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:

\[
S \rightarrow E \\
E \rightarrow E + T \mid T \\
T \rightarrow T \cdot F \mid F \\
F \rightarrow (E) \mid \text{Id}
\]

Table 3.5 shows a successful bottom-up analysis of the word \( \text{Id} \ast \text{Id} \) 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. □

<table>
<thead>
<tr>
<th>Stack</th>
<th>input</th>
<th>Erroneous alternative actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Id * Id</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Id</td>
<td>* Id</td>
<td>Reading of * misses a required reduction</td>
</tr>
<tr>
<td>F</td>
<td>* Id</td>
<td>reduction of ( T ) to ( E ) leads into a dead end</td>
</tr>
<tr>
<td>T</td>
<td>* Id</td>
<td></td>
</tr>
<tr>
<td>T * Id</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T * F</td>
<td></td>
<td>reduction of ( F ) to ( T ) leads into a dead end</td>
</tr>
<tr>
<td>T</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3.5. A successful analysis of the word \( \text{Id} \ast \text{Id} \) 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 \( \alpha \) of a production \( X \rightarrow \alpha \) 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 \( \text{Id} \) has been read. The sequence of three trees in the middle represents the state before the handle \( T \ast F \) 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
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 an 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 $[A \rightarrow \alpha, \beta]$ 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 $[S' \rightarrow .S]$ of the item pushdown-automaton $P_G$. The final states of the characteristic finite-state machine are the complete items $[X \rightarrow \alpha, .]$. 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 $A \rightarrow \alpha$ can be performed. The transition relation $\Delta$ of the characteristic finite-state machine consists of the transitions:

- $([X \rightarrow \alpha, Y, \beta], \epsilon, [Y \rightarrow \gamma])$ for $X \rightarrow \alpha Y \beta \in P$, $Y \rightarrow \gamma \in P$
- $([X \rightarrow \alpha, Y, \beta], Y, [X \rightarrow \alpha Y, \beta])$ for $X \rightarrow \alpha Y \beta \in P$, $Y \in V_N \cup V_T$

Reading a terminal symbols $a$ in $ch(G)$ corresponds to a shift transition of the item pushdown-automaton under $a$. $\epsilon$ transitions of $ch(G)$ correspond to the expansion transitions of the item.
pushdown-automaton. When \( \text{char}(G) \) reaches a final state \( [X \to \alpha.] \) \( PG \) 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 \( PG \).

**Example 3.4.2** Let \( G_0 \) again be the grammar for arithmetic expressions with the productions

\[
\begin{align*}
S & \to E \\
E & \to E + T \mid T \\
T & \to T \ast F \mid F \\
F & \to (E) \mid \text{Id}
\end{align*}
\]

Fig. 3.14 shows the characteristic finite-state machine to 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 \( ([S' \to S], \gamma) \xrightarrow{\text{char}(G)} ([A \to \alpha, \beta], \varepsilon) \) of the characteristic finite-state machine \( \text{char}(G) \).
2. There exists a computation \( (\rho[A \to \alpha, \beta], w) \xrightarrow{\text{PG}} ([S' \to S], \varepsilon) \) of the item pushdown-automaton \( PG \) such that \( \gamma = \text{hist}(\rho) \alpha \) holds.
3. There exists a rightmost derivation \( S' \rightharpoonup \alpha \beta w \xrightarrow{\text{rm}} \gamma' \alpha \beta w \) 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 \( \text{char}(G) \) are exactly the histories of stack contents of the item pushdown-automaton \( PG \) whose topmost symbol is this item and from which \( PG \) 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 \( \rho \) corresponds to a rightmost derivation that leads to a sentential form \( \alpha w \) where \( \alpha \) is the history of the stack contents \( \rho \).

We introduce some terminology before we prove Theorem 3.4.1. For a rightmost derivation \( S' \xrightarrow{r_m} \gamma' A v \xrightarrow{r_m} \gamma \alpha w \) and a production \( A \rightarrow \alpha \) we call \( \alpha \) the handle of the right sentential form \( \gamma \alpha w \).

Is the right side \( \alpha = \alpha'/\beta \), the prefix \( \gamma = \gamma'\alpha' \) is called a reliable prefix of \( G \) for the item \( [A \rightarrow \alpha', \beta] \). The item \([A \rightarrow \alpha, \beta]\) is valid for \( \gamma \). Theorem 3.4.1, thus, means, that the set of words under which the characteristic finite-state machine reaches an item \([A \rightarrow \alpha', \beta]\) is exactly the set of reliable prefixes for this item.

**Example 3.4.3** For the grammar \( G_0 \) we have:

<table>
<thead>
<tr>
<th>right sentential form</th>
<th>handle</th>
<th>reliable prefixes</th>
<th>reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>( E + F )</td>
<td>( F )</td>
<td>( E, E +, E + F )</td>
<td>( S \xrightarrow{r_m} E \xrightarrow{r_m} E + T \xrightarrow{r_m} E + F )</td>
</tr>
<tr>
<td>( T + ld )</td>
<td>( ld )</td>
<td>( T, T *, T * ld )</td>
<td>( S \xrightarrow{r_m} T + F \xrightarrow{r_m} T + ld )</td>
</tr>
</tbody>
</table>

\[ \square \]

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.

<table>
<thead>
<tr>
<th>reliable prefix</th>
<th>valid item</th>
<th>reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>( E + )</td>
<td>([E \rightarrow E + T])</td>
<td>( S \xrightarrow{r_m} E \xrightarrow{r_m} E + T )</td>
</tr>
<tr>
<td>( T \rightarrow F )</td>
<td></td>
<td>( S \xrightarrow{r_m} E + T \xrightarrow{r_m} E + F )</td>
</tr>
<tr>
<td>( F \rightarrow ld )</td>
<td></td>
<td>( S \xrightarrow{r_m} E + F \xrightarrow{r_m} E + ld )</td>
</tr>
<tr>
<td>( E + ( )</td>
<td>([F \rightarrow (E)])</td>
<td>( S \xrightarrow{r_m} (E + F) \xrightarrow{r_m} (E + (E)) )</td>
</tr>
<tr>
<td>( T \rightarrow F )</td>
<td></td>
<td>( S \xrightarrow{r_m} (E + (T)) \xrightarrow{r_m} (E + (F)) )</td>
</tr>
<tr>
<td>( F \rightarrow ld )</td>
<td></td>
<td>( S \xrightarrow{r_m} (E + (F)) \xrightarrow{r_m} (E + (ld)) )</td>
</tr>
</tbody>
</table>

\[ \square \]

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

Consider the rightmost derivation

\[ S' \xrightarrow{r_m} \gamma Aw \xrightarrow{r_m} \gamma \alpha \beta w \]

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

1. \( \beta \) is derived to a terminal word \( v \), and after that,
2. \( \alpha \) is derived to a terminal word \( u \).

Altogether,

\[ S' \xrightarrow{r_m} \gamma Aw \xrightarrow{r_m} \gamma \alpha \beta w \xrightarrow{r_m} \gamma \alpha uv \xrightarrow{r_m} \gamma uvw \xrightarrow{r_m} xuvw \]

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 \( \gamma \) in a number of steps, then \( u \) to \( \alpha \), then \( v \) to \( \beta \). The valid item \([A \rightarrow \alpha, \beta]\) for the reliable prefix \( \gamma \alpha \) describes the analysis situation in which the reduction of \( u \) to \( \alpha \) has already been done, while the reduction of \( v \) to \( \beta \) has not yet started. A possible long-range goal in this situation is the application of the production \( X \rightarrow \alpha \beta \).
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 each $G$ 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 $([S' \rightarrow \gamma S], \gamma)_{\text{char}(G)}^r ([A \rightarrow \alpha \beta], \varepsilon)$. By induction over the number $n$ of $\varepsilon$ transitions we construct a rightmost derivation $S' \xrightarrow{\alpha \beta} S \varepsilon \xrightarrow{\gamma} \gamma \alpha \beta w$.

Ist $n = 0$, dann ist $\gamma = \varepsilon$ und $[A \rightarrow \alpha \beta] = [S' \rightarrow S]$. Da $S' \xrightarrow{\gamma} S \varepsilon$ gilt, ist die Behauptung in diesem Fall erfüllt. Ist $n > 0$, betrachten wir den letzten $\varepsilon$-Ubergang. Dann l’asst sich die Berechnung der charakteristischen Automaten zerlegen in:

$$ ([S' \rightarrow S], \gamma)_{\text{char}(G)}^r ([X \rightarrow \alpha \beta], \varepsilon)_{\text{char}(G)} r ([A \rightarrow \alpha \beta], \varepsilon)_{\text{char}(G)} r ([A \rightarrow \alpha \beta], \varepsilon)$$

where $\gamma = \gamma' \alpha$. Nach Induktionsannahme gibt es eine rightmost derivation $S' \xrightarrow{\gamma'} \gamma'' X w' \xrightarrow{\gamma''} \gamma'' \alpha \beta w' \xrightarrow{\gamma''} \gamma' \alpha \beta w$ mit $\gamma'' = \gamma''' \alpha$. Da die Grammatik $G$ reduziert ist, gibt es ebenfalls eine rightmost derivation $\gamma' \xrightarrow{\beta} \varepsilon$. Deshalb haben wir:

$$S' \xrightarrow{\gamma \alpha \beta w} \gamma' \xrightarrow{\varepsilon} \gamma' \alpha \beta w$$

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

Nehmen wir an, wir hätten eine rightmost derivation $S' \xrightarrow{\alpha_1 \beta_1} \gamma \xrightarrow{\varepsilon} \gamma' \alpha \beta w$. Diese Ableitung l’asst sich zerlegen in:

$$S' \xrightarrow{\alpha_1 \beta_1} \gamma_1 \xrightarrow{\varepsilon} \gamma_2 \xrightarrow{\varepsilon} \gamma_3 \xrightarrow{\varepsilon} \gamma_4 \xrightarrow{\varepsilon} \gamma_5 \xrightarrow{\varepsilon} \gamma_6 \xrightarrow{\varepsilon} \gamma_7 \xrightarrow{\varepsilon} \gamma_8 \xrightarrow{\varepsilon} \gamma_9$$

mit $X_n = A$. Mit Induktion nach $n$ folgt, dass $(\rho, vw)_{\text{char}(G)}^r ([S' \rightarrow S], \varepsilon)$ gilt für

$$\rho = [S' \rightarrow \alpha_1 \beta_1] \ldots [S_{n-1} \rightarrow \alpha_n \beta_n]$$

$$w = vw_n \ldots v_1$$

sofern $\beta \xrightarrow{\alpha_1 \beta_1} \varepsilon$, $\alpha_1 = \beta_1 = \varepsilon$ and $X_1 = S$. Damit ergibt sich der Schluss $(2) \Rightarrow (3)$.

Für den letzten Schluss betrachten wir einen Kellerinhalt $\rho = \rho' [A \rightarrow \alpha \beta] \xrightarrow{\alpha_1 \beta_1} ([S' \rightarrow S], \varepsilon)$. Zuerst "überzeugen wir uns mit Induktion nach der Anzahl der "Uberg"änge in einer solchen Berechnung, dass $\rho'$ notwendigweise von der Form:

$$\rho' = [S' \rightarrow \alpha_1 \beta_1] \ldots [S_{n-1} \rightarrow \alpha_n \beta_n]$$

ist für ein $n \geq 0$ and $X_n = A$. Mit Induktion nach $n$ folgt aber, dass $([S' \rightarrow S], \gamma)_{\text{char}(G)}^r ([A \rightarrow \alpha \beta], \varepsilon)$ gilt für $\gamma = \alpha_1 \ldots \alpha_n \alpha$. Da $\gamma = \text{hist}(\rho)$, gilt auch die Behauptung $(1)$. Damit ist der Beweis vollständig. □

**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 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 \( \text{char}(G_0) \) of Fig. 3.14 on page 82. It is shown in Fig. 3.15 on page 85. It states are:

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

\[\Box\]

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

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 \( \Delta_G \) from symbols to words. We then have:

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

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:

\[
F, \ (F, \ ((F, \ \ldots \\
T \ast (F, \ T \ast ((F, \ T \ast ((F, \ \ldots \\
E + F, \ E + (F, \ E + ((F, \ \ldots
\]

The state \( S_3 \) in the canonical \( LR(0) \) automaton to \( G_0 \) contains all valid items for the reliable prefix \( E+ \), namely the items

\[
[E \rightarrow E + T], [T \rightarrow T \ast F], [T \rightarrow .F], [F \rightarrow .Id], [F \rightarrow .E].
\]

For \( E+ \) is a prefix of the right sentential form \( E + T \):

\[
\begin{align*}
S & \rightarrow_{rm} E \quad \Rightarrow \quad E + T \quad \Rightarrow \quad E + F \quad \Rightarrow \quad E + Id \\
\end{align*}
\]

\( \top \)

Valid are for instance \( [E \rightarrow E + T] \quad [T \rightarrow .F] \quad [F \rightarrow .Id] \)

\( \square \)

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 \( q_{G,0} \). The transition relation \( \Delta_0 \) 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 \( [X \rightarrow \alpha.a/\beta] \) is contained in \( q \).

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

Finish: \( \}q_{G,0}, q, \epsilon,f\) if \( [S' \rightarrow S.] \in q \). This transition is the reduction transition to the production \( S' \rightarrow S. \). The property \( [S' \rightarrow S.] \in q \) 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 \( [X \rightarrow .] \) merits special consideration. According to our description, \( |\epsilon| = 0 \) 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 \( [\cdots \rightarrow \cdots \cdot X \cdots] \) also the item \( [X \rightarrow \alpha.]\) is contained in state \( q \) for each right side \( \alpha \) of nonterminal \( X \). In the special case of a \( \epsilon \) production, the actual state \( q \) contains together with the item \( [\cdots \rightarrow \cdots \cdot X \cdots] \) also the complete item \( [X \rightarrow \alpha.]\). 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 \( q_0, \ldots, q_n \) with \( q_0 = q_{G,0} \) corresponds therefore to a uniquely determined word \( \alpha = X_1 \ldots X_n \in (V_T \cup V_N)^+ \) for which \( \Delta_G(q_n, X_{i+1}) = q_{i+1} \) holds. This word \( \alpha \) is a reliable prefix, and \( q_n \) is the set of all items valid for \( \alpha \).
Figure 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 $[X \rightarrow a \alpha \beta]$ and at least one complete item $[Y \rightarrow \gamma.]$; in the second case, $q$ contains two different complete items $[Y \rightarrow \alpha.]$, $[Z \rightarrow \beta.]$. 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 $[A \rightarrow \alpha.]$;
3. The state $q$ contains exactly one complete item $[A \rightarrow \alpha.]$, and all non-complete items in $q$ are of the form $[X \rightarrow \alpha.Y \beta]$, where all rightmost derivations for $Y$ that lead to a terminal word are of the form:

   \[
   \begin{align*}
   Y & \xrightarrow{rm} Aw \\
   & \xrightarrow{rm} w
   \end{align*}
   \]

   for a $w \in V_T^*$. \hfill \Box

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 $[S \rightarrow E.]$ and it can read the terminal symbol $+$ in the input (item $[E \rightarrow E. + T]$). In state $S_2$ the parser can reduce the right side $T$ to $E$ (complete item $[E \rightarrow T.]$) and it can read the terminal symbol $*$ (item $[T \rightarrow T. * F]$). In state $S_9$ finally, the parser can reduce the right side $E + T$ to $E$ (complete item $[E \rightarrow E + T.]$), and it can read the terminal symbol $*$ (item $[T \rightarrow T. * F]$). \hfill \Box

### 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 $\text{char}(G)$ 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 $\varepsilon$ transitions of the characteristic finite-state machine. The set $\Delta_{G,\varepsilon}(q)$ is the least solution of the following equation

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

Similar to the function $\text{closure}()$ of the subset construction it can be computed by
where \( V \) is the set of symbols \( V = \mathcal{V}_T \cup \mathcal{V}_N \). The set \( Q_G \) of states and the transition relation \( \Delta_G \) are computed by first constructing the initial state \( q_{G,0} = \Delta_G,\varepsilon([\{S' \rightarrow \cdot S\}]) \) 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 \( \text{nextState}() \) of the subset construction:

```plaintext
set(item) nextState(set(item) q, symbol X) {
    set(item) q' ← \emptyset;
    nonterminal A; string(symb) α, β;
    forall (A, α, β : ([A → α.X β] ∈ q))
        q' ← q' ∪ ([A → αX.β]);
    return closure(q');
}
```

As in the subset construction, the set of states \( \text{states} \) and the set of transitions \( \text{trans} \) can be computed iteratively:
list(set(item)) W;
set(item) q₀ ← closure([\{S' → .S\}]); states ← \{q₀\}; W ← [q₀];
trans ← ∅;
set(item) q, q';
while (W ≠ []) {
    q ← hd(W); W ← tl(W);
    forall (symbol X) {
        q' ← nextState(q, X);
        trans ← trans ∪ \{(q, X, q')\};
        if (q' ∈ states) {
            states ← states ∪ \{q'\};
            W ← q' :: W;
        }
    }
}

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

We call a context-free grammar \(G\) an LR(k)-grammar, if in each of its rightmost derivations \(S' = \alpha_0 \Rightarrow_r \cdots \Rightarrow_r \alpha_m = v\) and each right sentential forms \(\alpha_i\) occurring in the derivation

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

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

\[
S' \Rightarrow_r \alpha X w \Rightarrow_r \alpha \gamma \beta w \quad \text{and} \quad S' \Rightarrow_r \gamma Y x \Rightarrow_r \alpha \gamma x \beta w
\]

\(w|_k = y|_k\) implies \(\alpha = \gamma ∧ X = Y ∧ x = y\).

**Example 3.4.8** Let \(G\) be the grammar with the productions

\[
S \rightarrow A | B \quad A \rightarrow aAb | 0 \quad B \rightarrow aBbb | 1
\]

Then \(L(G) = \{a^n0^n | n ≥ 0\} \cup \{a^n1^n | n ≥ 0\}\). We know already that \(G\) is for no \(k ≥ 1\) an LL(k)-grammar. Grammar \(G\) is an LR(0)-grammar, though.

The right sentential forms of \(G\) have the form

\[
S, \ A, \ B, \ a^nAb^n, \ a^nBbb2^n, \ a^n0^nbb^n, \ a^n1^nbb2^n
\]

for \(n ≥ 0\). The handles are always underlined. Two different possibilities to reduce exist only in the case of right sentential forms \(a^nAb^n\) and \(a^nBbb2^n\). One could reduce \(a^nAb^n\) to \(a^nAb^n\) and to \(a^nAb^n\). The first choice belonged to the rightmost derivation

\[
S \Rightarrow_r a^nAb^n \Rightarrow_r a^nAb^n
\]

the second to no rightmost derivation. The prefix \(a^n\) of \(a^nAb^n\) 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 \(a^nBbb2^n\) are handled analogously. □
Example 3.4.9 The grammar $G_1$ with the productions

\[ S \to aAc \quad A \to Abb | b \]

and the language $L(G_1) = \{ab^{2n+1}c \mid n \geq 0\}$ is an $LR(0)$-grammar. In a right sentential form $aAbb^{2n}c$ 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 $ab^{2n}c$, $b$ is the handle, and the prefix $ab$ uniquely determines this. \( \square \)

Example 3.4.10 The grammar $G_2$ with the productions

\[ S \to aAc \quad A \to bbA | b \]

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. \( \square \)

Example 3.4.11 The grammar $G_3$ with the productions

\[ S \to aAc \quad A \to bAb | b \]

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

\[ S \xrightarrow{r} a^n Ab^n c \xrightarrow{r} a^n bb^n c \]
\[ S \xrightarrow{r} a^{n+1} Ab^{n+1} c \xrightarrow{r} a^n bb^{n+1} c \]

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. \( \square \)

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 $LR(0)$-grammat, und nehmen wir an, der canonical $LR(0)$ automaton $LR_0(G)$ habe einen $LR(0)$-inadequaten state $p$.

Fall 1: Der state $p$ hat einen reduce-reduce-conflit, d.h. $p$ enthält zwei verschiedene items $[X \to \beta_1, \beta_2]$. Dem state $p$ zugeordnet ist eine nichtleere Menge von reliable prefixes. Let $\gamma = \gamma' \beta$ ein solches reliable prefix. Weil beide items valid für $\gamma$ sind, gibt es rightmost derivations

\[ S' \xrightarrow{r_m} \gamma' X w \xrightarrow{r_m} \gamma' \beta w \quad \text{und} \]
\[ S' \xrightarrow{r_m} \nu Y y \xrightarrow{r_m} \nu \delta y \quad \text{mit} \ \nu \delta = \gamma' \beta = \gamma \]

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

Fall 2: Der state $p$ hat einen shift-reduce-conflit, d.h. $p$ enthält items $[X \to \beta, Y \to \delta$]. Let $\gamma$ ein reliable prefix für beide item Weil beide items valid für $\gamma$ sind, gibt es rightmost derivations

\[ S' \xrightarrow{r_m} \gamma' X w \xrightarrow{r_m} \gamma' \beta w \quad \text{und} \]
\[ S' \xrightarrow{r_m} \nu Y y \xrightarrow{r_m} \nu \delta a a y \quad \text{mit} \ \nu \delta = \gamma' \beta = \gamma \]

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

\[ \alpha \xrightarrow{r_m} v_1 X v_3 \xrightarrow{r_m} v_1 v_2 v_3 \]
Weil \( y \neq av_1v_2vy \) gilt, ist die \( LR(0) \)-Eigenschaft verletzt.

\[ \Rightarrow \] 
Nehmen wir an, der kanonische \( LR(0) \)-Automaton \( LR_0(G) \) habe keine \( LR(0) \)-inadequaten States. Betrachten wir die zwei rechtesten Ableitungen:

\[ S' \xrightarrow{\gamma} \alpha X w \xrightarrow{\gamma} \alpha \beta w \]
\[ S' \xrightarrow{\gamma} \gamma Y x \xrightarrow{\gamma} \alpha \beta y \]

Zu zeigen ist, dass \( \alpha = \gamma \), \( X = Y \), \( x = y \) gelten. Lassen Sie \( p \) das State of the kanonischen \( LR(0) \)-Automaton nach Lese von \( \alpha \beta \). Dann enthält \( p \) alle für \( \alpha \beta \) valid Items. Nach Voraussetzung ist \( p \) \( LR(0) \)-geeignet. Wir unterscheiden zwei Fälle:

\[ \text{Fall 1:} \; \beta \neq \epsilon \]

Wegen Lemma 3.4 ist \( p = \{ [X \rightarrow \beta] \} \), d.h. \( [X \rightarrow \beta] \) ist das einzige valid Item für \( \alpha \beta \). Daraus folgt, dass \( \alpha = \gamma \), \( X = Y \) und \( x = y \) sein muss.

\[ \text{Fall 2:} \; \beta = \epsilon \]

Nehmen wir an, die zweite rechteste Ableitung widerspricht der \( LR(0) \)-Bedingung. Dann gibt es ein weiteres Item \( [X \rightarrow \delta Y' \eta] \in p \), so dass \( \alpha = \alpha' \delta \) ist. Die letzte Anwendung einer Production in der unteren rechtesten Ableitung ist die letzte Anwendung einer Production in einer terminalen rechtesten Ableitung für \( Y' \). Nach Lemma 3.4 folgt daraus, dass die untere Ableitung gegeben ist durch:

\[ S' \xrightarrow{\alpha' \delta Y' w} \alpha' \delta X v w \xrightarrow{\alpha' \delta v w} \]

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

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 or through the characteristic finite-state machine \( \text{char}(G) \). From the deterministic finite-state machine \( LR_0(G) \) one can construct a pushdown automaton \( P_0 \). 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 \( i = [A \rightarrow \alpha \beta, x] \) for a Production \( A \rightarrow \alpha \beta \) of \( G \) and a word \( x \in (V_F^k \cup V_F^{<k})^\# \). The context-free item \( [A \rightarrow \alpha \beta] \) 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 \( [A \rightarrow \alpha \beta, x] \) is valid for a reliable prefix \( \gamma \), if there exists a rightmost derivation

\[ S' \# \xrightarrow{\gamma} \gamma' X w \# \xrightarrow{\gamma' \alpha \beta w \#} \]

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

**Example 3.4.12** Consider again grammar \( G_0 \). We have:

1. \[ [E \rightarrow E + T], \]
   \[ [E \rightarrow E + T, +] \]
   are valid \( LR(1) \)-items for the prefix \( E + \)

2. \[ [E \rightarrow T, \#] \]
   is not a valid \( LR(1) \)-item for any reliable prefix.

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

\[ S' \xrightarrow{E} (E) \xrightarrow{E + T} \]
\[ S' \xrightarrow{E} (E + T \#) \xrightarrow{E + T + \#} \]

Observation (2) follows since the subword \( E^k \) can occur in no right sentential form. \( \square \)
The following 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 $\gamma$ let $I(\gamma)$ be the set of $LR(k)$-items of $G$ that are valid for $\gamma$.

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

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

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 $[X \to \alpha,a\beta, x], [Y \to \beta, y] \in I$ 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

$$y \in \{a\} \circ_k \text{first}_k(\beta) \circ_k \{x\}$$

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 $\gamma$ the deterministic finite-state machine $LR_k(G)$ determines the set of $LR(k)$-items that are valid for $\gamma$. 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 $\gamma$ 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 its 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 canonical $LR(k)$-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 $\alpha$. 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:
3.4 Bottom-up Syntax Analysis

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

The function list(state) stack ← [q0];
terminal buffer ← scan();
state q; nonterminal X; string(symbol) α;
while (true) {
    q ← hd(stack);
    switch (action[q, buffer]) {
        case shift:
            stack ← goto[q, buffer] :: stack;
            buffer ← scan();
            break;
        case reduce(X → α):
            output(X → α);
            stack ← tl([α], stack); q ← hd(stack);
            stack ← goto[q, X] :: stack;
            break;
        case accept:
            stack ← f :: tl(2, stack);
            return accept;
        case error:
            output("...");
            goto err;
    }
}

The function list(state) tl(int n, list(state) 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 LR1(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 \(\varepsilon\)-transitions is formed. This set is the least solution of the following equation
\[
I = q \cup \{ [A \to \gamma, y] \mid \exists X \to \alpha A \beta \in P : [X \to \alpha A \beta, x] \in I, y \in \text{first}_1(\beta) \circ_1 \{ x \} \}
\]
It is computed by the following function
\[
\text{set}(\text{item}_1) \text{ closure(set}(\text{item}_1) \text{ q) } \{ \\
\hspace{1em} \text{set}(\text{item}_1) \text{ result } \leftarrow q; \\
\hspace{1em} \text{list}(\text{item}_1) \text{ W } \leftarrow \text{list_of}(q); \\
\hspace{1em} \text{nonterminal X; string(symbol) } \alpha, \beta; \text{ terminal } x, y; \\
\hspace{1em} \text{while } (W \neq \emptyset) \{ \\
\hspace{2em} \text{item}_1 \ i \leftarrow \text{hd}(W); \ W \leftarrow \text{tl}(W); \\
\hspace{2em} \text{switch } (i) \{ \\
\hspace{3em} \text{case } [\_ \to \_ . X \beta, x] : \\
\hspace{4em} \text{forall } (\alpha : (X \to \alpha) \in P) \\
\hspace{5em} \text{forall } (y \in \text{first}_1(\beta) \circ_1 \{ x \}) \\
\hspace{6em} \text{if } ([X \to \alpha, y] \notin \text{result} ) \{ \\
\hspace{7em} \text{result } \leftarrow \text{result } \cup \{ [X \to \alpha, y] \}; \\
\hspace{7em} W \leftarrow [X \to \alpha, y] :: W; \\
\hspace{6em} \}
\hspace{3em} \}
\hspace{2em} \}
\hspace{1em} \}
\hspace{1em} \text{default : break; \\
\hspace{1em} \}
\hspace{1em} \}
\hspace{1em} \text{return result; \\
\hspace{1em} }
\]
where \(V\) is the set of all symbols, \(V = V_T \cup V_N\). The initial state \(q_0\) of \(LR_1(G)\) is
\[
q_0 = \text{closure}(\{ [S' \to . S, \#] \})
\]
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:
\[
\text{set}(\text{item}_1) \text{ nextState(set}(\text{item}_1) \text{ q, symbol } X) \{ \\
\hspace{1em} \text{set}(\text{item}_1) \text{ q' } \leftarrow \emptyset; \\
\hspace{1em} \text{nonterminal A; string(symbol) } \alpha, \beta; \text{ terminal } x; \\
\hspace{1em} \text{forall } (A, \alpha, \beta, x : ([A \to \alpha X \beta, x] \in q)) \\
\hspace{2em} q' \leftarrow q' \cup \{ [A \to \alpha X \beta, x] \}; \\
\hspace{1em} \text{return \ closure(q'); \\
\hspace{1em} }
\]
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 $\alpha \neq x$. 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:

\[
\begin{align*}
\text{forall } (\text{state } q) \{ \\
\text{forall } (\text{terminal } x) \quad \text{action}[q, x] \leftarrow \text{error}; \\
\text{forall } ([X \to \alpha,\beta, x] \in q) \\
\quad \text{if } (\beta = \varepsilon) \\
\qquad \text{if } (X = S' \land \alpha = S \land x = \#) \quad \text{action}[q, \#] \leftarrow \text{accept}; \\
\qquad \text{else } \quad \text{action}[q, x] \leftarrow \text{reduce}(X \to \alpha); \\
\quad \text{else if } (\beta = a\gamma) \quad \text{action}[q, a] \leftarrow \text{shift}; \\
\}\end{align*}
\]

**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 $LR$-items are collected in one lookahead set $\#$. The next next input symbol $a$ indicates to shift, $\#$ and + to reduce; similarly in state $S_0$. 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 $S'_0, S'_1, S'_2, S'_0$ and $S'_0$. \hfill $\square$

**$SLR(1)$- 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
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 \( [X \rightarrow \alpha.\beta] \) an item in \( q \). We denote by \( \lambda(q, [X \rightarrow \alpha.\beta]) \) the lookahead set that is added to the item \( [X \rightarrow \alpha.\beta] \) in \( q \). The \( SLR(1) \)-method is different from the \( LALR(1) \)-method in the definition of the function

\[
\lambda : Q \times T \rightarrow 2^{V_r \cup \{\#\}}
\]

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

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

Relative to \( \lambda \), \( q \) has a \textit{shift-reduce}-conflict if it has items \( [X \rightarrow \alpha.\beta], [Y \rightarrow \gamma.] \in q \) with \( a \in \lambda(q, [Y \rightarrow \gamma.]) \).

If no state of the canonical \( LR(0) \) automaton has a conflict, the lookahead sets \( \lambda(q, [X \rightarrow \alpha.]) \) suffice to construct an \textit{action}-table.

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:

\[
\lambda_S(q, [X \rightarrow \alpha.\beta]) = \{ a \in V_r \cup \{\#\} | S' \# \xrightarrow{*} \gamma X aw \} = \text{follow}_1(X)
\]

for all states \( q \) with \( [X \rightarrow \alpha.] \in q \). A state \( q \) of the canonical \( LR(0) \) automaton is called \( SLR(1) \)-\textit{inadequate} if it contains conflicts with respect to the function \( \lambda_S \). \( G \) 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 \( S_6 \). We extend the complete items in the states by the \text{follow}_1\{-sets of their left sides to represent the function \( \lambda_S \) in a readable way. Since \text{follow}_1(S) = \{\#\} and \text{follow}_1(E) = \{\#, +, \} \) we obtain:

\[
S'_1 = \{ \ [S \rightarrow E., \{\#\}], \quad \ [E \rightarrow E. + T] \} \quad \text{conflict eliminated,}
\]

\[
S'_2 = \{ \ [E \rightarrow T., \{\#, +, \}], \quad \ [T \rightarrow T. * F] \} \quad \text{conflict eliminated,}
\]

\[
S'_6 = \{ \ [E \rightarrow E + T., \{\#, +, \}], \quad \ [T \rightarrow T. * F] \} \quad \text{conflict eliminated,}
\]

\[
\text{da} \ + \notin \{\#\}
\]

\[
\text{da} \ * \notin \{\#, +, \}
\]

\[
\text{da} \ * \notin \{\#, +, \}
\]

\[
\text{da} \ + \notin \{\#\}
\]

\[
\text{da} \ * \notin \{\#, +, \}
\]

\[
\text{da} \ * \notin \{\#, +, \}
\]

<table>
<thead>
<tr>
<th>( i.d )</th>
<th>( )</th>
<th>*</th>
<th>+</th>
<th>#</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S_0 )</td>
<td>( s )</td>
<td>( s )</td>
<td></td>
<td>1: ( S \rightarrow E )</td>
</tr>
<tr>
<td>( S_1 )</td>
<td>( s )</td>
<td>( acc )</td>
<td></td>
<td>2: ( E \rightarrow E + T )</td>
</tr>
<tr>
<td>( S_2 )</td>
<td>( s )</td>
<td>( r(3) )</td>
<td>( r(3) )</td>
<td>3: ( E \rightarrow T )</td>
</tr>
<tr>
<td>( S_6 )</td>
<td>( s )</td>
<td>( s )</td>
<td></td>
<td>4: ( T \rightarrow T * F )</td>
</tr>
<tr>
<td>( S_9 )</td>
<td>( s )</td>
<td>( r(2) )</td>
<td>( r(2) )</td>
<td>5: ( T \rightarrow F )</td>
</tr>
<tr>
<td>( S_9 )</td>
<td>( s )</td>
<td>( r(2) )</td>
<td>( r(2) )</td>
<td>6: ( F \rightarrow (E) )</td>
</tr>
<tr>
<td>( S_9 )</td>
<td>( s )</td>
<td>( r(2) )</td>
<td>( r(2) )</td>
<td>7: ( F \rightarrow \text{Id} )</td>
</tr>
</tbody>
</table>

\[\text{Table 3.6.} \text{ Some rows of the action-table of the canonical LR(1) parser for } G_0. \text{ s stands for shift, } r(i) \text{ for reduce by production } i, \text{ acc for accept. All empty entries represent error.}\]
So, \( G_0 \) is an SLR(1)-grammar and it has an SLR(1) parser. □

The set 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 \([X → α]\) occurs. The most precise lookahead set that considers the state is defined by:

\[
\lambda_L(q, [X → α.β]) = \{ a ∈ V_T ∪ \{ # \} | S'\# \xrightarrow{r_{tm}} γ.Xaw ∧ Δ_G(q_0, γα) = q \}
\]

Here, \( q_0 \) is the initial state, and \( Δ_G \) is the transition function of the canonical \( LR(0) \) automaton \( LR_0(G) \).

In \( λ_L(q, [X → α.] \} only terminal symbols are contained that can follow X in a right sentential form \( βXaw \) such that \( βα \) 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 \( λ_L(q, [A → α.β]) \) can be characterized as the least solution of the following system of equations:

\[
\begin{align*}
λ_L(q_0, [S' → .S]) &= \{ # \}, \\
λ_L(q, [A → αX.β]) &= \bigcup \{ λ_L(p, [A → α.Xβ]) | Δ_G(p, X) = q \}, \quad X ∈ (V_T ∪ V_N) \\
λ_L(q, [A → .α]) &= \bigcup \{ \text{first}_1(β) ⊕_1 λ_L(q, [X → γ.Aβ]) | [X → γ.Aβ] ∈ q' \}
\end{align*}
\]

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 \([A → α.X.β]\) in a state \( q \) result from the follow symbols after the dot in an item \([A → α.X.β]\) 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 \([A → .α]\) in a state \( q \) result from the follow symbols of \( A \) in items in \( q \) after the dot, that is, from sets \( \text{first}_1(β) ⊕_1 λ_L(q, [X → γ.Aβ]) \) for items \([X → γ.Aβ]\) in \( q \).

The system of equations for the sets \( λ_L(q, [A → α.β]) \) over the finite subset lattice \( 2^{V_T ∪ \{ # \}} \) 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.
A further possibility consists in the modification of Algorithm LR(1)-GEN. The conditional statement

\[
\text{if } (q' \text{ not in states}) \text{ states } \leftarrow \text{ states } \cup \{q'\};
\]

is replaced by

\[
\text{if } (\text{exists } q'' \text{ in states with samecores}(q', q'')) \text{ merge}(\text{states}, q', q'');
\]

where

\[
\text{bool samecores(set of item } p, \text{ set of item } p')
\]

\[
\text{if } (\text{set of cores of } p = \text{set of cores of } p')
\]

\[
\text{return true;}
\]

\[
\text{else}
\]

\[
\text{return false;}
\]

\[
\text{void merge(set of set of item } \text{ states, set of item } p, \text{ set of item } p')
\]

\[
\text{states } \leftarrow \text{ states } \setminus \{p'\} \cup \{|X \rightarrow \alpha.\beta, L_1 \cup L_2| [X \rightarrow \alpha.\beta, L_1] \in p \text{ and } [X \rightarrow \alpha.\beta, L_2] \in p'|};
\]

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

\[
S' \rightarrow S
\]

\[
S \rightarrow L = R | R
\]

\[
L \rightarrow *R | Id
\]

\[
R \rightarrow L
\]

This grammar is not an SLR(1)-grammar, but it is a LALR(1)-grammar. The states of the canonical LR(0) automaton are given by:
3.4 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 $action[q,a] = \text{error}$, 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 $shift(\beta a)$ for an item $[A \rightarrow \alpha.\beta a\gamma]$ in $q$, a $reduce$ for incomplete items in $q$, and a $skip$.

- The correction $shift(\beta a)$ assumes that the subword for $\beta$ is missing. It therefore pushes the states that the item-pushdown automaton would run through when reading the word $\beta$ starting in state $q$. After that the symbol $a$ is read and the associated $shift$-transition of the parser is performed.
- The correction $reduce(A \rightarrow \alpha.\beta)$ also assumes that the subword that belongs to $\beta$ is missing. Therefore $|\alpha|$ many states are removed from the stack. Let $p$ be the newly appeared state on top of that 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 $[A \rightarrow \alpha.\beta a\gamma]$, the parser could try to restart by reading $a$. As correction $shift(\beta a)$ is performed. If the symbol does not occur in any right side of an items in $q$, but as lookahead of a noncomplete item $[A \rightarrow \alpha.\beta]$ in $q$, then the correction $reduce(A \rightarrow \alpha.\beta)$ could be performed. If several such corrections are possible in $q$ a plausible correction is chosen. It could be plausible to choose the operation $shift(\beta a)$ or $reduce(A \rightarrow \alpha.\beta)$ in which the missing subword $\beta$ 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

$$
E \rightarrow E + T \\
E \rightarrow E * F \\
E \rightarrow T \\
T \rightarrow T * F \\
T \rightarrow F \\
F \rightarrow (E) \\
F \rightarrow \mathtt{Id}
$$

State $S_2$ is the only $LR(0)$-inadequate state. We have follow$_1(R) = \{\#, =\}$. This lookahead set for the item $[R \rightarrow L]$ 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 $\lambda_L(q,[A \rightarrow \alpha.\beta])$ were directly associated with the item $[A \rightarrow \alpha.\beta]$ of state $q$. In state $S_2$, the item $[R \rightarrow L]$ has now the lookahead set $\{\#\}$. The conflict is resolved since this set does not contain the next input symbol $=$. □
for which the canonical $LR(0)$ automaton was constructed in Example 3.4.5. As input we choose

( Id  + )

After reading the prefixes (Id  + ) the stack of an $SLR(1)$ parser contains the sequence of states $S_0S_1S_2S_3$, corresponding to the reliable prefix ( $E$  + . The actual state $S_6$ consists of the items :

$$S_6 = \{ [E \rightarrow E + .T],$$
$$[T \rightarrow .F],$$
$$[F \rightarrow .Id],$$
$$[F \rightarrow .(E)] \}$$

We consider an $SLR(1)$ parser. Its lookahead sets of the items in $S_6$ are the follow$_1$-sets of the left sides, i.e.

<table>
<thead>
<tr>
<th>$S_6$</th>
<th>$\lambda_S$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$[E \rightarrow E + .T]$</td>
<td>+, )</td>
</tr>
<tr>
<td>$[T \rightarrow .F]$</td>
<td>*, +, )</td>
</tr>
<tr>
<td>$[F \rightarrow .Id]$</td>
<td>*, +, )</td>
</tr>
<tr>
<td>$[F \rightarrow .(E)]$</td>
<td>*, +, )</td>
</tr>
</tbody>
</table>

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 $[E \rightarrow E + .T]$. The reduction produces a new stack content $S_0S_1S_2S_3$ since $S_0$ is the successor state of $S_1$ 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 $f$.  \[ \square \]
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 Parsetabellen eine Fehlerbehandlung erzeugt, dabei aber annimmt, dass das Programm im wesentlichen syntaktisch korrekt ist und deshalb nur minimal abgeändert werden muss. Das Verfahren geht ebenfalls vorwärts über das Input. Im Fehlerfall versucht es, den Input nach Möglichkeit nur an einer einzigen Stelle abzuändern. Das nennen wir die *Ein-Fehler-Hypothese*. Vorberechnete Informationen wird eingesetzt, um effizient zu entscheiden, wie der Fehler in den Input korrigiert werden sollte.

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

Eine Konfiguration \((\varphi q, a_i \ldots a_n)\) einer error configuration. Das Ziel der Fehlerkorrektur mit einer der drei operations lässt sich wie folgt beschreiben:

*Loschen:* Finde Kellerinhalte \(\varphi'p\) mit

\[ (\varphi q, a_{i+1} \ldots a_n) \vdash (\varphi' p, a_{i+1} \ldots a_n) \text{ und } action[p, a_{i+1}] = shift \]

*Ersetzen:* Finde ein Symbol \(a\) und Kellerinhalte \(\varphi'p\) mit

\[ (\varphi q, aa_{i+1} \ldots a_n) \vdash (\varphi' p, a_{i+1} \ldots a_n) \text{ und } action[p, a_{i+1}] = shift \]

*Einf"ugen:* Finde ein Symbol \(a\) und Kellerinhalte \(\varphi'p\) mit

\[ (\varphi q, a_{i} \ldots a_n) \vdash (\varphi' p, a_{i} \ldots a_n) \text{ und } action[p, a_{i}] = shift \]

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

Fehlerbehandlungsmethoden mit Zur"ucksetzen erlauben zus"atzlich, eine zuletzt angewandte production des Form \(X \rightarrow aY\ r"uckg"angig zu machen und \(Y_{a_1} \ldots a_n\) als input zu betrachten, wenn der anderen Korrekturversuche gescheitert sind.

Ein naives Verfahren wird die 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 Missserfolg ist dann der error configuration wiederherzustellen und die nächste Möglichkeit auszuprobieren. Die Suche nachrichtigen Abänderung eines Symbols kann damit sehr teuer sein. Deshalb interessieren wir uns für Vorberechnungen, die man bereits zur Generierungszeit of the Parsers durchführen kann, um Sackgassen bei der Fehlerkorrektur schneller zu erkennen. Let \((\varphi q, a_{i} \ldots a_n)\) wieder die error configuration. Betrachten wir die *Einf"ugen* eines Symbols \(a \in V_T\). Die Fehlerbehandlung kann aus der folgenden Sequenz von Schritten bestehen (siehe Abbildung 3.19 (a)):

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