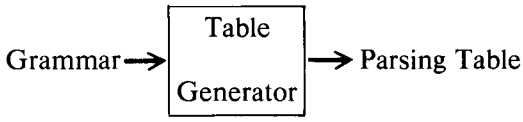
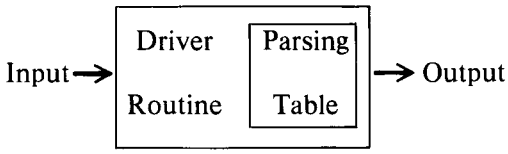


(a) Generating the parser.



(b) Operation of the parser.

Fig. 6.1. Generating an LR parser.

There are many different parsing tables that can be used in an LR parser for a given grammar. Some parsing tables may detect errors sooner than others, but they all accept the same sentences, exactly the sentences generated by the grammar. In this chapter we shall give three different techniques for producing LR parsing tables. The first method, called simple LR (SLR for short), is easiest to implement. Unfortunately, it may fail to produce a table for certain grammars on which the other methods succeed. The second method, called canonical LR, is the most powerful and will work on a very large class of grammars. Unfortunately, the canonical LR method can be very expensive to implement. The third method, called lookahead LR (LALR for short), is intermediate in power between the SLR and the canonical LR methods. The LALR method will work on most programming-language grammars and, with some effort, can be implemented efficiently. We then show how ambiguous grammars can be used to simplify the description of languages and produce efficient parsers.

6.1 LR Parsers

Figure 6.2 depicts an LR parser. The parser has an input, a stack, and a parsing table. The input is read from left to right, one symbol at a time. The stack contains a string of the form $s_0X_1s_1X_2s_2 \cdots X_ms_m$, where s_m is on top. Each X_i is a grammar symbol and each s_i is a symbol called a *state*. Each state symbol summarizes the information contained in the stack below it and is used to guide the shift-reduce decision. In an actual implementation, the grammar symbols need not appear on the stack. We include them there only to help explain the behavior of an LR parser. The parsing table consists of two parts, a parsing action function ACTION and a goto function GOTO.

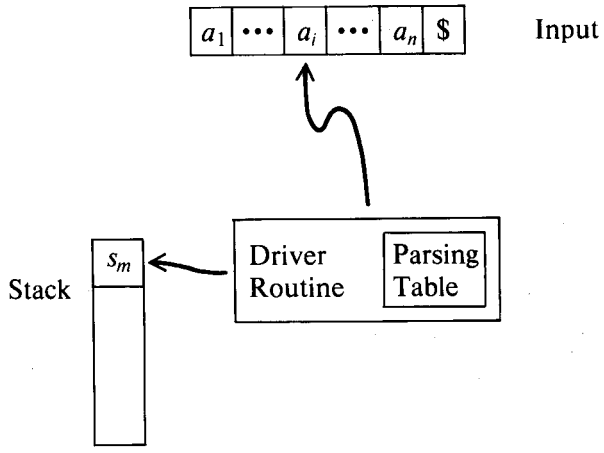


Fig. 6.2. LR parser.

The program driving the LR parser behaves as follows. It determines s_m, the state currently on top of the stack, and a_i, the current input symbol. It then consults ACTION$[s_m, a_i]$, the parsing action table entry for state s_m and input a_i. The entry ACTION$[s_m, a_i]$ can have one of four values:

1. shift s
2. reduce $A \rightarrow \beta$
3. accept
4. error

The function GOTO takes a state and grammar symbol as arguments and produces a state. It is essentially the transition table of a deterministic finite automaton whose input symbols are the terminals and nonterminals of the grammar.

A *configuration* of an LR parser is a pair whose first component is the stack contents and whose second component is the unexpended input:

$$(s_0\ X_1\ s_1\ X_2\ s_2\ \cdots\ X_m\ s_m,\quad a_i\ a_{i+1}\ \cdots\ a_n\ \$)$$

The next move of the parser is determined by reading a_i, the current input symbol, and s_m, the state on top of the stack, and then consulting the parsing action table entry ACTION$[s_m, a_i]$. The configurations resulting after each of the four types of move are as follows:

1. If ACTION$[s_m, a_i]$ = shift s, the parser executes a shift move, entering the configuration

$$(s_0\ X_1\ s_1\ X_2\ s_2\ \cdots\ X_m\ s_m\ a_i\ s,\quad a_{i+1}\ \cdots\ a_n\ \$)$$

Here the parser has shifted the current input symbol a_i and the next state $s = \text{GOTO}[s_m, a_i]$ onto the stack; a_{i+1} becomes the new current input symbol.

2. If $\text{ACTION}[s_m, a_i] = \text{reduce } A \rightarrow \beta$, then the parser executes a reduce move, entering the configuration

$$(s_0 X_1 s_1 X_2 s_2 \ldots X_{m-r} s_{m-r} A s, \quad a_i a_{i+1} \ldots a_n \$)$$

where $s = \text{GOTO}[s_{m-r}, A]$ and r is the length of β, the right side of the production. Here the parser first popped $2r$ symbols off the stack (r state symbols and r grammar symbols), exposing state s_{m-r}. The parser then pushed both A, the left side of the production, and s, the entry for $\text{ACTION}[s_{m-r}, A]$, onto the stack. The current input symbol is not changed in a reduce move. For the LR parsers we shall construct, $X_{m-r+1} \cdots X_m$, the sequence of grammar symbols popped off the stack, will always match β, the right side of the reducing production.

3. If $\text{ACTION}[s_m, a_i]$ = accept, parsing is completed.

4. If $\text{ACTION}[s_m, a_i]$ = error, the parser has discovered an error and calls an error recovery routine.

The LR parsing algorithm is very simple. Initially the LR parser is in the configuration $(s_0, a_1 a_2 \ldots a_n \$)$ where s_0 is a designated initial state and $a_1 a_2 \ldots a_n$ is the string to be parsed. Then the parser executes moves until an accept or error action is encountered. All our LR parsers behave in this fashion. The only difference between one LR parser and another is the information in the parsing action and goto fields of the parsing table.

Example 6.1. Figure 6.3 shows the parsing action and goto functions of an LR parser for the grammar

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

The codes for the actions are:

1. si means shift and stack state i,
2. rj means reduce by production numbered j,
3. acc means accept,
4. blank means error.

Note that the value of $\text{GOTO}[s, a]$ for terminal a is found in the action field connected with the shift action on input a for state s. The goto field gives $\text{GOTO}[s, A]$ for nonterminals A. Also, bear in mind that we have not yet

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

Fig. 6.3. Parsing table.

explained how the entries for Fig. 6.3 are selected; we shall deal with this issue shortly.

Consider the moves made by the parser on input **id** * **id** + **id**. The sequence of stack and input contents is shown in Fig. 6.4.

	Stack	Input
(1)	0	**id** * **id** + **id** $
(2)	0 **id** 5	* **id** + **id** $
(3)	0 *F* 3	* **id** + **id** $
(4)	0 *T* 2	* **id** + **id** $
(5)	0 *T* 2 * 7	**id** + **id** $
(6)	0 *T* 2 * 7 **id** 5	+ **id** $
(7)	0 *T* 2 * 7 *F* 10	+ **id** $
(8)	0 *T* 2	+ **id** $
(9)	0 *E* 1	+ **id** $
(10)	0 *E* 1 + 6	**id** $
(11)	0 *E* 1 + 6 **id** 5	$
(12)	0 *E* 1 + 6 *F* 3	$
(13)	0 *E* 1 + 6 *T* 9	$
(14)	0 *E* 1	$

Fig. 6.4. Moves of LR parser on **id** * **id** + **id**.

For example, at line (1) the LR parser is in state 0 with **id** the first input symbol. The action in row 0 and column **id** of the action field of Fig.

6.3 is s5, meaning shift and cover the stack with state 5. That is what has happened at line (2): the first token **id** and the state symbol 5 have both been pushed onto the stack, and **id** has been removed from the input.

Then, * becomes the current input symbol, and the action of state 5 on input * is to reduce by $F \rightarrow$ **id**. Two symbols are popped off the stack (one state symbol and one grammar symbol). State 0 is then exposed. Since the goto of state 0 on F is state 3, F and 3 are pushed onto the stack. We now have the configuration in line (3).

Each of the remaining moves are determined similarly. □

LR Grammars

Our primary question is, "How do we construct an LR parsing table from a grammar?" A grammar for which we can construct a parsing table in which every entry is uniquely defined is said to be an *LR grammar.* Unfortunately, there are context-free grammars which are not LR, but these can generally be avoided for typical programming-language constructs. Intuitively, in order for a grammar to be LR, it is sufficient that a left-to-right parser be able to recognize handles when they appear on top of the stack.

An LR parser does not have to scan the entire stack to know when the handle appears on top. Rather, the state symbol on top of the stack contains all the information it needs. It is a remarkable fact that if it is possible to recognize a handle knowing only what is in the stack, then a finite automaton can, by reading the stack from bottom to top, determine what handle, if any, is on top of the stack. The driver routine of an LR parser is essentially such a finite automaton. It need not, however, read the stack on every move. The state symbol stored on top of the stack is the state the handle-recognizing finite automaton would be in if it had read the stack from bottom to top. Thus, the LR parser can determine from the state on top of the stack everything that it needs to know about what is in the stack.

Another source of information than an LR parser can use to help make its shift-reduce decisions is the next k input symbols. In practice, k=0 or k=1 is sufficient, and we shall only consider LR parsers with $k \leqslant 1$ here. A grammar that can be parsed by an LR parser examining up to k input symbols on each move is called an *LR(k) grammar.*

It is worth noting that the LR requirement that we be able to recognize the occurrence of the right side of a production, having seen what is derived from that right side, is far less stringent than the requirement for a predictive parser, namely that we be able to recognize the apparent use of the production seeing only the first symbol it derives. Thus, it should be no surprise that LR parsers are more general than predictive parsers.

```
procedure ITEMS(G');
begin
    C := {CLOSURE({S' → ·S})};
    repeat
        for each set of items I in C and each grammar symbol X
                such that GOTO(I, X) is not empty and is not in C
            do add GOTO(I, X) to C
    until no more sets of items can be added to C
end
```

Fig. 6.6 The sets-of-items construction.

I_0: $E' \to \cdot E$
$E \to \cdot E+T$
$E \to \cdot T$
$T \to \cdot T*F$
$T \to \cdot F$
$F \to \cdot(E)$
$F \to \cdot$**id**

I_1: $E' \to E\cdot$
$E \to E\cdot+T$

I_2: $E \to T\cdot$
$T \to T\cdot*F$

I_3: $T \to F\cdot$

I_4: $F \to (\cdot E)$
$E \to \cdot E+T$
$E \to \cdot T$
$T \to \cdot T*F$
$T \to \cdot F$
$F \to \cdot(E)$
$F \to \cdot$**id**

I_5: $F \to$ **id**$\cdot$

I_6: $E \to E+\cdot T$
$T \to \cdot T*F$
$T \to \cdot F$
$F \to \cdot(E)$
$F \to \cdot$**id**

I_7: $T \to T*\cdot F$
$F \to \cdot(E)$
$F \to \cdot$**id**

I_8: $F \to (E\cdot)$
$E \to E\cdot+T$

I_9: $E \to E+T\cdot$
$T \to T\cdot*F$

I_{10}: $T \to T*F\cdot$

I_{11}: $F \to (E)\cdot$

Fig. 6.7. Collection of sets of items.

If each state of D is a final state and I_0 is the initial state, then D recognizes exactly the viable prefixes of grammar (6.1). This is no accident. For every grammar G, the GOTO function of the canonical collection of sets of items defines a deterministic finite automaton that recognizes the viable

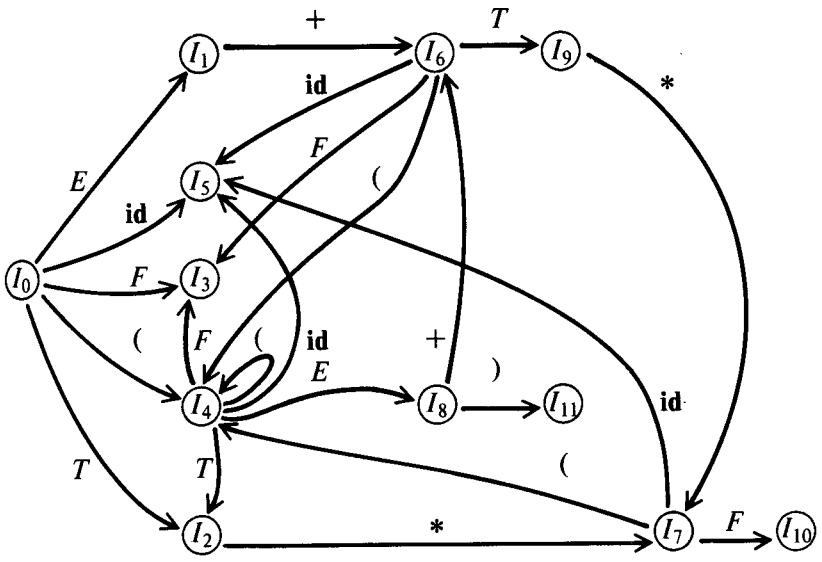


Fig. 6.8. Deterministic finite automaton D.

prefixes of G. In fact, one can visualize a nondeterministic finite automaton N whose states are the items themselves. There is a transition from $A \to \alpha \cdot X\beta$ to $A \to \alpha X \cdot \beta$ labeled X, and there is a transition from $A \to \alpha \cdot B\beta$ to $B \to \cdot\gamma$ labeled ϵ. Then CLOSURE(I) for set of items (states of N) I is exactly the ϵ-CLOSURE of a set of NFA states defined in Section 3.4. GOTO(I, X) gives the transition from I on symbol X in the DFA constructed from N by the subset construction. Viewed in this way, the procedure ITEMS(G') above is just the subset construction itself applied to the NFA N constructed from G' as we have described.

Valid Items

We say item $A \to \beta_1 \cdot \beta_2$ is *valid* for a viable prefix $\alpha\beta_1$ if there is a derivation $S' \underset{\text{rm}}{\overset{*}{\Rightarrow}} \alpha A w \underset{\text{rm}}{\overset{*}{\Rightarrow}} \alpha\beta_1\beta_2 w$. In general an item will be valid for many viable prefixes. The fact that $A \to \beta_1 \cdot \beta_2$ is valid for $\alpha\beta_1$ tells us a lot about whether to shift or reduce when we find $\alpha\beta_1$ on the parsing stack. In particular, if $\beta_2 \neq \epsilon$, then it suggests that we have not yet shifted the handle onto the stack, so shift is our move. If $\beta_2 = \epsilon$, then it looks as if $A \to \beta_1$ is the handle, and we should reduce by this production. Of course, two valid items may tell us to do different things for the same viable prefix. Some of these conflicts can be resolved by looking at the next input symbol, but we should not suppose that all parsing action conflicts can be

resolved if the LR method is used to construct a parsing table for an arbitrary grammar.

We can easily compute the set of valid items for each viable prefix that can appear on the stack of an LR parser. In fact, it is a major theorem of LR parsing theory that the set of valid items for a viable prefix γ is exactly the set of items reached from the initial state along a path labeled γ in the DFA constructed from the canonical collection of sets of items with transitions given by GOTO. In essence, the set of valid items embodies all the useful information that can be gleaned from the stack. While we shall not prove this theorem here, we shall give an example.

Example 6.6. Let us consider grammar (6.1) again, whose sets of items and GOTO function are exhibited in Figs. 6.7 and 6.8. The string $E + T *$ is a viable prefix of (6.1). The automaton of Fig. 6.8 will be in state I_7 after having read $E + T *$. State I_7 contains the items

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

which are precisely the items valid for $E + T *$. To see this, consider the following three rightmost derivations

(1) $E' \Rightarrow E$
$\Rightarrow E + T$
$\Rightarrow E + T * F$
$\Rightarrow E + T * \mathbf{id}$
$\Rightarrow E + T * F * \mathbf{id}$

(2) $E' \Rightarrow E$
$\Rightarrow E + T$
$\Rightarrow E + T * F$
$\Rightarrow E + T * (E)$

(3) $E' \Rightarrow E$
$\Rightarrow E + T$
$\Rightarrow E + T * F$
$\Rightarrow E + T * \mathbf{id}$

The first derivation shows the validity of $T \rightarrow T * \cdot F$, the second the validity of $F \rightarrow \cdot(E)$, and the third the validity of $F \rightarrow \cdot \mathbf{id}$ for the viable prefix $E + T *$. It can be shown that there are no other valid items for $E + T *$, and we leave a proof to the interested reader. □

In summary, when we construct C, the canonical collection of sets of items for an augmented grammar G', the sets of items become the states of a deterministic finite automaton D that recognizes the viable prefixes of G'. The GOTO function on C becomes the state transition function of D. In the next section we show how to convert D into an LR parsing table for G'.

6.3 Constructing SLR Parsing Tables

This section shows how to construct the SLR parsing action and goto functions from the deterministic finite automaton that recognizes viable prefixes. It will not produce uniquely-defined parsing action tables for all grammars but does succeed on many grammars for programming languages. Given a grammar G, we augment G to produce G', and from G' we construct C, the canonical collection of sets of items for G'. We construct ACTION, the parsing action function, and GOTO, the goto function, from C using the following "simple" LR (SLR for short) parsing table construction technique. It requires us to know FOLLOW(A) for each nonterminal A of a grammar (see Section 5.5).

Algorithm 6.1. Construction of an SLR parsing table.

Input. C, the canonical collection of sets of items for an augmented grammar G'.

Output. If possible, an LR parsing table consisting of a parsing action function ACTION and a goto function GOTO.

Method. Let $C = \{I_0, I_1, \ldots, I_n\}$. The states of the parser are $0, 1, \ldots, n$, state i being constructed from I_i. The parsing actions for state i are determined as follows:

1. If $[A \rightarrow \alpha \cdot a\beta]$ is in I_i and GOTO(I_i, a) = I_j, then set ACTION[i, a] to "shift j." Here a is a terminal.
2. If $[A \rightarrow \alpha \cdot]$ is in I_i, then set ACTION[i, a] to "reduce $A \rightarrow \alpha$" for all a in FOLLOW(A).
3. If $[S' \rightarrow S \cdot]$ is in I_i, then set ACTION[i, \$] to "accept."

If any conflicting actions are generated by the above rules, we say the grammar is not SLR(1).† The algorithm fails to produce a valid parser in this case.

† Recall that the 1 in SLR(1) indicates that one input symbol is used to help resolve conflicts.

The goto transitions for state i are constructed using the rule:

4. If GOTO$(I_i, A) = I_j$, then GOTO$[i, A] = j$.
5. All entries not defined by rules (1) through (4) are made "error."
6. The initial state of the parser is the one constructed from the set of items containing $[S' \rightarrow \cdot S]$. □

The parsing table consisting of the parsing action and goto functions determined by Algorithm 6.1 is called the *SLR table for G*. An LR parser using the SLR table for G is called the SLR parser for G, and a grammar having an SLR parsing table is said to be *SLR(1)*.

Example 6.7. Let us construct the SLR table for grammar (6.1). The canonical collection of sets of items for (6.1) was shown in Fig. 6.7. Consider I_0:

$$E' \rightarrow \cdot E$$
$$E \rightarrow \cdot E+T$$
$$E \rightarrow \cdot T$$
$$T \rightarrow \cdot T*F$$
$$T \rightarrow \cdot F$$
$$F \rightarrow \cdot(E)$$
$$F \rightarrow \cdot \mathbf{id}$$

The item $F \rightarrow \cdot(E)$ gives rise to the entry ACTION[0, (] = shift 4, the item $F \rightarrow .\mathbf{id}$ to the entry ACTION[0, **id**] = shift 5.

Consider I_1:

$$E' \rightarrow E\cdot$$
$$E \rightarrow E\cdot+T$$

The first item yields ACTION[1, \$] = accept, the second yields ACTION[1, +] = shift 6.

Consider I_2:

$$E \rightarrow T\cdot$$
$$T \rightarrow T\cdot*F$$

Since FOLLOW$(E) = \{\$, +,)\}$, the first item makes ACTION[2, \$] = ACTION[2, +] = ACTION[2,)] = reduce $E \rightarrow T$. The second item makes ACTION[2, *] = shift 7. Continuing in this fashion we obtain the parsing action and goto tables which were shown in Fig. 6.3. □

Example 6.8. Every SLR(1) grammar is unambiguous, but there are many unambiguous grammars that are not SLR(1). Consider the grammar with