Theoretische Informatik 3. Semester

Prof. Dr. Martin Plümicke

23. November 2023

Inhaltsverzeichnis

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

$T = \{class, \{, \}, ;, while, (,), true\}$
$N = \{S, Classdecl, Classbodydecl\}$
$\Pi = \{$

$$
\begin{aligned}
&S && \rightarrow Classdecl \\
&Classdecl && \rightarrow class\ Classbodydecl \\
&Classbodydecl && \rightarrow \{\} \mid \{while(true)\{;\}\}
\end{aligned}
$$

$\}$

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 $A \to BC$ mit $A, B, C \in N$ oder $A \to a$ mit $A \in N$ und $a \in \Sigma$ haben (Ausnahme: $S \to \epsilon \in \Pi$ 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. $(A \to a)$ in Π' ergänzt wird.
4. Für eine Folge von Produktionen $A_1 \to A_2, A_2 \to A_3, \ldots, A_{n-1} \to A_n$ mit $A_i \in N'$ muss eine Regel $A_n \to B_1 \ldots B_m$ oder $A_n \to a$ mit $B_i \in N'$ und $a \in \Sigma$ existieren.

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

   $A \to B_1 C_1$

   $C_i \to B_{i+1} C_{i+1}$ für $i \in \{1, \ldots, m-3\}$

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

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

□

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

$$\begin{aligned}\Pi = \{ & S \to C \mid \epsilon, \\ & C \to aCb \mid ab \}\end{aligned}$$

gegeben.

Schritt 3:

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

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

Schritt 4:

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

Schritt 5:

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

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

Die äquivalente Grammatik in Chomsky-Normalform lautet:

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

mit

$$\begin{aligned}\Pi' = \{ & S \to AD \mid AB \mid \epsilon \\ & D \to CB \\ & C \to AE \mid AB \\ & E \to CB \\ & A \to a \\ & B \to b\}\end{aligned}$$

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 $n)$

$\quad N_{ii} = \{A \mid (A \to a) \in \Pi \text{ mit } a = w[i]\}$

for $(d = 1$ to $n - 1)$

$\quad$ **for** $(i = 1$ to $n - d)$

$\quad\quad j = i + d$

$\quad\quad$ **for** $(k = i$ to $j - 1)$

$\quad\quad\quad N_{ij} = N_{ij} \cup \{A \mid (A \to BC) \in \Pi \text{ mit } B \in N_{ik} \text{ und } C \in N_{(k+1)j}\}$

if $(S \in N_{1n})$ // dann ist w aus S ableitbar

$\quad erg = True$

else

$\quad erg = False$

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

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

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

$\Pi = \{$

$\quad S \to AB$

$\quad A \to AA$

$\quad B \to BC$

$\quad B \to BB$

$\quad C \to CC$

$\quad A \to a$

$\quad B \to b$

$\quad C \to c$

$\}$

$w =$ "$abcc$"

$n = |w| = 4$

1. for-Schleife:

$N_{11} = \{A\} \to a$

$N_{22} = \{B\} \to b$

$N_{33} = \{C\} \to c$

$N_{44} = \{C\} \to c$

2. for-Schleife:

$$
\begin{array}{lll}
d=1,\ i=1,\ j=2: & N_{12}: k=1: & A \in N_{11}, B \in N_{22} \Rightarrow (S \to AB) \in \Pi \quad \Rightarrow N_{12} = \{S\} \\
i=2,\ j=3: & N_{23}: k=2: & B \in N_{22}, C \in N_{33} \Rightarrow (B \to BC) \in \Pi \Rightarrow N_{23} = \{B\} \\
i=3,\ j=4: & N_{34}: k=3: & C \in N_{33}, C \in N_{44} \Rightarrow (C \to CC) \in \Pi \quad \Rightarrow N_{34} = \{C\} \\
d=2,\ i=1,\ j=3: & N_{13}: k=1: & A \in N_{11}, B \in N_{23} \Rightarrow (S \to AB) \in \Pi \quad \Rightarrow N_{13} = \{S\} \\
 & k=2: & S \in N_{12}, C \in N_{33} \Rightarrow \nexists X : (X \to SC) \in \Pi \\
i=2,\ j=4: & N_{24}: k=2: & B \in N_{22}, C \in N_{34} \Rightarrow (B \to BC) \in \Pi \quad \Rightarrow N_{24} = \{B\} \\
 & k=3: & B \in N_{23}, C \in N_{44} \Rightarrow (B \to BC) \in \Pi \Rightarrow \text{---}\,\text{„}\,\text{---} \\
d=3,\ i=1,\ j=4: & N_{14}: k=1: & A \in N_{11}, B \in N_{24} \Rightarrow (S \to AB) \in \Pi \Rightarrow N_{14} = \{S\} \\
 & k=2: & S \in N_{12}, C \in N_{34} \Rightarrow \nexists X : (X \to SC) \in \Pi \\
 & k=3: & S \in N_{13}, C \in N_{44} \Rightarrow \nexists X : (X \to SC) \in \Pi
\end{array}
$$

Ergebnis:

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

Überprüfung:

Die Ableitung lautet: $S \to AB \to aBC \to aBCc \to abcc$

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: $Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \to \mathcal{P}(Q \times \Gamma^*)$[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 $(q, aw, sb) \to (q', w, sc)$ 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 erstem Fall ist die Übergangsfunktion gegeben durch:

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

[1] Man 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

$\delta = \{$

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

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

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

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

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

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

$\}$

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

$$(q_0, a^n b^n, \gamma_0) \xrightarrow{(1)} (q_0, a^{n-1} b^n, \gamma_0 a) \xrightarrow{(2)} (q_0, a^{n-2} b^n, \gamma_0 aa) \xrightarrow{(2)} \ldots \xrightarrow{(2)} (q_0, b^n, \gamma_0 a^n) \xrightarrow{(3)}$$
$$(q_1, b^{n-1}, \gamma_0 a^{n-1}) \xrightarrow{(4)} \ldots \xrightarrow{(4)} (q_1, \varepsilon, \gamma_0) \xrightarrow{(5)} (q_2, \varepsilon, \varepsilon)$$

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

$\delta = \{$

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

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

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

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

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

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

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

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

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

$(q_1, \varepsilon, \gamma_0) \mapsto \{(q_E, \varepsilon)\}$ (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 nichtdeter-

ministisch 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

$\delta = \{$

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

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

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

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

$(q, x, \gamma) \mapsto \emptyset,$ sonst (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 $G = (N, \Sigma, \Pi, S)$
$w \in \Sigma^*$

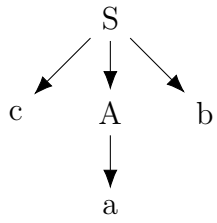
Ausgabe: $erg \in \{True, False\}$

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

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

Beispiel. $G = (N, T, \Pi, S)$ mit
$N = \{S, A\}$
$T = \{a, b, c\}$
$\Pi = \{S \to cAb, A \to ab|a\}$
$w = cab$
$\Rightarrow$ *Ableitung* : $S \to cAb \to cab$
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.

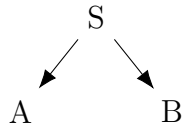
Definition 4.8 (Links- Rechtsableitungen)**.** Sei $w \to w'$ eine direkte Ableitung. Man bezeichnet $w \to_l w'$ als *direkte Linksableitung*, wenn das am weitesten links stehende Nichtterminal abgeleitet wird. Man bezeichnet $w \to_r w'$ als *direkte Rechtsableitung*, wenn das am weitesten rechts stehende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

Beispiel. Sei $G = (N, T, \Pi, S)$ eine Grammatik mit
$N = \{S, A, B, C\}$
$T = \{a, b, c\}$ und
$\Pi = \{S \to AB, A \to a, B \to bB, B \to c\}$

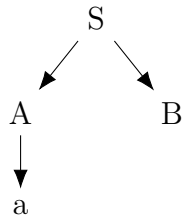
Sei weiterhin $w = abc$ das Eingabewort.

Aufbau des Ableitungsbaums durch Linksableitungen:

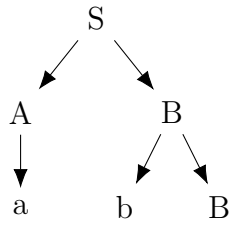
Man liest das nächste Token und wählt dann anhand diesem die Produktion aus.



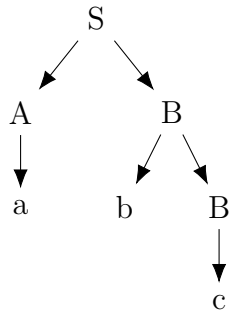
$nexttoken() = a$



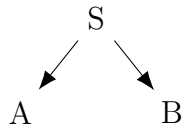
$nexttoken() = b$



$nexttoken() = c$



Aufbau des Ableitungsbaums durch Rechtsableitungen:



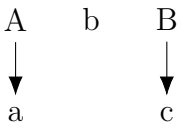
$nexttoken() = a$

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

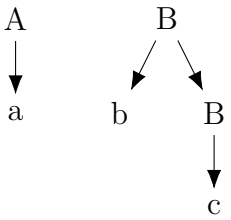
Anderer Ansatz: *Bottomup*: Wir bauen den Baum beginnend bei den *Blättern* zur *Wurzel* auf.

$nexttoken() = a$
wende an: $A \to a$

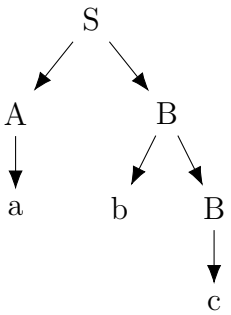
$nexttoken() = b$
$nexttoken() = c$
wende an: $B \to c$



wende an: $B \to bB$



wende an: $S \to AB$



Betrachtet man die Konstruktion rückwärts:

$$\underline{S} \to A\underline{B} \to Ab\underline{B} \to \underline{A}bc \to abc,$$

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.

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

Beispiel. Sei die Grammatik $G = (N, T, \Pi, S)$ mit
$N = \{S, A\}$,
$T = \{a, b\}$
$\Pi = \{S \to A, A \to aA \mid ab\}$.

Sei weiter $w = aab$

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

Mit 1 Zeichen lookahead, kann man nicht entscheiden ob $A \to aA$ oder $A \to ab$ 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
$N = \{S, A, B\}$,
$T = \{a, b\}$ und
$\Pi = \{S \to AB, A \to a, B \to b\}$

Es ist nur genau ein Wort ableitbar:

$S \to AB \to aB \to ab$

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

4.4 LR–Syntaxanalyse

Die LR–Syntaxanalyse ist eine Bottom–Up Syntaxanalyse. Man nennt die Syntaxanalyse auch eine ***Shift–Reduce–Syntaxanalyse***. Dabei wird für einen gegebenen Eingabestring (String von Lexemen) ein Parsbaum (Ableitungsbaum) von *Blättern zur Wurzel* (Bottom–Up) aufgebaut.

Beispiel. Sei $G = (N, T, \Pi, S)$ eine Grammatik mit
$N = \{S, A, B\}$,
$T = \{a, b, c, d, e\}$ und
$\Pi = \{S \to aABe, A \to Abc \mid b \mid d, B \to d\}$

Sei weiter $w = abbcde$ ein Eingabestring.

Bottom–Up Syntaxanalyse:

$$a.bbcde \longleftarrow ab.bcde \longleftarrow aA.b \longleftarrow aAb.cde \longleftarrow aAbc.de \longleftarrow aA.de \longleftarrow aAd.e \longleftarrow aAB.e \longleftarrow$$

Shift-Reduce-Konflikt: $aAA.cde$

Reduce-Reduce-Konflikt: $aAA.e$

$$aABe \longleftarrow S$$

Es entsteht wiederum eine umgekehrte Rechtsableitung

Bei der LR-Syntaxanalyse können sich zwei unterschiedliche Konflikttypen ergeben:

- **Shift–Reduce–Konflikt**: Es ist nicht eindeutig, ob geschiftet oder reduziert werden soll.
- **Reduce–Reduce–Konflikt**: Es ist nicht klar, auf welches Nichtterminal reduziert werden soll.

Bevor wir den Algorithmus der LR-Syntaxtanalyse angeben, benötigen wir einige Hilfsdefinitionen.

Im Folgenden sei $G = (N, T, \Pi, S)$ eine Grammatik.

Definition 4.10 (LR(0)–Element)**.** Unter einem LR(0)–Element einer Grammatik versteht man eine Produktion mit einem Punkt in der rechten Seite.

Beispiel. $S \to A.B$

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

$$\textit{Hülle}\colon \mathcal{P}(\{lr \mid lr \text{ ein LR(0)–Element}\}) \to \mathcal{P}(\{lr \mid lr \text{ ein LR(0)–Element}\})$$

die kleinste Menge , die durch folgende Regeln bestimmt ist:

- $I \subseteq \textit{Hülle}(I)$

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

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

Dann gilt:

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

Definition 4.12 (Sprung–Funktion)**.** Die Sprungfunktion

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

ist gegeben durch

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

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:

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

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

- Hülle $(\{S' \to .S\}) \in C$
- Wenn $I \in C$, dann gilt $\forall x \in (N \cup T)\ \mathit{Sprung}(I, x) \in C$, falls $\mathit{Sprung}(I, x) \neq \varnothing$

Beispiel. Sei die Grammatik $G = (N, T, \Pi, S)$ mit N, T, Π wie im vorigen Beispiel gegeben. Dann ist die erweiterte Grammatik gegeben durch $G' = (N \dot\cup \{S'\}, T, \Pi \dot\cup \{S' \to S\}, S')$
Im Folgenden geben wir die kanonische Sammlung an:

$H\ddot{u}lle\ (\{S' \to .S\}) = \underline{\{S' \to .S,\ S \to .E,\ E \to .E + E,\ E \to .E * E,\ E \to .id\}} =: \mathbf{I}_0$

$Sprung\ (I_0, +) = \varnothing,\ Sprung\ (I_0, *) = \varnothing$

$Sprung\ (I_0, id) = H\ddot{u}lle\ (\{E \to id.\}) = \underline{\{E \to id.\}} =: \mathbf{I}_1$

$Sprung(I_0, E) = H\ddot{u}lle(\{S \to E.\})\ \cup\ H\ddot{u}lle(\{E \to E. + E)\ \cup\ H\ddot{u}lle(\{E \to E. * E\})$

$= \underline{\{S \to E.,\ E \to E. + E,\ E \to E. * E\}} =: \mathbf{I}_2$

$Sprung\ (I_0, S) = H\ddot{u}lle\ (S' \to S.\}) = \underline{\{S' \to S.\}} =: \mathbf{I}_3$

$Sprung\ (I_1, x) = \varnothing$ für alle $x \in (N \cup T)$ //Sprungfunktion nicht anwendbar

$Sprung\ (I_2, +) = H\ddot{u}lle\ (\{E \to E{+}.E\}) = \underline{\{E \to E + .E,\ E \to .E + E,\ E \to .E * E,\ E \to .id\}} =: \mathbf{I}_4$

$Sprung\ (I_2, *) = H\ddot{u}lle\{E \to E * .E\} = \underline{\{E \to E * .E,\ E \to .E + E,\ E \to .E * E,\ E \to .id\}} =: \mathbf{I}_5$

$Sprung\ (I_2, E) = \varnothing,\ Sprung\ (I_2, id) = \varnothing,\ Sprung\ (I_2, S) = \varnothing$

$Sprung\ (I_3, x) = \varnothing$ für alle $x \in (N \cup T)$ //Sprungfunktion nicht anwendbar

$Sprung\ (I_4, +) = \varnothing\ Sprung\ (I_4, *) = \varnothing$

$Sprung\ (I_4, id) = \{E \to id.\} = \mathbf{I}_1$

$Sprung(I_4, E) = H\ddot{u}lle(\{E \to E + E.\})\ \cup\ H\ddot{u}lle(\{E \to E. + E\})\ \cup\ H\ddot{u}lle(\{E \to E. * E\})$

$= \underline{\{E \to E + E.,\ E \to E. + E,\ E \to E. * E\}} =: \mathbf{I}_6$

$Sprung\ (I_4, S) = \varnothing$

$Sprung\ (I_5, +) = \varnothing,\ Sprung\ (I_5, *) = \varnothing$

$Sprung\ (I_5, id) = I_1$

$Sprung(I_5, E) = H\ddot{u}lle(\ \{E \to E * E.\}\ )\ \cup\ H\ddot{u}lle(\ \{E \to E. + E\}\ )\ \cup\ H\ddot{u}lle(\ \{E \to E. * E\}\ )$

$= \underline{\{E \to E * E.,\ E \to E. + E,\ E \to E. * E\}} =: \mathbf{I}_7$

$Sprung(I_5, S) = \varnothing$

$Sprung(I_6, +) = I_4,\ Sprung(I_6, *) = I_5$

$Sprung(I_6, id) = \varnothing,\ Sprung(I_6, E) = \varnothing,\ Sprung(I_6, S) = \varnothing$

$Sprung(I_7, +) = I_4,\ Sprung(I_7, *) = I_5$

$Sprung(I_7, id) = \varnothing,\ Sprung(I_7, E) = \varnothing,\ Sprung(I_7, S) = \varnothing$

Damit ergibt sich die kanonische Sammlung $C = \{I_0, \ldots, I_7\}$.

Algorithmus 4.14 (LR(0)-Parser)**.**

Datentypen:	ADT Stack
Eingabe:	Eingabestring w Sprung-Funktion Kanonische Sammlung
Ausgabe:	Rechsableitung $rl \vee$ error
Nachbedingung:	$S' \xrightarrow{*}_r w = rl \Leftrightarrow \exists$ LR(0)-Ableitung $\wedge$ error $\Leftrightarrow \nexists$ LR(0)-Ableitung (vgl. 4.15)
Init:	Startzustand I_0 liegt auf dem Stack und $w\$$ im Eingabebuffer ip zeigt auf erstes Zeichen im Eingabebuffer

while $(true)$ {
  $I = Stack.top()$
  $a = w[ip]$
  if $(Sprung(I, a) = I'$ **&&** $\nexists(A \to \alpha.) \in I)$ {
    $Stack.push(I')$
    $ip++$
  }
  else if $(I == \{A \to \beta.\}$ **&&** $A \neq S')$ {
    entferne $|\beta|$ Elemente vom Stack
    $I' = Stack.top()$
    $Stack.push(Sprung(I', A))$
    gib aus: $A \to \beta$
  }
  else if $(I == \{S' \to S.\})$ {
    gib aus: $S' \to S$
    return
  }
  else throw error
}

$\$ = eof$ (end of file)
$|\beta|$ = Länge von β

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

Algorithmusdurchlauf:

Eingabestring: 1 + 1

stack	**Eingabestring**	**nächster Schritt**
I_0	$1 + 1\$$	$Sprung(I_0, 1) = I_1$
I_0, I_1	$+1\$$	$reduce\ E \to id.$ $Sprung(I_0, E) = I_2$
I_0, I_2	$+1\$$	**error**, da $S \to E. \in I_2$

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

$A \to \alpha.\beta$
$B \to \gamma.$

oder

$A \to \gamma_1.$
$B \to \gamma_2.$

enthält.

In beiden Fällen ergeben sich Mehrdeutigkeiten bei der LR-Syntaxtanalyse. 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.

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

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

oder Elemente der Form

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

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

Die LR(0)–Eigenschaft bedeutet, dass eine Bottom–Up LR-Syntaxtanalyse 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

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

gegeben durch folgende Funktion:

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

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

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

First(γ) 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

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

definiert durch die folgende Funktion:

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

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

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

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

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

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

gegeben durch:

action(I, a) = „shift I'“, falls $Sprung(I, a) = I', (a \in T), (A \rightarrow \alpha.a\beta) \in I, \alpha, \beta \in (N \cup T)^*$
action(I, a) = „reduziere $A \rightarrow \alpha$“, falls $(A \rightarrow \alpha.) \in I$, $(A \neq S')$ und a ∈ Follow(A)
action(I, $\underbrace{\$}_{=(eof)}$) = „akzeptiere“, falls $(S' \rightarrow S.) \in I$

Algorithmus 4.19 (SLR-Parser)**.** (**S**imple **L**eft to **R**ight)

Datentypen:	ADT Stack
Eingabe:	Eingabestring w Sprung-Funktion kanonische Sammlung von Mengen von LR(0)–Elementen Action-Funktion
Ausgabe:	Rechsableitung $rl \vee$ error
Nachbedingung:	$S' \xrightarrow{*}_r w = rl \Leftrightarrow \exists$ SLR-Ableitung $\wedge$ error $\Leftrightarrow \nexists$ SLR-Ableitung
Init:	Startzustand I_0 liegt auf dem Stack und $w\$$ im Eingabebuffer ip zeigt auf erstes Zeichen im Eingabebuffer

```
while (true) {
    I = Stack.top()
    a = w[ip]
    if (action(I, a) == „shift I'“) {
        Stack.push(I')
        ip++
    }
    else if (action(I, a) == „reduce A → β“) {
        endferne |β| Elemente vom Stack
        I' = Stack.top()
        Stack.push(Sprung(I', A))
        gib aus: A → β
    }
    else if (action(I, a) == „akzeptiere“) {
        gib aus: S' → S
        return
    }
    else throw error
}
```

$\$ = eof$ (end of file)
$|\beta|$ = Länge von β

Beispiel (SLR-Parser (1 von 2))**.** Sei $G = (N, T, \Pi, S')$ eine Grammatik mit:

$\Pi = \{S' \to S, S \to (S + S), S \to z\}$
$N = \{S', S\}$
$T = \{(,), +, z\}$

Kanonische Sammlung:

$I_0 = \mathit{Hülle}(\{S' \to .S\}) = \{S' \to .S,\ S \to .(S + S),\ S \to .z\}$
$I_1 = \mathit{Sprung}(I_0, S) = \{S' \to S.\}$
$I_2 = \mathit{Sprung}(I_0, () = \{S \to (.S + S),\ S \to .(S + S),\ S \to .z\}$
$I_3 = \mathit{Sprung}(I_0, z) = \{S \to z.\}$
$I_4 = \mathit{Sprung}(I_2, S) = \{S \to (S. + S)\}$
$\mathit{Sprung}(I_2, () = I_2$
$\mathit{Sprung}(I_2, z) = I_3$
$I_5 = \mathit{Sprung}(I_4, +) = \{S \to (S + .S),\ S \to .(S + S),\ S \to .z\}$
$I_6 = \mathit{Sprung}(I_5, S) = \{S \to (S + S.)\}$
$\mathit{Sprung}(I_5, () = I_2$
$\mathit{Sprung}(I_5, z) = I_3$
$I_7 = \mathit{Sprung}(I_6,)) = \{S \to (S + S).\}$

Sprung	S	z	$($	$)$	$+$
I_0	I_1	I_3	I_2	$\perp$	$\perp$
I_1	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
I_2	I_4	I_3	I_2	$\perp$	$\perp$
I_3	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
I_4	$\perp$	$\perp$	$\perp$	$\perp$	I_5
I_5	I_6	I_3	I_2	$\perp$	$\perp$
I_6	$\perp$	$\perp$	$\perp$	I_7	$\perp$
I_7	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$

$\perp \mathrel{\hat{=}}$ undefiniert

$\mathit{Follow}(S) = \mathit{First}(+S)) \cup \mathit{First}()) \cup \mathit{Follow}(S') = \{+,), \$\}$

Action	z	$($	$)$	$+$	\$
I_0	*shift* I_3	*shift* I_2	$\perp$	$\perp$	$\perp$
I_1	$\perp$	$\perp$	$\perp$	$\perp$	akzeptiere
I_2	*shift* I_3	*shift* I_2	$\perp$	$\perp$	$\perp$
I_3	$\perp$	$\perp$	*red* $S \to z$	*red* $S \to z$	*red* $S \to z$
I_4	$\perp$	$\perp$	$\perp$	*shift* I_5	$\perp$
I_5	*shift* I_3	*shift* I_2	$\perp$	$\perp$	$\perp$
I_6	$\perp$	$\perp$	*shift* I_7	$\perp$	$\perp$
I_7	$\perp$	$\perp$	*red* $S \to (S+S)$	*red* $S \to (S+S)$	*red* $S \to (S+S)$

1. Algorithmusdurchlauf:

Stack	**Eingabestring**	**action**
I_0	$(z+z)\$$	*shift* I_2
$I_0\ I_2$	$z+z)\$$	*shift* I_3
$I_0\ I_2\ I_3$	$+z)\$$	*reduce* $S \to z$, $Sprung(I_2, S) = I_4$
$I_0\ I_2\ I_4$	$+z)\$$	*shift* I_5
$I_0\ I_2\ I_4\ I_5$	$z)\$$	*shift* I_3
$I_0\ I_2\ I_4\ I_5\ I_3$	$)\$$	*reduce* $S \to z$, $Sprung(I_5, S) = I_6$
$I_0\ I_2\ I_4\ I_5\ I_6$	$)\$$	*shift* I_7
$I_0\ I_2\ I_4\ I_5\ I_6\ I_7$	\$	*reduce* $S \to (S+S)$, $Sprung(I_0, S) = I_1$
$I_0\ I_1$	\$	*akzeptiere*

Ausgabe:

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

2. Algorithmusdurchlauf:

Stack	**Eingabestring**	**action**
I_0	$()\$$	*shift* I_2
$I_0\ I_2$	$)\$$	$\perp \Rightarrow error$

Beispiel (SLR-Parser (2 von 2))**.** Sei $G = (N, T, \Pi, S')$ eine Grammatik mit:

$\Pi = \{S' \to S, S \to L = R \mid R, L \to^* R \mid id, R \to L\}$
$N = \{S', S, L, R\}$
$T = \{^*, =, id\}$

Kanonische Sammlung:

$I_0 = H\ddot{u}lle(\{S' \to .S\}) = \{S' \to .S,\ S \to .L = R,\ S \to .R,\ L \to .^*R,\ L \to .id,\ R \to .L\}$
$I_1 = Sprung(I_0, S) = \{S' \to S.\}$
$I_2 = Sprung(I_0, L) = \{S \to L. = R,\ R \to L.\}$
$I_3 = Sprung(I_0, R) = \{S \to R.\}$
$I_4 = Sprung(I_0, ^*) = \{L \to^* .R,\ R \to .L,\ L \to .^*R,\ L \to .id\}$
$I_5 = Sprung(I_0, id) = \{L \to id.\}$
$I_6 = Sprung(I_2, =) = \{S \to L = .R,\ R \to .L,\ L \to .^*R,\ L \to .id\}$
$I_7 = Sprung(I_4, R) = \{L \to^* R.\}$
$\ldots$

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

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

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

Definition 4.20 (SLR–Eigenschaft)**.** Eine Grammatik G hat die *SLR–Eigenschaft*, wenn die *action*–Funktion (Def. 4.18) eine wohldefinierte[2] partielle Funktion ist.

Es gibt weitere Algorithmen, die Konflikte in SLR lösen.

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

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

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 überein.

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

Für 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*)