6 Automaten und Formale Sprachen

6.1 Einführung grundlegender Begriffe

Alphabet, Zeichen, Wort

Ein **Alphabet** ist eine endliche, nicht leere Menge.

Die Elemente eines Alphabets heißen **Zeichen**.

Eine Folge von Zeichen aus einem Alphabet ist ein **Wort** über dem Alphabet.

Für ein Alphabet Σ beschreibt $\mathbf{\Sigma^*}$ die **Menge aller Wörter** über dem Alphabet Σ.

Die Menge Σ^* umfasst auch das **leere Wort** ε.

Die **Länge** eines Wortes $w \in \Sigma^*$ entspricht der Anzahl der Zeichen des Wortes und wird mit $|\mathbf{w}|$ bezeichnet.

Mehrfaches Hintereinanderstellen eines Wortes wird beschrieben durch $\mathbf{w^n} = \underbrace{\mathbf{ww...w}}_{\mathbf{n-mal}}$.

Es gilt $\mathbf{w^0} = \varepsilon$.

Formale Sprache

Eine **formale Sprache** über dem Alphabet Σ ist eine beliebige Teilmenge von Σ^*.

Formale Sprachen sind im Allgemeinen unendlich. Damit man mit Ihnen umgehen kann benötigt man **endliche Beschreibungenmöglichkeiten** für formale Sprachen. Das sind zum Beispiel **Grammatiken** und **Automaten**.

Beispiel

Beispiel 6.1. Sei $\Sigma_{\text{expr}} = \{\mathtt{0}, \mathtt{1}, \mathtt{2}, \mathtt{3}, \mathtt{4}, \mathtt{5}, \mathtt{6}, \mathtt{7}, \mathtt{8}, \mathtt{9}, +, -, *, /\}$ ein Alphabet.

Die Worte der Sprache $L_{\text{expr}} \subseteq \Sigma^*_{\text{expr}}$, der einfachen arithmetischen Ausdrücke über den natürlichen Zahlen, bestehen aus Zahlen und Operatoren.

- Zahlen sind Ziffernfolgen ohne führende Nullen.
- Operatoren sind die Zeichen $+$, $-$, $*$, $/$.

Weiterhin gelten folgende Regeln.

- Ein Wort beginnt mit einer Zahl.
- Jeder in einem Wort enthaltene Operator steht zwischen zwei Zahlen.

$$\begin{aligned} 1234567890 &\in L_{\text{expr}} \\ 1024 * 2 + 128/64 &\in L_{\text{expr}} \\ 1 + 2 + +012 &\notin L_{\text{expr}} \end{aligned}$$

6.2 Endliche Automaten

Automaten sind eine **endliche Beschreibungenmöglichkeiten** für formale Sprachen.

Ein Automat arbeitet ein Wort über einem Alphabet (die Eingabe) zeichenweise ab und akzeptiert es schließlich oder nicht. Die Menge der akzeptierten Wörter bildet die durch den Automaten beschriebene Sprache.

Ein Automat kann deterministisch oder nichtdeterministisch arbeiten.

Ein **deterministischer Automat** befindet sich beim Abarbeiten der Eingabe zu jedem Zeitpunkt in genau einem Zustand. Eine Eingabe wird akzeptiert, wenn der Automat sich nach dem Abarbeiten der gesamten Eingabe in einem akzeptierenden Zustand befindet.

Ein **nichtdeterministischer Automat** kann sich während des Abarbeitens der Eingabe in mehreren Zuständen gleichzeitig befinden. Eine Eingabe wird akzeptiert, wenn einer der Zustände, in dem sich der Automat nach dem Abarbeiten der gesamten Eingabe befindet, ein akzeptierender Zustand ist.

Das Zulassen von Nichtdeterminismus kann folgende Gründe haben.

Beschreibungsmächtigkeit.

> Es gibt einen nichtdeterministische Automaten, zu denen kein äquivalenter deterministischer Automat existiert.

Laufzeit.

Es gibt nichtdeterministische Automaten, die ihre Eingaben in sehr viel weniger Schritten akzeptieren als jeder äquivalente deterministische Automat.

Beschreibungskomplexität.

Ein nichtdeterministische Automaten kann unter Umständen sehr viel kleiner sein als jeder äquivalente deterministische Automat.

Für die im Folgenden betrachteten **endliche Automaten** erhöht der Nichtdeterminismus nur die Beschreibungskomplexität.

Definition 6.2. Ein **deterministischer endlicher Automat** (DEA) ist gegeben durch ein 5-Tupel $\mathcal{A} = (\Sigma, Q, q_0, F, \delta)$. Dabei ist

- Σ das **Alphabet** der Eingabewörter,
- Q die endliche Menge der **Zustände**,
- q_0 der **Startzustand**, $q_0 \in Q$,
- F die Menge der **akzeptierenden Zustände**, $F \subseteq Q$,
- δ die **Überführungsfunktion**, $\delta : Q \times \Sigma \to Q$.

Sei $w = w_1 \dots w_i w_{i+1} \dots w_n \in \Sigma^*$ eine Eingabe.

Der Automat befindet sich, nach dem Abarbeiten der Eingabe bis zum Zeichen w_i (inclusive), im Zustand $q \in Q$.

Dann **überführt** die Funktion δ den Automaten in den Zustand $\delta(q, w_{i+1}) \in Q$.

Beispiel

Beispiel 6.3.

$\mathcal{A}_{\text{expr}} = (\Sigma_{\text{expr}}, \{q_0, q_{\text{expr}}, q_{\text{op}}, q_{\text{error}}\}, q_0, \{q_{\text{expr}}\}, \delta_{\text{expr}})$

$\Sigma_{\text{expr}} = \{\mathtt{0}, \mathtt{1}, \mathtt{2}, \mathtt{3}, \mathtt{4}, \mathtt{5}, \mathtt{6}, \mathtt{7}, \mathtt{8}, \mathtt{9}, +, -, *, /\}$

Die Überführungsfunktion bildet wie folgt ab.

$$\begin{aligned}
\delta_{\text{expr}}(q_0, a) &= q_{\text{expr}} \text{ für alle } a \in \{1,2,3,4,5,6,7,8,9\} \\
\delta_{\text{expr}}(q_0, a) &= q_{\text{error}} \text{ für alle } a \in \{0,+,-,*,/\} \\
\delta_{\text{expr}}(q_{\text{expr}}, a) &= q_{\text{expr}} \text{ für alle } a \in \{0,1,2,3,4,5,6,7,8,9\} \\
\delta_{\text{expr}}(q_{\text{expr}}, a) &= q_{\text{op}} \text{ für alle } a \in \{+,-,*,/\} \\
\delta_{\text{expr}}(q_{\text{op}}, a) &= q_{\text{expr}} \text{ für alle } a \in \{1,2,3,4,5,6,7,8,9\} \\
\delta_{\text{expr}}(q_{\text{op}}, a) &= q_{\text{error}} \text{ für alle } a \in \{0,+,-,*,/\} \\
\delta_{\text{expr}}(q_{\text{error}}, a) &= q_{\text{error}} \text{ für alle } a \in \Sigma_{\text{expr}}
\end{aligned}$$

Beispiel 6.4.

$\mathcal{A}_{\text{expr}} = (\Sigma_{\text{expr}}, Q_{\text{expr}}, q_0, \{q_{\text{expr}}\}, \delta_{\text{expr}})$

$\Sigma_{\text{expr}} = \{\texttt{0}, \texttt{1}, \texttt{2}, \texttt{3}, \texttt{4}, \texttt{5}, \texttt{6}, \texttt{7}, \texttt{8}, \texttt{9}, +, -, *, /\}$

$Q_{\text{expr}} = \{q_0, q_{\text{expr}}, q_{\text{op}}, q_{\text{error}}\}$

Die Überführungsfunktion δ_{expr} bildet wie folgt ab.

$p \in Q_{\text{expr}}$	$\delta_{\text{expr}}(p, 0)$	$\delta_{\text{expr}}(p, z)$ $z \in \{1, \ldots, 9\}$	$\delta_{\text{expr}}(p, \text{op})$ $\text{op} \in \{+, -, *, /\}$
q_0	q_{error}	q_{expr}	q_{error}
q_{expr}	q_{expr}	q_{expr}	q_{op}
q_{op}	q_{error}	q_{expr}	q_{error}
q_{error}	q_{error}	q_{error}	q_{error}

Zustandsgraph

Deterministische endliche Automaten können durch ihren **Zustandsgraphen** veranschaulicht werden.

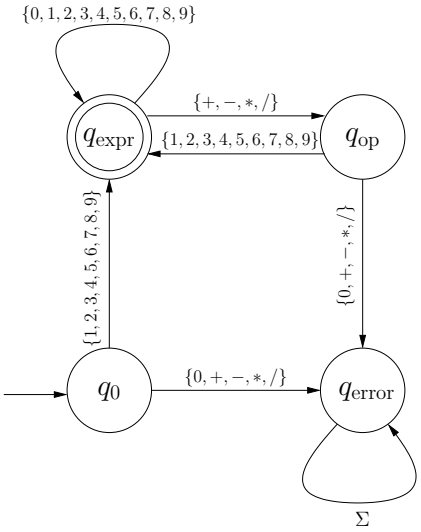
Ein Zustandsgraphen ist ein gerichteter Graph.

Für jeden Zustand enthält der Graph einen Knoten (dargestellt durch Kreise).

Der Anfangszustand ist durch eine unmarkierte eingehende Kante gekennzeichnet.

Akzeptierende Zustände sind durch Doppelkreise gekennzeichnet.

Jeder Knoten hat für jedes Zeichen aus dem Alphabet eine ausgehende Kante.



Folgezustand

Sei $\mathcal{A} = (\Sigma, Q, q_0, F, \delta)$ ein endlicher Automat.

- Für den Zustand $q \in Q$ bezeichne $qa = \delta(q, a)$ den **Folgezustand** von q nach dem Lesen des Zeichens $a \in \Sigma$.
- Für den Zustand $q \in Q$ und das Wort $w = w_1 w_2 \dots w_n \in \Sigma^*$ bezeichne $qw = (\cdots((qw_1)w_2), \dots, w_n)$ den **Folgezustand** von q nach dem Lesen des Wortes w.

Beispiel 6.5. $\mathcal{A}_{\text{expr}} = (\Sigma_{\text{expr}}, \{q_0, q_{\text{expr}}, q_{\text{op}}, q_{\text{error}}\}, q_0, \{q_{\text{expr}}\}, \delta_{\text{expr}})$

$$\begin{aligned} q_{\text{expr}}\ + &= q_{\text{op}} \\ q_0\ 1024 * 2 + 128/64 &= q_{\text{expr}} \\ q_{\text{expr}}\ + 012 &= q_{\text{error}} \end{aligned}$$

Ein Wort wird **akzeptiert**, wenn das Lesen seiner Zeichen den deterministischen Automaten von seinem Startzustand in einen akzeptierenden Zustand überführt.

Bemerkung

Das leere Wort ε wird genau dann akzeptiert, wenn der Startzustand ein akzeptierender Zustand ist.

Die Menge aller Wörter, die von dem Automaten akzeptiert werden können, ist die durch den Automaten akzeptierte Sprache.

Definition 6.6. Die durch den deterministischen endlichen Automaten $\mathcal{A} = (\Sigma, Q, q_0, F, \delta)$ **akzeptierte Sprache** $L_\mathcal{A} = \{w \in \Sigma^* \mid q_0 w \in F\}$.

Beispiel 6.7. Die von $\mathcal{A}_{\text{expr}}$ akzeptierte Sprache, ist die Sprache der einfachen arithmetischen Ausdrücke über den natürlichen Zahlen, $L_{\text{expr}} = L_{\mathcal{A}_{\text{expr}}}$.

Nichtdeterministische endliche Automaten

Definition 6.8. Ein **nichtdeterministischer endlicher Automat** (NEA) ist gegeben durch ein 5-Tupel $\mathcal{A} = (\Sigma, Q, q_0, F, \Delta)$. Dabei ist

- Σ das **Alphabet** der Eingabewörter,
- Q die endliche Menge der **Zustände**,
- q_0 der **Startzustand**, $q_0 \in Q$,
- F die Menge der **akzeptierenden Zustände**, $F \subseteq Q$,
- Δ die **Überführungsfunktion**, $\Delta : Q \times \Sigma \to \{U \mid U \subseteq Q\}$.

Sei $w = w_1 \dots w_i w_{i+1} \dots w_n \in \Sigma^*$ eine Eingabe.

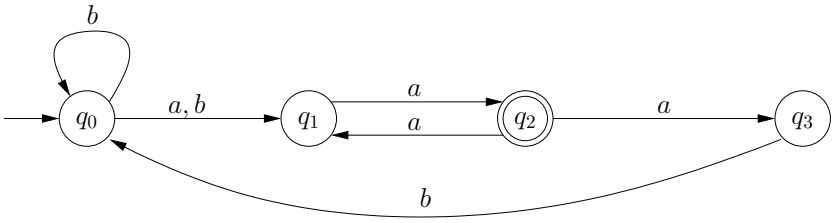
Der Automat befinde sich, nach dem Abarbeiten der Eingabe bis zum Zeichen w_i (inclusive), gleichzeitig in den Zuständen $U \subseteq Q$.

Dann **Überführt** die Funktion Δ den Automaten in folgende Zustandsmenge.

$$\bigcup_{q \in U} \Delta(q, w_{i+1}) \subseteq Q$$

Auch Nichtdeterministische endliche Automaten können durch einen Zustandsgraphen veranschaulicht werden.

Im Unterschied zu deterministischen endlichen Automaten kann ein Knoten für jedes Zeichen aus dem Alphabet **beliebig viele ausgehenden Kanten** (auch keine) haben.



Ein Wort wird akzeptiert, wenn das Lesen seiner Zeichen den nichtdeterministischen Automaten von seinem Startzustand in eine Menge von Zuständen überführt, die einen akzeptierenden Zustand enthält.

Bemerkung

Das leere Wort ε wird genau dann akzeptiert, wenn der Startzustand selbst ein akzeptierender Zustand ist.

Sei $\mathcal{A} = (\Sigma, Q, q_0, F, \Delta)$ ein nichtdeterministischer endlicher Automat und $q \in Q$ ein beliebiger Zustand.

Für $w = w_1 w_2 \dots w_n$ bezeichne qw die Menge der Folgezustände. Es gilt

$$\begin{aligned} M_1 &:= \Delta(q, w_1) \\ M_i &:= \bigcup_{p \in M_{i-1}} \Delta(p, w_i) \quad \text{für } i = 2, \dots, n \\ qw &:= M_n \end{aligned}$$

Definition 6.9. Die durch den nichtdeterministischen endlichen Automaten $\mathcal{A} = (\Sigma, Q, q_0, F, \Delta)$ **akzeptierte Sprache** $L_{\mathcal{A}} = \{w \in \Sigma^* \mid q_0 w \cap F \neq \emptyset\}$.

Satz 6.10. *Jeder nichtdeterministischen endlichen Automaten kann mit einem äquivalenten deterministicher Automat simuliert werden, der dieselbe formale Sprache akzeptiert.*

Beweis.

Durch Konstruktion.

Simulation eines NEA mit einem DEA über den Zustandsgraphen.

Schritt 1. Gegeben ist ein NEA mit n Zuständen.

- Erstelle einen Graphen, dessen Knoten markiert werden können und jeweils mehrere Zustände des NEA repräsentieren können.
- Der Graph enthält nur einen Knoten, der den Startzustand des NEA repräsentiert und nicht markiert ist.

Schritt 2. Betrachte einen nicht markierten Knoten v des Graphen.

- Sei Q_v die Menge der von v repräsentierten Zuständen des NEA.
- Für jedes $a \in \Sigma$ führe Folgendes aus.
  - Ermittle die Menge $Q_v a$ der Zustände, die unter a im NEA von den Zuständen in Q_v erreichbar sind, $Q_v a = \bigcup_{q \in Q_v} \Delta(q, a)$.
  - Erzeuge einen nicht markierten Knoten, der die Menge $Q_v a$ repräsentiert, wenn er noch nicht vorhanden ist (auch für $Q_v a = \emptyset$).

    Bemerkung. Die $Q_v a$ sind nicht notwendigerweise alle verschieden.
  - Füge eine mit a markierte Kante vom Knoten v zu dem $Q_v a$ repräsentierenden Knoten ein.
- Der abgearbeitete Knoten v wird markiert.

Wiederhole Schritt 2 bis keine nicht markierten Knoten mehr übrig sind.

Schritt 3. Graph zu DEA.

- Die Knoten sind die Zustände des DEA.

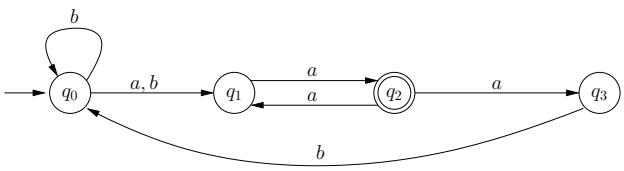
  Bemerkung. Es gibt nur noch markierte Knoten. Deshalb hat jeder Knoten für jedes $a \in \Sigma$ genau eine ausgehende Kante.
- Der Startzustand des DEA ist der Knoten, der den Startzustand des NEA repräsentiert.

- Die akzeptierenden Zustände des DEA sind die Konten, die mindestens einen akzeptierenden Zustand des NEA repräsentieren.

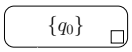
Beispiel

Schritt 1

NEA

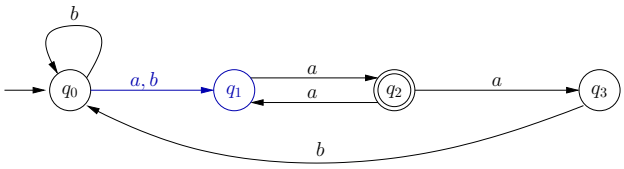


Graph

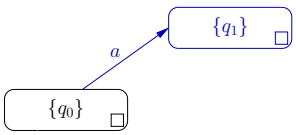


Schritt 2 (Auswahl)

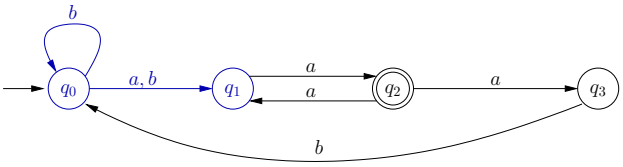
NEA



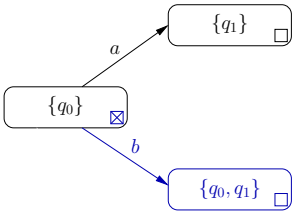
Graph



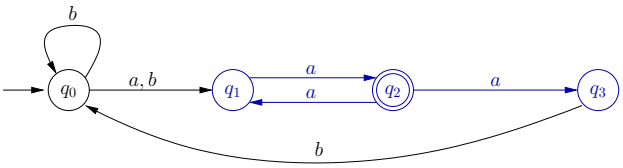
NEA



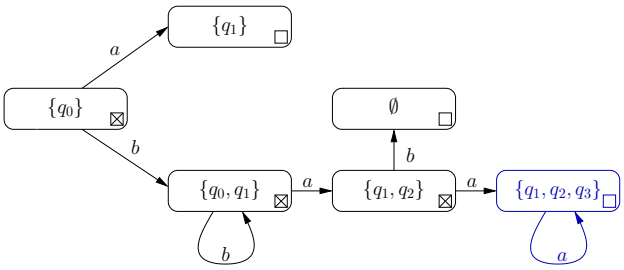
Graph

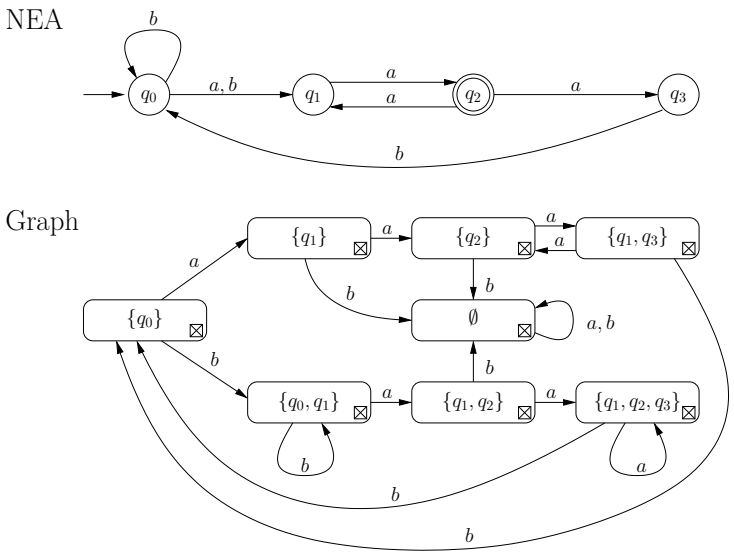


NEA

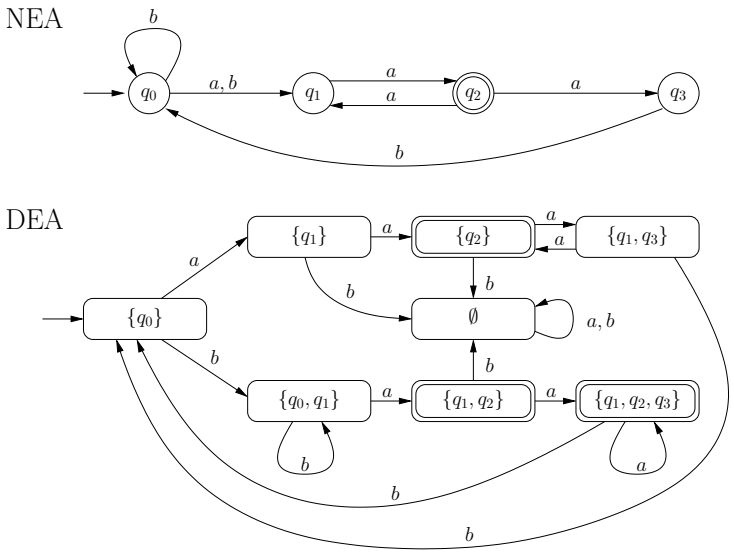


Graph




NEA
q_0
q_1
q_2
q_3
a, b
a
b
Graph
$\{q_1\}$
$\{q_2\}$
$\{q_1, q_3\}$
$\{q_0\}$
$\emptyset$
$\{q_0, q_1\}$
$\{q_1, q_2\}$
$\{q_1, q_2, q_3\}$

Schritt 3


NEA
q_0
q_1
q_2
q_3
a, b
a
b
DEA
$\{q_1\}$
$\{q_2\}$
$\{q_1, q_3\}$
$\{q_0\}$
$\emptyset$
$\{q_0, q_1\}$
$\{q_1, q_2\}$
$\{q_1, q_2, q_3\}$

Beschreibungsmächtigkeit.

Zu jedem NEA existiert ein äquivalenter DEA.

Laufzeit.

Ein Eingabe wird sowohl vom NEA als auch von DEA nach Abarbeitung aller Zeichen akzeptiert.

Beschreibungskomplexität.

Die Mächtigkeit der Zustandmenge eines DEA, der einen NEA $\mathcal{A} = (\Sigma, Q, q_0, F, \Delta)$ simuliert, ist im schlechtesten Fall

$$|\{U \mid U \subseteq Q\}| = 2^{|Q|} \ .$$

6.3 Reguläre Sprachen

Eine von einem endlichen Automaten akzeptierte Sprache ist eine formale Sprache. Aber nicht für jede formale Sprache kann ein endlicher Automat angegeben werden, der diese akzeptiert.

Definition 6.11. Eine formale Sprache $L \subseteq \Sigma^*$ heißt **reguläre Sprache**, wenn es einen endlichen Automaten $\mathcal{A}$ gibt der L akzeptiert, $L = L_{\mathcal{A}}$.

Regeln für reguläre Sprachen

Ist $L \subseteq \Sigma^*$ regulär, dann auch ist auch das *Komplement* regulär.

$$\overline{L} := \{w \in \Sigma^* | w \notin L\}$$

Sind L_1, L_2 regulär, dann ist auch der *Durchschnitt* regulär.

$$L_1 \cap L_2 := \{w | w \in L_1 \text{ und } w \in L_2\}$$

Sind L_1, L_2 regulär, dann ist auch die *Verkettung* regulär.

$$L_1 L_2 := \{vw | v \in L_1, w \in L_2\}$$

Sind L_1, L_2 regulär, dann ist auch die *Vereinigung* regulär.

$$L_1 \cup L_2 := \{w | w \in L_1 \text{ oder } w \in L_2\}$$

Die Kleenesche Hülle

Seien $L_1, L_2 \subseteq \Sigma^*$ Sprachen über Σ. Dann ist die Sprache

$$L_1 L_2 := \{vw \mid v \in L_1, w \in L_2\} \ .$$

Sei $L \subseteq \Sigma^*$, dann gilt

$$\begin{aligned} L^0 &:= \{\varepsilon\} \\ L^1 &:= L \\ L^i &:= LL^{i-1} \quad \text{für alle } i \in \mathbb{N} \\ L^* &:= \bigcup_{i \in \mathbb{N}} L^i \end{aligned}$$

Die Sprache L^* ist die **Kleenesche Hülle** der Sprache L. Beispiel: $L = \{a, bb\}$ und $L^* = \{\varepsilon, a, bb, aa, abb, bba, bbbb, aaa, \ldots\}$

Ist L regulär, dann ist auch L^* regulär.

Definition 6.12. Die **regulären Basissprachen** eine Alphabets Σ sind

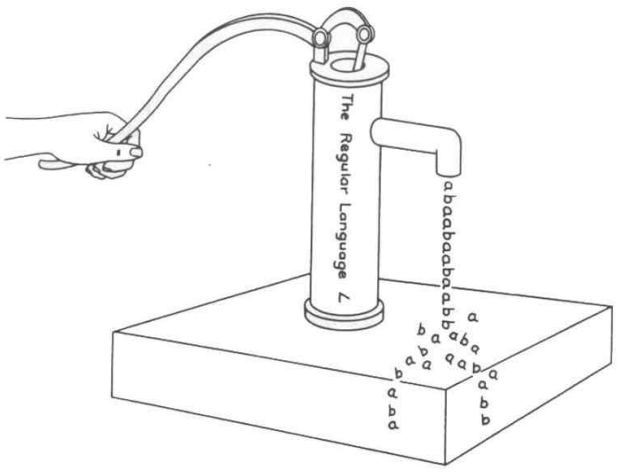
- $\emptyset$,
- $\{\varepsilon\}$,
- $\{a\}$ für jedes $a \in \Sigma$.

Satz 6.13. *Eine Sprache $L \subseteq \Sigma^*$ ist regulär genau dann, wenn sie durch eine endliche Folge der Operationen Vereinigung, Verkettung und Kleenesche Hülle aus den regulären Basissprachen von Σ erzeugt werden kann.*

Reguläre Sprachen sind *einfach gebaut.*

6.3.1 Pumping-Lemma für reguläre Sprachen

Das **Pumping-Lemma** liefert ein Kriterium um zu zeigen, dass eine Sprache **nicht regulär** ist.



Satz 6.14. *Sei L eine reguläre Sprache. Dann gibt es eine natürliche Zahl (Pumping-Konstante) $k \in \mathbb{N}$, sodass man jedes Wort $w \in L$ mit $|w| \geq k$ zerlegen kann in Wörter $xyz \in \Sigma^*$, für die gilt*

- $w = xyz$,
- $|y| \geq 1$, *das heißt* $y \neq \varepsilon$,
- $|xy| \leq k$,
- *für alle* $i \in \mathbb{N} \cup \{0\}$ *gilt,* $xy^iz \in L$.

Anhand des Pumping-Lemmas kann bewiesen werden, dass eine Sprache **nicht regulär** ist. Trifft das Pumping-Lemma nicht zu, dann ist die Sprache auch nicht regulär.

Satz 1. *Werden k Objekte in ℓ Fächer gegeben ($\ell < k$), dann enthält mindestens eins der Fächer mehr als ein Objekt.*

Beweis. Seien $F := \{F_1, \ldots, F_\ell\}$ die Menge der Fächer. Sei $|F_i|$ die Anzahl der Objekte im Fach F_i. Es gilt

$$\sum_{i=1}^{\ell} |F_i| = k$$

Annahme. Keins der Fächer enthält mehr als ein Objekt. Das heißt für alle F_i gilt $|F_i| \leq 1$, daraus folgt

$$\sum_{i=1}^{\ell} |F_i| \leq \ell < k$$

Das ist ein Widerspruch. □

Beweis

Da L regulär ist gibt es eine deterministischen endlichen Automaten $\mathcal{A} = (\Sigma, Q, q_0, F, \delta)$, der L akzeptiert.

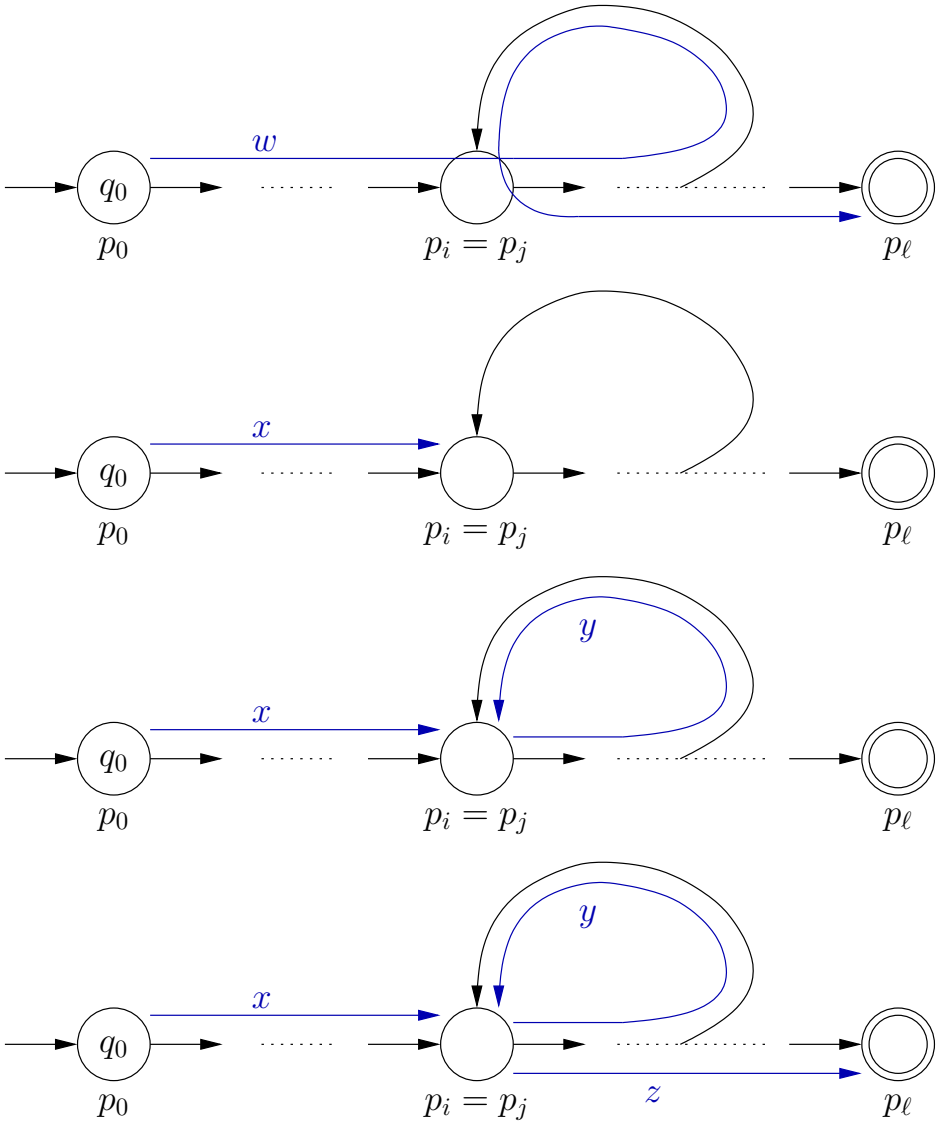
Wähle die Pumping-Konstante $k = |Q|$, wobei $|Q|$ die Anzahl der Zustände von $\mathcal{A}$ ist.

Sei $w \in L$ ein beliebiges Wort der Länge $|w| = \ell \geq k$.

Das Wort w wird vom Automaten $\mathcal{A}$ akzeptiert. Beim Akzeptieren von w durchläuft der Automat $\ell + 1$ Zustände $p_0, \ldots, p_\ell \in Q$, mit Startzustand $p_0 = q_0$ und akzeptierenden Zustand $p_\ell \in F$.

Da $k+1 > |Q|$ folgt mit dem Schubfachprinzip, dass in der Teilfolge $p_0, \ldots, p_k$ ein Zustand doppelt auftreten muss.

Das heißt, es gibt i, j mit $0 \leq i < j \leq k$ für die gilt $p_i = p_j$.



Das Wort w wird nun in $x, y, z \in \Sigma^*$ zerlegt, sodass Folgendes gilt.

$$p_0 x = p_i \qquad p_i y = p_j \qquad p_j z = p_\ell$$

Es gilt

- $xyz = w$ nach Konstruktion,

- $|y| \geq 1$, weil $i < j$,
- $|xy| \leq k$, weil $j \leq k$,
- für alle $i \in \mathbb{N} \cup \{0\}$ gilt $xy^iz \in L$ und mit $\hat{p} := p_i = p_j$ folgt

$$\begin{aligned} p_0 x &= \hat{p} \\ \hat{p}y &= \hat{p} \quad \text{also} \quad \hat{p}yy = \hat{p}y = \hat{p} \quad \text{und} \quad \hat{p}y^i = \hat{p} \\ \hat{p}z &= p_\ell \end{aligned}$$

  das heißt $p_0 xy^i z = p_\ell$.

□

6.3.2 Anwendung des Pumping-Lemmas

Behauptung.

Die Sprache L ist nicht regulär.

Beweis.

Durch Widerspruch.

Annahme. L ist regulär, daraus folgt das Pumping-Lemma gilt mit Pumping-Konstante $k \in \mathbb{N}$.

Betrachte **ein** Wort $w \in L$ mit $|w| \geq k$ und zeige **für jede** Zerlegung $xyz = w$ mit $|y| \geq 1$ und $|xy| \leq k$ **gibt es** ein $i \in \mathbb{N} \cup \{0\}$ mit $xy^iz \notin L$.

Das ist ein Widerspruch. □

Behauptung.

Die Sprache $L = \{a^n b^n \mid n \in \mathbb{N}\}$ ist nicht regulär.

Beweis. Durch Widerspruch.

Annahme. L ist regulär, daraus folgt das Pumping-Lemma gilt mit Pumping-Konstante $k \in \mathbb{N}$.

Betrachte das Wort $a^k b^k \in L$. Sei $xyz = a^k b^k$ eine beliebige Zerlegung.

$|xy| \leq k$, daraus folgt xy besteht nur aus a's, das heißt $xy = a^r$ mit $r \leq k$.

$|y| \geq 1$, daraus folgt y besteht aus mindestens einem a, das heißt $xy = a^{r-s}a^s$ mit $1 \leq s \leq r$.

Es gilt $a^k b^k = a^{r-s}a^s a^{k-r}b^k = xyz$.

Abpumpen von y. Es gilt $xy^0z = a^{k-s}b^k \notin L$, weil mit $s \geq 1$ gilt $k - s \neq k$. Das ist ein Widerspruch.

Aufpumpen von y. Sei $i \geq 2$, dann ist $xy^iz = a^{k-s}a^{si}b^k \notin L$, weil mit $s \geq 1$ gilt $k + s(i-1) \neq k$. Das ist ein Widerspruch. □

6.4 Grammatiken

Ergänzung.

Σ^+ ist die Menge der nichtleeren Zeichenketten über dem Alphabet Σ.

Definition 6.15. Eine **Grammatik** über einem Alphabet Σ ist ein 4-Tupel $G = (N, T, P, S)$ bestehend aus:

- einer Menge N von *Nichtterminalsymbolen*
- einer Menge $T \subseteq \Sigma$ von *Terminalsymbolen* mit $N \cap T = \emptyset$
- einer nichtleeren Menge $P \subseteq (N \cup T)^+ \times (N \cup T)^*$ von *Produktionen*

  [Produktion (p, q) wird mit $p \to q$ notiert.]
- einem Startsymbol $S \in N$.

Beispiel

Beispiel 6.16. $G_{\mathtt{a}^n\mathtt{b}^n} = (\{\mathtt{S}\}, \{\mathtt{a}, \mathtt{b}\}, \{S \to \mathtt{a}S\mathtt{b}, S \to \varepsilon\}, S)$

Abkürzung für *Produktionsgruppen* mit gleicher linker Seite.

Beispiel: $S \to \mathtt{a}S\mathtt{b}|\varepsilon$

Ableitung

Sei $x = vpw$ mit $v, w \in (N \cup T)^*$, $p \in (N \cup T)^+$ und $p \to q$ eine Produktion.

Durch Ersetzen von p durch q entsteht $y = vqw \in (N \cup T)^*$.

Notation. $x \Longrightarrow y$ ($x \overset{p \to q}{\Longrightarrow} y$).

Allgemein.

Seien $x \in (N \cup T)^+$ und $y \in (N \cup T)^*$. Dann heißt y mit G **ableitbar** aus x, falls

- $x = y$ oder
- es gibt $v_1, v_2, \ldots, v_n$ mit $x = v_1$, $y = v_n$ und $v_i \Longrightarrow v_{i+1}$ für $1 \leq i < n$

Notation. $x \overset{G}{\Longrightarrow} y$

Definition 6.17. Die von der Grammatik G **erzeugte Sprache** $L(G)$ ist die Menge aller aus dem Startsymbol S ableitbaren Worte aus T^*.

$$L(G) = \{w \mid S \overset{G}{\Longrightarrow} w \text{ und } w \in T^*\}$$

Beispiel 6.18. Die Grammatik $G_{\mathtt{a}^n\mathtt{b}^n}$ erzeugt (oder generiert) die Sprache $L_{\mathtt{a}^n\mathtt{b}^n} := \{\mathtt{a}^n\mathtt{b}^n \mid n \in \mathbb{N}\}$.

6.5 Reguläre Grammatiken

Definition 6.19. Die Grammatik (N, T, P, S) heißt **rechtslinear**, wenn jede Produktion die Form $A \to aB$ oder $A \to a$ oder $A \to \varepsilon$ hat, mit $A, B \in N$ und $a \in T$.

Definition 6.20. Die Grammatik (N, T, P, S) heißt **linkslinear**, wenn jede Produktion die Form $A \to Ba$ oder $A \to a$ oder $A \to \varepsilon$ hat, mit $A, B \in N$ und $a \in T$.

Definition 6.21. Eine **reguläre Grammatik** ist eine rechtslineare oder linkslineare Grammatik.

Satz 6.22. *Zu jedem endlichen Automaten $\mathcal{A}$ kann man eine reguläre Grammatik $G_\mathcal{A}$ konstruieren mit $L(G_\mathcal{A}) = L_\mathcal{A}$ und umgekehrt.*

Beweis durch Konstruktion.

- Erzeuge aus einer rechtslinearen Grammatik einen äquivalenten nichtdeterministischen endlichen Automaten.
- Es gilt, nichtdeterministischen und deterministische endlichen Automaten sind äquivalent.

- Erzeuge aus einem deterministischen endlichen Automaten eine äquivalente rechtslinearen Grammatik.

- Bleibt zu zeigen, rechtslineare und linkslineare Grammatiken sind äquivalent.

Rechtslineare Grammatik → NEA

Sei $G = (N, T, P, S)$ einen rechtslineare Grammatik. Erzeuge einen nichtdeterministischen endlichen Automat $\mathcal{A} = (\Sigma, Q, q_0, F, \Delta)$, sodass $L(G) = L_{\mathcal{A}}$ gilt.

- Setze Alphabet gleich Terminale, $\Sigma := T$.

- Setze Zustände gleich Nichterminale, $Q := N$.

- Setze Startzustand gleich Startsymbol, $q_0 := S$.

- Noch keine akzeptierenden Zustände, $F := \emptyset$.

- Initialisiere die Übergangsfunktion $\Delta(A, a) = \emptyset$ für alle $A \in Q$, $a \in \Sigma$.

- Gibt es eine Produktion $A \to a$, erzeuge neuen akzeptierenden Zustand E, setze $F := \{E\}$ und $Q := Q \cup \{E\}$.

- Für alle Produktionen $A \to \varepsilon$ setze $F := F \cup \{A\}$.

- Für alle Produktionen $A \to a$ erweitere Δ, sodass gilt $E \in \Delta(A, a)$.

- Für alle Produktionen $A \to aB$ erweitere Δ, sodass gilt $B \in \Delta(A, a)$.

DEA → rechtslineare Grammatik

Sei $\mathcal{A} = (\Sigma, Q, q_0, F, \delta)$ ein deterministischer endlicher Automat. Erzeuge eine rechtslineare Grammatik $G = (N, T, P, S)$, sodass $L_{\mathcal{A}} = L(G)$ gilt.

- Setze Terminale gleich Alphabet, $T := \Sigma$.
- Setze Nichterminale gleich Zustände $N := Q$.
- Setze Startsymbol gleich Startzustand, $S := q_0$.
- Initialisiere die Menge der Produktionen, $P := \emptyset$.
- Für alle $A \in F$ erweitere die Menge der Produktionen $P := P \cup \{A \to \varepsilon\}$.
- Für alle $A \in Q$, $a \in \Sigma$ erweitere die Menge der Produktionen $P := P \cup \{A \to aB\}$ mit $B = \delta(A, a)$.

  Bemerkung.

  Es müssen auch die akzeptierenden Zustände ($A \in F \subset Q$) berücksichtigt werden.

Links- und rechtslineare Grammatiken

Satz 6.23. *Für jede linkslineare Grammatik G_ℓ kann eine rechtslineare Grammatik G_r konstruiert werden mit $L(G_\ell) = L(G_r)$ und umgekehrt.*

Beweis durch Konstruktion.

Linkslineare Grammatik → rechtslineare Grammatik

Sei $G_\ell = (N_\ell, T_\ell, P_\ell, S_\ell)$ einen linkslineare Grammatik und $G_r = (N_r, T_r, P_r, S_r)$ die gesuchte rechtslineare Grammatik.

Die Terminale sind gleich.

- $T_r := T_\ell$

Die Nichterminale werden erweitert um eine neues Startsymbol S_r, das alte Startsymbol S_ℓ wird ein einfaches Nichterminal.

- $N_r := N_\ell \cup \{S_r\}$

Initalisiere die Produktionen. G_ℓ erzeugt die Wörter von rechts nach links, d.h. in G_r ist S_ℓ das Ende eines Wortes.

- $P_r := \{S_\ell \to \varepsilon\}$

Die Abschluss eines Wortes in G_ℓ ist der Anfang eines Wortes in G_r.

- Für alle $(A \to \varepsilon) \in P_\ell$ setze $P_r := P_r \cup \{S_r \to A\}$.
- Für alle $(A \to a) \in P_\ell$ setze $P_r := P_r \cup \{S_r \to aA\}$.

Alle Produktionen mit zwei Nichtterminalen ändern die Erzeugungsrichtung.

- Für alle $(A \to Ba) \in P_\ell$, setze $P_r := P_r \cup \{B \to aA\}$.

Entferne aus P_r alle Produktionen $S_r \to A$ durch Einsetzen.

- $P_r := P_r \backslash \{S_r \to A\}$.
- Für alle $(A \to \varepsilon) \in P_r$ setze $P_r := P_r \cup \{S_r \to \varepsilon\}$
- Für alle $(A \to a) \in P_r$ setze $P_r := P_r \cup \{S_r \to a\}$
- Für alle $(A \to aB) \in P_r$ setze $P_r := P_r \cup \{S_r \to aB\}$

Entferne aus P_r alle Produktionen mit linker Seite $A \neq S_r$, wenn A auf keiner rechten Seite einer Produktion vorkommt, deren linke Seite nicht A ist.

Bemerkung

Die Konstruktion einer äquivalenten linkslinearen Grammatik zu einer gegebenen rechtslinearen Grammatik kann analog durchgeführt werden.

Bemerkung

Wenn rechts- und linkslineare Regeln in der Menge der Produktionen gemischt werden, darf man nicht erwarten, dass das Ergebnis regulär ist.

Beispiel 6.24. Die Grammatik $G = \{\{S, A, B\}, \{a, b\}, \{S \to Ba, S \to aA, A \to Sa, B \to aS, S \to b\}, S\}$ erzeugt die Sprache $\{a^i b a^i\}$, die nicht regulär ist.

6.6 Parser

6.6.1 Kontextfreie Grammatiken

Definition 6.25. Eine Grammatik (N, T, P, S) heißt **kontextsensitive Grammatik**, wenn für alle Produktionen $p \to q$ aus P gilt $|p| \leq |q|$. [Keine rechte Seite einer Produktionen ist kürzer als die zugehörigen linke Seite.]

Definition 6.26. Eine formale Sprache ist genau dann ein **kontextsensitive Sprache** (*context sensitive language, CSL*), wenn es eine kontextsensitive Grammatik gibt, die diese Sprache erzeugt.

Definition 6.27. Eine Grammatik (N, T, P, S) heißt **kontextfreie Grammatik**, wenn für alle Produktionen $p \to q$ aus P gilt $p \in N$. [$p \to q$ ist unabhängig von umgebenden Symbolen anwendbar oder nicht.]

Definition 6.28. Eine formale Sprache ist genau dann ein **kontextfreie Sprache** (*context free language, CFL*), wenn es eine kontextfreie Grammatik gibt, die diese Sprache erzeugt.

Beispiel 6.29. Die kontextfreie Grammatik

$$G_{\mathtt{a}^n\mathtt{b}^n} = (\{\mathtt{S}\}, \{\mathtt{a}, \mathtt{b}\}, \{S \to \mathtt{a}S\mathtt{b} \mid \varepsilon\}, S)$$

erzeugt die Sprache

$$L_{\mathtt{a}^n\mathtt{b}^n} := \{\mathtt{a}^n\mathtt{b}^n \mid n \in \mathbb{N}\} \ .$$

Beispiel 6.30. Grammatik $G := \{N, T, P, S\}$

- Terminale $T := \{+,\ (,\ ),\ \text{int},\ \$\$\}$.
- Nichtterminale $N := \{\text{START},\ \text{SUM},\ \text{EXPR}\}$.
- Startsymbol $S := \text{START}$.

- Produktionen P

  START → EXPR $$
  EXPR → (SUM)
  EXPR → int
  SUM → EXPR + EXPR

6.6.2 Ableitung von kontextfreien Grammatiken

Wenn eine kontextfreie Grammatik G zur Analyse von Quelltexten einer Programmiersprache verwendet wird, dann gilt Folgendes.

- $L(G)$ ist die Menge der Tokenfolgen, die zu gültigen Quellprogrammen gehören.
- Der Scanner (lexikalischen Analyse) liefert ein Folge von Terminalsymbolen der Grammatik (genannt *Token*) zurück.

  Das $$ Token markiert das Ende der Eingabe (z.B. *EOF*). Es ist hilfreich, denn damit kann das Ereignis *Ende der Eingabe erreicht* in die Grammatik eingearbeitet werden.

  Beispiel

  Quellprogramm `(77+(5+10))` wird zur Tokenfolge `(int+(int+int))`$$.
- Der **Parser** (Syntaxanalyse) versucht die Tokenfolge entsprechend der Grammatik abzuleiten.

  Gelingt die Ableitung, ist die Tokenfolge und damit das Quellprogrammen gültig.

Der Prozess zur Erzeugung eines Wortes aus einer Grammatik ist die **Ableitung** (*derivation*).

Ein durch den Ableitungsprozess entstehende Zeichenkette, insbesondere wenn diese noch Nichtterminale enthält, wird **Satzform** genannt.

Das die abschließende Satzform (das Wort der Sprache), die nur Terminalsymbole enthält, wird auch **Satz** (*sentence*) der Grammatik oder **Ergebnis** (*yield*) des Ableitungsprozesses genannt.

Sei $G = (N, T, P, S)$ eine kontextfreie Grammatik.

- **Start.** Beginne mit einer Produktion $S \to u \in (N \cup T)^*$. Setze die Satzform sf gleich der rechten Seite dieser Produktion $sf := u$.
- **Abbruchbedingung.** Enthält die Satzform sf nur Terminale ist sf das Ergebnis des Ableitungsprozesses.
- **Einsetzen.** Wähle ein Nichtterminal A aus der Satzform $sf = vAw$ und eine Produktion $A \to q \in (N \cup T)^*$. Ersetze A durch q, das heißt $sf = vqw$.
- **Schleife.** Beginne wieder mit dem Prüfen der Abbruchbedingung.

Der durch den Ableitungsprozess erzeugte Satz sf ist ein Wort der von G erzeugten Sprache $L(G)$, das heißt $sf \in L(G)$.

Rechtsseitige Ableitungen (*rightmost derivations*)

- Ersetze jeweils das **erste rechte** Nichtterminalsymbol in jedem Ableitungsschritt.
- Manchmal auch **kanonische** Ableitung genannt.

Linksseitige Ableitungen (*leftmost derivations*)

- Ersetze jeweils das **erste linke** Nichtterminalsymbol in jedem Ableitungsschritt.

Andere Ableitungsreihenfolgen sind möglich.

Bemerkung

Die meisten Parser suchen entweder nach einer rechtsseitigen oder linksseitigen Ableitung

Beispiel 6.31. Grammatik $G := \{N, T, P, S\}$

- Terminale $T := \{+,\ (,\ ),\ \text{int},\ \$\$\}$.
- Nichtterminale $N := \{\text{START},\ \text{SUM},\ \text{EXPR}\}$.
- Startsymbol $S := \text{START}$.
- Produktionen

  START $\to$ EXPR \$\$
  EXPR $\to$ (SUM)
  EXPR $\to$ int
  SUM $\to$ EXPR + EXPR

Ist folgender Satz gültig?

(int + (int + int)) $$

Eine mögliche linksseitige Ableitung des Satzes.

START → EXPR $$
(SUM) $$
(EXPR + EXPR) $$
(int + EXPR) $$
(int + (SUM)) $$
(int + (EXPR + EXPR))$$
(int + (int + EXPR)) $$
(int + (int + int)) $$

6.6.3 Syntaxanalyse

Ableitungsbaum

Ein **Ableitungsbaum** (auch Parsebaum genannt) ist eine graphische Repräsentation des Ableitungsprozesses.

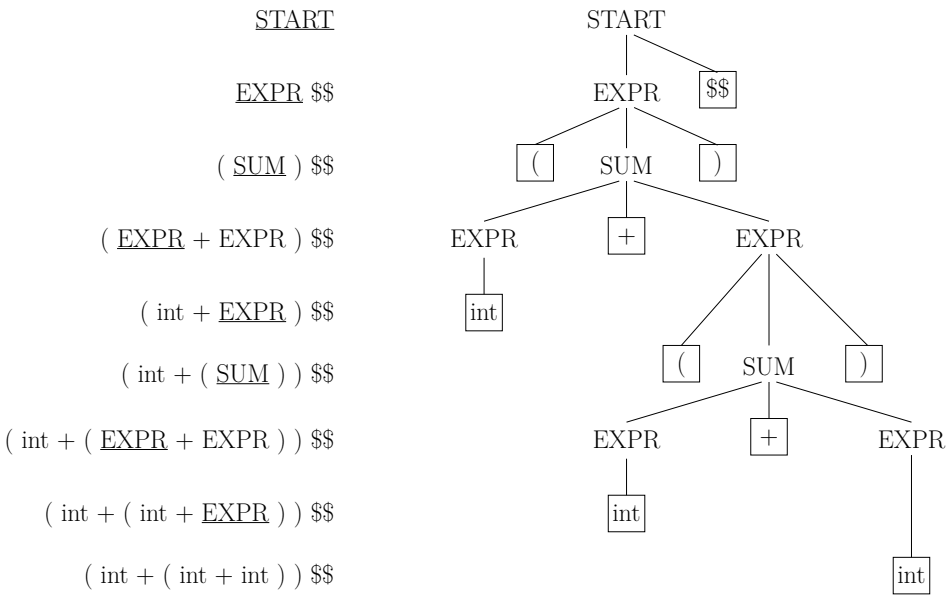
Die **Blätter** eines Ableitungsbaumes entsprechen den Terminalsymbolen der Grammatik.

Innere Knoten eines Ableitungsbaumes entsprechen den Nichtterminalsymbolen der Grammatik (linke Seite der Produktionen).

Bemerkung

Die meisten Parser konstruieren einen Ableitungsbaum während des Ableitungsprozesses für eine spätere Analyse.

Beispiel



Eindeutig/Mehrdeutig

Eindeutige Grammatiken haben genau einen Ableitungsbaum, wenn nur links- oder rechtsseitige Ableitungen verwendet werden.

Eine Grammatik ist **mehrdeutig**, wenn ein Satz mit (mindestens zwei) verschiedenen Ableitungsbäumen abgeleitet werden kann, obwohl nur links- oder rechtsseitige Ableitungen verwendet werden.

Es keinen allgemeingültigen Algorithmus zur Entdeckung von Mehrdeutigkeiten und deren Auflösung.

Vermeidung von Mehrdeutigkeiten durch Festlegung der Auswertungsreihenfolge.

- Bei verschiedenen Terminalen Prioritäten setzen.
- Bei gleichartigen Terminalen Gruppierung festlegen.

Beispiel

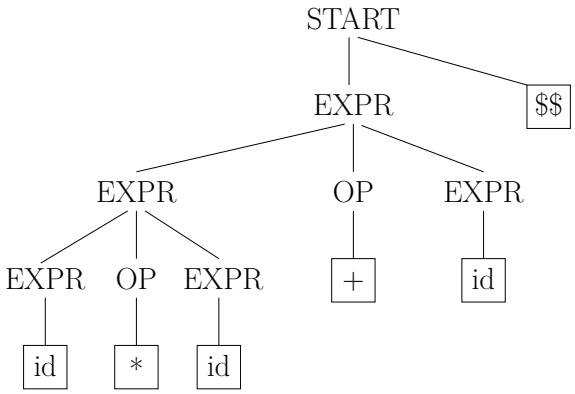
Beispiel 6.32.

- Terminale $T := \{+,\ -,\ *,\ /,\ \text{id},\ \$\$\}$.
- Nichtterminale $N := \{\text{START},\ \text{EXPR},\ \text{OP}\}$.
- Startsymbol $S :=$ START.
- Produktionen

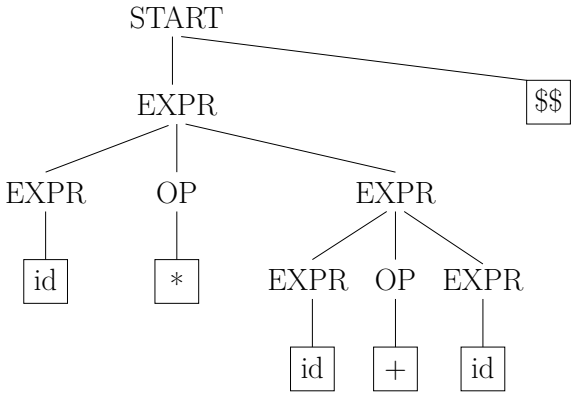
  START → EXPR $$
  EXPR → id | EXPR OP EXPR
  OP → + | - | * | /

Satz
`id * id + id $$`

Ableitungsbaum 1



Ableitungsbaum 2



Kontextfreie Grammatiken **erzeugen** durch den Ableitungsprozess Sätze.

Ein Parser für eine kontextfreie Grammatik **erkennt** Sätze in der von einer kontextfreien Grammatiken erzeugten Sprache.

Parser können automatisch aus einer kontextfreien Grammatik generiert werden.

Parser zum Erkennen von allgemeinen kontextfreien Sprachen können langsam sein.

Klassen von Grammatiken und Parsern

LL(k) Parser

- Eingabe wird von **links-nach-rechts** (erstes L) abgearbeitet.
- **Linksseitige Ableitung** (zweites L).
- Genannt *top-down*, *prädiktive* oder *voraussagende* Parser.

LR(k) Parser

- Eingabe wird von **links-nach-rechts** (L) abgearbeitet.
- **Rechtsseitige Ableitung** (R).
- Genannt *bottom up*, *shift-reduce* oder *schiebe-reduziere* Parser.

Der Wert **k** steht für die Anzahl von Token für die in der Eingabe **vorausgeschaut** werden muss um eine Entscheidung treffen zu können.

- LL(k). Welche nächste Produktion (rechten Seite) ist bei der linksseitigen Ableitung zu wählen.

Syntaxanalyse

Top-down oder LL-Syntaxanalyse.

- Baue den Ableitungsbaum von der Wurzel aus bis hinunter zu den Blättern auf.
- Berechne in jedem Schritt voraus welche Produktion zu verwenden ist um den aktuellen nichtterminalen Knoten des Ableitungsbaumes aufzuweiten (*expand*), indem die nächsten k Eingabesymbole betrachtet werden.

Beispiel

Grammatik
G = { {ID_LIST, ID_LIST_TAIL},
{`id`, `,`, `;`},
P,
ID_LIST }

Produktionen P
ID_LIST → `id` ID_LIST_TAIL
ID_LIST_TAIL → `,id` ID_LIST_TAIL
ID_LIST_TAIL → `;`$$

Quelltext
```
A, B, C;
```

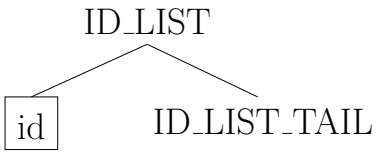
Satz (Tokenfolge)
`id,id,id;`$$

Ableitungsbaum

ID_LIST

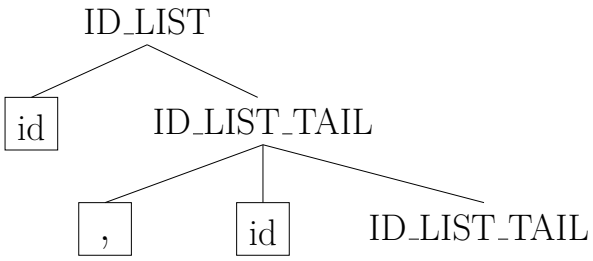
Satz (Tokenfolge)
`id`,id,id;$$

Ableitungsbaum



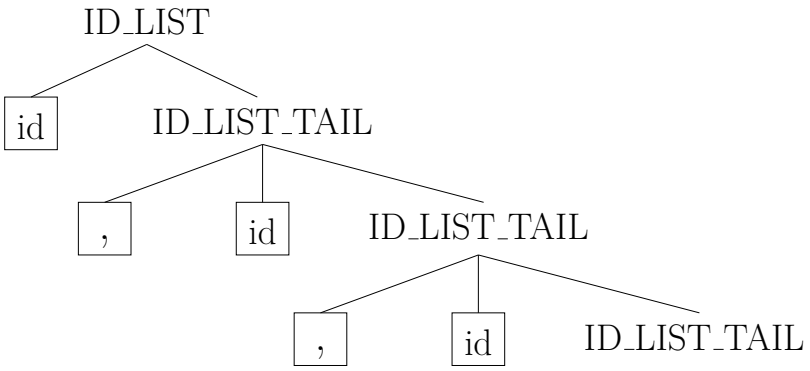
Satz (Tokenfolge)
id`,`id,id;$$

Ableitungsbaum



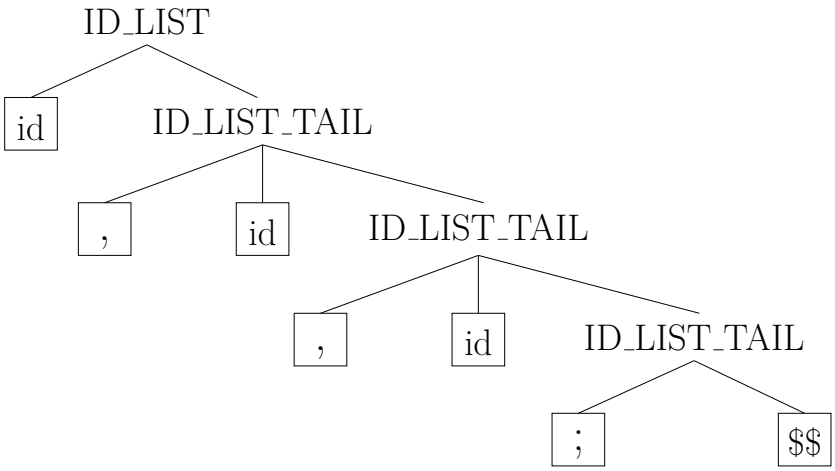
Satz (Tokenfolge)
id,id`,`id;$$

Ableitungsbaum



Satz (Tokenfolge)
id , id , id ; $$

Ableitungsbaum



Ergebnis
id , id , id ; $$

Der Satz ist ein Wort der Sprache $L(G)$.

6.6.4 Rekursive LL-Parser

Rekursiver Abstieg

Rekursiver Abstieg ist ein Weg um LL (top-down) Parser zu implementieren

- Es ist einfach von Hand zu schreiben.
- Es wird kein Parsergenerator benötigt.

Jedes nichtterminale Symbol in der Grammatik hat einen Prozeduraufruf.

- Die nachste anzuwendende linksseitige Ableitung wird bestimmt (*predict*), indem nur die nächsten k Symbole angeschaut werden.

Beispiel

Grammatik $G := \{N, T, P, S\}$ einfacher arithmetischen Ausdrücke.

$T = \{+, *, \text{const}, \$\$\}$
$N = \{\text{PROG, EXPR, TERM, TTAIL, FACTOR, FTAIL}\}$
$S = \text{PROG}$

$$
\begin{array}{lll}
\text{P} := \{ & \text{PROG} & \rightarrow \text{EXPR \$\$} \\
 & \text{EXPR} & \rightarrow \text{TERM TTAIL} \\
 & \text{TERM} & \rightarrow \text{FACTOR FTAIL} \\
 & \text{TTAIL} & \rightarrow + \text{ TERM TTAIL} \mid \varepsilon \\
 & \text{FACTOR} & \rightarrow \text{const} \\
 & \text{FTAIL} & \rightarrow * \text{ FACTOR FTAIL} \mid \varepsilon \quad \}
\end{array}
$$

Programm

```
void error() {
  // no derivation for input
}

token current_token() {
  // return current token of input
}

void next_token() {
  // goto next token of input
}

void match(token expected) {
```

```
    if (expected == current_tocken())
      next_token();
    else
      error();
  }

  // start symbol ----------------------------------
  void PROG() {
    EXPR();
    match($$);
  }

  void EXPR() {
    TERM();
    TTAIL();
  }

  void TTAIL() {
    if (current_token() == +) {
      match(+);
      TERM();
      TTAIL();
    }
  }

  void TERM() {
    FACTOR();
    FTAIL();
  }

  void FTAIL() {
    if (current_token() == *) {
      match(*);
      FACTOR();
      FTAIL();
    }
  }

  void FACTOR() {
    match(const);
  }
```

LL(k)-Syntaxanalyse

Finde zu einer Eingabe von Terminalsymbolen (Tokens) passende Produktionen in einer Grammatik durch Herstellung von linksseitigen Ableitungen.

Für eine gegebene Menge von Produktionen für ein Nichtterminal,

$$X \to \alpha_1 | \ldots | \alpha_n$$

und einen gegebenen, linksseitigen Ableitungsschritt

$$\gamma X \delta \Rightarrow \gamma \alpha_i \delta$$

muss bestimmt werden welches α_i zu wählen ist, indem nur die nächsten k Eingabesymbole betrachtet werden.

Bemerkung.

Für eine gegebene Menge von linksseitigen Ableitungsschritten, ausgehend vom Startsymbol $S \Rightarrow \gamma X \delta$, wird die Satzform γ nur aus Terminalen/Tokens bestehen und repräsentiert den passenden Eingabeabschnitt zu den bisherigen Grammatikproduktionen.

LL-Syntaxanalyse, Linksrekursion

Linksrekursion. Folgender Grammatikausschnitt enthält eine linksrekursive Produktionen.

$$A \to A\alpha$$
$$A \to \beta$$

Wenn eine Grammatik linksrekursive Produktionen enthält, dann kann es für diese Grammatik keinen LL Parser geben.

- LL Parser können in eine Endlosschleife eintreten, wenn versucht wird eine linksseitige Ableitung mit so einer Grammatik vorzunehmen.

Linksrekursion kann durch Umschreiben der Grammatik vermieden werden.

$$A \to \beta A'$$
$$A' \to \alpha A' \,|\, \varepsilon$$

Nicht formale Rechtfertigung.

Linkrekursion.

$$A \to A\alpha$$
$$A \to \beta$$

Ableitungen

$$A \Rightarrow A\alpha \Rightarrow A\alpha\alpha \Rightarrow A\alpha\alpha\alpha \Rightarrow \ldots$$
$$\beta\alpha\alpha\alpha\ldots$$

Auflösung.

$$A \to \beta A'$$
$$A' \to \alpha A' \,|\, \varepsilon$$

Ableitungen

$$A \Rightarrow \beta A' \Rightarrow \beta\alpha A' \Rightarrow \beta\alpha\alpha A' \Rightarrow \ldots$$
$$\beta\alpha\alpha\alpha\ldots$$

Beispiel

Produktionen mit Linksrekursion.

ID_LIST → ID_LIST_PREFIX `;`$$
ID_LIST_PREFIX → ID_LIST_PREFIX `,id`
ID_LIST_PREFIX → `id`

Umschreiben, neues Nichterminal ID_LIST_TAIL.

ID_LIST → ID_LIST_PREFIX `;`$$
ID_LIST_PREFIX → `id` ID_LIST_TAIL
ID_LIST_TAIL → `,id` ID_LIST_TAIL | ε

Vereinfachen, Nichterminal ID_LIST_PREFIX fällt weg.

ID_LIST → `id` ID_LIST_TAIL `;`$$
ID_LIST_TAIL → `,id` ID_LIST_TAIL | ε

LL-Syntaxanalyse, gemeinsame Präfixe

Gemeinsame Präfixe. Folgender Grammatikausschnitt enthält Produktionen mit gemeinsamen Präfixen.

$$A \to \alpha X$$
$$A \to \alpha Y$$

Wenn eine Grammatik Produktionen mit gemeinsamen Präfixen der Länge k enthält, dann kann es für diese Grammatik keinen LL(k) Parser geben. Denn der Parser kann nicht entscheiden mit welcher Produktion der nächste Ableitungsschritt vorgenommen werden soll.

Gemeinsame Präfixe können durch Umschreiben der Grammatik beseitigt werden.

$$A \to \alpha A'$$
$$A' \to X \mid Y$$

Beispiel

Produktionen mit gemeinsamen Präfixen.

STMT → `id :=` EXPR
STMT → `id (` ARGUMENT_LIST `)`

Umschreiben, neues Nichterminal STMT_LIST_TAIL.

STMT → `id` STMT_LIST_TAIL
STMT_LIST_TAIL → `:=` EXPR | `(` ARGUMENT_LIST `)`

Probleme mit der LL-Syntaxanalyse

Der Ausschluss von Linksrekursion und gemeinsamen Präfixen garantiert nicht, dass es für diese Grammatik einen LL(1)-Parser gibt.

Es gibt Algorithmen mit denen man für eine gegebene Grammatik ermitteln kann, ob sich ein LL(1)-Parser finden lässt.

Wenn man keinen LL(1)-Parser für eine Grammatik finden kann, dann muss man eine mächtigere Technik verwenden, z.B. LL(k)-Parser mit $k \geq 2$.

6.6.5 Nicht-rekursive LL-Parser

Literatur

Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman.

Compilers - Principles, Techniques and Tools (Dragon Book),

Addison-Wesley.

Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman.

Compiler, Prinzipien, Techniken und Werkzeuge

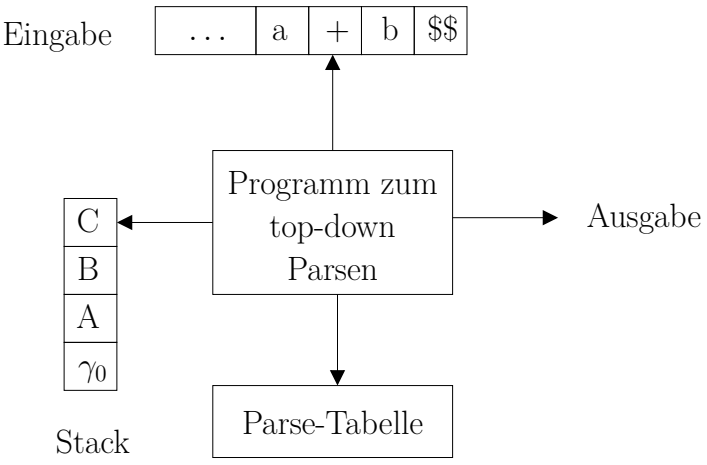
Pearson Studium.

Modell

Es ist möglich LL-Parser **nicht-rekursiv** zu implementieren.

Für die nicht-rekursive Implementation wird ein Stapel explizit verwaltet, anstatt die implizite Stapelverwaltung der rekursiven Aufrufe zu benutzen.

Für die Auswahl einer Produktion für ein zu ersetzendes Nichtterminal inspizieren nicht-rekursive Parser eine **Parse-Tabelle**.



Ein **tabellen-gesteuerter LL-Parser** besteht aus einem Eingabepuffer, einem Stapel, einer Parse-Tabelle und einem Ausgabestrom.

Der Eingabenpuffer enthält den Satz, der analysiert werden soll, abgeschlossen mit dem Symbol \$\$, das bei diesem Verfahren **nicht** zu den Terminalen der Grammatik gehört.

Der Stapel enthält eine Folge von Grammatiksymbolen (Terminale und Nichterminale) und eine Kennzeichnung γ_0 für das unterste Element, die nicht zu den Grammatiksymbolen gehört.

Zu Beginn enthält der Stapel γ_0 und darüber das Startsymbol der Grammatik.

Die Parse-Tabelle ist ein zweidimensionales Feld $M[A, a]$, wobei A ein Nichterminal und a ein Terminal der Grammatik oder \$\$ ist. Ein Eintrag der Parse-Tabelle enthält entweder eine Produktion $A \to \alpha$ oder ist leer.

Steuerung

Ein Programm mit folgenden Verhalten übernimmt die Steuerung des Parsers.

Das oberste Stapelsysmbol X und das aktuelle Eingabezeichen a werden inspiziert.

- $X = \gamma_0$, $a =$ \$\$. Der Parser stoppt und meldet Erfolg.
- $X = a$ ($X \neq \gamma_0$, $a \neq$ \$\$). Das oberste Element X wird vom Stapel entfernt und das nächste Zeichen der Eingabe wird zum aktuellen Eingabezeichen.
- X ist eine Nichterminal und der Eintrag $M[X, a]$ der Parse-Tabelle enthält genau eine Produktion. Das oberste Stapelelement X wird entfernt und die rechte Seite der Produktion wird Symbol für Symbol von rechts-nach-links auf den Stapel gelegt.

  Bemerkung. An dieser Stelle könnte beliebiger Code, z.B. zum Aufbauen des Parsenbaums, ausgeführt werden.
- In allen anderen Fällen stoppt der Parser und meldet einen Fehler.

Beispiel

Grammatik $G = (N, T, P, S)$.

Nichtterminale $N = \{E, E', F, T, T'\}$,

Terminal $T = \{\mathbf{id}, +, *, (,)\}$,

Startsymbol $S = E$,

Produktionen P wie folgt.

$$
\begin{array}{ll}
E & \to TE' \\
E' & \to +TE' \,|\, \varepsilon \\
T & \to FT' \\
T' & \to *FT' \,|\, \varepsilon \\
F & \to (\,E\,) \,|\, \mathbf{id}
\end{array}
$$

Parse-Tabelle

	id	+	*	(	)	$$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \varepsilon$	$E' \to \varepsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \varepsilon$	$T' \to *FT'$		$T' \to \varepsilon$	$T' \to \varepsilon$
F	$F \to \mathbf{id}$			$F \to (\,E\,)$		

Ablauf für Eingabe $\mathbf{id} + \mathbf{id} * \mathbf{id}$.

Eingabe (erzeugt)	Stapel	Eingabe (Rest)	Produktion
	$E\ \gamma_0$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\ \$\$$	$E \to TE'$
	$TE'\ \gamma_0$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\ \$\$$	$T \to FT'$
	$FT'E\ \gamma_0$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\ \$\$$	$F \to \mathbf{id}$
	$\mathbf{id}T'E'\ \gamma_0$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\ \$\$$	
$\mathbf{id}$	$T'E'\ \gamma_0$	$+\ \mathbf{id} * \mathbf{id}\ \$\$$	$T' \to \varepsilon$
$\mathbf{id}$	$E'\ \gamma_0$	$+\ \mathbf{id} * \mathbf{id}\ \$\$$	$E' \to +TE'$
$\mathbf{id}$	$+TE'\ \gamma_0$	$+\ \mathbf{id} * \mathbf{id}\ \$\$$	
$\mathbf{id} +$	$TE'\ \gamma_0$	$\mathbf{id} * \mathbf{id}\ \$\$$	$T \to FT'$
$\mathbf{id} +$	$FT'E'\ \gamma_0$	$\mathbf{id} * \mathbf{id}\ \$\$$	$F \to \mathbf{id}$
$\mathbf{id} +$	$\mathbf{id}T'E'\ \gamma_0$	$\mathbf{id} * \mathbf{id}\ \$\$$	
$\mathbf{id} + \mathbf{id}$	$T'E'\ \gamma_0$	$*\ \mathbf{id}\ \$\$$	$T' \to *FT'$
$\mathbf{id} + \mathbf{id}$	$*FT'E'\ \gamma_0$	$*\ \mathbf{id}\ \$\$$	
$\mathbf{id} + \mathbf{id} *$	$FT'E'\ \gamma_0$	$\mathbf{id}\ \$\$$	$F \to \mathbf{id}$
$\mathbf{id} + \mathbf{id} *$	$\mathbf{id}T'E'\ \gamma_0$	$\mathbf{id}\ \$\$$	
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$T'E'\ \gamma_0$	$\$\$$	$T' \to \varepsilon$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$E'\ \gamma_0$	$\$\$$	$E' \to \varepsilon$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	γ_0	$\$\$$	

FIRST und FOLLOW

Mit Hilfe einer Grammatik kann man Funktionen FIRST und FOLLOW definieren, die die Einträge für die Parse-Tabellen liefern.

Bemerkung

Die von FOLLOW gelieferten Symbole lassen sich als synchronisierende Symbole z.B. bei der Fehlersuche verwenden.

FIRST

Sei $G = (N, T, P, S)$ eine Grammatik, es gilt $\$\$ \notin T$.

FIRST(α) ist definiert für eine beliebige Folge von Grammatiksymbolen $\alpha \in (N \cup T)^*$ und liefert eine Teilmenge der Terminale vereinigt mit dem leeren Wort, $\text{FIRST}(\alpha) \subseteq \Big(T \cup \{\varepsilon\}\Big)$.

FIRST(α) liefert die Menge aller Terminale mit denen ein aus α abgeleiteter

Satz beginnen kann.

$$\text{FIRST}(\alpha) := \left\{ a \in T \mid \alpha \overset{G}{\Rightarrow} a\beta \text{ mit } \beta \in T^* \right\}$$

FIRST(α) enthält zusätzlich ε, wenn ε aus α abgeleitet werden kann, $\alpha \overset{G}{\Rightarrow} \varepsilon$.

FOLLOW

Sei $G = (N, T, P, S)$ eine Grammatik, es gilt \$\$ $\notin T$.

FOLLOW(A) ist definiert für jedes Nichtterminal $A \in N$ und liefert eine Teilmenge der Terminale vereinigt mit dem Endsymbol der Eingabe, $\text{FOLLOW}(A) \subseteq \left(T \cup \{\$\$\}\right)$.

FOLLOW(A) liefert die Menge aller Terminale, die in einer Satzform direkt rechts neben A stehen können.

$$\text{FOLLOW}(A) := \left\{ a \in T \mid S \overset{G}{\Rightarrow} \alpha A a \beta \text{ mit } \alpha, \beta \in (N \cup T)^* \} \right\}$$

Bemerkung

Zwischen A und a können während der Ableitung Grammatiksymbole gestanden haben, die aber verschwunden sind, weil ε aus ihnen abgeleitet wurde.

FOLLOW(A) enthält zusätzlich \$\$, wenn es eine Satzform gibt in der A das am weitesten rechts stehende Grammatiksymbol ist.

Berechnung von FIRST

Für alle Terminale $t \in T$ gilt $\text{FIRST}(t) := \{t\}$.

Für das leere Wort ε gilt $\text{FIRST}(\varepsilon) := \{\varepsilon\}$.

Für ein Nichtterminal $A \in N$ wird für jede Produktion $A \to \alpha$ die Menge FIRST(α) zu FIRST(A) hinzugefügt.

Bemerkung

Enthalten ist folgender Spezialfall.

Gibt es eine Produktion $A \to \varepsilon$, dann füge ε zu FIRST(A) hinzu.

FIRST(α) wird für $\alpha = \alpha_1\alpha_2 \dots \alpha_n$ mit $\alpha_i \in (N \cup T)$ wie folgt bestimmt.

- Zu FIRST(α) wird FIRST(α_1)$\backslash\{\varepsilon\}$ hinzugefügt.
- Zu FIRST(α) wird für $i = 2, \dots n$ die Menge FIRST(α_i)$\backslash\{\varepsilon\}$ hinzugefügt, wenn $\varepsilon \in$ FIRST(α_j) für alle $j = 1, \dots, i-1$, denn d.h. $\alpha_1 \dots \alpha_{i-1} \overset{G}{\Rightarrow} \varepsilon$.
- Zu FIRST(α) wird ε hinzugefügt, wenn ε in allen FIRST(α_1), ..., FIRST(α_n) enthalten ist.

Berechnung von FOLLOW

Für alle Nichterminale $A \in N$ werden die folgenden Regeln solange angewandt, bis keine FOLLOW-Menge mehr vergrößert werden kann.

- In FOLLOW(S) wird \$\$ aufgenommen, wobei S das Startsymbol und \$\$ die Endemarkierung der Eingabe ist.
- Wenn es eine Produktion $A \to \alpha B \beta$ gibt, wird jedes Element von FIRST(β) mit Ausnahme von ε in FOLLOW(B) aufgenommen.
- Wenn es Produktionen $A \to \alpha B$ gibt, dann wird jedes Element von FOLLOW(A) zu FOLLOW(B) hinzugefügt.
- Wenn es eine Produktion $A \to \alpha B \beta$ gibt und $\varepsilon \in$ FIRST(β) enthalten (d.h. $\beta \overset{G}{\Rightarrow} \varepsilon$), dann wird jedes Element von FOLLOW(A) zu FOLLOW(B) hinzugefügt.

Konstruktion der Parse-Tabelle

Angenommen $A \in N$ ist das zu ersetzende Nichterminal und $t \in T \cup \{\$\$\}$ das aktuelle Eingabezeichen.

Der Parser sucht eine Produktion $A \to \alpha$ durch deren rechte Seite A ersetzt werden kann.

- Das geht wenn $t \in$ FIRST(α) gilt.
- Das geht wenn $\alpha = \varepsilon$ oder $\alpha \overset{G}{\Rightarrow} \varepsilon$ und $t \in$ FOLLOW(A) gilt.

Zur Konstruktion der Parse-Tabelle führe für jede Produktion $A \to \alpha$ folgende Schritte durch.

- Trage für jedes Terminal $t \in \text{FIRST}(\alpha)$ die Produktion $A \to \alpha$ an der Stelle $M[A, t]$ ein.
- Gilt $\varepsilon \in \text{FIRST}(\alpha)$ trage $A \to \alpha$ für jedes Terminal $t \in \text{FOLLOW}(A)$ an der Stelle $M[A, t]$ ein.
- Gilt $\varepsilon \in \text{FIRST}(\alpha)$ und \$\$ $\in \text{FOLLOW}(A)$ trage $A \to \alpha$ an der Stelle $M[A, \$\$]$ ein.

Parse-Tabellen

Mit Hilfe von FIRST und FOLLOW läßt sich grundsätzlich für jede Grammatik eine Parse-Tabelle erstellen.

Es gibt Grammatiken, bei denen die Parse-Tabelle mehrere Einträge pro Zelle enthält. Z.B. wenn die Grammatik linksrekursiv ist oder es gemeinsame Präfixe gibt.

Eine **LL(1)-Grammatik** ist eine Grammatik deren Parse-Tabelle keine Mehrfacheinträge enthält, d.h. es gibt einen LL(1)-Parser, der genau die Sätze dieser Grammatik erkennt. Z.B. den nicht-rekursiven Parser mit der aus FIRST und FOLLOW erzeugten Parse-Tabelle.

Beispiel

Grammatik $G = (N, T, P, S)$

$N = \{E, E', F, T, T'\}, \quad T = \{\mathbf{id}, +, *, (,)\}, \quad S = E$

$P = \{$

$$\begin{aligned} E &\to TE' \\ E' &\to +TE' \mid \varepsilon \\ T &\to FT' \\ T' &\to *FT' \mid \varepsilon \\ F &\to (\,E\,) \mid \mathbf{id} \end{aligned}$$

$\}$

Ziel. Für jede Produktion Bestimmung der FIRST-Menge der rechten Seite.

$\text{FIRST}(\varepsilon) = \{\varepsilon\}$
$\text{FIRST}(a) = a$ für alle $a \in T$

$\text{FIRST}(\,(\,E\,)\,) = \{\,(\,\}$
$\text{FIRST}(*FT') = \{*\}$
$\text{FIRST}(+TE') = \{+\}$
$\text{FIRST}(*FT') = \{*\}$

$\text{FIRST}(FT')$?
- $\text{FIRST}(FT') \supset \text{FIRST}(F)$
- $\text{FIRST}(F)$?
  - $\text{FIRST}(F) \supset \text{FIRST}(\,(\,E\,)\,) = \{\,(\,\}$
  - $\text{FIRST}(F) \supset \text{FIRST}(\mathbf{id}) = \{\mathbf{id}\}$
  - es gibt keine weitere Produktion $F \to \ldots$
  - d.h. $\text{FIRST}(F) = \{\,(\,, \mathbf{id}\,\}$
- $\text{FIRST}(FT') \supset \text{FIRST}(F) = \{\,(\,, \mathbf{id}\,\}$
- $\varepsilon \notin \text{FIRST}(F)$
- d.h. $\text{FIRST}(FT') = \{\,(\,, \mathbf{id}\,\}$

$\text{FIRST}(TE')$?
- $\text{FIRST}(TE') \supset \text{FIRST}(T)$
- $\text{FIRST}(T)$?
  - $\text{FIRST}(T) \supset \text{FIRST}(FT') = \{\,(\,, \mathbf{id}\,\}$
  - es gibt keine weitere Produktion $T \to \ldots$
  - d.h. $\text{FIRST}(T) = \{\,(\,, \mathbf{id}\,\}$
- $\text{FIRST}(TE') \supset \text{FIRST}(T) = \{\,(\,, \mathbf{id}\,\}$
- $\varepsilon \notin \text{FIRST}(T)$
- d.h. $\text{FIRST}(TE') = \{\,(\,, \mathbf{id}\,\}$

Ziel. Für jede Produktion $A \to \alpha$ mit $\varepsilon \in \text{FIRST}(\alpha)$ Bestimmung von FOLLOW(A), d.h. gesucht ist FOLLOW(E') und FOLLOW(T').

FOLLOW(E')?
- $E' \to +TE'$
- d.h. FOLLOW(E') $\supset$ FOLLOW(E')
- $E \to TE'$
- d.h. FOLLOW(E') $\supset$ FOLLOW(E)
- FOLLOW(E)?
  - $S = E$
  - d.h. FOLLOW(E) $\ni$ \$\$
  - $F \to (\ E\ )$
  - d.h. FOLLOW(E) $\supset$ FIRST( ))$\backslash\{\varepsilon\} = \{\ )\ \}$
  - keine weitere rechte Seite $\ldots E \ldots$
  - d.h. FOLLOW(E) $= \{\ ),\$\$\ \}$
- FOLLOW(E') $\supset$ FOLLOW(E) $= \{\ ),\$\$\ \}$
- keine weitere rechte Seite $\ldots E' \ldots$
- d.h. FOLLOW(E') $= \{\ ),\$\$\ \}$

FOLLOW(T')?
- $T' \to *FT'$
  - d.h. FOLLOW(T') $\supset$ FOLLOW(T')
- $T \to FT'$
- d.h. FOLLOW(T') $\supset$ FOLLOW(T)
- FOLLOW(T)?
  - $E \to TE'$
  - d.h. FOLLOW(T) $\supset$ FIRST(E')$\backslash\{\varepsilon\}$
  - FIRST(E')?
    - FIRST(E') $= \{+, \varepsilon\}$
  - FOLLOW(T) $\supset$ FIRST(E')$\backslash\{\varepsilon\} = \{+\}$
  - $E \to TE'$ und $\varepsilon \in$ FIRST(E')
  - d.h. FOLLOW(T) $\supset$ FOLLOW(E) $= \{\ ),\$\$\ \}$
  - $E' \to +TE'$
  - genauso, weil FOLLOW(E') = FOLLOW(E)
  - d.h. FOLLOW(T) $= \{+,\ ),\$\$\ \}$
- FOLLOW(T') $\supset$ FOLLOW(T) $= \{+,\ ),\$\$\ \}$
- keine weitere rechte Seite $\ldots T' \ldots$
- d.h. FOLLOW(T') $= \{+,\ ),\$\$\ \}$

Ziel. Konstruiere die Parse-Tabelle wie folgt.

- Für $t \in \text{FIRST}(\alpha)$ setze $M[A, t] := A \to \alpha$.
- Gilt $\varepsilon \in \text{FIRST}(\alpha)$ setze $M[A, t] := A \to \alpha$ für jedes $t \in \text{FOLLOW}(A)$.

	id	$+$	$*$	$($	$)$	\$\$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \varepsilon$	$E' \to \varepsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \varepsilon$	$T' \to *FT'$		$T' \to \varepsilon$	$T' \to \varepsilon$
F	$F \to \mathbf{id}$			$F \to (\,E\,)$		