konstruieren. Die sich ergebenden LR(k)-Parser ähneln sich insofern, als ihre Zustände jeweils aus Mengen von LR(k)-Items der Grammatik bestehen, unterscheiden sich aber i.a. in der Zahl der Zustände und/oder in der Größe der Vorausschaumengen. Es ist zu erwarten, daß man mit mehr Zuständen genauere Information halten kann. Mächtigere Verfahren, die für eine größere Klasse von Grammatiken LR(k)-Parser erzeugen können, konstruieren deshalb i.a. Parser mit größeren Zustandsmengen. Bei den Vorausschaumengen verhält es sich umgekehrt; die mächtigsten Parsergenerierungsverfahren statten die Items in ihren Zuständen mit kleineren Vorausschaumengen aus. Das sieht man folgendermaßen ein: Jedes Wort u in der Menge L eines LR(k)-Items $[X \to \alpha., L]$ im Zustand q besagt: Ist der Parser im Zustand q und bilden die nächsten k Eingabesymbole das Wort u, so reduziere gemäß $X \to \alpha$. Gibt es ein zweites vollständiges Item $[Y \to \beta., L']$ in q, so müssen L und L' disjunkt sein, damit der Parser deterministisch ist. Je kleiner aber jeweils die beiden Mengen L und L' sind, desto eher sind sie disjunkt. Ähnlich verhält es sich bei gleichzeitig in einem Zustand vorhandenen shift- und reduce-Items.

Das mächtigste Verfahren ist das kanonische LR(k)-Verfahren. Ist eine Grammatik G eine LR(k)-Grammatik, so gibt es einen kanonischen LR(k)-Parser für G. Die anderen beiden Verfahren sind das SLR(k)- und das LALR(k)-Verfahren.

Das Schema für die drei Parsertypen ist also das gleiche: es gibt eine Menge von Zuständen, die allerdings auf verschiedene Weise berechnet werden, und Items in den Zuständen, die (teilweise) mit Mengen von Vorausschauworten der Maximallänge k versehen sind. Aus den Zuständen wird die action-Tabelle aufgebaut. Die Zustände und die Vorausschauworte dienen als Indizes in dieser Tabelle.

Im folgenden beschränken wir uns wieder auf den für die Praxis relevanten Fall $k = 1$. Dieser Fall ist prinzipiell sogar ausreichend, da gilt, daß es zu jeder Sprache, die einen LR(k)-Parser besitzt, sogar einen LR(1)-Parser gibt. Eine nicht-LR(1)-Grammatik für die Sprache muß allerdings in eine LR(1)-Grammatik transformiert werden.

Was ändert sich am Algorithmus LR(k)-PARSER? Die Variable *lookahead* hat jetzt den Typ *symbol*, und die Prozedur *scan* analysiert die weitere Eingabe und speichert das nächste Symbol in *lookahead* ab.

Enthält ein LR(1)-Parserzustand mehrere vollständige Items, so liegt trotzdem kein reduce-reduce-Konflikt vor, wenn ihre Vorausschaumengen paarweise disjunkt sind. Der Parser wird dann, wenn er in diesem Zustand ist, die Reduktion auswählen, in deren Vorausschaumenge das nächste Eingabesymbol liegt.

Enthält ein Parserzustand gleichzeitig ein vollständiges Item $[X \to \alpha., L]$ und ein shift-Item $[Y \to \beta.a\gamma, L']$, so liegt kein shift-reduce-Konflikt zwischen ihnen vor, wenn L das Symbol a nicht enthält. In diesem Fall wird der erzeugte Parser, wenn er in diesem Zustand ist, reduzieren, wenn das nächste Eingabesymbol in L ist, und lesen, wenn es gleich a ist. In allen Situationen, in denen das nächste Eingabesymbol weder in der Vorausschaumenge eines vollständigen Items noch das zu lesende Symbol eines shift-Items des aktuellen Zustandes ist, muß der Parser einen Syntaxfehler melden.

Definition 8.4.8 (Konflikt)
Sei I eine Menge von LR(1)-Items. I hat einen **shift-reduce-Konflikt**, wenn es ein Item $[X \to \alpha.a\beta, L_1]$ und ein Item $[Y \to \gamma., L_2]$ enthält, und $a \in L_2$ ist. I hat einen **reduce-reduce-Konflikt**, wenn es zwei Items $[X \to \alpha., L_1]$ und $[Y \to \beta., L_2]$ mit $L_1 \cap L_2 \neq \emptyset$ enthält. Tritt eine Itemmenge mit einem Konflikt bei der Konstruktion eines LR(1)-Parsers als Zustand auf, so heißt dieser Zustand **ungeeignet.** □

Die Konstruktion eines LR(1)-Parsers ist erfolgreich, wenn dabei keine ungeeigneten Zustände erzeugt werden.

Konstruktion einer LR(1)-action-Tabelle

Der letzte Schritt in der Konstruktion eines LR(1)-Parsers ist die Konstruktion der action-Tabelle. Er ist identisch für die verschiedenen Typen von LR-Parsern. Die Eingabe für diesen Schritt ist die LR(1)-Zustandsmenge. Diese ist i.a. für die gleiche Ausgangsgrammatik abhängig von der gewählten LR-Analysemethode.

Algorithmus LR(1)-Tabelle:
Eingabe: LR(1)-Zustandsmenge Q
Ausgabe: Action-Tabelle
Methode:

foreach $q \in Q$ **do**
  foreach *LR(1)-Item* $[K, L] \in q$ **do**
    if $K = [S' \to S.]$ **and** $L = \{\#\}$
    then *action*$[q, \#] :=$ *accept*
    elseif $K = [X \to \alpha.]$
    then **foreach** $a \in L$ **do**
      action$[q, a] :=$ *reduce*$(X \to \alpha)$
      od
    elseif $K = [X \to \alpha.a\beta]$
    then *action*$[q, a] :=$ *shift*
    fi
  od
od;
foreach $q \in Q$ **and** $a \in V_T$ *mit* action$[q, a]$ *ist undef.* **do**
  action$[q, a] :=$ *error*
od;

Berechnung von LR(1)-Zustandsmengen

Jetzt werden drei verschiedene Methoden vorgestellt, LR(1)-Zustandsmengen zu berechnen. Wir beginnen mit der Berechnung der kanonischen LR(1)-Zustandsmengen.

Algorithmus LR(1)-GEN:
Eingabe: kontextfreie Grammatik G
Ausgabe: charakteristischer endlicher Automat eines kanonischen LR(1)-Parsers für G.
Methode: Die Zustände und Übergänge des LR(1)-Parsers von G werden schrittweise mithilfe der folgenden drei Hilfsfunktionen *Start, Abschluss* und *Nachf* konstruiert.

```
var q, q' : set of item;
var Q : set of  set of item;
var δ : set of item × (V_N ∪ V_T) → set of item;
function Start: set of item;
   return({[S' → .S, {#}]});
function Abschluss(q : set of item) : set of item;
begin
   foreach [X → α.Yβ, L] in q and Y → γ in P do
      if exist. [Y → .γ, L'] in q
      then ersetze [Y → .γ, L'] durch [Y → .γ, L' ∪ ε-ffi(βL)]
      else q := q ∪ {[Y → .γ, ε-ffi(βL)]}
      fi
   od;
   return(q)
end;
function Nachf(q : set of item, Y : V_N ∪ V_T) : set of item;
   return({[X → αY.β, L] | [X → α.Yβ, L] ∈ q});
begin
   Q := {Abschluss(Start)};  δ := ∅;
   foreach q in Q and X in V_N ∪ V_T do
      let q' = Abschluss(Nachf(q, X)) in
         if q' ≠ ∅
         then
            if q' not in Q
            then Q := Q ∪ {q'}
            fi;
            δ := δ ∪ {q --X--> q'}
         fi
      tel
   od
end.
```

Wir sollten den Algorithmus LR(1)-GEN mit dem Algorithmus LR-DEA in Abschnitt 8.4.3 vergleichen. Was hat sich geändert? In der Hilfsfunktion *Start* konstruieren wir die Menge bestehend aus dem LR(1)-Item $[S' \to .S, \{\#\}]$, wobei

für das Textende- oder Dateiendesymbol steht. Wir nehmen an, daß jede Eingabe von irgendeinem Endesymbol gefolgt wird.

Ein Aufruf *Nachf* (Y, q) erzeugt die Menge aller Y-Nachfolge-Items zu einem Zustand q. Dabei wird die Vorausschaumenge jeweils übernommen. Das ist klar, wenn man die Definition der Gültigkeit von LR(k)-Items betrachtet. Ist $[X \to \alpha.Y\beta, L]$ gültig für ein beliebiges zuverlässiges Präfix $\gamma\alpha$, so ist $[X \to \alpha Y.\beta, L]$ gültig für das zuverlässige Präfix $\gamma\alpha Y$.

Die Funktion *Abschluss* berechnet neue Vorausschaumengen für die LR(1)-Items, die sie zu ihrem Argument hinzufügt. Ist nämlich $[X \to \alpha.Y\beta, L]$ gültig für das beliebige zuverlässige Präfix $\delta\alpha$ und ist $Y \to \gamma$ eine Alternative für Y, so muß das hinzugefügte Item mit Kern $[Y \to .\gamma]$ auch für $\delta\alpha$ gültig sein. Dann kann in einer Rechtssatzform jedes Symbol auf $\delta\alpha\gamma$ folgen, welches aus ε-*ffi*(βL) ist. Das sind die Symbole aus ε-*ffi*(β), und zusätzlich die aus L, wenn $\beta \stackrel{*}{\Longrightarrow} \varepsilon$ gilt.

Beim Test „q' **not in** Q“ wird eine Gleichheit auf LR(1)-Items benutzt, unter der zwei LR(1)-Items nur dann gleich sind, wenn sie in Kern und Vorausschaumenge übereinstimmen.

Beispiel 8.4.14
In diesem Beispiel werden einige LR(1)-Zustände für die kontextfreie Grammatik G_0 aufgeführt. Die Numerierung der Zustände ist dieselbe wie in Abbildung 8.19. Man sieht, daß die bisher ungeeigneten Zustände S_1, S_2 und S_9 nach der Erweiterung um Vorausschaumengen keine Konflikte mehr enthalten. Im Zustand S_1' wird bei nächstem Eingabesymbol "+" gelesen, bei "#" reduziert. In S_2' wird bei "*" gelesen, bei "#" und "+" reduziert; ebenso in S_9'.

$$
\begin{array}{lcl}
S_0' & = & \mathit{Abschluß}(\text{Start}) \\
 & = \{ & [S \to .E, \{\#\}] \\
 & & [E \to .E + T, \{\#, +\}], \\
 & & [E \to .T, \{\#, +\}], \\
 & & [T \to .T * F, \{\#, +, *\}], \\
 & & [T \to .F, \{\#, +, *\}], \\
 & & [F \to .(E), \{\#, +, *\}], \\
 & & [F \to .\mathbf{id}, \{\#, +, *\}] \ \}
\end{array}
$$

$$
\begin{array}{lcl}
S_1' & = & \mathit{Abschluß}(\mathit{Nachf}(S_0', E)) \\
 & = \{ & [S \to E., \{\#\}], \\
 & & [E \to E. + T, \{\#, +\}] \ \}
\end{array}
$$

$$
\begin{array}{lcl}
S_2' & = & \mathit{Abschluß}(\mathit{Nachf}(S_1', T)) \\
 & = \{ & [E \to T., \{\#, +\}], \\
 & & [T \to T. * F, \{\#, +, *\}] \ \}
\end{array}
$$

$$
\begin{array}{lcl}
S_6' & = & \mathit{Abschluß}(\mathit{Nachf}(S_1', +)) \\
 & = \{ & [E \to E + .T, \{\#, +\}], \\
 & & [T \to .T * F, \{\#, +, *\}], \\
 & & [T \to .F, \{\#, +, *\}], \\
 & & [F \to .(E), \{\#, +, *\}], \\
 & & [F \to .\mathbf{id}, \{\#, +, *\}] \ \}
\end{array}
$$

$$
\begin{array}{lcl}
S_9' & = & \mathit{Abschluß}(\mathit{Nachf}(S_6', T)) \\
 & = \{ & [E \to E + T., \{\#, +\}], \\
 & & [T \to T. * F, \{\#, +, *\}] \ \}
\end{array}
$$

□

Die Tabelle 8.6 zeigt die zu den obigen Zuständen gehörenden Zeilen der action-Tabelle des kanonischen LR(1)-Parsers für unsere Grammatik G_0.

Tabelle 8.6: Einige Zeilen der action-Tabelle des kanonischen LR(1)-Parsers für G_0. s steht für *shift*, ri für reduziere mit Produktion i, acc für accept. Alle unbesetzten Einträge sind error-Einträge.

	id	(	)	*	+	#
S_0'	s	s				
S_1'					s	acc
S_2'				s	$r3$	$r3$
S_6'	s	s				
S_9'				s	$r2$	$r2$

Verwendete Numerierung der Produktionen
1: $S \to E$
2: $E \to E + T$
3: $E \to T$
4: $T \to T * F$
5: $T \to F$
6: $F \to (E)$
7: $F \to$ **id**

SLR(1)- und LALR(1)-Parsergenerierung

Zwei häufig in der Praxis verwendete LR-Analyseverfahren sind die SLR(1)- (simple LR-) und LALR(1)- (lookahead LR-) Verfahren. Es gibt nicht für alle kanonischen LR(1)-Grammatiken einen SLR(1)- bzw. LALR(1)-Parser. Umgekehrt ist jede Grammatik, die einen SLR(1)- bzw. LALR(1)-Parser besitzt, eine LR(1)-Grammatik. Ein Vorteil der SLR- und LALR-Parser gegenüber den kanonischen LR-Parsern ist, daß ihre Zustandsmenge für eine Grammatik G nur so groß ist wie die des LR(0)-Parsers für G.

Der Ausgangspunkt bei der (praktischen) Konstruktion von SLR(1)- und LALR(1)-Parsern ist ein schon konstruierter LR(0)-Parser. Um ungeeignete Zustände zu beseitigen, werden die vollständigen Items in LR(0)-Zuständen um Vorausschaumengen erweitert. Sei q ein LR(0)-Zustand, $[X \to \alpha.\beta]$ ein Item in q. Dann bezeichnen wir mit $LA(q, [X \to \alpha.\beta])$ die zum Item $[X \to \alpha.\beta]$ in q hinzuzufügende Vorausschaumenge. LA ist also eine Funktion $LA : Q_d \times It_G \to 2^{V_T \cup \{\#\}}$. Sie ist in den Fällen SLR(1) und LALR(1) in verschiedener Weise definiert.

SLR(1)

Jedem reduce-Item $[X \to \alpha.]$ wird (in allen Zuständen) die Menge $FOLLOW_1(X)$ als Vorausschau-Menge zugeordnet.

$$LA_S(q, [X \to \alpha.]) = \{a \in V_T \cup \{\#\} \mid S'\# \stackrel{*}{\Longrightarrow} \beta X a \gamma\} = FOLLOW_1(X)$$

für alle q mit $[X \to \alpha.] \in q$.

Definition 8.4.9 (SLR(1)-Grammatik)
Seien die vollständigen Items des LR-DEA(G) einer Grammatik G auf obige Weise um Vorausschaumengen erweitert. Ein Zustand heißt **SLR(1)-ungeeignet**, wenn es in ihm zwei Items $[X_1 \to \alpha_1., L_1]$ und $[X_2 \to \alpha_2., L_2]$ mit $L_1 \cap L_2 \neq \emptyset$ oder zwei Items $[X \to \alpha.a\beta]$ und $[Y \to \gamma., L]$ mit $a \in L$ gibt. Gibt es keine SLR(1)-ungeeigneten Zustände, so ist G eine **SLR(1)-Grammatik**. □

Beispiel 8.4.15
Wir betrachten wieder die Grammatik G_0 aus Beispiel 8.4.1. Ihr LR-DEA(G_0) hatte die ungeeigneten Zustände S_1, S_2 und S_9. Wir erweitern deren vollständige Items durch die entsprechenden $FOLLOW_1$-Mengen. Es ergeben sich

$S_1'' = \{ [S \to E., \{\#\}], [E \to E. + T] \}$ Konflikt beseitigt, " + " ist nicht in $\{\#\}$

$S_2'' = \{ [E \to T., \{\#, +,)\}], [T \to T. * F] \}$ Konflikt beseitigt, " * " ist nicht in $\{\#, +,)\}$

$S_9'' = \{ [E \to E + T., \{\#, +,)\}], [T \to T. * F] \}$ Konflikt beseitigt, " * " ist nicht in $\{\#, +,)\}$

Damit ist G_0 SLR(1)-Grammatik. □

Beispiel 8.4.16
Die folgende Grammatik aus [ASU86] beschreibt eine Vereinfachung der C-Wertzuweisung. Sie ist LALR(1)-Grammatik, wie später gezeigt wird, aber nicht SLR(1)-Grammatik.

$S' \to S$
$S \to L = R \mid R$
$L \to *R \mid$ **id**
$R \to L$

Die Itemmengen, die die Zustände des LR-DEA bilden, sind

$S_0 = \{ [S' \to .S], [S \to .L = R], [S \to .R], [L \to . * R], [L \to .\mathbf{id}], [R \to .L] \}$

$S_1 = \{ [S' \to S.] \}$

$S_2 = \{ [S \to L. = R], [R \to L.] \}$

$S_3 = \{ [S \to R.] \}$

$S_4 = \{ [L \to * .R], [R \to .L], [L \to . * R], [L \to .\mathbf{id}] \}$

$S_5 = \{ [L \to \mathbf{id}.] \}$

$S_6 = \{ [S \to L = .R], [R \to .L], [L \to . * R], [L \to .\mathbf{id}] \}$

$S_7 = \{ [L \to * R.] \}$

$S_8 = \{ [R \to L.] \}$

$S_9 = \{ [S \to L = R.] \}$

S_2 ist der einzige ungeeignete Zustand. $FOLLOW_1(R) = \{\#, =\}$. Bei der Erweiterung des vollständigen Items $[R \to L.]$ um die Vorausschaumenge $\{\#, =\}$ bleibt der shift-reduce-Konflikt erhalten, da das zu lesende Symbol "=" in der Vorausschaumenge des vollständigen Items enthalten ist. □

LALR(1)

$FOLLOW_1(X)$ faßt alle Symbole zusammen, die auf X in Satzformen der Grammatik folgen können. Bei der SLR(1)-Parser-Konstruktion wird $FOLLOW_1(X)$ als Vorausschaumenge zu $[X \to \alpha.]$ in allen Zuständen benutzt, in denen dieses vollständige Item vorkommt. Jetzt konstruieren wir zu einem Vorkommen von $[X \to \alpha.]$ in einem Zustand q eine von q abhängige, oft kleinere Vorausschaumenge.

$$LA_L(q, [X \to \alpha.]) = \{a \in V_T \cup \{\#\} \mid S'\# \underset{rm}{\overset{*}{\Longrightarrow}} \beta X a w \text{ und } \delta_d^*(q_d, \beta\alpha) = q\}$$

Dabei ist δ_d die Übergangsfunktion des LR-DEA(G). Jetzt sind also nur die Terminalsymbole in $LA_L(q, [X \to \alpha.])$, die auf X in einer Rechtssatzform $\beta X a w$ folgen können, sodaß $\beta\alpha$ den charakteristischen endlichen Automaten LR-DEA(G) in den Zustand q bringt.

Definition 8.4.10 (LALR(1)-Grammatik)
Seien die vollständigen Items des LR-DEA(G) einer Grammatik G um die LA_L-Vorausschaumengen erweitert. Ein Zustand heißt **LALR(1)-ungeeignet**, wenn er zwei Items $[X \to \alpha_1., L_1]$ und $[Y \to \alpha_2., L_2]$ mit $L_1 \cap L_2 \neq \emptyset$ oder zwei Items $[X \to \alpha.a\beta]$ und $[Y \to \gamma., L]$ mit $a \in L$ enthält. Eine Grammatik ohne LALR(1)-ungeeignete Zustände heißt **LALR(1)-Grammatik**. □

Diese Definition ist nicht konstruktiv, da in ihr i.a. unendliche Mengen von Rechtssatzformen auftreten.

Es ist folgende Berechnungsmethode für LALR(1)-Parser denkbar, aber unnötigerweise aufwendig: Man versuche, einen kanonischen LR(1)-Parser zu konstruieren. Enthalten seine Zustände keine Konflikte, so verschmelze man solche Zustände p und q zu einem neuen Zustand p', wenn die Menge der Kerne der Items in p gleich der Menge der Kerne der Items von q ist, d.h. wenn der Unterschied zwischen den beiden Mengen von Items nur im Unterschied von Vorausschaumengen besteht. Der neue Zustand p' entsteht, indem die Vorausschaumengen von Items mit gleichem Kern vereinigt werden. Gibt es nach dieser Verschmelzung keine Konflikte, so ist die Grammatik LALR(1).

Eine weitere Möglichkeit besteht in der Modifikation des Algorithmus **LR(1)-GEN**. Man ersetzt darin die bedingte Anweisung

if q' **not in** Q **then** $Q := Q \cup \{q'\}$ **fi**;

durch

if *exist.* q'' **in** Q *mit kerngleich*(q', q'') **then** *verschmelze*(Q, q', q'') **fi**;

wobei

function *kerngleich*(p, p' : **set of item**):**bool**;
  if *Menge der Kerne von* p = *Menge der Kerne von* p'
    then return (true)
    else return (false)
  fi;

proc *verschmelze*(Q : **set of set of item**, p, p' : **set of item**);
  $Q := Q \cup \{[X \to \alpha.\beta, L_1 \cup L_2] \mid [X \to \alpha.\beta, L_1] \in p \text{ und } [X \to \alpha.\beta, L_2] \in p'\}$.

Beispiel 8.4.17 Die Grammatik aus Beispiel 8.4.16 ist eine LALR(1)-Grammatik. Das Übergangsdiagramm ihres LALR(1)-Parsers ist in Abbildung 8.22 gegeben. □

Effiziente Berechnung der Vorausschaumengen für LALR(1)-Parser

Wir gehen von einem erzeugten LR(0)-Parser aus. Sein charakteristischer endlicher Automat ist der LR-DEA(G) = $(Q_d, V_N \cup V_T, \delta_d, q_d, \{q_f\})$. Die Aufgabe ist es, die vollständigen Items in den Zuständen von Q_d (oder zumindest die an Konflikten beteiligten) mit Mengen von Vorausschausymbolen zu versehen, d.h. für jeden Zustand $q \in Q_d$ und jedes vollständige Item $[X \to \alpha.]$ in q die Menge

$$LA_L(q, [X \to \alpha.]) = \{a \in V_T \cup \{\#\} \mid S'\# \underset{rm}{\overset{*}{\Longrightarrow}} \beta X a w \text{ mit } \delta_d^*(q_d, \beta\alpha) = q\}$$

zu berechnen.

Wir erweitern die Definition konsistent auf unvollständige Items $[X \to \alpha.\beta]$ in q:

$$LA_L(q, [X \to \alpha.\beta]) = \{a \in V_T \cup \{\#\} \mid S'\# \underset{rm}{\overset{*}{\Longrightarrow}} \gamma X a w \text{ und } \delta_d^*(q_d, \gamma\alpha) = q\}.$$

Dann zerlegen wir die Berechnung des LR-DEA(G) von q_d unter $\gamma\alpha$ nach q in zwei Stücke, den Weg unter γ und den unter α.

$$\begin{aligned} LA_L(q, [X \to \alpha.\beta]) &= \{a \in V_T \mid S' \underset{rm}{\overset{*}{\Longrightarrow}} \gamma X a w \text{ und} \\ &\qquad \exists p \in Q_d : \delta_d^*(q_d, \gamma) = p \text{ und } \delta_d^*(p, \alpha) = q\} \\ &= \bigcup_{p \,:\, \delta_d^*(p,\alpha) = q} \{a \mid S'\# \underset{rm}{\Longrightarrow} \gamma X a w \text{ und } \delta_d^*(q_d, \gamma) = p\} \\ &= \bigcup_{p \,:\, \delta_d^*(p,\alpha) = q} LA_L(p, [X \to .\alpha\beta]) \end{aligned}$$

(LA0)

Damit haben wir eine Reduktion „innerhalb von Items" gefunden. Wie kann man $LA_L(q, [X \to .\alpha])$ definieren? $[X \to .\alpha]$ ist entweder das Item $[S' \to .S]$, dann wird $LA_L(q_d, [S' \to .S]) = \{\#\}$ gesetzt. Oder $[X \to .\alpha]$ ist durch Anwendung der Abschlußfunktion auf ein Item $[Y \to \beta.X\gamma] \in q$ hineingekommen. Dann

ist klar, daß alle Terminalsymbole b aus $FIRST_1(\gamma)$ in $LA_L(q,[X \to .\alpha])$ aufgenommen werden müssen. Denn es gilt für jedes solche b:

$$b \in \{a \in FIRST_1(\gamma) | S \underset{rm}{\overset{*}{\Longrightarrow}} \varphi Y w \underset{rm}{\Longrightarrow} \varphi\beta X\gamma w \text{ und } \delta_d^*(q_d, \varphi\beta) = q\}.$$

Diese Elemente von $LA_L(q,[X \to .\alpha])$ heißen **spontan erzeugt**. Wenn γ ε erzeugt oder ε ist, so gehören natürlich auch alle Elemente aus $LA_L(q,[Y \to \beta.X\gamma])$ zu $LA_L(q,[X \to .\alpha])$, denn es gilt:

$$S' \underset{rm}{\overset{*}{\Longrightarrow}} \varphi Y w \underset{rm}{\Longrightarrow} \varphi\beta X\gamma w \underset{rm}{\overset{*}{\Longrightarrow}} \varphi\beta X b v \text{ mit } \delta_d^*(\varphi\beta) = q.$$

Solche Elemente b heißen aus $LA_L(q,[Y \to \beta.X\gamma])$ in $LA_L(q,[X \to .\alpha])$ **propagiert**. Damit können wir das folgende Gleichungssystem aufstellen:

(LA1)

$$\begin{array}{lllll}
(1) & LA_L(q_d,[S' \to .S]) & = & \{\#\} & \\
(2) & LA_L(q,[X \to \alpha Y.\beta]) & = & \bigcup_{\delta_d(p,Y)=q} LA_L(p,[X \to \alpha.Y\beta]) & \text{für } Y \in V_N \cup V_T \\
(3) & LA_L(q,[X \to .\alpha]) & = & \bigcup_{[Y \to \beta.X\gamma] \in q} FIRST_1(\gamma) \oplus_1 LA_L(q,[Y \to \beta.X\gamma]) &
\end{array}$$

Unter Benutzung von (LA0) können wir dieses Gleichungssystem verkleinern, indem wir nur vollständige Items, $[X \to \alpha.]$, und Anfangsitems $[X \to .\alpha]$ betrachten.

(LA2)

$$\begin{array}{lllll}
(1') & LA_L(q_d,[S' \to .S]) & = & \{\#\} & \\
(2') & LA_L(q,[X \to \alpha.]) & = & \bigcup_{\delta_d^*(p,\alpha)=q} LA_L(p,[X \to .\alpha]) & \text{für } \alpha \in (V_N \cup V_T)^* \\
(3') & LA_L(q,[X \to .\alpha]) & = & \bigcup_{[Y \to \beta.X\gamma] \in q} \bigcup_{\delta_d^*(p,\beta)=q} FIRST_1(\gamma) \oplus_1 LA_L(p,[Y \to .\beta X\gamma]) &
\end{array}$$

Die in (2′) berechneten Vorausschaumengen sind die gewünschten Ergebnisse. Sie können mithilfe der Lösungen von (1′) und (3′) berechnet werden, gehen aber selbst nicht in die Rechnung ein.

In (3′) hängt der Ausdruck $LA_L(q,[X \to .\alpha])$ nicht von α ab. Sein Wert ist für alle Alternativen für X in q der gleiche. Deshalb kann man statt LA_L eine Funktion $\widetilde{LA}_L$: $Q_d \times V_N \to 2^{V_T \cup \{\#\}}$ berechnen, die definiert wird durch $\widetilde{LA}(q,X) = LA_L(q,[X \to .\alpha])$ für eine der Alternativen $X \to \alpha$ von X. Daraus ergibt sich als neues Gleichungssystem

$$\begin{array}{lllll}
(1'') & \widetilde{LA}(q_d,S') & = & \{\#\} & \\
(3'') & \widetilde{LA}(q,X) & = & \bigcup_{[Y \to \beta.X\gamma] \in q} \bigcup_{\delta_d^*(p,\beta)=q} FIRST_1(\gamma) \oplus_1 \widetilde{LA}(p,Y) &
\end{array} \qquad \text{(LA3)}$$

Hat man seine Lösung, so ergeben sich die gesuchten Vorausschaumengen für die vollständigen Items zu

$$LA_L(q,[X \to \alpha.]) = \bigcup_{\delta_d^*(p,\alpha)=q} \widetilde{LA}(p,X)$$

Aus dem Gleichungssystem (LA3) läßt sich ein reines Vereinigungsproblem gewinnen. Der Ausdruck $FIRST_1(\gamma) \oplus_1 \widetilde{LA}(p,Y)$ muß umgeformt werden. Wie schon im Falle der $FIRST_1$- und $FOLLOW_1$-Berechnungen können wir erst die ε-Produktivität von Nichtterminalen und damit von Wörtern aus $(V_N \cup V_T)^*$ vorberechnen. Mit der ε-Produktivität von γ wissen wir, ob der Wert des Ausdrucks von $\widetilde{LA}(p,Y)$ abhängt oder nicht. Damit können wir die Relation R_{LA} definieren:
$(q,X)\ R_{LA}\ (p,Y)$, wenn für das Wort γ in (3″) in (LA3) gilt $\varepsilon\text{-}prod(\gamma) = true$.
Die Funktion g_{LA} bestimmt sich für jedes Paar (q,X) gemäß (3″) zu

$$g_{LA}(q,X) = \bigcup_{[Y \to \beta.X\gamma] \in q} \varepsilon\text{-}ffi(\gamma)$$

Ein Vorteil dieser Gleichungen ist es, daß man die Vorausschaumengen auch nach Bedarf berechnen kann. Dabei geht man von vollständigen Items in ungeeigneten Zuständen aus und betrachtet nur die Zustände und Items, die zur Vorausschaumenge beitragen.

Beispiel 8.4.18 (Fortführung von Beispiel 8.4.16)
Wir berechnen $LA_L(S_2,[R \to L.]) = \widetilde{LA}(S_0,R)$. Der Zustand S_0 ist der einzige Vorgängerzustand von S_2. $\widetilde{LA}(S_0,R)$ ergibt sich gemäß (LA3) zu: $\widetilde{LA}(S_0,R) = \widetilde{LA}(S_0,S) = \widetilde{LA}(S_0,S') = \{\#\}$. Also ist $LA_L(S_2,[R \to L.]) = \{\#\}$. Damit ist der shift-reduce-Konflikt in Zustand S_2 beseitigt; es gibt keinen LALR(1)-ungeeigneten Zustand mehr; die Grammatik ist LALR(1). □

Damit ergibt sich das Übergangsdiagramm aus Abbildung 8.22

8.4.6 Fehlerbehandlung in LR-Parsern

LR-Parser besitzen ebenso wie LL-Parser die Eigenschaft des fortsetzungsfähigen Präfixes; d.h. jedes durch einen LR-Parser fehlerfrei analysierte Präfix der Eingabe kann zu einem korrekten Eingabewort, einem Satz der Sprache, fortgesetzt werden. Wir schreiben im folgenden Konfigurationen eines LR-Parsers als $(\varphi q, a_i \ldots a_n)$; d.h. der Kellerinhalt ist φq, der aktuelle Zustand q, und die restliche Eingabe ist $a_i \ldots a_n$. Ist ein LR-Parser in der Konfiguration $(\varphi q, a_i \ldots a_n)$ mit $action(q,a_i) = error$, so ist dies die frühestmögliche Situation, in der ein Fehler entdeckt werden kann.

Definition 8.4.11 (Fehlerkonfiguration)
Eine Konfiguration $(\varphi q, a_i \ldots a_n)$ eines LR(1)-Parsers mit $action(q,a_i) = error$ nennen wir eine **Fehlerkonfiguration**. q heißt der **Fehlerzustand** dieser Konfiguration. □

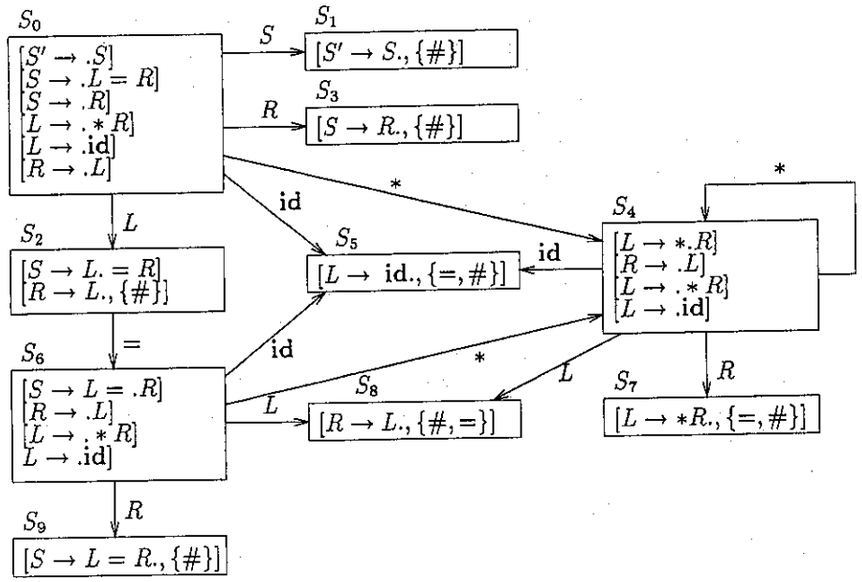

Abb. 8.22: Übergangsdiagramm des LALR(1)-Parsers für die Grammatik aus Beispiel 8.4.16.

Es gibt auch für LR-Parser ein ganzes Spektrum von Fehlerbehandlungsverfahren:

- Vorwärtsfehlerbehandlung; d.h. Modifikationen in der restlichen Eingabe ohne Manipulation auf dem Parserkeller, oder
- Rückwärtsfehlerbehandlung, also mit Veränderungen des Parserkellers;
- benutzerspezifizierte Fehlerbehandlung oder
- aus der Grammatik bzw. dem LR-Parser automatisch erzeugte Fehlerbehandlung.

Das im folgenden vorgestellte Verfahren ist im wesentlichen ein aus den Parsertabellen automatisch erzeugtes Vorwärtsfehlerbehandlungsverfahren. Es erlaubt lediglich, die letzte vollzogene Reduktion rückgängig zu machen.

Die Aufgabe des zu findenden Verfahrens ist es, zu der Fehlerkonfiguration $(\varphi q, a_i \dots a_n)$ eine „passende" Konfiguration zu finden, in der eine Fortsetzung der Analyse und zwar durch Lesen mindestens eines weiteren Eingabesymbols möglich ist. Eine Konfiguration paßt zu der Fehlerkonfiguration, wenn sie durch möglichst wenig Veränderungen aus der Fehlerkonfiguration hervorgeht.

Wir wollen die zugelassenen Veränderungen sogar durch die Annahme der 1-Fehlerhypothese drastisch einschränken. Sie besagt, daß der Fehler durch **ein** fehlendes, **ein** überflüssiges oder **ein** falsches Symbol an der Fehlerstelle verursacht wurde. Damit muß der Fehlerbehandlungsalgorithmus über drei Operationen verfügen, das Einsetzen, das Löschen und das Ersetzen **eines** Symbols.

Sei $(\varphi q, a_i \dots a_n)$ die Fehlerkonfiguration. Das Ziel bei einer Fehlerkorrektur mittels einer der drei Operationen ist das folgende:

Löschen: Finde Kellerinhalte $\varphi' p$ mit $(\varphi q, a_{i+1} \dots a_n) \vdash^* (\varphi' p, a_{i+1} \dots a_n)$ und mit $action[p, a_{i+1}] = shift$.

Ersetzen: Finde ein Symbol a und Kellerinhalte $\varphi' p$ mit $(\varphi q, a a_{i+1} \dots a_n) \vdash^* (\varphi' p, a_{i+1} \dots a_n)$ und $action[p, a_{i+1}] = shift$.

Einfügen: Finde ein Symbol a und Kellerinhalte $\varphi' p$ mit $(\varphi q, a a_i \dots a_n) \vdash^* (\varphi' p, a_i \dots a_n)$ und $action[p, a_i] = shift$.

Die gesuchten Kellerinhalte $\varphi' p$ können sich dadurch ergeben, daß unter dem jeweils neuen nächsten Eingabesymbol Reduktionen möglich sind, die in der Fehlerkonfiguration nicht möglich waren.

Man sieht schon eine wichtige Eigenschaft der drei Operationen; sie garantieren die Terminierung des Fehlerbehandlungsverfahren. Denn jeder der drei Schritte stellt im Erfolgfall den Lesezeiger um mindestens ein Symbol weiter.

Fehlerbehandlungsmethoden mit Zurücksetzen erlauben zusätzlich, eine zuletzt angewandte Produktion der Form $X \to \alpha Y$ rückgängig zu machen und $Y a_i \dots a_n$ als Eingabe zu betrachten, wenn die anderen Korrekturversuche gescheitert sind.

Die 1-Symbol-Korrekturoperationen sind zu teuer, wenn sie, wie oben beschrieben, ausgeführt werden. Die Suche nach (einem Symbol und) einer Konfiguration, in der man wieder aufsetzen kann, würde eventuell Reduktionen erfordern, dann einen Test, ob man ein Symbol lesen kann, bei Mißerfolg Wiederherstellen der Fehlerkonfiguration usw. Dies wäre ein Verfahren, welches dynamisch, also zur Übersetzungszeit verschiedene Möglichkeiten durchprobiert. Wir werden jetzt sehen, wie man zur Parsergenerierungszeit Vorberechnungen auf dem Parser machen kann, um Fehler effizienter zu behandeln.

Sei $(\varphi q, a_i \dots a_n)$ wieder die Fehlerkonfiguration. Betrachten wir das **Einfügen** eines Symbols $a \in V_T$. Die Fehlerbehandlung kann aus der folgenden Sequenz von Schritten bestehen (siehe Abbildung 8.23 (a)):

(1) eine Folge von Reduktionen unter Vorausschausymbol a, gefolgt von

(2) einer Leseaktion bezüglich a, gefolgt von

(3) einer Folge von Reduktionen unter Vorausschausymbol a_i.

Die Teilfolgen (1) - (3) lassen sich durch Vorberechnung effizient machen.

zu **(1)**: Berechne für alle $q \in Q_d$ und alle $a \in V_T$ die Menge $Succ(q)_a$ der **Reduktionsnachfolger** von q unter a. Das sind alle Zustände, in die der Parser aus q nur durch Reduktionen unter Vorausschausymbol a kommen kann (siehe Abbildung 8.24), und zusätzlich q selbst.

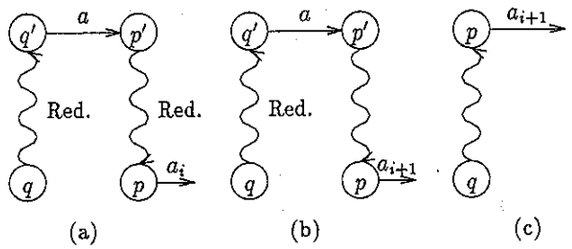

Abb. 8.23: Schließen der Brücke bei der Fehlerkorrektur, (a) beim Einfügen, (b) beim Ersetzen, (c) beim Löschen eines Symbols.

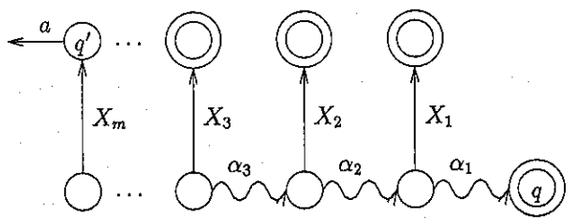

Abb. 8.24: $Succ(q)_a$. Die möglichen Reduktionen aus q unter a in q' erfolgen mittels der Produktionen $X_1 \to \alpha_1, X_2 \to \alpha_2 X_1, \ldots, X_m \to \alpha_m X_{m-1}$

zu **(2)**: Berechne für alle $a \in V_T$ die Menge $Sh(a)$ der Zustände, in denen a gelesen werden kann. Dies sind die Zustände q, für die $action[q, a] = shift$ gilt.

zu **(3)**: Berechne für alle $p \in Q_d$ und alle $a \in V_T$ die Menge $Pred(p)_a$ der **Reduktionsvorgänger** von p unter a. Das sind alle Zustände, aus denen der Parser nur durch Reduktion unter Vorausschausymbol a in den Zustand p kommen kann (siehe Abbildung 8.25), und zusätzlich p selbst.

Satz 8.4.6 *Es gilt $q \in Pred(p)_a$ genau dann, wenn $p \in Succ(q)_a$*

Die Korrektur mittels Einfügen eines Symbols a ist dann vielversprechend, wenn es einen Zustand q' aus $Succ(q)_a$, einen Zustand p aus $Sh(a_i)$ und einen

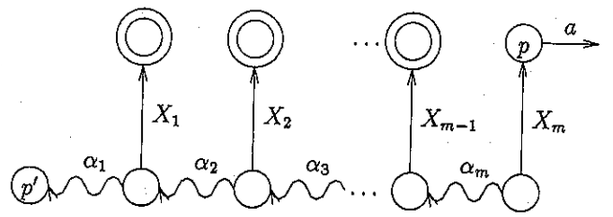

Abb. 8.25: $Pred(p)_a$. Die Reduktionen aus p' unter a in p benutzen die Produktionen $X_1 \to \alpha_1, X_2 \to \alpha_2 X_1, \ldots, X_m \to \alpha_m X_{m-1}$

Zustand p' aus $Pred(p)_{a_i}$ gibt, so daß $\delta_d(q', a) = p'$ ist. Das kann man so darstellen:

$$\underset{\in Succ(q)_a}{q'} \xrightarrow{a} \underset{\in Pred(p)_{a_i}}{p'}$$

a schließt die Brücke in der Fehlerkonfiguration.

Will man die Vorberechnung weitertreiben, so kann man für q und a_i die Menge

$$Bridge(q)_{a_i} = \{a \in V_T \mid \exists q' \in Succ(q)_a \text{ und } \delta_d(q', a) = p' \text{ und } Sh(a_i) \cap Succ(p')_{a_i} \neq \emptyset\}$$

berechnen.

Beispiel 8.4.19
Wir betrachten die Grammatik aus Beispiel 8.4.16 mit dem für sie berechneten LALR(1)-Parser aus Abbildung 8.22. Als erstes werden die Mengen, $Sh(a)$, der Zustände berechnet, in denen man a lesen kann.
$Sh(a)$:

$Sh(=)$	=	$\{S_2\}$
$Sh(*)$	=	$\{S_0, S_6, S_4\}$
$Sh(\mathbf{id})$	=	$\{S_0, S_6, S_4\}$

Dann werden die Reduktionsnachfolger $Succ(q)_a$ von q unter a und die Mengen $Bridge(q)_a$ berechnet.

$\underline{Succ(q)_a}$:

	=	*	id
S_0	S_0	S_0	S_0
S_1	S_1	S_1	S_1
S_2	S_2	S_2	S_2
S_3	S_3	S_3	S_3
S_4	S_4	S_4	S_4
S_5	S_2, S_5	S_5	S_5
S_6	S_6	S_6	S_6
S_7	S_2, S_7	S_7	S_7
S_8	S_2, S_8	S_8	S_8
S_9	S_9	S_9	S_9

$\underline{Bridge(q)_a}$:

	=	*	id
S_0	{id}	{*}	{*}
S_1	$\emptyset$	$\emptyset$	$\emptyset$
S_2	$\emptyset$	{=}	{=}
S_3	$\emptyset$	$\emptyset$	$\emptyset$
S_4	{id}	{*}	{*}
S_5	$\emptyset$	{=}	{=}
S_6	{id}	{*}	{*}
S_7	$\emptyset$	{=}	{=}
S_8	$\emptyset$	$\emptyset$	$\emptyset$
S_9	$\emptyset$	$\emptyset$	$\emptyset$

Jetzt können wir betrachten, welche Fehlerkorrekturen der erzeugte Fehlerbehandler machen würde.

Eingabe	Fehlerkonfiguration	Brücke	Korrektur
* = id #	(S_0S_4, = id #)	$Bridge(S_4)_= = \{id\}$	Einfügen von id
id == id #	($S_0S_2S_6$, = id #)	$Bridge(S_6)_{id} = \{*\}$	Ersetzen von = durch *

Ein Beispiel für eine Löschkorrektur:

Eingabe	Fehlerkonfiguration	Brücke	Korrektur
id id = id #	(S_0S_5, id = id #)	$Sh(=) \cap Succ(S_5)_= = \{S_2\}$	Löschen von id, Ersetzen von S_5 durch S_2

□

Auf der Basis der so vorberechneten Mengen kann auch effizient eine 1-Symbol-**Ersetzungskorrektur** versucht werden. Der Unterschied ist der, daß Symbole aus $Bridge(q)_{a_{i+1}}$ betrachtet werden müssen, siehe Abbildung 8.23 (b).

Der Test, ob eine 1-Symbol-**Löschkorrektur** vernünftig ist, wird jetzt ebenfalls effizient durchführbar. Die Löschung eines Symbols a ist offensichtlich dann eine Möglichkeit zur Fehlerbehandlung, wenn es einen Zustand p in $Sh(a_{i+1}) \cap Succ(q)_{a_{i+1}}$ gibt, siehe Abbildung 8.23 (c). Für jede Kombination aus einem Zustand q und einem Symbol a läßt sich vorberechnen, ob es einen solchen Zustand p gibt. Tabelliert man dieses Prädikat, so bleibt ein einfaches Nachschlagen übrig.

Einen Sonderfall haben wir bisher ignoriert, nämlich Korrekturen bei erschöpfter Eingabe. Da das Endesymbol "#" nicht gelesen wird, ist $Sh("\#") = \emptyset$. Löschaktionen und Ersetzungsaktionen sind nicht möglich, da nichts zu löschen bzw. zu ersetzen ist. Nur Einsetzungsaktionen bleiben übrig.

Die Einsetzung eines Symbols a ist sinnvoll, wenn nach eventuellen Reduktionen aus q unter a ein Zustand p erreicht wird, aus dem nach Lesen von a ein Zustand p' erreicht wird, aus dem wiederum unter "#" Reduktionen in *accept*-Konfigurationen möglich sind. Dazu kann man für jeden Zustand q die Menge $Acc(q)$ vorberechnen, die alle Terminalsymbole enthält, die dies garantieren.

Vorwärtsbewegung

Ein paar Fragen zu dem vorgestellten Verfahren sind noch offen und ein paar Verbesserungen noch möglich. Einmal ist nicht klar, welche Korrektur vorgenommen werden soll, wenn mehrere möglich sind. Damit zusammen hängt die Frage, wie man die Qualität einer Korrektur beurteilt. Um die Qualität von Korrekturen effizient zu prüfen, wird die Fehlerbehandlung eine Vorwärtsbewegung über die restliche Eingabe machen und dabei ein Präfix der restlichen Eingabe auf alle möglichen Arten reduzieren. Dieser „kondensierte" Rechtskontext wird dann benutzt, um zu testen, ob die Korrektur eine Fortsetzung über mehr als ein Symbol erlaubt. Der Parser startet eine Vorwärtsbewegung in der Fehlerkonfiguration $(\varphi q, a_i \dots a_n)$. Dabei versucht er, ein möglichst langes Präfix von $a_i \dots a_n$ zu reduzieren. Da der Parser diese Analyse nicht in einem eindeutig bestimmten Zustand beginnen kann – q paßt ja gerade nicht zu $a_i \dots a_n$ – beginnt er sie in der Menge von Zuständen $Sh(a_{i+1})$; das sind die Zustände, in denen er a_{i+1} lesen kann. Seine Konfigurationen bestehen auch weiterhin aus Folgen von **Mengen von Zuständen** Q in einem Fehlerkeller FK und der restlichen Eingabe. Ist der Parser in der Menge von Zuständen Q bei nächstem Eingabesymbol a, so macht er für alle $q \in Q$ alle Nichtfehler-Übergänge gemäß $action[q, a]$,

- wenn sie alle übereinstimmen, d.h. entweder alle *shift* oder alle $reduce(X \to \alpha)$ mit der gleichen Produktion $X \to \alpha$ sind, und
- im Falle $reduce(X \to \alpha)$ der Fehlerkeller nicht kürzer als $|\alpha|$ ist.

Die Vorwärtsbewegung stoppt,

- wenn für alle $q \in Q$ $action[q, a] = error$ gilt; das nennt man „2. Fehler gefunden",
- wenn die *action*-Tabelle für Q und a mehr als eine Aktion angibt,
- wenn sie die einzige Aktion *accept* angibt, und
- wenn sie eine Reduktion verlangt, wobei die Länge der rechten Seite größer als die Tiefe des Fehlerkellers ist; das wird „Reduktion über die Fehlerstelle" genannt.

Als Ergebnis gibt die Vorwärtsbewegung das Wort α zurück, zu dem sie das bis dahin gelesene Präfix der restlichen Eingabe reduziert hat, und die restliche Eingabe. Dies sei bezeichnet mit $VB(a_i \dots a_n) = (\alpha, a_{i+k} \dots a_n)$.

Eine Fehlerbehandlung mit Vorwärtsbewegung kann damit folgendermaßen beschrieben werden.

Sei die Fehlerkonfiguration $(\varphi q, a_i \dots a_n)$; $VB(a_i \dots a_n) = (\alpha, a_{i+k} \dots a_n)$.
Versuche zu löschen:
gibt es ein $p \in Sh(a_{i+1}) \cap Succ(q)_{a_{i+1}}$, so *teste* $(\varphi q, \alpha a_{i+k} \dots a_n)$;
Versuche zu ersetzen:
gibt es ein $a \in Bridge(q)_{a_{i+1}}$ mit $\underset{\in Succ(q)_a}{q'} \xrightarrow{a} \underset{\in Pred(p)_{a_{i+1}}}{p'}$, so *teste* $(\varphi q, a\alpha a_{i+k} \dots a_n)$;

Versuche einzusetzen:
gibt es ein $a \in Bridge(q)_{a_i}$ mit $\underset{\in Succ(q)_a}{q'} \xrightarrow{a} \underset{\in Pred(p)_{a_i}}{p'}$, so *teste* $(\varphi q, aa_i\alpha a_{i+k} \ldots a_n)$;

In der Prozedur *teste* werden Korrekturversuche bewertet. Dabei wird jeweils genau eine der bei der Vorwärtsbewegung parallel durchlaufenen Konfigurationen eingenommen. Eine Bestätigung der getesteten Fehlerkorrektur liegt dann vor, wenn man auf eine *accept*-Konfiguration stößt, oder wenn bei einer Reduktion über die Fehlerstelle jetzt bei Testen der fehlende Anfang der Produktion im Keller steht.

Im Gegensatz zur Vorwärtsbewegung, die ohne Linkskontext, d.h. Parserkellerinhalt, arbeitete, steht jetzt beim Testen des Korrekturvorschlags der Inhalt des Parserkellers zur Verfügung.

Falsche Reduktionen in SLR(1)- und LALR(1)-Parsern

Kanonische LR-Parser entdecken Fehler zum frühestmöglichen Zeitpunkt; sie lesen weder ein Symbol über die Fehlerstelle hinaus, noch reduzieren sie unter einem falschen Vorausschausymbol. SLR(1)- und LALR(1)-Parser lesen zwar auch nie ein Symbol über die Fehlerstelle hinaus, machen wegen der weniger differenzierten Vorausschaumengen jedoch eventuell noch Reduktionen, bevor sie bei einem *shift*-Zustand den Fehler entdecken. Dazu legt man einen zusätzlichen Keller an, auf dem man alle seit dem jeweils letzten Lesen durchgeführten Reduktionen speichert. Dieser Keller wird bei einer Leseaktion wieder geleert. Im Fehlerfall werden die gekellerten Reduktionen in umgekehrter Reihenfolge wieder rückgängig gemacht.

8.4.7 Scannergenerierung mit LR-Techniken

Die vorgestellten Techniken zur LR-Parser-Generierung erlauben die direkte Erzeugung von deterministischen endlichen Automaten für die lexikalische Analyse, also ohne den Umweg über nichtdeterministische endliche Automaten. Beim Einsatz eines LR-Parser-Generator würde man als Ergebnis erst einmal einen deterministischen Kellerautomaten erwarten. Wir gehen aber nach wie vor von einer Spezifikation der lexikalischen Analyse durch eine Folge von **regulären** Definitionen aus. Uns ist auch bekannt, daß die von einer solchen Folge beschriebene Sprache durch einen endlichen Automaten erkannt werden kann. Also muß der erzeugte LR-Automat auch ohne Keller auskommen können. Wozu wird im LR-Parser der Keller gebraucht? Auf ihm werden konsumierte Terminale und Nichtterminale, zu denen reduziert wurde, bzw. die dazu korrespondierenden Zustände abgespeichert. Wenn der oberste Zustand evtl. in Kombination mit Vorausschausymbolen eine Reduktion mittels einer Produktion $X \to \alpha$ anordnet, so werden $|\alpha|$ Einträge vom Keller entfernt und das Ergebnis der Reduktion auf dem Keller abgespeichert.

Bei der Spezifikation eines Scanners handelt es sich um eine Folge von regulären Definitionen

$$\begin{array}{lcl} X_1 & \to & r_1 \\ & \vdots & \\ X_n & \to & r_n \end{array}$$

Wir nehmen an, daß die Namen X_i schon durch Substitution aus den $r_1, \ldots, r_{i-1}$ beseitigt worden sind. Dann können wir diese Folge als eine Menge von erweiterten kontextfreien Produktionen betrachten. In ihren rechten Seiten treten keine Nichtterminale mehr auf. Reduktionen und anschließende Übergänge unter Nichtterminalen entfallen also. Ist ein Exemplar von r_i gefunden – der erzeugte Automat signalisiert eine Reduktion – und ist kein Leseübergang möglich, so wird dies als „ein X_i gefunden" gemeldet. Anschließend startet der Automat mit dem nächsten Eingabesymbol wieder im Anfangszustand. Gibt es einen Leseübergang, so muß er sich den erreichten Endzustand und den aktuellen Stand des Lesezeigers in zwei globalen Variablen merken. (Er soll ja das längste Präfix der restlichen Eingabe als das nächste Symbol abliefern, welches in einen Endzustand führt.) Insgesamt gibt es also keine Notwendigkeit einen Keller zu benutzen.

Der nach dem LR-Verfahren konstruierte Scanner hat Zustände, die aus Mengen von erweiterten kontextfreien Items bestehen. Jedes der Items in einem Zustand beschreibt eine mögliche Interpretation der Analysesituation. Bei der Konstruktion des LR-Automaten muß noch die Abschlußbildung auf reguläre rechte Seiten erweitert werden. Dies geschieht durch die folgende Funktion *ersetze*:

ersetze(I : **set of item**) : **set of item**;

Die Funktion *ersetze* produziert die Menge von Items, die sich durch die wiederholte Anwendung der folgenden Ersetzungsregeln auf I ergibt; (Wiederholung bis keine Ersetzung mehr möglich ist.):

ersetze	$\alpha.(r)^*\beta$	durch	$\alpha(.r)^*\beta, \alpha(r)^*.\beta$
ersetze	$\alpha(r.)^*\beta$	durch	$\alpha(.r)^*\beta, \alpha(r)^*.\beta$
ersetze	$\alpha.(r_1\|\cdots\|r_n)\beta$	durch	$\alpha(.r_1\|\cdots\|r_n)\beta, \ldots, \alpha(r_1\|\cdots\|.r_n)\beta$
ersetze	$\alpha(r_1\|\cdots\|r_i.\|\cdots\|r_n)\beta$	durch	$\alpha(r_1\|\cdots\|r_i\|\cdots\|r_n).\beta$ für $1 \le i \le n$

Damit ergibt sich für die direkte Konstruktion eines Scanners aus einer Folge von regulären Definitionen der Algorithmus LR-SCANGEN.

Algorithmus LR-SCANGEN:
Eingabe: Folge von regulären Definitionen
(nach Einsubstitution der r_i für X_i): $\begin{array}{lcl} X_1 & \to & r_1 \\ & \vdots & \\ X_n & \to & r_n \end{array}$

Ausgabe: Deterministischer endlicher Automat $(Q, \Sigma, \delta, q_0, Q_f)$ für die Vereinigung der Sprachen der $r_1, \ldots, r_n$; jeder Endzustand ist einer oder mehreren Definitionen zugeordnet.
Methode:

```
type state = set of item;
var q, q': state;
var S : set of state; (* Menge der noch zu bearbeitenden Zustände*)
var Q : set of state; (* Menge der schon erzeugten Zustände*)
func nachf(I : set of item, a : Σ) set of item;
var I': set of item;
begin
      I' := {[X → αa.β] | [X → α.aβ] ∈ I};
      return(ersetze(I'))
end;
begin
   q0 := ersetze({[X1 → .r1], ..., [Xn → .rn]});
   S := {q0}; Q := {q0};
   repeat
      wähle ein q ∈ S;
      foreach a ∈ Σ mit: exist [X → α.aβ] in q do
         q' := nachf(q, a);
         if q' not in Q then Q := Q ∪ {q'}; S := S ∪ {q'} fi;
         δ := δ ∪ {q, a, q'}   (* neuer Übergang q -a-> q' *)
      od;
      S := S - {q};    (* q erledigt *)
   until S = ∅
end.
```

Q_f ist die Menge aller Zustände mit mindestens einem vollständigen Item.

Beispiel 8.4.20
Iconst $\rightarrow$ $Zi(Zi)^*$
Rconst $\rightarrow$ $Zi(Zi)^*.Zi(Zi)^*(e(+|-)\ Zi\ Zi \mid \varepsilon)$
Der zugehörige LR-Scanner ist in Abbildung 8.26 dargestellt. □

Satz 8.4.7 *Sei $X_1 \rightarrow r_1, \ldots, X_n \rightarrow r_n$ eine Folge von regulären Definitionen (nach Substitution). Sei M der gemäß Algorithmus LR-SCANGEN erzeugte DEA. Dann wird jede der durch die r_i beschriebenen regulären Mengen von M erkannt und zwar durch Halten in einem Endzustand, welcher das Item $[X_i \rightarrow r_i.]$ enthält.*

Dieser Satz besagt, daß M tatsächlich ein Akzeptor für die beschriebenen regulären Sprachen ist, und daß Endzustände von M den regulären Sprachen zugeordnet werden können.

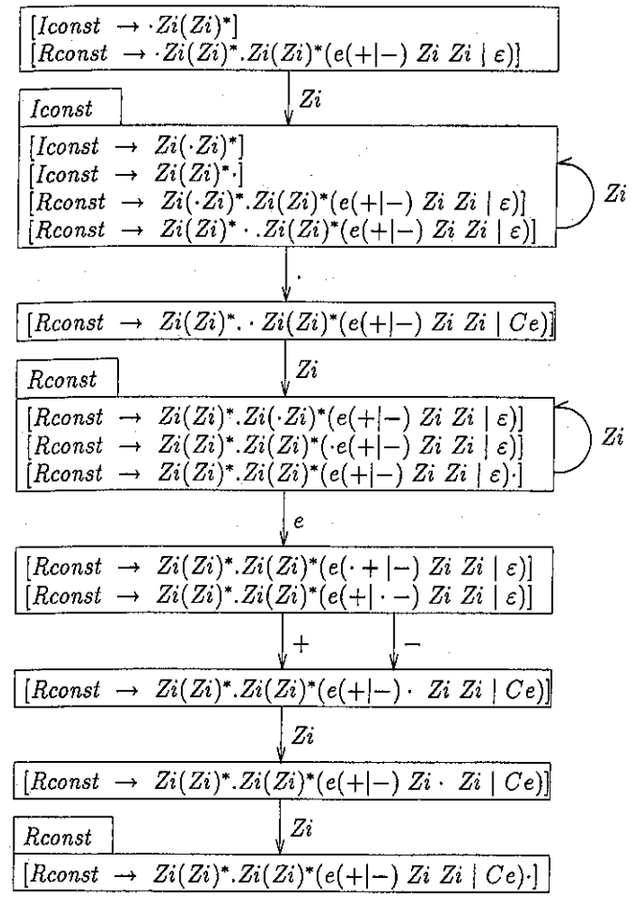

Abb. 8.26: Mit LR-Technik erzeugter deterministischer endlicher Automat. Auf den Kästchen für Endzustände steht der Name der zugehörigen Definition.

Korollar 8.4.7.1
Die von r_i und r_j beschriebenen regulären Mengen sind genau dann nicht disjunkt, wenn es einen Endzustand von M gibt, welcher die beiden vollständigen Items $[X_i \rightarrow r_i.]$ und $[X_j \rightarrow r_j.]$ enthält.

Sind die regulären Mengen paarweise disjunkt, so kann M durch Interpretation des Endzustandes jedes akzeptierte Wort eindeutig einer regulären Menge zuordnen.