

Prof. Dr. Reinhard Wilhelm
Dipl.-Inform. Andreas Kerren
Fachbereich Informatik
Universität des Saarlandes



Proseminar im Wintersemester 1999/2000

Syntaktische Analyse von Programmiersprachen

Vortrag 3: Einführung in die Syntaxanalyse und Syntaxanalyatoren

- Kathrin Roberts -

Inhaltsangabe:

I) Was ist Parsen?

II) Kellerautomat

III) Linksparser und Rechtsparser

IV) Starke LL(k) Syntaxanalyse

V) Eigenschaften starker LL(k) Grammatiken

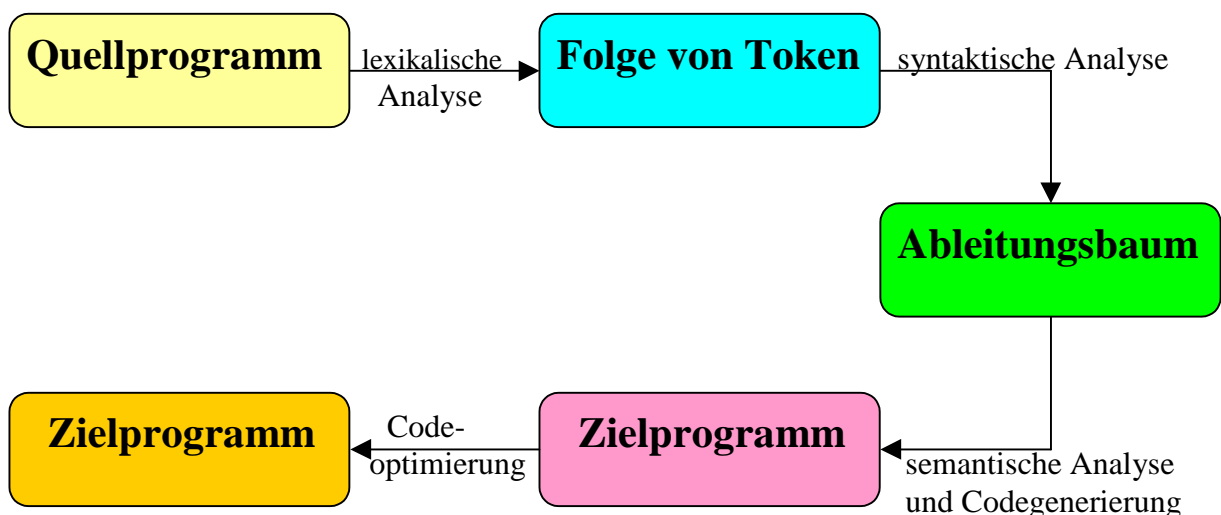
VI) Implementierung eines starken LL(1) - Parsers

D) Was ist Parsen?

Parsen heißt aus dem Englischen übersetzt grammatisch analysieren und spielt beim Compilerbau für Programmiersprachen eine entscheidende Rolle, wie wir später erfahren werden. Da die Syntax von Programmiersprachen mit Hilfe von kontextfreien Grammatiken erzeugt werden, betrachten wir nur das Parsen von kontextfreien Sprachen.

Bei der Programmübersetzung wollen wir das in einer Programmiersprache verfaßte Quellprogramm in ein äquivalentes Zielprogramm umwandeln. Dabei durchläuft das Programm mehrere Phasen:

Als erstes unterzieht man es der lexikalischen Analyse, welche aus dem Quelltext eine Folge von Token hervorbringt. Im zweiten Schritt wird die Folge von Token syntaktisch analysiert, dies ist genau der Teil, den wir in dieser Ausarbeitung behandeln. Die syntaktische Analyse wird durch den Parser durchgeführt. Seine Aufgaben setzen sich zum einen zusammen aus der Überprüfung des zu compilierenden Quellprogrammes auf syntaktische Korrektheit, die aus der kontextfreien Grammatik der Programmiersprache ableitbar ist daher auch die englische Bezeichnung "language recognizer". Zum anderen übernimmt der Parser auch die Aufgabe, den Quelltext in eine Eingabe für den Codegenerator zu übersetzen (text transformer). Diese Eingabe ist nichts anderes als ein erweiterter Ableitungsbaum. Nachdem der Ableitungsbaum noch einer semantischen Analyse unterzogen wurde, erstellt der Codegenerator mit den Ableitungsregeln ein Zielprogramm z.B. in der Maschinsprache Assembler.



Nachdem das vorläufige Zielprogramm erstellt wurde, kann optional noch eine Codeoptimierung durchgeführt werden.

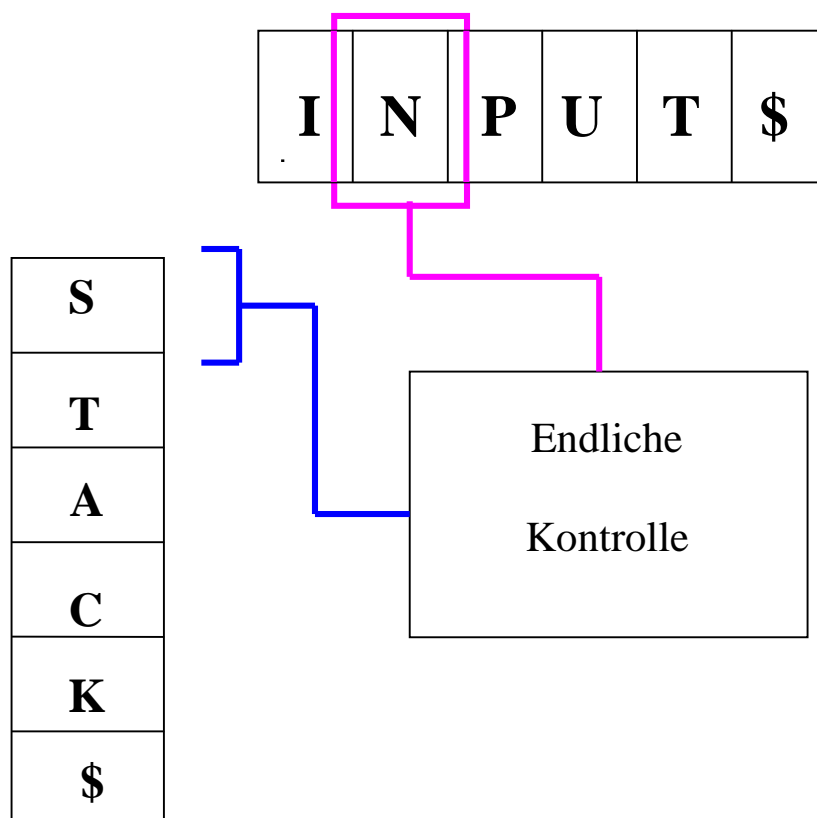
Der Parser ist ein Programm, das die von der kontextfreien Grammatik G erzeugte Sprache $L(G)$ erkennt und für jeden Satz aus der Sprache $L(G)$ einen Ableitungsbaum produziert. Das formale Modell des Parsers ist ein Kellerautomat mit Ausgabe, der genau die kontextfreien Sprachen akzeptiert und einen Homomorphismus besitzt, der die Ableitung des zugrundeliegenden Kellerautomaten ausgibt.

Zunächst aber zur Grundlage des Kellerautomaten.

II) Kellerautomat KA (push down automat)

Der **Kellerautomat** M besteht aus einem 7-Tupel $M=(Q,T,P,\gamma_s,F,\$,|)$, wobei Q das Kelleralphabet ist, T das Eingabealphabet, γ_s der initiale Kellerinhalt mit $\gamma_s \in Q^*$, F die Menge der Endinhalte im Keller mit $F \subseteq Q^*$, $\$$ das Endmarkierungssymbol, $|$ das Trennzeichen mit $\$, | \notin Q \cup T$ und P die Menge der Produktionsregeln, wobei jede Regel von P die Form $\alpha|xy \rightarrow \beta|y$ besitzt mit $\alpha, \beta \in Q^* \cup \Q^* , $x \in T^*$ und $y \in T^* \cup T^*\$$.

Die Konfiguration (momentane Zustandsbeschreibung) ist ein String der Form $\$y|w\$$, wobei $y \in Q^*$ und $w \in T^*$. Das letzte Symbol des auf der linken Seite von $|$ stehenden Kellerstrings $\$y$ wird das oberste Kellersymbol und das erste Symbol des auf der rechten Seite stehenden Eingabebandstrings $w\$$ wird das aktuelle Eingabesymbol genannt. Eine Produktion der Form $\alpha|xy \rightarrow \beta|y$ ist nur auf eine Konfiguration anwendbar, wenn das oberste Symbol des Kellers α und der Anfang des verbleibenden Eingabestrings xy ist. Durch Anwendung dieser Produktion wird α durch β ersetzt und x vom Eingabeband gelöscht.



Der Kellerautomat ist initialisiert, wenn auf dem Eingabeband das Eingabewort w steht und der Keller mit γ_s gefüllt ist, er akzeptiert ein Wort, wenn er nach

seinen Berechnungen ein leeres Eingabeband aufweist, also $w = \epsilon$, und im Keller ein γ steht, das in der Menge der Endinhalte enthalten ist.

Ist auf eine Konfiguration keine Produktionsregel mehr anwendbar, so spricht man von einer Fehlerkonfiguration. Die vom Kellerautomaten akzeptierte Sprache $L(M)$ ist demnach definiert als:

$$L(M) = \{ w \in T^* \mid \exists \gamma \in F \mid w \Rightarrow^* \gamma \mid \$ \text{ in } M \text{ f\"ur } \gamma \in F \}$$

Beispiel für einen Kellerautomaten M_1 , der die Sprache $L(M_1) = \{0^n 1^n \mid n \geq 0\}$ akzeptiert:

$$M_1 = (\{0,1,c\}, \{0,1\}, P, \epsilon, \{c\}, \$, |)$$

wobei die Menge P der Produktionsregeln enthält:

- 1.) $\$|0 \rightarrow \$0|$, 2.) $0|0 \rightarrow 00|$,
- 3.) $0|1 \rightarrow 0c|$, 4.) $0c|1 \rightarrow c|$,
- 5.) $\$|\$ \rightarrow \$c|\$$

Nun ein paar Konfigurationsübergänge bei verschiedenen n :

$$n = 0:$$

$$\$|\$ \Rightarrow \$c|\$$$

$$n > 0:$$

$$\$|0^n 1^n \$ \Rightarrow^n \$0^n | 1^n \$ \Rightarrow \$0^n c | 1^n \$ \Rightarrow^n \$c|\$$$

Die jeweilige Endkonfiguration $\$c|\$$ enthält auf der rechten Seite ein leeres Eingabeband ($w = \epsilon$) und ein verbleibender Kellerstring c mit $c \in \{c\}$ (entspricht der Menge der Endinhalte von M_1) auf der linken Seite, somit akzeptiert der Kellerautomat M_1 die Wörter der Sprache $L(M_1)$.

Ein Kellerautomat ist **mehrdeutig**, wenn es mindestens zwei unterschiedliche akzeptierende Berechnungen für eine Eingabe gibt, sonst ist er **eindeutig**.

Ein Kellerautomat ist **nichtdeterministisch** genau dann, wenn es eine Konfiguration gibt, auf die zwei Konfigurationen anwendbar sind. Dies ist z.B. genau dann der Fall, wenn es verschiedene Produktionsregeln $\alpha|x \rightarrow \alpha'|x'$ und $\beta|y \rightarrow \beta'|y'$ gibt, wobei x Präfix von y und α Suffix von β ist.

Nun werde ich beweisen, dass ein Kellerautomat kontextfreie Sprachen akzeptiert und dass jede Sprache, die ein Kellerautomat akzeptiert, kontextfrei ist. Dies ist eine wichtige Grundlage, um später kontextfreie Sprachen analysieren zu können.

Satz: Eine Sprache ist kf. \Leftrightarrow sie wird von einem KA akzeptiert

“ \Rightarrow “

Um diese Richtung des Satzes zu beweisen, konstruiere ich eine **Predictive Machine**, die einen Kellerautomaten darstellt, der genau die kontextfreie Sprache der gegebenen kontextfreien Grammatik akzeptiert:

Sei also $G = (V, T, P, S)$ kontextfrei. Dann werden die Produktionsregeln des Kellerautomaten $M = (V, T, P_m, S, \{\epsilon\}, \$, |)$ wie folgt definiert:

P_m : (pa) $A| \rightarrow \omega^R|$ für jede Ableitung $A \rightarrow \omega \in P$
 (sa) $a|a \rightarrow |$ für jedes $a \in T$

wobei (pa) für produce action und (sa) für shift-action steht. Bei der (pa) wird die rechte Seite einer Produktionsregel, die das Nichtterminal A in der Grammatik ableitet umgekehrt in den Keller geschrieben (ist mit R gekennzeichnet). Falls also das oberste Symbol des Kellers A ist, geht der Kellerautomat in die nächste Konfiguration, indem er das abgeleitete Wort von A umgekehrt in den Keller schreibt. Bei der (sa) wird das oberste Kellersymbol a gepoppt, wenn das aktuelle Eingabesymbol auch das Terminal a ist.

Der Kellerautomat akzeptiert also ein Wort w der Sprache $L(G)$, in dem er die Ableitung der Grammatik G für dieses Wort simuliert. Der Buchstabe eines Wortes ist dann akzeptiert, wenn er aus dem Keller gepoppt wird.

Beispiel für die Predictive Machine:

Die gegebene Grammatik sei $G_{0^*1^n} = (\{0,1,S\}, \{0,1\}, \{S \rightarrow \epsilon | 0S1\}, S)$, so dass der Kellerautomat $M = (\{0,1,S\}, \{0,1\}, P_m, S, \{\epsilon\}, \$, |)$ definiert werden kann. P_m besteht aus folgenden Produktionen:

$r_1 = S| \rightarrow |$,
 $r_2 = S| \rightarrow 1S0|$,
 $r_3 = 0|0 \rightarrow |$,
 $r_4 = 1|1 \rightarrow |$,

Die Konfigurationsübergänge bei verschiedenen Wörtern w

$w = \epsilon$: $\$S|\$ \rightarrow \$|\$$,

$w = 0^n 1^n$: $\$S|0^n 1^n\$ \Rightarrow \$1S0|0^n 1^n\$ \Rightarrow^n \$1^n S|1^n\$ \Rightarrow \$1^n|1^n\$ \Rightarrow^n \$|\$$

Zurück zum Beweis:

$\forall w \in T^*$ gilt :

$$\begin{aligned} w \in L(G) &\Leftrightarrow \exists S \Rightarrow^* w \\ &\Leftrightarrow \exists \text{Folge von Konfigurationen von } M \\ &\quad \$S|w$, \dots, \$|\$ \\ &\Leftrightarrow w \in L(M) \end{aligned}$$

“ \Leftarrow “

Bei der Rückrichtung des Beweises wird eine kontextfreie Grammatik konstruiert, die die Sprache eines gegebenen Kellerautomaten erzeugt.

Sei $L = L(M)$ für den Kellerautomaten $M = (Q, T, P_m, \gamma_s, F, \$, |)$. Für jede Produktionsregel soll gelten, dass sie sich in der folgenden Normalform befindet:

$\alpha|x \rightarrow \beta|$ mit $|\alpha| \leq 2$ und $|x| \leq 1$.

Dabei sei $\alpha \in Q^* \cup \$Q^*$, $\beta \in T^* \cup T^*\$$ und $x \in T$.

Im folgenden ist A_i ein Buchstabe von dem String α .

Wäre $\alpha = A_1 A_2 \dots A_k$ mit $k > 2$, dann kann man die Produktionsregel von P_m ($A_1 A_2 \dots A_k |x \rightarrow \beta|$) unter Einhaltung der Normalform umformen in:

$$\begin{aligned} A_{k-1} A_k |x &\rightarrow A_{k-1}| \\ A_{k-2} A_{k-1} |x &\rightarrow A_{k-2}| \\ &\vdots \\ A_2 A_1 |x &\rightarrow A_1| \\ A_1 |x &\rightarrow \beta| \end{aligned}$$

Nun zur Konstruktion von G :

Die Grammatik simuliert die Rechenschritte des Kellerautomaten M durch Linksableitungsschritte. Die Variablen von G haben zwei Bestandteile:

- 1.) die Konfiguration vor einer Folge von Rechenschritten für ein bestimmtes zu verarbeitendes Eingabesymbol
- 2.) Die Nachfolgekonfiguration von der unter 1.) angeführten Konfiguration

Sei $G = (V, T, P, S)$

mit $V = \{S\} \cup \{ [(\$ \gamma_a | \omega_a \$), (\$ \gamma_b | \omega_b \$)] \mid \gamma_a, \gamma_b \in Q^*; \omega_a, \omega_b \in T^* \}$

Kurz noch ein paar Definitionen zu den im folgenden verwendeten Variablen:

$\gamma_s = \gamma_1 \gamma_2 \dots \gamma_m$ $\gamma = \gamma_1 \dots \gamma_i$ $i \leq m$ $\gamma, \gamma_s \in Q^*$

unter anderem folgt damit auch $\gamma_s = \gamma \gamma_{i+1} \dots \gamma_m$

$w = w_1 w_2 \dots w_n$ $E \in T$

$\delta_i = \gamma_1 \dots \gamma_i$ $i \in \mathbb{N}$ $r_j = w_j w_{j+1} \dots w_n$ $\delta \in Q^*$ $r \in T^*$

Die Menge der Produktionsregeln setzt sich zusammen aus:

$$\begin{aligned}
 P = & \{ S \rightarrow [(\$ \gamma_s | w \$) , (\$ E | \$)] \mid \text{für } (\gamma_s | w \rightarrow E |) \in P_m \} \\
 \cup & \{ S \rightarrow [(\$ \gamma \alpha | w_1 w_2 \dots w_n \$) , (\$ \gamma \beta | w_2 \dots w_n \$)] \mid \text{für } (\alpha | w_1 \rightarrow \beta |) \in P_m \} \\
 \cup & \{ S \rightarrow [(\$ \gamma \alpha | w \$) , (\$ \gamma \beta | w \$)] \mid \text{für } (\alpha | \rightarrow \beta |) \in P_m \} \\
 \cup & \{ [(\$ \delta_i \alpha | r_j \$) , (\$ \delta_i \beta | r_{j+1} \$)] \\
 & \quad \rightarrow w_j [(\$ \delta_i \beta | r_{j+1} \$) , (\$ \delta | r \$)] \mid \text{für } (\alpha | r_j \rightarrow \beta |) \in P_m \} \\
 \cup & \{ [(\$ \delta_i \alpha | r_j \$) , (\$ \delta_i \beta | r_j \$)] \\
 & \quad \rightarrow [(\$ \delta_i \beta | r_j \$) , (\$ \delta | r \$)] \mid \text{für } (\alpha | \rightarrow \beta |) \in P_m \} \\
 \cup & \{ [(\$ \gamma_i | w_n \$) , (\$ E | \$)] \rightarrow w_n \mid \text{für } (\gamma_i | w_n \rightarrow E |) \in P_m \}
 \end{aligned}$$

Die entstandene Grammatik ist kontextfrei, so dass die Voraussetzungen für den Beweis gegeben sind.

Behauptung :

$$\begin{aligned}
 \forall \text{ Teilwörter } x \in T^* \text{ gilt: } & \quad x = x_1 x_2 \dots x_n \\
 [(\$ \delta_v | x r \$) , (\$ \delta_w | x_2 \dots x_n r \$)] \Rightarrow^* & x \\
 \Leftrightarrow (\$ \delta_v | x r \$) , (\$ \delta_w | x_2 \dots x_n r \$) , \dots , & (\$ \delta | r \$)
 \end{aligned}$$

Beweis:

$$\begin{aligned}
 \text{Für } a \in T \cup \{ \epsilon \} \text{ gilt:} \\
 [(\$ \delta \alpha | a r \$) , (\$ \delta \beta | r \$)] \Rightarrow a \\
 \Leftrightarrow ([(\$ \delta \alpha | a r \$) , (\$ \delta \beta | r \$)] \Rightarrow a [(\$ \delta \beta | r \$) , (\$ \delta_i r_j \$)]) \in P \\
 \Leftrightarrow (\alpha | a \rightarrow \beta |) \in P_m \\
 \Leftrightarrow \exists \text{ Folge von Konfigurationen } (\$ \delta \alpha | a r \$) , (\$ \delta \beta | r \$)
 \end{aligned}$$

Der weitere Beweis, auf den ich nicht näher eingehe, wird mittels Induktion durchgeführt. Die Hinrichtung geht dabei über die Länge der Linksableitung k , die Rückrichtung über die Anzahl der Rechenschritte n .

Es folgt danach die Behauptung $L(M) = L(G)$:

$$\begin{aligned}
 w \in L(M) & \Leftrightarrow \$ \gamma_s | w \$, \dots , \$ E | \$ \\
 & \Leftrightarrow S \Rightarrow [(\$ \gamma \alpha | w_1 w_2 \dots w_n \$) , (\$ \gamma \beta | w_2 \dots w_n \$)] \Rightarrow^* w \\
 & \Leftrightarrow w \in L(G)
 \end{aligned}$$

Im übrigen sei noch gesagt, dass eine Sprache **deterministisch** ist, wenn sie von einem deterministischen Kellerautomaten akzeptiert wird.

III) Linksparser und Rechtsparser

Bevor wir zur Definition eines Parsers kommen gebe ich noch kurz eine Grundlage an.

Sei $G = (V, T, P, S)$ und $w \in L(G)$.

Eine Folge von Ableitungsregeln $\pi \in P^*$ ist eine **Linkableitung** eines Wortes w in G , wenn bei jedem Ableitungsschritt eine Produktionsregel auf die linksstehende Variable angewendet wird.

Eine Folge von Ableitungsregeln $\pi^R \in P^*$ ist eine **Rechtsableitung** eines Wortes w in G , wenn bei jedem Ableitungsschritt eine Produktionsregel auf die rechtsstehende Variable angewendet wird. π^R ist die Umkehrung von π .

Nun die Definition der Parser:

Ein RAM¹-Programm ist ein **Linksparser** für eine Grammatik G , wenn es die Sprache $L(G)$ erkennt und für jeden Satz in G eine Linksableitung erzeugt.

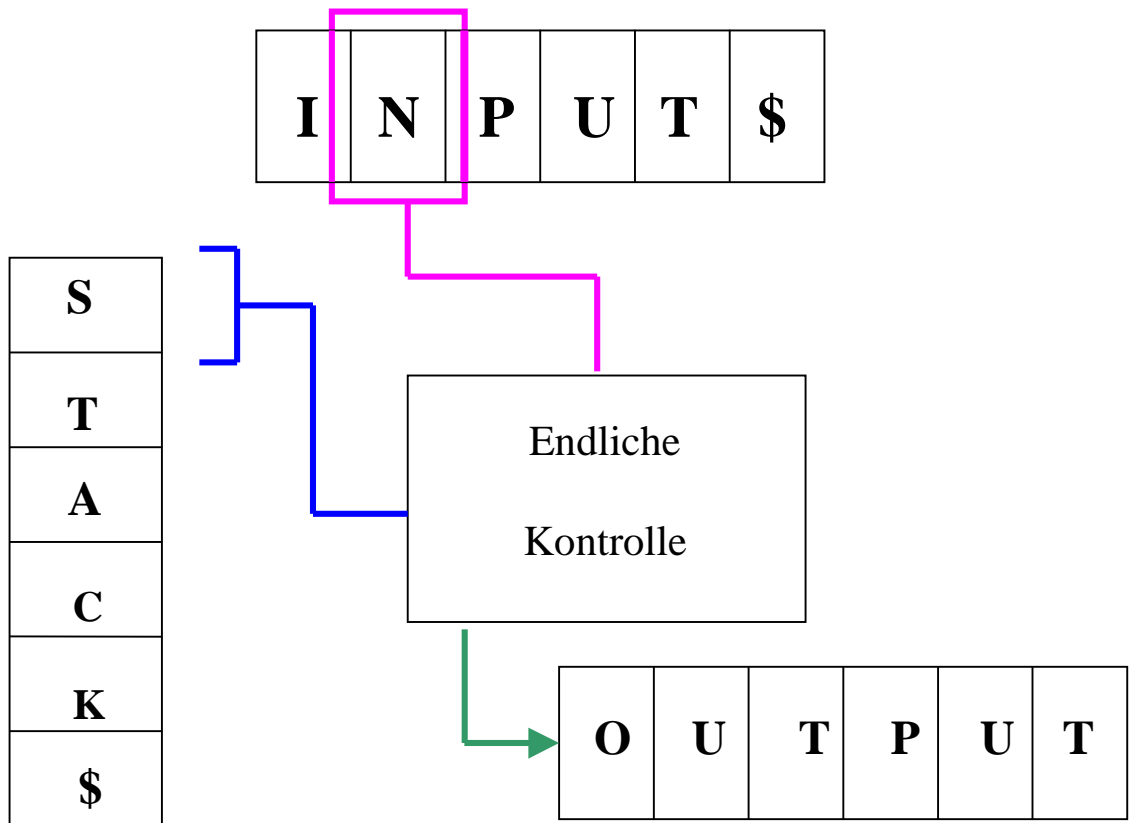
Analog natürlich die Definition des **Rechtsparsers**.

Beide produzieren also Ableitungsbäume für jeden Satz in $L(G)$, wobei man sie jeweils an der Art der Erstellung des Ableitungsbaumes unterscheiden kann. Beim Linksparser wird der Ableitungsbaum von oben nach unten aufgebaut, daher der Begriff **top-down parser** aus dem Englischen. Der Rechtsparser wird **bottom-up parser** genannt, da er den Ableitungsbaum von unten her aufbaut.

Im folgenden wird der Kellerautomat mit Ausgabe definiert, um das formale Modell des Parsers beschreiben zu können.

Der Kellerautomat muß eine Ausgabereinheit besitzen, um die jeweilige Ableitung eines Wortes ausgeben zu können, welche durch den unten angegebenen Homomorphismus τ realisiert ist.

¹ RAM: random access machine, die RAM ist ein formales Berechnungsmodell, das sich von einer realen Maschine dadurch unterscheidet, dass es keine Ressourcenlimitierung (wie z.B. Speicherplatz) gibt.



Definition des **Kellerautomaten mit Ausgabe** (M, τ) (pushdown transducer):

Sei Δ Ausgabealphabet und τ die Ausgabefunktion bzw. der Homomorphismus $\tau: P^* \rightarrow \Delta^*$. Sei w ein Satz in $N(M)$ und π' eine Folge von Produktionsregeln. π' sei dann eine Ableitung von w , wenn ein mit w initialisierter Kellerautomat eine akzeptierende Berechnung hervorruft, bei der π' verwendet wird. Ein Kellerautomat mit Ausgabe produziert eine Ausgabe π für ein Wort w , wenn gilt: $\tau(\pi') = \pi$ für eine Ableitung π' von w .

Definition:

Der Kellerautomat mit Ausgabe ist ein **Linksparser** (produce-shift parser) für eine Grammatik G , wenn folgendes erfüllt ist:

- (1) Eingabealphabet von $M =$ Terminalalphabet von G
- (2) $N(M) = L(G)$
- (3) Ausgabealphabet von $M =$ Menge der Produktionsregeln von G
- (4) Ausgabe von M für w ist eine Linksableitung

Der Kellerautomat ist wie die Predictive Machine aus dem vorherigen Kapitel definiert:

Sei also $G = (V, T, P, S)$ die gegebene kontextfreie Grammatik.
 $M = (V, T, P_1, S, \{\epsilon\}, \$, |)$ mit den entsprechenden Produktionen in P_1 .

$$P_1: \quad (pa) A| \rightarrow \omega^R| \quad \text{für } A \rightarrow \omega \in P$$

$$\quad (sa) a|a \rightarrow | \quad \text{für } a \in T$$

Die Ausgabefunktion von M ist demnach definiert als:

$$\tau(A| \rightarrow \omega^R|) = A \rightarrow \omega$$

$$\tau(a|a \rightarrow |) = \epsilon$$

- **Beispiel mit Grammatik $G_{0^n 1^n}$:**

Im folgenden sind die Produktionen P_1 des Kellerautomaten mit der entsprechend definierten Ausgabefunktion angegeben

$$r_1 = S| \rightarrow |, \quad \tau(r_1) = S \rightarrow \epsilon$$

$$r_2 = S| \rightarrow 1S0|, \quad \tau(r_2) = S \rightarrow 0S1$$

$$r_3 = 0|0 \rightarrow |, \quad \tau(r_3) = \epsilon$$

$$r_4 = 1|1 \rightarrow |, \quad \tau(r_4) = \epsilon$$

Die Ausgabe des Kellerautomaten mit Ausgabe bei verschiedenen Wörtern w

$$w = \epsilon \quad \tau(r_1) = S \rightarrow \epsilon$$

$$w = 0011 \quad \tau(r_2 r_3 r_2 r_3 r_1 r_4 r_4) = (S \rightarrow 0S1) (S \rightarrow 0S1) (S \rightarrow \epsilon)$$

$$w = 0^n 1^n \quad \tau((r_2 r_3)^n r_1 r_4^n) = (S \rightarrow 0S1)^n (S \rightarrow \epsilon), \text{ mit } n \geq 0$$

- Analog zum Linksparser ist auch der **Rechtsparser** (shift-reduce parser) definiert. Wobei aber Punkt (4) variiert wird, so dass M für w eine Rechtsableitung ausgibt. Daher ist der Kellerautomat mit Ausgabe (M, τ) anders strukturiert als der Linksparser.

Sei $G = (V, T, P, S)$ und $M = (V, T, P_r, \epsilon, \{S\}, \$, |)$

$$\text{mit } P_r: \quad (ra) \omega| \rightarrow A| \quad \text{für } A \rightarrow \omega \in P$$

$$\quad (sa) |a \rightarrow a| \quad \text{für } a \in T$$

und die Ausgabefunktion wird dargestellt als:

$$\tau(\omega| \rightarrow A|) = A \rightarrow \omega$$

$$\tau(|a \rightarrow a|) = \epsilon$$

Beispiel mit der Grammatik $G_{0^n 1^n}$:

Die Produktionsregeln für einen Rechtsparser und die Ausgabefunktion sind definiert als:

$$\begin{aligned} r_1 &= | \rightarrow S |, & \tau(r_1) &= S \rightarrow \epsilon \\ r_2 &= 0S1 | \rightarrow S |, & \tau(r_2) &= S \rightarrow 0S1 \\ r_3 &= |0 \rightarrow 0|, & \tau(r_3) &= \epsilon \\ r_4 &= |1 \rightarrow 1|, & \tau(r_4) &= \epsilon \end{aligned}$$

Die Ausgabe bei bestimmten Wörtern w :

$$\begin{aligned} w &= \epsilon & \tau(r_1) &= S \rightarrow \epsilon \\ w &= 0011 & \tau(r_3 r_3 r_1 r_4 r_2 r_4 r_2) &= (S \rightarrow \epsilon) (S \rightarrow 0S1) (S \rightarrow 0S1) \\ w &= 0^n 1^n & \tau(r_3^n r_1 (r_4 r_2)^n) &= (S \rightarrow \epsilon) (S \rightarrow 0S1)^n, \quad n \geq 0 \end{aligned}$$

Nach der derzeitigen Definition der Rechts- und Linksparser kann man bei Anwendung von Produktionsregeln auf Ausführungskonflikte stoßen.

Zum Beispiel beim Linksparser der produce-produce conflict: mit den gegebenen Produktionsregeln $A | \rightarrow \omega_1 |$ und $A | \rightarrow \omega_2 |$. Bei diesem Konflikt können auf das Nichtterminal A , das als oberstes Symbol im Keller liegt, zwei Produktionsregeln angewendet werden.

Beim Rechtsparser ist es einmal der shift-reduce conflict, mit folgenden gegebenen Produktionsregeln: $\omega | \rightarrow A |$ und $|a \rightarrow a|$, denn bei Anwendung auf die Konfiguration $\$ \gamma \omega | a \gamma \$$ hat man ebenfalls verschiedene Möglichkeiten:

Einerseits das a zu shiften, andererseits zu A zu reduzieren. Zum anderen kann der reduce-reduce conflict vorkommen, wenn die folgenden Produktionsregeln gegeben sind: $\omega_1 | \rightarrow A_1 |$ und $\omega_2 | \rightarrow A_2 |$ mit ω_1 Suffix von ω_2 . Es stellt sich die Frage, ob der Suffix oder ω_2 weiter abgeleitet werden soll.

Der hier beschriebene Nichtdeterminismus darf bei einem Parser für den Einsatz in der Praxis nicht vorkommen. Man muß die Sicherheit haben, dass er auf allen Eingaben hält und immer das richtige Ergebnis liefert. Um diesen auftretenden Nichtdeterminismus zu bekämpfen wird im nächsten Kapitel das lookahead-Symbol eingeführt. Da der Linksparser mit Hilfe von lookaheads deterministisch wird, kann die Definition des Linksparsers (man spricht später von einem Starken LL(k) Parser) verschärft werden.

IV) Starke LL(k)-Syntaxanalyse

Zuvor noch ein paar Grundlagen zur Definition eines Starke LL(k)-Parsers :

Sei $G = (V, T, P, S)$ eine Grammatik, $k \in \mathbb{N}$, $\gamma \in V^*$.

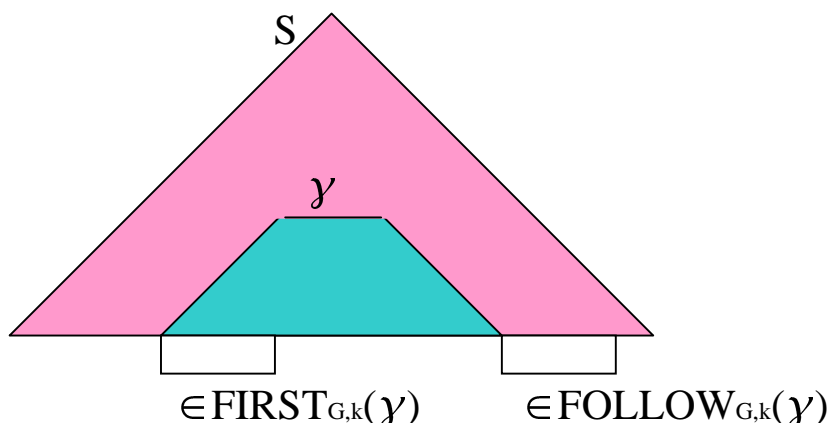
Dann kann man zwei Mengen aus der Sprache der Grammatik definieren:

1.) $\text{FIRST}_{G,k}(\gamma) = k:\text{L}_G(\gamma)$

Diese Menge spiegelt genau die Menge der k-Präfixe (= Präfixe der Länge k) der Terminalwörter wider, die von dem String γ abgeleitet werden können.

2.) $\text{FOLLOW}_{G,k}(\gamma) = \{ y \in T^* \mid S \Rightarrow^* \alpha \gamma \beta \text{ in } G \text{ und } y \in \text{FIRST}_{G,k}(\beta) \}$
 $\alpha, \beta \in V^*$

Diese Menge ist die Menge der k-Präfixe des Terminalwortes, die nach der Ableitung des Strings γ abgeleitet werden können.



Zum besseren Verständnis siehe bitte obige Abbildung, die einen Ableitungsbaum zeigt, der bei dem Startsymbol S beginnt und den oben angesprochenen String γ enthält.

Ein **starker LL(k)-Parser (SLL(k)-Parser)** für eine Grammatik $G = (V, T, P, S)$ ist ein Kellerautomat (M, τ) mit Ausgabe, der wie folgt definiert ist:

$M = (V, T, P_m, S, \{\epsilon\}, \$, |)$

Die Menge der Produktionsregeln P_m bestehen aus:

(pa) $A | \gamma \rightarrow \omega^R | \gamma$ für ein $A \rightarrow \omega \in P$

(sa) $a | a \rightarrow |$ $a \in T$,

$\gamma \in \text{FIRST}_k(\omega \text{ FOLLOW}_k(A))$ ist das lookahead-Symbol, welches neu eingeführt wurde

und die Ausgabefunktion τ :

(1) $\tau(A | \gamma \rightarrow \omega^R | \gamma) = A \rightarrow \omega$

(2) $\tau(a | a \rightarrow |) = \epsilon$

Noch einige Erläuterungen zum lookahead-Symbol:

Mit dem lookahead-Symbol wird bestimmt, welches Eingabesymbol auf dem Eingabeband als nächstes zu stehen hat, damit man eine bestimmte Produktion anwenden kann.

Beispiel:

Es sei in P_m gegeben:

a) $A|a \rightarrow \omega^R|a$ für $A \rightarrow \omega \in P$

b) $a|a \rightarrow |$ für $a \in T$

c) $a \in \text{FIRST}_k(\omega \text{ FOLLOW}_k(A))$

Die Benutzung des lookaheads ist sehr nützlich, z.B. bei 2 verschiedenen Produktionsregeln $A|a \rightarrow w_1|a$ $A|b \rightarrow w_2|b$, d.h. dass das Nichtterminal A in der Grammatik entweder nach w_1 oder w_2 abgeleitet werden kann. Damit der SLL(k)-Parser erkennt, welche Produktion er anwenden soll, braucht er nur zu untersuchen, was als aktuelles Eingabesymbol auf dem Eingabeband steht. Ist es a, dann wird $A \rightarrow w_1$ abgeleitet, ist es b, dann wird $A \rightarrow w_2$ abgeleitet. Man merkt, dass das lookahead-Symbol hier zur eindeutigen Zuordnung der Produktion auf einer gegebenen Konfiguration verhilft und eventuell auftretenden Nichtdeterminismus der Grammatik beseitigt.

Beispiele für lookahead – Auswertung der Grammatik G_{block} :

$G_{\text{block}} = (V, T, P, S)$

P : $S \rightarrow E|B$
 $E \rightarrow \epsilon$
 $B \rightarrow a| \text{begin } S \text{ C end}$
 $C \rightarrow \epsilon | ; S C$

$k=1$:

$\text{FIRST}_1(a \text{ FOLLOW}_1(B)) = \{a\}$

$\text{FIRST}_1(\text{begin } S \text{ C end FOLLOW}_1(B)) = \{\text{begin}\}$

$\text{FIRST}_1(; S C \text{ FOLLOW}_1(C)) = \{;\}$

$\text{FIRST}_1(B \text{ FOLLOW}_1(S)) = \text{FIRST}_1(B) = \{a, \text{begin}\}$

$\text{FIRST}_1(\epsilon \text{ FOLLOW}_1(C)) = \text{FOLLOW}_1(C) = \{\text{end}\}$

$\text{FIRST}_1(\epsilon \text{ FOLLOW}_1(S)) = \text{FOLLOW}_1(S) = \{\text{end}, ;, \$\}$

$\text{FIRST}_1(\epsilon \text{ FOLLOW}_1(E)) = \text{FOLLOW}_1(E) = \{\text{end}, ;, \$\}$

Auswertung eines starken LL(1)-Parsers für die Grammatik G_{block} bei der Beispieleingaben für w :

Produktionsregeln für den Kellerautomaten laut Definition:

$B|a \rightarrow a|a$
 $B|begin \rightarrow end \ C \ S \ begin|begin$
 $C|; \rightarrow C \ S \ ;|;$
 $S|a \rightarrow B|a$
 $S|begin \rightarrow B|begin$
 $C|end \rightarrow |end$
 $S|end \rightarrow E|end$
 $S|; \rightarrow E|;$
 $E|end \rightarrow |end$
 $E|; \rightarrow |;$
 $S|\$ \rightarrow E|\$$
 $E|\$ \rightarrow |\$$

Eingabe: $w = \epsilon$

Auswertung: $\$S|\$ \Rightarrow \$E|\$ \Rightarrow \$|\$$

Ausgabe: $(S \rightarrow E)(E \rightarrow \epsilon)$

Eingabe: $w = \text{begin } a \ ; \ \text{begin } a \ ; \ ; \ \text{end } \text{end}$

Auswertung:

$\$ \ S|begin \ a \ ; \ begin \ a \ ; \ ; \ end \ end \ \$$
 $\$ \ B|begin \ a \ ; \ begin \ a \ ; \ ; \ end \ end \ \$$
 $\$ \ end \ C \ S \ begin|begin \ a \ ; \ begin \ a \ ; \ ; \ end \ end \ \$$
 $\$ \ end \ C \ S|a \ ; \ begin \ a \ ; \ ; \ end \ end \ \$$
 $\$ \ end \ C \ B|a \ ; \ begin \ a \ ; \ ; \ end \ end \ \$$
 $\$ \ end \ C \ a|a \ ; \ begin \ a \ ; \ ; \ end \ end \ \$$
 $\$ \ end \ C \ |; \ begin \ a \ ; \ ; \ end \ end \ \$$
 $\$ \ end \ C \ S \ ;|; \ begin \ a \ ; \ ; \ end \ end \ \$$
 $\$ \ end \ C \ S \ | \ begin \ a \ ; \ ; \ end \ end \ \$$
 $\$ \ end \ C \ B \ | \ begin \ a \ ; \ ; \ end \ end \ \$$
 $\$ \ end \ C \ end \ C \ S \ begin| \ begin \ a \ ; \ ; \ end \ end \ \$$
 $\$ \ end \ C \ end \ C \ S \ | \ a \ ; \ ; \ end \ end \ \$$
 $\$ \ end \ C \ end \ C \ B \ | \ a \ ; \ ; \ end \ end \ \$$
 $\$ \ end \ C \ end \ C \ a \ | \ a \ ; \ ; \ end \ end \ \$$
 $\$ \ end \ C \ end \ C \ |; \ ; \ end \ end \ \$$
 $\$ \ end \ C \ end \ C \ S \ ;|; \ ; \ end \ end \ \$$
 $\$ \ end \ C \ end \ C \ S \ | \ ; \ end \ end \ \$$
 $\$ \ end \ C \ end \ C \ E \ | \ ; \ end \ end \ \$$
 $\$ \ end \ C \ end \ C \ | \ ; \ end \ end \ \$$
 $\$ \ end \ C \ end \ C \ S \ ;|; \ end \ end \ \$$

$\$ \text{ end C end C S } | \text{ end end } \$$
 $\$ \text{ end C end C E } | \text{ end end } \$$
 $\$ \text{ end C end C } | \text{ end end } \$$
 $\$ \text{ end C end } | \text{ end end } \$$
 $\$ \text{ end C } | \text{ end } \$$
 $\$ \text{ end } | \text{ end } \$$
 $\$ | \$$

Ausgabe:

$(S \rightarrow B)(B \rightarrow \text{begin S C end})(S \rightarrow B)(B \rightarrow a)(C \rightarrow ; SC)(S \rightarrow B)(B \rightarrow \text{begin S C end})$
 $(S \rightarrow B)(B \rightarrow a)(C \rightarrow ; S C)(S \rightarrow E)(E \rightarrow \epsilon)(C \rightarrow ; S C)(S \rightarrow E)(E \rightarrow \epsilon)(C \rightarrow \epsilon)(C \rightarrow \epsilon)$

Der Begriff starker LL(k)-Parser noch kurz erklärt: „stark“ zeigt an welche Parserkonstruktionsmethode verwendet wird, hier hängt die Auswahl der Alternative für das aktuelle Nichtterminal nicht vom konsumierten Linkskontext ab (unabhängig vom Linkskontext). LL bedeutet, dass der Eingabestring von links nach rechts geparkt wird und dass eine Linksableitung produziert wird. k bedeutet, dass die Länge des verwendeten lookaheads höchstens k ist.

Im nächsten Kapitel werden die Eigenschaften von SLL(k)-Grammatiken vorgestellt.

V) Eigenschaften starker LL(k) Grammatiken

Vorab ein paar Definitionen:

Eine **Grammatik ist stark LL(k)**, wenn ihr SLL(k)-Parser deterministisch ist.

Eine **Sprache** über ein Alphabet T **ist stark LL(k)**, wenn sie von einer starken LL(k) Grammatik mit Terminalalphabet T akzeptiert wird.

Wir betrachten im folgenden kurz die Nichtterminale der Produktionsregeln eines SLL(k)-Parsers, bei denen es zu einem produce-produce Konflikt kommen kann:

$$A|y \rightarrow w_1|y \quad A|y \rightarrow w_2|y \quad \text{mit } w_1 \neq w_2$$

Da das Auftreten von Nichtdeterminismus unerwünscht ist, definiert man, dass jedes **Nichtterminal SLL(k) Eigenschaft** hat, wenn folgendes eingehalten wird:

$$\text{FIRST}_k(w_1 \text{ FOLLOW}_k(A)) \cap \text{FIRST}_k(w_2 \text{ FOLLOW}_k(A)) = \emptyset$$

\forall disjunkte Regeln $A \rightarrow w_1 | w_2$

d.h., dass die lookahead-Symbole, die für w_1 und w_2 verwendet werden, nicht gleich sein dürfen.

Man kann somit folgende Charakterisierungen treffen:

\forall reduzierte Grammatiken und $k \in \mathbf{N}$ sollen die folgenden Behauptungen äquivalent gelten:

- I) der SLL(k) – Parser für G ist deterministisch
- II) es tritt kein produce-produce Konflikt auf
- III) alle Nichtterminale haben die SLL(k)-Eigenschaft
- IV) folgendes wird immer erfüllt:

$$x_1, x_2 \in T^* \quad w_1, w_2, \delta_1, \delta_2 \in V^*$$

$$S \Rightarrow^* x_1 A \delta_1 \Rightarrow x_1 w_1 \delta_1 \Rightarrow^* x_1 y_1, \quad (\text{wobei } \Rightarrow \text{ und } \Rightarrow^* \text{ Linksablei-}$$

$$S \Rightarrow^* x_2 A \delta_2 \Rightarrow x_2 w_2 \delta_2 \Rightarrow^* x_2 y_2, \quad \text{tungen sind})$$

$$\text{und } k:y_1 = k:y_2$$

$$\Rightarrow w_1 = w_2$$

Unter anderem ist von diesen Eigenschaften ableitbar, dass eine SLL(k)-Grammatik eindeutig ist, hierauf gehe ich aber nicht näher ein.

Im folgenden Abschnitt wird noch das Problem der Rekursivität in einer Grammatik behandelt.

Vorab wieder ein paar Definitionen:

Eine Konfiguration ϕ eines KA M durchläuft eine **Schleife**, falls es $\forall n \in \mathbb{N}$ Konfigurationen ϕ_n gibt für die gilt:

$$\phi \Rightarrow^n \phi_n$$

Sei ϕ eine Schleifen-Konfiguration eines Kellerautomaten M , dann gibt es ein $y \in T^*$ und eine unendliche Reihe von $y_i \in V^*$, $i \geq 0$, so dass gilt:

$$\phi \Rightarrow^* \$y_0|y\$ \quad \text{und} \quad \$y_i|y\$ \Rightarrow \$y_{i+1}|y\$ \quad , \forall i \geq 0.$$

Daraus ist leicht ersichtlich, dass der Kellerautomat M unendlich lang eine Schleife für ein Eingabewort $w \in T^*$ durchläuft, genau dann, wenn es eine Konfiguration von M für w gibt, die eine Schleife durchläuft.

Ein **Nichtterminal** A einer Grammatik G ist **links-rekursiv**, wenn es eine Ableitung der Form $A \Rightarrow^+ A\beta$ mit $\beta \in V^*$ gibt.

Eine **Grammatik ist links-rekursiv**, wenn sie mindestens ein links-rekursives Nicht-terminal enthält.

Nun zu den Auswirkungen der Rekursivität auf den SLL(k)-Parser:

Sei G reduzierte Grammatik und $k \in \mathbb{N}$.

Ist die Grammatik G links-rekursiv, dann ist das genau dann der Fall, wenn der SLL(k)-Parser von G für $w \in L(G)$ unendlich lang eine Schleife durchläuft. Somit hält ein SLL(k)-Parser für jede nicht links-rekursive Grammatik an. Und es folgt, dass eine reduzierte, linksrekursive Grammatik nicht stark LL(k) ist für $k \geq 0$.

VI) Implementierung eines starken LL(k)-Parsers

In diesem Kapitel werden RAM-Programme eingesetzt, um einen SLL(1)-Parser zu simulieren.

Wir betrachten dabei zwei verschiedene Ansätze:

- 1.) Der Keller des zugrundeliegenden Kellerautomaten wird exakt implementiert
- 2.) Rekursives Absteigen

1.1) Grundlagen:

Gegeben sei eine SLL(1) – Grammatik $G = (V, T, P, S)$, bei der T aus Zeichenklassennamen besteht (siehe erster Vortrag).

Beispiel für Grammatik G_{block} :

$T = \{ \text{a-token, begin-keyword, end-keyword, semicolon} \}$, wobei die Elemente der Menge für die folgende Token Tokenklassennamen sind:

a-token = `a` ,
begin-keyword = `begin` ,
end-keyword = `end` ,
semicolon = `;`
eof-token = `\$` (speziell für das Endmarkierungssymbol)

Im folgenden wird kurz auf die Konstruktion bzw. Implementierung des Scanners und des Kellers eingegangen.

a) **Konstruktion Scanner:**

Das Wort $w = x_1 x_2 \dots x_k$ der Eingabe wird durch den Scanner zu einer Folge von Tupeln verarbeitet: $(x_1, m_1) \dots (x_k, m_k) (\$, eof\text{-token})$, dabei steht m_i mit $i = 1, \dots, k$ für die jeweilige Tokenklasse in der x_i enthalten ist. Bei der Verarbeitung des Eingabewortes wird so auch eine lexikalische Analyse durchgeführt, denn bei einem unkorrekten lexikalischen Ausdruck für ein x_i bekommt das entsprechende m_i den Wert error-token zugewiesen. Die Tokenklasse error-token enthält alle nicht korrekten lexikalischen Ausdrücke.

Der Parser ruft den Scanner auf, wenn das aktuelle Eingabesymbol geshiftet wurde und ein neues Eingabesymbol bestimmt werden muß. Der Befehl **scan** veranlaßt, dass das nächste token vom Eingabewort extrahiert wird und in der globalen Variable **token** gespeichert wird. Diese Variable besteht aus zwei Feldern in Anlehnung an das Tupel (x, m) . Das Feld mit der Bezeichnung token.kind enthält m , den Zeichenklassenname von x , wobei das andere Feld natürlich x enthält, den aktuellen Tokenstring.

b) Konstruktion Keller:

Der Keller kann alle Symbole der Grammatik inklusiv \$ enthalten und wird mit den Operationen **empty**, **push**, **pop** und **isempty** bearbeitet.

empty: Der Keller wird vollständig geleert.

push(X): Das Symbol X wird auf den Keller gelegt

pop(symbol): wirft das oberste Symbol aus dem Keller und speichert es in der Variable symbol

isempty: überprüft ob der Keller leer ist.

Der Keller wird mit \$ und S initialisiert. Um die nächste Aktion des SLL(1)-Parsers bestimmen zu können, wird das oberste Symbol des Kellers mit dem Befehl **pop(symbol)** gepoppt. Ist dabei das gepoppte Symbol ein Terminal und das aktuelle Eingabesymbol a stimmt mit diesem überein, dann wird a aus dem Keller geschiftet ($a|a \rightarrow |$) und der nächste Token wird eingelesen. Ist das gepoppte Symbol dagegen ein Nichtterminal, dann wird die ihm durch die Grammatik zugewiesene Produktionsregel angewendet, natürlich mit Überprüfung des aktuellen Eingabesymbol ($A|a \rightarrow \omega^R|a$).

Die Simulation ist zu Ende, wenn:

- 1.) Der Keller ist leer und das letztes Eingabezeichen ist \$
- 2.) Der Keller ist leer aber das aktuelle Eingabezeichen ist \neq \$
- 3.) Der Keller ist leer aber keine Produktionsregeln sind mehr anwendbar

Dabei terminiert bei 2.) und 3.) der Prozess und es wird eine Fehlermeldung ausgegeben (**error(m)**).

Nun zur Implementierung.

1.2) Implementierung

Die Kellerimplementierung eines SLL(1)-Parsers:

```
empty;
push(eof-token);
push(S);
scan;

repeat
  pop(symbol);
  case symbol of

    X1: parse(X1);
```

```

    X2: parse(X2);
    .
    .
    Xn: parse(Xn);
end
until isempty;

```

Nun die Parsingprogramme für ein gepopptes Nichtterminal und ein gepopptes Terminal:

Parsingprogramm für ein **Nichtterminal A** mit den Produktionsregeln
 $A \rightarrow X_{11} \dots X_{1n_1} \mid \dots \mid X_{m1} \dots X_{mn_m}$

```

parse(A) =
    if token.kind in FIRST1(X11...X1n1 FOLLOW1(A)) then
        begin write ``A → X11...X1n1``;
            push(X1n1); ...;push(X11);
        end else
        .
        .
        .
        if token.kind in FIRST1(Xm1...Xmnm FOLLOW1(A)) then
            begin write ``A → Xm1...Xmnm``;
                push(Xmnm); ...;push(Xm1);
            end else
                error(``No A can start with this``);

```

Parsingprogramm für ein **Terminal a** und das **Endmarkierungssymbol \$**:

```

parse(a) =
    if token.kind = a then
        scan;
    else
        error(``a erwartet``);

```

```

parse(eof-token) =
    if token.kind <> eof-token then
        error(`` Ende der Eingabe erwartet``);

```

Beispiel für eine Kellerimplementation durch Parsingprogramme für die Grammatik
 $G = (V, T, P, S)$

P : $S \rightarrow E \mid B$
 $E \rightarrow \epsilon$
 $B \rightarrow a \mid \text{begin } S \text{ } C \text{ end}$
 $C \rightarrow \epsilon \mid ; S C$

```
parse(S) =  
  if token.kind in {a-token, begin-keyword} then  
    begin write ``S → B``;  
          push(B);  
    end else  
    if token.kind in {end-keyword, semicolon, eof-token} then  
      begin write ``S → E``;  
            push(E);  
    end else  
      error(``No E can start with this``);
```

```
parse(E) =  
  if token.kind in { end-keyword, semicolon, eof-token } then  
    begin write ``E → ε``;  
  end else  
    error(``No E can start with this``);
```

```
parse(B) =  
  if token.kind in {a-token} then  
    begin write ``B → a``;  
          push(a-token);  
    end else  
    if token.kind in {begin-keyword} then  
      begin write ``B → begin C S end``;  
            push(end-keyword); push(C); push(S); push(begin-keyword);  
    end else  
      error(``No B can start with this``);
```

```
parse(C) =  
  if token.kind in {end-keyword} then  
    begin write ``C → ε``;  
  end else  
    if token.kind in {semicolon} then  
      begin write ``C → ; S C``;
```

```

        push(C); push(S);push(semicolon);
    end else
        error(`No C can start with this`);

```

2.) Rekursives Absteigen

2.1) Grundlagen

Die in 1.1 beschriebenen Funktionen werden auch in dieser Implementierung ebenfalls verwendet.

Beim rekursiven Absteigen wird kein Keller implementiert, sondern eine Menge von Prozeduren, die sich gegenseitig untereinander rekursiv aufrufen (man nutzt also den Runtimestack des laufenden Programms). Dabei hat jedes Nichtterminal A eine eigene Prozedur mit dem Ziel, die Sätze zu parsen, die von der Grammatik (V, T, P, A) abgeleitet werden können. Die Prozedur A wird aufgerufen, wenn das aktuelle Eingabesymbol einen Satz in der Sprache $L(A)$ einleitet; sie parst den Satz und gibt die Kontrolle an die aufgerufene Prozedur zurück. Nach diesem **return** schreitet das Einlesen der Eingabe weiter fort, so dass das aktuelle Eingabesymbol der wirkliche Nachfolger von A ist. Das Hauptprogramm besteht aus der Prozedur S , die nach dem Einlesen des ersten Eingabesymbols aufgerufen wird. Wird ein **return** der Prozedur S zurückgegeben, überprüft das Programm, ob die Eingabe abgearbeitet ist, d.h. ob das Endmarkierungssymbol $\$$ auf dem Eingabeband steht.

2.2) Implementierung

Das Hauptprogramm:

```

scan;
S;
if token.kind < > eof.token then
    error(`Ende der Eingabe erwartet`);

```

Parsing-Prozedur für ein Nichtterminal A :

$$A \rightarrow \omega_1 | \omega_2 \dots | \omega_n$$

```

procedure A;
begin
    if token.kind in FIRST1 ( $\omega_1$  FOLLOW1 (A)) then
        begin
            write `A  $\rightarrow$   $\omega_1$ `;
            parse( $\omega_1$ );
        end

```

```

end else
.
.
if token.kind in FIRST1( $\omega_n$  FOLLOW1(A)) then
  begin
    write ``A  $\rightarrow$   $\omega_n$  ``;
    parse( $\omega_n$ );
  end else
    error(`` No A can start with this. ``);
end;

```

Parsing-Prozedur für die **Suffixe der rechten Seite**:

$a \in T, A \in V \setminus T, \beta \in V^*$

parse($a\beta$) =	parse($A\beta$) =
scan;	A;
check(β);	check(β);

check($a\beta$) =	check($A\beta$) =
if token.kind = a then	parse($A\beta$);
scan;	
else	
error(`` a erwartet ``);	
check(β)	

parse(ϵ) = check(ϵ) = ϵ

Beispiel zum rekursiven Absteigen für die Grammatik $G_{\text{block}} = (V, T, P, S)$:

P : $S \rightarrow E \mid B$
 $E \rightarrow \epsilon$
 $B \rightarrow a \mid \text{begin } S \ C \ \text{end}$
 $C \rightarrow \epsilon \mid ; \ S \ C$

```

procedure S;
begin
  if token.kind in {a-token,being-keyword} then
    begin
      write ``S  $\rightarrow$  B ``;
      B;
    end else
    if token.kind in {end-keyword,eof-token,semicolon} then
      begin
        write ``S  $\rightarrow$  E ``;
        E;

```

```
    end else
        error(` No S can start with this.`);
end;
```

```
procedure E;
begin
    if token.kind in {end-keyword,semicolon,eof-token} then
        begin
            write``E → ε``;
            B;
        end else
            error(` No E can start with this.`);
    end;
```

```
procedure B;
begin
    if token.kind in {a-token} then
        begin
            write``B → a``;
            scan;
        end else
            if token.kind in {begin-keyword} then
                begin
                    write``B → begin S C end ``;
                    scan;
                    S;
                    C;
                    if token.kind = end-keyword then
                        scan;
                    else
                        error (`end-keyword expected. `);
                end else
                    error(` No B can start with this.`);
            end;
```

```
procedure C;
begin
    if token.kind in {end-keyword} then
        begin
            write``C → ε``;
        end else
            if token.kind in {semicolon} then
                begin
```

```
write``C → ; S C ``;  
scan;  
S;  
C;  
end else  
    error(`` No C can start with this. ``);  
end;
```