! **Exercise 4.6.8:** We suggested that individual items could be regarded as states of a nondeterministic finite automaton, while sets of valid items are the states of a deterministic finite automaton (see the box on "Items as States of an NFA" in Section 4.6.5). For the grammar $S \rightarrow S\,S\,+ \mid S\,S\,* \mid a$ of Exercise 4.2.1:

a) Draw the transition diagram (NFA) for the valid items of this grammar according to the rule given in the box cited above.

b) Apply the subset construction (Algorithm 3.20) to your NFA from part (a). How does the resulting DFA compare to the set of LR(0) items for the grammar?

!! c) Show that in all cases, the subset construction applied to the NFA that comes from the valid items for a grammar produces the LR(0) sets of items.

! **Exercise 4.6.9:** The following is an ambiguous grammar:

$$\begin{array}{lcl} S & \rightarrow & A\,S \mid b \\ A & \rightarrow & S\,A \mid a \end{array}$$

Construct for this grammar its collection of sets of LR(0) items. If we try to build an LR-parsing table for the grammar, there are certain conflicting actions. What are they? Suppose we tried to use the parsing table by nondeterministically choosing a possible action whenever there is a conflict. Show all the possible sequences of actions on input *abab*.

4.7 More Powerful LR Parsers

In this section, we shall extend the previous LR parsing techniques to use one symbol of lookahead on the input. There are two different methods:

1. The "canonical-LR" or just "LR" method, which makes full use of the lookahead symbol(s). This method uses a large set of items, called the LR(1) items.

2. The "lookahead-LR" or "LALR" method, which is based on the LR(0) sets of items, and has many fewer states than typical parsers based on the LR(1) items. By carefully introducing lookaheads into the LR(0) items, we can handle many more grammars with the LALR method than with the SLR method, and build parsing tables that are no bigger than the SLR tables. LALR is the method of choice in most situations.

After introducing both these methods, we conclude with a discussion of how to compact LR parsing tables for environments with limited memory.

4.7.1 Canonical LR(1) Items

We shall now present the most general technique for constructing an LR parsing table from a grammar. Recall that in the SLR method, state i calls for reduction by $A \rightarrow \alpha$ if the set of items I_i contains item $[A \rightarrow \alpha\cdot]$ and a is in FOLLOW(A). In some situations, however, when state i appears on top of the stack, the viable prefix $\beta\alpha$ on the stack is such that βA cannot be followed by a in any right-sentential form. Thus, the reduction by $A \rightarrow \alpha$ should be invalid on input a.

Example 4.51: Let us reconsider Example 4.48, where in state 2 we had item $R \rightarrow L\cdot$, which could correspond to $A \rightarrow \alpha$ above, and a could be the = sign, which is in FOLLOW(R). Thus, the SLR parser calls for reduction by $R \rightarrow L$ in state 2 with = as the next input (the shift action is also called for, because of item $S \rightarrow L\cdot{=}R$ in state 2). However, there is no right-sentential form of the grammar in Example 4.48 that begins $R = \cdots$. Thus state 2, which is the state corresponding to viable prefix L only, should not really call for reduction of that L to R. □

It is possible to carry more information in the state that will allow us to rule out some of these invalid reductions by $A \rightarrow \alpha$. By splitting states when necessary, we can arrange to have each state of an LR parser indicate exactly which input symbols can follow a handle α for which there is a possible reduction to A.

The extra information is incorporated into the state by redefining items to include a terminal symbol as a second component. The general form of an item becomes $[A \rightarrow \alpha \cdot \beta, a]$, where $A \rightarrow \alpha\beta$ is a production and a is a terminal or the right endmarker \$. We call such an object an *LR(1) item.* The 1 refers to the length of the second component, called the *lookahead* of the item.[6] The lookahead has no effect in an item of the form $[A \rightarrow \alpha\cdot\beta, a]$, where β is not ϵ, but an item of the form $[A \rightarrow \alpha\cdot, a]$ calls for a reduction by $A \rightarrow \alpha$ only if the next input symbol is a. Thus, we are compelled to reduce by $A \rightarrow \alpha$ only on those input symbols a for which $[A \rightarrow \alpha\cdot, a]$ is an LR(1) item in the state on top of the stack. The set of such a's will always be a subset of FOLLOW(A), but it could be a proper subset, as in Example 4.51.

Formally, we say LR(1) item $[A \rightarrow \alpha\cdot\beta, a]$ is *valid* for a viable prefix γ if there is a derivation $S \underset{rm}{\overset{*}{\Rightarrow}} \delta A w \underset{rm}{\Rightarrow} \delta\alpha\beta w$, where

1. $\gamma = \delta\alpha$, and

2. Either a is the first symbol of w, or w is ϵ and a is \$.

Example 4.52: Let us consider the grammar

[6] Lookaheads that are strings of length greater than one are possible, of course, but we shall not consider such lookaheads here.

$$S \rightarrow B\ B$$
$$B \rightarrow a\ B \mid b$$

There is a rightmost derivation $S \underset{rm}{\overset{*}{\Rightarrow}} aaBab \underset{rm}{\Rightarrow} aaaBab$. We see that item $[B \rightarrow a{\cdot}B, a]$ is valid for a viable prefix $\gamma = aaa$ by letting $\delta = aa$, $A = B$, $w = ab$, $\alpha = a$, and $\beta = B$ in the above definition. There is also a rightmost derivation $S \underset{rm}{\overset{*}{\Rightarrow}} BaB \underset{rm}{\Rightarrow} BaaB$. From this derivation we see that item $[B \rightarrow a{\cdot}B, \$]$ is valid for viable prefix Baa. □

4.7.2 Constructing LR(1) Sets of Items

The method for building the collection of sets of valid LR(1) items is essentially the same as the one for building the canonical collection of sets of LR(0) items. We need only to modify the two procedures CLOSURE and GOTO.

```
SetOfItems CLOSURE(I) {
    repeat
        for ( each item [A → α·Bβ, a] in I )
            for ( each production B → γ in G' )
                for ( each terminal b in FIRST(βa) )
                    add [B → ·γ, b] to set I;
    until no more items are added to I;
    return I;
}

SetOfItems GOTO(I, X) {
    initialize J to be the empty set;
    for ( each item [A → α·Xβ, a] in I )
        add item [A → αX·β, a] to set J;
    return CLOSURE(J);
}

void items(G') {
    initialize C to 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;
}
```

Figure 4.40: Sets-of-LR(1)-items construction for grammar G'

To appreciate the new definition of the CLOSURE operation, in particular, why b must be in FIRST(βa), consider an item of the form $[A \rightarrow \alpha \cdot B\beta, a]$ in the set of items valid for some viable prefix γ. Then there is a rightmost derivation $S \underset{rm}{\overset{*}{\Rightarrow}} \delta Aax \underset{rm}{\Rightarrow} \delta\alpha B\beta ax$, where $\gamma = \delta\alpha$. Suppose βax derives terminal string by. Then for each production of the form $B \rightarrow \eta$ for some η, we have derivation $S \underset{rm}{\overset{*}{\Rightarrow}} \gamma Bby \underset{rm}{\Rightarrow} \gamma\eta by$. Thus, $[B \rightarrow \cdot\eta, b]$ is valid for γ. Note that b can be the first terminal derived from β, or it is possible that β derives ϵ in the derivation $\beta ax \underset{rm}{\overset{*}{\Rightarrow}} by$, and b can therefore be a. To summarize both possibilities we say that b can be any terminal in FIRST(βax), where FIRST is the function from Section 4.4. Note that x cannot contain the first terminal of by, so FIRST(βax) = FIRST(βa). We now give the LR(1) sets of items construction.

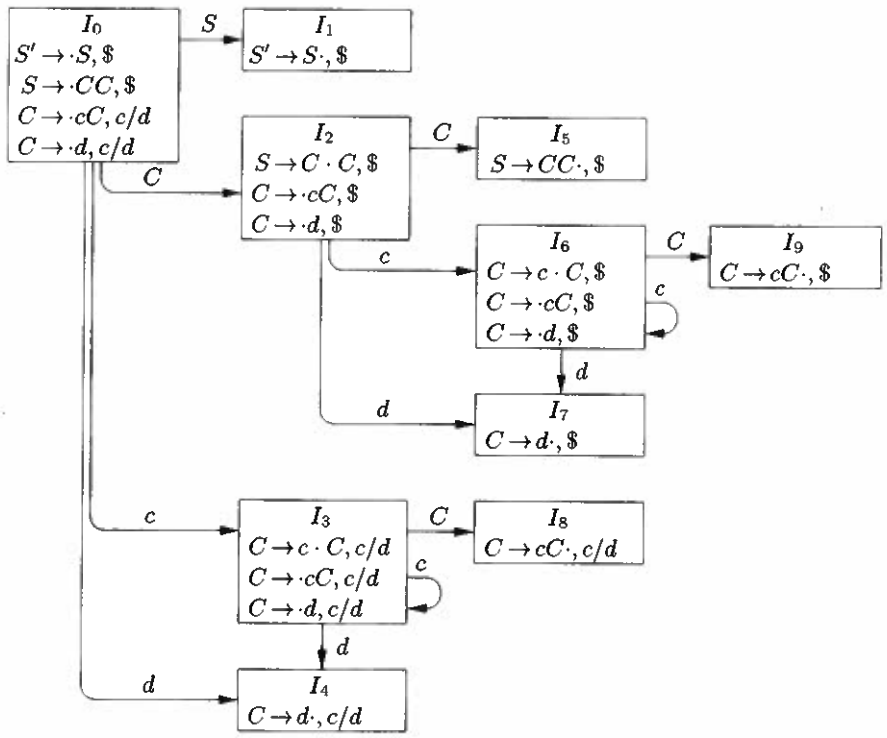

Figure 4.41: The GOTO graph for grammar (4.55)

Algorithm 4.53: Construction of the sets of LR(1) items.

INPUT: An augmented grammar G'.

OUTPUT: The sets of LR(1) items that are the set of items valid for one or more viable prefixes of G'.

METHOD: The procedures CLOSURE and GOTO and the main routine *items* for constructing the sets of items were shown in Fig. 4.40. □

Example 4.54: Consider the following augmented grammar.

$$\begin{array}{lcl} S' & \rightarrow & S \\ S & \rightarrow & C\,C \\ C & \rightarrow & c\,C \mid d \end{array} \tag{4.55}$$

We begin by computing the closure of $\{[S' \rightarrow \cdot S, \$]\}$. To close, we match the item $[S' \rightarrow \cdot S, \$]$ with the item $[A \rightarrow \alpha \cdot B\beta, a]$ in the procedure CLOSURE. That is, $A = S'$, $\alpha = \epsilon$, $B = S$, $\beta = \epsilon$, and $a = \$$. Function CLOSURE tells us to add $[B \rightarrow \cdot\gamma, b]$ for each production $B \rightarrow \gamma$ and terminal b in FIRST(βa). In terms of the present grammar, $B \rightarrow \gamma$ must be $S \rightarrow CC$, and since β is ϵ and a is \$, b may only be \$. Thus we add $[S \rightarrow \cdot CC, \$]$.

We continue to compute the closure by adding all items $[C \rightarrow \cdot\gamma, b]$ for b in FIRST$(C\$)$. That is, matching $[S \rightarrow \cdot CC, \$]$ against $[A \rightarrow \alpha \cdot B\beta, a]$, we have $A = S$, $\alpha = \epsilon$, $B = C$, $\beta = C$, and $a = \$$. Since C does not derive the empty string, FIRST$(C\$)$ = FIRST(C). Since FIRST(C) contains terminals c and d, we add items $[C \rightarrow \cdot cC, c]$, $[C \rightarrow \cdot cC, d]$, $[C \rightarrow \cdot d, c]$ and $[C \rightarrow \cdot d, d]$. None of the new items has a nonterminal immediately to the right of the dot, so we have completed our first set of LR(1) items. The initial set of items is

$$\begin{array}{ll} I_0: & S \rightarrow \cdot S, \$ \\ & S \rightarrow \cdot CC,\ \$ \\ & C \rightarrow \cdot cC,\ c/d \\ & C \rightarrow \cdot d,\ c/d \end{array}$$

The brackets have been omitted for notational convenience, and we use the notation $[C \rightarrow \cdot cC,\ c/d]$ as a shorthand for the two items $[C \rightarrow \cdot cC,\ c]$ and $[C \rightarrow \cdot cC,\ d]$.

Now we compute GOTO(I_0, X) for the various values of X. For $X = S$ we must close the item $[S' \rightarrow S\cdot,\ \$]$. No additional closure is possible, since the dot is at the right end. Thus we have the next set of items

$$I_1: \quad S' \rightarrow S\cdot,\ \$$$

For $X = C$ we close $[S \rightarrow C\cdot C,\ \$]$. We add the C-productions with second component \$ and then can add no more, yielding

$$\begin{array}{ll} I_2: & S \rightarrow C\cdot C,\ \$ \\ & C \rightarrow \cdot cC,\ \$ \\ & C \rightarrow \cdot d,\ \$ \end{array}$$

Next, let $X = c$. We must close $\{[C \rightarrow c\cdot C,\ c/d]\}$. We add the C-productions with second component c/d, yielding

$$
\begin{aligned}
I_3: \quad & C \to c{\cdot}C,\ c/d \\
& C \to {\cdot}cC,\ c/d \\
& C \to {\cdot}d,\ c/d
\end{aligned}
$$

Finally, let $X = d$, and we wind up with the set of items

$$I_4: \quad C \to d{\cdot},\ c/d$$

We have finished considering GOTO on I_0. We get no new sets from I_1, but I_2 has goto's on C, c, and d. For GOTO(I_2, C) we get

$$I_5: \quad S \to CC{\cdot}, \$$$

no closure being needed. To compute GOTO(I_2, c) we take the closure of $\{[C \to c{\cdot}C,\ \$]\}$, to obtain

$$
\begin{aligned}
I_6: \quad & C \to c{\cdot}C,\ \$ \\
& C \to {\cdot}cC,\ \$ \\
& C \to {\cdot}d,\ \$
\end{aligned}
$$

Note that I_6 differs from I_3 only in second components. We shall see that it is common for several sets of LR(1) items for a grammar to have the same first components and differ in their second components. When we construct the collection of sets of LR(0) items for the same grammar, each set of LR(0) items will coincide with the set of first components of one or more sets of LR(1) items. We shall have more to say about this phenomenon when we discuss LALR parsing.

Continuing with the GOTO function for I_2, GOTO(I_2, d) is seen to be

$$I_7: \quad C \to d{\cdot},\ \$$$

Turning now to I_3, the GOTO's of I_3 on c and d are I_3 and I_4, respectively, and GOTO(I_3, C) is

$$I_8: \quad C \to cC{\cdot},\ c/d$$

I_4 and I_5 have no GOTO's, since all items have their dots at the right end. The GOTO's of I_6 on c and d are I_6 and I_7, respectively, and GOTO(I_6, C) is

$$I_9: \quad C \to cC{\cdot},\ \$$$

The remaining sets of items yield no GOTO's, so we are done. Figure 4.41 shows the ten sets of items with their goto's. □

4.7.3 Canonical LR(1) Parsing Tables

We now give the rules for constructing the LR(1) ACTION and GOTO functions from the sets of LR(1) items. These functions are represented by a table, as before. The only difference is in the values of the entries.

Algorithm 4.56: Construction of canonical-LR parsing tables.

INPUT: An augmented grammar G'.

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

METHOD:

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

2. State i of the parser is constructed from I_i. The parsing action for state i is determined as follows.

   (a) If $[A \rightarrow \alpha \cdot a\beta, b]$ 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 $[A \rightarrow \alpha \cdot, a]$ is in I_i, $A \neq S'$, then set ACTION$[i, a]$ to "reduce $A \rightarrow \alpha$."

   (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 LR(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 $[S' \rightarrow \cdot S, \$]$.

□

The table formed from the parsing action and goto functions produced by Algorithm 4.44 is called the *canonical* LR(1) parsing table. An LR parser using this table is called a canonical-LR(1) parser. If the parsing action function has no multiply defined entries, then the given grammar is called an *LR(1) grammar*. As before, we omit the "(1)" if it is understood.

Example 4.57: The canonical parsing table for grammar (4.55) is shown in Fig. 4.42. Productions 1, 2, and 3 are $S \rightarrow CC$, $C \rightarrow cC$, and $C \rightarrow d$, respectively. □

Every SLR(1) grammar is an LR(1) grammar, but for an SLR(1) grammar the canonical LR parser may have more states than the SLR parser for the same grammar. The grammar of the previous examples is SLR and has an SLR parser with seven states, compared with the ten of Fig. 4.42.

STATE	ACTION			GOTO	
	c	d	\$	S	C
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

Figure 4.42: Canonical parsing table for grammar (4.55)

4.7.4 Constructing LALR Parsing Tables

We now introduce our last parser construction method, the LALR (*lookahead*-LR) technique. This method is often used in practice, because the tables obtained by it are considerably smaller than the canonical LR tables, yet most common syntactic constructs of programming languages can be expressed conveniently by an LALR grammar. The same is almost true for SLR grammars, but there are a few constructs that cannot be conveniently handled by SLR techniques (see Example 4.48, for example).

For a comparison of parser size, the SLR and LALR tables for a grammar always have the same number of states, and this number is typically several hundred states for a language like C. The canonical LR table would typically have several thousand states for the same-size language. Thus, it is much easier and more economical to construct SLR and LALR tables than the canonical LR tables.

By way of introduction, let us again consider grammar (4.55), whose sets of LR(1) items were shown in Fig. 4.41. Take a pair of similar looking states, such as I_4 and I_7. Each of these states has only items with first component $C \rightarrow d\cdot$. In I_4, the lookaheads are c or d; in I_7, \$ is the only lookahead.

To see the difference between the roles of I_4 and I_7 in the parser, note that the grammar generates the regular language $\mathbf{c^*dc^*d}$. When reading an input $cc\cdots cdcc\cdots cd$, the parser shifts the first group of c's and their following d onto the stack, entering state 4 after reading the d. The parser then calls for a reduction by $C \rightarrow d$, provided the next input symbol is c or d. The requirement that c or d follow makes sense, since these are the symbols that could begin strings in $\mathbf{c^*d}$. If \$ follows the first d, we have an input like ccd, which is not in the language, and state 4 correctly declares an error if \$ is the next input.

The parser enters state 7 after reading the second d. Then, the parser must