Chapter 8

Introduction to LR-Parsing

8.1 $LR(0)$-Characteristic Automata

The purpose of *LR-parsing*, invented by D. Knuth in the mid sixties, is the following: Given a context-free grammar G, for any terminal string $w \in \Sigma^*$, find out whether w belongs to the language $L(G)$ generated by G, and if so, construct a rightmost derivation of w, in a deterministic fashion.

Of course, this is not possible for all context-free grammars, but only for those that correspond to languages that can be recognized by a *deterministic* PDA (DPDA).

Knuth's major discovery was that for a certain type of grammars, the $LR(k)$-grammars, a certain kind of DPDA could be constructed from the grammar (*shift/reduce parsers*).

The k in $LR(k)$ refers to the amount of *lookahead* that is necessary in order to proceed deterministically.

It turns out that $k = 1$ is sufficient, but even in this case, Knuth construction produces very large DPDA's, and his original $LR(1)$ method is not practical.

Fortunately, around 1969, Frank DeRemer, in his MIT Ph.D. thesis, investigated a practical restriction of Knuth's method, known as $SLR(k)$, and soon after, the $LALR(k)$ method was discovered.

The $SLR(k)$ and the $LALR(k)$ methods are both based on the construction of the *$LR(0)$-characteristic automaton* from a grammar G, and we begin by explaining this construction.

The additional ingredient needed to obtain an $SLR(k)$ or an $LALR(k)$ parser from an $LR(0)$ parser is the computation of lookahead sets.

In the SLR case, the FOLLOW sets are needed, and in the $LALR$ case, a more sophisticated version of the FOLLOW sets is needed.

For simplicity of exposition, we first assume that grammars have no ϵ-rules.

Given a reduced context-free grammar $G = (V, \Sigma, P, S')$ augmented with start production $S' \to S$, where S' does not appear in any other productions, the set C_G of *characteristic strings of G* is the following subset of V^* (watch out, not Σ^*):

$$C_G = \{\alpha\beta \in V^* \mid S' \overset{*}{\underset{rm}{\Longrightarrow}} \alpha B v \underset{rm}{\Longrightarrow} \alpha\beta v,\\ \alpha, \beta \in V^*, v \in \Sigma^*, B \to \beta \in P\}.$$

In words, C_G is a certain set of prefixes of sentential forms obtained in rightmost derivations.

The fundamental property of LR-parsing, due to D. Knuth, is that C_G is a *regular language*. Furthermore, a DFA, DCG, accepting C_G, can be constructed from G.

Conceptually, it is simpler to construct the DFA accepting C_G in two steps:

(1) First, construct a nondeterministic automaton with ϵ-rules, NCG, accepting C_G.

(2) Apply the subset construction (Rabin and Scott's method) to NCG to obtain the DFA DCG.

In fact, careful inspection of the two steps of this construction reveals that it is possible to construct DCG directly in a single step, and this is the construction usually found in most textbooks on parsing.

The nondeterministic automaton NCG accepting C_G is defined as follows:

The states of N_{C_G} are "marked productions", where a marked production is a string of the form $A \to \alpha``."\beta$, where $A \to \alpha\beta$ is a production, and "." is a symbol not in V called the "dot" and which can appear anywhere within $\alpha\beta$.

The start state is $S' \to ``."S$, and the transitions are defined as follows:

(a) For every terminal $a \in \Sigma$, if $A \to \alpha``."a\beta$ is a marked production, with $\alpha, \beta \in V^*$, then there is a transition on input $a$ from state $A \to \alpha``."a\beta$ to state $A \to \alpha a``."\beta$ obtained by "shifting the dot." Such a transition is shown in Figure 8.1.

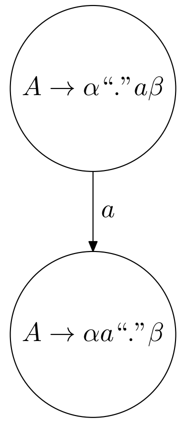

Figure 8.1: Transition on terminal input a

(b) For every nonterminal $B \in N$, if $A \to \alpha``."B\beta$ is a marked production, with $\alpha, \beta \in V^*$, then there is a transition on input $B$ from state $A \to \alpha``."B\beta$ to state $A \to \alpha B``."\beta$ (obtained by "shifting the dot"), and transitions on input $\epsilon$ (the empty string) to all states $B \to ``."\gamma_i$, for all productions $B \to \gamma_i$ with left-hand side B. Such transitions are shown in Figure 8.2.

(c) A state is *final* if and only if it is of the form $A \to \beta``."$ (that is, the dot is in the rightmost position).

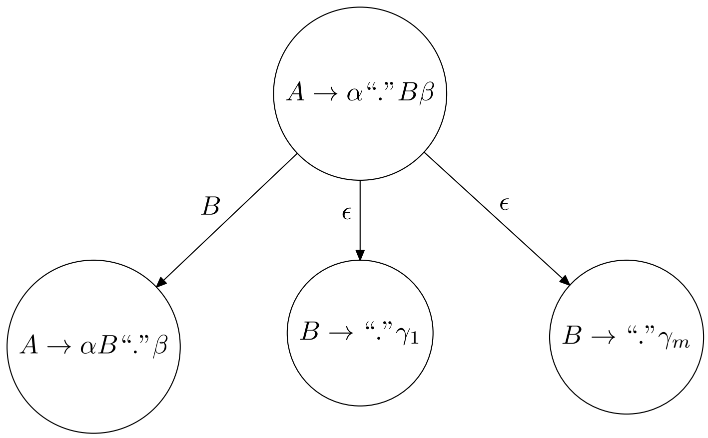

Figure 8.2: Transitions from a state $A \to \alpha$“.”$B\beta$

The above construction is illustrated by the following example:

Example 1. Consider the grammar G_1 given by:

$$
\begin{aligned}
S &\longrightarrow E \\
E &\longrightarrow aEb \\
E &\longrightarrow ab
\end{aligned}
$$

The NFA for C_{G_1} is shown in Figure 8.3.

The result of making the NFA for C_{G_1} deterministic is shown in Figure 8.4 (where transitions to the “dead state” have been omitted). The internal structure of the states $1, \ldots, 6$ is shown below:

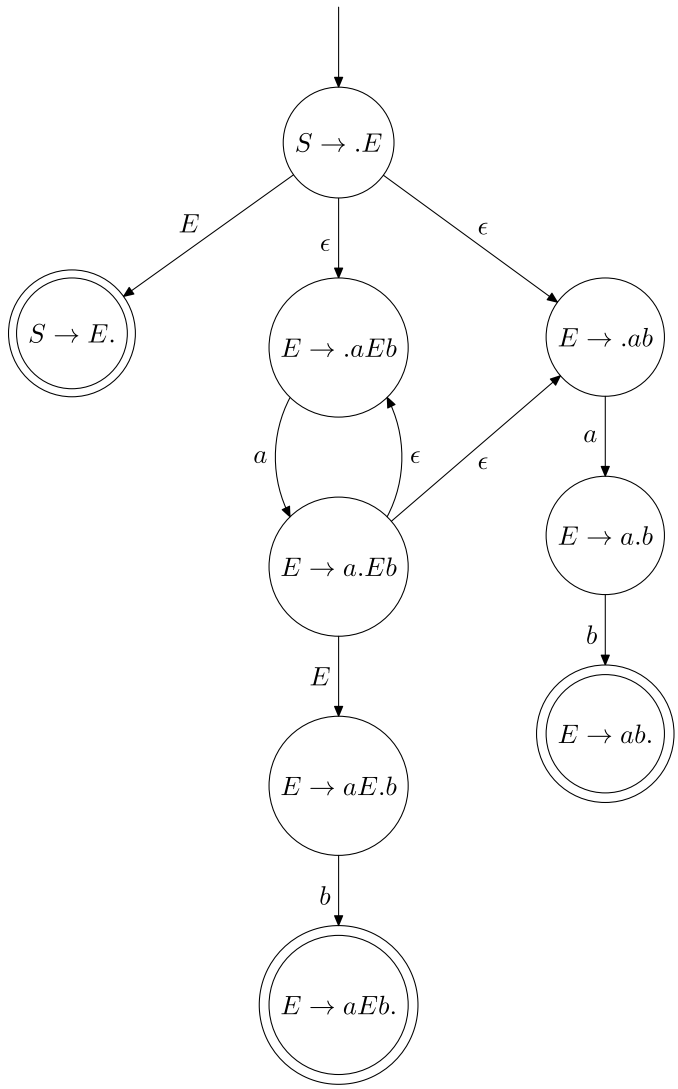

Figure 8.3: NFA for C_{G_1}

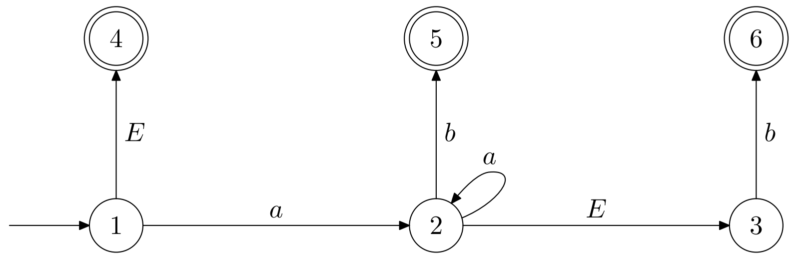

Figure 8.4: DFA for C_{G_1}

$$
\begin{aligned}
1: S &\longrightarrow .E \\
E &\longrightarrow .aEb \\
E &\longrightarrow .ab \\
2: E &\longrightarrow a.Eb \\
E &\longrightarrow a.b \\
E &\longrightarrow .aEb \\
E &\longrightarrow .ab \\
3: E &\longrightarrow aE.b \\
4: S &\longrightarrow E. \\
5: E &\longrightarrow ab. \\
6: E &\longrightarrow aEb.
\end{aligned}
$$

The next example is slightly more complicated.

Example 2. Consider the grammar G_2 given by:

$$
\begin{aligned}
S &\longrightarrow E \\
E &\longrightarrow E + T \\
E &\longrightarrow T \\
T &\longrightarrow T * a \\
T &\longrightarrow a
\end{aligned}
$$

The result of making the NFA for C_{G_2} deterministic is shown in Figure 8.5 (where transitions to the "dead state" have been omitted).

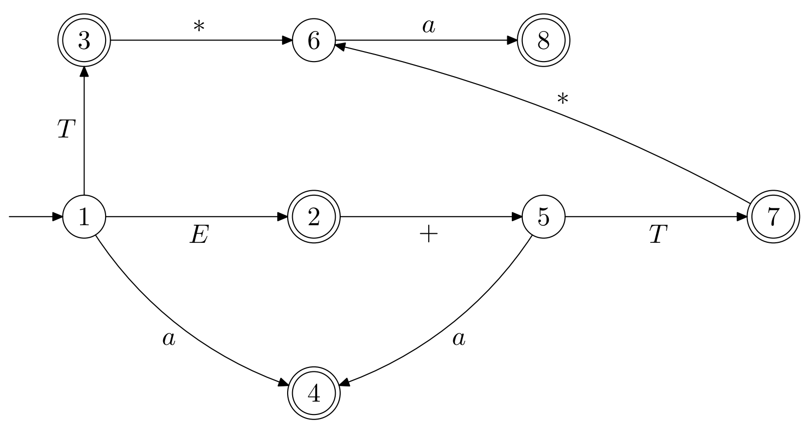

Figure 8.5: DFA for C_{G_2}

The internal structure of the states $1, \ldots, 8$ is shown below:

$$
\begin{aligned}
1: S &\longrightarrow .E \\
E &\longrightarrow .E + T \\
E &\longrightarrow .T \\
T &\longrightarrow .T * a \\
T &\longrightarrow .a \\
2: E &\longrightarrow E. + T \\
S &\longrightarrow E. \\
3: E &\longrightarrow T. \\
T &\longrightarrow T. * a \\
4: T &\longrightarrow a. \\
5: E &\longrightarrow E + .T \\
T &\longrightarrow .T * a \\
T &\longrightarrow .a \\
6: T &\longrightarrow T * .a \\
7: E &\longrightarrow E + T. \\
T &\longrightarrow T. * a \\
8: T &\longrightarrow T * a.
\end{aligned}
$$

Note that some of the marked productions are more important than others.

For example, in state 5, the marked production $E \longrightarrow E + .T$ determines the state.

The other two items $T \longrightarrow .T * a$ and $T \longrightarrow .a$ are obtained by ϵ-closure.

We call a marked production of the form $A \longrightarrow \alpha.\beta$, where $\alpha \neq \epsilon$, a *core item*.

A marked production of the form $A \longrightarrow \beta.$ is called a *reduce item*. Reduce items only appear in final states.

If we also call $S' \longrightarrow .S$ a core item, we observe that every state is completely determined by its subset of core items.

The other items in the state are obtained via ϵ-closure.

We can take advantage of this fact to write a more efficient algorithm to construct in a single pass the $LR(0)$-automaton.

Also observe the so-called *spelling property*: All the transitions entering any given state have the same label.

Given a state s, if s contains both a reduce item $A \longrightarrow \gamma.$ and a shift item $B \longrightarrow \alpha.a\beta$, where $a \in \Sigma$, we say that there is a *shift/reduce conflict* in state s on input a.

If s contains two (distinct) reduce items $A_1 \longrightarrow \gamma_1.$ and $A_2 \longrightarrow \gamma_2.$, we say that there is a *reduce/reduce conflict* in state s.

A grammar is said to be $LR(0)$ if the DFA DCG has no conflicts. This is the case for the grammar G_1.

However, it should be emphasized that this is extremely rare in practice. The grammar G_1 is just very nice, and a toy example.

In fact, G_2 is not $LR(0)$.

To eliminate conflicts, one can either compute $SLR(1)$-lookahead sets, using FOLLOW sets, or sharper lookahead sets, the $LALR(1)$ sets.

For example, the computation of $SLR(1)$-lookahead sets for G_2 will eliminate the conflicts.

In order to motivate the construction of a shift/reduce parser from the DFA accepting C_G, let us consider a rightmost derivation for $w = aaabbb$ in reverse order for the grammar

$$
\begin{aligned}
0\colon\ & S \longrightarrow E \\
1\colon\ & E \longrightarrow aEb \\
2\colon\ & E \longrightarrow ab
\end{aligned}
$$

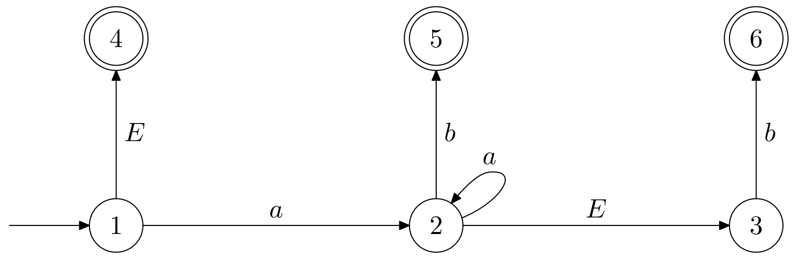

Figure 8.6: DFA for C_G

$aaabbb$	$\alpha_1\beta_1v_1$		
$aaEbb$	$\alpha_1B_1v_1$		$E \longrightarrow ab$
$aaEbb$	$\alpha_2\beta_2v_2$		
aEb	$\alpha_2B_2v_2$		$E \longrightarrow aEb$
aEb	$\alpha_3\beta_3v_3$	$\alpha_3 = v_3 = \epsilon$	
E	$\alpha_3B_3v_3$	$\alpha_3 = v_3 = \epsilon$	$E \longrightarrow aEb$
E	$\alpha_4\beta_4v_4$	$\alpha_4 = v_4 = \epsilon$	
S	$\alpha_4B_4v_4$	$\alpha_4 = v_4 = \epsilon$	$S \longrightarrow E$

Observe that the strings $\alpha_i\beta_i$ for $i = 1, 2, 3, 4$ are all accepted by the DFA for C_G shown in Figure 8.6.

Also, every step from $\alpha_i\beta_i v_i$ to $\alpha_i B_i v_i$ is the inverse of the derivation step using the production $B_i \longrightarrow \beta_i$, and the marked production $B_i \longrightarrow \beta_i$" ." is one of the reduce items in the final state reached after processing $\alpha_i\beta_i$ with the DFA for C_G.

This suggests that we can parse $w = aaabbb$ by recursively running the DFA for C_G.

The first time (which correspond to step 1) we run the DFA for C_G on w, some string $\alpha_1\beta_1$ is accepted and the remaining input in v_1.

Then, we "reduce" β_1 to B_1 using a production $B_1 \longrightarrow \beta_1$ corresponding to some reduce item $B_1 \longrightarrow \beta_1$" ." in the final state s_1 reached on input $\alpha_1\beta_1$.

We now run the DFA for C_G on input $\alpha_1 B_1 v_1$. The string $\alpha_2\beta_2$ is accepted, and we have

$$\alpha_1 B_1 v_1 = \alpha_2\beta_2 v_2.$$

We reduce β_2 to B_2 using a production $B_2 \longrightarrow \beta_2$ corresponding to some reduce item $B_2 \longrightarrow \beta_2$" ." in the final state s_2 reached on input $\alpha_2\beta_2$.

We now run the DFA for C_G on input $\alpha_2 B_2 v_2$, and so on.

At the $(i+1)$th step $(i \geq 1)$, we run the DFA for C_G on input $\alpha_i B_i v_i$. The string $\alpha_{i+1}\beta_{i+1}$ is accepted, and we have

$$\alpha_i B_i v_i = \alpha_{i+1}\beta_{i+1}v_{i+1}.$$

We reduce β_{i+1} to B_{i+1} using a production $B_{i+1} \longrightarrow \beta_{i+1}$ corresponding to some reduce item $B_{i+1} \longrightarrow \beta_{i+1}$" ." in the final state s_{i+1} reached on input $\alpha_{i+1}\beta_{i+1}$.

The string β_{i+1} in $\alpha_{i+1}\beta_{i+1}v_{i+1}$ if often called a *handle*.

Then we run again the DFA for C_G on input $\alpha_{i+1}B_{i+1}v_{i+1}$.

Now, because the DFA for C_G is *deterministic* there is no need to rerun it on the entire string $\alpha_{i+1}B_{i+1}v_{i+1}$, because *on input* α_{i+1} it will take us to *the same state*, say p_{i+1}, that it reached on input $\alpha_{i+1}\beta_{i+1}v_{i+1}$!

The trick is that we can use a *stack* to keep track of the sequence of states used to process $\alpha_{i+1}\beta_{i+1}$.

Then, to perform the reduction of $\alpha_{i+1}\beta_{i+1}$ to $\alpha_{i+1}B_{i+1}$, we simply *pop* a number of states equal to $|\beta_{i+1}|$, encovering a new state p_{i+1} on top of the stack, and from state p_{i+1} we perform the transition on input B_{i+1} to a state q_{i+1} (in the DFA for C_G), so we *push* state q_{i+1} on the stack which now contains the sequence of states on input $\alpha_{i+1}B_{i+1}$ that takes us to q_{i+1}.

Then we resume scanning v_{i+1} using the DGA for C_G, *pushing* each state being traversed on the stack until we hit a final state.

At this point we find the new string $\alpha_{i+2}\beta_{i+2}$ that leads to a final state and we continue as before.

The process stops when the remaining input v_{i+1} becomes empty and when the reduce item $S' \longrightarrow S.$ (here
$S \longrightarrow E.$) belongs to the final state s_{i+1}.

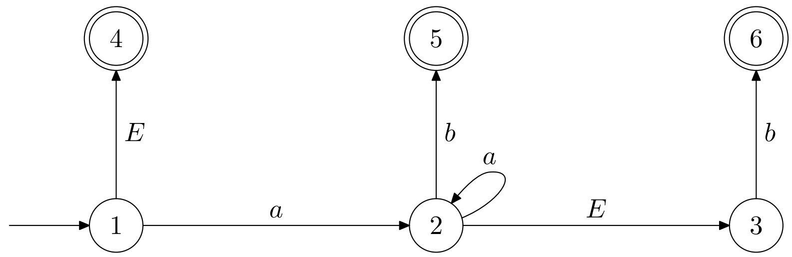

Figure 8.7: DFA for C_G

For example, on input $\alpha_2\beta_2 = aaEbb$, we have the sequence of states:

$$1\ 2\ 2\ 3\ 6$$

State 6 contains the marked production $E \longrightarrow aEb$“.”, so we pop the three topmost states 2 3 6 obtaining the stack

$$1\ 2$$

and then we make the transition from state 2 on input E, which takes us to state 3, so we push 3 on top of the stack, obtaining

$$1\ 2\ 3$$

We continue from state 3 on input b.

Basically, the recursive calls to the DFA for C_G are implemented using a stack.

What is not clear is, during step $i+1$, when reaching a final state s_{i+1}, how do we know which production $B_{i+1} \longrightarrow \beta_{i+1}$ to use in the reduction step?

Indeed, state s_{i+1} could contain several reduce items $B_{i+1} \longrightarrow \beta_{i+1}$"."

This is where we assume that we were able to compute some *lookahead information*, that is, for every final state s and every input a, we know which unique production $n\colon B_{i+1} \longrightarrow \beta_{i+1}$ applies. This is recorded in a table name "action," such that $\mathrm{action}(s,a) = rn$, where "r" stands for reduce.

Typically we compute SLR(1) or LALR(1) lookahead sets.

Otherwise, we could pick some reducing production nondeterministicallly and use backtracking. This works but the running time may be exponential.

The DFA for C_G and the action table giving us the reductions can be combined to form a bigger action table which specifies completely how the parser using a stack works.

This kind of parser called a *shift-reduce parser* is discussed in the next section.

In order to make it easier to compute the reduce entries in the parsing table, we assume that the end of the input w is signalled by a special endmarker traditionally denoted by \$.

8.2 Shift/Reduce Parsers

A shift/reduce parser is a modified kind of DPDA.

Firstly, push moves, called *shift moves*, are restricted so that exactly one symbol is pushed on top of the stack.

Secondly, more powerful kinds of pop moves, called *reduce moves*, are allowed. During a reduce move, a finite number of stack symbols may be popped off the stack, and the last step of a reduce move, called a *goto move*, consists of pushing one symbol on top of new topmost symbol in the stack.

Shift/reduce parsers use *parsing tables* constructed from the $LR(0)$-characteristic automaton DCG associated with the grammar.

The shift and goto moves come directly from the transition table of DCG, but the determination of the reduce moves requires the computation of *lookahead sets*.

The $SLR(1)$ lookahead sets are obtained from some sets called the FOLLOW sets, and the $LALR(1)$ lookahead sets $\mathrm{LA}(s, A \longrightarrow \gamma)$ require fancier FOLLOW sets.

The construction of shift/reduce parsers is made simpler by assuming that the end of input strings $w \in \Sigma^*$ is indicated by the presence of an *endmarker*, usually denoted \$, and assumed not to belong to Σ.

Consider the grammar G_1 of Example 1, where we have numbered the productions $0, 1, 2$:

$$\begin{aligned} 0: S &\longrightarrow E \\ 1: E &\longrightarrow aEb \\ 2: E &\longrightarrow ab \end{aligned}$$

The parsing tables associated with the grammar G_1 are shown below:

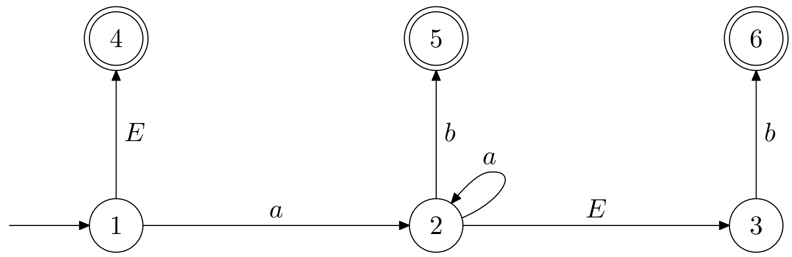

Figure 8.8: DFA for C_G

	a	b	$\$$	E
1	$s2$			4
2	$s2$	$s5$		3
3		$s6$		
4			acc	
5	$r2$	$r2$	$r2$	
6	$r1$	$r1$	$r1$	

Entries of the form si are *shift actions*, where i denotes one of the states, and entries of the form rn are *reduce actions*, where n denotes a production number (*not* a state).

The special action acc means accept, and signals the successful completion of the parse.

Entries of the form i, in the rightmost column, are *goto actions*.

All blank entries are **error** entries, and mean that the parse should be aborted.

We will use the notation action(s, a) for the entry corresponding to state s and terminal $a \in \Sigma \cup \{\$\}$, and goto$(s, A)$ for the entry corresponding to state s and nonterminal $A \in N - \{S'\}$.

Assuming that the input is $w\$$, we now describe in more detail how a shift/reduce parser proceeds.

The parser uses a stack in which states are pushed and popped. Initially, the stack contains state 1 and the cursor pointing to the input is positioned on the leftmost symbol.

There are four possibilities:

(1) If action$(s, a) = sj$, then push state j on top of the stack, and advance to the next input symbol in $w\$$. This is a *shift move*.

(2) If action$(s, a) = rn$, then do the following: First, determine the length $k = |\gamma|$ of the righthand side of the production $n\colon A \longrightarrow \gamma$. Then, pop the topmost k symbols off the stack (if $k = 0$, no symbols are popped). If p is the new top state on the stack (after the k pop moves), push the state goto(p, A) on top of the stack, where A is the lefthand side of the "reducing production" $A \longrightarrow \gamma$. Do not advance the cursor in the current input. This is a *reduce move*.

(3) If action$(s, \$) =$ acc, then accept. The input string w belongs to $L(G)$.

(4) In all other cases, **error**, abort the parse. The input string w does not belong to $L(G)$.

Observe that no explicit state control is needed. The current state is always the current topmost state in the stack.

We illustrate below a parse of the input $aaabbb\$$.

stack	remaining input	action
1	$aaabbb\$$	$s2$
12	$aabbb\$$	$s2$
122	$abbb\$$	$s2$
1222	$bbb\$$	$s5$
12225	$bb\$$	$r2$
1223	$bb\$$	$s6$
12236	$b\$$	$r1$
123	$b\$$	$s6$
1236	$\$$	$r1$
14	$\$$	acc

Observe that the sequence of reductions read from bottom-up yields a rightmost derivation of $aaabbb$ from E (or from S, if we view the action acc as the reduction by the production $S \longrightarrow E$).

This is a general property of LR-parsers.