a) $SSS+a*+$.

b) $SS+a*a+$.

c) $aaa*a++$.

Exercise 4.5.3: Give bottom-up parses for the following input strings and grammars:

a) The input 000111 according to the grammar of Exercise 4.5.1.

b) The input $aaa*a++$ according to the grammar of Exercise 4.5.2.

4.6 Introduction to LR Parsing: Simple LR

The most prevalent type of bottom-up parser today is based on a concept called LR(k) parsing; the "L" is for left-to-right scanning of the input, the "R" for constructing a rightmost derivation in reverse, and the k for the number of input symbols of lookahead that are used in making parsing decisions. The cases $k = 0$ or $k = 1$ are of practical interest, and we shall only consider LR parsers with $k \leq 1$ here. When (k) is omitted, k is assumed to be 1.

This section introduces the basic concepts of LR parsing and the easiest method for constructing shift-reduce parsers, called "simple LR" (or SLR, for short). Some familiarity with the basic concepts is helpful even if the LR parser itself is constructed using an automatic parser generator. We begin with "items" and "parser states;" the diagnostic output from an LR parser generator typically includes parser states, which can be used to isolate the sources of parsing conflicts.

Section 4.7 introduces two, more complex methods — canonical-LR and LALR — that are used in the majority of LR parsers.

4.6.1 Why LR Parsers?

LR parsers are table-driven, much like the nonrecursive LL parsers of Section 4.4.4. A grammar for which we can construct a parsing table using one of the methods in this section and the next is said to be an *LR grammar*. Intuitively, for a grammar to be LR it is sufficient that a left-to-right shift-reduce parser be able to recognize handles of right-sentential forms when they appear on top of the stack.

LR parsing is attractive for a variety of reasons:

- LR parsers can be constructed to recognize virtually all programming-language constructs for which context-free grammars can be written. Non-LR context-free grammars exist, but these can generally be avoided for typical programming-language constructs.

- The LR-parsing method is the most general nonbacktracking shift-reduce parsing method known, yet it can be implemented as efficiently as other, more primitive shift-reduce methods (see the bibliographic notes).
- An LR parser can detect a syntactic error as soon as it is possible to do so on a left-to-right scan of the input.
- The class of grammars that can be parsed using LR methods is a proper superset of the class of grammars that can be parsed with predictive or LL methods. For a grammar to be LR(k), we must be able to recognize the occurrence of the right side of a production in a right-sentential form, with k input symbols of lookahead. This requirement is far less stringent than that for LL(k) grammars where we must be able to recognize the use of a production seeing only the first k symbols of what its right side derives. Thus, it should not be surprising that LR grammars can describe more languages than LL grammars.

The principal drawback of the LR method is that it is too much work to construct an LR parser by hand for a typical programming-language grammar. A specialized tool, an LR parser generator, is needed. Fortunately, many such generators are available, and we shall discuss one of the most commonly used ones, `Yacc`, in Section 4.9. Such a generator takes a context-free grammar and automatically produces a parser for that grammar. If the grammar contains ambiguities or other constructs that are difficult to parse in a left-to-right scan of the input, then the parser generator locates these constructs and provides detailed diagnostic messages.

4.6.2 Items and the LR(0) Automaton

How does a shift-reduce parser know when to shift and when to reduce? For example, with stack contents $\$T$ and next input symbol $*$ in Fig. 4.28, how does the parser know that T on the top of the stack is not a handle, so the appropriate action is to shift and not to reduce T to E?

An LR parser makes shift-reduce decisions by maintaining states to keep track of where we are in a parse. States represent sets of "items." An *LR(0) item* (*item* for short) of a grammar G is a production of G with a dot at some position of the body. Thus, production $A \rightarrow XYZ$ yields the four items

$$
\begin{aligned}
A &\rightarrow \cdot XYZ \\
A &\rightarrow X \cdot YZ \\
A &\rightarrow XY \cdot Z \\
A &\rightarrow XYZ \cdot
\end{aligned}
$$

The production $A \rightarrow \epsilon$ generates only one item, $A \rightarrow \cdot$.

Intuitively, an item indicates how much of a production we have seen at a given point in the parsing process. For example, the item $A \rightarrow \cdot XYZ$ indicates that we hope to see a string derivable from XYZ next on the input. Item

Representing Item Sets

A parser generator that produces a bottom-up parser may need to represent items and sets of items conveniently. Note that an item can be represented by a pair of integers, the first of which is the number of one of the productions of the underlying grammar, and the second of which is the position of the dot. Sets of items can be represented by a list of these pairs. However, as we shall see, the necessary sets of items often include "closure" items, where the dot is at the beginning of the body. These can always be reconstructed from the other items in the set, and we do not have to include them in the list.

$A \to X \cdot YZ$ indicates that we have just seen on the input a string derivable from X and that we hope next to see a string derivable from YZ. Item $A \to XYZ \cdot$ indicates that we have seen the body XYZ and that it may be time to reduce XYZ to A.

One collection of sets of LR(0) items, called the *canonical* LR(0) collection, provides the basis for constructing a deterministic finite automaton that is used to make parsing decisions. Such an automaton is called an *LR(0) automaton*.[3] In particular, each state of the LR(0) automaton represents a set of items in the canonical LR(0) collection. The automaton for the expression grammar (4.1), shown in Fig. 4.31, will serve as the running example for discussing the canonical LR(0) collection for a grammar.

To construct the canonical LR(0) collection for a grammar, we define an augmented grammar and two functions, CLOSURE and GOTO. If G is a grammar with start symbol S, then G', the *augmented grammar* for G, is G with a new start symbol S' and production $S' \to S$. The purpose of this new starting production is to indicate to the parser when it should stop parsing and announce acceptance of the input. That is, acceptance occurs when and only when the parser is about to reduce by $S' \to S$.

Closure of Item Sets

If I is a set of items for a grammar G, then CLOSURE(I) is the set of items constructed from I by the two rules:

1. Initially, add every item in I to CLOSURE(I).
2. If $A \to \alpha \cdot B\beta$ is in CLOSURE(I) and $B \to \gamma$ is a production, then add the item $B \to \cdot\gamma$ to CLOSURE(I), if it is not already there. Apply this rule until no more new items can be added to CLOSURE(I).

[3]Technically, the automaton misses being deterministic according to the definition of Section 3.6.4, because we do not have a dead state, corresponding to the empty set of items. As a result, there are some state-input pairs for which no next state exists.

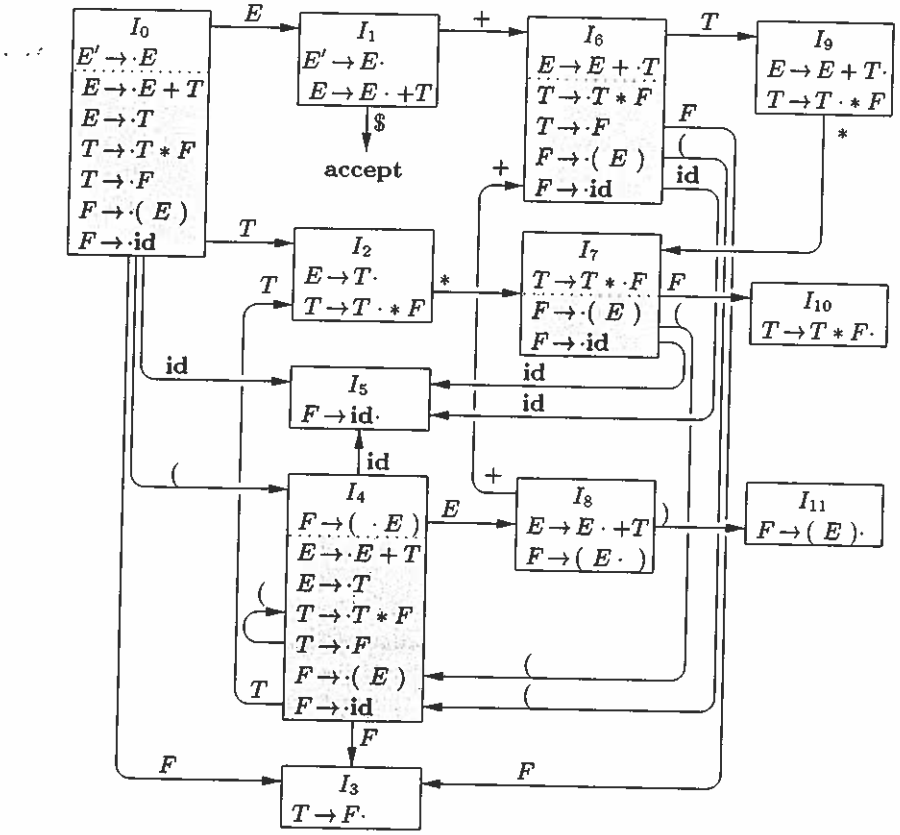

Figure 4.31: LR(0) automaton for the expression grammar (4.1)

Intuitively, $A \to \alpha \cdot B\beta$ in CLOSURE(I) indicates that, at some point in the parsing process, we think we might next see a substring derivable from $B\beta$ as input. The substring derivable from $B\beta$ will have a prefix derivable from B by applying one of the B-productions. We therefore add items for all the B-productions; that is, if $B \to \gamma$ is a production, we also include $B \to \cdot\gamma$ in CLOSURE(I).

Example 4.40: Consider the augmented expression grammar:

$$\begin{array}{lcl} E' & \to & E \\ E & \to & E + T \mid T \\ T & \to & T * F \mid F \\ E & \to & (E) \mid \mathbf{id} \end{array}$$

If I is the set of one item $\{[E' \to \cdot E]\}$, then CLOSURE(I) contains the set of items I_0 in Fig. 4.31.

To see how the closure is computed, $E' \to \cdot E$ is put in CLOSURE(I) by rule (1). Since there is an E immediately to the right of a dot, we add the E-productions with dots at the left ends: $E \to \cdot E + T$ and $E \to \cdot T$. Now there is a T immediately to the right of a dot in the latter item, so we add $T \to \cdot T * F$ and $T \to \cdot F$. Next, the F to the right of a dot forces us to add $F \to \cdot(E)$ and $F \to \cdot$**id**, but no other items need to be added. □

The closure can be computed as in Fig. 4.32. A convenient way to implement the function *closure* is to keep a boolean array *added*, indexed by the nonterminals of G, such that *added*[B] is set to **true** if and when we add the item $B \to \cdot\gamma$ for each B-production $B \to \gamma$.

```
SetOfItems CLOSURE(I) {
    J = I;
    repeat
        for ( each item A → α·Bβ in J )
            for ( each production B → γ of G )
                if ( B → ·γ is not in J )
                    add B → ·γ to J;
    until no more items are added to J on one round;
    return J;
}
```

Figure 4.32: Computation of CLOSURE

Note that if one B-production is added to the closure of I with the dot at the left end, then all B-productions will be similarly added to the closure. Hence, it is not necessary in some circumstances actually to list the items $B \to \cdot\gamma$ added to I by CLOSURE. A list of the nonterminals B whose productions were so added will suffice. We divide all the sets of items of interest into two classes:

1. *Kernel items*: the initial item, $S' \to \cdot S$, and all items whose dots are not at the left end.

2. *Nonkernel items*: all items with their dots at the left end, except for $S' \to \cdot S$.

Moreover, each set of items of interest is formed by taking the closure of a set of kernel items; the items added in the closure can never be kernel items, of course. Thus, we can represent the sets of items we are really interested in with very little storage if we throw away all nonkernel items, knowing that they could be regenerated by the closure process. In Fig. 4.31, nonkernel items are in the shaded part of the box for a state.

The Function GOTO

The second useful function is GOTO(I, X) where I is a set of items and X is a grammar symbol. GOTO(I, X) is defined to be the closure of the set of all items $[A \to \alpha X \cdot \beta]$ such that $[A \to \alpha \cdot X\beta]$ is in I. Intuitively, the GOTO function is used to define the transitions in the LR(0) automaton for a grammar. The states of the automaton correspond to sets of items, and GOTO(I, X) specifies the transition from the state for I under input X.

Example 4.41: If I is the set of two items $\{[E' \to E\cdot], [E \to E\cdot + T]\}$, then GOTO$(I, +)$ contains the items

$$
\begin{aligned}
E &\to E + \cdot T \\
T &\to \cdot T * F \\
T &\to \cdot F \\
F &\to \cdot(E) \\
F &\to \cdot \mathbf{id}
\end{aligned}
$$

We computed GOTO$(I, +)$ by examining I for items with $+$ immediately to the right of the dot. $E' \to E\cdot$ is not such an item, but $E \to E\cdot + T$ is. We moved the dot over the $+$ to get $E \to E + \cdot T$ and then took the closure of this singleton set. □

We are now ready for the algorithm to construct C, the canonical collection of sets of LR(0) items for an augmented grammar G' — the algorithm is shown in Fig. 4.33.

```
void items(G') {
    C = CLOSURE({[S' → ·S]});
    repeat
        for ( each set of items I in C )
            for ( each grammar symbol X )
                if ( GOTO(I, X) is not empty and not in C )
                    add GOTO(I, X) to C;
    until no new sets of items are added to C on a round;
}
```

Figure 4.33: Computation of the canonical collection of sets of LR(0) items

Example 4.42: The canonical collection of sets of LR(0) items for grammar (4.1) and the GOTO function are shown in Fig. 4.31. GOTO is encoded by the transitions in the figure. □

Use of the LR(0) Automaton

The central idea behind "Simple LR," or SLR, parsing is the construction from the grammar of the LR(0) automaton. The states of this automaton are the sets of items from the canonical LR(0) collection, and the transitions are given by the GOTO function. The LR(0) automaton for the expression grammar (4.1) appeared earlier in Fig. 4.31.

The start state of the LR(0) automaton is CLOSURE($\{[S' \to \cdot S]\}$), where S' is the start symbol of the augmented grammar. All states are accepting states. We say "state j" to refer to the state corresponding to the set of items I_j.

How can LR(0) automata help with shift-reduce decisions? Shift-reduce decisions can be made as follows. Suppose that the string γ of grammar symbols takes the LR(0) automaton from the start state 0 to some state j. Then, shift on next input symbol a if state j has a transition on a. Otherwise, we choose to reduce; the items in state j will tell us which production to use.

The LR-parsing algorithm to be introduced in Section 4.6.3 uses its stack to keep track of states as well as grammar symbols; in fact, the grammar symbol can be recovered from the state, so the stack holds states. The next example gives a preview of how an LR(0) automaton and a stack of states can be used to make shift-reduce parsing decisions.

Example 4.43: Figure 4.34 illustrates the actions of a shift-reduce parser on input **id** $*$ **id**, using the LR(0) automaton in Fig. 4.31. We use a stack to hold states; for clarity, the grammar symbols corresponding to the states on the stack appear in column SYMBOLS. At line (1), the stack holds the start state 0 of the automaton; the corresponding symbol is the bottom-of-stack marker \$.

LINE	STACK	SYMBOLS	INPUT	ACTION
(1)	0	\$	**id** $*$ **id** \$	shift to 5
(2)	0 5	\$ **id**	$*$ **id** \$	reduce by $F \to$ **id**
(3)	0 3	\$ F	$*$ **id** \$	reduce by $T \to F$
(4)	0 2	\$ T	$*$ **id** \$	shift to 7
(5)	0 2 7	\$ $T*$	**id** \$	shift to 5
(6)	0 2 7 5	\$ $T*$ **id**	\$	reduce by $F \to$ **id**
(7)	0 2 7 10	\$ $T*F$	\$	reduce by $T \to T*F$
(8)	0 2	\$ T	\$	reduce by $E \to T$
(9)	0 1	\$ E	\$	accept

Figure 4.34: The parse of **id** $*$ **id**

The next input symbol is **id** and state 0 has a transition on **id** to state 5. We therefore shift. At line (2), state 5 (symbol **id**) has been pushed onto the stack. There is no transition from state 5 on input $*$, so we reduce. From item $[F \to \textbf{id}\cdot]$ in state 5, the reduction is by production $F \to$ **id**.

With symbols, a reduction is implemented by popping the body of the production from the stack (on line (2), the body is **id**) and pushing the head of the production (in this case, F). With states, we pop state 5 for symbol **id**, which brings state 0 to the top and look for a transition on F, the head of the production. In Fig. 4.31, state 0 has a transition on F to state 3, so we push state 3, with corresponding symbol F; see line (3).

As another example, consider line (5), with state 7 (symbol $*$) on top of the stack. This state has a transition to state 5 on input **id**, so we push state 5 (symbol **id**). State 5 has no transitions, so we reduce by $F \rightarrow$ **id**. When we pop state 5 for the body **id**, state 7 comes to the top of the stack. Since state 7 has a transition on F to state 10, we push state 10 (symbol F). □

4.6.3 The LR-Parsing Algorithm

A schematic of an LR parser is shown in Fig. 4.35. It consists of an input, an output, a stack, a driver program, and a parsing table that has two parts (ACTION and GOTO). The driver program is the same for all LR parsers; only the parsing table changes from one parser to another. The parsing program reads characters from an input buffer one at a time. Where a shift-reduce parser would shift a symbol, an LR parser shifts a *state*. Each state summarizes the information contained in the stack below it.

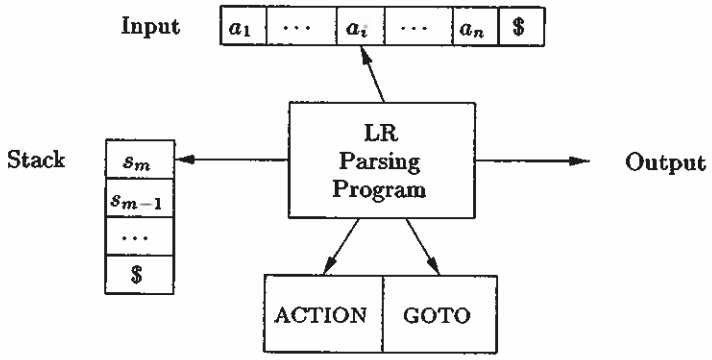

Figure 4.35: Model of an LR parser

The stack holds a sequence of states, $s_0 s_1 \cdots s_m$, where s_m is on top. In the SLR method, the stack holds states from the LR(0) automaton; the canonical-LR and LALR methods are similar. By construction, each state has a corresponding grammar symbol. Recall that states correspond to sets of items, and that there is a transition from state i to state j if GOTO$(I_i, X) = I_j$. All transitions to state j must be for the same grammar symbol X. Thus, each state, except the start state 0, has a unique grammar symbol associated with it.[4]

[4]The converse need not hold; that is, more than one state may have the same grammar

Structure of the LR Parsing Table

The parsing table consists of two parts: a parsing-action function ACTION and a goto function GOTO.

1. The ACTION function takes as arguments a state i and a terminal a (or \$, the input endmarker). The value of ACTION$[i, a]$ can have one of four forms:
   (a) Shift j, where j is a state. The action taken by the parser effectively shifts input a to the stack, but uses state j to represent a.
   (b) Reduce $A \rightarrow \beta$. The action of the parser effectively reduces β on the top of the stack to head A.
   (c) Accept. The parser accepts the input and finishes parsing.
   (d) Error. The parser discovers an error in its input and takes some corrective action. We shall have more to say about how such error-recovery routines work in Sections 4.8.3 and 4.9.4.
2. We extend the GOTO function, defined on sets of items, to states: if GOTO$[I_i, A] = I_j$, then GOTO also maps a state i and a nonterminal A to state j.

LR-Parser Configurations

To describe the behavior of an LR parser, it helps to have a notation representing the complete state of the parser: its stack and the remaining input. A *configuration* of an LR parser is a pair:

$$(s_0 s_1 \cdots s_m,\ a_i a_{i+1} \cdots a_n\$)$$

where the first component is the stack contents (top on the right), and the second component is the remaining input. This configuration represents the right-sentential form

$$X_1 X_2 \cdots X_m a_i a_{i+1} \cdots a_n$$

in essentially the same way as a shift-reduce parser would; the only difference is that instead of grammar symbols, the stack holds states from which grammar symbols can be recovered. That is, X_i is the grammar symbol represented by state s_i. Note that s_0, the start state of the parser, does not represent a grammar symbol, and serves as a bottom-of-stack marker, as well as playing an important role in the parse.

---

symbol. See for example states 1 and 8 in the LR(0) automaton in Fig. 4.31, which are both entered by transitions on E, or states 2 and 9, which are both entered by transitions on T.

Behavior of the LR Parser

The next move of the parser from the configuration above is determined by reading a_i, the current input symbol, and s_m, the state on top of the stack, and then consulting the entry ACTION$[s_m, a_i]$ in the parsing action table. 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; it shifts the next state s onto the stack, entering the configuration

$$(s_0 s_1 \cdots s_m s,\ a_{i+1} \cdots a_n \$)$$

   The symbol a_i need not be held on the stack, since it can be recovered from s, if needed (which in practice it never is). The current input symbol is now a_{i+1}.

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

$$(s_0 s_1 \cdots s_{m-r} s,\ a_i a_{i+1} \cdots a_n \$)$$

   where r is the length of β, and s = GOTO$[s_{m-r}, A]$. Here the parser first popped r state symbols off the stack, exposing state s_{m-r}. The parser then pushed s, the entry for GOTO$[s_{m-r}, A]$, 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 corresponding to the states popped off the stack, will always match β, the right side of the reducing production.

   The output of an LR parser is generated after a reduce move by executing the semantic action associated with the reducing production. For the time being, we shall assume the output consists of just printing the reducing production.

3. If ACTION$[s_m, a_i]$ = accept, parsing is completed.

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

The LR-parsing algorithm is summarized below. All LR parsers behave in this fashion; the only difference between one LR parser and another is the information in the ACTION and GOTO fields of the parsing table.

Algorithm 4.44: LR-parsing algorithm.

INPUT: An input string w and an LR-parsing table with functions ACTION and GOTO for a grammar G.

OUTPUT: If w is in $L(G)$, the reduction steps of a bottom-up parse for w; otherwise, an error indication.

METHOD: Initially, the parser has s_0 on its stack, where s_0 is the initial state, and $w\$$ in the input buffer. The parser then executes the program in Fig. 4.36. □

```
let a be the first symbol of w$;
while(1) { /* repeat forever */
        let s be the state on top of the stack;
        if ( ACTION[s, a] = shift t ) {
                push t onto the stack;
                let a be the next input symbol;
        } else if ( ACTION[s, a] = reduce A → β ) {
                pop |β| symbols off the stack;
                let state t now be on top of the stack;
                push GOTO[t, A] onto the stack;
                output the production A → β;
        } else if ( ACTION[s, a] = accept ) break; /* parsing is done */
        else call error-recovery routine;
}
```

Figure 4.36: LR-parsing program

Example 4.45: Figure 4.37 shows the ACTION and GOTO functions of an LR-parsing table for the expression grammar (4.1), repeated here with the productions numbered:

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

The codes for the actions are:

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

Note that the value of 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 GOTO[s, A] for nonterminals A. Although we have not yet explained how the entries for Fig. 4.37 were selected, we shall deal with this issue shortly.

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			

Figure 4.37: Parsing table for expression grammar

On input **id** * **id** + **id**, the sequence of stack and input contents is shown in Fig. 4.38. Also shown for clarity, are the sequences of grammar symbols corresponding to the states held on the stack. For example, at line (1) the LR parser is in state 0, the initial state with no grammar symbol, and with **id** the first input symbol. The action in row 0 and column **id** of the action field of Fig. 4.37 is s5, meaning shift by pushing state 5. That is what has happened at line (2): the state symbol 5 has 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**. One state symbol is popped off the stack. State 0 is then exposed. Since the goto of state 0 on F is 3, state 3 is pushed onto the stack. We now have the configuration in line (3). Each of the remaining moves is determined similarly. □

4.6.4 Constructing SLR-Parsing Tables

The SLR method for constructing parsing tables is a good starting point for studying LR parsing. We shall refer to the parsing table constructed by this method as an SLR table, and to an LR parser using an SLR-parsing table as an SLR parser. The other two methods augment the SLR method with lookahead information.

The SLR method begins with LR(0) items and LR(0) automata, introduced in Section 4.5. That is, given a grammar, G, we augment G to produce G', with a new start symbol S'. From G', we construct C, the canonical collection of sets of items for G' together with the GOTO function.

	STACK	SYMBOLS	INPUT	ACTION
(1)	0		**id** $*$ **id** $+$ **id** \$	shift
(2)	0 5	**id**	$*$ **id** $+$ **id** \$	reduce by $F \rightarrow$ **id**
(3)	0 3	F	$*$ **id** $+$ **id** \$	reduce by $T \rightarrow F$
(4)	0 2	T	$*$ **id** $+$ **id** \$	shift
(5)	0 2 7	$T*$	**id** $+$ **id** \$	shift
(6)	0 2 7 5	$T*$ **id**	$+$ **id** \$	reduce by $F \rightarrow$ **id**
(7)	0 2 7 10	$T*F$	$+$ **id** \$	reduce by $T \rightarrow T*F$
(8)	0 2	T	$+$ **id** \$	reduce by $E \rightarrow T$
(9)	0 1	E	$+$ **id** \$	shift
(10)	0 1 6	$E+$	**id** \$	shift
(11)	0 1 6 5	$E+$ **id**	\$	reduce by $F \rightarrow$ **id**
(12)	0 1 6 3	$E+F$	\$	reduce by $T \rightarrow F$
(13)	0 1 6 9	$E+T$	\$	reduce by $E \rightarrow E+T$
(14)	0 1	E	\$	accept

Figure 4.38: Moves of an LR parser on **id** $*$ **id** $+$ **id**

The ACTION and GOTO entries in the parsing table are then constructed using the following algorithm. It requires us to know FOLLOW(A) for each nonterminal A of a grammar (see Section 4.4).

Algorithm 4.46: Constructing an SLR-parsing table.

INPUT: An augmented grammar G'.

OUTPUT: The SLR-parsing table functions ACTION and GOTO for G'.

METHOD:

1. Construct $C = \{I_0, I_1, \ldots, I_n\}$, the collection of sets of LR(0) items for G'.

2. State i is constructed from I_i. The parsing actions for state i are determined as follows:

   (a) 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 must be a terminal.

   (b) 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); here A may not be S'.

   (c) If $[S' \rightarrow S \cdot]$ is in I_i, then set ACTION$[i, \$]$ to "accept."

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

3. The goto transitions for state i are constructed for all nonterminals A using the rule: If GOTO$(I_i, A) = I_j$, then GOTO$[i, A] = j$.

4. All entries not defined by rules (2) and (3) are made "error."

5. The initial state of the parser is the one constructed from the set of items containing $[S' \to \cdot S]$.

□

The parsing table consisting of the ACTION and GOTO functions determined by Algorithm 4.46 is called the *SLR(1) table for G*. An LR parser using the SLR(1) table for G is called the SLR(1) parser for G, and a grammar having an SLR(1) parsing table is said to be *SLR(1)*. We usually omit the "(1)" after the "SLR," since we shall not deal here with parsers having more than one symbol of lookahead.

Example 4.47: Let us construct the SLR table for the augmented expression grammar. The canonical collection of sets of LR(0) items for the grammar was shown in Fig. 4.31. First consider the set of items I_0:

$$
\begin{aligned}
&E' \to \cdot E\\
&E \to \cdot E + T\\
&E \to \cdot T\\
&T \to \cdot T * F\\
&T \to \cdot F\\
&F \to \cdot (E)\\
&F \to \cdot \mathbf{id}
\end{aligned}
$$

The item $F \to \cdot(E)$ gives rise to the entry ACTION$[0, (]$ = shift 4, and the item $F \to \cdot\mathbf{id}$ to the entry ACTION$[0, \mathbf{id}]$ = shift 5. Other items in I_0 yield no actions. Now consider I_1:

$$
\begin{aligned}
&E' \to E\cdot\\
&E \to E\cdot + T
\end{aligned}
$$

The first item yields ACTION$[1, \$]$ = accept, and the second yields ACTION$[1, +]$ = shift 6. Next consider I_2:

$$
\begin{aligned}
&E \to T\cdot\\
&T \to T\cdot * F
\end{aligned}
$$

Since FOLLOW$(E) = \{\$, +,)\}$, the first item makes

$$\text{ACTION}[2, \$] = \text{ACTION}[2, +] = \text{ACTION}[2,)] = \text{reduce } E \to T$$

The second item makes ACTION$[2, *]$ = shift 7. Continuing in this fashion we obtain the ACTION and GOTO tables that were shown in Fig. 4.31. In that figure, the numbers of productions in reduce actions are the same as the order in which they appear in the original grammar (4.1). That is, $E \to E + T$ is number 1, $E \to T$ is 2, and so on. □

Example 4.48: Every SLR(1) grammar is unambiguous, but there are many unambiguous grammars that are not SLR(1). Consider the grammar with productions

$$\begin{array}{lcl} S & \rightarrow & L = R \mid R \\ L & \rightarrow & *R \mid \mathbf{id} \\ R & \rightarrow & L \end{array} \tag{4.49}$$

Think of L and R as standing for l-value and r-value, respectively, and $*$ as an operator indicating "contents of."[5] The canonical collection of sets of LR(0) items for grammar (4.49) is shown in Fig. 4.39.

I_0: $S' \rightarrow \cdot S$
$S \rightarrow \cdot L = R$
$S \rightarrow \cdot R$
$L \rightarrow \cdot * R$
$L \rightarrow \cdot \mathbf{id}$
$R \rightarrow \cdot L$

I_1: $S' \rightarrow S \cdot$

I_2: $S \rightarrow L \cdot = R$
$R \rightarrow L \cdot$

I_3: $S \rightarrow R \cdot$

I_4: $L \rightarrow * \cdot R$
$R \rightarrow \cdot L$
$L \rightarrow \cdot * R$
$L \rightarrow \cdot \mathbf{id}$

I_5: $L \rightarrow \mathbf{id} \cdot$

I_6: $S \rightarrow L = \cdot R$
$R \rightarrow \cdot L$
$L \rightarrow \cdot * R$
$L \rightarrow \cdot \mathbf{id}$

I_7: $L \rightarrow * R \cdot$

I_8: $R \rightarrow L \cdot$

I_9: $S \rightarrow L = R \cdot$

Figure 4.39: Canonical LR(0) collection for grammar (4.49)

Consider the set of items I_2. The first item in this set makes ACTION[2, =] be "shift 6." Since FOLLOW(R) contains = (to see why, consider the derivation $S \Rightarrow L = R \Rightarrow *R = R$), the second item sets ACTION[2, =] to "reduce $R \rightarrow L$." Since there is both a shift and a reduce entry in ACTION[2, =], state 2 has a shift/reduce conflict on input symbol =.

Grammar (4.49) is not ambiguous. This shift/reduce conflict arises from the fact that the SLR parser construction method is not powerful enough to remember enough left context to decide what action the parser should take on input =, having seen a string reducible to L. The canonical and LALR methods, to be discussed next, will succeed on a larger collection of grammars, including

[5] As in Section 2.8.3, an l-value designates a location and an r-value is a value that can be stored in a location.

grammar (4.49). Note, however, that there are unambiguous grammars for which every LR parser construction method will produce a parsing action table with parsing action conflicts. Fortunately, such grammars can generally be avoided in programming language applications. □

4.6.5 Viable Prefixes

Why can LR(0) automata be used to make shift-reduce decisions? The LR(0) automaton for a grammar characterizes the strings of grammar symbols that can appear on the stack of a shift-reduce parser for the grammar. The stack contents must be a prefix of a right-sentential form. If the stack holds α and the rest of the input is x, then a sequence of reductions will take αx to S. In terms of derivations, $S \underset{rm}{\overset{*}{\Rightarrow}} \alpha x$.

Not all prefixes of right-sentential forms can appear on the stack, however, since the parser must not shift past the handle. For example, suppose

$$E \underset{rm}{\overset{*}{\Rightarrow}} F * \mathbf{id} \underset{rm}{\Rightarrow} (E) * \mathbf{id}$$

Then, at various times during the parse, the stack will hold (, $(E$, and (E), but it must not hold $(E)*$, since (E) is a handle, which the parser must reduce to F before shifting $*$.

The prefixes of right sentential forms that can appear on the stack of a shift-reduce parser are called *viable prefixes*. They are defined as follows: a viable prefix is a prefix of a right-sentential form that does not continue past the right end of the rightmost handle of that sentential form. By this definition, it is always possible to add terminal symbols to the end of a viable prefix to obtain a right-sentential form.

SLR parsing is based on the fact that LR(0) automata recognize viable prefixes. We say item $A \rightarrow \beta_1 \cdot \beta_2$ is *valid* for a viable prefix $\alpha\beta_1$ if there is a derivation $S' \underset{rm}{\overset{*}{\Rightarrow}} \alpha A w \underset{rm}{\Rightarrow} \alpha\beta_1\beta_2 w$. In general, an item will be valid for many viable prefixes.

The fact that $A \rightarrow \beta_1 \cdot \beta_2$ 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 $A \rightarrow \beta_1$ 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, and others can be resolved by the methods of Section 4.8, but we should not suppose that all parsing action conflicts can be resolved if the LR method is applied to 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 central 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 the path labeled γ in the LR(0) automaton for the grammar. In essence, the set of valid items embodies

Items as States of an NFA

A nondeterministic finite automaton N for recognizing viable prefixes can be constructed by treating the items themselves as states. 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 3.7.1. Thus, 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') in Fig. 4.33 is just the subset construction itself applied to the NFA N with items as states.

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 4.50: Let us consider the augmented expression grammar again, whose sets of items and GOTO function are exhibited in Fig. 4.31. Clearly, the string $E + T*$ is a viable prefix of the grammar. The automaton of Fig. 4.31 will be in state 7 after having read $E + T*$. State 7 contains the items

$$\begin{aligned} T &\to T * \cdot F \\ F &\to \cdot(E) \\ F &\to \cdot\mathbf{id} \end{aligned}$$

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

$$\begin{aligned} E' &\underset{rm}{\Rightarrow} E \\ &\underset{rm}{\Rightarrow} E + T \\ &\underset{rm}{\Rightarrow} E + T * F \end{aligned} \qquad \begin{aligned} E' &\underset{rm}{\Rightarrow} E \\ &\underset{rm}{\Rightarrow} E + T \\ &\underset{rm}{\Rightarrow} E + T * F \\ &\underset{rm}{\Rightarrow} E + T * (E) \end{aligned} \qquad \begin{aligned} E' &\underset{rm}{\Rightarrow} E \\ &\underset{rm}{\Rightarrow} E + T \\ &\underset{rm}{\Rightarrow} E + T * F \\ &\underset{rm}{\Rightarrow} E + T * \mathbf{id} \end{aligned}$$

The first derivation shows the validity of $T \to T * \cdot F$, the second the validity of $F \to \cdot(E)$, and the third the validity of $F \to \cdot\mathbf{id}$. It can be shown that there are no other valid items for $E + T*$, although we shall not prove that fact here. □

4.6.6 Exercises for Section 4.6

Exercise 4.6.1: Describe all the viable prefixes for the following grammars:

a) The grammar $S \to 0\ S\ 1 \mid 0\ 1$ of Exercise 4.2.2(a).

! b) The grammar $S \rightarrow S\ S\ +\ |\ S\ S\ *\ |\ a$ of Exercise 4.2.1.

! c) The grammar $S \rightarrow S\ (\ S\ )\ |\ \epsilon$ of Exercise 4.2.2(c).

Exercise 4.6.2: Construct the SLR sets of items for the (augmented) grammar of Exercise 4.2.1. Compute the GOTO function for these sets of items. Show the parsing table for this grammar. Is the grammar SLR?

Exercise 4.6.3: Show the actions of your parsing table from Exercise 4.6.2 on the input $aa*a+$.

Exercise 4.6.4: For each of the (augmented) grammars of Exercise 4.2.2(a)-(g):

a) Construct the SLR sets of items and their GOTO function.

b) Indicate any action conflicts in your sets of items.

c) Construct the SLR-parsing table, if one exists.

Exercise 4.6.5: Show that the following grammar:

$$\begin{array}{lcl} S & \rightarrow & A\ a\ A\ b\ |\ B\ b\ B\ a \\ A & \rightarrow & \epsilon \\ B & \rightarrow & \epsilon \end{array}$$

is LL(1) but not SLR(1).

Exercise 4.6.6: Show that the following grammar:

$$\begin{array}{lcl} S & \rightarrow & S\ A\ |\ A \\ A & \rightarrow & a \end{array}$$

is SLR(1) but not LL(1).

!! **Exercise 4.6.7:** Consider the family of grammars G_n defined by:

$$\begin{array}{lcll} S & \rightarrow & A_i\ b_i & \text{for } 1 \leq i \leq n \\ A_i & \rightarrow & a_j\ A_i\ |\ a_j & \text{for } 1 \leq i, j \leq n \text{ and } i \neq j \end{array}$$

Show that:

a) G_n has $2n^2 - n$ productions.

b) G_n has $2^n + n^2 + n$ sets of LR(0) items.

c) G_n is SLR(1).

What does this analysis say about how large LR parsers can get?