

3.17 $LR(0)$ -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 on *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 $S' \rightarrow S$, 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^*):

$$C_G = \{ \alpha\beta \in V^* \mid S' \xrightarrow[rm]{*} \alpha Bv \xrightarrow[rm]{} \alpha\beta v, \\ \alpha, \beta \in V^*, v \in \Sigma^*, B \rightarrow \beta \in P \}.$$

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 .

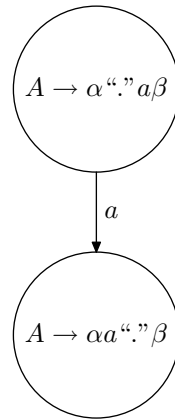
In fact, careful inspection of the two steps of this construction reveals that it is possible to construct DCG directly in a single step, and this is the construction usually found in most textbooks on parsing.

The nondeterministic automaton NCG accepting C_G is defined as follows:

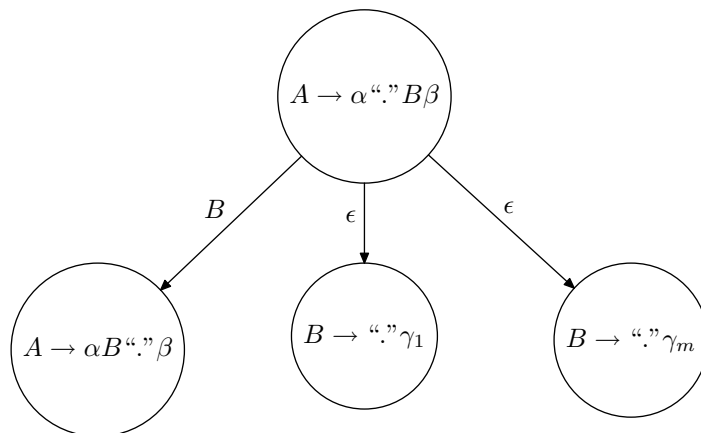
The states of N_{C_G} are “marked productions”, where a marked production is a string of the form $A \rightarrow \alpha \text{“.”} \beta$, where $A \rightarrow \alpha\beta$ is a production, and “.” is a symbol not in V called the “dot” and which can appear anywhere within $\alpha\beta$.

The start state is $S' \rightarrow \text{“.”} S$, and the transitions are defined as follows:

- (a) For every terminal $a \in \Sigma$, if $A \rightarrow \alpha \text{“.”} a\beta$ is a marked production, with $\alpha, \beta \in V^*$, then there is a transition on input a from state $A \rightarrow \alpha \text{“.”} a\beta$ to state $A \rightarrow \alpha a \text{“.”} \beta$ obtained by “shifting the dot.” Such a transition is shown in Figure 3.1.

Figure 3.1: Transition on terminal input a

- (b) For every nonterminal $B \in N$, if $A \rightarrow \alpha \cdot B \beta$ is a marked production, with $\alpha, \beta \in V^*$, then there is a transition on input B from state $A \rightarrow \alpha \cdot B \beta$ to state $A \rightarrow \alpha B \cdot \beta$ (obtained by “shifting the dot”), and transitions on input ϵ (the empty string) to all states $B \rightarrow \cdot \gamma_i$, for all productions $B \rightarrow \gamma_i$ with left-hand side B . Such transitions are shown in Figure 3.2.
- (c) A state is *final* if and only if it is of the form $A \rightarrow \beta \cdot$ (that is, the dot is in the rightmost position).

Figure 3.2: Transitions from a state $A \rightarrow \alpha \cdot B \beta$

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 3.3.

The result of making the NFA for C_{G_1} deterministic is shown in Figure 3.4 (where transitions to the “dead state” have been omitted). The internal structure of the states $1, \dots, 6$ is shown below:

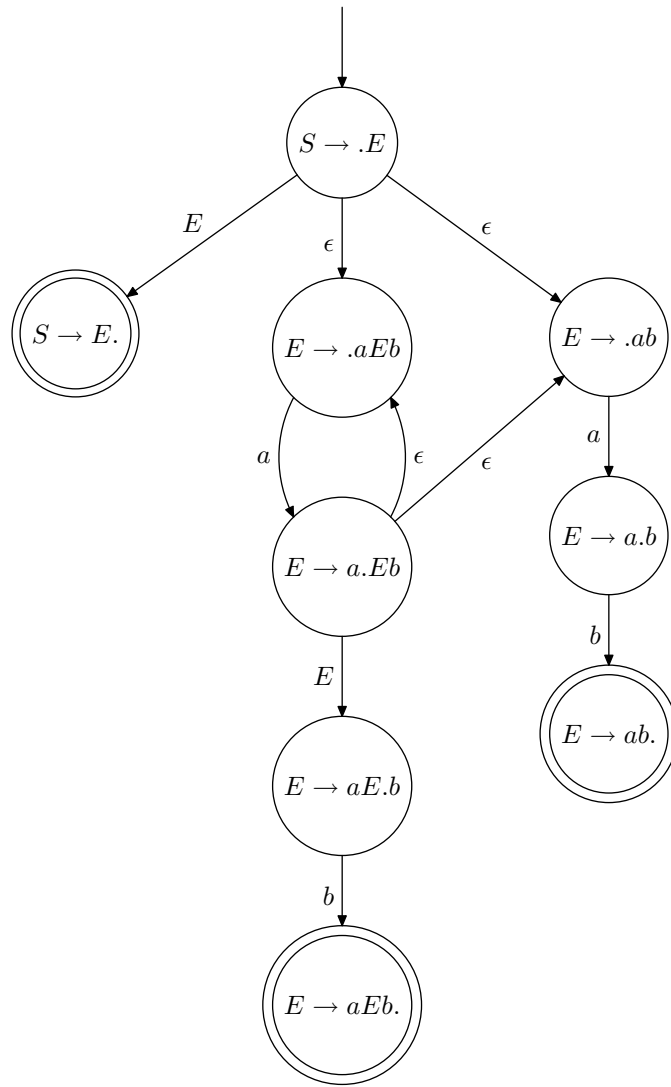
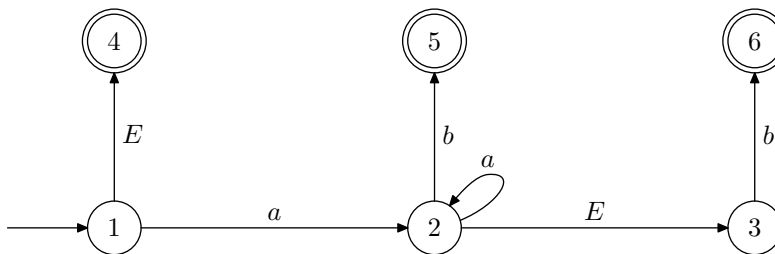


Figure 3.3: NFA for C_{G_1}

Figure 3.4: DFA for C_{G_1}

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

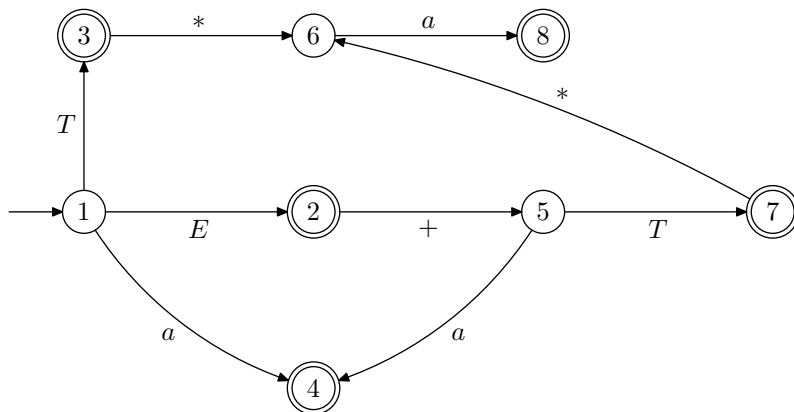
The next example is slightly more complicated.

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 3.5 (where transitions to the “dead state” have been omitted). The internal structure of the states $1, \dots, 8$ is shown below:

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

Figure 3.5: DFA for C_{G_2}

$$5 : E \longrightarrow E + .T$$

$$T \longrightarrow .T * a$$

$$T \longrightarrow .a$$

$$6 : T \longrightarrow T * .a$$

$$7 : E \longrightarrow E + T.$$

$$T \longrightarrow T. * a$$

$$8 : E \longrightarrow T * a.$$

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*.

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.

3.18 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 $LA(s, A \longrightarrow \gamma)$ 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:

	a	b	$\$$	E
1	s_2			4
2	s_2	s_5		3
3		s_6		
4			acc	
5	r_2	r_2	r_2	
6	r_1	r_1	r_1	

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

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 $\text{action}(s, a) = sj$, 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 $\text{action}(s, a) = rn$, then do the following: First, determine the length $k = |\gamma|$ of the righthand side of the production $n: A \longrightarrow \gamma$. 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 $\text{goto}(p, A)$ 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 $\text{action}(s, \$) = \text{acc}$, 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 $S \rightarrow E$).

This is a general property of LR -parsers.

