Leibniz Universität Hannover
Fakultät für Elektrotechnik und Informatik
Institut für Theoretische Informatik

Vergleich verschiedener Parsing-Verfahren für Kontextfreie Sprache

Bachelorarbeit

im Studiengang Informatik

von

Yichen Xie
Matrikelnummer: 3218030

Prüfer: Prof. Dr. Heribert Vollmer

Zweitprüfer: Dr. Arne Meier

Betreuer: Prof. Dr. Heribert Vollmer

Hannover, 14. September, 2019

Erklärung

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel und Quellen genutzt habe.

____

Yichen Xie

Hannover, den 14. September, 2019

Inhaltverzeichnis

1 **Einleitung** **1**

2 Cocke-Kasami-Younger Algorithmus **2**

2.1 Chomsky-Normalform 2

2.2 Prinzip 2

2.3 Erstellung der Tabelle 3

3 LR(k)-Parser **5**

3.1 Umformen von der Grammatik 5

3.2 First() 6

3.3 LR-Item 6

3.4 Closure() 7

3.5 Erstellung von DFA 8

3.6 Erstellung von Action-Tabelle und GoTo-Tabelle 9

3.7 Prüfung von Eingabe 10

4 Implementierung der beiden Parser **12**

4.1 Komponente einer Grammatik 12

4.2 Die Klasse CYK 14

4.3 Die Klasse LR1 14

4.4 Die Klasse DFA 15

4.5 MainClass: Comparison 16

5 Messungen und Vergleich **17**

6 Fazit **19**

Literaturverzeichnis **20**

1 Einleitung

Die Untersuchung der formalen Sprache ist seit langem Kern der Themen im Bereich theoretische Informatik. Vor allem spielt der Parser eine wichtige Rolle, das für eine bestimmte Programmiersprache eingesetzt werden und damit die Semantik der Eingabe erschließen könnten. Konkreter wird geprüft, ob die Eingabe von der Grammtik dieser Programmiersprache akzeptiert werden könnte. Um solcher Prozess zu ermöglichen, gibt es viele unterschiedlichen Verfahren, wie z.B. Cocke-Kasami-Younger Algorithm (im Folgenden als CYK gekürzt), LL- und LR-Parsing. In dieser Arbeit werden spezifisch CYK- und LR- Parser betrachtet.
Wegen verschiedenen Mechanismen der oben genannten Verfahren unterscheidet sich auch ihren Laufzeiten. Theoretisch ist für eine Eingabe der Länge n beim CYK-Parser die Laufzeit kubisch, und beim LR-Parser linear. Aber trotz der eigenen Parsing-Prozesse, braucht die beiden im Praxis auch verschiedene Vorbereitungsprozesse. Bei CYK muss die originale kontextfreie Grammatik zunächst in Chomsky-Normal-Form umformt werden. Anderseits beim LR-Parser wird erstens eine deterministische finite Automat (im Folgenden als DFA gekürzt) erstellen und dadurch ACTION- und GOTO-Tabelle erfüllen.
Wie verhalten sich die beiden Laufzeiten wenn die beiden Parser im Praxis eingesetzt werden? In dieser Arbeit soll am erstens die Theorie der beiden Parser erklären, dannach die Parser implementieren und schließlich die jeweilige Laufzeit durch Testen und Messungen vergleichen. Als Programmiersprache wird Java für die gesamte Implementierung verwendet.

2 Cocke-Kasami-Younger Algorithmus

CYK Algorithmus ist die eine einfache Prozedure um den eingegebene Text in einer kontextfreien Sprache zu erkennen. Die zu parsende Grammatik soll im Chomsky-Normalform sein. Dadurch ist der entstehende Parsebaum binär. [MAK88, Chapter 4.1]

2.1 Chomsky-Normalform

Definition 2.1 Eine kontextfreie Grammatik G = (V, X, S, P) is im Chomsky-Normalform(CNF) falls alle Produktionen davon im folgenden Form sind:

$A \rightarrow BC$ oder $A \rightarrow a$,

wobei A, B, C die Variablen sind und a ein Terminal von G ist. [HU79, S.92]

Man kann eine beliebige kontextfreie Grammatik im CNF umformen, wie durch folgendes Beispiel aus [HU79, S.93f] gezeigt werden:

Bsp.: G = ({S, A, B}, {a,b}, P, S) mit P = {S→bA|aB, A→bAA|aS|a, B→aBB|bS|b}

Zuerst betrachten wir A→a und B→b, wobei merkt man sofort dass die beiden Produktionen schon im geforderten Form ist. Dann können alle in anderen Produktionsregeln auftretende Terminalen a und b durch Variablen A, B ersetzt werden. So bekommen wir jetzt S →BA, S→AB, A→BAA, A→AS, B→ABB und B→BS. Dannach suchen wir alle Productionsregeln aus, die mehr als zwei Variablen auf der rechten Seite haben, in diesem Fall also A→BAA und B→ABB. Dazu erstellen wir neue Variablen D und E mit entsprechende Productionsregeln: D→AA und E→BB. Einsetzen wir dies im oben genannten Regeln, so bekommt man schließlich A→BD und B→BE. Dies erfolgt eine Umformung einer kontextfreien Grammatik in CNF, die bereit für Anfang eines CYK-Parsingprozesses ist.

2.2 Prinzip

Gegeben ist eine kontextfreie Grammatik G im CNF, jetzt soll der Parser entscheiden, ob die Eingabe $w = x_1 \ldots x_n$ mit der Länge n von L(G) akzeptiert wird. Um dies zu erklären, wird im Folgenden das gesamte Prozess mit demselben Beispiel von Kapitel 2.1 dargestellt.

Bsp.: G = ({S, A, B}, {a,b}, P, S) mit P = { S→BA, S→AB, A→BD, A→AS, A→a, B→b, B→AE, B→BS, D→AA, E→BB} mit der Eingabe „abbbaa".

Das Hauptprinzip von CYK ist eine Zerlegung der originalen Eingabe mit Länge n in vielen Stringteilen der unterschiedlichen Länge x mit $x \leq n$ und damit alle möglichen Stringpaaren zu betrachten, die durch deren Kombinierung zu der originalen Eingabe wieder führen könnten.
Im Fall des oben genannten Beispiels wird die Eingabe „abbbaa" für x = 1 in sechs String geteilt, also „a", „b", „b"… „a", d.h. aus jeder einzelnen Buchstaben entsteht ein neuer Stringteil. Für x = 2 gibt es fünf Stringteilen, „ab", „bb", „bb", „ba", „aa" , für x = 3 vier usw. , bis zum x = n = 6, wobei es nur ein String gibt, welcher genau die originale Eingabe entspricht.

2.3 Erstellung der Tabelle

Um alle Kombinationen der Stringteile zu betrachten, erstellen wir so eine Pyramide, deren Basis eine Größe von n Zellen hat. Wir erfüllen die Zelle (1, j) für j = 1…n mit Variable $A \in V$, falls es eine Production $A \rightarrow x_j$ gibt. In diesem Fall findet man nur A→a und B→b für die beiden Terminalen a und b, deswegen geben wir in der Zelle (1,1), (1,5), (1,6) die Variable A und in (1,2), (1,3), (1,4) B ein.
Dannach sollen wir alle Kombinationen der Stringteilen betrachten. Zum Beispiel nehmen wir erstens die Zelle (1,1) (steht für Stringteil „a") und Zelle (1,2) (steht für Stringteil „b"), so bekommen wir Kombination „AB". In der Produktion von G finden wir S→AB, d.h. die durch dieses Stringpaar entstehender Stringteil „ab" von S akzeptieren könnte. Deswegen erfüllen wir S in die entsprechende Zelle (2,1).
Im allgemein können wir alle restlichen Zellen (i, j) mit i >1 und j = 1…n-i+1 wie folgendes erfüllen:
Für ein m = 1…i-1, prüft die Zellenpaare (m, j) und (i-m, j+m). Falls in der Zelle (m,j) steht Variable A und in (i-m, j+m) steht Variable B und es gibt eine Produktion $D \rightarrow AB$, dann erfüllen wir Variable D in Zelle (i, j). Falls in einer der zu prüfenden Zellen steht mehr als eine Variable z.B. in (m,j) steht EF, dann muss alle Varible-Kombinationen von

EF und B geprüft werden, d.h. EB, FB.

z.B. sieht diese Prozess wie folgenden aus:

Zelle	**m**	**die zu prüfende Paare**	**entsprechende Produktion**	
(3,1)	m=1	(1,1) & (2,2)	AE:	B→AE
	m=2	(2,1) & (1,3)	SB:	-
(3,2)	m=1	(1,2) & (2,3)	BE:	-
	m=2	(2,2) & (1,4)	EB:	-
(3,3)	m=1	(1,3) & (2,4)	BS:	B→BS
	m=2	(2,3) & (1,5)	EA:	-
(3,4)	m=1	(1,4) & (2,5)	BD:	A→BD
	m=2	(2,4) & (1,6)	SA:	-
(4,1)	m=1	(1,1) & (3,2)	A-:	-
	m=2	(2,1) & (2,3)	SE:	-
	m=3	(3,1) & (1,4)	BB:	E→BB
...				
(6,1)	m=1	(1,1) & (5,2)	AB:	S→AB
	[B und A in (5,2)]		AA:	D→AA
	m=2	(2,1) & (4,3)	SS:	-
	m=3	(3,1) & (3,4)	BA:	S→BA
	m=4	(4,1) & (2,5)	ED:	-
	m=5	(5,1) & (1,6)	BA:	S→BA

Und schließlich bekommen wir die Ergebnistablle ((i,j) entspricht die Zellenszahl):

(6,1) S, D					
(5,1) B	(5,2) B, A				
(4,1) E	(4,2) E	(4,3) S			
(3,1) B	(3,2) -	(3,3) B	(3,4) A		
(2,1) S	(2,2) E	(2,3) E	(2,4) S	(2,5) D	
(1,1) A	(1,2) B	(1,3) B	(1,4) B	(1,5) A	(1,6) A
a	b	b	b	a	a

Nach dem ganzen Prozess prüfen wir nun die Zelle (n,i), ob dessen Ergebnis die Startvaribale S enthält, z.B. in oben gezeigte Tabelle enthält die Zelle (6,1) Variable S und D, deswegen ist die Eingabe „abbbaa“ von Grammatik G akzeptiert. [MAK88, Chapter 4.1]

3 LR(1)-Parser

Im Vergleich zum CYK Algorithmus, die LR(k) hat aber ein ganz anders Prinzip, und zwar so genannte „Bottom-up” Parsing. LR(k) steht für Scan des eingegebenen Strings von links nach rechts mit k Symbole als „lookahead”[vgl. HU79, S.248]. Es beginnt am den linksten Zeichen, und prüft ob es in der gegebenen Grammatik eine entprechende Produktionsregel gibt und dadurch eine Möglichkeit zur Auflösung der Variablen existiert. [MAK88, S.132]

Da der LR(0)-Parser nur bei eine sehr beschränkte Menge von Grammtiken funktionieren kann, betrachten wir in folgender Arbeit spezifisch den LR(1)-Parser. Durch ein lookahead-Symbol von LR(1) werden die meisten mögliche Konflikte verschwinden, die bei LR(0) erscheinen könnten.

Definition 3.1 Eine Grammtik G ist LR(1), falls:

1) Das Startsymbol tritt in keiner rechten Seite der Produktion auf;

2) Für alle Itemset *I*, die eine komplette LR(1)-Item $A \to \alpha\cdot$, $\{a_1, a_2, ..., a_n\}$ enthält, dann gilt folgendes:

i) kein a_i tritt sofort an der rechten Seite des Punktes von einem Item in *I* auf, und

ii) falls $B \to \beta\cdot$, $\{b_1, b_2, ..., b_k\}$ ist ein anderes komplette LR(1)-Item in *I*, dann ist $a_i \neq b_i$ für alle $1 \leq i \leq n$ und $1 \leq j \leq k$. [HU79, S.263]

3.1 Umformen von der Grammatik

Dazu nehmen wir wieder das Beispiel von Kapitel 2.1:

Bsp.: $G = (\{S, A, B\}, \{a,b\}, P, S)$ mit $P = \{S \to bA|aB, A \to bAA|aS|a, B \to aBB|bS|b\}$

Bevor wir den Parsing-Prozess richtig anfangen, sollen zunächst alle Produktionen geteilt werden, sodass jede nur eine Regel auf ihre rechter Seite enthält. Aus oben genanntem Beispiel bekommen wir dann die neue Produktionen $P' = \{S \to bA, S \to aB, A \to bAA, A \to aS, A \to a, B \to aBB, B \to bS, B \to b\}$.

So merken wir uns sofort, dass die Beispielsgrammatik die Definition 3.1 widerspricht, weil die Startsvariable S auf der rechten Seite der Produktion $A \to aS$ und $B \to bS$ auftritt. Deswegen erzeugen wir eine Variable S’ als neues Startsymbol mit Produktion $S' \to S$

und fügt ein Endsymbol *$* am Ende der Produktion vom Startsymbol, damit bekommen wir die neue **Grammatik G'**:

G' = ({S', S, A, B}, {a,b,*$*}, P', S') mit P' = { S'→ S*$*, S→bA, S→aB, A→bAA, A→aS, A→a, B→aBB, B→bS, B→b}

Nun ist diese Grammtik bereit für die Erzeugung von LR-Items. Dazu brauchen wir aber noch ein spezielles Set, falls wir nicht nur LR(0)-Item, sondern auch LR(1)-Item erstellen möchten: First().

3.2 First()

First(x) beschreibt eine Funktion, die für ein eingegebenes Symbol x alle seine mögliche Präfixe zurückgibt. Für ein Präfix wird nur das erste Symbol t der Produktionsregel betrachtet. Falls t ein leeres Symbol ist, dann wird das Endsymbol zurückgegeben. Falls t ∈X ist, wobei X die Menge der Terminals von G ist, wird direkt t zurückgegeben, ansonsten geht es weiter zum First(t).

Beispielweise erstellen wir First(S'), durch seine Produktion S'→ S*$* bekommen wir First(S') = First(S), und geht's weiter mit S→bA und S→aB. Da a und b Terminals sind, ist First(S) = {a, b}, dann ist First(S') = {a, b}.

Ähnlich wie oben bekommen wir auch die restliche First-Sets:

First(S') = {a, b}, First(S) = {a, b}, First(A) = {b, a}, First(B) = {a, b}.

3.3 LR-Item

Die LR(0)-Items kann man schon am Anfang des Prozesses erzeugt werden, und zwar aus jeder Produktionsregel:

z.B. für S'→ S*$* bekommen wir {S'→ ·S*$*, S'→ S·*$*, S'→ S*$*·}

Der Punkt bewegt sich jedes mal genau eine Characterposition nach rechts und beschreibt dadurch den aktuellen Zustand des Scans, d.h. nach dem Punkt soll das einzugebendes Zeichen stehen. Alle durch die Produktionen von G' entstehende Items gehört zum LR(0)-Itemset *I'*.

Ein LR(1)-Item besteht aus einem LR(0)-Item und einem Set von lookahead-Symbole.

3.4 Closure()

Closure(x) ist eine Funktion, ein Itemset zu komplementieren, wobei soll ein LR(1)-Item x eingegeben und schließlich ein Set von LR(1)-Items zurückgegeben werden. Es geht wie folgendes:

Wir merken uns das einzugebende Symbol B des LR(1)-Items I_n = [I'_n, t] und ein sofort nach diesem stehendes Symbol β, so gilt für alle von Closure() erzeugende LR(1)-Items folgendes:

1) falls B ∈ V, dann sucht alle I'_i aus der LR(0)-Itemset *I'* von G aus, auf deren linker Seite B steht und auf deren rechter Seite der Punkt noch am Anfangsposition steht;

2) für alle ausgefundene I'_i erzeugt man I_i = [I'_i , t'] und fügt diese in LR(1)-Itemset I_{Erg} ein, falls I_i noch nicht existiert, wobei t' = t falls β = ε (Leerzeichen), sonst ist t' = First(β) ;

3) schließlich gibt LR(1)-Itemset I_{Erg} zurück, für jedes Item I_j im I_{Erg} führt es mit Closure(I_j) weiter, bis zum kein neues Item erzeugt wird.

Falls wir am Anfang ein Startset I_0 mit S'→ ·S, {*$*}; erzeugen, das komplette Set muss durch Closure([S'→ ·S, {*$*}]) erzeugt werden. Sei B = S eine Variable ist, muss alle LR(0)-Items, die S auf der linken Seite und Punkt am linksten steht, ausgefunden werden: S→ ·bA und S→ ·aB.

In diesem Fall ist β = ε (Leerzeichen), deswegen nehmen wir First(*$*) = *$* als neues lookahead-Symbol, damit erzeugen wir zwei neue LR(1)-Items: [S→ ·bA, {*$*}] und [S→ ·aB, {*$*}]. Dieses Prozess wird mit Closure([S→ ·bA, {*$*}]) und Closure([S→ ·aB, {*$*}]) immer weitergeführt, bis zum es kein neues Item mehr einzufügen gibt. In diesem Fall sind die beiden einzugenbendes Symbole Terminals, deswegen wird es bei die beiden Funktionen nichts passieren. Schließlich bekommen wir:

I_0 = Closure([S'→ ·S, {*$*}])

= {[S'→ ·S, {*$*}], [S→ ·bA, {*$*}], [S→ ·aB, {*$*}]} + Closure([S→ ·bA, {*$*}]) + Closure([S→ ·aB, {*$*}])

= {[S'→ ·S, {*$*}], [S→ ·bA, {*$*}], [S→ ·aB, {*$*}]}

3.5 Erstellung von DFA

Nach der Erzeugung der LR(1)-Items sind wir schon bereit für die Erstellung eines DFAs. Falls so ein DFA als Graph bezeichnet wird, wird es aus vielen Itemsets als Knoten und Transaktionen mit Eingabe als Kanten bestehen.

Ein Zustand S_i enthält eine Menge von LR(1)-Items. Wir fangen immer mit der S_0 an. Sei I_s das LR(0)-Item von Startsymbol mit derem Punkt an der Anfangsposition und *$* das Endsymbol, so können wir den Angfangszustand so erstellen: $S_0 = \{Closure(I_s, \$)\}$. Dann bekommen wir $S_0 = \{[S' \rightarrow \cdot S, \{\$\}], [S \rightarrow \cdot bA, \{\$\}], [S \rightarrow \cdot aB, \{\$\}]\}$. Falls alle Punkte der Items sich eine Position nach rechts bewegen, dann gibt es drei mögliche Eingabe S, a und b, d.h. es existiert drei Transaktionen von S_0 zu anderen Zustände, dies beschreiben wir mit Funktion GoTo(S_0, S), GoTo(S_0, b) und GoTo(S_0, a). Dadurch erzeugen wir drei neue LR(1)-Items: $[S' \rightarrow S\cdot, \{\$\}]$, $[S \rightarrow b\cdot A, \{\$\}]$, $[S \rightarrow a\cdot B, \{\$\}]$.

1) Es wird ein neuer Zustand erstellt, falls ein neu erzeugendes LR(1)-Item in keinem existierenden Zustand auftreten. So geht folgender Prozess:

2) Falls das neu erzeugende LR(1)-Item schon im existierenden Zustand S' auftritt, ist nun eine neue Transaktion in S' hinzufügen: GoTo(aktuelle Zustand, einzugebendes Symbol) = S' ;

S_0:	Closure([S'→ ·S, {$}]) = {[S'→ ·S, {$}], [S→ ·bA, {$}], [S→ ·aB, {$}]};
	GoTo(S_0, S) = S_1, GoTo(S_0, b) = S_2, GoTo(S_0, a) = S_3
S_1:	Closure([S'→ S·, {$}]) = {[S'→ S·, {$}]};
S_2:	Closure([S→ b·A, {$}]) = {[S→ b·A, {$}], [A→·bAA, {$}], [A→·aS, {$}], [A→·a, {$}]};
	GoTo(S_2, A) = S_4, GoTo(S_2, b) = S_5, GoTo(S_2, a) = S_6
S_3:	Closure([S→ a·B, {$}]) = {[S→ a·B, {$}], [B→·bS,{$}], [B→·b,{$}], [B→·aBB,{$}]};
	GoTo(S_3, B) = S_7, GoTo(S_3, b) = S_8, GoTo(S_3, a) = S_9
S_4:	Closure([S→ bA·, {$}]) = {[S→ bA·, {$}]};
S_5:	Closure([A→b·AA, {$}]) = {[A→ b·AA, {$}], [A→·bAA, {a, b}], [A→·aS, {a, b}], [A→·a, {a, b}]};
	GoTo(S_5, A) = S_{10}, GoTo(S_5, b) = S_{11}, GoTo(S_5, a) = S_{12}

S_6:	Closure([A→a·S, {\$}], [A→a·, {\$}]) = {[A→a·S, {\$}], [A→a·, {\$}], [S→ ·bA, {\$}], [S→ ·aB, {\$}]};
	GoTo(S_6, S) = S_{13} , GoTo(S_6, b) = S_2 , GoTo(S_6, a) = S_3
S_7:	Closure([S→ aB·, {\$}]) = { S→ aB·, {\$}] };
S_8:	Closure([B→b·S,{\$}], [B→b·,{\$}])= {[B→b·S,{\$}], [B→b·,{\$}], [S→ ·bA, {\$}], [S→ ·aB, {\$}]};
	GoTo(S_8, S) = S_{14} , GoTo(S_8, b) = S_2 , GoTo(S_8, a) = S_3
S_9:	Closure([B→a·BB,{\$}])= {[B→a·BB,{\$}], [B→·bS,{a, b}], [B→·b,{a, b}], [B→·aBB,{a, b}]};
	GoTo(S_9, B) = S_{15} , GoTo(S_9, b) = S_{16} , GoTo(S_9, a) = S_{17}
S_{10}:	Closure([S→ bA·A, {\$}])= {[S→ bA·A, {\$}], [A→·bAA, {\$}], [A→·aS, {\$}], [A→·a, {\$}]};
	GoTo(S_{10}, A) = S_{18} , GoTo(S_{10}, b) = S_5 , GoTo(S_{10}, a) = S_6
S_{11}:	Closure([A→b·AA, {a, b}])= {[A→b·AA, {a, b}], [A→·bAA, {a, b}], [A→·aS, {a, b}], [A→·a, {a, b}]};
	GoTo(S_{11}, A) = S_{19} , GoTo(S_{11}, b) = S_{11} , GoTo(S_{11}, a) = S_{12}
S_{12}:	Closure([A→a·S, {a, b}], [A→a·, {a, b}])= {[A→a·S, {a, b}], [A→a·, {a, b}], [S→ ·bA, {a, b}], [S→ ·aB, {a, b}]};
	GoTo(S_{12}, S) = S_{20} , GoTo(S_{12}, a) = S_{21} , GoTo(S_{12}, b) = S_{22}
...	

Dieser Prozess wiederholt sich immer bis zum es keinen neuen Zustand im DFA zu erzeugen werden. Vgl.[Wat07]

3.6 Erstellung von Action-Tabelle und GoTo-Tabelle

Nachdem wir schon ein komplettes DFA haben, sind nun die zugehörige Tabellen zu erstellen.

Da ein eingegebenes Symbol entweder ein Terminal oder eine Variable sein soll, erstellen wir auch zwei unterschiedliche Tabellen dafür - die Action-Tablle für Terminals und GoTo-Tablle für Variablen.

Zuerst wir nummieren alle Produktionen von G':

P' = { (1)S'→ S\$, (2)S→bA, (3)S→aB, (4)A→bAA, (5)A→aS, (6)A→a, (7)B→aBB, (8)B→bS, (9)B→b}

Die Spalten entprechen alle Symbole, und die Reihen entprechen den Zustandsindexe.

Wir prüfen für jeder Zustand S_x im DFA:

1) Falls Zustand S_x ein LR(1)-Item enthält, auf deren linker Seite die Startvariable steht, und der Punkt am Endposition ist, trägt in Zelle (x, \$) = „acc".

2) Falls Zustand S_x ein LR(1)-Item mit Punkt an der Endposition enthält, welches zum

Produktion P_i gehört und hat lookahead-Symbol t, dann trägt in Zelle (x,t) = „r"+i.

3) Falls beim Zustand S_x eine Transaktion mit Eingabe von Symbol n nach Ziel von Zustand S_z existiert:

i) falls n ∈ X, trägt in Zelle (x, n) = „s"+z;

ii) falls n ∈ V, trägt in Zelle (x, n) = „g"+z;

Damit erfüllen wir folgende Tabelle:

Zustand	Action-Tablle			GoTo-Tabelle			
	a	b	$	S'	S	A	B
0	s3	s2			g1		
1			acc				
2	s6	s5				g4	
3	s9	s8					g7
4			r2				
5	s12	s11				g10	
6	s3	s2	r6		g13		
7			r3				
8	s3	s2	r9		g14		
9	s17	s16					g15
…							

3.7 Prüfung von Eingabe

Eine beliebige Eingabe können wir nach der vorher erstellten Action/GoTo-Tabelle prüfen, ob diese von der gegebenen Grammatik akzeptiert wird. Für so einen Prozess brauchen wir einen Stack, wo Zustandsindex oder eingegebenes Symbol am Top des Stacks hingefügt wird (push). Das oberste Zahl auf dem Stack repränsentiert den Index aktuelles Zustands.

Die Ergebnistablle hat drei Spalten - Stack, Remaining-input und Comments.

Beispielweise ist die Eingabe „ab".

Es fängt nur mit eine 0 als Zustandindex auf dem Stack an, gleichzeitig ist Remaning-input ab$ und Comments „Initial". Jetzt sollen wir das erste Symbol „a" eingeben, deswegen suchen wir in der Action/GoTo- Tabelle nach Zelle (0, a) und es steht „s3" für „Shift and go to State 3". Dann suchen wir nach Zelle (3, b) wegen aktuelles Zustands 3

und einzugebenes Symbols „b”.

1) Shift: Das einzugebendes Symbol sollen wir auf dem Stack pushen und nach entsprechendem Zustand gehen. Dannach soll das aktuelle Zustandsindex auch gepusht werden.

Stack	Remaining input	Comments
0	ab$	Initial
0a3	b$	Shift
0a3b8	$	Shift

In der Zelle (8, $) finden wir aber etwas anders „r9”, welche repräsentiert „Reduction by 9. Production”. Da wir im Kapitel 3.6 alle Produktionen nummeriert haben, ist es hier nach der Produktion B→b aufzulösen.

2)Reduce: Sei n Länge der rechten Seite von entsprechender Produktion, pop 2*n mals, damit das oberste Stack das aktuelle Zustandindex i nach der Reduktion entspricht. Push die Variable V an der linken Seite. Suchen wir dann nach Zelle (i,V).

3)Goto: Für alle eingegebene Variable muss dem aktuellen Zustand wecheseln und deren Index auf dem Stack pushen.

Stack	**Remaining input**	**Comments**
0a3b8	$	Shift
0a3B7	$	Reduce by (9)B→b
0S1	$	Reduce by (3)S→aB
0S1	$	acc

Schließlich kommen wir zum Zustand 1 mit Eingabe $, wobei in der Zelle (1, $) „acc” für akzeptiert steht. Das bedeutet wir können das Prozess beenden und das Ergebnis ausgeben.

4)Rejected : Falls während eines Parsing-Prozesses eine gesuchte Zelle der Action/GoTo-Tabelle leer ist, so wissen wir dass die gesamte Eingabe nicht von der gegebenen Grammtik akzeptierbar ist.

4 Implementierung der beiden Parser

Für die Implementierung der zwei Parsers habe ich mich entscheidet, mit Java zu arbeiten. Da Java ein Objekt orientierte Sprache ist, ist es einfacher und deutlicher, jede Klasse nur um seine eigene Aufgaben kümmert. Vorallem habe ich eine Klasse für kontextfreie Grammatik und eine Klasse für deren Produktion aufgebaut. Dann überlege ich, wie der Benutzer seine eigene Grammatik als eine txt-Datei eingeben könnte. Für CYK habe ich nur eine Klasse implementiert, aber für LR(1) mehrere wegen die Komplexität der DFA.
Alle Funktionen werden nach dem Konzept implementiert, wie es im Kapitel 2 und 3 erklärt, deshalb wird im Folgenden nur seine Struktur vorgestellt.

4.1 Komponente einer Grammatik

Nach Definition 2.1 hat eine Grammatik hauptsächlich 4 Komponente, und zwar Menge der Terminals, Menge der Variablen, Startvariable und Menge der Produktionen, diese werden alle als Attribute in der Grammar-Klasse bezeichnet.
Jedes Symbol wird als einzelnen Character bezeichnet, deshalb ist die Menge der Variablen und Terminals in einem String gepeichert. Die Produktion hat auf linker Seite nur ein Symbol, deswegen hat die Klasse Produktion ein String Attribut „left". Auf rechter Seite ist es aber möglich, mehr als eine Regel zu speichern, die jeweilig mehrere Symbole erhalten und als einen String bezeichnet werden. Dafür hat die Klasse Produktion ein ArrayList „right", wobei alle Regeln auf der rechten Seite gespeichert werden.
Im folgenden werden die Hauptfunktionen von der Klasse Grammar vorgestellt. Einige dafür entwickelte Hilfsfunktionen werden nicht betrachtet.

void getGrammarFrom(File f, File f2, int countX) :
Um eine vollständige Grammatik zu erzeugen, braucht man zwei Text-Dateien und die Anzahl der Terminals in dieser Funktion eingeben. Einer davon speichert alle Terminals und Variablen, der andere speichert das Startsymbol und Regeln. Falls man die Grammatik richtig erzeugen möchte, muss die Trenner und Format beachtet werden.

Für Alphabet.txt sollen erstens die Terminlas, dann Variablen gespeichert werden. Alle Symbole werden durch Zeilenumbruchszeichen „\n" getrennt.
Für Productions.txt soll am Anfang mit „#start" das Startsymbol verdeutlichen. Die Produktionen werden durch ein „#" am Anfang getrennt, dahinter steht das Symbol auf der linker Seite. Die Regeln werden dann mit Zeilenumbruchszeichen „\n" getrennt, wobei die Symbole durch eine Leertaste getrennt werden.
Nehmen wir wieder das Beispielsgrammatik:
Bsp.: G = ({S, A, B}, {a,b}, P, S) mit P = {S→bA|aB, A→bAA|aS|a, B→aBB|bS|b}, so sieht die Textdateien wie folgenden aus:
(Bitte beachten, am Ende der Datei immer eine Zeile frei zu lassen)

Alphabet.txt:

```
a
b
S
A
B

```

Productions.txt:

```
#start
S
#S
b A
a B
#A
b A A
a S
a
#B
a B B
b S
b

```

Zusammen mit einer Verschlüsselung ermöglicht dies eine Benennung von Symbolen als langer String. Jedes eingegebene Symbol wird zu einem einzelnen Character verschlüsselt und zusammen in einem Map eingefügt, wobei das originale Symbol als Key und der Character als Value gespeichert.

void convertToCNF():

Falls eine Grammatik diese Funktion aufrufen, wird sie sich in Chomsky Normalform umwandeln. Die konkrete Funktionsweise entspricht das Kapitel 2.1. Dies dient als eine wichtige Vorbereitung des CYK-Parsers.

void simplizeLR():

Funktioniert wie im Kapitel 3.1 beschrieben, es wird geprüft, ob die Startvariable auf der

rechter Seite auftritt. Dann werden alle Produktionen getrennt, bis zum jede nur eine Regel auf rechter Seite enthält. Die Funktion wird nur als Vorbereitung des LR-Parsers aufgerufen.

4.2 Die Klasse CYK

Um eine CYK-Klasse zu erstellen braucht man beim Konstruktor eine Grammatik einzugeben. In dieser Klasse wird es geprüft, ob eine Eingabe von gespeicherte Grammatik nach dem CYK Algorithmus akzeptiert wird.

boolean checkGrammar(String input):

Zunächst wird die Eingabe durch eine Verschlüsselungsfunktion der Grammtik auch umwandelt. Dann konventiert es die originale Grammatik in CNF. Nun erstellt diese Funktion eine leere Tabelle im Form von einem Array. Für eine Eingabe mit n Symbole braucht diese Tabelle n Zeilen, d.h. Array der Länge n. Diese Tabelle wird dann erfüllt, wie es im Kapitel 2.3 vorgestellt ist. Schließliche wird geprüft, ob die letzte Zelle das Startvariable enthält, falls es so stimmt, dann wird „true" zurückgegeben, ansonst „false".

4.3 Die Klasse LR1

Die Klasse LR1 ist aber nicht so einfach wie CYK aufgebaut. Als Attribute hat diese Klasse trotz der gegebenen Grammtik noch ein DFA, eine Tabelle und ein Stack. Für Erzeugung eines neuen LR1-Objekts wird automatisch ein DFA der gegebenen Grammtik erstellen, dafür benötigt man auch andere Klassen wie LR0Item, LR1Item, State und Transition.

boolean checkGrammar(String input):

Ähnlich wie in der Klasse CYK wird erstens die Eingabe verschlüsselt. Nachdem das DFA schon komplett ist, erstellt diese Funktion eine Action/GoTo-Tabelle nach Kapitel 3.6. Diese Tabelle ist ein zwei-dimensionales Array von String. Wie im Kapitel 3.7 beschrieben wird es Schritt für Schritt geprüft, ob „acc" in der gesuchte Zelle auftritt. In diesem Fall wird „true" zurückgegeben, ansonsten „false".

4.4 Die Klasse DFA

Bei der Erzeugung eines neuen DFAs muss eine Grammatik eingegeben werden. Durch einen Aufruf der simplizeLR() Funktion ist diese Grammtik bereit für das weitere Prozess umformt. Dann erzeugt dies die LR0Items von aller Produktionen. Es wird auch ein Array für First-Symbole erzeugt. Bevor man mit der Zustanderzeugung anfangt wird es noch ein Anfangszustand mit Index 0 erstellen, der einen LR1Item von der Startvariable mit Punkt am linksten und das Endsymbol als lookahead enthält. Die komplette Erstellung des DFAs erfolgt sofort im Konstruktor.

ArrayList<LR0Item> createLR0Items(Production p)

In dieser Funktion wird eine Produktion eingegeben und dadurch eine Menge der LR0Items erzeugt. Das neue LR0Item weißt, wie es im nächsten Schritt aussehen soll und welche Position der Punkt steht. Die Attribut ruleIndex speichert das Index der eingegebenen Produktion von der Produktionsmenge der Grammtik. Alle dadurch entstehende LR0Item wird in eine Ergebnismenge eingefügt und schließlich zurückgegeben.

void createFirst()

Wie im Kapitel 3.2 vorgestellt, wird in dieser Funktion eine Menge im Form von String von First-Symbole für jede Terminal und Variable erstellt und in dem Array first gespeichert.

void createNextState(State s)

Mit einem Anfangszustand kann diese Funktion mithilfe der Closure()-Funktion ein neuer Zustand oder neue Transaktion in das DFA einfügen. Die closure() funktioniert genauso wie im Kapitel 3.4 beschrieben. Welche LR1Item zu erzeugen und wo diese hin sollen, entscheidet sich diese Funktion nach dem Schritte vom Kapitel 3.5.

Die Klassen State und Transition

In der Klasse State wird nur ihrer Index, Inhalt und Transitionen zu anderen als Attribute gespeichert. In der Transition gibt es ein String für die Eingabe und ein State für den

Zielzustand. Falls wir das DFA als ein Graph bezeichnet, dann dient die beiden Klassen als Knoten und Kanten, also diese gehören zur Komponenten des DFAs.

4.5 MainClass: Comparison

In der MainClass des ganzen Programms sind die beiden Klassen CYK und LR1 zu erstellen und dann die Parsing-Prozesse durchzuführen. Deren Ergebnisse und Laufzeit werden in der Konsole ausgegeben.

Es gibt in dem Folder der Jar-Datei ein Defaultbeispiel von YACC C Grammatik[Deg95], aber ist es auch möglich, eigene Grammatik zu nutzen. Am Anfang des Programms kann man durch Eingabe auf der Konsole den Mode selbst wählen, also „1” für Default C Grammatik und „2” für eigene Grammatik.

Falls man Mode 1 ausgewählt hat, dann wird gefragt ob die Parser seine eigene Eingabe prüfen soll oder einfache für die Laufzeit testen. Für Testen wird nochmal gefragt, wie lange die Eingabe sein soll. Damit wird eine Eingabe mit n mal „IDENTIFIER { }“ geprüft werden, wobei diese von den beiden Parser immer akzeptiert werden soll.

Falls man Mode 2 ausgewählt hat, dann muss man die Location der Txt-Dateien und Anzahl der Terminals eingeben. In diesem Mode könnten die Parser nur die Eingabe durch Konsole prüfen.

Bitte bemerken, dass für jeder Lauf des Programms nur eine Grammtik erstellt werden kann. Nach dem Ausgabe der Ergebnisse hat man aber noch die Möglichkeit, weiter mit anderer Eingaben zu testen.

Ein vollständiges Programmlauf sieht wie folgenden aus:

```
Select a mode please:
 1 - use the default C grammar
 2 - use your own grammar(Format same as the default)
1
Default C grammar chosen
Select a mode please:
 1 - check your own phrase
 2 - test run
2
How long should the input be?
(A test for n*"IDENTIFIER { } ")
Please insert n:
200
CYK Approved.
LR1 Approved.
CYK runs 58817 ms.
LR1 runs 41770 ms.
Continue? (j/n)
j
Select a mode please:
 1 - check your own phrase
 2 - test run
1
Please insert your phrase(with space bewteen the terminals):
 For example: b b a
INT IDENTIFIER = CONSTANT ;
CYK Approved.
LR1 Approved.
CYK runs 18 ms.
LR1 runs 84012 ms.
Continue? (j/n)
n
```

```
Select a mode please:
 1 - use the default C grammar
 2 - use your own grammar(Format same as the default)
2
Please insert the location of Alphabet text file:
C:\Users\ychen\Desktop\Alphabets-ExampleGrammar.txt
Please insert the amout of Terminals:
2
Please insert the location of Production rules text file:
C:\Users\ychen\Desktop\Rules-ExampleGrammar.txt
Read text file sucessed.
Grammar:
X = {a b }
V = {S A B }
S = S
P = {S  ->
        A
A  ->
        B A

B  ->
        a B
        b
}
Please insert your phrase(with space bewteen the terminals):
 For example: b b a
a a b b
CYK Approved.
LR1 Approved.
CYK runs 2 ms.
LR1 runs 7 ms.
Continue? (j/n)
```

5 Messungen und Vergleich

Das oben genannte Programm ermöglicht die Tests mit unterschiedliche Grammtiken und Eingaben, die zeigen, wie sich die Laufzeiten der beiden Parser im Praxis verhalten werden. Dafür nehme ich die YACC C Grammatik als Beispiel, welche insgesamt 63 Produktionen und 210 Regeln enthält. Ziel des Tests ist um das Verhältnis der jeweiligen Laufzeiten von CYK-Parser und LR(1)-Parser zu finden. Dies erfolgt durch Prüfung mit der Eingaben von unterschiedlicher Länge.

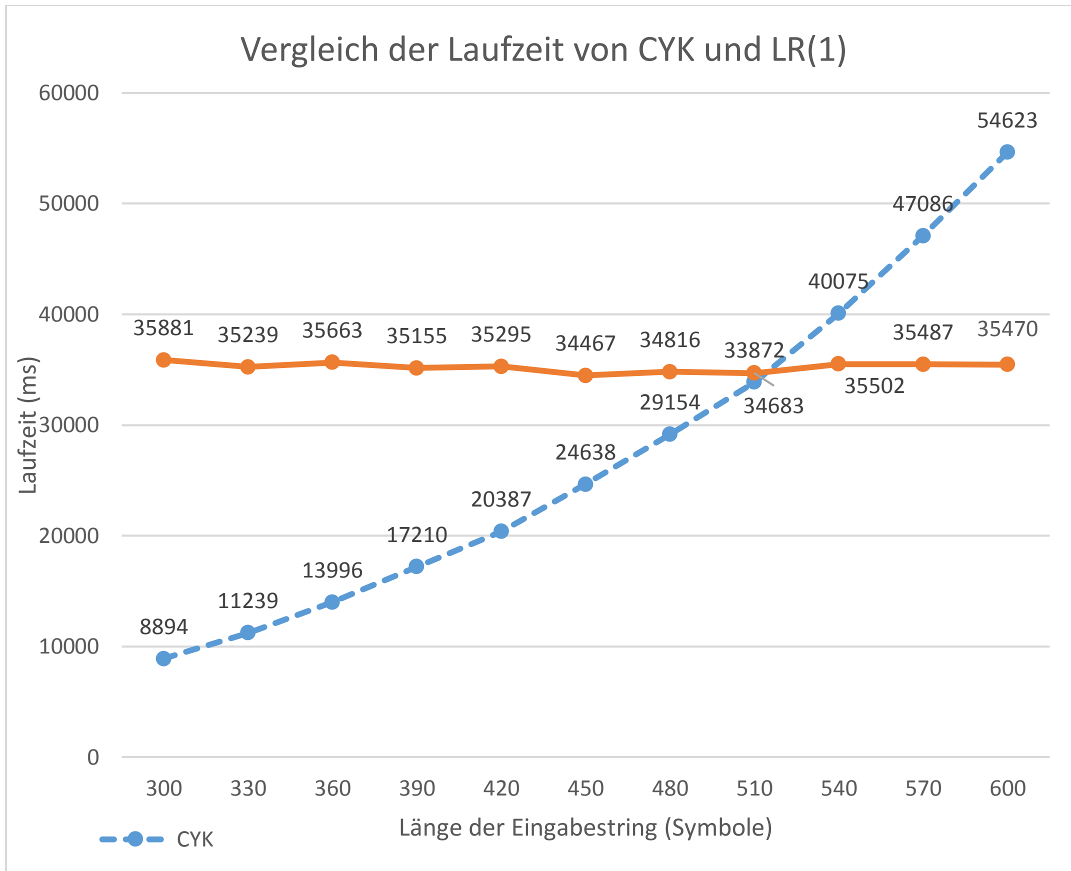
Da bei einer abgelehnten Eingabe nicht alle Schritte der Parsing-Prozess durchgeführt werden und damit die Laufzeit instabil ist, wähle ich nur die Eingabe, die von C Grammar akzeptiert wird. In dem Testfälle ist die Eingabe n mal „IDENTIFIER { } “, also n*3 Symbole, wobei n das von Benutzer eingegebenes Zahl ist.

Beim n = 1 ist es deutlich zu sehen, dass LR(1)-Parser viel mehr Laufzeiten braucht als CYK-Parser wegen seine komplizierte Vorbereitungen des Prozesses. Besonders für so eine umfangreiche Grammatik, die Anzahl der Zustände im DFA und die Aufwand der Action/GoTo-Tabelle ist viel größer als die Tabelle von CYK.

```
How long should the input be?
(A test for n*"IDENTIFIER { } ")
Please insert n:
1
CYK Approved.
LR1 Approved.
CYK runs 23 ms.
LR1 runs 39282 ms.
```

Aber können wir nun bestimmen, dass die LR(1)-Parser immer länger dauert? Um diese Frage zu beantworten, müssen wir erstens mit längere Eingabe testen.

Deshalb werden im folgenden Diagramm nur die Laufzeiten mit Eingabe ab n = 100 bzw. 300 Symbole ausgewählt. Es wurde insgesamt vier Tests durchgeführt, die im Diagramm gezeigte Daten sind die Durchschnitte davon.



Von diesem Diagramm kann man sofort ablesen, dass die Linie von LR(1) eher linear aussieht, aber die von CYK eine Kurve ist. Die beiden Linien treffen sich ungefähr bei der Eingabelänge von 520 Symbole. D.h. für eine Eingabestring, der länger als 510 Symbole ist, ist die Laufzeit von CYK-Parser größer als LR(1)-Parser. Beim n = 200 bzw. 600 Symbole ist den Unterschied sehr groß geworden.

6 Fazit

In dieser Arbeit wird gezeigt, dass LR(1)-Parser nicht immer wenigere Laufzeit als CYK-Parser hat, obwohl es theoretisch eine lineare Laufzeit hat. Falls wir nur die Vorbereitungen der beiden Parser betrachtet, werden wir uns sofort merken, dass der Vorbereitungsprozess von LR(1) viel kompliezierter als der von CYK ist. Es ist besonders deutlich zu sehen bei einer umfangreichen Grammatik. Dies entspricht auch die im Kapitel 6 gezeigte Ergebnisse der Testfälle: bei Prüfung einer kurzen Eingabe ist CYK-Parser deutlich effizienter als LR(1)-Parser.

Trotzdem ist dieses Verhältnis bei einer sehr langen Eingabe nicht mehr gültig. Wie es im Diagramm von Kapitel 6 gezeigt wird, je länger die Eingabe ist, desto schneller erwächst die Laufzeit von CYK. Da bei den Tests nur einen kleinen Teil von der Beispielsgrammatik genutzt werden, stimmit die gezeigte Grenze der Eingabeslänge nicht immer, wobei die Laufzeit von LR1 die von CYK überschreitet. Dies unterscheidet sich bei verschiedener Eingabe. Falls wir im Praxis einen sehr langen Text durch den Parser kompilieren möchte, ist der LR(1)-Parser immer noch die erste Wahl.

Literaturverzeichnis

[MAK88] Robert N. Moll, Michael A. Arbib und A.J. Kfoury, *An Introduction To Formal Language Threory*, 1988, Springer-Verlag New York Inc.

[HU79] John E. Hopcroft, Jeffrey D. Ullman, *Introduction to Automata Theory, Languages and Computation*, 1979, Addision-Wesley Publishing Company Inc.

[Wat07] Stephen M. Watt, *LR(1) Parsing Table Example*, 2007, University of Western Ontario, http://www.orcca.on.ca/~watt/home/courses/2007-08/cs447a/notes/LR1%20Parsing%20Tables%20Example.pdf

[Deg95] Jutta Degener, *ANSI C Yacc grammar*, 1995, https://www.lysator.liu.se/c/ANSI-C-grammar-y.html#direct-declarator