

Theoretische Informatik 3. Semester

Prof. Dr. Martin Plümicke

19. November 2024

Inhaltsverzeichnis

1	Berechenbarkeit und rekursive Funktionen	4
1.1	Primitive rekursive Funktionen	4
1.2	LOOP-Programme	8
1.3	μ -Rekursive Funktionen	13
1.4	WHILE-Programme	15
1.5	Einführung Turingmaschine	17
1.6	Turingmaschine als Automat zur Berechnung von rekursiven Funktionen .	19
2	Grundlagen der Theorie der formalen Sprache	25
3	Reguläre Sprachen	30
3.1	Reguläre Ausdrücke	30
3.2	Endliche Automaten	31
3.3	Von regulären Sprachen zu Scannern	40
3.4	Der Scannergenerator JLex	46
3.4.1	Die JLex - Spezifikation	46
3.4.2	JLex-Direktiven (Anweisungen)	46
3.4.3	Java-Klasse erzeugen	47
3.4.4	Beispiel: html-Lexeme	47
4	Kontextfreie Sprachen	50
4.1	Cocke-Younger-Kasami-Algorithmus	51
4.2	Push-Down-Automaten	55
4.3	Parsertypen	58
4.4	LR-Syntaxanalyse	63

4 Kontextfreie Sprachen

Nachdem wir nun im letzten Kapitel die Funktionsweise von Scanner betrachtet haben, wenden wir uns nun den Parsern zu. Parser lesen Texte und entscheiden, ob der Text in der Sprache einer vorgegebenen Grammatik liegt. Die Grammatik wird über einem Alphabet von Tokens definiert. Dabei werden die Tokens durch die Lex-Spezifikation eines zugehörigen Scanner festgelegt. Der Parser seinerseits ruft den Scanner auf um das nächste Lexem von der Eingabe zu lesen.

Verkürzt zusammengefasst kann man sagen:

Scanner: Zerlegt den Eingabetext in einzelne Tokens

Parser: Soll erkennen, ob die Anordnung der Tokens in einer „richtigen Reihenfolge“ ist.

WICHTIG: Terminal-Symbole können jetzt auch Strings sein, nicht wie bisher nur einzelne Symbole.

Beispiel (Super-Mini-Java). Sei die Grammatik $G = (N, T, \Pi, S)$ gegeben mit

$T = \{class, \{, \}, ;, while, (,), true\}$

$N = \{S, Classdecl, Classbodydecl\}$

$\Pi = \{$

$S \quad \rightarrow Classdecl$

$Classdecl \quad \rightarrow class Classbodydecl$

$Classbodydecl \rightarrow \{\} \mid \{while(true)\{;\}\}$

$\}$

4.1 Cocke-Younger-Kasami-Algorithmus

Definition 4.1 (Chomsky-Normalform). Eine kontextfreie Grammatik $G = (N, \Sigma, \Pi, S)$ ist in Chomsky-Normalform, wenn alle Produktionen aus Π entweder die Form $A \rightarrow BC$ mit $A, B, C \in N$ oder $A \rightarrow a$ mit $A \in N$ und $a \in \Sigma$ haben (Ausnahme: $S \rightarrow \epsilon \in \Pi$ erlaubt)

Satz 4.2. Für jede kontextfreie Grammatik $G = (N, \Sigma, \Pi, S)$ existiert eine äquivalente kontextfreie Grammatik $G' = (N', \Sigma, \Pi', S)$ in Chomsky-Normalform mit $\mathcal{L}(G) = \mathcal{L}(G')$, falls $\mathcal{L}(G) \neq \emptyset$.

Beweis.

1. O.B.d.A. können wir davon ausgehen, dass alle Produktionen nach ausschließlich Terminal-Symbolen ableitbar sind. (Alle anderen Ableitungen können entfernt werden.)
2. $G' := G$
3. Alle Produktionen in Π' , die bereits in Chomsky-Normalform sind, bleiben bestehen. In allen anderen Produktionen werden alle Terminal-Symbole $a \in \Sigma$ durch neue Nichtterminal-Symbole A besetzt, wobei $A \notin \Sigma$ und A in N' bzw. $(A \rightarrow a)$ in Π' ergänzt wird.
4. Für eine Folge von Produktionen $A_1 \rightarrow A_2, A_2 \rightarrow A_3, \dots, A_{n-1} \rightarrow A_n$ mit $A_i \in N'$ muss eine Regel $A_n \rightarrow B_1 \dots B_m$ oder $A_n \rightarrow a$ mit $B_i \in N'$ und $a \in \Sigma$ existieren.

\Rightarrow Jede Folge $A_1 \rightarrow A_2, A_2 \rightarrow A_3, \dots, A_{n-1} \rightarrow A_n$ wird durch $A_1 \rightarrow B_1 \dots B_m$ bzw. $A_1 \rightarrow a$ ersetzt.
5. Jede Produktion $A \rightarrow B_1 \dots B_m$ mit $m > 2$ wird ersetzt durch,

$$A \rightarrow B_1 C_1$$

$$C_i \rightarrow B_{i+1} C_{i+1} \text{ für } i \in \{1, \dots, m-3\}$$

$$C_{m-2} \rightarrow B_{m-1} B_m$$

wobei $C_1, \dots, C_{m-2} \in N'$ neue Nichtterminal-Symbole sind.

□

Beispiel: Sie folgende Grammatik $G = (\{S, C\}, \{a, b\}, \Pi, S)$ mit

$$\Pi = \{S \rightarrow C \mid \epsilon, \\ C \rightarrow aCb \mid ab\}$$

gegeben.

Schritt 3:

$C \rightarrow aCb \mid ab$ wird ersetzt durch $C \rightarrow ACB \mid AB$.

$A \rightarrow a, B \rightarrow b$ wird hinzugefügt.

Schritt 4:

$S \rightarrow C$ wird ersetzt durch $S \rightarrow ACB \mid AB$.

Schritt 5:

$S \rightarrow ACB$ wird ersetzt durch $S \rightarrow AD, D \rightarrow CB$.

$C \rightarrow ACB$ wird ersetzt durch $C \rightarrow AE, E \rightarrow CB$.

Die äquivalente Grammatik in Chomsky-Normalform lautet:

$$G' = (\{S, A, B, C, D, E\}, \{a, b\}, \Pi', S)$$

mit

$$\Pi' = \{S \rightarrow AD \mid AB \mid \epsilon \\ D \rightarrow CB \\ C \rightarrow AE \mid AB \\ E \rightarrow CB \\ A \rightarrow a \\ B \rightarrow b\}$$

Algorithmus 4.3 (Cocke-Younger-Kasami-Algorithmus). Sei eine kontextfreie Grammatik $G = (N, \Sigma, \Pi, S)$ in Chomsky-Normalform gegeben.

Eingabe: $w = a_1 \dots a_n \in \Sigma^*$

Ausgabe: $erg \in \{True, False\}$

Nachbedingung: $erg = True$, falls $w \in \mathcal{L}(G)$, sonst $erg = False$

```

n = |w|
for (i = 1 to n)
    Nii = {A | (A → a) ∈ Π mit a = w[i]}
for (d = 1 to n - 1)
    for (i = 1 to n - d)
        j = i + d
        for (k = i to j - 1)
            Nij = Nij ∪ {A | (A → BC) ∈ Π mit B ∈ Nik und C ∈ N(k+1)j}
if (S ∈ N1n) // dann ist w aus S ableitbar
    erg = True
else
    erg = False

```

Beispiel (CYK-Algorithmus). Grammatik $G = (N, T, \Pi, S)$ mit

```

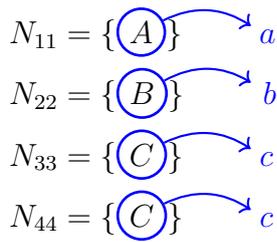
T = {a, b, c}
N = {S, A, B, C}
Π = {
    S → AB
    A → AA
    B → BC
    B → BB
    C → CC
    A → a
    B → b
    C → c
}

```

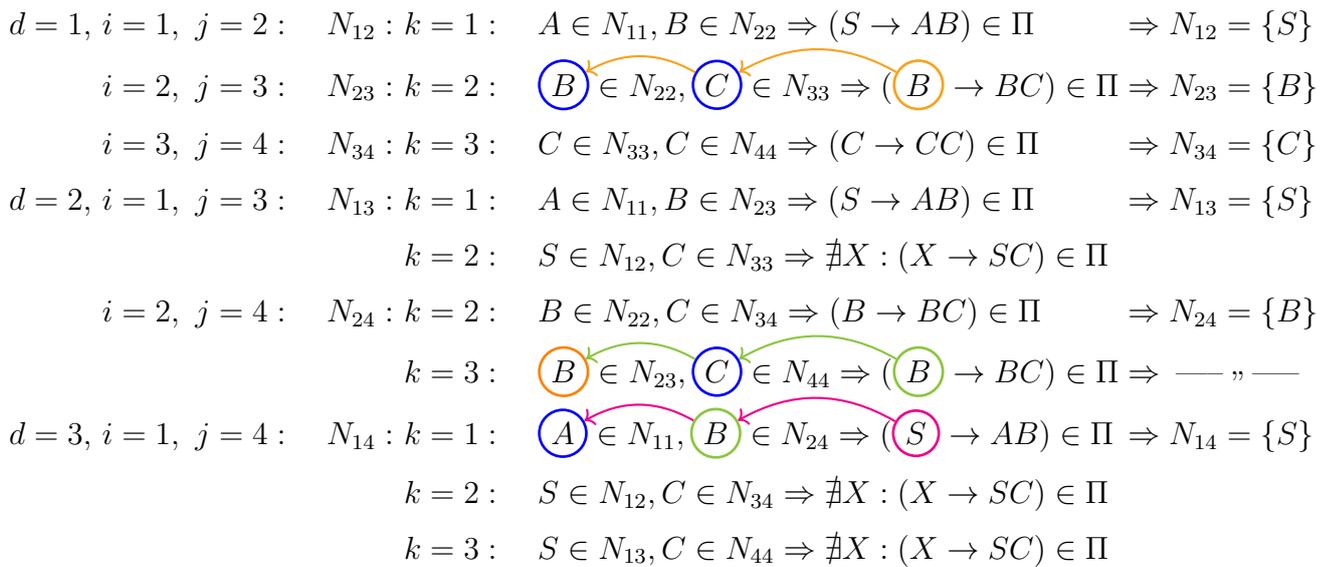
$w = "abcc"$

$n = |w| = 4$

1. for-Schleife:



2. for-Schleife:



Ergebnis:

$$S \in N_{14} \Rightarrow \text{erg} = \text{True}$$

Überprüfung:

Die Ableitung lautet: $\textcircled{S} \rightarrow \textcircled{A}\textcircled{B} \rightarrow a\textcircled{B}\textcircled{C} \rightarrow a\textcircled{B}\textcircled{C}c \rightarrow abcc$

Komplexität von CYK

Es handelt sich um zwei ineinander geschachtelte for-Schleifen. Zudem muss noch bei der Mengenbildung von N_{ij} über $1 \leq k < j$ gelaufen werden.

$$\Rightarrow O(n^3)$$

Dies ist keine ideale Laufzeit. Für eine Unterklasse/Teilmenge der kontextfreien Sprachen gibt es jedoch bessere Algorithmen, die eine Laufzeitkomplexität von $O(n)$ besitzen.

4.2 Push-Down-Automaten

Definition 4.4 (Push-Down-Automat (PDA)). Der Automat $A = (\Sigma, \Gamma, Q, \delta, \gamma_0, q_0, F)$ mit

1. Σ - nicht leeres, endliches Eingabealphabet
2. Γ - nicht leeres, endliches Kelleralphabet
3. Q - nicht leere, endliche Zustandsmenge
4. δ - Übergangsfunktion: $Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \rightarrow \mathcal{P}(Q \times \Gamma^*)^1$
5. $\gamma_0 \in \Gamma$ - Anfangssymbol auf dem Keller/Stack
6. $q_0 \in Q$ - Anfangszustand
7. $F \subseteq Q$ - Menge der Finalzustände

heißt Push-Down-Automat (oder Kellerautomat).

Definition 4.5 (Sprache von einem PDA akzeptiert). Die Menge $L \subseteq \Sigma^*$ heißt von einem PDA $A = (\Sigma, \Gamma, Q, \delta, \gamma_0, q_0, F)$ akzeptiert, wenn gilt:

$$L = \{p \mid p \in \Sigma^* \wedge (\exists q \exists w : q \in F \wedge w \in \Gamma^* \wedge (q_0, p, \gamma_0) \xrightarrow{*} (q, \varepsilon, w))\},$$

wobei $(q, aw, sb) \rightarrow (q', w, sc)$ gilt, wenn $(q', c) \in \delta(q, a, b)$. (Interpretation: Vom Zustand q ausgehend kann ein a nur dann gelesen werden, wenn das oberste Element des Stapels (Stapelspitze) ein b ist. In diesem Fall wird q' zum aktuellen Zustand und die Stapelspitze wird durch ein c ersetzt.)

Gilt für jedes Tupel $(q, x, \gamma) \in (Q \times \Sigma \times \Gamma)$ und jedes $w \in \Gamma$: $|\delta(q, x, \gamma)| + |\delta(q, \varepsilon, w)| \leq 1$, so arbeitet der PDA deterministisch (DPDA), sonst nicht-deterministisch (NPDA). In ersterem Fall ist die Übergangsfunktion gegeben durch:

$$\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \rightarrow (Q \times \Gamma^*).$$

¹Man kann δ auch als Relation betrachten $Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \times (Q \times \Gamma^*)$.

Beispiel (DPDA). $A = (\{a, b\}, \{a, \gamma_0\}, \{q_0, q_1, q_2\}, \delta, \gamma_0, q_0, \{q_2\})$ mit

$$\delta = \{$$

$$(q_0, a, \gamma_0) \mapsto \{(q_0, \gamma_0 a)\} \quad (1)$$

$$(q_0, a, a) \mapsto \{(q_0, aa)\} \quad (2)$$

$$(q_0, b, a) \mapsto \{(q_1, \varepsilon)\} \quad (3)$$

$$(q_1, b, a) \mapsto \{(q_1, \varepsilon)\} \quad (4)$$

$$(q_1, \varepsilon, \gamma_0) \mapsto \{(q_2, \varepsilon)\} \quad (5)$$

$$(q, x, \gamma) \mapsto \emptyset, \text{sonst}$$

$$\}$$

A akzeptiert die Sprache $a^n b^n$:

$$(q_0, a^n b^n, \gamma_0) \xrightarrow{(1)} (q_0, a^{n-1} b^n, \gamma_0 a) \xrightarrow{(2)} (q_0, a^{n-2} b^n, \gamma_0 aa) \xrightarrow{(2)} \dots \xrightarrow{(2)} (q_0, b^n, \gamma_0 a^n) \xrightarrow{(3)}$$

$$(q_1, b^{n-1}, \gamma_0 a^{n-1}) \xrightarrow{(4)} \dots \xrightarrow{(4)} (q_1, \varepsilon, \gamma_0) \xrightarrow{(5)} (q_2, \varepsilon, \varepsilon)$$

Die Sprache $a^n b^n$ ist nicht regulär, d.h. DPDAs akzeptieren auch nicht-reguläre Sprachen.

Beispiel (Palindrom-Sprache). $\mathcal{L}_{pal} = \{w\bar{w} \mid w \in \{0, 1\}^* \wedge \bar{w} \text{ ist die Umkehrung von } w\}$

Der NPDA $A = (\{0, 1\}, \{\gamma_0, \gamma^0, \gamma^1\}, \{q_0, q_1, q_E\}, \delta, \gamma_0, q_0, \{q_E\})$ mit

$$\delta = \{$$

$$(q_0, 0, \gamma_0) \mapsto \{(q_0, \gamma_0 \gamma^0)\} \quad (1)$$

$$(q_0, 1, \gamma_0) \mapsto \{(q_0, \gamma_0 \gamma^1)\} \quad (2)$$

$$(q_0, 0, \gamma^0) \mapsto \{(q_0, \gamma^0 \gamma^0), (q_1, \varepsilon)\} \quad (3)$$

$$(q_0, 0, \gamma^1) \mapsto \{(q_0, \gamma^1 \gamma^0)\} \quad (4)$$

$$(q_0, 1, \gamma^0) \mapsto \{(q_0, \gamma^0 \gamma^1)\} \quad (5)$$

$$(q_0, 1, \gamma^1) \mapsto \{(q_0, \gamma^1 \gamma^1), (q_1, \varepsilon)\} \quad (6)$$

$$(q_1, 0, \gamma^0) \mapsto \{(q_1, \varepsilon)\} \quad (7)$$

$$(q_1, 1, \gamma^1) \mapsto \{(q_1, \varepsilon)\} \quad (8)$$

$$(q_0, \varepsilon, \gamma_0) \mapsto \{(q_E, \varepsilon)\} \quad (9)$$

$$(q_1, \varepsilon, \gamma_0) \mapsto \{(q_E, \varepsilon)\} \quad (10)$$

$$\}$$

Idee des Automaten: Gelesene 0er werden mit γ^0 und gelesene 1er mit γ^1 gemerkt. Wenn zwei gleiche Zeichen hintereinander gelesen werden ((3) und (6)), wird nichtdeterministisch geraten, ob w fortgesetzt wird oder ob es sich bereits um das erste Zeichen von \bar{w} handelt. q_0 ist dabei für die linke Hälfte und q_1 für die rechte Hälfte des Wortes zuständig.

Bemerkung:

- \mathcal{L}_{pal} ist eine kontextfreie Sprache
- es gibt keinen DPDA, der \mathcal{L}_{pal} akzeptiert (*ohne Beweis*)

Satz 4.6. Die Menge der Sprachen, die durch einen NPDA akzeptiert werden, ist eine echte Obermenge der Sprachen, die durch einen DPDA akzeptiert werden.

Beweis: siehe z.B. Winter, Theoretische Informatik

Im Folgenden wollen wir den Zusammenhang zwischen kontextfreien Sprachen und PDAs betrachten.

Zunächst soll aus einer kontextfreien Grammatik ein PDA konstruiert werden.

Idee des Automaten:

- (1) S wird ohne ein Zeichen zu lesen auf den Stack geschrieben
- (2) Ist auf dem Stack ein Nichtterminalsymbol, wird dieses durch seine Ableitung (rückwärts) ersetzt. Dabei wird kein Zeichen gelesen.
- (3) Sind die gleichen Terminalsymbole auf Eingabe und Stack, werden beide gelöscht (= akzeptiert)
- (4) Wenn alle Zeichen gelesen sind und γ_1 auf dem Stack liegt, ist das Wort akzeptiert
- (5) Alle anderen Eingaben werden auf die leere Menge abgebildet, d.h. das Wort wird nicht akzeptiert

Satz 4.7 (Chomsky, Evey). Eine Sprache ist genau dann kontextfrei, wenn sie von einem (N)PDA akzeptiert wird.

Beweis. Sei $G = (N, \Sigma, \Pi, S)$ eine kontextfreie Grammatik und $L = \mathcal{L}(G)$ die zugehörige Sprache.

Zu zeigen: Es gibt einen PDA, der L akzeptiert.

Herangehensweise: Wir konstruieren einen PDA, der L akzeptiert:

$A = (\Sigma, \Sigma \dot{\cup} N \dot{\cup} \{\gamma_0, \gamma_1\}, \{q_0, q_E\}, \delta, \gamma_0, q_0, \{q_E\})$ mit

$\delta = \{$

$$(q_0, \varepsilon, \gamma_0) \mapsto \{(q_0, \gamma_1 S)\} \quad (1)$$

$$(q_0, \varepsilon, A) \mapsto \{(q_0, \bar{p} \mid (A \rightarrow p) \in \Pi)\} \quad \forall A \in N \quad (2)$$

$$(q_0, x, x) \mapsto \{(q_0, \varepsilon)\} \quad \forall x \in \Sigma \quad (3)$$

$$(q_0, \varepsilon, \gamma_1) \mapsto \{(q_E, \varepsilon)\} \quad (4)$$

$$(q, x, \gamma) \mapsto \emptyset, \text{ sonst} \quad (5)$$

$\}$

Für die Rückrichtung des Beweises wird z.B. auf Winter, Theoretische Informatik verwiesen. □

Im Folgenden betrachten wir nur noch die Klasse der kontextfreien Sprachen, die von DPDAs akzeptiert werden.

Wir werden sehen, dass es einen Algorithmus gibt, der diese Sprachen in linearer Zeit akzeptiert.

4.3 Parsertypen

Eingabe: Grammatik $G = (N, \Sigma, \Pi, S)$

$w \in \Sigma^*$

Ausgabe: $erg \in \{True, False\}$

Nachbedingung: $erg = (w \in \mathcal{L}(G))$

Mit anderen Worten: Es muss eine Ableitung $S \xrightarrow{*} w$ gefunden werden.

Beispiel. $G = (N, T, \Pi, S)$ mit

$$N = \{S, A\}$$

$$T = \{a, b, c\}$$

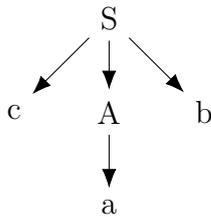
$$\Pi = \{S \rightarrow cAb, A \rightarrow ab|a\}$$

$$w = cab$$

$$\Rightarrow \text{Ableitung} : S \rightarrow cAb \rightarrow cab$$

Ergebnis: $erg = True$

Man kann die Ableitung als Baum darstellen.



Die Blätter bilden das abgeleitete Wort.

Es gibt verschiedene Ansätze Algorithmen zu entwickeln, die der Spezifikation eines Parsers entsprechen.

- Man kann den Eingabestring von **L**inks oder von **R**echts lesen.
- Man kann die Ableitung unterschiedlich bilden. Man kann das am weitesten **L**inks stehende Nichtterminal oder das am weitesten **R**echts stehende Nichtterminal ableiten.

Definition 4.8 (Links- Rechtsableitungen). Sei $w \rightarrow w'$ eine direkte Ableitung. Man bezeichnet $w \rightarrow_l w'$ als *direkte Linksableitung*, wenn das am weitesten links stehende Nichtterminal abgeleitet wird. Man bezeichnet $w \rightarrow_r w'$ als *direkte Rechtsableitung*, wenn das am weitesten rechts stehende Nichtterminal abgeleitet wird. Analog zu Definition 2.4 definiert man *Links-* bzw. *Rechtsableitung*. (Bez.: $w \xrightarrow{*}_l w'$ bzw. $w \xrightarrow{*}_r w'$)

Beispiel. Sei $G = (N, T, \Pi, S)$ eine Grammatik mit

$$N = \{S, A, B, C\}$$

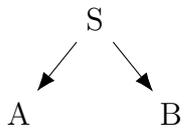
$$T = \{a, b, c\} \text{ und}$$

$$\Pi = \{S \rightarrow AB, A \rightarrow a, B \rightarrow bB, B \rightarrow c\}$$

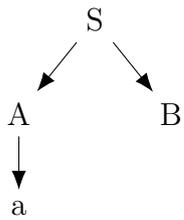
Sei weiterhin $w = abc$ das Eingabewort.

Aufbau des Ableitungsbaums durch Linksableitungen:

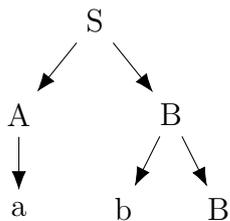
Man liest das nächste Token und wählt dann anhand diesem die Produktion aus.



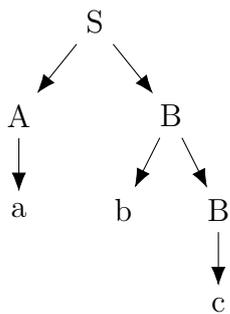
$nexttoken() = a$



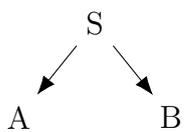
$nexttoken() = b$



$nexttoken() = c$



Aufbau des Ableitungsbaums durch Rechtsableitungen:



$nexttoken() = a$

Problem: Es ist nicht offensichtlich wie B abgeleitet wird, weil nicht klar ist welche Tokens von A abgeleitet werden können.

Anderer Ansatz: *Bottomup*: Wir bauen den Baum beginnend bei den *Blättern* zur *Wurzel* auf.

$nexttoken() = a$

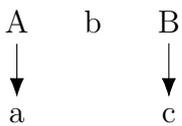
wende an: $A \rightarrow a$



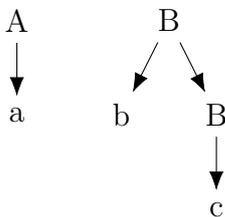
$nexttoken() = b$

$nexttoken() = c$

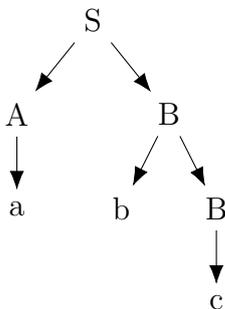
wende an: $B \rightarrow c$



wende an: $B \rightarrow bB$



wende an: $S \rightarrow AB$



Betrachtet man die Konstruktion rückwärts:

$$\underline{S} \rightarrow \underline{A}\underline{B} \rightarrow \underline{A}b\underline{B} \rightarrow \underline{A}bc \rightarrow abc,$$

so erhält man die Rechtsableitung.

Definition 4.9 (Klassifikation der Parser). Parser werden in Klassen eingeteilt, die angeben, ob ein Tokenstring von rechts oder links gelesen werden, ob Rechts- oder Linksableitungen gebildet werden und wieviele Zeichen vom Tokenstring gelesen werden, um die jeweilige Regel auszuwählen.

$LL(n) \hat{=}$ Tokenstring von links lesen, links ableiten, n Zeichen vorausschauen (lookahead)
 $LR(n) \hat{=}$ Tokenstring von links lesen, rechts ableiten, n Zeichen vorausschauen

Beispiel. Sei die Grammatik $G = (N, T, \Pi, S)$ mit

$$N = \{S, A\},$$

$$T = \{a, b\}$$

$$\Pi = \{S \rightarrow A, A \rightarrow aA \mid ab\}.$$

Sei weiter $w = aab$

Linksableitungen: $S \rightarrow A$ nexttoken() = a

Mit 1 Zeichen lookahead, kann man nicht entscheiden ob $A \rightarrow aA$ oder $A \rightarrow ab$ ausgewählt werden muss.

$\Rightarrow G$ liegt nicht in $LL(1)$. G liegt in $LL(2)$.

Wie sieht die Sprache einer $LL(0)$ - Grammatik aus?

Eigenschaften:

- Es gibt keine Alternativen der Ableitung
- Die Sprache umfasst genau ein Wort

Beispiel ($LL(0)$ -Grammatik). $G = (N, T, \Pi, S)$ mit

$$N = \{S, A, B\},$$

$$T = \{a, b\} \text{ und}$$

$$\Pi = \{S \rightarrow AB, A \rightarrow a, B \rightarrow b\}$$

Es ist nur genau ein Wort ableitbar:

$$S \rightarrow AB \rightarrow aB \rightarrow ab$$

$$L(G) = \{ab\}$$

Definition 4.11 (Hülle). Sei I eine Menge von LR(0)–Elementen einer Grammatik G , so ist die

$$\text{Hülle: } \mathcal{P}(\{lr \mid lr \text{ ist ein LR(0)–Element}\}) \rightarrow \mathcal{P}(\{lr \mid lr \text{ ist ein LR(0)–Element}\})$$

die kleinste Menge, die durch folgende Regeln bestimmt ist:

- $I \subseteq \text{Hülle}(I)$
- Wenn $(A \rightarrow \alpha.B\beta) \in \text{Hülle}(I)$ und $(B \rightarrow \gamma) \in \Pi$ ist, so ist $(B \rightarrow .\gamma) \in \text{Hülle}(I)$, wobei $A, B \in N, \alpha, \beta, \gamma \in (N \cup T)^*$

Beispiel. Seien $\Pi = \{S \rightarrow E, E \rightarrow E + E \mid E * E \mid id\}$, $N = \{S, E\}$ und $T = \{+, *, id\}$

Dann gilt:

$$\text{Hülle}(\{S \rightarrow .E\}) = \{S \rightarrow .E, E \rightarrow .E + E, E \rightarrow .E * E, E \rightarrow .id\}$$

$$\text{Hülle}(\{E \rightarrow E + .E\}) = \{E \rightarrow E + .E, E \rightarrow .E + E, E \rightarrow .E * E, E \rightarrow .id\}$$

$$\text{Hülle}(\{E \rightarrow E. + E\}) = \{E \rightarrow E. + E\}$$

Definition 4.12 (Sprung–Funktion). Die Sprungfunktion

$$\text{Sprung} : \mathcal{P}\{lr \mid lr \text{ ist ein LR(0)–Element}\} \times (N \cup T) \rightarrow \mathcal{P}\{lr \mid lr \text{ ist ein LR(0)–Element}\}$$

ist gegeben durch

$$\text{Sprung}(I, x) = \bigcup_{(A \rightarrow \alpha.x\beta) \in I} \text{Hülle}\{A \rightarrow \alpha.x.\beta\}$$

wobei $x \in (N \cup T), \alpha, \beta, \gamma \in (N \cup T)^*$

Beispiel. Seien N, T, Π wie im vorigen Beispiel gegeben.

Dann gilt:

$$\begin{aligned} \text{Sprung}(\{S \rightarrow E., E \rightarrow E. + E\}, +) &= \text{Hülle}(\{E \rightarrow E + .E\}) \\ &= \{E \rightarrow E + .E, E \rightarrow .E + E, E \rightarrow .E * E, E \rightarrow .id\} \end{aligned}$$

Definition 4.13 (Kanonische Sammlung von Mengen LR(0)–Elementen). Sei eine Grammatik $G = (N, T, \Pi, S)$ gegeben. Dann ist $G' = (N \cup \{S'\}, T, \Pi \cup \{S' \rightarrow S\}, S')$ die erweiterte Grammatik. Die Kanonische Sammlung von LR(0)–Elementen C ist die kleinste Menge mit folgenden Bedingungen:

- $\text{Hülle}(\{S' \rightarrow .S\}) \in C$

- Wenn $I \in C$, dann gilt $\forall x \in (N \cup T) \text{Sprung}(I, x) \in C$, falls $\text{Sprung}(I, x) \neq \emptyset$

Beispiel. Sei die Grammatik $G = (N, T, \Pi, S')$ mit

$$\Pi = \{S' \rightarrow S, S \rightarrow (S + S), S \rightarrow z\}$$

$$N = \{S', S\}$$

$$T = \{(\cdot), +, z\}$$

gegeben.

Kanonische Sammlung:

$$I_0 = \text{H\u00fclle}(\{S' \rightarrow \cdot S\}) = \{S' \rightarrow \cdot S, S \rightarrow \cdot (S + S), S \rightarrow \cdot z\}$$

$$I_1 = \text{Sprung}(I_0, S) = \{S' \rightarrow S \cdot\}$$

$$I_2 = \text{Sprung}(I_0, (\cdot)) = \{S \rightarrow (\cdot S + S), S \rightarrow \cdot (S + S), S \rightarrow \cdot z\}$$

$$I_3 = \text{Sprung}(I_0, z) = \{S \rightarrow z \cdot\}$$

$$I_4 = \text{Sprung}(I_2, S) = \{S \rightarrow (S \cdot + S)\}$$

$$\text{Sprung}(I_2, (\cdot)) = I_2$$

$$\text{Sprung}(I_2, z) = I_3$$

$$I_5 = \text{Sprung}(I_4, +) = \{S \rightarrow (S + \cdot S), S \rightarrow \cdot (S + S), S \rightarrow \cdot z\}$$

$$I_6 = \text{Sprung}(I_5, S) = \{S \rightarrow (S + S \cdot)\}$$

$$\text{Sprung}(I_5, (\cdot)) = I_2$$

$$\text{Sprung}(I_5, z) = I_3$$

$$I_7 = \text{Sprung}(I_6, (\cdot)) = \{S \rightarrow (S + S) \cdot\}$$

Algorithmus 4.14 (LR(0)-Parser).

<i>Datentypen:</i>	ADT Stack
<i>Eingabe:</i>	Eingabestring w Sprung-Funktion Kanonische Sammlung
<i>Ausgabe:</i>	Rechsableitung $rl \vee \text{error}$
<i>Nachbedingung:</i>	$S' \xrightarrow{*}_r w = rl \Leftrightarrow \exists \text{ LR(0)-Ableitung}$ \wedge $\text{error} \Leftrightarrow \nexists \text{ LR(0)-Ableitung (vgl. 4.15)}$
<i>Init:</i>	Startzustand I_0 liegt auf dem Stack und $w\$$ im Eingabebuffer ip zeigt auf erstes Zeichen im Eingabebuffer
<pre> while (<i>true</i>) { $I = \text{Stack.top}()$ $a = w[ip]$ if ($\text{Sprung}(I, a) = I'$ && $\nexists (A \rightarrow \alpha.) \in I$) { $\text{Stack.push}(I')$ $ip++$ } else if ($I == \{A \rightarrow \beta.\}$ && $A \neq S'$) { entferne β Elemente vom Stack $I' = \text{Stack.top}()$ $\text{Stack.push}(\text{Sprung}(I', A))$ gib aus: $A \rightarrow \beta$ } else if ($I == \{S' \rightarrow S.\}$) { gib aus: $S' \rightarrow S$ return } else throw error } </pre>	
$\$ = \text{eof}$ (end of file) $ \beta = \text{L\u00e4nge von } \beta$	

Beispiel. Sei die Grammatik $G = (N, T, \Pi, S)$ und die kanonische Sammlung wie im vorigen Beispiel gegeben.

Algorithmusdurchlauf:

Stack	Eingabestring	nächster Schritt
I_0	$(z + z)\$$	$Sprung(I_0, () = I_2$
$I_0 I_2$	$z + z)\$$	$Sprung(I_2, z) = I_3$
$I_0 I_2 I_3$	$+z)\$$	$reduce\ S \rightarrow z, Sprung(I_2, S) = I_4$
$I_0 I_2 I_4$	$+z)\$$	$Sprung(I_4, +) = I_5$
$I_0 I_2 I_4 I_5$	$z)\$$	$Sprung(I_5, z) = I_3$
$I_0 I_2 I_4 I_5 I_3$	$)\$$	$reduce\ S \rightarrow z, Sprung(I_5, S) = I_6$
$I_0 I_2 I_4 I_5 I_6$	$)\$$	$Sprung(I_6,)) = I_7$
$I_0 I_2 I_4 I_5 I_6 I_7$	$\$$	$reduce\ S \rightarrow (S + S), Sprung(I_0, S) = I_1$
$I_0 I_1$	$\$$	<i>akzeptiere</i>

Ausgabe:

$$\left. \begin{array}{l} S \rightarrow z \quad (1) \\ S \rightarrow z \quad (2) \\ S \rightarrow (S + S) \quad (3) \end{array} \right\} \Rightarrow S' \rightarrow S \xrightarrow{(3)} (S + S) \xrightarrow{(2)} (S + z) \xrightarrow{(1)} (z + z)$$

Beispiel. Sei die folgende erweiterte Grammatik $G = (N, T, \Pi, S')$ mit $N = \{S', E, T, F\}$, $T = \{+, *, id\}$ und $\Pi = \{S' \rightarrow E, E \rightarrow E + T \mid T, T \rightarrow T * F \mid F, F \rightarrow id\}$

gegeben. Daraus ergibt sich die Kanonische Sammlung:

$$I_0 = \text{Hülle}(S' \rightarrow .E) = \{S' \rightarrow .E, E \rightarrow .E + T, E \rightarrow .T, T \rightarrow .T * F, T \rightarrow .F, F \rightarrow .id\}$$

$$I_1 = \text{Sprung}(I_0, E) = \text{Hülle}(S' \rightarrow E., E \rightarrow E. + T)$$

$$I_2 = \text{Sprung}(I_0, T) = \text{Hülle}(E \rightarrow T., T \rightarrow T. * F)$$

$$I_3 = \text{Sprung}(I_0, F) = \text{Hülle}(T \rightarrow F.) = \{T \rightarrow F.\}$$

$$I_4 = \text{Sprung}(I_0, id) = \text{Hülle}(F \rightarrow id.) = \{F \rightarrow id.\}$$

$$I_5 = \text{Sprung}(I_1, +) = \text{Hülle}(E \rightarrow E+.T) = \{E \rightarrow E+.T, T \rightarrow .T * F, T \rightarrow .F, F \rightarrow .id\}$$

$$I_6 = \text{Sprung}(I_2, *) = \text{Hülle}(T \rightarrow T * .F) = \{T \rightarrow T * .F, F \rightarrow .id\}$$

$$I_7 = \text{Sprung}(I_5, T) = \text{Hülle}(E \rightarrow E + T.) = \{E \rightarrow E + T., T \rightarrow T. * F\}$$

$$\text{Sprung}(I_5, F) = \text{Hülle}(T \rightarrow F.) = I_3$$

$$\text{Sprung}(I_5, id) = \text{Hülle}(F \rightarrow id.) = I_4$$

$$I_8 = \text{Sprung}(I_6, F) = \text{Hülle}(T \rightarrow T * F.) = \{T \rightarrow T * F.\}$$

$$\text{Sprung}(I_6, id) = \text{Hülle}(F \rightarrow id.) = I_4$$

$$\text{Sprung}(I_7, *) = I_6$$

Es kann passieren, dass eine Menge von LR(0)–Elementen entweder

$$\begin{aligned} A &\rightarrow \alpha.\beta \\ B &\rightarrow \gamma. \end{aligned}$$

oder

$$\begin{aligned} A &\rightarrow \gamma_1. \\ B &\rightarrow \gamma_2. \end{aligned}$$

enthält.

In beiden Fällen ergeben sich Mehrdeutigkeiten bei der LR-Syntaxanalyse. Im ersten Fall ist nicht klar, ob geschiftet oder reduziert werden soll. Im zweiten Fall ist nicht klar, welches LR(0)–Element reduziert werden soll.

Im vorigen Beispiel ist das in I_1 , I_2 und I_7 der Fall.

Definition 4.15 (LR(0)–Eigenschaft). Eine Grammatik G hat die LR(0)–Eigenschaft, wenn es keine Elemente ihrer Kanonischen Sammlung gibt, die entweder Elemente der Form

$$A \rightarrow \alpha.\beta \text{ und } B \rightarrow \gamma.$$

oder Elemente der Form

$$A \rightarrow \gamma_1. \text{ und } B \rightarrow \gamma_2.$$

enthalten, wobei A eine Nichtterminal und α , β , γ_1 und γ_2 Satzformen aus Terminalen und Nichtterminalen sind.

Die Grammatik im vorigen Beispiel erfüllt nicht die LR(0)–Eigenschaft.

Die LR(0)–Eigenschaft bedeutet, dass eine Bottom–Up LR-Syntaxanalyse ohne Vorausschau eindeutig aufgebaut werden kann. Man spricht in dem Fall von einer LR(0)–Ableitung.

Definition 4.16 (First). Sei $G = (N, T, \Pi, S)$ eine Grammatik, dann ist

$$\textit{First} : (N \cup T)^* \rightarrow \mathcal{P}(T)$$

gegeben durch folgende Funktion:

$$\textit{First}(a\beta) = \{a\}, \text{ falls } a \in T, \beta \in (N \cup T)^*$$

$$First(A\beta) = \bigcup_{(A \rightarrow \alpha) \in \Pi} First(\alpha), \text{ falls } A \in N, (A \rightarrow \varepsilon) \notin \Pi \text{ und } \beta, \alpha \in (N \cup T)^*$$

$$First(A\beta) = \left(\bigcup_{\substack{(A \rightarrow \alpha) \in \Pi, \\ \alpha \neq \varepsilon}} First(\alpha) \right) \cup First(\beta), \text{ falls } A \in N, (A \rightarrow \varepsilon) \in \Pi \text{ und } \beta, \alpha \in (N \cup T)^*$$

$First(\gamma)$ ergibt alle Terminalsymbole, die als erstes Zeichen aus γ abgeleitet werden können.

Definition 4.17 (Follow). Sei $G = (N, T, \Pi, S)$ eine Grammatik. Dann ist

$$Follow : N \rightarrow \mathcal{P}(T)$$

definiert durch die folgende Funktion:

$$\begin{aligned} Follow(A) = & \left(\bigcup_{(B \rightarrow \alpha A \beta) \in \Pi} (First(\beta)) \right) \\ & \cup \left(\bigcup_{(B \rightarrow \alpha A) \in \Pi} (Follow(B)) \right) \\ & \cup \left(\bigcup_{(B \rightarrow \alpha A \beta) \in \Pi, \beta \xrightarrow{*} \varepsilon} (Follow(B)) \right), \end{aligned}$$

wobei $A, B \in N$ und $\alpha, \beta \in (N \cup T)^*$

Die Funktion Follow bestimmt die Terminale, die in einem beliebigen Ableitungsschritt auf ein Nichtterminal folgen können.

Beispiel. $\Pi = \{S \rightarrow ABC, A \rightarrow BC \mid a \mid d \mid \varepsilon, B \rightarrow b, C \rightarrow c \mid \varepsilon\}$

$$N = \{A, B, C, S\}$$

$$T = \{a, b, c, d\}$$

$$First(S) = First(ABC) = \underbrace{First(BC)}_{=\{b\}} \cup First(a) \cup First(d) \cup \underbrace{First(BC)}_{=\{b\}} = \{a, d, b\}$$

$$Follow(B) = \underbrace{First(C)}_{=\{c\}} \cup \underbrace{Follow(S)}_{=\emptyset} \cup \underbrace{Follow(A)}_{=First(B)=\{b\}} = \{c, b\}$$

Die nun folgende Action-Funktion gibt an, wie shift-reduce-Konflikte bzw. reduce-reduce-Konflikte aufgelöst werden können.

Definition 4.18 (Action-Funktion). Sei G' eine erweiterte Grammatik und C die zugehörige kanonische Sammlung von Mengen von LR(0)-Elementen. Dann ist die Action-Funktion

$$\text{action: } C \times T \rightarrow \text{auszuführende Funktion}$$

gegeben durch:

$\text{action}(I, a) = \text{„shift } I' \text{“}$, falls $\text{Sprung}(I, a) = I'$, $(a \in T)$, $(A \rightarrow \alpha.a\beta) \in I$, $\alpha, \beta \in (N \cup T)^*$

$\text{action}(I, a) = \text{„reduziere } A \rightarrow \alpha \text{“}$, falls $(A \rightarrow \alpha.) \in I$, $(A \neq S')$ und $a \in \text{Follow}(A)$

$\text{action}(I, \underbrace{\$}_{=(eof)}) = \text{„akzeptiere“}$, falls $(S' \rightarrow S.) \in I$

Algorithmus 4.19 (SLR-Parser). (Simple Left to Right) In Abb. 4.1 ist der SLR-Parser-Algorithmus gegeben.

Beispiel (SLR-Parser). Sie die Grammatik und dazugehörige kanonische Sammlung wie in zweiten Beispiel zum LR(0)-Algorithmus gegeben.

Die Follow-Betrachtung ergibt:

$$\text{Follow}(E) = \text{First}(+T) \cup \$ = \{+, \$\}$$

$$\text{Follow}(T) = \text{First}(*F) \cup \text{First}(F) = \{*, +, \$\}$$

$$\text{Follow}(F) = \{*, +, \$\}.$$

Daraus erhalten wir folgende Action-Funktion.

Action	id	$+$	$*$	$\$$
I_0	$shift\ I_4$	\perp	\perp	\perp
I_1	\perp	$shift\ I_5$	\perp	akzeptiere
I_2	\perp	$red\ E \rightarrow T$	$shift\ I_6$	$red\ E \rightarrow T$
I_3	\perp	$red\ T \rightarrow F$	$red\ T \rightarrow F$	$red\ T \rightarrow F$
I_4	\perp	$red\ F \rightarrow id$	$red\ F \rightarrow id$	$red\ F \rightarrow id$
I_5	$shift\ I_4$	\perp	\perp	\perp
I_6	$shift\ I_4$	\perp	\perp	\perp
I_7	\perp	$red\ E \rightarrow E + T$	$shift\ I_6$	$red\ E \rightarrow E + T$
I_8	\perp	$red\ T \rightarrow T * F$	$red\ T \rightarrow T * F$	$red\ T \rightarrow T * F$

<i>Datentypen:</i>	ADT Stack
<i>Eingabe:</i>	Eingabestring w Sprung-Funktion kanonische Sammlung von Mengen von LR(0)-Elementen Action-Funktion
<i>Ausgabe:</i>	Rechtsableitung $rl \vee \text{error}$
<i>Nachbedingung:</i>	$S' \xrightarrow{*}_r w = rl \Leftrightarrow \exists \text{ SLR-Ableitung}$ \wedge $\text{error} \Leftrightarrow \nexists \text{ SLR-Ableitung}$
<i>Init:</i>	Startzustand I_0 liegt auf dem Stack und $w\$$ im Eingabebuffer ip zeigt auf erstes Zeichen im Eingabebuffer
<pre> while (<i>true</i>) { $I = \text{Stack.top}()$ $a = w[ip]$ if ($\text{action}(I, a) == \text{„shift } I' \text{“}$) { $\text{Stack.push}(I')$ $ip++$ } else if ($\text{action}(I, a) == \text{„reduce } A \rightarrow \beta \text{“}$) { entferne β Elemente vom Stack $I' = \text{Stack.top}()$ $\text{Stack.push}(\text{Sprung}(I', A))$ gib aus: $A \rightarrow \beta$ } else if ($\text{action}(I, a) == \text{„akzeptiere“}$) { gib aus: $S' \rightarrow S$ return } else throw error } </pre>	
$\$ = \text{eof}$ (end of file) $ \beta = \text{L\u00e4nge von } \beta$	

Abbildung 4.1: SLR-Parser

Algorithmusdurchlauf:

Stack	Eingabestring	action
I_0	$z + z * z \$$	<i>shift</i> I_4
$I_0 I_4$	$+z * z \$$	<i>reduce</i> $F \rightarrow z$, <i>Sprung</i> (I_0, F) = I_3
$I_0 I_3$	$+z * z \$$	<i>reduce</i> $T \rightarrow F$, <i>Sprung</i> (I_0, T) = I_2
$I_0 I_2$	$+z * z \$$	<i>reduce</i> $E \rightarrow T$, <i>Sprung</i> (I_0, E) = I_1
$I_0 I_1$	$+z * z \$$	<i>shift</i> I_5
$I_0 I_1 I_5$	$z * z \$$	<i>shift</i> I_4
$I_0 I_1 I_5 I_4$	$*z \$$	<i>reduce</i> $F \rightarrow z$, <i>Sprung</i> (I_5, F) = I_3
$I_0 I_1 I_5 I_3$	$*z \$$	<i>reduce</i> $T \rightarrow F$, <i>Sprung</i> (I_5, T) = I_7
$I_0 I_1 I_5 I_7$	$*z \$$	<i>shift</i> I_6
$I_0 I_1 I_5 I_7 I_6$	$z \$$	<i>shift</i> I_4
$I_0 I_1 I_5 I_7 I_6 I_4$	$\$$	<i>reduce</i> $F \rightarrow z$, <i>Sprung</i> (I_0, F) = I_8
$I_0 I_1 I_5 I_7 I_6 I_8$	$\$$	<i>reduce</i> $T \rightarrow T * F$, <i>Sprung</i> (I_5, F) = I_7
$I_0 I_1 I_5 I_7$	$\$$	<i>reduce</i> $E \rightarrow E + T$, <i>Sprung</i> (I_0, E) = I_1
$I_0 I_1$	$\$$	<i>akzeptiere</i>

Ausgabe:

$$\left. \begin{array}{ll}
 F \rightarrow z & (1) \\
 T \rightarrow F & (2) \\
 E \rightarrow T & (3) \\
 F \rightarrow z & (4) \\
 T \rightarrow F & (5) \\
 F \rightarrow z & (6) \\
 T \rightarrow T * F & (7) \\
 E \rightarrow E + T & (8) \\
 S' \rightarrow E & (9)
 \end{array} \right\}$$

$$\begin{aligned}
 \Rightarrow & S' \xrightarrow{(9)} E \xrightarrow{(8)} E + T \xrightarrow{(7)} E + T * F \xrightarrow{(6)} E + T * z \xrightarrow{(5)} E + F * z \xrightarrow{(4)} E + z * z \xrightarrow{(3)} \\
 & T + z * z \xrightarrow{(2)} F + z * z \xrightarrow{(1)} z + z * z
 \end{aligned}$$

Beispiel (SLR-Parser). Sei $G = (N, T, \Pi, S')$ eine Grammatik mit:

$$\Pi = \{S' \rightarrow S, S \rightarrow L = R \mid R, L \rightarrow^* R \mid id, R \rightarrow L\}$$

$$N = \{S', S, L, R\}$$

$$T = \{*, =, id\}$$

Kanonische Sammlung:

$$I_0 = \text{H\u00fclle}(\{S' \rightarrow .S\}) = \{S' \rightarrow .S, S \rightarrow .L = R, S \rightarrow .R, L \rightarrow .*R, L \rightarrow .id, R \rightarrow .L\}$$

$$I_1 = \text{Sprung}(I_0, S) = \{S' \rightarrow S.\}$$

$$I_2 = \text{Sprung}(I_0, L) = \{S \rightarrow L. = R, R \rightarrow L.\}$$

$$I_3 = \text{Sprung}(I_0, R) = \{S \rightarrow R.\}$$

$$I_4 = \text{Sprung}(I_0, *) = \{L \rightarrow *.R, R \rightarrow .L, L \rightarrow .*R, L \rightarrow .id\}$$

$$I_5 = \text{Sprung}(I_0, id) = \{L \rightarrow id.\}$$

$$I_6 = \text{Sprung}(I_2, =) = \{S \rightarrow L = .R, R \rightarrow .L, L \rightarrow .*R, L \rightarrow .id\}$$

$$I_7 = \text{Sprung}(I_4, R) = \{L \rightarrow *.R.\}$$

...

$$\text{Follow}(R) = \text{Follow}(S) \cup \text{Follow}(L) = \text{Follow}(S') \cup \text{First}(= R) = \{=, \$\}$$

Bereits jetzt ist ersichtlich, dass es zu Konflikten kommen wird:

$$\left. \begin{array}{l} \text{Sprung}(I_2, =) = I_6 \\ \implies \text{action}(I_2, =) = \text{„shift } I_6 \text{“} \\ ((R \rightarrow L.) \in I_2) \wedge (= \in \text{Follow}(R)) \\ \implies \text{action}(I_2, =) = \text{„reduce } R \rightarrow L \text{“} \end{array} \right\} \text{shift-reduce-Konflikt}$$

Definition 4.20 (SLR-Eigenschaft). Eine Grammatik G hat die *SLR-Eigenschaft*, wenn die *action*-Funktion (Def. 4.18) eine wohldefinierte² partielle Funktion ist.

Es gibt weitere Algorithmen, die Konflikte in SLR l\u00f6sen.

Anmerkung. Bei SLR-Parser wird die Follow-Menge f\u00fcr die gesamte Grammatik betrachtet. Man k\u00f6nnte die Follow-Menge auch f\u00fcr einzelne Produktionen betrachten. Dies nennt man LR(1)-Parsen (LR-Parsen mit Vorausschau von einem Symbol).

Satz 4.21 (ohne Beweis). Die Klassen der Sprachen die durch einen deterministischen Push-Down-Automaten akzeptiert werden, stimmt mit der Klasse der LR(1)-Sprachen \u00fcberein.

Die Klasse der LR(1)-Sprachen stimmt mit der Klasse der LR(k)-Sprachen mit $k \geq 1$ (k Symbole Vorausschau) \u00fcberein.

F\u00fcr LR(1) und LALR(1) sei auf die Literatur verwiesen (z.B. Aho, Alfred V. and Lam, Sethi, Ullman, Jeffrey: *Compiler: Prinzipien, Techniken und Werkzeuge*)

²Wohldefiniertheit einer Funktion bedeutet, dass jedem Wert des Definitionsbereichs maximal ein Wert des Wertebereichs zugeordnet ist.