Parsingalgorithmen für kontextfreie Sprachen

Bachelorarbeit im Studiengang Informatik
Hannover, 10. Juli 2019
Alexander Koch

Erstprüfer:	Prof. Dr. Heribert Vollmer
Zweitprüfer:	Dr. Arne Meier
Betreuer:	Martin Lück
Matrikelnummer:	10004714

Erklärung der Selbstständigkeit

Hiermit versichere ich, die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet zu haben. Die Arbeit hat in gleicher oder ähnlicher Form noch keinem anderen Prüfungsamt vorgelegen.

Hannover, den 10. Juli 2019

Alexander Koch

Inhaltsverzeichnis

1 Einleitung

Die Formalismen kontextfreier Grammatiken wurden Ende der 1950er Jahre durch den Linguisten Noam Chomsky als eine Art von formalen Grammatiken definiert, auch bezeichnet als Phrasenstrukturgrammatiken. Diese sind auch als Konstituentengrammatiken bekannt, die im Gegensatz zu den Dependenzgrammatiken auf dem Prinzip der Konstituenz aufbauen. Ein Satz wird hierfür in mehrere Teileinheiten, den sogenannten Konstituenten zerlegt. Es werden Regeln definiert, die genau beschreiben, wie ein Konstituent zerlegt werden kann. Sprachen, die durch diese kontextfreien Phrasenstrukturgrammatiken generiert werden, heißen kontextfreie Sprachen.

Kontextfreie Grammatiken sind einfache, aber ausdrucksstarke Formalismen, die es ermöglichen, natürliche Sprachen, wenn auch begrenzt, zu beschreiben und effizient zu analysieren. Außerdem werden sie auch zur Beschreibung von künstlichen Sprachen, wie z. B. Programmiersprachen verwendet. So wurde die Syntax der Programmiersprache ALGOL 60 als erste mithilfe einer formalen Grammatik in Backus-Naur-Form beschrieben.

Heute werden kontextfreie Grammatiken z. B. in der Auswertung von natürlichsprachlichen Texten, wie in z. B. Wortkorrektur- und Grammatikanalysesystemen verwendet, in der Linguistik zur Beschreibung von Satzstrukturen und auch in der Biologie zur Modellierung von RNA.

Diese Bachelorarbeit befasst sich mit Parsingalgorithmen für kontextfreie Sprachen. Es wird für einen beliebigen Satz bzw. ein Wort betrachtet, ob und wie es von einer gegebenen Grammatik erzeugt werden kann.

Zuerst werden die LL(k)- und LR(k)-Parser eingeführt, welche in linearer Laufzeit das Wortproblem entscheiden können. Diese Verfahren sind allerdings auf echte Teilmengen der kontextfreien Sprachen beschränkt. Um die Gesamtheit aller kontextfreien Sprachen zu parsen, werden anschließend Parsingalgorithmen von Earley und Harrison vorgestellt. Diese nutzen das Konzept der dynamischen Programmierung und lösen des Wortproblem in kubischer Laufzeit. Zum Schluss wird ein Softwaretool entwickelt, das die vorgestellten Verfahren implementiert.

2 Theoretische Grundlagen

2.1 Grammatiken und kontextfreie Sprachen

Ein Alphabet ist eine endliche nichtleere Menge von Symbolen. Ist Σ ein Alphabet, so wird eine Folge von Symbolen als Wort bezeichnet. Für die Gesamtheit aller nichtleeren Worte über Σ schreibt man Σ^+. Um das leere Wort darzustellen wird ε verwendet. Man verwendet $\Sigma^* := \Sigma^+ \cup \{\varepsilon\}$ um alle Wörter über Σ darzustellen. Eine Sprache bezeichnet eine Menge von Wörtern aus Σ^*.

Definition 1. *Eine Grammatik ist ein 4-Tupel $G = (V, \Sigma, P, S)$ bestehend aus einer endlichen Menge von Variablen V, einem Alphabet von Terminalsymbolen Σ, sodass gilt $V \cap \Sigma = \emptyset$, einer endlichen Menge von Produktionen $P \subseteq (V \cup \Sigma)^+ \times (V \cup \Sigma)^*$ und einer Startvariable $S \in V$. Variablen werden auch als Nichtterminalsymbole bezeichnet.*

Für kontextfreie Grammatiken werden die Produktionen in der Form $A \to \alpha$ mit $A \in V$ und $a \in (V \cup \Sigma)^*$ angegeben. Die linke Seite einer Produktion darf jeweils nur aus einem Nichtterminalsymbol bestehen, d. h. unabhängig vom Kontext darf A durch α ersetzt werden. Die rechte Seite der Produktion α kann eine beliebige Folge von Nichtterminal- und Terminalsymbolen sein, sowie auch das leere Wort.
In alternativen Definitionen wird der Gebrauch von ε-Produktionen eingeschränkt und $\alpha \in (V \cup \Sigma)^+$ verwendet. Falls das leere Wort ein Teil der Sprache sein soll, darf $S \to \varepsilon$ weiterhin existieren, aber S darf in keiner rechten Seite einer Produktion vorkommen. Es kann jedoch gezeigt werden, dass für jede Sprache L, die durch eine kontextfreie Grammatik mit ε-Regeln erzeugt wird, eine äquivalente kontextfreie Grammatik ohne ε-Regeln mit Ausnahme von $S \to \varepsilon$ existiert [Har78].

Definition 2. *Sei $G = (V, \Sigma, P, S)$ eine Grammatik. Man schreibt $u \Rightarrow_G v$, falls $x, z \in (V \cup \Sigma)^*$ existieren mit $u = xyz$ und $v = xy'z$, sowie $y \to y'$ eine Regel in P ist. Des Weiteren gilt $u \Rightarrow_G^* v$, falls $u = v$ oder es Wörter $w_1, w_2, ..., w_n \in (V \cup \Sigma)^*$ gibt, sodass gilt:*

$$u = w_1 \Rightarrow w_2 \Rightarrow ... \Rightarrow w_n = v$$

Die Folge $(w_1, w_2, ..., w_n)$ heißt Ableitung. Falls eindeutig ist, welche Grammatik für die Ableitung verwendet wird, so kann der Index G kann weggelassen werden. Die von G erzeugte Sprache ist $L(G) = \{ w \in \Sigma^* \mid S \Rightarrow_G^* w \}$.

Definition 3. *Eine Sprache heißt kontextfrei, wenn sie von einer kontextfreien Grammatik erzeugt werden kann.*

Definition 4. *Als Linksableitung bzw. Rechtsableitung wird eine Folge von Ableitungsschritten bezeichnet, bei denen das jeweils am weitesten links bzw. rechts stehende Nichtterminalsymbol durch Anwendung einer Produktion ersetzt wird.*

Definition 5. *Als Satzform (engl. sentential form) wird eine Zeichenfolge $s \in (V \cup \Sigma)^*$ bezeichnet für die gilt: $S \Rightarrow^* s$. Eine Satzform heißt Linkssatzform bzw. Rechtssatzform, wenn eine Linksableitung bzw. eine Rechtsableitung von S nach s existiert. Die Menge aller Satzformen einer Grammatik G ist eine echte Obermenge von $L(G)$.*

2.2 Parsing und Syntaxanalyse

Definition 6. *Ein Ableitungsbaum ist ein geordneter Baum $(K, E, <)$ mit einer Markierungsabbildung $f : K \mapsto (V \cup \Sigma)^*$ für den folgende Bedingungen gelten:*

1. *Für den Wurzelknoten $r \in K$ gilt: $f(r) \in V$*
2. *Ist x ein Knoten und $\{ k_1, k_2, ..., k_n \}$ die Menge seiner direkten Nachfolger mit $k_i < k_{i+1}$ für $1 \leq i < n$, dann ist $f(x) \to f(k_1)f(k_2)...f(k_n)$ eine Produktionsregel in P.*

Beispiel 1. *Sei* $G = (\{S, \mathrm{NP}, \mathrm{VP}, \mathrm{Det}, \mathrm{N}, \mathrm{Verb}\}, \{\mathrm{the}, \mathrm{ate}, \mathrm{cat}, \mathrm{homework}\}, P, S)$
NP *steht hier für „Noun Phrase“,* VP *für „Verb Phrase“,* Det *für „Determinant“ und* N *für „Noun“.*

Produktionsregeln:

$S \to \mathrm{NP\,VP}$
$\mathrm{NP} \to \mathrm{Det\,N}$
$\mathrm{VP} \to \mathrm{Verb\,NP}$
$\mathrm{Det} \to \mathrm{the}$
$\mathrm{N} \to \mathrm{cat}$
$\mathrm{N} \to \mathrm{homework}$
$\mathrm{Verb} \to \mathrm{ate}$

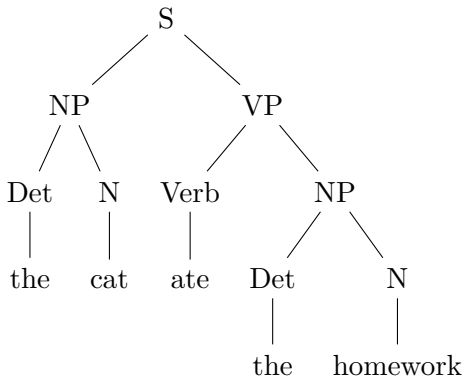

Abbildung 1: Ableitungsbaum für „the cat ate the homework“

Definition 7. *Das Entscheidungsproblem, festzustellen, ob ein gegebenes Wort bestehend aus einer beliebigen Zeichenfolge $w \in \Sigma^*$ in $L(G)$ liegt, wird als Wortproblem bezeichnet. Es wird auch Erkennungsproblem genannt.*

Um zu überprüfen, ob ein Wort Teil einer Sprache ist bzw. von einer Grammatik erzeugt werden kann, wird ein Recognizer oder Parser verwendet. Ein Recognizer ist ein Algorithmus, der das Wortproblem entscheidet. Um die genaue syntaktische Struktur des Eingabewortes bezüglich der Grammatik zu beschreiben, wird ein Parser verwendet. Dieser ist ein Recognizer, der zusätzlich zur Überprüfung alle Ableitungsbäume des Eingabeworts erzeugt, falls das Wort in $L(G)$ liegt. Die Bestimmung der syntaktischen Struktur wird auch Syntaxanalyse bzw. Parsing genannt.

Definition 8. *Eine Grammatik $G = (V, \Sigma, P, S)$ liegt in Chomsky-Normalform vor, wenn sie nur Produktionen der folgenden Gestalt enthält:*

(1) $A \to BC$, *mit* $A, B, C \in V$
(2) $A \to a$, *mit* $A \in V$ *und* $a \in \Sigma$
(3) $S \to \varepsilon$, *falls* S *in keiner Produktion auf der rechten Seite auftritt*

Beispiel 2. $G = (\{E\}, \{a, *, +\}, P, E)$
P sei die Menge der folgenden Produktionen:

$$E \to E + E$$
$$E \to E * E$$
$$E \to a$$

G' sei eine Chomsky-Normalform zu G $G' = (\{E, A, M, B, C\}, \{a, *, +\}, P', E)$
P' sei die Menge der folgenden Produktionen:

$$E \to E\,B$$
$$B \to A\,E$$
$$E \to E\,C$$
$$C \to M\,E$$
$$E \to a$$
$$A \to +$$
$$M \to *$$

Definition 9. *Eine Grammatik G heißt eindeutig, falls für jedes Wort $w \in L(G)$ nur eine mögliche Ableitung aus der Startvariable existiert.*

Beispiel 3. $G = (\{E\}, \{a, *, +\}, P, E)$
P sei die Menge der folgenden Produktionen:

$$E \to E + E$$
$$E \to E * E$$
$$E \to a$$

G ist keine eindeutige Grammatik, da für das Wort $w = a + a * a$ zwei Ableitungsbäume von der Startvariable aus existieren.

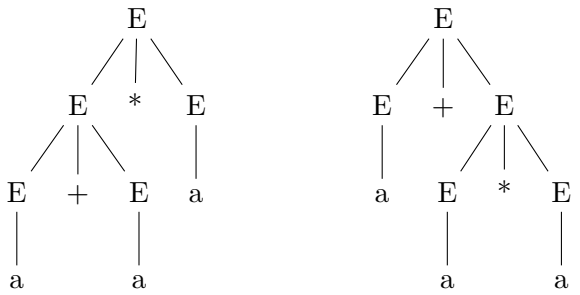

Abbildung 2: Ableitungsbäume für $a + a * a$

Definition 10. *Sei $G = (V, \Sigma, P, S)$ eine kontextfreie Grammatik und $\cdot$ ein Trennsymbol, das nicht in V auftritt. Eine geteilte Produktion (engl. dotted rule) ist eine Datenstruktur, die während des Parsings verwendet wird um eine partielle Lösung einer Ableitung zu speichern. Sie werden in der Form $A \to \alpha \cdot \beta$ angegeben. Das Trennsymbol stellt den Fortschritt der Verarbeitung dar.*

3 LL(k)- und LR(k)-Parser

Definition 11. *Sei* $w = w_1 w_2 ... w_n$, $n \in \mathbb{N}_0$

$$k{:}w := \begin{cases} w, & |w| \leq k \\ w_1 w_2 ... w_k, & \text{sonst} \end{cases}$$

Definition 12. $\mathrm{FIRST}_k(\alpha)$ *ist die Menge aller Zeichenketten, die von* α *durch die Grammatik* G *ableitbar sind und deren Länge kleiner oder gleich* k *ist, mit* $\alpha \in (V \cup \Sigma)^*$. *Für das leere Wort ist* $\mathrm{FIRST}_k(\varepsilon) = \{\varepsilon\}$. *Falls* ε *durch* α *ableitbar ist, enthält* $\mathrm{FIRST}_k(\alpha)$ *auch das leere Wort.*

$$\mathrm{FIRST}_k(\alpha) := \{\, k{:}w \mid \alpha \Rightarrow^* w,\, w \in \Sigma^* \,\} \cup \{\, \varepsilon \mid \alpha \Rightarrow^* \varepsilon \,\}$$

Definition 13. $\mathrm{FOLLOW}_k(A)$ *ist die Menge aller Zeichenketten* $a \in \Sigma^+$, *deren Länge kleiner oder gleich* k *ist, die in einer Satzform unmittelbar auf* A *folgen können, so das eine Ableitung der Form* $S \Rightarrow^* \alpha A a \beta$ *existiert. Falls auf das Nichtterminalsymbol* A *keine Zeichen folgen können und somit das Ende der Eingabe vorliegt, so enthält* $\mathrm{FOLLOW}_k(A)$ *die Endmarke* $\dashv$.

$$\begin{aligned} \mathrm{FOLLOW}_k(A) := \{\, \mathrm{FIRST}_k(\beta) \mid S \Rightarrow^* \alpha A \beta,\, \alpha \in \Sigma^*,\, \beta \in (V \cup \Sigma)^* \,\} \\ \cup \{\, \dashv \mid S \Rightarrow^* \alpha A,\, \alpha \in (V \cup \Sigma)^* \,\} \end{aligned}$$

3.1 LL(k)- und LR(k)-Sprachen

Sei $G = (V, \Sigma, P, S)$ eine kontextfreie Grammatik. G heißt LL(k)-Grammatik, wenn für jede Linkssatzform $xA\delta$, mit $x \in \Sigma^*$, $\delta \in (V \cup \Sigma)^*$ gilt: falls die Produktionen $A \to \alpha_1$, $A \to \alpha_2$ existieren, so müssen die jeweiligen FIRST_k-Mengen disjunkt sein.

$$\mathrm{FIRST}_k(\alpha_1 \delta) \cap \mathrm{FIRST}_k(\alpha_2 \delta) = \emptyset$$

G nennt man starke LL(k)-Grammatik oder SLL(k)-Grammatik [RS70], falls gilt:

$$\mathrm{FIRST}_k(\alpha_1 \mathrm{FOLLOW}_k(A)) \cap \mathrm{FIRST}_k(\alpha_2 \mathrm{FOLLOW}_k(A)) = \emptyset$$

Das Hintereinanderschreiben von $\alpha_1 \text{FOLLOW}_k(A)$ stellt hier die Menge der Konkatenationen von α_1 mit jedem Element aus $\text{FOLLOW}_k(A)$ dar (analog für $\alpha_2 \text{FOLLOW}_k(A)$). Eine Sprache wird LL($k$)-Sprache genannt, wenn es eine LL(k)-Grammatik gibt, die sie erzeugt. Die Menge der LL(k)-Sprachen ist eine echte Teilmenge der LL($k+1$)-Sprachen für jedes $k \geq 0$ [RS70].

$$\text{LL}(0) \subsetneq \text{LL}(1) \subsetneq \text{LL}(2) \subsetneq ... \subsetneq \text{DCFL}^1$$

Ist eine Grammatik eine LL(k)-Grammatik, so kann ein LL(k)-Parser konstruiert werden. Dieser analysiert ein Eingabewort, indem er von links nach rechts arbeitet um eine Linksableitung der Eingabe zu erzeugen (daher die Abkürzung LL). Jeder Ableitungsschritt ist dabei eindeutig durch die k Vorschausymbole (engl. *lookahead*) bestimmt. Es handelt sich um ein Top-Down Verfahren, bei der vom Wurzelknoten des Ableitungsbaumes gestartet wird, bis die Ebene der Blätter erreicht wird. Besonders oft werden LL(1)-Sprachen verwendet, da es für diese Sprachen sehr einfach ist einen Parser zu schreiben. Eine Implementierung kann entweder über Parsingtabellen oder über rekursiven Abstieg (engl. *recursive descent*) erfolgen.

Beispiel 4. *Arithmetische LL(1)-Grammatik,* $G = (\{E, E', T, T', F\}, \{+, *, a\}, P, E)$
P sei die Menge der folgenden Produktionen:

(1) $E \to TE'$
(2) $E' \to +TE'$
(3) $E' \to \varepsilon$
(4) $T \to FT'$
(5) $T' \to *FT'$
(6) $T' \to \varepsilon$
(7) $F \to (E)$
(8) $F \to a$

[1] DCFL - Deterministische kontextfreie Sprachen

$\text{FIRST}(T') = \{\, \varepsilon, * \,\}$
$\text{FIRST}(T) = \text{FIRST}(F) = \text{FIRST}(E) = \{\, (, a \,\}$
$\text{FIRST}(E') = \{\, +, \varepsilon \,\}$
$\text{FOLLOW}(T) = \text{FOLLOW}(T') = \{\, \dashv, +,) \,\}$
$\text{FOLLOW}(F) = \{\, *, \dashv, +,) \,\}$
$\text{FOLLOW}(E) = \text{FOLLOW}(E') = \{\, \dashv,) \,\}$

Linksableitung für $w = a + a$:

$$\begin{aligned} E &\overset{(1)}{\Rightarrow} TE' \overset{(4)}{\Rightarrow} FT'E' \overset{(8)}{\Rightarrow} aT'E' \\ &\overset{(6)}{\Rightarrow} aE' \overset{(2)}{\Rightarrow} aTE' \overset{(4)}{\Rightarrow} a + FT'E' \\ &\overset{(8)}{\Rightarrow} a + aT'E' \overset{(6)}{\Rightarrow} a + aE' \overset{(3)}{\Rightarrow} a + a \end{aligned}$$

G heißt LR(k)-Grammatik, falls ein deterministischer LR(k)-Parser für G existiert und falls die folgenden Bedingungen

$$\begin{aligned} &S \underset{rm}{\Rightarrow}^* \alpha A z_1 \underset{rm}{\Rightarrow} \alpha\beta z_1 \\ &S \underset{rm}{\Rightarrow}^* \alpha' B z_2 \underset{rm}{\Rightarrow} \alpha'\beta' z_2 = \alpha\beta z_2' \\ &\text{FIRST}_k(z_1) = \text{FIRST}_k(z_2') \end{aligned}$$

implizieren, dass $\alpha = \alpha'$, $A = B$ und $\beta = \beta'$ für $A, B \in V$, $\alpha, \beta, \alpha', \beta' \in (V \cup \Sigma)^*$ und $z_1, z_2, z_2' \in \Sigma^*$ [Soi80].
LR-Parser erzeugen im Gegensatz zum LL-Parser eine Rechtsableitung und verwenden ein Bottom-Up Verfahren. Eine Reduktion wird erst ausgeführt, wenn eine vollständige Regel erkannt wurde. Die Reduktion bzw. der Ableitungsschritt ist wiederum eindeutig durch die k Vorschausymbole bestimmt.
Die durch LL(k)-Grammatiken erzeugten Sprachen sind eine echte Teilmenge der LR(k)-Sprachen [May86]. Jede LL(k)-Grammatik ist eine LR(k)-Grammatik [RS70].
Während die LL(k)-Sprachen jeweils echte Teilmengen der nächst höheren LL($k+1$)-Sprachen sind, gilt dies nicht für LR(k) Sprachen. Für jede LR(k)-Sprache mit $k > 1$ gibt es eine LR(1)-Grammatik, die durch Umformung der entsprechenden LR(k)-Grammatik erzeugt werden kann [Knu65].

Jede Sprache L ist eine deterministiche kontextfreie Sprache, genau dann wenn L eine LR(k)-Grammatik besitzt für $k \geq 1$. Eine Sprache L besitzt eine $LR(0)$-Grammatik, genau dann wenn L deterministisch kontextfrei ist und die Präfix-Eigenschaft erfüllt, d. h. kein Wort ist das Präfix eines anderen Wortes der Sprache [HU90]. LR(0)-Sprachen sind daher eine echte Teilmenge der LR(1)-Sprachen. Sie sind schwächer, da sie keine Vorschau besitzen und nur Entscheidungen basierend auf den bisher verarbeitenden Produktionen treffen.

$$\text{LR}(0) \subsetneq \text{LR}(1) = \text{LR}(2) = ... = \text{LR}(k)$$

Durch LR(k)-Parser können alle deterministischen kontextfreien Grammatiken erkannt werden.

$$\text{LL}(0) \subsetneq \text{LL}(1) \subsetneq ... \subsetneq \text{LL}(k) \subsetneq \text{LR}(1) = \text{LR}(2) = ... = \text{LR}(k) = \text{DCFL} \subsetneq \text{CFL}^2$$

Beispiel 5. *LR(1)-Grammatik,* $G = (\{S, P\}, \{a, b\}, P, S)$
P sei die Menge der folgenden Produktionen:

$$S \to aS \mid A$$
$$A \to aAb \mid \varepsilon$$

Beispielwörter: $w_1 = aaaabb, w_2 = aaa, w_3 = ab$, *...*
Die von G erzeugt Sprache ist $L(G) = \{\, a^i b^j \mid i \geq j \,\}$. *Es existiert keine äquivalente LL(k)-Grammatik, da nicht entschieden werden kann, ob bei einem „a“ ein Paar „ab“ oder ein einfaches „a“ eingelesen und verarbeitet werden soll. Ein LR(k)-Parser kann dies verarbeiten, da ein „b“ oder das Ende der Eingabe vorliegt.*

3.2 LL(k)-Parsingtabellen

Da der LL(k)-Parser Linksableitungen erzeugt und Anwendung der Regeln nur aufgrund der k Vorschausymbole entscheidet, ist es sehr leicht eine Parsingtabelle für alle Möglichkeiten zu entwickeln.
Es wird wie folgt vorgegangen:

[2] CFL - Kontextfreie Sprachen

Algorithmus 1 Erzeugung einer LL(k)-Parsingtabelle

Eingabe : LL(k) Grammatik $G = (V, \Sigma, P, S)$

Ausgabe : Matrix M

Berechne die FIRST_k und FOLLOW_k Mengen aller Produktionen.

foreach $A \to \alpha \in P$ **do**
  foreach $a \in \text{FIRST}_k(\alpha)$, *mit* $a \in \Sigma^+$ **do**
    $M[A, a] := A \to \alpha$
  if $\varepsilon \in \text{FIRST}_k(\alpha)$ **then**
    foreach $b \in \text{FOLLOW}_k(A)$ **do**
      $M[A, b] := A \to \alpha$

Für jede Produktion $A \to \alpha \in P$ wird für jedes $\text{FIRST}_k(\alpha)$-Symbol aus α der Eintrag $A \to \alpha$ in die Tabelle M an der Position $M[A, a]$ eingefügt. Befindet sich das leere Wort in der FIRST_k-Menge, so wird zusätzlich für jedes Symbol aus der FOLLOW_k-Menge von A der Eintrag $A \to \alpha$ in die Tabelle an der Position $M[A, b]$ eingefügt.
Um nun für ein Wort den Parsingalgorithmus auszuführen, wird eine Stack-Maschine verwendet. Dazu wird zum Start das Startsymbol auf den Stack gelegt. Nun wird in jedem Schritt das oberste Stack-Symbol mit dem derzeitigen Eingabezeichen verglichen. Handelt es sich um ein Nichtterminalsymbol, so wird die entsprechende Regel in der Parsingtabelle gesucht und das Stack-Symbol durch die rechte Seite der gefundenen Produktion ersetzt. Damit das Wort korrekt weiterverarbeitet werden kann, müssen hierfür die Symbole der Produktion in umgekehrter Reihenfolge abgelegt werden. Befindet sich nun ein Terminalsymbol auf dem Stack und stimmt es mit dem Eingabezeichen überein, so kann das Element jeweils vom Stack und von der Eingabe entfernt werden. Dies wird solange wiederholt, bis eine Fehlersituation auftritt, oder auf dem Stack und in der Eingabe keine Zeichen mehr vorhanden sind und die Eingabe akzeptiert wird.

Beispiel 6. *Sei* $G = (\{S, A\}, \{a, b, c\}, P, S)$ *eine kontextfreie LL(2)-Grammatik.*
P sei die Menge der folgenden Produktionen:

$$S \to aSA \mid \varepsilon$$
$$A \to abS \mid c$$

Die Grammatik erzeugt eine LL(2)-Sprache, jedoch keine LL(1)-Sprache, da nur durch ein Lookahead von 2 entschieden werden kann, ob die Produktion $S \to aSA$ *oder* $A \to abS$

angewendet werden soll.

$\text{FIRST}_2(A) = \{\, ab,\, c \,\}$
$\text{FIRST}_2(S) = \{\, aa,\, ac,\, \varepsilon \,\}$
$\text{FOLLOW}_2(S) = \{\, ab,\, c,\, ca,\, cc,\, \dashv \,\}$

	Vorschau						
Variable	c	ab	aa	ac	ca	cc	$\dashv$
S	$S \to \varepsilon$	$S \to \varepsilon$	$S \to aSA$	$S \to aSA$	$S \to \varepsilon$	$S \to \varepsilon$	$S \to \varepsilon$
A	$A \to c$	$A \to abS$					

Tabelle 1: Parsingtabelle

Beispiel für Eingabewort $w = aaabc$
Der Stack beginnt mit der Endmarke und dem darauffolgenden Startsymbol S. An das Ende der Eingabe wird ebenfalls die Endmarke angehängt.

Stack	Eingabe	anzuwendende Produktion
$\vdash S$	$aaabc \dashv$	$S \to aSA$
$\vdash ASa$	$aaabc \dashv$	
$\vdash AS$	$aabc \dashv$	$S \to aSA$
$\vdash AASa$	$aabc \dashv$	
$\vdash AAS$	$abc \dashv$	$S \to \varepsilon$
$\vdash AA$	$abc \dashv$	$A \to abS$
$\vdash ASba$	$abc \dashv$	
$\vdash AS$	$c \dashv$	$S \to \varepsilon$
$\vdash A$	$c \dashv$	$A \to c$
$\vdash$	$\dashv$	

Tabelle 2: LL-Parsing für Eingabewort

Das Wort $w = aaabc$ ist in $L(G)$ und wird akzeptiert, da sich weder in der Eingabe noch

auf dem Stack ein Symbol befindet. Beide beinhalten nur noch die Endmarke.

3.3 LR(k)-Parsingtabellen

Der LR(k)-Parser erzeugt Rechtsableitungen mit k Vorschausymbolen und arbeitet im Gegensatz zum LL(k)-Parser Bottom-Up, d. h. die Ableitung wird zunächst nicht vom Startsymbol erzeugt, sondern direkt von den Eingabezeichen, den Blättern des Ableitungsbaumes. Der Parser versucht Teilwörter zu finden, die mit der rechten Seite der Produktionen übereinstimmen. Konnte ein passendes Teilwort gefunden werden, so wird eine Reduktion durchgeführt, eine Ersetzung mit der linken Seite der Produktion. Dies wird wiederholt, bis durch Reduktion das Startsymbol erreicht wird.
Ein Bottom-Up Verfahren ist das Shift-Reduce Parsing. Die Produktionen werden hierbei in umgekehrter Reihenfolge zum Top-Down Verfahren entdeckt. Wie bei dem LL(k)-Parser wird wieder ein Stack verwendet, um den Parsingzustand zu speichern, sowie eine Tabelle um den nächsten Zustandsübergang zu berechnen. Die Eingabe wird in zwei Teile aufgeteilt, den verarbeiteten und den unverarbeiteten Teil. Für jeden Schritt wird eine der zwei Operationen „Shift“ oder „Reduce“ ausgeführt.
Bei einer Shift-Operation werden Zeichen von der unverarbeiteten Eingabe eingelesen und auf dem Stack abgelegt, sofern keine Reduktion ausgeführt werden konnte. Es wird eine Reduce-Operation ausgeführt, falls man eine Produktion $A \to a$ finden kann und der Inhalt des Stacks αa ist, sodass man eine Reduktion zu αA ausführen kann. a bzw. $A \to a$ wird Handle von $\alpha a \beta$ genannt, falls gilt $S \Rightarrow^* \alpha A \beta \Rightarrow^* \alpha a \beta$ ist eine Rechtsableitung einer kontextfreien Grammatik.
Anstatt nun Eingabezeichen auf den Stack abzulegen, werden bei einem LR-Parser Zustandsnummern abgelegt. Durch den obersten Zustand auf dem Stack sowie durch die Vorschausymbole wird entschieden ob ein Zustandswechsel durch einen Shift stattfindet oder ein Handle gefunden wurde und eine Reduktion durchgeführt werden kann. Ein LR-Parser verwendet hierfür zwei Tabellen: eine Action-Tabelle mit Shift-/Reduce-Einträgen und eine Goto-Tabelle. Die Action-Tabelle kodiert für jeden Zustand S und für jedes Element β der $\text{FIRST}_k(\alpha)$ Menge einen Shift- oder Reduce-Eintrag in $\text{ACTION}[S, \beta]$. Die Goto-Tabelle gibt für den obersten Zustand S auf dem Stack nach einer erfolgreichen Reduktion zu X an, welcher neue Zustand auf den Stack gelegt werden soll ($\text{GOTO}[S, X]$).
Die Zustände eines LR(0)-Parsers bestehen aus Mengen von geteilten Produktionen, den sogenannten LR(0)-Items. Verwendet man ein $k > 0$, so werden die LR(0)-Items zu LR(k)-Items erweitert, die jeweils aus einem LR(0)-Item und einem Lookahead von k Symbolen bestehen. Um die Zustände zu berechnen wird die Closure-Funktion verwendet.

Diese gibt für eine Menge von LR-Items diejenigen zurück, die der Parser als nächstes verarbeiten könnte, ausgehend von dem derzeitigen Fortschritt des jeweiligen Items. Ein Zustandsübergangswechsel findet über eine Shift-Operation oder durch einen Eintrag in Goto-Tabelle statt. Closure sei wie folgt definiert:

Algorithmus 2 Closure

Eingabe : LR(1) Grammatik $G = (V, \Sigma, P, S)$
q, eine Menge von $(A \to \alpha \cdot B\beta, a)$-Tupeln

Wiederhole folgende Anweisungen bis q sich nicht mehr ändert:

foreach $(A \to \alpha \cdot B\beta, a) \in q$ **do**
  foreach $B \to \lambda \in P$ **do**
    foreach $b \in \text{FIRST}_1(\beta a)$ **do**
      $q := q \cup \{(B \to \cdot\lambda, b)\}$

Um nun die Übergänge zwischen den Zuständen zu berechnen kann für ein Symbol $B \in V \cup \Sigma$ die Goto-State-Funktion verwendet werden:

$$\text{Goto-State}(q, B) := \text{Closure}(\{(A \to \alpha B \cdot \beta, a) \mid (A \to \alpha \cdot B\beta, a) \in q\})$$

Diese berechnet für alle Items aus dem Zustand q, deren nächstes Symbol B ist, die Nachfolgerelemente und bildet diese auf den Zustand ab, der durch das Anwenden der Closure-Funktion auf diese Ergebnismenge erreicht wird. Für eine Menge Q von Zuständen kann ausgehend vom Startzustand $q_0 = Closure(\{(S' \to \cdot S, \dashv)\}) \in Q$ jeder weitere Zustand berechnet werden. Jeder Zustand wird eindeutig nummeriert. Um nun die LR(1)-Parsingtabelle zu konstruieren wird wie folgt vorgegangen:

Algorithmus 3 Erzeugung einer LR(1)-Parsingtabelle

Eingabe : Q Menge der Zustände bestehend aus LR(0)-Item Mengen
$\#A \to \alpha$ ist die kodierte Produktionsnummer von $A \to \alpha$

foreach $q_i \in Q$ **do**
  foreach $a \in \Sigma$ **do**
    $q_j :=$ Goto-State(q_i, a)
    ACTION$[i, a] :=$ Shift(j)
  foreach $X \in V$ **do**
    $q_j :=$ Goto-State(q_i, a)
    GOTO$[i, X] :=$ Goto(j)
  if $(A \to \alpha\cdot, a) \in q_i$ **then**
    ACTION$[i, a] :=$ Reduce$(\#A \to \alpha)$
  if $(S' \to S\cdot, \dashv) \in q_i$ **then**
    ACTION$[i, \dashv] :=$ Accept

Beispiel 7. $G = (\{S, C\}, \{c, d\}, P, S')$
P sei die Menge der folgenden Produktionen:

(0) $S' \to S$
(1) $S \to CC$
(2) $C \to cC$
(3) $C \to d$

Ausgehend von $Closure(\{\, (S' \to \cdot S, \dashv)\,\})$ *erhält man folgenden Startzustand:*

$$q_0 = \{\, (S \to \cdot CC, \dashv), (C \to \cdot cC, c), (C \to \cdot cC, d), (C \to \cdot d, c), (C \to \cdot d, d)\,\}$$

Anschließend wird für jedes Nichtterminal- und Terminalsymbol der Nachfolgerzustand berechnet: $Goto\text{-}State(q_0, S) = q_4, Goto\text{-}State(q_0, C) = q_2, ...$
Wiederholt man dies für jeden neu generierten Zustand, erhält man als Resultat folgende Menge von Zuständen (Nummerierung kann abweichen):

q_0: $(S' \to \cdot S, \dashv)$
$(S \to \cdot CC, \dashv)$
$(C \to \cdot cC, c)$
$(C \to \cdot cC, d)$
$(C \to \cdot d, c)$
$(C \to \cdot d, d)$

q_1: $(C \to d\cdot, c)$
$(C \to d\cdot, d)$

q_2: $(S \to C \cdot C, \dashv)$
$(C \to \cdot cC, \dashv)$
$(C \to \cdot d, \dashv)$

q_3: $(C \to \cdot cC, c)$
$(C \to \cdot cC, d)$
$(C \to c \cdot C, c)$
$(C \to c \cdot C, d)$
$(C \to \cdot d, c)$
$(C \to \cdot d, d)$

q_4: $(S' \to S\cdot, \dashv)$

q_5: $(C \to d\cdot, \dashv)$

q_6: $(S \to CC\cdot, \dashv)$

q_7: $(C \to \cdot cC, \dashv)$
$(C \to c \cdot C, \dashv)$
$(C \to \cdot d, \dashv)$

q_8: $(C \to cC\cdot, c)$
$(C \to cC\cdot, d)$

q_9: $(C \to cC\cdot, \dashv)$

	Action			Goto	
Zustand	c	d	$\dashv$	C	S
0	s3	s1		2	4
1	r3	r3			
2	s7	s5		6	
3	s3	s1		8	
4			acc		
5			r3		
6			r1		
7	s7	s5		9	
8	r2	r2			
9			r2		

Stack	Eingabe	Reduktion
0	$cdcd \dashv$	
$0, 3$	$dcd \dashv$	
$0, 3, 1$	$cd \dashv$	(3) $C \to d$
$0, 3, 8$	$cd \dashv$	(2) $C \to cC$
$0, 2$	$cd \dashv$	
$0, 2, 7$	$d \dashv$	
$0, 2, 7, 5$	$\dashv$	(3) $C \to d$
$0, 2, 7, 9$	$\dashv$	(2) $C \to cC$
$0, 2, 6$	$\dashv$	(1) $S \to CC$
$0, 4$	$\dashv$	

Tabelle 3: LR(1)-Parsingtabelle und Parsing für $w = cdcd$

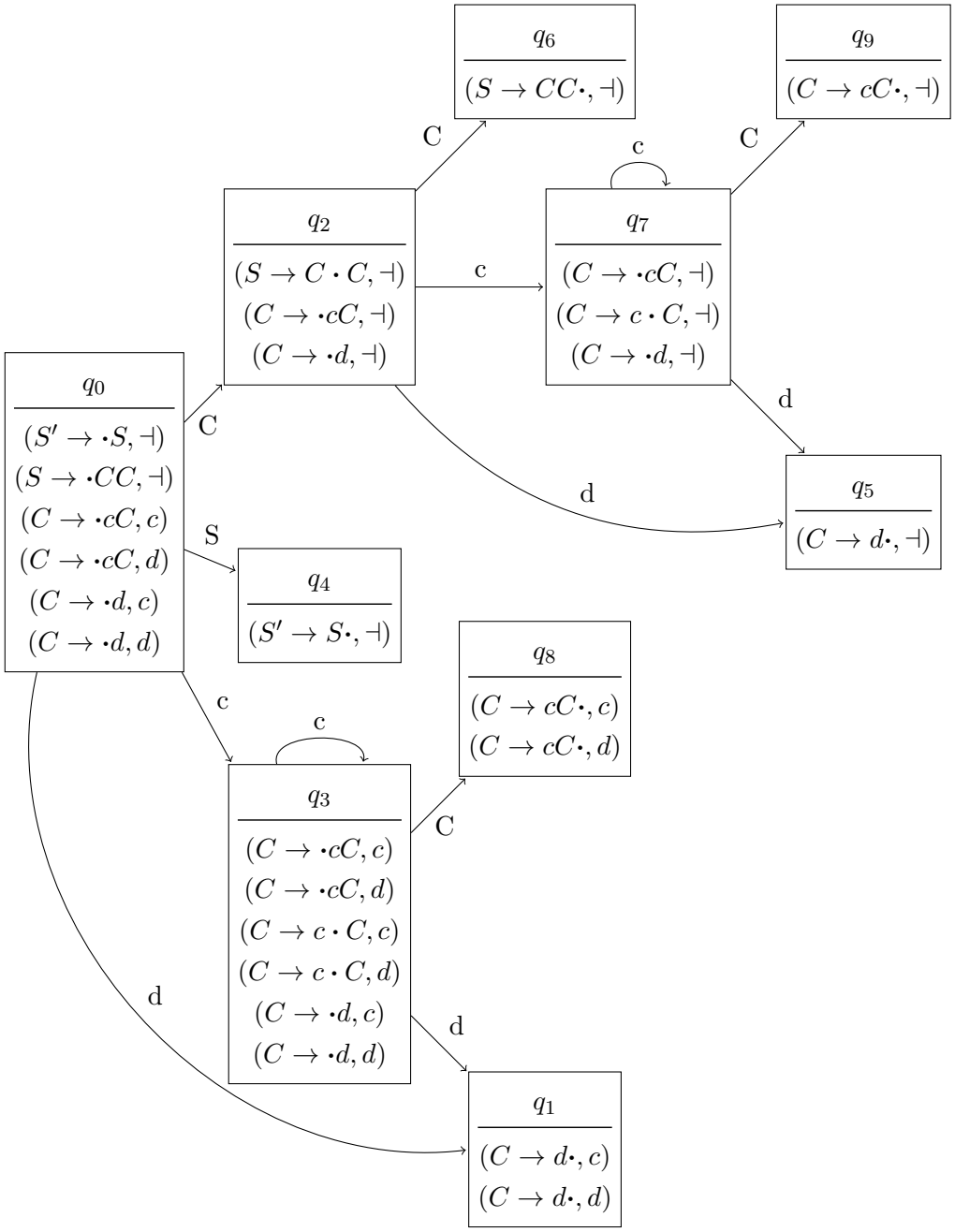

Abbildung 3: LR(1)-Parsingautomat zu Beispiel 7

4 Earley-Algorithmus

Der Earley-Algorithmus berechnet mithilfe von geteilten Produktionen für jeden Schritt aus der gegebenen Grammatik Teilableitungen für das Eingabewort. Es wird gespeichert wie weit die Analyse fortgeschritten ist, welcher Teil einer Regel schon gefunden wurde, sowie die Start- und Endpositionen der Objekte links des Trennsymbols.

4.1 Analysealgorithmus

Der Analysealgorithmus erzeugt für ein Eingabewort $w = w_1 w_2 \dots w_n$ eine Liste von $n + 1$ Zustandsmengen und setzt voraus, dass die Produktionen der Grammatik nummeriert sind. Ein Zustand ist definiert als ein Tripel (p, j, k), wobei p den Index der Produktion beschreibt, j den Index des Trennsymbols innerhalb der Produktion und k die Anfangsposition innerhalb des Wortes, an der das Matching der Produktion begann. Die Zustandsmengen seien als $S_0, S_1, \dots, S_n$ definiert. Earley beschreibt im originalen Algorithmus [Ear68; Ear70] zusätzlich noch ein Vorschau-Attribut. Es hat sich jedoch gezeigt, dass dieses nur wenig zur Verbesserung der Laufzeit beiträgt und wird daher in Implementierungen nicht mehr verwendet. Der Algorithmus besteht aus drei Routinen die für jedes $i \in [0, n]$ ausgeführt werden und die Zustandsmengen modifizieren: dem Predictor, dem Scanner und dem Completer.

Zu Beginn wird die Startvariable S der Grammatik durch eine neue Variable S' ersetzt in der Form $S' \to S$. Dies ist notwendig, damit der Predictor zum Start des Algorithmus alle möglichen alternativen Produktionen für das Startsymbol finden und verarbeiten kann. Danach wird S_0 die Startproduktion $(S' \to \cdot S, 0)$ hinzugefügt.

Algorithmus 4 Earley-Algorithmus

Eingabe : $w = w_0 w_1 ... w_{n-1} \dashv$

Sei S eine Liste von Mengen der Größe $n + 1$

Füge $(S' \to \cdot S, 0)$ zu S_0 hinzu

for $k = 0$ **to** n **do**
  foreach *nicht-markiertes* state $\in S_k$ **do**
    markiere state
    if ¬final(state) **then**
      if next−symbol(state) *ist ein Nichtterminalsymbol* **then**
        predict(state, ...)
      else
        scan(state, w_k, ...)
    else
      complete(state, ...)

if $(S' \to S\cdot, 0) \in S_n$ **then**
  accept
else
  reject

Der Predictor berechnet für einen Zustand $(A \to \alpha \cdot B\beta, k) \in S_i$ alle möglichen Produktionen, die das nächste Nichtterminalsymbol ersetzen können. Dabei werden alle Produktionen der Grammatik durchsucht, die mit dem Nichtterminalsymbol B anfangen und jeweils als $(B \to \cdot\lambda, i)$ zur Zustandsmenge S_i hinzugefügt.

Daraufhin wird der Scanner ausgeführt. Dieser fügt für jede Produktion der Form $(A \to \alpha \cdot a\beta, j)$ im nächsten Berechnungsschritt S_{i+1} die Produktionen hinzu, bei denen ein Terminalsymbol a eingelesen werden kann $(A \to \alpha a \cdot \beta, j)$. Dafür muss sich an der i-ten Stelle der Eingabe ebenfalls das gleiche Terminalsymbol a befinden.

Der Completer vervollständigt zuletzt die verarbeiteten Produktionen des Scanners. Konnte eine Teilproduktion beendet werden, d. h. alle Terminalsymbole einer Produktion $(A \to \lambda\cdot, k)$ wurden gelesen, so wird der Positionsindex der Produktion B inkrementiert die sich gerade vor der Produktion A befindet. Es wird ausgehend von $(B \to \alpha\cdot A\beta, j) \in S_k$ der Zustand $(B \to \alpha A \cdot \beta, j)$ zur Menge S_i hinzugefügt. Enthält zum Schluss die Zustandsmengenliste am n-ten Index einen Zustand der Form $(S' \to \alpha\cdot, 0)$ und S' ist die Startproduktion, so liegt das Wort der in Sprache von G, falls n und die Länge der

Eingabe übereinstimmen.

Beispiel 8. *Palindrom-Grammatik,* $G = (\{S', S\}, \{a, b\}, P, S')$
P sei die Menge der folgenden Produktionen:

$$S' \to S$$
$$S \to aSa \mid bSb \mid \varepsilon$$

Die erzeugte kontextfreie Sprache von G ist:

$$L(G) = \{\, ww^R \mid w \in \{a, b\}^* \,\}$$

Ist w ein Wort, so ist w^R *das rückwärts geschriebene Wort w. Die Sprache ist nicht deterministisch, daher kann das Wortproblem nicht von einem LR(1)-Parser entscheiden werden.*

Das Eingabewort sei $w = abba$.
Durch den Earley-Algorithmus wird dann die folgende Liste von Zustandsmengen generiert:

a	b	b	a	
$(S \to \cdot aSa, 0)$	$(S \to aS \cdot a, 0)$	$(S \to \cdot aSa, 2)$	$(S \to \cdot, 3)$	$(S \to \cdot aSa, 4)$
$(S \to \cdot, 0)$	$(S \to \cdot aSa, 1)$	$(S \to \cdot bSb, 2)$	$(S \to bS \cdot b, 2)$	$(S \to aS \cdot a, 3)$
$(S' \to S\cdot, 0)$	$(S \to a \cdot Sa, 0)$	$(S \to \cdot, 2)$	$(S \to \cdot aSa, 3)$	$(S \to \cdot, 4)$
$(S' \to \cdot S, 0)$	$(S \to \cdot, 1)$	$(S \to b \cdot Sb, 1)$	$(S \to \cdot bSb, 3)$	$(S \to a \cdot Sa, 3)$
$(S \to \cdot bSb, 0)$	$(S \to \cdot bSb, 1)$	$(S \to bS \cdot b, 1)$	$(S \to aS \cdot a, 0)$	$(S' \to S\cdot, 0)$
			$(S \to bSb\cdot, 1)$	$(S \to \cdot bSb, 4)$
			$(S \to b \cdot Sb, 2)$	$(S \to aSa\cdot, 0)$

Tabelle 4: Earley Zustandsmengenliste

Da $(S' \to S\cdot, 0) \in S_4$ *liegt* $w \in L(G)$.

4.2 Zeit- und Platzbedarf

Sei $w = w_1 w_2 \dots w_n$ das Eingabewort und $n = |w|$ die Länge des Wortes. Jede beliebige Zustandsmenge ist proportional in ihrer Anzahl an Zuständen durch n beschränkt. Für einen beliebigen Zustand (p, j, k) gilt $0 \leq p \leq c_1$, $0 \leq j \leq c_2$, $0 \leq k \leq n$, wobei c_1

die konstante Anzahl von Produktionen der Grammatik ist und c_2 der maximale Index einer geteilten Produktion. Für die Eingabe kann nur der Parameter k variieren, da er die derzeitige Position in der Eingabe angibt und diese ist durch n beschränkt. Daher ist der Platzbedarf in $\mathcal{O}(n)$ für jede Zustandsmenge. Insgesamt ergibt sich für $n + 1$ Zustandsmengen ein Platzbedarf von $\mathcal{O}(n^2)$.

Der Predictor und Scanner durchlaufen in jeden Schritt eine Zustandsmenge, die Berechnungen werden genau $n + 1$ mal wiederholt. Diese Zustandsmengen sind proportional zu n beschränkt. Daher liegt die Laufzeit der Berechnung in $\mathcal{O}(n^2)$.

Der Completer muss jeweils die derzeitige Zustandsmenge durchlaufen, sowie die Menge des Startindex des übergebenen Zustands. Die Laufzeit für eine Zustandsmenge liegt daher in $\mathcal{O}(n^2)$. Da es $n + 1$ Zustandsmengen gibt, liegt der Zeitbedarf in $\mathcal{O}(n^3)$.

Das Wortproblem wird mithilfe des Earley-Algorithmus mit Zeitaufwand $\mathcal{O}(n^3)$ gelöst. Er ist somit vergleichbar mit dem Algorithmus von Cocke, Younger [You67] und Kasami [Kas66], auch bekannt als CYK-Algorithmus. Earleys Algorithmus hat jedoch den Vorteil, dass die Grammatik nicht in Chomsky-Normalform vorliegen muss. Spezielle Grammatiken können eine bessere Performanz als $\mathcal{O}(n^3)$ erzielen: für eindeutige Grammatiken wird das Wortproblem in Laufzeit $\mathcal{O}(n^2)$ und für alle nicht rechtsrekursiven LR(k)-Grammatiken in linearer Zeit gelöst [Leo91; Ear68]. Um alle LR(k)-Grammatiken in linearer Zeit zu parsen beschreibt Leo [Leo91] eine Adaption für Tabellenparser die ggf. ein Lookahead verwenden.

4.3 Konstruktion der Ableitungsbäume

Earleys ursprüngliches Paper [Ear68] gibt eine Beschreibung an, wie man alle möglichen Ableitungsbäume bilden kann. Hierfür werden Zeiger von Nichtterminalsymbolen zurück auf Earley-Zustände gebildet. Es wird behauptet, dass die Konstruktion in einer Laufzeit von $\mathcal{O}(n^3)$ möglich ist. Dies wurde durch Tomita [Tom85] wiederlegt, der einen Fehler in Earleys Algorithmus aufzeigte. Scott [Sco08] stellte einen neuen Algorithmus vor, der einen binarised SPPF (Shared Packed Parse Forest) erzeugt und in Zeit $\mathcal{O}(n^3)$ läuft. Ein SPPF ermöglicht Teilbäume eines Ableitungsbaums wiederzuverwenden und diese für mehrere Knoten gemeinsam zu nutzen. Dies reduziert den Speicherbedarf, da bei mehrdeutigen Grammatiken die Anzahl der Ableitungsbäume für ein Wort exponentiell wachsen kann. Der resultierende Graph besteht aus einzelnen SPPF-Nodes, die mit einem

Tripel (X, i, j) benannt werden. X stellt hier das Terminal- bzw. Nichtterminalsymbol dar, das die Wortfolge $a_{i+1}...a_j$ abbildet. Damit eine kubische Laufzeit erreicht wird, werden zusätzlich Intermediate-Nodes verwendet, die den Ausgangsgrad eines Knotens auf höchstens zwei beschränken, sofern die Kindknoten keine Packed-Nodes sind. Daher wird der SPPF auch „binarised“ genannt. Packed-Nodes fassen Teilableitungsbäume zusammen, falls mehrere Ableitungen möglich sind. Scotts Algorithmus für Earley Recognizer [Sco08] wird nun im folgenden vorgestellt.
Um einen Parse Forest aufzubauen werden die Earley-Sets, die durch das Parsen eines Eingabeworts entstanden sind noch einmal iteriert. Dabei werden Rückwärtszeiger und Reduktionszeiger generiert. Für jedes $p = (A \to \alpha w_i \cdot \beta, j) \in E_i$ mit $i > 0$ und $\alpha \neq \varepsilon$, welches einen Vorgänger $q = (A \to \alpha \cdot w_i \beta, j) \in E_{i-1}$ hat wird ein Vorgängerzeiger mit der Bezeichnung $i-1$ von q zu p hinzugefügt. Des Weiteren wird für jedes $t = (B \to \tau \cdot, k) \in E_i$ und jedem korrespondierenden $q = (D \to \alpha \cdot B\mu, h)$, dass sich in E_k vor einem B befindet wird für das Nachfolger Element $p = (D \to \alpha B \cdot \mu, h) \in E_i$ ein Reduktionszeiger von p zu t mit Bezeichnung k erstellt. Falls außerdem gilt $\alpha \neq \varepsilon$, so wird zusätzlich ein Vorgängerzeiger k von p zu q hinzugefügt.

Algorithmus 5 BuildTree

Eingabe : u: SPPF-Node
p: Earley-Zustand der Form $(A \to \alpha \cdot \beta, j)$ aus E_i

markiere p als verarbeitet

if $p = (A \to \cdot, j)$ **then**
- falls keine SPPF-Node $v = (A, i, i)$ existiert so erstelle sie mit Kindknoten ε
- wenn u nicht (v) als Nachfolger besitzt, füge (v) zu u hinzu

if $p = (A \to \alpha \cdot \beta, j)$ *mit* α *ist ein Terminalsymbol* **then**
- falls keine SPPF-Node $v = (a, i-1, i)$ existiert so erstelle sie
- wenn u nicht (v) als Nachfolger besitzt, füge (v) zu u hinzu

if $p = (A \to C \cdot \beta, j)$ *mit* C *ist ein Nichtterminalsymbol* **then**
- falls keine SPPF-Node $v = (C, j, i)$ existiert so erstelle sie
- wenn u nicht (v) als Nachfolger besitzt, füge (v) zu u hinzu
- **foreach** *Reduktionzeiger* j *von* p *nach* q **do**
  - falls q noch nicht markiert wurde, BuildTree(v, q)

if $p = (A \to \alpha'\alpha \cdot \beta, j)$ *mit* α *ist ein Terminalsymbol,* $\alpha' \neq \varepsilon$ **then**
- falls keine SPPF-Node $v = (a, i-1, i)$ existiert so erstelle sie
- falls keine SPPF-Node $w = (A \to \alpha' \cdot a\beta, j, i-1)$ existiert so erstelle sie
- **foreach** *Vorgängerzeiger* $i-1$ *von* p *nach* q **do**
  - falls q noch nicht markiert wurde, BuildTree(w, q)
- wenn u nicht (w, v) als Nachfolger besitzt, füge (w, v) zu u hinzu

if $p = (A \to \alpha C \cdot \beta, j)$ *mit* C *ist ein Nichtterminalsymbol,* $\alpha' \neq \varepsilon$ **then**
- **foreach** *Reduktionszeiger* l *von* p *nach* q **do**
  - falls keine SPPF-Node $v = (C, l, i)$ existiert so erstelle sie
  - falls q noch nicht markiert wurde, BuildTree(v, q)
  - falls keine SPPF-Node $w = (A \to \alpha' x \cdot C\beta, j, l)$ existiert so erstelle sie
  - **foreach** *Vorgängerzeiger* l *von* p *nach* p' **do**
    - falls p' noch nicht markiert wurde, BuildTree(w, p')
  - wenn u nicht (w, v) als Nachfolger besitzt, füge (w, v) zu u hinzu

Der gesamte Baum kann nun mit BuildTree(u_0, p) erstellt werden, wobei u_0 der Wurzelknoten des Baumes mit Bezeichnung $(S, 0, n)$ ist und p die beendeten Earley-Zustände $(S \to \alpha\cdot, n) \in S_n$.

Beispiel 9. $G = (\{S\}, \{b\}, P, S)$
P sei die Menge der folgenden Produktionen:

$$S \to SS$$

$$S \to b$$

Das Eingabewort sei $w = bbb$*, mit* $n = |w| = 3$.
Zuerst werden die Earley-Sets für w *berechnet.*

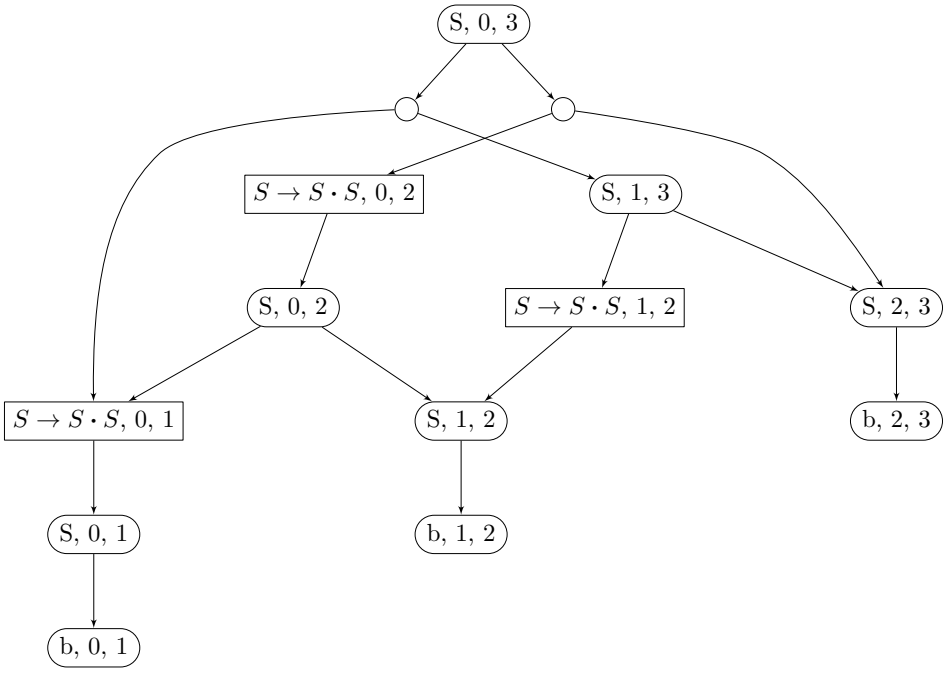

Abbildung 4: Shared Packed Parse Forest (SPPF)

S_0	S_1	S_2	S_3
$(S \to \cdot SS, 0)$	$(S \to b\cdot, 0)$	$(S \to \cdot b, 2)$	$(S \to \cdot SS, 3)$
$(S \to \cdot b, 0)$	$(S \to \cdot b, 1)$	$(S \to SS\cdot, 0)$	$(S \to SS\cdot, 0)$
	$(S \to S \cdot S, 0)$	$(S \to S \cdot S, 0)$	$(S \to b\cdot, 2)$
	$(S \to \cdot SS, 1)$	$(S \to S \cdot S, 1)$	$(S \to S \cdot S, 1)$
		$(S \to \cdot SS, 2)$	$(S \to S \cdot S, 0)$
		$(S \to b\cdot, 1)$	$(S \to SS\cdot, 1)$
			$(S \to S \cdot S, 2)$
			$(S \to \cdot b, 3)$

Tabelle 5: Earley-Sets

Nach der Berechnung enthält der SPPF nun zwei mögliche Ableitungsbäume die zusammengefasst sind. Ausgehend von $S \to SS$ wendet man entweder $S \to SS$ für das erste S oder das zweite S der Produktion an und leitet jeweils alle drei S nach b ab.

$$S \Rightarrow SS \Rightarrow SS \cdot S \Rightarrow bb \cdot b = bbb$$
$$S \Rightarrow SS \Rightarrow S \cdot SS \Rightarrow b \cdot bb = bbb$$

(Das Trennsymbol wird hier zur Visualisierung zwischen den verschiedenen Ableitungsmöglichkeiten verwendet.)

5 Harrison-Algorithmus

Im folgenden wird ein Algorithmus von Harrison vorgestellt [Har78], der sich für die Anwendung mit einer Laufzeit von cn^3 eignet, wobei c proportional zu $|G|$ ist. Es wird eine $(n+1) \times (n+1)$ Dreiecksmatrix erstellt, die Mengen von geteilten Produktionen erhält.

5.1 Voraussetzungen

Damit der Algorithmus formuliert werden kann, müssen noch einige Operationen und Hilfsfunktionen definiert werden, die zur Berechnung der Matrix benötigt werden.

Definition 14. *Sei $G = (V, \Sigma, P, S)$ eine kontextfreie Grammatik. Seien X, Y Mengen von geteilten Produktionen. Wir definieren $\times$ als eine Operation, die die Regeln aus X zurückgibt, bei denen ein Nichtterminalsymbol mithilfe von geteilten Produktionen bzw. Symbolen aus Y übersprungen werden kann.*

$$X \times Y := \{\, A \to \alpha B \beta \cdot \gamma \mid A \to \alpha \cdot B\beta\gamma \in X,\ \beta \Rightarrow^* \varepsilon,\ B \to \lambda\cdot \in Y \,\}$$
$$X \times Y := \{\, A \to \alpha B \beta \cdot \gamma \mid A \to \alpha \cdot B\beta\gamma \in X,\ \beta \Rightarrow^* \varepsilon,\ B \in Y,\ B \in (V \cup \Sigma)^* \,\}$$

Damit auch Terminalsymbole und Nichtterminalsymbole verarbeitet werden können, wird $\times$ nicht nur für geteilte Produktionen, sondern auch für Symbole definiert. Außerdem soll eine Verschachtelung ermöglicht werden, d. h. die vervollständigte Regel die in Y vorkommt, muss nicht direkt in einer Regel von X vorkommen, sondern darf auch durch eine Ableitung erzeugt werden. Hierfür wird ein neuer Operator $*$ definiert.

Definition 15. *Sei $G = (V, \Sigma, P, S)$ eine kontextfreie Grammatik. Seien X, Y Mengen von geteilten Produktionen.*

$$X * Y := \{\, A \to \alpha B \beta \cdot \gamma \mid A \to \alpha \cdot B\beta\gamma \in X,\ \beta \Rightarrow^* \varepsilon,\ B \Rightarrow^* C,\ C \to \lambda\cdot \in Y \,\}$$

Es gilt: $X \times Y \subseteq X * Y$

Beispiel 10. *Sei $G = (\{S, A, B, C, D\}, \{a\}, P, S)$ eine Grammatik*
P seien die folgenden Produktionen:

$$S \to ABAC$$
$$A \to \varepsilon$$
$$B \to CDC$$
$$C \to \varepsilon$$
$$D \to a$$

Für $X = \{S \to A \cdot BAC\}$ *und* $Y = \{B \to CDC\cdot\}$ *erhält man für die* $\times$*-Operation:*

$$X \times Y = \{\, S \to AB \cdot AC, S \to ABA \cdot C, S \to ABAC\cdot \,\}$$

Für $X = \{S \to A \cdot BAC\}$ *und* $Z = \{D \to a\cdot\}$ *erhält man für die* $*$*-Operation die gleiche Ergebnismenge wie für* $Y = \{B \to CDC\cdot\}$*:*

$$X * Z = \{\, S \to AB \cdot AC, S \to ABA \cdot C, S \to ABAC\cdot \,\}$$

Bei der $*$*-Operation kann jedoch erkannt werden, dass* $B \Rightarrow^* D$ *und somit auch* $B \Rightarrow^* a$ *gilt. Daher kann schon durch* $D \to a\cdot$ *erkannt werden, dass* B *vollständig verarbeitet werden kann.*

Wie in Earleys Algorithmus wird für Harrisons Algorithmus auch ein Predictor benötigt. Dieser wird wie folgt definiert:

Definition 16. *Sei* $G = (V, \Sigma, P, S)$ *eine kontextfreie Grammatik. Sei* $R \subseteq V$*.*

$$\text{predict}(R) := \{\, A \to \alpha \cdot \beta \mid A \to \alpha\beta \in P,\, \alpha \Rightarrow^* \varepsilon,\, B \Rightarrow^* A\gamma,\, B \in R,\, \gamma \in (V \cup \Sigma)^* \,\}$$

Handelt es sich bei R *um eine Menge von geteilten Produktionen, so wird predict definiert als:*

$$\text{predict}(R) := \text{predict}(\{\, B \mid A \to \alpha \cdot B\beta \in R \,\})$$

5.2 Analysealgorithmus

Sei $G = (V, \Sigma, P, S)$ eine kontextfreie Grammatik, $w = w_1 w_2 ... w_n$, $n \geq 0$ mit $w_i \in \Sigma$.

Algorithmus 6 Harrison-Algorithmus

Eingabe : $w = w_1 w_2 ... w_n$

Sei $T = (t_{i,j})$ eine $(n+1) \times (n+1)$ Matrix

$t_{0,0} := predict(\{S\})$

for $j = 1$ **to** n **do**

```
// Scanner
```

for $i = 0$ **to** $j\text{-}1$ **do**

$t_{i,j} := t_{i,j-1} \times w_j$

```
// Completer
```

for $k = j\text{-}1$ **downto** 0 **do**

$t_{k,j} := t_{k,j} \cup t_{k,k} * t_{k,j}$

for $i = k\text{-}1$ **downto** 0 **do**

$t_{i,j} := t_{i,j} \cup t_{i,k} \times t_{k,j}$

```
// Predictor
```

$$t_{j,j} := predict\left(\bigcup_{0 \le i \le j-1} t_{i,j}\right)$$

Der Algorithmus akzeptiert, sofern $S \to \alpha\bullet \in t_{0,n}$ wobei $\alpha \in (V \cup \Sigma)^*$ und $n = |w|$.
Es werden spaltenweise die einzelnen Mengen der Matrixeinträge berechnet. Wurden die Spalten 0 bis $j-1$ berechnet, so wird die j-te Spalte wie folgt berechnet: der Scanner liest ein Teil der Eingabe und schreibt die verarbeiteten Produktionen in die nächste Spalte neben der Diagonalen.

$$\begin{pmatrix} t_{0,0} \longrightarrow t_{0,1} \longrightarrow t_{0,2} \longrightarrow t_{0,3} & \cdots & t_{0,n} \\ t_{1,1} \longrightarrow t_{1,2} \longrightarrow t_{1,3} & \cdots & t_{1,n} \\ t_{2,2} \longrightarrow t_{2,3} & \cdots & t_{2,n} \\ \ddots & & \vdots \\ & \ddots & \vdots \\ & & t_{n,n} \end{pmatrix}$$

Abbildung 5: Verlauf Scanner

Dann geht der Completer die Spalte von Zeile $j-1$ bis zu 0 entlang. Jede Zelle $t_{k,j}$ wird gefüllt mit $t_{k,k} \times t_{k,j}$, d. h. die Diagonaleinträge werden, falls möglich, weiterverarbeitet durch die Zwischenresultate der j-ten Spalte. Weiter wird für jeden dieser Zelleneinträge

$t_{k,j}$ die $\times$-Operation durchgeführt für die Einträge über der jeweiligen Zelle mit den Ergebnissen $t_{i,k}$ der k-ten Spalte. Die Resultate werden in $t_{i,j}$ zurückgeschrieben.

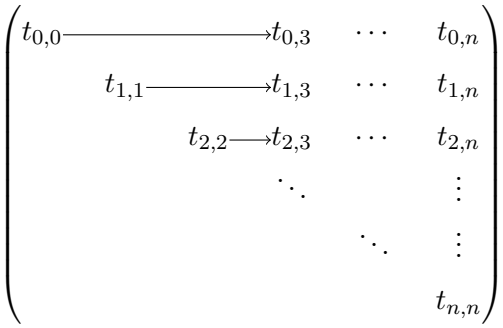

Abbildung 6: Verlauf Completer - äußere Schleife

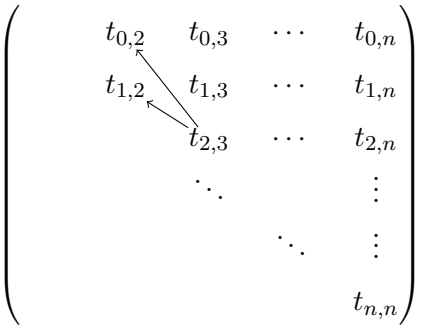

Abbildung 7: Verlauf Completer - innere Schleife

Zuletzt wird $t_{j,j}$ vervollständigt. Hierfür werden die Ergebnisse des Predictors hinzugefügt, der auf der Vereinigung aller vorherigen Zwischenresultate der Spalte ausgeführt wird.

Beispiel 11. *Sei $G = (\{S\}, \{(,)\}, P, S)$, die Grammatik der Klammerausdrücke.*
P seien die folgenden Produktionen:

$$S \to SS$$
$$S \to (S)$$
$$S \to ()$$

Das Eingabewort sei $w = (())$.

$S \to \cdot()$ $S \to \cdot SS$ $S \to \cdot(S)$	$S \to (\cdot S)$ $S \to (\cdot)$		$S \to (S\cdot)$	$S \to (S)\cdot$ $S \to S \cdot S$
	$S \to \cdot(S)$ $S \to \cdot SS$ $S \to \cdot()$	$S \to (\cdot)$ $S \to (\cdot S)$	$S \to ()\cdot$ $S \to S \cdot S$	
		$S \to \cdot SS$ $S \to \cdot()$ $S \to \cdot(S)$		
			$S \to \cdot SS$ $S \to \cdot(S)$ $S \to \cdot()$	
				$S \to \cdot SS$ $S \to \cdot()$ $S \to \cdot(S)$

Tabelle 6: Harrison-Analysematrix zu Beispiel 11

Zu Beginn werden durch den Predictor in $t_{0,0}$ alle LR(0)-Items aufgelistet, vor denen sich der Algorithmus gerade befinden könnte: $\text{predict}(\{S\}) = \{S \to \cdot() \ , \ S \to \cdot SS \ , \ S \to \cdot(S)\}$.
Der Scanner berechnet für $t_{0,1}$ dann die Items, die durch das Einlesen einer öffnenden Klammer fortgeführt werden können. Die beiden Möglichkeiten sind $S \to \cdot()$ und $S \to \cdot(S)$ deren Index des Trennsymbols zu $S \to (\cdot S)$, $S \to (\cdot)$ inkrementiert wird. Zum Schluss wird für die Spalte das Diagonalelement mithilfe des Predictors berechnet. Da sich $S \to (\cdot S)$ in $t_{0,1}$ befindet, werden die LR(0)-Items in $t_{0,2}$ eingetragen die direkt durch die Produktionen von S gebildet werden können. Der Prozess wird für jede Spalte bis zum Ende der Eingabe wiederholt. Da $S' \to S\cdot \in S_{0,4}$, folgt $w \in L(G)$.

5.3 Unterschiede zum Earley-Algorithmus

Der vorgestellte Algorithmus unterscheidet sich wesentlich in drei Aspekten von Earleys Algorithmus. Erstens werden die ε-Ableitungen statisch vorberechnet und können direkt durch die Operationen übersprungen werden. Earleys Algorithmus berechnet diese jeweils für jeden Schritt neu, da sie durch den Predictor neu entdeckt werden. Darauffolgend

können sie erst durch den Completer übersprungen werden. Zweitens wird eine Matrix anstatt einer Liste zur Berechnung verwendet. Zuletzt unterscheidet sich außerdem die Reihenfolge der Berechnungen. In Earleys Algorithmus werden für die Zustandsmenge S_i die geteilten Produktionen nach der Reihenfolge ihrer Entdeckung verarbeitet, während Harrisons Algorithmus für die i-te Spalte die Einträge nach absteigender Zeilennummer verarbeitet.

6 Implementierungen

Ein weiteres Ziel dieser Arbeit war es einen Teil der vorgestellten Algorithmen zu implementieren, um den Unterschied zwischen der theoretischen Definition und der praktischen Umsetzung aufzuzeigen. Im Folgenden wird die Entwicklung eines Analysetools beschrieben.

6.1 Allgemeines

Die Implementierung erfolgte in der Programmiersprache Rust, mit der 2018 Edition. Es wurde ein Programm entwickelt, das über ein Commandline-Interface für verschiedene Eingabegrammatiken einen Parser bereitstellt, der interaktiv für beliebige Eingabeworte das Wortproblem löst und ggf. die Ableitungsschritte bzw. die verschiedenen Ableitungsbäume ausgibt. Für das Commandline-Interface wird die Bibliothek *clap* verwendet.

Parameter		Erklärung
`-p`	`--parser`	Wählt den Parsingalgorithmus aus der genutzt werden soll. Mögliche Optionen: *earley*, *harrison*, *ll1*, *lr1*. Falls nicht angegeben wird die Option *earley* verwendet.
`-g`	`--grammar`	Pfad zu einer Datei, die eine Grammatikdefinition enthält.
`-i`	`--input`	Das zu parsende Eingabewort. Ist der Parameter nicht gesetzt, wird eine REPL [1] ausgeführt.
`-o`	`--output`	Ausgabepfad für den SPPF-Graphen im DOT Format. Wird nur für den Parser *earley* generiert.
`-f`	`--firstfollow`	Ausgabe der FIRST und FOLLOW Mengen der Grammatik.

Tabelle 7: Eingabeparameter für das Analysetool

Der Parameter *input* und die REPL [1] nehmen beliebige Zeichenfolgen entgegen und trennen diese jeweils an den gefundenen Leerzeichen. Die gesamte Eingabe wird als ein Wort betrachtet. Das Eingabeformat ist stark limitiert, da kein lexikalisches Analysetool

[1] Read-Eval-Print-Loop: Interaktives Computerprogramm, das Eingaben entgegen nimmt, direkt auswertet und das Ergebnis ausgibt

vorliegt, welches z. B. wie *Lex* die Definition von regulären Ausdrücken zur Trennung von Tokens ermöglicht.

6.2 Definition von Grammatiken

Um Grammatiken zu analysieren, werden diese mithilfe ihrer Produktionsregeln in der Erweiterten Backus-Naur Form angegeben. Die Syntax ist wie folgt definiert:

```
Rule = VARIABLE "=" Expr "." .
Expr = Alt { "|" Alt } .
Alt = { Term } .
Term = VARIABLE
     | LITERAL
     | "(" Expr ")"
     | "[" Expr "]"
     | "{" Expr "}"
     .
```

Der Punkt stellt das Ende einer Produktion dar, es wird = zur Definition einer Regel verwendet. Elemente können durch Klammern gruppiert werden, als optional markiert werden durch „[“, „]“ oder zur mehrfachen Wiederholung zugelassen werden durch „{“, „}“. Alternativen werden durch das |-Symbol getrennt, können aber auch einzeln durch mehrfache Zuweisungen desselben Nichtterminalsymbols definiert werden.
Die Menge der Terminalsymbole und Nichtterminalsymbole müssen nicht explizit angegeben werden. Nach dem Parsing des Eingabetexts wird die jeweilige Menge aus den Produktionen errechnet. Um das leere Wort darzustellen, folgt auf das Gleichzeichen direkt der Punkt um die Produktion zu terminieren. Das Startsymbol ist als die linke Seite der ersten aufgeführten Produktion definiert. Terminalsymbole werden mit Anführungszeichen maskiert und Bezeichner sind Zeichenketten bestehend aus alphanumerischen Symbolen oder Unterstrichen. Ein Bezeichner muss nicht mit einem Buchstaben beginnen.
In der Implementierung wird hierfür ein EBNF-Parser bereitgestellt, der die Datei in eine Grammatik-Struktur einliest, die aus einer Menge von Nichtterminalsymbolen, Terminalsymbolen und einer Liste von Produktionen besteht. Hierfür wird ein recursive descent LL(1)-Parser verwendet um einen Syntaxbaum zu erstellen. Dieser wird darauffolgend in einzelne Produktionen übersetzt, die keine Alternativen, Gruppierungen, optionale Elemente oder Wiederholungen mehr enthalten.

Produktionen bestehen aus einem Produktionsbezeichner und einem Vektor von Tupeln, die jeweils aus einem Symbol und einem Wahrheitswert bestehen. Der Wahrheitswert gibt an, ob es sich um ein Terminalsymbol handelt, um Konflikte zwischen gleichbenannten Nichtterminalsymbolen und Terminalsymbolen zu vermeiden.

Produktionsregel	Umformung
A = B | C .	A = B .
	A = C .
A = (B | C).	A = D .
	D = B .
	D = C .
A = [B | C] .	A = D .
	D = B .
	D = C .
	D = .
A = { B | C } .	A = D .
	D = E D
	D = .
	E = B .
	E = C .

Tabelle 8: Umformung der Grammatikregeln

Beispiel 12. *Regeln für die LOOP-Programmiersprache [Sch97]*

```
S = ident ":=" ident "+" const
  | ident ":=" ident "-" const
  | S ";" S
  | "loop" ident "do" S "end"
  .

ident = "x" const .
const = digit { digit } .
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" .
```

6.3 LL(1)- und LR(1)-Parser

Um diese Parser zu realisieren werden die FIRST und FOLLOW Mengen benötigt. Hierfür wurde eine Struktur *FFSets* definiert, die bereits errechnete FIRST bzw. FOLLOW Mengen zwischenspeichert. Diese besitzt drei wesentliche Funktionen `first`, `scan_over` und `follow`:

- `first` - ermöglicht die Berechnung der FIRST-Menge für einzelne Variablen aus V
- `scan_over` - berechnet die FIRST-Menge für eine Folge aus $(V \cup \Sigma)^*$
- `follow` - berechnet die Follow-Menge für eine Variable aus V.

Die Funktionen `first` und `follow` speichern Zwischenergebnisse in den *FFSets* ab, sodass keine erneute Berechnung benötigt wird, sollte eine bereits bekannte FIRST-Menge benötigt werden. Dies ermöglicht eine Lazy-Evaluation Auswertungsstrategie. Für die Endmarke wird das Symbol *$* verwendet. Die Follow-Mengen werden wie folgt berechnet:

Algorithmus 7 follow

Eingabe : B – Nichtterminalsymbol

if $B = S$ **then**
  FOLLOW$(B) := \{\$\}$
else
  FOLLOW$(B) := \emptyset$
foreach $A \to \alpha B \beta \in P$ **do**
  FOLLOW$(B) :=$ FOLLOW$(B) \cup$ FIRST(β)
  if $\beta \Rightarrow^* \varepsilon$ **then**
    FOLLOW$(B) :=$ FOLLOW$(B) \cup$ FOLLOW(A)

Für den LL-Parser wird eine verschachtelte HashMap verwendet, die zwei Zeichenketten auf einen Index der entsprechenden Produktion in der Liste der Grammatik abbildet. Mehrfacheinträge werden als Konflikte erkannt und führen zu einer Fehlermeldung. Die Implementierung folgt Algorithmus 1. Jeder Eintrag für ein Nichtterminalsymbol A und einer Zeichenfolge a enthält den Index der jeweiligen Produktionsregel.
Um den LR(1)-Parser zu implementieren werden LR(0)-Items verwendet, die als Strukturen definiert werden. Diese bestehen aus einem Produktionsindex und dem Index des Trennsymbols. Eine LR-Parsingtabelle besteht aus zwei Vektoren von HashMaps, die jeweils auf eine Aktion oder einen Sprungindex verweisen. Aktionen werden als ein Enum

mit den möglichen Werten *Shift*, *Reduce* oder *Acc* definiert. *Shift* und *Reduce* erhalten jeweils einen Index als Parameter, der auf den nächsten Zustand, bzw. auf die nächste zu reduzierende Produktion verweist.
Um die Konstruktion der Parsingtabelle in `compute_states` zu realisieren, werden zwei Methoden verwendet: `closure`, `goto_state`. Diese folgen in der Implementierung den bereits vorgestellten Algorithmen (siehe Algorithmus 3, Definition Goto-State auf Seite 14). `compute_states` fasst die Erzeugung und Nummerierung aller möglichen Zustände mit der Konstruktion der Tabelleneinträge zusammen (siehe Algorithmus 2). Hierfür wird eine Liste von LR(0)-Mengen verwendet, die durch `closure` Aufrufe berechnet wurde.

6.4 Earley-Algorithmus

Recognizer Earleys Algorithmus basiert auf Zustandstripeln. Diese werden als ein C-struct *State* definiert, bestehend aus dem Produktions- / Regelindex `rule_index`, dem Trennsymbolindex *dot* und dem Start des Matchings *start*. Der Earley Parser wird ebenfalls als ein C-struct deklariert mit einer Referenz auf die Grammatik, einen Zeiger auf das derzeitige zu verarbeitende Wort und einer Zustandsmengenliste, die während des Parsings modifiziert bzw. ausgelesen wird. Die Zustandsmengenliste S wird hierfür als ein Vektor von Hash-Sets definiert.
Ausgehend von dem ersten Eintrag der Liste, mit dem Startzustand $(S' \rightarrow \cdot S, 0)$ werden die Listenelemente und die Eingabe gleichzeitig iteriert über `process_state_set` ausgewertet. Da durch die Verarbeitung einer Zustandsmenge neue Zustände entstehen können, müssen diese ebenfalls berücksichtigt werden. Hierfür werden alle benötigten Updates berechnet und nur die weiterverarbeitet, die noch nicht vorhanden sind. Dies wird solange wiederholt, bis keine neuen Zustände mehr verfügbar sind.
Zur Auswertung eines Zustands wird `process_state` verwendet. Diese Methode wendet den folgenden Teilschritt des Earley-Algorithmus an:

Algorithmus 8 process_state

Eingabe : state $\in S_k$
w_k – Teil der Eingabe an k-ter Stelle

```
if ¬final(state) then
    if next−symbol(state) ist ein Nichtterminalsymbol then
        return predict(state, ...)
    else
        S_{k+1} := S_{k+1} ∪ scan(state, w_k, ...)
        return ∅
else
    return complete(state, ...)
```

Wurde die Analyse beendet, so erhält man die Zustandsmengenliste als Rückgabewert. Zur Überprüfung für das Wortproblem wird zuletzt die Bedingung State$(0, 1, 0) \in S_n$ getestet.

Shared Packed Parse Forest (SPPF) Um nun die Ableitungsbäume aus der Zustandsmengenliste erzeugen zu können, wird Scott's Algorithmus „A cubic parser which walks the Earley sets“ [Sco08] adaptiert. Der Shared Packed Parse Forest (SPPF) wird über Knoten definiert. Diese werden als *SPPFNode* definiert, die entweder vom Typ *Epsilon*, *Symbol* oder *Intermediate* sind. Zusätzlich erhält jeder Knoten eine Menge von sogenannten *FamilyNodes*. Diese stellen die Unterknoten da und können entweder ein einzelner Knoten sein oder eine Gruppe von zwei Knoten. Da für eine Gruppe von zwei Knoten (x, y) die Reihenfolge irrelevant ist, d. h. $(y, x) = (x, y)$, wird der Vergleichsoperator des Tupels überschrieben.
Zur Konstruktion wird der *ForestBuilder* verwendet, der die Liste der bereits existierenden *SPPFNodes* speichert und auf Anfrage deren Index zurückgibt. Des Weiteren enthält dieser eine Hash-Map von Vorgängerzeigern und Reduktionszeigern. Es wird jeweils ein Tupel, bestehend aus einem Earley-Zustand und einem Set-Index auf ein zweites Tupel abgebildet. Dieses zweite Tupel zeigt auf das Vorgänger-, bzw. das Reduktionselement. Dadurch können die benötigten Zeiger während der Konstruktion für einen Earley-Zustand schnell gefunden werden. Zusätzlich wird eine Menge von bereits besuchten Earley-Zuständen gespeichert.
Der *BuildTree*-Algorithmus wird angepasst und erhält einen weiteren Parameter, der den Listenindex des übergebenen Earley-Zustands speichert. Dies ermöglicht den Zugriff auf die Vorgänger- / Reduktionszeiger. Außerdem wird der Index des SPPF-Knotens anstatt

der Knoten selbst übergeben. Der SPPF-Teilbaum kann dann über die gleichnamige Methode `build_tree` konstruiert werden.
Als Voraussetzung müssen jedoch die benötigten Zeiger generiert werden. Dies geschieht durch eine weitere Iteration der bereits existierenden Zustandsmengenliste, die für das gegebene Wort bereits erstellt wurde.
Die vorgestellte Implementierung benötigt neben den Earley-Zuständen auch die Indizes der Zustände innerhalb der Liste. In Scott's originalem Paper [Sco08] wird der Index als Bezeichnung / Label deklariert. Hier wird es als das zweite Element des Tupels verwendet. Für Reduktionszeiger ist das Label immer der Start des Matchings der Produktion die die Inkrementierung des Trennsymbols ermöglicht hat. Für Vorgängerzeiger ist das Label der Index der vorherigen Menge S_{i-1}. Es wird wie im Paper beschrieben vorgegangen. Die Methode `build_forest` fasst die Konstruktion des SPPF-Startknotens, die Generierung der Zeiger und den Aufbau des SPPFs für eine gegebene Zustandsmengenliste zusammen.

6.5 Harrison-Algorithmus

Um Harrisons Algorithmus zu implementieren werden die vordefinierten Hilfsoperationen $\times$, $*$ und predict(R) benötigt. Da für den Scanner eine $\times$-Operation verwendet wird, die Terminalsymbole anstatt Nichtterminalsymbole bzw. geteilte Produktionen verwendet, muss dieser Fall ebenfalls behandelt werden.
Zuerst wird eine Hilfsfunktion `skip_epsilon` definiert. Diese wird für die Matrixoperationen $\times$ und $*$ benötigt. Für ein LR(0)-Item erhält man die Menge aller Nachfolgerzustände, die durch das Überspringen von Nichtterminalsymbolen erreicht werden können, die sich zum leeren Wort ableiten lassen. Um die Nachfolgersymbole zu testen, wird die FIRST Funktion verwendet. Gilt für ein Nichtterminalsymbol N, dass $\varepsilon \in \text{FIRST}(N)$, so wird der Index weiter inkrementiert, bis ein Nichtterminalsymbol N mit $\varepsilon \notin \text{FIRST}(N)$ oder ein Terminalsymbol gefunden wird. Die Ergebnismenge enthält außerdem das Eingabeelement selbst.

Beispiel 13. *Seien folgende Produktionsregeln gegeben:*

$$A \to BCBD$$
$$B \to \varepsilon$$
$$C \to \varepsilon$$
$$D \to a$$

Eingabe: das LR(0)-Item $I = A \to \cdot BCBD$
`skip_epsilon`$(I) = \{A \to \cdot BCBD,\ A \to B \cdot CBD,\ A \to BC \cdot BD,\ A \to BCB \cdot D\}$

Die Operatoren $\times$ und $*$ werden in einer Funktion zusammengefasst und wie folgt implementiert:

Algorithmus 9 complete

Eingabe : q: Menge von LR(0)-Items
r: Menge von LR(0)-Items
chained: Mehrfache Ableitungen erlauben

Sei $T = \emptyset$
foreach *terminiertes LR(0)-Item* $A \to \alpha\bullet \in R$ **do**
  if *chained* **then**
    $T := T \cup \{A\} \cup$ `reduction_map`(A)
  else
    $T := T \cup \{A\}$
foreach *symbol* S *aus* T **do**
  foreach *unterminiertes LR(0)-Item* $B \in Q$ **do**
    token := Symbol der Regel von B, das auf das Trennsymbol folgt
    if *token* $= S$ **then**
      item := LR(0)-Item B mit inkrementiertem Index
      füge `skip_epsilon(item)` zur Lösungsmenge hinzu

Mögliche Reduktionen werden über eine Funktion `find_reductions` statisch vorberechnet und in einer `reduction_map` gespeichert, die ein Nichtterminalsymbol C auf die Symbole aller möglichen Produktionen abbildet für die gilt: $B \Rightarrow^* C$.
Dieser Ansatz wird ebenfalls für Ableitungen angewendet. Für ein Nichtterminalsymbol werden die Menge aller möglichen LR(0)-Items berechnet und auf diese jeweils der Completer angewandt, bis keine Aktualisierungen der Menge mehr vorhanden sind. Diese Ableitungen werden ebenfalls vorberechnet und in einer `derivation_map` gespeichert, die ein Nichtterminalsymbol auf alle durch sie erreichbaren LR(0)-Items abbildet.
Der Predictor `predict` verhält sich ähnlich zu Earleys Verfahren. Allerdings wird hier nur auf die vorberechneten Ableitungen in `derivation_map` zugegriffen. Statt einen Earley-Zustand nimmt der Predictor eine Menge von Nichtterminalsymbolen entgegen, für die die erreichbare Menge von LR(0)-Items zusammengefasst zurückgegeben wird.
Der Scanner wird auch als eine Methode `scan` implementiert und betrachtet alle übergebenen LR(0)-Items der Form $(\text{index}, \text{dot})$ an der Stelle des jeweiligen Trennsymbols. Stimmen das gefundene Terminalsymbol mit dem derzeitigen Teilwort überein, so erhält die Lösungsmenge alle neuen LR(0)-Items aus `skip_epsilon` für $(\text{index}, \text{dot} + 1)$.

Die Matrix des Algorithmus wird durch eine Liste von Listen dargestellt, die jeweils als Eintrag ein HashSet von LR(0)-Items enthält. Die Implementierung des Parsingalgorithmus `accepts` folgt dann dem vorgestellten Algorithmus 6, mithilfe der Methoden `predict`, `scan` und `complete`.

7 Zusammenfassung und Ausblick

Zu Beginn der Arbeit wurden die LL(k)- / LR(k)-Grammatiken eingeführt, die alle deterministisch kontextfreien Sprachen erzeugen und durch entsprechende Parser das Wortproblem in linearer Laufzeit lösen können. Es wurden Sprachhierarchien angegeben, sowie Algorithmen zur Konstruktion der jeweiligen Parsingtabellen.
Anschließend wurde Earleys Algorithmus vorgestellt, der die Gesamtheit aller kontextfreien Sprachen parsen kann und in kubischer Laufzeit das Wortproblem entscheidet. Im Vergleich zu anderen Parsingalgorithmen für kontextfreie Sprachen ist die Besonderheit des Algorithmus von Earley, dass dieser das Wortproblem für eindeutige Grammatiken in quadratischer und für LR(k)-Sprachen in linearer Laufzeit löst.
Weiter wurde ein Algorithmus von Harrison vorgestellt, der sich an dem CYK-Algorithmus orientiert und versucht den Algorithmus von Earley zu verbessern. Das Wortproblem wird auch hier in kubischer Laufzeit gelöst.
Außerdem wurden die vorgestellten Algorithmen implementiert. Es wurde ein Tool entwickelt, das für beliebige Eingabegrammatiken, die in einer Backus-Naur-Form vorliegen, das Wortproblem löst. Über die Software kann der zu verwendende Parsingalgorithmus eingestellt werden und optional ein Syntaxbaum, bzw. die Ableitungsschritte ausgegeben werden.
In Zukunft könnte man die Implementierung noch weiter optimieren, indem man die Hilfsmethoden effizienter durch statische Vorberechnungen umsetzt. Außerdem benötigt der LR-Parser viel Speicherplatz für große Grammatiken und könnte durch Techniken z. B. von Korenjak [Kor69] und DeRemer [DeR71] verbessert werden. Des Weiteren könnten die LL-Parser und LR-Parser erweitert werden, sodass das Lookahead k einstellbar ist.

Die vorgestellten Algorithmen können für kontextfrei Sprachen das Wortproblem in kubischer Laufzeit lösen. Valiant [Val75] konnte durch seine Erweiterung des CYK-Algorithmus eine bessere Laufzeit von $\mathcal{O}(n^{2.81})$ erzielen. Die Lösung des Wortproblems wird hierfür auf Boole'sche Matrixmultiplikation reduziert. Allerdings ist dieser Algorithmus nicht für die Anwendung geeignet, durch die Optimierung entstehen zu große Laufzeitkonstanten. Lee zeigte die Reduktion von Boole'scher Matrixmultiplikation auf das Parsingproblem für kontextfreie Sprachen [Lee02]. Ein effizienter Parser benötigt daher effiziente Boole'sche Matrixmultiplikation und vice versa. Jeder Parsingalgorithmus mit Laufzeit $\mathcal{O}(|G|n^{3-\epsilon})$ kann zu einem Algorithmus konvertiert werden, der Boole'sche Matrixmultiplikation

verwendet und in Zeit $\mathcal{O}(m^{3-\epsilon/3})$ [1] arbeitet [Lee02].
Ein weiterer Algorithmus, der die gleiche Laufzeitkomplexität des CYK- und Earley-Algorithmus besitzt, ist der Generalized LR-Parser, auch bekannt als GLR-Parser [Tom85]. GLR-Parser sind eine Erweiterung der LR-Parser, die Nichtdeterminismus verarbeiten können, indem mehrere Threads erzeugt werden, die parallel arbeiten und eine Breitensuche ausführen.
Neuere Fortschritte umfassen den Generalized LL-Parser (GLL) [SJ10a]. Dieser basiert auf der Technik des rekursiven Abstiegs und ermöglicht es alle kontextfreien Grammatiken mit einer worst-time Laufzeitkomplexität von $\mathcal{O}(n^3)$ zu parsen, mit linearer Laufzeit für LL-Grammatiken. Da das Verfahren auf LL(1)-Parsern basiert, ist die Implementierung einfach nachzuvollziehen, sodass ein Parser auch einfach manuell selbst geschrieben werden kann.

[1] für Boole'sche $m \times m$ Matrizen

8 Literatur

[Aho+06] Alfred V. Aho u. a. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN: 0321486811.

[Cho56] Noam Chomsky. „Three models for the description of language“. In: *IRE Trans. Information Theory* 2.3 (1956), S. 113–124.

[DeR71] Frank DeRemer. „Simple LR(k) Grammars“. In: *Commun. ACM* 14.7 (1971), S. 453–460.

[Ear68] Jay Earley. „An Efficient Context-Free Parsing Algorithm“. Diss. Pittsburgh, PA, USA: Carnegie Mellon University, 1968.

[Ear70] Jay Earley. „An Efficient Context-Free Parsing Algorithm“. In: *Commun. ACM* 13.2 (1970), S. 94–102.

[GHR80] Susan L. Graham, Michael A. Harrison und Walter L. Ruzzo. „An Improved Context-Free Recognizer“. In: *ACM Trans. Program. Lang. Syst.* 2.3 (1980), S. 415–462.

[GJ08] Dick Grune und Ceriel J. H. Jacobs. *Parsing Techniques - A Practical Guide*. Monographs in Computer Science. Springer, 2008.

[Har78] M. A. Harrison. *Introduction to Formal Language Theory*. 1st. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1978. ISBN: 0201029553.

[HU90] John E. Hopcroft und Jeffrey D. Ullman. *Einführung in die Automatentheorie, formale Sprachen und Komplexitätstheorie (2. Aufl.)* Internationale Computer-Bibliothek. Addison-Wesley, 1990.

[JM09] Dan Jurafsky und James H. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition, 2nd Edition*. Prentice Hall series in artificial intelligence. Prentice Hall, Pearson Education International, 2009.

[Kas66] Tadao Kasami. „An efficient recognition and syntax-analysis algorithm for context-free languages“. In: *Coordinated Science Laboratory Report no. R-257* (1966).

[Knu65] Donald E. Knuth. „On the Translation of Languages from Left to Right“. In: *Information and Control* 8.6 (1965), S. 607–639.

[Kor69] A. J. Korenjak. „A practical method for constructing LR(k) processors“. In: *Commun. ACM* 12.11 (1969), S. 613–623.

[Lee02] Lillian Lee. „Fast context-free grammar parsing requires fast boolean matrix multiplication“. In: *J. ACM* 49.1 (2002), S. 1–15.

[Leo91] Joop M. I. M. Leo. „A General Context-Free Parsing Algorithm Running in Linear Time on Every LR (k) Grammar Without Using Lookahead“. In: *Theor. Comput. Sci.* 82.1 (1991), S. 165–176.

[May86] Otto Mayer. *Syntaxanalyse, 3. Auflage*. Bd. 27. Reihe Informatik. Bibliographisches Institut, 1986.

[Par93] Terence John Parr. „Obtaining Practical Variants of LL (K) and LR (K) for K Greater Than 1 by Splitting the Atomic K-tuple“. Diss. West Lafayette, IN, USA: Purdue University, 1993.

[RS70] Daniel J. Rosenkrantz und Richard Edwin Stearns. „Properties of Deterministic Top-Down Grammars“. In: *Information and Control* 17.3 (1970), S. 226–256.

[Sch97] Uwe Schöning. *Theoretische Informatik - kurzgefaßt (3. Aufl.)* Hochschultaschenbuch. Spektrum Akademischer Verlag, 1997.

[Sco08] Elizabeth Scott. „SPPF-Style Parsing From Earley Recognisers“. In: *Electr. Notes Theor. Comput. Sci.* 203.2 (2008), S. 53–67.

[SJ10a] Elizabeth Scott und Adrian Johnstone. „GLL Parsing“. In: *Electr. Notes Theor. Comput. Sci.* 253.7 (2010), S. 177–189.

[SJ10b] Elizabeth Scott und Adrian Johnstone. „Recognition is not parsing - SPPF-style parsing from cubic recognisers“. In: *Sci. Comput. Program.* 75.1-2 (2010), S. 55–70.

[Soi80] Eljas Soisalon-Soininen. „On Comparing LL(k) and LR(k) Grammars“. In: *Mathematical Systems Theory* 13 (1980), S. 323–329.

[SS82] Seppo Sippu und Eljas Soisalon-Soininen. „On LL(k) Parsing“. In: *Information and Control* 53.3 (1982), S. 141–164.

[Tom85] Masaru Tomita. *Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems*. Norwell, MA, USA: Kluwer Academic Publishers, 1985. ISBN: 0898382025.

[Val75] Leslie G. Valiant. „General Context-Free Recognition in Less than Cubic Time“. In: *J. Comput. Syst. Sci.* 10.2 (1975), S. 308–315.

[You67] Daniel H. Younger. „Recognition and Parsing of Context-Free Languages in Time n^3“. In: *Information and Control* 10.2 (1967), S. 189–208.

Abbildungsverzeichnis

Tabellenverzeichnis

Liste der Algorithmen