3

Syntaktische Analyse

3.1 Die Aufgabe der syntaktischen Analyse

Der Parser realisiert die *syntaktische Analyse* von Programmen. Als Eingabe erhält er eine Folge von Symbolen. Seine Aufgabe ist es, in dieser Folge von Symbolen die syntaktische Struktur des Programms zu identifizieren. Die syntaktische Struktur gibt an, wie die verschiedenen syntaktischen Einheiten des Programms ineinander geschachtelt sind. Syntaktische Einheiten in imperativen Sprachen sind z.B. Variablen, Ausdrücke, Anweisungen, Anweisungsfolgen und Deklarationen. In funktionalen Sprachen gibt es Variablen, Ausdrücke, Muster, Definitionen und Deklarationen und in logischen Sprachen wie PROLOG Variablen, Terme, Listen von Termen, Ziele und Klauseln.

Der Parser repräsentiert die syntaktische Struktur des Eingabeprogramms in einer Datenstruktur, mit deren Hilfe weitere Übersetzerkomponenten auf die einzelnen Programmbestandteile zugreifen können. Eine mögliche Darstellung ist der *Syntaxbaum* (syntax tree, parse tree) des Programms. Der Syntaxbaum kann später mit weiteren Informationen angereichert werden. Mit seiner Hilfe können Transformationen des Programms durchgeführt oder direkt Code für eine Zielmaschine generiert werden.

Manchmal ist die Übersetzungsaufgabe für eine Programmiersprache so leicht, dass die Programme sogar in einem Durchgang durch den Programmtext übersetzt werden können. Dann kann der Parser gegebenenfalls auf eine explizite Darstellung der syntaktischen Struktur verzichten und stattdessen an geeigneten Stellen der Syntaxanalyse Hilfsfunktionen zur semantischen Analyse und zur Codeerzeugung aufrufen.

Viele Programme, die einem Übersetzer präsentiert werden, enthalten jedoch Fehler. Ein beträchtlicher Teil davon besteht aus Verstößen gegen die Syntaxregeln der Programmiersprache. Solche *Syntaxfehler* entstehen meist durch Flüchtigkeitsfehler, wie Buchstabendreher oder nicht korrekt ausgezählte Klammern oder fehlende Semikolons. Von jedem Übersetzer wird erwartet, dass er syntaktische Fehler möglichst genau lokalisiert. Oft kann jedoch nicht die Fehlerstelle selbst festgestellt werden, sondern nur die früheste Stelle, an welcher der Fehler zu einer Situation geführt hat, in der keine Fortsetzung der bisher analysierten Eingabe zu einem korrekten Programm möglich ist. Man erwartet von einem Übersetzer auch, dass er nach nach dem ersten entdeckten Fehler nicht abbricht, sondern möglichst bald wieder Tritt fasst, d.h. in einen Zustand kommt, in dem das restliche Programm analysiert bzw. weitere Fehler entdeckt werden können.

Die syntaktische Struktur der Programme einer Programmiersprache lässt sich durch eine kontextfreie Grammatik beschreiben. Unser Ziel ist, aus einer solchen Grammatik automatisch einen Syntaxanalysator zu generieren. Aus Effizienz- und Eindeutigkeitsgründen beschränkt man sich im Übersetzerbau meist auf deterministisch analysierbare kontextfreie Grammatiken, für welche automatisch Parser generiert werden können. Die in der Praxis eingesetzten Syntaxanalyseverfahren fallen in zwei Klaseen, nämlich *top-down*- und *bottom-up*-Verfahren. Beide lesen die Eingabe von links and rechts. Die Unterschiede in der Arbeitsweise macht man sich am besten daran klar, wie sie jeweils Syntaxbäume aufbauen.

Top-down-Analysierer beginnen die Analyse und die Konstruktion des Syntaxbaums mit dem Startsymbol der Grammatik, der Markierung der Wurzel des Syntaxbaums. Man nennt *top-down-*

Analysierer auch *Voraussageparser*. Sie treffen Voraussagen darüber, wie das Programm bzw. Teile des Programms aussehen sollten, und versuchen anschließend, diese Voraussagen zu bestätigen. Die erste Prognose besteht aus dem Startsymbol der Grammatik. Sie besagt, dass die Eingabe ein Wort für das Startsymbol ist. Nehmen wir jetzt an, ein Teil der Eingabe sei bereits bestätigt. Dann gibt es zwei Fälle:

- Beginnt der unbestätigte Teil der Prognose mit einem Nichtterminal, verfeinert der *top-down*-Parser seine aktuelle Prognose, indem er eine der Produktionen für dieses Nichtterminal auswählt.
- Beginnt die aktuelle Prognose mit einem Terminalsymbol, vergleicht er dieses mit dem nächsten Eingabesymbol. Sind diese gleich, so ist ein weiteres Symbol der Prognose bestätigt. Andernfalls liegt ein Fehler vor.

Der *top-down*-Parser ist fertig, wenn die gesamte Eingabe als Prognose vorausgesagt und bestätigt wurde.

bottom-up-Parser beginnen dagegen die Analyse und die Konstruktion des Syntaxbaums mit der Eingabe, d.h. dem zu analysierenden Programm und versuchen, für immer längere Anfangsstücke der Eingabe die syntaktische Struktur zu identifizieren. Dazu versuchen sie, sukzessive Vorkommen rechter Seiten von Produktionen der Grammatik durch die linke Seite zu ersetzen. Eine solche Ersetzung wird *reduce*-Schritt genannt. Anstatt zu reduzieren, kann der Parser auch beschließen, ein weiteres Symbol der Eingabe zu konsumieren. Ein solcher Schritt heißt *shift*. Wegen dieser beiden Arten von Schritten heißt ein *bottom-up*-Parster auch *shift-reduce*-Parser. Die Analyse des Parsers ist erfolgreich, wenn sich aus der Eingabe durch eine Folge von *shift*- und *reduce*-Schritten das Startsymbol der Grammatik ergibt.

Die Behandlung von Syntaxfehlern

Die meisten Programme, mit denen ein Übersetzer konfrontiert wird, sind fehlerhaft. Ein Grund ist, dass fehlerhafte Programme i.A. mehrfach übersetzt werden, fehlerfreie Programme dagegen nur nach Modifikationen oder Portierungen auf andere Rechner. Deshalb sollte ein Übersetzer mit dem *Normalfall*, dem inkorrekten Quellprogramm, möglichst gut umgehen können. Lexikalische Fehler und auch Fehler in der statischen Semantik, also etwa Typfehler in imperativen Sprachen, lassen sich einfacher lokal diagnostizieren und behandeln. Syntaxfehler wie etwa Fehler in der Klammerstruktur des Programms sind schwieriger zu diagnostizieren. In diesem Abschnitt beschreiben wir die erwünschten und die möglichen Reaktionen eines Parsers auf Syntaxfehler.

Der Parser kann auf syntaktisch inkorrekte Programme auf eine oder mehrere der folgenden Weisen reagieren:

1. Der Fehler wird lokalisiert und gemeldet;
2. der Fehler wird diagnostiziert;
3. der Fehler wird korrigiert;
4. der Parser fasst wieder Tritt, um eventuell vorhandene weitere Fehler zu entdecken.

Der erste Schritt ist zwingend erforderlich: andere Übersetzerteile gehen nach der syntaktischen Analyse von einem syntaktisch fehlerfreien Programm aus und erwarten einen Syntaxbaum der Sprache. Auch ist der Benutzer des Übersetzers darauf angewiesen, dass ihm syntaktische Fehler angezeigt werden. Allerdings muss man zwei Einschränkungen machen. In der Nähe eines anderen Syntaxfehlers kann ein Fehler leicht unbemerkt bleiben. Die zweite Einschränkung ist gewichtiger. I.A. entdeckt der Parser einen Fehler dadurch, dass für seine aktuelle Konfiguration keine legale Fortsetzung existiert. Dies ist aber oft nur das *Symptom* für einen vorhandenen Fehler, nicht aber der Fehler selbst.

Beispiel 3.1.1 Betrachten wir die folgende fehlerhafte Zuweisung:

$$a = a * (b + c * d \quad ;$$
$$\uparrow$$
Fehlersymptom: $')'$ fehlt

Hier gibt es mehrere Fehlermöglichkeiten. Entweder ist die öffnende Klammer zuviel, oder es fehlt eine schließende Klammer hinter c oder hinter d. Die Bedeutung der drei möglichen Ausdrücke ist jeweils unterschiedlich. □

Bei Fehlern mit überflüssigen oder fehlenden Klammern {, }, begin, end, if, usw. können Fehlerstelle und Stelle des Fehlersymptoms weit voneinander entfernt liegen. Allerdings haben $LL(k)$- wie $LR(k)$-Parser, die wir im Folgenden betrachten werden, die Eigenschaft des *fortsetzungsfähigen Präfixes*:

> Verarbeitet der Parser für eine kontextfreie Grammatik G ein Präfix u eines Wortes, ohne einen Fehler zu melden, so gibt es ein Wort w, so dass uw ein Satz von G ist.

Parser mit dieser Eigenschaft melden Fehler(symptome) zum frühestmöglichen Zeitpunkt. Obwohl wir i.A. nur das Fehlersymptom und nicht den Fehler selbst entdecken können, werden wir in Zukunft meist von Fehlern sprechen. In diesem Sinne führen die hier vorgestellten Parser den ersten Schritt der Fehlerbehandlung aus: sie melden und lokalisieren Syntaxfehler.

Beispiel 3.1.1 zeigt, dass der zweite Schritt nicht so leicht zu erfüllen ist. Der Parser kann eine Diagnose des Fehlersymptoms lediglich *versuchen*. Diese sollte zumindest folgende Information bereit stellen:

- die Stelle des Fehlersymptoms im Programm;
- die Beschreibung der Parserkonfiguration (der aktuelle Zustand, das erwartete Symbol, das stattdessen gefundene Symbol etc.)

Um den dritten Schritt, die Korrektur eines gefundenen Fehlers, auszuführen, müsste der Parser die Intention des Programmierers ahnen. Dies ist i.A. nicht möglich. Etwas realistischer ist die Suche nach einer global optimalen Fehlerkorrektur. Hierzu wird der Parser um die Fähigkeit erweitert, Symbole in einem Eingabewort einzusetzen bzw. zu löschen. Die *global optimale* Fehlerkorrektur für ein ungültiges Eingabewort w ist ein Wort w', welches durch eine minimale Zahl solcher Einsetz- und Löschoperationen aus w hervorgeht. Solche Verfahren haben aber wegen ihres Aufwands keinen Eingang in die Praxis gefunden.

Stattdessen begnügt man sich meist mit lokalen Einsetzungen oder Ersetzungen, welche den Parser aus der Fehlerkonfiguration in eine neue Konfiguration überführen, in der er zumindest das nächste Eingabesymbol lesen kann. Damit ist gesichert, dass der Parser durch lokale Veränderungen nicht in eine Endlosschleife gerät. Auch sollte das Entstehen von Folgefehlern möglichst vermieden werden.

Der Aufbau dieses Kapitels

Im Abschnitt 3.2 werden die Grundlagen der Syntaxanalyse dargestellt. Kontextfreie Grammatiken mit ihrem Ableitungsbegriff und Kellerautomaten, die zugehörigen Erkennungsmechanismen, werden behandelt. Ein spezieller, nichtdeterministischer Kellerautomat zu einer gegebenen kontextfreien Grammatik G wird konstruiert, der die von G definierte Sprache akzeptiert. Aus diesem Kellerautomaten werden später deterministische *top-down*- und *bottom-up*-Analyseverfahren abgeleitet.

In den Abschnitten 3.3 und 3.4 werden die *top-down*- und die *bottom-up*-Syntaxanalyse vorgestellt. Die entsprechenden Grammatikklassen werden charakterisiert und Generierungsverfahren beschrieben. Ausführlich behandeln wir auch Verfahren zur Fehlerbehandlung.

3.2 Grundlagen

So wie die lexikalische Analyse durch reguläre Ausdrücke spezifiert und durch endliche Automaten implementiert wird, wird die syntaktische Analyse durch kontextfreien Grammatiken spezifiziert und durch Kellerautomaten implementiert. Reguläre Ausdrücke selbst reichen nicht zur Beschreibung der Syntax von Programmiersprachen aus, da reguläre Ausdrücke nicht beliebige rekursive Schachtelungen auszudrücken können, wie sie bei Programmkonstrukten für Blöcke, Anweisungen und Ausdrücke auftreten.

In Abschnitt 3.2.1 und 3.2.3 führen wir die notwendigsten Begriffe über kontextfreie Grammatiken und Kellerautomaten ein. Der Leser, der mit diesen Begriffen vertraut ist, kann diese Abschnitte überschlagen und mit Abschnitt 3.2.4 fortfahren. In Abschnitt 3.2.4 wird zu einer kontextfreien Grammatik ein Kellerautomat eingeführt, der die von der Grammatik definierte Sprache akzeptiert.

3.2.1 Kontextfreie Grammatiken

Mit kontextfreien Grammatiken lässt sich die syntaktische Struktur der Programme einer Programmiersprache beschreiben. Die Grammatik gibt an, wie Programme aus Teilprogrammen zusammengesetzt sind, d.h. welche elementaren Konstrukte es gibt und wie komplexe Konstrukte aus anderen Konstrukten zusammengesetzt werden können.

Beispiel 3.2.1 Ein Auschnitt aus einer Grammatik für eine C-ähnliche Programmiersprache könnte so aussehen:

$$
\begin{array}{lcl}
\langle stat \rangle & \rightarrow & \langle if\_stat \rangle \mid \\
 & & \langle while\_stat \rangle \mid \\
 & & \langle do\_while\_stat \rangle \mid \\
 & & \langle exp \rangle \ ; \mid \\
 & & ; \mid \\
 & & \{ \ \langle stats \rangle \ \} \\
\langle if\_stat \rangle & \rightarrow & \mathsf{if} \ (\ \langle exp \rangle \ ) \ \mathsf{else} \ \langle stat \rangle \mid \\
 & & \mathsf{if} \ (\ \langle exp \rangle \ ) \ \langle stat \rangle \\
\langle while\_stat \rangle & \rightarrow & \mathsf{while} \ (\ \langle exp \rangle \ ) \ \langle stat \rangle \\
\langle do\_while\_stat \rangle & \rightarrow & \mathsf{do} \ \langle stat \rangle \ \mathsf{while} \ (\ \langle exp \rangle \ ); \\
\langle exp \rangle & \rightarrow & \langle assign \rangle \mid \\
 & & \langle call \rangle \mid \\
 & & \mathsf{Id} \mid \\
 & & \ldots \\
\langle call \rangle & \rightarrow & \mathsf{Id} \ (\ \langle exps \rangle \ ) \mid \\
 & & \langle exp \rangle () \\
\langle assign \rangle & \rightarrow & \mathsf{Id} \ '=' \ \langle exp \rangle \\
\langle stats \rangle & \rightarrow & \langle stat \rangle \mid \\
 & & \langle stats \rangle \ \langle stat \rangle \\
\langle exps \rangle & \rightarrow & \langle exp \rangle \mid \\
 & & \langle exps \rangle, \ \langle exp \rangle
\end{array}
$$

Das Nichtterminalsymbol $\langle stat \rangle$ steht für Anweisungen. Wie bei regulären Ausdrücken wird der Operator | verwendet, um mehrere Alternativen für ein Nichtterminal zusammenzufassen. Gemäß diesem Ausschnitt der C-Grammatik ist eine Anweisung eine *if*-Anweisung, eine *while*-Anweisung, eine *do-while*-Anweisung, ein Ausdruck – gefolgt von einem Semikolon, eine leere Anweisung oder eine geklammerte Folge von Anweisungen. Das Nichtterminalsymbol $\langle if\_stat \rangle$ beschreibt *if*-Anweisungen, bei denen der *else*-Teil fehlen kann. Stets beginnen sie mit dem Schlüsselwort if, gefolgt von einem Ausdruck, der in runden Klammern steht, und einer Anweisung. Auf diese Anweisung kann das Schlüsselwort else folgen, zusammen mit einer weiteren Anweisung. Weitere Produktionen beschreiben, wie *while*- und *do-while*-Anweisungen und Ausdrücke aufgebaut sind. Bei Ausdrücken sind nur einige Beispielalternativen angegeben. Weitere Alternativen sind durch ... angedeutet. □

Formal ist eine *kontextfreie Grammatik* ein Quadrupel $G = (V_N, V_T, P, S)$, wobei V_N, V_T disjunkte Alphabete sind, V_N ist die Menge der *Nichtterminale*, V_T die Menge der *Terminale*, $P \subseteq V_N \times (V_N \cup V_T)^*$ ist die endliche Menge der *Produktionsregeln*, und $S \in V_N$ ist das *Startsymbol*.

Die Terminalsymbole (oder kurz: Terminale) sind die Symbole, aus denen zu analysierende Programme aufgebaut sind. Während wir bei der Behandlung der lexikalischen Analyse von einem Alphabet von *Zeichen* gesprochen haben – in der Praxis von den in einem Programm erlaubten Zeichen eines

geeigneten Zeichensatzes wie ASCII oder Unicode – reden wir in diesem Kapitel von *Symbolen*, wie sie etwa von einem Scanner oder Sieber geliefert werden. Solche Symbole sind z.B. Schlüsselwörter oder Bezeichner von Symbolklassen wie z.B. Id, die eine ganze Meng möglicher Symbole repräsentieren.

Die Nichtterminale der Grammatik stehen für die Menge von Wörtern, die aus ihnen produziert werden kann. In der Beispielgrammatik 3.2.1 wurden sie in spitzen Klammern gesetzt. Eine *Produktionsregel* (A, α) aus der Relation P beschreibt mögliche Ersetzungen: ein Vorkommen der linken Seite A in einem Wort $\beta = \gamma_1 A \gamma_2$ kann durch die rechte Seite $\alpha \in (V_T \cup V_N)^*$ ersetzt werden. In der Sicht eines *top-down*-Parsers wird so aus dem Wort β ein neues Wort $\beta' = \gamma_1 \alpha \gamma_2$ *produziert* oder *abgeleitet*.

Ein *bottom-up*-Parser liest die Produktion (A, α) dagegen von rechts nach links und fasst sie als Möglichkeit auf, ein Vorkommen der rechten Seite α in einem Wort $\beta' = \gamma_1 \alpha \gamma_2$ auf das Nichtterminal A der linken Seite zu *reduzieren*. Diese Reduktion führt dann zu dem Wort $\beta = \gamma_1 A \gamma_2$.

Für eine kontextfreie Grammatik $G = (V_N, V_T, P, S)$ führen wir die folgenden Konventionen ein. Lateinische Großbuchstaben vom Anfang des Alphabets, z.B. A, B, C für Nichtterminale aus V_N; lateinische Großbuchstaben vom Ende des Alphabets, z.B. X, Y, Z für Terminale oder Nichtterminale; Kleinbuchstaben am Anfang des lateinischen Alphabets, also $a, b, c, \ldots$, stehen für Terminale, also Elemente aus V_T; Kleinbuchstaben am Ende des lateinischen Alphabets, etwa u, v, w, x, y, z, stehen für Terminalworte, also Elemente aus V_T^*; griechische Kleinbuchstaben, z.B. $\alpha, \beta, \gamma, \varphi, \psi$ stehen für Wörter aus $(V_T \cup V_N)^*$.

Die Relation P fassen wir als Menge von Produktionsregeln auf. Jedes Element (A, α) der Relation schreiben wir deshalb intuitiver als $A \to \alpha$. Alle Produktionen $A \to \alpha_1, A \to \alpha_2, \ldots, A \to \alpha_n$ für ein Nichtterminal A fassen wir auch zu *einer* Deklaration zusammen, die wir als

$$A \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$$

schreiben. Die $\alpha_1, \alpha_2, \ldots, \alpha_n$ heißen die *Alternativen* für A.

Beispiel 3.2.2 Die beiden Grammatiken G_0 und G_1 beschreiben die gleiche Sprache:

$$\begin{aligned}
G_0 = (\{E,T,F\}, \{+,*,(,),\mathsf{Id}\}, P_0, E) \quad & \text{wobei } P_0 \text{ gegeben ist durch:}\\
E &\to E + T \mid T,\\
T &\to T * F \mid F,\\
F &\to (E) \mid \mathsf{Id}\\
G_1 = (\{E\}, \{+,*,(,),\mathsf{Id}\}, P_1, E) \quad & \text{wobei } P_1 \text{ gegeben ist durch:}\\
E &\to E + E \mid E * E \mid (E) \mid \mathsf{Id}
\end{aligned}$$

□

Wir sagen, ein Wort φ *produziert* ein Wort ψ gemäß G *direkt*, i.Z. $\varphi \underset{G}{\Longrightarrow} \psi$, wenn $\varphi = \sigma A \tau, \psi = \sigma \alpha \tau$ gilt für geeignete Wörter σ, τ und eine Produktion $A \to \alpha \in P$. Wir sagen, eine Wort φ *produziert* ein Wort ψ gemäß G (oder auch ψ ist aus φ gemäß G *ableitbar*), i.Z. $\varphi \overset{*}{\underset{G}{\Longrightarrow}} \psi$, wenn es eine endliche Folge $\varphi_0, \varphi_1, \ldots \varphi_n, (n \geq 0)$ von Wörtern gibt, so dass

$$\varphi = \varphi_0, \psi = \varphi_n \text{ und } \varphi_i \underset{G}{\Longrightarrow} \varphi_{i+1} \text{ für alle } 0 \leq i < n.$$

Die Folge $\varphi_0, \varphi_1, \ldots, \varphi_n$ heißt dann eine *Ableitung* von ψ aus φ gemäß G. Falls die Ableitung in n Schritten erfolgt, schreiben wir auch $\varphi \overset{n}{\underset{G}{\Longrightarrow}} \psi$. Die Relation $\overset{*}{\underset{G}{\Longrightarrow}}$ bezeichnet die reflexive und transitive Hülle von $\underset{G}{\Longrightarrow}$.

Beispiel 3.2.3 Die Grammatiken aus Beispiel 3.2.2 erlauben die Ableitungen:

$$\begin{aligned}
E \underset{G_0}{\Longrightarrow} E + T \underset{G_0}{\Longrightarrow} T + T \underset{G_0}{\Longrightarrow} T * F + T \underset{G_0}{\Longrightarrow} T * \mathsf{Id} + T \underset{G_0}{\Longrightarrow} F * \mathsf{Id} + T \underset{G_0}{\Longrightarrow}\\
F * \mathsf{Id} + F \underset{G_0}{\Longrightarrow} \mathsf{Id} * \mathsf{Id} + F \underset{G_0}{\Longrightarrow} \mathsf{Id} * \mathsf{Id} + \mathsf{Id},\\
E \underset{G_1}{\Longrightarrow} E + E \underset{G_1}{\Longrightarrow} E * E + E \underset{G_1}{\Longrightarrow} \mathsf{Id} * E + E \underset{G_1}{\Longrightarrow} \mathsf{Id} * E + \mathsf{Id} \underset{G_1}{\Longrightarrow} \mathsf{Id} * \mathsf{Id} + \mathsf{Id}\ .
\end{aligned}$$

Wir schließen, dass sowohl $E \overset{*}{\underset{G_1}{\Longrightarrow}} \mathsf{Id} * \mathsf{Id} + \mathsf{Id}$ gilt wie $E \overset{*}{\underset{G_0}{\Longrightarrow}} \mathsf{Id} * \mathsf{Id} + \mathsf{Id}$. □

Für eine kontextfreie Grammatik $G = (V_N, V_T, P, S)$ definieren wir die von G *definierte* (erzeugte) Sprache als die Menge

$$L(G) = \{u \in V_T^* \mid S \overset{*}{\underset{G}{\Longrightarrow}} u\}$$

Ein Wort $x \in L(G)$ nennen wir einen *Satz* von G. Ein Wort $\alpha \in (V_T \cup V_N)^*$ mit $S \overset{*}{\underset{G}{\Longrightarrow}} \alpha$ nennen wir eine *Satzform* von G.

Beispiel 3.2.4 Betrachten wir erneut die beiden Grammatiken aus Beispiel 3.2.3. Das Wort $\mathsf{Id} * \mathsf{Id} + \mathsf{Id}$ ist ein Satz sowohl von G_0 als auch von G_1 , da sowohl $E \overset{*}{\underset{G_0}{\Longrightarrow}} \mathsf{Id} * \mathsf{Id} + \mathsf{Id}$ als auch $E \overset{*}{\underset{G_1}{\Longrightarrow}} \mathsf{Id} * \mathsf{Id} + \mathsf{Id}$ gilt. □

Wenn die Grammatik aus dem Kontext klar ist, auf die sich Ableitungen beziehen, lassen wir im Folgenden den Index G in $\underset{G}{\Longrightarrow}$ weg.

Die syntaktische Struktur eines Programms, wie sie sich als Resultat der syntaktischen Analyse ergibt, ist ein *geordneter Baum*, d.h. ein Baum, in dem die Ausgangskanten jedes Knoten geordnet sind. Dieser Baum beschreibt eine standardisierte Sicht auf den Ableitungsvorgang. Innerhalb eines Übersetzers dient er als *Schnittstelle* zu nachfolgenden Übersetzerkomponenten. Die meisten Verfahren zur Auswertung semantischer Attribute im Kapitel 4 über semantische Analyse arbeiten auf dieser Baumstruktur.

Sei $G = (V_N, V_T, P, S)$ eine kontextfreie Grammatik Sei t ein geordneter Baum, dessen innere Knoten mit Symbolen aus V_N und dessen Blätter mit Symbolen aus $V_T \cup V_N \cup \{\varepsilon\}$ beschriftet sind. Dann ist t ein *Syntaxbaumfragment* oder *Strukturbaumfragment*, wenn die Beschriftung X jedes inneren Knotens n von t zusammen mit der Folge der Beschriftungen $X_1, \ldots, X_k$ der Kinder von n in t die folgenden Eigenschaften besitzt:

1. $X \to X_1 \ldots X_k$ ist eine Produktion aus P.
2. Ist $X_1 \ldots X_k = \varepsilon$, dann ist $k = 1$, d.h. der Knoten n hat genau ein Kind und dieses is mit ε beschriftet.
3. Ist $X_1 \ldots X_k \neq \varepsilon$, dann ist $X_i \neq \varepsilon$ für jedes i.

Ist das Symbol an der Wurzel von t das Nichtterminal A und ergibt die Konkatenation der Beschriftungen der Folge der Blätter von t das Wort α, dann nennen wir t ein Syntaxbaumfragment t für das Wort α und X gemäß G. Ist die Folge der Beschriftungen α ein Terminalwort, d.h. aus V_T^*, dann nennen wir t einen *Syntaxbaum* für α und A.

Beispiel 3.2.5 Abbildung 3.1 zeigt zwei Syntaxbäume gemäß der Grammatik G_1 aus Beispiel 3.2.2 für das Wort $\mathsf{Id} * \mathsf{Id} + \mathsf{Id}$. □

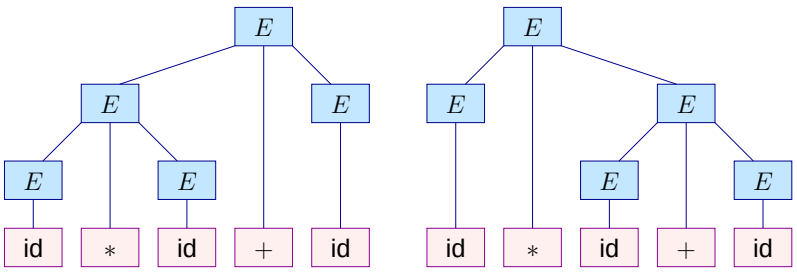


Abb. 3.1. Zwei Syntaxbäume gemäß der Grammatik G_1 aus Beispiel 3.2.2 für das Wort $\mathsf{Id} * \mathsf{Id} + \mathsf{Id}$.

Ein Syntaxbaum kann als eine Darstellung von Ableitungen aufgefasst werden, bei der von der Reihenfolge abstrahiert wird, in der die Vorkommen von Nichtterminalsymbolen ersetzt werden. Besitzt

derselbe Satz $w \in L(G)$ dennoch *mehr* als einen Syntaxbaum für das Startsymbol S gemäß der Grammatik G, dann heißt der Satz w *mehrdeutig*. Entsprechend heißt die Grammatik G *mehrdeutig*, wenn $L(G)$ mindestens einen mehrdeutigen Satz enthält. Eine kontextfreie Grammatik, die nicht mehrdeutig ist, nennen wir *eindeutig*.

Beispiel 3.2.6 Die Grammatik G_1 ist mehrdeutig, da der Satz $\mathsf{Id} * \mathsf{Id} + \mathsf{Id}$ mehr als einen Syntaxbaum besitzt. Die Grammatik G_0 ist dagegen eindeutig. □

Aus der Definition ergibt sich, dass jeder Satz $x \in L(G)$ mindestens eine Ableitung aus S besitzt. Zu jeder Ableitung für den Satz x gehört wiederum ein Syntaxbaum für x. Jeder Satz $x \in L(G)$ besitzt deshalb mindestens einen Syntaxbaum. Umgekehrt gibt es zu jedem Syntaxbaum für einen Satz x mindestens eine Ableitung für x. Diese Ableitung lässt sich leicht aus dem Syntaxbaum ablesen.

Beispiel 3.2.7 Das Wort $\mathsf{Id} + \mathsf{Id}$ hat gemäß der Grammatik G_1 den einen Syntaxbaum aus Abbildung 3.2. Je nachdem, in welcher Reihenfolge die Nichtterminale abgeleitet werden, ergeben sich die beiden Ableitungen:

$$E \Rightarrow E + E \Rightarrow \mathsf{Id} + E \Rightarrow \mathsf{Id} + \mathsf{Id}$$
$$E \Rightarrow E + E \Rightarrow E + \mathsf{Id} \Rightarrow \mathsf{Id} + \mathsf{Id}$$

□

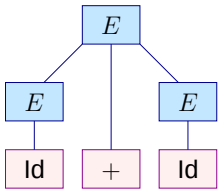


Abb. 3.2. Der eindeutige Syntaxbaum für den Satz $\mathsf{Id} + \mathsf{Id}$.

Wir haben in Beispiel 3.2.7 gesehen, dass – auch bei eindeutigen Wörtern – zu einem Syntaxbaum mehrere Ableitungen korrespondieren können. Diese ergeben sich aus den verschiedenen Möglichkeiten, in einer Satzform Nichtterminale für die nächste Anwendung einer Produktion auszuwählen. Wenn festgelegt wird, dass jeweils das am weitesten links bzw. das am weitesten rechts stehende Nichtterminal ersetzt wird, ergeben sich ausgezeichnete Ableitungen, nämlich *Links*- bzw. *Rechts*-Ableitungen.

Eine Ableitung $\varphi_1 \Longrightarrow \ldots \Longrightarrow \varphi_n$ von $\varphi = \varphi_n$ aus $S = \varphi_1$ ist eine *Linksableitung* von φ, i.Z. $S \overset{*}{\underset{lm}{\Longrightarrow}} \varphi$, wenn beim Schritt von φ_i nach φ_{i+1} jeweils das in φ_i am weitesten links stehende Nichtterminal ersetzt wird, d.h. $\varphi_i = uA\tau,\ \varphi_{i+1} = u\alpha\tau$ für ein Wort $u \in V_T^*$ und eine Produktion $A \to \alpha \in P$.

Analog nennen wir die Ableitung $\varphi_1 \Longrightarrow \ldots \Longrightarrow \varphi_n$ *Rechtsableitung* von φ, geschrieben $S \overset{*}{\underset{rm}{\Longrightarrow}} \varphi$, wenn jeweils das am weitesten rechts stehende Nichtterminal ersetzt wird, d.h. $\varphi_i = \sigma Au$, $\varphi_{i+1} = \sigma\alpha u$ mit $u \in V_T^*$ und $A \to \alpha \in P$.

Eine Satzform, die in einer Linksableitung (Rechtsableitung) auftritt, heißt *Linkssatzform (Rechtssatzform)*.

Zu jedem Syntaxbaum für S gibt es genau eine Links- und genau eine Rechtsableitung. Folglich gibt es zu jedem *eindeutigen* Satz auch genau eine Links- und genau eine Rechtsableitung.

Beispiel 3.2.8 Das Wort $\mathsf{Id} * \mathsf{Id} + \mathsf{Id}$ hat gemäß der Grammatik G_1 die Linksableitungen

$$E \underset{lm}{\Longrightarrow} E + E \underset{lm}{\Longrightarrow} E * E + E \underset{lm}{\Longrightarrow} \mathsf{Id} * E + E \underset{lm}{\Longrightarrow} \mathsf{Id} * \mathsf{Id} + E \underset{lm}{\Longrightarrow} \mathsf{Id} * \mathsf{Id} + \mathsf{Id} \quad \text{und}$$
$$E \underset{lm}{\Longrightarrow} E * E \underset{lm}{\Longrightarrow} \mathsf{Id} * E \underset{lm}{\Longrightarrow} \mathsf{Id} * E + E \underset{lm}{\Longrightarrow} \mathsf{Id} * \mathsf{Id} + E \underset{lm}{\Longrightarrow} \mathsf{Id} * \mathsf{Id} + \mathsf{Id}.$$

Es hat die Rechtsableitungen

$$E \underset{rm}{\Longrightarrow} E + E \underset{rm}{\Longrightarrow} E + \mathsf{Id} \underset{rm}{\Longrightarrow} E * E + \mathsf{Id} \underset{rm}{\Longrightarrow} E * \mathsf{Id} + \mathsf{Id} \underset{rm}{\Longrightarrow} \mathsf{Id} * \mathsf{Id} + \mathsf{Id} \quad \text{und}$$
$$E \underset{rm}{\Longrightarrow} E * E \underset{rm}{\Longrightarrow} E * E + E \underset{rm}{\Longrightarrow} E * E + \mathsf{Id} \underset{rm}{\Longrightarrow} E * \mathsf{Id} + \mathsf{Id} \underset{rm}{\Longrightarrow} \mathsf{Id} * \mathsf{Id} + \mathsf{Id}.$$

Das Wort $\mathsf{Id} + \mathsf{Id}$ hat in G_1 nur jeweils eine Linksableitung, nämlich

$$E \underset{lm}{\Longrightarrow} E + E \underset{lm}{\Longrightarrow} \mathsf{Id} + E \underset{lm}{\Longrightarrow} \mathsf{Id} + \mathsf{Id}$$

und eine Rechtsableitung, nämlich

$$E \underset{rm}{\Longrightarrow} E + E \underset{rm}{\Longrightarrow} E + \mathsf{Id} \underset{rm}{\Longrightarrow} \mathsf{Id} + \mathsf{Id}.$$

□

Wird ein Wort einer eindeutigen Grammatik einmal durch eine Links- und einmal durch eine Rechtsableitung abgeleitet, dann bestehen die beiden Ableitungen aus den gleichen Produktionen. Sie unterscheiden sich lediglich in der *Reihenfolge* ihrer Anwendungen. Die Frage ist daher, ob man in beiden Ableitungen jeweils Satzformen finden kann, die dadurch miteinander korrespondieren, dass jeweils im nächsten Schritt das gleiche Vorkommen eines Nichtterminals ersetzt wird? Das folgende Lemma stellt eine solche Beziehung her.

Lemma 3.1. 1. Wenn $S \underset{lm}{\overset{*}{\Longrightarrow}} uA\varphi$ gilt, dann gibt es ein ψ, mit $\psi \overset{*}{\Longrightarrow} u$, so dass für alle v mit $\varphi \overset{*}{\Longrightarrow} v$ gilt: $S \underset{rm}{\Longrightarrow} \psi Av$.

2. Wenn $S \underset{rm}{\overset{*}{\Longrightarrow}} \psi Av$ gilt, dann gibt es ein φ mit $\varphi \overset{*}{\Longrightarrow} v$, so dass für alle u mit $\psi \overset{*}{\Longrightarrow} u$ gilt: $S \underset{lm}{\overset{*}{\Longrightarrow}} uA\varphi$. □

Abbildung 3.3 verdeutlicht die Zusammenhänge zwischen φ und v einerseits und ψ und u andererseits.

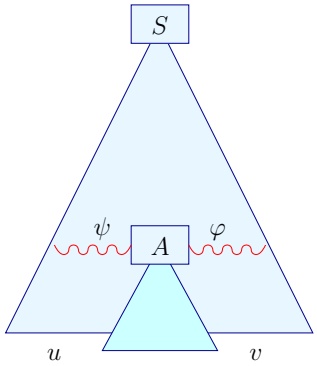


Abb. 3.3. Zusammenhang zwischen Rechts- und Linksableitung.

Von kontextfreien Grammatiken, welche die Syntax von Programmiersprachen beschreiben, verlangt man meist, dass sie eindeutig sind. Dann gibt es zu jedem syntaktisch korrekten Programm, d.h. zu jedem Satz der Grammatik, genau einen Syntaxbaum und folglich auch nur genau eine Links- und genau eine Rechtsableitung.

3.2.2 Produktivität und Erreichbarkeit von Nichtterminalen

Kontextfreie Grammatiken haben eventuell überflüssige Nichtterminale und Produktionen. Deren Beseitigung macht die Grammatiken kleiner, verändert aber nicht die erzeugte Sprache. Im Folgenden

werden zwei Eigenschaften von Nichtterminalen definiert, welche diese als nützlich ausweisen, und es werden Verfahren angegeben, diese Eigenschaften festzustellen und die in diesem Sinne nutzlosen Nichtterminale zu entfernen.

Die erste Eigenschaft eines nützlichen Nichtterminals ist seine *Produktivität*. Ein Nichtterminal X einer kontextfreien Grammatik $G = (V_N, V_T, P, S)$ nennen wir *produktiv*, wenn es eine Ableitung $X \overset{*}{\underset{G}{\Longrightarrow}} w$ gibt für ein Wort $w \in V_T^*$, d.h., wenn es einen Ableitungsbaum gibt, dessen Wurzel mit X beschriftet ist.

Beispiel 3.2.9 Betrachten wir die Grammatik $G = (\{S', S, X, Y, Z\}, \{a, b\}, P, S')$, wobei P aus den Produktionen:

$$\begin{aligned} S' &\rightarrow S \\ S &\rightarrow aXZ \mid Y \\ X &\rightarrow bS \mid aYbY \\ Y &\rightarrow ba \mid aZ \\ Z &\rightarrow aZX \end{aligned}$$

besteht. Dann ist Y produktiv und deshalb sind auch X, S und S' produktiv. Das Nichtterminal Z ist dagegen nicht produktiv, da die einzige Produktion für Z selbst wieder Z enthält. □

Eine zweistufige Charakterisierung, die zu einem Algorithmus für die Berechnung der Produktivität führt, ist die folgende:

(1) X ist *produktiv über die Produktion* p genau dann, wenn X die linke Seite von p ist, und wenn alle Nichtterminale, die in der rechten Seite von p vorkommen, produktiv sind.
(2) X ist *produktiv*, wenn X über mindestens eine seiner Alternativen produktiv ist.

Insbesondere ist X damit produktiv, wenn es eine Produktion $X \rightarrow u \in P$ gibt, deren rechte Seite u keine Nichtterminale enthält, d.h. es gilt $u \in V_T^*$.

Eigenschaft (1) beschreibt die Abhängigkeit der Information zu X von den Informationen über die Symbole in der rechten Seite einer Produktion für X; Eigenschaft (2) gibt an, wie die Information, die aus den verschiedenen Alternativen für X erhalten wurde, zu kombinieren ist.

Wir stellen ein Verfahren vor, das zu einer kontextfreien Grammatik G die Menge aller produktiven Nichtterminal berechnet. Das Verfahren führt für jede Produktion p einen *Zähler* $\mathsf{count}[p]$ ein, der die Anzahl der Vorkommen von Nichtterminalen zählt, von denen noch nicht bekannt ist, dass sie produktiv sind. Sinkt der Zähler einer Produktion p auf 0, müssen alle Nichtterminale auf der rechten Seite produktiv sein. Daraus folgt, dass dann die linke Seite von p produktiv über p ist. Zur Verwaltung der Produktionen, deren Zähler auf 0 gesunken ist, verwendet der Algorithmus die *Arbeitsliste* W.

Weiterhin wird für jedes Nichtterminal X eine Liste $\mathsf{occ}[X]$ der Vorkommen dieses Nichtterminals in rechten Seiten von Produktionen verwaltet:

```
set⟨nonterminal⟩   productive ← ∅;          //  Ergebnis-Menge
int   count[production];                    //  Zähler für jede Regel
list⟨nonterminal⟩   W ← [ ];
list⟨production⟩   occ[nonterminal];        //  Vorkommen in rechten Seiten

forall (nonterminal X)   occ[X] ← [ ];      //  Initialisierung
forall (production p)   { count[p] ← 0;
                          init(p);
                          }
...
```

Der Aufruf $\mathsf{init}(p)$ der Hilfsprozedur $\mathsf{init}()$ für eine Produktion p, deren Code wir nicht angegeben haben, iteriert über die Folge der Symbole auf der rechten Seite von p. Bei jedem Vorkommen eines Nichtterminals X wird der Zähler $\mathsf{count}[p]$ inkrementiert und p zu der Liste $\mathsf{occ}[X]$ hinzugefügt. Gilt

am Ende immer noch $\mathsf{count}[p] = 0$, dann fügt $\mathsf{init}(p)$ die Produktion p in die Liste W ein. Damit ist die Initialisierung abgeschlossen.

In der Hauptiteration werden sukzessive die Produktionen aus W abgearbeitet. Für jede Produktion p aus W ist die linke Seite produktiv über p und damit produktiv. Wird andererseits ein Nichtterminal X neuerdings als produktiv entdeckt, dann wird über die Liste $\mathsf{occ}[X]$ der Produktionen iteriert, in denen X vorkommt. Bei jeder Produktion r aus dieser Liste wird der Zähler $\mathsf{count}[r]$ erniedrigt. Das liefert uns den folgenden Algorithmus:

```
...
while (W ≠ [])  {
    X ← hd(W); W ← tl(W);
    if (X ∉ productive)  {
        productive ← productive ∪ {X};
        forall ((r : A → α) ∈ occ[X])  {
            count[r]−−;
            if (count[r] = 0)  W ← A :: W;
            }                                   // end of forall
        }                                       // end of if
    }                                           // end of while
```

Im Wesentlichen läuft die Initialisierungsphase einmal über die Grammatik und führt dabei pro Symbol nur konstant viel Arbeit aus. Während der Hauptiteration über die Arbeitsliste wird für jede Regel die linke Seite maximal einmal in die Liste W eingefügt und kann damit auch nur maximal einmal aus der Liste W entfernt werden. Bei jeder Entnahme eines Nichtterminals X aus W fällt nur dann mehr als konstant viel Arbeit an, wenn X noch nicht als produktiv markiert war. Der Aufwand für ein solches X ist proportional zu der Länge der Liste $\mathsf{occ}[X]$. Die *Summe* dieser Längen ist beschränkt durch die Gesamtgröße der Grammatik G. Daraus folgt, dass der Gesamtaufwand linear in der Größe der Grammatik ist.

Zur Korrektheit des Verfahrens vergewissern wir uns, dass es die folgenden Eigenschaften besitzt:

- Falls in der j-ten Iteration der *while*-Schleife X in die Menge productive eingefügt wird, gibt es einen Syntaxbaum für X der Höhe maximal $j - 1$.
- Für jeden Syntaxbaum wird die Wurzel einmal in W eingefügt.

Der effiziente Algorithmus, den wir gerade vorgestellt haben, hat Bedeutung auch über seine Anwendungen im Übersetzerbau hinaus. Mit geringen Änderungen kann er auch eingesetzt werden, um die *kleinsten* Lösungen von *Booleschen* Gleichungssystemen zu bestimmen, d.h. von Gleichungssystemen, bei denen die rechten Seiten Disjunktionen beliebiger Konjunktionen von Unbekannten sind. In unserem Beispiel stammen die Konjunktionen von den rechten Seiten, während eine Disjunktion die Existenz verschiedener Alternativen für ein Nichtterminal repräsentiert.

Die zweite Eigenschaft eines nützlichen Nichtterminals ist seine *Erreichbarkeit*. Ein Nichtterminal X nennen wir *erreichbar* bzgl. einer kontextfreien Grammatik $G = (V_N, V_T, P, S)$, wenn es eine Ableitung $S \overset{*}{\underset{G}{\Longrightarrow}} \alpha X \beta$ gibt.

Beispiel 3.2.10 Betrachten wir die Grammatik $G = (\{S, U, V, X, Y, Z\}, \{a, b, c, d\}, P, S)$, wobei die Menge P aus den folgenden Produktionen besteht:

$$\begin{array}{lll} S \to Y & & X \to c \\ Y \to YZ \mid Ya \mid b & & V \to Vd \mid d \\ U \to V & & Z \to ZX \end{array}$$

Dann sind die Nichtterminale S, Y, Z und X erreichbar, U und V dagegen nicht. □

Auch für Erreichbarkeit gibt es eine Charakterisierung, aus der wir einen Algorithmus ableiten können.

(1) Das Startsymbol S ist stets erreichbar.
(2) Ist ein Nichtterminal X erreichbar und $X \to \alpha \in P$, dann ist auch jedes Nichtterminal erreichbar, das in der rechten Seite α vorkommt.

Sei für jedes Nichtterminal X $\mathsf{rhs}[X]$ die Menge aller Nichtterminale, die in rechten Seiten von Produktionen der Grammatik vorkommen, deren linke Seite X ist. Diese Mengen können in linearer Zeit berechnet werden. Dann können wir die Menge $reachable$ der erreichbaren Nichtterminale einer Grammatik berechnen durch:

$$
\begin{array}{l}
\mathbf{set}\langle nonterminal\rangle\ \ reachable \leftarrow \emptyset;\\
\mathbf{list}\langle nonterminal\rangle\ \ W \leftarrow S :: [\,];\\
nonterminal\ Y;\\
\mathbf{while}\ (W \neq [\,])\ \{\\
\quad X \leftarrow \mathsf{hd}(W);\ \ W \leftarrow \mathsf{tl}(W);\\
\quad \mathbf{if}\ (X \notin reachable)\ \{\\
\qquad reachable \leftarrow reachable \cup \{X\};\\
\qquad \mathbf{forall}\ (Y \in \mathsf{rhs}[X])\ W \leftarrow W \cup \{Y\};\\
\qquad \}
\end{array}
$$

Um eine Grammatik G zu reduzieren, werden zuerst die unproduktiven Nichtterminale zusammen mit allen Produktionen, in denen sie vorkommen, beseitigt. Erst in einem zweiten Schritt werden die nicht erreichbaren Nichtterminale beseitigt. In diesem zweiten Schritt kann man wir deshalb davon ausgehen, dass sämtliche noch vorhandenen Nichtterminale produktiv sind.

Beispiel 3.2.11 Betrachten wir erneut die Grammatik aus Beispiel 3.2.9 mit den Produktionen:

$$
\begin{array}{lcl}
S' & \to & S\\
S & \to & aXZ \mid Y\\
X & \to & bS \mid aYbY\\
Y & \to & ba \mid aZ\\
Z & \to & aZX
\end{array}
$$

Die Menge der produktiven Nichtterminale ist $\{S', S, X, Y\}$, während Z nicht produktiv ist. Um eine reduzierte Grammatik zu berechnen, werden deshalb im ersten Schritt alle Produktionen gestrichen, in denen Z vorkommt. Es ergibt sich die Menge P_1:

$$
\begin{array}{lcl}
S' & \to & S\\
S & \to & Y\\
X & \to & bS \mid aYbY\\
Y & \to & ba
\end{array}
$$

Obwohl X bzgl. der ursprünglichen Menge von Produktionen erreichbar war, ist X nicht mehr erreichbar. Die Menge der erreichbaren Nichtterminale ist $V'_N = \{S', S, Y\}$. Indem alle Produktionen gestrichen werden, deren linke Seite nicht erreichbar ist, ergibt sich die Menge:

$$
\begin{array}{lcl}
S' & \to & S\\
S & \to & Y\\
Y & \to & ba
\end{array}
$$

□

Wir nehmen im Folgenden immer an, dass Grammatiken reduziert sind.

3.2.3 Kellerautomaten

In diesem Abschnitt behandeln wir das Automatenmodell, das kontextfreien Grammatiken entspricht, den Kellerautomaten. Für die Realisierung einer Komponente zur Syntaxanalyse benötigen wir ein Verfahren, das zu einer gegebenen kontextfreien Grammatik einen Kellerautomaten konstruiert, welcher die wohlgeformten Sätze der Grammatik analysieren kann. In Abschnitt 3.2.4 beschreiben wir eine solche Konstruktion. Allerdings hat der Kellerautomat, den wir zu einer kontextfreien Grammatik konstruieren, einen kleinen Schönheitsfehler: für fast alle Grammatiken ist er nichtdeterministisch. In den Abschnitten 3.3 und 3.4 wird beschrieben, wie diese Konstruktion für geeignete Unterklassen kontextfreier Grammatiken so modifiziert werden kann, dass wir deterministische Automaten erhalten.

Im Gegensatz zu den endlichen Automaten, die im letzten Kapitel behandelt wurden, verfügt ein Kellerautomat über eine unbegrenzte Speicherfähigkeit.

Die (konzeptuell) unbeschränkte Datenstruktur *Keller*, mit der er ausgestattet ist, arbeitet nach dem *last-in-first-out*-Prinzip. Abbildung 3.4 zeigt eine schematische Darstellung eines Kellerautomaten. Wie

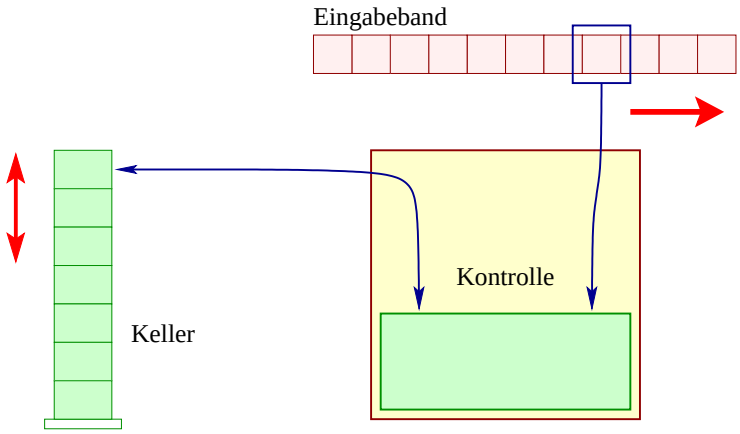


Abb. 3.4. Schematische Darstellung eines Kellerautomaten.

bei einem endlichen Automaten darf sich der Lesekopf nur von links nach rechts bewegen. Im Unterschied zu einem endlichen Automaten wird ein Übergang des Automaten nicht durch das aktuelle Symbol auf dem Eingabeband und und einen Zustand, sondern durch das aktuelle Symbol auf dem Eingabeband und einem oberen Abschnitt auf dem Keller bestimmt. Der Übergang kann diesen oberen Abschnitt des Kellers verändern und den Eingabekopf gegebenenfalls auf dem Eingabeband nach rechts rücken.

Formal definieren wir einen *Kellerautomaten* als ein Tupel $P = (Q, V_T, \Delta, q_0, F)$, wobei

- Q die endliche Menge der *Zustände*,
- V_T das *Eingabealphabet*,
- $q_0 \in Q$ der *Anfangszustand* und
- $F \subseteq Q$ die Menge der *Endzustände* sind, und
- Δ, eine endliche Teilmenge von $Q^+ \times V_T \times Q^*$, die *Übergangsrelation* ist. Die Übergangsrelation Δ kann auch als eine endliche partielle Funktion Δ von $Q^+ \times V_T$ in die endlichen Teilmengen von Q^* aufgefasst werden.

Unsere Definition eines Kellerautomaten ist insofern etwas ungewöhnlich, als sie nicht zwischen Zuständen des Automaten und Kellersymbolen unterscheidet, sondern für beide das gleiche Alphabet verwendet. Entsprechend wird das oberste Kellersymbol als der *aktuelle* Zustand angesehen und der Automat in die Lage versetzt, auch die Symbole unterhalb des obersten Kellersymbols zu berücksichtigen.

Die Übergangsrelation beschreibt die möglichen Berechnungsschritte des Kellerautomaten. Sie listet endlich viele Übergänge auf. Die Anwendung eines Übergangs (γ, x, γ') ersetzt den oberen Abschnitt $\gamma \in Q^+$ des Kellers durch die neue Folge $\gamma' \in Q^*$ von Zuständen und liest dabei $x \in V_T \cup \{\varepsilon\}$ in der Eingabe. Der ersetzte Kellerabschnitt hat dabei mindestens die Länge 1. Ein Übergang, bei dem das nächste Eingabesymbol nicht angesehen wird, heißt ε-*Übergang*.

Auch für Kellerautomaten führen wir den Begriff einer *Konfiguration* ein. Eine Konfiguration umfasst alle Komponenten, welche für die zukünftigen Schritte des Automaten relevant sind. Bei unserer Art von Kellerautomaten sind das der Kellerinhalt und die restliche Eingabe. Formal ist damit eine *Konfiguration* des Kellerautomaten P ein Paar $(\gamma, w) \in Q^+ \times V_T^*$. In dieser linearisierten Darstellung befindet sich das obere Ende des Kellers am *rechten* Ende von γ, während das nächste zu lesende Symbol *links* am Anfang von w steht. Ein *Übergang* von P wird durch die binäre Relation $\vdash_P$ zwischen Konfigurationen dargestellt. Diese Relation ist definiert durch:

$$(\gamma, w) \vdash_P (\gamma', w') \quad \text{, falls} \quad \gamma = \alpha\beta,\ \gamma' = \alpha\beta', \quad w = xw' \quad \text{und} \quad (\beta, x, \beta') \in \Delta$$

für ein geeignetes $\alpha \in Q^*$. Wie bei endlichen Automaten ist eine *Berechnung* eine Folge von Konfigurationen, zwischen denen jeweils ein Übergang besteht. Wir schreiben $C \vdash_P^n C'$, wenn es Konfigurationen $C_1, \ldots, C_{n+1}$ gibt, so dass $C_1 = C,\ C_{n+1} = C'$ und $C_i \vdash_P C_{i+1}$ für $1 \leq i \leq n$ gilt. Die Relationen $\vdash_P^+$ und $\vdash_P^*$ sind die transitive bzw. die reflexive und transitive Hülle von $\vdash_P$. Es gilt:

$$\vdash_P^+ \ = \bigcup_{n \geq 1} \vdash_P^n \quad \text{und} \quad \vdash_P^* \ = \bigcup_{n \geq 0} \vdash_P^n$$

Eine Konfiguration (q_0, w) für ein $w \in V_T^*$ heißt eine *Anfangskonfiguration*, (q, ε), für $q \in F$, eine *Endkonfiguration* des Kellerautomaten P. Ein Wort $w \in V_T^*$ wird von dem Kellerautomaten P *akzeptiert* wenn $(q_0, w) \vdash_P^* (q, \varepsilon)$ gilt für ein $q \in F$. Die *Sprache* $L(P)$ des Kellerautomaten P ist die Menge der von P akzeptierten Wörter:

$$L(P) = \{w \in V_T^* \mid \exists f \in F :\ (q_0, w) \vdash_P^* (f, \varepsilon)\}$$

Ein Wort w wird also von einem Kellerautomaten akzeptiert, wenn es *mindestens* eine Berechnung gibt, die von der Anfangskonfiguration (q_0, w) zu einer Endkonfiguration führt. Solche Berechnungen heißen *akzeptierend*. Für ein Wort kann es gegebenenfalls mehrere akzeptierende Berechnungen geben zusammen mit vielen Berechnungen, die nur einen Präfix des Worts w lesen können oder zwar w lesen, aber trotzdem keine Endkonfiguration erreichen. Da man in der Praxis akzeptierende Berechnungen nicht durch Probieren herausfinden möchte, sind *deterministische* Kellerautomaten von besonderer Bedeutung.

Ein Kellerautomat P heißt *deterministisch*, wenn die Übergangsrelation Δ die folgende Eigenschaft erfüllt:

(D) Falls $(\gamma_1, x, \gamma_2), (\gamma_1', x', \gamma_2')$ zwei verschiedende Übergänge aus Δ sind und γ_1' ein Suffix von γ_1 ist, dann sind x und x' aus Σ und verschieden, d.h. $x \neq \varepsilon \neq x'$ und $x \neq x'$.

Besitzt die Übergangsrelation die Eigenschaft (D), dann gibt es in jeder Konfiguration höchstens einen Übergang.

3.2.4 Der Item-Kellerautomat zu einer kontextfreien Grammatik

In diesem Abschnitt wird ein Verfahren angegeben, mit dem zu jeder kontextfreien Grammatik ein Kellerautomat konstruiert werden kann, der die von der Grammatik definierte Sprache akzeptiert. Dieser Automat ist nichtdeterministisch und deshalb für einen praktischen Einsatz nur bedingt geeignet. Durch geeignete Entwurfsentscheidungen können wir aus ihm jedoch sowohl die LL-Analysatoren aus Abschnitt 3.3, als auch die LR-Analysatoren aus Abschnitt 3.4 ableiten.

Eine entscheidende Rolle spielt der Begriff des kontextfreien *Items*. Sei $G = (V_N, V_T, P, S)$ eine kontextfreie Grammatik. Ein *kontextfreies Item* von G ist ein Tripel (A, α, β) mit $A \to \alpha\beta \in P$. Dieses Tripel schreiben wir intuitiver als $[A \to \alpha.\beta]$. Das Item $[A \to \alpha.\beta]$ beschreibt die Situation, dass bei

dem Versuch, ein Wort w aus A abzuleiten, bereits ein Präfix von w erfolgreich aus α abgeleitet wurde. α heißt deshalb die *Geschichte* des Items.

Ein Item $[A \to \alpha.\beta]$ mit $\beta = \varepsilon$ heißt *vollständig*. Die Menge aller kontextfreien Items von G bezeichnen wir mit It_G. Ist ρ die Folge von Items:

$$\rho = [A_1 \to \alpha_1.\beta_1][A_2 \to \alpha_2.\beta_2] \ldots [A_n \to \alpha_n.\beta_n]$$

dann bezeichnet $\mathsf{hist}(\rho)$ die Konkatenation der Geschichten der Items von ρ, d.h.

$$\mathsf{hist}(\rho) = \alpha_1\alpha_2 \ldots \alpha_n.$$

Zu der kontextfreien Grammatik $G = (V_N, V_T, P, S)$ konstruieren wir nun den *Item-Kellerautomaten*. Als Zustände und damit auch als Kellersymbole verwendet er die Items der Grammatik. Der aktuelle Zustand ist das Item, an dessen rechter Seite der Automat gerade arbeitet. Im Keller darunter stehen die Items, von deren rechten Seiten die Bearbeitung bereits begonnen, aber noch nicht vollendet wurde.

Bevor wir den Item-Kellerautomaten konstruieren, wollen wir die Grammatik G jedoch so erweitern, dass die Terminierung des konstruierten Kellerautomaten eindeutig am aktuellen Zustand abgelesen werden kann. Ist S das Startsymbol der Grammatik, dann sind die Kandidaten für die Endzustände des Item-Kellerautomaten alle vollständigen Items $[S \to \alpha.]$ der Grammatik. Tritt S auch auf der rechten Seite einer Produktion auf, können solche vollständigen Items jedoch auch oben auf dem Keller auftreten, ohne dass der Automat deswegen terminieren sollte, da darunter noch unvollständig bearbeitete Items liegen. Darum erweitern wir die Grammatik G um ein neues Startsymbol S', das in keiner rechten Seite vorkommt. Für S' fügen wir die Produktion $S' \to S$ zu der Menge der Produktionen von G hinzu. Als Startzustand des Item-Kellerautomaten für die erweiterte Grammatik wählen wir das Item $[S' \to .S]$ und als einzigen Endzustand das vollständige Item $[S' \to S.]$. Der *Item-Kellerautomat* zu der Grammatik G ist dann der Kellerautomat

$$K_G = (\mathsf{It}_G, V_T, \Delta, [S' \to .S], \{[S' \to S.]\})$$

wobei die Übergangsrelation Δ drei Typen von Übergängen bereit stellt:

$$\begin{array}{lll}
(E) & \Delta([X \to \beta.Y\gamma], \varepsilon) & = \{[X \to \beta.Y\gamma][Y \to .\alpha] \mid Y \to \alpha \in P\} \\
(L) & \Delta([X \to \beta.a\gamma], a) & = \{[X \to \beta a.\gamma]\} \\
(R) & \Delta([X \to \beta.Y\gamma][Y \to \alpha.], \varepsilon) & = \{[X \to \beta Y.\gamma]\}.
\end{array}$$

Übergänge gemäß (E) heißen *Expansionsübergänge*, solche gemäß (L) *Leseübergänge* und solche gemäß (R) *Reduktionsübergänge*.

Für eine Folge von Items, die als Kellerinhalt in einer Berechnung des Item-Kellerautomaten auftreten, gilt die folgende Invariante (I):

(I) Falls $([S' \to .S], uv) \vdash_{K_G}^* (\rho, v)$, dann gilt: $\mathsf{hist}(\rho) \underset{G}{\Longrightarrow}^* u$.

Diese Invariante ist ein wesentlicher Bestandteil des Beweises, dass der Item-Kellerautomat K_G nur Sätze von G akzeptiert, d.h. dass $L(K_G) \subseteq L(G)$ ist. Wir erläutern nun die Arbeitsweise des Automaten K_G und führen gleichzeitig einen Induktionsbeweis (über die Länge von Berechnungen), dass die Invariante (I) für jede aus einer Anfangskonfiguration erreichbare Konfiguration erfüllt ist.

Betrachten wir zuerst die Anfangskonfiguration für die Eingabe w. Diese Anfangskonfiguration ist $([S' \to .S], w)$. Das Wort $u = \varepsilon$ wurde bereits gelesen, $\mathsf{hist}([S' \to .S]) = \varepsilon$, und es gilt $\varepsilon \underset{G}{\Longrightarrow}^* \varepsilon$. Deshalb gilt die Invariante in dieser Konfiguration.

Betrachten wir nun Ableitungen, die mindestens einen Übergang enthalten. Nehmen wir als erstes an, der letzte Übergang sei ein Expansionsübergang. Dann wurde vor diesem Übergang aus der Anfangskonfiguration $([S' \to .S], uv)$ eine Konfiguration $(\rho[X \to \beta.Y\gamma], v)$ erreicht. Diese Konfiguration erfüllt nach Induktionsvoraussetzung die Invariante (I), d.h. es gilt $\mathsf{hist}(\rho)\beta \underset{G}{\Longrightarrow}^* u$. Als aktueller Zustand legt das Item $[X \to \beta.Y\gamma]$ nahe, einen Präfix von v aus Y abzuleiten. Dazu sollte der Automat nichtdeterministisch eine der Alternativen für Y auswählen. Gerade das beschreiben die Übergänge gemäß (E). Alle möglichen Folgekonfigurationen $(\rho[X \to \beta.Y\gamma][Y \to .\alpha], v)$ für $Y \to \alpha \in P$ erfüllen ebenfalls die Invariante (I), denn

$$\mathsf{hist}(\rho[X \to \beta.Y\gamma][Y \to .\alpha]) = \mathsf{hist}(\rho)\beta \overset{*}{\Longrightarrow} u\,.$$

Als nächstes nehmen wir an, der letzte Übergang sei ein Leseübergang. Dann wurde vor diesem Übergang aus $([S' \to .S], uav)$ eine Konfiguration $(\rho[X \to \beta.a\gamma], av)$ erreicht. Diese Konfiguration erfüllt nach Induktionsvoraussetzung die Invariante (I), d.h. es gilt $\mathsf{hist}(\rho)\beta \overset{*}{\Longrightarrow} u$. Dann erfüllt die Nachfolgekonfiguration $(\rho[X \to \beta a.\gamma], v)$ ebenfalls die Invariante (I), denn

$$\mathsf{hist}(\rho[X \to \beta a.\gamma]) = \mathsf{hist}(\rho)\beta a \overset{*}{\Longrightarrow} ua$$

Nehmen wir schließlich an, dass der letzte Schritt ein Reduktionsübergang ist. Vor dem letzten Schritt wurde aus $([S' \to .S], uv)$ eine Konfiguration $(\rho[X \to \beta.Y\gamma][Y \to \alpha.], v)$ erreicht. Nach Induktionsvoraussetzung erfüllt sie die Invariante (I), d.h. es gilt $\mathsf{hist}(\rho)\beta\alpha \overset{*}{\underset{G}{\Longrightarrow}} u$. Der aktuelle Zustand ist das vollständige Item $[Y \to \alpha.]$. Er ist das Ergebnis einer Berechnung, die mit dem Item $[Y \to .\alpha]$ begonnen wurde, als $[X \to \beta.Y\gamma]$ aktueller Zustand war und die Alternative $Y \to \alpha$ für Y ausgewählt wurde. Diese Alternative wurde erfolgreich abgearbeitet. Die Nachfolgekonfiguration $(\rho[X \to \beta Y.\gamma], v)$ erfüllt ebenfalls die Invariante (I), denn aus $\mathsf{hist}(\rho)\beta\alpha \overset{*}{\underset{G}{\Longrightarrow}} u$ folgt $\mathsf{hist}(\rho)\beta Y \overset{*}{\underset{G}{\Longrightarrow}} u$. □

Insgesamt gilt der folgende Satz:

Satz 3.2.1 Für jede kontextfreie Grammatik G gilt $L(K_G) = L(G)$.

Beweis. Nehmen wir an, $w \in L(K_G)$. Dann gilt

$$([S' \to .S], w) \overset{*}{\underset{K_G}{\vdash}} ([S' \to S.], \varepsilon)\,.$$

Wegen der Invariante (I), die wir bereits bewiesen haben, folgt, dass

$$S = \mathsf{hist}([S' \to S.]) \overset{*}{\underset{G}{\Longrightarrow}} w$$

Deshalb ist $w \in L(G)$. Für die umgekehrte Richtung nehmen wir an, dass $w \in L(G)$ ist. Dann gilt $S \overset{*}{\underset{G}{\Longrightarrow}} w$. Um zu zeigen, dass dann auch

$$([S' \to .S], w) \overset{*}{\underset{K_G}{\vdash}} ([S' \to S.], \varepsilon)$$

gilt, zeigt man allgemeiner, dass für jede Ableitung $A \underset{G}{\Longrightarrow} \alpha \overset{*}{\underset{G}{\Longrightarrow}} w$ mit $A \in V_N$ gilt, dass

$$(\rho[A \to .\alpha], wv) \overset{*}{\underset{K_G}{\vdash}} (\rho[A \to \alpha.], v)$$

für beliebige $\rho \in \mathsf{lt}_G^*$ und beliebige $v \in V_T^*$. Diese allgemeinere Behauptung lässt sich durch Induktion über die Länge der Ableitung $A \underset{G}{\Longrightarrow} \alpha \overset{*}{\underset{G}{\Longrightarrow}} w$ beweisen. □

Beispiel 3.2.12 Sei $G' = (\{S, E, T, F\}, \{+, *, (,), \mathsf{Id}\}, P', S)$ die Erweiterung der Grammatik G_0 um das neue Startsymbol S. Die Menge der Produktionen P' ist gegeben durch:

$$\begin{array}{lcl} S & \to & E \\ E & \to & E + T \mid T \\ T & \to & T * F \mid F \\ F & \to & (E) \mid \mathsf{Id} \end{array}$$

Die Übergangsrelation Δ von K_{G_0} ist in Tabelle 3.1 beschrieben. Eine akzeptierende Berechnung von K_{G_0} für das Wort $\mathsf{Id} + \mathsf{Id} * \mathsf{Id}$ zeigt Tabelle 3.2. □

oberes Kellerende	Eingabe	neues oberes Kellerende
$[S \to .E]$	ε	$[S \to .E][E \to .E + T]$
$[S \to .E]$	ε	$[S \to .E][E \to .T]$
$[E \to .E + T]$	ε	$[E \to .E + T][E \to .E + T]$
$[E \to .E + T]$	ε	$[E \to .E + T][E \to .T]$
$[F \to (.E)]$	ε	$[F \to (.E)][E \to .E + T]$
$[F \to (.E)]$	ε	$[F \to (.E)][E \to .T]$
$[E \to .T]$	ε	$[E \to .T][T \to .T * F]$
$[E \to .T]$	ε	$[E \to .T][T \to .F]$
$[T \to .T * F]$	ε	$[T \to .T * F][T \to .T * F]$
$[T \to .T * F]$	ε	$[T \to .T * F][T \to .F]$
$[E \to E + .T]$	ε	$[E \to E + .T][T \to .T * F]$
$[E \to E + .T]$	ε	$[E \to E + .T][T \to .F]$
$[T \to .F]$	ε	$[T \to .F][F \to .(E)]$
$[T \to .F]$	ε	$[T \to .F][F \to .\mathsf{Id}]$
$[T \to T * .F]$	ε	$[T \to T * .F][F \to .(E)]$
$[T \to T * .F]$	ε	$[T \to T * .F][F \to .\mathsf{Id}]$
$[F \to .(E)]$	(	$[F \to (.E)]$
$[F \to .\mathsf{Id}]$	Id	$[F \to \mathsf{Id}.]$
$[F \to (E.)]$	)	$[E \to (E).]$
$[E \to E. + T]$	$+$	$[E \to E + .T]$
$[T \to T. * F]$	$*$	$[T \to T * .F]$
$[T \to .F][F \to \mathsf{Id}.]$	ε	$[T \to F.]$
$[T \to T * .F][F \to \mathsf{Id}.]$	ε	$[T \to T * F.]$
$[T \to .F][F \to (E).]$	ε	$[T \to F.]$
$[T \to T * .F][F \to (E).]$	ε	$[T \to T * F.]$
$[T \to .T * F][T \to F.]$	ε	$[T \to T. * F]$
$[E \to .T][T \to F.]$	ε	$[E \to T.]$
$[E \to E + .T][T \to F.]$	ε	$[E \to E + T.]$
$[E \to E + .T][T \to T * F.]$	ε	$[E \to E + T.]$
$[T \to .T * F][T \to T * F.]$	ε	$[T \to T. * F]$
$[E \to .T][T \to T * F.]$	ε	$[E \to T.]$
$[F \to (.E)][E \to T.]$	ε	$[F \to (E.)]$
$[F \to (.E)][E \to E + T.]$	ε	$[F \to (E.)]$
$[E \to .E + T][E \to T.]$	ε	$[E \to E. + T]$
$[E \to .E + T][E \to E + T.]$	ε	$[E \to E. + T]$
$[S \to .E][E \to T.]$	ε	$[S \to E.]$
$[S \to .E][E \to E + T.]$	ε	$[S \to E.]$

Tabelle 3.1. Tabellarische Darstellung der Übergangsrelation aus Beispiel 3.2.12. Die mittlere Spalte gibt die konsumierte Eingabe an.

Kellerinhalt	restliche Eingabe
$[S \to .E]$	$\mathsf{Id} + \mathsf{Id} * \mathsf{Id}$
$[S \to .E][E \to .E + T]$	$\mathsf{Id} + \mathsf{Id} * \mathsf{Id}$
$[S \to .E][E \to .E + T][E \to .T]$	$\mathsf{Id} + \mathsf{Id} * \mathsf{Id}$
$[S \to .E][E \to .E + T][E \to .T][T \to .F]$	$\mathsf{Id} + \mathsf{Id} * \mathsf{Id}$
$[S \to .E][E \to .E + T][E \to .T][T \to .F][F \to .\mathsf{Id}]$	$\mathsf{Id} + \mathsf{Id} * \mathsf{Id}$
$[S \to .E][E \to .E + T][E \to .T][T \to .F][F \to \mathsf{Id}.]$	$+\mathsf{Id} * \mathsf{Id}$
$[S \to .E][E \to .E + T][E \to .T][T \to F.]$	$+\mathsf{Id} * \mathsf{Id}$
$[S \to .E][E \to .E + T][E \to T.]$	$+\mathsf{Id} * \mathsf{Id}$
$[S \to .E][E \to E. + T]$	$+\mathsf{Id} * \mathsf{Id}$
$[S \to .E][E \to E + .T]$	$\mathsf{Id} * \mathsf{Id}$
$[S \to .E][E \to E + .T][T \to .T * F]$	$\mathsf{Id} * \mathsf{Id}$
$[S \to .E][E \to E + .T][T \to .T * F][T \to .F]$	$\mathsf{Id} * \mathsf{Id}$
$[S \to .E][E \to E + .T][T \to .T * F][T \to .F][F \to .\mathsf{Id}]$	$\mathsf{Id} * \mathsf{Id}$
$[S \to .E][E \to E + .T][T \to .T * F][T \to .F][F \to \mathsf{Id}.]$	$*\mathsf{Id}$
$[S \to .E][E \to E + .T][T \to .T * F][T \to F.]$	$*\mathsf{Id}$
$[S \to .E][E \to E + .T][T \to T. * F]$	$*\mathsf{Id}$
$[S \to .E][E \to E + .T][T \to T * .F]$	Id
$[S \to .E][E \to E + .T][T \to T * .F][F \to .\mathsf{Id}]$	Id
$[S \to .E][E \to E + .T][T \to T * .F][F \to \mathsf{Id}.]$	
$[S \to .E][E \to E + .T][T \to T * F.]$	
$[S \to .E][E \to E + T.]$	
$[S \to E.]$	

Tabelle 3.2. Die akzeptierende Berechnung von K_G für das Wort $\mathsf{Id} + \mathsf{Id} * \mathsf{Id}$.

Kellerautomaten mit Ausgabe

Kellerautomaten als solche sind nur Akzeptoren, d.h. sie entscheiden nur, ob ein vorgelegtes Wort ein Satz der Sprache ist oder nicht. Will man dagegen einen Kellerautomaten zur syntaktischen Analyse in einem Übersetzer benutzen, so interessiert aber auch die syntaktische Struktur akzeptierter Wörter. Diese kann in der Form eines Syntaxbaums oder in der Folgen der in einer Rechts- bzw. Linksableitung angewandten Produktionen darstellt werden. Deshalb erweitern wir Kellerautomaten um entsprechende Ausgabemöglichkeiten.

Ein Kellerautomat *mit Ausgabe* ist ein Tupel $P = (Q, V_T, O, \Delta, q_0, F)$, wobei Q, V_T, q_0, F wie bei einem normalen Kellerautomaten definiert sind und O ein endliches Ausgabealphabet ist. Die Relation Δ ist eine endliche Relation zwischen $Q^+ \times (V_T \cup \{\varepsilon\})$ und $Q^* \times (O \cup \{\varepsilon\})$. Eine *Konfiguration* beschreibt den aktuellen Kellerinhalt, die verbleibende Eingabe und die bereits getätigte Ausgabe. Sie ist deshalb ein Element aus $Q^+ \times V_T^* \times O^*$.

Bei jedem Übergang kann der Automat ein Symbol aus O ausgeben. Setzen wir einen Kellerautomaten mit Ausgabe als Parser ein, so besteht sein Ausgabealphabet aus den Produktionen der kontextfreien Grammatik oder ihren Nummern.

Den Item-Kellerautomaten können wir auf zwei Weisen um Möglichkeiten zur Ausgabe erweitern. Wird bei jeder Expansion die angewandte Produktion ausgegeben, ist die gesamte Ausgabe für jede akzeptierende Berechnung eine Linksableitung für das akzeptierte Wort. Einen Kellerautmaten mit einer solchen Ausgabe nennen wir einen *Linksparser*.

Anstatt bei den Expansionen könnte der Item-Kellerautomat auch bei jeder Reduktion die angewandte Produktion ausgeben. Dann liefert seine Ausgabe für eine akzeptierende Berechnung eine *gespiegelte Rechtsableitung* für das akzeptierte Wort. Einen Kellerautomaten mit einer solchen Ausgabe nennen wir einen *Rechtsparser*.

Deterministische Parser

In Satz 3.2.1 haben wir bewiesen, dass der Item-Kellerautomat K_G zu einer kontextfreien Grammatik G genau deren Sprache $L(G)$ akzeptiert. Jedoch ist die nichtdeterministische Arbeitsweise des Kellerautomaten für die Praxis ungeeignet. Die Quelle des Nichtdeterminismus liegt in den Übergängen des Typs (E): der Item-Kellerautomat kann bei jedem Expansionsübergang für ein Nichtterminal zwischen mehreren Alternativen auswählen. Bei einer eindeutigen Grammatik ist jedoch höchstens eine davon die richtige, um einen Präfix der restlichen Eingabe abzuleiten. Die anderen Alternativen führen früher oder später in die Irre. Der Item-Kellerautomat kann diese richtige Alternative jedoch nur *raten*.

In den Abschnitten 3.3 und 3.4 geben wir zwei Methoden an, die das Raten beseitigen. Die LL-Analysatoren aus Abschnitt 3.3 wählen deterministisch eine Alternative für das aktuelle Nichtterminal aus und benutzen dazu eine beschränkte Vorausschau in die restliche Eingabe. Nicht für alle Grammatiken, aber für Grammatiken vom Typ $LL(k)$ kann deterministisch ein (E)-Übergang ausgewählt werden, wenn die bereits gelesene Eingabe, das zu expandierende Nichtterminal und die nächsten k Eingabesymbole in Betracht gezogen werden. LL-Analysatoren sind Linksparser.

Die LR-Analysatoren arbeiten anders. Sie *verzögern* die Entscheidung, die der LL-Analysator bei der Expansion trifft, bis zu den Reduktionsstellen. Zu jedem Zeitpunkt der Analyse verfolgen sie parallel alle Möglichkeiten, die noch zu einer gespiegelten Rechtsableitung für das Eingabewort führen können. Erst, wenn eine der Möglichkeiten eine Reduktion verlangt, muss der Analysator eine Entscheidung treffen. Diese Entscheidung betrifft die Frage, ob weiter gelesen oder reduziert werden soll, und im zweiten Fall mit welcher Produktion reduziert werden soll. Für diese Entscheidung wird der aktuelle Kellerinhalt und ebenfalls eine beschränkte Zahl von Symbolen Vorausschau herangezogen. Weil LR-Analysatoren Reduktionen melden, sind sie Rechtsparser. Auch LR-Analysatoren existieren nicht für jede kontextfreie Grammatik, sondern nur für Grammatiken vom Typ $LR(k)$, wobei k wieder die Anzahl der benötigten Symbole Vorausschau angibt.

3.2.5 first- und follow-Mengen

Betrachten wir den Item-Kellerautomaten K_G zu einer kontextfreien Grammatik G bei einer Expansion, d.h. einem (E)-Übergang. Vor einem solchen Übergang ist K_G in einem aktuellen Zustand der Form $[X \to \alpha.Y\beta]$. In diesem Zustand muss der Kellerautomat K_G nichtdeterministisch aus den Alternativen $Y \to \alpha_1 \mid \ldots \mid \alpha_n$ für das Nichtterminal Y auswählen. Für diese Entscheidung ist es hilfreich, die Mengen der Wörter zu kennen, die von den verschiedenen Alternativen produziert werden können. Wenn der Anfang der restlichen Eingabe nur zu Wörtern einer Alternative $Y \to \alpha_i$ passt, wird diese Alternative auszuwählen sein. Wenn einige der Alternativen auch kurze Wörter, z.B. ε, produzieren können, ist ebenfalls die Menge der Wörter oder ihrer Anfänge interessant, die auf Y folgen können.

Da die Mengen der Wörter, die von einer Alternative produziert werden können, i.A. unendlich sind, begnügt man sich mit den Mengen der *Präfixe* dieser Wörter von einer vorgegebenen Länge k. Diese Mengen sind endlich, Der erzeugte Parser gründet seine Entscheidungen auf einem Vergleich eines Präfixes der restlichen Eingabe der Länge k mit den Elementen dieser vorberechneten endlichen Mengen. Für diesen Zweck führen wir die Funktionen first_k und follow_k ein.

Für ein Alphabet V_T schreiben wir $V_T^{\leq k}$ für $\bigcup_{i=0}^{k} V_T^i$ und $V_{T,\#}^{\leq k}$ für $V^{\leq k} \cup (V_T^{\leq k-1}\{\#\})$, wobei $\#$ ein Symbol ist, das nicht in V_T enthalten ist. Wie das Dateiendesymbol eof markiert es das Ende eines Wortes. Sei $w = a_1 \ldots a_n$ ein Wort mit $a_i \in V_T$ für $(1 \leq i \leq n)$, $n \geq 0$. Für $k \geq 0$ definieren wir den *k-Präfix* von w als:

$$w|_k = \begin{cases} a_1 \ldots a_n & \text{falls} \quad n \leq k \\ a_1 \ldots a_k & \text{sonst} \end{cases}$$

Weiterhin führen wir den Operator $\odot_k : V_T \times V_T \to V_T^{\leq k}$ ein, der definiert ist durch

$$u \odot_k v = (uv)|_k$$

Diesen Operator nennen wir *k-Konkatenation*. Beide Operationen erweitern wir auf Mengen von Wörtern. Für Mengen $L \subseteq V_T^*$ und $L_1, L_2 \subseteq V^{\leq k}$ definieren wir

$$L|_k = \{w|_k \mid w \in L\} \quad \text{und} \quad L_1 \odot_k L_2 = \{x \odot_k y \mid x \in L_1, y \in L_2\}\,.$$

Sei $G = (V_N, V_T, P, S)$ eine kontextfreie Grammatik. Für $k \geq 1$ definieren wir die Funktion $\mathsf{first}_k : (V_N \cup V_T)^* \to 2^{V_T^{\leq k}}$, die zu jedem Wort α die Menge aller Präfixe der Länge k von Terminalworten liefert, die aus α ableitbar sind:

$$\mathsf{first}_k(\alpha) = \{u|_k \mid \alpha \stackrel{*}{\Longrightarrow} u\}$$

Entsprechend liefert die Funktion $\mathsf{follow}_k : V_N \to 2^{V_{T,\#}^{\leq k}}$ zu jedem Nichtterminal X die Menge der Terminalworte der Länge kleiner oder gleich k, die in einer Satzform direkt auf X folgen können:

$$\mathsf{follow}_k(X) = \{w \in V_T^* \mid S \stackrel{*}{\Longrightarrow} \beta X \gamma \text{ und } w \in \mathsf{first}_k(\gamma\#)\}$$

Die Menge $\mathsf{first}_k(X)$ besteht aus den k-Präfixen der Blattwörter aller Bäume für X. $\mathsf{follow}_k(X)$ aus den k-Präfixen des zweiten Teils der Blattwörter aller oberen Baumfragmente für X (siehe Abbildung 3.5). Einige Eigenschaften der k-Konkatenation und der Funktion first_k beschreibt das folgende Lemma.

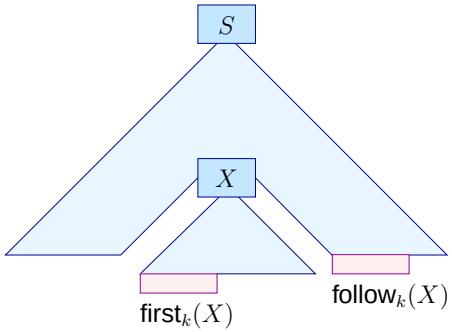


Abb. 3.5. first_k und follow_k im Syntaxbaum

Lemma 3.2. Sei $k \geq 1$, und seien $L_1, L_2, L_3 \subseteq V^{\leq k}$ gegeben. Dann gilt:

(a) $L_1 \odot_k (L_2 \odot_k L_3) = (L_1 \odot_k L_2) \odot_k L_3$
(b) $L_1 \odot_k \{\varepsilon\} = \{\varepsilon\} \odot_k L_1 = L_1|_k$
(c) $L_1 \odot_k L_2 = \emptyset$ gdw. $L_1 = \emptyset \vee L_2 = \emptyset$
(d) $\varepsilon \in L_1 \odot_k L_2$ gdw. $\varepsilon \in L_1 \wedge \varepsilon \in L_2$
(e) $(L_1 L_2)|_k = L_1|_k \odot_k L_2|_k$
(f) $\mathsf{first}_k(X_1 \ldots X_n) = \mathsf{first}_k(X_1) \odot_k \ldots \odot_k \mathsf{first}_k(X_n)$ für $X_1, \ldots, X_n \in (V_T \cup V_N)$

Die Beweise zu (b), (c), (d) und (e) sind trivial. (a) ergibt sich aus Fallunterscheidungen über die Länge von Wörtern $x \in L_1$, $y \in L_2$, $z \in L_3$. Der Beweis zu (f) benutzt (e) sowie die Beobachtung, dass $X_1 \ldots X_n \stackrel{*}{\Longrightarrow} u$ gilt genau dann, wenn $u = u_1 \ldots u_n$ gilt für geeignete Worte u_i mit $X_i \stackrel{*}{\Longrightarrow} u_i$.

Wegen der Eigenschaft (f) lässt sich die Berechnung der Menge $\mathsf{first}_k(\alpha)$ auf die Berechnung der Mengen $\mathsf{first}_k(X)$ für einzelne Symbole $X \in V_T \cup V_N$ zurückführen. Da $\mathsf{first}_k(a) = \{a\}$ für $a \in V_T$ gilt, genügt es, die Mengen $\mathsf{first}_k(X)$ für Nichtterminale X zu ermitteln. Ein Wort $w \in V_T^{\leq k}$ ist genau dann in $\mathsf{first}_k(X)$ enthalten, wenn w für eine der Produktionen $X \to \alpha \in P$ in der Menge $\mathsf{first}_k(\alpha)$ enthalten ist. Wegen der Eigenschaft (f) des Lemmas 3.2 erfüllen die first_k-Mengen deshalb das Gleichungssystem (fi):

$$\mathsf{first}_k(X) = \bigcup \{\mathsf{first}_k(X_1) \odot_k \ldots \odot_k \mathsf{first}_k(X_n) \mid X \to X_1 \ldots X_n \in P\}\,, X_i \in V_N \qquad \text{(fi)}$$

Beispiel 3.2.13 Sei G_2 die kontextfreie Grammatik mit den Produktionen:

$$\begin{array}{llll} 0: & S \to E & 3: & E' \to +E \quad 6: \quad T' \to *T \\ 1: & E \to TE' & 4: & T \to FT' \quad 7: \quad F \to (E) \\ 2: & E' \to \varepsilon & 5: & T' \to \varepsilon \quad 8: \quad F \to \mathsf{Id} \end{array}$$

G_2 erzeugt die gleiche Sprache der arithmetischen Ausdrücke wie G_0 und G_1. Als Gleichungssystem zur Berechnung der first_1-Mengen erhalten wir:

$$\begin{array}{l} \mathsf{first}_1(S) = \mathsf{first}_1(E) \\ \mathsf{first}_1(E) = \mathsf{first}_1(T) \odot_1 \mathsf{first}_1(E') \\ \mathsf{first}_1(E') = \{\varepsilon\} \cup \{+\} \odot_1 \mathsf{first}_1(E) \\ \mathsf{first}_1(T) = \mathsf{first}_1(F) \odot_1 \mathsf{first}_1(T') \\ \mathsf{first}_1(T') = \{\varepsilon\} \cup \{*\} \odot_1 \mathsf{first}_1(T) \\ \mathsf{first}_1(F) = \{\mathsf{Id}\} \cup \{(\} \odot_1 \mathsf{first}_1(E) \odot_1 \{)\} \end{array}$$

□

Die rechten Seiten des Gleichungssystems zur Berechnung der first_k-Mengen können als Ausdrücke repräsentiert werden, die aus Unbekannten $\mathsf{first}_k(Y), Y \in V_N$ und den Mengenkonstanten $\{x\}, x \in V_T \cup \{\varepsilon\}$, mit Hilfe der Operatoren $\odot_k$ und $\cup$ aufgebaut sind. Es stellen sich sofort mehrere Fragen:

- Ist dieses Gleichungssystem lösbar, d.h. besitzt dieses Gleichungssystem Lösungen?
- Wenn ja, welche Lösung ist diejenige, die den first_k-Mengen entspricht?
- Wie berechnet man diese gewünschte Lösung?

Zur Beantwortung dieser Fragen betrachten wir zuerst einmal generell Gleichungssysteme wie (fi) und untersuchen eine algorithmische Idee, wie eine Lösung berechnet werden könnte. Sei $\mathbf{x}_1, \ldots, \mathbf{x}_n$ eine Folge von Unbekannten,

$$\begin{array}{c} \mathbf{x}_1 = f_1(\mathbf{x}_1, \ldots, \mathbf{x}_n) \\ \mathbf{x}_2 = f_2(\mathbf{x}_1, \ldots, \mathbf{x}_n) \\ \vdots \\ \mathbf{x}_n = f_n(\mathbf{x}_1, \ldots, \mathbf{x}_n) \end{array}$$

ein Gleichungssystem, das über einem Bereich $\mathbb{D}$ gelöst werden soll. Dabei bezeichnet jedes f_i auf der rechten Seite eine Funktion $f_i : \mathbb{D}^n \to \mathbb{D}$. Eine Lösung I^* dieses Gleichungssystems ordnet jeder Unbekannten $\mathbf{x}_i$ einen Wert $I^*(\mathbf{x}_i)$ zu, so dass alle Gleichungen erfüllt sind, d.h. dass

$$I^*(\mathbf{x}_i) = f_i(I^*(\mathbf{x}_1), \ldots, I^*(\mathbf{x}_n))$$

gilt für alle $i = 1, \ldots, n$.

Nehmen wir an, $\mathbb{D}$ enthalte ein ausgezeichnetes Element d_0, das sich als Startwert zur Berechnung einer Lösung anbietet. Eine einfache Idee zur Berechnung einer Lösung besteht dann darin, zuerst einmal alle Unbekannten $\mathbf{x}_1, \ldots, \mathbf{x}_n$ auf diesen Startwert d_0 zu setzen. Sei $I^{(0)}$ diese Variablenbelegung. Mit dieser Belegung der Unbekannten werden nun alle rechten Seiten f_i ausgewertet. Dann erhält jede Variable $\mathbf{x}_i$ einen neuen Wert. Alle diese Werte bilden eine neue Variablenbelegung $I^{(1)}$, in der erneut die rechten Seiten ausgewertet werden, usw. Nehmen wir also an, wir haben bereits eine Variablenbelegung $I^{(j)}$ berechnet. Dann erhalten wir die Variablenbelegung $I^{(j+1)}$ durch:

$$I^{(j+1)}(\mathbf{x}_i) = f_i(I^{(j)}(\mathbf{x}_1), \ldots, I^{(j)}(\mathbf{x}_n))$$

Es ergibt sich eine Folge $I^{(0)}, I^{(1)}, \ldots$ von Variablenbelegungen. Gilt für irgend ein $j \geq 0$, dass $I^{(j+1)} = I^{(j)}$, dann folgt, dass

$$I^{(j)}(\mathbf{x}_i) = f_i(I^{(j)}(\mathbf{x}_1), \ldots, I^{(j)}(\mathbf{x}_n)) \qquad (i = 1, \ldots, n)$$

gilt. Damit ist $I^{(j)} = I^*$ eine Lösung.

Ohne weitere Voraussetzungen ist allerdings unklar, ob ein j mit $I^{(j+1)} = I^{(j)}$ jemals erreicht wird. In den speziellen Fällen, die wir in diesem Band betrachten, können wir allerdings garantieren, dass dieses Verfahren nicht nur gegen eine Lösung, sondern sogar gegen die Lösung konvergiert, die wir benötigen. Denn wir wissen einiges über die Bereiche $\mathbb{D}$, die in unserer Anwendung auftreten:

- Auf $\mathbb{D}$ gibt es stets eine *Halbordnung*, dargestellt durch das Symbol $\sqsubseteq$. Bei den first_k-Mengen besteht die Menge $\mathbb{D}$ aus allen Teilmengen der (endlichen) Grundmenge $V_T^{\leq k}$ aller Terminalwörter der Länge höchstens k. Die Halbordnung auf diesem Bereich ist die *Teilmengenrelation*.
- $\mathbb{D}$ enthält als ausgezeichnetes Element, mit dem die Iteration starten kann, ein eindeutig bestimmtes *kleinstes Element*. Dieses Element bezeichnen wir mit $\bot$ (bottom). Bei den first_k-Mengen ist dieses kleinste Element die *leere* Menge.
- Für jede Teilmenge $Y \subseteq \mathbb{D}$ gibt es bzgl. der Relation $\sqsubseteq$ stets eine *kleinste obere Schranke* $\bigsqcup Y$. Bei den first_k-Mengen ist die kleinste obere Schranke einer Menge von Mengen ihre Vereinigung. Halbordnungen mit dieser Eigenschaft heißen *vollständige Verbände*.

Weiterhin sind alle Funktionen f_i *monoton*, d.h. sie respektieren die Ordnungsrelation $\sqsubseteq$ auf ihren Argumenten. Bei den first_k-Mengen gilt dies, weil die rechten Seiten der Gleichungen aus den Operatoren Vereinigung und k-Konkatenation aufgebaut sind, die beide monoton sind, und Kompositionen monotoner Funktionen wieder monoton sind.

Wird der Algorithmus mit $d_0 = \bot$ gestartet, dann folgt daraus, dass $I^{(0)} \sqsubseteq I^{(1)}$ gilt. Dabei nehmen wir an, dass eine Variablenbelegung kleiner oder gleich einer anderen ist, wenn dies für die Werte jeder Variablen gilt. Aus der Monotonie der Funktionen f_i folgt dann mittels Induktion, dass wir eine *aufsteigende* Folge:

$$I^{(0)} \sqsubseteq I^{(1)} \sqsubseteq I^{(2)} \sqsubseteq \ldots I^{(k)} \sqsubseteq \ldots$$

von Variablenbelegungen erhalten. Ist der Bereich $\mathbb{D}$ endlich, gibt es ein j, sodass $I^{(j)} = I^{(j+1)}$ gilt, d.h. der Algorithmus findet tatsächlich eine Lösung. Es lässt sich sogar zeigen, dass diese Lösung die *kleinste* Lösung ist. Eine solche kleinste Lösung existiert auch dann, wenn der vollständige Verband nicht endlich ist und die einfache Iteration nicht terminiert. Dies folgt aus dem Fixpunktsatz von Knaster-Tarski, den wir im dritten Band *Übersetzerbau: Analyse und Transformation* ausführlich behandeln.

Beispiel 3.2.14 Wenden wir dieses Verfahren zur Bestimmung einer Lösung auf das Gleichungssystem aus Beispiel 3.2.13 an. Am Anfang wird allen Nichtterminalen die leere Menge zugeordnet. Die Wörter, die in der i-ten Iteration zu den jeweiligen first_1-Mengen hinzugefügt werden, zeigt die folgende Tabelle:

	1	2	3	4	5	6	7	8
S				Id			(	
E			Id			(		
E'	ϵ			+				
T		Id			(			
T'	ϵ		*					
F	Id			(				

Als Ergebnis erhalten wir deshalb:

$$\begin{aligned} \mathsf{first}_1(S) &= \{\mathsf{Id}, (\} & \mathsf{first}_1(T) &= \{\mathsf{Id}, (\} \\ \mathsf{first}_1(E) &= \{\mathsf{Id}, (\} & \mathsf{first}_1(T') &= \{\varepsilon, *\} \\ \mathsf{first}_1(E') &= \{\varepsilon, +\} & \mathsf{first}_1(F) &= \{\mathsf{Id}, (\} \end{aligned}$$

□

Um die Anwendbarkeit des iterativen Algorithmus auf ein gegebenes Gleichungssystem über einem vollständigen Verband zu garantieren, reicht es also nachzuweisen, dass alle rechten Seiten monoton sind und der Wertebereich endlich ist.

Mit dem folgenden Satz vergewissern wir uns, dass die *kleinste* Lösung des Gleichungssystems (fi) tatsächlich die first_k-Mengen charakterisiert.

Satz 3.2.2 (Korrektheit der first_k-Mengen) Sei $G = (V_N, V_T, P, S)$ eine kontextfreie Grammatik, $\mathbb{D}$ der vollständige Verband der Teilmengen von $V_T^{\leq k}$ und $I : V_N \to \mathbb{D}$ die kleinste Lösung des Gleichungssystems (fi). Dann gilt:

$$I(X) = \mathsf{first}_k(X) \quad \text{für alle } X \in V_N$$

Beweis. Für $i \geq 0$ sei $I^{(i)}$ die Variablenbelegung nach der i-ten Iteration des Lösungsverfahrens für (fi). Durch Induktion über i zeigt man, dass für alle $i \geq 0$ $I^{(i)}(X) \subseteq \mathsf{first}_k(X)$ gilt für alle $X \in V_N$. Deshalb gilt auch $I(X) = \bigcup_{i\geq 0}(X) \subseteq \mathsf{first}_k(X)$ für alle $X \in V_N$. Für die umgekehrte Richtung genügt es zu zeigen, dass für jede Ableitung $X \overset{*}{\underset{lm}{\Longrightarrow}} w$ ein $i \geq 0$ existiert mit $w|_k \in I^{(i)}(X)$. Diese Behauptung wird erneut mit Induktion bewiesen, diesmal durch Induktion über die Länge $n \geq 1$ der Linksableitung. Ist $n = 1$, besitzt die Grammatik eine Produktion $X \to w$. Dann gilt:

$$I^{(1)}(X) \supseteq \mathsf{first}_k(w) = \{w|_k\}$$

und die Behauptung folgt mit $i = 1$. Ist $n > 1$, dann gibt es eine Produktion $X \to u_0X_1u_1 \ldots X_mu_m$ mit $u_0, \ldots, u_m \in V_T^*$ und $X_1, \ldots, X_m \in V_N$ und Linksableitungen $X_i \overset{*}{\underset{lm}{\Longrightarrow}} w_j, j = 1, \ldots, k$, die alle eine Länge kleiner als n haben, mit $w = u_0w_1u_1 \ldots w_mu_m$. Für jedes $j \in \{1, \ldots, m\}$ gibt es deshalb nach Induktionsvoraussetzung ein i_j, so dass $(w_i|_k) \in I^{(i_j)}(X_i)$ gilt. Sei i' das Maximum dieser i_j. Für $i = i' + 1$ gilt dann:

$$\begin{aligned} I^{(i)}(X) &\supseteq \{u_0\} \odot_k I^{(i')}(X_1) \odot_k \{u_1\} \ldots \odot_k I^{(i')}(X_m) \odot_k \{u_m\} \\ &\supseteq \{u_0\} \odot_k \{w_1|_k\} \odot_k \{u_1\} \ldots \odot_k \{w_m|_k\} \odot_k \{u_m\} \\ &\supseteq \{w|_k\} \end{aligned}$$

Die Behauptung folgt. □

Kleinste Lösungen von Gleichungssystemen oder gelegentlich auch Ungleichungssystemen über einem vollständigen Verband zu berechnen, ist eine Aufgabe, die auch bei der Bestimmung statischer Programminvarianten auftritt. Solche Programminvarianten können ausgenutzt werden, um das Programm in ein möglicherweise effizienteres Programm umzuschreiben. Solche Analysen und Transformationen stellen wir im Band *Übersetzerbau: Analyse und Transformation* vor. Der global iterative Ansatz, den wir eben skizzierten, ist nicht unbedingt das beste Verfahren, um Gleichungssysteme zu lösen. Im Band *Übersetzerbau: Analyse und Transformation* beschreiben wir deshalb effizientere Verfahren.

Zur Berechnung der follow_k-Mengen für eine erweiterte kontextfreie Grammatik G beginnen wir wieder mit einer geeigneten rekursiven Eigenschaft. Für ein Wort $w \in V_T^k \cup V_T^{\leq k-1}\{\#\}$ gilt $w \in \mathsf{follow}_k(X)$, falls

(1) $X = S'$ das Startsymbol der erweiterten Grammatik ist und $w = \#$ ist, oder
(2) es eine Produktion $Y \to \alpha X \beta$ in G gibt, so dass $w \in \mathsf{first}_k(\beta) \odot_k \mathsf{follow}_k(Y)$ ist.

Die Mengen $\mathsf{follow}_k(X)$ erfüllen also das folgende Gleichungssystem:

$$\begin{aligned} &\mathsf{follow}_k(S') = \{\#\} \\ &\mathsf{follow}_k(X) = \bigcup\{\mathsf{first}_k(\beta) \odot_k \mathsf{follow}_k(Y) \mid Y \to \alpha X \beta \in P\}, \qquad S' \neq X \in V_N \end{aligned} \tag{fo}$$

Beispiel 3.2.15 Betrachten wir erneut die kontextfreie Grammatik G_2 aus Beispiel 3.2.13. Für die Bestimmung der follow_1-Mengen für die Grammatik G_2 ergibt sich das Gleichungssystem:

$$\begin{aligned} \mathsf{follow}_1(S) &= \{\#\} \\ \mathsf{follow}_1(E) &= \mathsf{follow}_1(S) \cup \mathsf{follow}_1(E') \cup \{)\} \odot_1 \mathsf{follow}_1(F) \\ \mathsf{follow}_1(E') &= \mathsf{follow}_1(E) \\ \mathsf{follow}_1(T) &= \{\varepsilon, +\} \odot_1 \mathsf{follow}_1(E) \cup \mathsf{follow}_1(T') \\ \mathsf{follow}_1(T') &= \mathsf{follow}_1(T) \\ \mathsf{follow}_1(F) &= \{\varepsilon, *\} \odot_1 \mathsf{follow}_1(T) \end{aligned}$$

□

Das Gleichungssystem (fo) muss wieder über einem Teilmengenverband gelöst werden, wobei die rechten Seiten aus konstanten Mengen und Unbekannten mit Hilfe monotoner Operatoren aufgebaut sind. Deshalb besitzt auch (fo) eine kleinste Lösung, welche durch globale Iteration berechnet werden kann. Wir vergewissern uns, dass dieser Algorithmus tatsächlich das richtige berechnet.

Satz 3.2.3 (Korrektheit der follow_k-Mengen) Sei $G = (V_N, V_T, P, S')$ eine erweiterte kontextfreie Grammatik, $\mathbb{D}$ der vollständige Verband der Teilmengen von $V_T^k \cup V_T^{\leq k-1}\{\#\}$ und $I : V_N \to \mathbb{D}$ die kleinste Lösung des Gleichungssystems (fo). Dann gilt:

$$I(X) = \mathsf{follow}_k(X) \quad \text{für alle } X \in V_N$$

□

Der Beweis ist ähnlich dem Beweis von Satz 3.2.2 und wird dem Leser zur Übung empfohlen (Aufgabe 6).

Beispiel 3.2.16 Betrachten wir das Gleichungssystem aus Beispiel 3.2.15. Zur Berechnung der kleinsten Lösung beginnt die Iteration wieder mit dem Wert $\emptyset$ für jedes Nichtterminal. Die Wörter, die bei den folgenden Iterationen hinzukommen, zeigt die Tabelle:

	1	2	3	4	5	6	7
S	$\#$						
E		$\#$			$)$		
E'			$\#$			$)$	
T			$+, \#,)$				
T'				$+, \#,)$			
F				$*, +, \#,)$			

Insgesamt erhalten wir die folgenden Mengen:

$$\begin{aligned} \mathsf{follow}_1(S) &= \{\#\} & \mathsf{follow}_1(T) &= \{+, \#,)\} \\ \mathsf{follow}_1(E) &= \{\#,)\} & \mathsf{follow}_1(T') &= \{+, \#,)\} \\ \mathsf{follow}_1(E') &= \{\#,)\} & \mathsf{follow}_1(F) &= \{*, +, \#,)\} \end{aligned}$$

□

3.2.6 Der Spezialfall first_1 und follow_1

Die Iteration, mit der wir die kleinsten Lösungen der Gleichungssysteme für die first_1- bzw. follow_1-Mengen in unseren Beispielen berechnet haben, ist nicht sehr effizient. Aber auch für effizientere Lösungsmethoden wird die Berechnung von first_k- bzw. follow_1-Mengen für größere Werte von k schnell sehr mühsam. Deshalb wird bei den Verfahren zur syntaktischen Analyse meist nur Vorausschau der Länge $k = 1$ eingesetzt. In diesem Fall kann die Berechnung der first- und follow-Mengen besonders effizient durchgeführt werden. Das folgende Lemma ist die Basis für unser weiteres Vorgehen:

Lemma 3.3. Seien $L_1, L_2 \subseteq V_T^{\leq 1}$ nichtleere Sprachen. Dann gilt:

$$L_1 \odot_1 L_2 = \begin{cases} L_1 & \text{falls } L_2 \neq \emptyset \text{ und } \varepsilon \notin L_1 \\ (L_1 \backslash \{\varepsilon\}) \cup L_2 & \text{falls } L_2 \neq \emptyset \text{ und } \varepsilon \in L_1 \end{cases}$$

Nach Voraussetzung sind die betrachteten Grammatiken stets reduziert. Deshalb enthalten sie weder unproduktive noch unerreichbare Nichtterminale. Es gilt für alle $X \in V_N$, dass sowohl $\mathsf{first}_1(X)$ wie $\mathsf{follow}_1(X)$ nichtleer sind. Zusammen mit Lemma 3.3 erlaubt uns das, die Transferfunktionen für first_1

und follow_1 so zu vereinfachen, dass die 1-Konkatenation (im Wesentlichen) durch *Vereinigungen* ersetzt wird. Um die Fallunterscheidung, ob ε in first_1-Mengen enthalten ist oder nicht, zu eliminieren, gehen wir in zwei Schritten vor. Im ersten Schritt wird die Menge der Nichtterminale X mit $\varepsilon \in \mathsf{first}_1(X)$ bestimmt. Im zweiten Schritt werden anstelle der first_1-Menge zu jedem Nichtterminal X die *ε-freie* first_1-Menge

$$\begin{aligned}\mathsf{eff}(X) &= \mathsf{first}_1(X)\backslash\{\varepsilon\}\\ &= \{(w|_k) \mid X \stackrel{*}{\underset{G}{\Longrightarrow}} w, w \neq \varepsilon\}\end{aligned}$$

bestimmt. Zur Implementierung des ersten Schritts bemerken wir, dass für jedes Nichtterminal X

$$\varepsilon \in \mathsf{first}_1(X) \quad \text{genau dann, wenn} \quad X \stackrel{*}{\Longrightarrow} \varepsilon$$

In einer Ableitung des Worts ε kann jedoch keine Produktion eingesetzt werden, die ein Terminalsymbol $a \in V_T$ enthält. Sei also G_ε die Grammatik, die man aus G erhält, indem man alle diese Produktionen streicht. Dann gilt deshalb $X \stackrel{*}{\underset{G}{\Longrightarrow}} \varepsilon$ genau dann, wenn X bzgl. der Grammatik G_ε produktiv ist. Für dieses Problem kann unser effizienter Löser für Produktivität aus Abschnitt 3.2.2 eingesetzt werden.

Beispiel 3.2.17 Betrachten wir die Grammatik G_2 aus Beispiel 3.2.13. Die Menge der Produktionen, in denen kein Terminalsymbol vorkommt, ist dann gegeben durch:

$$\begin{array}{llll} 0: & S & \to & E \\ 1: & E & \to & TE' \\ 2: & E' & \to & \varepsilon \end{array} \qquad \begin{array}{llll} 4: & T & \to & FT' \\ 5: & T' & \to & \varepsilon \end{array}$$

Bezüglich dieser Menge von Produktionen sind nur die Nichtterminale E' und T' produktiv. Diese beiden Nichtterminale sind folglich die beiden einzigen Nichtterminale der Grammatik G_2, die ε-produktiv sind. □

Wenden wir uns nun dem zweiten Schritt, der Berechnung der ε-freien first_1-Mengen zu. Betrachten wir eine Produktion von der Form $X \to X_1 \ldots X_m$. Der zugehörige Beitrag für $\mathsf{eff}(X)$ lässt sich schreiben als:

$$\bigcup\{\mathsf{eff}(X_j) \mid X_1 \ldots X_{j-1} \stackrel{*}{\underset{G}{\Longrightarrow}} \varepsilon\}$$

Insgesamt erhalten wir deshalb das Gleichungssystem:

$$\mathsf{eff}(X) = \bigcup\{\mathsf{eff}(Y) \mid X \to \alpha Y \beta \in P, \alpha \stackrel{*}{\underset{G}{\Longrightarrow}} \varepsilon\}, \quad X \in V_N \tag{eff}$$

Beispiel 3.2.18 Schauen wir uns erneut die kontextfreie Grammatik G_2 aus Beispiel 3.2.13 an. Zur Berechnung der ε-freien first_1-Mengen ergibt sich das Gleichungsystem:

$$\begin{array}{ll} \mathsf{eff}(S) = \mathsf{eff}(E) & \mathsf{eff}(T) = \mathsf{eff}(F) \\ \mathsf{eff}(E) = \mathsf{eff}(T) & \mathsf{eff}(T') = \emptyset \cup \{*\} \\ \mathsf{eff}(E') = \emptyset \cup \{+\} & \mathsf{eff}(F) = \{\mathsf{Id}\} \cup \{(\} \end{array}$$

Alle Vorkommen von $\odot_1$-Operatoren sind verschwunden. Stattdessen tauchen nur noch konstante Mengen, Vereinigungen bzw. Variablen $\mathsf{eff}(X)$ auf der rechten Seite auf. Als kleinste Lösung erhalten wir:

$$\begin{array}{ll} \mathsf{eff}(S) = \{\mathsf{Id}, (\} & \mathsf{eff}(T) = \{\mathsf{Id}, (\} \\ \mathsf{eff}(E) = \{\mathsf{Id}, (\} & \mathsf{eff}(T') = \{*\} \\ \mathsf{eff}(E') = \{+\} & \mathsf{eff}(F) = \{\mathsf{Id}, (\} \end{array}$$

□

Bei der Berechnung der ε-freien first_1-Mengen leisten Nichtterminale, die rechts von Terminalen stehen, keinen Beitrag. Für die Korrektheit der Konstruktion ist es deshalb wichtig, dass alle Nichtterminale der Grammatik produktiv sind.

Mithilfe der ε-freien first_1-Mengen $\mathsf{eff}(X)$ lässt sich auch das Gleichungssystem zur Berechnung der follow_1-Mengen vereinfachen. Betrachten wir eine Produktion von der Form $Y \to \alpha X X_1 \ldots X_m$. Der zugehörige Beitrag des Vorkommens von X in der rechten Seite von Y zu der Menge $\mathsf{follow}_1(X)$ ist gegeben durch:

$$\bigcup\{\mathsf{eff}(X_j) \mid X_1 \ldots X_{j-1} \overset{*}{\underset{G}{\Longrightarrow}} \varepsilon\} \cup \{\mathsf{follow}_1(Y) \mid X_1 \ldots X_m \overset{*}{\underset{G}{\Longrightarrow}} \varepsilon\}$$

Falls alle Nichtterminale nicht nur produktiv, sondern auch erreichbar sind, vereinfacht sich das Gleichungsystem zur Berechnung der follow_1-Mengen damit zu:

$$\begin{aligned}
\mathsf{follow}_1(S') &= \{\#\} \\
\mathsf{follow}_1(X) &= \bigcup\{\mathsf{eff}(Y) \mid A \to \alpha X \beta Y \gamma \in P, \beta \overset{*}{\underset{G}{\Longrightarrow}} \varepsilon\} \\
&\quad \cup \bigcup\{\mathsf{follow}_1(A) \mid A \to \alpha X \beta, \beta \overset{*}{\underset{G}{\Longrightarrow}} \varepsilon\}, \quad X \in V_N \backslash \{S'\}
\end{aligned}$$

Beispiel 3.2.19 Das vereinfachte Gleichungssystem zur Berechnung der follow_1-Mengen der kontextfreien Grammatik G_2 aus Beispiel 3.2.13 ergibt sich zu:

$$\begin{aligned}
\mathsf{follow}_1(S) &= \{\#\} \\
\mathsf{follow}_1(E) &= \mathsf{follow}_1(S) \cup \mathsf{follow}_1(E') \cup \{)\} \\
\mathsf{follow}_1(E') &= \mathsf{follow}_1(E) \\
\mathsf{follow}_1(T) &= \{+\} \cup \mathsf{follow}_1(E) \cup \mathsf{follow}_1(T') \\
\mathsf{follow}_1(T') &= \mathsf{follow}_1(T) \\
\mathsf{follow}_1(F) &= \{*\} \cup \mathsf{follow}_1(T)
\end{aligned}$$

Wieder beobachten wir, dass alle Vorkommen des Operators $\oplus_1$ beseitigt wurden. Neben konstanten Mengen und Variablen $\mathsf{follow}_1(X)$ taucht auf den rechten Seiten der Gleichungen nur noch der Vereinigungsoperator auf. □

Im nächsten Abschnitt wird ein Verfahren vorgestellt, das beliebige Gleichungssysteme, welche die besonderen Eigenschaften der vereinfachten Gleichungssysteme für die Mengen $\mathsf{eff}(X)$ und $\mathsf{follow}_1(X)$ haben, besonders effizient löst. Nach der Beschreibung des Verfahrens wenden wir es auf die Berechnung der first_1- und follow_1-Mengen an.

3.2.7 Reine Vereinigungsprobleme

Nehmen wir an, wir haben ein Gleichungssystem

$$\mathbf{x}_i = e_i \quad , \qquad i = 1, \ldots, n$$

über einem beliebigen vollständigen Verband $\mathbb{D}$, wobei auf der rechten Seite Ausdrücke e_i stehen, die nur aus Konstanten aus $\mathbb{D}$, Variablen $\mathbf{x}_j$ und Anwendungen des Operators $\sqcup$ (kleinste obere Schranke des vollständigen Verbands $\mathbb{D}$) aufgebaut sind. Unsere Aufgabe besteht darin, die kleinste Lösung dieses Gleichungssystems zu bestimmen. Ein solches Problem nennen wir ein *reines Vereinigungsproblem*.

Die Berechnung der Menge der erreichbaren Nichtterminale einer kontextfreien Grammatik ist ein reines Vereinigungsproblem über dem Booleschen Verband $\mathbb{B} = \{\mathit{false}, \mathit{true}\}$. Auch die Berechnung der ε-freien first_1-Mengen und der follow_1-Mengen für eine reduzierte kontextfreie Grammatik erfüllen die Bedingungen eines reinen Vereinigungsproblems. In diesem Fall sind die vollständigen Verbände 2^{V_T} bzw. $2^{V_T \cup \{\#\}}$, geordnet durch die Teilmengenrelation.

Beispiel 3.2.20 Als laufendes Beispiel betrachten wir den Teilmengenverband $\mathbb{D} = 2^{\{a,b,c\}}$ zusammen mit dem Gleichungssystem

$$\begin{aligned}\mathbf{x}_0 &= \{a\}\\ \mathbf{x}_1 &= \{b\} \cup \mathbf{x}_0 \cup \mathbf{x}_3\\ \mathbf{x}_2 &= \{c\} \cup \mathbf{x}_1\\ \mathbf{x}_3 &= \{c\} \cup \mathbf{x}_2 \cup \mathbf{x}_3\end{aligned}$$

□

Zu einem reinen Vereinigungsproblem konstruieren wir den Variablenabhängigkeitsgraphen. Die Knoten dieses Graphen sind gegeben durch die Variablen $\mathbf{x}_i$ des Gleichungssystems. Eine Kante $(\mathbf{x}_i, \mathbf{x}_j)$ gibt es genau dann in dem Variablen-Abhängigkeitsgraphen, wenn die Variable $\mathbf{x}_i$ in der rechten Seite der Variablen $\mathbf{x}_j$ vorkommt. Den Variablenabhängigkeitsgraphen zu dem Gleichungssystem aus Beispiel 3.2.20 zeigt Abbildung 3.6.

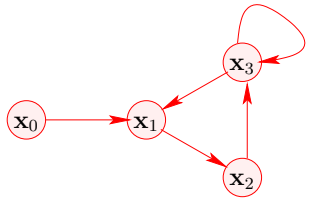


Abb. 3.6. Der Variablenabhängigkeitsgraph zu dem Gleichungssystem aus Beispiel 3.2.20.

Sei I die kleinste Lösung des Gleichungssystems. Als erstes bemerken wir, dass stets $I(\mathbf{x}_i) \sqsubseteq I(\mathbf{x}_j)$ gelten muss, wenn es einen Weg von $\mathbf{x}_i$ nach $\mathbf{x}_j$ im Variablenabhängigkeitsgraphen gibt. Folglich sind die Werte für die Variablen in jeder *starken Zusammenhangskomponente* des Variablenabhängigkeitsgraphen gleich.

Wir markieren jede Variable $\mathbf{x}_i$ mit der kleinsten oberen Schranke aller Konstanten, die auf rechten Seiten von Gleichungen für die Variable $\mathbf{x}_i$ vorkommen. Nennen wir diesen Wert $I_0(\mathbf{x}_i)$. Dann gilt für alle j, dass

$$I(\mathbf{x}_j) = \sqcup\{I_0(\mathbf{x}_i) \mid \mathbf{x}_j \text{ ist von } \mathbf{x}_i \text{ erreichbar}\}$$

Beispiel 3.2.21 (Fortsetzung von Beispiel 3.2.20)
Für das Gleichungssystem aus Beispiel 3.2.20 finden wir:

$$\begin{aligned}I_0(\mathbf{x}_0) &= \{a\}\\ I_0(\mathbf{x}_1) &= \{b\}\\ I_0(\mathbf{x}_2) &= \{c\}\\ I_0(\mathbf{x}_3) &= \{c\}\end{aligned}$$

Damit gilt:

$$\begin{aligned}I(\mathbf{x}_0) &= I_0(\mathbf{x}_0) &&= \{a\}\\ I_0(\mathbf{x}_1) &= I_0(\mathbf{x}_0) \cup I_0(\mathbf{x}_1) \cup I_0(\mathbf{x}_2) \cup I_0(\mathbf{x}_3) &&= \{a,b,c\}\\ I_0(\mathbf{x}_2) &= I_0(\mathbf{x}_0) \cup I_0(\mathbf{x}_1) \cup I_0(\mathbf{x}_2) \cup I_0(\mathbf{x}_3) &&= \{a,b,c\}\\ I_0(\mathbf{x}_3) &= I_0(\mathbf{x}_0) \cup I_0(\mathbf{x}_1) \cup I_0(\mathbf{x}_2) \cup I_0(\mathbf{x}_3) &&= \{a,b,c\}\end{aligned}$$

□

Diese Beobachtung legt das folgende Vorgehen nahe, um die kleinste Lösung I des Gleichungssystems zu berechnen. Zuerst werden die starken Zusammenhangskomponenten des Variablenabhängigkeitsgraphen berechnet. Dafür reichen linear viele Schritte aus. Dann wird über die Liste der starken Zusammenhangskomponenten iteriert.

Wir beginnen mit einer starken Zusammenhangskomponente Q, die keine eingehenden Kanten aus anderen starken Zusammenhangskomponenten besitzt. Die Werte aller Variablen $\mathbf{x}_j \in Q$ sind:

$$I(\mathbf{x}_j) = \bigsqcup\{I_0(\mathbf{x}_i) \mid \mathbf{x}_i \in Q\}$$

Die Werte $I(\mathbf{x}_j)$ können folglich berechnet werden durch die beiden Schleifen:

$$\begin{array}{l}\mathbb{D}\ t \leftarrow \bot;\\ \textbf{forall}\ (\mathbf{x}_i \in Q)\\ \qquad t \leftarrow t \sqcup I_0(\mathbf{x}_i);\\ \textbf{forall}\ (\mathbf{x}_i \in Q)\\ \qquad I(\mathbf{x}_i) \leftarrow t;\end{array}$$

Die Laufzeit jeder der beiden Schleifen ist proportional zu der Anzahl der Elemente in der starken Zusammenhangskomponente Q. Die Werte der Variablen aus Q werden entlang der ausgehenden Kanten propagiert. Sei E_Q die Menge der Kanten $(\mathbf{x}_i, \mathbf{x}_j)$ des Variablenabhängigkeitsgraphen mit $\mathbf{x}_i \in Q$ und $\mathbf{x}_j \notin Q$, d.h. der von Q ausgehenden Kanten. Dann setzen wir:

$$\begin{array}{l}\textbf{forall}\ ((\mathbf{x}_i, \mathbf{x}_j) \in E_Q)\\ \qquad I_0(\mathbf{x}_j) \leftarrow I_0(\mathbf{x}_j) \sqcup I(\mathbf{x}_i);\end{array}$$

Die Anzahl der Schritte für die Propagation ist proportional zu der Anzahl der Kanten in E_Q.

Nun wird die starke Zusammenhangskomponente Q zusammen mit der Menge E_Q der ausgehenden Kanten aus dem Graphen entfernt und mit der nächsten starken Zusammenhangskomponente ohne eingehende Kanten fortgefahren. Dies wird wiederholt, bis keine weiteren starken Zusammenhangskomponenten übrig sind. Insgesamt erhalten wir ein Verfahren, das linear viele Operationen $\sqcup$ in dem vollständigen Verband $\mathbb{D}$ ausführt.

Beispiel 3.2.22 (Fortsetzung von Beispiel 3.2.20) Der Abhängigkeitsgraph zu dem Gleichungssystem aus Beispiel 3.2.20 hat die starken Zusammenhangskomponenten

$$Q_0 = \{\mathbf{x}_0\} \quad \text{und} \quad Q_1 = \{\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3\}\,.$$

Für Q_0 erhalten wir den Wert $I_0(\mathbf{x}_0) = \{a\}$. Nach Beseitigung von Q_0 und der Kante $(\mathbf{x}_0, \mathbf{x}_1)$ erhalten wir die neue Zuordnung:

$$\begin{aligned} I_0(\mathbf{x}_1) &= \{a, b\}\\ I_0(\mathbf{x}_2) &= \{c\}\\ I_0(\mathbf{x}_3) &= \{c\}\end{aligned}$$

Die Werte aller Variablen in der starken Zusammenhangskomponente Q_1 ergeben sich als $I_0(\mathbf{x}_1) \cup I_0(\mathbf{x}_2) \cup I_0(\mathbf{x}_3) = \{a, b, c\}$. □

3.3 *Top-down*-Syntaxanalyse

3.3.1 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. In der Anfangssituation besteht das Fragment des Syntaxbaums aus der Wurzel, markiert mit dem Startsymbol der kontextfreien Grammatik; das gesamte w steht noch in der Eingabe. Jetzt wird eine Alternative für das Startsymbol zur Expansion ausgewählt. Die Symbole der rechten Seite dieser Alternative werden unter die Wurzel gehängt. Dadurch wird das obere Fragment des Syntaxbaums erweitert. Das nächste Nichtterminal, das betrachtet wird, ist das jeweils

am weitesten links stehende. Die Auswahl einer Alternative für dieses Nichtterminal und deren Anbau an das aktuelle Fragment des Syntaxbaums werden solange wiederholt, bis der Syntaxbaum vollständig ist. Durch den Anbau der Symbole der rechten Seite einer Produktion können Terminalsymbole im Blattwort des Baumfragments auftreten. Stehen links von ihnen keine Nichtterminale im Blattwort, so vergleicht der *top-down*-Analysator sie mit dem jeweils nächsten Symbol in der Eingabe. Stimmen die auftretenden Terminalsymbole mit der Eingab überein, das Symbol der Eingabe konsumiert. Andernfalls wird ein Fehler gemeldet.

Die *top-down*-Analyse führt also die folgenden zwei Arten von Schritten aus:

- Auswahl einer Produktion und Anbau der rechten Seite der Produktion an das aktuelle Baumfragment;
- Vergleich der Terminalsymbole links vom nächsten Nichtterminal mit der Eingabe.

Abbildungen 3.7, 3.8, 3.9 und 3.10 zeigen einige Syntaxbaumfragmente für den arithmetischen Ausdruck $\mathsf{Id} + \mathsf{Id} * \mathsf{Id}$ bzgl. der Grammatik G_2. Die Auswahl der Alternativen für die zu expandierenden Nichtterminale wurde jeweils so vorgenommen, dass die Analyse erfolgreich zu Ende geführt werden kann.

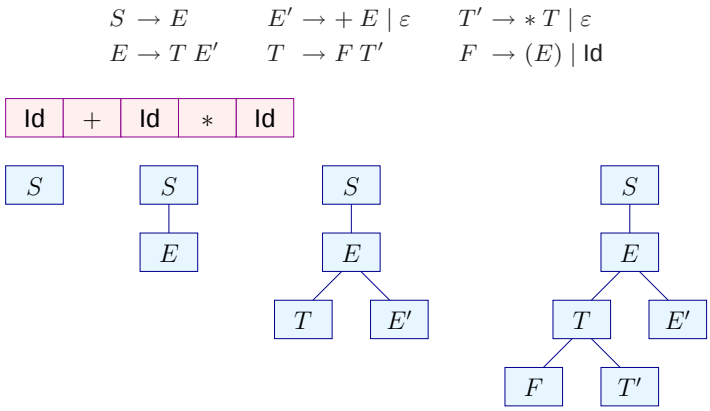


Abb. 3.7. Die ersten Syntaxbaumfragmente einer *top-down*-Analyse des Satzes $\mathsf{Id} + \mathsf{Id} * \mathsf{Id}$ der Grammatik G_2, die aufgebaut werden, ohne ein Symbol der Eingabe zu lesen.

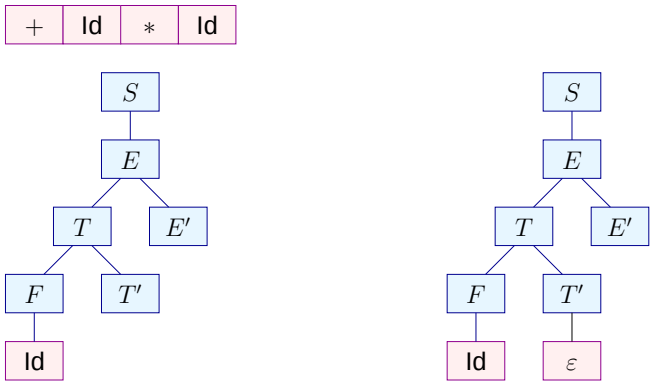


Abb. 3.8. Die Syntaxbaumfragmente nach Lesen des Symbols Id und bevor das Terminalsymbol $+$ an das Fragment angehängt wird.

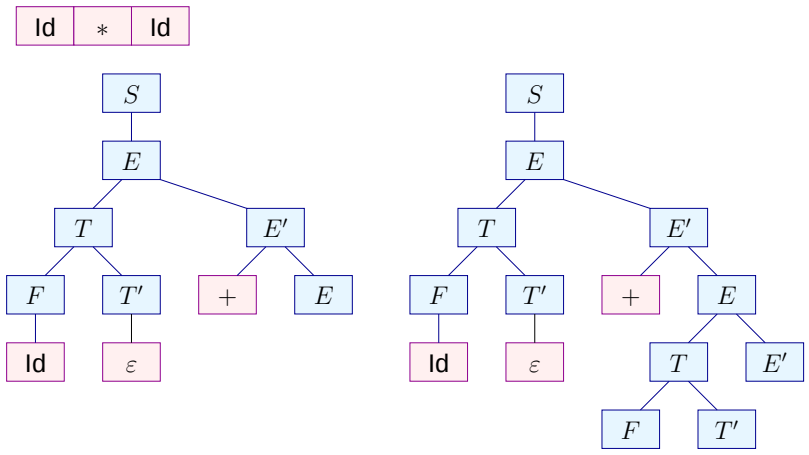


Abb. 3.9. Der erste und der letzte Syntaxbaum nach Lesen des Symbols $+$ und bevor das zweite Symbol Id im Syntaxbaum erscheint.

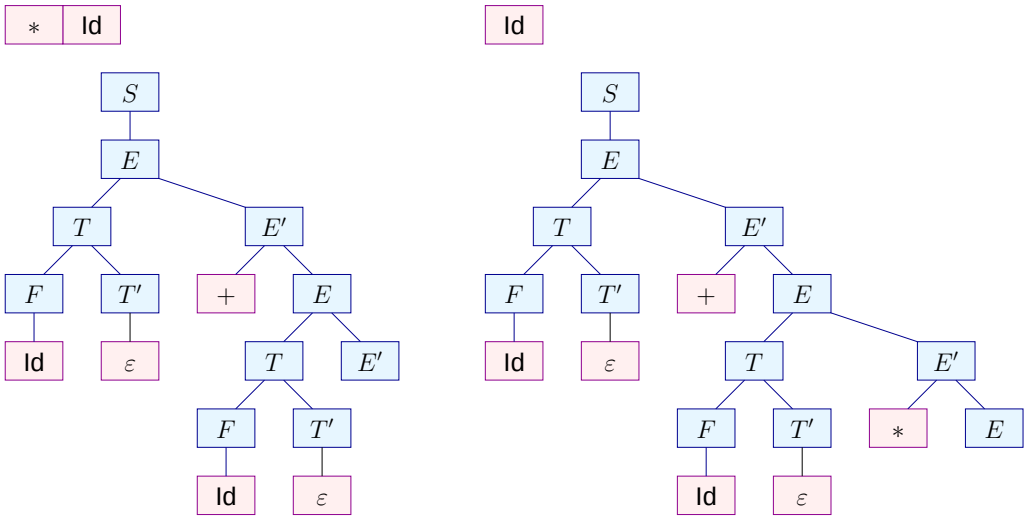


Abb. 3.10. Der Syntaxbaum nach der Reduktion für das zweite Vorkommen von T' und der Syntaxbaum nach Lesen des Symbols $*$, jeweils mit der verbleibenden Eingabe.

3.3.2 $LL(k)$: 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 der Item-Kellerautomat K_G diese Auswahl nichtdeterministisch trifft. Der ganze Nichtdeterminismus 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 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 dem Item-Kellerautomaten K_G einen deterministischen Automaten abzuleiten, gestatten wir eine *begrenzte* Vorausschau auf die restliche Eingabe. Wir geben eine natürliche Zahl $k \geq 1$ vor und lassen den Item-Kellerautomaten bei jedem (E)-Übergang als Entscheidungshilfe die k ersten Symbole der restlichen Eingabe zu Rate ziehen. Ist sichergestellt, dass diese Vorausschau der Länge k immer

ausreicht, um die richtige Alternative auszuwählen, nennen wir die Ausgangsgrammatik G eine $LL(k)$-Grammatik.

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

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

Wegen der Invariante (I) aus Abschnitt 3.2.4 gilt $\mathsf{hist}(\rho)\beta \overset{*}{\Longrightarrow} u$.

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. Dann nennen wir die Folge

$$\mathsf{fut}(\rho) = \gamma_n \ldots \gamma_1$$

die *Zukunft* von ρ. Sei $\delta = \mathsf{fut}(\rho)$. Bisher haben wir die Linksableitung $S' \overset{*}{\underset{lm}{\Longrightarrow}} uY\gamma\delta$ gefunden. Wenn sich diese Ableitung zur Ableitung des Terminalwortes uv fortsetzen lässt, d.h. $S' \overset{*}{\underset{lm}{\Longrightarrow}} uY\gamma\delta \overset{*}{\underset{lm}{\Longrightarrow}} uv$, dann hängt in einer $LL(k)$-Grammatik die Alternative, die für Y ausgewählt werden muss, nur von u, Y und $v|_k$ ab.

Sei $k \geq 1$ eine natürliche Zahl. Die reduzierte kontextfreie Grammatik G ist eine $LL(k)$-*Grammatik*, wenn für je zwei Linksableitungen:

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

und $x|_k = y|_k$ folgt, dass auch $\beta = \gamma$ gilt.

Bei einer $LL(k)$-Grammatik kann damit die Auswahl der Alternative für das nächste Nichtterminal Y im Allgemeinen nicht nur von Y und den nächsten k Symbolen, sondern auch von dem bereits konsumierten Präfix u der Eingabe abhängen. Hängt die Auswahl dagegen nicht von dem bereits konsumierten Linkskontext u ab, nennen wir die Grammatik *stark-*$LL(k)$.

Beispiel 3.3.1 Sei G_1 die kontextfreie Grammatik mit den Produktionen:

$$\begin{array}{lcl}
\langle \mathit{stat} \rangle & \to & \mathsf{if}\ (\mathsf{Id})\ \langle \mathit{stat} \rangle\ \mathsf{else}\ \langle \mathit{stat} \rangle \ \mid \\
 & & \mathsf{while}\ (\mathsf{Id})\ \langle \mathit{stat} \rangle \ \mid \\
 & & \{\ \langle \mathit{stats} \rangle\ \} \ \mid \\
 & & \mathsf{Id}\ '='\ \mathsf{Id};\\
\langle \mathit{stats} \rangle & \to & \langle \mathit{stat} \rangle\ \langle \mathit{stats} \rangle \ \mid \\
 & & \varepsilon
\end{array}$$

Die Grammatik G_1 ist eine $LL(1)$-Grammatik. Tritt $\langle \mathit{stat} \rangle$ als linkestes Nichtterminal in einer Satzform auf, dann bestimmt das nächste Eingabesymbol, welche Alternative angewendet werden muss. Genauer bedeutet das für zwei Ableitungen der Form:

$$\begin{array}{l}
\langle \mathit{stat} \rangle \overset{*}{\underset{lm}{\Longrightarrow}} w\ \langle \mathit{stat} \rangle\ \alpha \underset{lm}{\Longrightarrow} w\,\beta\,\alpha \overset{*}{\underset{lm}{\Longrightarrow}} w\,x \\
\langle \mathit{stat} \rangle \overset{*}{\underset{lm}{\Longrightarrow}} w\ \langle \mathit{stat} \rangle\ \alpha \underset{lm}{\Longrightarrow} w\,\gamma\,\alpha \overset{*}{\underset{lm}{\Longrightarrow}} w\,y
\end{array}$$

dass aus $x|_1 = y|_1$ folgt, dass $\beta = \gamma$ ist. Ist z.B. $x|_1 = y|_1 = \mathsf{if}$, dann ist $\beta = \gamma = \mathsf{if}\ (\mathsf{Id})\ \langle \mathit{stat} \rangle\ \mathsf{else}\ \langle \mathit{stat} \rangle$. □

Beispiel 3.3.2 Nun fügen wir zu der Grammatik G_1 aus Beispiel 3.3.1 die folgenden Produktionen hinzu:

$$\begin{array}{lcll}
\langle \mathit{stat} \rangle & \to & \mathsf{Id}:\ \langle \mathit{stat} \rangle \ \mid & // \quad \text{markierte Anweisung} \\
 & & \mathsf{Id}\,(\mathsf{Id}); & // \quad \text{Prozeduraufruf}
\end{array}$$

Die Grammatik G_2, die wir so erhalten, ist keine $LL(1)$-Grammatik mehr. Denn es gilt:

$$\langle stat\rangle \overset{*}{\underset{lm}{\Longrightarrow}} w\,\langle stat\rangle\,\alpha \underset{lm}{\Longrightarrow} w\,\overbrace{\mathsf{Id}\,'{=}'\,\mathsf{Id};}^{\beta}\,\alpha \overset{*}{\underset{lm}{\Longrightarrow}} w\,x$$

$$\langle stat\rangle \overset{*}{\underset{lm}{\Longrightarrow}} w\,\langle stat\rangle\,\alpha \underset{lm}{\Longrightarrow} w\,\overbrace{\mathsf{Id}\,:\,\langle stat\rangle}^{\gamma}\,\alpha \overset{*}{\underset{lm}{\Longrightarrow}} w\,y$$

$$\langle stat\rangle \overset{*}{\underset{lm}{\Longrightarrow}} w\,\langle stat\rangle\,\alpha \underset{lm}{\Longrightarrow} w\,\overbrace{\mathsf{Id}(\mathsf{Id});}^{\delta}\,\alpha \overset{*}{\underset{lm}{\Longrightarrow}} w\,z$$

wobei $x|_1 = y|_1 = z|_1 =$ Id, aber β, γ, δ paarweise verschieden sind.

G_2 ist aber eine $LL(2)$-Grammatik. Für die drei eben angegebenen Linksableitungen sind

$$x|_2 = \mathsf{Id}\,'{=}' \qquad y|_2 = \mathsf{Id}\ : \qquad z|_2 = \mathsf{Id}\,($$

paarweise verschieden. Und dies sind tatsächlich die einzigen kritischen Fälle. □

Beispiel 3.3.3 G_3 enthalte die Produktionen

$$\begin{array}{lcl}
\langle stat\rangle & \rightarrow & \mathsf{if}\ (\langle var\rangle)\ \langle stat\rangle\ \mathsf{else}\ \langle stat\rangle \ \mid \\
 & & \mathsf{while}\ (\langle var\rangle)\ \langle stat\rangle \ \mid \\
 & & \{\ \langle stats\rangle\ \} \ \mid \\
 & & \langle var\rangle\,'{=}'\,\langle var\rangle; \ \mid \\
 & & \langle var\rangle; \\
\langle stats\rangle & \rightarrow & \langle stat\rangle\ \langle stats\rangle \ \mid \\
 & & \varepsilon \\
\langle var\rangle & \rightarrow & \mathsf{Id} \ \mid \\
 & & \mathsf{Id}() \ \mid \\
 & & \mathsf{Id}(\langle vars\rangle) \\
\langle vars\rangle & \rightarrow & \langle var\rangle,\ \langle vars\rangle \ \mid \\
 & & \langle var\rangle
\end{array}$$

Die Grammatik G_3 ist für kein $k \geq 1$ eine $LL(k)$-Grammatik. Nehmen wir für einen Widerspruch an, G_3 wäre eine $LL(k)$-Grammatik für ein $k > 0$.

Sei $\langle stat\rangle \Rightarrow \beta \overset{*}{\underset{lm}{\Longrightarrow}} x$ und $\langle stat\rangle \Rightarrow \gamma \overset{*}{\underset{lm}{\Longrightarrow}} y$ mit

$x = \mathsf{Id}\,(\underbrace{\mathsf{Id}, \mathsf{Id}, \ldots, \mathsf{Id}}_{k})\,'{=}'\,\mathsf{Id};$ und $y = \mathsf{Id}\,(\underbrace{\mathsf{Id}, \mathsf{Id}, \ldots, \mathsf{Id}}_{k});$

Dann gilt zwar $x|_k = y|_k$, aber

$$\beta = \langle var\rangle\,'{=}'\,\langle var\rangle \qquad \gamma\ =\ \langle var\rangle;$$

und damit $\beta \neq \gamma$. □

Zu der Sprache $L(G_3)$ der Grammatik G_3 kann allerdings eine $LL(2)$-Grammatik aus der Grammatik G_3 durch durch *Faktorisierung* konstruiert werden. Kritisch in G_3 sind die Produktionen für die Wertzuweisung und den Prozeduraufruf. Faktorisierung fasst gemeinsame Anfänge solcher Produktionen zusammen. Auf diesen gemeinsamen Präfix folgt ein neues Nichtterminalsymbol, aus dem die unterschiedlichen Fortsetzungen abgeleitet werden können. Die Produktionen:

$$\langle stat\rangle \rightarrow \langle var\rangle\,'{=}'\,\langle var\rangle; \ \mid\ \langle var\rangle;$$

werden ersetzt durch:

$$\begin{array}{lcl}
\langle stat\rangle & \rightarrow & \langle var\rangle\ Z \\
Z & \rightarrow & '{=}'\,\langle var\rangle; \ \mid ;
\end{array}$$

Jetzt kann ein $LL(1)$-Parser zwischen den kritischen Alternativen mit Hilfe der nächsten Zeichen Id bzw. ; entscheiden.

Beispiel 3.3.4 Sei $G_4 = (\{S, A, B\}, \{0, 1, a, b\}, P_4, S)$, wobei die Menge P_4 der Produktionen gegeben ist durch:

$$\begin{aligned} S &\rightarrow A \mid B \\ A &\rightarrow aAb \mid 0 \\ B &\rightarrow aBbb \mid 1 \end{aligned}$$

Dann ist

$$L(G_4) = \{a^n 0 b^n \mid n \geq 0\} \cup \{a^n 1 b^{2n} \mid n \geq 0\}$$

und G_4 ist keine $LL(k)$-Grammatik für irgendein $k \geq 1$. Um das einzusehen, betrachten wir die beiden Linksableitungen:

$$\begin{aligned} S &\underset{lm}{\Longrightarrow} A \overset{*}{\underset{lm}{\Longrightarrow}} a^k 0 b^k \\ S &\underset{lm}{\Longrightarrow} B \overset{*}{\underset{lm}{\Longrightarrow}} a^k 1 b^{2k} \end{aligned}$$

Da für jedes $k \geq 1$, $(a^k 0 b^k)|_k = (a^k 1 b^{2k})|_k$ gilt, aber die rechten Seiten A und B für S verschieden sind, kann G_4 für kein $k \geq 1$ eine $LL(k)$-Grammatik sein. In diesem Fall kann man sogar zeigen, dass es zu der Sprache $L(G_4)$ für kein $k \geq 1$ eine $LL(k)$-Grammatik gibt. □

Satz 3.3.1 Die reduzierte kontextfreie Grammatik $G = (V_N, V_T, P, S)$ ist genau dann eine $LL(k)$-Grammatik, wenn für jeweils zwei verschiedene Produktionen $A \rightarrow \beta$ und $A \rightarrow \gamma$ von G gilt:

$$\mathsf{first}_k(\beta\alpha) \cap \mathsf{first}_k(\gamma\alpha) = \emptyset \text{ für alle } \alpha \text{ mit } S \overset{*}{\underset{lm}{\Longrightarrow}} wA\alpha$$

Beweis. Zum Beweis der Richtung " $\Rightarrow$ " nehmen wir an, G sei eine $LL(k)$-Grammatik, es existiere aber ein $x \in \mathsf{first}_k(\beta\alpha) \cap \mathsf{first}_k(\gamma\alpha)$. Nach Definition von first_k und wegen der Reduziertheit von G gibt es dann Ableitungen:

$$\begin{aligned} S &\overset{*}{\underset{lm}{\Longrightarrow}} uA\alpha \underset{lm}{\Longrightarrow} u\beta\alpha \overset{*}{\underset{lm}{\Longrightarrow}} uxy \\ S &\overset{*}{\underset{lm}{\Longrightarrow}} uA\alpha \underset{lm}{\Longrightarrow} u\gamma\alpha \overset{*}{\underset{lm}{\Longrightarrow}} uxz, \end{aligned}$$

wobei in dem Fall, dass $|x| < k$ ist, $y = z = \varepsilon$ gelten muss. Aus $\beta \neq \gamma$ folgt, dass G keine $LL(k)$-Grammatik sein kann – im Widerspruch zu unserer Annahme.

Zum Beweis der anderem Richtung " $\Leftarrow$ " nehmen wir an, G sei keine $LL(k)$-Grammatik. Dann gibt es zwei Linksableitungen

$$\begin{aligned} S &\overset{*}{\underset{lm}{\Longrightarrow}} uA\alpha \underset{lm}{\Longrightarrow} u\beta\alpha \overset{*}{\underset{lm}{\Longrightarrow}} ux \\ S &\overset{*}{\underset{lm}{\Longrightarrow}} uA\alpha \underset{lm}{\Longrightarrow} u\gamma\alpha \overset{*}{\underset{lm}{\Longrightarrow}} uy \end{aligned}$$

mit $x|_k = y|_k$, wobei $A \rightarrow \beta,\ A \rightarrow \gamma$ verschiedene Produktionen sind. Dann ist aber das Wort $x|_k = y|_k$ in $\mathsf{first}_k(\beta\alpha) \cap \mathsf{first}_k(\gamma\alpha)$ enthalten – im Widerspruch zu der Aussage des Satzes. □

Satz 3.3.1 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 3.3.1 kann man gute Kriterien für die Zugehörigkeit zu gewissen Teilklassen der $LL(k)$-Grammatiken ableiten. Die ersten betreffen den Fall $k = 1$.

Die Menge $\mathsf{first}_1(\beta\alpha) \cap \mathsf{first}_1(\gamma\alpha)$ für alle Linkssatzformen $wA\alpha$ und je zwei verschiedene Alternativen $A \rightarrow \beta$ und $A \rightarrow \gamma$ lässt sich zu $\mathsf{first}_1(\beta) \cap \mathsf{first}_1(\gamma)$ vereinfachen, wenn weder β noch γ das leere Wort ε produzieren. Dies ist dann der Fall, wenn kein Nichtterminal von G ε-produktiv ist. Für die Praxis wäre es jedoch eine zu starke Einschränkung, ε-Produktionen zu verbieten. Betrachten wir deshalb den Fall, dass eine der beiden rechten Seiten β bzw. γ das leere Wort produzieren kann. Produzieren sowohl β als auch γ das leere Wort, kann G keine $LL(1)$-Grammatik sein. Nehmen wir deshalb an, dass $\beta \overset{*}{\Longrightarrow} \varepsilon$, dass sich aber aus γ nicht ε ableiten lässt. Dann aber gilt für alle Linkssatzformen $uA\alpha, u'A\alpha'$:

$$\begin{aligned}\mathsf{first}_1(\beta\alpha) \cap \mathsf{first}_1(\gamma\alpha') &= \mathsf{first}_1(\beta\alpha) \ \cap \ \mathsf{first}_1(\gamma) \odot_1 \mathsf{first}_1(\alpha') \\ &= \mathsf{first}_1(\beta\alpha) \ \cap \ \mathsf{first}_1(\gamma) \\ &= \mathsf{first}_1(\beta\alpha) \ \cap \ \mathsf{first}_1(\gamma\alpha) \\ &= \emptyset\end{aligned}$$

Daraus folgt aber, dass

$$\begin{aligned}&\mathsf{first}_1(\beta) \odot_1 \mathsf{follow}_1(A) \ \cap \ \mathsf{first}_1(\gamma) \odot_1 \mathsf{follow}_1(A) \\ &\quad = \ \bigcup\{\mathsf{first}_1(\beta\alpha) \mid S \underset{lm}{\overset{*}{\Longrightarrow}} uA\alpha\} \ \cap \ \bigcup\{\mathsf{first}_1(\gamma\alpha') \mid S \underset{lm}{\overset{*}{\Longrightarrow}} u'A\alpha'\} \\ &\quad = \ \emptyset\end{aligned}$$

Damit erhalten wir den folgenden Satz.

Satz 3.3.2 Eine reduzierte kontextfreie Grammatik G ist eine $LL(1)$-Grammatik genau dann, wenn für je zwei verschiedene Produktionen $A \to \beta$ und $A \to \gamma$ gilt:

$$\mathsf{first}_1(\beta) \odot_1 \mathsf{follow}_1(A) \ \cap \ \mathsf{first}_1(\gamma) \odot_1 \mathsf{follow}_1(A) = \emptyset \ .$$

□

Im Gegensatz zu den Charakterisierungen aus dem Satz 3.3.1 lässt sich die Charakterisierung aus Satz 3.3.2 leicht überprüfen. Wenn wir zusätzlich die Eigenschaften der 1-Konkatenation in Betracht ziehen, erhalten wir sogar eine noch etwas handlichere Formulierung.

Korollar 3.3.2.1 Eine reduzierte kontextfreie Grammatik G ist genau dann eine $LL(1)$-Grammatik, wenn für alle Alternativen $A \to \alpha_1 \mid \ldots \mid \alpha_n$ gilt:

1. $\mathsf{first}_1(\alpha_1), \ldots, \mathsf{first}_1(\alpha_n)$ sind paarweise disjunkt; insbesondere enthält höchstens eine dieser Mengen ε;
2. Gilt $\varepsilon \in \mathsf{first}_1(\alpha_i)$, dann folgt: $\mathsf{first}_1(\alpha_j) \cap \mathsf{follow}_1(A) = \emptyset$ für alle $1 \leq j \leq n, \ j \neq i$. □

Die Eigenschaft aus dem Satz 3.3.2 verallgemeinern wir auf beliebige Vorausschaulängen $k \geq 1$.

Eine reduzierte kontextfreie Grammatik $G = (V_N, V_T, P, S)$ heißt *starke* $LL(k)$-Grammatik, wenn für je zwei verschiedene Produktionen $A \to \beta$ und $A \to \gamma$ eines Nichtterminals A stets

$$\mathsf{first}_k(\beta) \odot_k \mathsf{follow}_k(A) \ \cap \ \mathsf{first}_k(\gamma) \odot_k \mathsf{follow}_k(A) = \emptyset$$

gilt.

Gemäß dieser Definition und Satz 3.3.2 ist jede $LL(1)$-Grammatik eine starke $LL(1)$-Grammatik. Für $k > 1$ ist eine $LL(k)$-Grammatik jedoch nicht automatisch bereits eine starke $LL(k)$-Grammatik. Der Grund dafür ist, dass die Menge $\mathsf{follow}_k(A)$ die Folgeworte aus *allen* Linkssatzformen mit A enthält: in der $LL(k)$-Bedingung treten jedoch nur Folgeworte zu *einer* Linkssatzform auf.

Beispiel 3.3.5 Sei G die kontextfreie Grammatik mit den Produktionen

$$S \to aAaa \mid bAba \qquad\qquad A \to b \mid \varepsilon$$

Wir überprüfen:

1. Fall: Die Ableitung fängt mit $S \Rightarrow aAaa$ an. Dann gilt: $\mathsf{first}_2(baa) \cap \mathsf{first}_2(aa) = \emptyset$.
2. Fall: Die Ableitung fängt mit $S \Rightarrow bAba$ an. Dann gilt: $\mathsf{first}_2(bba) \cap \mathsf{first}_2(ba) = \emptyset$.

Also ist G nach Satz 3.3.1 eine $LL(2)$-Grammatik. Die Grammatik G ist jedoch keine starke $LL(2)$-Grammatik, denn

$$\begin{aligned}&\mathsf{first}_2(b) \odot_2 \mathsf{follow}_2(A) \ \cap \ \mathsf{first}_2(\varepsilon) \odot_2 \mathsf{follow}_2(A) \\ &\quad = \ \{b\} \odot_2 \{aa, ba\} \ \cap \ \{\varepsilon\} \odot_2 \{aa, ba\} \\ &\quad = \ \{ba, bb\} \ \cap \ \{aa, ba\} \\ &\quad = \ \{ba\}\end{aligned}$$

In dem Beispiel ist $\mathsf{follow}_1(A)$ also zu undifferenziert, da es die terminalen Folgewörter zusammenfasst, die bei *verschiedenen* Satzformen möglich sind. □

3.3.3 Linksrekursion

Parser, die den Syntaxbaum für die Eingabe *topdown* konstruieren, können nicht mit *linksrekursiven* Nichtterminalen umgehen. Eine Nichtterminal A einer reduzierten kontextfreien Grammatik G heißt dabei linksrekursiv, wenn es eine Ableitung $A \overset{+}{\Longrightarrow} A\beta$ gibt.

Satz 3.3.3 Sei G eine reduzierte kontextfreie Grammatik. Falls ein Nichtterminal der Grammatik G linksrekursiv ist, ist G für kein $k \geq 1$ eine $LL(k)$-Grammatik.

Beweis. Sei X ein linksrekursives Nichtterminal der Grammatik G. Zur Vereinfachung nehmen wir an, dass G direkt eine Produktion $X \to X\beta$ besitzt. Da G reduziert ist, muss es eine weitere Produktion $X \to \alpha$ geben. Tritt in einer Linkssatzform X auf, d.h. gilt $S \underset{lm}{\overset{*}{\Longrightarrow}} uX\gamma$, kann beliebig oft die Alternative $X \to X\beta$ angewandt werden. Für jedes $n \geq 1$ gibt es damit eine Linksableitung:

$$S \underset{lm}{\overset{*}{\Longrightarrow}} wX\gamma \underset{lm}{\overset{n}{\Longrightarrow}} wX\beta^n\gamma \ .$$

Nehmen wir an, die Grammatik G wäre eine $LL(k)$-Grammatik. Dann ist nach Satz 3.3.1

$$\mathsf{first}_k(X\alpha^{n+1}\gamma) \cap \mathsf{first}_k(\alpha\beta^n\gamma) = \emptyset$$

Wegen $X \to \alpha$ ist

$$\mathsf{first}_k(\alpha\beta^{n+1}\gamma) \subseteq \mathsf{first}_k(X\beta^{n+1}\gamma)$$

Also ist auch

$$\mathsf{first}_k(\alpha\beta^{n+1}\gamma) \cap \mathsf{first}_k(\alpha\beta^n\gamma) = \emptyset$$

Falls $\beta \overset{*}{\Longrightarrow} \varepsilon$ gilt, erhalten wir sofort den Widerspruch. Andernfalls wählen wir $n \geq k$ und erhalten ebenfalls einen Widerspruch. Folglich kann G keine $LL(k)$-Grammatik sein. □

Wir schließen, dass kein Generator für $LL(k)$-Parser linksrekursive Grammatiken behandeln kann. Jede Grammatik kann jedoch in eine Grammatik transformiert werden, die nicht mehr linksrekursiv ist, aber die gleiche Sprache beschreibt. Nehmen wir zur Vereinfachung an, die Grammatik G habe keine ε-Produktionen (siehe Aufg. **??**) und außerdem keine rekursiven Kettenproduktionen, d.h. es gibt kein Nichtterminal A mit $A \overset{+}{\underset{G}{\Longrightarrow}} A$. Sei $G = (V_N, V_T, P, S)$. Dann konstruieren wir zu G eine kontextfreie Grammatik $G' = (V'_N, V_T, P', S)$ mit der gleichen Menge V_T an Terminalsymbolen und dem gleichen Startsymbol S, dessen Menge V'_N an Nichtterminalsymbolen gegeben ist durch:

$$V'_N = V_N \cup \{\langle A, B\rangle \mid A, B \in V_N\}$$

und P' aus den folgenden Produktionen besteht:

- Ist $B \to a\beta \in P$ für ein Terminalsymbol $a \in V_T$, dann ist $A \to a\beta\langle A, B\rangle \in P'$ für jedes $A \in V_N$;
- Ist $C \to B\beta \in P$, dann ist $\langle A, B\rangle \to \beta\langle A, C\rangle \in P'$;
- Schließlich ist $\langle A, A\rangle \to \varepsilon \in P'$ für alle $A \in V_N$.

Beispiel 3.3.6 Für die Grammatik G_0 mit den Produktionen:

$$\begin{aligned} E &\to E + T \mid T \\ T &\to T * F \mid F \\ F &\to (E) \mid \mathsf{Id} \end{aligned}$$

erhalten wir nach Beseitigung der nichtproduktiven Nichterminale:

$$
\begin{array}{ll}
E & \to\ (E)\,\langle E,F\rangle \mid \mathsf{Id}\,\langle E,F\rangle \\
\langle E,F\rangle & \to\ \langle E,T\rangle \\
\langle E,T\rangle & \to\ *\,F\,\langle E,T\rangle \mid \langle E,E\rangle \\
\langle E,E\rangle & \to\ +T\,\langle E,E\rangle \mid \varepsilon \\
T & \to\ (E)\,\langle T,F\rangle \mid \mathsf{Id}\,\langle E,F\rangle \\
\langle T,F\rangle \to \langle T,T\rangle & \\
\langle T,T\rangle \to\ *\,F\,\langle T,T\rangle \mid \varepsilon & \\
F & \to\ (E)\,\langle F,F\rangle \mid \mathsf{Id}\,\langle F,F\rangle \\
\langle F,F\rangle & \to\ \varepsilon
\end{array}
$$

Die Grammatik G_0 benötigt drei Nichtterminale und sechs Produktionen, die Grammatik G_1, benötigt dagegen neun Nichtterminale und fünfzehn Produktionen.

Den Syntaxbaum für $\mathsf{Id} + \mathsf{Id}$ gemäß G_0 zeigt Abbildung 3.11 (a), denjenigen gemäß G_1 dagegen Abbildung 3.11 (b). Dieser Ableitungsbaum hat eine deutlich andere Struktur. Intuitiv erzeugt die Grammatik direkt das erste mögliche Terminalsymbol, um dann rückwärts die Reste der rechten Seiten aufzusammeln, die rechts auf das jeweilige Nichtterminalsymbol links folgen. Das Nichtterminal $\langle A,B\rangle$ steht deshalb für die Aufgabe, von dem B aus zurück zu A zu gelangen. □

Wir überzeugen uns, dass die Grammatik G', die wir zu der Grammatik G construiert haben, die folgenden Eigenschaften besitzt:

- Die Grammatik G' enthält keine linksrekursiven Nichtterminale.
- Es gibt eine Linksableitung
$$A \overset{*}{\underset{G}{\Longrightarrow}} B\gamma \underset{G}{\Longrightarrow} a\beta\gamma$$
genau dann, wenn es eine Rechtsableitung
$$A \underset{G'}{\Longrightarrow} a\beta\,\langle A,B\rangle \overset{*}{\underset{G'}{\Longrightarrow}} a\beta\gamma\,\langle A,A\rangle$$
gibt, bei der nach dem ersten Schritt nur Nichtterminale der Form $\langle X,Y\rangle$ ersetzt werden.

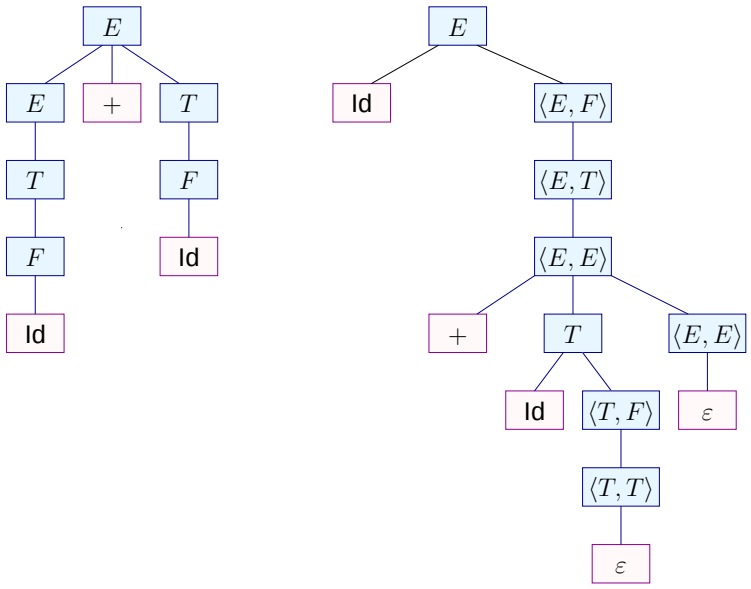
Aus der letzten Eigenschaft folgt insbesondere, dass die Grammatiken G und G' äquivalent sind, d.h. dass $L(G) = L(G')$ gilt.

In einigen Fällen ist die Grammatik nach Beseitigung der Linksrekursion eine $LL(k)$-Grammatik. Dies ist etwa für die Grammatik G_0 aus Beispiel 3.3.6 der Fall. Die Transformation zur Beseitigung der Linksrekursion hat jedoch auch Nachteile. Sei n die Anzahl der Nichtterminale. Dann kann sich sowohl die Anzahl der Nichtterminale wie der Produktionen um einen Faktor $n+1$ erhöhen. Bei großen Grammatiken wird es sich deshalb i.A. nicht lohnen, die Transformation *manuell* auszuführen. Ein Parsergenerator könnte dagegen diese Transformation vornehmen, um automatisch aus dem Ableitungsbaum für die transformierte Grammatik einen Ableitungsbaum für die ursprüngliche Grammatik zu rekonstruieren (siehe Aufgabe **??** des nächsten Kapitels). Dem Benutzer bliebe die Transformation der Grammatik, die der Parser aus rein pragmatischen Gründen vornimmt, damit verborgen.

Wie sehr sich der Syntaxbaum eines Ausdrucks gemäß der transformierten Grammatik von dem Synta gemäss der ursprünglichen Grammatik unterscheidet, illustriert Beispiel 3.3.6: der Operator sitzt etwas isoliert zwischen seinen weit entfernten Operanden. Eine Alternative zur nachträglichen Eliminierung der Linksrekursion sind Grammatiken mit *regulären* rechten Seiten, wie wir sie im nächsten Abschnitt betrachten werden.

3.3.4 Rechtsreguläre kontextfreie Grammatiken

Linksrekursion wird oft für Auflistungen benutzt, etwa von Parameterspezifikationen in einer Funktionsdeklaration oder aktuellen Parametern in einem Funktionsaufruf, von Indexausdrücken in einem mehrdimensionalen Feldzugriff oder von arithmetischen Teilausdrücken, die durch den gleichen (assoziativen) Operator verknüpft sind. Diese Fälle können aber auch elegant durch reguläre Ausdrücke reguläre Ausdrücke auf den rechten Seiten von Produktionen beschrieben werden.



(a) (b)

Abb. 3.11. Syntaxbaum für Id + Id bzgl. der Grammatik G_0 aus Beispiel 3.3.6 bzw. der zugehörigen Grammatik ohne Linksrekursion.

Eine *rechtsreguläre* kontextfreie Grammatik ist ein Tupel $G = (V_N, V_T, P, S)$, wobei V_N, V_T, S wie üblich die Menge der Nichtterminale, die Menge der Terminale und das Startsymbol sind, aber $P : V_N \to \mathit{RA}$ nun eine Abbildung der Nichtterminale in die Menge *RA* der regulären Ausdrücke über $V_N \cup V_T$ ist. Ein Paar (X, r) mit $P(X) = r$ schreiben wir auch $X \to r$.

Beachten Sie, dass die Menge der Produktionen auch bei einer normalen kontextfreien Grammatik als eine Abbildung $V_N \to \mathit{RA}$ aufgefasst werden kann, wenn wir die verschiedenen Alternativen für ein Nichtterminal mithilfe des Alternativoperators zu einem regulären Ausdruck zusammenfügen.

Beispiel 3.3.7 Eine rechtsreguläre kontextfreie Grammatik für arithmetische Ausdrücke ist gegeben durch:

$$G_e = (\{S, E, T, F\}, \{\mathsf{Id}, (,), +, -, *, /\}, P, S)$$

mit der Menge der Produktionen P:

$$\begin{aligned} S &\to E \\ E &\to T\{\{+ \mid -\}\, T\}^* \\ T &\to F\{\{* \mid /\}\, F\}^* \\ F &\to (E) \mid \mathsf{Id} \end{aligned}$$

Um die runden Klammern (und) unter den Terminalsymbolen von den Metasymbolen zur Notierung regulärer Ausdrücke zu unterscheiden, verwenden wir hier als Metazeichen *geschwungene* Klammern { und }. □

Für eine rechtsreguläre kontextfreie Grammatik definieren wir eine Relation $\underset{R,lm}{\Longrightarrow}$ auf Wörtern aus $(V_N \cup V_T)^*$ durch:

$$w\, X\, \beta \underset{R,lm}{\Longrightarrow} w\, \alpha\, \beta \qquad \text{sofern} \qquad \alpha \in L(P(X))$$

Dabei ist $L(r)$ die Sprache, die durch den regulären Ausdruck r beschrieben wird. Gilt $\beta_1 \underset{R,lm}{\Longrightarrow} \beta_2$ sagen wir, β_2 wird in einem Schritt aus β_1 abgeleitet. Die reflexive, transitive Hülle von $\underset{R,lm}{\Longrightarrow}$ nennen

wir reguläre Linksableitung und bezeichnen sie mit $\overset{*}{\underset{R,lm}{\Longrightarrow}}$. Die Relationen $\underset{R,lm}{\Longrightarrow}$ und $\overset{*}{\underset{R,lm}{\Longrightarrow}}$ setzen wir auf Paare von regulären Ausdrücken und Wörtern fort durch:

$$r \underset{R,lm}{\Longrightarrow} \beta_2 \qquad \text{sofern} \qquad \beta_1 \underset{R,lm}{\Longrightarrow} \beta_2 \quad \text{für ein } \beta_1 \in L(r)$$

bzw.

$$r \overset{*}{\underset{R,lm}{\Longrightarrow}} \beta_2 \qquad \text{sofern} \qquad \beta_1 \overset{*}{\underset{R,lm}{\Longrightarrow}} \beta_2 \quad \text{für ein } \beta_1 \in L(r)$$

Dann ist die von G definierte Sprache gegeben durch:

$$L(G) = \{w \in V_T^* \mid S \overset{*}{\underset{R,lm}{\Longrightarrow}} w\}$$

Beispiel 3.3.8 Eine reguläre Linksableitung zum Wort $\mathsf{Id} + \mathsf{Id} * \mathsf{Id}$ der Grammatik G_e aus Beispiel 3.3.7 ist etwa gegeben durch:

$$\begin{aligned} S &\underset{R,lm}{\Longrightarrow} E \underset{R,lm}{\Longrightarrow} T + T \underset{R,lm}{\Longrightarrow} F + T \underset{R,lm}{\Longrightarrow} \mathsf{Id} + T \\ &\underset{R,lm}{\Longrightarrow} \mathsf{Id} + F * F \underset{R,lm}{\Longrightarrow} \mathsf{Id} + \mathsf{Id} * F \underset{R,lm}{\Longrightarrow} \mathsf{Id} + \mathsf{Id} * \mathsf{Id} \end{aligned}$$

□

Anstelle einer eigenen Parserkonstruktion für Grammatiken mit regulären rechten Seiten stellen wir eine Konstruktion bereit, die solche Grammatiken durch normale kontextfreie Grammatiken simuliert. Dazu konstruieren wir zu jeder rechten Seite einen endlichen Automaten. Diesen endlichen Automaten erhalten wir aus dem endlichen Automaten, den wir in Kapitel 2.2 für einen regulären Ausdruck konstruiert haben, indem wir zusätzlich die ε-Übergänge beseitigen. Sei r ein regulärer Ausdruck über $V_N \cup V_T$ und $M = (Q, V_N \cup V_T, \Delta, q_0, \{q_f\})$ der endliche Automat zu r gemäß der Konstruktion aus dem Kapitel 2.2. Dann definieren wir einen ε-freien endlichen Automaten

$$M_r = (Q', V_N \cup V_T, \Delta', q_0', F')$$

wie folgt:

- $q_0' = q_0$.
- Q' und Δ' sind die kleinsten Mengen für die das folgende gilt. Der Anfangszustand q_0' ist in Q'. Ist weiterhin $q \in Q'$, $(q, \varepsilon, q_1) \in \Delta^*$ und $(q_1, X, p) \in \Delta$ für ein Nichtterminal- oder Terminalsymbol X, dann ist auch $p \in Q'$ und $(q, X, p) \in \Delta'$.
- $F' = \{q \in Q' \mid (q, \varepsilon, q_f) \in \Delta^*\}$.

Beispiel 3.3.9 Betrachten wir den regulären Ausdruck $r \equiv T\{\{+ \mid -\}T\}^*$. Die beiden endlichen Automaten zu r mit und ohne ε-Übergänge zeigt Abbildung 3.12. Alle ε-freien endlichen Automaten zu den regulären rechten Seiten der Grammatik G_e zeigt Abbildung 3.13. Beachten Sie, dass gemäß seiner Definition der endlichen Automat ohne ε-Übergänge im Allgemeinen einige Zustände weniger besitzt als der endliche Automat M mit ε-Übergängen. In unserem Fall sind die Zustände 2 und 5 weggefallen.

In unserem Beispiel bleiben dabei auch weniger Übergänge übrig. Die Beseitigung der ε-Übergänge kann jedoch die Anzahl der benötigten Übergänge auch *erhöhen*. □

Die Zustände des ε-freien endlichen Automaten M_r entsprechen den Endpunkten von Kanten des endlichen Automaten zu dem regulären Ausdruck r, an denen ein Symbol gelesen wird. Offenbar gilt $L(M_r) = L(M) = L(r)$.

Sei nun $G = (V_N, V_T, P, S)$ eine kontextfreie Grammatik mit rechtsregulären Produktionen. Nehmen wir an, die Mengen der Zustände der Automaten $M_{P(A)}, A \in N$, seien paarweise disjunkt. Den Anfangszustand des Automaten $M_{P(A)}$ bezeichnen wir dabei mit q_A. Seien Q, Δ, F die Vereinigungen der Mengen der Zustände, der Übergangsrelationen und der Endzustände der Automaten $M_{P(A)}$. Zu der Grammatik G konstruieren wir dann eine normale kontextfreie Grammatik $G' = (V_N', V_T, P', S)$.

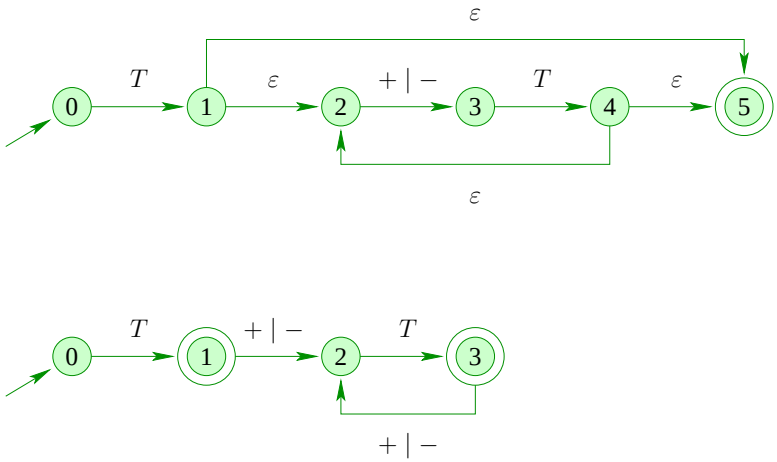


Abb. 3.12. Die beiden endlichen Automaten mit bzw. ohne ε-Übergänge für den regulären Ausdruck $T\{\{+ \mid -\}T\}^*$.

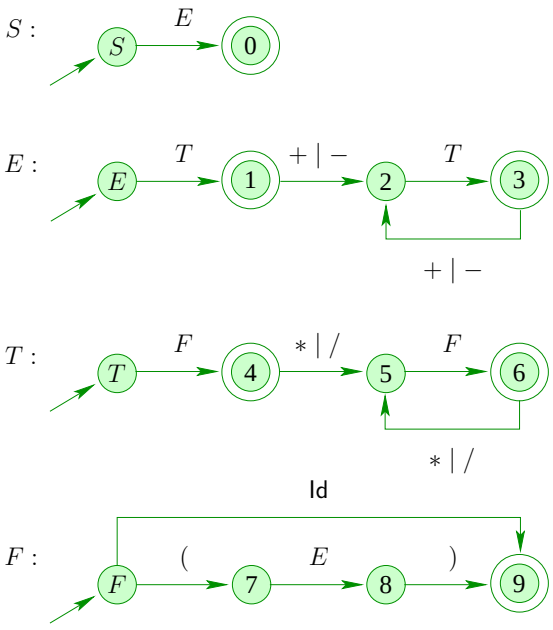


Abb. 3.13. Die ε-freien endlichen Automaten zu den rechten Seiten der Grammatik G_e mit fortlaufender Numerierung der Zustände, die keine Anfangszustände sind.

Die Menge der Nichtterminale der Grammatik G' ist gegeben durch $V'_N = V_N \cup Q$. Die Menge P' der Produktionen von G' ist gegeben durch:

$$\begin{array}{lll} A \to q_A & , & A \in V_N \\ p \to Xq & , & (p, X, q) \in \Delta \\ p \to \varepsilon & , & p \in F \end{array}$$

Man überzeugt sich, dass $L(G) = L(G')$ gilt. Aus einer Linksableitung von G' lässt sich leicht eine Linksableitung von G rekonstruieren. Auf die Grammatik G' können wir nun z.B. die Konstruktion eines $LL(1)$-Parsers aus dem nächsten Kapitel anwenden. Ist diese Konstruktion möglich, nennen wir die Grammatik G eine $RLL(1)$-Grammatik.

Beispiel 3.3.10 (Fortführung von Beispiel 3.3.7) Unsere Konstruktion liefert die folgende Grammatik:

$$
\begin{array}{ll}
S \to Z_S & E \to Z_E \\
Z_S \to E Z_0 & Z_E \to T Z_1 \\
S \to E Z_0 & Z_1 \to +Z_2 \mid -Z_2 \\
Z_0 \to \varepsilon & Z_2 \to T Z_3 \\
 & Z_3 \to +Z_2 \mid -Z_2 \mid \varepsilon \\
T \to Z_T & F \to Z_F \\
Z_T \to F Z_4 & Z_F \to \mathsf{Id}\ Z_9 \mid (\ Z_7 \\
Z_4 \to * Z_5 \mid / Z_5 & Z_7 \to E Z_8 \\
Z_5 \to F Z_6 & Z_8 \to)\ Z_9 \\
Z_6 \to * Z_5 \mid / Z_5 \mid \varepsilon & Z_9 \to \varepsilon
\end{array}
$$

Um unsere Namenskonventionen für kontextfreie Grammatiken einzuhalten, haben wir das Nichtterminal zu dem Zustand i mit Z_i bezeichnet. Die first_1- und follow_1-Mengen für diese Grammatik sind gegeben durch:

$$
\begin{array}{ll}
\mathsf{first}_1(S) = \{\mathsf{Id}, (\} & \mathsf{first}_1(E) = \{\mathsf{Id}, (\} \\
\mathsf{first}_1(Z_S) = \{\mathsf{Id}, (\} & \mathsf{first}_1(Z_E) = \{\mathsf{Id}, (\} \\
\mathsf{first}_1(Z_0) = \{\varepsilon\} & \mathsf{first}_1(Z_1) = \{+, -\} \\
 & \mathsf{first}_1(Z_2) = \{\mathsf{Id}, (\} \\
 & \mathsf{first}_1(Z_3) = \{\varepsilon\} \\
\mathsf{first}_1(T) = \{\mathsf{Id}, (\} & \mathsf{first}_1(F) = \{\mathsf{Id}, (\} \\
\mathsf{first}_1(Z_T) = \{\mathsf{Id}, (\} & \mathsf{first}_1(Z_F) = \{\mathsf{Id}, (\} \\
\mathsf{first}_1(Z_4) = \{*, /\} & \mathsf{first}_1(Z_7) = \{\mathsf{Id}, (\} \\
\mathsf{first}_1(Z_5) = \{\mathsf{Id}, (\} & \mathsf{first}_1(Z_8) = \{)\} \\
\mathsf{first}_1(Z_6) = \{\varepsilon\} & \mathsf{first}_1(Z_9) = \{\varepsilon\}
\end{array}
$$

sowie:

$$
\begin{array}{l}
\mathsf{follow}_1(S) = \mathsf{follow}_1(Z_S) = \mathsf{follow}_1(Z_0) = \{\#\} \\
\mathsf{follow}_1(E) = \mathsf{follow}_1(EZ_)) = \mathsf{follow}_1(Z_1) = \mathsf{follow}_1(Z_2) = \mathsf{follow}_1(Z_3) = \{\#,)\} \\
\mathsf{follow}_1(T) = \mathsf{follow}_1(TZ_)) = \mathsf{follow}_1(Z_4) = \mathsf{follow}_1(Z_5) = \mathsf{follow}_1(Z_6) = \{+, -, \#,)\} \\
\mathsf{follow}_1(F) = \mathsf{follow}_1(Z_F) = \mathsf{follow}_1(Z_7) = \mathsf{follow}_1(Z_8) = \mathsf{follow}_1(Z_9) = \{*, /, +, -, \#,)\}
\end{array}
$$

Die kontextfreie Grammatik, die wir zu G_e konstruiert haben, ist somit eine $LL(1)$-Grammatik. □

3.3.5 Starke $LL(k)$-Parser

Die Struktur eines Parsers für starke $LL(k)$-Grammatiken zeigt Abbildung 3.14. 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 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.

Bei einer Expansion verwendet der Parser die Parser-*Tabelle*, um die richtige Alternative für das Nichtterminal auszuwählen. Die Parser-Tabelle M ist ein zweidimensionales Feld, dessen Zeilen durch Nichtterminale und dessen Spalten durch Wörter der Länge (maximal) k indiziert werden. Sie repräsentiert eine Auswahlfunktion

$$V_N \times V_{T\#}^{\leq k} \to (V_T \cup V_N)^* \cup \{\mathsf{error}\}$$

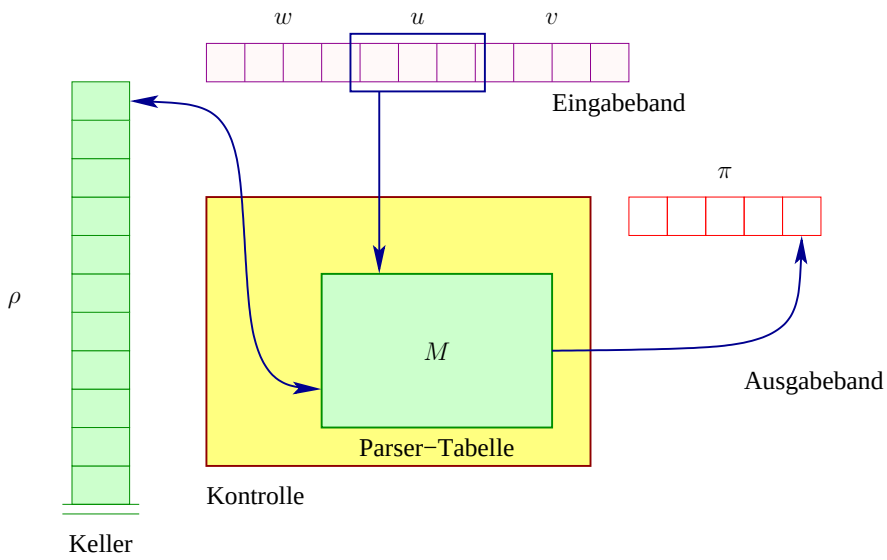


Abb. 3.14. Schematische Darstellung eines starken $LL(k)$-Parsers.

die jedem Nichtterminal die Alternative zuordnet, die bei der gegebenen Vorausschau ausgewählt werden soll – oder einen Fehler anzeigt, falls es keine passende Alternative gibt. Sei $[X \to \beta.Y\gamma]$ das oberste Item auf dem Keller und u der Präfix der Länge k der restlichen Eingabe. Gilt $M[Y,u] = (Y \to \alpha)$, dann wird $[Y \to .\alpha]$ neues oberstes Kellersymbol und die Produktion $Y \to \alpha$ auf das Ausgabeband geschrieben.

die Tabelleneinträge in M werden für ein Nichtterminal Y auf die folgende Weise berechnet. Seien $Y \to \alpha_1 \mid \ldots \mid \alpha_r$ die Alternativen für Y. Bei einer starken $LL(k)$-Grammatik sind die Mengen $\mathsf{first}_k(\alpha_i) \odot_k \mathsf{follow}_k(Y)$ disjunkt. Für jedes $u \in \mathsf{first}_k(\alpha_1) \odot_k \mathsf{follow}_k(Y) \cup \ldots \cup \mathsf{first}_k(\alpha_r) \odot_k \mathsf{follow}_k(Y)$ ist deshalb:

$$M[Y,u] \leftarrow \alpha_i \qquad \text{gdw.} \qquad u \in \mathsf{first}_k(\alpha_i) \odot_k \mathsf{follow}_k(Y)$$

Andernfalls wird $M[Y,u] \leftarrow \mathsf{error}$ gesetzt. Der Eintrag $M[Y,u] = \mathsf{error}$ bedeutet, dass das aktuelle Nichtterminal und das 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 3.3.6 beschrieben.

Für $k = 1$ ist die Konstruktion der Parser-Tabelle besonders einfach. Wegen Korollar 3.3.2.1 kommt sie ohne k-Konkatenation aus. Stattdessen reicht es, u auf Enthaltensein in einer der Mengen $\mathsf{first}_1(\alpha_i)$ bzw. gegebenenfalls noch $\mathsf{follow}_1(Y)$ zu testen.

Beispiel 3.3.11 Tabelle 3.3 ist die $LL(1)$-Parser-Tabelle für die Grammatik aus Beispiel 3.2.13. Tabelle 3.4 beschreibt den Lauf des zugehörigen Parsers für die Eingabe $\mathsf{Id} * \mathsf{Id}\#$. □

Anstelle mit Hilfe eines iterativen Treiberprogramms, das den Keller *explizit* verwaltet, lässt sich ein $LL(k)$-Parser auch durch ein *rekursives* Treiberprogramm implementieren. Einen solchen Parser nennen wir *Recursive-Descent–Parser*. Wir betrachten nur den Fall $k = 1$ und nehmen an, wir hätten bereits die first_1-und follow_1-Mengen berechnet und die Tabelle $M[X,a]$ aufgestellt. Das rekursive Treiberprogramm ist dann gegeben durch:

	(	)	+	∗	Id	#
S	E	error	error	error	E	error
E	$(E)\,\langle E,F\rangle$	error	error	error	Id $\langle E,F\rangle$	error
T	$(E)\,\langle T,F\rangle$	error	error	error	Id $\langle T,F\rangle$	error
F	$(E)\,\langle F,F\rangle$	error	error	error	Id $\langle F,F\rangle$	error
$\langle E,F\rangle$	error	$\langle E,T\rangle$	$\langle E,T\rangle$	$\langle E,T\rangle$	error	$\langle E,T\rangle$
$\langle E,T\rangle$	error	$\langle E,E\rangle$	$\langle E,E\rangle$	$*\,F\,\langle E,T\rangle$	error	$\langle E,E\rangle$
$\langle E,E\rangle$	error	ε	$+\,T\,\langle E,E\rangle$	error	error	ε
$\langle T,F\rangle$	error	$\langle T,T\rangle$	$\langle T,T\rangle$	$\langle T,T\rangle$	error	$\langle T,T\rangle$
$\langle T,T\rangle$	error	ε	ε	$*\,F\,\langle T,T\rangle$	error	ε
$\langle F,F\rangle$	error	ε	ε	ε	error	ε

Tabelle 3.3. $LL(1)$-Parsertabelle für die Grammatik aus Beispiel 3.2.13.

Kellerinhalt	Eingabe
$[S \to .E]$	Id $*$ Id#
$[S \to .E][E \to .\mathsf{Id}\,\langle E,F\rangle]$	Id $*$ Id#
$[S \to .E][E \to \mathsf{Id}\,.\,\langle E,F\rangle]$	$*$Id#
$[S \to .E][E \to \mathsf{Id}\,.\,\langle E,F\rangle][\langle E,F\rangle \to .\,\langle E,T\rangle]$	$*$Id#
$[S \to .E][E \to \mathsf{Id}\,.\,\langle E,F\rangle][\langle E,F\rangle \to .\,\langle E,T\rangle][\langle E,T\rangle \to .*\,F\,\langle E,T\rangle]$	$*$Id#
$[S \to .E][E \to \mathsf{Id}\,.\,\langle E,F\rangle][\langle E,F\rangle \to .\,\langle E,T\rangle][\langle E,T\rangle \to *\,.F\,\langle E,T\rangle]$	Id#
$[S \to .E][E \to \mathsf{Id}\,.\,\langle E,F\rangle][\langle E,F\rangle \to .\,\langle E,T\rangle][\langle E,T\rangle \to *\,.F\,\langle E,T\rangle][F \to .\mathsf{Id}\,\langle F,F\rangle]$	Id#
$[S \to .E][E \to \mathsf{Id}\,.\,\langle E,F\rangle][\langle E,F\rangle \to .\,\langle E,T\rangle][\langle E,T\rangle \to *\,.F\,\langle E,T\rangle][F \to \mathsf{Id}.\,\langle F,F\rangle]$	#
$[S \to .E][E \to \mathsf{Id}\,.\,\langle E,F\rangle][\langle E,F\rangle \to .\,\langle E,T\rangle][\langle E,T\rangle \to *\,.F\,\langle E,T\rangle][F \to \mathsf{Id}.\,\langle F,F\rangle][\langle F,F\rangle \to .]$	#
$[S \to .E][E \to \mathsf{Id}\,.\,\langle E,F\rangle][\langle E,F\rangle \to .\,\langle E,T\rangle][\langle E,T\rangle \to *\,.F\,\langle E,T\rangle][F \to \mathsf{Id}\,\langle F,F\rangle\,.]$	#
$[S \to .E][E \to \mathsf{Id}\,.\,\langle E,F\rangle][\langle E,F\rangle \to .\,\langle E,T\rangle][\langle E,T\rangle \to *\,F.\,\langle E,T\rangle]$	#
$[S \to .E][E \to \mathsf{Id}\,.\,\langle E,F\rangle][\langle E,F\rangle \to .\,\langle E,T\rangle][\langle E,T\rangle \to *\,F.\,\langle E,T\rangle][\langle E,T\rangle \to .\,\langle E,E\rangle]$	#
$[S \to .E][E \to \mathsf{Id}\,.\,\langle E,F\rangle][\langle E,F\rangle \to .\,\langle E,T\rangle][\langle E,T\rangle \to *\,F.\,\langle E,T\rangle][\langle E,T\rangle \to .\,\langle E,E\rangle][\langle E,E\rangle \to .]$	#
$[S \to .E][E \to \mathsf{Id}\,.\,\langle E,F\rangle][\langle E,F\rangle \to .\,\langle E,T\rangle][\langle E,T\rangle \to *\,F.\,\langle E,T\rangle][\langle E,T\rangle \to \langle E,E\rangle\,.]$	#
$[S \to .E][E \to \mathsf{Id}\,.\,\langle E,F\rangle][\langle E,F\rangle \to .\,\langle E,T\rangle][\langle E,T\rangle \to *\,F\,\langle E,T\rangle.]$	#
$[S \to .E][E \to \mathsf{Id}\,.\,\langle E,F\rangle][\langle E,F\rangle \to \langle E,T\rangle\,.]$	#
$[S \to .E][E \to \mathsf{Id}\,\langle E,F\rangle\,.]$	#
$[S \to E.]$	#

Ausgabe:
$(S \to E)\ (E \to \mathsf{Id}\,\langle E,F\rangle)\ (\langle E,F\rangle \to \langle E,T\rangle)\ (\langle E,T\rangle \to *\,F\,\langle E,T\rangle)\ (F \to \mathsf{Id}\,\langle F,F\rangle)$
$(\langle F,F\rangle \to \varepsilon)\ (\langle E,T\rangle \to \langle E,E\rangle)\ (\langle E,E\rangle \to \varepsilon)$

Tabelle 3.4. Parserlauf für die Eingabe: $Id * Id\#$

```
enum result { error; accept; }
terminal nextsymbol;

enum result parse() {
        nextsymbol ← scan();
        enum result r ← process(S);
        if (r = error) return error;
        if (nextsymbol = #) return accept;
        else return error;
}
```

Die Funktion process() analysiert die einzelnen Nichtterminale:

```
enum result process(nonterminal X) {
    string⟨symbol⟩ α;
    symbol Y;
    if (M[X, nextsymbol]) = error) { output("..."); goto err; }
    α ← M[X, nextsymbol]; output(X → α);
    enum result r;
    while (α ≠ ε) {
        Y ← hd(α); α ← tl(α);
        if (is_terminal(Y))
            if (Y = nextsymbol) nextsymbol ← scan();
            else { output("..."); goto err; }
        else {
            r ← process(Y);
            if (r = error) goto err;
        }
    }
    return accept;
err : return error;
}
```

Die Hilfsfunktion scan() stellt das jeweils nächste Symbol aus der Eingabe zur Verfügung. Der Datentyp **enum** *result* fasst die möglichen Ergebnisse der Analyse zusammen. In einer praktischen Implementierung wird man im Erfolgsfall nicht nur die Meldung *accept* zurück liefern, sondern eine (gegebenenfalls bereits weiter verarbeitete) Repräsentation des Syntaxbaums. Entsprechend sollte auch im Fehlerfall dem Anwender detaillierte Informationen über die Stelle zurück geliefert werden, an der der Fehler auftrat. Gegebenenfalls sollte im Fehlerfall die Analyse auch nicht sofort beendet werden. Entsprechende Techniken werden im Abschnitt 3.3.6 behandelt.

Wie die iterative Implementierung des $LL(k)$-Parsers kann die rekursive Implementierung automatisch aus einer starken $LL(k)$-Grammatik und ihren first_k- und follow_k-Mengen erzeugt werden. Die rekursive Implementierung kann noch weiter spezialisiert werden, indem die Funktion

$$\textbf{enum}\ \textit{result}\ \mathsf{process}(\textit{nonterminal}\ X)$$

zu einem System

$$\textbf{enum}\ \textit{result}\ \mathsf{process}_X(), \quad X \in V_N$$

rekursiver Funktionen instantiiert wird. Die Funktion $\mathsf{process}_X()$ ist für die Analyse von Worten für das Nichtterminal X zuständig. Im Rumpf der Funktion $\mathsf{process}_X()$ wird das Nachschlagen in der Tabelle M durch eine Fallunterscheidung ersetzt und die Iteration über die einzelnen Symbole der ausgewählten Alternative α für X zu einer Folge von Anweisungen abgewickelt. Für die Generierung dieser Funktionen benötigen wir die folgenden Code-Generator-Funktionen:

gen_proc X :	generiert die spezialisierte Funktion $\mathsf{process}_X()$;
gen_alt α :	generiert Code zur Abarbeitung der ausgewählten Alternative;
gen_symb X :	generiert Code für ein Symbol in einer rechten Seite.

Nehmen wir an, die Alternativen für das Nichtterminal X seien $\alpha_1 \mid \ldots \mid \alpha_n$ und $a_i = \mathsf{first}_1(\alpha_i) \odot \mathsf{follow}_1(X)$ das Terminalsymbol, bei dessen Vorausschau die i-te Alternative ausgewählt wird. Dann ist:

```
gen_proc X =  enum result process_X() {
                  enum result r;
                  switch (nextsymbol) {
                  case a_1 :     output(X → α_1);
                                 gen_alt α_1
                  ...
                  case a_n :     output(X → α_n);
                                 gen_alt α_n
                  default :      output("...");
                                 goto err;
                  }
              err :  return error;
              }
```

Der Code für die einzelnen Alternativen wird dann auf die folgende Weise generiert:

```
gen_alt (X_1 ... X_k) =  gen_symb X_1
                         ...
                         gen_symb X_k
                         return accept;

gen_symb a       =  if (a = nextsymbol) nextsymbol ← scan();
                    else { output("..."); goto err; }
gen_symb A       =  r ← process_A();
                    if (r = error) goto err;
```

Unsere Konstruktionen von $LL(k)$-Parsern sind nur auf *starke* $LL(k)$-Grammatiken anwendbar. Diese Einschränkung ist jedoch nicht so gravierend, wie es den Anschein hat:

- Der Fall, der am häufigsten in der Praxis vorkommt ist $k = 1$. Jede $LL(1)$-Grammatik ist aber stets eine starke $LL(1)$-Grammatik.
- Wird dennoch eine Vorausschau $k > 1$ benötigt und ist die Grammatik $LL(k)$, aber nicht stark $LL(k)$, dann gibt es immerhin eine allgemeine Konstruktion, wie zu der Grammatik eine starke $LL(k)$-Grammatik konstruiert werden kann, die die gleiche Sprache beschreibt (siehe Aufgabe 7).

Wir verzichten deshalb darauf, das Parse-Verfahren für starke $LL(k)$-Grammatiken auf beliebige $LL(k)$-Grammatiken zu verallgemeinern.

3.3.6 Fehlerbehandlung in $LL(k)$-Parsern

$LL(k)$-Parser haben die Eigenschaft des *fortsetzungsfähigen Präfixes*: jeder von einem $LL(k)$-Parser bestätigte Anfang eines Eingabewortes hat mindestens eine Fortsetzung zu einem Satz der Sprache. Obwohl Parser i.A. nur Fehlersymptome und nicht die Fehler selbst finden, legt es diese Eigenschaft nahe, auf Korrekturen in dem bereits gelesenen Teil der Eingabe zu verzichten und stattdessen durch Veränderung oder teilweises Überlesen der restlichen Eingabe wieder eine Parserkonfiguration zu suchen, aus der eine Analyse der restlichen Eingabe möglich ist. Das Verfahren, das nun vorgestellt wird, bemüht sich, durch möglichst geschicktes Überlesen eines Anfangs der restlichen Eingabe wieder eine Kombination von Kellerinhalt und restlicher Eingabe herzustellen, in der die Analyse zuende geführt werden kann.

Eine naheliegende Vorgehensweise besteht darin, eine Endklammer oder ein Trennsymbol für das aktuelle Nichtterminal zu suchen und alle dazwischenliegenden Eingabesymbole zu überlesen. Wird

kein solches Symbol gefunden, aber stattdessen ein aussagekräftiges Endesymbol für ein anderes Nichtterminal, dann werden Einträge im Keller so lange gelöscht, bis das zu diesem Endesymbol gehörende Nichtterminal oben auf dem Keller erscheint. Das hieße in C oder ähnlichen Sprachen etwa:

- Bei der Analyse einer Zuweisung nach einem Semikolon zu suchen,
- bei der Analyse einer Deklaration ein Komma oder ein Semikolon,
- bei einer bedingten Anweisungen nach einem else,
- zu einer öffnenden Klammer { für einen Block eine schließende Klammer } zu suchen.

Ein solcher *Panik-Modus* hat jedoch mehrere gravierende Nachteile. Selbst wenn das gesuchte Symbol im Programm vorhanden ist, muss der Parser gegebenenfalls größere Folgen von Wörtern ohne Analyse überlesen, bis er das Symbol findet. Fehlt das Symbol ganz oder gehört es nicht zu der gegenwärtigen Inkarnation des aktuellen Nichtterminals, gerät der Parser meist aus dem Tritt.

Unser Fehlerbehandlungsverfahren geht deshalb differenzierter vor. Es verfügt über zwei Modi, den *Parser-Modus* und den *Fehler-Modus*. Im Parser-Modus für ein Nichtterminal X wird ein Wort für X analysiert, seit dem Beginn dieser Analyse ist noch keinen Fehler entdeckt worden bzw. alle entdeckten Fehler wurden (vermeintlich) vollständig behandelt. Der Parser-Modus für X wird verlassen, wenn ein Wort für X gefunden ist. Wurde der Parser-Modus in der Anfangskonfiguration gestartet, ist also X gleich dem Startsymbol, ist das Ende der syntaktischen Analyse erreicht – vorausgesetzt, die Eingabe ist vollständig abgearbeitet. Der Parser-Modus kann allerdings auch im Fehlermodus rekursiv gestartet worden sein; dann wird nach seiner Beendigung auch dorthin zurückgekehrt. In den Fehlermodus wird bei Auftreten eines Fehlers gewechselt. Dann werden die nächsten Eingabesymbole überlesen, bis zu der aktuellen Analysesituation passende Endesymbole gefunden wurden. Damit keine großen Eingabeteile unanalysiert überlesen werden, wird rekursiv in den Parser-Modus für ein Nichtterminal X gewechselt, wenn ein *charakteristisches Anfangssymbol* für X gefunden wurde. Wie zwischen Parser- und Fehlermodus gewechselt wird, zeigt Abbildung 3.15. Dort sind a_X, a_Y bzw. a_Z Anfangssymbole und b_X, b_Y bzw. b_Z Endesymbole für die Nichtterminale X, Y bzw. Z.

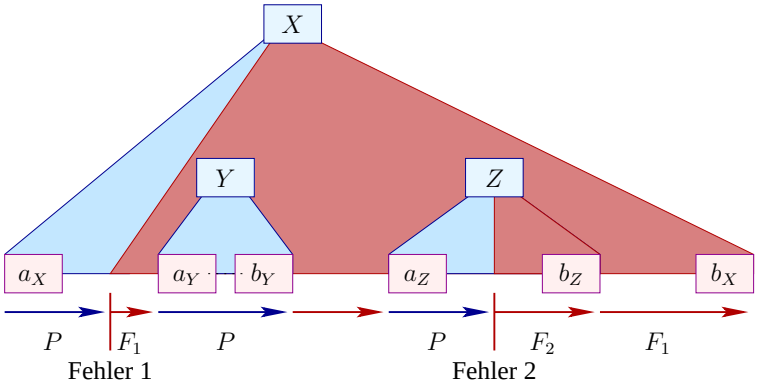


Abb. 3.15. Schematische Fehlerbehandlung; die Abkürzungen F_i und P stehen für Fehler- bzw. Parser-Modi.

Das Zurückschalten vom Fehler-Modus in den Parser-Modus bei Auffinden eines charakteristischen Endesymbols nennen wir *Fortsetzen bei einem Fortsetzungssymbol*, das rekursive Umschalten vom Fehler-Modus in den Parser-Modus beim Finden eines charakteristischen Anfangsymbols *Aufsetzen bei einem Aufsetzsymbol*.

Wir betrachten zwei Klassen von Aufsetzsymbolen und und zwei Klassen von Fortsetzungssymbolen und modifizieren das rekursive Treiberprogramm für $LL(1)$-Parser für diese vier Klassen so, dass die generierten $LL(1)$-Parser das Fortsetzen bzw. Aufsetzen im Fehlerfall beherrschen. Der Generator ist dabei auf Benutzerangaben angewiesen, welche festlegen, für welche Nichtterminale bzw. welche Vorkommen von Nichtterminalen diese Fehlerbehandlungen wirksam werden sollen.

Eine Fehlermeldung für ein Vorkommen eines Nichtterminals X soll nur einmal ausgegeben werden, d.h. Folgefehlermeldungen für den gleichen Fehler aus verschiedenen Parserfunktionsaufrufen

sollen vermieden werden. Den Parserfunktionsaufrufen, die zu dem aktuellen Aufruf process(X) geführt haben, wird deshalb mitgeteilt, ob in X ein Fehler entdeckt und gemeldet wurde, der sie auch betreffen kann. In diesem Fall wurde innerhalb von process(X) in den Fehler-Modus gewechselt, aber vor Beendigung von process(X) nicht mehr verlassen.

Fortsetzen bei last**-Symbolen**

Nehmen wir an, der Parser ist dabei, ein Wort für ein Nichtterminal X zu analysieren. Dabei stößt er auf einen Fehler, d.h. eine Situation, in der das nächste Eingabesymbol nicht zum aktuellen Zustand passt. Eine naheliegende Fehlerbehandlung besteht darin, den Fehler zu melden und die nächsten Eingabesymbole zu überlesen, bis ein Symbol gefunden wird, welches das letzte Symbol in einem Wort für X sein kann. In einem C-Block würde man nach einer schließenden Klammer } suchen, in einer Parameterliste nach der Klammer) usw. Dazu definieren wir in Analogie zu den ε-freien first$_1$-Mengen die ε-freien last$_1$-Mengen.

Für ein Wort $w = a_1 \dots a_n \in V_T^n, n \geq 0$, und $k \in \mathbb{N}$ sei $w|^k$ der *k-Suffix* des Worts w, d.h.

$$w|^k = \begin{cases} a_1 \dots a_n \text{ falls} & n < k \\ a_1 \dots a_k \text{ falls} & n \geq k \end{cases}$$

Für ein Nichtterminal X sei dann

$$\mathsf{last}_k(X) = \{w|^k \mid X \underset{R,lm}{\overset{*}{\Longrightarrow}} w\}$$

Die ε-freie last$_1$-Menge für X ist dann gegeben durch

$$\mathsf{efl}(X) = \mathsf{last}_1(X) \backslash \{\varepsilon\}$$

Zur Berechnung dieser Mengen können geeignete Gleichungssysteme aufgestellt werden, ganz analog zu den Gleichungssystemen für die first$_1$- bzw. die ε-freien first$_1$-Mengen. Wir überlassen dies als Übung unseren Lesern.

Die Fortsetzung bei last-Symbolen kann implementiert werden, indem die Funktion process() der Abschnitt zur Behandlung eines gefundenen Fehlers erweitert wird:

```
err : while (nextsymbol ∉ efl(X) ∪ {#}) nextsymbol ← scan();
      if (nextsymbol ∉ efl(X)) return error;
      nextsymbol ← scan();
      return accept;
```

Nach Finden eines Symbols aus efl(X) wird angenommen, dass ein erfolgreiches Fortsetzen im Parser-Modus möglich ist und keine weitere Fehlerbehandlung durch den Aufrufer von X erforderlich ist.

Ideal für diese Fehlerbehandlung sind Klammern, die eindeutig einem Nichtterminal zugeordnet werden können. Nicht alle Nichtterminale verfügen jedoch über charakteristische last-Symbole. Vorsicht ist auch geboten bei rekursiven Nichtterminalen. Hier kann es geschehen, dass eine Endklammer der falschen Inkarnation des Nichtterminals zugeordnet wird. Deshalb sollten nur solche rekursiven Nichtterminale für die Fortsetzung bei last-Symbolen ausgezeichnat werden, bei denen auch der Anfang ihrer Wörter erkannt werden kann. Der Benutzer des $LL(1)$-Parser-Generators bzw. der Schreiber der Eingabegrammatik sollten deshalb die Nichtterminale spezifizieren können, bei denen eine Fortsetzung über last-Symbole versucht werden soll. Im Folgenden bezeichnen wir die Menge der Nichtterminale, für die eine Fortsetzung bei last-Symbolen gewünscht wird, mit last(G). In die Fehlerbehandlung bauen wir dann eine entsprechende Fallunterscheidung ein.

Nichtlokale Fortsetzung bei last**-Symbolen**

Die bisher beschriebene Fehlerbehandlung setzt ein zu großes Vertrauen in das Vorhandensein des last-Symbols für das aktuelle Nichtterminal. Es können aber zwei Probleme auftreten. Erstens kann

der Syntaxfehler gerade darin bestehen, dass dieses last-Symbol fehlt. Zweitens könnte das aktuelle Nichtterminal gar kein charakteristisches last-Symbol besitzen, wie etwa das Nichtterminal für arithmetische Ausdrücke.

Deshalb werden Parserfunktionen zusätzlich in die Lage versetzt, auch nach last-Symbolen *umfassender* Nichtterminale zu suchen. Betrachten wir Vorkommen von Nichtterminalen, die in dem schon konstruierten Anfang der Linksableitung das aktuelle Nichtterminal produziert haben. Die zugehörigen Inkarnationen der Parserfunktion sind noch nicht beendet und werden in die Fehlerbehandlung mit einbezogen, wenn diese lokal in der aktuellen Funktion nicht möglich ist.

Dazu wird die Parserfunktion process() um einen weiteren Parameter erweitert, in dem die Menge der last-Symbole der umfassenden Nichtterminalvorkommen übergeben werden. Im Fehlerfall beendet die aktuelle Parserfunktion das Überlesen der Eingabesymbole nicht nur, wenn sie ein eigenes last-Symbol findet sondern auch, wenn sie eines der übergebenen Symbole identifiziert. Die entsprechend modifizierte rekursive Treiberfunktion process() sieht dann so aus:

```
enum result parse() {
        nextsymbol ← scan();
        enum result r ← process(S, {#});
        if (r = error) return error;
        if (nextsymbol = #) return accept;
        else return error;
}

enum result process(nonterminal X, set⟨terminal⟩ l) {
        set⟨terminal⟩ lasts;
        if (X ∈ last(G)) lasts ← l ∪ efl(X);
        else lasts ← l;

        ...

err :   if (X ∉ last(G)) return error;
        while (nextsymbol ∉ lasts) nextsymbol ← scan();
        if (nextsymbol ∉ efl(X)) return error;
        nextsymbol ← scan();
        return accept;
}
```

Den unveränderten Mittelteil der Funktion process() haben wir weggelassen. Ab der Marke *err* steht der Code für die Fehlerbehandlung. Falls das aktuelle Nichtterminal X nicht in last(G) enthalten ist, wird sofort ein Fehlerwert zurückgegeben. Ist das erste last-Symbol, das in der Eingabe gefunden wird, aus efl(X), wird dieses Zeichen konsumiert und dann *accept* zurückgeliefert. Andernfalls wird sofort ein Fehlerwert zurückgeliefert. Dann testen die Aufrufer in der Reihenfolge der Rückkehr, ob das gefundene Symbol für sie ein last-Symbol ist.

Fortsetzung bei follow-Symbolen

Eine Fehlerbehandlung nur über Fortsetzung bei last-Symbolen ist oft unzureichend. Ausdrücke haben i.A. keine charakteristischen Endesymbole. Sie kommen aber in verschiedenen Kontexten vor, in denen Symbole auf sie folgen müssen, die sehr aussagekräftig für die syntaktische Position sind. Dazu gehören Zuweisungen (gefolgt von einem Semikolon), Parameterpositionen (gefolgt von einem Komma oder einer schließenden Klammer), Bedingungen in *if*-Anweisungen (gegebenenfalls gefolgt von then). Deshalb nehmen wir eine Fortsetzung bei *Folgesymbolen* hinzu. Folgesymbole für ein Vorkommen des Nichtterminals X sind alle Symbole, die auf dieses Vorkommen in Satzformen folgen können.

Der Benutzer sollte wieder spezifizieren, für welche Nichtterminale er eine Fortsetzung bei Folgesymbolen wünscht. Die entsprechende Menge bezeichnen wir mit follow(G).

Das rekursive Treiberprogramm für $LL(1)$-Parser wird ein weiteres Mal abgeändert, um die Fortsetzung bei Folgesymbolen zu ermöglichen. Der Parserfunktion process() wird in einem weiteren Parameter die Menge der Folgesymbole für das jeweilige angewandte Vorkommen von X übergeben. Die entsprechend modifizierte Funktion process() ist gegeben durch:

```
enum result process(nonterminal X, set⟨terminal⟩ l, set⟨terminal⟩ f) {
    set⟨terminal⟩ lasts, follows;
    if (X ∈ last(G)) lasts ← l ∪ efl(X);
    else lasts ← l;

    set⟨symbol⟩ α;
    symbol Y;
    α ← M[X, nextsymbol]; output(X → α);
    enum result r;
    while (α ≠ ε) {
        Y ← hd(α); α ← tl(α);
        if (is_terminal(Y))
            if (Y = nextsymbol) nextsymbol ← scan();
            else { output("..."); goto err; }
        else {
            if (Y ∈ follow(G)) follows ← first_1(α) ⊙_1 follow_1(X);
            else follows ← ∅;
            r ← process(Y, lasts, follows);
            if (r = error) goto err;
        }
    }
    return accept;

err : while (nextsymbol ∉ lasts ∪ f) nextsymbol ← scan();
    if (X ∈ last(G) ∧ nextsymbol ∈ efl(X)) {
        nextsymbol ← scan();
        return accept;
    }
    if (X ∈ follow(G) ∧ nextsymbol ∈ f) {
        return accept; }
    return error;
}
```

Aufsetzen bei Anfangssymbolen

An der Fehlerbehandlung, die wir bisher entwickelt haben, stört noch, dass in einigen Situationen bei der Suche nach einem last- oder follow-Symbol zuviel überlesen wird. Ganze Anweisungen oder sogar Anweisungsfolgen können übersprungen werden, wenn das Ende einer bedingten Anweisung oder eines Schleifenrumpfes gesucht wird. Ein spezielles Problem tritt bei rekursiven Nichtterminalen auf, für die die Fortsetzung bei last-Symbolen spezifiziert ist. Bei geschachtelten Vorkommen wird die bisher entwickelte Strategie immer das erste Endesymbol nach der Fehlerstelle zum Fortsetzen benutzen, auch wenn es zu einer tieferen Schachtelung gehört. Dadurch ergeben sich oft Folgefehler.

Beispiel 3.3.12 Betrachten wir das folgende Programm:

$$\ldots \mathsf{while}\ \ldots \{a = a + 1 \ \underset{\uparrow}{}\ \mathsf{while}\ \ldots \{\ldots\} \ldots\} \ldots$$

Der Fehler besteht darin, dass das Semikolon nach der Wertzuweisung fehlt. Eine Fehlerbehandlung, die nur auf der Fortsetzung bei last-Symbolen beruht, würde das Klammersymbol } der inneren Schleife als Ende der äußeren Schleife auffassen und damit einen Folgefehler produzieren. Eine Fortsetzung bei Folgesymbolen würde den gleichen Fehler machen oder ein Semikolon aus dem Rumpf der inneren Schleife als Folgesymbol für die Wertzuweisung benutzen – was im Wesentlichen dieselben Folgen hätte. Diese falsche Fehlerbehandlung kann vermieden werden, wenn der Beginn der inneren Schleife erkannt wird und der Parser dort wieder in den Parser-Modus wechselt. □

Die Fehlerbehandlung wird darum so abgeändert, dass wieder in den Parser-Modus für ein Nichtterminal Y gewechselt wird, wenn ein für Y *charakteristisches Anfangssymbol* gefunden wird. Dabei werden nur solche Nichtterminale Y in Betracht gezogen, die von dem aktuellen Nichtterminal erreichbar sind. In Beispiel 3.3.12 wäre das Nichtterminal $\langle while\_stat \rangle$ aus dem angefangenen, aber noch nicht beendeten Nichtterminal ⟨*stat*⟩ ableitbar. Damit kann bei dem Anfangssymbol while des Nichtterminals ⟨*while_stat*⟩ aufgesetzt werden. Diese Wahl schließt eine Reihe von Folgefehlern aus.

Bisher sah es so aus, als ob im Fehler-Modus nur Symbole überlesen würden – was einem Löschen der Symbole aus der Eingabe entspricht. Das Aufsetzen, welches jetzt hinzugenommen wird, entspricht dagegen häufig dem *Einfügen* eines oder mehrerer Symbole, in Beispiel 3.3.12 etwa dem Einfügen eines Semikolons. Anfangssymbole, bei denen in einer Fehlersituation aufgesetzt werden soll, müssen die folgenden Anforderungen erfüllen:

- Sie sollten ausschließlich als erste Symbole von Wörtern für Nichtterminale auftreten.
- Ist das aktuelle Nichtterminal X, und sind $Y_1, \ldots, Y_n$ die aus X ableitbaren Nichtterminale, muss jedes Anfangssymbol von X eindeutig einem der Y_j $(1 \leq j \leq n)$ zugeordnet werden können.

Wir nehmen an, dass der Benutzer unseres Generators eine Teilmenge *BEG*(G) der Nichtterminale spezifizierte, bei denen mit Anfangssymbolen wieder aufgesetzt werden soll.

Zur Formalisierung der Mengen von Anfangssymbolen definieren wir für ein Nichtterminal X die Menge $er(X) \subseteq V_N$ als die Menge aller Nichtterminale, die in Syntaxbäumen für X vorkommen können. Sei *BEG*$(X) = er(X) \cap$ *BEG*(G). Dann definieren wir die Menge der *charakteristischen Anfangssymbole* für das Nichtterminal Y im Kontext des Nichtterminals X als die Menge

$$\mathsf{begsymbs}(X, Y) = \mathsf{eff}(Y) \setminus \bigcup \{\mathsf{eff}(Z) \mid Z \in \mathit{BEG}(X), Z \neq Y\}$$

Die Menge aller charakteristischen Anfangssymbole im Kontext X ist dann gegeben durch:

$$\mathsf{begsymbs}(X) = \bigcup \{\mathsf{begsymbs}(X, Y) \mid Y \in \mathit{BEG}(X)\}$$

Die charakteristischen Anfangssymbole von Y im Kontext X bilden eine Teilmenge der Menge $\mathsf{eff}(Y)$. Entfernt werden alle Symbole, die in den first_1-Mengen anderer, aus X ableitbarer Nichtterminale aus *BEG*(G) vorkommen. Spezifiziert der Benutzer unglücklicherweise etwa, dass die Nichtterminale $\langle stat \rangle$ und $\langle if\_stat \rangle$ beide in *BEG*(G) sein sollen, entfällt if als charakteristisches Anfangssymbol, da es in den first_1-Mengen beider Nichtterminale liegt.

Sei X das aktuelle Nichtterminal. Geht der Parser in den Fehler-Modus über, sucht er bei der Fehlerbehandlung nun nicht nur last- oder follow-Symbole. Vielmehr berücksichtigt er auch die Anfangssymbole aus $\mathsf{begsymbs}(X)$. Findet er ein Symbol aus $\mathsf{begsymbs}(X, Y)$, geht er rekursiv in den Parser-Modus für das Nichtterminal Y über. Nach Konstruktion sind die Mengen $\mathsf{begsymbs}(X, Y)$ und $\mathsf{begsymbs}(X, Z)$ für alle $Y \neq Z$ in *BEG*(X) disjunkt. Damit bestimmt jedes Symbol aus $\mathsf{begsymbs}(X)$ die Fortsetzung mit genau einem Nichtterminal. Sei

$$nonterminal\ \mathsf{recover}(nonterminal\ X, terminal\ a)$$

eine Implementierung dieser Funktion.

```
enum result process(nonterminal X, set⟨terminal⟩ l, set⟨terminal⟩ f, set⟨terminal⟩ b) {
    set⟨terminal⟩ lasts, follows, begins;
    begins ← b ∪ begsymbs(X);
    if (X ∈ last(G)) lasts ← l ∪ efl(X);
    else lasts ← l;

    string⟨symbol⟩ α;
    symbol Y;
    if (M[X, nextsymbol]) = error) { output("..."); goto err; }
    α ← M[X, nextsymbol]; output(X → α);
    enum result r;
    while (α ≠ ε) {
        Y ← hd (α); α ← tl (α);
        if (is_terminal(Y))
            if (Y = nextsymbol) nextsymbol ← scan();
            else { output("..."); goto err; }
        else {
            if (Y ∈ follow(G)) follows ← first_1(α) ⊙_1 follow_1(X);
            else follows ← ∅;
            r ← process(Y, lasts, follows, begins);
            if (r = error) goto err;
        }
    }
    return accept;

err : while (nextsymbol ∉ lasts ∪ f) {
        if (nextsymbol ∈ begins)
            if (nextsymbol ∈ begsymbs(X)) {
                Y ← recover(X, nextsymbol);
                process(Y, lasts, ∅, begins);
            } else return error;
        nextsymbol ← scan();
    }
    if (X ∈ last(G) ∧ nextsymbol ∈ efl(X)) {
        nextsymbol ← scan();
        return accept;
    }
    if (X ∈ follow(G) ∧ nextsymbol ∈ f) return accept;
    return error;
}
```

Weil der Kontext für das Nichtterminalsymbol Y, das die Funktion recover() zurück liefert, unbekannt ist, kann dem Aufruf von process() für Y nur die leere Menge von Folgesymbolen übergeben werden.

Aufsetzen bei Vorgängersymbolen

Im letzten Abschnitt wurde die Fehlerbehandlung so erweitert, dass sie im Fehler-Modus wieder aufsetzte, d.h. in den Parser-Modus für ein Nichtterminal Y umschaltete, wenn ein charakteristisches

Anfangssymbol für Y gefunden wurde. Es kommt jedoch häufig vor, dass Nichtterminale keine charakteristischen Anfangssymbole besitzen, aber dafür Symbole, die einem speziellen Vorkommen des Nichtterminals immer vorangehen. Solche Symbole nennen wir *Vorgängersymbole* für ein Nichtterminalvorkommen. In Analogie zu follow nennen wir sie precede-Symbole. Den zwei Vorkommen von Typbeschreibungen in PASCAL-Deklarationen etwa geht einmal ein Doppelpunkt und einmal ein Gleichheitszeichen voran. In einer bedingten Anweisung in C oder JAVA geht den Vorkommen des Nichtterminals $\langle stat \rangle$ einmal eine schließende Klammer und einmal ein else voraus.

Von dem Benutzer erwarten wir eine Spezifikation einer Menge von Nichtterminalen, bei denen nach Auffinden eines Vorgängersymbols wieder aufgesetzt werden kann. Die Menge der spezifizierten Nichtterminale nennen wir $PREC(G)$. Bei der Berechnung der Aufsetzsymbole, also der Anfangssymbole und der Vorgängersymbole müssen folgende Bedingungen erfüllt werden:

- Die Spezifikationen des Benutzers sollten dazu führen, dass kein Symbol gleichzeitig Anfangs- und Vorgängersymbol ist.
- Jedes Aufsetzsymbol sollte eindeutig ein Nichtterminal bestimmen, für welches rekursiv in den Parser-Modus gewechselt wird.

Wir bestimmen in mehreren Schritten die Mengen $\mathsf{begs}(X,Y)$ und $\mathsf{precs}(X,Y)$ der Anfangssymbole bzw. Vorgängersymbole von Y im Kontext X. Wir definieren der Reihe nach die Mengen:

$$
\begin{aligned}
B_1(X,Y) &= \begin{cases} \mathsf{eff}(Y)\ , & \text{falls } Y \in BEG(X) \\ \emptyset\ , & \text{sonst} \end{cases} \\
P_1(X,Y) &= \begin{cases} \bigcup\{\mathsf{last}_1(\alpha) \mid Z \in er(X) \cup \{X\}, Z \to \alpha Y \beta \in P\}\ , & \text{falls } Y \in PREC(G) \\ \emptyset\ , & \text{sonst} \end{cases} \\
B_2(X,Y) &= B_1(X,Y) \setminus \textstyle\bigcup_Y P_1(X,Y) \\
P_2(X,Y) &= P_1(X,Y) \setminus \textstyle\bigcup_Y B_1(X,Y) \\
\mathsf{begs}(X,Y) &= B_2(X,Y) \setminus \textstyle\bigcup_{Z \neq Y} B_2(X,Z) \\
\mathsf{precs}(X,Y) &= P_2(X,Y) \setminus \textstyle\bigcup_{Z \neq Y} P_2(X,Z) \\
\mathsf{begsymbs}(X) &= \textstyle\bigcup_Y (\mathsf{begs}(X,Y) \cup \mathsf{precs}(X,Y))
\end{aligned}
$$

Die Menge $P_1(X,Y)$ enthält alle Vorgängersymbole von Vorkommen von Y aus $PREC(G)$ für solche Nichtterminale, die von X aus erreichbar sind. Die Mengen B_2 erhalten wir, indem wir alle first-Symbole eleminieren, die gleichzeitig precede-Symbole sind. Die Mengen P_2 erhalten wir, indem wir alle precede-Symbole eliminieren, die gleichzeitig first-Symbole sind. Die Mengen begs erhalten wir, indem wir alle mehrfach vorkommenden Anfangssymbole streichen. Die Mengen precs erhalten wir, indem wir alle mehrfach vorkommenden Vorgängersymbole streichen. Damit erhalten wir eine neue Definition der Menge der Aufsetzsymbole $\mathsf{begsymbs}(X)$ im Kontext X. Zu dieser neuen Definition konstruieren wir entsprechend eine neue Funktion

$$nonterminal\ \mathsf{recover}(nonterminal\ Y, terminal\ a)$$

Das rekursive Treiberprogramm für die $LL(1)$-Parser, das jetzt bei Anfangs- und Vorgängersymbolen wieder aufsetzt, erhalten wir, indem der Fehlerfall um eine Behandlung von Vorgängersymbolen erweitert wird:

```
err :  while (nextsymbol ∉ lasts ∪ f) {
           if (nextsymbol ∈ begins)
               if (nextsymbol ∈ begsymbs(X)) {
                   Y ← recover(X, nextsymbol);
                   if (nextsymbol ∈ precs(X, Y)) nextsymbol ← scan();
                   process(Y, lasts, ∅, begins);
               } else return error;
           nextsymbol ← scan();
       }
       if (X ∈ last(G) ∧ nextsymbol ∈ efl(X)) {
           nextsymbol ← scan();
           return accept;
       }
       if (X ∈ follow(G) ∧ nextsymbol ∈ f) return accept;
       return error;
```

Damit ist die Fehlerbehandlung zu $LL(1)$-Parsern vollständig beschrieben. Ausgehend von einer reduzierten erweiterten rechtsregulären kontextfreien Grammatik G und der Angabe der Mengen $\mathsf{last}(G)$, $\mathsf{follow}(G)$, $BEG(G)$ und $PREC(G)$ wurde ein $LL(1)$-Parser konstruiert, der nicht nur korrekt die Sätze der von G beschriebenen Sprache analysiert, sondern auch eine ausgefeilte Fehlerbehandlung bereitstellt. Dafür wurde das rekursive Treiberprogramm mit immer mächtigeren Fehlerkorrekturmechanismen ausgestattet. Für eine gegebene Grammatik G kann die Funktion $\mathsf{process}()$ natürlich wieder zu Funktionen $\mathsf{process}_A$ für die Nichtterminale A der Grammatik G spezialisiert werden umd diese Spezialisierungen direkt aus der Grammatik G zusammen mit der Parsertabelle für G generiert werden. Die Beschreibung eines solchen Generators beschreibt Aufgabe 17.

3.4 *Bottom-up*-Syntaxanalyse

3.4.1 Einführung

Bottom-up-Parser lesen ihre Eingabe wie *top-down*-Parser von links nach rechts. Sie sind Kellerautomaten, die im wesentlichen zwei Operationen ausführen können:

- Kellern des nächsten Eingabesymbols (*shift*), und
- Lokalisieren der rechten Seite einer Produktion $X \to \alpha$ am oberen Kellerende und ihr Ersetzen durch einen Zustand, der der linken Seite X der Produktion entspricht (*reduce*).

Wegen dieser beiden Operationen heißen sie *shift-reduce*-Parser. Weil stets am oberen Kellerende reduziert wird, ist ein *shift-reduce*-Parser ein Rechtsparser: als Ergebnis einer erfolgreichen Analyse wird eine Rechtsableitung in gespiegelter Reihenfolge geliefert.

Ein *shift-reduce*-Parser darf niemals eine *gebotene* Reduktion verpassen d.h. durch ein eingelesenes Symbol im Keller zudecken. Dabei ist eine Reduktion *geboten*, wenn ohne sie keine gespiegelte Rechtsableitung bis zum Startsymbol möglich ist. Ist eine rechte Seite nämlich erst einmal zugedeckt, taucht sie nie mehr am oberen Ende des Kellers auf und kann darum nie mehr reduziert werden. Eine rechte Seite am oberen Kellerende, die reduziert werden muss, um eine Ableitung zu erhalten, nennen wir *Griff* (engl.: handle).

Nicht alle Vorkommen rechter Seiten am oberen Kellerende sind jedoch Griffe: manchmal führen Reduktionen am oberen Kellerende in Sackgassen, d.h. können nicht zu einer gespiegelten Rechtsableitung fortgesetzt werden.

Beispiel 3.4.1 G_0 sei wieder die Ausdrucksgrammatik mit den Produktionen:

$$
\begin{aligned}
S &\rightarrow E \\
E &\rightarrow E + T \mid T \\
T &\rightarrow T * F \mid F \\
F &\rightarrow (E) \mid \mathsf{Id}
\end{aligned}
$$

Tabelle 3.5 zeigt eine erfolgreiche *bottom-up*-Analyse des Satzes $\mathsf{Id} * \mathsf{Id}$ von G_0. In der dritten Spalte sind Aktionen angegeben, welche in der jeweiligen Analysesituation ebenfalls möglich wären, aber in Sackgassen führen. Im dritten Schritt würde man eine gebotene Reduktion verpassen. In den anderen beiden Schritten würden die alternativen Reduktionen in Sackgassen, d.h. nicht zu Rechtssatzformen führen. □

Keller	Eingabe	fehlerhafte alternative Aktionen
	$\mathsf{Id} * \mathsf{Id}$	
Id	$* \mathsf{Id}$	
F	$* \mathsf{Id}$	Lesen von $*$ verpasst gebotene Reduktion
T	$* \mathsf{Id}$	Reduktion von T zu E führt in eine Sackgasse
$T *$	Id	
$T * \mathsf{Id}$		
$T * F$		Reduktion von F zu T führt in eine Sackgasse
T		
E		
S		

Tabelle 3.5. Eine erfolgreiche Analyse des Satzes $\mathsf{Id} * \mathsf{Id}$ mit möglichen Sackgassen.

Bottom-up-Parser konstruieren den Syntaxbaum von *unten nach oben*. Sie beginnen bei dem Blattwort des Syntaxbaums, dem Eingabewort. Sie konstruieren für immer größere Teile der gelesenen Eingabe Unterbäume des Syntaxbaums, indem sie bei der Reduktion mit einer Produktion $X \rightarrow \alpha$ die Teilbäume für die rechte Seite α unter dem neuen Nichtterminalknoten X zusammenhängen. Die Analyse ist erfolgreich, wenn ein Syntaxbaum für die gesamte Eingabe konstruiert wurde, dessen Wurzelknoten mit dem Startsymbol der Grammatik markiert ist.

Abbildung 3.16 zeigt zwei Schnappschüsse des schrittweisen Aufbau des Syntaxbaums zur Ableitung aus Tabelle 3.5. Der Baum links versammelt alle Knoten, die nach Lesen des Symbols Id aufgebaut werden können. Die Folge der drei Bäume in der Mitte repräsentiert den Zustand, bevor der Griff $T * F$ reduziert wird, während rechts der vollständige Syntaxbaum abgebildet ist.

3.4.2 $LR(k)$-Analysatoren

In diesem Abschnitt wird das mächtigste deterministisch *bottom-up*-arbeitende Syntaxanalyseverfahren behandelt, die $LR(k)$-Analyse. Der Buchstabe steht L dafür, dass die Analysatoren dieser Klasse ihre Eingabe von links nach rechts lesen, und R kennzeichnet sie als Rechtsparser; k gibt an, wieviele Symbole sie in der Eingabe vorausschauen dürfen, um Entscheidungen zu treffen.

Wir gehen wieder von dem Item-Kellerautomaten K_G für eine kontextfreie Grammatik G aus und bauen ihn zu einem *shift-reduce*-Parser um. Bei der *top-down*-Analyse wurden aus der Grammatik Vorausschauworte berechnet, die für die *Expansionsübergänge* von K_G eine eindeutige Auswahl der zu wählenden Alternative gestatten. $LR(k)$-Analysatoren verfolgen stattdessen *alle* Möglichkeiten zu expandieren oder zu lesen *parallel*. Eine Entscheidung muss getroffen werden, wenn eine der verfolgten Möglichkeiten mit einem Reduktionsübergang fortgesetzt werden kann. Gibt es unterschiedliche Produktionen, für die eine Reduktion möglich ist, oder ist gleichzeitig auch ein Leseübergang möglich, werden zur Lösung dieses Konflikts die nächsten k Symbole betrachtet.

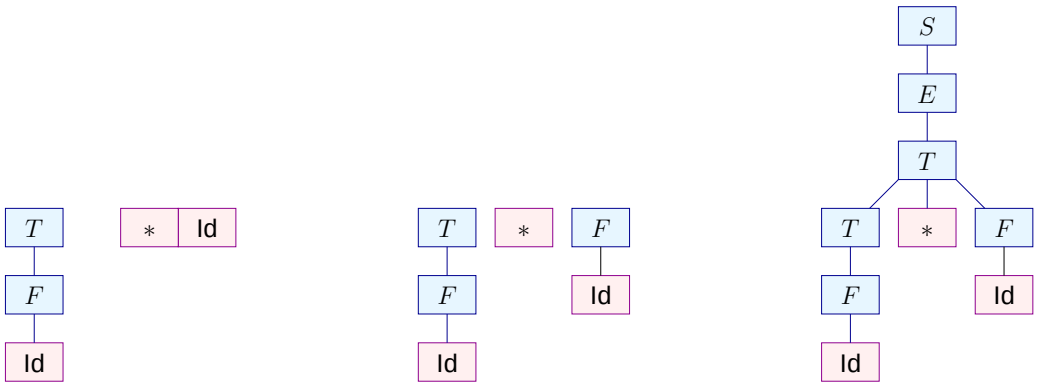


Abb. 3.16. Teilweiser Aufbau des Syntaxbaums nach Lesen des ersten Symbols Id mit verbleibender Eingabe, vor der Reduktion des Griffs $T * F$ sowie der vollständige Syntaxbaum.

In diesem Abschnitt wird zuerst ein $LR(0)$-Analysator entwickelt, der noch keine Vorausschau berücksichtigt. In Abschnitt 3.4.3 wird dann der *kanonische* $LR(k)$-Parser vorgestellt. In Abschnitt 3.4.3 werden schwächere, aber für die Praxis oft ausreichend starke Varianten der $LR(k)$-Analyse eingeführt. Zum Abschluss wird in Abschnitt 3.4.4 ein Verfahren zur Fehlerbehandlung für die $LR(k)$-Analysemethode beschrieben. Beachten Sie, dass bei allen gegebenen kontextfreien Grammatiken immer angenommen wird, dass sie um unproduktive und unerreichbare Nichtterminale reduziert und um ein neues Startsymbol erweitert sind.

Der charakteristische endliche Automat zu einer kontextfreien Grammatik

Unser Ziel ist ein Kellerautomat, der mögliche Expansions- und Leseübergänge des Item-Kellerautomaten parallel verfolgt und sich erst bei einer Reduktion festlegt, welche Regel im letzten Schritt angewendet wurde. Zu einer reduzierten erweiterten kontextfreien Grammatik G definieren wir deshalb den *charakteristischen* endlichen Automaten $\mathsf{char}(G)$. Die Zustände des charakteristischen Automaten $\mathsf{char}(G)$ sind die Items $[A \to \alpha.\beta]$ der Grammatik G, d.h. die Menge der Zustände des Item-Kellerautomaten K_G. Der Automat $\mathsf{char}(G)$ soll genau dann in den Zustand $[X \to \alpha.\beta]$ übergehen, wenn der Item-Kellerautomat einen Keller ρ erreichen kann, dessen *Geschichte* mit dem gelesenen Wort des charakteristischen Automaten übereinstimmt. Der charakteristische Automat liest also die Konkatenation der bereits abgearbeiteten Präfixe von Produktionen ein, deren Abarbeitung im Item-Kellerautomaten zu dem aktuellen Item führten. Weil diese Präfixe von Produktionen sowohl Terminal- wie Nichtterminalsymbole enthalten, ist die Menge der Eingabesymbole von $\mathsf{char}(G)$ gegeben durch $V_T \cup V_N$. Der Anfangszustand des charakteristischen Automaten stimmt mit dem Start-Item $[S' \to .S]$ des Item-Kellerautomaten K_G überein. Die Endzustände des charakteristischen Automaten sind die vollständigen Items $[X \to \alpha.]$. Ein solcher Endzustand signalisiert, dass das gelesene Wort einem Kellerinhalt des Item-Kellerautomaten entspricht, in dem eine Reduktion mit der Produktion $A \to \alpha$ durchgeführt werden kann. Die Übergangsrelation Δ des charakteristischen Automaten besteht aus den Übergängen:

$$\begin{array}{lll} ([X \to \alpha.Y\beta], \varepsilon, [Y \to .\gamma]) & \text{für} & X \to \alpha Y\beta \in P, \quad Y \to \gamma \in P \\ ([X \to \alpha.Y\beta], Y, [X \to \alpha Y.\beta]) & \text{für} & X \to \alpha Y\beta \in P, \quad Y \in V_N \cup V_T \end{array}$$

Das Lesen eines Terminalsymbols a entspricht einem *shift*-Übergang des Item-Kellerautomaten unter a. Die ε-Übergänge entsprechen den Expansionsübergängen, während das Lesen eines Nichtterminalsymbols dem Fortschalten des Punkts nach einem Reduktionsübergang des Item-Kellerautomaten entspricht.

Beispiel 3.4.2 Sei G_0 wieder die Grammatik für arithmetische Ausdrücke mit den Produktionen

$$
\begin{aligned}
S &\to E \\
E &\to E + T \mid T \\
T &\to T * F \mid F \\
F &\to (E) \mid \mathsf{Id}
\end{aligned}
$$

Den charakteristischen endlichen Automaten zu der Grammatik G_0 zeigt Abbildung 3.17. □

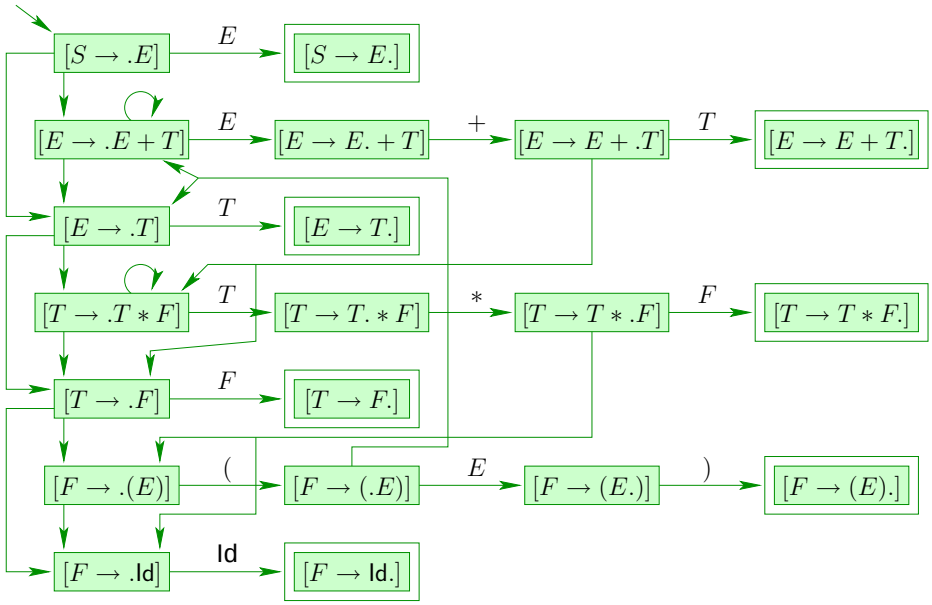


Abb. 3.17. Der charakteristische endliche Automat $\mathsf{char}(G_0)$ für die Grammatik G_0.

Das genaue Verhältnis zwischen dem charakteristischen endlichen Automaten und dem Item-Kellerautomaten klärt der folgende Satz:

Satz 3.4.1 Sei G eine erweiterte reduzierte kontextfreie Grammatik und $\gamma \in (V_T \cup V_N)^*$. Die folgenden drei Aussagen sind äquivalent:

1. Es gibt eine Berechnung $([S' \to .S], \gamma) \vdash^*_{\mathsf{char}(G)} ([A \to \alpha.\beta], \varepsilon)$ des charakteristischen Automaten $\mathsf{char}(G)$.
2. Es gibt eine Berechnung $(\rho\,[A \to \alpha.\beta], w) \vdash^*_{K_G} ([S' \to S.], \varepsilon)$ des Item-Kellerautomaten K_G, so dass $\gamma = \mathsf{hist}(\rho)\ \alpha$ gilt.
3. Es gibt eine Rechtsableitung $S' \underset{rm}{\overset{*}{\Longrightarrow}} \gamma' A w \underset{rm}{\Longrightarrow} \gamma'\alpha\beta w$ mit $\gamma = \gamma'\alpha$. □

Die Äquivalenz der Aussagen (1) und (2) besagt, dass die Wörter, die zu einem Item im charakteristischen endlichen Automaten zu G führen, genau den Geschichten von Kellerinhalten des Item-Kellerautomaten zu G entsprechen, deren oberstes Kellersymbol dieses Item ist, von denen aus bei Vorliegen einer geeigneten Eingabe w der Endzustand des Item-Kellerautomaten erreicht werden kann. Die Äquivalenz der Aussagen (2) und (3) besagt, dass eine akzeptierende Berechnung des Item-Kellerautomaten für ein Eingabewort w, die mit einem bestimmten Kellerinhalt ρ beginnt, einer Rechtsableitung entspricht, die zu einer Satzform αw führt, wobei α die Geschichte des Kellerinhalts ρ ist.

Bevor wir den Satz 3.4.1 beweisen, führen wir die folgende Sprechweise ein. Für eine Rechtsableitung $S' \underset{rm}{\overset{*}{\Longrightarrow}} \gamma' A v \underset{rm}{\Longrightarrow} \gamma\alpha v$ und eine Produktion $A \to \alpha$ nennen wir α den *Griff* der Rechtssatzform $\gamma\alpha v$. Ist die rechte Seite $\alpha = \alpha'\beta$, dann heißt das Präfix $\gamma = \gamma'\alpha'$ ein *zuverlässiges Präfix* von G für das Item $[A \to \alpha'.\beta]$. Das Item $[A \to \alpha.\beta]$ ist *gültig* für γ. Satz 3.4.1 besagt also, dass die Menge der

Wörter, mit denen der charakteristische Automat ein Item $[A \to \alpha'.\beta]$ erreicht, gerade die Menge der zuverlässigen Präfixe für dieses Item ist.

Beispiel 3.4.3 Für die Grammatik G_0 haben wir:

Rechtssatzform	Griff	zuverlässige Präfixe	Begründung
$E + F$	F	$E,\ E+,\ E + F$	$S \underset{rm}{\Longrightarrow} E \underset{rm}{\Longrightarrow} E + T \underset{rm}{\Longrightarrow} E + F$
$T * \mathsf{Id}$	Id	$T,\ T*,\ T * \mathsf{Id}$	$S \overset{3}{\underset{rm}{\Longrightarrow}} T * F \underset{rm}{\Longrightarrow} T * \mathsf{Id}$

□

In einer eindeutigen Grammatik ist der Griff einer Rechtssatzform das eindeutig bestimmte Teilwort, welches der *bottom-up*-Analysator im nächsten Reduktionsschritt durch ein Nichtterminal ersetzen muss, um zu einer Rechtsableitung zu kommen.

Beispiel 3.4.4 Wir geben zwei zuverlässige Präfixe von G_0 und einige für sie gültige Items an.

zuverlässiges Präfix	gültiges Item	Begründung
$E +$	$[E \to E + .T]$	$S \underset{rm}{\Longrightarrow} E \underset{rm}{\Longrightarrow} E + T$
	$[T \to .F]$	$S \overset{*}{\underset{rm}{\Longrightarrow}} E + T \underset{rm}{\Longrightarrow} E + F$
	$[F \to .\mathsf{Id}]$	$S \overset{*}{\underset{rm}{\Longrightarrow}} E + F \underset{rm}{\Longrightarrow} E + \mathsf{Id}$
$(E + ($	$[F \to (.E)]$	$S \overset{*}{\underset{rm}{\Longrightarrow}} (E + F) \underset{rm}{\Longrightarrow} (E + (E))$
	$[T \to .F]$	$S \overset{*}{\underset{rm}{\Longrightarrow}} (E + (.T) \underset{rm}{\Longrightarrow} (E + (F))$
	$[F \to .\mathsf{Id}]$	$S \overset{*}{\underset{rm}{\Longrightarrow}} (E + (F) \underset{rm}{\Longrightarrow} (E + (\mathsf{Id}))$

□

Ist beim Versuch, eine Rechtsableitung für einen Satz zu erstellen, das bisher gelesene Präfix u des Satzes zu einem zuverlässigen Präfix γ reduziert worden, dann beschreibt jedes für γ gültige Item $[X \to \alpha.\beta]$ eine mögliche Interpretation der Analysesituation. Es gibt also eine Rechtsableitung, in der γ Präfix einer Rechtssatzform und $X \to \alpha\beta$ eine der möglichen gerade bearbeiteten Produktionen ist. Alle solchen Produktionen sind Kandidaten für spätere Reduktionen.

Betrachten wir die Rechtsableitung $S' \overset{*}{\underset{rm}{\Longrightarrow}} \gamma A w \underset{rm}{\Longrightarrow} \gamma\alpha\beta w$. Da sie als Rechtsableitung fortgesetzt werden soll, muss eine Reihe von Schritten folgen, die β zu einem Terminalwort v ableitet, darauf eine Reihe von Schritten, die α zu einem Terminalwort u ableitet. Insgesamt ergibt sich $S' \overset{*}{\underset{rm}{\Longrightarrow}} \gamma A w \underset{rm}{\Longrightarrow} \gamma\alpha\beta w \overset{*}{\underset{rm}{\Longrightarrow}} \gamma\alpha v w \overset{*}{\underset{rm}{\Longrightarrow}} \gamma u v w \overset{*}{\underset{rm}{\Longrightarrow}} x u v w$. Jetzt betrachten wir diese Rechtsableitung in Reduktionsrichtung, d.h. in der Richtung, in welcher der *bottom-up*-Parser sie aufbaut. Es wird erst in einer Reihe von Schritten x zu γ reduziert, dann u zu α, dann v zu β. Das gültige Item $[A \to \alpha.\beta]$ für das zuverlässige Präfix $\gamma\alpha$ beschreibt die Analysesituation, in welcher die Reduktion von u nach α bereits geschehen ist, während die Reduktion von v nach β noch nicht begonnen hat. Ein mögliches Fernziel in dieser Situation ist die Anwendung der Produktion $X \to \alpha\beta$.

Wir kommen zu der Frage zurück, welche Sprache der charakteristische endliche Automat von K_G akzeptiert. Satz 3.4.1 besagt, dass er unter einem zuverlässigen Präfix in einen Zustand übergeht, der ein gültiges Item für dieses Präfix ist. Endzustände, d.h. vollständige Items, sind nur gültig für zuverlässige Präfixe, an deren Ende eine Reduktion möglich ist.

Beweis von Satz 3.4.1. Wir führen einen Ringschluss $(1) \Rightarrow (2) \Rightarrow (3) \Rightarrow (1)$ durch. Nehmen wir zuerst an, dass $([S' \to .S], \gamma) \overset{*}{\vdash}_{\mathsf{char}(G)} ([A \to \alpha.\beta], \varepsilon)$ gilt. Mit Induktion nach der Anzahl n der ε-Übergänge konstruieren wir eine Rechtsableitung $S' \overset{rm}{\underset{*}{\Longrightarrow}} \gamma A w \overset{rm}{\Longrightarrow} \gamma\alpha\beta w$.

Ist $n = 0$, dann ist $\gamma = \varepsilon$ und $[A \to \alpha.\beta] = [S' \to .S]$. Da $S' \overset{rm}{\underset{*}{\Longrightarrow}} S'$ gilt, ist die Behauptung in diesem Fall erfüllt. Ist $n > 0$, betrachten wir den letzten ε-Übergang. Dann lässt sich die Berechnung des charakteristischen Automaten zerlegen in:

$$([S' \to .S], \gamma) \overset{*}{\vdash}_{\mathsf{char}(G)} ([X \to \alpha'.A\beta'], \varepsilon) \vdash_{\mathsf{char}(G)} ([A \to .\alpha\beta], \alpha) \overset{*}{\vdash}_{\mathsf{char}(G)} ([A \to \alpha.\beta], \varepsilon)$$

wobei $\gamma = \gamma'\alpha$. Nach Induktionsannahme gibt es eine Rechtsableitung $S' \overset{rm}{\underset{*}{\Longrightarrow}} \gamma''Xw' \overset{rm}{\Longrightarrow} \gamma''\alpha'A\beta'w'$ mit $\gamma' = \gamma''\alpha'$. Da die Grammatik G reduziert ist, gibt es ebenfalls eine Rechtsableitung $\beta' \overset{rm}{\underset{*}{\Longrightarrow}} v$. Deshalb haben wir:

$$S' \overset{rm}{\underset{*}{\Longrightarrow}} \gamma'Avw' \overset{rm}{\Longrightarrow} \gamma'\alpha\beta w$$

mit $w = vw'$. Damit ist die Richtung $(1) \Rightarrow (2)$ bewiesen.

Nehmen wir an, wir hätten eine Rechtsableitung $S' \overset{rm}{\underset{*}{\Longrightarrow}} \gamma'Aw \overset{rm}{\Longrightarrow} \gamma'\alpha\beta w$. Diese Ableitung lässt sich zerlegen in:

$$S' \overset{rm}{\Longrightarrow} \alpha_1X_1\beta_1 \overset{rm}{\underset{*}{\Longrightarrow}} \alpha_1X_1v_1 \overset{rm}{\underset{*}{\Longrightarrow}} \ldots \overset{rm}{\underset{*}{\Longrightarrow}} (\alpha_1 \ldots \alpha_n)X_n(v_n \ldots v_1) \overset{rm}{\Longrightarrow} (\alpha_1 \ldots \alpha_n)\alpha\beta(v_n \ldots v_1)$$

für $X_n = A$. Mit Induktion nach n folgt, dass $(\rho, vw) \overset{*}{\underset{K_G}{\vdash}} ([S' \to S.], \varepsilon)$ gilt für

$$\begin{aligned} \rho &= [S' \to \alpha_1.X_1\beta_1] \ldots [X_{n-1} \to \alpha_n.X_n\beta_n] \\ w &= vv_n \ldots v_1 \end{aligned}$$

sofern $\beta \overset{*}{\underset{rm}{\Longrightarrow}} v$, $\alpha_1 = \beta_1 = \varepsilon$ und $X_1 = S$. Damit ergibt sich der Schluss $(2) \Rightarrow (3)$.

Für den letzten Schluss betrachten wir einen Kellerinhalt $\rho = \rho'\,[A \to \alpha.\beta]$ mit $(\rho, w) \overset{*}{\underset{K_G}{\vdash}} ([S' \to S.], \varepsilon)$. Zuerst überzeugen wir uns mit Induktion nach der Anzahl der Übergänge in einer solchen Berechnung, dass ρ' notwendigerweise von der Form:

$$\rho' = [S' \to \alpha_1.X_1\beta_1] \ldots [X_{n-1} \to \alpha_n.X_n\beta_n]$$

ist für ein $n \geq 0$ und $X_n = A$. Mit Induktion nach n folgt aber, dass $([S' \to .S], \gamma) \overset{*}{\underset{\mathsf{char}(G)}{\vdash}} ([A \to \alpha.\beta], \varepsilon)$ gilt für $\gamma = \alpha_1 \ldots \alpha_n\alpha$. Da $\gamma = \mathsf{hist}(\rho)$, gilt auch die Behauptung (1). Damit ist der Beweis vollständig. □

Der kanonische $LR(0)$-Automat

In Kapitel 2 wurde ein Verfahren vorgestellt, welches aus einem nichtdeterministischen endlichen Automaten einen äquivalenten deterministischen endlichen Automaten erzeugt. Dieser deterministische endliche Automat verfolgt alle Pfade parallel, die der nichtdeterministische für eine Eingabe durchlaufen könnte. Seine Zustände sind Mengen von Zuständen des nichtdeterministischen Automaten. Diese Teilmengenkonstruktion wird jetzt auf den charakteristischen endlichen Automaten $\mathsf{char}(G)$ einer kontextfreien Grammatik G angewendet. Den deterministischen endlichen Automaten, der sich so ergibt, nennen wir den *kanonischen* $LR(0)$-Automaten für G und bezeichnen ihn mit $LR_0(G)$.

Beispiel 3.4.5 Der $LR(0)$-Automat für die kontextfreie Grammatik G_0 aus Beispiel 3.2.2 auf Seite 41 ergibt sich durch Anwendung der Teilmengenkonstruktion auf den charakteristischen endlichen Automaten $\mathsf{char}(G_0)$ aus Abbildung 3.17 auf Seite 90. Ihn zeigt Abbildung 3.18 auf Seite 93. Seine Zustände sind gegeben durch:

$$
\begin{aligned}
S_0 = \{\ & [S \to .E], \\
& [E \to .E + T], \\
& [E \to .T], \\
& [T \to .T * F], \\
& [T \to .F], \\
& [F \to .(E)], \\
& [F \to .\mathsf{Id}]\ \}
\end{aligned}
$$

$$
\begin{aligned}
S_1 = \{\ & [S \to E.], \\
& [E \to E. + T]\ \}
\end{aligned}
$$

$$
\begin{aligned}
S_2 = \{\ & [E \to T.], \\
& [T \to T. * F]\ \}
\end{aligned}
$$

$$
S_3 = \{\ [T \to F.]\ \}
$$

$$
\begin{aligned}
S_4 = \{\ & [F \to (.E)], \\
& [E \to .E + T], \\
& [E \to .T], \\
& [T \to .T * F] \\
& [T \to .F] \\
& [F \to .(E)] \\
& [F \to .\mathsf{Id}]\ \}
\end{aligned}
$$

$$
S_5 = \{\ [F \to \mathsf{Id}.]\ \}
$$

$$
\begin{aligned}
S_6 = \{\ & [E \to E + .T], \\
& [T \to .T * F], \\
& [T \to .F], \\
& [F \to .(E)], \\
& [F \to .\mathsf{Id}]\ \}
\end{aligned}
$$

$$
\begin{aligned}
S_7 = \{\ & [T \to T * .F], \\
& [F \to .(E)], \\
& [F \to .\mathsf{Id}]\ \}
\end{aligned}
$$

$$
\begin{aligned}
S_8 = \{\ & [F \to (E.)], \\
& [E \to E. + T]\ \}
\end{aligned}
$$

$$
\begin{aligned}
S_9 = \{\ & [E \to E + T.], \\
& [T \to T. * F]\ \}
\end{aligned}
$$

$$
S_{10} = \{\ [T \to T * F.]\}
$$

$$
S_{11} = \{\ [F \to (E).]\ \}
$$

$$
S_{12} = \emptyset
$$

□

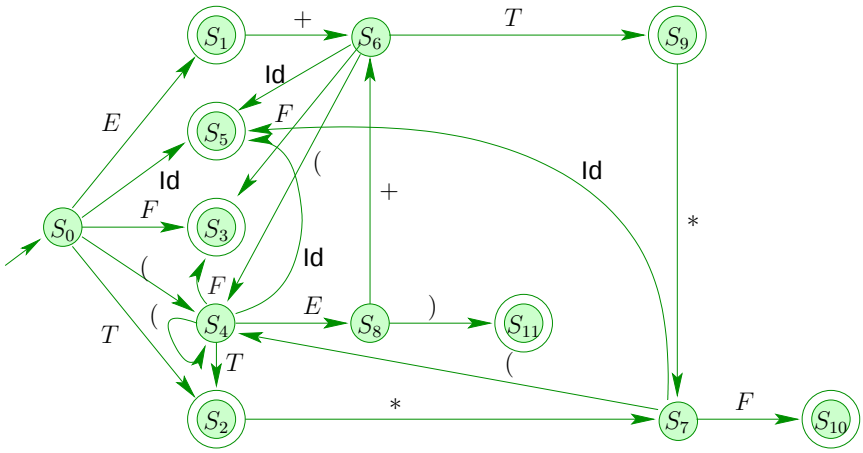


Abb. 3.18. Das Übergangsdiagramm des $LR(0)$-Automaten für die Grammatik G_0, der sich aus dem charakteristischen Automaten $\mathsf{char}(G_0)$ in Abbildung 3.17 ergibt. Der Fehlerzustand $S_{12} = \emptyset$ und alle Übergänge in den Fehlerzustand wurden weggelassen.

Der kanonische $LR(0)$-Automat $LR_0(G)$ zu einer kontextfreien Grammatik G hat einige interessante Eigenschaften. Sei $LR_0(G) = (Q_G, V_T \cup V_N, \Delta_G, q_{G,0}, F_G)$ und $\Delta_G^* : Q_G \times (V_T \cup V_N)^* \to Q_G$ die Fortsetzung der Übergangsfunktion Δ_G von Symbolen auf Wörter. Dann gilt:

1. $\Delta_G^*(q_{G,0}, \gamma)$ ist die Menge aller Items aus $\mathcal{I}_G$, für die γ ein zuverlässiges Präfix ist.
2. $L(LR_0(G))$ ist die Menge aller zuverlässigen Präfixe für vollständige Items $[A \to \alpha.] \in \mathcal{I}_G$.

Zuverlässige Präfixe sind Präfixe von Rechtssatzformen, wie sie während der Reduktion eines Eingabewortes auftreten. Wenn in ihnen eine Reduktion möglich ist, die wieder zu einer Rechtssatzform führt, dann kann dies nur am äußersten rechten Ende passieren. Ein für ein zuverlässiges Präfix gültiges Item beschreibt eine mögliche Sicht der aktuellen Analysesituation.

Beispiel 3.4.6 $E + F$ ist ein zuverlässiges Präfix für die Grammatik G_0. Der Zustand $\Delta_{G_0}^*(S_0, E + F) = S_3$ wird auch durch die folgenden zuverlässigen Präfixe erreicht:

$$\begin{array}{llll} F\,, & (F\,, & ((F\,, & (((F\,, \quad \ldots \\ T*(F\,, & T*((F\,, & T*(((F\,, & \ldots \\ E+F\,, & E+(F\,, & E+((F\,, & \ldots \end{array}$$

Der Zustand S_6 in dem kanonischen $LR(0)$-Automaten zu G_0 enthält alle gültigen Items für das zuverlässige Präfix $E+$, nämlich die Items:

$$[E \to E + .T], [T \to .T * F], [T \to .F], [F \to .\mathsf{Id}], [F \to .(E)].$$

Denn $E+$ ist Präfix der Rechtssatzform $E + T$:

$$\begin{array}{lllllll} S \underset{rm}{\Longrightarrow} E \underset{rm}{\Longrightarrow} & E+T & \underset{rm}{\Longrightarrow} & E+F & \underset{rm}{\Longrightarrow} & E+\mathsf{Id} \\ & \uparrow & & \uparrow & & \uparrow \\ \text{Gültig sind z.B.} & [E \to E + .T] & & [T \to .F] & & [F \to .\mathsf{Id}] \end{array}$$

□

Der kanonische $LR(0)$-Automat $LR_0(G)$ zu einer kontextfreien Grammatik G ist ein deterministischer endlicher Automat, der die zuverlässigen Präfixe zu vollständigen Items akzeptiert. Weil er so Reduktionsstellen identifiziert, bietet er sich für die Konstruktion eines Rechtsparsers an. Anstelle von Items (wie der Item-Kellerautomat) kellert dieser Parser Zustände des kanonischen $LR(0)$-Automaten, also *Mengen* von Items. Den zugrunde liegenden Kellerautomaten K_0 definieren wir durch das Tupel $K_0 = (Q_G \cup \{f\}, V_T, \Delta_0, q_{G,0}, \{f\})$. Die Menge der Zustände ist gegeben durch die Menge Q_G der Zustände des kanonischen $LR(0)$-Automaten $LR_0(G)$, erweitert um einen neuen Zustand f; der Anfangszustand von K_0 ist gleich dem Anfangszustand $q_{G,0}$ von $LR_0(G)$; der Endzustand ist gegeben durch den neuen Zustand f. Die Übergangsrelation Δ_0 besteht aus den folgenden Übergängen:

Lesen: $(q, a, q\,\delta_G(q, a)) \in \Delta_0$, falls $\delta_G(q, a) \neq \emptyset$ ist. In diesem Übergang werden das nächste Eingabesymbol a gelesen und der Nachfolgezustand von q unter a gekellert. Er ist nur dann möglich, wenn mindestens ein Item der Form $[X \to \alpha.a\beta]$ in q vorhanden ist.

Reduzieren: $(qq_1 \ldots q_n, \varepsilon, q\,\delta_G(q, X)) \in \Delta$, falls $[X \to \alpha.] \in q_n$ gilt mit $|\alpha| = n$. Das vollständige Item $[X \to \alpha.]$ im obersten Kellereintrag signalisiert eine mögliche Reduktion. Daraufhin werden soviele Einträge aus dem Keller entfernt, wie die rechte Seite lang ist. Danach wird der X-Nachfolger des neuen obersten Kellereintrags gekellert. In Abbildung 3.19 wird ein Ausschnitt aus dem Übergangsdiagramm eines $LR_0(G)$ gezeigt, der diese Situation widerspiegelt. Dem α-Weg im Übergangsdiagramm entsprechen $|\alpha|$ Einträge oben auf dem Keller. Diese Einträge werden bei der Reduktion entfernt. Der darunterliegende neue oberste Zustand hat einen Übergang unter X, der jetzt durchlaufen wird.

Abschließen: $(q_{G,0}\, q, \varepsilon, f)$ falls $[S' \to S.] \in q$. Dieser Übergang ist der Reduktionsübergang zu der Produktion $S' \to S$. Die Eigenschaft $[S' \to S.] \in q$ signalisiert, dass ein Satz erfolgreich auf das Startsymbol reduziert wurde. Der Abschlussübergang leert den Keller und ersetzt ihn durch den Endzustand f.

Der Sonderfall $[X \to .\,]$ verdient Beachtung. Nach unserer Darstellung sind bei einer Reduktion $|\varepsilon| = 0$ oberste Kellereinträge zu entfernen, und aus dem neuen (wie alten) aktuellen Zustand q ein Übergang unter X vorzunehmen und den Zustand $\Delta_G(q, X)$ zu kellern. Durch diesen Reduktionsübergang *verlängert* sich also der Keller.

Die Konstruktion von $LR_0(G)$ garantiert, dass es zu jedem seiner Zustände q, die verschieden von Start- und Fehlerzustand sind, jeweils genau ein Eingangssymbol gibt, mit dem der Automat in q übergehen kann. Deshalb entspricht einem Kellerinhalt $q_0, \ldots, q_n$ mit $q_0 = q_{G,0}$ eindeutig ein Wort $\alpha = X_1 \ldots X_n \in (V_T \cup V_N)^*$, für das $\Delta_G(q_i, X_{i+1}) = q_{i+1}$ gilt. Dieses Wort α ist ein zuverlässiges Präfix, und q_n ist die Menge aller für α gültigen Items.

Der Kellerautomat K_0, den wir gerade konstruiert haben, ist jedoch nicht notwendigerweise deterministisch. Es gibt zwei Arten von Konflikten, die Nichtdeterminismus verursachen:

shift-reduce–Konflikt: ein Zustand q erlaubt sowohl einen Leseübergang unter einem Symbol $a \in V_T$ als auch einen Reduktions- bzw. Abschlussübergang, und

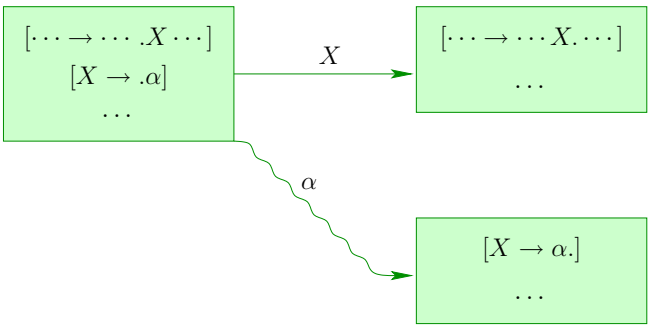


Abb. 3.19. Ausschnitt aus dem Übergangsdiagramm eines kanonischen $LR(0)$-Automaten.

reduce-reduce–Konflikt: ein Zustand q erlaubt Reduktions- bzw. Abschlussübergänge gemäß zweier verschiedener Produktionen.

Im ersten Fall enthält der aktuelle Zustand mindestens ein Item $[X \to \alpha.a\beta]$ und mindestens ein vollständiges Item $[Y \to \gamma.]$; im zweiten Fall enthält q zwei verschiedene vollständige Items $[Y \to \alpha.]$, $[Z \to \beta.]$. Einen Zustand q des $LR(0)$-Automaten mit einer dieser Eigenschaften nennen wir auch $LR(0)$-*ungeeignet*. Andernfalls nennen wir q $LR(0)$-*geeignet*. Es gilt:

Lemma 3.4. Für einen $LR(0)$-*geeigneten* Zustand q gibt es drei Möglichkeiten:

1. Der Zustand q enthält kein vollständiges Item.
2. Der Zustand q besteht aus genau einem vollständigen Item $[A \to \alpha.]$;
3. Der Zustand q enthält genau ein vollständiges Item $[A \to .]$, und alle unvollständigen Items in q sind von der Form $[X \to \alpha.Y\beta]$, wobei alle Rechtsableitungen für Y, die zu einem terminalen Wort führen, von der Form:
$$Y \underset{rm}{\overset{*}{\Longrightarrow}} Aw \underset{rm}{\Longrightarrow} w$$
sind für ein $w \in V_T^*$. □

Ungeeignete Zustände des kanonischen $LR(0)$-Automaten machen den Kellerautomaten K_0 nichtdeterministisch. Deterministische Parser erhalten wir, indem eine Vorausschau in die restliche Eingabe eingesetzt wird, um die richtige Aktion in ungeeigneten Zuständen auswählen.

Beispiel 3.4.7 Die Zustände S_1, S_2 und S_9 des kanonischen $LR(0)$-Automat in Abbildung 3.18 sind $LR(0)$-ungeeignet. Im Zustand S_1 kann die rechte Seite E zu der linken Seite S reduziert (vollständiges Item $[S \to E.]$) oder das Terminalsymbol $+$ in der Eingabe gelesen werden (Item $[E \to E. + T]$). Im Zustand S_2 kann die rechte Seite T zu der linken Seite E reduzier (vollständiges Item $[E \to T.]$) oder das Terminalsymbol $*$ gelesen werden (Item $[T \to T. * F]$). Im Zustand S_9 schließlich kann der Parser die rechte Seite $E + T$ zu E reduzieren (vollständiges Item $[E \to E + T.]$) oder das Terminalsymbol $*$ lesen (Item $[T \to T. * F]$). □

Direkte Konstruktion des kanonischen $LR(0)$-Automaten

Der kanonische $LR(0)$-Automat $LR_0(G)$ zu einer kontextfreien Grammatik G muss nicht über den Umweg der Konstruktion des charakteristischen endlichen Automaten $\mathsf{char}(G)$ und die Teilmengenkonstruktion erstellt werden. Er lässt sich auch direkt aus G konstruieren. Als Hilfsfunktion benötigen wir dazu die Funktion $\Delta_{G,\varepsilon}$, die zu einer Menge q von Items alle Items hinzufügt, die durch ε-Übergänge des charakteristischen Automaten erreichbar sind. Die Menge $\Delta_{G,\varepsilon}(q)$ ist deshalb die kleinste Lösung der Gleichung:

$$I = q \cup \{[A \to .\gamma] \mid \exists X \to \alpha A\beta \in P : \ [X \to \alpha.A\beta] \in I\}$$

In Anlehnung an die Funktion closure() der Teilmengenkonstruktion können wir sie deshalb berechnen durch:

```
set⟨item⟩ closure(set⟨item⟩ q) {
    set⟨item⟩ result ← q;
    list⟨item⟩ W ← list_of(q);
    symbol X; string⟨symbol⟩ α;
    while (W ≠ []) {
        item i ← hd(W); W ← tl(W);
        switch (i) {
        case [_ → _ .X _] : forall (α : (X → α) ∈ P)
                                if ([X → .α] ∉ result) {
                                    result ← result ∪ {[X → .α]};
                                    W ← [X → .α] :: W;
                                }
        default :           break;
        }
    }
    return result;
}
```

wobei V die Menge aller Symbole $V = V_T \cup V_N$ bezeichnet. Die Menge Q_G der Zustände und die Übergangsrelation Δ_G berechnen wir, indem wir zuerst den Anfangszustand $q_{G,0} = \Delta_{G,\varepsilon}(\{[S' \to .S]\})$ konstruieren und so lange Nachfolgezustände und Übergänge hinzufügen, bis alle Nachfolgezustände bereits in der Menge der gefundenen Zustände enthalten sind. Zur Implementierung spezialisieren wir die Hilfsfunktion nextState() der Teilmengenkonstruktion:

```
set⟨item⟩ nextState(set⟨item⟩ q, symbol X) {
    set⟨item⟩ q' ← ∅;
    nonterminal A; string⟨symbol⟩ α, β;
    forall (A, α, β : ([A → α.Xβ] ∈ q))
        q' ← q' ∪ {[A → αX.β]};
    return closure(q');
    }
```

Wie bei der Teilmengenkonstruktion kann die Menge aller Zustände *states* und die Menge aller Übergänge *trans* iterativ berechnet werden durch:

```
list⟨set⟨item⟩⟩ W;
set⟨item⟩ q0 ← closure({[S' → .S]});
states ← {q0}; W ← [q0];
trans ← ∅;
set⟨item⟩ q, q';
while (W ≠ []) {
    q ← hd(W); W ← tl(W);
    forall (symbol X) {
        q' ← nextState(q, X);
        trans ← trans ∪ {(q, X, q')};
        if (q' ∉ states) {
            states ← states ∪ {q'};
            W ← q' :: W;
        }
    }
}
```

3.4.3 $LR(k)$: Definition, Eigenschaften, Beispiele

Sei $S' = \alpha_0 \underset{rm}{\Longrightarrow} \alpha_1 \underset{rm}{\Longrightarrow} \alpha_2 \cdots \underset{rm}{\Longrightarrow} \alpha_m = v$ eine Rechtsableitung zu einer kontextfreien Grammatik G. Wir nennen G eine $LR(k)$-Grammatik, wenn in jeder solchen Rechtsableitung und jeder darin auftretenden Rechtssatzform α_i

- der Griff lokalisiert werden kann, und
- die anzuwendende Produktion bestimmt werden kann,

indem man α_i von links bis höchstens k Symbole hinter dem Griff betrachtet. In einer $LR(k)$-Grammatik ist also die Aufteilung von α_i in $\gamma\beta w$ und die Bestimmung von $X \to \beta$, so dass $\alpha_{i-1} = \gamma X w$ ist, eindeutig durch $\gamma\beta$ und $w|_k$ bestimmt. Formal nennen wir G deshalb eine $LR(k)$-Grammatik, wenn aus

$$\begin{array}{lll} S' \overset{*}{\underset{rm}{\Longrightarrow}} \alpha X w \underset{rm}{\Longrightarrow} \alpha\beta w & & \text{und} \\ S' \overset{*}{\underset{rm}{\Longrightarrow}} \gamma Y x \underset{rm}{\Longrightarrow} \alpha\beta y & & \text{und} \\ w|_k = y|_k & \text{folgt, dass} & \alpha = \gamma \wedge X = Y \wedge x = y. \end{array}$$

Beispiel 3.4.8 Sei G die Grammatik mit den Produktionen:

$$S \to A \mid B \qquad A \to aAb \mid 0 \qquad B \to aBbb \mid 1$$

Dann ist $L(G) = \{a^n 0 b^n \mid n \geq 0\} \cup \{a^n 1 b^{2n} \mid n \geq 0\}$. Wir wissen schon, dass G für kein $k \geq 1$ eine $LL(k)$-Grammatik ist. Die Grammatik G ist jedoch eine $LR(0)$-Grammatik.

Die Rechtssatzformen von G haben die Formen

$$S, \quad \underline{A}, \quad \underline{B}, \quad a^n \underline{aAb} b^n, \quad a^n \underline{aBbb} b^{2n}, \quad a^n a \underline{0} b b^n, \quad a^n a \underline{1} b b b^{2n}$$

für $n \geq 0$. Dabei wurde der Griff jeweils unterstrichen. Nur im Falle der Rechtssatzformen $a^n aAbb^n$ und $a^n aBbbb^{2n}$ gibt es jeweils zwei verschiedene mögliche Reduktionen. Man könnte $a^n aAbb^n$ zu $a^n Ab^n$ und zu $a^n aSbb^n$ reduzieren. Die erste gehört zu der Rechtsableitung

$$S \overset{*}{\underset{rm}{\Longrightarrow}} a^n Ab^n \underset{rm}{\Longrightarrow} a^n aAbb^n$$

die zweite gehört zu keiner Rechtsableitung. Aus dem Präfix a^n von $a^n Ab^n$ ergibt sich eindeutig, ob A Griff ist, nämlich im Fall $n = 0$, oder ob aAb Griff ist, nämlich im Fall $n > 0$. Die Rechtssatzformen $a^n Bb^{2n}$ werden analog behandelt. □

Beispiel 3.4.9 Die Grammatik G_1 mit den Produktionen

$$S \to aAc \qquad A \to Abb \mid b$$

und der Sprache $L(G_1) = \{ab^{2n+1}c \mid n \geq 0\}$ ist eine $LR(0)$-Grammatik. In einer Rechtssatzform $aAbbb^{2n}c$ gibt es nur die Reduktion zu $aAb^{2n}c$ als Teil einer Rechtsableitung. Der Präfix $aAbb$ bestimmt dies eindeutig. Für die Rechtssatzform $abb^{2n}c$ ist b der Griff, und der Präfix ab bestimmt dies eindeutig. □

Beispiel 3.4.10 Die Grammatik G_2 mit den Produktionen

$$S \to aAc \qquad A \to bbA \mid b$$

mit der Sprache $L(G_2) = L(G_1)$ ist eine $LR(1)$-Grammatik. Die kritischen Rechtssatzformen haben die Form ab^nw. Falls $1 : w = b$, so liegt der Griff in w; falls $1 : w = c$, so bildet das letzte b in b^n den Griff. □

Beispiel 3.4.11 Die Grammatik G_3 mit den Produktionen

$$S \to aAc \qquad A \to bAb \mid b$$

und mit $L(G_3) = L(G_1)$ ist keine $LR(k)$-Grammatik für irgendein $k \geq 0$. Denn sei k beliebig, aber fest gewählt. Man betrachte die zwei Rechtsableitungen

$$S \underset{rm}{\overset{*}{\Longrightarrow}} ab^nAb^nc \underset{rm}{\Longrightarrow} ab^nbb^nc$$
$$S \underset{rm}{\overset{*}{\Longrightarrow}} ab^{n+1}Ab^{n+1}c \underset{rm}{\Longrightarrow} ab^{n+1}bb^{n+1}c$$

mit $n \geq k$. Hier sind mit den Bezeichnungen aus der Definition einer $LR(k)$-Grammatik: $\alpha = ab^n, \beta = b, \gamma = ab^{n+1}, w = b^nc, y = b^{n+2}c$. Dabei ist $w|_k = y|_k = b^k$. Aus $\alpha \neq \gamma$ folgt, dass G_3 keine $LR(k)$-Grammatik sein kann. □

Der folgende Satz stellt eine Beziehung zwischen der Definition einer $LR(0)$-Grammatik und den Eigenschaften des kanonischen $LR(0)$-Automaten her.

Satz 3.4.2 Eine kontextfreie Grammatik G ist genau dann eine $LR(0)$-Grammatik, wenn der kanonische $LR(0)$-Automat zu G keine $LR(0)$-ungeeigneten Zustände hat.

Beweis: ” $\Rightarrow$ ” Sei G eine $LR(0)$-Grammatik, und nehmen wir an, der kanonische $LR(0)$-Automat $LR_0(G)$ habe einen einen $LR(0)$-ungeeigneten Zustand p.

Fall 1: Der Zustand p hat einen *reduce-reduce*-Konflikt, d.h. p enthält zwei verschiedene Items $[X \to \beta.],\ [Y \to \delta.]$. Dem Zustand p zugeordnet ist eine nichtleere Menge von zuverlässigen Präfixen. Sei $\gamma = \gamma'\beta$ ein solches zuverlässiges Präfix. Weil beide Items gültig für γ sind, gibt es Rechtsableitungen

$$S' \underset{rm}{\overset{*}{\Longrightarrow}} \gamma' Xw \underset{rm}{\Longrightarrow} \gamma'\beta w \qquad \text{und}$$
$$S' \underset{rm}{\overset{*}{\Longrightarrow}} \nu Yy \underset{rm}{\Longrightarrow} \nu\delta y \qquad \text{mit} \quad \nu\delta = \gamma'\beta = \gamma$$

Das ist aber ein Widerspruch zur $LR(0)$-Eigenschaft.

Fall 2: Zustand p hat einen *shift-reduce*-Konflikt, d.h. p enthält Items $[X \to \beta.]$ und $[Y \to \delta.a\alpha]$. Sei γ ein zuverlässiges Präfix für beide Item Weil beide Items gültig für γ sind, gibt es Rechtsableitungen

$$S' \underset{rm}{\overset{*}{\Longrightarrow}} \gamma' Xw \underset{rm}{\Longrightarrow} \gamma'\beta w \qquad \text{und}$$
$$S' \underset{rm}{\overset{*}{\Longrightarrow}} \nu Yy \underset{rm}{\Longrightarrow} \nu\delta a\alpha y \qquad \text{mit} \quad \nu\delta = \gamma'\beta = \gamma$$

Ist $\beta' \in V_T^*$, erhalten wir sofort einen Widerspruch. Andernfalls gibt es eine Rechtsableitung

$$\alpha \underset{rm}{\overset{*}{\Longrightarrow}} v_1 X v_3 \underset{rm}{\Longrightarrow} v_1 v_2 v_3$$

Weil $y \neq a v_1 v_2 v_3 y$ gilt, ist die $LR(0)$-Eigenschaft verletzt.

" $\Leftarrow$ " Nehmen wir an, der kanonische $LR(0)$-Automat $LR_0(G)$ habe keine $LR(0)$-ungeeigneten Zustände. Betrachten wir die zwei Rechtsableitungen:

$$S' \underset{rm}{\overset{*}{\Longrightarrow}} \alpha X w \underset{rm}{\Longrightarrow} \alpha\beta w$$
$$S' \underset{rm}{\overset{*}{\Longrightarrow}} \gamma Y x \underset{rm}{\Longrightarrow} \alpha\beta y$$

Zu zeigen ist, dass $\alpha = \gamma,\ X = Y,\ x = y$ gelten. Sei p der Zustand des kanonischen $LR(0)$-Automaten nach Lesen von $\alpha\beta$. Dann enthält p alle für $\alpha\beta$ gültigen Items. Nach Voraussetzung ist p $LR(0)$-geeignet. Wir unterscheiden zwei Fälle:

Fall 1: $\beta \neq \varepsilon$. Wegen Lemma 3.4 ist $p = \{[X \to \beta.]\}$, d.h. $[X \to \beta.]$ ist das einzige gültige Item für $\alpha\beta$. Daraus folgt, dass $\alpha = \gamma,\ X = Y$ und $x = y$ sein muss.

Fall 2: $\beta = \varepsilon$. Nehmen wir an, die zweite Rechtsableitung widerspreche der $LR(0)$-Bedingung. Dann gibt es ein weiteres Item $[X \to \delta.Y'\eta] \in p$, so dass $\alpha = \alpha'\delta$ ist. Die letzte Anwendung einer Produktion in der unteren Rechtsableitung ist die letzte Anwendung einer Produktion in einer terminalen Rechtsableitung für Y'. Nach Lemma 3.4 folgt daraus, dass die untere Ableitung gegeben ist durch:

$$S' \underset{rm}{\overset{*}{\Longrightarrow}} \alpha'\delta Y' w \underset{rm}{\overset{*}{\Longrightarrow}} \alpha'\delta X v w \underset{rm}{\Longrightarrow} \alpha'\delta v w$$

wobei $y = vw$ ist. Damit gilt $\alpha = \alpha'\delta = \gamma$, $Y = X$ und $x = vw = y$ – im Widerspruch zu unserer Annahme. □

Fassen wir zusammen. Ausgehend von einer kontextfreien Grammatik G können wir den kanonischen $LR(0)$-Automaten $LR_0(G)$ konstruieren: entweder direkt oder auf dem Umweg über den charakteristischen endlichen Automaten $\mathsf{char}(G)$. Mit Hilfe des deterministischen endlichen Automaten $LR_0(G)$ können wir einen Kellerautomaten K_0 konstruieren. Der Kellerautomat K_0 ist deterministisch, wenn $LR_0(G)$ keine $LR(0)$-ungeeigneten Zustände enthält. Satz 3.4.2 besagt, dass dies genau dann der Fall ist, wenn die Grammatik G eine $LR(0)$-Grammatik ist. Damit haben wir ein Verfahren zur Generierung eines Parsers für $LR(0)$-Grammatiken.

In der Praxis kommen $LR(0)$-Grammatiken jedoch nicht sehr häufig vor. Oft muss eine Vorausschau der Länge $k > 0$ eingesetzt werden, um zwischen unterschiedlichen Aktionen des Parsers auswählen zu können. In einem $LR(0)$-Parser legt der aktuelle Zustand fest, was die nächste Aktion ist – unabhängig von den nächsten Symbolen in der Eingabe. $LR(k)$-Parser für $k > 0$ haben Zustände, die ebenfalls aus Mengen von Items bestehen. Als Items werden hier allerdings nicht mehr kontextfreie Items verwendet, sondern $LR(k)$-Items. $LR(k)$-Items sind kontextfreie Items, erweitert um Vorausschauwörter. Ein $LR(k)$-Item ist von der Form $i = [A \to \alpha.\beta, x]$ für eine Produktion $A \to \alpha\beta$ von G und ein Wort $x \in (V_T^k \cup V_T^{<k}\#)$. Das kontextfreie Item $[A \to \alpha.\beta]$ heißt der *Kern*, das Wort x die *Vorausschau* des $LR(k)$-Items i. Die Menge aller $LR(k)$-Items für die Grammatik G bezeichnen wir mit $\mathcal{I}_{G,k}$. Das $LR(k)$-Item $[A \to \alpha.\beta, x]$ ist *gültig* für ein zuverlässiges Präfix γ, wenn es eine Rechtsableitung

$$S'\# \underset{rm}{\overset{*}{\Longrightarrow}} \gamma' X w\# \underset{rm}{\Longrightarrow} \gamma'\alpha\beta w\#$$

gibt mit $x = (w\#)|_k$. Ein kontextfreies Item $[A \to \alpha.\beta]$ können wir als $LR(0)$-Item auffassen, indem wir es um die Vorausschau ε erweitern.

Beispiel 3.4.12 Betrachten wir wieder die Grammatik G_0. Dann gilt:

(1)	$[E \to E + .T,)]$ $[E \to E + .T, +]$	sind gültige $LR(1)$-Items für das Präfix $(E+$
(2)	$[E \to T., *]$	ist kein gültiges $LR(1)$-Item für irgendein zuverlässiges Präfix.

Um die Beobachtung (1) einzusehen, betrachtet man die beiden Rechtsableitungen:

$$S' \underset{rm}{\overset{*}{\Longrightarrow}} (E) \underset{rm}{\Longrightarrow} (E+T)$$
$$S' \underset{rm}{\overset{*}{\Longrightarrow}} (E+\mathsf{Id}) \underset{rm}{\Longrightarrow} (E+T+\mathsf{Id})$$

Die Beobachtung (2) folgt, weil in keiner Rechtssatzform das Teilwort $E*$ auftreten kann. □

Der folgende Satz gibt eine Charakterisierung der $LR(k)$-Eigenschaft mit Hilfe gültiger $LR(k)$-Items.

Satz 3.4.3 Sei G eine kontextfreie Grammatik. Für ein zuverlässiges Präfix γ sei $It(\gamma)$ die Menge der $LR(k)$-Items von G, die für γ gültig sind.

Die Grammatik G ist genau dann eine $LR(k)$-Grammatik, wenn für alle zuverlässigen Präfixe γ und alle $LR(k)$-Items $[A \to \alpha., x] \in It(\gamma)$ gilt:

1. Gibt es ein weiteres $LR(k)$-Item $[X \to \delta., y] \in It(\gamma)$, dann ist $x \neq y$.
2. Gibt es ein weiteres $LR(k)$-Item $[X \to \delta.a\beta, y] \in It(\gamma)$, dann ist $x \notin \mathsf{first}_k(a\beta) \odot_k \{y\}$. □

Satz 3.4.3 gibt Anlass, auch für $k > 0$ $LR(k)$-geeignete bzw. $LR(k)$-ungeeignete Mengen von Items zu definieren. Sei I eine Menge von $LR(k)$-Items. Dann hat I einen *reduce-reduce*-Konflikt, wenn es $LR(k)$-Items $[X \to \alpha., x], [Y \to \beta., y] \in I$ gibt mit $x = y$. I hat einen *shift-reduce*-Konflikt, wenn es $LR(k)$-Items $[X \to \alpha.a\beta, x], [Y \to \gamma., y] \in I$ gibt mit

$$y \in \{a\} \odot_k \mathsf{first}_k(\beta) \odot_k \{x\}$$

Für $k = 1$ vereinfacht sich diese Bedingung zu $y = a$.

Die Menge I nennen wir $LR(k)$-ungeeignet, wenn sie einen *reduce-reduce*- oder einen *shift-reduce*-Konflikt besitzt. Andernfalls nennen wir sie $LR(k)$-geeignet.

Die $LR(k)$-Eigenschaft bedeutet, dass man beim Lesen einer Rechtssatzform einen Kandidaten für eine Reduktion zusammen mit der anzuwendenden Produktion eindeutig mithilfe des dazugehörigen zuverlässigen Präfixes und der k nächsten Symbole der Eingabe identifizieren kann. Wenn wir jedoch alle Kombinationen aus zuverlässigen Präfixen und Wörtern der Länge k tabellieren wollten, hätten wir Schwierigkeiten, weil es i.A. unendlich viele zuverlässige Präfixe gibt. Analog zu unserem Vorgehen bei $LR(0)$-Grammatiken können wir jedoch einen kanonischen $LR(k)$-Automaten konstruieren. Der kanonische $LR(k)$-Automat $LR_k(G)$ ist ein deterministischer endlicher Automat. Seine Zustände sind Mengen von $LR(k)$-Items. Für jedes zuverlässige Präfix γ liefert der deterministische endliche Automat $LR_k(G)$ die Menge aller $LR(k)$-Items, die für γ gültig sind. Nun hilft uns Satz 3.4.3 weiter. Wegen Satz 3.4.3 bestimmt für eine $LR(k)$-Grammatik die Menge der für γ gültigen $LR(k)$-Items zusammen mit der Vorausschau eindeutig, ob im nächsten Schritt reduziert werden muss, und wenn ja, mit welcher Produktion.

So wie der $LR(0)$-Parser Zustände des kanonischen $LR(0)$-Automaten kellert, kellert der $LR(k)$-Parser Zustände des kanonischen $LR(k)$-Automaten. Die Auswahl zwischen verschiedenen möglichen Aktionen des $LR(k)$-Parsers wird durch die *action*-Tabelle gesteuert. Diese Tabelle enthält für jede Kombination von Zustand und Vorausschau einen der folgenden Einträge:

shift:	lies das nächste Eingabesymbol;
reduce$(X \to \alpha)$:	reduziere mittels der Produktion $X \to \alpha$;
error:	melde Fehler und
accept:	melde erfolgreiches Ende des Parserlaufs.

Eine zweite Tabelle, die *goto*-Tabelle, enthält die Darstellung der Übergangsfunktion des kanonischen $LR(k)$-Automaten $LR_k(G)$. Sie wird konsultiert, wenn eine *shift*-Aktion oder eine *reduce*-Aktion passiert ist, um den neuen aktuellen Zustand oben auf dem Keller zu berechnen. Bei einem *shift* bestimmt sie den Übergang aus dem aktuellen Zustand unter dem gelesenen Symbol; bei einer Reduktion mittels $X \to \alpha$ den Übergang unter X aus dem Zustand unterhalb der Kellerzustände oben auf dem Keller, die zu α gehören.

Neben *action*- und *goto*-Tabelle benötigt der $LR(k)$-Parser für eine Grammatik G ein Programm, welches diese Tabellen interpretiert. Wir betrachten wieder nur den Fall $k = 1$. Zumindest im Prinzip ist dieser Fall sogar ausreichend, da es zu jeder Sprache, für die es eine $LR(k)$-Grammatik und damit einen $LR(k)$-Parser gibt, auch eine $LR(1)$-Grammatik und dementsprechend auch ein $LR(1)$-Parser konstruiert werden kann. Nehmen wir an, die Menge der Zustände des $LR(1)$-Parsers sei Q. Dann lässt sich dieses Treiberprogramm implementieren durch:

```
list⟨state⟩ stack ← [q_0];
terminal buffer ← scan();
state q; nonterminal X; string⟨symbol⟩ α;
while (true) {
    q ← hd(stack);
    switch (action[q, buffer]) {
    case shift :          stack ← goto[q, buffer] :: stack;
                          buffer ← scan();
                          break;
    case reduce(X → α) :  output(X → α);
                          stack ← tl(|α|, stack); q ← hd(stack);
                          stack ← goto[q, X] :: stack;
                          break;
    case accept :         stack ← f :: tl(2, stack);
                          return accept;
    case error :          output("..."); goto err;
    }
```

Die Funktion **list**⟨*state*⟩ tl(**int** n, **list**⟨*state*⟩ s) liefert die Liste s in ihrem zweiten Argument zurück, von der die obersten n Elemente entfernt wurden. Wie bei dem Treiberprogramm für $LL(1)$-Parser wird im Fehlerfall zu einer Marke *err* gesprungen, an der der Code zur Fehlerbehandlung steht.

Wir stellen drei Ansätze vor, wie aus einer kontextfreien Grammatik G einen $LR(1)$-Parser für G konstruiert werden kann. Das allgemeinste Verfahren ist das kanonische $LR(1)$-Verfahren. Für jede $LR(1)$-Grammatik G gibt es einen kanonischen $LR(1)$-Parser. Die Anzahl der Zustände dieses Parsers kann jedoch gegebenenfalls groß sein. Deshalb wurden vereinfachte Verfahren vorgeschlagen, die mit den Zuständen des kanonischen $LR(0)$-Automaten auskommen. Von diesen betrachten wir nur das $SLR(1)$- und das $LALR(1)$-Verfahren.

Unser Treiberprogramm für $LR(1)$-Parser ist auf alle drei Parsertypen anwendbar: es gibt eine Menge Q von Zuständen sowie eine *action*- und eine *goto*-Tabelle, die das Treiberprogramm des Parsers steuern. Die verschiedenen Verfahren unterscheiden sich einerseits in dem zu Grunde liegenden deterministischen endlichen Automaten. Das hat unterschiedliche Mengen von Zuständen zur Folge und entsprechend unterschiedliche *goto*-Tabellen. Andererseits wird bei den verschiedenen Verfahren auch die *action*-Tabelle, d.h. die Vorausschausymbole für unterschiedliche Aktionen auf unterschiedliche Weise berechnet.

Konstruktion eines $LR(1)$-Parsers

Der $LR(1)$-Parser basiert auf dem kanonischen $LR(1)$-Automaten $LR_1(G)$. Seine Zustände sind deshalb Mengen von $LR(1)$-Items. Bei der Konstruktion des kanonischen $LR(1)$-Automaten gehen wir wie bei der Konstruktion des kanonischen $LR(0)$-Automaten vor, nur dass wir anstelle von $LR(0)$-Items $LR(1)$-Items verwenden. Das bedeutet, dass wir beim Abschluss einer Menge q von $LR(1)$-Items unter ε-Übergängen für die neu hinzugefügten Items die Vorausschausymbole berechnen müssen. Diese Menge ist die kleinste Lösung der Gleichung:

$$I = q \cup \{[A \to .\gamma, y] \mid \exists X \to \alpha A \beta \in P : \ [X \to \alpha.A\beta, x] \in I, y \in \mathsf{first}_1(\beta) \odot_1 \{x\}\}$$

Sie wird durch die folgende Funktion berechnet:

```
set⟨item_1⟩ closure(set⟨item_1⟩ q) {
    set⟨item_1⟩ result ← q;
    list⟨item_1⟩ W ← list_of(q);
    nonterminal X; string⟨symbol⟩ α, β; terminal x, y;
    while (W ≠ []) {
        item_1 i ← hd(W); W ← tl(W);
        switch (i) {
        case [_ → _ .Xβ, x] :
                forall (α : (X → α) ∈ P)
                    forall (y ∈ first_1(β) ⊙_1 {x})
                        if ([X → .α, y] ∉ result) {
                            result ← result ∪ {[X → .α, y]};
                            W ← [X → .α, y] :: W;
                        }
        default : break;
        }
    }
    return result;
}
```

wobei V die Menge aller Symbole $V = V_T \cup V_N$ bezeichnet. Der Anfangszustand q_0 von $LR_1(G)$ ist gegeben durch:

$$q_0 = \mathsf{closure}(\{[S' \rightarrow .S, \#]\})$$

Weiterhin benötigen wir eine Hilfsfunktion nextState(), die uns zu einer gegebenen Menge q von LR_1-Items und einem Symbol $X \in V = V_N \cup V_T$ den Nachfolgezustand des kanonischen $LR(1)$-Automaten berechnet. Die entsprechende Funktion bei der Konstruktion von $LR_0(G)$ muss nun um eine Behandlung der Vorausschausymbole erweitert werden:

```
set⟨item_1⟩ nextState(set⟨item_1⟩ q, symbol X) {
    set⟨item_1⟩ q' ← ∅;
    nonterminal A; string⟨symbol⟩ α, β; terminal x;
    forall (A, α, β, x : ([A → α.Xβ, x] ∈ q))
        q' ← q' ∪ {[A → αX.β, x]};
    return closure(q');
    }
```

Die Menge der Zustände und die Übergangsrelation des kanonischen $LR(1)$-Automaten wird analog zum kanonischen $LR(0)$-Automaten berechnet. Der Generator geht von dem Anfangszustand und einer leeren Menge von Übergängen aus und fügt so lange Nachfolgezustände und Übergänge hinzu, bis alle Nachfolgezustände bereits in der Menge der gefundenen Zustände enthalten sind. Die Übergangsfunktion des kanonischen $LR(1)$-Automaten liefert die *goto*-Tabelle des $LR(1)$-Parsers.

Wenden wir uns der Konstruktion der *action*-Tabelle des $LR(1)$-Parsers zu. Enthält ein Zustand q des kanonischen $LR(1)$-Automaten vollständige $LR(1)$-Items $[X \rightarrow \alpha., x], [Y \rightarrow \beta., y]$, liegt kein *reduce-reduce*-Konflikt vor, sofern nur $x \neq y$ ist. Wenn der $LR(1)$-Parser in dem Zustand q ist, wird er die Reduktion auswählen, deren Vorausschausymbol das nächste Eingabesymbol ist.

Enthält der Zustand q gleichzeitig ein vollständiges $LR(1)$-Item $[X \rightarrow \alpha., x]$ und ein $LR(1)$-Item $[Y \rightarrow \beta.a\gamma, y]$, liegt kein *shift-reduce*-Konflikt zwischen ihnen vor, wenn $a \neq x$ ist. Im Zustand q wird

der erzeugte Parser reduzieren, wenn das nächste Eingabesymbol x ist und lesen, wenn es gleich a ist. Deshalb kann die *action*-Tabelle durch die folgende Iteration berechnet werden:

```
forall (state q) {
    forall (terminal x) action[q, x] ← error;
    forall ([X → α.β, x] ∈ q)
        if (β = ε)
            if (X = S' ∧ α = S ∧ x = #) action[q, #] ← accept;
            else action[q, x] ← reduce(X → α);
        else if (β = aβ') action[q, a] ← shift;
}
```

Beispiel 3.4.13 Wir betrachten einige Zustände des kanonischen $LR(1)$-Automaten für die kontextfreie Grammatik G_0. Die Nummerierung der Zustände ist dieselbe wie in Abbildung 3.18. Um eine Menge S von $LR(1)$-Items übersichtlicher darzustellen, werden alle Vorausschauwörter in $LR(1)$-Items aus S mit demselben Kern $[A \to \alpha.\beta]$ zu einer Vorausschaumenge

$$L = \{x \mid [A \to \alpha.\beta, x] \in q\}$$

zusammen gefasst. Die Teilmenge $\{[A \to \alpha.\beta, x] \mid x \in L\}$ repräsentieren wir durch $[A \to \alpha.\beta, L]$. Damit erhalten wir:

$$\begin{aligned}
S'_0 = \;& \mathsf{closure}(\{[S \to .E, \{\#\}]\}) \\
= \{\; & [S \to .E, \{\#\}] \\
& [E \to .E + T, \{\#, +\}], \\
& [E \to .T, \{\#, +\}], \\
& [T \to .T * F, \{\#, +, *\}], \\
& [T \to .F, \{\#, +, *\}], \\
& [F \to .(E), \{\#, +, *\}], \\
& [F \to .\mathsf{Id}, \{\#, +, *\}] \;\}
\end{aligned}$$

$$\begin{aligned}
S'_1 = \;& \mathsf{nextState}(S'_0, E) \\
= \{\; & [S \to E., \{\#\}], \\
& [E \to E. + T, \{\#, +\}] \;\}
\end{aligned}$$

$$\begin{aligned}
S'_2 = \;& \mathsf{nextState}(S'_1, T) \\
= \{\; & [E \to T., \{\#, +\}], \\
& [T \to T. * F, \{\#, +, *\}] \;\}
\end{aligned}$$

$$\begin{aligned}
S'_6 = \;& \mathsf{nextState}(S'_1, +) \\
= \{\; & [E \to E + .T, \{\#, +\}], \\
& [T \to .T * F, \{\#, +, *\}], \\
& [T \to .F, \{\#, +, *\}], \\
& [F \to .(E), \{\#, +, *\}], \\
& [F \to .\mathsf{Id}, \{\#, +, *\}] \;\}
\end{aligned}$$

$$\begin{aligned}
S'_9 = \;& \mathsf{nextState}(S'_6, T)) \\
= \{\; & [E \to E + T., \{\#, +\}], \\
& [T \to T. * F, \{\#, +, *\}] \;\}
\end{aligned}$$

Nach der Erweiterung um Vorausschausymbole enthalten die Zustände S_1, S_2 und S_9, die $LR(0)$-ungeeignet waren, keine Konflikte mehr. In Zustand S'_1 wird bei nächstem Eingabesymbol $+$ gelesen, bei $\#$ reduziert. In Zustand S'_2 wird bei $*$ gelesen, bei $\#$ und $+$ reduziert; ebenso in Zustand S'_9.

Die Tabelle 3.6 zeigt die Zeilen der *action*-Tabelle des kanonischen $LR(1)$-Parsers für die Grammatik G_0, die zu den Zuständen S'_0, S'_1, S'_2, S'_6 und S'_9 gehören. □

$SLR(1)$- und $LALR(1)$-Parser

Die Zustandsmengen von $LR(1)$-Parsern können eventuell sehr groß werden. Deshalb werden oft LR-Analyseverfahren eingesetzt, die nicht ganz so mächtig sind, aber mit weniger Zuständen auskommen.

	id	(	)	$*$	$+$	$\#$
S_0'	s	s				
S_1'					s	*acc*
S_2'				s	$r(3)$	$r(3)$
S_6'	s	s				
S_9'				s	$r(2)$	$r(2)$

Verwendete Nummerierung der Produktionen:

$1 : S \to E$
$2 : E \to E + T$
$3 : E \to T$
$4 : T \to T * F$
$5 : T \to F$
$6 : F \to (E)$
$7 : F \to \mathsf{id}$

Tabelle 3.6. Einige Zeilen der *action*-Tabelle des kanonischen $LR(1)$-Parsers für G_0. s steht für *shift*, $r(i)$ für *reduce* mit Produktion i, *acc* für den Eintrag *accept*. Alle unbesetzten Einträge sind *error*-Einträge.

Zwei häufig verwendete LR-Analyseverfahren sind die $SLR(1)$- (*simple* LR-) und $LALR(1)$- (*lookahead* LR-) Verfahren. Jeder $SLR(1)$-Parser ist ein spezieller $LALR(1)$-Parser, und jede Grammatik, die einen $LALR(1)$-Parser besitzt, ist eine $LR(1)$-Grammatik.

Anstelle von Mengen von $LR(1)$-Items werden bei $SLR(1)$- und $LALR(1)$-Parsern nur Mengen kontextfreier Items als Zustände verwendet. Der Ausgangspunkt bei der Konstruktion von $SLR(1)$- und $LALR(1)$-Parsern ist deshalb der kanonische $LR(0)$-Automat $LR_0(G)$. Die Menge Q der Zustände und die *goto*-Tabelle für diese Parser stimmen mit der Menge der Zustände und der *goto*-Tabelle des entsprechenden $LR(0)$-Parsers überein. Um eventuell auftretende Konflikte in den Zuständen aus Q aufzulösen, wird Vorausschau eingesetzt. Sei $q \in Q$ ein Zustand des kanonischen $LR(0)$-Automaten und $[X \to \alpha.\beta]$ ein Item in q. Dann bezeichnen wir mit $\lambda(q, [X \to \alpha.\beta])$ die Vorausschaumenge, die zu dem Item $[X \to \alpha.\beta]$ in q hinzu gefügt wird. Das $SLR(1)$-Verfahren unterscheidet sich von dem $LALR(1)$-Verfahren in der Definition der Funktion

$$\lambda : Q \times \mathcal{I}_G \to 2^{V_T \cup \{\#\}}$$

Relativ zu einer solchen Funktion λ enthält der Zustand q von $LR_0(G)$ einen *reduce-reduce*-Konflikt, wenn es verschiedene vollständige Items $[X \to \alpha.], [Y \to \beta.] \in q$ gibt mit

$$\lambda(q, [X \to \alpha.]) \cap \lambda(q, [Y \to \beta.]) \neq \emptyset$$

Relativ zu λ enthält q einen *shift-reduce*-Konflikt, wenn es Items $[X \to \alpha.a\beta], [Y \to \gamma.] \in q$ gibt mit $a \in \lambda(q, [Y \to \gamma.])$.

Gibt es in keinem Zustand des kanonischen $LR(0)$-Automaten einen Konflikt, reichen die Voraussschaumengen $\lambda(q, [X \to \alpha.])$ aus, um eine *action*-Tabelle zu konstruieren.

Bei $SLR(1)$-Parsern sind die Vorausschaumengen für Items unabhängig von den Zuständen, in denen sie auftreten: die Vorausschau hängt einzig von der linken Seite der Produktion in dem Item ab:

$$\lambda_S(q, [X \to \alpha.\beta]) = \{a \in V_T \cup \{\#\} \mid S'\# \stackrel{*}{\Longrightarrow} \gamma X a w\} = \mathsf{follow}_1(X)$$

für alle Zustände q mit $[X \to \alpha.] \in q$. Einen Zustand q des kanonischen $LR(0)$-Automaten nennen wir $SLR(1)$-*ungeeignet*, wenn er bzgl. der Funktion λ_S Konflikte enthält. Gibt es keine $SLR(1)$-ungeeigneten Zustände, nennen wir G eine $SLR(1)$-*Grammatik*.

Beispiel 3.4.14 Wir betrachten wieder die Grammatik G_0 aus Beispiel 3.4.1. Ihr kanonischer $LR(0)$-Automat $LR_0(G_0)$ besitzt die ungeeigneten Zustände S_1, S_2 und S_9. Um die Funktion λ_S übersichtlich darzustellen, erweitern die vollständigen Items in den Zuständen durch die follow_1-Mengen ihrer linken Seiten. Weil $\mathsf{follow}_1(S) = \{\#\}$ und $\mathsf{follow}_1(E) = \{\#, +,)\}$ ist, erhalten wir:

$$S_1'' = \{\ [S \to E.,\{\#\}], \quad \text{Konflikt beseitigt,}$$
$$\qquad [E \to E. + T]\} \quad \text{da} \quad + \notin \{\#\}$$

$$S_2'' = \{\ [E \to T.,\{\#,+,)\}], \quad \text{Konflikt beseitigt,}$$
$$\qquad [T \to T. * F]\ \} \quad \text{da} \quad * \notin \{\#,+,)\}$$

$$S_9'' = \{\ [E \to E + T.,\{\#,+,)\}], \quad \text{Konflikt beseitigt,}$$
$$\qquad [T \to T. * F]\ \} \quad \text{da} \quad * \notin \{\#,+,)\}$$

Also ist G_0 eine $SLR(1)$-Grammatik und besitzt einen $SLR(1)$-Parser. □

Die Menge $\mathsf{follow}_1(X)$ fasst alle Symbole zusammen, die auf das Nichtterminal X in Satzformen der Grammatik folgen können. Bei der Konstruktion eines $SLR(1)$-Parsers werden alleine die follow_1-Mengen eingesetzt, um Konflikte zu lösen. In vielen Fällen ist reicht das jedoch nicht aus. Mehr Konflikte lassen sich lösen, wenn der Zustand berücksichtigt wird, in dem das vollständige Item $[X \to \alpha.]$ auftritt. Die *genaueste* Vorausschaumenge, die den Zustand berücksichtigt, ist definiert durch:

$$\lambda_L(q,[X \to \alpha.\beta]) = \{a \in V_T \cup \{\#\} \mid S'\# \underset{rm}{\overset{*}{\Longrightarrow}} \gamma X a w \wedge \Delta_G^*(q_0,\gamma\alpha) = q\}$$

Dabei ist q_0 der Anfangszustand und Δ_G die Übergangsfunktion des kanonischen $LR(0)$-Automaten $LR_0(G)$. In $\lambda_L(q,[X \to \alpha.])$ sind nur die Terminalsymbole enthalten, die auf X in einer Rechtssatzform $\beta X a w$ folgen können, sodass $\beta\alpha$ den kanonischen $LR(0)$-Automaten in den Zustand q überführt.

Wir nennen einen Zustand q des kanonischen $LR(0)$-Automaten $LALR(1)$-*ungeeignet*, wenn er bzgl. der Funktion λ_L Konflikte enthält. Die Grammatik G ist eine $LALR(1)$-Grammatik, wenn der kanonische $LR(0)$-Automat keine $LALR(1)$-ungeeigneten Zustände besitzt.

Zu einer $LALR(1)$-Grammatik gibt es also stets einen $LALR(1)$-Parser. Die Definition der Funktion λ_L ist jedoch nicht konstruktiv, da in ihr Mengen von Rechtssatzformen auftreten, die i.A. unendlich sind. Die Mengen $\lambda_L(q,[A \to \alpha.\beta])$ lassen sich jedoch als kleinste Lösung des folgenden Gleichungssystems charakterisieren:

$$\begin{aligned}
\lambda_L(q_0,[S' \to .S]) &= \{\#\} \\
\lambda_L(q,[A \to \alpha X.\beta]) &= \bigcup\{\lambda_L(p,[A \to \alpha.X\beta]) \mid \Delta_G(p,X) = q\}\ , \qquad X \in (V_T \cup V_N) \\
\lambda_L(q,[A \to .\alpha]) &= \bigcup\{\mathsf{first}_1(\beta) \odot_1 \lambda_L(q,[X \to \gamma.A\beta]) \mid [X \to \gamma.A\beta] \in q'\}
\end{aligned}$$

Das Gleichungssystem gibt an, wie die Mengen von Nachfolgesymbolen von Items in Zuständen zustande kommen. Die erste Gleichung gibt an, dass hinter dem Startsymbol S' nur $\#$ kommen kann. Die zweite Klasse von Gleichungen beschreibt, dass die Folgesymbole eines Items $[A \to \alpha X.\beta]$ in einem Zustand q sich aus den Folgesymbolen hinter dem Item $[A \to \alpha.X\beta]$ in Zuständen p ergeben, aus denen man unter Lesen von X nach q gelangen kann. Die dritte Klasse von Gleichungen formalisiert, dass die Folgesymbole eines Items $[A \to .\alpha]$ in einem Zustand q sich ergeben aus den Folgesymbolen von *Vorkommen* von A in Items aus q hinter dem Punkt, d.h. aus den Mengen $\mathsf{first}_1(\beta) \odot_1 \lambda_L(q,[X \to \gamma.A\beta])$ für Items $[X \to \gamma.A\beta]$ in q.

Auf das Gleichungssystem für die Mengen $\lambda_L(q,[A \to \alpha.\beta])$ über dem endlichen Teilmengenverband $2^{V_T \cup \{\#\}}$ ist das iterative Verfahren zur Berechnung kleinster Lösungen anwendbar. Indem wir berücksichtigen, welche Nichtterminale ε produzieren, können allerdings die Vorkommen der 1-Konkatenation durch Vereinigungen ersetzt werden. Wir erhalten so ein äquivalentes reines Vereinigungsproblem, dessen Lösung sich mit dem schnellen Verfahren aus Abschnitt 3.2.7 lösen lässt.

Beispiel 3.4.15 Die folgende Grammatik aus [ASU86] beschreibt eine Vereinfachung der C-Wertzuweisung:

$$\begin{aligned}
S' &\to S \\
S &\to L = R \mid R \\
L &\to *R \mid \mathsf{Id} \\
R &\to L
\end{aligned}$$

Diese Grammatik ist keine $SLR(1)$-Grammatik, aber eine $LALR(1)$-Grammatik. Die Zustände des kanonischen $LR(0)$-Automaten sind gegeben durch:

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

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

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

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

$$S_4 = \{\ [L \to\ * .R],\ [R \to .L],\ [L \to . * R],\ [L \to .\mathsf{Id}]\ \}$$

$$S_5 = \{\ [L \to \mathsf{Id}.]\ \}$$

$$S_6 = \{\ [S \to L = .R],\ [R \to .L],\ [L \to . * R],\ [L \to .\mathsf{Id}]\ \}$$

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

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

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

Der Zustand S_2 ist der einzige $LR(0)$-ungeeignete Zustand. Es gilt $\mathsf{follow}_1(R) = \{\#, =\}$. Diese Vorausschaumenge für das Item $[R \to L.]$ reicht nicht, um den *shift-reduce*-Konflikt in S_2 zu lösen, da das nächste zu lesende Symbol $=$ in der Vorausschaumenge enthalten ist. Folglich ist die Grammatik keine $SLR(1)$-Grammatik.

Die Grammatik ist jedoch eine $LALR(1)$-Grammatik. Das Übergangsdiagramm ihres $LALR(1)$-Parsers zeigt Abbildung 3.20. Der Übersichtlichkeit halber wurden die Vorausschaumengen $\lambda_L(q, [A \to \alpha.\beta])$ direkt an dem Item $[A \to \alpha.\beta]$ des Zustands q vermerkt. Im Zustand S_2 besitzt das Item $[R \to L.]$ nun die Vorausschaumenge $\{\#\}$. Weil diese Menge das nächste zu lesende Symbol $=$ nicht enthält, ist der Konflikt gelöst. □

3.4.4 Fehlerbehandlung in LR-Parsern

LR-Parser besitzen ebenso wie LL-Parser die Eigenschaft des fortsetzungsfähigen Präfixes. Das bedeutet, dass jedes durch einen LR-Parser fehlerfrei analysierte Präfix der Eingabe zu einem korrekten Eingabewort, einem Satz der Sprache, fortgesetzt werden kann. Trifft ein LR-Parser in einer Konfiguration auf ein Eingabesymbol a mit $action[q, a] = \mathsf{error}$, ist dies die frühestmögliche Situation, in der ein Fehler entdeckt werden kann. Diese Konfiguration nennen wir *Fehlerkonfiguration* und q den *Fehlerzustand* dieser Konfiguration. Auch für LR-Parser gibt es ein Spektrum von Fehlerbehandlungsverfahren:

- Vorwärtsfehlerbehandlung. Modifikationen werden in der restlichen Eingabe, nicht aber auf dem Parserkeller vorgenommen.
- Rückwärtsfehlerbehandlung. Modifikationen werden auch auf dem Parserkeller vorgenommen.

Nehmen wir an, q sei der aktuelle Zustand und a das nächste Symbol in der Eingabe. Als mögliche Korrekturen bieten sich die Aktionen ein verallgemeinertes *shift*(βa) für ein Item $[A \to \alpha.\beta a\gamma]$ aus q, ein *reduce* für unvollständige Items aus q oder *skip* an:

- Die Korrektur *shift*(βa) nimmt an, dass das Teilwort zu β ausgefallen ist. Es kellert deshalb die Zustände, die der Item-Kellerautomat bei Lesen der Symbolfolge β von q aus durchläuft. Anschließend wird das Symbol a gelesen und der entsprechende *shift*-Übergang des Parsers ausgeführt.
- Die Korrektur *reduce*$(A \to \alpha.\beta)$ nimmt an, dass das Teilwort, das zu β gehört, fehlt. Deshalb werden $|\alpha|$ viele Zustände vom Keller entfernt. Sei p der Zustand, der danach oben auf dem Keller liegt. Dann wird der Zustand gekellert, der sich aus p und A gemäß der *goto*-Tabelle ergibt.
- Die Korrektur *skip* fährt mit dem nächsten Symbol a' in der Eingabe fort.

Eine einfache Standardfehlerbehandlung könnte so aussehen. Nehmen wir an, es gebe keinen korrekten Übergang unter a. Enthält der aktuelle Zustand ein Item $[A \to \alpha.\beta a\gamma]$, könnte versucht werden, mit Lesen von a wiederaufzusetzen. Als Korrektur wird dann *shift*(βa) ausgeführt. Tritt das Symbol in keiner rechten Seite eines Items aus q auf, aber als Vorausschau eines nichtvollständigen Item $[A \to \alpha.\beta]$

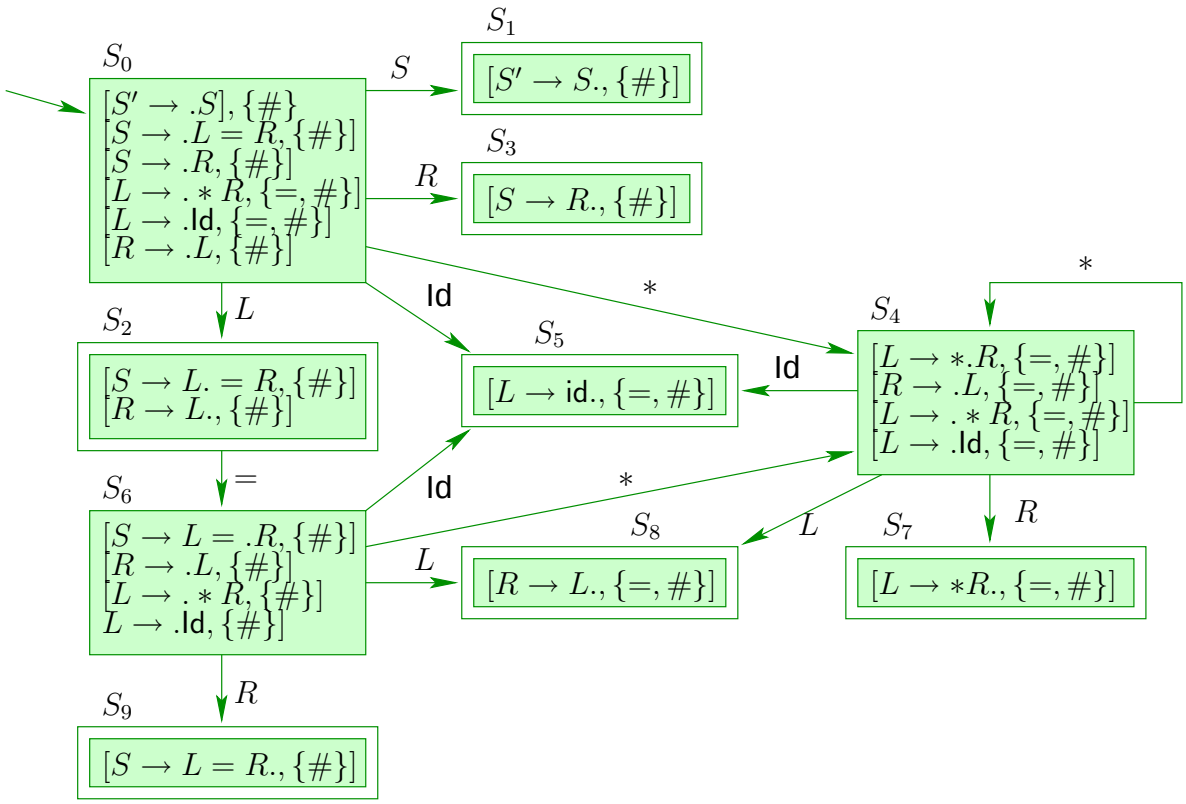


Abb. 3.20. Übergangsdiagramm des $LALR(1)$-Parsers für die Grammatik aus Beispiel 3.4.15.

in q, dann könnte als Korrektur *reduce*$(A \to \alpha.\beta)$ ausgeführt werden. Sind in q mehrere solcher Korrekturen möglich, wird eine *plausible* Korrektur ausgewählt. Plausibel könnte etwa sein, die Operation *shift*(βa) bzw. *reduce*$(A \to \alpha.\beta)$ auszuwählen, bei das fehlende Teilwort β am kürzesten ist. Ist weder eine *shift*- noch eine *reduce*-Korrektur möglich, wird die Korrektur *skip* angewendet.

Beispiel 3.4.16 Betrachten Sie die Grammatik G_0 mit den Produktionen

$$\begin{array}{lll} E \to E + T & T \to T * F & F \to (E) \\ E \to T & T \to F & F \to \mathsf{Id} \end{array}$$

für die wir in Beispiel 3.4.5 den kanonischen $LR(0)$-Automaten konstruiert haben. Als Eingabe wählen wir

$$(\ \mathsf{Id} + \ )$$

Nach Lesen des Präfixes ($\mathsf{Id}\ \ +$ enthält der Keller eines $SLR(1)$-Parsers die Folge der Zustände $S_0S_4S_8S_6$, die dem zuverlässigen Präfix $(\ E\ \ +$ entspricht. Der aktuelle Zustand S_6 besteht aus den Items:

$$\begin{array}{rcl} S_6 & = & \{\, [E \to E + .T], \\ & & \ [T \to .F], \\ & & \ [F \to .\mathsf{Id}], \\ & & \ [F \to .(E)] \quad \} \end{array}$$

Weil wir einen $SLR(1)$-Parser betrachten, sind die Vorausschaumengen der Items in S_6 jeweils gegeben durch die follow_1-Mengen der linken Seiten, d.h.

S_6	λ_S
$[E \to E + .T]$	$+,)$
$[T \to .F]$	$*, +,)$
$[F \to .\mathsf{Id}]$	$*, +,)$
$[F \to .(E)]$	$*, +,)$

Lesen von) im Zustand S_6 liefert den Wert error. Es gibt jedoch in S_6 unvollständige Items mit Vorausschau). Deshalb wird eines dieser Items zur Reduktion verwendet. Dazu kommt etwa das Item $[E \to E + .T]$ in Frage. Die Reduktion liefert als neuen Kellerinhalt $S_0S_4S_8$, da S_8 der Nachfolgezustand von S_4 unter Lesen der linken Seite E ist. In S_8 kann dann ein *shift*-Übergang unter Lesen von) erfolgen. Dies liefert den neuen Zustand S_{11} auf dem Keller. In einer Folge von Reduktionen wird nun der Endzustand f erreicht. □

Diese Fehlerbehandlung ist eine reine Vorwärtsbehandlung. Sie wird in ähnlicher Form etwa von dem Parsergenerator CUP für JAVA angeboten.

Die Ein-Fehler-Hypothese

Im Folgenden stellen wir ein verfeinertes Verfahren vor, das aus den Parsertabellen eine Fehlerbehandlung erzeugt, dabei aber annimmt, dass das Programm *im wesentlichen* syntaktisch korrekt ist und deshalb nur minimal abgeändert werden muss. Das Verfahren geht ebenfalls vorwärts über die Eingabe. Im Fehlerfall versucht es, die Eingabe nach Möglichkeit nur an einer einzigen Stelle abzuändern. Das nennen wir die *Ein-Fehler-Hypothese*. Vorberechnete Informationen wird eingesetzt, um effizient zu entscheiden, wie der Fehler in der Eingabe korrigiert werden sollte.

Eine Konfiguration des LR-Parsers notieren wir als $(\varphi q, a_i \dots a_n)$, wobei φq der Kellerinhalt ist mit aktuellem Zustand q, und die restliche Eingabe $a_i \dots a_n$. Das verfeinerte Verfahren versucht, zu jeder Fehlerkonfiguration $(\varphi q, a_i \dots a_n)$ eine *passende* Konfiguration zu finden, in der eine Fortsetzung der Analyse durch Lesen mindestens eines weiteren Eingabesymbols möglich ist. Eine Konfiguration *passt* zu der Fehlerkonfiguration, wenn sie durch möglichst wenig Veränderungen aus der Fehlerkonfiguration hervorgeht. Mit der Annahme der *ein-Fehler-Hypothese* schränken wir die zugelassenen Veränderungen drastisch ein. Die ein-Fehler-Hypothese besagt, dass der Fehler an der gegebenen Stelle durch *ein* fehlendes, *ein* überflüssiges oder *ein* falsches Symbol an der Fehlerstelle verursacht wurde. Der Fehlerbehandlungsalgorithmus verfügt deshalb über eine Operation für das Einsetzen, eine Operation für das Löschen und eine Operation für das Ersetzen *eines* Symbols.

Sei $(\varphi q, a_i \dots a_n)$ eine Fehlerkonfiguration. Das Ziel der Fehlerkorrektur mit einer der drei Operationen lässt sich wie folgt beschreiben:

Löschen: Finde Kellerinhalte $\varphi' p$ mit

$$(\varphi q, a_{i+1} \dots a_n) \vdash^* (\varphi' p, a_{i+1} \dots a_n) \quad \text{und} \quad action[p, a_{i+1}] = shift$$

Ersetzen: Finde ein Symbol a und Kellerinhalte $\varphi' p$ mit

$$(\varphi q, a a_{i+1} \dots a_n) \vdash^* (\varphi' p, a_{i+1} \dots a_n) \quad \text{und} \quad action[p, a_{i+1}] = shift$$

Einfügen: Finde ein Symbol a und Kellerinhalte $\varphi' p$ mit

$$(\varphi q, a a_i \dots a_n) \vdash^* (\varphi' p, a_i \dots a_n) \quad \text{und} \quad action[p, a_i] = shift$$

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. Eine wichtige Eigenschaft der drei Operationen ist, dass sie die Terminierung des Fehlerbehandlungsverfahren garantieren: jeder der drei Schritte stellt im Erfolgsfall 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.

Ein naives Verfahren wird die verschiedenen Möglichkeiten einer Fehlerkorrektur dynamisch, d.h. während des Parserlaufs durchsuchen, bis eine geeignete Korrektur gefunden ist. Das Überprüfen einer Möglichkeit verlangt eventuell, eine Reihe Reduktionen durchzuführen, gefolgt von einem Test, ob man ein Symbol lesen kann. Bei Misserfolg ist dann die Fehlerkonfiguration wiederherzustellen und die nächste Möglichkeit auszprobieren. Die Suche nach der *richtigen* Abänderung eines Symbols kann damit sehr teuer sein. Deshalb interessieren wir uns für *Vorberechnungen*, die man bereits zur Generierungszeit des Parsers durchführen kann, um Sackgassen bei der Fehlerkorrektur schneller zu erkennen. 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 3.21 (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.

Eine Vorberechnung ermöglicht es, viele Symbole a von vornherein auszuschließen, weil es keine Teilfolgen für (1) oder (3) geben kann. Dazu berechnen wird für jeden Zustand q und jedes $a \in V_T$ die Menge *Succ*(q, a) möglicher *Reduktionsnachfolger* von q unter a berechnet. Die Menge *Succ*(q, a) enthält den Zustand q zusammen mit allen Zuständen, in die der Parser aus q durch Reduktionen unter Vorausschau a kommen kann. Die Menge *Succ*(q, a) ist die kleinste Menge Q' mit den folgenden Eigenschaften:

- $q \in Q'$;
- sei $q' \in Q'$ und enthalte q' ein vollständiges Item zu einer Produktion $A \to X_1 \dots X_k$. Dann ist auch *goto*$[p, A] \in Q'$ für jeden Zustand p mit

$$\textit{goto}[\dots \textit{goto}[p, X_1] \dots, X_k] = q$$

Mit Hilfe der Menge *Succ*(q, a) lässt sich die Menge $Sh(q, a)$ aller Zustände definieren, die von Reduktionsnachfolgern von q unter a durch einen *shift*-Übergang für a erreicht werden kann:

$$Sh(q, a) = \{\textit{goto}[q', a] \mid q' \in \textit{Succ}(q, a)\}$$

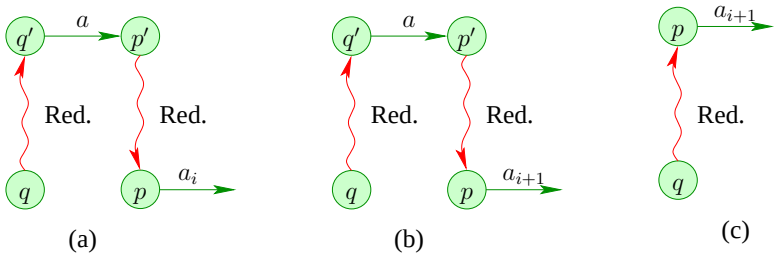


Abb. 3.21. Schließen der Brücke bei der Fehlerkorrektur, (a) beim Einfügen, (b) beim Ersetzen, (c) beim Löschen eines Symbols.

Mit Hilfe der Mengen $Sh(q', a')$ für alle Zustände q' und Terminalsymbole a wird nun die Menge aller Zustände definiert, die sich aus den Zuständen in $Sh(q, a)$ durch Reduktionen mit Vorausschau a_i ergeben, gefolgt von einer Leseoperation für a_i:

$$Sh(q, a, a_i) = \bigcup \{Sh(q', a_i) \mid q' \in Sh(q, a)\}$$

Eine Korrektur mittels Einfügen eines Symbols a ist vielversprechend, wenn die Menge $Sh(q, a, a_i)$ nichtleer ist. Das Terminalsymbol a ist ein Kandidat zum Schließen der Brücke in der Fehlerkonfiguration. Soll die Vorberechnung weitergetrieben werden, kann für den Zustand q und das Terminalsymbol a_i die Menge

$$\mathit{Bridge}(q, a_i) = \{a \in V_T \mid Sh(q, a, a_i) \neq \emptyset\}$$

aller Kandidaten berechnet werden, die zur Fehlerkorrektur durch Einfügen in Betracht kommen.

Beispiel 3.4.17 Wir betrachten die Grammatik aus Beispiel 3.4.15 mit dem $LALR(1)$-Parser aus Abbildung 3.20. Die Reduktionsnachfolger $Succ(q, a)$ von q unter a, die Mengen $Sh(q, a)$ und die Mengen $Bridge(q, a)$ für $a \in \{=, *, \mathsf{Id}\}$ ergeben sich zu:

$\underline{Succ(q,a)}$:

q	$=$	$*$	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

$\underline{Sh(q,a)}$:

q	$=$	$*$	Id
S_0		S_4	S_5
S_1			
S_2	S_6		
S_3			
S_4		S_4	S_5
S_5	S_6		
S_6		S_4	S_5
S_7	S_6		
S_8	S_6		
S_9			

$\underline{Bridge(q,a)}$:

q	$=$	$*$	Id
S_0	Id	$*$	$*$
S_1			
S_2		$=$	$=$
S_3			
S_4	Id	$*$	$*$
S_5		$=$	$=$
S_6	Id	$*$	$*$
S_7		$=$	$=$
S_8			
S_9			

Betrachten wir, welche Fehlerkorrekturen unsere Vorberechnung vorschlagen würde.

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

Ein Beispiel für eine Löschkorrektur ist:

Eingabe	Fehlerkonfiguration	Brücke	Korrektur
$\mathsf{Id}\ \mathsf{Id} = \mathsf{Id}\ \#$	$(S_0S_5, \mathsf{Id} = \mathsf{Id}\ \#)$	$\mathit{Sh}(S_5, =) \neq \emptyset$	Löschen von Id

□

Auf der Basis der vorberechneten Mengen kann ebenfalls effizient eine ein-Symbol-*Ersetzungskorrektur* versucht werden. Der einzige Unterschied ist, dass Symbole aus $\mathit{Bridge}(q, a_{i+1})$ betrachtet werden müssen (siehe Abbildung 3.21 (b)).

Analoge Überlegungen liefern uns einen Test, ob eine ein-Symbol-*Löschkorrektur* vernünftig ist. Das Löschen eines Symbols a wird zur Fehlerbehandlung in Betracht gezogen, wenn die Menge $\mathit{Sh}(q, a_{i+1})$ nichtleer ist (siehe Abbildung 3.21 (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, lässt sich der Test durch einfaches Nachschlagen implementieren.

Einen Sonderfall haben wir bisher ignoriert, nämlich Korrekturen bei erschöpfter Eingabe. Da das Endesymbol # nie gelesen wird, sind Löschaktionen und oder Ersetzungsaktionen nicht möglich. Hier bleiben nur Einsetzungskorrekturen übrig.

Die Einsetzung eines Terminalsymbols 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 für jeden Zustand q die Menge $\mathit{Acc}(q)$ vorberechnet werden, die alle Terminalsymbole enthält, die dazu in Frage kommen.

Sei $(\varphi q, a_i \dots a_n)$ wieder die Fehlerkonfiguration. Eine optimierte Fehlerbehandlung kann nun so beschrieben werden:

Versuch zu löschen:	Ist *Sh*$(q, a_{i+1}) \neq \emptyset$, dann *teste*$(\varphi q, a_{i+1} \dots a_n)$;
Versuch zu ersetzen:	Gibt es ein $a \in$ *Bridge*(q, a_{i+1}), dann *teste*$(\varphi q, a a_{i+1} \dots a_n)$;
Versuch einzusetzen:	Gibt es ein $a \in$ *Bridge*(q, a_i), dann *teste*$(\varphi q, a a_i a_{i+1} \dots a_n)$.

In der Prozedur *teste* wird das Parsen nach dem Korrekturversuch fortgesetzt. Wird der Rest der Eingabe erfolgreich abgearbeitet, ist der Korrekturversuch gelungen. Schlägt der Versuch, erfolgreich den Rest der Eingabe zu verarbeiten, jedoch fehl, kann der Parser einen weiteren Fehler annehmen und erneut eine Fehlerkorrektur versuchen.

In einer ambitionierteren Implementierung wird der Parser nach Scheitern eines Korrekturversuchs an die Fehlerstelle zurückkehren und einen weiteren Korrekturvorschlag testen. Schlagen sämtliche Korrekturvorschläge fehl, wird ein besonders *erfolgreicher* Korrekturversuch ausgewählt und in der dabei erreichten Konfiguration erneut eine Fehlerkorrektur versucht. Als Maß für den Erfolg einer Korrektur könnte etwa die Länge der danach fehlerfrei konsumierten Eingabe dienen. Beachten Sie, dass sich der Parser in diesem Fall nur jeweils einen erfolgreichsten bisher unternommenen Korrekturversuch merken muss.

Die Vorwärtsbewegung

Unsere Vorberechnungen haben die Anzahl der zu überprüfenden Korrekturvorschläge deutlich eingeschränkt. Nichtsdestoweniger kann es mehrere Möglichkeiten zur Korrektur geben. In dem Fall kann es sich lohnen, eine Teilberechnung heraus zu faktorisieren, die von sämtlichen Korrekturversuchen durchgeführt wird. Eine solche Teilberechnung ist eine *Vorwärtsbewegung*, die allen Versuchen gemeinsam ist. Um eine solche Vorwärtsbewegung zu identifizieren, starten wir nicht mit einem spezifischen Kellerinhalt, sondern betrachten sämtliche Zustände, in denen das Zeichen a_{i+1} gelesen werden kann. Dann wird versucht, ein möglichst langes Präfix von $a_{i+1} \dots a_n$ zu reduzieren. Die Konfigurationen bestehen dabei aus Folgen von *Mengen* von Zuständen Q in einem Fehlerkeller und der jeweiligen 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 entweder alle *shift* liefern oder alle *reduce*$(X \to \alpha)$ mit der gleichen Produktion $X \to \alpha$ und der Fehlerkeller nicht kürzer als $|\alpha|$ ist. Die Vorwärtsbewegung stoppt,

- wenn für alle $q \in Q$ *action*$[q, a] =$ error gilt: dann liegt ein *zweiter* Fehler vor;
- wenn die *action*-Tabelle für Q und a mehr als eine Aktion angibt;
- wenn sie die einzige Aktion *accept* angibt: dann ist das Parsen beendet; oder
- wenn sie eine Reduktion verlangt, wobei die Länge der rechten Seite größer als die Tiefe des Fehlerkellers ist: das würde zu einer *Reduktion über die Fehlerstelle* hinaus führen.

Als Ergebnis gibt die Vorwärtsbewegung das Wort γ zurück, zu dem sie das bis dahin gelesene Präfix $a_{i+1} \dots a_k$ reduziert hat, gefolgt von der restlichen Eingabe $a_{k+1} \dots a_n$. Beachten Sie, dass das Wort γ oft sehr viel kürzer sein wird als das Teilwort $a_{i+1} \dots a_k$. In der Eingabe für die Aufrufe der Prozedur *teste* kann dann das Teilwort $a_{i+1} \dots a_n$ durch $\gamma\ a_{k+1} \dots a_n$ ersetzt werden, wobei der Parser ein Nichtterminal A in der Eingabe stets wie ein *shift* des Symbols A behandelt.

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

Kanonische $LR(1)$-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.