

Proseminar

Syntaktische Analyse von Programmiersprachen

Thema

LR(k) Syntaxanalyse Teil I

ausgearbeitet von
Claudia Bieg
Mat.nr.:2012201

Inhaltsverzeichnis

Einleitung	S.3
1. Zuverlässige Präfixe	S.5
• zuverlässige Kellerstrings	S.5
• zuverlässige Präfixe	S.10
2. LR(k) gültige Items	S.12
• k-Item / LR(k)-gültiges Item	S.12
• LR(k)-Äquivalenz	S.13
• Kanonische LR(k)-Maschine	S.14
• Algorithmus zur Konstruktion der kanonischen LR(k)-Maschine für eine Grammatik	S.18
3. Kanonische LR(k)-Parser	S.21
• LR(k)-reduce- und shift-actions	S.21
• Kanonische LR(k)-Parser	S.22
• Algorithmus zur Konstruktion von Parsingaktionen	S.24
4. LR(k)-Grammatiken	S.28
• LR(k)-Grammatik	S.28
• Charakterisierung von LR(k)-Grammatiken	S.29

Einleitung:

Im Folgenden wird ein Verfahren vorgestellt, die sogenannte **LR(k) Syntaxanalyse**, das kontextfreie Grammatiken syntaktisch analysiert. Das „**L**“ bedeutet, daß in der LR(k)-Syntaxanalyse die Eingabe von links nach rechts abgearbeitet wird. Das „**R**“ steht für einen Rechtsparser, d.h. für einen Parser, der eine Rechtsableitung in umgekehrter Reihenfolge bildet. Das „**k**“ steht für die Anzahl der Symbole, die in der Eingabe vorausgeschaut werden dürfen, bevor eine Syntaxanalyseentscheidung getroffen wird. Wird (k) weggelassen, so wird $k=1$ angenommen.

LR(k)-Parser arbeiten wie folgt: bevor sie ein weiteres Eingabesymbol einlesen, machen sie für den bereits gelesenen und analysierten Teil der Eingabe alle möglichen Reduktionen, d.h. LR(k)-Parser konstruieren einen Syntaxbaum, indem sie einzelne Teilbäume erzeugen. Dabei beginnen sie beim Blattwort, dem Eingabewort, und bilden über die Reduktionen einzelne Teilbäume mit Nichtterminalen als Wurzeln. Für jeden neu gelesenen Teil des Eingabeworts werden Teilbäume unter einem Nichtterminalknoten verbunden, bis man beim Startsymbol S angelangt ist, der Wurzel des Syntaxbaumes. Man kann LR(k)-Parser so konstruieren, daß sie beinahe alle gebildeten Konstrukte einer kontextfreien Grammatik auf diese Art und Weise analysieren können.

Im 1. Kapitel werden wir uns mit Kellerstrings, die während einer akzeptierenden Rechnung im Keller erscheinen, beschäftigen, den sogenannten *zuverlässigen Präfixen*. Dazu betrachten wir die Eigenschaften der Kellerstrings von shift-reduce Parsern. Unser Ziel ist es, zu zeigen, daß zuverlässige Präfixe eine reguläre Sprache über dem Alphabet einer Grammatik bilden. Denn dann können wir einen endlichen Automaten finden, der diese Sprache akzeptiert.

Wir werden im 2. Kapitel eine Äquivalenzrelation über der Menge der zuverlässigen Präfixe kennenlernen, die sogenannte LR(k) Äquivalenz. Dann geben wir einen Algorithmus an, mit dessen Hilfe man die deterministische LR(k) Maschine einer Grammatik konstruieren kann. Im 3. Kapitel definieren wir den **kanonischen LR(k) Parser** mit Hilfe der LR(k)-Äquivalenz. Mit einem Algorithmus kann man diesen kanonischen LR(k)-Parser für eine kontextfreie Grammatik generieren. Es ist also möglich, eine

kontextfreie Grammatik zu schreiben und automatisch einen Parser für diese Grammatik zu erzeugen. Im 4. Kapitel schließlich werden wir sehen, daß der kanonische LR(k) Parser einer Grammatik korrekt ist und daß er terminiert.

1. zuverlässige Präfixe

Unser Ziel ist es, einen deterministischen Rechtsparser am Beispiel einer kontextfreien Grammatik zu konstruieren. Dazu betrachten wir die folgende Grammatik G_{ab} :

$$\begin{aligned} S &\rightarrow aA \mid bB, \\ A &\rightarrow c \mid dAd, \\ B &\rightarrow c \mid dBd. \end{aligned}$$

Die von der Grammatik erzeugte Sprache ist $L(G_{ab}) = \{(a|b)d^*cd^*\}$.

Da G_{ab} zwei Produktionsregeln mit der gleichen rechten Seite hat ($A \rightarrow c$, $B \rightarrow c$), ist eine Reduktion hier nicht eindeutig.

Wir versuchen, die Reduktion eindeutig zu machen, indem wir die Vorschau auf Strings der Länge k erweitern. Doch dabei stoßen wir direkt auf einen neuen Konflikt:

$$d^k c \mid d^k \rightarrow d^k A \mid d^k \quad \text{oder} \quad d^k c \mid d^k \rightarrow d^k B \mid d^k,$$

denn die reduce-action $A \rightarrow c$ ist auf $\$ad^k c \mid d^k\$$ anwendbar, wenn $ad^k cd^k$ akzeptiert werden soll. Analog ist die reduce-action $B \rightarrow c$ auf $\$bd^k c \mid d^k\$$ anwendbar, wenn $bd^k cd^k$ akzeptiert werden soll. Es handelt sich also um einen reduce-reduce Konflikt. Allein durch Erweiterung der Vorschau ist es nicht möglich, einen deterministischen Rechtsparser für die Grammatik G_{ab} zu entwickeln. Um dieses Problem zu lösen, führen wir den Begriff des **zuverlässigen Kellerstrings** ein:

Definition: (zuverlässiger Kellerstring)

Man nennt die Strings, die während einer akzeptierenden Rechnung eines shift-reduce Parsers im Keller erscheinen, **zuverlässige Kellerstrings**.

Ein String γ ist ein **zuverlässiger Kellerstring** eines Kellerautomaten M , falls für Eingabestrings w und y und finale Kellerinhalte γ_f gilt:

$$\gamma_s \mid w\$ \xRightarrow{*} \$\gamma \mid y\$ \xRightarrow{*} \$\gamma_f \mid \$,$$

wobei γ_s der ursprüngliche Kellerinhalt von M ist.

Beispiel:

Die Menge der zuverlässigen Kellerstrings für den shift-reduce Parser von G_{ab} ist:

$$\begin{aligned} &\{\epsilon\} \cup \{S\} \\ &\cup \{ad^n \mid n \geq 0\} \cup \{ad^n c \mid n \geq 0\} \\ &\cup \{ad^n A \mid n \geq 0\} \cup \{ad^n Ad \mid n \geq 1\} \\ &\cup \{bd^n \mid n \geq 0\} \cup \{bd^n c \mid n \geq 0\} \\ &\cup \{bd^n B \mid n \geq 0\} \cup \{bd^n Bd \mid n \geq 1\} \end{aligned}$$

Auf jeden zuverlässigen Kellerstring können Parsingaktionen angewandt werden, aber nur wenige ergeben auch wieder einen zuverlässigen Kellerstring. Man fordert daher, daß eine Parsingaktion nur angewandt werden darf, wenn sie „gültig“ ist, d.h. wenn aus ihr ein zuverlässiger Kellerstring resultiert.

Definition : (gültige Aktion)

Eine Aktion r eines Kellerautomaten M heißt für einen zuverlässigen Kellerstring γ von M *gültig*, falls gilt:

$$\gamma | y \stackrel{r}{\implies} \gamma' | y' \text{ in } M,$$

für Eingabewörter y und y' und für einen zuverlässigen Kellerstring γ' .

Da aber die Menge der zuverlässigen Kellerstrings meistens unendlich groß ist, ist es im allgemeinen unmöglich herauszufinden, welche Aktionen für welchen Kellerstring gültig sind. Daher teilt man die Menge der zuverlässigen Kellerstrings in Äquivalenzklassen auf. Zwei Kellerstrings gehören zur selben Äquivalenzklasse, wenn sie die gleichen gültigen Aktionen besitzen.

Beispiel:

Für die Grammatik G_{ab} gilt:

Äquivalenzklasse:

- [ϵ]
- [$ad^* \cup bd^*$]
- [ad^*c]
- [aA]
- [$ad^+A \cup bd^+B$]
- [ad^+Ad]
- [bd^*c]
- [bB]
- [bd^+Bd]
- [S]

gültige Aktionen:

- shift a, shift b
- shift c, shift d
- reduce by $A \rightarrow c$
- reduce by $S \rightarrow aA$
- shift d
- reduce by $A \rightarrow dAd$
- reduce by $B \rightarrow c$
- reduce by $S \rightarrow bB$
- reduce by $B \rightarrow dBd$
-

Die Idee dabei ist, die Äquivalenzklassen als Kellersymbole des Parsers zu verwenden. Wir ersetzen in den Parsingaktionen die Grammatiksymbole X , die ursprünglich links der Abgrenzung $|$ standen, durch Äquivalenzklassen der Form $[\delta X]$, wobei δX ein zuverlässiger Kellerstring ist. Dann definiert man entsprechend:

shift – action

Für jeden zuverlässigen Kellerstring δ und für jedes Terminal a gilt die shift-action:

$$(sa) \quad [\delta] | a \rightarrow [\delta][\delta a] |$$

δa muß hierbei ein zuverlässiger Kellerstring sein.

reduce – action

Für jeden zuverlässigen Kellerstring δ und jede Produktionsregel $A \rightarrow X_1 \dots X_n$ (jedes X_i sei ein einzelnes Symbol, $1 \leq i \leq n$) gilt die reduce-action :

$$(ra) \quad [\delta][\delta X_1] \dots [\delta X_1 \dots X_n] | \rightarrow [\delta][\delta A] |$$

$\delta X_1, \dots, \delta X_1 \dots X_n$ und δA müssen hierbei zuverlässige Kellerstrings sein.

Beispiel:

Beim Parser für G_{ab} sind **shift-actions**

$$[\epsilon] | a \rightarrow [\epsilon][ad^* \cup bd^*] \quad (\text{shift } a)$$

$$[\epsilon] | b \rightarrow [\epsilon][ad^* \cup bd^*] \quad (\text{shift } b)$$

und **reduce-actions**

$$[ad^* \cup bd^*][ad^*c] | \rightarrow [ad^* \cup bd^*][ad^+A \cup bd^+B] | \quad (\text{reduce by } A \rightarrow c)$$

$$[ad^* \cup bd^*][bd^*c] | \rightarrow [ad^* \cup bd^*][ad^+A \cup bd^+B] | \quad (\text{reduce by } B \rightarrow c)$$

Wir benutzen reguläre Ausdrücke, um die Äquivalenzklassen zu bezeichnen. Sei E ein regulärer Ausdruck. Dann bezeichnet $[E]$ die Äquivalenzklasse für ein w in $L(E)$, d.h. $[E] = [w]$ für alle w in $L(G)$. Wir werden noch sehen, daß Äquivalenzklassen tatsächlich reguläre Sprachen von Grammatiken sind, und daß sie daher durch reguläre Ausdrücke bezeichnet werden können.

Leider führt unsere Konstruktion zu einem neuen **reduce-reduce** Konflikt

$$[ad^* \cup bd^*][ad^*c] | \rightarrow [ad^* \cup bd^*][ad^+A \cup bd^+B] | \quad (\text{reduce by } A \rightarrow c),$$

$$[ad^* \cup bd^*][ad^*c] | \rightarrow [ad^* \cup bd^*][aA] | \quad (\text{reduce by } A \rightarrow c),$$

$$[ad^* \cup bd^*][bd^*c] | \rightarrow [ad^* \cup bd^*][ad^+A \cup bd^+B] | \quad (\text{reduce by } B \rightarrow c),$$

$$[ad^* \cup bd^*][bd^*c] | \rightarrow [ad^* \cup bd^*][bB] | \quad (\text{reduce by } B \rightarrow c),$$

und zu neuen **shift-shift** Konflikten:

$$[ad^* \cup bd^*] | c \rightarrow [ad^* \cup bd^*][ad^*c] | \quad (\text{shift } c),$$

$$[ad^* \cup bd^*] | c \rightarrow [ad^* \cup bd^*][bd^*c] | \quad (\text{shift } c),$$

$$\begin{aligned}
[ad^+A \cup bd^+B] | d &\rightarrow [ad^+A \cup bd^+B] [ad^+Ad] | \quad (\text{shift } d), \\
[ad^+A \cup bd^+B] | d &\rightarrow [ad^+A \cup bd^+B] [bd^+Bd] | \quad (\text{shift } d).
\end{aligned}$$

Der Grund für diese Konflikte ist, daß die Einteilung in Äquivalenzklassen nicht fein genug ist.

Beispiel:

Betrachte die zuverlässigen Kellerstrings ad^n und ad^nA für $n \geq 0$. Für alle $n \geq 0$ gehören die Strings ad^n zu der selben Äquivalenzklasse $[ad^* \cup bd^*]$. Die Strings ad^nA für $n \geq 0$ jedoch werden in zwei verschiedene Äquivalenzklassen aufgeteilt: $[aA]$ und $[ad^+A \cup bd^+B]$. Daher kommt es zum obigen reduce-reduce Konflikt.

Wenn man zwei äquivalente Kellerstrings δ_1 und δ_2 mit dem selben Symbol X zu Kellerstrings δ_1X und δ_2X erweitert, erwartet man, daß δ_1X und δ_2X auch wieder äquivalent sind. Wir fordern daher:

Die Äquivalenzklassen sollen rechts – invariant sein.

Weiter fordern wir:

Zwei äquivalente Kellerstrings g und g sollen mit dem selben Symbol enden, d.h.: $g:1 = g:1$.

Ansonsten ist nicht gewährleistet, daß die Parsingaktionen eindeutig sind.

Beispiel:

Um die beiden obigen Forderungen zu erfüllen, verfeinert man die Äquivalenzklassen der Grammatik G_{ab} :

- (1) Die Klasse $[ad^* \cup bd^*]$ wird unterteilt in $[a]$, $[ad^+]$, $[b]$, $[bd^+]$.
- (2) Die Klasse $[ad^+A \cup bd^+B]$ wird unterteilt in $[ad^+A]$ und $[bd^+B]$.

Die verfeinerten Äquivalenzklassen stellen die Knoten im folgenden Übergangsgraphen dar. Die Kanten von Knoten $[\delta]$ zu Knoten $[\delta X]$ sind mit dem Symbol X gekennzeichnet, wenn δ und δX zuverlässige Kellerstrings sind. Man kann den Graph als endlichen Automaten interpretieren, wobei $[\epsilon]$ der Startzustand ist und eine vorgegebene Äquivalenzklasse $[\gamma]$ der einzige Endzustand ist. Dann entspricht die vom Automat akzeptierte Sprache gerade $[\gamma]$.

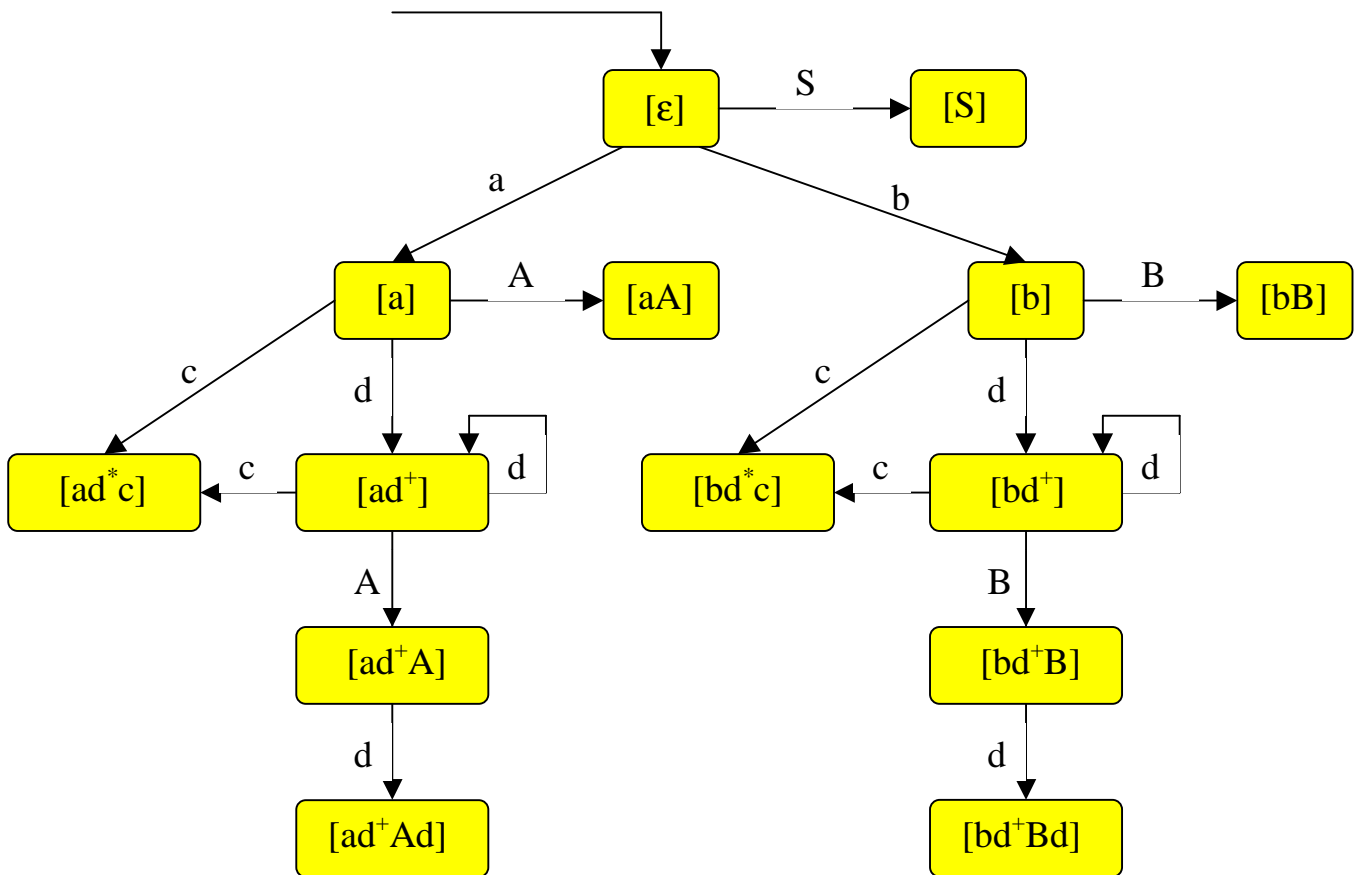


Bild 1.1 Übergangsgraph für zuverlässige Kellerstrings des shift-reduce Parsers der Grammatik G_{ab} : $S \rightarrow aA \mid bB$, $A \rightarrow c \mid dAd$, $B \rightarrow c \mid dBd$

Beispiel:

Ausgewählte shift-actions des Parsers für die Grammatik G_{ab} sind:

$$\begin{aligned}
 r_1 &= [\epsilon] \mid a \rightarrow [\epsilon][a] \mid & \tau(r_1) &= \epsilon \\
 r_2 &= [a] \mid d \rightarrow [a][ad^+] \mid & \tau(r_2) &= \epsilon \\
 r_3 &= [a] \mid c \rightarrow [a][ad^*c] \mid & \tau(r_3) &= \epsilon
 \end{aligned}$$

Ausgewählte reduce-actions des Parsers für die Grammatik G_{ab} sind:

$$\begin{aligned}
 r_4 &= [a][ad^*c] \mid \rightarrow [a][aA] \mid & \tau(r_4) &= A \rightarrow c \\
 r_5 &= [\epsilon][a][aA] \mid \rightarrow [\epsilon][S] \mid & \tau(r_5) &= S \rightarrow aA
 \end{aligned}$$

Wenn man alle Parsingaktionen entwickelt, sieht man, daß der entstandene Parser ein deterministischer Rechtsparser für G_{ab} ist (genauer: ein LR(0) Parser).

Beispiel:

An der folgenden Berechnung kann man erkennen, daß wir einen Rechtsparser erhalten haben:

$$\begin{aligned}
 & \$[\epsilon] \mid ac\$ \xRightarrow{r_1} \$[\epsilon][a] \mid c\$ \xRightarrow{r_3} \$[\epsilon][a][ad^*c] \mid \$ \xRightarrow{r_4} \$[\epsilon][a][aA] \mid \\
 & \xRightarrow{r_5} \$[\epsilon][S] \mid \$.
 \end{aligned}$$

Zum Abschluß dieses 1. Kapitels wollen wir zuverlässige Kellerstrings von shift-reduce Parsern grammatisch charakterisieren und einige Eigenschaften von LR Parsern betrachten.

Definition : (zuverlässiges Präfix , Griff)

Sei $G = (V, T, P, S)$ eine Grammatik. Ein String $\gamma \in V^*$ ist ein **zuverlässiges Präfix** von G , wenn in G gilt:

$$S \xrightarrow{rm}^* \delta A y \implies \delta \alpha \beta y = \gamma \beta y$$

für Strings $\delta \in V^*$, $y \in T^*$ und Produktionsregel $A \rightarrow \alpha \beta$ aus P .
 $\alpha \beta$ bezeichnet man als **Griff (handle)**.

Es gilt: ein zuverlässiges Präfix ist bis auf das Ende soweit wie möglich reduziert. Mit Hilfe der Definition ist das folgende Lemma offensichtlich:

Lemma 1.1:

Jedes Präfix eines zuverlässigen Präfixes ist ein zuverlässiges Präfix.

Der nächste Satz zeigt den Zusammenhang zwischen den zuverlässigen Präfixen einer Grammatik und den zuverlässigen Kellerstrings des zugehörigen shift-reduce Parsers:

Satz 1.1:

Sei $G = (V, T, P, S)$ eine Grammatik und M ihr shift-reduce-Parser. Jeder zuverlässige Kellerstring von M ist entweder S oder ein zuverlässiges Präfix von G . Umgekehrt ist jedes zuverlässige Präfix von G ein zuverlässiger Kellerstring von M , falls G reduziert ist.

Wenn also G eine reduzierte Grammatik ist, dann entsprechen die zuverlässigen Präfixe von G gerade den zuverlässigen Kellerstrings des entsprechenden Parsers.

Man kann sogar zeigen, daß für eine Grammatik $G = (V, T, P, S)$ die Menge aller zuverlässigen Präfixe eine reguläre Sprache über dem Alphabet V ist. Dazu geben wir eine rechtslineare Grammatik an, d.h. eine reguläre Grammatik, die die zuverlässigen Präfixe erzeugt. Zu einer gegebenen Grammatik $G = (V, T, P, S)$ definiert man eine Grammatik $G_{VP} = (V_{VP}, V, P_{VP}, [S])$ mit:

$$\begin{aligned} V_{VP} &= \{ [A] \mid A \text{ ist Nichtterminal in } V \setminus T \} \\ P_{VP} &= \{ [A] \rightarrow \alpha \mid A \rightarrow \alpha \beta \text{ ist eine Vorschrift in } P \} \\ &\cup \{ [A] \rightarrow \alpha [B] \mid A \rightarrow \alpha B \beta \text{ ist eine Vorschrift in } P, B \text{ ist ein} \\ &\quad \text{Nichtterminal in } V \setminus T \text{ und } \beta \text{ leitet einen String aus } T^* \text{ ab.} \} \end{aligned}$$

Dabei ist $[A]$, für ein Nichtterminal A in $V \setminus T$, ein neues Symbol, daß nicht in V enthalten ist. Die Bedingung „ β leitet einen String aus T^* ab“ ist wichtig, wenn G nicht reduziert ist.

Beispiel:

Die Grammatik $(G_{ab})_{VP}$ für G_{ab} ist:

$$\begin{aligned} [S] &\rightarrow \varepsilon \mid a \mid aA \mid b \mid bB \mid a[A] \mid b[B] \\ [A] &\rightarrow \varepsilon \mid c \mid d \mid dA \mid dAd \mid d[A] \\ [B] &\rightarrow \varepsilon \mid c \mid d \mid dB \mid dBd \mid d[B] \end{aligned}$$

Damit kann man nun den folgenden Satz beweisen:

Satz 1.2:

Jede Grammatik G kann in eine rechtslineare Grammatik transformiert werden, die die Menge aller zuverlässigen Präfixe erzeugt.

Beweisidee zu Satz 1.2:

Man kann zeigen: G kann in G_{VP} transformiert werden. In G gilt:

$$(1) \quad S \xRightarrow{*} \delta Ay \xRightarrow{*} \delta \alpha \beta y = \gamma \beta y$$

Dann kann man zeigen, daß in G_{VP} gilt:

$$(2) \quad [S] \xRightarrow{*} \delta [A] \xRightarrow{*} \delta \alpha = \gamma$$

Beachte: $[A] \rightarrow \alpha$ ist Vorschrift in G_{VP}

γ ist also ein zuverlässiges Präfix von G , das auch in G_{VP} erzeugt wird.

Umgekehrt:

Wenn $[S]$ in G_{VP} einen String $\gamma \in V^*$ ableitet, dann gilt (2) für ein $\delta \in V^*$ und $A \rightarrow \alpha \beta$ in P . Dann zeigt man, daß (1) für ein y gilt.

Damit ist jedes γ , daß G_{VP} von erzeugt wird, ein zuverlässiges Präfix von G .

Also:

$L(G_{VP})$ ist gerade die Menge der zuverlässigen Präfixe von G .

Da rechtslineare Grammatiken auch regulär sind, folgt aus Satz 1.2 direkt Satz 1.3 und wir sind am Ziel dieses Kapitels:

Satz 1.3:

Für jede Grammatik $G = (V, T, P, S)$ gilt:

Die Menge aller zuverlässigen Präfixe von G ist eine reguläre Sprache über V .

2. LR(k) - gültige Items

In diesem Kapitel wollen wir eine Äquivalenzrelation über der Menge der zuverlässigen Präfixe definieren, die sogenannte LR(k)-Äquivalenz. Unser Ziel dabei wird sein, die kanonische LR(k)-Maschine für eine Grammatik zu konstruieren. Dazu benötigen wir zuerst ein paar grundlegende Definitionen:

Definition : (Position, k-Item, Kern, LR(k)-gültig, Vorschaustring)

Sei $G = (V, T, P, S)$ eine Grammatik. Eine **Position** von G ist eine punktierte Produktionsregel der Form $A \rightarrow \alpha.\beta$, wobei $A \rightarrow \alpha\beta$ eine Produktionsregel in P ist, und der Punkt ein Symbol, das nicht in V enthalten ist.

Ein Paar der Form $[A \rightarrow \alpha.\beta, y]$ ist ein **k-Item** für $k \geq 0$, wenn

$A \rightarrow \alpha.\beta$ eine Position von G ist und der String $y \in k:T^*$.

Die Position $A \rightarrow \alpha.\beta$ wird als **Kern** des Items bezeichnet und der String y als **Vorausschaustring** (**lookahead**).

Ein Item $[A \rightarrow \alpha.\beta, y]$ ist **LR(k)-gültig** für einen String $\gamma \in V^*$, wenn eine Rechtsableitung

$$S \xRightarrow{rm}^* \delta Az \xRightarrow{rm} \delta \alpha \beta z = \gamma \beta z \quad \text{und } k:z = y$$

in G für String $\delta \in V^*$ und $z \in T^*$ existiert.

Das LR(k)-gültige Item $[A \rightarrow \alpha.\beta, y]$ beschreibt dann die folgende Analysesituation:

$$S \xRightarrow{rm}^* \delta Az \xRightarrow{rm} \delta \alpha \beta z \xRightarrow{rm}^* \delta \alpha v z \xRightarrow{rm}^* \delta u v z,$$

wobei $\delta\alpha$ ein zuverlässiges Präfix ist (d.h. wenn α zu einem Terminalwort u abgeleitet wird, und β zu einem Terminalwort v , dann ist die Reduktion von u nach α bereits geschehen, während die Reduktion von v nach β noch nicht begonnen hat.).

Ist ein Item $[A \rightarrow \alpha.\beta, y]$ LR(k) gültig für einen String $\gamma \in V^*$, dann bedeutet das: Es gibt eine Rechtsableitung, in der γ zuverlässiges Präfix einer Rechtssatzform ist (d.h. in einer Satzform, die in einer Rechtsableitung auftritt) und $A \rightarrow \alpha\beta$ eine der möglicherweise gerade „bearbeiteten“ Produktionen ist. $A \rightarrow \alpha\beta$ ist dann später ein Kandidat für eine Reduktion.

Den nächsten Satz folgert man direkt aus der Definition:

Satz 2.1:

Wenn γ ein zuverlässiges Präfix ist , dann ist mindestens ein Item LR(k)-gültig für γ .

Beispiel:

Die Items $[S \rightarrow .aA, \varepsilon]$ und $[S \rightarrow .bB, \varepsilon]$ sind LR(k)-gültig für ε , in der Grammatik G_{ab} , denn:

$$S \xrightarrow[\text{rm}]{}^0 S \xrightarrow[\text{rm}]{} aA ,$$

$$S \xrightarrow[\text{rm}]{}^0 S \xrightarrow[\text{rm}]{} bB ,$$

und $k:\varepsilon = \varepsilon$ für $k \geq 0$.

Für die Definition der LR(k) Äquivalenz wird die folgende Menge aller LR(k)-gültigen Items für einen String von Bedeutung sein:

Definition : ($VALID_k$)

Sei $G = (V,T,P,S)$ eine Grammatik und $\gamma \in V^*$, dann gilt für alle $k \geq 0$:

$$VALID_{LR(k)}^G(\gamma) = \{ I \mid I \text{ ist ein LR(k)-gültiges Item für } \gamma \}$$

Beispiel :

In der Grammatik G_{ab} gilt für alle $k, n \geq 0$:

$$VALID_k(\varepsilon) = \{ [S \rightarrow .aA, \varepsilon], [S \rightarrow .bB, \varepsilon] \}$$

$$VALID_k(a) = \{ [S \rightarrow a.A, \varepsilon], [A \rightarrow .c, \varepsilon], [A \rightarrow .dAd, \varepsilon] \}$$

$$VALID_k(ad^{n+1}) = \{ [A \rightarrow d.Ad, k:d^n], [A \rightarrow .c, k:d^{n+1}], [A \rightarrow .dAd, k:d^{n+1}] \}$$

Damit sind wir jetzt in der Lage, die LR(k)-Äquivalenz zu definieren:

Definition : (LR(k)-äquivalent)

String γ_1 ist **LR(k)-äquivalent** zu String γ_2 , wenn gilt:

$$VALID_k(\gamma_1) = VALID_k(\gamma_2)$$

Die Relation ρ_k bezeichnet die LR(k)-Äquivalenz für G . Sind zwei Strings γ_1 und γ_2 LR(k)-äquivalent, dann schreibt man: $\gamma_1 \rho_k \gamma_2$.

Die Definition beruht auf einer Relation, nämlich der Identität. Reflexivität, Symmetrie und Transitivität sind damit erfüllt. Daraus kann man folgern, daß es sich bei der LR(k)-Äquivalenz auch tatsächlich um eine Äquivalenzrelation handelt:

Satz 2.2:

Für eine Grammatik $G = (V,T,P,S)$ und $k \in \mathbb{N}$ ist die LR(k)-Äquivalenz ρ_k für G eine Äquivalenzrelation über V^* . Außerdem ist ρ_k von endlichem Index, d.h. die Anzahl der verschiedenen Äquivalenzklassen unter ρ_k ist endlich.

Beweisidee zu Satz 2.2:

Da $[\gamma_1]_{\Delta k} = [\gamma_2]_{\Delta k}$ genau dann gilt, wenn $\text{VALID}_k(\gamma_1) = \text{VALID}_k(\gamma_2)$, kann man die Äquivalenzklassen bijektiv auf die Mengen der k-Items $\text{VALID}_k(\gamma)$ abbilden, d.h. beide Mengen haben gleichen Index. Es gibt endlich viele verschiedene k-Itemmengen $\text{VALID}_k(\gamma)$, also ist ρ_k von endlichem Index.

Beispiel :

In der Grammatik G_{ab} gilt für alle $n \geq 0$:

die zuverlässigen Präfixe ad^{n+1} sind alle LR(0)-äquivalent, weil für alle $n, m \geq 0$ gilt: $\text{VALID}_0(ad^{n+1}) = \text{VALID}_0(ad^{m+1})$.

Wir bezeichnen die LR(0)-Äquivalenzklasse für ad^{n+1} mit $[ad^+]_0$.

Die LR(0)-Äquivalenzklasse für ad^{n+1} kann wie folgt in LR(k)-Äquivalenzklassen zerlegt werden:

$$[ad^+]_0 = [ad]_k \cup \dots \cup [ad^{k+1}d^*]_k$$

Aus dem Beispiel kann man erkennen, daß die LR(k+1)-Äquivalenz eine Verfeinerung der LR(k)-Äquivalenz ist. Insbesondere ist bei der Grammatik G_{ab} die LR(k)-Äquivalenzklasse entweder eine einzige LR(k+1)-Äquivalenzklasse oder eine Vereinigung von zwei Äquivalenzklassen, wie beispielsweise die LR(k)-Äquivalenzklasse $[ad^{k+1}d^*]_k$ die Vereinigung aus den LR(k+1)-Äquivalenzklassen $[ad^{k+1}]_{k+1}$ und $[ad^{k+2}d^*]_{k+1}$ ist.

Das legt den folgenden Satz nahe:

Satz 2.3:

Für eine Grammatik $G = (V, T, P, S)$ und $k, l \in \mathbb{N}$ mit $k \leq l$ ist die LR(l)-Äquivalenz eine Verfeinerung der LR(k)-Äquivalenz. Das bedeutet, daß jede LR(k)-Äquivalenzklasse die Vereinigung von LR(l)-Äquivalenzklassen ist.

Da für jede Grammatik $G = (V, T, P, S)$ die LR(k)-Äquivalenzklassen $[\gamma]_k$ von den k-Itemmengen $\text{VALID}_k(\gamma)$ abhängen, kann man jede Menge $\text{VALID}_k(\gamma)$ als endliche Darstellung der entsprechenden Klasse $[\gamma]_k$ betrachten. Deswegen stellt die Vereinigung aller Mengen $\text{VALID}_k(\gamma)$, $\gamma \in V^*$, eine endliche Repräsentation der gesamten LR(k)-Äquivalenz dar. Diese kann als endlicher Automat aufgefaßt werden, den man die *kanonische LR(k)-Maschine* nennt.

Definition : (deterministische LR(k)-Maschine)

Die **kanonische** (oder **deterministische**) **LR(k)-Maschine** der Grammatik G ist ein endlicher Automat. Die Zustandsmenge ist die Vereinigung aller Mengen $\text{VALID}_k(\gamma)$, das Eingabealphabet ist V , der Startzustand ist $\text{VALID}_k(\epsilon)$ und die Übergangsmenge besteht aus allen Funktionen der Form

$$\text{VALID}_k(\gamma)X \rightarrow \text{VALID}_k(\gamma X),$$

wobei γ ein String aus V^* ist und X ein Symbol aus V .

Beispiel:

Man nennt die kanonische LR(0)-Maschine kurz LR(0)-Maschine.

Wir betrachten nun die LR(0)-Maschine für die Grammatik G_{ab} :

$S \rightarrow aA \mid bB, A \rightarrow c \mid dAd, B \rightarrow c \mid dBd.$

Sie ist in Bild 2.1 dargestellt. Den leeren Zustand \emptyset lassen wir weg, ebenso alle Übergänge von und zu ihm. Um die Äquivalenz zu dem Übergangsgraph aus Bild 1.1 zu betonen, beschriften wir jeden Zustand $VALID_k(\gamma)$ mit $[E]$, wobei E ein regulärer Ausdruck ist, der die LR(0)-Äquivalenzklasse $[\gamma]_0$ bezeichnet.

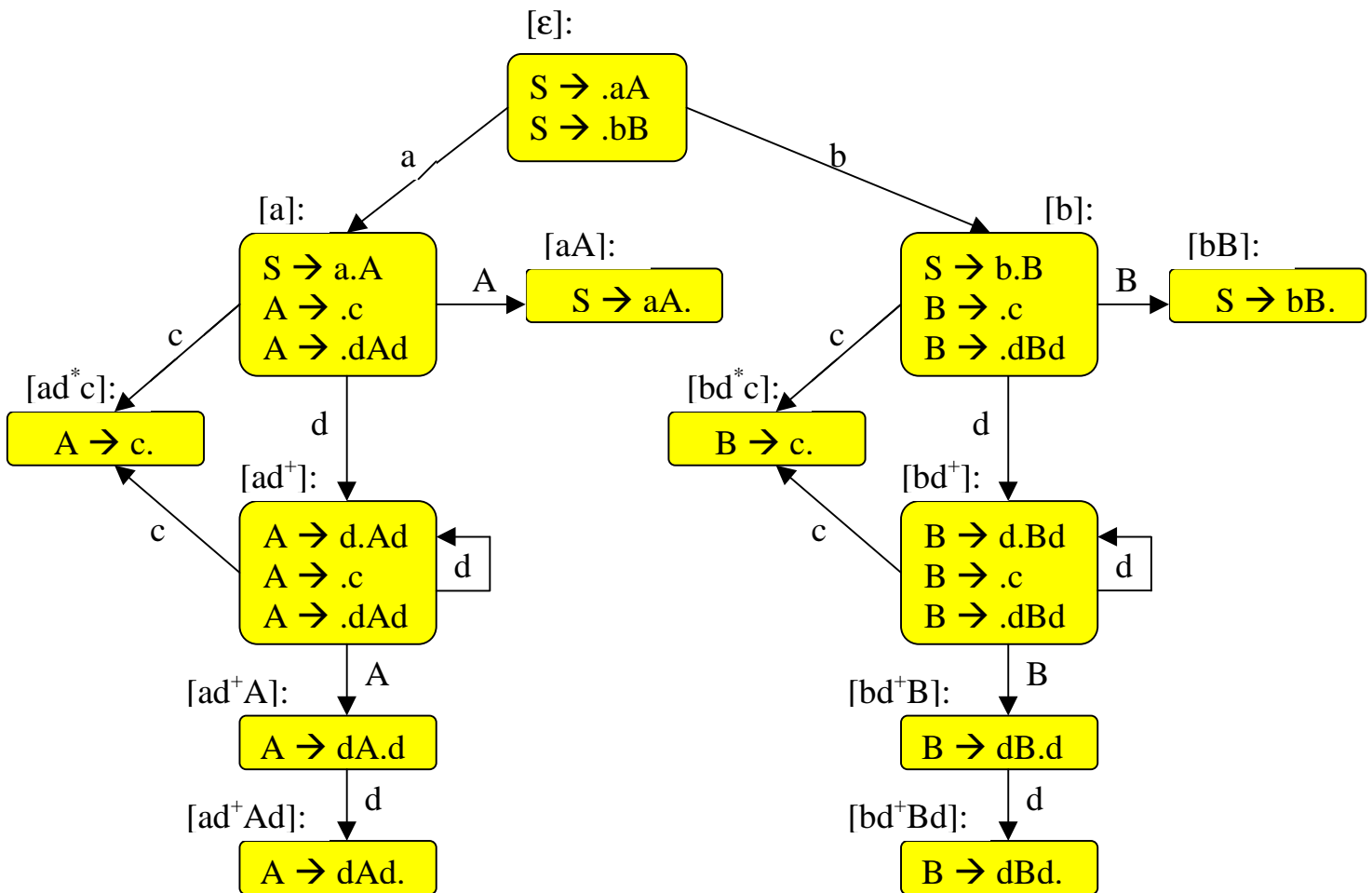


Bild 2.1 Die LR(0)-Maschine für die Grammatik G_{ab} :
 $S \rightarrow aA \mid bB, A \rightarrow c \mid dAd, B \rightarrow c \mid dBd.$

Wir wollen nun zeigen, daß die LR(k)-Äquivalenz beide Forderungen aus Kapitel 1 erfüllt. Zur Erinnerung: die LR(k)-Äquivalenz soll rechts-invariant sein und zwei äquivalente zuverlässige Präfixe sollen immer mit dem selben Symbol enden. Außerdem werden wir sehen:

Für eine Grammatik G und $k \in \mathbb{N}$ ist die kanonische LR(k)-Maschine ein ϵ -freier, endlicher, deterministischer Automat, der die LR(k)-Äquivalenz induziert, d.h.

zwei Strings γ_1 und γ_2 sind genau dann LR(k)-äquivalent, wenn der angenommene Zustand beim Lesen von γ_1 mit dem angenommenen Zustand beim Lesen von γ_2 übereinstimmt. Damit ist die Rechts-Invarianz erfüllt. Außerdem gilt, daß jeder nichtleere Zustand $VALID_k(\gamma)$ ein eigenes Eingangssymbol hat, d.h. ein Symbol X, so daß alle Übergänge zu $VALID_k(\gamma)$ über X gehen. Das bedeutet, daß zwei äquivalente zuverlässige Präfixe immer mit dem selben Symbol enden.

Wir wollen noch einmal die LR(0) Maschine aus Bild 2.1 genauer betrachten. Wie sind die Mengen $VALID_k(\gamma)$, also die Zustände der Maschine zusammengesetzt ?

Man beobachtet, daß wenn $VALID_k(\gamma)$ ein Item der Form $[A \rightarrow \alpha.B\beta, y]$ enthält, wobei B ein Nichtterminal ist, $VALID_k(\gamma)$ auch alle Items $[B \rightarrow \cdot\omega, z]$ enthält, wobei $B \rightarrow \omega$ eine Produktionsregel in der Grammatik ist und z ein String aus $FIRST_k(\beta y)$. In unserem Beispiel bedeutet das, wenn $[S \rightarrow a.A, \epsilon]$ in [a] enthalten ist, dann auch $[A \rightarrow \cdot c, \epsilon]$ und $[A \rightarrow \cdot dAd, \epsilon]$ in [a] enthalten. Das legt die folgende Definition nahe:

Definition: (LR(k)-Nachfolger, LR(k)-Vorgänger)

$[B \rightarrow \cdot\omega, z]$ ist ein **direkter LR(k)-Nachfolger** von $[A \rightarrow \alpha.B\beta, y]$, wenn gilt : $z \in FIRST_k(\beta y)$. Man schreibt : $[A \rightarrow \alpha.B\beta, y] \mathbf{desc}_{LR(k)} [B \rightarrow \cdot\omega, z]$

Ein Item ist ein **LR(k)-Nachfolger** von k-Item I, wenn es zu $\mathbf{desc}_{LR(k)}^*(I)$ gehört. Item I_1 ist ein (**direkter**) **LR(k)-Vorgänger** von Item I_2 , falls I_2 ein (direkter) LR(k)-Nachfolger von I_1 ist.

Weiter kann man erkennen, daß jede Itemmenge $VALID_k(\gamma X)$, wobei X ein einzelnes Symbol ist, alle Items der Form $[A \rightarrow \alpha.X\beta, y]$ enthält, wenn $[A \rightarrow \alpha.X\beta, y]$ ein Item aus $VALID_k(\gamma)$ ist. In unserem Beispiel, vgl. Bild 2.1, ist das Item $[A \rightarrow \cdot dAd, \epsilon]$ in [a]. Daher ist dann $[A \rightarrow d.Ad, \epsilon]$ in $[ad^+]$. Wir versuchen, diese Beobachtungen mit den folgenden Definitionen zu beschreiben:

Definition : (LR- (un-) wesentlich)

$[A \rightarrow \alpha.\beta, y]$ heißt **LR-wesentlich**, wenn α nicht leer ist. Wenn α leer ist, heißt das Item **LR-unwesentlich**. Wenn q eine Itemmenge ist, definiert man :

$$ESS_{LR}(q) = \{ I \in q \mid I \text{ ist LR-wesentlich} \}.$$

Definition :

Sei $X \in V$ und sei **passes-X** eine Relation in der Menge der k-Items, definiert durch :

$$[A \rightarrow \alpha.X\beta, y] \text{ passes-X } [A \rightarrow \alpha X.\beta, y]$$

für $A \rightarrow \alpha X\beta$ in P und $y \in k:T^*$.

Sei q eine k-Itemmenge und $X \in V$. **passes-X(q)** wird als die **Basis des X-Nachfolgers von q** bezeichnet. Man schreibt: $\text{BASIS}_{\text{LR}}(q,X)$. Damit gilt:

$$\text{BASIS}_{\text{LR}}(q,X) = \{ [A \rightarrow \alpha X.\beta, y] \mid [A \rightarrow \alpha.X\beta, y] \in q \}$$

Damit definiert man $\text{GOTO}_{\text{LR}}(q,X)$ als:

$$\text{GOTO}_{\text{LR}}(q,X) := \text{desc}_k^*(\text{BASIS}_{\text{LR}}(q,X)) := \text{passes-X desc}_k^*(q)$$

Die Menge $\text{GOTO}_{\text{LR}}(q,X)$ besteht also aus allen LR(k)-Nachfolgern aller Items der Form $[A \rightarrow \alpha X.\beta, y]$, für welche das Item $[A \rightarrow \alpha.X\beta, y]$ in q ist.

Wir wollen nun einen Algorithmus zur Konstruktion der kanonischen LR(k) Maschine einer Grammatik vorstellen. Dazu wird die Menge $\text{GOTO}_{\text{LR}}(q,X)$ von Bedeutung sein, denn sie beinhaltet alle Items aus der Menge $\text{BASIS}_{\text{LR}}(q,X)$ und alle deren Nachfolger. Damit kann man die Zustände der LR(k) Maschine konstruieren, wenn man den Anfangszustand kennt. Nach den folgenden beiden Lemmas, die für den Algorithmus wichtig sind, stellen wir diesen an einem Beispiel vor:

Lemma 2.4:

Sei $G = (V,T,P,S)$ eine Grammatik und γ ein String in V^* . Dann gilt:

$$\text{VALID}_k(\epsilon) = \text{desc}_k^*(\{ [S \rightarrow \cdot\omega, \epsilon] \mid S \rightarrow \omega \text{ in } P \})$$

$$\text{VALID}_k(\gamma) = \text{desc}_k^*(\text{ESS} (\text{VALID}_k(\gamma))) \text{ für } \gamma \neq \epsilon.$$

Lemma 2.5:

Sei $G = (V,T,P,S)$ eine Grammatik, γ ein String in V^* und X ein Symbol in V. Dann gilt für alle $k \geq 0$:

$$\text{VALID}_k(\gamma X) = \text{GOTO}(\text{VALID}_k(\gamma), X)$$

Lemma 2.4 beschreibt die Zustände der LR(k)-Maschine allgemein, während Lemma 2.5 die Konstruktion des Nachfolgezustands aus einem Zustand beschreibt. Damit läßt sich nun der folgende Algorithmus angeben:

Algorithmus zur Konstruktion der kanonischen LR(k)-Maschine für eine Grammatik $G = (V, T, P, S)$

Q_M enthält alle Zustände von M.

P_M enthält alle Übergänge in M.

q_s ist der Startzustand von M.

Berechne die Relation **desc_k**;

for alle $X \in V$ **do**

Berechne die Relation **passes** – X;

$q_s := \text{desc}_k^*(\{ [S \rightarrow \cdot \omega, \varepsilon] \mid S \rightarrow \omega \text{ in } P \});$

$Q_M := \{ q_s \};$

$P_M := \emptyset;$

repeat

for alle $q \in Q_M$ und $X \in V$ **do begin**

$q' := \text{passes} - X \text{ desc}_k^*(q);$

$Q_M := Q_M \cup \{ q' \};$

$P_M := P_M \cup \{ qX \rightarrow q' \};$

end

until nichts kann mehr zu Q_M und P_M hinzugefügt werden;

Damit der Algorithmus verständlicher wird, wollen wir ihn an einem Beispiel ausführen.

Beispiel :

Konstruktion der LR(0)-Maschine für die Grammatik G_{ab}

1. *Berechne die Relation **desc₀**;*

An dieser Stelle müßte man nun alle auftretenden Items untersuchen und alle ihre direkten LR(0)-Nachfolger bestimmen. Da dies zu aufwendig wird, beschränken wir uns auf die folgenden beiden Items:

$[S \rightarrow \cdot aA, \varepsilon]$

$[S \rightarrow \cdot bB, \varepsilon]$

Beide haben keinen direkten LR(0)-Nachfolger, da in jedem Item dem Punkt ein Terminal folgt.

2. ***for** alle $X \in V$ **do***

*Berechne die Relation **passes** – X;*

Auch hier beschränken wir uns auf $a, b \in V$, da nur hier der Punkt in jedem Item verschoben werden kann. (vgl. Schritt 1)

passes – a ($[S \rightarrow \cdot aA, \varepsilon]$) = $[S \rightarrow a \cdot A, \varepsilon]$

passes – b ($[S \rightarrow \cdot bB, \varepsilon]$) = $[S \rightarrow b \cdot B, \varepsilon]$

3. Wir bestimmen nun den Startzustand. Auch hier erhalten wir keine direkten LR(0)-Nachfolger, da in jedem Item dem Punkt ein Terminal folgt.

$$q_s := \text{desc}_0^* (\{ [S \rightarrow .aA, \epsilon], [S \rightarrow .bB, \epsilon] \}) \\ = \{ [S \rightarrow .aA, \epsilon], [S \rightarrow .bB, \epsilon] \}$$

4. Die Menge Q_M enthält anfangs nur den Startzustand, die Menge P_M ist leer.

$$Q_M := \{ q_s \}; \\ P_M := \mathbf{\emptyset};$$

5. repeat

for alle $q \in Q_M$ und $X \in V$ do begin

$$q' := \text{passes} - X \text{desc}_0^* (q);$$

$$Q_M := Q_M \hat{\cup} \{ q' \};$$

$$P_M := P_M \hat{\cup} \{ qX \rightarrow q' \};$$

end

until nichts kann mehr zu Q_M und P_M hinzugefügt werden;

Also: q_s ist zur Zeit der einzige Zustand in Q_M . Wir beschränken uns wieder auf $a, b \in V$. Damit erhält man die folgenden beiden neuen Zustände:

$$q_1' = \text{passes} - a \text{desc}_0^* (q_s) \\ = \{ [S \rightarrow a.A, \epsilon], [A \rightarrow .c, \epsilon], [A \rightarrow .dAd, \epsilon] \} =: [a]$$

$$q_1'' = \text{passes} - b \text{desc}_0^* (q_s) \\ = \{ [S \rightarrow b.B, \epsilon], [B \rightarrow .c, \epsilon], [B \rightarrow .dBd, \epsilon] \} =: [b]$$

Q_M und P_M werden die neuen Zustände, bzw. die neuen Übergänge hinzugefügt. Damit fährt man so lange fort, bis der Graph aus Bild 2.1 entsteht.

Wir haben also eine Äquivalenzrelation für eine Grammatik kennengelernt, mit deren Hilfe man einen Algorithmus zur Konstruktion der kanonische-LR(k) Maschine der Grammatik findet. Damit sind wir am Ziel dieses 2. Kapitels und können abschließend festhalten:

Satz 2.6:

Sei $G = (V, T, P, S)$ eine Grammatik, $k \in \mathbb{N}$, und M die kanonische LR(k)-Maschine für G . Dann gelten die folgenden Aussagen:

- (a) M ist deterministisch.
 (b) Wenn alle nicht leeren Zustände $\text{VALID}_k(\gamma)$ als Endzustände von M bestimmt sind, dann ist die von M akzeptierte Sprache die Menge der zuverlässigen Präfixe von G .

Beweisidee zu Satz 2.6:

(a) Seien γ_1, γ_2 Strings in V^* und X_1, X_2 Symbole aus V .
Man betrachtet das Übergangspaar:

$$\text{VALID}_k(\gamma_1)X_1 \rightarrow \text{VALID}_k(\gamma_1X_1) \text{ und } \text{VALID}_k(\gamma_2)X_2 \rightarrow \text{VALID}_k(\gamma_2X_2),$$

wobei beide auf die selbe Konfiguration anwendbar sein sollen. Dann gilt:

$$\text{VALID}_k(\gamma_1) = \text{VALID}_k(\gamma_2) \text{ und } X_1 = X_2.$$

Nach Lemma 2.5 :

$$\begin{aligned} \text{VALID}_k(\gamma_1X_1) &= \text{GOTO}(\text{VALID}_k(\gamma_1), X_1) \\ &= \text{GOTO}(\text{VALID}_k(\gamma_2), X_2) = \text{VALID}_k(\gamma_2X_2). \end{aligned}$$

Damit sind die Übergänge die selben, also ist M deterministisch.

(b) Wenn alle Zustände $\text{VALID}_k(\gamma)$ nicht leer sind, dann enthalten sie mindestens ein gültiges Item. Und zu jedem gültigen Item gehört gerade ein zuverlässiges Präfix.

Satz 2.7:

Für eine Grammatik $G = (V, T, P, S)$, $k \in \mathbb{N}$ gelten die folgenden Aussagen:

(a) Die LR(k)-Äquivalenz von G ist gerade die Äquivalenz, die durch die kanonische LR(k)-Maschine von G impliziert wird, d.h. Strings γ_1 und γ_2 sind genau dann LR(k)-äquivalent, wenn der angenommene Zustand beim Lesen von γ_1 mit dem angenommenen Zustand beim Lesen von γ_2 übereinstimmt.

(b) Die LR(k)-Äquivalenz von G ist rechts-invariant, d.h. immer dann, wenn γ_1 und γ_2 LR(k)-äquivalente Strings in V^* sind und X ein Symbol in V ist, sind γ_1X und γ_2X LR(k)-äquivalent.

(c) Zwei LR(k)-äquivalente zuverlässige Präfixe enden immer mit dem selben Symbol, d.h. immer wenn γ_1 und γ_2 LR(k)-äquivalente zuverlässige Präfixe sind, gilt: $\gamma_1:1 = \gamma_2:1$.

3. kanonische LR(k)-Parser

In diesem Kapitel wollen wir den Begriff des kanonischen LR(k)-Parsers für eine Grammatik $G = (V, T, P, S)$ anhand der Ergebnisse aus dem vorhergehenden Kapitel definieren. Um es einfacher zu machen, benutzen wir die $\$$ -erweiterte Grammatik für G , d.h. wir führen ein neues Startsymbol S' ein und erweitern P um $S' \rightarrow \$\$$. Damit erreichen wir, daß die Berechnungen sauber terminieren. Im Folgenden bezeichnen wir diese Grammatik mit G' . Das Kelleralphabet des kanonischen LR(k)-Parsers wird die Vereinigung aller LR(k)-Äquivalenzklassen der zuverlässigen Präfixe von G' sein. Wir bezeichnen diese Vereinigung mit $[G']_k$.

Definition : (LR(k)-reduce-action)

Sei $G = (V, T, P, S)$ eine Grammatik G' ihre $\$$ -erweiterte Grammatik und $k, n \in \mathbb{N}$. Eine Vorschrift der Form

$$(ra) \quad [\delta]_k [\delta X_1]_k \dots [\delta X_1 \dots X_n]_k \mid y \rightarrow [\delta]_k [\delta A]_k \mid y$$

wird als die **kanonische LR(k)-reduce-action** durch Produktionsregel $A \rightarrow X_1 \dots X_n$ mit **Vorausschau** y bezeichnet, wenn δ ein String in $\$V^*$ ist, X_1, \dots, X_n Symbole in V sind, $A \rightarrow X_1 \dots X_n$ eine Produktionsregel in P ist und y ein String in $k:T^*\$$, so daß gilt:

$$[A \rightarrow X_1 \dots X_n \dots, y] \in \text{VALID}_k(\delta X_1 \dots X_n)$$

Zu beachten sei, daß $\delta, \delta X_1, \dots, \delta X_1 \dots X_n$ zuverlässige Präfixe von G' sind, denn sie sind alle Präfixe von $\delta X_1 \dots X_n$. $\delta X_1 \dots X_n$ ist ein zuverlässiges Präfix, da ein Item $[A \rightarrow X_1 \dots X_n \dots, y]$ für $\delta X_1 \dots X_n$ gültig ist.

Definition : (LR(k)-shift-action)

Sei $G = (V, T, P, S)$ eine Grammatik, G' ihre $\$$ -erweiterte Grammatik und $k \in \mathbb{N}$. Eine Vorschrift der Form

$$(sa) \quad [\delta]_k \mid ay \rightarrow [\delta]_k [\delta a]_k \mid y$$

wird als die **kanonische LR(k)-shift-action** an Terminal a und **Vorausschau** ay bezeichnet, wenn δ ein String in $\$V^*$ ist, a ein Terminal in T ist und y ein String in $\max\{k-1, 0\}: T^*\$$ ist, so daß gilt:

$$[A \rightarrow \alpha a \beta, z] \in \text{VALID}_k(\delta) \text{ und } y \in \text{FIRST}_{\max\{k-1, 0\}}(\beta z)$$

für Produktionsregel $A \rightarrow \alpha a \beta$ in P und String z in $k: T^*\$$.

Auch hier sei zu beachten, daß δ und δa zuverlässige Präfixe von G' sind.

Beispiel:

Betrachte die $\$$ -erweiterte Grammatik G_a :

$S' \rightarrow \$S\$$

$S \rightarrow aA$

$A \rightarrow c$

$A \rightarrow dAd$

Wie erhält man z.B. das Terminalwort **adcd** ?
(Betrachte Vorausschaustrings der Länge 6)

shift \$
[ϵ]₆ | **\$adcd**\$ → [ϵ]₆[\$]₆ | **adcd**\$
shift a
→ [ϵ]₆[\$]₆[\$a]₆ | **dcd**\$
shift d
→ [ϵ]₆[\$]₆[\$a]₆[\$ad]₆ | **cd**\$
shift c
→ [ϵ]₆[\$]₆[\$a]₆[\$ad]₆[\$adc]₆ | **d**\$
reduce $A \rightarrow c$
→ [ϵ]₆[\$]₆[\$a]₆[\$ad]₆[\$adA]₆ | **d**\$
shift d
→ [ϵ]₆[\$]₆[\$a]₆[\$ad]₆[\$adA]₆[\$adAd]₆ | **\$**
reduce $A \rightarrow dAd$
→ [ϵ]₆[\$]₆[\$a]₆[\$aA]₆ | **\$**
reduce $S \rightarrow aA$
→ [ϵ]₆[\$]₆[\$S]₆ | **\$**
shift \$
→ [ϵ]₆[\$]₆[\$S]₆[\$S\$]₆ |
reduce $S' \rightarrow \$S\$$
→ [ϵ]₆[\$S']₆ |

Nun können wir den kanonischen LR(k)-Parser für eine Grammatik definieren:

Definition : (kanonischer LR(k)-Parser)

Der **kanonische LR(k)-Parser** für G ist der Kellerautomat mit Ausgabe mit Kelleralphabet $[G']_k$, Eingabealphabet T , Anfangskellerinhalten $[\$]_k$, Menge der Endkellerinhalte $\{ [\$]_k [\$S]_k \}$ und der Aktionenmenge, die aus allen reduce- und shift-actions besteht. Der Outputeffekt τ wird definiert, um jede reduce-action von Vorschrift r nach Vorschrift r und jede shift-action zum leeren String ϵ wie folgt zu bezeichnen:

(1) $\tau ([\delta]_k [\delta X_1]_k \dots [\delta X_1 \dots X_n]_k | y \rightarrow [\delta]_k [\delta A]_k | y) = A \rightarrow X_1 \dots X_n$

(2) $\tau ([\delta]_k | ay \rightarrow [\delta]_k [\delta a]_k | y) = \epsilon$

Man beachte, daß jede reduce-action der Form (ra) die Produktionsregel $A \rightarrow X_1 \dots X_n$ definiert. Wenn eine reduce-action auf zwei Arten dargestellt wird, sagen wir durch

$$[\delta]_k[\delta X_1]_k \dots [\delta X_1 \dots X_n]_k \mid y \rightarrow [\delta]_k [\delta A]_k \mid y$$

und

$$[\delta]_k[\delta Y_1]_k \dots [\delta Y_1 \dots Y_n]_k \mid y \rightarrow [\delta]_k [\delta B]_k \mid y,$$

dann gilt notwendig : $X_1=Y_1, X_2=Y_2, \dots, X_n=Y_n$ und $A=B$.

Also ist τ wohldefiniert .

Beispiel:

Betrachte die Grammatik $G_{abe} : S \rightarrow aA \mid bB, A \rightarrow \epsilon \mid cAd, B \rightarrow \epsilon \mid cBd$.

Dann ist die von der Grammatik erzeugte Sprache: $L(G_{abe}) = \{ (a|b)c^*d^* \}$.

Die Zustände VALID(ϵ) und VALID($$$$$) lassen wir weg, da sie im Parser nicht benötigt werden. Alle anderen Zustände sind durchnummeriert. Im Text bezeichnen wir Zustand i durch q_i . Betrachten wir nun die LR(0)-Maschine für

G_{abe} :

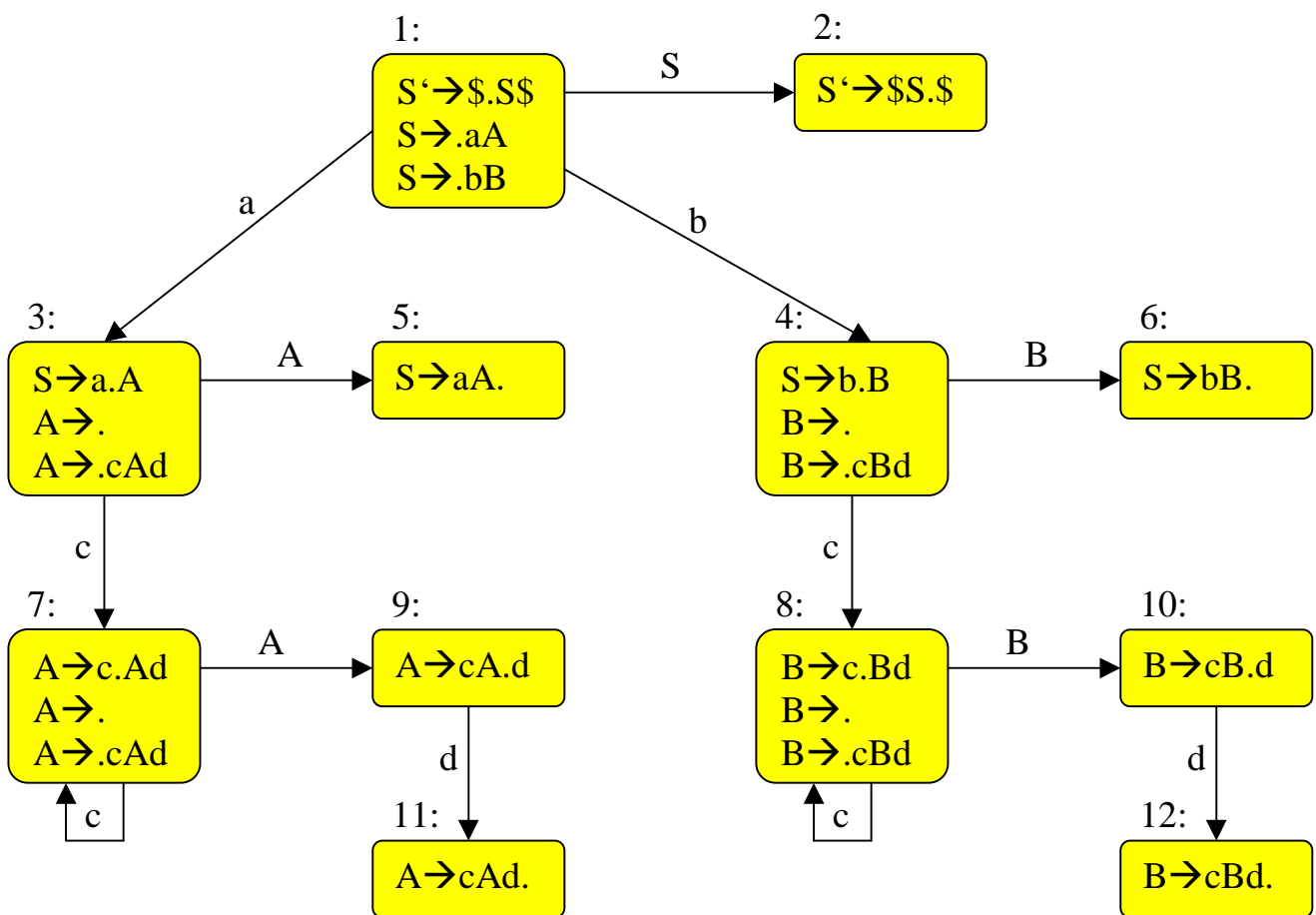


Bild 2.2 Die LR(0)-Maschine für die ϵ -erweiterte Grammatik G_{abe} :

$$S' \rightarrow $$$$, $S \rightarrow aA \mid bB, A \rightarrow \epsilon \mid cAd, B \rightarrow \epsilon \mid cBd$.$$

Die Aktionen des kanonischen LR(k)-Parsers erhält man aus der kanonischen LR(k)-Maschine, indem man den folgenden Algorithmus benutzt:

Algorithmus zur Konstruktion der Parsing Aktionen :

```

for alle Zustände q do begin
  for alle Produktionsregeln  $A \rightarrow X_1 \dots X_n$  von G (  $n \geq 0$  ), so daß gilt :
  ein Item in q hat den Kern  $A \rightarrow .X_1 \dots X_n$  do begin
    Sei  $q_0, q_1, q_2, \dots, q_n$  eine Folge von Zuständen, die von q bis zu den
    eingelesenen  $X_1, \dots, X_n$  reichen,
    dabei sei  $q_0 = q$  und
     $q_i = \text{GOTO}(q, X_i)$  mit  $i = 1, \dots, n$ 
     $q_A = \text{GOTO}(q, A)$ 
    for alle Items  $[A \rightarrow X_1 \dots X_n \cdot, y]$  in  $q_n$  do
      generiere die reduce-action
       $qq_1q_2 \dots q_n \mid y \rightarrow qq_A \mid y$ 
    end;
    for alle Items  $[A \rightarrow \alpha \cdot a \beta, z]$  in q mit  $a \in T$  do begin
       $q_a = \text{GOTO}(q, a)$ ;
      for alle Strings  $y \in \text{FIRST}_{\max\{k-1, 0\}}(\beta z)$  do
        generiere die shift-action
         $q \mid ay \rightarrow qq_a \mid y$ 
      end;
    end;
  end;
end;

```

Auch hier wollen wir den Algorithmus anhand eines Beispiels verständlicher machen:

Beispiel:

Konstruktion der Parsing Aktionen des LR(0)-Parsers der Grammatik G_{abE} :

1. *for alle Zustände q do begin*

Da es viel zu aufwendig ist, alle Zustände zu bearbeiten, beschränken wir uns hier auf Zustand $q_1 : S' \rightarrow \$.S\$$

$S \rightarrow .aA$

$S \rightarrow .bB$

(Am Ende werden alle anderen Parsing Aktionen angegeben.)

2. **for** alle Produktionsregeln $A \rightarrow X_1 \dots X_n$ von $G(n^30)$, so
 daß gilt : ein Item in q hat den Kern $A \rightarrow .X_1 \dots X_n$ **do begin**

Ein Item in q_1 hat den Kern $S \rightarrow .aA$, ein anderes den Kern $S \rightarrow .bB$.
 Betrachte also die beiden Produktionsregeln:

$$S \rightarrow aA, \quad S \rightarrow bB$$

3. Sei $q_0, q_1, q_2, \dots, q_n$ eine Folge von Zuständen, die von q bis zu den
 eingelesenen X_1, \dots, X_n reichen, dabei sei $q_0 = q$ und
 $q_i = \text{GOTO}(q, X_1, \dots, X_i)$ mit $i = 1, \dots, n$
 $q_A = \text{GOTO}(q, A)$

(1) Betrachte als erstes die Produktionsregel $S \rightarrow aA$

Dann handelt es sich also um die folgenden Zustände:

$$q_1' = \text{GOTO}(q_1, a) = \{ [S \rightarrow a.A], [A \rightarrow .cAd], [A \rightarrow .\epsilon] \} (=q_3)$$

$$q_2' = \text{GOTO}(q_1, aA) = \{ [S \rightarrow aA.] \} (=q_5)$$

(2) Betrachte als zweites die Produktionsregel $S \rightarrow bB$

Dann handelt es sich also analog um die folgenden Zustände:

$$q_1' = \text{GOTO}(q_1, b) = \{ [S \rightarrow b.B], [B \rightarrow .cBd], [B \rightarrow .\epsilon] \} (=q_4)$$

$$q_2' = \text{GOTO}(q_1, bB) = \{ [S \rightarrow bB.] \} (=q_6)$$

(3) $q_S = \text{GOTO}(q_1, S) = \{ [S' \rightarrow \$S.\$] \} (=q_2)$

4. **for** alle Items $[A \rightarrow X_1 \dots X_n ., y]$ in q_n **do**
 generiere die reduce-action
 $q_0 q_1 q_2 \dots q_n / y \rightarrow q q_A / y$
end;

Da es sich um einen LR(0)-Parser handelt, gilt: $y = \epsilon$, d.h. man verwendet
 keine Vorschau. Man generiert also die folgenden reduce-actions :

$$r_1 = q_1 q_3 q_5 | \rightarrow q_1 q_2 | \text{ mit } \tau(r_1) = S \rightarrow aA$$

$$r_2 = q_1 q_4 q_6 | \rightarrow q_1 q_2 | \text{ mit } \tau(r_2) = S \rightarrow bB$$

5. **for** alle Items $[A \rightarrow a.ab, z]$ in q mit $a \in T$ **do begin**

$$q_a = \text{GOTO}(q, a);$$

for alle Strings $y \in \text{FIRST}_{\max\{k-1, 0\}}(bz)$ **do**

generiere die shift-action

$$q / ay \rightarrow q q_a / y$$

end;

end;

Da es sich um einen LR(0)-Parser handelt, gilt auch hier wieder: $y = \varepsilon$, d.h. man verwendet keine Vorschau. Man betrachtet also die folgenden Zustände :

$$\begin{aligned} q_a &= \text{GOTO}(q_1, a) = q_3 \\ q_b &= \text{GOTO}(q_1, b) = q_4 \end{aligned}$$

und erhält daraus die shift-actions:

$$\begin{aligned} s_1 &= q_1 \mid a \rightarrow q_1q_3 \mid \\ \text{und} \\ s_2 &= q_1 \mid b \rightarrow q_1q_4 \mid \end{aligned}$$

Im Beispiel haben wir nur den Zustand q_1 betrachtet. Führt man den Algorithmus für alle Zustände aus, so erhält man für den LR(0)-Parser der Grammatik G_{abe} die folgenden **reduce-actions**:

$$\begin{array}{ll} r_1 = q_1q_3q_5 \mid \rightarrow q_1q_2 \mid & \tau(r_1) = S \rightarrow aA \\ r_2 = q_1q_4q_6 \mid \rightarrow q_1q_2 \mid & \tau(r_2) = S \rightarrow bB \\ r_3 = q_3 \mid \rightarrow q_3q_5 \mid & \tau(r_3) = A \rightarrow \varepsilon \\ r_4 = q_3q_7q_9q_{11} \mid \rightarrow q_3q_5 \mid & \tau(r_4) = A \rightarrow cAd \\ r_5 = q_4 \mid \rightarrow q_4q_6 \mid & \tau(r_5) = B \rightarrow \varepsilon \\ r_6 = q_4q_8q_{10}q_{12} \mid \rightarrow q_4q_6 \mid & \tau(r_6) = B \rightarrow cBd \\ r_7 = q_7 \mid \rightarrow q_7q_9 \mid & \tau(r_7) = A \rightarrow \varepsilon \\ r_8 = q_7q_7q_9q_{11} \mid \rightarrow q_7q_9 \mid & \tau(r_8) = A \rightarrow cAd \\ r_9 = q_8 \mid \rightarrow q_8q_{10} \mid & \tau(r_9) = B \rightarrow \varepsilon \\ r_{10} = q_8q_8q_{10}q_{12} \mid \rightarrow q_8q_{10} \mid & \tau(r_{10}) = B \rightarrow cBd \end{array}$$

Außerdem erhält man die folgenden **shift-actions**:

$$\begin{array}{ll} s_1 = q_1 \mid a \rightarrow q_1q_3 \mid , & s_5 = q_7 \mid c \rightarrow q_7q_7 \mid , \\ s_2 = q_1 \mid b \rightarrow q_1q_4 \mid , & s_6 = q_8 \mid c \rightarrow q_8q_8 \mid , \\ s_3 = q_3 \mid c \rightarrow q_3q_7 \mid , & s_7 = q_9 \mid d \rightarrow q_9q_{11} \mid , \\ s_4 = q_4 \mid c \rightarrow q_4q_8 \mid , & s_8 = q_{10} \mid d \rightarrow q_{10}q_{12} \mid . \end{array}$$

Der **LR(0)-Parser für G_{abe} ist nicht deterministisch**, denn man erkennt einen “shift-reduce conflict” bei den Zuständen q_3 , q_4 , q_7 und q_8 . Der Konflikt besteht zwischen einer reduce-action der Produktionsregel $A \rightarrow \varepsilon$ und einer shift-action des Terminals c . Beim Zustand q_3 beispielsweise entsteht der Konflikt bei den Aktionspaaren

$$r_3 = q_3 \mid \rightarrow q_3q_5 \mid \quad \text{und} \quad s_3 = q_3 \mid c \rightarrow q_3q_7 \mid .$$

Beide Aktionen sind auf eine Konfiguration der Form $\$ \phi q_3 \mid cy \$$ anwendbar (ϕ ist dabei eine Folge von Zuständen). Entwickelt man über den Algorithmus die reduce- und shift-actions des LR(1)-Parsers für G_{abe} analog, so erkennt man: **der LR(1)-Parser für G_{abe} ist deterministisch**.

Entwickelt man zusätzlich über den Algorithmus die reduce- und shift-actions des LR(2)-Parsers für $G_{ab\epsilon}$, und vergleicht das Verhalten des LR(1)- und LR(2)-Parsers, so stellt man fest: Wird ein korrekter String eingegeben, d.h. ein String aus $L(G_{ab\epsilon})$, dann verhalten sich beide Parser gleich. Wird jedoch ein inkorrekt String eingegeben, d.h. ein String, der in $L(G_{ab\epsilon})$ nicht vorkommt, dann findet der LR(2)-Parser den Fehler früher als der LR(1)-Parser, denn der LR(2) Parser verwendet eine längere Vorausschau. Je länger also die Vorausschau ist, desto schneller findet der Parser einen Fehler in der Eingabe.

Es bleibt immer noch die Frage offen, ob der entwickelte LR(k)-Parser korrekt arbeitet. Wir werden sehen, daß es sich wirklich um einen Rechtsparser für die Grammatik handelt :

Satz 3.1:

Der kanonische LR(k)-Parser (M, τ) für eine Grammatik G ist ein Rechtsparser. Außerdem gilt für jeden Satz w in $L(G)$: M produziert alle Rechtsparse von w in G . Also ist der kanonische LR(k)-Parser korrekt.

4. LR(k)-Grammatiken

Als erstes wollen wir definieren, was man unter einer LR(k)-Grammatik versteht. Um zu verhindern, daß auch nichteindeutige Grammatiken als LR(k)-Grammatiken bezeichnet werden, legen wir fest, daß sich das Startsymbol nicht selbst ableiten darf.. Da wir zusätzlich fordern, daß der kanonische LR(k)-Parser der Grammatik deterministisch sein soll, kann es ohne die erste Forderung zu folgendem Problem kommen: Betrachte beispielsweise die Grammatik $S \rightarrow a \mid S$. Diese Grammatik hat zwar einen deterministischen LR(0)-Parser, ist aber nicht eindeutig. Daher definieren wir:

Definition : (LR(k)-Grammatik)

Eine **Grammatik** $G = (V, T, P, S)$ heißt **LR(k)**, wenn ihr kanonischer LR(k)-Parser deterministisch ist, und wenn zusätzlich $S \xRightarrow{+} S$ in G unmöglich ist. Eine **Sprache** über dem Alphabet T heißt **LR(k)**, wenn sie durch eine LR(k)-Grammatik über dem Terminalalphabet T erzeugt wird.

Satz 4.1:

Jede LR(k)-Grammatik ist eindeutig.

Wir wollen nun eine grammatischen Charakterisierung für LR(k)-Grammatiken ableiten. Zum Abschluß werden wir dann den Zusammenhang zwischen dem Nichtdeterminismus eines kanonischen LR(k)-Parsers und dem Auftreten von „reduce-reduce Konflikten“ und „shift-reduce Konflikten“ klären.

Definition : (Konflikt aufzeigen)

Sei $G = (V, T, P, S)$ eine Grammatik, G' ihre $\$$ -erweiterte Grammatik und $k \in \mathbb{N}$. Man sagt, daß die k-Items $[A_1 \rightarrow \omega_1., y_1]$ und $[A_2 \rightarrow \omega_2., y_2]$ von G' einen **reduce-reduce Konflikt** aufzeigen, wenn gilt: $A_1 \rightarrow \omega_1$ und $A_2 \rightarrow \omega_2$ sind verschiedene Produktionsregeln und zusätzlich $y_1 = y_2$. Analog zeigen zwei k-Items $[A \rightarrow \alpha.a\beta, z]$ und $[B \rightarrow \omega., y]$ von G' einen **shift-reduce Konflikt** auf, wenn gilt: a ist ein Terminal in T und $y \in \text{FIRST}_k(a\beta z)$.

Beispiel:

Die im 3. Kapitel vorgestellte Grammatik $G_{ab\epsilon}$ ist keine LR(0)-Grammatik, denn die LR(0)-Maschine hat Zustände, die ein Itempaar enthalten, das einen shift-reduce Konflikt aufzeigt. (Betrachte das Itempaar $[A \rightarrow .], [A \rightarrow .cAd]$ in den Zuständen q_3 und q_7 und das Itempaar $[B \rightarrow .], [B \rightarrow .cBd]$ in den Zuständen q_4 und q_8 .)

Satz 4.2: (Charakterisierung von LR(k)-Grammatiken)

Die folgenden Aussagen sind für alle Grammatiken $G = (V, T, P, S)$ und $k \in \mathbb{N}$ äquivalent:

- (a) Der kanonische LR(K)-Parser von G ist deterministisch.
- (b) In der kanonische LR(k)-Maschine der $\$$ -erweiterten Grammatik G' gibt es keinen Zustand, der ein Itempaar enthält, das einen reduce-reduce oder einen shift-reduce Konflikt aufzeigt.
- (c) Aus den Bedingungen

$$S \xRightarrow{\text{rm}}^* \delta_1 A_1 y_1 \xRightarrow{\text{rm}} \delta_1 \omega_1 y_1 = \gamma y_1 \quad \text{in } G ,$$

$$S \xRightarrow{\text{rm}}^* \delta_2 A_2 y_2 \xRightarrow{\text{rm}} \delta_2 \omega_2 y_2 = \gamma y_2 \quad \text{in } G ,$$

$$\text{und } k:y_1 = k:vy_2$$

folgt, daß: $\delta_1 = \delta_2$, $A_1 = A_2$ und $\omega_1 = \omega_2$.

Beweisidee zu Satz 4.2:

(a) \hat{U} (b)

Man zeigt : Der Parser ist nichtdeterministisch

\Leftrightarrow LR(k)-Maschine enthält ein Itempaar mit reduce-reduce oder shift-reduce Konflikt.

(\Leftarrow)

Sei γ ein zuverlässiges Präfix von G' und $\text{VALID}_k(\gamma)$ enthalte ein Itempaar I_1, I_2 , das einen reduce-reduce oder shift-reduce Konflikt aufzeigt.

Dann zeigt man in beiden Fällen über die jeweilige Definition, daß es Aktionspaare gibt, die beide auf die gleiche Konfiguration anwendbar sind.

\Rightarrow Der Parser ist nicht deterministisch.

(\Rightarrow)

Sei also der Parser nicht deterministisch. Dann gibt es zwei verschiedene Aktionen r_1, r_2 , die beide auf dieselbe Konfiguration anwendbar sind. Man unterscheidet drei Fälle:

- (1) r_1 und r_2 sind beide reduce-actions.
- (2) r_1 ist eine shift-action und r_2 eine reduce-action.
- (3) r_1 und r_2 sind beides shift-actions.

Zuerst zeigt man, daß Fall (3) nicht auftreten kann.

Dann zeigt man über die Definition von reduce-reduce oder shift-reduce Konflikt, daß ein Zustand der LR(k)-Maschine

- (1) einen reduce-reduce Konflikt aufzeigt.
- (2) einen shift-reduce Konflikt aufzeigt.

(b) \hat{U} (c)

Man folgert aus der Definition von reduce-reduce Konflikt und aus der Definition der LR(k)-Gültigkeit direkt die Behauptung für den Fall, daß kein Itempaar einen reduce-reduce Konflikt aufzeigt.

Für den Fall, daß ein Itempaar einen shift-reduce Konflikt aufzeigt, zeigt man über die Definition, daß (c) nicht gilt.

Umgekehrt nimmt man an, daß (c) nicht gilt, und zeigt damit, daß ein Itempaar einen shift-reduce Konflikt aufzeigen muß.

Satz 4.3:

Für alle $k \geq 0$ ist die Klasse der LR(k)-Grammatiken in der Klasse der LR(k+1)-Grammatiken enthalten.

Nach Definition ist jede LR(k)-Sprache eine deterministische Sprache, d.h. sie wird von einem deterministischen Kellerautomaten akzeptiert. Es gilt auch die Umkehrung: Jede deterministische Sprache kann von einer LR(k) Grammatik erzeugt werden:

Satz 4.4:

Jeder Kellerautomat M mit Eingabealphabet T kann in eine äquivalente Grammatik G mit Terminalalphabet T transformiert werden, so daß M genau dann deterministisch ist, wenn G LR(k) ist für $k \geq 0$.

Als nächstes werden wir sehen, daß der kanonische LR(k)-Parser einer LR(k)-Grammatik einen Fehler im Eingabestring erkennt. Gibt man also einen String w ein, der nicht in der von der Grammatik erzeugten Sprache ist, dann führt der Parser über w eine Rechnung aus, die mit einer Fehlerkonfiguration endet. Dazu muß sichergestellt werden, daß der kanonische LR(k)-Parser **jeder** Grammatik diese Eigenschaft besitzt. Zusätzlich muß der Parser einer LR(k)-Grammatik bei einem Eingabestring terminieren, d.h. er darf nicht für immer in einer Schleife festhängen. Diese beiden Forderungen halten wir in den folgenden beiden Sätzen fest:

Satz 4.5:

Sei $G = (V, T, P, S)$ eine Grammatik, $k \geq 1$, und M ihr kanonischer LR(k)-Parser. Dann findet M den Fehler in jedem Eingabestring in $T^* \setminus L(G)$.

Satz 4.5 gilt nicht für $k=0$.

Beispiel:

Betrachten wir den LR(0)-Parser der nicht eindeutigen Grammatik $S \rightarrow S \mid a$.
Der LR(0)-Parser dieser Grammatik führt die folgenden Aktionen aus:

$$\begin{aligned} [\$]_0 \mid a &\rightarrow [\$]_0[\$a]_0 \mid, \\ [\$]_0[\$a]_0 \mid &\rightarrow [\$]_0[\$S]_0 \mid, \\ [\$]_0[\$S]_0 \mid &\rightarrow [\$]_0[\$S]_0 \mid. \end{aligned}$$

Obwohl der Parser deterministisch ist, findet er den Fehler in dem falschen Eingabestring aa nicht. Der Parser hängt bei aa für immer in einer Schleife fest:

$$\begin{aligned} [\$]_0 \mid aa\$ &\implies [\$]_0[\$a]_0 \mid a\$ \implies [\$]_0[\$S]_0 \mid a\$ \\ &\implies^n [\$]_0[\$S]_0 \mid a\$ \quad \text{für alle } n \geq 0. \end{aligned}$$

Im Fall $k \geq 1$ garantiert Satz 4.5, daß es bei Eingabe eines falschen Strings eine Berechnung gibt, die mit einer Fehlerkonfiguration endet. Das schließt aber nicht aus, daß der Parser möglicherweise in einer Schleife hängen bleibt. Dazu benötigen wir abschließend noch den folgenden Satz:

Satz 4.6:

Sei $G = (V, T, P, S)$ eine Grammatik, $k \geq 0$, und M ihr kanonischer LR(k)-Parser. Dann hängt M bei der Eingabe eines Strings nicht in einer Schleife fest.