$S \mapsto \#$ $\quad E \mapsto \{\#,)\}$ $\quad T \mapsto \{+\}$
$F \mapsto \{*\}$ $\quad E' \mapsto \emptyset$ $\quad T' \mapsto \emptyset$

Der Algorithmus SZK arbeitet auf diesem Graphen wie folgt: (Die Zuordnung der Menge $\{\#\}$ zum Startsymbol S bleibt natürlich konstant und ist weggelassen worden):

Schritt /Anw	Keller S	Berechnung der Funktion Fo E	E'	T	T'	F
(1)/(1)	$\boxed{F\|T\|T'\|E\|E'}$	$\{\#,)\}$	$\emptyset$	$\{+\}$	$\emptyset$	$\{*\}$
(2)/(3)	$\boxed{F\|T\|T'\|E}$	$\{\#,)\}$	$\{\#,)\}$	$\{+\}$	$\emptyset$	$\{*\}$
(3)/(2)	$\boxed{F\|T\|T'}$	$\{\#,)\}$	$\{\#,)\}$	$\{+,\#,)\}$	$\emptyset$	$\{*\}$
(4)/(3)	$\boxed{F\|T}$	$\{\#,)\}$	$\{\#,)\}$	$\{+,\#,)\}$	$\{+,\#,)\}$	$\{*\}$
(5)/(2)	$\boxed{F}$	$\{\#,)\}$	$\{\#,)\}$	$\{+,\#,)\}$	$\{+,\#,)\}$	$\{*,+,\#,)\}$
(6)/(1)	$\boxed{S}$	$\{\#,)\}$	$\{\#,)\}$	$\{+,\#,)\}$	$\{+,\#,)\}$	$\{*,+,\#,)\}$

□

8.3 Top down-Syntaxanalyse

8.3.1 Einführung

Die Arbeitsweise von Parsern kann man sich intuitiv am besten klar machen, wenn man sich vorstellt, wie sie den Syntaxbaum zu einem Eingabewort konstruieren. Top down-Parser beginnen die Konstruktion des Syntaxbaums an der Wurzel. Die Anfangssituation ist die in Abbildung 8.12 gezeigte; das erste Fragment des Syntaxbaums besteht aus der Wurzel, markiert mit dem Startsymbol S der kontextfreien Grammatik; das Eingabewort w ist gegeben, und der Eingabezeiger steht vor dem ersten Symbol von w.

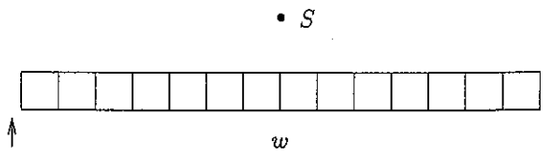


Abb. 8.12: Start einer top down-Analyse

Jetzt wird – auf später beschriebene Weise – eine Alternative für S zur Expansion ausgewählt. Die Symbole der rechten Seite dieser Alternative werden in ihrer dortigen Reihenfolge unter die Wurzel gehängt. Dadurch entsteht ein neues Fragment des Syntaxbaums. Die Auswahl eines Nichtterminals, im folgenden immer des am weitesten links stehenden, die Auswahl einer seiner Alternativen und deren „Anbau" an den aktuellen Syntaxbaum wiederholen sich. Durch den

Anbau einer Produktion können Terminalsymbole im Blattwort des Baumfragments auftreten. Stehen links von ihnen keine Nichtterminale im Blattwort, so vergleichen wir sie mit noch nicht gelesenen Symbolen, die im Eingabewort hinter dem Lesezeiger stehen, und haken sie bei Übereinstimmung durch Vorrücken des Lesezeigers als bestätigt ab.

Die top down-Analyse läuft also als eine Folge von folgenden zwei Arten von Aktivitäten ab: „baue Produktion an aktuelles Baumfragment an" und „verifiziere Übereinstimmung zwischen angebauten und Eingabesymbolen" .

Eine Folge von solcherart konstruierten Syntaxbaumfragmenten für die Grammatik G_2 für arithmetische Ausdrücke ist in Abbildung 8.13 dargestellt. Man beachte, daß die Folge der Blattwörter der Fragmente eine Linksableitung für den erzeugten Satz $\mathbf{id} + \mathbf{id} * \mathbf{id}$ ist. Die Auswahl der Alternativen für die jeweils zu expandierenden Nichtterminale wurde jeweils so vorgenommen, daß die Analyse Erfolg hat. Daß dabei eine nicht immer vorauszusetzende „höhere Einsicht" vorhanden war, zeigt der nächste Abschnitt.

8.3.2 Top down-Syntaxanalyse mit Zurücksetzen in Prolog

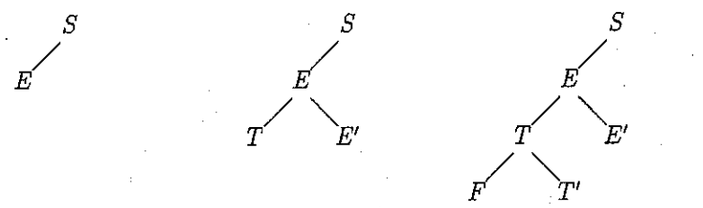
Eine mögliche Spezifikation eines Parsers in Prolog besteht aus einer Darstellung der kontextfreien Grammatik und einem „Treiber", das ist ein Programm, welches die Grammatik interpretiert[1]. In der Darstellung von Grammatiken verwenden wir die Funktoren t (für Terminal) und n (für Nichtterminal). Manche der Terminalsymbole müssen in ' ' eingeschlossen werden. Das Eingabewort wird als eine Liste von so dargestellten Terminalen angegeben, etwa `[t(id), t('+'),t(id),t('*'),t(id),t('#')]`. Die Produktionen der Grammatik werden als Fakten des Prolog-Parsers angegeben, etwa `rule(n,factor),[t('('), n(expr),t(')')])`, d.h. die rechte Seite der Produktion wird als Liste ihrer Komponenten dargestellt.

Top down-Parser sind Vorhersage-Parser (predictive parsers); durch die Expansion eines Nichtterminals mithilfe einer seiner Alternativen macht der Parser eine Vorhersage darüber, welche Struktur er für einen Präfix der restlichen Eingabe erwartet. Diese Vorhersagen werden auf einen Keller gespeichert und bei gefundener Bestätigung daraus entfernt. Der Keller wird ebenfalls als Liste dargestellt. Da in Prolog nur auf das erste, also am weitesten links stehende Element einer Liste effizient zugegriffen werden kann, ist jetzt – im Gegensatz zum Keller von K_G – das obere Kellerende am linken Ende der Liste.

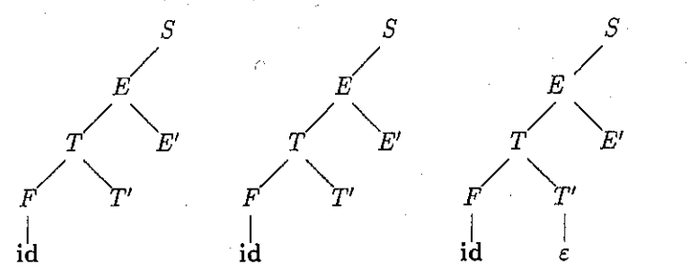
Das Prädikat `predict(Input, Stack)` hat als Eingabe die (restliche) Eingabeliste und den aktuellen Kellerinhalt. Beim ersten Aufruf sind dies das zu analysierende Eingabewort, und als Kellerinhalt $n(s)$, wenn s das Startsymbol der Grammatik ist. Das Prädikat hat keine Ausgabe, weil wir auf den Aufbau des Syntaxbaums verzichten. Akzeptiert wird, wenn die Eingabe erschöpft ist

[1] Es gibt allerdings einen in Prolog eingebauten Formalismus, die „definite clause grammars", die eine unserer bisherigen Notation ähnliche Aufschreibung von Grammatiken erlauben.

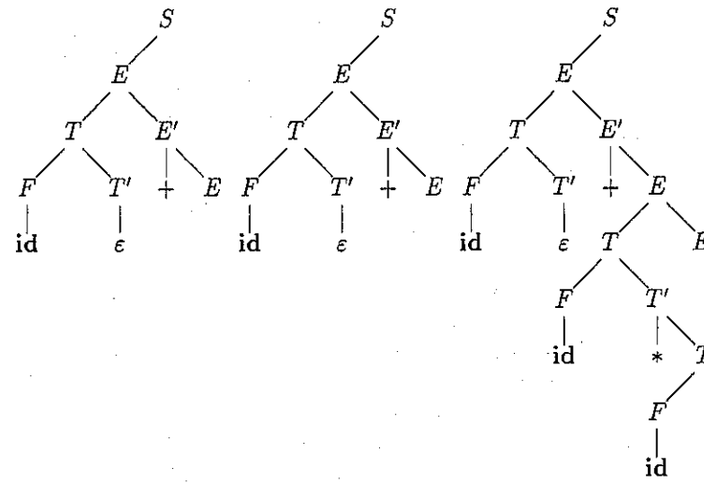
$$S \longrightarrow E \qquad E' \longrightarrow +\,E \mid \varepsilon \qquad T' \longrightarrow *\,T \mid \varepsilon$$
$$E \longrightarrow T\,E' \qquad T \longrightarrow F\,T' \qquad F \longrightarrow (E) \mid \mathrm{id}$$



| id + id ∗ id | id + id ∗ id | id + id ∗ id



| id + id ∗ id | id | + id ∗ id | id | + id ∗ id



id | + id ∗ id | id + | id ∗ id | id + id ∗ id |

Abb. 8.13: Einige Schritte einer top down-Analyse; der Stand des Lesezeigers in der Eingabe wird durch das |-Zeichen angegeben.

und der Keller leer ist. Der Treiber, hier gegeben durch drei Klauseln für das Prädikat `predict`, ist:

```
%  akzeptiere
predict([], []).

%  expandiere Nichtterminal mithilfe einer seiner Alternativen
predict(Input, [n(X) | Stack]) :-
      rule(n(X), Rhs),
      append(Rhs, Stack, Newstack),
      predict(Input, Newstack).

%  verifiziere Terminal
predict([t(A) | Restinput], [t(A) | Stack]) :-
      predict(Restinput, Stack).
```

Dieser Parser wendet die Produktionsregeln in der Reihenfolge ihrer Aufschreibung an. Er setzt zurück, wenn ein Terminal oben auf dem Keller, aber nicht am Anfang der restlichen Eingabe steht, bzw. wenn alle Alternativen für ein Nichtterminal erfolglos versucht wurden. Eine linksrekursive Produktion kann ihn in eine endlose Berechnung bringen. Selbst wenn die Grammatik eine sehr schnelle Syntaxanalyse erlaubt – mit einem der Verfahren aus den nächsten Abschnitten – kann dieser Prolog-Parser durch das Backtracking ein nicht akzeptables Zeitverhalten zeigen. Wir werden aber später auf Verfahren zur top down-Syntaxanalyse in Prolog zurückkommen, die ohne Backtracking arbeiten.

Beispiel 8.3.1
Gegeben seien obiger Parser-Treiber und die Menge von Produktionen als Fakten:

```
rule(n(expr),   [n(term), n(expr1)]).
rule(n(expr1),  []).
rule(n(expr1),  [ t('+'), n(expr)]).
rule(n(term),   [n(factor), n(term1)]).
rule(n(term1),  []).
rule(n(term1),  [t('*'), n(term)]).
rule(n(factor), [t(id)]).
rule(n(factor), [t('('), n(expr), t(')')]).
```

Die Anfrage

```
?- predict([t(id), t('*'), t(id)], [n(expr)]).
```

führt zu folgendem Trace:

```
C-Prolog version 1.5

Spy-point placed on predict/2.
```

```
Debug mode switched on.
No leashing.
*  (4)  2 Call: predict([t(id),t(*),t(id)],[n(expr)])
*  (9)  3 Call: predict([t(id),t(*),t(id)],[n(term),n(expr1)])
*  (14) 4 Call: predict([t(id),t(*),t(id)],[n(factor),n(term1),
                n(expr1)])
*  (18) 5 Call: predict([t(id),t(*),t(id)],[t(id),n(term1),
                n(expr1)])
*  (19) 6 Call: predict([t(*),t(id)],[n(term1),n(expr1)])
*  (22) 7 Call: predict([t(*),t(id)],[n(expr1)])
*  (25) 8 Call: predict([t(*),t(id)],[])
*  (25) 8 Fail: predict([t(*),t(id)],[])
*  (29) 8 Call: predict([t(*),t(id)],[t(+),n(expr)])
*  (29) 8 Fail: predict([t(*),t(id)],[t(+),n(expr)])
*  (22) 7 Back to: predict([t(*),t(id)],[n(expr1)])
*  (22) 7 Fail: predict([t(*),t(id)],[n(expr1)])
*  (33) 7 Call: predict([t(*),t(id)],[t(*),n(term),n(expr1)])
*  (34) 8 Call: predict([t(id)],[n(term),n(expr1)])
*  (39) 9 Call: predict([t(id)],[n(factor),n(term1),n(expr1)])
*  (43) 10 Call: predict([t(id)],[t(id),n(term1),n(expr1)])
*  (44) 11 Call: predict([],[n(term1),n(expr1)])
*  (47) 12 Call: predict([],[n(expr1)])
*  (50) 13 Call: predict([],[])
*  (50) 13 Exit: predict([],[])
*  (47) 12 Exit: predict([],[n(expr1)])
```

weiteres Verlassen von erfolgreichen Klauselanwendungen

```
*  (4) 2 Exit: predict([t(id),t(*),t(id)],[n(expr)])
yes
```

Betrachten wir die Schritte, die dieser Parser bei der Analyse von **id** $*$ **id** macht. In den Schritten (4), (9), (14), (18) und (19) stellt er den Anfang einer Linksableitung her

$$expr \Rightarrow term\ expr1 \Rightarrow factor\ term1\ expr1 \Rightarrow \mathbf{id}\ term1\ expr1$$

und verifiziert die Übereinstimmung des Symbols **id** in der Eingabe und in der produzierten Satzform. Im Schritt (22) benutzt er die erste, die leere Alternative für *term1* und in Schritt (25) die erste, die leere Alternative für *expr1*. Damit hat er einen leeren Keller, aber noch die Eingabe $*$**id**. Daraufhin macht er die letzte Wahl rückgängig. Diese und die folgenden Schritte lassen sich folgendermaßen darstellen:

$$\mathbf{id}\ term1\ expr1 \overset{(22)}{\Rightarrow} \mathbf{id}\ expr1 \underset{(25)}{\overset{(25)}{\rightleftarrows}} \mathbf{id}$$

$$\overset{(29)}{\Rightarrow} \mathbf{id} + expr$$
$$\overset{(22)}{\Leftarrow} \qquad \overset{(29)}{\Leftarrow}$$
$$\overset{(33)}{\Rightarrow} \mathbf{id} * term\ expr1$$

Von da ab verläuft die Analyse erfolgreich.

8.3.3 LL(k): Definition, Beispiele, Eigenschaften

Der Item-Kellerautomat K_G zu einer kontextfreien Grammatik G, arbeitet im Prinzip wie ein top down-Parser; seine (E)-Übergänge machen eine Voraussage, welche Alternative für das aktuelle Nichtterminal auszuwählen ist, um das Eingabewort abzuleiten. Störend daran ist, daß K_G diese Auswahl nichtdeterministisch trifft. Der ganze Nichtdeterminismus von K_G steckt in den (E)-Übergängen. Wenn $[X \to \beta.Y\gamma]$ der aktuelle Zustand ist und Y die Alternativen $Y \to \alpha_1 \mid \ldots \mid \alpha_n$ hat, so gibt es unter δ die n Übergänge

$$\delta([X \to \beta.Y\gamma], \varepsilon) = \{[X \to \beta.Y\gamma][Y \to .\alpha_i] \mid 1 \leq i \leq n\}$$

Um aus K_G einen deterministischen Automaten zu machen, werden wir eine begrenzte Vorausschau in die restliche Eingabe gestatten. Genauer, wir geben eine natürliche Zahl k vor und lassen K_G bei jedem (E)-Übergang als Entscheidungshilfe die k ersten Symbole der restlichen Eingabe anschauen. Ist sichergestellt, daß diese Vorausschau der Länge k immer ausreicht, um die richtige Alternative auszuwählen, so werden wir die Ausgangsgrammatik G eine LL(k)-Grammatik nennen.

Schauen wir uns eine entsprechende Konfiguration an, die K_G aus einer Anfangskonfiguration erreicht hat:

$$([S' \to .S], uv) \vdash_{K_G}^{*} (\rho[X \to \beta.Y\gamma], v)$$

Die Invariante (I) aus Abschnitt 8.2.3 besagt, daß $hist(\rho)\beta \overset{*}{\Longrightarrow} u$ gilt. Sei $\rho = [\,X_1 \longrightarrow \beta_1.X_2\gamma_1\,]\ldots[\,X_n \longrightarrow \beta_n.X_{n+1}\gamma_n\,]$ eine Folge von Items. Wir definieren die **Zukunft** von ρ, $fut(\rho)$, als $\gamma_n \ldots \gamma_1$. Sei $\delta = fut(\rho)$. Wenn gilt, daß sich die bisher gefundene Linksableitung $S' \underset{lm}{\overset{*}{\Longrightarrow}} uY\gamma\delta$ zur Ableitung des Terminalwortes uv fortsetzen läßt, d.h. $S' \underset{lm}{\overset{*}{\Longrightarrow}} uY\gamma\delta \underset{lm}{\overset{*}{\Longrightarrow}} uv$, dann kann in einer LL($k$)-Grammatik die Auswahl der für Y anzuwendenden Alternative immer durch Betrachtung von u, Y und $k : v$ getroffen werden.

Definition 8.3.1 (LL(k)-Grammatik)
Sei $G = (V_N, V_T, P, S)$ eine kontextfreie Grammatik und k eine natürliche Zahl. G ist eine **LL(k)-Grammatik**, wenn gilt:

Existieren zwei Linksableitungen

$S \overset{*}{\underset{lm}{\Longrightarrow}} uY\alpha \underset{lm}{\Longrightarrow} u\beta\alpha \overset{*}{\underset{lm}{\Longrightarrow}} ux$ und

$S \overset{*}{\underset{lm}{\Longrightarrow}} uY\alpha \underset{lm}{\Longrightarrow} u\gamma\alpha \overset{*}{\underset{lm}{\Longrightarrow}} uy$, und ist $k:x = k:y$,

dann gilt $\beta = \gamma$. □

Diese Definition besagt, daß die Auswahl der Alternative für das aktuelle Nichtterminal Y bei festem Linkskontext u durch die k ersten Symbole der restlichen Eingabe eindeutig festgelegt ist. Man beachte, daß die Auswahl aber nicht nur von Y und den nächsten k Symbolen, sondern i.a. auch von dem bereits konsumierten Präfix u der Eingabe abhängt.

Mit obiger Definition haben wir noch keinen Ansatz für einen Test auf die LL(k)-Eigenschaft einer kontextfreien Grammatik. Im folgenden werden wir einfache Tests für die Mitgliedschaft zu Unterklassen der LL(k)-Grammatiken kennenlernen, insbesondere einen praktikablen Test für die Klasse der starken LL(k)-Grammatiken. In dieser Klasse hängt die Auswahl einer Alternative für das aktuelle Nichtterminal nicht vom konsumierten Linkskontext ab.

Beispiel 8.3.2
Sei G_1 die kontextfreie Grammatik mit den Produktionen:

$$\begin{array}{lll} STAT \rightarrow & \textbf{if id then } STAT \textbf{ else } STAT \textbf{ fi} & | \\ & \textbf{while id do } STAT \textbf{ od} & | \\ & \textbf{begin } STAT \textbf{ end} & | \\ & \textbf{id := id} & \end{array}$$

G_1 ist LL(1)-Grammatik. Tritt $STAT$ als linkestes Nichtterminal in einer Satzform auf, so bestimmt das nächste Eingabesymbol die anzuwendende Alternative. Genauer:

$$STAT \overset{*}{\underset{lm}{\Longrightarrow}} w\ STAT\ \alpha \underset{lm}{\Longrightarrow} w\beta\alpha \overset{*}{\underset{lm}{\Longrightarrow}} wx$$
$$STAT \overset{*}{\underset{lm}{\Longrightarrow}} w\ STAT\ \alpha \underset{lm}{\Longrightarrow} w\gamma\alpha \overset{*}{\underset{lm}{\Longrightarrow}} wy$$

Gilt $1:x = 1:y$, dann folgt, daß $\beta = \gamma$ ist.
Z.B. folgt aus $1:x = 1:y =$ **if** , daß $\beta = \gamma =$ **if id then** $STAT$ **else** $STAT$ **fi** □

Definition 8.3.2 (einfache LL(1)-Grammatik)
Sei G kontextfreie Grammatik ohne ε-Produktionen. Beginnt für jedes Nichtterminal N jede seiner Alternativen mit einem anderen Terminalsymbol, dann heißt G **einfache LL(1)-Grammmatik.** □

Dies ist ein erstes, leicht zu überprüfendes Testkriterium für einen Sonderfall. Die Grammatik G_1 aus Beispiel 8.3.2 ist eine einfache LL(1)-Grammatik.

Beispiel 8.3.3
Sei G_1 aus Beispiel 8.3.2 erweitert um die Produktionen

$$\begin{array}{lll} STAT \rightarrow \textbf{id: } STAT & | & (* \text{ markierte Anweisung } *) \\ \quad \textbf{id (id)} & & (* \text{ Prozeduraufruf } *) \end{array}$$

zur Grammatik G_2. G_2 ist nicht mehr LL(1), insbesondere nicht mehr einfach LL(1), denn:

$$STAT \overset{*}{\underset{lm}{\Longrightarrow}} w\ STAT\ \alpha \underset{lm}{\Longrightarrow} w\overbrace{\textbf{id := id}}^{\beta}\alpha \overset{*}{\underset{lm}{\Longrightarrow}} wx$$
$$STAT \overset{*}{\underset{lm}{\Longrightarrow}} w\ STAT\ \alpha \underset{lm}{\Longrightarrow} w\overbrace{\textbf{id : } STAT}^{\gamma}\alpha \overset{*}{\underset{lm}{\Longrightarrow}} wy$$
$$STAT \overset{*}{\underset{lm}{\Longrightarrow}} w\ STAT\ \alpha \underset{lm}{\Longrightarrow} w\overbrace{\textbf{id(id)}}^{\delta}\alpha \overset{*}{\underset{lm}{\Longrightarrow}} wz$$

und es gilt $1:x = 1:y = 1:z =$ **"id"**, und β, γ, δ sind paarweise verschieden. G_2 ist aber LL(2)-Grammatik; denn für die obigen drei Linksableitungen gilt: Es sind $2:x =$ **"id :="**, $2:y =$ **"id :"**, $2:z =$ **"id("** paarweise verschieden, und dies sind die einzigen kritischen Fälle.

Erinnern Sie sich, daß ein Scanner, wie er im letzten Kapitel beschrieben wurde, immer den längsten Präfix der restlichen Eingabe zum nächsten abzuliefernden Symbol zusammenfaßt. Daher wird er, wenn der Wertzuweisungsoperator ':=' in der Eingabe steht, nicht den Doppelpunkt als nächstes Symbol abliefern. □

Beispiel 8.3.4
G_3 enthalte die Produktionen

$$\begin{array}{lll} STAT \rightarrow & \textbf{if id then } STAT \textbf{ else } STAT \textbf{ fi} & | \\ & \textbf{while id do } STAT \textbf{ od} & | \\ & \textbf{begin } STAT \textbf{ end} & | \\ & VAR := VAR & | \\ & \textbf{id(} IDLIST \textbf{)} & (* \text{ Prozeduraufruf } *) \\ VAR \rightarrow & \textbf{id} \mid \textbf{id (} IDLIST \textbf{)} & (* \text{ indizierte Variable } *) \\ IDLIST \rightarrow & \textbf{id} \mid \textbf{id, } IDLIST & \end{array}$$

G_3 ist für kein k eine LL(k)-Grammatik.
Annahme: G_3 sei LL(k) für ein $k > 0$.

Sei $STAT \Rightarrow \beta \overset{*}{\underset{lm}{\Longrightarrow}} x$ und $STAT \Rightarrow \gamma \overset{*}{\underset{lm}{\Longrightarrow}} y$ mit
$x = \textbf{id}\,(\underbrace{\textbf{id, id, \ldots, id}}_{\lceil \frac{k}{2} \rceil \text{ mal}}) := \textbf{id}$ und $y = \textbf{id}\,(\underbrace{\textbf{id, id, \ldots, id}}_{\lceil \frac{k}{2} \rceil \text{ mal}})$

Dann ist $k:x = k:y$, aber $\beta = VAR := VAR \neq \gamma = \textbf{id}\,(IDLIST)$. □

Zu $L(G_3)$ gibt es allerdings eine LL(2)-Grammatik. Diese gewinnt man aus G_3 durch eine Transformation, genannt Ausfaktorisierung. Die kritischen Produktionen sind die für die Wertzuweisung und den Prozeduraufruf. Da sowohl die Dimension von Feldern als auch die Parameterzahl von Prozeduren nicht beschränkt ist, kann kein LL(k)-Parser das Vorhandensein eines ':=' nach der schließenden Klammer mit k Symbolen Vorausschau prüfen. Die Faktorisierung faßt gemeinsame Anfänge von solchen Produktionen unter einem neuen Nichtterminal zusammen.

Deshalb werden die Produktionen

$$\begin{array}{lll} \mathit{STAT} & \rightarrow \mathit{VAR} := \mathit{VAR} \mid \mathbf{id}(\mathit{IDLIST}) \end{array}$$

ersetzt durch

$$\begin{array}{lll} \mathit{STAT} & \rightarrow \mathit{ZUWPROZ} \mid \mathbf{id} := \mathit{VAR} \\ \mathit{ZUWPROZ} & \rightarrow \mathbf{id}(\mathit{IDLIST})\ \mathit{ZPREST} \\ \mathit{ZPREST} & \rightarrow := \mathit{VAR} \mid \varepsilon \end{array}$$

Jetzt kann sich ein LL(2)-Parser zwischen den kritischen Alternativen mittels der Kombinationen "**id**:=" bzw. "**id**(" für *STAT* entscheiden. Hat er sich für das Nichtterminal *ZUWPROZ* entschieden, so arbeitet er dessen Anfang bis *ZPREST* ab. Da auf ein *STAT* kein ':=' folgen darf, führt die Vorausschau eines Symboles zur richtigen Entscheidung für eine der Alternativen von *ZPREST*. Die Grundlage für diese Argumentation werden wir bald präzise einführen.

Beispiel 8.3.5
Sei $G_4 = (\{S, A, B\}, \{0, 1, a, b\}, P_4, S)$

$$P_4 = \left\{ \begin{array}{lll} S & \rightarrow & A \mid B \\ A & \rightarrow & aAb \mid 0 \\ B & \rightarrow & aBbb \mid 1 \end{array} \right\} \qquad L(G_4) = \{a^n 0 b^n \mid n \geq 0\} \cup \{a^n 1 b^{2n} \mid n \geq 0\}$$

G_4 ist nicht LL(k) für irgendein k.
Wir wählen für die Bezeichnungen aus Definition 8.3.1 $u = \alpha = \varepsilon$, $\beta = A$, $\gamma = B$, $x = a^k 0 b^k$, $y = a^k 1 b^{2k}$. Dann ergeben sich die beiden Linksableitungen:

$$S \underset{lm}{\overset{0}{\Longrightarrow}} S \underset{lm}{\Longrightarrow} A \underset{lm}{\overset{*}{\Longrightarrow}} a^k 0 b^k$$

$$S \underset{lm}{\overset{0}{\Longrightarrow}} S \underset{lm}{\Longrightarrow} B \underset{lm}{\overset{*}{\Longrightarrow}} a^k 1 b^{2k}$$

Es gilt $k : x = k : y$, aber $\beta \neq \gamma$. Da k beliebig gewählt werden kann, ist G_4 nicht LL(k) für beliebiges k. Es kann sogar gezeigt werden: Zu $L(G_4)$ gibt es für kein k eine LL(k)-Grammatik. □

Satz 8.3.1
Sei $G = (V_N, V_T, P, S)$ eine kontextfreie Grammatik. G ist genau dann LL(k), wenn die folgende Bedingung erfüllt ist:
Sind $A \rightarrow \beta$ und $A \rightarrow \gamma$ verschiedene Produktionen in P, dann gilt

$$\mathit{FIRST}_k(\beta\alpha) \cap \mathit{FIRST}_k(\gamma\alpha) = \emptyset \quad \textit{für alle } \alpha \quad \textit{mit } S \underset{lm}{\overset{*}{\Longrightarrow}} wA\alpha$$

Beweis:
" $\Rightarrow$ " Sei G LL(k).
Annahme: Es existiert $x \in \mathit{FIRST}_k(\beta\alpha) \cap \mathit{FIRST}_k(\gamma\alpha)$. Nach Definition von FIRST_k und wegen der Reduziertheit von G gibt es Ableitungen:

$$S \underset{lm}{\overset{*}{\Longrightarrow}} wA\alpha \underset{lm}{\Longrightarrow} w\beta\alpha \underset{lm}{\overset{*}{\Longrightarrow}} wxy$$

$$S \underset{lm}{\overset{*}{\Longrightarrow}} wA\alpha \underset{lm}{\Longrightarrow} w\gamma\alpha \underset{lm}{\overset{*}{\Longrightarrow}} wxz \text{ , (falls } |x| < k \text{, dann } y = z = \varepsilon)$$

Aus $\beta \neq \gamma$ folgt, G ist nicht LL(k).
" $\Leftarrow$ " Sei G nicht LL(k). Dann gibt es zwei Ableitungen

$$S \underset{lm}{\overset{*}{\Longrightarrow}} wA\alpha \underset{lm}{\Longrightarrow} w\beta\alpha \underset{lm}{\overset{*}{\Longrightarrow}} wx$$

$$S \underset{lm}{\overset{*}{\Longrightarrow}} wA\alpha \underset{lm}{\Longrightarrow} w\gamma\alpha \underset{lm}{\overset{*}{\Longrightarrow}} wy \text{ mit } k : x = k : y,$$

wobei $A \rightarrow \beta$, $A \rightarrow \gamma$ verschiedene Produktionen sind. Aber $k : x = k : y$ liegt in $\mathit{FIRST}_k(\beta\alpha) \cap \mathit{FIRST}_k(\gamma\alpha)$. Damit ist ein Widerspruch hergeleitet. □

Satz 8.3.1 besagt, daß in einer LL(k)-Grammatik die Anwendung zweier verschiedener Produktionen auf eine Linkssatzform immer zu verschiedenen k-Präfixen der restlichen Eingabe führt.

Aus Satz 8.3.1 kann man gute Kriterien für die Zugehörigkeit zu gewissen Teilklassen der LL(k)-Grammatiken ableiten. Die ersten betreffen den Fall $k = 1$. Die Menge $\mathit{FIRST}_1(\beta\alpha) \cap \mathit{FIRST}_1(\gamma\alpha)$ für alle Linkssatzformen $wA\alpha$ und je zwei verschiedene Alternativen $A \rightarrow \beta$ und $A \rightarrow \gamma$ läßt sich zu $\mathit{FIRST}_1(\beta) \cap \mathit{FIRST}_1(\gamma)$ vereinfachen, wenn weder β noch γ das leere Wort ε produzieren. Dies ist dann der Fall, wenn kein Nichtterminal von G ε-produktiv ist.

Satz 8.3.2
Sei G eine ε-freie kontextfreie Grammatik, d.h. ohne Produktionen der Form $X \rightarrow \varepsilon$. Dann ist G LL(1)-Grammatik genau dann, wenn für jedes Nichtterminal X mit den Alternativen $X \rightarrow \alpha_1 | \ldots | \alpha_n$ gilt: Die Mengen $\mathit{FIRST}_1(\alpha_1), \ldots, \mathit{FIRST}_1(\alpha_n)$ sind paarweise disjunkt.

Für die Praxis wäre es eine zu starke Einschränkung, ε-Produktionen zu verbieten. Was können wir sagen, wenn in Satz 8.3.1 eine der beiden rechten Seiten ε produzieren kann? Produzieren sowohl β als auch γ das leere Wort, so ist G offensichtlich nicht LL(k). Nehmen wir oBdA. an, daß $\beta \overset{*}{\Longrightarrow} \varepsilon$ gilt. Dann verlangt die Bedingung aus Satz 8.3.1:

$$\mathit{FIRST}_1(\alpha) \cap \mathit{FIRST}_1(\gamma) = \emptyset$$

für alle Linkssatzformen $wA\alpha$. Das läßt sich äquivalent umformulieren in

$$\bigcup \{\mathit{FIRST}_1(\alpha) \mid S \underset{lm}{\overset{*}{\Longrightarrow}} wA\alpha\} \cap \mathit{FIRST}_1(\gamma) = \emptyset$$

Der erste Teil ist aber gleich $\mathit{FOLLOW}_1(A)$. Damit ergibt sich:

Satz 8.3.3 *G ist LL(1) genau dann, wenn gilt:*
Sind $A \to \beta$ und $A \to \gamma$ verschiedene Produktionen, so ist
$FIRST_1(\beta) \oplus_1 FOLLOW_1(A) \cap FIRST_1(\gamma) \oplus_1 FOLLOW_1(A) = \emptyset$.

Korollar 8.3.3.1
G ist genau dann LL(1), wenn für alle Alternativen $A \to \alpha_1 | \ldots | \alpha_n$ gilt:

1. $FIRST_1(\alpha_1), \ldots, FIRST_1(\alpha_n)$ sind paarweise disjunkt; insbesondere enthält höchstens eine der Mengen ϵ, und
2. Aus $\alpha_i \stackrel{*}{\Longrightarrow} \epsilon$ folgt:

   $FIRST_1(\alpha_j) \cap FOLLOW_1(A) = \emptyset$ für $1 \leq j \leq n,\ j \neq i$.

Definition 8.3.3 (starke LL(k)-Grammatik)
Sei $G = (V_N, V_T, P, S)$ eine kontextfreie Grammatik. Wenn für je zwei verschiedene Produktionen $A \to \beta$ und $A \to \gamma$ eines Nichtterminals A gilt:

$$FIRST_k(\beta) \oplus_k FOLLOW_k(A) \cap FIRST_k(\gamma) \oplus_k FOLLOW_k(A) = \emptyset,$$

dann heißt G **starke LL(k)-Grammatik.** □

Bemerkung:

- Es gibt kontextfreie Grammatiken, welche LL(k)-Grammatiken, aber nicht starke LL(k)-Grammatiken sind, d.h. die Bedingung aus Satz 8.3.3 kann nicht von 1 auf k, $k > 1$, verallgemeinert werden. Der Grund dafür ist, daß $FOLLOW_k(A)$ die Folgeworte aus **allen** Linkssatzformen mit A enthält; in der LL(k)-Bedingung treten nur Folgeworte in **einer** Linkssatzform auf.
- Jede LL(1)-Grammatik ist stark (Satz 8.3.3).

Beispiel 8.3.6
G sei die kontextfreie Grammatik mit den Produktionen
$S \to aAaa \mid bAba$
$A \to b \mid \epsilon$
G ist LL(2).

1.Fall Die Ableitung fängt mit $S \Rightarrow aAaa$ an.
$FIRST_2(baa) \cap FIRST_2(aa) = \emptyset$

2.Fall Die Ableitung fängt mit $S \Rightarrow bAba$ an.
$FIRST_2(bba) \cap FIRST_2(ba) = \emptyset$

Also ist G nach Satz 8.3.1 LL(2). G ist nicht stark LL(2);
denn $FIRST_2(b\ FOLLOW_2(A)) \cap FIRST_2(\epsilon\ FOLLOW_2(A)) =$
$FIRST_2\{baa, bba\} \cap FIRST_2\{aa, ba\} = \{ba\}$. □

Also ist *FOLLOW* zu undifferenziert, da es die terminalen Folgewörter zusammenfaßt, die in **allen** Satzformen möglich sind.

Definition 8.3.4 (linksrekursiv)
Sei G eine kontextfreie Grammatik. Eine Produktion von G heißt **direkt rekursiv**, wenn sie die Form $A \to \alpha A \beta$ hat. Sie heißt **direkt linksrekursiv**, wenn $\alpha = \epsilon$, **direkt rechtsrekursiv**, wenn $\beta = \epsilon$ ist. Ein Nichtterminal A heißt **rekursiv**, wenn es eine Ableitung $A \stackrel{+}{\Longrightarrow} \alpha A \beta$ gibt. A heißt **linksrekursiv**, wenn $\alpha = \epsilon$, **rechtsrekursiv**, wenn $\beta = \epsilon$ ist. Eine kontextfreie Grammatik G heißt **linksrekursiv**, wenn G mindestens ein linksrekursives Nichtterminal enthält. □

Satz 8.3.4 *Sei G eine kontextfreie Grammatik.*

(a) *Ist G linksrekursiv, so ist G nicht LL(k) für jedes k.*

(b) *Ist G LL(k)-Grammatik, dann ist G nicht mehrdeutig.*

Beweis:
zu (a): G ist linksrekursiv; also gibt es mindestens ein linksrekursives Nichtterminal X. Zur Vereinfachung nehmen wir an, daß X eine direkt linksrekursive Produktion hat. Es gelte $X \to X\alpha$. Da G nach impliziter Voraussetzung reduziert ist, gibt es zusätzlich auch noch eine Produktion $X \to \beta$. Tritt in einer Linkssatzform X auf, d.h. $S \stackrel{*}{\underset{lm}{\Longrightarrow}} wX\gamma$, so kann beliebig oft die Alternative $X \to X\alpha$ angewandt werden. Man erhält $S \stackrel{*}{\underset{lm}{\Longrightarrow}} wX\gamma \stackrel{n}{\underset{lm}{\Longrightarrow}} wX\alpha^n\gamma$. Nehmen wir an, G sei LL(k). Dann ist nach Satz 8.3.1 $FIRST_k(X\alpha^{n+1}\gamma) \cap FIRST_k(\beta\alpha^n\gamma) = \emptyset$. Wegen $X \to \beta$ ist $FIRST_k(\beta\alpha^{n+1}\gamma) \subseteq FIRST_k(X\alpha^{n+1}\gamma)$. Also ist auch $FIRST_k(\beta\alpha^{n+1}\gamma) \cap FIRST_k(\beta\alpha^n\gamma) = \emptyset$. Produziert α ϵ, so erhalten wir sofort den Widerspruch. Produziert α nicht ϵ, so wählen wir $n \geq k$ und erhalten ebenfalls einen Widerspruch. Also ist G nicht LL(k). Der allgemeine Fall indirekter Linksrekursion ist nicht viel schwerer. Er bleibt dem Leser überlassen.
zu (b): folgt aus der Definition von LL(k). □

Welche Konsequenzen hat es, daß ein eventuell vorhandener LL(k)-Parser-Generator keine linksrekursiven Grammatiken akzeptiert? Der Benutzer muß seine Grammatik so transformieren, daß sie nicht mehr linksrekursiv ist und die sich ergebende Grammatik LL(k) wird. Diese Transformation ist immer möglich, kann auch automatisch durchgeführt werden, hat aber einige gravierende Nachteile:

1. die Größe der Grammatik wächst stark,
2. die Struktur der Grammatik wird eventuell stark verändert.

Beispiel 8.3.7
Wir betrachten zwei Varianten der Ausdrucksgrammatik.

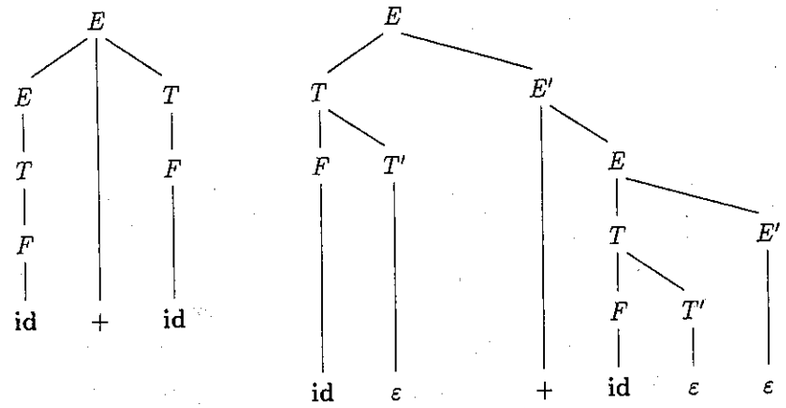
G_0: $E \to E+T \mid T$, $T \to T*F \mid F$, $F \to (E) \mid \mathbf{id}$

3 Nichtterminale, 6 Produktionen

G_1: $E \to TE'$, $E' \to \varepsilon \mid +E$, $T \to FT'$, $T' \to \varepsilon \mid *T$, $F \to (E) \mid \mathbf{id}$

5 Nichtterminale, 8 Produktionen

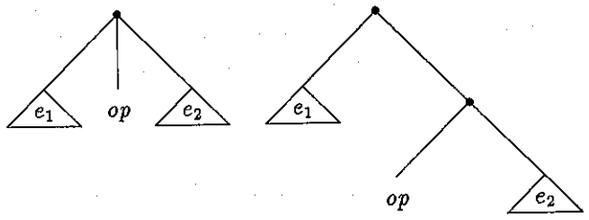
Der Syntaxbaum für **id** + **id** gemäß G_0 ist in Abbildung 8.14 (a), der gemäß G_1 in Abbildung 8.14 (b) dargestellt.



(a) (b)

Abb. 8.14: Syntaxbaum für **id** + **id**

Wie man im Beispiel sieht, ist die Baumstruktur zu einem Teilausdruck e_1 *op* e_2 in G_0 wie in Abbildung 8.15(a) und in G_1 wie in Abbildung 8.15(b) dargestellt.



(a) (b)

Abb. 8.15:

Die Struktur zu G_1 ist (etwas) ungünstiger für die Behandlung der Semantik, für die Codeerzeugung usw., da der Operator *op* nicht so günstig zu seinen Operanden steht wie in der Originalgrammatik. Deshalb ist die Frage, ob es eine Alternative zu dieser Eliminierung von Linksrekursion gibt. Meist werden in Übersetzern statt der Syntaxbäume zur kfG der Programmiersprache, der sogenannten konkreten Syntax, kompaktere Bäume benutzt, die auf die wesentliche Struktur reduziert sind. Diese „abstrakte" Syntax könnte für G_0 und G_1 ähnlich oder gleich aussehen.

Betrachten wir, wozu Linksrekursion benutzt wird; meist für Auflistungen, etwa von Parameterspezifikationen in einer Prozedurdeklaration oder aktuellen Parametern in einem Prozeduraufruf, von Indexausdrücken in einem Feldzugriff, von Anweisungen in einer Anweisungsfolge oder von arithmetischen Teilausdrücken, die durch den gleichen (assoziativen) Operator verknüpft sind. Diese Fälle können aber auch durch reguläre Ausdrücke beschrieben werden. Deshalb werden wir im übernächsten Abschnitt reguläre Ausdrücke auf der rechten Seite kontextfreier Produktionen erlauben, $FIRST_1$- und $FOLLOW_1$-Berechnungen dafür beschreiben und Parser sowie Parser-Generator dafür angeben.

8.3.4 (Starke) LL(k)-Parser

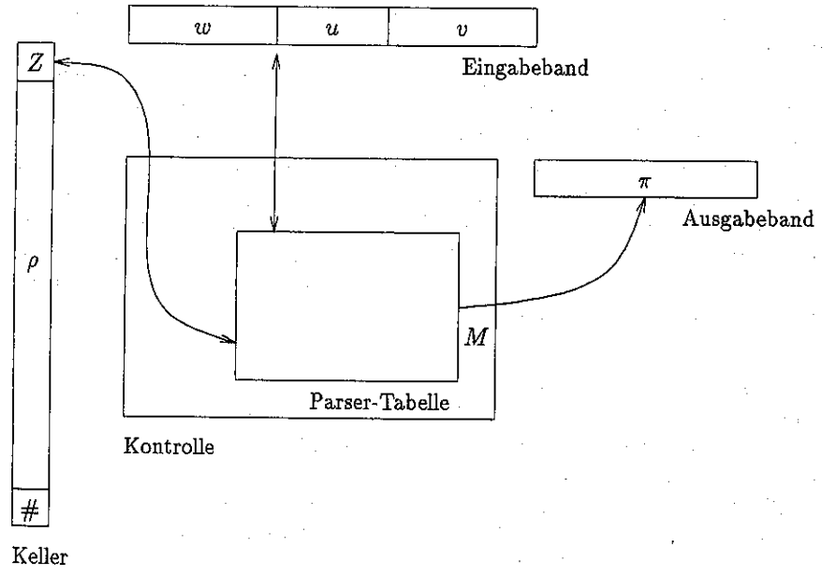


Abb. 8.16: LL-Parser

Die Struktur eines Parsers für (starke) LL(k)-Grammatiken ist in Abbildung

8.16 dargestellt. Von der Eingabe auf dem Eingabeband ist der Präfix w bereits gelesen. Die restliche Eingabe beginnt mit einem Präfix u der Länge k. Der Keller enthält über einem Kellerendezeichen $\#$ eine Folge von Items der kontextfreien Grammatik. Das oberste Item, der aktuelle Zustand Z, bestimmt, ob als nächstes

- das nächste Eingabesymbol gelesen,
- auf das Ende der Analyse getestet oder
- das aktuelle Nichtterminal expandiert werden soll.

Im letzteren Fall zieht der Parser die Parsertabelle zu Rate, um mithilfe der k nächsten Symbole die richtige Alternative für das Nichtterminal auszuwählen. Die Parsertabelle M ist ein 2-dimensionales Feld, deren Zeilen durch Nichtterminale und deren Spalten durch Wörter der Länge k indiziert werden. Sie stellt eine Auswahlfunktion $m : V_N \times V_{T\#}^{\leq k} \to P \cup \{error\}$ dar. Sei $[X \to \beta.Y\gamma]$ das oberste Kelleritem und u der Präfix der Länge k der restlichen Eingabe. $m(Y,u) = Y \to \alpha$ hat zur Folge, daß $[Y \to .\alpha]$ neues oberstes Kelleritem und $(Y \to \alpha)$ auf das Ausgabeband geschrieben wird. $m(Y,u) = error$ heißt, daß in dieser Situation aktuelles Nichtterminal und Präfix der restlichen Eingabe nicht zusammenpassen. Ein syntaktischer Fehler liegt vor. Deshalb wird eine Fehlerdiagnose- und -behandlungsroutine gestartet, die eine Fortsetzung der Analyse ermöglichen soll. Solche Verfahren werden im Abschnitt „Fehlerbehandlung in LL(k)-Parsern“ beschrieben.

Jetzt werden wir uns auf den praktisch relevanten Fall, $k = 1$, einschränken. Ein **LL(1)-Parsergenerator** erwartet als Eingabe eine kontextfreie Grammatik G. Er berechnet für die Nichtterminale von G die $FIRST_1$- und $FOLLOW_1$-Mengen und führt mit ihrer Hilfe den LL(1)-Test gemäß Satz 8.3.3 durch. Ist der Testausgang positiv, d.h. ist G eine LL(1)-Grammatik, so wird mit dem jetzt folgenden Algorithmus die Parsertabelle erzeugt. Lehnt der Parsergenerator G ab, d.h. ist G keine LL(1)-Grammatik, so muß der Benutzer versuchen, durch Transformation von G, etwa durch Ausfaktorisierung, eine LL(1)-Grammatik für die gleiche Sprache zu erhalten.

Algorithmus LL(1)-GEN
Eingabe: LL(1)-Grammatik G, $FIRST_1$ und $FOLLOW_1$ für G.
Ausgabe: Parsertabelle M für LL(1)-Parser für G.
Methode: M wird folgendermaßen aufgebaut:
Für alle $X \to \alpha \in P$ und für alle Terminalsymbole $a \in FIRST_1(\alpha)$ wird $M[X,a] = (X \to \alpha)$ gesetzt. Falls $eps(\alpha) = true$, dann wird $M[X,b] = (X \to \alpha)$ gesetzt für alle $b \in FOLLOW_1(X)$.
Alle übrigen Einträge von M werden auf $error$ gesetzt.

Eine mögliche Alternative für den LL(1)-Test gemäß Satz 8.3.3 bietet der Algorithmus LL(1)-GEN an: Wenn eine Komponente $M[X,a]$ im Laufe der Besetzung zwei verschiedene Einträge bekommt, so ist die Grammatik nicht LL(1).

Beispiel 8.3.8
In Tabelle 8.3 ist die LL(1)-Parsertabelle für die Grammatik aus Beispiel 8.2.18 dargestellt, in Tabelle 8.4 ein Lauf des zugehörigen Parsers für die Eingabe **id** $*$ **id**#. □

Tabelle 8.3: LL(1)-Parsertabelle für die Grammatik aus Beispiel 8.2.18

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

Tabelle 8.4: Parserlauf für die Eingabe: **id** $*$ **id**#

Kellerinhalt	Eingabe
$\#[S \to .E]$	**id** $*$ **id**#
$\#[S \to .E][E \to .TE']$	**id** $*$ **id**#
$\#[S \to .E][E \to .TE'][T \to .FT']$	**id** $*$ **id**#
$\#[S \to .E][E \to .TE'][T \to .FT'][F \to .\mathbf{id}]$	**id** $*$ **id**#
$\#[S \to .E][E \to .TE'][T \to .FT'][F \to \mathbf{id}.]$	$*$**id**#
$\#[S \to .E][E \to .TE'][T \to F.T']$	$*$**id**#
$\#[S \to .E][E \to .TE'][T \to F.T'][T' \to .*T]$	$*$**id**#
$\#[S \to .E][E \to .TE'][T \to F.T'][T' \to *.T]$	**id**#
$\#[S \to .E][E \to .TE'][T \to F.T'][T' \to *.T][T \to .FT']$	**id**#
$\#[S \to .E][E \to .TE'][T \to F.T'][T' \to *.T][T \to .FT'][F \to .\mathbf{id}]$	**id**#
$\#[S \to .E][E \to .TE'][T \to F.T'][T' \to *.T][T \to .FT'][F \to \mathbf{id}.]$	#
$\#[S \to .E][E \to .TE'][T \to F.T'][T' \to *.T][T \to F.T']$	#
$\#[S \to .E][E \to .TE'][T \to F.T'][T' \to *.T][T \to F.T'][T' \to \varepsilon.]$	#
$\#[S \to .E][E \to .TE'][T \to F.T'][T' \to *.T][T \to FT'.]$	#
$\#[S \to .E][E \to .TE'][T \to F.T'][T' \to *T.]$	#
$\#[S \to .E][E \to .TE'][T \to FT'.]$	#
$\#[S \to .E][E \to T.E']$	#
$\#[S \to .E][E \to T.E'][E' \to \varepsilon.]$	#
$\#[S \to .E][E \to TE'.]$	#
$\#[S \to E.]$	#
#	#

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

LL(1)-Parser in Prolog

Der in Prolog geschriebene top down-Parser aus Abschnitt 8.3.2 wird mithilfe berechneter $FIRST_1$- und $FOLLOW_1$- Mengen deterministisch gemacht. Die Parsertabelle wird in Form einer Menge von Fakten dargestellt. Ein Eintrag $M[X, a] = (X \to \alpha)$ wird durch das Faktum $parstab(t(a), n(x), <$Liste der Elemente von $\alpha>)$ dargestellt. Der Parsertreiber wird geändert zu:

```
predict([],[]).
predict( [ t(A) | Input], [n(X) | Stack]) :-
   parstab(t(A),n(X), Rhs),
   append( Rhs, Stack, Newstack),
   predict( [t(A) | Input], Newstack).
predict( [ t(A) | Restinput], [t(A) | Stack] ) :-
   predict( Restinput, Stack).
```

Beispiel 8.3.9
Für die Grammatik aus Beispiel 8.2.18 wurden in den Beispielen 8.2.19 und 8.2.20 die $FIRST_1$- und die $FOLLOW_1$-Mengen berechnet. Damit können wir die Parsertabelle für diese Grammatik als Folge von Prolog-Fakten aufstellen:

```
parstab( t(id),    n(s),     [n(expr),t('#')]).
parstab( t('('),   n(s),     [n(expr),t('#')]).
parstab( t(id),    n(expr),  [n(term), n(expr1)]).
parstab( t('('),   n(expr),  [n(term), n(expr1)]).
parstab( t('+'),   n(expr1), [t('+'), n(expr)]).
parstab( t(')'),   n(expr1), []).
parstab( t('#'),   n(expr1), []).
parstab( t(id),    n(term),  [n(factor), n(term1)]).
parstab( t('('),   n(term),  [n(factor), n(term1)]).
parstab( t('*'),   n(term1), [t('*'), n(term)]).
parstab( t('+'),   n(term1), []).
parstab( t('#'),   n(term1), []).
parstab( t(')'),   n(term1), []).
parstab( t('('),   n(factor),[t('('), n(expr), t(')')]).
parstab( t(id),    n(factor),[t(id)]).
```

Jetzt ergibt die Anfrage

```
?- predict([t(id), t('*'), t(id), t(#)], [n(s)]).
```

den Trace:

```
C-Prolog version 1.5

Spy-point placed on predict/2.
```



```
Debug mode switched on.
No leashing.
*  (4)  2 Call:  predict([t(id),t(*),t(id),t(#)],[n(s)])
*  (9)  3 Call:  predict([t(id),t(*),t(id),t(#)],[n(expr),
                 t(#)])
*  (14) 4 Call:  predict([t(id),t(*),t(id),t(#)],[n(term),
                 n(expr1), t(#)])
*  (19) 5 Call:  predict([t(id),t(*),t(id),t(#)],[n(factor),
                 n(term1),n(expr1),t(#)])
*  (23) 6 Call:  predict([t(id),t(*),t(id),t(#)],[t(id),
                 n(term1), n(expr1),t(#)])
*  (24) 7 Call:  predict([t(*),t(id),t(#)],[n(term1),n(expr1),
                 t(#)])
*  (29) 8 Call:  predict([t(*),t(id),t(#)],[t(*),n(term),
                 n(expr1), t(#)])
*  (30) 9 Call:  predict([t(id),t(#)],[n(term),n(expr1),
                 t(#)])
*  (35) 10 Call: predict([t(id),t(#)],[n(factor),n(term1),
                 n(expr1), t(#)])
*  (39) 11 Call: predict([t(id),t(#)],[t(id),n(term1),n(expr1),
                 t(#)])
*  (40) 12 Call: predict([t(#)],[n(term1),n(expr1),t(#)])
*  (43) 13 Call: predict([t(#)],[n(expr1),t(#)])
*  (46) 14 Call: predict([t(#)],[t(#)])
*  (47) 15 Call: predict([],[])
*  (47) 15 Exit: predict([],[])
*  (46) 14 Exit: predict([t(#)],[t(#)])
*  (43) 13 Exit: predict([t(#)],[n(expr1),t(#)])
```

weitere erfolgreiche Anwendungen werden verlassen

```
*  (4) 2 Exit: predict([t(id),t(*),t(id),t(#)],[n(s)])
```

```
yes
```

Der entscheidende Unterschied zum Rücksetzparser in 8.3.1 ist im Schritt (29) zu sehen. Hier benutzt der Parser die Information aus Tabelle *parstab*, um für *term1* die richtige Alternative $term1 \to *\, term$ auszuwählen.

8.3.5 LL-Parser für rechtsreguläre kontextfreie Grammatiken

Linksrekursive Nichtterminale zerstören die LL-Eigenschaft von kontextfreien Grammatiken. Wir werden jetzt reguläre Ausdrücke auf den rechten Seiten von Produktionen zulassen, um einen Ersatz für die Linksrekursion in Auflistungen zu haben. Dies sind die häufigsten Verwendungen von Linksrekursion bei der Beschreibung von Programmiersprachen.

Definition 8.3.5 (rechtsreguläre kontextfreie Grammatik)
Eine **rechtsreguläre kontextfreie Grammatik** (rrkfG) ist ein Tupel $G = (V_N, V_T, p, S)$, wobei V_N, V_T, S wie üblich definiert sind, und $p : V_N \to RA$ eine Abbildung der Nichtterminale in die Menge RA der regulären Ausdrücke über $V_N \cup V_T$ ist. □

Es ist möglich, p als eine Abbildung zu sehen, da mithilfe des Alternativoperators verschiedene Alternativen für ein Nichtterminal in einem regulären Ausdruck zusammengefaßt werden können.

Beispiel 8.3.10
Eine rrkfG für arithmetische Ausdrücke ist

$$G_e = (\{S, E, T, F\}, \{\mathbf{id}, (,), +, -, *, /\}, p, S),$$

wobei p die folgende Abbildung ist ('{' und '}' werden als Metazeichen benutzt, um den Konflikt mit den Terminalsymbolen '(' und ')' zu vermeiden):
$S \to E$
$E \to T\{\{+|-\}T\}^*$
$T \to F\{\{*|/\}F\}^*$
$F \to (E) \,|\, \mathbf{id}$ □

Definition 8.3.6 (reguläre Ableitung)
Sei G eine rechtsreguläre kontextfreie Grammatik. Die Relation $\underset{R,lm}{\Longrightarrow}$ auf RA, „leitet regulär direkt links ab", ist definiert durch:

(a)	$w\,X\,\beta$	$\underset{R,lm}{\Longrightarrow}$	$w\,\alpha\,\beta$	mit $\alpha = p(X)$
(b)	$w\,(r_1 \,|\, \ldots \,|\, r_n)\,\beta$	$\underset{R,lm}{\Longrightarrow}$	$w\,r_i\,\beta$	für $1 \le i \le n$
(c)	$w\,(r)^*\,\beta$	$\underset{R,lm}{\Longrightarrow}$	$w\,\beta$	
(d)	$w\,(r)^*\,\beta$	$\underset{R,lm}{\Longrightarrow}$	$w\,r\,(r)^*\,\beta$	

$\overset{*}{\underset{R,lm}{\Longrightarrow}}$ sei die reflexive, transitive Hülle von $\underset{R,lm}{\Longrightarrow}$. Die von G definierte Sprache ist dann $L(G) = \{w \in V_T^* \mid S \overset{*}{\underset{R,lm}{\Longrightarrow}} w\}$ □

Beispiel 8.3.11
Eine reguläre Linksableitung zum Wort $\mathbf{id} + \mathbf{id} * \mathbf{id}$ der Grammatik G_e aus Beispiel 8.3.10 ist:
$S \underset{R,lm}{\Longrightarrow} E \underset{R,lm}{\Longrightarrow} T\{\{+|-\}T\}^*$
$\underset{R,lm}{\Longrightarrow} F\{\{*|/\}F\}^*\{\{+|-\}T\}^*$
$\underset{R,lm}{\Longrightarrow} \{(E)|\mathbf{id}\}\{\{*|/\}F\}^*\{\{+|-\}T\}^*$
$\underset{R,lm}{\Longrightarrow} \mathbf{id}\{\{*|/\}F\}^*\{\{+|-\}T\}^*$

$\underset{R,lm}{\Longrightarrow} \mathbf{id}\{\{+|-\}T\}^*$
$\underset{R,lm}{\Longrightarrow} \mathbf{id}\{+|-\}T\{\{+|-\}T\}^*$
$\underset{R,lm}{\Longrightarrow} \mathbf{id} + T\{\{+|-\}T\}^*$
$\underset{R,lm}{\Longrightarrow} \mathbf{id} + F\{\{*|/\}F\}^*\{\{+|-\}T\}^*$
$\underset{R,lm}{\Longrightarrow} \mathbf{id} + \{(E)|\mathbf{id}\}\{\{*|/\}F\}^*\{\{+|-\}T\}^*$
$\underset{R,lm}{\Longrightarrow} \mathbf{id} + \mathbf{id}\{\{*|/\}F\}^*\{\{+|-\}T\}^*$
$\underset{R,lm}{\Longrightarrow} \mathbf{id} + \mathbf{id}\{*|/\}F\{\{*|/\}F\}^*\{\{+|-\}T\}^*$
$\underset{R,lm}{\Longrightarrow} \mathbf{id} + \mathbf{id} * F\{\{*|/\}F\}^*\{\{+|-\}T\}^*$
$\underset{R,lm}{\Longrightarrow} \mathbf{id} + \mathbf{id} * \{(E)|\mathbf{id}\}\{\{*|/\}F\}^*\{\{+|-\}T\}^*$
$\underset{R,lm}{\Longrightarrow} \mathbf{id} + \mathbf{id} * \mathbf{id}\{\{*|/\}F\}^*\{\{+|-\}T\}^*$
$\underset{R,lm}{\Longrightarrow} \mathbf{id} + \mathbf{id} * \mathbf{id}\{\{+|-\}T\}^*$
$\underset{R,lm}{\Longrightarrow} \mathbf{id} + \mathbf{id} * \mathbf{id}$ □

Unser Ziel ist es, einen RLL-Parser, also einen deterministischen top down-Parser für rechtsreguläre kontextfreie Grammatiken zu entwickeln. Dieser wird für eine korrekte Eingabe eine reguläre Linksableitung erstellen. Bei der obigen Definition fällt auf, daß jetzt der Expansionfall (a) – ein Nichtterminal wird durch seine (einzige) rechte Seite ersetzt – nicht mehr kritisch ist. Anstelle dessen sind die Fälle (b), (c) und (d) jetzt deterministisch zu machen. Wir werden also einen Parser für eine rechtsreguläre kontextfreie Grammatik einen RLL(1)-Parser nennen, wenn er bei Vorliegen einer regulären Linkssatzform $w(r_1 \,|\, \ldots \,|\, r_n)\beta$ die Entscheidung für die richtige der Alternativen und bei Vorliegen einer Linkssatzform $w(r)^*\beta$ die Entscheidung für die Fortsetzung oder Beendigung der Iteration mithilfe des nächsten Eingabesymbols der restlichen Eingabe treffen kann. Wir übertragen jetzt einige Begriffe auf den Fall der rechtsregulären kontextfreien Grammatiken.

Definition 8.3.7 (regulärer Unterausdruck)
r_i, $1 \le i \le n$, ist **direkter regulärer Unterausdruck** von $(r_1 \,|\, \ldots \,|\, r_n)$ und $(r_1 \ldots r_n)$; r ist **direkter regulärer Unterausdruck** von $(r)^*$ und von r selbst; r_1 ist **regulärer Unterausdruck** von r_2, wenn $r_1 = r_2$ oder wenn r_1 direkter regulärer Unterausdruck von r_2 oder regulärer Unterausdruck eines direkten regulären Unterausdrucks von r_2 ist. □

Definition 8.3.8 (erweitertes kontextfreies Item)
Ein Tupel $(X, \alpha, \beta, \gamma)$ ist ein **erweitertes kontextfreies Item** einer rrkfG $G = (V_N, V_T, p, S)$, wenn gilt $X \in V_N$, $\alpha, \beta, \gamma \in (V_N \cup V_T \cup \{(,), {}^*, |, \varepsilon\})^*$, $p(X) = \beta\alpha\gamma$ und α ist regulärer Unterausdruck von $\beta\alpha\gamma$. Dieses Item wird geschrieben als $[X \to \beta.\alpha\gamma]$. □

Die Generierung eines RLL(1)-Parsers aus einer rrkfG benutzt wieder $FIRST_1$- und $FOLLOW_1$-Mengen, und zwar von regulären Unterausdrücken rechter Produktionenseiten.

$FIRST_1$- und $FOLLOW_1$-Berechnung für rechtsreguläre kontextfreie Grammatiken

Die Berechnungen von $FIRST_1$- und $FOLLOW_1$ für rrkfGs lassen sich wieder als reine Vereinigungsprobleme darstellen und deshalb effizient erledigen. Genau wie im konventionellen Fall geht ihnen die Berechnung der ε-Produktivität voraus. Die Gleichungen für die ε-Produktivität lassen sich über die Struktur der regulären Ausdrücke definieren. Zusätzlich überträgt sich die ε-Produktivität rechter Seiten auf das Nichtterminal der linken Seite.

$$
\begin{array}{l}
eps(a) = false, \quad \text{für } a \in V_T \\
eps(\varepsilon) = true \\
eps(r^*) = true \\
eps(X) = eps(r), \text{ falls } p(X) = r \text{ für } X \in V_N \\
eps((r_1|\ldots|r_n)) = \bigvee_{i=1}^{n} eps(r_i) \\
eps((r_1 \ldots r_n)) = \bigwedge_{i=1}^{n} eps(r_i)
\end{array}
\qquad (eps)
$$

Beispiel 8.3.12 (Fortführung von Beispiel 8.3.10)
Für alle Nichtterminale von G_e gilt: $eps(X) = false$ □

Wenn die ε-Produktivität berechnet ist, kann man jetzt die ε-freie first-Funktion berechnen. Dafür ergeben sich die folgenden Gleichungen.

$$
\begin{array}{l}
\varepsilon\text{-}ffi(\varepsilon) = \emptyset \\
\varepsilon\text{-}ffi(a) = \{a\} \\
\varepsilon\text{-}ffi(r^*) = \varepsilon\text{-}ffi(r) \\
\varepsilon\text{-}ffi(X) = \varepsilon\text{-}ffi(r), \text{ falls } p(X) = r \\
\varepsilon\text{-}ffi((r_1|\ldots|r_n)) = \bigcup_{1 \le i \le n} \varepsilon\text{-}ffi(r_i) \\
\varepsilon\text{-}ffi((r_1 \ldots r_n)) = \bigcup_{1 \le j \le n} \{\varepsilon\text{-}ffi(r_j) \mid \bigwedge_{1 \le i < j} eps(r_i)\}
\end{array}
\qquad (\varepsilon\text{-}ffi)
$$

Beispiel 8.3.13 (Fortführung von Beispiel 8.3.10)
Die ε-*ffi*- und deshalb auch die $FIRST_1$-Mengen für die Nichtterminale der Grammatik G_e sind
$FIRST_1(S) = FIRST_1(E) = FIRST_1(T) = FIRST_1(F) = \{(, \mathrm{id}\}$ □

ε-Produktivität und ε-freie first-Funktionen ließen sich induktiv über die Struktur von regulären Ausdrücken definieren. Die $FIRST_1$-Menge eines regulären Ausdrucks ist unabhängig davon, in welchem Kontext er steht. Für die $FOLLOW_1$-Menge ist dies anders; zwei verschiedene Vorkommen eines regulären (Unter-) Ausdrucks haben i.a. verschiedene $FOLLOW_1$- Mengen. Wir sind für die Generierung von RLL(1)-Parsern an den $FOLLOW_1$-Mengen von Vorkommen von regulären (Unter-) Ausdrücken interessiert. Ein bestimmtes Vorkommen eines regulären Ausdrucks in einer rechten Seite entspricht aber genau einem erweiterten regulären Item, in dem der Punkt vor diesem regulären Ausdruck steht.

Bei den folgenden Gleichungen für $FOLLOW_1$ wird vorausgesetzt, daß Konkatenationen und Alternativlisten außen geklammert und innen ohne überflüssige Klammern sind.

(1) $FOLLOW_1([S' \to .S]) = \{\#\}$ Das Endesymbol '#' folgt auf jeden Satz.

(2) $FOLLOW_1([X \to \cdots(r_1|\cdots|.r_i|\cdots|r_n)\cdots]) =$
$\quad FOLLOW_1([X \to \cdots.(r_1|\cdots|r_i|\cdots|r_n)\cdots])$ für $1 \le i \le n$

(3) $FOLLOW_1([X \to \cdots(\cdots.r_i r_{i+1}\cdots)\cdots]) =$
$\quad \varepsilon\text{-}ffi(r_{i+1}) \cup \begin{cases} FOLLOW_1([X \to \cdots(\cdots r_i.r_{i+1}\cdots)\cdots]), \\ \quad \text{falls } eps(r_{i+1}) = true \\ \emptyset \quad \text{sonst} \end{cases}$

(4) $FOLLOW_1([X \to \cdots(r_1 \cdots r_{n-1}.r_n)\cdots]) =$ $\qquad (FOLLOW_1)$
$\quad FOLLOW_1([X \to \cdots.(r_1 \cdots r_{n-1} r_n)\cdots])$

(5) $FOLLOW_1([X \to \cdots(.r)^*\cdots]) =$
$\quad \varepsilon\text{-}ffi(r) \cup FOLLOW_1([X \to \cdots.(r)^*\cdots])$

(6) $FOLLOW_1([X \to .r]) = \bigcup FOLLOW_1([Y \to \cdots.X\cdots])$

Beispiel 8.3.14 (Fortführung von Beispiel 8.3.10)
Die $FOLLOW_1$-Mengen für einige Items zur Grammatik G_e ergeben sich zu:
$FOLLOW_1([S \to .E]) = \{\#\}$
$FOLLOW_1([E \to T.\{\{+|-\}T\}^*]) \overset{(4)}{=}$
$\quad FOLLOW_1([E \to .T\{\{+|-\}T\}^*]) \overset{(6)}{=}$
$\quad FOLLOW_1([S \to .E]) \cup FOLLOW_1([F \to (.E)]) =$
$\quad (\{\#\} \cup FOLLOW_1([F \to (.E)])) \overset{(3)}{=} \{), \#\}$
$FOLLOW_1([T \to F.\{\{*|/\}F\}^*]) = \{+, -,), \#\}$ □

Um die Berechnung der Lösungen von ε-*ffi* und $FOLLOW_1$ möglichst effizient, sprich linear zu machen, müssen diese Gleichungssysteme wieder in die Form

$$f(X) = g(X) \cup \bigcup\{f(Y) \mid X \; R \; Y\}$$

mit einer bekannten mengenwertigen Funktion g und einer binären Relation R gebracht werden.

Im Falle der Berechnung von ε-*ffi* ist die Grundmenge von R bzw. die Knotenmenge des von R induzierten gerichteten Graphen die Menge der in den Produktionen auftretenden regulären (Unter-) Ausdrücke. Eine gerichtete Kante von X nach Y existiert genau dann, wenn entweder Y ein direkter Unterausdruck von X ist und Y zur $FIRST_1$-Menge von X beiträgt, oder wenn X ein Nichtterminal (-vorkommen) und Y seine rechte Seite ist. Die Funktion g ist nur für den Fall eines terminalen Symbols als nichtleer definiert.

Im Falle der Berechnung von $FOLLOW_1$ ist die Grundmenge die Menge der erweiterten Items, und die Relation bringt wieder solche Items j zu einem Item