```
            else
              error("*␣expected");
          break;
          default:
            if(nextsym == "/")
              scan();
            else
              error("/␣expected");
        }

        F();
      }
    }

    void F() {
      switch (nextsym) {
        case "(":
          E();
          if(nextsym == ")")
            scan();
          else
            error(")␣expected");
        default:
          if(nextsym == "id")
            scan();
          else
            error("id␣expected");
      }
    }

    void parser() {
      scan();
      S();
      if(nextsym == "#")
        accept();
      else
        error("#␣expected");
    }
```

Some inefficiencies result from the schematic generation of this parser program. A more sophisticated generation scheme will avoid most of these inefficiencies.

3.4 Bottom-up Syntax Analysis

3.4.1 Introduction

Bottom-up parsers read their input like *top-down* parsers from left to right. They are pushdown automata that can essentially do two kinds of operations:

- Read the next input symbol (*shift*), and
- Reduce the right side of a production $X \to \alpha$ at the top of the stack by the left side X of the production (*reduce*).

Because of these operations they are called *shift-reduce* parsers. *Shift-reduce* parsers are right parsers; they output the application of a production when they do a reduction. The result of the successful analysis of an input word is a rightmost derivation in reverse order because *shift-reduce* parsers always reduce at the top of the stack.

A *shift-reduce* parser must never miss a *required* reduction, that is, cover it in the stack by a newly read input symbol. A reduction is *required*, if no rightmost derivation to the start symbol is possible without it. A right side covered by an input symbol will never reappear at the top of the stack and can therefore never be reduced. A right side at the top of the stack that must be reduced to obtain a derivation is called a *handle*.

Not all occurrences of right sides that appear at the top of the stack are handles. Some reductions performed at the top of the stack lead into dead ends, that is, they can not continued to a reverse rightmost derivation although the input is correct.

Example 3.4.1 Let G_0 be again the grammar for arithmetic expressions with the productions:

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

Table 3.5 shows a successful *bottom-up* analysis of the word $\mathsf{Id} * \mathsf{Id}$ of G_0. The third column lists actions that were also possible, but would lead into dead ends. In the third step, the parser would miss a required reduction. In the other two steps, the alternative reductions would lead into dead ends, that is, not to right sentential forms. □

Stack	input	Erroneous alternative actions
	$\mathsf{Id} * \mathsf{Id}$	
Id	$* \mathsf{Id}$	
F	$* \mathsf{Id}$	Reading of $*$ misses a required reduction
T	$* \mathsf{Id}$	reduction of T to E leads into a dead end
$T\,*$	Id	
$T * \mathsf{Id}$		
$T * F$		reduction of F to T leads into a dead end
T		
E		
S		

Table 3.5. A successful analysis of the word $\mathsf{Id} * \mathsf{Id}$ together with potential dead ends.

Bottom-up parsers construct the parse tree from the *bottom up*. They start with the leaf word of the parse tree, the input word, and construct for ever larger parts of the read input subtrees of the parse tree by attaching the subtrees for the right side α of a production $X \to \alpha$ below a newly created X node upon a reduction by this production. The analysis is successful if a parse tree with root label S, the start symbol of the grammar, has been constructed for the whole input word.

Fig. 3.13 shows some snapshots during the construction of the parse tree according to the derivation shown in Table 3.5. The tree on the left contains all nodes that can be created when the input Id has been read. The sequence of three trees in the middle represents the state before the handle $T * F$ is being reduced, while the tree on the right shows the complete parse tree.

3.4.2 $LR(k)$ Parsers

This section presents the most powerful deterministic method that works *bottom-up*, $LR(k)$ analysis. The letter L says that the parsers of this class read their input from left to right, The R characterizes

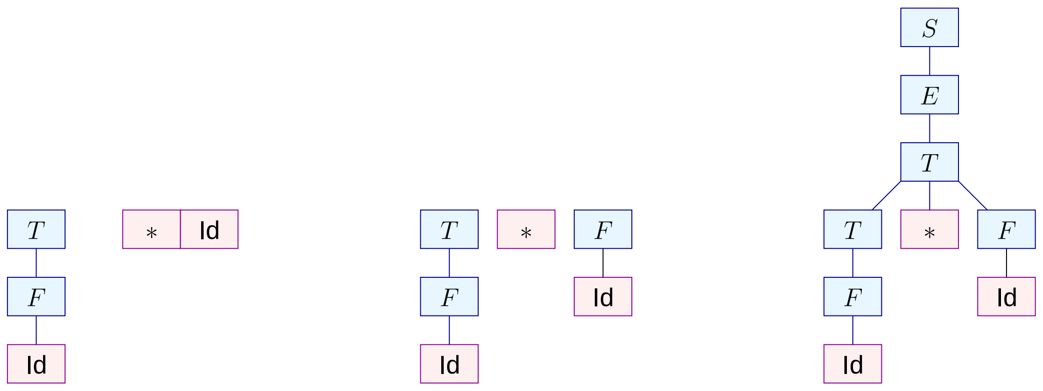

Fig. 3.13. Construction of the parse tree after reading the first symbol, Id, together with the remaining input, before the reduction of the handle $T * F$, and the complete parse tree.

them as Right parser; k is the length of the considered lookahead.

We start again with the item-pushdown automaton P_G for a context-free grammar G and transform it into a *shift-reduce* parser. Let us look back at what we did in the case of *top-down* analysis. Sets of lookahead words were computed from the grammar, which were used to select the right alternative for a nonterminal at *expansion transitions* of P_G. So, the $LL(k)$ parser decides about the alternative for a nonterminal at the earliest possible time, when the nonterminal has to be expanded. $LR(k)$ parsers follow a different strategy; they pursue *all* possibilities to expand and to read in *parallel*.

A decision has to be taken when one of the possibilities to continue asks for a reduction. What is there to decide? There could be several productions by which to reduce, and a shift could be possible in addition to a reduction. The parser uses the next k symbols to take its decision.

In this section, first an $LR(0)$ parser is developed, which does not yet consider any lookahead. Section 3.4.3 presents the *canonical* $LR(k)$ parser. In Section 3.4.3, less powerful variants of $LR(k)$ are described, which are often powerful enough for practice. Finally, Section 3.4.4 describes a error recovery method for $LR(k)$. Note that all context-free grammars are assumed to be reduced of non-productive and unreachable nonterminals and extended by a new start symbol.

The Characteristic Finite-state Machine to a Context-free Grammar

We attempt to represent P_G by a non-deterministic finite-state machine, its *characteristic finite-state machine*, $ch(G)$. Since P_G is a pushdown automaton, this cannot easily work. An additional specification of actions on the stack is necessary. These are associated with some states and some transitions of $ch(G)$.

Our goal is to arrive at a pushdown automaton who pursues all potential expansion and read transitions of the item pushdown-automaton in parallel and only at reduction decides which production is the one to select. We define the *characteristic* finite-state machine $ch(G)$ to a reduced context-free grammar G. The states of the characteristic finite-state machine $ch(G)$ are the items $[A \to \alpha.\beta]$ of the grammar G, that is, the states of the item pushdown-automaton P_G. The set of input symbols of the characteristic finite-state machine $ch(G)$ is $V_T \cup V_N$, its initial state is the start item $[S' \to .S]$ of the item pushdown-automaton P_G. The final states of the characteristic finit-state machine are the complete items $[X \to \alpha.]$. Such a final state signals that the word just read corresponds to a stack contents of the item pushdown-automaton in which a reduction with the production $A \to \alpha$ can be performed. The transition relation $\varDelta$ of the characteristic finite-state machine consists of the transitions:

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

Reading a terminal symbols a in $\mathsf{char}(G)$ corresponds to a *shift* transition of the item pushdown-automaton under a. ε transitions of $\mathsf{char}(G)$ correspond to the expansion transitions of the item pushdown-automaton. When $\mathsf{char}(G)$ reaches a final state $[X \to \alpha.]$ P_G undertakes the following actions: it removes the item $[X \to \alpha.]$ on top of its stack and makes a transition under X from the new state that has appears on top of the stack. This is a reduction move of the item pushdown-automaton P_G.

Example 3.4.2 Let G_0 again be the grammar for arithmetic expressions with the productions

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

Fig. 3.14 shows the characteristic finite-state machine to grammar G_0. □

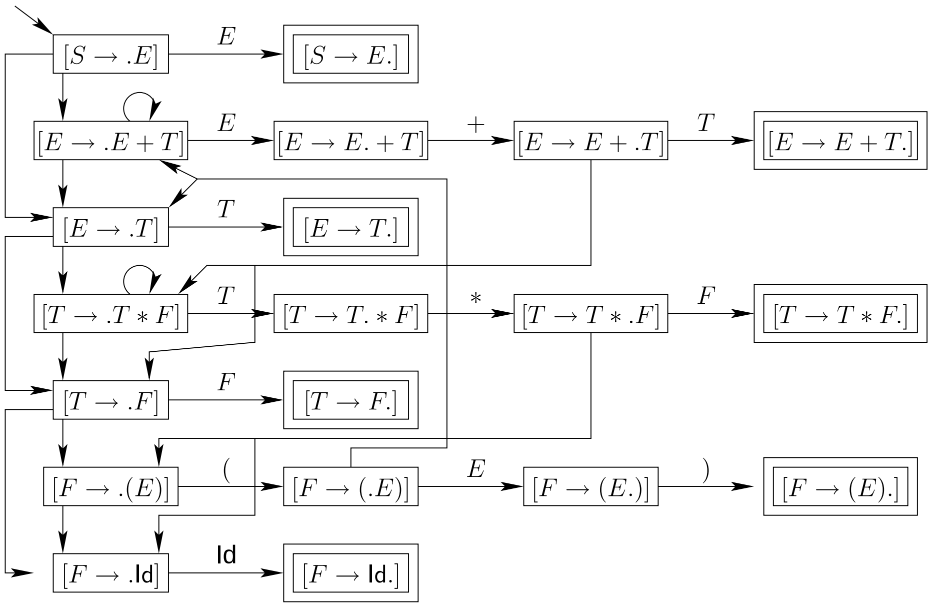

Fig. 3.14. The characteristic finite-state machine $\mathsf{char}(G_0)$ for the grammar G_0.

The following theorem clarifies the exact relation between the characteristic finite-state machine and the item pushdown automaton:

Theorem 3.4.1 Let G be a context-free grammar and $\gamma \in (V_T \cup V_N)^*$. The following three statements are equivalent:

1. There exists a computation $([S' \to .S], \gamma) \vdash^*_{\mathsf{char}(G)} ([A \to \alpha.\beta], \varepsilon)$ of the characteristic finite-state machine $\mathsf{char}(G)$.
2. There exists a computation $(\rho\,[A \to \alpha.\beta], w) \vdash^*_{P_G} ([S' \to S.], \varepsilon)$ of the item pushdown-automaton P_G such that $\gamma = \mathsf{hist}(\rho)\ \alpha$ holds.
3. There exists a rightmost derivation $S' \underset{rm}{\overset{*}{\Longrightarrow}} \gamma' A w \underset{rm}{\Longrightarrow} \gamma' \alpha\beta w$ with $\gamma = \gamma'\alpha$. □

The equivalence of statements (1) and (2) means that words that lead to an item of the characteristic finite-state machine $\mathsf{char}(G)$ are exactly the histories of stack contents of the item pushdown-automaton P_G whose topmost symbol is this item and from which P_G can reach one of its final states assuming appropriate input w. The equivalence of statements (2) and (3) means that an accepting computation of the item pushdown-automaton for an input word w that starts with a stack contents ρ corresponds to a rightmost derivation that leads to a sentential form αw where α is the history of the stack contents ρ.

We introduce some terminology before we prove Theorem 3.4.1. For a rightmost derivation $S' \underset{rm}{\overset{*}{\Longrightarrow}} \gamma' A v \underset{rm}{\Longrightarrow} \gamma\alpha v$ and a production $A \to \alpha$ we call α the *handle* of the right sentential form $\gamma\alpha v$. Is the right side $\alpha = \alpha'\beta$, the prefix $\gamma = \gamma'\alpha'$ is called a *reliable prefix* of G for the item $[A \to \alpha'.\beta]$. The item $[A \to \alpha.\beta]$ is *valid* for γ. Theorem 3.4.1, thus, means, that the set of words under which the characteristic finite-state machine reaches an item $[A \to \alpha'.\beta]$ is exactly the set of reliable prefixes for this item.

Example 3.4.3 For the grammar G_0 we have:

right sentential form	handle	reliable prefixess	reason
$E + F$	F	$E,\ E+,\ E+F$	$S \underset{rm}{\Longrightarrow} E \underset{rm}{\Longrightarrow} E+T \underset{rm}{\Longrightarrow} E+F$
$T * \mathsf{Id}$	Id	$T,\ T*,\ T*\mathsf{Id}$	$S \underset{rm}{\overset{3}{\Longrightarrow}} T*F \underset{rm}{\Longrightarrow} T*\mathsf{Id}$

□

In a non-ambiguous grammar, the handle of a right sentential form is the uniquely determined word that the *bottom-up* parser should replace by a nonterminal in the next reduction step to arrive at a rightmost derivation. A reliable prefix is a prefix of a right sentential form that does not properly extend beyond the handle.

Example 3.4.4 We give two reliable prefixes of G_0 and some items that are valid for them.

relaible prefix	valid item	reason
$E+$	$[E \to E + .T]$	$S \underset{rm}{\Longrightarrow} E \underset{rm}{\Longrightarrow} E+T$
	$[T \to .F]$	$S \underset{rm}{\overset{*}{\Longrightarrow}} E+T \underset{rm}{\Longrightarrow} E+F$
	$[F \to .\mathsf{Id}]$	$S \underset{rm}{\overset{*}{\Longrightarrow}} E+F \underset{rm}{\Longrightarrow} E+\mathsf{Id}$
$(E+($	$[F \to (.E)]$	$S \underset{rm}{\overset{*}{\Longrightarrow}} (E+F) \underset{rm}{\Longrightarrow} (E+(E))$
	$[T \to .F]$	$S \underset{rm}{\overset{*}{\Longrightarrow}} (E+(.T) \underset{rm}{\Longrightarrow} (E+(F))$
	$[F \to .\mathsf{Id}]$	$S \underset{rm}{\overset{*}{\Longrightarrow}} (E+(F) \underset{rm}{\Longrightarrow} (E+(\mathsf{Id}))$

□

Has, in the attempt to construct a rightmost derivation for a word, the prefix u of the word been reduced to a reliable prefix γ, then each item $[X \to \alpha.\beta]$, valid for γ, describes one possible interpretation of the analysis situation. Thus, there is a rightmost derivation in which γ is prefix of a right sentential form and $X \to \alpha\beta$ is one of the possibly just processed productions. All such productions are candidates for later reductions.

Consider the rightmost derivation

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

It should be extended to a rightmost derivation of a terminal word. This requires that

1. β is derived to a terminal word v, and after that,
2. α is derived to a terminal word u.

Altogether,

$$S' \underset{rm}{\overset{*}{\Longrightarrow}} \gamma A w \underset{rm}{\Longrightarrow} \gamma\alpha\beta w \underset{rm}{\overset{*}{\Longrightarrow}} \gamma\alpha v w \underset{rm}{\overset{*}{\Longrightarrow}} \gamma u v w \underset{rm}{\overset{*}{\Longrightarrow}} x u v w$$

We now consider this rightmost derivation in the direction of reduction, that is, in the direction in which a *bottom-up* parser constructs it. First, x is reduced to γ in a number of steps, then u to α, then v to β. The valid item $[A \to \alpha.\beta]$ for the reliable prefix $\gamma\alpha$ describes the analysis situation in which the reduction of u to α has already been done, while the reduction of v to β has not yet started. A possible long-range goal in this situation is the application of the production $X \to \alpha\beta$.

We come back to the question which language is accepted by the characteristic finite-state machine of P_G. Theorem 3.4.1 says that chG goes under a reliable prefix into a state that is a valid item for this prefix. Final states, i.e. complete items, are only valid for reliable prefixes where a reduction is possible at their ends.

Proof of Theorem 3.4.1. We do a circular proof $(1) \Rightarrow (2) \Rightarrow (3) \Rightarrow (1)$. Let us first assume $([S' \to .S], \gamma) \vdash^*_{\mathsf{char}(G)} ([A \to \alpha.\beta], \varepsilon)$. By induction over the number n of ε transitions we construct a rightmost derivation $S' \overset{rm}{\underset{*}{\Longrightarrow}} \gamma A w \overset{rm}{\Longrightarrow} \gamma\alpha\beta w$.

Ist $n = 0$, dann ist $\gamma = \varepsilon$ und $[A \to \alpha.\beta] = [S' \to .S]$. Da $S' \overset{rm}{\underset{*}{\Longrightarrow}} S'$ gilt, ist the Behauptung in diesem Fall erf"ullt. Ist $n > 0$, betrachten wir den letzten ε-"Ubergang. Dann l"asst sich the Berechnung of the characteristic automaton zerlegen in:

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

where $\gamma = \gamma'\alpha$. Nach Induktionsannahme gibt es eine rightmost derivation $S' \overset{rm}{\underset{*}{\Longrightarrow}} \gamma'' X w' \overset{rm}{\Longrightarrow} \gamma''\alpha' A\beta' w'$ mit $\gamma' = \gamma''\alpha'$. Da the grammar G reduziert ist, gibt es ebenfalls eine rightmost derivation $\beta' \overset{rm}{\underset{*}{\Longrightarrow}} v$. Deshalb haben wir:

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

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

Nehmen wir an, wir h"atten eine rightmost derivation $S' \overset{rm}{\underset{*}{\Longrightarrow}} \gamma' A w \overset{rm}{\Longrightarrow} \gamma'\alpha\beta w$. Diese Ableitung l"asst sich zerlegen in:

$$S' \overset{rm}{\Longrightarrow} \alpha_1 X_1 \beta_1 \overset{rm}{\underset{*}{\Longrightarrow}} \alpha_1 X_1 v_1 \overset{rm}{\underset{*}{\Longrightarrow}} \ldots \overset{rm}{\underset{*}{\Longrightarrow}} (\alpha_1 \ldots \alpha_n) X_n (v_n \ldots v_1) \overset{rm}{\Longrightarrow} (\alpha_1 \ldots \alpha_n)\alpha\beta(v_n \ldots v_1)$$

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

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

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

F"ur den letzten Schluss betrachten wir einen Kellerinhalt $\rho = \rho'\ [A \to \alpha.\beta]$ mit $(\rho, w) \vdash^*_{K_G} ([S' \to S.], \varepsilon)$. Zuerst "uberzeugen wir uns mit Induktion nach the Anzahl der "Uberg"ange in einer solchen Berechnung, dass ρ' notwendigerweise von the Form:

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

ist for ein $n \geq 0$ and $X_n = A$. Mit Induktion nach n folgt aber, dass $([S' \to .S], \gamma) \vdash^*_{\mathsf{char}(G)} ([A \to \alpha.\beta], \varepsilon)$ gilt for $\gamma = \alpha_1 \ldots \alpha_n\alpha$. Da $\gamma = \mathsf{hist}(\rho)$, gilt auch the Behauptung (1). Damit ist the Beweis vollst"andig. $\square$

The Canonical $LR(0)$ Automaton

In Chapter 2, we presented an algorithm which takes a non-deterministic finite-state machine and constructs an equivalent deterministic finite-state machine. This deterministic finite-state machine pursues all paths in parallel which the non-deterministic automaton could take for a given input. Its states are sets of states of the non-deterministic automaton. This *subset construction* is now applied to the characteristic finite-state machine $\mathsf{char}(G)$ of a context-free grammar G. The resulting deterministic finite-state machine is called the *canonical* $LR(0)$ automaton for G and denote it by $LR_0(G)$.

Example 3.4.5 The canonical $LR(0)$ automaton for the context-free grammar G_0 of Example 3.2.2 on page 39 is obtained by the application of the subset construction to the characteristic finite-state machine $\mathsf{char}(G_0)$ of Fig. 3.14 on page 82. It is shown in Fig. 3.15 on page 85. It states are:

$$\begin{aligned}
S_0 = \{ & [S \to .E], [E \to .E+T], [E \to .T], [T \to .T*F], [T \to .F], [F \to .(E)], [F \to .\mathsf{Id}] \} \\
S_1 = \{ & [S \to E.], [E \to E.+T] \} \\
S_2 = \{ & [E \to T.], [T \to T.*F] \} \\
S_3 = \{ & [T \to F.] \} \\
S_4 = \{ & [F \to (.E)], [E \to .E+T], [E \to .T], [T \to .T*F] [T \to .F] [F \to .(E)] [F \to .\mathsf{Id}] \} \\
S_5 = \{ & [F \to \mathsf{Id}.] \} \\
S_6 = \{ & [E \to E+.T], [T \to .T*F], [T \to .F], [F \to .(E)], [F \to .\mathsf{Id}] \} \\
S_7 = \{ & [T \to T*.F], [F \to .(E)], [F \to .\mathsf{Id}] \} \\
S_8 = \{ & [F \to (E.)], [E \to E.+T] \} \\
S_9 = \{ & [E \to E+T.], [T \to T.*F] \} \\
S_{10} = \{ & [T \to T*F.]\} \\
S_{11} = \{ & [F \to (E).] \} \\
S_{12} = & \emptyset
\end{aligned}$$

□

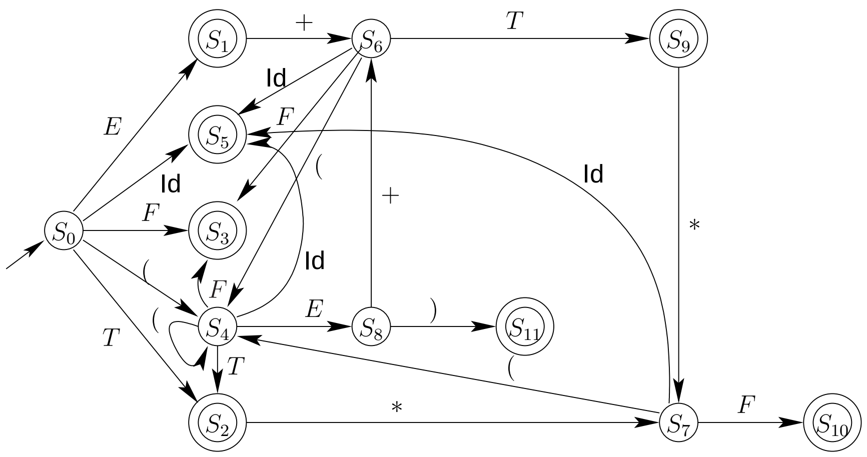

Fig. 3.15. The transition diagram of the $LR(0)$ automaton for the grammar G_0 obtained from the characteristic finite-state machine$\mathsf{char}(G_0)$ in Fig. 3.14. The error state $S_{12} = \emptyset$ and all transitions into it are left out.

The canonical $LR(0)$ automaton $LR_0(G)$ to a context-free grammar G has some interesting properties. Let $LR_0(G) = (Q_G, V_T \cup V_N, \Delta_G, q_{G,0}, F_G)$, and let $\Delta_G^* : Q_G \times (V_T \cup V_N)^* \to Q_G$ be the lifting of the transition function Δ_G from symbols to words. We then have:

1. $\Delta_G^*(q_{G,0}, \gamma)$ is the set of all items in $\mathcal{I}_G$ for which γ is a reliable prefix.
2. $L(LR_0(G))$ is the set of all reliable prefixes for complete items $[A \to \alpha.] \in \mathcal{I}_G$.

Reliable prefixes are prefixes of right-sentential forms, as they occur during the reduction of an input word. When a reduction is possible that will again lead to a right sentential-form This can only happen at the right end of this sentential form. An item valid for a reliable prefix describes one possible interpretation of the actual analysis situation.

Example 3.4.6 $E+F$ is a reliable prefix for the grammar G_0. The state $\Delta^*_{G_0}(S_0, E+F) = S_3$ is also reached by the following reliable prefixes:

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

The state S_6 in the canonical $LR(0)$ automaton to G_0 contains all valid items for the reliable prefix $E+$, namely the items

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

For $E+$ is a prefix of the right sentential form $E+T$:

$$\begin{array}{lccccc} & S \underset{rm}{\Longrightarrow} E \underset{rm}{\Longrightarrow} & E+T & \underset{rm}{\Longrightarrow} & E+F & \underset{rm}{\Longrightarrow} & E+\mathsf{Id} \\ & & \uparrow & & \uparrow & & \uparrow \\ \text{Valid are for instance} & & [E \to E+.T] & & [T \to .F] & & [F \to .\mathsf{Id}] \end{array}$$

□

The canonical $LR(0)$ automaton $LR_0(G)$ to a context-free grammar G is a deterministic finite-state machine that accepts the set of reliable prefixes to complete items. In this way, it identifies positions for reduction, and therefore offers itself for the construction of a right parser. Instead of items (as the item-pushdown automaton) this parser stores on its stack states of the canonical $LR(0)$ automaton, that is *sets* of items. The underlying pushdown automata P_0 is defined as the tuple $K_0 = (Q_G \cup \{f\}, V_T, \Delta_0, q_{G,0}, \{f\})$. The set of states is the set Q_G of states of the canonical $LR(0)$ automaton $LR_0(G)$, extended by a new state f, the final state. The initial state of P_0 is identical to the initial state $q_{G,0}$ of $LR_0(G)$; The transition relation Δ_0 consists of the following kinds of transitions:

Read: $(q, a, q\,\delta_G(q,a)) \in \Delta_0$, if $\delta_G(q,a) \neq \emptyset$. This transition reads the next input symbol a and pushes the successor state q under a onto the stack. It can only be taken if at least one item of the form $[X \to \alpha.a\beta]$ is contained in q.

Reduce: $(qq_1 \ldots q_n, \varepsilon, q\,\delta_G(q,X)) \in \Delta$ if $[X \to \alpha.] \in q_n$ holds with $|\alpha| = n$. The complete item $[X \to \alpha.]$ in the topmost stack entry signals a potential reduction. As many entries are removed from the top of the stack as the length of the right side indicates. After that, the X successor of the new topmost stack entry is pushed onto the stack.
Fig. 3.16 shows a part of the transition diagram of a $LR(0)$ automaton $LR_0(G)$ that demonstrates this situation. The α path in the transition diagram corresponds to $|\alpha|$ entries on top of the stack. These entries are removed at reduction. The new actual state, previously below these removed entries, has a transition under X, which is now taken.

Finish: $(q_{G,0}\,q, \varepsilon, f)$ if $[S' \to S.] \in q$. This transition is the reduction transition to the production $S' \to S$. The property $[S' \to S.] \in q$ signals that a word was successfully reduced to the start symbol. This transition empties the stack and inserts the final state f.

The special case $[X \to .]$ merits special consideration. According to our description, $|\varepsilon| = 0$ topmost stack entries need to be removed from the stack upon this reduction, and a transition from the new, and old, actual state q under X should be taken, and the state $\Delta_G(q,X)$ is pushed onto the stack. This transition is possible since by construction it holds that with the item $[\cdots \to \cdots .X \cdots]$ also the item $[X \to .\alpha]$ is contained in state q for each right side α of nonterminal X. In the special case of a ε production, the actual state q contains together with the item $[\cdots \to \cdots .X \cdots]$ also the complete item $[X \to .]$. This latter reduction transition *extends the length* of the stack.

The construction of $LR_0(G)$ guarantees that for each non-initial and non-final state q there exists exactly one entry symbol under which the automaton can make a transition into q. The stack contents $q_0, \ldots, q_n$ mit $q_0 = q_{G,0}$ corresponds therefore to a uniquely determined word $\alpha = X_1 \ldots X_n \in (V_T \cup V_N)^*$ for which $\Delta_G(q_i, X_{i+1}) = q_{i+1}$ holds. This word α is a reliable prefix, and q_n is the set of all items valid for α.

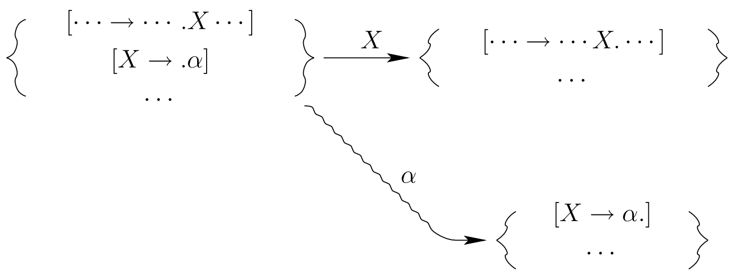

Fig. 3.16. Part of the transition diagram of a canonical $LR(0)$ automaton.

The pushdown automaton P_0 just constructed is not necessarily deterministic. There are two kinds of conflicts that cause non-determinism:

shift-reduce conflict: a state q allows a read transition under a symbol $a \in V_T$ as well as a reduce or finish transition, and
reduce-reduce conflict: a state q permits reduction transitions according to two different productions.

In the first case, the actual state contains at least one item $[X \to \alpha.a\beta]$ and at least one complete item $[Y \to \gamma.]$; in the second case, q contains two different complete items $[Y \to \alpha.]$, $[Z \to \beta.]$. A state q of the $LR(0)$ automaton with one of these properties is called $LR(0)$ *inadequate*. Otherwise, we call q $LR(0)$ *adequate*. Es gilt:

Lemma 3.4. For an $LR(0)$ *adequate* state q there are three possibilities:

1. The state q contains no complete item.
2. The state q consists of exactly one complete item $[A \to \alpha.]$;
3. The state q contains exactly one complete item $[A \to .]$, and all non-complete items in q are of the form $[X \to \alpha.Y\beta]$, where all rightmost derivations for Y that lead to a terminal word are of the form:
$$Y \underset{rm}{\overset{*}{\Longrightarrow}} Aw \underset{rm}{\Longrightarrow} w$$
for a $w \in V_T^*$. $\square$

Inadequate states of the canonical $LR(0)$ automaton make the pushdown automata P_0 non-deterministic. We obtain deterministic parsers by permitting the parser to look ahead into the remaining input to select the correct action in inadequate states.

Example 3.4.7 The states $S_1,\ S_2$ and S_9 of the canonical $LR(0)$ automaton in Fig. 3.15 are $LR(0)$ inadequate. In state S_1, the parser can reduce the right side E to the left side S (complete item $[S \to E.]$) and it can read the terminal symbol $+$ in the input (item $[E \to E. + T]$). In state S_2 the parser can reduce the right side T to E (complete item $[E \to T.]$) and it can read the terminal symbol $*$ (item $[T \to T. * F]$). In state S_9 finally, the parser can reduce the right side $E + T$ to E (complete item $[E \to E + T.]$), and it can read the terminal symbol $*$ (item $[T \to T. * F]$). $\square$

Direct Construction of the Canonical $LR(0)$ Automaton

The canonical $LR(0)$ automaton $LR_0(G)$ to a context-free grammar G needs not be derived through the construction of the characteristic finite-state machine $\mathsf{char}(G)$ and the subset construction. It can be constructed directly from G. The construction uses a function $\Delta_{G,\varepsilon}$ that adds to each set q of items all items that are reachable by ε transitions of the characteristic finite-state machine. The set $\Delta_{G,\varepsilon}(q)$ is the least solution of the following equation

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

Similar to the function $\mathsf{closure}()$ of the subset construction it can be computed by

```
$\mathbf{set}\langle item\rangle$ closure($\mathbf{set}\langle item\rangle\ q$) {
    $\mathbf{set}\langle item\rangle\ result \leftarrow q$;
    $\mathbf{list}\langle item\rangle\ W \leftarrow$ list_of($q$);
    $symbol\ X$; $\mathbf{string}\langle symbol\rangle\ \alpha$;
    while ($W \neq []$) {
        $item\ i \leftarrow$ hd($W$); $W \leftarrow$ tl($W$);
        switch ($i$) {
        case $[\_ \to \_ .X\ \_]$ : forall ($\alpha : (X \to \alpha) \in P$)
                                    if ($[X \to .\alpha] \notin result$) {
                                        $result \leftarrow result \cup \{[X \to .\alpha]\}$;
                                        $W \leftarrow [X \to .\alpha] :: W$;
                                    }
        default :                  break;
        }
    }
    return $result$;
}
```

where V is the set of symbols $V = V_T \cup V_N$. The set Q_G of states and the transition relation Δ_G are computed by first constructing the initial state $q_{G,0} = \Delta_{G,\varepsilon}(\{[S' \to .S]\})$ and then adding successor states and transitions until all successor states are already in the set of constructed states. To implement it we specialize the function nextState() of the subset construction:

```
$\mathbf{set}\langle item\rangle$ nextState($\mathbf{set}\langle item\rangle\ q, symbol\ X$) {
    $\mathbf{set}\langle item\rangle\ q' \leftarrow \emptyset$;
    $nonterminal\ A$; $\mathbf{string}\langle symbol\rangle\ \alpha, \beta$;
    forall ($A, \alpha, \beta : ([A \to \alpha.X\beta] \in q)$)
        $q' \leftarrow q' \cup \{[A \to \alpha X.\beta]\}$;
    return closure($q'$);
    }
```

As in the subset construction, the set of states *states* and the set of transitions *trans* can be computed iteratively:

```
$\mathbf{list}\langle\mathbf{set}\langle item\rangle\rangle\ \ W;$
$\mathbf{set}\langle item\rangle\ \ q_0 \leftarrow \mathsf{closure}(\{[S' \to .S]\});$
$states \leftarrow \{q_0\};\ \ W \leftarrow [q_0];$
$trans \leftarrow \emptyset;$
$\mathbf{set}\langle item\rangle\ \ q, q';$
$\mathbf{while}\ \ (W \neq [])\ \ \{$
    $q \leftarrow \mathsf{hd}(W);\ \ W \leftarrow \mathsf{tl}(W);$
    $\mathbf{forall}\ (symbol\ X)\ \ \{$
        $q' \leftarrow \mathsf{nextState}(q, X);$
        $trans \leftarrow trans \cup \{(q, X, q')\};$
        $\mathbf{if}\ \ (q' \notin states)\ \ \{$
            $states \leftarrow states \cup \{q'\};$
            $W \leftarrow q' :: W;$
        $\}$
    $\}$
$\}$
```

3.4.3 $LR(k)$: Definition, Properties, and Examples

We call a context-free grammar G an $LR(k)$-grammar, if in each of its rightmost derivations $S' = \alpha_0 \underset{rm}{\Longrightarrow} \alpha_1 \underset{rm}{\Longrightarrow} \alpha_2 \cdots \underset{rm}{\Longrightarrow} \alpha_m = v$ and each right sentential forms α_i occurring in the derivation

- the handle can be localized, and
- the production to be applied can be determined

by considering α_i from the left to at most k symbols following the handle. In an $LR(k)$-grammar, the decomposition of α_i into $\gamma\beta w$ and the determination of $X \to \beta$, such that $\alpha_{i-1} = \gamma X w$ holds is uniquely determined by $\gamma\beta$ and $w|_k$. Formally, we call G an $LR(k)$-grammar if

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

Example 3.4.8 Let G be the grammar with the productions

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

Then $L(G) = \{a^n 0 b^n \mid n \geq 0\} \cup \{a^n 1 b^{2n} \mid n \geq 0\}$. We know already that G is for no $k \geq 1$ an $LL(k)$-grammar. Grammar G is an $LR(0)$-grammar, though.

The right sentential forms of G have the form

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

for $n \geq 0$. The handles are always underlined. Two different possibilities to reduce exist only in the case of right sentential forms $a^n aAbb^n$ and $a^n aBbbb^{2n}$ One could reduce $a^n aAbb^n$ to $a^n Ab^n$ and to $a^n aSbb^n$. The first choice belonged to the rightmost derivation

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

the second to no rightmost derivation. The prefix a^n of $a^n Ab^n$ uniquely determines, whether A is the handle, namely in the case $n = 0$, or whether aAb is the handle, namely in the case $n > 0$. The right sentential forms $a^n Bb^{2n}$ are handled analogously. □

Example 3.4.9 The grammar G_1 with the productions

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

and the language $L(G_1) = \{ab^{2n+1}c \mid n \geq 0\}$ is an $LR(0)$-grammar. In a right sentential form $aAbbb^{2n}c$ only the reduction to $aAb^{2n}c$ is possible as part of a rightmost derivation. The prefix $aAbb$ uniquely determines this. For the right sentential form $abb^{2n}c$, b is the handle, and the prefix ab uniquely determines this. □

Example 3.4.10 The grammar G_2 with the productions

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

and the language $L(G_2) = L(G_1)$ is an $LR(1)$-grammar. The critical right sentential forms have the form $ab^n w$. If $1 : w = b$, the handle lies in w; if $1 : w = c$, the last b in b^n forms the handle. □

Example 3.4.11 The grammar G_3 with the productions

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

and the language $L(G_3) = L(G_1)$ is not an $LR(k)$-grammar for any $k \geq 0$. For, let k be arbitrary, but fix. Consider the two rightmost derivations

$$\begin{aligned} S &\overset{*}{\underset{rm}{\Longrightarrow}} ab^n A b^n c \underset{rm}{\Longrightarrow} ab^n b b^n c \\ S &\overset{*}{\underset{rm}{\Longrightarrow}} ab^{n+1} A b^{n+1} c \underset{rm}{\Longrightarrow} ab^{n+1} b b^{n+1} c \end{aligned}$$

with $n \geq k$. With the names introduced in the definition of $LR(k)$-grammar, we have $\alpha = ab^n, \beta = b, \gamma = ab^{n+1}, w = b^n c, y = b^{n+2}c$. Here $w|_k = y|_k = b^k$. $\alpha \neq \gamma$ implies that G_3 can be no $LR(k)$-grammar. □

The following theorem clarifies the relation between the definition of $LR(0)$-grammar and the properties of the canonic $LR(0)$ automaton.

Theorem 3.4.2 A context-free grammar G is an $LR(0)$-grammar if and only if the canonical $LR(0)$ automaton for G has no $LR(0)$-inadequate states.

Proof: $"\Rightarrow"$ Let G eine $LR(0)$-grammar, and nehmen wir an, der canonical $LR(0)$ automaton $LR_0(G)$ habe einen einen $LR(0)$-inadequaten state p.
Fall 1: The state p hat einen *reduce-reduce*-conflict, d.h. p enth"alt zwei verschiedene items $[X \rightarrow \beta.], [Y \rightarrow \delta.]$. Dem state p zugeordnet ist eine nichtleere Menge von reliable prefixesn. Let $\gamma = \gamma'\beta$ ein solches reliable prefix. Weil beide items valid for γ sind, gibt es rightmost derivations

$$\begin{aligned} S' &\overset{*}{\underset{rm}{\Longrightarrow}} \gamma' X w \underset{rm}{\Longrightarrow} \gamma'\beta w && \text{und} \\ S' &\overset{*}{\underset{rm}{\Longrightarrow}} \nu Y y \underset{rm}{\Longrightarrow} \nu\delta y && \text{mit} \quad \nu\delta = \gamma'\beta = \gamma \end{aligned}$$

Das ist aber ein Widerspruch zur $LR(0)$-Eigenschaft.
Fall 2: state p hat einen *shift-reduce*-conflict, d.h. p enth"alt items $[X \rightarrow \beta.]$ and $[Y \rightarrow \delta.a\alpha]$. Let γ ein reliable prefix for beide item Weil beide items valid for γ sind, gibt es rightmost derivations

$$\begin{aligned} S' &\overset{*}{\underset{rm}{\Longrightarrow}} \gamma' X w \underset{rm}{\Longrightarrow} \gamma'\beta w && \text{und} \\ S' &\overset{*}{\underset{rm}{\Longrightarrow}} \nu Y y \underset{rm}{\Longrightarrow} \nu\delta a\alpha y && \text{mit} \quad \nu\delta = \gamma'\beta = \gamma \end{aligned}$$

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

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

Weil $y \neq av_1v_2v_3y$ gilt, ist the $LR(0)$-Eigenschaft verletzt.

"$\Leftarrow$" Nehmen wir an, the canonical $LR(0)$ automaton $LR_0(G)$ habe keine $LR(0)$-inadequaten states. Betrachten wir the zwei rightmost derivations:

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

Zu zeigen ist, dass $\alpha = \gamma,\ X = Y,\ x = y$ gelten. Let p the state of the canonical $LR(0)$ automaton nach Lesen von $\alpha\beta$. Dann enth"alt p alle for $\alpha\beta$ valid items . Nach Voraussetzung ist p $LR(0)$-geeignet. Wir unterscheiden zwei F"alle:

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

Fall 2: $\beta = \varepsilon$. Nehmen wir an, the zweite rightmost derivation widerspreche the $LR(0)$-Bedingung. Dann gibt es ein weiteres item $[X \to \delta.Y'\eta] \in p$, so dass $\alpha = \alpha'\delta$ ist. The letzte Anwendung einer production in the unteren rightmost derivation ist the letzte Anwendung einer production in einer terminalen rightmost derivation for Y'. Nach Lemma 3.4 folgt daraus, dass the 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 $\alpha = \alpha'\delta = \gamma$, $Y = X$ and $x = vw = y$ – im Widerspruch zu unserer Annahme. □

Let us conclude. We have seen how to construct the $LR(0)$ automaton $LR_0(G)$ from a given context-free grammar G. This can be done either directly of through the characteristic finite-state machine $\mathsf{char}(G)$. From the deterministic finite-state machine $LR_0(G)$ one can construct a pushdown automata P_0. This pushdown automaton P_0 is deterministic if $LR_0(G)$ does not contain $LR(0)$-inadequate states. Theorem 3.4.2 states this is exactly the case if the grammar G is an $LR(0)$-grammar. We have thereby met a method to generate parsers for $LR(0)$-grammars.

In real life, $LR(0)$-grammars are rather rare. Often lookahead of length $k > 0$ needs to be used to select between the different choices of a parsing situation. In an $LR(0)$ parser, the actual state determines what the next action is, independently of the next input symbols. $LR(k)$ parsers for $k > 0$ have states consisting of sets of items. A different kind of items are used, though, so-called $LR(k)$-items. $LR(k)$-items are context-free items, extended by lookahead words. An $LR(k)$-item is of the form $i = [A \to \alpha.\beta, x]$ for a production $A \to \alpha\beta$ of G and a word $x \in (V_T^k \cup V_T^{<k}\#)$. The context-free item $[A \to \alpha.\beta]$ is called the *core*, the word x the *lookahead* of the $LR(k)$-items i. The set of $LR(k)$-items of grammar G is written as $\mathcal{I}_{G,k}$. The $LR(k)$-item $[A \to \alpha.\beta, x]$ is *valid* for a reliable prefix γ, if there exists a rightmost derivation

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

with $x = (w\#)|_k$. A context-free item $[A \to \alpha.\beta]$ can be understood as an $LR(0)$-item that is extended by lookahead ε.

Example 3.4.12 Consider again grammar G_0. We have:

(1)	$[E \to E + .T,)]$	
	$[E \to E + .T, +]$	are valid $LR(1)$-items for the prefix $(E+$
(2)	$[E \to T., *]$	is not a valid $LR(1)$-item for any reliable prefix.

To see observation (1), consider the two rightmost derivations:

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

Observation (2) follows since the subword $E*$ can occur in no right sentential form. □

The folllowing theorem gives a characterization of the $LR(k)$-property based on valid $LR(k)$-items.

Theorem 3.4.3 Let G be a context-free grammar. For a reliable prefix γ let $It(\gamma)$ be the set of $LR(k)$-items of G that are valid for γ.

The grammar G is an $LR(k)$-grammar if and only if for all reliable prefixes γ and all $LR(k)$-items $[A \to \alpha., x] \in It(\gamma)$ holds:

1. if there is another $LR(k)$-item $[X \to \delta., y] \in It(\gamma)$, then $x \neq y$.
2. is there another $LR(k)$-item $[X \to \delta.a\beta, y] \in It(\gamma)$, then $x \notin \mathsf{first}_k(a\beta) \odot_k \{y\}$. $\square$

Theorem 3.4.3 suggests to define $LR(k)$ adequate and $LR(k)$-inadequate sets of items also for $k > 0$. Let I be a set of $LR(k)$-items. I has a *reduce-reduce*-conflict, if there are $LR(k)$-items $[X \to \alpha., x], [Y \to \beta., y] \in I$ with $x = y$. I has a *shift-reduce*-conflict, if there are $LR(k)$-items $[X \to \alpha.a\beta, x], [Y \to \gamma., y] \in I$ with

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

For $k = 1$ this condition is simplified to $y = a$.

The set I is called $LR(k)$-inadequate, if it has a *reduce-reduce*- or a *shift-reduce*-conflict. Otherwise, we call it $LR(k)$ adequate.

The $LR(k)$-property means that when reading a right sentential form, a candidate for a reduction together with production to be applied can be uniquely determined by the help of the associated reliable prefixes and the k next symbols of the input. However, if we were to tabulate all combinations of reliable prefixes with words of length k this would be infeasible since, in general, there are infinitely many reliable prefixes. In analogy to our way of dealing with $LR(0)$-grammars one could construct a canonical $LR(k)$-automaton. The canonical $LR(k)$-automaton $LR_k(G)$ is a deterministic finite-state machine. Its states are sets of $LR(k)$-items. For each reliable prefix γ the deterministic finite-state machine $LR_k(G)$ determines the set of $LR(k)$-items that are valid for γ. Theorem 3.4.3 helps us in our derivation. It says that for an $LR(k)$-grammar, the set of $LR(k)$-items valid for γ together with the lookahead determines uniquely whether to reduce in the next step, and if so, by which production.

In much the same way as the $LR(0)$ parser stores states of the canonical $LR(0)$ automaton on its stack, the $LR(k)$ parser stores states of the canonical $LR(k)$-automaton on is stack. The selection of the right of several possible actions of the $LR(k)$ parser is controlled by the *action*-table. This table contains for each combination of state and lookahead one of the following entries:

shift:	read the next input symbol;
reduce$(X \to \alpha)$:	reduce by production $X \to \alpha$;
error:	report error
accept:	announce successful end of the parser run

A second table, the *goto*-table, contains the representation of the transition function of the canonic $LR(k)$-automaton $LR_k(G)$. It is consulted after a *shift*-action or a *reduce*-action to determine the new state on top of the stack. Upon a *shift*, it computes the transition under the read symbol out of the actual state. Upon a reduction by $X \to \alpha$, it gives the transition under X out of the state underneath those stack symbols that belong to α. These two tables for $k = 1$ are shown in Fig. 3.17.

The $LR(k)$ parser for a grammar G needs a program that interprets the *action*- and *goto*-table, the *driver*. Again, we consider the case $k = 1$. This is, in principle, sufficient because for each language that has an $LR(k)$-grammar and therefore also an $LR(k)$ parser one can construct an $LR(1)$-grammar and consequently also an $LR(1)$ parser. Let us assume that the set of states of the $LR(1)$ parser were Q. One such driver program then is:

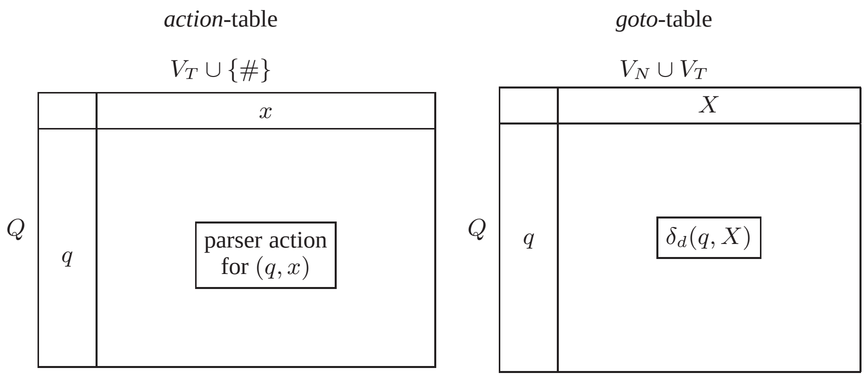

Fig. 3.17. Schematic representation of *action*- and *goto*-table of an $LR(1)$ parser with set of states Q.

```
$\mathbf{list}\langle state\rangle$  $stack \leftarrow [q_0]$;
$terminal$  $buffer \leftarrow$ scan();
$state$  $q$;  $nonterminal$  $X$;  $\mathbf{string}\langle symbol\rangle$  $\alpha$;
while (true) {
    $q \leftarrow$ hd($stack$);
    switch ($action[q, buffer]$) {
    case $shift$ :              $stack \leftarrow goto[q, buffer] :: stack$;
                                $buffer \leftarrow$ scan();
                                break;
    case $reduce(X \to \alpha)$ :  output($X \to \alpha$);
                                $stack \leftarrow$ tl($|\alpha|, stack$);   $q \leftarrow$ hd($stack$);
                                $stack \leftarrow goto[q, X] :: stack$;
                                break;
    case $accept$ :             $stack \leftarrow f ::$ tl($2, stack$);
                                return $accept$;
    case $error$ :              output("..."); goto $err$;
    }
```

The function $\mathbf{list}\langle state\rangle$ tl($\mathbf{int}$ n, $\mathbf{list}\langle state\rangle$ s) returns in its second argument the list s with the topmost n elements removed. As with the driver program for $LL(1)$ parsers, in the case of an error, it jumps to a label *err* at which the code for error handling is to be found.

We present three approaches to construct an $LR(1)$ parser for a context-free grammar G. The most general method is the canonical $LR(1)$-method. For each $LR(1)$-grammar G there exists a canonical $LR(1)$ parser. The number of states of this parser can be large. Therefore, other methods were proposed that have state sets of the size of the $LR(0)$ automaton. Of these we consider the $SLR(1)$- and the $LALR(1)$-method.

The described driver program for $LR(1)$ parsers works for all three parsing methods; the driver interprets the *action*- and a *goto*-table, but their contents are computed in different ways. In consequence, the actions for some combinations of state and lookahead may be different.

Construction of an $LR(1)$ Parser

The $LR(1)$ parser is based on the canonical $LR(1)$-automaton $LR_1(G)$. Its states therefore are sets of $LR(1)$-items. We construct the canonical $LR(1)$-automaton much in the same way as we constructed the canonical $LR(0)$ automaton. The only difference is that $LR(1)$-items are used instead of $LR(0)$-items. This means that the lookahead symbols need to be computed when the closure of a set q of

$LR(1)$-items under ε-transitions is formed. This set is the least solution of the following equation

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

It is computed by the following function

```
$\mathbf{set}\langle item_1\rangle$ $\mathsf{closure}(\mathbf{set}\langle item_1\rangle\ q)$ {
    $\mathbf{set}\langle item_1\rangle\ result \leftarrow q;$
    $\mathbf{list}\langle item_1\rangle\ W \leftarrow \mathsf{list\_of}(q);$
    $nonterminal\ X;\ \mathbf{string}\langle symbol\rangle\ \alpha, \beta;\ terminal\ x, y;$
    while $(W \neq [])$ {
        $item_1\ i \leftarrow \mathsf{hd}(W);\ W \leftarrow \mathsf{tl}(W);$
        switch $(i)$ {
        case $[\_ \to \_\ .X\beta, x]$ :
                    forall $(\alpha : (X \to \alpha) \in P)$
                        forall $(y \in \mathsf{first}_1(\beta) \odot_1 \{x\})$
                            if $([X \to .\alpha, y] \notin result)$ {
                                $result \leftarrow result \cup \{[X \to .\alpha, y]\};$
                                $W \leftarrow [X \to .\alpha, y] :: W;$
                            }
        default : break;
        }
    }
    return $result;$
}
```

where V is the set of all symbols, $V = V_T \cup V_N$. The initial state q_0 of $LR_1(G)$ is

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

We need a function nextState() that computes the successor state to a given set q of LR_1-items and a symbol $X \in V = V_N \cup V_T$. The corresponding function for the construction of $LR_0(G)$ needs to be extended by the compute the lookahead symbols:

```
$set\langle item_1\rangle$ $\mathsf{nextState}(set\langle item_1\rangle\ q, symbol\ X)$ {
    $set\langle item_1\rangle\ q' \leftarrow \emptyset;$
    $nonterminal\ A;\ \mathbf{string}\langle symbol\rangle\ \alpha, \beta;\ terminal\ x;$
    forall $(A, \alpha, \beta, x : ([A \to \alpha.X\beta, x] \in q))$
        $q' \leftarrow q' \cup \{[A \to \alpha X.\beta, x]\};$
    return $\mathsf{closure}(q');$
    }
```

The set of states and the transition relation of the canonical $LR(1)$-automaton is computed in analogy to the canonical $LR(0)$-automaton. The generator starts with the initial state and an empty set of transitions and adds successors states until all successor states are already contained in the set of computed states. The transition function of the canonical $LR(1)$-automaton gives the *goto*-table of the $LR(1)$ parser.

Let us turn to the construction of the *action*-table of the $LR(1)$ parser. No *reduce-reduce*-conflict exists in a state q of the canonical $LR(1)$-automaton with complete $LR(1)$-items $[X \to \alpha., x], [Y \to \beta., y]$ if $x \neq y$. If the $LR(1)$ parser is in state q it will decide to reduce with the production whose lookahead symbol is the next input symbol. If state q contains at the same time a complete $LR(1)$-item $[X \to \alpha., x]$ and an $LR(1)$-item $[Y \to \beta.a\gamma, y]$, it still has no *shift-reduce*-conflict if $a \neq x$. In state q the generated parser will reduce if the next next input symbol is x and shift if it is a. Therefore, the *action*-table can be computed by the following iteration:

```
forall $(state\ \ q)$ {
    forall $(terminal\ \ x)$  $action[q, x] \leftarrow error$;
    forall $([X \to \alpha.\beta, x] \in q)$
        if $(\beta = \varepsilon)$
            if $(X = S' \wedge \alpha = S \wedge x = \#)$  $action[q, \#] \leftarrow accept$;
            else $action[q, x] \leftarrow reduce(X \to \alpha)$;
        else if $(\beta = a\beta')$  $action[q, a] \leftarrow shift$;
}
```

Example 3.4.13 We consider some states of the canonical $LR(1)$-automaton for the context-free grammar G_0. The numbering of states is the same as in Fig. 3.15. To make the representation of sets S of $LR(1)$-items more readable all lookahead symbols in $LR(1)$-items from S with the same kernel $[A \to \alpha.\beta]$ are collected in one lookahead set

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

We represent subsets $\{[A \to \alpha.\beta, x] \mid x \in L\}$ as $[A \to \alpha.\beta, L]$ and obtain

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

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

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

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

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

After the extension by lookahead symbols, the states S_1, S_2 and S_9, which were $LR(0)$ inadequate, have no longer conflicts. In state S'_1 the next input symbol $+$ indicates to shift, the next input symbol $\#$ indicates to reduce. In state S'_2 lookahead symbol $*$ indicates to shift, $\#$ and $+$ to reduce; similarly in state S'_9.

The table 3.6 shows the rows of the *action*-table of the canonical $LR(1)$ parser for the grammar G_0, which belong to the states S'_0, S'_1, S'_2, S'_6 and S'_9. □

$SLR(1)$- and $LALR(1)$ parser

The set of states of $LR(1)$ parsers can become quite large. Therefore, often LR analysis methods are employed that are not as powerful as canonical LR parsers, but have fewer states. Two such LR analysis

	Id	(	)	$*$	$+$	$\#$
S'_0	s	s				
S'_1					s	*acc*
S'_2				s	$r(3)$	$r(3)$
S'_6	s	s				
S'_9				s	$r(2)$	$r(2)$

The used numbering of the productions:

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

Table 3.6. Some rows of the *action*-table of the canonical $LR(1)$ parser for G_0. s stands for *shift*, $r(i)$ for *reduce* by production i, *acc* for *accept*. All empty entries represent *error*.

methods are the $SLR(1)$- (*simple* LR-) and $LALR(1)$- (*lookahead* LR-)methods. Ist $SLR(1)$ parser is a special $LALR(1)$ parser, and each grammar that has an $LALR(1)$ parser is an $LR(1)$-grammar.

The starting point of the construction of $SLR(1)$- and $LALR(1)$ parsers is the canonical $LR(0)$ automaton $LR_0(G)$. The set Q of states and the *goto*-table for these parsers are the set of states and the *goto*-table of the corresponding $LR(0)$ parser. Lookahead is used to resolve conflicts in the states in Q. Let $q \in Q$ be a state of the canonical $LR(0)$ automaton and $[X \to \alpha.\beta]$ an item in q. We denote by $\lambda(q, [X \to \alpha.\beta])$ the lookahead set that is added to the item $[X \to \alpha.\beta]$ in q. The $SLR(1)$-method is different from the $LALR(1)$-method in the definition of the function

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

Relative to such a function λ, the state q of $LR_0(G)$ has a *reduce-reduce*-conflict, if it has different complete items $[X \to \alpha.], [Y \to \beta.] \in q$ with

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

Relative to λ , q has a *shift-reduce*-conflict if it has items $[X \to \alpha.a\beta], [Y \to \gamma.] \in q$ with $a \in \lambda(q, [Y \to \gamma.])$.

If no state of the canonic $LR(0)$ automaton has a conflict, the lookahead sets $\lambda(q, [X \to \alpha.])$ suffice to construct an *action*-table zu.

In $SLR(1)$ parsers, the lookahead sets for items are independent of the states in which they occur; the lookahead only depends on the left side of the production in the item:

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

for alle states q mit $[X \to \alpha.] \in q$. A state q of the canonical $LR(0)$ automaton is called $SLR(1)$-*inadequate* if it contains conflicts with respect to the function λ_S. G is an $SLR(1)$-*grammar* if there are no $SLR(1)$-inadequate states.

Example 3.4.14 We consider again grammar G_0 of Example 3.4.1. Its canonical $LR(0)$ automaton $LR_0(G_0)$ has the inadequate states S_1, S_2 and S_9. We extend the complete items in the states by the follow_1-sets of their left sides to represent the function λ_S in a readable way. Since $\mathsf{follow}_1(S) = \{\#\}$ and $\mathsf{follow}_1(E) = \{\#, +,)\}$ we obtain:

$$\begin{array}{ll}
S''_1 = \{\ [S \to E., \{\#\}], & \text{conflict eliminated,} \\
\qquad\quad [E \to E. + T]\} & \text{da} \quad + \notin \{\#\} \\
 & \\
S''_2 = \{\ [E \to T., \{\#, +,)\}], & \text{conflict eliminated,} \\
\qquad\quad [T \to T. * F]\ \} & \text{da} \quad * \notin \{\#, +,)\} \\
 & \\
S''_9 = \{\ [E \to E + T., \{\#, +,)\}], & \text{conflict eliminated,} \\
\qquad\quad [T \to T. * F]\ \} & \text{da} \quad * \notin \{\#, +,)\}
\end{array}$$

So, G_0 i an $SLR(1)$-grammar and it has an $SLR(1)$ parser. $\square$

The set $\mathsf{follow}_1(X)$ collects all symbols that can follow the nonterminal X in a sentential form of the grammar. Only the follow_1-sets are used to resolve conflicts in the construction of an $SLR(1)$ parser. In many cases this is not sufficient. More conflicts can be resolved if the state is taken into consideration in which the complete item $[X \to \alpha.]$ occurs. The *most precise* lookahead set that considers the state is defined by:

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

Here, q_0 is the initial state, and Δ_G is the transition function of the canonic $LR(0)$ automaton $LR_0(G)$. In $\lambda_L(q, [X \to \alpha.])$ only terminal symbols are contained that can follow X in a right sentential form $\beta X a w$ such that $\beta\alpha$ drives the canonical $LR(0)$ automaton into the state q. We call state q of the canonical $LR(0)$ automaton $LALR(1)$-*inadequate* if it contains conflicts with respect to the function λ_L. The grammar G is an $LALR(1)$-grammar if the canonical $LR(0)$ automaton has no $LALR(1)$-inadequate states.

There always exists an $LALR(1)$ parser to an $LALR(1)$-grammar. The definition of the function λ_L however is not constructive since sets of right sentential forms appear in it that are in general infinite. The sets $\lambda_L(q, [A \to \alpha.\beta])$ can be characterized as the least solution of the following system of equations:

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

The system of equations describes how sets of successor symbols of items in states originate. The first equation says that only $\#$ can follow the start symbol S'. The second class of equations describes that the follow symbols of an item $[A \to \alpha X.\beta]$ in a state q result from the follow symbols after the dot in an item $[A \to \alpha.X\beta]$ in states p from which one can reach q by reading X. The third class of equations formalizes that the follow symbols of an item $[A \to .\alpha]$ in a state q result from the follow symbols of *occurrences* of A in items in q after the dot, that is, from sets $\mathsf{first}_1(\beta) \odot_1 \lambda_L(q, [X \to \gamma.A\beta])$ for items $[X \to \gamma.A\beta]$ in q.

The system of equations for the sets $\lambda_L(q, [A \to \alpha.\beta])$ over the finite subset lattice $2^{V_T \cup \{\#\}}$ can be solved by the iterative method for the computation of least solutions. Considering which nonterminal may produce ε allows us to replace the occurrences of 1-concatenation by unions. We so obtain an equivalent pure union problem that can be solved by the efficient method of Section 3.2.7.

$LALR(1)$ parsers can be constructed in the following, albeit inefficient way: One constructs a canonical $LR(1)$ parser. Consider two $LR(1)$ adequate states p and q where the cores of the items in p are the same as the cores in the items of q, that is, where the difference of the two sets of items consists only in the lookahead sets. Such states p and q are merged to a new state p'. The lookahead sets in the new state p' are obtained as the union of the lookahead sets of items with the same core. The grammar is an $LALR(1)$-grammar if the new states have no conflicts.

```
**list**$\langle$**set**$\langle item\rangle\rangle$  $W$;
**set**$\langle item\rangle$  $q_0 \leftarrow \mathsf{closure}(\{[S' \rightarrow .S]\})$;
$states \leftarrow \{q_0\}$;  $W \leftarrow [q_0]$;
$trans \leftarrow \emptyset$;
**set**$\langle item\rangle$  $q, q'$;
**while** $(W \neq [])$ {
    $q \leftarrow \mathsf{hd}(W)$;  $W \leftarrow \mathsf{tl}(W)$;
    **forall** $(symbol\ X)$ {
        $q' \leftarrow \mathsf{nextState}(q, X)$;
        $trans \leftarrow trans \cup \{(q, X, q')\}$;
        **if** $(q' \notin states)$ {
            $states \leftarrow states \cup \{q'\}$;
            $W \leftarrow q' :: W$;
        }
    }
}
```

A further possibility consists in the modification of Algorithm *LR(1)-GEN*. The conditional statement

if (q' **not in** $states$) $states \leftarrow states \cup \{q'\}$;

is replaced by

if (*exists* q'' **in** $states$ *with samecores*(q', q'')) *merge*$(states, q', q'')$;

where

```
**bool** *samecores*(**set of item** $p$, **set of item** $p'$)
  **if** (*set of cores of* $p$ = *set of cores of* $p'$)
     **return** true;
  **else**
     **return** false;

**void** *merge* (**set of set of item** $states$, **set of item** $p$, **set of item** $p'$)
  $states \leftarrow states \setminus \{p'\} \cup \{[X \rightarrow \alpha.\beta, L_1 \cup L_2] \mid [X \rightarrow \alpha.\beta, L_1] \in p$ *and* $[X \rightarrow \alpha.\beta, L_2] \in p'\}$;
```

Example 3.4.15 The following grammar taken from [ASU86] describes a simplified version of the C assignment statement:

$$\begin{array}{lcl} S' & \rightarrow & S \\ S & \rightarrow & L = R \mid R \\ L & \rightarrow & *R \mid \mathsf{Id} \\ R & \rightarrow & L \end{array}$$

This grammar is not an $SLR(1)$-grammar, but t is a $LALR(1)$-grammar. The states of the canonical $LR(0)$ automaton are given by:

$$
\begin{array}{lll}
S_0 = \{\ [S' \to .S], & S_2 = \{\ [S \to L. = R], & S_6 = \{\ [S \to L = .R], \\
\quad [S \to .L = R], & \quad [R \to L.]\ \} & \quad [R \to .L], \\
\quad [S \to .R], & & \quad [L \to .*R], \\
\quad [L \to .*R], & S_3 = \{\ [S \to R.]\ \} & \quad [L \to .\mathsf{Id}]\ \} \\
\quad [L \to .\mathsf{Id}], & S_4 = \{\ [L \to *.R], & \\
\quad [R \to .L]\ \} & \quad [R \to .L], & S_7 = \{\ [L \to *R.]\ \} \\
 & \quad [L \to .*R], & S_8 = \{\ [R \to L.]\ \} \\
S_1 = \{\ [S' \to S.]\ \} & \quad [L \to .\mathsf{Id}]\ \} & S_9 = \{\ [S \to L = R.]\ \} \\
 & S_5 = \{\ [L \to \mathsf{Id}.]\ \} &
\end{array}
$$

State S_2 is the only $LR(0)$-inadequate state. We have $\mathsf{follow}_1(R) = \{\#, =\}$. This lookahead set for the item $[R \to L.]$ is not sufficient to resolve the *shift-reduce*-conflict in S_2 since the next input symbol $=$ is in the lookahead set. Therefore, the grammar is not an $SLR(1)$-grammar.

The grammar however is a $LALR(1)$-grammar. The transition diagram of its $LALR(1)$ parser is shown in Fig. 3.18. To increase readability, the lookahead sets $\lambda_L(q, [A \to \alpha.\beta])$ were directly associated with the item $[A \to \alpha.\beta]$ of state q. In state S_2, the item $[R \to L.]$ has now the lookahead set $\{\#\}$. The conflict is resolved since this set does not contain the next input symbol $=$. □

3.4.4 Error Handling in LR Parsers

LR parsers like LL parsers have the viable-prefix property. This means that each prefix of the input that could be analyzed by an LR parser without finding an error can be completed to a correct input word, a word of the language. When an LR parser meets a configuration where the combination of state q and input symbol a leads to an entry $action[q, a] = \mathsf{error}$ this is the earliest situation in which an error can be detected. We call this configuration an *error configuration* and q the *error state* of this configuration. There exist a number of error handling methods for LR parsers:

- Forward error recovery: Modifications are made in the remaining input, not in the parse stack.
- Backward error recovery: Modifications are also made in the parse stack.

Let us assume, q were the actual state and a the next symbol in the input. Potential corrections are the following actions: A generalized *shift*(βa) for an item $[A \to \alpha.\beta a \gamma]$ in q, a *reduce* for incomplete items in q, and *skip*.

- The correction *shift*(βa) assumes that the subword for β is missing. It therefore pushes the states that the item-pushdown automaton would run through when reading the word β starting in state q. After that the symbol a is read and the associated *shift*-transition of the parser is performed.
- The correction *reduce*$(A \to \alpha.\beta)$ also assumes that the subword that belongs to β is missing. Therefore $|\alpha|$ many states are removed from the stack. Let p be the newly appeared state on top of the stack. That state is pushed onto the stack that results from p and A according to the *goto*-table.
- The correction *skip* continues with the next Symbol a' in the input.

A simple method for error recovery could look as follows: Let us assume there were no transition under a. If the actual state contains an item $[A \to \alpha.\beta a \gamma]$, the parser could try to restart by reading a. As correction *shift*(βa) is performed. If the symbol does not occur in any right side of an items in q, but as lookahead of a noncomplete item $[A \to \alpha.\beta]$ in q, then the correction *reduce*$(A \to \alpha.\beta)$ could be performed. If several such corrections are possible in q a *plausible* correction is chosen. It could be plausible to choose the operation *shift*(βa) or *reduce*$(A \to \alpha.\beta)$ in which the missing subword β is the shortest. If neither a *shift*- nor a *reduce*-correction is possible the correction *skip* is applied.

Example 3.4.16 Consider the grammar G_0 with the productions

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

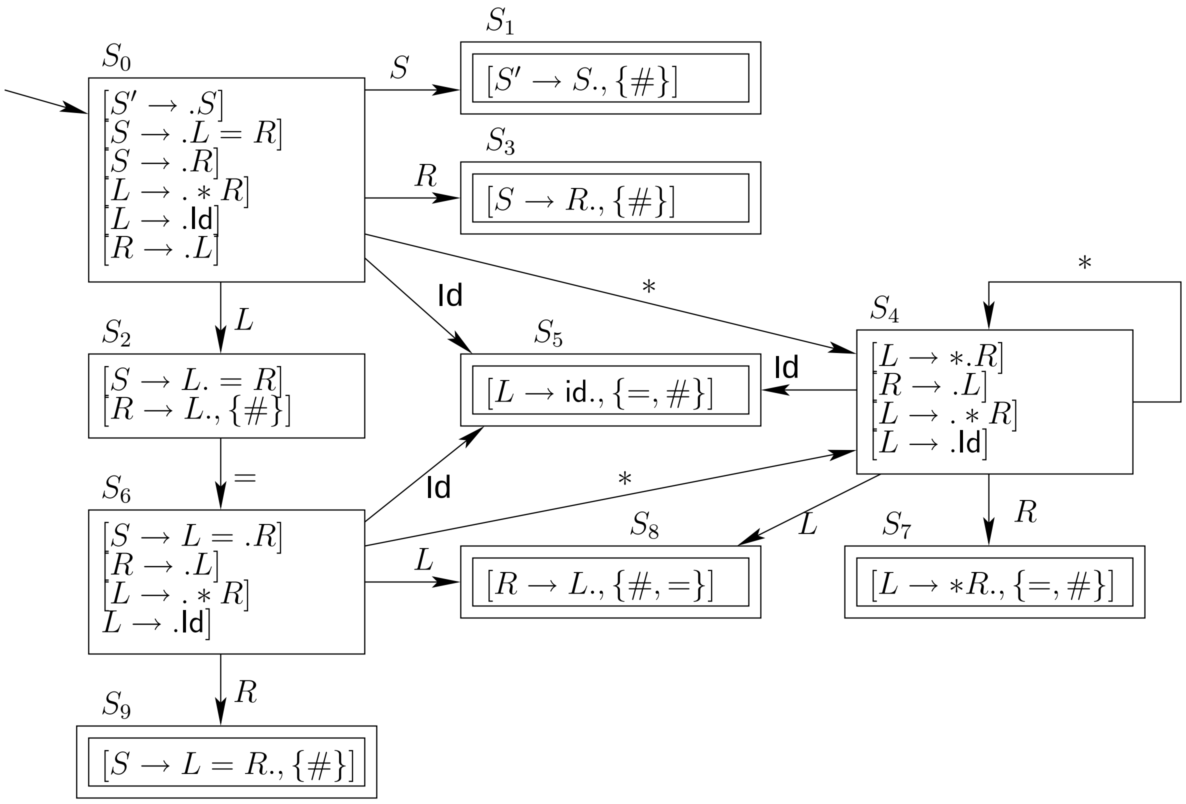

Fig. 3.18. Transition diagram of the $LALR(1)$ parser for the grammar of Example 3.4.15

for which the canonical $LR(0)$ automaton was constructed in Example 3.4.5. As input we choose

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

After reading the prefixes $(\mathsf{Id} \ +$ the stack of an $SLR(1)$ parser contains the sequence of states $S_0S_4S_8S_6$, corresponding to the reliable prefix $(\ E \ +$. The actual state S_6 consists of the items :

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

We consider am $SLR(1)$ parser. Its lookahead sets of the items in S_6 are the follow_1-sets of the left sides, i.,e.

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

Reading $)$ in state S_6 leads to error. However, there are incomplete items in S_6 with lookahead $)$. One of these items is used for reduction. One such item is $[E \to E + .T]$. The reduction produces a new stack content $S_0S_4S_8$ since S_8 is the successor state of S_4 under the left side E. A *shift*-transition reading $)$ is possible in state S_8. This leads to new state S_{11} on top of the stack. A sequence of reductions reaches the final state f. □

This error recovery is a pure forward recovery. It is similar to the one offered by the parser generator CUP for JAVA.

The One-Error-Hypothesis

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

Eine Konfiguration of the LR parsers notieren wir als $(\varphi q, a_i \dots a_n)$, wobei φq der Kellerinhalt ist mit actualm state q, und the restliche input $a_i \dots a_n$. Das verfeinerte Verfahren versucht, zu jeder error configuration $(\varphi q, a_i \dots a_n)$ eine *passende* Konfiguration zu finden, in the eine Fortsetzung the analysis durch Lesen mindestens eines weiteren input symbol m"oglich ist. Eine Konfiguration *passt* zu the error configuration , wenn sie durch m"oglichst wenig Ver"anderungen aus the error configuration hervorgeht. Mit the Annahme the *ein-Fehler-Hypothese* schr"anken wir the zugelassenen Ver"anderungen drastisch ein. The ein-Fehler-Hypothese besagt, dass the Fehler an the gegebenen Stelle durch *ein* fehlendes, *ein* "uberfl"ussiges oder *ein* falsches Symbol an the Fehlerstelle verursacht wurde. Der Fehlerbehandlungsalgorithmus verf"ugt deshalb über eine operation for the Einsetzen, eine operation for the L"oschen und eine operation for the Ersetzen *eines* Symbols.

Let $(\varphi q, a_i \dots a_n)$ eine error configuration . Das Ziel the Fehlerkorrektur mit einer the drei operations lässt sich wie folgt beschreiben:

L"oschen: Finde Kellerinhalte $\varphi' p$ mit

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

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

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

Einf"ugen: Finde ein Symbol a and Kellerinhalte $\varphi' p$ mit

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

The gesuchten Kellerinhalte $\varphi' p$ k"onnen sich dadurch ergeben, dass unter dem jeweils neuen next input symbol reductions m"oglich sind, die in the error configuration nicht m"oglich waren. Eine wichtige Eigenschaft the drei operations ist, dass sie the Terminierung of the Fehlerbehandlungsverfahren garantieren: jeder the drei Schritte stellt im Erfolgsfall den Lesezeiger um mindestens ein Symbol weiter.

Fehlerbehandlungsmethoden mit Zur"ucksetzen erlauben zus"atzlich, eine zuletzt angewandte production the Form $X \to \alpha Y$ r"uckg"angig zu machen and $Y a_i \dots a_n$ als input zu betrachten, wenn the anderen Korrekturversuche gescheitert sind.

Ein naives Verfahren wird the verschiedenen M"oglichkeiten einer Fehlerkorrektur dynamisch, d.h. während of the Parserlaufs durchsuchen, bis eine geeignete Korrektur gefunden ist. Das Überprüfen einer Möglichkeit verlangt eventuell, eine Reihe reductions durchzuführen, gefolgt von einem Test, ob man ein Symbol lesen kann. Bei Misserfolg ist dann the error configuration wiederherzustellen und the nächste Möglichkeit auszprobieren. The Suche nach the *richtigen* Abänderung eines Symbols kann damit sehr teuer sein. Deshalb interessieren wir uns for *Vorberechnungen*, die man bereits zur Generierungszeit of the Parsers durchf"uhren kann, um Sackgassen bei the Fehlerkorrektur schneller zu erkennen. Let $(\varphi q, a_i \dots a_n)$ wieder the error configuration . Betrachten wir the *Einf"ugen* eines Symbols $a \in V_T$. The Fehlerbehandlung kann aus the folgenden Sequenz von Schritten bestehen (siehe Abbildung 3.19 (a)):

(1) eine Folge von reductions unter lookaheadsymbol a, gefolgt von