6
LR Parsing

Because of its power and efficiency, LR parsers are commonly used for the syntax-checking phase of a compiler. In this chapter, we study the operation and construction of LR parsers. Tools for the automatic construction of such parsers are available for a variety of platforms. These parser generators are useful not only because they automatically construct tables that drive LR parsers, but also because they are powerful diagnostic tools for developing or modifying grammars.

The basic properties and actions of a generic LR parser are introduced in Sections 6.1 and 6.2. Section 6.3 presents the most basic table-construction method for LR parsers. Section 6.4 considers problems that prevent automatic LR parser construction. Sections 6.5 and 6.6 discuss table-building algorithms of increasing sophistication and power. Of particular interest is the LALR(1) technique, which is used in most LR parser generators. The formal definition of most modern programming languages includes an LALR(1) grammar to specify the language's syntax.

6.1 Introduction

In Chapter Chapter:global:five, we learned how to construct LL (top-down) parsers based on **context-free grammars** (CFGs) that had certain properties. The fundamental concern of a LL parser is which production to choose in expanding a given nonterminal. This choice is based on the parser's current state and on a peek at the unconsumed portion of the parser's input string. The derivations and parse trees produced by LL parsers are easy to follow: the leftmost nonterminal is expanded at each step, and the parse tree grows systematically—top-down, from left to right. The LL parser begins with the tree's root, which is labeled with the grammar's goal symbol. Suppose that A is the next nonterminal to be expanded, and that the parser chooses the production $A \rightarrow \gamma$. In the parse tree, the node corresponding to this A is supplied with children that are labeled with the symbols in γ.

In this chapter, we study LR (bottom-up) parsers, which appear to operate in the reverse direction of LL parsers.

- The LR parser begins with the parse tree's leaves and moves toward its root.
- The *rightmost* derivation is produced *in reverse*.
- An LR parser uses a grammar rule to replace the rule's **right-hand side** (RHS) with its **left-hand side** (LHS).

Figures Figure:four:tdparse and Figure:four:buparse illustrate the differences between a top-down and bottom-up parse. Section 6.2 considers bottom-up parses in greater detail.

Unfortunately, the term “LR” denotes both the generic bottom-up parsing engine as well as a particular technique for building the engine's tables. Actually, the style of parsing considered in this chapter is known by the following names.

- **Bottom-up**, because the parser works its way from the terminal symbols to the grammar's goal symbol
- **Shift-reduce**, because the two most prevalent actions taken by the parser are to *shift* symbols onto the parse stack and to *reduce* a string of such symbols located at the top-of-stack to one of the grammar's nonterminals
- **LR**(k), because such parsers scan the input from the left (L) producing a rightmost derivation (R) in reverse, using k symbols of lookahead

In an LL parser, each state is committed to expand a particular nonterminal. On the other hand, an LR parser can concurrently anticipate the eventual success of multiple nonterminals. This flexibility makes LR parsers more general than LL parsers. For example, LL parsers cannot accommodate left-recursive grammars and they cannot automatically handle the “dangling-else” problem described in Chapter Chapter:global:five. LR parsers have no systematic problem with either of these areas.

Tools for the automatic construction of LR parsers are available for a variety of platforms, including ML, Java, C, and C++. Chapter Chapter:global:seven discusses such tools in greater detail. It is important to note that a parser generator for a given platform performs syntax analysis for the language of its provided grammar. For example, the Yacc is a popular parser generator that emits C code. If Yacc is given a grammar for the syntax of FORTRAN, then the resulting parser compiles using C but performs syntax analysis for FORTRAN. The syntax of most modern programming languages is defined by grammars that are suitable for automatic parser generation using LR techniques.

1	Start	$\rightarrow$	E $
2	E	$\rightarrow$	plus E E
3		$\mid$	num

Rule	Derivation
1	Start $\Rightarrow_{\mathrm{rm}}$ E $
2	$\Rightarrow_{\mathrm{rm}}$ plus E E $
3	$\Rightarrow_{\mathrm{rm}}$ plus E num $
3	$\Rightarrow_{\mathrm{rm}}$ plus num num $

Figure 6.1: Rightmost derivation of plus num num $.

6.2 Shift-Reduce Parsers

In this section, we examine the operation of an LR parser, assuming that an LR parse table has already been constructed to guide the parser's actions. The reader may be understandably curious about how the table's entries are determined. However, table-construction techniques are best considered after obtaining a solid understanding of an LR parser's operation.

We describe the operation of an LR parser informally in Sections 6.2.1 and 6.2.2. Section 6.2.3 describes a generic LR parsing engine whose actions are guided by the LR parse table defined in Section 6.2.4. Section 6.2.5 presents LR(k) parsing more formally.

6.2.1 LR Parsers and Rightmost Derivations

One method of understanding an LR parse is to appreciate that such parses construct rightmost derivations in reverse. Given a grammar and a rightmost derivation of some string in its language, the sequence of productions applied by an LR parser is the sequence used by the rightmost derivation—played backwards. Figure 6.1 shows a grammar and the rightmost derivation of a string in the grammar's language. The language is suitable for expressing sums in a prefix (Lisp-like) notation. Each step of the derivation is annotated with the production number used at that step. For this example, the derivation uses Rules 1, 2, 3, and 3.

A bottom-up parse is accomplished by playing this sequence backwards—Rules 3, 3, 2, and 1. In contrast with LL parsing, an LR parser finds the RHS of a production and replaces it with the production's LHS. First, the leftmost num is **reduced** to an E by the rule E$\rightarrow$num. This rule is applied again to obtain plus E E $. The sum is then reduced by E$\rightarrow$plus E E to obtain E $. This can then be reduced by Rule 1 to the goal symbol Start.

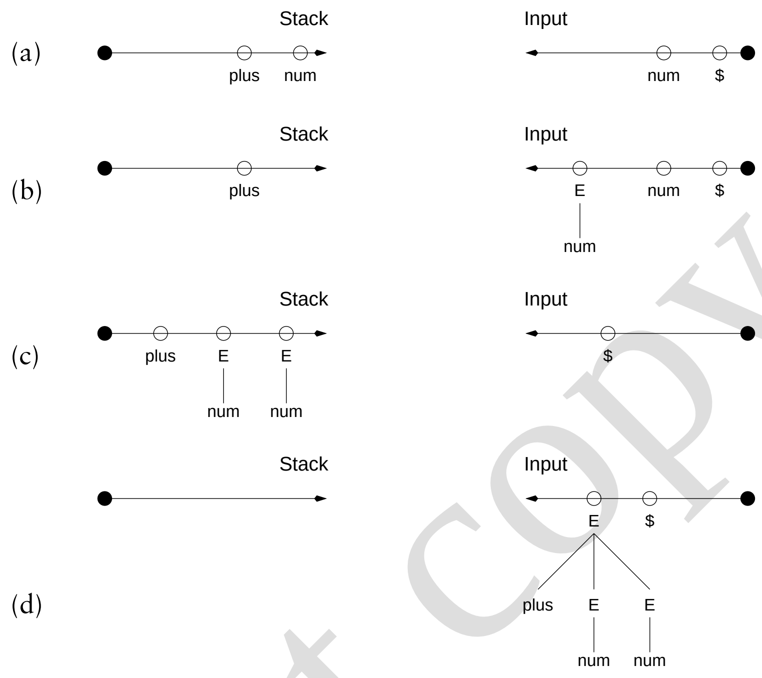


Figure 6.2: Bottom-up parsing resembles knitting.

6.2.2 LR Parsing as Knitting

Section 6.2.1 presents the order in which productions are applied to perform a bottom-up parse. We next examine *how* the RHS of a production is found so that a reduction can occur. The actions of an LR parser are analogous to *knitting*. Figure 6.2 illustrates this by showing a parse in progress for the grammar and string of Figure 6.1. The right needle contains the currently unprocessed portion of the string. The left needle is the parser's stack, which represents the processed portion of the input string.

A **shift** operation transfers a symbol from the right needle to the left needle. When a **reduction** by the rule $\mathsf{A}\rightarrow\gamma$ is performed, the symbols in γ must occur at the pointed end of the left needle—at the *top* of the parse stack. Reduction by $\mathsf{A}\rightarrow\gamma$ removes the symbols in γ and prepends the LHS symbol A to the unprocessed input of the right needle. A is then treated as an input symbol to be shifted onto the left needle. To illustrate the parse tree under construction, Figure 6.2 shows the symbols in γ as children of A.

We now follow the parse that is illustrated in Figure 6.2. In Figure 6.2(a), the left needle shows that two shifts have been performed. With plus num on the left needle, it is time to reduce by $\mathsf{E}\rightarrow\mathsf{num}$. Figure 6.2(b) shows the effect of this reduction, with the resulting E prepended to the input. This same sequence of

```
call $Stack$.PUSH( $StartState$ )
$accepted \leftarrow$ false
while not $accepted$ do
    $action \leftarrow Table[Stack$.TOS( )][$InputStream$.PEEK( )]        1
    if $action =$ shift $s$
    then
        call $Stack$.PUSH( $s$ )                                        2
        if $s \in AcceptStates$                                         3
        then   $accepted \leftarrow$ true
        else   call $InputStream$.ADVANCE( )
    else
        if $action =$ reduce $A \rightarrow \gamma$
        then
            call $Stack$.POP( $|\gamma|$ )                              4
            call $InputStream$.PREPEND( A )                             5
        else
            call ERROR( )                                               6
```

Figure 6.3: Driver for a bottom-up parser.

activities is repeated to obtain the state shown in Figure 6.2(c)—the left needle contains plus E E. When reduced by E→plus E E, we obtain Figure 6.2(d). The resulting E $ would then be shifted, reduced by Start→E $, and the parse would be accepted.

Based on the input string and the sequence of shift and reduce actions, symbols transfer back and forth between the needles. The "product" of the knitting is the parse tree, shown on the left needle if the input string is accepted.

6.2.3 LR Parsing Engine

Before considering an example in some detail, we present a simple driver for our shift-reduce parser in Figure 6.3. The parsing engine is driven by a table, whose entries are discussed in Section 6.2.4. The table is indexed at Step **1**, using the parser's *current state* and the next (unprocessed) input symbol. The **current state** of the parser is defined by the contents of the parser's stack. To avoid rescanning the stack's contents prior to each parser action, state information is computed and stored with each symbol shifted onto the stack. Thus, Step **1** need only consult the state information associated with the stack's topmost symbol. The parse table calls for a shift or reduce as follows.

- Step **2** performs a shift of the next input symbol to state *s*.
- A reduction occurs at Steps **4** and **5**. The RHS of a production is popped off the stack and its LHS symbol is prepended to the input.

The parser continues to perform shift and reduce actions until one of the following situations occurs.

- The input is reduced to the grammar's goal symbol at Step **3**. The input string is **accepted**.
- No valid action is found at Step **1**. In this case, the input string has a syntax error.

6.2.4 The LR Parse Table

We have seen that an LR parse constructs a rightmost derivation in reverse. Each reduction step in the LR parse uses a grammar rule such as $\mathsf{A}\rightarrow\gamma$ to replace γ by A. A sequence of **sentential forms** is thus constructed, beginning with the input string and ending with the grammar's goal symbol.

Given a sentential form, the **handle** is defined as the sequence of symbols that will next be replaced by reduction. The difficulties lie in identifying the handle and in knowing which production to employ in the reduction (should there be multiple productions with the same RHS). These activities are choreographed by the parse table.

A suitable parse table for the grammar in Figure 6.4 is shown in Figure 6.5. This grammar appeared in Figure Figure:five:simplegram to illustrate top-down parsing. Readers familiar with top-down parsing can use this grammar to compare the methods.

To conserve space, a shift and reduce actions are distinguished graphically in our parse tables. A shift to State *s* is denoted by $\boxed{s}$. An unboxed number is the production number for a reduction. Blank entries are error actions, and the parser accepts when the Start symbol is shifted in the parser's starting state. Using the table in Figure 6.5, Figure 6.6 shows the steps of a bottom-up parse. For pedagogical purposes, each stack cell is shown as two elements: $\boxed{\begin{matrix}\mathsf{a}\\ \mathsf{n}\end{matrix}}$. The bottom element *n* is the parser state entered when the cell is pushed. The top symbol *a* is the symbol causing the cell to be pushed. The parsing engine described in Figure 6.3 keeps track only of the state.

The reader should verify that the reductions taken in Figure 6.6 trace a rightmost derivation in reverse. Moreover, the shift actions are essentially implied by the reductions: tokens are shifted until the handle appears at the top of the parse stack, at which time the next reduction in the reverse derivation can be applied.

Of course, the parse table plays a central role in determining the shifts and reductions that are necessary to recognize a valid string. For example, the rule $\mathsf{C}\rightarrow\lambda$ could be applied at any time, but the parse table calls for this only in certain states and only when certain tokens are next in the input stream.

1	Start	$\rightarrow$	S $
2	S	$\rightarrow$	A C
3	C	$\rightarrow$	c
4		|	λ
5	A	$\rightarrow$	a B C d
6		|	B Q
7	B	$\rightarrow$	b B
8		|	λ
9	Q	$\rightarrow$	q
10		|	λ

Rule	Derivation
1	Start $\Rightarrow_{\text{rm}}$ S $
2	$\Rightarrow_{\text{rm}}$ A C $
3	$\Rightarrow_{\text{rm}}$ A c $
5	$\Rightarrow_{\text{rm}}$ a B C d c $
4	$\Rightarrow_{\text{rm}}$ a B d c $
7	$\Rightarrow_{\text{rm}}$ a b B d c $
7	$\Rightarrow_{\text{rm}}$ a b b B d c $
8	$\Rightarrow_{\text{rm}}$ a b b d c $

Figure 6.4: Rightmost derivation of a b b d c $.

6.2.5 LR(k) Parsing

The concept of LR parsing was introduced by Knuth [?]. As was the case with LL parsers, LR parsers are parameterized by the number of lookahead symbols that are consulted to determine the appropriate parser action: an LR(k) parser can peek at the next k tokens. This notion of "peeking" and the term LR(0) are confusing, because even an LR(0) parser must refer to the next input token, for the purpose of indexing the parse table to determine the appropriate action. The "0" in LR(0) refers not to the lookahead at parse-time, but rather to the lookahead used in *constructing* the parse table. At parse-time, LR(0) and LR(1) parsers index the parse table using one token of lookahead; for $k \geq 2$, an LR(k) parser uses k tokens of lookahead.

The number of columns in an LR(k) parse table grows dramatically with k. For example, an LR(3) parse table is indexed by the parse state to select a row, and by the next 3 input tokens to select a column. If the terminal alphabet has n symbols, then the number of distinct three-token sequences is n^3. More generally, an LR(k) table has n^k columns for a token alphabet of size n. To keep the size of parse tables within reason, most parser generators are limited to one token of lookahead. Some parser generators do make selected use of extra lookahead where such information is helpful.

Most of this chapter is devoted to the problems of constructing LR parse tables.

State	a	b	c	d	q	$	Start	S	A	B	C	Q
0	$\boxed{3}$	$\boxed{2}$	8		8	8	accept	$\boxed{4}$	$\boxed{1}$	$\boxed{5}$		
1			$\boxed{11}$			4					$\boxed{14}$	
2		$\boxed{2}$	8	8	8	8				$\boxed{13}$		
3		$\boxed{2}$	8	8						$\boxed{9}$		
4						$\boxed{8}$						
5			10		$\boxed{7}$	10						$\boxed{6}$
6			6			6						
7			9			9						
8						1						
9			$\boxed{11}$	4							$\boxed{10}$	
10				$\boxed{12}$								
11				3		3						
12			5			5						
13			7	7	7	7						
14						2						

Figure 6.5: Parse table for the grammar shown in Figure 6.4.

Before we consider such techniques, it is instructive to formalize the definition of LR(k) in terms of the properties an LR(k) parser must possess. All shift-reduce parsers operate by shifting symbols and examining lookahead information until the end of the handle is found. Then the handle is reduced to a nonterminal, which replaces the handle on the stack. An LR(k) parser, guided by its parse table, must decide whether to shift or reduce, knowing only the symbols already shifted (left context) and the next k lookahead symbols (right context).

A grammar is LR(k) if and only if it is possible to construct an LR parse table such that k tokens of lookahead allows the parser to recognize *exactly* those strings in the grammar's language. An important property of an LR parse table is that each cell accommodates only one entry. In other words, the LR(k) parser is **deterministic**—exactly one action can occur at each step.

We next formalize the properties of an LR(k) grammar, using the following definitions from Chapter Chapter:global:four.

- If $S \Rightarrow^{\star} \beta$, then β is a **sentential form** of a grammar with goal symbol S.
- $\text{First}_k(\alpha)$ is the set of length-k terminal prefixes that can be derived from α.

$\boxed{0}$	Initial Configuration	a b b d c $
$\boxed{0}\,\boxed{a\ 3}$	shift a	b b d c $
$\boxed{0}\,\boxed{a\ 3}\,\boxed{b\ 2}$	shift b	b d c $
$\boxed{0}\,\boxed{a\ 3}\,\boxed{b\ 2}\,\boxed{b\ 2}$	shift b	d c $
$\boxed{0}\,\boxed{a\ 3}\,\boxed{b\ 2}\,\boxed{b\ 2}$	Reduce λ to B	B d c $
$\boxed{0}\,\boxed{a\ 3}\,\boxed{b\ 2}\,\boxed{b\ 2}\,\boxed{B\ 13}$	shift B	d c $
$\boxed{0}\,\boxed{a\ 3}\,\boxed{b\ 2}$	Reduce b B to B	B d c $
$\boxed{0}\,\boxed{a\ 3}\,\boxed{b\ 2}\,\boxed{B\ 13}$	shift B	d c $
$\boxed{0}\,\boxed{a\ 3}$	Reduce b B to B	B d c $
$\boxed{0}\,\boxed{a\ 3}\,\boxed{B\ 9}$	shift B	d c $
$\boxed{0}\,\boxed{a\ 3}\,\boxed{B\ 9}$	Reduce λ to C	C d c $
$\boxed{0}\,\boxed{a\ 3}\,\boxed{B\ 9}\,\boxed{C\ 10}$	shift C	d c $
$\boxed{0}\,\boxed{a\ 3}\,\boxed{B\ 9}\,\boxed{C\ 10}\,\boxed{d\ 12}$	shift d	c $
$\boxed{0}$	Reduce a B C d to A	A c $
$\boxed{0}\,\boxed{A\ 1}$	shift A	c $
$\boxed{0}\,\boxed{A\ 1}\,\boxed{c\ 11}$	shift c	$
$\boxed{0}\,\boxed{A\ 1}$	Reduce c to C	C $
$\boxed{0}\,\boxed{A\ 1}\,\boxed{C\ 14}$	shift C	$
$\boxed{0}$	Reduce A C to S	S $
$\boxed{0}\,\boxed{S\ 4}$	shift S	$
$\boxed{0}\,\boxed{S\ 4}\,\boxed{\$\ 8}$	shift $	$
$\boxed{0}$	Reduce S $ to Start	Start $
$\boxed{0}\,\boxed{Start\ 0}$	shift Start	$
	Accept	

Figure 6.6: Bottom-up parse of a b b d c $.

Assume that in some $\mathsf{LR}(k)$ grammar there are two sentential forms $\alpha\beta w$ and $\alpha\beta y$, with $w, y \in \Sigma^\star$. These sentential forms share a common prefix $\alpha\beta$. Further, with the prefix $\alpha\beta$ on the stack, their k-token lookahead sets are identical: $\mathsf{First}_k(w) = \mathsf{First}_k(y)$. Suppose the parse table calls for reduction by $\mathsf{A} \rightarrow \beta$ given the left context of $\alpha\beta$ and the k-token lookahead present in w; this results in $\alpha\mathsf{A}w$. With the same lookahead information in y, the $\mathsf{LR}(k)$ parser insists on making the same decision: $\alpha\beta y$ becomes $\alpha\mathsf{A}y$. Formally, a grammar is $\mathsf{LR}(k)$ if and only if the following conditions imply $\alpha\mathsf{A}y = \gamma\mathsf{B}x$.

1. $\mathsf{S} \Rightarrow^\star_{\mathrm{rm}} \alpha\mathsf{A}w \Rightarrow_{\mathrm{rm}} \alpha\beta w$
2. $\mathsf{S} \Rightarrow^\star_{\mathrm{rm}} \gamma\mathsf{B}x \Rightarrow_{\mathrm{rm}} \alpha\beta y$
3. $\mathsf{First}_k(w) = \mathsf{First}_k(y)$

This implication allows reduction by $\mathsf{A} \rightarrow \beta$ whenever β is on top-of-stack and the k-symbol lookahead is $\mathsf{First}_k(w)$.

This definition is instructive in that it defines the minimum properties a grammar must possess to be parsable by $\mathsf{LR}(k)$ techniques. It does not tell us how to *build* a suitable $\mathsf{LR}(k)$ parser; in fact, the primary contribution of Knuth's early work was an algorithm for $\mathsf{LR}(k)$ construction. We begin with the simplest LR(0) parser, which lacks sufficient power for most applications. After examining problems that arise in LR(0) construction we turn to the more powerful LR(1) parsing method and its variants.

When LR parser construction fails, the associated grammar may be *ambiguous* (discussed in Section 6.4.1). For other grammars, the parser may require more information about the unconsumed input string (lookahead). In fact, some grammars require an *unbounded* amount of lookahead (Section 6.4.2). In either case, the parser-generator identifies *inadequate* parsing states that are useful for resolving the problem. While it can be shown that there can be no algorithm to determine if a grammar is ambiguous, Section 6.4 describes techniques that work well in practice.

6.3 LR(0) Table Construction

The table-construction methods discussed in this chapter analyze a grammar to devise a parse table suitable for use in the generic parser presented in Figure 6.3. Each symbol in the terminal and nonterminal alphabets corresponds to a column of the table. The analysis proceeds by exploring the state-space of the parser. Each state corresponds to a row of the parse table. Because the state-space is necessarily finite, this exploration must terminate at some point. After the parse table's rows have been determined, analysis then attempts to fill in the cells of the table. Because we are interested only in *deterministic* parsers, each table cell can hold one entry. An important outcome of the LR construction methods is the determination of **inadequate states**—states that lack sufficient information to place at most one parsing action in each column.

LR(0) item	Progress of rule in this state
E→ • plus E E	Beginning of rule
E→plus • E E	Processed a plus, expect an E
E→plus E • E	Expect another E
E→plus E E •	Handle on top-of-stack, ready to reduce

Figure 6.7: LR(0) items for production E→plus E E.

We next consider LR(0) table construction for the grammar shown in Figure 6.1. In constructing the parse table, we are required to consider the parser's progress in recognizing a rule's **right-hand side** (RHS). For example, consider the rule E→plus E E. Prior to reducing the RHS to E, each component of the RHS must be found. A plus must be identified, then two Es must be found. Once these three symbols are on top-of-stack, then it is possible for the parser to apply the reduction and replace the three symbols with the **left-hand side** (LHS) symbol E.

To keep track of the parser's progress, we introduce the notion of an **LR(0) item**—a grammar production with a bookmark that indicating the current progress through the production's RHS. The bookmark is analogous to the "progress bar" present in many applications, indicating the completed fraction of a task. Figure 6.7 shows the possible LR(0) items for the production E→plus E E. A *fresh* item has its marker at the extreme left, as in E→ • plus E E. When the marker is at the extreme right, as in E→plus E E •, we say the item is *reducible*. As a special case, consider the rule A→λ. The symbol "λ" denotes that there is *nothing* on this rule's RHS. We make this clear when representing such rules as items. For A→λ, the only possible item is the reducible A→ • .

We now define a **parser state** as a set of LR(0) items. While each state is formally a set, we drop the usual braces notation and simply list the the set's elements (items). The LR(0) construction algorithm is shown in Figure 6.8.

The start state for our parser—nominally state 0—is formed at Step **7** by including fresh items for each of the grammar's goal-symbol productions. For our example grammar in Figure 6.1, we initialize the start state with Start→ • E $. The algorithm maintains *WorkList*—a set of states that need to be processed by the loop at Step **8**. Each state formed during processing is passed through ADDSTATE, which determines at Step **9** if the set of items has already been identified as a state. If not, then a new state is constructed at Step **10**. The resulting state is added to *WorkList* at Step **11**. The state's row in the parse table is initialized at Step **12**.

The processing of a state begins when the loop at Step **8** extracts a state *s* from the *WorkList*. When COMPUTEGOTO is called on state *s*, the following steps are performed.

1. The **closure** of state *s* is computed at Step **17**. If a nonterminal B appears just after the bookmark symbol (•), then in state *s* we can process a B once one has been found. Transitions from state *s* must include actions that can lead to discovery of a B. CLOSURE in Figure 6.9 returns a set that includes its supplied set of items along with fresh items for B's rules. The addition of fresh

```
function ComputeLR0( Grammar ) : (Set, State)
    States ← ∅
    StartItems ← { Start→ • RHS( p ) | p ∈ ProductionsFor( Start ) }        7
    StartState ← AddState( States, StartItems )
    while (s ← WorkList.ExtractElement( )) ≠ ⊥ do                           8
        call ComputeGoto( States, s )
    return ((States, StartState))
end
function AddState( States, items ) : State
    if items ∉ States                                                       9
    then
        s ← newState(items)                                                 10
        States ← States ∪ { s }
        WorkList ← WorkList ∪ { s }                                         11
        Table[s][⋆] ← error                                                 12
    else   s ← FindState(items)
    return (s)
end
function AdvanceDot( state, X ) : Set
    return ({ A→αX • β | A→α • Xβ ∈ state })                                13
end
```

Figure 6.8: LR(0) construction.

items can trigger the addition of still more fresh items. Because the computed answer is a set, no item is added twice. The loop at Step **14** continues until nothing new is added. Thus, this loop eventually terminates.

2. Step **18** determines transitions from s. When a new state is added during LR(0) construction, Step **12** sets all actions for this state as error. Transitions are defined for each grammar symbol $\mathcal{X}$ that appears after the bookmark. ComputeGoto in Figure 6.9 defines a transition at Step **20** from s to a (potentially new) state that reflects the parser's progress after shifting across *every* item in this state with $\mathcal{X}$ after the bookmark. All such items indicate transition to the same state, because the parsers we construct must operate deterministically. In other words, the parse table has only one entry for a given state and symbol.

We now construct an LR(0) parse table for the grammar shown in Figure 6.1. In Figure 6.10 each state is shown as a separate box. The **kernel** of state s is the set of items explicitly represented in the state. We use the convention of drawing a line within a state to separate the kernel and closure items, as in States 0, 1, and 5; the other states did not require any closure items. Next to each item in each state is the state number reached by shifting the symbol next to the item's bookmark.

```
function CLOSURE( state ) : Set
    ans ← state
    repeat                                                          14
        prev ← ans
        foreach $A \to \alpha \bullet B\gamma \in ans$ do                            15
            foreach $p \in$ PRODUCTIONSFOR( B ) do
                $ans \leftarrow ans \cup \{ B \to \bullet \text{RHS}(p) \}$                    16
    until ans = prev
    return (ans)
end
procedure COMPUTEGOTO( States, s )
    closed ← CLOSURE( s )                                           17
    foreach $\mathcal{X} \in (N \cup \Sigma)$ do                                      18
        RelevantItems ← ADVANCEDOT( closed, $\mathcal{X}$ )                   19
        if $RelevantItems \neq \emptyset$
        then
            Table[s][$\mathcal{X}$] ← shift ADDSTATE( States, RelevantItems )    20
end
```

Figure 6.9: LR(0) closure and transitions.

In Figure 6.10 the transitions are also shown with labeled edges between the states. If a state contains a reducible item, then the state is double-boxed. The edges and double-boxed states emphasize that the basis for LR parsing is a **deterministic finite-state automaton** (DFA), called the **characteristic finite-state machine** (CFSM).

A **viable prefix** of a right sentential form is any prefix that does not extend beyond its handle. Formally, a CFSM recognizes its grammar's viable prefixes. Each transition shifts the symbols of a (valid) sentential form. When the automaton arrives in a double-boxed state, it has processed a viable prefix that ends with a handle. The handle is the RHS of the (unique) reducible item in the state. At this point, a reduction can be performed. The sentential form produced by the reduction can be processed anew by the CFSM. This process can be repeated until the grammar's goal symbol is shifted (successful parse) or the CFSM blocks (an input error).

For the input string plus plus num num num $, Figure 6.11 shows the results of repeatedly presenting the (reverse) derived sentential forms to the CFSM. This approach serves to illustrate how a CFSM recognizes viable prefixes. However, it is wasteful to make repeated passes over an input string's sentential forms. For example, the repeated passes over the input string's first two tokens (plus plus) always cause the parser to enter State 1. Because the CFSM is deterministic, processing a given sequence of vocabulary symbols always has the same effect. Thus, the pars-

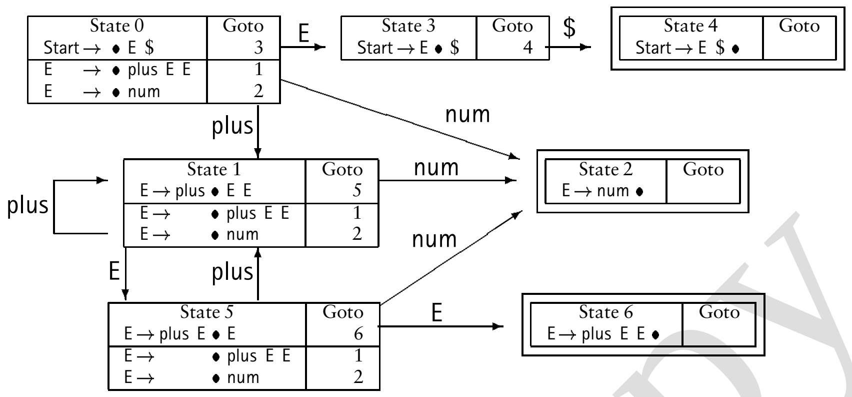


Figure 6.10: LR(0) computation for Figure 6.1

Sentential Prefix	Transitions	Resulting Sentential Form
		plus plus num num num $
plus plus num	States 1, 1, and 2	plus plus E num num $
plus plus E num	States 1, 1, 5, and 2	plus plus E E num $
plus plus E E	States 1, 1, 5, and 6	plus E num $
plus E num	States 1, 5, and 2	plus E E $
plus E E	States 1, 5, and 6	E $
E $	States 1, 3, and 4	Start

Figure 6.11: Processing of plus plus num num num by the LR(0) machine in Figure 6.10.

ing algorithm given in Figure 6.3 does not make repeated passes over the derived sentential forms. Instead, the parse state is recorded after each shift so that current parse state is always associated with whatever symbol happens to be on top of stack. As reductions eat into the stack, the symbol exposed at the new top-of-stack bears the current state, which is the state the CFSM would reach if it rescanned the entire prefix of the current sentential form up to and including the current top-of-stack symbol.

If a grammar is LR(0), then the construction discussed in this section has the following properties:

- Given a syntactically correct input string, the CFSM will block in only double-boxed states, which call for a reduction. The CFSM clearly shows that no progress can occur unless a reduction takes place.

- There is at most one item present in any double-boxed state—the rule that should be applied upon entering the state. Upon reaching such states, the

```
procedure CompleteTable( Table, grammar )
    call ComputeLookahead( )
    foreach state ∈ Table do
        foreach rule ∈ Productions(grammar) do
            call TryRuleInState( state, rule )
    call AssertEntry( StartState, GoalSymbol, accept )                21
end
procedure AssertEntry( state, symbol, action )
    if Table[state][symbol] = error                                   22
    then   Table[state][symbol] ← action
    else
        call ReportConflict( Table[state][symbol], action )           23
end
```

Figure 6.12: Completing an LR(0) parse table.

CFSM has completely processed a rule. The associated item is *reducible*, with the marker moved as far right as possible.

- If the CFSM's input string is syntactically invalid, then the parser will enter a state such that the offending terminal symbol cannot be shifted.

The table established during LR(0) construction at Step **20** in Figure 6.9 is almost suitable for parsing by the algorithm in Figure 6.3. Each state is a row of the table, and the columns represent grammar symbols. Entries are present only where the LR(0) construction allows transition between states—these are the shift actions. To complete the table, we apply the algorithm in Figure 6.12, which establishes the appropriate reduce actions.

For LR(0), the decision to call for a reduce is reflected in the code of Figure 6.13; arrival in a double-boxed state signals a reduction irrespective of the next input token. As reduce actions are inserted, AssertEntry reports any *conflicts* that arise when a given state and grammar symbol call for multiple parsing actions. Step **22** allows an action to be asserted only if the relevant table cell was previously undefined (error). Finally, Step **21** calls for acceptance when the goal symbol is shifted in the table's start state. Given the construction in Figure 6.10 and the grammar in Figure 6.1, LR(0) analysis yields the parse table is shown in Figure 6.14.

6.4 Conflict Diagnosis

Sometimes LR construction is not successful, even for simple languages and grammars. In the following sections we consider table-construction methods that are

```
procedure COMPUTELOOKAHEAD( )
    /*  Reserved for the LALR($k$) computation given in Section 6.5.2  */
end
procedure TRYRULEINSTATE($s, r$)
    if LHS($r$)$\rightarrow$RHS($r$)$\bullet$ $\in s$
    then
        foreach $\mathcal{X} \in (\Sigma \cup N)$ do call ASSERTENTRY($s, \mathcal{X}$, reduce r)
end
```

Figure 6.13: LR(0) version of TRYRULEINSTATE.

State	num	plus	$	Start	E
0	$\boxed{2}$	$\boxed{1}$		accept	$\boxed{3}$
1	$\boxed{2}$	$\boxed{1}$			$\boxed{5}$
2	reduce 3				
3			$\boxed{4}$		
4	reduce 1				
5	$\boxed{2}$	$\boxed{1}$			$\boxed{6}$
6	reduce 2				

Figure 6.14: LR(0) parse table for the grammar in Figure 6.1.

more powerful than LR(0), thereby accommodating a much larger class of grammars. In this section, we examine why *conflicts* arise during LR table construction. We develop approaches for understanding and resolving such conflicts.

The generic LR parser shown in Figure 6.3 is *deterministic*. Given a parse state and an input symbol, the parse table can specify exactly one action to be performed by the parser—shift, reduce, accept, or error. In Chapter Chapter:global:three we tolerated nondeterminism in the scanner specification because we knew of an efficient for transforming a nondeterministic DFA into a deterministic DFA. Unfortunately, no such algorithm is possible for stack-based parsing engines. Some CFGs cannot be parsed deterministically. In such cases, *perhaps* there is another grammar that generates the same language. but for which a deterministic parser can be constructed. There are **context-free languages** (CFLs) that provably cannot be parsed using the (deterministic) LR method (see Exercise 9). However, programming languages are typically designed to be parsed deterministically.

A parse-table **conflict** arises when the table-construction method cannot decide between multiple alternatives for some table-cell entry. We then say that the associated state (row of the parse table) is **inadequate** for that method. An inadequate state for a weaker table-construction algorithm can sometimes be resolved by a stronger algorithm. For example, the grammar of Figure 6.4 is not LR(0)—a mix of shift and reduce actions can be seen in State 0. However, the table-construction algorithms introduced in Section 6.5 resolve the LR(0) conflicts for this grammar.

If we consider the possibilities for multiple table-cell entries, only the following two cases are troublesome for LR(k) parsing.

- **shift/reduce conflicts** exist in a state when table construction cannot use the next k tokens to decide whether to shift the next input token or call for a reduction. The bookmark symbol must occur before a terminal symbol t in one of the state's items, so that a shift of t could be appropriate. The bookmark symbol must also occur at the end of some other item, so that a reduction in this state is also possible.
- **reduce/reduce conflicts** exist when table construction cannot use the next k tokens to distinguish between multiple reductions that could be applied in the inadequate state. Of course, a state with such a conflict must have at least two reducible items.

Other combinations of actions in a table cell do not make sense. For example, it cannot be the case that some terminal t could be shifted but also cause an error. Also, we cannot obtain a shift/shift error: if a state admits the shifting of terminal symbols t and u, then the target state for the two shifts is different, and there is no conflict. Exercise 11 considers the impossibility of a shift/reduce conflict on a nonterminal symbol.

Although the table-construction methods we discuss in the following sections vary in power, each is capable of reporting conflicts that render a state inadequate. Conflicts arise for one of the following reasons.

- The grammar is *ambiguous*. No (deterministic) table-construction method can resolve conflicts that arise due to ambiguity. Ambiguous grammars are considered in Section 6.4.1, but here we summarize possibilities for addressing the ambiguity. If a grammar is ambiguous, then some input string has at least two distinct parse trees.
  - If both parse trees are desirable (as in Exercise 37), then the grammar's language contains a *pun*. While puns may be tolerable in natural languages, they are undesirable in the design of computer languages. A program specified in a computer language should have an unambiguous interpretation.
  - If only one tree has merit, then the grammar can often be modified to eliminate the ambiguity. While there are inherently ambiguous languages (see Exercise 12), computer languages are not designed with this property.
- The grammar is not ambiguous, but the current table-building approach could not resolve the conflict. In this case, the conflict might disappear if one or more of the following approaches is followed:

1	Start	$\rightarrow$	E $
2	E	$\rightarrow$	E plus E
3		|	num

State 0	Goto
Start → • E $	2
E → • E plus E	2
E → • num	1

State 1	Goto
E → num •	

State 2	Goto
E → E • plus E	3
Start → E • $	4

State 3	Goto
E → E plus • E	5
E → • E plus E	5
E → • num	1

State 4	Goto
Start → E $ •	

State 5	Goto
E → E plus E •	
E → E • plus E	3

Figure 6.15: An ambiguous expression grammar.

– The current table-construction method is given more lookahead.

– A more powerful table-construction method is used.

It is possible that no amount of lookahead or table-building power can resolve the conflict, even if the grammar is unambiguous. We consider such a grammar in Section 6.4.2 and in Exercise 36.

When an LR(k) construction algorithm develops an inadequate state, it is an unfortunate but important fact is that it is not possible automatically to decide which of the above problem afflicts the grammar [?]. It is impossible to construct an algorithm to determine if a CFG is ambiguous. It is therefore also impossible to determine whether a bounded amount of lookahead can resolve an inadequate state. As a result, human (instead of mechanical) reasoning is required to understand and repair grammars for which conflicts arise. Sections 6.4.1 and 6.4.2 develop intuition and strategies for such reasoning.

6.4.1 Ambiguous Grammars

Consider the grammar and its LR(0) construction shown in Figure 6.15. The grammar generates sums of numbers using the familiar *infix* notation. In the LR(0) construction, all states are adequate except State 5. In this state a plus can be shifted to arrive in State 3. However, State 5 also allows reduction by the rule E→E plus E. This inadequate state exhibits a shift/reduce conflict for LR(0). To resolve this conflict, it must be decided how to fill in the LR parse table for State 5 and the symbol plus. Unfortunately, this grammar is ambiguous, so a unique entry cannot be determined.

While there is no automatic method for determining if an arbitrary grammar is ambiguous, the inadequate states can provide valuable assistance in finding a string with multiple derivations—should one exist. Recall that a parser state represents transitions made by the CFSM when recognizing viable prefixes. The bookmark

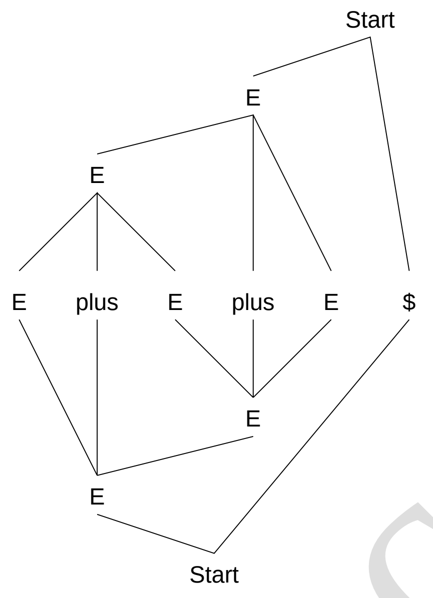


Figure 6.16: Two derivations for E plus E plus E $. The parse tree on top favors reduction in State 5; the parse tree on bottom favors a shift.

symbol shows the progress made so far; symbols appearing after the bookmark are symbols that can be shifted to make progress toward a successful parse. While our ultimate goal is the discovery of an input string with multiple derivations, we begin by trying to find an ambiguous sentential form. Once identified, the sentential form can easily be extended into a terminal string, by replacing nonterminals using the grammar's productions.

Using State 5 in Figure 6.15 as an example, the steps taken to understand conflicts are as follows.

1. Using the parse table or CFSM, determine a sequence of vocabulary symbols that cause the parser to move from the start state to the inadequate state. For Figure 6.15, the simplest such sequence is E plus E, which passes through States 0, 2, 3, and 5. Thus, in State 5 we have E plus E on the top-of-stack; one option is a reduction by E→E plus E. However, with the item E→E • plus E it is also possible to shift a plus and then an E.

2. If we line up the dots of these two items, we obtain a snapshot of what is on the stack upon arrival in this state and what may be successfully shifted in the future. Here we obtain the sentential form prefix E plus E • plus E. The shift/reduce conflict tells us that there are two potentially successful parses. We therefore try to construct two derivation trees for E plus E plus E, one assuming the reduction at the bookmark symbol and one assuming the shift.

1	Start	→	E $
2	E	→	E plus num
3		|	num

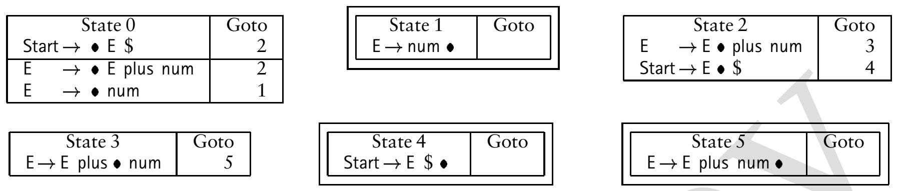


Figure 6.17: Unambiguous grammar for infix sums and its LR(0) construction.

> Completing either derivation may require extending this sentential prefix so that it becomes a sentential form: a string of vocabulary symbols derivable (in two different ways) from the goal symbol.

For our example, E plus E plus E is almost a complete sentential form. We need only append $ to obtain E plus E plus E $.

To emphasize that the derivations are constructed for the same string, Figure 6.16 shows the derivations above and below the sentential form. If the reduction is performed, then the early portion of E plus E plus E $ is structured under a nonterminal; otherwise, the input string is shifted so that the latter portion of the sentential form is reduced first. The parse tree that favors the reduction in State 5 corresponds to **left-association** for addition, while the shift corresponds to **right-association**.

Having analyzed the ambiguity in the grammar of Figure 6.15, we next eliminate the ambiguity by creating a grammar that favors left-association—the reduction instead of the shift. Such a grammar and its LR(0) construction are shown in Figure 6.17. The grammars in Figures 6.15 and 6.17 generate the same language. In fact, the language is regular, denoted by the regular expression num (plus num)* $. So we see that even simple languages can have ambiguous grammars. In practice, diagnosing ambiguity can be more difficult. In particular, finding the ambiguous sentential form may require significant extension of a viable prefix. Exercises 36 and 37 provide practice in finding and fixing a grammar's ambiguity.

6.4.2 Grammars that are not LR(k)

Figure 6.18 shows a grammar and a portion of its LR(0) construction for a language similar to infix addition, where expressions end in either a or b. The complete LR(0) construction is left as Exercise 13. State 2 contains a reduce/reduce conflict. In this state, it is not clear whether num should be reduced to an E or an F. The viable prefix that takes us to State 2 is simply num. To obtain a sentential form, this must be extended either to num a $ or num b $. If we use the former sentential form,

1	Start	→	Exprs $
2	Exprs	→	E a
3		|	F b
4	E	→	E plus num
5		|	num
6	F	→	F plus num
7		|	num

State 0	Goto
Start → • Exprs $	1
Exprs → • E a	4
Exprs → • F b	3
E → • E plus num	4
E → • num	2
F → • F plus num	3
F → • num	2

State 2	Goto
E → num • F → num •	

Figure 6.18: A grammar that is not LR(k).

then F cannot be involved in the derivation. Similarly, if we use the latter sentential form, E is not involved. Thus, progress past num cannot involve more than one derivation, and the grammar is not ambiguous.

Since LR(0) construction failed for the grammar in Figure 6.18, we could try a more ambitious table-construction method from among those discussed in Sections 6.5 and 6.6. It turns out that none can succeed. All LR(k) constructions analyze grammars using k lookahead symbols. If a grammar is LR(k), then there is some value of k for which all states are adequate in the LR(k) construction described in Section 6.6. The grammar in Figure 6.18 is not LR(k) for any k. To see this, consider the following rightmost derivation of num plus ... plus num a.

$$\begin{aligned} \text{Start} &\Rightarrow_{\text{rm}} \text{Exprs \$} \\ &\Rightarrow_{\text{rm}} \text{E a \$} \\ &\Rightarrow_{\text{rm}} \text{E plus num a \$} \\ &\Rightarrow^{\star}_{\text{rm}} \text{E plus ... plus num a \$} \\ &\Rightarrow_{\text{rm}} \text{num plus ... plus num a \$} \end{aligned}$$

A bottom-up parse must play the above derivation backwards. Thus, the first few steps of the parse will be:

$\boxed{0}$	Initial Configuration	num plus ..... plus num a $
$\boxed{0}\ \boxed{\text{num } 2}$	shift num	plus ..... plus num a $

With num on top-of-stack, we are in State 2. A deterministic, bottom-up parser must decide at this point whether to reduce num to an E or an F. If the decision were delayed, then the reduction would have to take place in the middle of the

1	Start	→	E $
2	E	→	E plus num
3		|	E times num
4		|	num

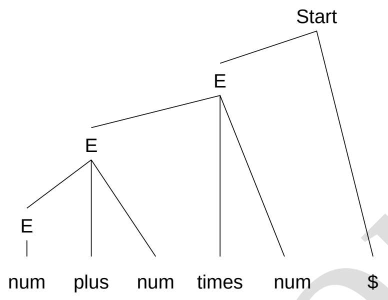


Figure 6.19: Expressions with sums and products.

stack, and this is not allowed. The information needed to resolve the reduce/reduce conflict appears just before the $ symbol. Unfortunately, the relevant a or b could be arbitrarily far ahead in the input, because strings derived from E or F can be arbitrarily long.

In summary, simple grammars and languages can have subtle problems that prohibit generation of a bottom-up parser. It follows that a top-down parser would also fail on such grammars (Exercise 14). The LR(0) construction can provide important clues for diagnosing a grammar's inadequacies; understanding and resolving such conflicts requires human intelligence. Also, the LR(0) construction forms the basis of the SLR(k) and LALR(k) constructions—techniques we consider next.

6.5 Conflict Resolution for LR(0) Tables

While LR(0) construction succeeded for the grammar in Figure 6.17, most grammars require some lookahead during table construction. In this section we consider methods that are based on the LR(0) construction. Where conflicts arise, these methods analyze the grammar to determine if the conflict can be resolved without expanding the number of states (rows) in the LR(0) table. Section 6.5.1 presents the SLR(k) construction, which is simple but does not work as well as the LALR(k) construction introduced in Section 6.5.2.

6.5.1 SLR(k) Table Construction

The SLR(k) method attempts to resolve inadequate states using grammar analysis methods presented in Chapter Chapter:global:four. To demonstrate the SLR(k) construction, we require a grammar that is not LR(0). We begin by extending the grammar in Figure 6.17 to accommodate expressions involving sums and products. Figure 6.19 shows such a grammar along with a parse tree for the string num plus num times num $. Exercise 15 shows that this grammar is LR(0). However, it does not produce the parse trees that structure the expressions appropriately. The parse tree shown in Figure 6.19 structures the computation by adding the first two nums and then multiplying that sum by the third num. As such, the input string 3+4∗7 would produce a value of 49 if evaluation were guided by the computation's parse tree.

A common convention in mathematics is that multiplication has precedence over addition. Thus, the computation $3 + 4 * 7$ should be viewed as adding 3 to the product $4 * 7$, resulting in the value 31. Such conventions are typically adopted in programming language design, in an effort to simplify program authoring and readability. We therefore seek a parse tree that appropriately structures expressions involving multiplication and addition.

To develop the grammar that achieves the desired effect, we first observe that a string in the language of Figure 6.19 should be regarded as a *sum of products*. The grammar in Figure 6.17 generates sums of nums. A common technique to expand a language involves replacing a terminal symbol in the grammar by a nonterminal whose role in the grammar is equivalent. To produce a sum of Ts rather than a sum of nums, we need only replace num with T to obtain the rules for E shown in Figure 6.20. To achieve a sum of products, each T can now derive a product, with the simplest product consisting of a single num. Thus, the rules for T are based on the rules for E, substituting times for plus. Figure 6.20 shows a parse tree for the input string from Figure 6.19, with multiplication having precedence over addition.

Figure 6.21 shows a portion of the LR(0) construction for our precedence-respecting grammar. States 1 and 6 are inadequate for LR(0): in each of these states, there is the possibility of shifting a times or applying a reduction to E. Figure 6.21 shows a sequence of parser actions for the sentential form E plus num times num $, leaving the parser in State 6.

Consider the shift/reduce conflict of State 6. To determine if the grammar in Figure 6.20 is ambiguous, we turn to the methods described in Section 6.4. We proceed by assuming the shift and reduce are *each* possible given the sentential form E plus T times num $.

- If the shift is taken, then we can continue the parse in Figure 6.21 to obtain the parse tree shown in Figure 6.20.
- Reduction by rule E→E plus T yields E times num $, which causes the CFSM in Figure 6.21 to block in State 3 with no progress possible. If we try to reduce using T→num, then we obtain E times T $, which can be further reduced to E times E $. Neither of phrases can be further reduced to the goal symbol.

Thus, E times num $ is not a valid sentential form for this grammar and a reduction in State 6 for this sentential form is inappropriate.

With the item E→E plus T • in State 6, reduction by E→E plus T must be appropriate under some conditions. If we examine the sentential forms E plus T $ and

1	Start	→	E $
2	E	→	E plus T
3		|	T
4	T	→	T times num
5		|	num

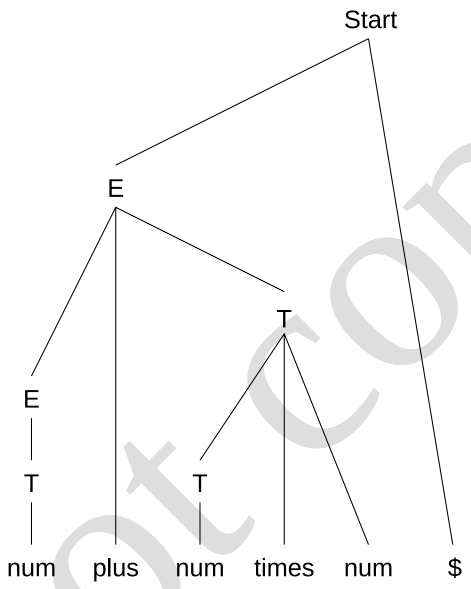


Figure 6.20: Grammar for sums of products.

E plus T plus num $, we see that the E→E plus T must be applied in State 6 when the next input symbol is plus or $, but not times. LR(0) could not selectively call for a reduction in any state; perhaps methods that can consult lookahead information in TRYRULEINSTATE can resolve this conflict.

Consider the sequence of parser actions that could be applied between a reduction by E→E plus T and the next shift of a terminal symbol. Following the reduction, E must be shifted onto the stack. At this point, assume terminal symbol plus is the next input symbol. If the reduction to E can lead to a successful parse, then plus can appear next to E in *some* valid sentential form. An equivalent statement is plus ∈ Follow(E), using the Follow computation from Chapter Chapter:global:four.

SLR(k) parsing uses Follow_k(A) to call for a reduction to A in any state containing a reducible item for A. Algorithmically, we obtain SLR(k) by performing the LR(0) construction in Figure 6.8; the only change is to the method TRYRULEINSTATE, whose SLR(1) version is shown in Figure 6.22. For our example, States 1 and 6 are resolved by computing Follow(E) = { plus, $ }. The SLR(1) parse table that results from this analysis is shown in Figure 6.23.

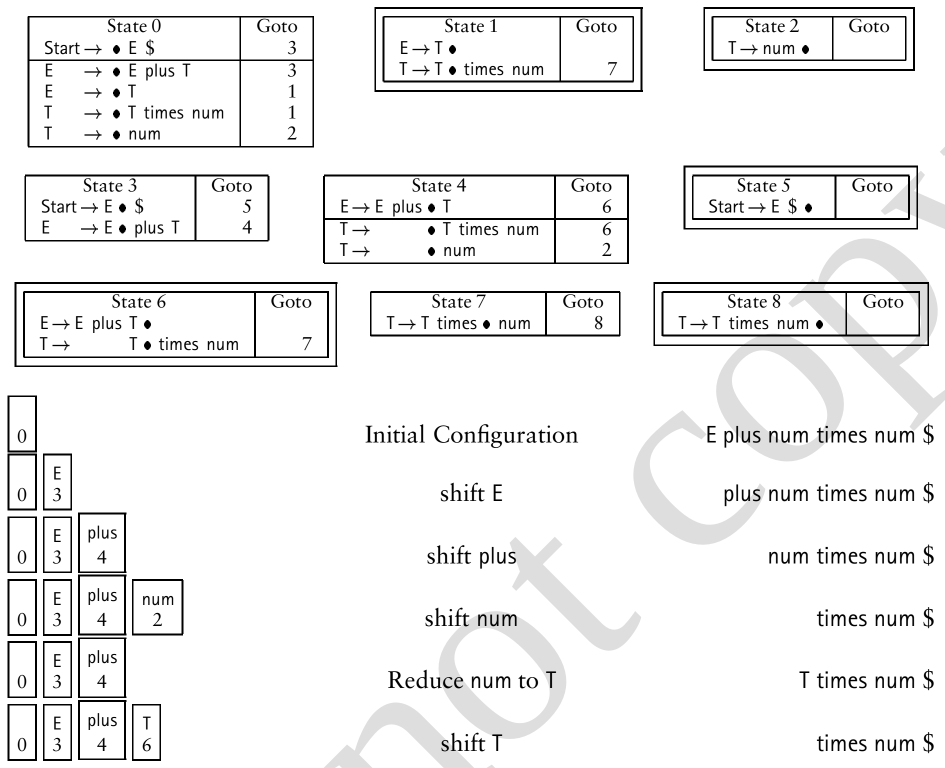


Figure 6.21: LR(0) construction and parse leading to inadequate State 6.

```
procedure TryRuleInState( $s, r$ )
    if LHS( $r$ )→RHS( $r$ ) • $\in s$
    then
        foreach $\mathcal{X} \in$ Follow( LHS( $r$ ) ) do
            call AssertEntry( $s, \mathcal{X}$, reduce r )
end
```

Figure 6.22: SLR(1) version of TryRuleInState.

State	num	plus	times	$	Start	E	T
0	$\boxed{2}$				accept	$\boxed{3}$	$\boxed{1}$
1		3	$\boxed{7}$	3			
2		5	5	5			
3		$\boxed{4}$		$\boxed{5}$			
4	$\boxed{2}$						$\boxed{6}$
5				1			
6		2	$\boxed{7}$	2			
7	$\boxed{8}$						
8		4	4	4			

Figure 6.23: SLR(1) parse table for the grammar in Figure 6.20.

6.5.2 LALR(k) Table Construction

SLR(k) attempts to resolve LR(0) inadequate states using Follow_k information. Such information is computed *globally* for a grammar. Sometimes SLR(k) construction fails only because the Follow_k information is not *rule-specific*. Consider the grammar and its partial LR(0) construction shown in Figure 6.24. This grammar generates the language $\{$ a, ab, ac, xac $\}$. The grammar is not ambiguous, because each of these strings has a unique derivation. However, State 3 has an LR(0) shift/reduce conflict. SLR(k) tries to resolve the shift/reduce conflict, computing $\text{Follow}_k(\text{A}) = \{ \text{b\$}^{k-1}, \text{c\$}^{k-1}, \$^{k} \}$. A can be followed in some sentential form by any number of end-of-string symbols, possibly prefaced by b or c. If we examine States 0 and 3 more carefully, we see that it is not possible for c to occur after the expansion of A in State 3. In the closure of State 0, the fresh item for A was created by the item S→ • A B. Following the shift of A, only b or $ can occur—there is no sentential form A c $. Given the above analysis, we can fix the shift/reduce conflict by modifying the grammar to have two "versions" of A. Using A_1 and A_2, the resulting SLR(1) grammar is shown in Figure 6.25. Here, State 2 is resolved, since $\text{Follow}(\text{A}_1) = \{ \$, \text{b} \}$.

SLR(k) has difficulty with the grammar in Figure 6.24 because Follow sets are computed *globally* across the grammar. Copying productions and renaming nonterminals can cause the Follow computation to become more production-specific, as in Figure 6.25. However, this is tedious and can complicate the understanding and maintenance of a programming language's grammar. In this section we consider LALR(k) parsing, which offers a more specialized computation of "follow" information. Short for "lookahead" LR, the term LALR is not particularly informative—SLR and LR also use lookahead. However, LALR offers superior lookahead analysis

1	Start	→	S $
2	S	→	A B
3		|	a c
4		|	x A c
5	A	→	a
6	B	→	b
7		|	λ

State 0	Goto
Start → • S $	4
S → • A B	2
S → • a c	3
S → • x A c	1
A → • a	3

State 3	Goto
S → a • c	6
A → a •	

Figure 6.24: A grammar that is not SLR(k).

1	Start	→	S $
2	S	→	A_1 B
3		|	a c
4		|	x A_2 c
5	A_1	→	a
6	A_2	→	a
7	B	→	b
8		|	λ

State 0	Goto
Start → • S $	3
S → • A_1 B	4
S → • a c	2
S → • x A_2 c	1
A_1 → • a	2

State 2	Goto
S → a • c	8
A_1 → a •	

Figure 6.25: An SLR(1) grammar for the language defined in Figure 6.24.

for the LR(0) table.

Like SLR(k), LALR(k) is based on the LR(0) construction given in Section 6.3. Thus, an LALR(k) table has the same number of rows (states) as does an LR(0) table for the same grammar. While LR(k) (discussed in Section 6.6) offers more powerful lookahead analysis, this is achieved at the expense of introducing more states.

Due to its balance of power and efficiency, LALR(1) is the most popular LR table-building method. Chapter Chapter:global:seven describes tools that can automatically construct parsers for grammars that are LALR(1). For LALR(1), we redefine the following two methods from Figure 6.13.

COMPUTELOOKAHEAD Figure 6.26 contains code to build and evaluate a *lookahead propagation graph*. This computation establishes *ItemFollow*((*state*, *item*)) as the set of (terminal) symbols that can follow the reducible *item* as of the given *state*. The propagation graph is described below in greater detail.

TRYRULEINSTATE In the LALR(1) version, the *ItemFollow* set for a reducible item is consulted for symbols that can apparently occur after this reduction.

Propagation Graph

We have not formally named the LR(0) items, but an item occurs at most once in any state. Thus, the pair $(s, \mathsf{A}\rightarrow\alpha \bullet \beta)$ suffices to identify the item $\mathsf{A}\rightarrow\alpha \bullet \beta$ that occurs in state s. For each valid state and item pair, Step **24** in Figure 6.26 creates a vertex v in the propagation graph. Each item in an LR(0) construction is represented by a vertex in this graph. The *ItemFollow* sets are initially empty, except for the augmenting item $\mathsf{Start}\rightarrow \bullet \mathsf{S}\ \$$ in the LR(0) start-state. Edges are placed in the graph between items i and j when the symbols that follow the reducible form of item i should be included in the corresponding set of symbols for item j. For the purposes of lookahead analysis, the input string can be considered to have an arbitrary number of "end-of-string" characters. Step **25** forces the entire program, derived from any rule for Start, to be followed by $.

Step **26** of the algorithm in Figure 6.26 considers items of the form $\mathsf{A}\rightarrow\alpha \bullet \mathsf{B}\gamma$ in state s. This generic item indicates that the bookmark is just before the symbol B, with grammar symbols in α appearing before the bookmark, and grammar symbols in γ appearing after B. Note that α or γ could be absent, in which case $\alpha = \lambda$ or $\gamma = \lambda$, respectively. The symbol B is always present, unless the grammar rule is $\mathsf{A}\rightarrow\lambda$. The lookahead computation is specifically concerned with $\mathsf{A}\rightarrow\alpha \bullet \mathsf{B}\gamma$ when B is a nonterminal, because CLOSURE in Figure 6.9 adds items to state s for each of B's productions. The symbols that can follow B depend on γ, which is either absent or present in the item. Also, even when γ is present, it is possible that $\gamma \Rightarrow^\star \lambda$. The algorithm in Figure 6.26 takes these cases into account as follows.

- For the item $\mathsf{A}\rightarrow\alpha \bullet \mathsf{B}\gamma$, any symbol in $\mathsf{First}(\gamma)$ can follow each closure item $\mathsf{B}\rightarrow \bullet \delta$. This is true even when γ is absent: in such cases, $\mathsf{First}(\lambda) = \emptyset$. Thus, Step **28** places symbols from $\mathsf{First}(\gamma)$ in the *ItemFollow* sets for each $\mathsf{B}\rightarrow \bullet \delta$ in state s.

```
procedure COMPUTELOOKAHEAD( )
    call BUILDITEMPROPGRAPH( )
    call EVALITEMPROPGRAPH( )
end
procedure BUILDITEMPROPGRAPH( )
    foreach $s \in States$ do
        foreach $item \in state$ do
            $v \leftarrow Graph$.ADDVERTEX( $(s, item)$ )                          24
            $ItemFollow(v) \leftarrow \emptyset$
    foreach $p \in$ PRODUCTIONSFOR( Start ) do
        $ItemFollow((StartState, \text{Start} \rightarrow \bullet \text{RHS}(p))) \leftarrow \{ \$ \}$    25
    foreach $s \in States$ do
        foreach $\text{A} \rightarrow \alpha \bullet \text{B}\gamma \in s$ do                       26
            $v \leftarrow Graph$.FINDVERTEX( $(s, \text{A} \rightarrow \alpha \bullet \text{B}\gamma)$ )
            call $Graph$.ADDEDGE( $v, (Table[s][B], \text{A} \rightarrow \alpha\text{B} \bullet \gamma)$ )     27
            foreach $(w \leftarrow (s, \text{B} \rightarrow \bullet \delta)) \in Graph.Vertices$ do
                $ItemFollow(w) \leftarrow ItemFollow(w) \cup \text{First}(\gamma)$          28
                if ALLDERIVEEMPTY( $\gamma$ )                                     29
                then call $Graph$.ADDEDGE( $v, w$ )
end
procedure EVALITEMPROPGRAPH( )
    repeat                                                                30
        $changed \leftarrow$ false
        foreach $(v, w) \in Graph.Edges$ do
            $old \leftarrow ItemFollow(w)$
            $ItemFollow(w) \leftarrow ItemFollow(w) \cup ItemFollow(v)$
            if $ItemFollow(w) \neq old$
            then $changed \leftarrow$ true
    until not $changed$
end
```

Figure 6.26: LALR(1) version of COMPUTELOOKAHEAD.

```
procedure TRYRULEINSTATE( $s, r$ )
    if $\text{LHS}(r) \rightarrow \text{RHS}(r) \bullet \in s$
    then
        foreach $\mathcal{X} \in \Sigma$ do
            if $\mathcal{X} \in ItemFollow((s, \text{LHS}(r) \rightarrow \text{RHS}(r) \bullet))$
            then call ASSERTENTRY( $s, \mathcal{X}$, reduce $r$ )
end
```

Figure 6.27: LALR(1) version of TRYRULEINSTATE.

State	LR(0) Item		Goto State	Prop Edges Placed by Step		Initialize *ItemFollow*	
				27	**29**	First(γ)	**28**
0	1	Start→ •S $	4	13		$	2,3,4
	2	S→ •A B	2	8	5	b	5
	3	S→ •a c	3	11			
	4	S→ •x A c	1	6			
	5	A→ •a	3	12			
1	6	S→x•A c	9	18		c	7
	7	A→ •a	10	19			
2	8	S→A•B	8	17	9,10		
	9	B→ •b	7	16			
	10	B→ •					
3	11	S→a•c	6	15			
	12	A→a•					
4	13	Start→S•$	5	14			
5	14	Start→S $•					
6	15	S→a c•					
7	16	B→b•					
8	17	S→A B•					
9	18	S→x A•c	11	20			
10	19	A→a•					
11	20	S→x A c•					

Figure 6.28: LALR(1) analysis of the grammar in Figure 6.24.

ItemFollow sets are useful only when the bookmark progresses to the point of a reduction. B→ •δ is only the *promise* of a reduction to B once δ has been found. Thus, the lookahead symbols must accompany the bookmark's progress across δ so they are available for B→δ • in the appropriate state. Such migration of lookahead symbols is represented by propagation edges, as described next.

- Edges must be placed in the propagation graph when the symbols that are associated with one item should be added to the symbols associated with another item. Following are two situations that call for the addition of propagation edges.
  - As described above, the lookahead symbols introduced by Step **28** are useful only when the bookmark has advanced to the end of the rule. In LR(0), the CFSM contains an edge from state *s* to state *t* when the ad-

vance of the bookmark symbol for an item in state *s* creates an item in state *t*. For lookahead propagation in LALR, the edges are more specific—Step **27** places edges in the propagation graph between *items*. Specifically, an edge is placed from an item $\mathsf{A} \rightarrow \alpha \bullet \mathsf{B}\gamma$ in state *s* to the item $\mathsf{A} \rightarrow \alpha\mathsf{B} \bullet \gamma$ in state *t* obtained by advancing the bookmark, if *t* is the CFSM state reached by processing a B in state *s*.

– Consider again the item $\mathsf{A} \rightarrow \alpha \bullet \mathsf{B}\gamma$ and the closure items introduced when B is a nonterminal. When $\gamma \Rightarrow^\star \lambda$, either because γ is absent or because the string of symbols in γ can derive λ, then any symbol that can follow A can also follow B. Thus, Step **29** places a propagation edge from the item for A to the item for B.

The edges placed by these steps are used at Step **30** to affect the appropriate *ItemFollow* sets. The loop at Step **30** continues until no changes are observed in any *ItemFollow* set. This loop must eventually terminate, because the lookahead sets are only increased, by symbols drawn from a finite alphabet (Σ).

Now consider the grammar in Figure 6.24 and its LALR(1) construction shown in Figure 6.28. The items listed under the column for Step **27** are the targets of edges placed in the propagation graph to carry symbols to the point of reduction. For example, consider Items 6 and 7. For the item $\mathsf{S} \rightarrow \mathsf{x} \bullet \mathsf{A}\ \mathsf{c}$, we have $\gamma = \mathsf{c}$. Thus, when the item $\mathsf{A} \rightarrow \bullet\, \mathsf{a}$ is generated in Item 7, c can follow the reduction to A. Step **28** therefore adds c directly to Item 5's *ItemFollow* set. However, c is not useful until it is time to apply the reduction $\mathsf{A} \rightarrow \mathsf{a}$. Thus, propagation edges are placed between Items 7 and 19 by Step **27**.

In most cases, lookahead is either *generated* (when $\mathsf{First}(\gamma) \neq \emptyset$) or *propagated* (when $\gamma = \lambda$). However, it is possible that $\mathsf{First}(\gamma) \neq \emptyset$ and $\gamma \Rightarrow^\star \lambda$, as in Item 2. Here, $\gamma = \mathsf{B}$; we have $\mathsf{First}(\mathsf{B}) = \{\, b \,\}$ but we also have $\mathsf{B} \Rightarrow^\star \lambda$. Thus, Step **28** causes b to contribute to Item 5's *ItemFollow* set. Additionally, the *ItemFollow* set at Item 2 is forwarded to Item 5 by a propagation edge placed by Step **29**. Finally, the lookahead present at Item 2 must make its way to Item 17 where it can be consulted when the reduction $\mathsf{S} \rightarrow \mathsf{A}\ \mathsf{B}$ is applied. Thus, propagation edges are placed by Step **27** between Items 2 and 8 and between Items 8 and 17.

Constructing the propagation graph is only half of the process we need to compute lookahead sets. Once LALR(1) construction has established the propagation graph and has initialized the *ItemFollow* sets, as shown in Figure 6.28, the propagation graph can be evaluated. EVALITEMPROPGRAPH in Figure 6.26 evaluates the graph by iteratively propagating lookahead information along the graph's edges until no new information appears.

In Figure 6.29 we trace the progress of this algorithm on our example. The "Initial" column shows the lookahead sets established by Step **28**. The loop at Step **30** unions lookahead sets as determined by the propagation graph's edges. For the example we have considered thus far, the loop at Step **30** converges after a single pass. As written, the algorithm requires a second pass to detect that no

Item	Prop To	Initial	Pass 1
1	13	$	
2	5,8	$	
3	11	$	
4	6	$	
5	12	b	$
6	18		$
7	19	c	
8	9,10,17		$
9	16		$
10			$
11	15		$
12			b $
13	14		$
14			$
15			$
16			$
17			$
18	20		$
19			c
20			$

Figure 6.29: Iterations for LALR(1) follow sets.

lookahead sets change after the first pass. We do not show the second pass in Figure 6.29.

The loop at Step **30** continues until no *ItemFollow* set is changed from the previous iteration. The number of iterations prior to convergence depends on the structure of the propagation graph. The graph with edges specified in Figure 6.29 is *acyclic*—such graphs can be evaluated completely in a single pass.

In general, multiple passes can be required for convergence. We illustrate this using Figure 6.31, which records how an LALR(1) propagation graph is constructed for the grammar shown in Figure 6.30. Figure 6.32 shows the progress of the loop at Step **30**. The lookahead sets for a given item are the union of the symbols displayed in the three right-most columns. For this example, the sets converge after two passes, but a third pass is necessary to detect this. Two passes are necessary, because the propagation graph embeds the graph shown in Figure 6.33. This graph contains a cycle with one "retreating" backedge. Information cannot propagate from item 20 to 12 in a single pass. Exercise 27 explores how to extend the grammar in Figure 6.30 so that convergence can require *any number* of iterations. In practice, LALR(1) lookahead computations converge quickly, usually in one or two passes.

In summary, LALR(1) is a powerful parsing method and is the basis for most

1	Start	$\rightarrow$	S $
2	S	$\rightarrow$	x C1 y1 Cn yn
3		|	A1
4	A1	$\rightarrow$	b1 C1
5		|	a1
6	An	$\rightarrow$	bn Cn
7		|	an
8	C1	$\rightarrow$	An
9	Cn	$\rightarrow$	A1

Figure 6.30: LALR(1) analysis: grammar.

bottom-up parser generators. To achieve greater power, more lookahead can be applied, but this is rarely necessary. LALR(1) grammars are available for all popular programming languages.

6.6 LR(k) Table Construction

In this section we describe an LR table-construction method that accommodates all deterministic, context-free languages. While this may seem a boon, LR(k) parsing is not very practical, because even LR(1) tables are typically much larger than the LR(0) tables upon which SLR(k) and LALR(k) parsing are based. Moreover, it is rare that LR(1) can handle a grammar for which LALR(1) construction fails. We present such a grammar in Figure 6.34, but gramamrs such as these do not arise very often in practice. When LALR(1) fails, it is typically for one of the following reasons.

- The grammar is ambiguous—LR(k) cannot help.
- More lookahead is needed—LR(k) can help (for $k > 1$). However, LALR(k) would probably also work in such cases.
- No amount of lookahead suffices—LR(k) cannot help.

The grammar in Figure 6.34 allows strings generated by the nonterminal M to be surrounded by *matching* parentheses (lp and rp) or braces (lb and rb). The grammar also allows S to generate strings with *unmatched* punctuation. The unmatched expressions are generated by the nonterminal U. The grammar can easily be expanded by replacing the terminal expr with a nonterminal that derives arithmetic expressions, such as E in the grammar of Figure 6.20. While M and U generate the same terminal strings, the grammar distinguishes between them so that a semantic action can report the mismatched punctuation—using reduction by U$\rightarrow$expr.

A portion of the LALR(1) analysis of the grammar in Figure 6.34 is shown in Figure 6.36; the complete analysis is left for Exercise 28. Consider the lookaheads that will propagate into State 6. For Item 14, which calls for the reduction

State		LR(0) Item	Goto State	Prop Edges Placed by Step		Initialize *ItemFollow*	
				27	**29**	First(γ)	**28**
0	1	Start→ • S $	3	11		$	2,3
	2	S→ • x C1 y1 Cn yn	1	6			
	3	S→ • A1	5	16	4,5		
	4	A1→ • b1 C1	4	12			
	5	A1→ • a1	2	10			
1	6	S→x • C1 y1 Cn yn	13	27		y1	7
	7	C1→ • An	7	18	8,9		
	8	An→ • bn Cn	8	19			
	9	An→ • an	9	23			
2	10	A1→a1 •					
3	11	Start→S • $	12	26			
4	12	A1→b1 • C1	6	17	13		
	13	C1→ • An	7	18	14,15		
	14	An→ • bn Cn	8	19			
	15	An→ • an	9	23			
5	16	S→A1 •					
6	17	A1→b1 C1 •					
7	18	C1→An •					
8	19	An→bn • Cn	10	24	20		
	20	Cn→ • A1	11	25	21,22		
	21	A1→ • b1 C1	4	12			
	22	A1→ • a1	2	10			
9	23	An→an •					
10	24	An→bn Cn •					
11	25	Cn→A1 •					
12	26	Start→S $ •					
13	27	S→x C1 • y1 Cn yn	14	28			
14	28	S→x C1 y1 • Cn yn	15	32		yn	29
	29	Cn→ • A1	11	25	30,31		
	30	A1→ • b1 C1	4	12			
	31	A1→ • a1	2	10			
15	32	S→x C1 y1 Cn • yn	16	33			
16	33	S→x C1 y1 Cn yn •					

Figure 6.31: LALR(1) analysis.

Item	Prop To	Initial	Pass 1	Pass 2
1	11	\$		
2	6	\$		
3	4,5,16	\$		
4	12		\$	
5	10		\$	
6	27		\$	
7	8,9,18	y1		
8	19		y1	
9	23		y1	
10			\$ y1 yn	
11	26		\$	
12	13,17		\$ y1 yn	
13	14,15,18		\$	y1 yn
14	19		\$	y1 yn
15	23		\$	y1 yn
16			\$	
17			\$	y1 yn
18			y1 \$	yn
19	20,24		y1 \$	yn
20	21,22,25		y1 \$	yn
21	12		y1 \$	yn
22	10		y1 \$	yn
23			y1 \$	yn
24			y1 \$	yn
25			y1 \$ yn	
26			\$	
27	28		\$	
28	32		\$	
29	25,30,31	yn		
30	12		yn	
31	10		yn	
32	33		\$	
33			\$	

Figure 6.32: Iterations for LALR(1) follow sets.

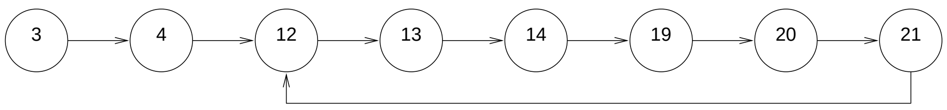


Figure 6.33: Embedded propagation subgraph.

1	Start	→	S $
2	S	→	lp M rp
3		|	lb M rb
4		|	lp U rb
5		|	lb U rp
6	M	→	expr
7	U	→	expr

Figure 6.34: A grammar that is not LALR(k).

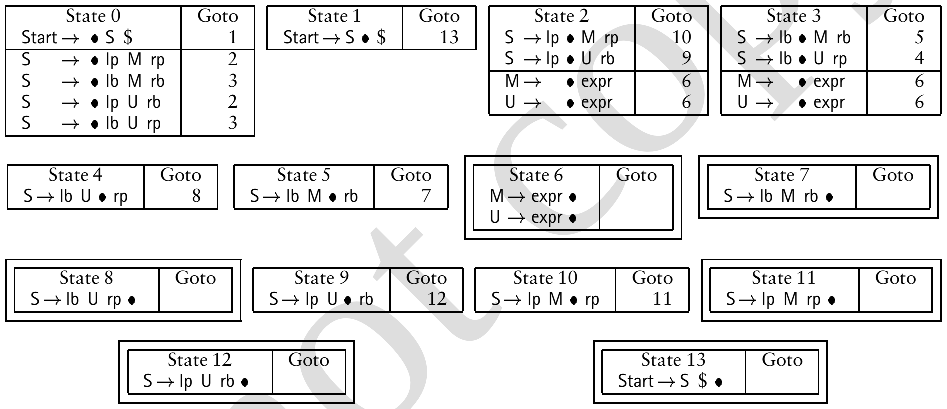


Figure 6.35: LR(0) construction.

M→expr, rp is sent to Item 8 and then to Item 14. Also, rb is sent to Item 12 and then to Item 14. Thus, *ItemFollow*(14) = { rb, rp }. Similarly, we compute *ItemFollow*(15) = { rb, rp }. Thus, State 6 contains a reduce/reduce conflict. For LALR(1), the rules M→expr and U→expr can each be followed by either rp or rb.

Because LALR(1) is based on LR(0), there is exactly one state with the kernel of State 6. Thus, States 2 and 3 must share State 6 when shifting an expr. If only we could split State 6, so that State 2 shifts to one version and State 3 shifts to the other, then the lookaheads in each state could resolve the conflict between M→expr and U→expr. The LR(1) construction causes such splitting, because a state is uniquely identified not only by its kernel from LR(0) but also its lookahead information.

SLR(k) and LALR(k) supply information to LR(0) states to help resolve conflicts. In LR(k), such information is part of the items themselves. For LR(k), we extend an item's notation from $A \to \alpha \bullet \beta$ to $[A \to \alpha \bullet \beta, w]$. For LR(1), w is a (terminal) symbol that can follow A when this item becomes reducible. For LR(k), $k \geq 0$, w is a k-length string that can follow A after reduction. If symbols x and y can both

State	LR(0) Item		Goto State	Prop Edges Placed by Step		Initialize *ItemFollow*	
				27	**29**	First(γ)	**28**
0	1	Start→ • S \$	1	??		\$	2,3,4,5
	2	S→ • lp M rp	2	6			
	3	S→ • lb M rb	3	10			
	4	S→ • lp U rb	2	7			
	5	S→ • lb U rp	3	11			
2	6	S→lp • M rp	10	??		rp	8
	7	S→lp • U rb	9	??		rb	9
	8	M→ • expr	6	14			
	9	U→ • expr	6	15			
3	10	S→lb • M rb	5	??		rb	12
	11	S→lb • U rp	4	??		rp	13
	12	M→ • expr	6	14			
	13	U→ • expr	6	15			
6	14	M→expr •					
	15	U→expr •					

Figure 6.36: Partial LALR(1) analysis.

follow A when A$\rightarrow\alpha \bullet \beta$ becomes reducible, then the corresponding LR(1) state contains both [A$\rightarrow\alpha \bullet \beta$, x] and [A$\rightarrow\alpha \bullet \beta$, y].

Notice how nicely the notation for LR(k) generalizes LR(0). For LR(0), w must be a 0-length string. The only such string is λ, which provides no information at a possible point of reduction, since λ does not occur as input.

Examples of LR(1) items for the grammar in Figure 6.34 include [S→lp • M rp , $] and [M→expr • , rp]. The first item is not ready for reduction, but indicates that $ will follow the reduction to S when the item becomes reducible in State 11. The second item calls for a reduction by rule M→expr when rp is the next input token. It is possible and likely that a given state contains several items that differ only in their follow symbol.

In LR(k), a state is a set of LR(k) items, and construction of the CFSM is basically the same as with LR(0). States are represented by their kernel items, and new states are generated as needed. Figure 6.37 presents an LR(1) construction algorithm in terms of modifications to the LR(0) algorithm shown in Figures 6.8 and 6.9. At Step **31**, any symbol that can follow B due to the presence of γ is considered; when $\gamma \Rightarrow^{\star} \lambda$, then any symbol a that can follow A can also follow B. Thus, Step **31** considers each symbol in First(γa). The current state receives an item for reach rule for B and each possible follow symbol. Figure 6.13 shows TRYRULEINSTATE—the LR(0) method for determining if a state calls for a particular

The following steps must be modified in Figures 6.8 and 6.9.

Step 7: We initialize *StartItems* by including LR(1) items that have \$ as the follow symbol:

$StartItems \leftarrow \{ [\,\mathsf{Start} \rightarrow \bullet\, \mathrm{RHS}(p)\,, \$\,] \mid p \in \textsc{ProductionsFor}(\mathsf{Start}) \}$

Step 13: We augment the LR(0) item so that AdvanceDot returns the appropriate LR(1) items:

return $(\{ [\,\mathsf{A} \rightarrow \alpha \mathcal{X} \bullet \beta\,, \mathsf{a}\,] \mid [\,\mathsf{A} \rightarrow \alpha \bullet \mathcal{X}\beta\,, \mathsf{a}\,] \in state \})$

Step 15: This entire loop is replaced by the following:

```
foreach $[\,\mathsf{A} \rightarrow \alpha \bullet \mathsf{B}\gamma\,, \mathsf{a}\,] \in ans$ do
    foreach $p \in \textsc{ProductionsFor}(B)$ do
        foreach $b \in \mathsf{First}(\gamma \mathsf{a})$ do                    31
            $ans \leftarrow ans \cup \{ [\,\mathsf{B} \rightarrow \bullet\, \mathrm{RHS}(p)\,, b\,] \}$
```

Figure 6.37: LR(1) construction.

```
procedure TryRuleInState($s, r$)
    if $[\,\mathrm{LHS}(r) \rightarrow \mathrm{RHS}(r) \bullet\,, w\,] \in s$
    then call AssertEntry($s, w$, reduce r)
end
```

Figure 6.38: LR(1) version of TryRuleInState.

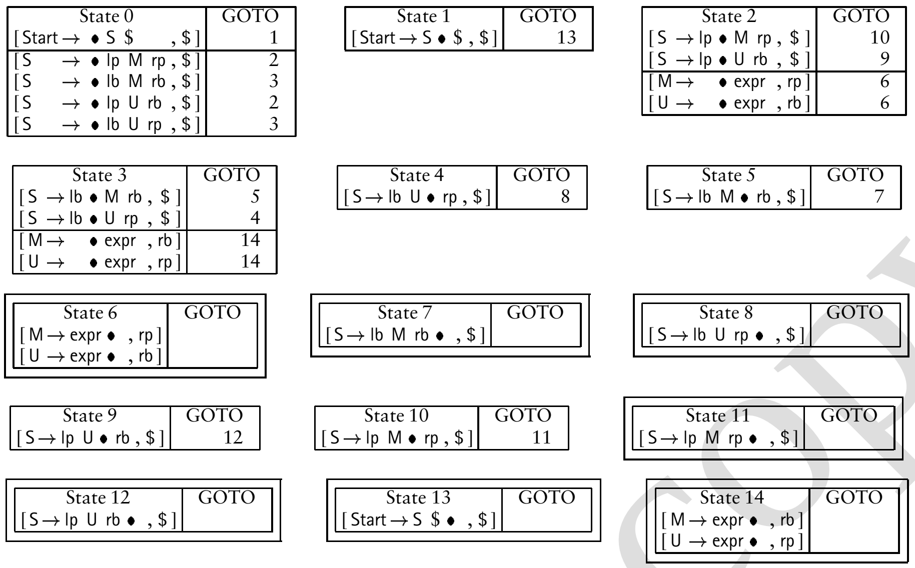
reduction. The LR(1) version of TryRuleInState is shown in Figure 6.38.

Figure 6.39 shows the LR(1) construction for the grammar in Figure 6.34. States 6 and 14 would be merged under LR(0). For LR(1), these states differ by the lookaheads associated with the reducible items. Thus, LR(1) is able to resolve what would have been a reduce/reduce conflict under LR(0).

The number of states such as States 6 and 14 that split during LR(1) construction is usually much larger. Instead of constructing a full LR(1) parse table, one could begin with LALR(1), which is based on the LR(0) construction. States could then be split selectively. As discussed in Exercise 34, LR(k) can resolve only the reduce/reduce conflicts that arise during LALR(k) construction. A shift/reduce conflict in LALR(k) will also be present in the correponding LR(k) construction. Exercise 35 considers how to split LR(0) states on-demand, in response to reduce/reduce conflicts that arise in LALR(k) constructions.

Summary

This concludes our study of LR parsers. We have investigated a number of LR table-building methods, from LR(0) to LR(1). The intermediate methods—SLR(1) and LALR(1)—are the most practical. In particular, LALR(1) provides excellent conflict



State 0	GOTO
[Start → • S $, $]	1
[S → • lp M rp , $]	2
[S → • lb M rb , $]	3
[S → • lp U rb , $]	2
[S → • lb U rp , $]	3

State 1	GOTO
[Start → S • $, $]	13

State 2	GOTO
[S → lp • M rp , $]	10
[S → lp • U rb , $]	9
[M → • expr , rp]	6
[U → • expr , rb]	6

State 3	GOTO
[S → lb • M rb , $]	5
[S → lb • U rp , $]	4
[M → • expr , rb]	14
[U → • expr , rp]	14

State 4	GOTO
[S → lb U • rp , $]	8

State 5	GOTO
[S → lb M • rb , $]	7

State 6	GOTO
[M → expr • , rp]	
[U → expr • , rb]	

State 7	GOTO
[S → lb M rb • , $]	

State 8	GOTO
[S → lb U rp • , $]	

State 9	GOTO
[S → lp U • rb , $]	12

State 10	GOTO
[S → lp M • rp , $]	11

State 11	GOTO
[S → lp M rp • , $]	

State 12	GOTO
[S → lp U rb • , $]	

State 13	GOTO
[Start → S $ • , $]	

State 14	GOTO
[M → expr • , rb]	
[U → expr • , rp]	

Figure 6.39: LR(1) construction.

resolution and generates very compact tables. Tools based on LALR(1) grammars are discussed in Chapter Chapter:global:seven. Such tools are indispensible for language modification and extension. Changes can be prototyped using an LALR(1) grammar for the language's syntax. When conflicts occur, the methods discussed in Section 6.4 help identify why the proposed modification may not work. Because of their efficiency and power, LALR(1) grammars are available for most modern programming languages. Indeed, the syntax of modern programming languages is commonly designed with LALR(1) parsing in mind.

Exercises

1. Build the CFSM and the parse table for the grammar shown in Figure 6.1.
2. Using the knitting analogy of Section 6.2.2, show the sequence of LR shift and reduce actions for the grammar of Figure 6.1 on the following strings.
   (a) plus plus num num num $
   (b) plus num plus num num $
3. Figure 6.6 traces a bottom-up parse of an input string using the table shown in Figure 6.5. Trace the parse of the following strings.
   (a) q $
   (b) c $
   (c) a d c $
4. Build the CFSM for the following grammar.

1	Prog	→	Block $
2	Block	→	begin StmtList end
3	StmtList	→	StmtList semi Stmt
4		|	Stmt
5	Stmt	→	Block
6		|	Var assign Expr
7	Var	→	id
8		|	id lb Expr rb
9	Expr	→	Expr plus T
10		|	T
11	T	→	Var
12		|	lp Expr rp

5. Show the LR parse table for the CFSM constructed in Exercise 4.
6. Which of following grammars are LR(0)? Explain why.

(a)

1	S	→	StmtList $
2	StmtList	→	StmtList semi Stmt
3		|	Stmt
4	Stmt	→	s

(b)

1	S	→	StmtList $
2	StmtList	→	Stmt semi StmtList
3		|	Stmt
4	Stmt	→	s

(c)

1	S	$\rightarrow$	StmtList \$
2	StmtList	$\rightarrow$	StmtList semi StmtList
3		|	Stmt
4	Stmt	$\rightarrow$	s

(d)

1	S	$\rightarrow$	StmtList \$
2	StmtList	$\rightarrow$	s StTail
3	StTail	$\rightarrow$	semi StTail
4		|	λ

7. Show that the CFSM corresponding to a LL(1) grammar has the following property. Each state has exactly one kernel item if the grammar is λ-free.

8. Prove or disprove that all λ-free LL(1) grammars are LR(0).

9. Explain why the language defined by the following grammar is *inherently nondeterministic*—there is no LALR(k) grammar for this language.

1	Start	$\rightarrow$	Single a
2		|	Double b
3	Single	$\rightarrow$	0 Single 1
4		|	0 1
5	Double	$\rightarrow$	0 Double 1 1
6		|	0 1 1

10. Given the claim of Exercise 9, explain why the following statement is true or false. There is no LR(k) grammar for the language

$$\{ 0^n 1^n \mathsf{a} \} \cup \{ 0^n 1^{2n} \mathsf{b} \}.$$

11. Discuss why is it not possible during LR(0) construction to obtain a shift/reduce conflict on a nonterminal.

12. Discuss why there cannot be an unambiguous CFG for the language

$$\{ \mathsf{a}^i \mathsf{b}^j \mathsf{c}^k \mid i = j \text{ or } j = k; i, j, k \geq 1 \}.$$

13. Complete the LR(0) construction for the grammar in Figure 6.18.

14. Show that LL(1) construction fails for an unambiguous grammar that is not LR(1).

15. Show that the grammar in Figure 6.19 is LR(0).

16. Complete the LR(0) construction for the grammar shown in Figure 6.24. Your state numbers should agree with those shown in the partial LR(0) construction.

17. Show the LR(0) construction for the following grammars.

	1	Start	→	S $
	2	S	→	id assign E semi
	3	E	→	E plus P
(a)	4		|	P
	5	P	→	id
	6		|	lp E rp
	7		|	id assign E

	1	Start	→	S $
	2	S	→	id assign A semi
	3	A	→	id assign A
	4		|	E
(b)	5	E	→	E plus P
	6		|	P
	7	P	→	id
	8		|	lp A rp

	1	Start	→	S $
	2	S	→	id assign A semi
	3	A	→	id assign A
	4		|	E
(c)	5	E	→	E plus P
	6		|	P
	7		|	P plus
	8	P	→	id
	9		|	lp A rp

	1	Start	→	S $
	2	S	→	id assign A semi
	3	A	→	Pre E
	4	Pre	→	Pre id assign
(d)	5		|	λ
	6	E	→	E plus P
	7		|	P
	8	P	→	id
	9		|	lp A rp

	1	Start	$\rightarrow$	S $
	2	S	$\rightarrow$	id assign A semi
	3	A	$\rightarrow$	Pre E
	4	Pre	$\rightarrow$	id assign Pre
(e)	5		|	λ
	6	E	$\rightarrow$	E plus P
	7		|	P
	8	A	$\rightarrow$	id
	9		|	lp A rp

	1	Start	$\rightarrow$	S $
	2	S	$\rightarrow$	id assign A semi
	3	A	$\rightarrow$	id assign A
	4		|	E
	5	E	$\rightarrow$	E plus P
	6		|	P
(f)	7	P	$\rightarrow$	id
	8		|	lp A semi A rp
	9		|	lp V comma V rp
	10		|	lb A comma A rb
	11		|	lb V semi V rb
	12	V	$\rightarrow$	id

	1	Start	$\rightarrow$	S $
	2	S	$\rightarrow$	id assign A semi
	3	A	$\rightarrow$	id assign A
	4		|	E
(g)	5	E	$\rightarrow$	E plus P
	6		|	P
	7	P	$\rightarrow$	id
	8		|	lp id semi id rp
	9		|	lp A rp

18. Which of the grammars in Exercise 17 are LR(0)? Justify your answers.

19. Complete the SLR(1) construction for the grammar shown in Figure 6.25. Show the resulting parse table.

20. Extend the grammar given in Figure 6.20 to accommodate standard expressions involving addition, subtraction, multiplication, and division. Model the syntax and semantics for these operators according to Java or C.

21. Extend the grammar as directed in Exercise 20, but introduce an exponentiation operator that is *right-associating*. Let the operator (denoted by "$\star$") have the highest priority, so that the value of $3 + 4 \times 5 \star 2$ is 103.

22. Repeat Exercise 21 but add the ability to enclose expressions with parentheses, to control how expressions are grouped together. Thus, the value of $((3+4)\times 5)\star 2$ is 1225.

23. Which of the grammars in Exercise 17 are SLR(1)? Justify your answers.

24. Generalize the algorithm given in Section 6.5.1 from SLR(1) to SLR(k).

25. Show that

    (a) For any state s containing the item $\mathsf{A}\rightarrow\alpha\bullet\beta$, $\mathit{ItemFollow}((s,\mathsf{A}\rightarrow\alpha\bullet\beta))\subseteq$ Follow(A)

    (b)

$$\bigcup_{s}\ \bigcup_{\mathsf{A}\rightarrow\alpha_i\bullet\beta_i\in s}\mathit{ItemFollow}((s,\mathsf{A}\rightarrow\alpha_i\bullet\beta_i)) = \mathsf{Follow}(\mathsf{A})$$

26. Perform the LALR(1) construction for the following grammar:

1	Start	→	S $
2	S	→	x C1 y1 C2 y2 C3 y3
3		|	A1
4	A1	→	b1 C1
5		|	a1
6	A2	→	b2 C2
7		|	a2
8	A3	→	b3 C3
9		|	a3
10	C1	→	A2
11	C2	→	A1
12		|	A3
13	C3	→	A2

27. Using the grammars in Figure 6.30 and Exercise 26 as a guide, show how to generate a grammar that requires n iterations for LALR(1) convergence.

28. For the grammar shown in Figure 6.34, complete the LALR(1) construction from Figure 6.36.

29. Which of the grammars in Exercise 17 are LALR(1)? Justify your answers.

30. Show the LR(1) construction for the grammar in Exercise 4.

31. Define the **quasi-identical states** of an LR(1) parsing machine to be those states whose kernel productions are identical. Such states are distinguished only by the lookahead symbols associated with their productions. Given the LR(1) machine built for Exercise 30, do the following.

(a) List the quasi-identical states of the LR(1) machine.

(b) Merge each set of quasi-identical states to obtain an LALR(1) machine.

32. Starting with the CFSM built in Exercise 4, compute the LALR(1) lookahead information. Compare the resulting LALR(1) machine with the machine obtained in Exercise 31

33. Which of the grammars in Exercise 17 are LR(1)? Justify your answers.

34. Consider a grammar G and its LALR(1) construction. Suppose that a shift/reduce conflict occurs in G's LALR(1) construction. Prove that G's LR(1) construction also contains a shift/reduce conflict.

35. Describe an algorithm that computes LALR(1) and then splits states as needed in an attempt to address conflicts. Take note of the issue raised in Exercise 34.

36. Using a grammar for the C programming language, try to extend the syntax to allow *nested* function definitions. For example, you might allow function definitions to occur inside any compound statement.

    Report on any difficulties you encounter, and discuss possible solutions. Justify the particular solution you adopt.

37. Using a grammar for the C programming language, try to extend the syntax so that a compound statement can appear to compute a value. In other words, allow a compound statement to appear wherever a simple constant or identifier could appear. Semantically, the value of a compound statement could be the value associated with its last statement.

    Report on any difficulties you encounter, and discuss possible solutions. Justify the particular solution you adopt.

38. In Figure 6.3, Step **2** pushes a state on the parse stack. In the bottom-up parse shown in Figure 6.6, stack cells show both the state and the input symbol causing the state's shift onto the stack. Explain why the input symbol's presence in the stack cell is superfluous.

39. Recall the **dangling else** problem introduced in Chapter Chapter:global:five. Following is a grammar for a simplified language that allows conditional statements.

1	Start	→	Stmt \$
2	Stmt	→	if e then Stmt else Stmt
3		|	if e then Stmt
4		|	other

    Explain why the following grammar is or is not LALR(1).

40. Consider the following grammar.

```
1   Start      →   Stmt $
2   Stmt       →   Matched
3              |   Unmatched
4   Matched    →   if e then Matched else Matched
5              |   other
6   Unmatched  →   if e then Matched else Unmatched
7              |   if e then Unmatched
```

(a) Explain why the grammar is or is not LALR(1).

(b) Is the language of this grammar the same as the language of the grammar in Exercise 39? Why or why not?

41. Repeat Exercise 40, adding the production Unmatched→other to the grammar.

42. Consider the following grammar.

```
1   Start      →   Stmt $
2   Stmt       →   Matched
3              |   Unmatched
4   Matched    →   if e then Matched else Matched
5              |   other
6   Unmatched  →   if e then Matched else Unmatched
7              |   if e then Stmt
```

(a) Explain why the grammar is or is not LALR(1).

(b) Is the language of this grammar the same as the language of the grammar in Exercise 39? Why or why not?

43. Based on the material in Exercises 39, 40, and 42, construct an LALR(1) grammar for the language defined by the following grammar.

```
1   Start  →   Stmt $
2   Stmt   →   if e then Stmt else Stmt
3          |   if e then Stmt
4          |   while e Stmt
5          |   repeat Stmt until e
6          |   other
```

44. Show that there exist non-LL(1) grammars that are

(a) LR(0)

(b) SLR(1)

(c) LALR(1)

45. Normally, an LR parser traces a rightmost derivation (in reverse).

(a) How could an LR parser be modified to produce a leftmost parse as LL(1) parsers do? Describe your answer in terms of the algorithm in Figure 6.3.

(b) Would it help if we knew that the LR table was constructed for an LL grammar?

46. For each of the following, construct an appropriate grammar.

    (a) The grammar is SLR(3) but not SLR(2).

    (b) The grammar is LALR(2) but not LALR(1).

    (c) The grammar is LR(2) but not LR(1).

    (d) The grammar is LALR(1) and SLR(2) but not SLR(1).

47. Construct a single grammar that has *all* of the following properties.

    - It is SLR(3) but not SLR(2).
    - It is LALR(2) but not LALR(1).
    - It is LR(1).

48. For every $k > 1$ show that there exist grammars that are SLR($k+1$), LALR($k+1$), and LR($k+1$) but not SLR(k), LALR(k), or LR(k).

49. Consider the grammar generated by $1 \leq i, j \leq n,\ i \neq j$ using the following template.

$$\begin{array}{lcl} \mathsf{S} & \rightarrow & \mathsf{X}_i\ \mathsf{z}_i \\ \mathsf{X}_i & \rightarrow & \mathsf{y}_j\ \mathsf{X}_i \\ & | & \mathsf{y}_j \end{array}$$

The resulting grammar has $O(n^2)$ productions.

(a) Show that the CFSM for this grammar has $O(2^n)$ states.

(b) Is the grammar SLR(1)?