliste von v''_c gleich $\{v'_1\}$ gesetzt wird. Anschließend werden alle Reduktionen auf v'' nochmals auf v''_c ausgeführt.

Das folgende Bild stellt die Situation graphisch dar. Gestrichelte Linien entsprechen Knotenkopien.



Zurück zu (11): Falls mit einer ε-Produktion reduziert wird, so liefert *Ancestors* den Knoten selbst als seinen Vorgängerknoten zurück. Wenn die Liste der Elternknoten von v'' nun eine Variante v'_2 von v'_1 enthält, dann ist v'_1 eine Kopie von v'_2. Zu diesem Zeitpunkt wurde v'' schon verarbeitet, was bedeutet, daß es Reduktionen mit dem Vaterknoten von v'_1 geben könnte, die noch nicht verarbeitet wurden. Um dies zu korrigieren, wird v'' erneut nach v''_c kopiert und alle Reduktionen auf v'' werden auf der Kopie v''_c wiederholt.

Das folgende Bild stellt die Situation graphisch dar.



Die Funktion *Ancestors* schließlich verfolgt rekursiv die Verbindungen zu den Elternknoten und gibt die Menge der Vorgängerknoten, die k Knoten von v entfernt sind zurück.

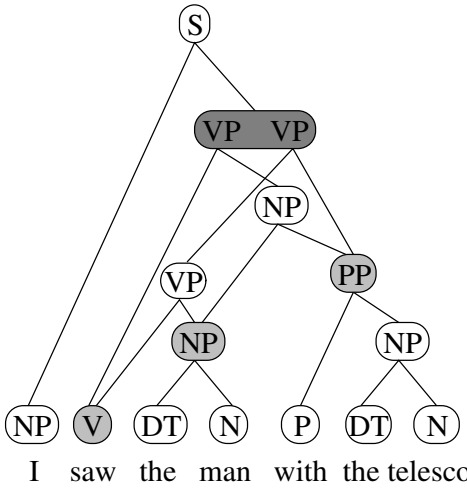
Parsewald-Repräsentation

Der beschriebene Erkenner wird zu einem Parser erweitert, indem während der Analyse die Ableitungsbäume aufgebaut werden. Da ein Satz exponentiell viele Analysen (und damit Ableitungsbäume) besitzen kann, ist es notwendig, die Parsebäume in kompakter Form zu repräsentieren. Hierzu dient die *Parsewald*-Repräsentation (packed shared parse forest).

Ein *Parsewald* ist dadurch gekennzeichnet, daß

- Teilbäume, die mehreren Analysen gemeinsam sind und denselben Teil des Eingabestrings überdecken, zusammengefaßt und nur einmal repräsentiert werden (subtree sharing). Es ergeben sich dadurch Knoten mit mehreren Elternknoten.
- Parsebäume, die sich nur in einem Teilbaum unterscheiden, zusammengefaßt werden. Die obersten Knoten der beiden Teilbäume werden dabei zu einem *gepackten* Knoten vereinigt (local ambiguity packing). Die Wurzelknoten der Teilbäume müssen dieselbe Kategorie haben.

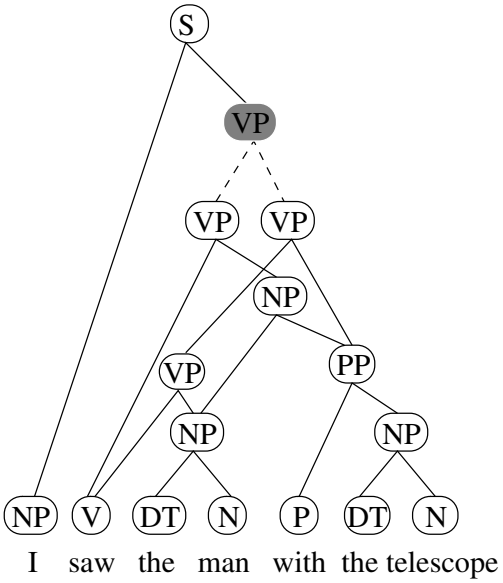
Die folgende Graphik zeigt einen solchen Parsewald. Der dunkelgrau unterlegte VP-Knoten ist ein gepackter Knoten. Er repräsentiert die beiden möglichen VP-Analysen. Die hellgrau unterlegten Knoten sind zusammengefaßte Knoten.



Parsewälder lassen sich auch als und-oder-Graphen darstellen. Sie unterscheiden sich nur durch eine etwas andere Darstellung der gepackten Knoten von der obigen Darstellung.

Und-Oder-Graph: Ein *Und-Oder-Graph* ist ein gerichteter, azyklischer Graph mit *einer* Wurzel und zwei Typen von nicht-terminalen Knoten, den Und-Knoten und den Oder-Knoten. Für jeden Und-Knoten mit der Kategorie A und einer Liste von Tochterknoten $B_1, B_2, \ldots, B_n$ existiert eine Produktion $A \rightarrow B_1 B_2 \ldots B_n$. Jeder Oder-Knoten hat zwei oder mehr Und-Knoten derselben Kategorie als Tochterknoten.

Der obige Parsewald sieht in dieser Darstellung folgendermaßen aus (gestrichelte Kanten sind Oder-Kanten, der grau unterlegte Knoten ist ein Oder-Knoten):



Parsewälder werden häufig zur kompakten Repräsentation mehrdeutiger Parseergebnisse verwendet.

Als weiterführende Literatur zum Tomita-Parser werden [Tomita 86], [Tomita 87] und [TomitaNg 91] empfohlen.

3.4 Chartorientierte Parser

Neben dem Tomita-Parser gibt es eine ganze Familie von weiteren Parsern für nicht-LR-Grammatiken. Sie alle verwenden eine Tabelle (Chart) zur Speicherung von partiellen Analyseergebnissen.

3.4.1 CYK-Parser

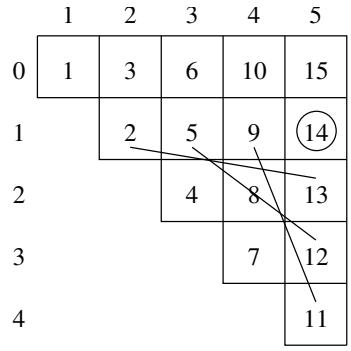
Cocke, Younger und Kasami entwickelten unabhängig voneinander einen Chart-basierten Bottom-up-Parser für Grammatiken in Chomsky-Normalform. Der Algorithmus basiert darauf, daß für einen gegebenen Eingabestring $x_1 \dots x_n$ eine dreickige Parsetabelle (Chart) konstruiert wird, deren einzelne Elemente mit t_{ij} $(0 \leq i < j \leq n)$ bezeichnet werden. Jedes t_{ij} hat eine Teilmenge von V (der Menge der Nichtterminale) als Wert, und für jedes Nichtterminal A in t_{ij} gilt $A \stackrel{*}{\Rightarrow} x_{i+1} \dots x_j$, das heißt der Teilstring $x_{i+1} \dots x_j$ ist aus A ableitbar. Die Eingabe wird akzeptiert, falls $S \in t_{0n}$.

CYK-Erkenner:

(0) Recognize($x_1 \dots x_n$)
(1) for $k := 1$ to n do % für alle Eingabesymbole
(2) $t_{k-1,k} := \{A | A \rightarrow x_k \in P\}$ % Nachschlagen im Lexikon
(3) for $i := k-2$ downto 0 do % für alle Felder in der k-ten Spalte
(4) $t_{ik} := \{\}$
(5) for $j := i+1$ to $k-1$ do % für alle Zerlegungen von $x_{i+1} \dots x_k$
(6) for all $A \rightarrow B\ C \in P$ do % für alle Grammatikregeln
(7) if $B \in t_{ij}$ and $C \in t_{jk}$ then % falls $B \stackrel{*}{\Rightarrow} x_{i+1} \dots x_j$ und $C \stackrel{*}{\Rightarrow} x_{j+1} \dots x_k$
(8) $t_{ik} := t_{ik} \cup \{A\}$ % dann $A \stackrel{*}{\Rightarrow} x_{j+1} \dots x_k$
(9) if $S \in t_{0n}$ then accept, otherwise reject

Die einzelnen Felder der Chart werden von links nach rechts und von unten nach oben gefüllt. Dadurch ist gewährleistet, daß die Felder t_{ij} und t_{jk} für bel. j vollständig berechnet sind, wenn in Zeile 4 ff t_{ik} berechnet wird.

Das folgende Bild zeigt die Reihenfolge, in der die einzelnen Felder der Chart gefüllt werden. t_{ij} im Algorithmus entspricht dabei dem Feld in Zeile i und Spalte j. Die Striche zeigen, welche Felderkombinationen beim Füllen des 14. Feldes betrachtet werden.



Der CYK-Erkenner kann zu einem Parser erweitert werden, indem nach dem Füllen der Chart top-down ein Parsewald zur kompakten Repräsentation aller Analysen generiert wird. Die erzeugten

Teilbäume werden in einer Tabelle p_{ik} gespeichert, um die Zusamenfassung identischer Teilanalysen zu erleichtern.

CYK-Parser1:

Parse$(x_1 \dots x_n)$

```
Recognize(x_1 ... x_n)                    % Chart aufbauen
if S ∈ t_{0,n} then                       % Analyse existiert
   p_ik := {} for all 0 ≤ i < k ≤ n       % Initialisierung
   return Gen(S, 0, n)                    % Erzeuge den Parsewald
else
   return fail                            % Keine Analyse
```

Gen(A, i, k)

```
if ∃n ∈ p_ik with n = ⟨A, l⟩ then         % Wurde Parsewald für A bereits generiert?
   return n
else if i = k − 1 then                    % Terminalsymbol
   n := x_k                               % Erzeuge Terminalknoten
   l := {n}
else                                      % Nichtterminal-Symbol
   l := {}
   for j := i + 1 to k − 1 do
      for all A → B C ∈ P do
         if B ∈ t_ij and C ∈ t_jk then    % Analyse für A gefunden
            n_1 := Gen(B, i, j)           % Generiere Parsewald für B
            n_2 := Gen(C, j, k)           % Generiere Parsewald für C
            l := l ∪ {⟨n_1, n_2⟩}
n := ⟨A, l⟩                               % Erzeuge neuen Knoten
p_ik := p_ik ∪ {n}                        % Speichere den Knoten in einer Tabelle
return n
```

Alternativ kann der Parsewald auch schon während des Bottom-up-Schrittes aufgebaut werden. Statt der Symbole werden in diesem Fall Paare bestehend aus einem Symbol und einer Liste von Verweisen auf Tochterknoten in der Tabelle gespeichert.

CYK-Parser2:

```
(0)  Parse(x_1 ... x_n)
(1)     for k := 1 to n do
(2)        for all A → x_k ∈ P do
(2a)          Add(A, k − 1, k, x_k, nil)
(3)     ...
(7)                if n_1 ∈ p_ij with n_1 = ⟨B, l⟩ and n_2 ∈ p_jk with n_2 = ⟨C, l'⟩ then
(8)                   Add(A, i, k, n_1, n_2)
(9)     if ∃n ∈ p_0n with n = ⟨S, l⟩ then return n, otherwise return fail
(10)
(11) Add(A, i, k, n_1, n_2)
```

```
(12)  if n2 = nil then                          % A → x_k
(13)    p_ik := p_ik ∪ {⟨A, {n1}⟩}
(14)  else                                      % A → B C
(15)    if ∃⟨A, l⟩ ∈ p_ik then
(16)      l := l ∪ {⟨n1, n2⟩}
(17)    else
(18)      p_ik := p_ik ∪ {⟨A, {⟨n1, n2⟩}⟩}
```

In Zeile 13 wird ein Parsebaum bestehend aus einem Knoten mit Symbol A und einem einzigen terminalen Tochterknoten mit Symbol x_k generiert. In Zeile 15 wird geprüft, ob die neue Analyse mit einer bereits vorhandenen Analyse vereinigt werden kann. Falls ja, wird die Liste der Analysen des entsprechenden Knotens erweitert (Zeile 16), andernfalls wird ein neuer Knoten mit einer einzigen Analyse generiert.

Der CYK-Parser wird in der vorgestellten Form selten tatsächlich fürs Parsen verwendet. Dies hat zwei Gründe:

- Er erfordert Grammatiken in Chomsky-Normalform.
- Konstituenten werden auch dann generiert, wenn aufgrund des linken Kontextes klar ist, daß letztendlich keine Reduktion zum Startsymbol möglich ist.

Der CYK-Parser läßt sich zu einem Parser für allgemeine kontextfreie Grammatiken erweitern, indem statt nach Paaren von reduzierbaren Konstituenten nach N-Tupeln von reduzierbaren Konstituenten gesucht wird. Die Komplexität des Algorithmus steigt in diesem Fall jedoch exponentiell mit der Länge der rechten Seite der Grammatikregeln N an, so daß der erweiterte CYK-Algorithmus ineffizient ist. Der Grund für den exponentiellen Anstieg ist darin zu suchen, daß der Parser für eine gegebene Regel $A \rightarrow BCD$ mehrfach prüft, ob und wie sich ein Teilstring $x_j \ldots x_k$ zu $B\ C$ reduzieren läßt.

Der exponentielle Anstieg kann vermieden werden, indem der Parser inkrementell für Präfixe der rechten Seite einer Regel berechnet, welche Teile der Eingabe zu diesem Präfix reduziert werden können. Diese Technik findet im Left-Corner-Parser und im Earley-Parser Verwendung, die nachfolgend vorgestellt werden.

3.4.2 Left-Corner-Parser

Bei einem *Left-Corner-Parse* werden die Knoten des Parsebaumes in einer solchen Reihenfolge generiert, daß jeder Knoten nach seinem ersten (nichtleeren) Tochterknoten und vor allen anderen Tochterknoten generiert wird.

Ein Parser, der einen Left-Corner-Parse erzeugt, heißt entsprechend *Left-Corner-Parser*. Ein Left-corner-(Chart-)Parser[1] ist wie der CYK-Parser ein Bottom-up-Parser. Im Gegensatz zum CYK-Parser, ist er jedoch auf beliebige kontextfreie Grammatiken anwendbar. Die unten vorgestellte Variante erfordert allerdings Grammatiken, in denen Terminalsymbole durch Kettenproduktionen der Form $A \rightarrow x$ abgeleitet werden, was bei computerlinguistischen Grammatiken üblicherweise der Fall ist.

[1] Es gibt auch eine ableitungsorientierte Variante des Left-Corner-Parsers.

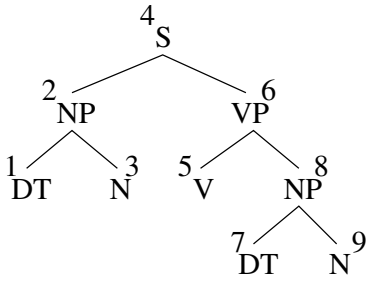


Abbildung 1: Left-Corner-Parse

Der Left-Corner-Parser merkt sich mit Hilfe von *geteilten Produktionen* (dotted rules, vgl. LR-Parsing), welche Präfixe von Regeln einen bestimmten Teil des Eingabestrings generieren können. Wenn die Grammatik die Regel $A \to BCD$ enthält und $BC \stackrel{*}{\Rightarrow} x_{i+1} \dots x_k$ gilt, so fügt der Parser $\langle A \to BC \cdot D, i, k\rangle$ in die Chart ein.

Allgemein führt der Left-Corner-Parser die beiden folgenden Aktionen wiederholt aus:

Predict: Wenn $\langle A \to \alpha\cdot, i, k\rangle \in t_k$, dann füge $\{B \to A \cdot \beta, i, k\}$ für alle $A \to A\beta \in P$ zu t_k hinzu.

Complete: Wenn $\langle A \to \alpha \cdot B\beta, i, j\rangle \in t_j$ und $\langle B \to \gamma\cdot, j, k\rangle \in t_k$, dann füge $\langle A \to \alpha B \cdot \beta, i, k\rangle$ zu t_k hinzu.

Left-Corner-Erkenner1:

Recognize($x_1 \dots x_n$)

```
t_0 := {}                                   % Initialisierung der Chart
for j := 1 to n do                          % für alle Eingabesymbole
   t_j := {}                                % Initialisierung
   for all A → x_j ∈ P do
      Add(A → x_j·, j − 1, j)               % Scan
if ⟨S → α·, 0, n⟩ ∈ t_n for some α then
   accept
else
   reject
```

Add($A \to \alpha \cdot \beta, i, k$)

```
if ⟨A → α · β, i, k⟩ ∉ t_k then             % neue Kante?
   t_k := t_k ∪ {⟨A → α · β, i, k⟩}          % neue Kante eintragen
   if β = ε then                            % passive Kante?
      Complete(A, i, k)
      Predict(A,i,k)
```

Predict(A, i, k)

```
if not predicted[A, i, k] then              % Predict-Schritt noch nicht ausgeführt?
   predicted[A, i, k] := true
```

```
    for all B → Aα ∈ P do
      Add(B → A · α, i, k)                    % erwartete Kanten eintragen

Complete(A, j, k)
  if not completed[A, j, k] then
    completed[A, j, k] := true
    for all ⟨B → β · Aγ, i, j⟩ ∈ t_j do
      Add(B → βA · γ, i, k)                   % neue Kante eintragen
```

In der vorgestellten Form kann der Left-Corner-Parser keine Grammatiken mit ε-Produktionen verarbeiten. Wie der CYK-Erkenner und der unten vorgestellte Earley-Parser läßt sich auch der Left-Corner-Erkenner zu einem Parser erweitern, indem bei jedem Charteintrag die Liste der möglichen Kombinationen von Tochtereinträgen gespeichert wird.

In der vorgestellten Form ist der Left-Corner-Parser ein reiner Bottom-up-Parser und hat daher genauso wie der CYK-Parser das Problem, daß Konstituenten generiert werden, die im gegebenen Linkskontext keine Reduktion zum Startsymbol erlauben.

3.4.3 Earley-Parser

Der Earley-Parser unterscheidet sich vom Left-Corner-Parser darin, daß der Predict-Schritt nicht bottom-up durch vollständig erkannte Konstituenten (passive Kanten in der Chart) getriggert wird, sondern top-down durch unvollständig erkannte Konstituenten (aktive Kanten in der Chart).

Earley-Erkenner:

```
Recognize(x_1 ... x_n)
  t_0 := {}                                   % Initialisierung der Chart
  Predict(S, 0)
  for j := 1 to n do                          % für alle Eingabesymbole
    t_j := {}                                 % Initialisierung
    for all ⟨A → α · x_j β, i, j − 1⟩ ∈ t_{j−1} do
      Add(A → αx_j · β, i, j)                 % Scan
  if ⟨S → α·, 0, n⟩ ∈ t_n for some α then
    accept
  else
    reject

Add(A → α · β, i, k)
  if ⟨A → α · β, i, k⟩ ∉ t_k then             % neue Kante?
    t_k := t_k ∪ {⟨A → α · β, i, k⟩}          % neue Kante eintragen
    if β = ε then                             % passive Kante?
      Complete(A, i, k)
    else if β = Bγ for some B, γ then         % aktive Kante?
      if ⟨B → δ·, k, k⟩ ∈ t_k then            % ε-Kante für B eingetragen?
        Add(A → αB · γ, i, k)                 % Complete-Schritt für neues Item nachholen
      else
```

Predict(B,k)

Predict(A, i)

if not predicted$[A, i]$ then % Predict-Schritt noch nicht ausgeführt?
  predicted$[A, i] :=$ true
  for all $A \to \alpha \in P$ do
    Add$(A \to \cdot\alpha, i, i)$ % erwartete Kanten eintragen

Complete(A, j, k)

if not completed$[A, j, k]$ then
  completed$[A, j, k] :=$ true
  for all $\langle B \to \beta \cdot A\gamma, i, j\rangle \in t_j$ do
    Add$(B \to \beta A \cdot \gamma, i, k)$ % neue Kante eintragen

Der Earley-Parser hat den Vorteil, daß alle Kanten in der Chart von ihrem linken Kontext lizensiert sind. In der vorgestellten Form hat er aber den Nachteil, daß vor allem der Predict-Schritt viele aktive Kanten erzeugt, die später keine Fortsetzung finden. Dies kann vermieden werden, indem wie bei den tabellengesteuerten Verfahren ein Vorschauzeichen betrachtet wird. Eine Kante $A \to \alpha \cdot \beta, i, k$ wird nur noch dann in die Chart eingetragen, wenn $\beta \stackrel{*}{\Rightarrow} x_{k+1}\gamma$ oder $\beta \stackrel{*}{\Rightarrow} \varepsilon$ und $x_{k+1} \in Follow[A]$ gilt (zur Definition von Follow siehe Abschnitt 3.3.2). Die dadurch definierte Relation $L(A \to \alpha \cdot \beta, x)$ kann vorab berechnet werden. Die Prüfung des Vorschauzeichens erfordert somit nur konstante Zeit (Tabellenzugriff).

Der Earley-Parser mit Lookahead unterscheidet sich vom Earley-Parser ohne Lookahead nur in der Add-Funktion, an deren Anfang eine if-Anweisung eingefügt wird, und in der Recognize-Funktion, die ein neues Terminalsymbol \$ als Endemarkierung an den Eingabestring anfügt.

Recognize$(x_1 \ldots x_n)$

$x_{n+1} =$ \$
$t_0 := \{\}$ % Initialisierung der Chart
...

Add$(A \to \alpha \cdot \beta, i, k)$

if $L(A \to \alpha \cdot \beta, x_{k+1})$ then % Vorschauzeichen prüfen
  if $\langle A \to \alpha \cdot \beta, i, k\rangle \notin t_k$ then % neue Kante?
    ...

3.4.4 Top-Down-Filter

Der Left-Corner-Parser kann verbessert werden, indem durch einen *Top-Down-Filter* die Generierung von Kanten, die von ihrem linken Kontext nicht lizensiert sind, verhindert wird. Dazu wird vorab eine *Erreichbarkeits-Funktion* R berechnet. $R(A)$ ist die Menge aller Nichtterminale B, für die $A \stackrel{*}{\Rightarrow} B\alpha$ gilt, d.h. die Menge der Nichtterminale, die am Anfang einer aus A abgeleiteten Satzform stehen können.

Oft ist mit dem Begriff *Left-Corner-Parser* die Variante mit Top-Down-Filter gemeint.

Left-Corner-Erkenner2:

Recognize$(x_1 \ldots x_n)$

```
t_0 := {}
r_0 := R(S)                                          % neu
for j := 1 to n do
    t_j := {}
    r_j := {}                                        % neu
    for all A → x_j ∈ P such that A ∈ r_{j-1} do     % neu
        Add(A → x_j·, j − 1, j)
if ⟨S → α·, 0, n⟩ ∈ t_n for some α then
    accept
else
    reject
```

Add$(A \rightarrow \alpha \cdot \beta, i, k)$

```
if ⟨A → α · β, i, k⟩ ∉ t_k then
    t_k := t_k ∪ {⟨A → α · β, i, k⟩}
    if β = ε then
        Complete(A, i, k)
        Predict(A,i,k)
    else if β = Bγ for some B, γ then                % hier immer wahr
        r_k := r_k ∪ R(B)                            % neu
```

Predict(A, i, k)

```
if not predicted[A, i, k] then
    predicted[A, i, k] := true
    for all B → Aα ∈ P such that B ∈ r_i do          % neu
        Add(B → A · α, i, k)
```

Complete(A, j, k)

```
if not completed[A, j, k] then
    completed[A, j, k] := true
    for all ⟨B → β · Aγ, i, j⟩ ∈ t_j do
        Add(B → βA · γ, i, k)
```

Der Left-Corner-Parser mit Top-Down-Filter ist dem Earley-Parser recht ähnlich. Im Gegensatz zum Earley-Parser erzeugt er aktive Kanten aber erst, wenn das erste Symbol auf der rechten Seite der Regel vollständig erkannt wurde. Dadurch werden weniger Kanten als beim Earley-Parser erzeugt.

Wie beim Earley-Parser kann auch beim Left-Corner-Parser eine Lookahead-Relation verwendet werden, um die Zahl der erzeugten aktiven Kanten weiter zu verringern.

3.4.5 Grammatiktransformationen

Die Einführung der geteilten Produktionen beim Earley-Parser und beim Left-Corner-Parser kann als eine implizite Grammatiktransformation aufgefaßt werden, durch die eine Regel der Form

$A \to B_1 \ldots B_n$ durch eine Menge von binären Regeln ersetzt wird.

$$\begin{aligned}
A &\to \langle A \to B_1 \ldots B_n \cdot \rangle \\
\langle A \to B_1 \ldots B_n \cdot \rangle &\to \langle A \to B_1 \ldots B_{n-1} \cdot B_n \rangle \; B_n \\
\langle A \to B_1 \ldots B_{n-1} \cdot B_n \rangle &\to \langle A \to B_1 \ldots B_{n-2} \cdot B_{n-1} B_n \rangle \; B_{n-1} \\
&\ldots \\
\langle A \to B_1 \cdot B_2 \ldots B_n \rangle &\to B_1
\end{aligned}$$

Diese Form der Transformation hat den Nachteil, daß für jede Regel der Länge n jedesmal n neue Grammatikregeln generiert werden. Wenn die rechten Seiten zweier Regeln einen identischen Anfang besitzen, wäre es günstiger, die entsprechenden Regeln zusammenzufassen (Linksfaktorisierung). Dies wird erreicht, indem die noch zu erwartenden Symbole nicht in den Namen der Hilfssymbole "gespeichert" werden.

Zwei Regeln $A \to BCD$ und $A \to BCDE$ werden dann folgendermaßen transformiert:

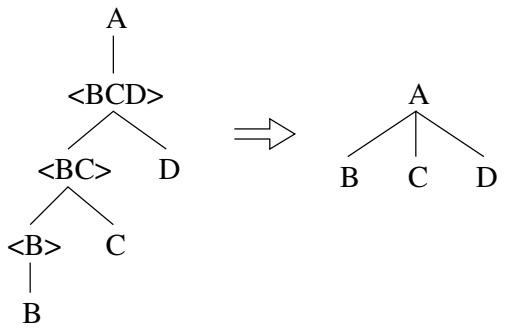
$$\begin{aligned}
A &\to \langle A \to BCD \rangle \\
A &\to \langle A \to BCDE \rangle \\
\langle A \to BCDE \rangle &\to \langle A \to BCD \rangle \; E \\
\langle A \to BCD \rangle &\to \langle A \to BC \rangle \; D \\
\langle A \to BC \rangle &\to \langle A \to B \rangle \; C \\
\langle A \to B \rangle &\to B
\end{aligned}$$

Der Vorteil dieser Transformation besteht darin, daß der Parser nun nicht mehr beide Analysen parallel generieren muß. Die Grammatik läßt sich weiter vereinfachen, indem auch die linke Seite der Regel nicht mehr in den Hilfssymbolen der transformierten Grammatik gespeichert wird.

Zwei Regeln $A \to BCD$ und $F \to BCDE$ werden dann folgendermaßen transformiert:

$$\begin{aligned}
A &\to \langle BCD \rangle \\
F &\to \langle BCDE \rangle \\
\langle BCDE \rangle &\to \langle BCD \rangle \; E \\
\langle BCD \rangle &\to \langle BC \rangle \; D \\
\langle BC \rangle &\to \langle B \rangle \; C \\
\langle B \rangle &\to B
\end{aligned}$$

Wenn ein Parsebaum generiert werden soll, so muß darauf geachtet werden, daß sich der Parsebaum einfach aus den Charteinträgen ableiten läßt. Bei den obigen Transformationen ist dies der Fall: Alle Regeln der transformierten Grammatik mit einem normalen Symbol auf der linken Seite lassen sich eindeutig auf Regeln der Ursprungsgrammatik abbilden. Alle Regeln mit Hilfssymbolen auf der linken Seite führen lediglich Zwischenebenen in die Analyse ein und werden bei der Suche nach den Tochterknoten eines Knotens übersprungen.



Bei anderen Grammatik-Transformationen wie der Eliminierung von Linksrekursion ist eine einfache Generierung des Parsewaldes nicht mehr gegeben.

3.5 Vergleich der Analyseverfahren

Die verschiedenen Analyseverfahren lassen sich bzgl. der Eigenschaften Analyserichtung, Typ der Behandlung von Nichtdeterminismus, Verwendung eines Top-Down-Filters, Verwendung eines Vorschauzeichens und Grammatikeinschränkungen miteinander vergleichen.

Richtung	Typ	TDF	Lookahead	Grammatik	Verfahren
top-down	Backtracking	–	nein	kontextfrei	TD mit Backtr.
top-down	deterministisch	–	ja	LL(1)	LL(1)-Parser
top-down	Chart	–	nein	kontextfrei	Earley ohne Lookahead
top-down	Chart	–	ja	kontextfrei	Earley mit Lookahead
bottom-up	Backtracking	nein	nein	kontextfrei	BU mit Backtr.
bottom-up	deterministisch	ja	nein	LR(0)	LR(0)-Parser
bottom-up	deterministisch	ja	ja	LR(1)	LR(1)-Parser
bottom-up	pseudo-det.	ja	ja	kontextfrei	Tomita-Parser
bottom-up	Chart	nein	nein	Chomsky-NF	CYK-Parser
bottom-up	Chart	nein	nein	kontextfrei	LC-Parser
bottom-up	Chart	ja	nein	kontextfrei	LC-Parser mit Pred.
bottom-up	Chart	ja	ja	kontextfrei	LC m. Pred. u. LA

3.6 Komplexität des kontextfreien Parsens

Da die einzelnen Parser für kontextfreie Grammatiken im wesentlichen diesselbe Ausgabe liefern, kommt es bei einem Vergleich der Parser vor allem auf die unterschiedlichen Speicherplatz- und Rechenzeitanforderungen (Komplexität) an.

Zur Beurteilung der Speicherplatz- und Rechenzeitkomplexität eines Algorithmus wird in der Regel nur das asymptotische Verhalten für wachsende Eingabelänge betrachtet. Das hat den Vorteil, daß von Details der Computer-Hardware abstrahiert werden kann. Ferner wird meist nur die im ungünstigsten Fall zu erwartende Komplexität eines Algorithmus (worst case complexity) betrachtet.

$O(f(n))$: Die Komplexitätsklasse $O(f(n))$ umfaßt alle Algorithmen, für deren Komplexität $g(n)$ gilt: $\exists_{K,x_0}\forall_{n>x_0} Kf(n) > g(n)$

Das heißt durch $O(f(n))$ wird die Komplexität eines Algorithmus nach oben abgeschätzt. Asymptotisch benötigt der Algorithmus maximal $Kf(n)$ Schritte für ein entsprechend gewähltes K.

Ein Algorithmus mit der Komplexität $100n^2 + 1000n + 10000\ log(n)$ beispielsweise ist in der Komplexitätsklasse $O(n^2)$, da für $K = 10000$ und bel. n gilt $10000n^2 > 100n^2 + 1000n + 10000\ log(n)$.

Andererseits läßt sich zeigen, daß ein Algorithmus mit der Komplexität 2^n nicht in $O(n^p)$ für beliebiges p ist. Algorithmen, die nicht in $O(n^p)$ sind, heißen *nicht-polynomiell*.

$\Omega(f(n))$: Die Komplexitätsklasse $\Omega(f(n))$ umfaßt alle Algorithmen, für deren Komplexität $g(n)$ gilt: $\exists_{K,x_0}\forall_{n>x_0} Kf(n) < g(n)$

Ω schätzt also die Komplexität eines Algorithmus nach unten ab. Der Algorithmus benötigt mindestens $Kf(n)$ Schritte für ein entsprechend gewähltes K.

3.6.1 Top-Down-Parser mit Backtracking

Der Top-Down-Parser durchläuft mit Backtracking eine Folge von Konfigurationen, bis eine erfolgreiche Linksableitung gefunden wurde, oder bis alle möglichen Konfigurationen erfolglos durchlaufen wurden.

Es wird nun gezeigt werden, daß die Länge einer Konfiguration und somit der Speicherplatzbedarf durch $O(n)$ begrenzt ist, d.h. linear ist, und daß die Zahl der Konfigurationsübergänge und damit die Rechenzeit durch $O(c^n)$ begrenzt ist und somit exponentiell ist.

Die folgenden Konfigurationsübergänge sind möglich:

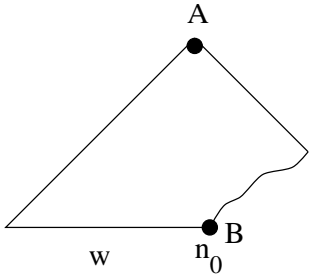
- Von $(a\alpha, ar)$ nach (α, r), d.h. ein erwartetes Terminalsymbol wird "konsumiert"; oder
- von $(A\beta, r)$ nach $(\alpha\beta, r)$, falls $A \to \alpha \in P$, d.h. ein erwartetes Nichtterminalsymbol wird durch die rechte Seite einer Produktion ersetzt (expandiert).

Es wird angenommen, daß ein Konfigurationsübergang in konstanter Zeit erfolgt.

Lemma 1: Wenn $G = (V, \Sigma, P, S)$ eine nicht-linksrekursive Grammatik ist, dann gibt es eine Konstante c, so daß für alle Linkssatzformen $wB\alpha$ mit $|w| = n$, die in i Schritten aus einem Nichtterminal A ableitbar sind, $i \leq c^{n+2}$ gilt.

Beweis: Angenommen die Grammatik G hat k Nichtterminale. Dann hat der partielle Ableitungsbaum, der der Ableitung der Linkssatzform $A \stackrel{i}{\Rightarrow} wB\alpha$ entspricht, eine Höhe von maximal $k(n+2)$. Wenn dem nicht so wäre, so müßte ein Pfad der Länge $l > k(n+2)$ von der Wurzel zu einem Blatt des Ableitungsbaumes existieren. Wir zeigen nun, daß dies nicht der Fall sein kann.

Angenommen n_0 ist der Knoten im Ableitungsbaum, der dem Nichtterminal B in der Linkssatzform $wB\alpha$ entspricht. Da $wB\alpha$ eine Linkssatzform ist, bei deren Erzeugung immer das am weitesten links stehende Nichtterminal ersetzt wird, gilt daß der Pfad von der Wurzel bis n_0 mindestens so lang ist wie jeder beliebige Pfad zu einem Blatt rechts von n_0 im Ableitungsbaum. Jedes Blatt rechts von n_0 hat nämlich einen Vorgängerknoten von n_0 als Mutterknoten. Es genügt daher, die Länge der Pfade zu n_0 und zu den Blattknoten links von n_0 zu betrachten.



Angenommen es existiert ein Pfad der Länge $l > k(n+2)$. Da wB die Länge $n+1$ hat und die Knoten auf diesem Pfad somit nur $n+1$ viele unterschiedliche nicht-leere Teilstrings von wB abdecken können, gibt es $k+1$ oder mehr untereinanderhängende Knoten, die alle denselben Substring r von wB erzeugen. Da es nur k viele unterschiedliche Nichtterminale gibt, haben mindestens zwei dieser Knoten dieselbe Kategorie C. Es müßte also eine Ableitung $C \stackrel{+}{\Rightarrow} C\beta \stackrel{*}{\Rightarrow} r$ geben. Die Grammatik wäre also entgegen der Annahme linksrekursiv.

Es kann somit geschlossen werden, daß es im Ableitungsbaum keinen Pfad der Länge größer $k(n+2)$ gibt. Wenn l die Länge der rechten Seite der längsten Produktion in P ist, dann hat der Ableitungsbaum nicht mehr als $l^{k(n+2)}$ innere Knoten. Wenn $A \stackrel{i}{\Rightarrow} wB\alpha$, so ist demnach $i \leq l^{k(n+2)}$ und für $c = l^k$ folgt $i \leq c^{n+2}$. □

Lemma 2: Die Länge der Konfigurationen eines Top-Down-Parsers ist maximal $(l-1)k(n+2)$.

Beweis: Gemäß Lemma 1 hat jeder Ableitungsbaum für die Rechtssatzform $wB\alpha$ eine maximale Tiefe von $k(n+2)$, wobei k die Zahl der Nichtterminale in der Grammatik ist. Es gilt die Länge von α abzuschätzen. Jeder Knoten im Ableitungsbaum entsprechend den Symbolen in α hat einen Vorgängerknoten von n_0 (dem Knoten, der B entspricht) zum Mutterknoten. Da n_0 maximal $k(n+1)$ Vorgängerknoten hat und jeder Vorgängerknoten maximal l Tochterknoten hat, gilt $|\alpha| \leq (l-1)k(n+1)$. Es folgt für alle Konfigurationen $wB\alpha$, daß $|wB\alpha| \leq n+1+(l-1)k(n+1) = O(|G|n)$ für $|G| = kl$. □

Der Speicherplatzbedarf eines Top-Down-Parsers ist also linear in der Länge der Eingabe und der Größe der Grammatik.

Aus Lemma 1 folgt, daß alle Ableitungen der Form $A \stackrel{*}{\Rightarrow} \varepsilon$ und $A \stackrel{*}{\Rightarrow} B$ in konstanter Zeit c bezüglich der Eingabelänge erfolgen. Da G eine nicht-linksrekursive Grammatik ist, erzeugt ein Top-Down-Parser im Mittel nach maximal $c' = k(l-1)c$ vielen Schritten ein neues terminales Symbol. Der Ausdruck setzt sich zusammen aus maximal k Rekursionsschritten mit jeweils maximal $(l-1)c$ vielen Schritten, um vorausgehende Symbole nach ε abzuleiten. Der Parser benötigt insgesamt für die Ableitung eines Strings der Länge n ohne Backtracking maximal nc' viele Schritte. In jedem Schritt gibt es maximal $|P|$ viele alternative Schritte. Der Parser verarbeitet daher eine Eingabe der Länge n in nicht mehr als $|P|^{c'n} = (|P|^{c'})^n = \hat{c}^n$ Schritten (siehe auch [Aho/Ullman 72]).

3.6.2 Bottom-up-Parser mit Backtracking

Der Bottom-up-Parser mit Backtracking weist dieselbe Zeitkomplexität $O(c^n)$ und Speicherkomplexität $O(n)$ auf, wie der Top-Down-Parser.

Beweis: Übung

3.6.3 xLR(k)-Parser

Ein LL(k)-Parser hat dieselbe Speicherplatzkomplexität wie ein Top-Down-Parser, da dieselbe Repräsentation der Konfigurationen verwendet wird und die Größe der Steuertabelle von der Grammatik nicht aber von der Eingabelänge abhängt.

Ein LL(k)-Parser benötigt soviele Schritte wie ein Top-Down-Parser, der kein Backtracking macht. Die Rechenzeitkomplexität ist daher $O(n)$.

Analog ergibt sich auch für die xLR(k)-Parser eine lineare Speicherplatz- und Rechenzeit-Komplexität.

3.6.4 CYK-Parser

Der CYK-Erkenner benötigt $|V|O(n^2)$ viel Speicherplatz (wobei $|V|$ die Zahl der Nichtterminale ist) für die Chart und $O(|G|n^3)$ viel Rechenzeit (wobei G die Zahl der Grammatikregeln ist).

CYK-Erkenner:

(0) Recognize($x_1 \dots x_n$)
(1) for $k := 1$ to n do % $O(n)$ Durchläufe
(2) $t_{k-1,k} := \{A | A \to x_k \in P\}$ % $O(|N|)$ Schritte
(3) for $i := k-2$ downto 0 do % $O(n)$ Durchläufe
(4) $t_{ik} := \{\}$
(5) for $j := i+1$ to $k-1$ do % $O(n)$ Durchläufe
(6) for all $A \to B\ C \in P$ do % $O(|P|)$ Durchläufe
(7) if $B \in t_{ij}$ and $C \in t_{jk}$ then
(8) $t_{ik} := t_{ik} \cup \{A\}$
(9) if $S \in t_{0n}$ then accept, otherwise reject

Durch die Schleifenschachtelung ergeben sich $O(n(|N| + |P|n^2)) = O(|G|n^3)$ Schritte bei der Verarbeitung einer Eingabe der Länge n. Dabei wird davon ausgegangen, daß das Einfügen und Nachschauen in der Chart nur konstante Zeit erfordert.

Wenn der CYK-Erkenner zu einem Parser erweitert wird, sind zusätzlich die Verweise auf Tochterknoten zu speichern. Für jedes Feld t_{ij} der Chart gibt es maximal $|P| * n$ viele Verweise. Es ergibt sich dadurch ein Speicherplatzbedarf von $O(n^3 * |P|)$.

3.6.5 Earley-Parser

Der Earley-Erkenner speichert maximal $n^2|P|(l+1)$ viele Kanten in der Chart, wobei l die Länge der längsten rechten Seite einer Regel ist. Für jede der maximal $n^2|P|l$ aktiven Kanten wird die Predict-Funktion mit $O(|P|)$ Schritten aufgerufen. Für jede der maximal $n^2|V|$ erkannten Konstituenten wird die Complete-Funktion mit maximal $O(n|P|l)$ Schritten aufgerufen. Es ergeben sich insgesamt maximal $O(n^2|P|^2l + n^2|V|n|P|l) = O(n^3|V||P|l)$ Schritte.

Wenn die Grammatik eindeutig ist und keine unerreichbaren oder unproduktiven Symbole enthält, so wird für jede Kante der Form $A \to \alpha \cdot \beta, i, j$ mit $\alpha \neq \varepsilon$ maximal einmal versucht, sie in die Chart einzutragen. Also wird die Schleife in der Complete-Funktion insgesamt höchstens $O(n^2|P|l)$ mal durchlaufen. Somit macht der Erkenner auch nur $O(n^2|G|)$ viele Schritte.

Wenn die Grammatik eine SLR-Grammatik ist, so macht der oben beschriebene Earley-Erkenner mit Lookahead sogar nur linear viele Schritte.

Wenn der Earley-Erkenner zu einem Parser erweitert wird, indem in der Chart Verweise auf die Tochterknoten einer Konstituente gespeichert werden, dann steigt der Speicherplatzbedarf des Parsers auf $O(|G|n^3)$.

Anmerkung: Damit in konstanter Zeit geprüft werden kann, ob eine Kante bereits in der Chart ist, muß jede Kante wie beim CYK-Parser in eine dreidimensionale Tabelle ($n * n * |P| * L(P)$) eingetragen werden. In diesem Fall kann der Parser allerdings wegen der nötigen Initialisierungsschritte LR(k)-GHrammatiken nicht mehr in linearer Zeit verarbeiten. Wenn eine zweidimensionale Chart verwendet wird, bei der jedes Feld eine verkettete Liste von Kanten enthält, steigt die Komplexität

bzgl. der Grammatikgröße auf $O(|G|^2)$ an. Für lineares Parsen von LR(k)-Grammatiken muß eine eindimensionale Chart verwendet werden. Der Test, ob eine Kante bereits existiert, kann auch mit einer Hashtabelle erfolgen. In diesem Falle erhält man das beschriebene Laufzeitverhalten für LR(k)-Grammatiken, eindeutige Grammatiken und allgemeine kontextfreie Grammatiken nur mit sehr hoher Wahrscheinlichkeit, nicht aber mit Sicherkeit.

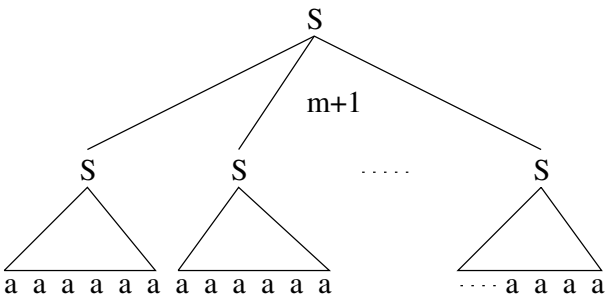
3.6.6 Tomita-Parser

Die Zeitkomplexität des Tomita-Parsers (siehe auch [Johnson 91, Kipps 91]) ist für manche Grammatiken schlechter als $O(n^3)$, d.h. schlechter als die des Earley-Parsers. Beispiele sind die Grammatiken

S → a
S → S S
S → S^{m+2}

mit $m > 0$.

Alle diese Grammatiken erzeugen dieselbe Sprache a^+. Wir betrachten den Eingabestring a^{n+1} mit $n > m$. Mittels der beiden ersten Regeln kann jeder nichtleere Teilstring der Eingabe als S analysiert werden. Daraus folgt, daß es für den obersten S-Knoten $\binom{n}{m+1}$ Möglichkeiten gibt, Tochterknoten gemäß der dritten Regel auszuwählen (= Zahl der Möglichkeiten, die rechten Stringpositionen der Tochterknoten auszuwählen). $\binom{n}{m+1}$ ist ein Polynom in n der Ordnung $m+1$. Eine untere Schranke für dieses Polynom ist durch cn^m gegeben. Daraus folgt, daß die Größe des Parsewaldes und damit auch die Zeitkomplexität des Tomita-Parsers, der den Parsewald generiert, mit $\Omega(n^m)$ nach unten abgeschätzt werden kann.



Dieses Argument bezieht sich allein auf den Parsewald, den der Tomita-Parser aufbaut, und gilt für alle anderen Parser, die dieselbe Repräsentation verwenden, ebenso. Es gilt jedoch nicht für den Left-Corner-Parser und den Earley-Parser, da dort jeder Charteintrag nur Verweise auf maximal zwei Tochtereinträge enthält.

Weiterhin läßt sich beweisen, daß die Zeitkomplexität des Tomita-Parsers für manche Grammatiken exponentiell mit der Größe des Grammatik ansteigt. Dazu wird gezeigt, daß die Zahl der Zustände des Tomita-Parsers exponentiell mit der Größe der Grammatik ansteigen kann, und daß es Eingabestrings gibt, bei denen alle diese Zustände tatsächlich durchlaufen werden.

Ein Beispiel ist die folgende Folge von Grammatiken mit $m > 0$:

$$\begin{array}{ll} S \rightarrow A_i & 0 \leq i \leq m \\ A_i \rightarrow B_j\ A_i & 0 \leq i,j \leq m; i \neq j \\ A_i \rightarrow B_j & 0 \leq i,j \leq m; i \neq j \\ B_j \rightarrow a & 0 \leq j \leq m \end{array} \tag{1}$$

Wiederum erzeugen alle diese Grammatiken die Sprache a^+. Da diese Grammatiken mehrdeutig sind, sind sie nicht LR(k) für beliebiges k.

Die Items des Startzustandes sind folgende:

$$\left[\begin{array}{l} S \rightarrow \cdot A_i \\ A_i \rightarrow \cdot B_j\ A_i \\ A_i \rightarrow \cdot B_j \\ B_j \rightarrow \cdot a \end{array}\right] \quad 0 \leq i, j \leq m; i \neq j \tag{2}$$

Nach dem Shiften des ersten Eingabesymbols a befindet sich der Parser in dem Zustand

$$[B_j \rightarrow a\cdot] \quad 0 \leq j \leq m \tag{3}$$

Dieser Zustand enthält $m+1$ Items und erlaubt ebensoviele verschiedene Reduktionen. Wenn eine Reduktion zu dem Symbol $B_{k_1} \rightarrow \alpha$ gewählt wird, so geht der Parser in den folgenden Zustand über:

$$\left[\begin{array}{l} A_i \rightarrow B_{k_1} \cdot A_i \\ A_i \rightarrow B_{k_1} \cdot \\ A_i \rightarrow \cdot B_j\ A_i \\ A_i \rightarrow \cdot B_j \\ B_j \rightarrow \cdot a \end{array}\right] \quad 0 \leq i, j \leq m; i \neq j, k_1 \tag{4}$$

Nach dem Shiften des nächsten Eingabesymbols erreicht der Parser wieder den in (3) gezeigten Zustand. Wenn anschließend zu B_{k_2} reduziert wird, so wird für $B_{k_1} = B_{k_2}$ wieder der in (4) gezeigte Zustand erreicht, andernfalls wird folgender Zustand erreicht:

$$\left[\begin{array}{l} A_i \rightarrow B_{k_2} \cdot A_i \\ A_i \rightarrow B_{k_2} \cdot \\ A_i \rightarrow \cdot B_j\ A_i \\ A_i \rightarrow \cdot B_j \\ B_j \rightarrow \cdot a \end{array}\right] \quad 0 \leq i, j \leq m; i \neq j, k_1, k_2 \tag{5}$$

Aufgrund der Bedingung $i \neq k_1$ enthält (5) weniger Items als (4). Insgesamt gibt es $m(m+1)/2$ verschiedene Zustände der Form (5).

Wenn $n \geq m$ Symbole gelesen und zu $B_{k_1} \ldots B_{k_n}$ reduziert wurden, ist folgender Zustand erreicht:

$$\left[\begin{array}{l} A_i \rightarrow B_{k_n} \cdot A_i \\ A_i \rightarrow B_{k_n} \cdot \\ A_i \rightarrow \cdot B_j\ A_i \\ A_i \rightarrow \cdot B_j \\ B_j \rightarrow \cdot a \end{array}\right] \quad 0 \leq i, j \leq m; i \neq j, k_1, \ldots, k_n \tag{6}$$

Die Sequenz $k_1, \ldots, k_n$ umfaßt zwischen 1 und m viele verschiedene Elemente. Es gibt $\binom{m+1}{j}$ viele Zustände bei j vielen verschiedenen Elementen in der Sequenz. Jeder Zustand ist durch die Menge $I \subset \{0, 1, ..., m\}$ der bereits aufgetretenen Indizes charakterisiert. Insgesamt gibt es 2^{m+1} mögliche Teilmengen der Menge $\{0, 1, ..., m\}$. Nur für zwei dieser Mengen (die leere Menge und die Menge $\{0, 1, ..., m\}$ selbst) gibt es keinen entsprechenden Zustand. Die Zahl der Zustände der

Form (5) ist somit $2^{m+1}-2$. Daher muß der Tomita-Parser mindestens $2^{m+1}-2$ Operationen pro Eingabesymbol durchführen, nachdem mindestens m Eingabesymbole gelesen wurden. Für die asymptotische Zeitkomplexität läßt sich somit $\Omega(2^m)$ als untere Schranke angeben. Da die Größe der Grammatik (= Zahl der Punktregeln) $|G_m| = (3+2)(m^2-m)+(2+2)m = 5m^2-m < 6m^2$ beträgt, läßt sich die Zeitkomplexität auch durch $\Omega(c^{\sqrt{|G_m|}})$ mit $c = 2^{1/6} \approx 1,3$ nach unten abschätzen.

Literatur

[Aho/Ullman 72] Aho, Ullman: *The Theory of Parsing, Translation, and Compiling. Volume 1 (Parsing)*. Prentice Hall, Englewood Cliffs, NJ, 1972.

[Aho/Sethi/Ullman 86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman: *Compilers. Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA., 1986.

[Aho/Sethi/Ullman 88] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman: *Compilerbau* (Teil 1). Addison-Wesley, Reading, MA., 1986.

[Earley 70] Jay Earley (1970): An efficient context-free parsing algorithm. In: *Communications of the ACM* 13, 2, 94-102.

[Graham/Harrison/Ruzzo 80] Graham, Harrison, Ruzzo (1980) An Improved Context-Free Recognizer. In: *ACM Transactions on Prog. Lang. and Sys.* 2(3)

[Johnson 91] Mark Johnson: The computational complexity of Tomita's algorithm. In: *Generalized LR Parsing*. Masaru Tomita (ed.), pp. 43–59. Kluwer Academic Publishers, Boston, 1991.

[Hopcroft/Ullman 79] Hopcroft/Ullman: *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, Reading, MS, 1979.

[Kipps 91] James R. Kipps: Analysis of Tomita's algorithm for general context-free parsing. In: *Generalized LR Parsing*. Masaru Tomita (ed.), pp. 43–59. Kluwer Academic Publishers, Boston, 1991.

[Mayer 78] Otto Mayer: *Syntaxanalyse*. BI, 1982.

[Nederhof 91] Mark-Jan Nederhof: An Optimal Tabular Parsing Algorithm, *ACL 94*.

[Schabes 91] Yves Schabes: Polynomial Time and Space Shift-Reduce Parsing of Arbitrary Context-free Grammars, *ACL 91*.

[Tomita 86] M. Tomita: *Efficient Parsing for Natural Languages*. Kluwer Academic Publishers, Boston, 1986.

[Tomita 87] M. Tomita: An Efficient Augmented-Context-Free Parsing Algorithm. In: *Computational Linguistics*, 13(1–2):31–46, 1987.

[TomitaNg 91] M. Tomita und S.-K. Ng.: The Generalized LR Parsing Algorithm. In: *Generalized LR Parsing*. Masaru Tomita (ed.), pp. 1–16. Kluwer Academic Publishers, Boston, 1991.