Inhalt

Lehrziele

Nach dem Durcharbeiten dieser Kurseinheit sollten Sie

- das allgemeine Prinzip der Bottom-up-Analyse erklären und an einem Beispiel vorführen können,
- das allgemeine Prinzip der Operator-Vorrang-Analyse erklären und an einem Beispiel vorführen können,
- die Begriffe LR(0)-Element, Abschluss einer Menge von LR(0)-Elementen, kanonische LR(0)-Kollektion und SLR-Parser erklären können,
- für eine SLR(1)-Grammatik die Analysetabelle aufstellen und erläutern können,
- die Begriffe kanonischer LR-Parser, LR(1)-Element und LALR(1)-Parser erklären können,
- für eine LR(1)-Grammatik die Analysetabelle aufstellen und erläutern können,
- für einfache Problemstellungen Yacc-Spezifikationen zur Implementierung eines LALR(1)-Parsers schreiben können.

3.3 Bottom-up-Analyse

Thema dieses Abschnitts ist die zweite Strategie für die Syntaxanalyse, bei der man *bottom-up* vorgeht, den Syntaxbaum also von den Blättern aus nach oben aufbaut. Bei der Top-down-Analyse werden jeweils *Nichtterminale* im Ableitungsbaum betrachtet und Regeln benutzt, bei denen dieses Nichtterminal auf der *linken Seite* steht, um das Nichtterminal zu expandieren; dabei werden die Symbole der rechten Seite als Söhne an den Knoten des Nichtterminals angehängt. Bei der Bottom-up-Analyse werden jeweils *rechte Seiten* von Produktionen entdeckt, wobei zu den Symbolen der rechten Seite bereits separate Ableitungsbäume erkannt sind; mit Hilfe der Produktion wird dann das Nichtterminal auf der linken Seite als Vater (bzw. neue Wurzel) mit den Bäumen der Söhne verbunden.

Wir betrachten im folgenden Abschnitt 3.3.1 zunächst das Prinzip der Bottom-up-Analyse genauer. In Abschnitt 3.3.2 untersuchen wir die am wenigsten mächtige, aber dafür am einfachsten zu verstehende Technik der Bottom-up-Analyse, die sogenannte Operator-Vorrangmethode. Die mächtigsten effizienten Verfahren zur Syntaxanalyse überhaupt sind *LR-Parser*, die wir in Abschnitt 3.3.3 besprechen. LR-Parser sind allerdings so komplex, dass sie normalerweise „von Hand“ nicht mehr vernünftig konstruierbar sind. Deshalb spielen Werkzeuge eine besondere Rolle. In Abschnitt 3.3.4 stellen wir das sehr verbreitete Werkzeug *Yacc* vor, das aus einer Grammatikspezifikation automatisch einen LR-Parser generiert.

3.3.1 Das Prinzip der Bottom-up-Analyse

Wie gerade skizziert, werden bei der Bottom-up-Analyse Syntaxbäume von den Blättern aus nach oben zusammengebaut; dabei werden *Rechtsableitungen in umgekehrter Reihenfolge* erkannt. Wir betrachten als Beispiel wieder eine einfache Grammatik für arithmetische Ausdrücke mit den Produktionen (wie in Abschnitt 3.1):

$E \rightarrow E + T$
$E \rightarrow T$
$T \rightarrow T * F$
$T \rightarrow F$
$F \rightarrow (E)$
$F \rightarrow \mathbf{id}$

Gegeben sei eine Tokenfolge **id** + **id** * **id**.

Ein Bottom-up-Parser liest Symbole der Eingabefolge; sobald eine rechte Seite einer Produktion komplett vorliegt, wird sie durch das Nichtterminal der linken Seite ersetzt (das ist allerdings nicht die ganze Wahrheit, wie wir gleich sehen werden). Wir notieren links, was bereits gelesen und verarbeitet ist, rechts die noch übrige Restfolge.

id + **id** * **id**

Das Token **id** ist eine komplette rechte Seite, wir ersetzen durch *F*. Man sagt, die rechte Seite wird zu *F reduziert*.

F	+ **id** * **id**
T	+ **id** * **id**
E	+ **id** * **id**

Mit *E* kann man nichts weiter anfangen, wir müssen ein weiteres Zeichen hinzunehmen.

E +	**id** * **id**
E + **id**	* **id**
E + *F*	* **id**
E + *T*	* **id**

An dieser Stelle müssten wir reduzieren, da wieder eine komplette rechte Seite vorliegt. Tatsächlich ist das Kriterium „vollständige rechte Seite vorhanden“ aber nicht ausreichend. Wenn wir hier *E* + *T* zu *E* reduzieren, wird es nicht gelingen, die gesamte Eingabefolge zu reduzieren, da es keine Produktion gibt, in der „*E* *“ vorkommt. Das zentrale Problem in der Bottom-up-Analyse besteht darin, zu entscheiden, ob bei Vorliegen einer kompletten rechten Seite reduziert werden soll oder ob zunächst noch weitere Zeichen hinzugenommen werden sollen. Hier wäre letzteres die richtige Entscheidung:

E + *T* *	**id**
E + *T* * **id**	
E + *T* * *F*	
E + *T*	
E	

Wenn wir diese Folge in umgekehrter Reihenfolge hinschreiben, sehen wir, dass eine Rechtsableitung erkannt worden ist:

$$\underline{E} \Rightarrow E + \underline{T} \Rightarrow E + T * \underline{F} \Rightarrow E + \underline{T} * \mathbf{id} \Rightarrow E + \underline{F} * \mathbf{id} \Rightarrow \underline{E} + \mathbf{id} * \mathbf{id}$$

$$\Rightarrow \underline{T} + \mathbf{id} * \mathbf{id} \Rightarrow \underline{F} + \mathbf{id} * \mathbf{id} \Rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id}$$

Zeilen, in denen nicht reduziert, sondern nur ein Zeichen gelesen wurde, erscheinen natürlich nicht in der Ableitung. Warum wird eine *Rechtsableitung* erkannt? Weil bei jedem Reduktionsschritt das *Ende* der bereits gelesenen Folge durch ein Nichtterminal ersetzt wird – in der Ableitung ist dieses dann das Nichtterminal, das ersetzt wird – und rechts davon nur die noch nicht gelesenen Zeichen, also Terminale, stehen.

Wie wir gesehen haben, besteht das wesentliche Problem darin, zu entscheiden, ob eine vollständige rechte Seite reduziert werden soll oder nicht. Die rechte Seite β sollte reduziert werden, wenn sie in einer Rechtsableitung vorkommt und somit der Reduktionsprozess erfolgreich zu Ende geführt werden kann. Ob eine rechte Seite diese Rolle spielt, wird präzisiert im Begriff des *Handle*:

Definition 3.16: Sei G eine kontextfreie Grammatik und sei

$$S \underset{r}{\Rightarrow}^{*} \alpha A w \underset{r}{\Rightarrow} \alpha\beta w$$

eine Rechtsableitung in G. Dann heißt β ein *Handle*[1] der Rechtssatzform $\alpha\beta w$. □

Die Definition spricht von „einem" Handle, weil es mehrere geben könnte, wenn die Grammatik mehrdeutig ist, wenn also die Rechtssatzform $\alpha\beta w$ auf verschiedene Arten abgeleitet werden kann. Bei einer eindeutigen Grammatik hat jede Rechtssatzform genau ein Handle.

Innerhalb einer Rechtsableitung sieht man Handles, wenn man nicht die Nichtterminale unterstreicht, die ersetzt werden, sondern die rechten Seiten, durch die ersetzt wird.

$$E \Rightarrow \underline{E + T} \Rightarrow E + \underline{T * F} \Rightarrow E + T * \underline{\mathbf{id}} \Rightarrow E + \underline{F} * \mathbf{id} \Rightarrow E + \underline{\mathbf{id}} * \mathbf{id}$$

$$\Rightarrow \underline{T} + \mathbf{id} * \mathbf{id} \Rightarrow \underline{F} + \mathbf{id} * \mathbf{id} \Rightarrow \underline{\mathbf{id}} + \mathbf{id} * \mathbf{id}$$

Bevor wir uns der Frage zuwenden, wie man Handles findet, überlegen wir, wie man eine Bottom-up-Analyse implementieren kann. Das obige Beispiel hat schon deutlich gemacht, dass eine sehr natürliche Implementierung mit Hilfe eines *Stacks* möglich ist. Eingabesymbole werden jeweils auf den Stack gelegt. Sobald am oberen Ende des Stacks ein Handle β einer Produktion $A \to \beta$ erscheint, wird reduziert, das heißt, die Symbole von β werden vom Stack entfernt, und A wird an ihrer Stelle auf den Stack gelegt. Genau genommen führt ein solcher Parser 4 Arten von Aktionen durch:

1. *Shift*. Entnimm das nächste Symbol der Eingabefolge und lege es auf den Stack.
2. *Reduce*. Ein Handle β einer Produktion $A \to \beta$ bildet das obere Ende des Stacks. Ersetze β auf dem Stack durch A.
3. *Accept*. Auf dem Stack liegt nur noch das Startsymbol; die Eingabefolge ist leer. Akzeptiere die Eingabefolge.
4. *Error*. Entscheide, dass ein Syntaxfehler vorliegt.

Da *shift* und *reduce* die wesentlichen Aktionen sind, wird ein solcher Parser auch *Shift-reduce*-Parser genannt.

Wir spielen unser obiges Beispiel noch einmal etwas deutlicher unter Verwendung eines Stacks durch. Das Symbol „$" wird wieder benutzt, um das untere Ende des Stacks bzw. das Ende der Eingabefolge zu kennzeichnen. Handles sind unterstrichen.

---

[1] In deutschen Lehrbüchern findet man verschiedene Übersetzungen für *Handle*, u. a. „Griff", „Henkel" oder „Ansatz". Wegen der fehlenden Einheitlichkeit bleiben wir lieber beim englischen Begriff.

Stack	*Eingabe*	*Aktion*
\$	**id** + **id** * **id** \$	*shift*
\$ **id**	+ **id** * **id** \$	*reduce* mit $F \rightarrow$ **id**
\$ *F*	+ **id** * **id** \$	*reduce* mit $T \rightarrow F$
\$ *T*	+ **id** * **id** \$	*reduce* mit $E \rightarrow T$
\$ *E*	+ **id** * **id** \$	*shift*
\$ *E* +	**id** * **id** \$	*shift*
\$ *E* + **id**	* **id** \$	*reduce* mit $F \rightarrow$ **id**
\$ *E* + *F*	* **id** \$	*reduce* mit $T \rightarrow F$
\$ *E* + *T*	* **id** \$	*shift*
\$ *E* + *T* *	**id** \$	*shift*
\$ *E* + *T* * **id**	\$	*reduce* mit $F \rightarrow$ **id**
\$ *E* + *T* **F*	\$	*reduce* mit $T \rightarrow T$ **F*
\$ *E* + *T*	\$	*reduce* mit $E \rightarrow E + T$
\$ *E*	\$	*accept*

Die Folgen von Symbolen, die während der Analyse auf dem Stack erscheinen können, sind für spätere Betrachtungen von Interesse, sie heißen *geeignete Präfixe* („viable[2] prefixes“).

Definition 3.17: Sei G eine kontextfreie Grammatik und sei

$$S \underset{r}{\Rightarrow} {}^{*} \alpha A w \underset{r}{\Rightarrow} \alpha\beta w$$

eine Rechtsableitung in G. Jedes Anfangsstück der Folge $\alpha\beta$ heißt *geeignetes Präfix* von G. □

Ein geeignetes Präfix ist also ein Anfangsstück einer Rechtssatzform, das nicht über das Handle hinausgeht. Solange auf dem Stack ein geeignetes Präfix steht, können in der Eingabe noch Terminalsymbole folgen, mit denen zusammen eine Rechtssatzform entsteht. Das bedeutet, dass bis dahin noch kein Syntaxfehler vorliegt.

Die (ggf. explizite) Konstruktion des Ableitungsbaums im Rahmen des Shift-Reduce-Parsens ist ebenfalls leicht mit Hilfe des Stacks möglich. Dann werden auf dem Stack Teilbäume verwaltet, deren Wurzel jeweils das (nach bisheriger Auffassung) auf dem Stack gespeicherte Symbol ist. Bei einem Reduktionsschritt werden Symbole der rechten Seite β als Söhne an das Nichtterminal A angehängt und vom Stack entfernt; der neue Teilbaum mit Wurzel A wird auf den Stack gelegt. Für unser Beispiel sieht das so aus (horizontal sind Stackplätze für Teilbäume angeordnet):

---

[2] Auch hier gibt es verschiedene deutsche Übersetzungen, z. B. „zuverlässiges“ oder „lebensfähiges“ Präfix. Diesmal erfinden wir selbst noch eine neue. Am besten merkt man sich auch den englischen Begriff.

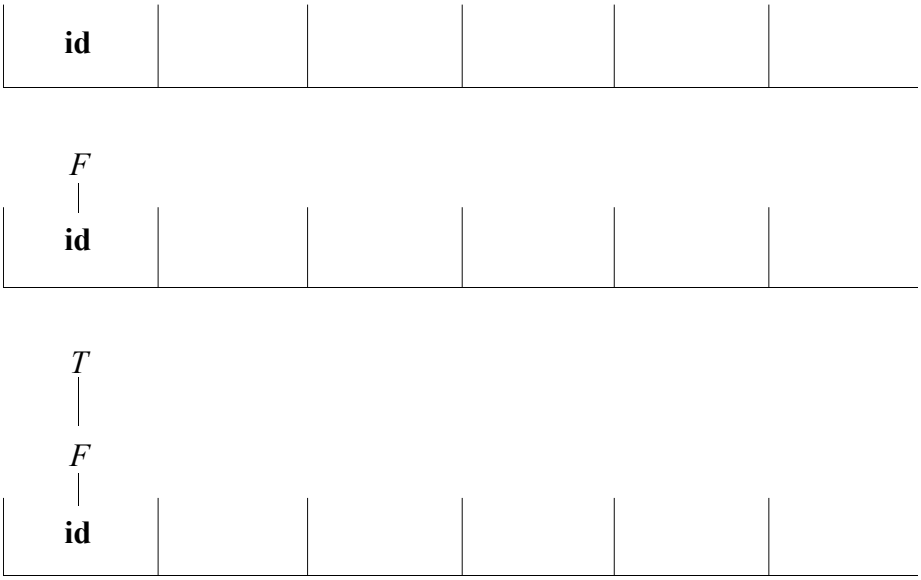

Abb. 3.22. Teilbäume auf dem Stack beim Shift-Reduce-Parsen (1)

Nach zwei weiteren Schritten:

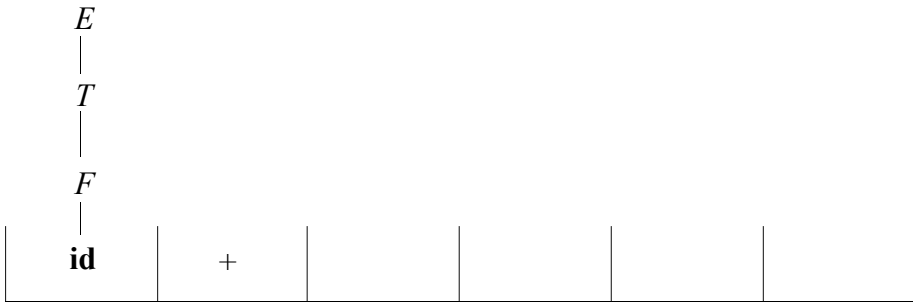

Abb. 3.23. Teilbäume auf dem Stack beim Shift-Reduce-Parsen (2)

Dies ist die Situation nach dem zweiten Shift-Schritt. Nach einigen weiteren Schritten erhalten wir folgende Situation:

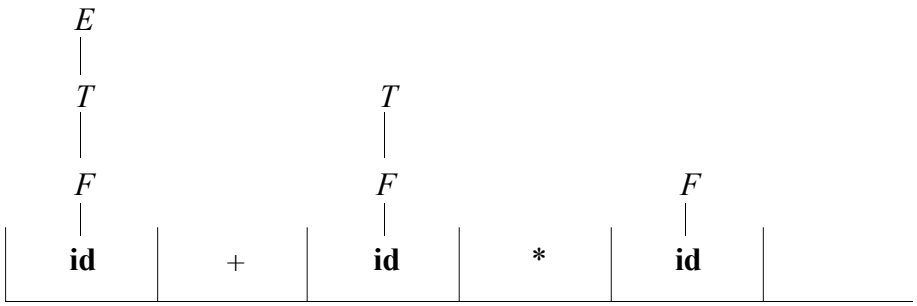

Abb. 3.24. Teilbäume auf dem Stack beim Shift-Reduce-Parsen (3)

Es folgt eine Reduktion mit $T \rightarrow T * F$. Beachten Sie, dass der entstehende Teilbaum insgesamt nur noch einen Stackplatz belegt.

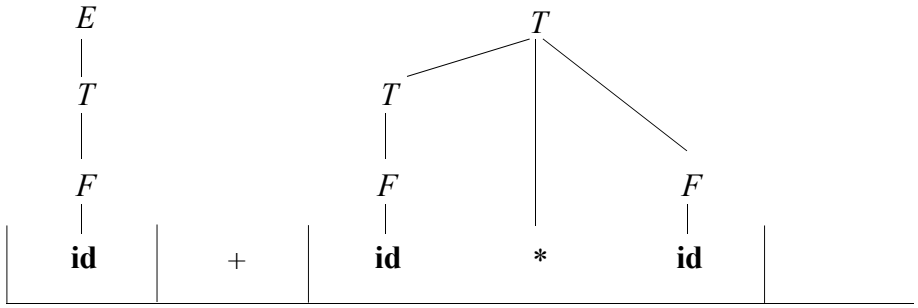

Abb. 3.25. Teilbäume auf dem Stack beim Shift-Reduce-Parsen (4)

Schließlich folgt die letzte Reduktion, der gesamte Baum steht auf der untersten Stackposition.

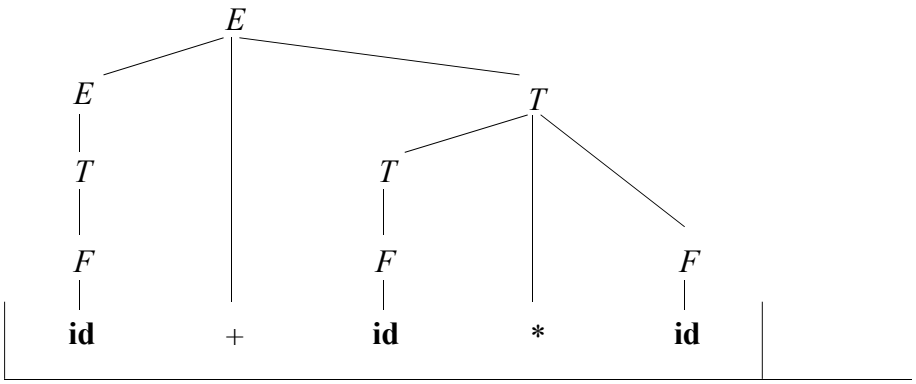

Abb. 3.26. Teilbäume auf dem Stack beim Shift-Reduce-Parsen (5)

Diesen Ableitungsbaum haben wir schon mal in Abb. 3.4 gesehen.

Die spannende Frage ist nun, wie der Parser Handles erkennen kann. Er müßte erstens feststellen, dass nach einem Shift-Schritt ein Handle auf dem Stack liegt, und zweitens die Anfangsposition (das erste Symbol) des Handles erkennen. Eine erste einfache Methode dazu betrachten wir im folgenden Abschnitt.

3.3.2 Operator-Vorranganalyse

Die im folgenden beschriebene Methode ist ursprünglich speziell für die Syntaxanalyse arithmetischer Ausdrücke entwickelt worden. Es ist das schwächste Bottom-up-Verfahren insofern, als nur eine recht eingeschränkte Klasse von Grammatiken damit behandelt werden kann. Dafür ist es einfach, kann relativ leicht von Hand implementiert werden und bietet uns einen guten Einstieg in das Shift-Reduce-Parsen.

Die Grundidee besteht darin, drei Relationen zu definieren, die zwischen aufeinanderfolgenden Symbolen auf dem Stack bestehen können, notiert als $<\cdot$, $\doteq$ und $\cdot>$. Das Ziel ist ja, in einer Rechtssatzform $\alpha\beta w$ das Handle β zu erkennen. Sei $\alpha = a_1 \ldots a_n$, $\beta = b_1 \ldots b_m$ und $w = w_1 \ldots w_k$. Dann sollten folgende Relationen zwischen den einzelnen Symbolen bestehen:

Stack		Eingabe
$\ldots a_n <\cdot b_1 \doteq b_2 \doteq \ldots \doteq b_m$	$\cdot>$	$w_1 \ldots$

Die spitzen Klammern zeigen also gerade die Grenzen des Handles an, die Relation $\doteq$ gilt zwischen Symbolen innerhalb des Handles. Der Nutzen ist offensichtlich: Solange zwischen dem obersten Stacksymbol und dem nächsten Eingabesymbol die Beziehung $<\cdot$ oder $\doteq$ gilt, ist *shift* die richtige Aktion. Sobald die Beziehung $\cdot>$ auftritt, ist das Ende eines Handles erreicht und *reduce* ist auszuführen; darüber hinaus

findet man das linke Ende des Handles, indem man über alle Paare von Symbolen mit Beziehung $\doteq$ hinweg zurückgeht, bis die Beziehung $<\cdot$ auftritt.

Tatsächlich definiert man die drei Relationen nur zwischen *Terminalsymbolen*. Bei diesem Verfahren spielen Nichtterminale praktisch keine Rolle; die Analyse wird vollständig durch die Relationen gesteuert. Bei der Betrachtung aufeinanderfolgender Symbole auf dem Stack beschränkt man sich daher auf Terminale und ignoriert dazwischenliegende Nichtterminale. Die drei Relationen heißen *Operator-Vorrangrelationen*, wohl deshalb, weil man ursprünglich Beziehungen zwischen arithmetischen Operatoren wie +, -, *, / usw. festlegen wollte. Zwischen arithmetischen Operatoren gelten ja gewisse Vorrangregeln: Multiplikation und Division binden stärker als Addition und Subtraktion usw. Zwischen + und * definiert man daher die Beziehungen

$$+ \;<\cdot\; *$$

$$* \;\cdot\!> \; +$$

Das führt etwa dazu, dass in einer Folge

$$E + E * E + E$$

Relationen gelten (Nichtterminale werden ignoriert)

$$E + <\cdot\; E * E \;\cdot\!> + E$$

so dass, wie gewünscht, der Teilausdruck $E * E$ zuerst reduziert wird.

Wir betrachten als Beispiel wieder unsere Grammatik:

$$E \to E + T$$
$$E \to T$$
$$T \to T * F$$
$$T \to F$$
$$F \to (E)$$
$$F \to \mathbf{id}$$

Da die Nichtterminale in der Analyse keine Rolle spielen, können wir mit einem einzigen auskommen und die Grammatik vereinfachen:

$$E \to E + E$$
$$E \to E * E$$
$$E \to (E)$$
$$E \to \mathbf{id}$$

Es müssen Vorrangrelationen festgelegt werden zwischen allen Terminalsymbolen, hier also den Symbolen {+, *, (,), **id**} sowie dem speziellen Symbol $, das in der Analyse benutzt wird. Die drei Vorrangrelationen müssen disjunkt sein, d. h., es darf zwischen je zwei Symbolen nur eine der drei Beziehungen bestehen. Daher kann man die Relationen gut in einer einzigen Tabelle darstellen. Wir betrachten zunächst

den Ablauf des Parsens bei vorgegebener Tabelle; später werden wir sehen, wie man Tabellen konstruiert. Hier ist die Tabelle für unser Beispiel:

		Eingabe					
		+	*	(	)	id	$
Stack	+	·>	<·	<·	·>	<·	·>
	*	·>	·>	<·	·>	<·	·>
	(	<·	<·	<·	≐	<·	
	)	·>	·>		·>		·>
	id	·>	·>		·>		·>
	$	<·	<·	<·		<·	

Dabei sind Leereinträge als Fehler zu interpretieren. Eine Ausnahme bildet der fehlende Eintrag für das Paar ($, $); an dieser Stelle akzeptiert der Parser.

Wir analysieren wieder die Folge **id** + **id** * **id**:

Stack		*Eingabe*	*Aktion*
$	<·	**id** + **id** * **id** $	*shift*

Wir sehen die Beziehung zwischen $ und **id** in der Tabelle nach und finden <·. Also ist *shift* die richtige Aktion. Auf dem Stack tragen wir die Beziehung mit ein.

$ <· **id**	·>	+ **id** * **id** $	*reduce* mit $E \rightarrow$ **id**
$ E	<·	+ **id** * **id** $	*shift*

Hier wird die Beziehung zwischen $ und + nachgesehen; E wird ignoriert.

$ E <· +	<·	**id** * **id** $	*shift*
$ E <· + <· **id**	·>	* **id** $	*reduce* mit $E \rightarrow$ **id**
$ E <· + E	<·	* **id** $	*shift*
$ E <· + E <· *	<·	**id** $	*shift*
$ E <· + E <· * <· **id**	·>	$	*reduce* mit $E \rightarrow$ **id**
$ E <· + E <· * E	·>	$	*reduce* mit $E \rightarrow E * E$

Die genaue „Spielregel“ für Reduktionen lautet, dass zu den Terminalsymbolen innerhalb des durch <· und ·> bestimmten Handles die angrenzenden Nichtterminale hinzugenommen werden. Deshalb erhalten wir jetzt:

$ E <· + E	·>	$	*reduce* mit $E \rightarrow E + E$
$ E		$	*accept*

Wir betrachten noch ein weiteres Beispiel, um zu sehen, wie Klammern behandelt werden bzw. um auch die Relation ≐ einmal zu benutzen. Die zu analysierende Folge sei **id** * (**id** + **id**).

Stack		*Eingabe*	*Aktion*
\$	$\lessdot$	**id** * (**id** + **id**) \$	*shift*
\$ $\lessdot$ **id**	$\gtrdot$	* (**id** + **id**) \$	*reduce* mit $E \to$ **id**
\$ *E*	$\lessdot$	* (**id** + **id**) \$	*shift*
\$ *E* $\lessdot$ *	$\lessdot$	(**id** + **id**) \$	*shift*
\$ *E* $\lessdot$ * $\lessdot$ (	$\lessdot$	**id** + **id**) \$	*shift*
\$ *E* $\lessdot$ * $\lessdot$ ($\lessdot$ **id**	$\gtrdot$	+ **id**) \$	*reduce* mit $E \to$ **id**
\$ *E* $\lessdot$ * $\lessdot$ (*E*	$\lessdot$	+ **id**) \$	*shift*
\$ *E* $\lessdot$ * $\lessdot$ (*E* $\lessdot$ +	$\lessdot$	**id**) \$	*shift*
\$ *E* $\lessdot$ * $\lessdot$ (*E* $\lessdot$ + $\lessdot$ **id**	$\gtrdot$	) \$	*reduce* mit $E \to$ **id**
\$ *E* $\lessdot$ * $\lessdot$ (*E* $\lessdot$ + *E*	$\gtrdot$	) \$	*reduce* mit $E \to E + E$
\$ *E* $\lessdot$ * $\lessdot$ (*E*	$\doteq$	) \$	*shift*
\$ *E* $\lessdot$ * $\lessdot$ (*E* $\doteq$)	$\gtrdot$	\$	*reduce* mit $E \to (E)$
\$ *E* $\lessdot$ * *E*	$\gtrdot$	\$	*reduce* mit $E \to E * E$
\$ *E*		\$	*accept*

Man beachte, wie in der drittletzten Zeile zum ersten Mal ein Handle mit mehr als einem Terminalsymbol, nämlich „(“ und „)“, reduziert wird. Dazu ist die Relation $\doteq$ erforderlich.

Wie berechnet man nun die Tabelle mit den Vorrangrelationen? Prinzipiell kann man entweder von einer gegebenen Grammatik ausgehen oder direkt Operatoren wie +, * usw. aufgrund bekannter Regeln zu Vorrang und Assoziativität miteinander in Beziehung setzen. Damit eine Grammatik sich eignet, müssen einige Regeln gelten, u. a.:

- Auf keiner rechten Seite einer Produktion darf es zwei aufeinanderfolgende Nichtterminale geben.
- Es darf keine Produktionen mit gleicher rechter Seite geben.
- Es darf keine ε-Produktionen geben.

Man kann dann die Grammatik analysieren und Vorrangrelationen anhand folgender Regeln definieren:

1. Wenn eine Folge Ab auf der rechten Seite einer Produktion erscheint und wenn aus A eine Folge αa ableitbar ist, dann muss gelten $a \gtrdot b$.

   Abbildung 3.27 macht klar, warum diese Regel gilt; offensichtlich liegt ein Handle auf dem Stack, wenn a mit b verglichen wird. Die Gültigkeit der weiteren Regeln kann man sich auf ähnliche Art klarmachen.

2. Wenn eine Folge aA auf der rechten Seite einer Produktion erscheint und wenn aus A eine Folge $b\alpha$ ableitbar ist, dann muss gelten $a \lessdot b$.

3. Wenn auf der rechten Seite einer Produktion eine Folge aAb erscheint, dann muss gelten $a \doteq b$.

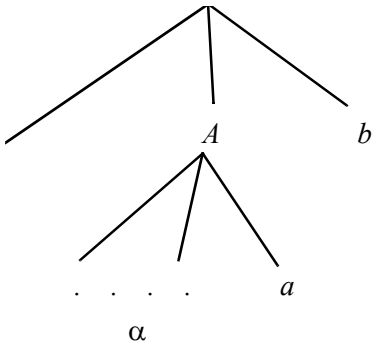

Abb. 3.27. Beispiel zu Regel 1: Vorrangrelation $a \cdot\!> b$ muss gelten.

Wenn nach Untersuchung der Grammatik herauskommt, dass es keine Konflikte zwischen den Vorrangrelationen gibt (das heißt, für jedes Paar von Terminalen gilt genau eine Relation), dann kann man die Operator-Vorrangmethode benutzen. Diese Vorgehensweise ist aber mühsam und bei vielen Grammatiken treten Konflikte auf. Deshalb wird allgemein empfohlen, dieses Verfahren nur für die Analyse von Ausdrücken einzusetzen und die Vorrangrelationen aus Vorrang (Priorität) und Assoziativität der Operatoren zu ermitteln. Dafür gelten folgende Regeln:

1. Wenn a höhere Priorität hat als b, dann setze $b <\!\cdot\, a$ und $a \cdot\!> b$.

   Zum Beispiel für + und * setze $+ <\!\cdot\, *$ und $* \cdot\!> +$.

2. Wenn a und b gleiche Priorität haben (insbesondere, wenn $a = b$), dann betrachte die Assoziativität. Falls die Operatoren links-assoziativ sind, setze $a \cdot\!> b$ und $b \cdot\!> a$; falls sie rechts-assoziativ sind, setze $a <\!\cdot\, b$ und $b <\!\cdot\, a$.

   Die Operatoren +, – sind z. B. links-assoziativ. Das heißt, der Ausdruck

   $$a - b - c + d$$

   ist auszuwerten als

   $$((a - b) - c) + d$$

   Hingegen ist der Operator $\uparrow$ (Exponentiation) rechts-assoziativ. Ein Ausdruck

   $$a \uparrow b \uparrow c$$

   sollte ausgewertet werden als

   $$a \uparrow (b \uparrow c)$$

   Die Regel erzwingt jeweils die gewünschte Auswertung.

3. Alle Arten von Klammern (etwa ( ) oder []) stehen in Beziehung $\doteq$ zueinander (also ($\doteq$) und [$\doteq$]). Für alle Symbole a, die benachbart zu Klammern auftreten können, setzen wir $a <\!\cdot$ (und) $\cdot\!> a$. Damit erzwingen wir, dass die

Klammer mit ihrem Inhalt vor der Umgebung reduziert wird. Andererseits setzen wir ($\lessdot$ *a* und *a* $\gtrdot$) und erreichen dadurch, dass der Ausdruck in den Klammern zunächst vollständig reduziert wird.

4. Bezeichner sollten vor Operatoren reduziert werden; deshalb setzen wir **id** $\gtrdot$ *a* und *a* $\lessdot$ **id**.

5. Das Endsymbol \$ hat niedrigere Priorität als jedes andere Symbol *a*, also \$ $\lessdot$ *a* und *a* $\gtrdot$ \$.

Selbsttestaufgabe 3.7: Konstruieren Sie eine Vorrangtabelle für die Operatoren +, –, *, / und ↑. Daneben gibt es natürlich weiterhin die Symbole **id**, (,) und \$. □

Selbsttestaufgabe 3.8: Parsen Sie mit Ihrer Tabelle den Ausdruck

id – **id** ↑ **id** * **id** – (**id** + **id**) □

3.3.3 LR-Analyse

Wir betrachten nun die mächtigste Klasse von Shift-Reduce-Parsern, genannt *LR-Parser*. Die Namenskonvention haben wir schon bei LL(*k*)-Grammatiken kennengelernt: Ein LR(*k*)-Parser liest von **l**inks nach rechts, erkennt eine **R**echtsableitung (in umgekehrter Reihenfolge) unter Vorausschau auf die nächsten *k* Zeichen. In der Praxis beschränkt man sich auch wieder auf Vorausschau um ein Zeichen, arbeitet also mit LR(1)-Parsern. Wir lassen die (1) gewöhnlich weg. Eine Grammatik ist – nichtformal gesprochen – vom Typ LR(*k*), wenn die (unten zu beschreibende) Konstruktion einer entsprechenden Analysetabelle konfliktfrei gelingt. Wir verzichten auf eine formale Definition.

LR-Parser sind aus folgenden Gründen attraktiv:

- Praktisch alle in Programmiersprachen vorkommenden Konstrukte können damit analysiert werden.
- Dies ist die allgemeinste Shift-Reduce-Technik, die ohne Backtracking auskommt; sie kann ebenso effizient implementiert werden wie die anderen Shift-Reduce-Verfahren.
- LR-Parser sind echt mächtiger als LL-Parser, das heißt, für alle Grammatiken, für die man „predictive parser“ gemäß Abschnitt 3.2 konstruieren kann, kann man auch LR-Parser bauen. Darüber hinaus gibt es Grammatiken und Konstrukte, die mit LR-Parsern analysierbar sind, nicht aber mit LL-Parsern.
- Mit LR-Parsern kann man beim Lesen der Eingabe Fehler so früh erkennen wie irgend möglich.

Ein intuitives Argument, warum LR(*k*)-Parser mächtiger sind als LL(*k*)-Parser, ist das folgende: Ein LR-Parser entscheidet, dass in der Ableitung eines zu analysierenden Wortes die Produktion $A \to \beta$ angewandt wurde, nachdem er alles gesehen hat, was aus den Symbolen von β abgeleitet wurde (zu jedem Symbol aus β liegt der

Ableitungsbaum schon auf dem Stack) sowie die nächsten *k* Zeichen der Eingabe. Ein LL-Parser muss diese Entscheidung treffen, nachdem er nur die ersten *k* Zeichen des aus β abgeleiteten Terminalwortes gesehen hat.

Ein Nachteil von LR-Parsern besteht darin, dass ihre Analysetabellen von Hand nur sehr schwer konstruierbar sind. Dies wird kompensiert durch die Möglichkeit, Werkzeuge zu benutzen (wie z. B. Yacc, das wir in Abschnitt 3.4 besprechen), die zu einer gegebenen Grammatik automatisch einen LR-Parser erzeugen.

Wie schon erwähnt, benutzen LR-Parser ebenfalls die Grundmethodik des Shift-Reduce-Parsens. Unterschiede liegen nur in der auf dem Stack verwalteten Information und in der Struktur der Analysetabelle. Es gibt noch verschiedene Varianten des LR-Parsens, die sich aber nur in der Methode zur Konstruktion der Tabelle unterscheiden. Die allgemeine Struktur eines LR-Parsers ist in Abb. 3.28 gezeigt.

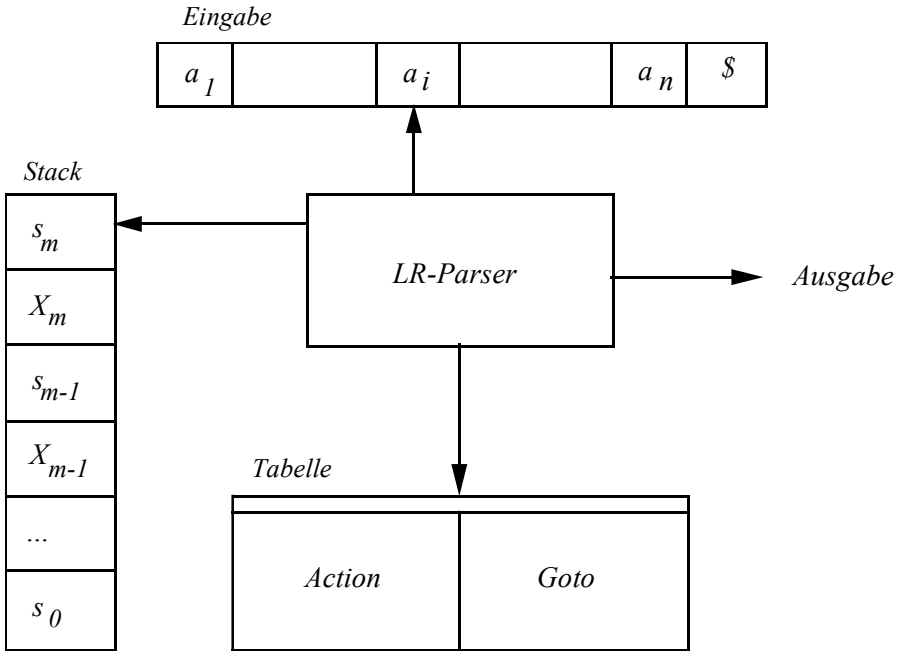

Abb. 3.28. LR-Parser als abstrakte Maschine

Neu ist gegenüber den bisher gezeigten Techniken des Shift-Reduce-Parsens, dass auf dem Stack eine alternierende Folge $s_0 X_1 s_1 X_2 \ldots s_{m-1} X_m s_m$ verwaltet wird, wobei die X_i Grammatiksymbole sind (wie bisher) und die s_i *Zustände*. Genau genommen braucht man auf dem Stack sogar *nur* die Zustände; es ist aber leichter zu verstehen, was passiert, wenn man die Grammatiksymbole mitbetrachtet.

Die Tabelle, die den Ablauf steuert, besteht bei LR-Parsern aus zwei Teilen, genannt *action* und *goto*. Für unsere Standard-Beispielgrammatik

1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow$ **id**

ist die Tabelle in Abb. 3.29 gezeigt. Wir sehen uns zunächst wieder an, wie das Parsen bei gegebener Tabelle abläuft, und wenden uns dann der Frage zu, wie die Tabelle konstruiert wird.

	Action						*Goto*		
Zustand	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Abb. 3.29. Tabelle für einen LR-Parser

Der *Action*-Teil der Tabelle enthält Einträge zu jedem Zustand und jedem Terminalsymbol der Grammatik. Es gibt 4 Arten von Einträgen, nämlich

1. *shift s*, wobei *s* ein Zustand ist
2. *reduce* $A \rightarrow \beta$
3. *accept*
4. *error*

In Abb. 3.29 ist *shift* mit *s* abgekürzt, *reduce* mit *r*, *accept* mit *acc*. Leere Felder enthalten wieder den Eintrag *error*. Der Eintrag „s5“ steht also für „*shift* mit Zustand 5“. In „r6“ bezeichnet 6 eine Produktionsnummer; dieser Eintrag bedeutet also „*reduce* mit Produktion $F \rightarrow$ **id**“.

Der *Goto*-Teil der Tabelle enthält Einträge zu Zuständen und Nichtterminalen; der Eintrag ist jeweils ein Zustand. Ein LR-Parser arbeitet mit diesen Tabellen wie folgt:

Zu Beginn steht der Eingabezeiger auf dem ersten Zeichen der Eingabe; auf dem Stack liegt der Zustand s_0. In jedem Schritt betrachtet der Parser den Zustand s_m oben auf dem Stack und das aktuelle Eingabesymbol a_i und führt die in der Tabelle an der Stelle $action[s_m, a_i]$ beschriebene Aktion durch:

1. Falls $action[s_m, a_i] = shift\ s$, so wird das Eingabesymbol a_i und der Zustand s auf den Stack gelegt. Die Eingabe wird auf das nächste Zeichen gesetzt.
2. Falls $action[s_m, a_i] = reduce\ A \rightarrow \beta$: Sei $l = |\beta|$ die Länge von β. Die obersten l Zustände und l Grammatiksymbole (also die Symbole von β) werden vom Stack genommen. Sei nun s' der oberste Stackzustand. Das Nichtterminal A und der Zustand $Goto[s', A]$ werden auf den Stack gelegt. Produktion $A \rightarrow \beta$ wird ausgegeben. – An dieser Stelle kommt also der *Goto*-Teil der Tabelle ins Spiel.
3. Falls $action[s_m, a_i] = accept$, so wird das Parsen erfolgreich beendet.
4. Falls $action[s_m, a_i] = error$, so wird eine Fehlerbehandlung eingeleitet.

Wir spielen die Analyse wieder an unserem Beispiel **id** + **id** * **id** durch. Unter „Action“ notieren wir, was in der Tabelle gefunden wird; bei *reduce*-Schritten fügen wir die benutzte und ausgegebene Produktion hinzu:

Stack	*Eingabe*	*Action*
0	**id** + **id** * **id** $	s5
0 **id** 5	+ **id** * **id** $	r6 $F \rightarrow$ **id**
0 *F* 3	+ **id** * **id** $	r4 $T \rightarrow F$
0 *T* 2	+ **id** * **id** $	r2 $E \rightarrow T$
0 *E* 1	+ **id** * **id** $	s6
0 *E* 1 + 6	**id** * **id** $	s5
0 *E* 1 + 6 **id** 5	* **id** $	r6 $F \rightarrow$ **id**
0 *E* 1 + 6 *F* 3	* **id** $	r4 $T \rightarrow F$
0 *E* 1 + 6 *T* 9	* **id** $	s7
0 *E* 1 + 6 *T* 9 * 7	**id** $	s5
0 *E* 1 + 6 *T* 9 * 7 **id** 5	$	r6 $F \rightarrow$ **id**
0 *E* 1 + 6 *T* 9 * 7 *F* 10	$	r3 $T \rightarrow T * F$
0 *E* 1 + 6 *T* 9	$	r1 $E \rightarrow E + T$
0 *E* 1	$	acc

Konstruktion der Tabelle

LR-Parsen bei gegebener Tabelle ist also relativ leicht zu verstehen. Die Kernfrage ist wieder, wie man zu der Tabelle kommt.

Zunächst kann man beobachten, dass die alternierende Folge von Zuständen und Grammatiksymbolen, die auf dem Stack verwaltet wird, sehr stark an Pfade in endlichen Automaten erinnert. Offenbar handelt es sich hier um einen endlichen Automaten, dessen Kanten (im Zustandsdiagramm) mit Grammatiksymbolen beschriftet sind. Einen Teil des Automaten sehen wir in der zweiten Zeile der obigen Beispielanalyse:

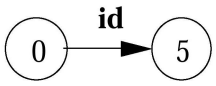

Abb. 3.30. Zustandsübergang gemäß Zeile 2 der Beispielanalyse

Der gesamte Teil des endlichen Automaten, der in der Beispielanalyse benutzt wird, ist in Abb. 3.31 gezeigt.

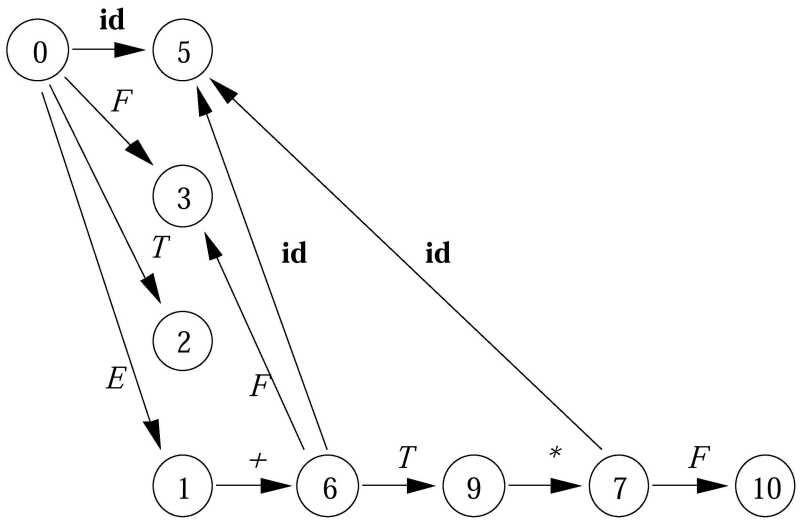

Abb. 3.31. Darstellung von Zuständen in der LR-Analyse als endlicher Automat

Tatsächlich ist die von einem solchen Automaten akzeptierte Sprache exakt die Menge der geeigneten Präfixe (Definition 3.17) der Bottom-up-Analyse, wenn wir alle Zustände als Endzustände auffassen. Ein Shift-Schritt beim Parsen entspricht einem Zustandsübergang im Automaten mit einem Terminalsymbol. Ein Reduce-Schritt bedeutet, dass man im Automaten zurückgeht zu dem Zustand, von dem ab Symbole in der rechten Seite der reduzierenden Produktion vorkommen, und dann

einen Zustandsübergang mit einem Nichtterminal zu einem weiteren Zustand durchführt.

In diesem Automaten beschreibt ein *einzelner* Zustand, nämlich jeweils der oberste Stackzustand, vollständig das Ergebnis der bisherigen Analyse. Das Ergebnis der bisherigen Analyse ist gerade die Folge der Symbole, die auf dem Stack steht. Wenn man diese Folge von links nach rechts (auf dem Stack von unten nach oben) liest, gewinnt man genausoviel Information wie durch Ansehen des obersten Stackzustands.

Man könnte einwenden, dass man aus dem Zustand, z. B. Zustand 5, nicht erkennen kann, ob auf dem Stack lediglich das Symbol „**id**“ oder etwa eine Folge „$E + T$ *** id**“ steht. Offenbar sind diese Folgen aber für die weitere Analyse gleichwertig, brauchen also nicht unterschieden zu werden.

Es bleibt die Frage, wie man die Zustände gewinnt. Ein Zustand soll „den bisherigen Fortschritt in der Analyse“ darstellen. Während der Analyse sind gewisse Teile des noch unbekannten Ableitungsbaums gesehen und reduziert worden. Das heißt, dass in jeder der im Ableitungsbaum benutzten Produktionen gewisse Fortschritte zu verzeichnen sind. Betrachten wir dazu noch einmal den Ableitungsbaum für „**id** + **id * id**“ aus Abb. 3.26 und die Zeile innerhalb der obigen Analyse mit dem Stackzustand (erstes Auftreten)

0 *E* 1 + 6 *T* 9

Zu diesem Zeitpunkt sind aus dem kompletten Ableitungsbaum nur die Reduktionen durchgeführt worden, die in Abb. 3.32 fett gezeichnet sind.

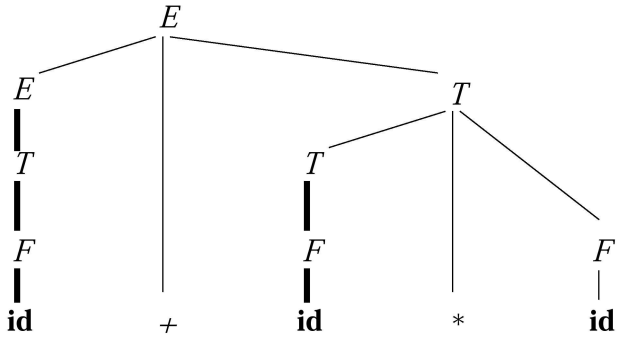

Abb. 3.32. Teilweise erkannter Ableitungsbaum in der LR-Analyse

Innerhalb der Produktionen

$E \rightarrow E + T$
$T \rightarrow T * F$

sind wir bis zu einer gewissen Stelle vorgedrungen, die wir mit einem Punkt markieren:

$$E \to E + \cdot\, T$$
$$T \to T \cdot *\, F$$

Die im linken Ast verwendeten Produktionen sind vollständig abgearbeitet, ebenso die im mittleren Ast:

$$F \to \mathbf{id}\ \cdot$$
$$T \to F \cdot$$
$$E \to T \cdot$$

Schließlich ist die Produktion im rechten Ast noch unbearbeitet:

$$F \to \cdot\ \mathbf{id}$$

Solche mit einer Position markierten Produktionen heißen *LR(0)-Elemente* oder kurz *Elemente* (engl. *LR(0)-Items* bzw. kurz *Items*).

Definition 3.18: Sei $A \to \alpha$ eine Produktion einer kontextfreien Grammatik G. Für jede Zerlegung $\beta\gamma$ von α (d. h. $\alpha = \beta\gamma$) ist $A \to \beta \cdot \gamma$ ein *LR(0)-Element* von G. Falls $\alpha = \varepsilon$, so ist $A \to \cdot$ ein LR(0)-Element von G. □

Man beachte, dass in der Definition auch $\beta = \varepsilon$ oder $\gamma = \varepsilon$ erlaubt ist. Zu einer Produktion $A \to XYZ$ gibt es also vier Elemente:

$$A \to \cdot\, XYZ$$
$$A \to X \cdot YZ$$
$$A \to XY \cdot Z$$
$$A \to XYZ \cdot$$

Die Grundidee zur Konstruktion des endlichen Automaten besteht nun darin, alle Produktionen der Grammatik und die in ihnen in der Analyse erzielten Fortschritte in Form von LR(0)-Elementen parallel zu betrachten. Ein Zustand des Automaten entspricht einer *Menge von LR(0)-Elementen*. Ein Zustandsübergang mit einem gegebenen Symbol führt zu einer neuen Menge von Elementen. Ein Zustandsübergang mit dem Symbol X überführt z. B. das Element $A \to \cdot\, XYZ$ in das Element $A \to X \cdot YZ$.

Die Konstruktion beginnt mit den Produktionen, die zum Startsymbol S der Grammatik gehören. Aus technischen Gründen ergänzt man die Grammatik um ein neues Startsymbol S' sowie eine Produktion $S' \to S$. Das Ziel dabei ist, genau eine Produktion zu bekommen, bei der der Shift-Reduce-Parser nicht reduziert, sondern akzeptiert und damit die Analyse beendet. In unserer Beispielgrammatik fügen wir also ein neues Startsymbol E' und die Produktion $E' \to E$ hinzu, diese Produktion bekommt die Nummer 0, so dass die Grammatik jetzt die Form hat:

0. $E' \to E$
1. $E \to E + T$
2. $E \to T$
3. $T \to T * F$
4. $T \to F$
5. $F \to (E)$
6. $F \to \mathbf{id}$

Der Startzustand 0 des endlichen Automaten entspricht offenbar dem LR(0)-Element

$$E' \to \cdot\ E$$

Das besagt soviel wie „ich erwarte, dass im weiteren Verlauf der Analyse die Produktion $E' \to E$ zu reduzieren sein wird; von der rechten Seite E ist noch nichts verarbeitet“. Wenn wir aber den Ableitungsbaum in Abb. 3.32 betrachten, sehen wir, dass zuerst die Produktion $F \to \mathbf{id}$ reduziert werden muss; wir müssten also ein Element $F \to \cdot\ \mathbf{id}$ haben. Wir erreichen das, indem wir zu dem Element $E' \to \cdot\ E$ alle Elemente hinzunehmen, die sich aus der rechten Seite ableiten lassen. In diesen Produktionen ist auch jeweils noch nichts von der rechten Seite verarbeitet, der Punkt steht also am Anfang. Wir müssen folglich hinzunehmen:

$$E \to \cdot\ E + T$$
$$E \to \cdot\ T$$

Natürlich könnte die erste zu reduzierende rechte Seite auch aus T abgeleitet sein; wir müssen daher weitere Elemente hinzunehmen, bis es keine Produktionen mehr gibt, an deren Anfang die Analyse stehen könnte:

$$T \to \cdot\ T * F$$
$$T \to \cdot\ F$$
$$F \to \cdot\ (\ E\ )$$
$$F \to \cdot\ \mathbf{id}$$

Dieses Hinzunehmen weiterer Elemente, solange es geht, nennt man „den Abschluss bilden“ und definiert eine entsprechende Operation *closure*:

Definition 3.19: Sei M eine Menge von LR(0)-Elementen. Der *Abschluss von M*, notiert *closure*(M), wird nach folgenden Regeln gebildet:

(i) Jedes Element in M ist auch in *closure*(M).
(ii) Wenn $A \to \alpha \cdot B\beta$ in *closure*(M) ist, dann ist für jede Produktion $B \to \gamma$ auch das Element $B \to \cdot\ \gamma$ in *closure*(M). □

Bei einer Berechnung von *closure*(M) ist der Definition gemäß die zweite Regel iteriert anzuwenden, bis nichts mehr hinzukommt.

Der Startzustand des zu konstruierenden endlichen Automaten ist nun gerade die Menge von LR(0)-Elementen *closure*($\{S' \to \cdot\ S\}$), in unserem Beispiel also

$$s_0 := closure(\{E' \to \cdot\, E\}) =$$

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

Weitere Zustände erhält man, indem man Zustandsübergänge auf die Elemente eines gegebenen Zustands anwendet. Ein Zustandsübergang entspricht dem Verschieben des Punktes über ein Symbol hinweg. Vom Zustand s_0 aus sind solche Übergänge möglich mit den Symbolen *E*, *T*, *F*, (, und **id**. Nach einem Zustandsübergang ergibt sich eine neue Menge von LR(0)-Elementen, von der dann noch der Abschluss gebildet wird, um einen endgültigen neuen Zustand zu bilden.

Vom Zustand s_0 aus erhalten wir so durch einen Übergang mit *E* die Menge von Elementen

$$\begin{aligned}& E' \to E\ \cdot\\ & E \to E\ \cdot + T\end{aligned}$$

Zustandsübergänge sind formal über eine Funktion *goto* definiert:

Definition 3.20: Sei *M* eine Menge von LR(0)-Elementen, *X* ein Grammatiksymbol. Dann ist $goto(M, X) := closure(\{A \to \alpha X \cdot \beta \mid A \to \alpha \cdot X\beta \in M\})$. □

Um den neuen Zustand s_1 zu erhalten, bilden wir also

$$s_1 := closure(\{E' \to E\ \cdot\ ,\ E \to E\ \cdot + T\}) =$$

$$\begin{aligned}\{\ & E' \to E\ \cdot\ ,\\ & E \to E\ \cdot + T\}\end{aligned}$$

In diesem Fall ist durch die Abschlussbildung nichts hinzugekommen, da in keinem der Elemente ein Nichtterminal unmittelbar rechts vom Punkt steht. Damit haben wir einen ersten kleinen Teil des endlichen Automaten konstruiert, den man z. B. so darstellen könnte:

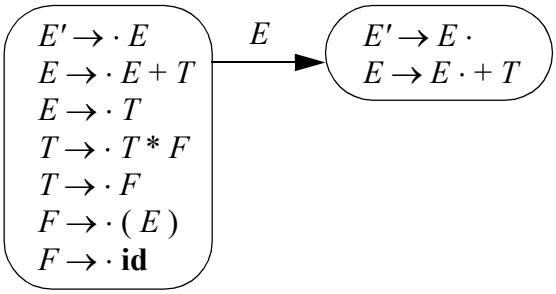

Abb. 3.33. Konstruktion eines endlichen Automaten, dessen Zustände Mengen von LR(0)-Elementen sind

Natürlich kann man nach der Konstruktion auch die knappere Darstellung in Abb. 3.34 wählen:

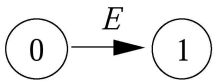

Abb. 3.34. Äquivalente Darstellung der Zustände aus Abb. 3.33

Wir berechnen nun den gesamten endlichen Automaten. Die Darstellung soll den Ablauf der Berechnung illustrieren. Der Vollständigkeit halber geben wir die bereits bekannten Zustände 0 und 1 noch einmal mit an:

$0 \xrightarrow{E} 1$

0: $E' \to \cdot\, E$ $E \to \cdot\, E + T$ $E \to \cdot\, T$ $T \to \cdot\, T * F$ $T \to \cdot\, F$ $F \to \cdot\, (E)$ $F \to \cdot\, \mathbf{id}$	1: $E' \to E \cdot$ $E \to E \cdot + T$

Den Ablauf der Berechnung muss man sich so vorstellen:

$0 \xrightarrow{T}$

$E \to T \cdot$ $T \to T \cdot * F$

Wir berechnen zunächst die Menge von LR(0)-Elementen, die sich als $goto(s_0, T)$ ergibt. Anschließend stellen wir fest, ob es eine solche Menge bzw. diesen Zustand schon gibt. Wenn das nicht der Fall ist, vergeben wir eine neue Zustandsnummer. In jedem Fall tragen wir in der oberen Zeile die Nummer des Zielzustands ein. Als Ergebnis bekommen wir also in diesem Fall die Darstellung:

$0 \xrightarrow{T} 2$

2: $E \to T \cdot$ $T \to T \cdot * F$

Die weiteren Zustände ergeben sich wie folgt:

$0 \xrightarrow{F} 3$ $\qquad$ $0 \xrightarrow{(} 4$

3: $T \to F \cdot$	4: $F \to (\cdot E)$ $E \to \cdot E + T$ $E \to \cdot T$ $T \to \cdot T * F$ $T \to \cdot F$ $F \to \cdot (E)$ $F \to \cdot$ **id**

$0 \xrightarrow{\textbf{id}} 5$ $\qquad$ $1 \xrightarrow{+} 6$

5: $F \to$ **id** $\cdot$	6: $E \to E + \cdot T$ $T \to \cdot T * F$ $T \to \cdot F$ $F \to \cdot (E)$ $F \to \cdot$ **id**

$2 \xrightarrow{*} 7$ $\qquad$ $4 \xrightarrow{E} 8$

7: $T \to T * \cdot F$ $F \to \cdot (E)$ $F \to \cdot$ **id**	8: $F \to (E \cdot)$ $E \to E \cdot + T$

$4 \xrightarrow{T} 2$	$4 \xrightarrow{F} 3$
$E \to T \cdot$ $T \to T \cdot * F$	$T \to F \cdot$

$4 \xrightarrow{(} 4$	$4 \xrightarrow{\textbf{id}} 5$
$F \to (\cdot E)$ ...	$F \to \textbf{id} \cdot$

Hier haben wir uns das Auflisten des Abschlusses des Elementes $F \to (\cdot E)$ erspart. Durch dieses Element ist der Zustand 4 bereits eindeutig identifiziert.

$6 \xrightarrow{T} 9$	$6 \xrightarrow{F} 3$
9: $E \to E + T \cdot$ $T \to T \cdot * F$	$T \to F \cdot$

$6 \xrightarrow{(} 4$	$6 \xrightarrow{\textbf{id}} 5$
$F \to (\cdot E)$ ...	$F \to \textbf{id} \cdot$

$7 \xrightarrow{F} 10$	$7 \xrightarrow{(} 4$
10: $T \to T * F \cdot$	$F \to (\cdot E)$ ...

$7 \xrightarrow{\textbf{id}} 5$	$8 \xrightarrow{)} 11$
$F \to \textbf{id} \cdot$	11: $F \to (E) \cdot$

$8 \xrightarrow{+} 6$	$9 \xrightarrow{*} 7$
$E \to E + \cdot T$ ...	$T \to T * \cdot F$ ...

Die Gesamtheit dieser Mengen von LR(0)-Elementen, von denen jede einzelne einen Zustand des endlichen Automaten definiert, heißt *kanonische LR(0)-Kollektion* für die Grammatik G' (G' ist die um das neue Startsymbol S' und die Produktion $S' \to S$ erweiterte Grammatik G). Die gerade gezeigte Vorgehensweise zur Berechnung kann man so zusammenfassen:

procedure *sets_of_items*(*G'*)
begin
  C := {*closure*({*S'* → · *S*})}; (*C* ist eine Menge von Mengen von LR(0)-Elementen)
  repeat für jede Menge *M* in *C* und für jedes Grammatiksymbol *X*, für die *goto*(*M*, *X*) nicht leer und noch nicht in *C* ist:
    füge *goto*(*M*, *X*) zu *C* hinzu
  until keine weitere Menge kann zu *C* hinzugefügt werden
end

Der vollständige endliche Automat, den wir so konstruiert haben, ist in Abb. 3.35 gezeigt.

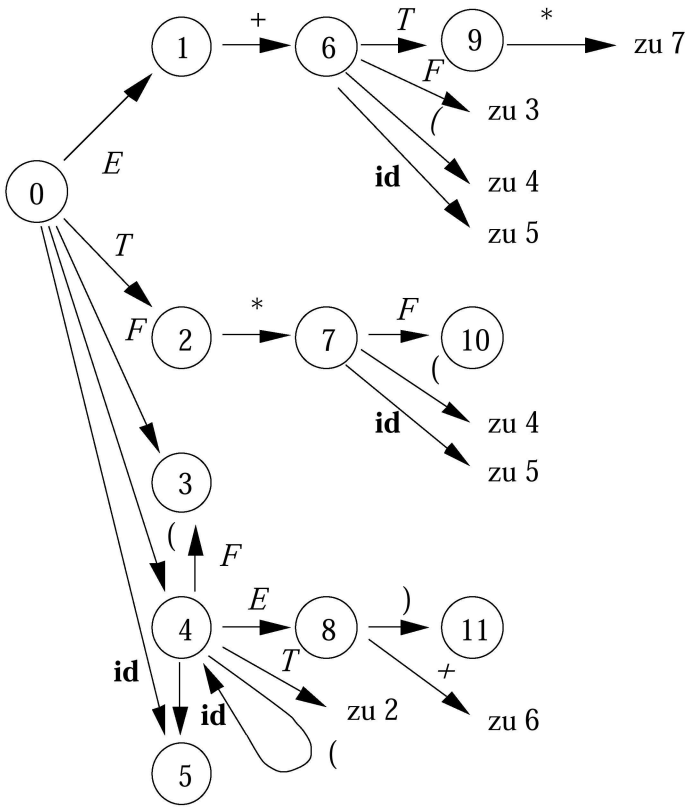

Abb. 3.35. Vollständiger endlicher Automat für die Beispielgrammatik

Selbsttestaufgabe 3.9: Gegeben sei die erweiterte Grammatik $G = (N, \Sigma, P, S')$ mit

$$
\begin{aligned}
N &= \{S', A, B\} \\
\Sigma &= \{\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}, \mathbf{e}\} \\
P &= \{\ S' \to A \\
&\qquad A \to \mathbf{a}B\mathbf{b} \mid \mathbf{ade} \mid \mathbf{b}B\mathbf{c} \mid \mathbf{bdd} \\
&\qquad B \to \mathbf{d} \qquad\qquad\qquad\qquad \}
\end{aligned}
$$

Berechnen Sie die kanonische LR(0)-Kollektion für G. □

Die letzte noch offene Frage ist, wie man aus dem endlichen Automaten die Steuertabelle (mit *Action- und Goto*-Teilen) für den Shift-Reduce-Parser erhält, für unser Beispiel also die Tabelle aus Abb. 3.29. Dazu kann man sich Folgendes überlegen:

1. In der *Goto*-Tabelle stehen Übergänge von Zuständen aus mit Nichtterminalen. Diese kann man unmittelbar aus dem endlichen Automaten übernehmen. Also für jeden Zustand i, von dem aus eine mit einem Nichtterminal A beschriftete Kante zu einem Zustand j führt, erzeugen wir einen Eintrag $goto[i, A] = j$ in der *Goto*-Tabelle. Alle anderen Felder bleiben leer.

2. Die *Action*-Tabelle enthält Einträge zu jedem Zustand und Terminalsymbol. Wenn im endlichen Automaten von einem Zustand i eine mit einem Terminalsymbol a beschriftete Kante zu einem Zustand j führt, muss das Terminalsymbol und der neue Zustand auf den Stack gelegt werden. Wir erzeugen also einen Eintrag $action[i, a] = shift\ j$ in der *Action*-Tabelle.

3. Zu reduzieren ist offenbar, wenn in einem Zustand eine Produktion vollständig abgearbeitet ist, das heißt, wenn die LR(0)-Menge des Zustands ein Element $A \to \alpha \cdot$ enthält. Für welche Terminalsymbole sind dann Einträge „*reduce* $A \to \alpha$“ zu erzeugen? Die mehr oder weniger überraschende Antwort lautet, dass es genau die Terminalsymbole in der Menge FOLLOW(A) sind, die wir in Abschnitt 3.2 definiert haben. Denn das sind ja die Terminale, die in beliebigen Satzformen der Grammatik auf das Nichtterminal A folgen können.

   Falls die LR(0)-Menge zwei oder mehr Elemente $A \to \alpha \cdot$, $B \to \alpha \cdot$, … enthält, für die die FOLLOW-Mengen nicht disjunkt sind, haben wir einen *reduce/reduce-Konflikt*, denn man kann nicht entscheiden, mit welcher Produktion reduziert werden soll. Die Konstruktion der Tabelle schlägt dann fehl; die Grammatik eignet sich nicht für diese Art der Analyse.

   Nun gibt es vom Zustand i im Allgemeinen auch Übergänge mit Terminalsymbolen in andere Zustände, und wir haben oben schon gesehen, dass wir für diese Terminale *shift*-Einträge in der *Action*-Tabelle erzeugen müssen. Daher ist es möglich, dass einerseits vom Zustand i ein Übergang mit Terminalsymbol a in Zustand j vorliegt, andererseits aber das gleiche Symbol a auch in FOLLOW(A) liegt. In diesem Fall haben wir einen *shift/reduce-Konflikt*, und auch dann schlägt die Konstruktion der Tabelle fehl; die Grammatik ist nicht geeignet.

Wenn es auch keine *shift*/*reduce*-Konflikte gibt, so erzeugen wir für jedes a in FOLLOW(A) einen Eintrag $action[i, a] = reduce\ A \rightarrow \alpha$.

4. Sei m der Zustand, der das Element $S' \rightarrow S \cdot$ enthält. Wir setzen $action[m, \$] = accept$ (überschreiben damit einen *reduce*-Eintrag, den wir gemäß Regel 3 erzeugt haben).

5. Alle anderen Einträge der *Action*-Tabelle werden auf *error* gesetzt.

Wir fassen diese Überlegungen noch einmal kurz zusammen in Algorithmus 3.21. Die bisher beschriebene Grundform der Konstruktion einer LR-Analysetabelle heißt *simple LR* oder kurz *SLR-Verfahren.* Eine Grammatik, bei der die Konstruktion konfliktfrei gelingt, heißt *vom Typ SLR*(1). Dabei steht die (1) wieder für Vorausschau um 1 Zeichen.

Algorithmus 3.21: Berechnung einer SLR(1)-Analysetabelle

Eingabe Eine erweiterte kontextfreie Grammatik G'

Ausgabe Tabellen *Action* und *Goto* für G'

Methode

1. Berechne $C = \{M_0, \ldots, M_n\}$, die kanonische LR(0)-Kollektion für G'.

2. Berechne die *Goto*-Tabelle: Für jedes Paar (M_i, M_j) in C und Nichtterminal A, für die gilt $goto(M_i, A) = M_j$, setze $goto[i, A] = j$.

3. Berechne die *Action*-Tabelle:

   3.1 Für jedes Paar (M_i, M_j) in C und Terminalsymbol a, für die gilt $goto(M_i, a) = M_j$, setze $action[i, a] = shift\ j$.

   3.2 Für jede Menge M_i in C, die ein LR(0)-Element $A \rightarrow \alpha \cdot$ enthält, und für jedes Terminal a in FOLLOW(A), setze $action[i, a] = reduce\ A \rightarrow \alpha$. Abbruch bei *reduce/reduce*- oder *shift/reduce*-Konflikten.

   3.3 Für die Menge M_m in C, die das Element $S' \rightarrow S \cdot$ enthält, setze $action[m, \$] = accept$.

   3.4 Setze alle bisher nicht belegten Einträge in der *Action*-Tabelle auf *error*.

Um unser Beispiel zu Ende führen zu können, müssen wir also noch die FOLLOW-Mengen für die gegebene (erweiterte) Grammatik G' mit dem Algorithmus 3.15 aus Abschnitt 3.2.4 berechnen. Dabei ergibt sich der in Abb. 3.36 gezeigte Graph; die Terminalsymbole an den Knoten entsprechen den FOLLOW-Mengen.

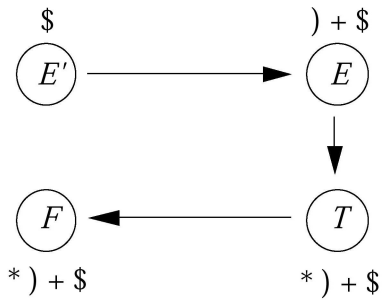

Abb. 3.36. Berechnung von FOLLOW-Mengen

Wenn man nun den Algorithmus 3.21 auf unsere Beispielgrammatik anwendet, erhält man genau die in Abb. 3.29 gezeigte Analysetabelle.

Selbsttestaufgabe 3.10:

(a) Geben Sie die Steuertabelle für die Grammatik aus Aufgabe 3.9 an.

(b) Analysieren Sie das Wort **adb**. □

Kanonische LR-Parser

Die bisher beschriebenen SLR-Parser sind schon sehr mächtig und decken weitgehend die in Programmiersprachen benutzten Konstrukte ab. Es gibt allerdings noch eine kleine Steigerung in der Leistungsfähigkeit des Analyseverfahrens, die man durch eine Verfeinerung der gezeigten Technik erreicht, genannt *kanonisches LR-Verfahren*.

Zunächst brauchen wir ein Beispiel, bei dem die SLR-Methode versagt. Es ist relativ schwierig, einfache praktisch interessante Beispiele zu finden. In Ermangelung eines besseren Beispiels betrachten wir die folgende Grammatik:

0. $S' \rightarrow S$
1. $S \rightarrow E = E$
2. $S \rightarrow$ **id**
3. $E \rightarrow E +$ **id**
4. $E \rightarrow$ **id**

Wir versuchen, eine SLR-Tabelle zu konstruieren. Das heißt, wir beginnen mit der Konstruktion der kanonischen LR(0)-Kollektion. Ausgangszustand ist

0: $S' \to \cdot\, S$ $S \to \cdot\, E = E$ $S \to \cdot\, \mathbf{id}$ $E \to \cdot\, E+ \mathbf{id}$ $E \to \cdot\, \mathbf{id}$

Wir berechnen Zustandsübergänge mit *S*, *E* und **id**:

$0 \xrightarrow{S} 1$ $\qquad$ $0 \xrightarrow{E} 2$

1: $S' \to S \cdot$	2: $S \to E \cdot = E$ $E \to E \cdot + \mathbf{id}$

$0 \xrightarrow{\mathbf{id}} 3$

3: $S \to \mathbf{id} \cdot$ $E \to \mathbf{id} \cdot$	

An dieser Stelle wird bereits das Problem erkennbar, nämlich ein reduce/reduce-Konflikt im Zustand 3. Genaugenommen tritt der Konflikt allerdings erst auf, wenn auch die FOLLOW-Mengen nicht disjunkt sind. Wir berechnen die FOLLOW-Mengen (Abb. 3.37):

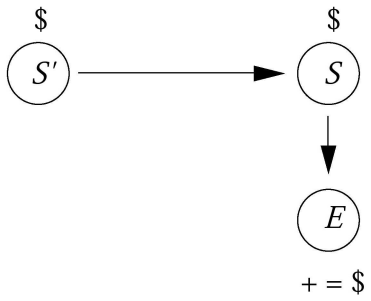

Abb. 3.37. Berechnung von FOLLOW-Mengen

Wie man sieht, enthalten in der Tat beide FOLLOW-Mengen (von *S* und *E*) das Symbol $, und ein reduce/reduce-Konflikt liegt vor.

Wenn man die Grammatik genauer untersucht, stellt man allerdings fest, dass beim Übergang aus dem Startzustand mit einem einzigen Symbol **id** zwar beide Reduktionen möglich sind, bei Reduktion mit $E \to$ **id** das nächste Symbol aber nicht \$ sein kann. Man betrachte dazu die Ableitungsbäume in Abb. 3.38.

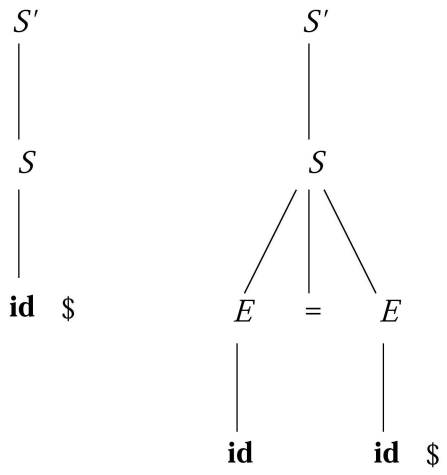

Abb. 3.38. Mögliche Folgesymbole bei Reduktion von **id**

Bei Reduktion mit $E \to$ **id** kann das Folgesymbol „=“ sein (gemäß dem rechten Ableitungsbaum in Abb. 3.38) oder „+“, falls aus dem Startsymbol $E + E$ abgeleitet wurde. Der Parser könnte dies eigentlich wissen, arbeitet also nicht präzise genug. Das Problem stammt daher, dass die FOLLOW-Mengen nur global berechnet werden, daher nicht den gerade erreichten Zustand (des endlichen Automaten) berücksichtigen.

Die Idee zur Lösung dieses Problems besteht darin, bei der Berechnung der Zustände des Automaten die jeweils möglichen Folgezeichen mitzuberechnen. Um diese Information mitzuverwalten, benutzt man anstelle der LR(0)-Elemente nun *LR(1)-Elemente*. Das sind Paare der Form $[A \to \alpha \cdot \beta, s]$, wobei s eine Menge von Terminalsymbolen (einschließlich \$) der Länge 1 ist, also eine Menge von einzelnen Symbolen. Dies ist sozusagen die spezielle FOLLOW-Menge für dieses Element.

Definition 3.22: Sei $A \to \alpha$ eine Produktion einer kontextfreien Grammatik $G = (N, \Sigma, P, S)$ und sei $s \subseteq (\Sigma \cup \{\$\})$. Für jede Zerlegung $\beta\gamma$ von α (d. h. $\alpha = \beta\gamma$) ist das Paar $[A \to \beta \cdot \gamma, s]$ ein *LR(1)-Element* von G. Falls $\alpha = \varepsilon$, so ist $[A \to \cdot\,, s]$ ein LR(1)-Element von G. Die erste Komponente, also $A \to \beta \cdot \gamma$, heißt *Kern*, die zweite Komponente *s Vorausschau-Menge* des Elementes. □

Allgemeiner kann man LR(k)-Elemente betrachten, wobei die Menge s Terminalzeichenfolgen der Länge k enthält. Damit ist auch der Ursprung der Bezeichnung „LR(0)-Element“ geklärt; diese enthalten eben eine Vorausschau um 0 Zeichen, also keine Vorausschau. Man sollte sich dabei aber nicht verwirren las-

sen: Das SLR(1)-Verfahren insgesamt hat trotzdem eine Vorausschau um 1 Zeichen, nur wird diese nicht innerhalb der Elemente berücksichtigt.

Wie wir sehen werden, sind am SLR-Verfahren insgesamt folgende Änderungen vorzunehmen:

1. Verwendung von LR(1)- anstelle von LR(0)-Elementen.
2. Definition des Abschlusses (*closure*) ändern.
3. Definition der *goto*-Funktion ändern.
4. Anfangszustand ist *closure*({[$S' \to \cdot\, S$, {\$}]}).
5. Bei der Konstruktion der *Action*-Tabelle werden nicht mehr die FOLLOW-Mengen benutzt, sondern Vorausschau-Mengen der LR(1)-Elemente.

Wir betrachten diese Änderungen nun im Einzelnen.

Definition 3.23: Sei M eine Menge von LR(1)-Elementen. Der *Abschluss von M*, notiert *closure*(M), wird nach folgenden Regeln gebildet:

(i) Jedes Element in M ist auch in *closure*(M).

(ii) Wenn $[A \to \alpha \cdot B\beta, s]$ in *closure*(M) ist, dann bilde für jede Produktion $B \to \gamma$ und jedes $a \in s$ ein Element $[B \to \cdot\, \gamma, \text{FIRST}(\beta a)]$. Falls es in *closure*($M$) schon ein Element $[B \to \cdot\, \gamma, t]$ gibt, so ersetze dieses durch $[B \to \cdot\, \gamma, t \cup \text{FIRST}(\beta a)]$. Andernfalls füge $[B \to \cdot\, \gamma, \text{FIRST}(\beta a)]$ als neues Element hinzu. □

Auch hier ist die zweite Regel anzuwenden, bis nichts mehr hinzukommt. Neu in Regel (ii) ist die Berechnung der Vorausschau-Menge. Welche Terminalsymbole können in dem neuen Element $[B \to \cdot\, \gamma, \ldots]$ (nach Reduktion) auf B folgen? Da dieses Element aus dem Element $[A \to \alpha \cdot B\beta, s]$ abgeleitet ist, sind es offensichtlich gerade die Symbole in FIRST(β). Darüber hinaus könnten Symbole aus der Menge s folgen, falls aus β das leere Wort ε ableitbar ist. Durch Verwendung von FIRST(βa) für jedes $a \in s$ in der Definition wird dieser Fall automatisch mitbehandelt. Falls es schon ein Element mit Kern $B \to \cdot\, \gamma$ in *closure*(M) gibt, ist dieses mit dem neu erzeugten Element zu verschmelzen, indem die Vorausschau-Mengen vereinigt werden.

Wir berechnen nach dieser Regel den Abschluss des Startelementes $[S' \to \cdot\, S, \{\$\}]$ für unsere Beispielgrammatik:

0. $S' \to S$
1. $S \to E = E$
2. $S \to$ **id**
3. $E \to E +$ **id**
4. $E \to$ **id**

Der besseren Lesbarkeit wegen lassen wir hier einige Klammern weg und notieren die Vorausschau-Mengen einfach durch Auflisten der Symbole rechts vom Kern:

$S' \to \cdot\, S, \$$

Im ersten Schritt kommen Elemente für die Produktionen mit Nichtterminal S (kurz S-Produktionen) hinzu:

$S \to \cdot\, E = E, \$$
$S \to \cdot\, \mathbf{id}, \$$

In diesem Fall ist β aus Regel (ii) leer, und die FIRST-Menge für das neue Element ist jeweils FIRST($) = {$}. Mit anderen Worten, wenn rechts vom betrachteten Nichtterminal nichts mehr steht, kann man einfach die Vorausschau-Menge des Ausgangselementes kopieren. – Im nächsten Schritt kommen E-Produktionen hinzu, abgeleitet aus dem jetzt vorhandenen Element $[S \to \cdot\, E = E, \{\$\}]$:

$E \to \cdot\, E + \mathbf{id}, =$
$E \to \cdot\, \mathbf{id}, =$

Hier ist FIRST(= $E$$) berechnet worden, was die Menge {=} ergibt. Beim SLR-Verfahren wären wir jetzt schon fertig. Hier aber ist E noch in einem neuen Kontext hinzugekommen, nämlich innerhalb des Elementes $[E \to \cdot\, E+ \mathbf{id}, \{=\}]$, und wir müssen Regel (ii) noch einmal anwenden. Damit erhalten wir zunächst neue Elemente:

$E \to \cdot\, E + \mathbf{id}, +$
$E \to \cdot\, \mathbf{id}, +$

Da diese Kerne schon vorhanden sind, verschmelzen wir die Elemente:

$E \to \cdot\, E + \mathbf{id}, = +$
$E \to \cdot\, \mathbf{id}, = +$

Damit ist der Abschluss von $[S' \to \cdot\, S, \{\$\}]$ vollständig berechnet, und der Startzustand des Automaten ist:

0: $S' \to \cdot\, S, \$$
$S \to \cdot\, E = E, \$$
$S \to \cdot\, \mathbf{id}, \$$
$E \to \cdot\, E + \mathbf{id}, = +$
$E \to \cdot\, \mathbf{id}, = +$

Zustandsübergänge, also die *goto*-Funktion, werden im Grunde genauso berechnet wie bisher. Die Definition ändert sich nur technisch, da jetzt LR(1)-Elemente zu behandeln sind. Natürlich wird auch die neue Definition des Abschlusses verwendet.

Definition 3.24: Sei M eine Menge von LR(1)-Elementen, X ein Grammatiksymbol. Dann ist $goto(M, X) := closure(\{[A \to \alpha X \cdot \beta, s] \mid [A \to \alpha \cdot X\beta, s] \in M\})$. □

Wir berechnen für unser Beispiel wieder Zustandsübergänge mit *S*, *E* und **id**:

$0 \xrightarrow{S} 1$ $\qquad$ $0 \xrightarrow{E} 2$

1: $S' \to S \cdot, \$$	2: $S \to E \cdot = E, \$$ $E \to E \cdot + \mathbf{id}, = +$

$0 \xrightarrow{\mathbf{id}} 3$

3: $S \to \mathbf{id} \cdot, \$$ $E \to \mathbf{id} \cdot, = +$

Man sieht sehr schön, dass jetzt der reduce/reduce-Konflikt nicht auftritt, da die Vorausschau-Mengen der beiden Kerne $S \to \mathbf{id} \cdot$ und $E \to \mathbf{id} \cdot$ disjunkt sind.

Die letzte Änderung gegenüber dem SLR-Verfahren betrifft die Konstruktion der *Action*-Tabelle. In Algorithmus 3.21 ist Anweisung 3.2 zu ersetzen durch:

> 3.2 Für jede Menge M_i in C, die ein LR(1)-Element $[A \to \alpha \cdot, s]$ enthält, und für jedes Terminal a in s setze $action[i, a] = reduce\ A \to \alpha$. Abbruch bei *reduce/reduce*- oder *shift/reduce*-Konflikten.

Wir verwenden also jetzt die Vorausschau-Mengen der LR(1)-Elemente anstelle der globalen FOLLOW-Mengen. Wenn die Konstruktion der Tabellen mit diesem Verfahren konfliktfrei gelingt, so haben wir einen *kanonischen LR(1)-Parser* erhalten, und die Grammatik heißt *vom Typ LR(1)*.

Selbsttestaufgabe 3.11: Gegeben sei die erweiterte Grammatik $G = (N, \Sigma, P, S')$ mit

$$
\begin{aligned}
N &= \{S', S, A, B\} \\
\Sigma &= \{\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}, \mathbf{e}\} \\
P &= \{\ S' \to S \\
&\qquad S \to \mathbf{a}A\mathbf{b} \mid \mathbf{c}B\mathbf{b} \mid \mathbf{a}B\mathbf{d} \mid \mathbf{c}A\mathbf{d} \\
&\qquad A \to \mathbf{e} \\
&\qquad B \to \mathbf{e} \\
&\quad \}.
\end{aligned}
$$

(a) Zeigen Sie: *G* ist keine SLR(1)-Grammatik.

(b) Erstellen Sie eine geeignete Steuertabelle für *G*. □

LALR(1)-Parser

Wir haben jetzt alle grundlegenden Konzepte der LR-Analyse besprochen. In der Praxis werden LR-Parser von Werkzeugen konstruiert wie etwa von Yacc, das Thema des folgenden Abschnitts ist. Die praktisch eingesetzten LR-Parser gehen in

der Verfeinerung des Grundkonzepts noch zwei Schritte weiter; wir skizzieren sie noch kurz.

Der erste Schritt resultiert aus einer Beobachtung, die man bei der Konstruktion von kanonischen LR-Parsern im Vergleich mit SLR-Parsern macht. Man stellt fest, dass ein kanonischer LR-Parser oft sehr viel mehr Zustände besitzt als ein entsprechender SLR-Parser. Bei typischen Programmiersprachen kann z. B. ein SLR-Parser mehrere hundert, ein kanonischer LR-Parser mehrere tausend Zustände besitzen. Andererseits beobachtet man, dass beim kanonischen LR-Parser viele dieser Zustände gleich sind in Bezug auf die Kerne ihrer Elemente und sich nur in den Vorausschau-Mengen unterscheiden.

Die Idee besteht nun darin, alle Zustände, deren Kerne gleich sind (genauer: die Mengen der Kerne ihrer Elemente), zu verschmelzen. Man verschmilzt zwei Zustände, indem man darin jeweils die beiden Elemente mit gleichem Kern nimmt und ihre Vorausschau-Mengen vereinigt. Zustandsübergänge werden beim Verschmelzen auch richtig mitbehandelt (das werden wir unten noch genauer sehen). Anschließend konstruiert man die *Action*- und *Goto*-Tabellen wie vorher beim kanonischen LR(1)-Verfahren. Wenn die Konstruktion konfliktfrei gelingt, dann hat man einen *LALR(1)-Parser* erhalten und die Grammatik heißt *vom Typ LALR(1)*. Dabei steht LALR für *lookahead-LR*.

Wenn die Ausgangsgrammatik vom Typ LR(1) ist, dann können durch das Verschmelzen keine shift/reduce-Konflikte entstehen. Es ist allerdings möglich, dass reduce/reduce-Konflikte entstehen; dies kommt aber nur sehr selten vor. Insofern ist die LALR(1)-Sprachklasse minimal schwächer als die LR(1)-Klasse. Der Vorteil ist aber, dass der LALR(1)-Parser nur genauso viele Zustände hat wie der entsprechende SLR-Parser.

Eine einfache, allerdings ineffiziente Vorgehensweise zur Konstruktion eines LALR-Parsers ist also die folgende:

Algorithmus 3.25: Berechnung einer LALR(1)-Analysetabelle

Eingabe Eine erweiterte kontextfreie Grammatik G'

Ausgabe LALR(1)-Tabellen *Action* und *Goto* für G'

Methode

1. Berechne $C = \{M_0, \ldots, M_n\}$, die kanonische Kollektion von Mengen von LR(1)-Elementen. Bezeichne K_i den Kern von M_i, d. h. die Menge der Kerne der Elemente von M_i.

2. Für jeden Kern K_j, der in C vorkommt, bestimme alle Mengen von LR(1)-Elementen in C mit diesem Kern und ersetze sie durch ihre Vereinigung.

3. Sei $C' = \{L_0, \ldots L_m\}$ die resultierende neue Kollektion von Mengen von LR(1)-Elementen. Zustandsübergänge (die *goto*-Funktion) werden wie folgt berechnet: Wenn L die Vereinigung von $M_{i1} \cup M_{i2} \cup \ldots \cup M_{ip}$ ist, dann sind die Kerne von $goto(M_{i1}, X)$, $goto(M_{i2}, X)$, …, $goto(M_{ip}, X)$ alle gleich. Sei L' die Menge aus C',

die den gleichen Kern hat wie alle diese Mengen. Definiere dann die neue *goto*-Funktion für C' als $goto(L, X) = L'$.

4. Konstruiere *Action*- und *Goto*-Tabellen wie beim kanonischen LR(1)-Verfahren. Wenn Konflikte auftreten, schlägt die Konstruktion fehl; dann ist die Grammatik nicht vom Typ LALR(1).

Der zweite Verbesserungsschritt besteht darin, die LALR(1)-Tabelle zu bestimmen, ohne dabei die riesige kanonische LR(1)-Kollektion zu berechnen. Die grobe Strategie dabei sieht so aus, dass man zunächst die LR(0)-Kollektion bestimmt (also die Zustände des SLR-Parsers) und dann Vorausschau-Mengen hinzufügt. Das genaue Verfahren ist relativ kompliziert; es ist z. B. in (Aho, Sethi und Ullman 1986) beschrieben.

3.3.4 Yacc: Ein Parsergenerator

Das Werkzeug *Yacc*[3], das mit UNIX-Systemen verfügbar ist, erzeugt aus einer Grammatik-Spezifikation einen LALR(1)-Parser. Ähnlich wie bei Lex ist es möglich, C-Programmcode einzubetten und dadurch Übersetzungsaktionen ausführen zu lassen. Natürlich kooperieren von Yacc erzeugte Parser auch mit Scannern, die von Lex generiert wurden, verbrauchen also die Folge von Token, die von diesen geliefert werden. Das Wort *Yacc* steht übrigens für „yet another compiler-compiler“, deutet also an, dass es schon eine ganze Reihe solcher Systeme gab, als Yacc geschrieben wurde. Nichtsdestoweniger ist Yacc das populärste von allen geworden, sicherlich auch wegen der Verbreitung mit UNIX.

Die grundsätzliche Vorgehensweise bei Spezifikation und Erzeugen des Parsers (Abb. 3.39) ist ähnlich wie die für Lex in Abb. 2.9 gezeigte.

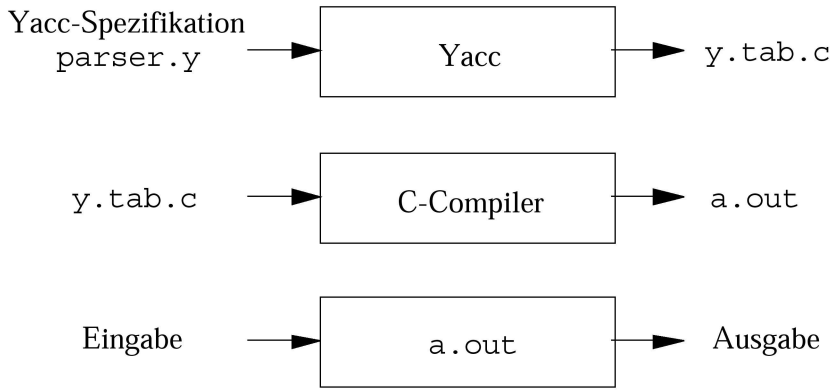

Abb. 3.39. Generierung eines Parsers mit Yacc

[3] In Linux-Systemen heißen die Werkzeuge Flex und Bison anstelle von Lex und Yacc.

Das Spezifikationsfile, hier benannt *parser.y*, wird von Yacc übersetzt in ein C-Programm mit dem festen Namen *y.tab.c* durch ein Kommando (unter UNIX):

```
yacc parser.y
```

Anschließend übersetzt man das C-Programm z. B. mit einem Befehl

```
cc y.tab.c -ly
```

Dabei dient der Parameter „-ly“ dazu, die Yacc-Bibliotheken hinzuzubinden, insbesondere das eigentliche LR-Parse-Programm, das ja fest ist, wie wir oben gesehen haben (nur die Tabellen werden jeweils neu berechnet). Bei dieser Befehlsform wird ein ausführbares Programm *a.out* erzeugt, das den generierten Parser darstellt. Natürlich kann man auch *y.tab.c* mit anderen Programmen zusammen übersetzen oder binden. Dann steht der erzeugte Parser als Funktion *yyparse*() zur Verfügung.

Die Yacc-Spezifikation selbst hat folgende Struktur:

```
Deklarationen
%%
Grammatik (Produktionen) mit semantischen Aktionen
%%
Hilfsprozeduren
```

Ein beliebtes Beispiel ist die Yacc-Spezifikation eines einfachen Taschenrechners. Grundlage ist in etwa die Grammatik für arithmetische Ausdrücke, die wir schon so oft gesehen haben. Hier werden semantische Aktionen angehängt, die die Berechnung jeweils sofort ausführen. Das Folgende ist ein komplettes Yacc-Programm:

```
%{
#include <ctype.h>
#include <stdio.h>
%}
%token NUMBER
%%
lines   : lines expr '\n'{ printf("%d\n", $2); }
        | lines '\n'
        |
        ;
expr    : expr '+' term { $$ = $1 + $3; }
        | expr '-' term { $$ = $1 - $3; }
        | term
        ;
term    : term '*' factor{ $$ = $1 * $3; }
        | term '/' factor{ $$ = $1 / $3; }
        | factor
        ;
factor  : '(' expr ')'  { $$ = $2; }
        | NUMBER
        ;
%%
int yylex() {
  int c;
  c = getchar();
```

```
  if (isdigit(c)) {
    ungetc(c, stdin); scanf("%d", &yylval); return NUMBER;
  }
  return c;
}
```

Diese Spezifikation steht (beim Autor) in einem File *calculator.y*. Man kann es mit Yacc, anschließend mit dem C-Compiler übersetzen und dann aufrufen; anschließend kann man arithmetische Ausdrücke eintippen und bekommt jeweils das Ergebnis ausgegeben. Ein Beispiel (Ausgaben des Systems sind fett gedruckt, „>“ ist der System-Prompt):

```
>yacc calculator.y
>cc y.tab.c -ly
>a.out
6*3
18
10
10
(3*5+(22-7))
30
500-50-20
430
500-(50-20)
470
100 + 100
syntax error
>
```

Die Zeile „100 + 100“ ist fehlerhaft, da sie Leerzeichen enthält, die die Spezifikation nicht vorsieht. Der Parser gibt die Meldung „syntax error“ aus und bricht ab.

Wir betrachten nun die Yacc-Spezifikation genauer und beginnen mit dem mittleren, dem *Grammatik-Teil*. Hier gelten folgende Regeln:

- Nichtterminale werden durch einfache Zeichenfolgen oder Zeichen, ohne besondere Kennzeichnung oder Klammerung, dargestellt. Im Beispiel sind *lines*, *expr*, *term* usw. Nichtterminale. Nichtterminale brauchen nicht deklariert zu werden.
- Terminale („Token“) können entweder einzelne Zeichen sein (Typ *char* oder *int* in C); in diesem Fall werden sie in Produktionen in einfache Anführungszeichen gesetzt. Im Beispiel sind '+', '(', '\n' so notiert. Oder es sind explizit deklarierte Token (im Beispiel NUMBER); die Deklaration steht dann im ersten Teil der Spezifikation in der Form

  ```
  %token NUMBER
  ```

- Eine Menge von A-Produktionen $A \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$ wird in Yacc notiert als

  ```
  A : <alpha1> | <alpha2> | … | <alpha-n>;
  ```

  Meist schreibt man Produktionen untereinander, also

```
A       : <alpha1>
        | <alpha2>
        …
        | <alpha-n>
        ;
```

insbesondere deshalb, weil man mit jeder Alternative noch eine *semantische Aktion* (s.u.) verbinden will, die man dann ans Ende der Zeile setzen kann, also:

```
A       : <alpha1> {<sem. Aktion für alpha1>}
        | <alpha2> {<sem. Aktion für alpha2>}
        …
        | <alpha-n>{<sem. Aktion für alpha-n>}
        ;
```

Eine ε-Produktion notiert man als leere rechte Seite, z. B. das Paar $A \rightarrow \alpha_1 \mid \varepsilon$:

```
A       : <alpha1>
        |
        ;
```

Semantische Aktionen sind C-Code, der auszuführen ist, wenn eine Reduktion mit der zugehörigen rechten Seite vorgenommen wird. Dort können beliebige Programmstücke stehen; insbesondere manipuliert man aber *Attribute*, die den Grammatiksymbolen zugeordnet sind (dies ist ein Vorgriff auf Kapitel 4, in dem wir *attributierte Grammatiken* besprechen). Attribute sind einfach Behälter für beliebige Informationen. Yacc verwaltet für jedes Symbol auf dem Parserstack eine Datenstruktur zur Darstellung des Attributs. Solange man diese Datenstruktur nicht besonders deklariert, ist jedem Symbol ein *Integer*-Wert zugeordnet. Im Programmcode für die semantische Aktion bezieht man sich auf die Attribute der Symbole der *rechten Seite* mit den Bezeichnern $1, $2, …, $*n*, auf das Attribut des Symbols auf der *linken Seite* mit dem Bezeichner $$. In der Zeile

```
expr    : expr '+' term { $$ = $1 + $3; }
```

sind daher $1, $2 und $3 die Attribute von *expr*, + und *term* auf der rechten Seite, und $$ gehört zu *expr* auf der linken Seite. Die semantische Aktion berechnet also bei der Reduktion den Wert des Ausdrucks links als Summe der bereits vorliegenden Werte von *expr* und *term* auf der rechten Seite.

Wenn für eine Alternative mit einem einzelnen Symbol auf der rechten Seite keine semantische Aktion angegeben wird, so greift die *Default-Aktion* {$$ = $1;}. Deshalb braucht im obigen Beispiel etwa für

```
expr    : term
```

keine Aktion angegeben zu werden; das Attribut von *term* wird automatisch dem Attribut von *expr* zugewiesen.

Im *Deklarationsteil* stehen zunächst, eingeklammert durch %{ und %}, reguläre C-Deklarationen, die für semantische Aktionen des Teils 2 oder für Hilfsprozeduren des Teils 3 benötigt werden. Darüber hinaus enthält dieser Teil Tokendeklarationen.

In unserem Scannerbeispiel in Abschnitt 2.3 hatten wir Token explizit als Integer-Konstanten definiert. In Yacc könnte man schreiben:

```
%token OPEN, CLOSE, INTEGER, REAL, BOOLEAN, STRING, SYMBOL
%token OPENTEXT, CLOSETEXT
```

Damit werden die Token automatisch als Integer-Konstanten definiert. Yacc plaziert diese Definitionen in einem File *y.tab.h*. Um diese Konstanten auch einem Lex-generierten Scanner bekanntzumachen, genügt es daher, im Deklarationsteil der Lex-Spezifikation zu sagen

```
#include "y.tab.h"
```

Schließlich stehen im Teil 3 *Hilfsprozeduren*, die somit in semantischen Aktionen verwendet werden können. Im Beispiel haben wir hier die Funktion *yylex*() für die lexikalische Analyse untergebracht. Eine solche als

```
int yylex() {…}
```

deklarierte Prozedur muss entweder explizit vom Programmierer geschrieben oder durch Lex erzeugt werden; der von Yacc erzeugte Parser ruft sie jeweils auf, um das nächste Token zu bekommen. Token sind einfach Integer-Werte; insofern können einzelne Zeichen wie auch deklarierte Token auf gleiche Art übergeben werden.

Die lexikalische Analyse kann einem Token bereits einen Wert zuordnen. Dieser wird über eine globale Variable

```
int yylval;
```

vom Scanner an den Parser übergeben. In unserem Beispiel werden Token und Wert in den Anweisungen

```
scanf("%d", &yylval); return NUMBER;
```

übergeben. Auf der Seite des Parsers kann auf diesen Wert als Attribut des Terminalsymbols (Tokens) zugegriffen werden.

Konflikte. Man kann Yacc auch mit mehrdeutigen Grammatiken verwenden bzw. Grammatiken, die nicht vom Typ LALR(1) sind. In diesem Fall gibt Yacc nicht einfach auf, sondern löst die Konflikte nach folgenden Regeln auf:

1. Bei shift/reduce-Konflikten wird stets zugunsten von shift entschieden. Dies ist normalerweise das gewünschte Verhalten. Zum Beispiel wird bei

   ```
   cond    : IF boolexpr THEN statement
           | IF boolexpr THEN statement ELSE statement
           ;
   ```

   bei Auftreten von ELSE shift gewählt, der ELSE-Zweig bei geschachtelten bedingten Anweisungen also der inneren Anweisung zugeordnet, was die übliche Interpretation ist.

2. Bei reduce/reduce-Konflikten wird mit der Produktion reduziert, die in der Yacc-Spezifikation zuerst aufgelistet ist.

Beim Übersetzen der Spezifikation meldet Yacc die Anzahlen aufgetretener shift/reduce- und reduce/reduce-Konflikte. Man kann Yacc mit einer Option „–v“ aufrufen, also

```
yacc -v parser.y
```

Dadurch wird ein File *y.output* generiert, das eine lesbare Beschreibung der erzeugten LALR-Parse-Tabellen enthält. Dort kann man überprüfen, wie Konflikte aufgelöst wurden. Insbesondere bei reduce/reduce-Konflikten ist es ratsam, dort nachzuschauen.

In Kapitel 5 betrachten wir als ein ausführliches Anwendungsbeispiel für Yacc (und Lex) die Konstruktion eines Compilers für eine Dokument-Beschreibungssprache.

3.4 Literaturhinweise

Gute Darstellungen der in diesem Kapitel behandelten Themen bieten (Aho et al. 2006) und (Parsons 1992); formale Definitionen bzw. Darstellungen der LL(k)- und LR(k)-Analyse aus formaler Sicht finden sich in (Sudkamp 2005). Eine umfassende Behandlung der Theorie der Syntaxanalyse bieten (Sippu und Soisalon-Soininen 1988, 1990).

Die Operator-Vorranganalyse geht zurück auf Floyd (1963). Operator-Vorrangmethoden können übrigens auch in einer Top-down-Analyse eingesetzt werden (Pratt 1973).

Die Theorie der LR(k)-Analyse stammt von Knuth (1965). Dies wurde zunächst für ein rein theoretisches Ergebnis gehalten, bis Korenjak (1969) und DeRemer die Methode im Hinblick auf praktische Einsatzfähigkeit verbesserten; DeRemer entwickelte dabei die SLR- und LALR-Verfahren (DeRemer 1969, 1971). Effiziente Methoden zur Berechnung von LALR(1)-Vorausschau-Mengen wurden u. a. von Kristensen und Madsen (1981) und DeRemer und Pennello (1982) vorgeschlagen. Ein Übersichtsartikel zur LR(k)-Analyse ist (Aho und Johnson 1974). Dem zweiten Autor, S.C. Johnson, verdanken wir auch das Yacc-System (Johnson 1975). Nähere Informationen und eine gute Einführung zu Yacc bietet (Levine, Mason und Brown 1992).

Lösungen zu den Selbsttestaufgaben

Aufgabe 3.7

Wir berücksichtigen, dass der Operator ↑ höhere Priorität besitzt als +, –, * und /. Dann erhalten wir die folgende Vorrangtabelle.

		Eingabe								
		+	–	*	/	↑	**id**	(	)	$
Stack	+	·>	·>	<·	<·	<·	<·	<·	·>	·>
	–	·>	·>	<·	<·	<·	<·	<·	·>	·>
	*	·>	·>	·>	·>	<·	<·	<·	·>	·>
	/	·>	·>	·>	·>	<·	<·	<·	·>	·>
	↑	·>	·>	·>	·>	<·	<·	<·	·>	·>
	id	·>	·>	·>	·>	·>			·>	·>
	(	<·	<·	<·	<·	<·	<·	<·	≐	
	)	·>	·>	·>	·>	·>			·>	·>
	$	<·	<·	<·	<·	<·	<·	<·		

Aufgabe 3.8

Wir analysieren die Folge **id** – **id** ↑ **id** * **id** – (**id** + **id**). Dabei sei die vereinfachte Beispielgrammatik durch die Produktionen $E \rightarrow E \uparrow E \mid E - E$ vervollständigt worden.

Stack		*Eingabe*	*Aktion*
\$	$\lessdot$	**id** – **id** ↑ **id** * **id** – (**id** + **id**) \$	*shift*
\$ $\lessdot$ **id**	$\gtrdot$	– **id** ↑ **id** * **id** – (**id** + **id**) \$	*reduce* mit $E \rightarrow$ **id**
\$ E	$\lessdot$	– **id** ↑ **id** * **id** – (**id** + **id**) \$	*shift*
\$ $E \lessdot$ –	$\lessdot$	**id** ↑ **id** * **id** – (**id** + **id**) \$	*shift*
\$ $E \lessdot$ – $\lessdot$ **id**	$\gtrdot$	↑ **id** * **id** – (**id** + **id**) \$	*reduce* mit $E \rightarrow$ **id**
\$ $E \lessdot$ – E	$\lessdot$	↑ **id** * **id** – (**id** + **id**) \$	*shift*
\$ $E \lessdot$ – $E \lessdot$ ↑	$\lessdot$	**id** * **id** – (**id** + **id**) \$	*shift*
\$ $E \lessdot$ – $E \lessdot$ ↑ $\lessdot$ **id**	$\gtrdot$	* **id** – (**id** + **id**) \$	*reduce* mit $E \rightarrow$ **id**
\$ $E \lessdot$ – $E \lessdot$ ↑ E	$\gtrdot$	* **id** – (**id** + **id**) \$	*reduce* mit $E \rightarrow E \uparrow E$
\$ $E \lessdot$ – E	$\lessdot$	* **id** – (**id** + **id**) \$	*shift*
\$ $E \lessdot$ – $E \lessdot$ *	$\lessdot$	**id** – (**id** + **id**) \$	*shift*
\$ $E \lessdot$ – $E \lessdot$ * $\lessdot$ **id**	$\gtrdot$	– (**id** + **id**) \$	*reduce* mit $E \rightarrow$ **id**
\$ $E \lessdot$ – $E \lessdot$ * E	$\gtrdot$	– (**id** + **id**) \$	*reduce* mit $E \rightarrow E * E$
\$ $E \lessdot$ – E	$\gtrdot$	– (**id** + **id**) \$	*reduce* mit $E \rightarrow E - E$
\$ E	$\lessdot$	– (**id** + **id**) \$	*shift*
\$ $E \lessdot$ –	$\lessdot$	(**id** + **id**) \$	*shift*
\$ $E \lessdot$ – $\lessdot$ (	$\lessdot$	**id** + **id**) \$	*shift*
\$ $E \lessdot$ – $\lessdot$ ($\lessdot$ **id**	$\gtrdot$	+ **id**) \$	*reduce* mit $E \rightarrow$ **id**
\$ $E \lessdot$ – $\lessdot$ (E	$\lessdot$	+ **id**) \$	*shift*
\$ $E \lessdot$ – $\lessdot$ ($E \lessdot$ +	$\lessdot$	**id**) \$	*shift*
\$ $E \lessdot$ – $\lessdot$ ($E \lessdot$ + $\lessdot$ **id**	$\gtrdot$	) \$	*reduce* mit $E \rightarrow$ **id**
\$ $E \lessdot$ – $\lessdot$ ($E \lessdot$ + E	$\gtrdot$	) \$	*reduce* mit $E \rightarrow E + E$
\$ $E \lessdot$ – $\lessdot$ (E	$\doteq$	) \$	*shift*
\$ $E \lessdot$ – $\lessdot$ ($E \doteq$)	$\gtrdot$	\$	*reduce* mit $E \rightarrow$ (E)
\$ $E \lessdot$ – E	$\gtrdot$	\$	*reduce* mit $E \rightarrow E - E$
\$ E		\$	*accept*

Aufgabe 3.9

Wir erhalten

$0 \xrightarrow{A} 1$

0: $S' \to \cdot A$ $A \to \cdot \mathbf{a}B\mathbf{b}$ $A \to \cdot \mathbf{ade}$ $A \to \cdot \mathbf{b}B\mathbf{c}$ $A \to \cdot \mathbf{bdd}$	1: $S' \to A\cdot$

$0 \xrightarrow{\mathbf{a}} 2$ $0 \xrightarrow{\mathbf{b}} 3$

2: $A \to \mathbf{a}\cdot B\mathbf{b}$ $A \to \mathbf{a}\cdot \mathbf{de}$ $B \to \cdot \mathbf{d}$	3: $A \to \mathbf{b}\cdot B\mathbf{c}$ $A \to \mathbf{b}\cdot \mathbf{dd}$ $B \to \cdot \mathbf{d}$

$2 \xrightarrow{B} 4$ $2 \xrightarrow{\mathbf{d}} 5$

4: $A \to \mathbf{a}B\cdot \mathbf{b}$	5: $A \to \mathbf{ad}\cdot \mathbf{e}$ $B \to \mathbf{d}\cdot$

$3 \xrightarrow{B} 6$ $3 \xrightarrow{\mathbf{d}} 7$

6: $A \to \mathbf{b}B\cdot \mathbf{c}$	7: $A \to \mathbf{bd}\cdot \mathbf{d}$ $B \to \mathbf{d}\cdot$

$4 \xrightarrow{\mathbf{b}} 8$ $5 \xrightarrow{\mathbf{e}} 9$

8: $A \to \mathbf{a}B\mathbf{b}\cdot$	9: $A \to \mathbf{ade}\cdot$

$6 \xrightarrow{\mathbf{c}} 10$ $7 \xrightarrow{\mathbf{d}} 11$

10: $A \to \mathbf{b}B\mathbf{c}\cdot$	11: $A \to \mathbf{bdd}\cdot$

Aufgabe 3.10

(a) Wir benötigen die FOLLOW-Mengen von G. Die Anwendung von Algorithmus 3.15 liefert uns

FOLLOW(S') = {\$}
FOLLOW(A) = {\$}
FOLLOW(B') = {**b**, **c**}.

Wir nummerieren die Produktionen:

(1) S' → A
(2) A → **a***B***b**
(3) A → **ade**
(4) A → **b***B***c**
(5) A → **bdd**
(6) B → **d**

und wenden Algorithmus 3.21 an. Als Ergebnis erhalten wir die folgende Steuertabelle.

	Action						*Goto*		
Zustand	**a**	**b**	**c**	**d**	**e**	$	*S'*	*A*	*B*
0	s2	s3						1	
1						acc			
2				s5					4
3				s7					6
4		s8							
5		r6	r6		s9				
6			s10						
7		r6	r6	s11					
8						r2			
9						r3			
10						r4			
11						r5			

(b) Wir analysieren das Wort **adb**.

Stack	Eingabe	Action
0	**adb**\$	s2
0**a**2	**db**\$	s5
0**a**2**d**5	**b**\$	r6 $B \to \mathbf{d}$
0**a**2*B*4	**b**\$	s8
0**a**2*B*4**b**8	\$	r2 $A \to \mathbf{a}B\mathbf{b}$
0*A*1	\$	acc

Aufgabe 3.11

(a) Wir erstellen zunächst die kanonische LR(0)-Kollektion für *G*.

$0 \xrightarrow{S} 1$

0: $S' \to \cdot S$ $S \to \cdot\mathbf{a}A\mathbf{b}$ $S \to \cdot\mathbf{c}B\mathbf{b}$ $S \to \cdot\mathbf{a}B\mathbf{d}$ $S \to \cdot\mathbf{c}A\mathbf{d}$	1: $S' \to S\cdot$

$0 \xrightarrow{\mathbf{a}} 2$ $\quad$ $0 \xrightarrow{\mathbf{c}} 3$

2: $S \to \mathbf{a}\cdot A\mathbf{b}$ $S \to \mathbf{a}\cdot B\mathbf{d}$ $A \to \cdot\mathbf{e}$ $B \to \cdot\mathbf{e}$	3: $S \to \mathbf{c}\cdot B\mathbf{b}$ $S \to \mathbf{c}\cdot A\mathbf{d}$ $A \to \cdot\mathbf{e}$ $B \to \cdot\mathbf{e}$

$2 \xrightarrow{A} 4$ $\quad$ $2 \xrightarrow{B} 5$

4: $S \to \mathbf{a}A\cdot\mathbf{b}$	5: $S \to \mathbf{a}B\cdot\mathbf{d}$

$2 \xrightarrow{\mathbf{e}} 6$

6: $A \to \mathbf{e}\cdot$ $B \to \mathbf{e}\cdot$	…

Betrachten wir Zustand 6 und berechnen die FOLLOW-Mengen für A und B. Es ist FOLLOW(A) = {**b**, **d**} = FOLLOW(B). Es liegt ein *reduce*/*reduce*-Konflikt vor, damit ist G keine SLR(1)-Grammatik.

(b) Wir versuchen nun, einen kanonischen LR-Parser zu konstruieren und berechnen dazu zunächst die LR(1)-Elemente der Grammatik. Die Vorausschau-Mengen werden wieder durch Auflisten der Symbole rechts vom Kern notiert.

$0 \xrightarrow{S} 1$

0: $S' \to \cdot S,\$$ $S \to \cdot \mathbf{a}A\mathbf{b},\$$ $S \to \cdot \mathbf{c}B\mathbf{b},\$$ $S \to \cdot \mathbf{a}B\mathbf{d},\$$ $S \to \cdot \mathbf{c}A\mathbf{d},\$$	1: $S' \to S\cdot,\$$

$0 \xrightarrow{\mathbf{a}} 2$ $\quad$ $0 \xrightarrow{\mathbf{c}} 3$

2: $S \to \mathbf{a}\cdot A\mathbf{b},\$$ $S \to \mathbf{a}\cdot B\mathbf{d},\$$ $A \to \cdot\mathbf{e},\mathrm{b}$ $B \to \cdot\mathbf{e},\mathrm{d}$	3: $S \to \mathbf{c}\cdot B\mathbf{b},\$$ $S \to \mathbf{c}\cdot A\mathbf{d},\$$ $A \to \cdot\mathbf{e},\mathrm{d}$ $B \to \cdot\mathbf{e},\mathrm{b}$

$2 \xrightarrow{A} 4$ $\quad$ $2 \xrightarrow{B} 5$

4: $S \to \mathbf{a}A\cdot\mathbf{b},\$$	5: $S \to \mathbf{a}B\cdot\mathbf{d},\$$

$2 \xrightarrow{\mathbf{e}} 6$ $\quad$ $3 \xrightarrow{B} 7$

6: $A \to \mathbf{e}\cdot,\mathrm{b}$ $B \to \mathbf{e}\cdot,\mathrm{d}$	7: $S \to \mathbf{c}B\cdot\mathbf{b},\$$

$3 \xrightarrow{A} 8$ $\quad$ $3 \xrightarrow{\mathbf{e}} 9$

8: $S \to \mathbf{c}A\cdot\mathbf{d},\$$	9: $A \to \mathbf{e}\cdot,\mathrm{d}$ $B \to \mathbf{e}\cdot,\mathrm{b}$

$4 \xrightarrow{\mathbf{b}} 10$ $\quad$ $5 \xrightarrow{\mathbf{d}} 11$

10: $S \to \mathbf{a}A\mathbf{b}\cdot,\$$	11: $S \to \mathbf{a}B\mathbf{d}\cdot,\$$

$7 \xrightarrow{\mathbf{b}} 12$ $\quad$ $8 \xrightarrow{\mathbf{d}} 13$

12: $S \to \mathbf{c}B\mathbf{b}\cdot,\$$	13: $S \to \mathbf{c}A\mathbf{d}\cdot,\$$

Da keine Konflikte auftreten, ist die Grammatik vom Typ LR(1). Wir erstellen die Steuertabelle unter Verwendung des Algorithmus 3.21, dessen Anweisung 3.2 gemäß **Kurstext** modifiziert wurde. Dazu nummerieren wir zunächst die Produktionen

(1) S' → S
(2) S → **a***A***b**
(3) S → **c***B***b**
(4) S → **a***B***d**
(5) S → **c***A***d**
(6) A → **e**
(7) B → **e**

	Action						*Goto*			
Zustand	**a**	**b**	**c**	**d**	**e**	$	*S'*	*S*	*A*	*B*
0	s2		s3					1		
1						acc				
2					s6				4	5
3					s9				8	7
4		s10								
5				s11						
6		r6		r7						
7		s12								
8				s13						
9		r7		r6						
10						r2				
11						r4				
12						r3				
13						r5				

VIII Lösungen zu den Selbsttestaufgaben

Literatur

Aho, A.V. und Johnson, S.C. (1974). LR Parsing. *Computing Surveys 6*, 99-124.

Aho, A.V., Lam, M.S., Sethi, R. und Ullman, J.D. (2006). *Compilers: Principles, Techniques, and Tools*. 2nd Edition, Addison-Wesley, Reading, MA.

DeRemer, F. (1969). *Practical Translators for LR(k) Languages*. Ph.D. Thesis, M.I.T., Cambridge, MA.

DeRemer, F. (1971). Simple LR(*k*) Grammars. *Communications of the ACM 14*, 453-460.

DeRemer, F. und Pennello, T. (1982). Efficient Computation of LALR(1) Look-ahead Sets. *Transactions on Programming Languages and Systems 4*, 615-649.

Floyd, R.W. (1963). Syntactic Analysis and Operator Precedence. *Journal of the ACM 10*, 316-333.

Johnson, S.C. (1975). Yacc – Yet Another Compiler-Compiler. Technical Report 32, AT & T Bell Laboratories, Murray Hill, NJ.

Knuth, D.E. (1965). On the Translation of Languages from Left to Right. *Information and Control 8*, 607-639.

Korenjak, A.J. (1969). A Practical Method for Constructing LR(*k*) Processors. *Communications of the ACM 12*, 613-623.

Kristensen, B.B. und Madsen, O.L. (1981). Methods for Computing LALR(*k*) Lookahead. *Transactions on Programming Languages and Systems 3*, 60-82.

Levine, J.R., Mason, T. und Brown, D. (1992). *lex & yacc*. 2nd Edition, O'Reilly & Associates, Sebastopol.

Parsons, T.W. (1992). *Introduction to Compiler Construction*. Computer Science Press, New York.

Pratt, V.R. (1973). Top-down Operator Precedence. *1st ACM Symposium on Principles of Programming Languages*, S. 41-51.

Sippu, S. und Soisalon-Soininen, E. (1988). *Parsing Theory. Vol. I: Languages and Parsing*. EATCS Monographs on Theoretical Computer Science 15, Springer-Verlag, Berlin.

Sippu, S. und Soisalon-Soininen, E. (1990). *Parsing Theory. Vol. II: LR(k) and LL(k) Parsing*. EATCS Monographs on Theoretical Computer Science 20, Springer-Verlag, Berlin.

Sudkamp, T. A. (2005). *Languages and Machines*: An Introduction to the Theory of Computer Science. 3rd Edition, Addison-Wesley, Reading, MA.

Index