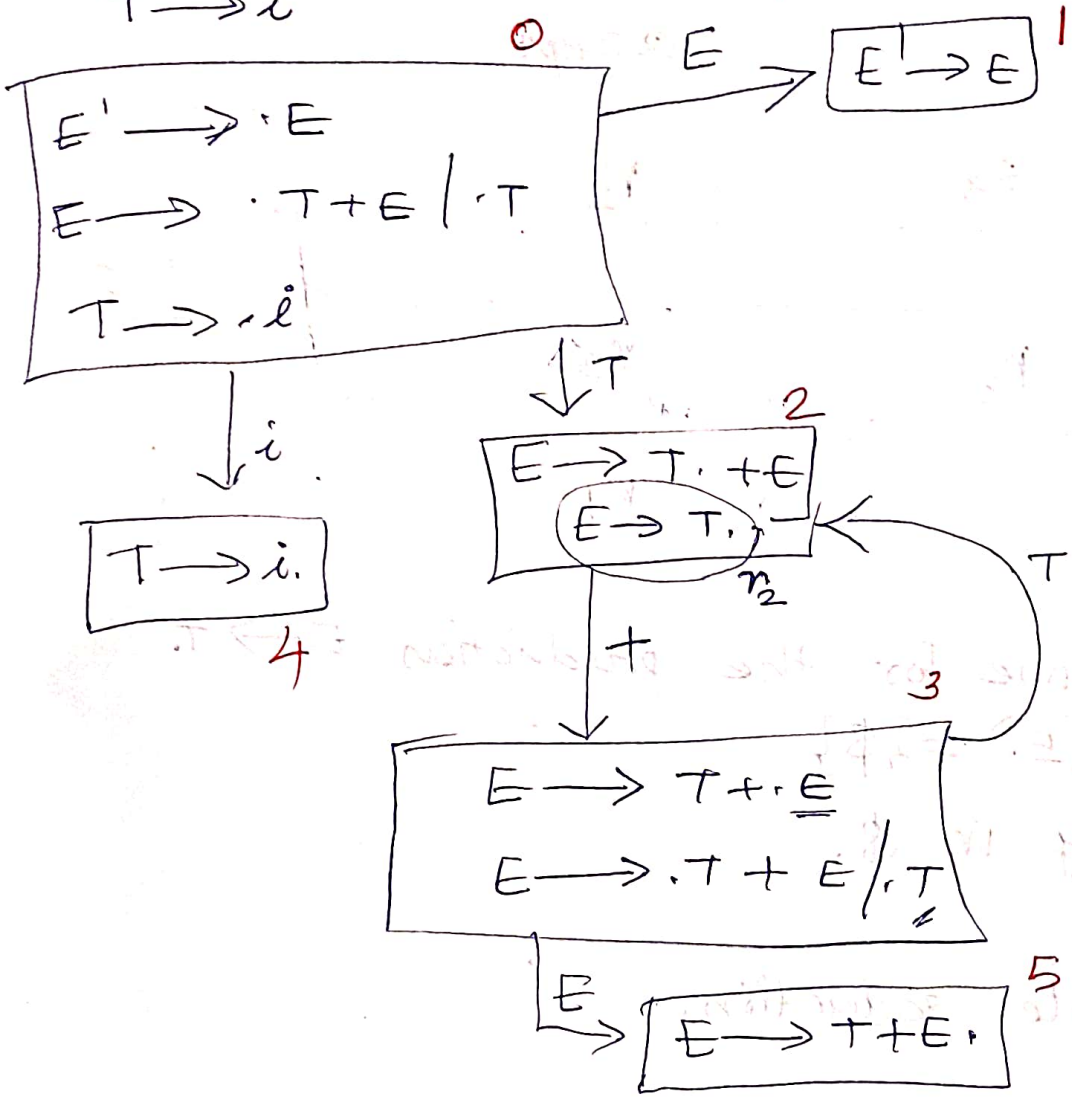
SLR(1)

$E \rightarrow T+E \mid T$

$T \rightarrow i$



State	action			goto	
	+	i	$	E	T
0		s_4		1	2
1			accept		
2	s_3 / r_2	r_2	r_2		
3				5	2
4	r_3	r_3	r_3		
5	r_1	r_1	r_1		

State	action			Goto	
	+	i	$	E	T
0		s_4		1	2
1			accept		
2	s_3		r_2		
3				5	2
4	r_3		r_3		
5			r_1		

- reduction came for the production $E \rightarrow T$.

  Follow $(E) = \{\$\}$

  put r_2 only in $\$$.

- 4th state reduction.

  $T \rightarrow i$.

  Follow $(T) = \{+, \$\}$

- 5th state reduction

  follow $(E) = \{\$\}$.

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 $[A \to \alpha \cdot a\beta]$ 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 $[A \to \alpha \cdot]$ is in I_i, then set ACTION$[i, a]$ to "reduce $A \to \alpha$" for all a in FOLLOW(A); here A may not be S'.

   (c) If $[S' \to S \cdot]$ 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 $\text{GOTO}(I_i, A) = I_j$, then $\text{GOTO}[i, A] = j$.

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 $[S' \to \cdot S]$.

□

P. Q

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.

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.

for con

Example 4.54: Consider the following augmented grammar.

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

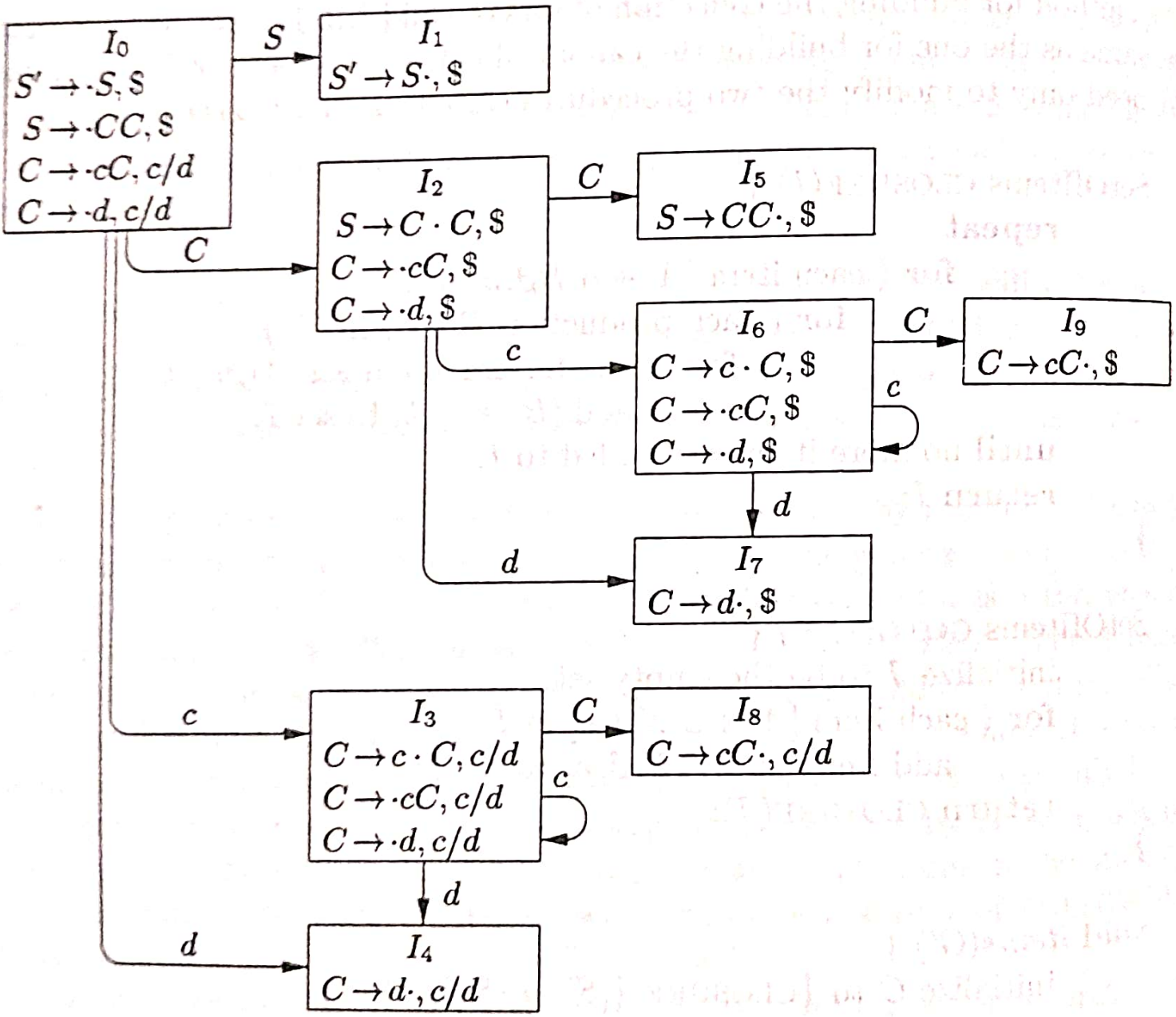


Figure 4.41: The GOTO graph for grammar (4.55)

	Action			goto	
	a	b	$	S	A
0	s_3	s_4		1	2
1			accept		
2	s_6	s_7			5
3	s_3	s_4			8
4	r_3	r_3			
5			r_1		
6	s_6	s_7			9
7			r_3		
8	r_2	r_2			
9			r_2		

I_3 & I_6

I_4 & I_7

I_8 & I_9

	action			goto	
	a	b	$	S	A
0	S_{36}	S_{47}			2
1			Accept		
2	S_{36}	S_{47}			5
3,6	S_{36}	S_{47}			89
4,7	r_3	r_3	r_3		
5			r_1		
8,9	r_2	r_2	r_2		

LALR(1) parsing table for the previous grammar.

Example

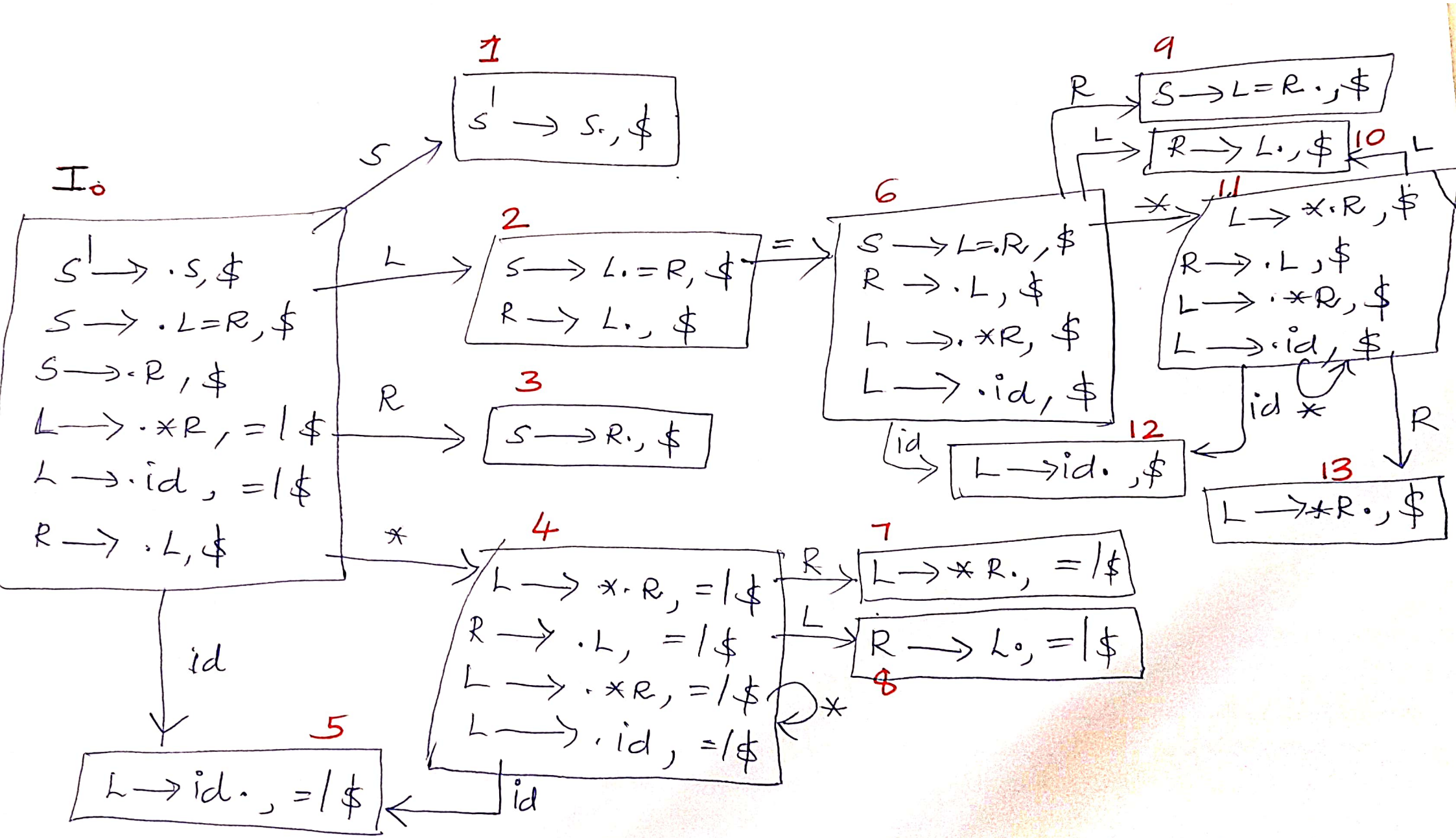
$S \longrightarrow L = R$

$S \longrightarrow R$

$L \longrightarrow *R$

$L \longrightarrow id$

$R \longrightarrow L$


I0
S' → .S, $
S → .L=R, $
S → .R, $
L → .*R, =/$
L → .id, =/$
R → .L, $
S
L
R
*
id
1
S' → S., $
2
S → L.=R, $
R → L., $
=
3
S → R., $
4
L → *.R, =/$
R → .L, =/$
L → .*R, =/$
L → .id, =/$
*
id
5
L → id., =/$
6
S → L=.R, $
R → .L, $
L → .*R, $
L → .id, $
7
L → *R., =/$
8
R → L., =/$
9
S → L=R., $
10
R → L., $
11
L → *.R, $
R → .L, $
L → .*R, $
L → .id, $
12
L → id., $
13
L → *R., $

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 $[A \to \alpha \cdot a\beta, b]$ is in I_i and $\text{GOTO}(I_i, a) = I_j$, then set ACTION$[i, a]$ to "shift j." Here a must be a terminal.
   (b) If $[A \to \alpha \cdot, a]$ is in I_i, $A \neq S'$, then set ACTION$[i, a]$ to "reduce $A \to \alpha$."
   (c) If $[S' \to S \cdot, \$]$ 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 $\text{GOTO}(I_i, A) = I_j$, then $\text{GOTO}[i, A] = j$.
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 $[S' \to \cdot S, \$]$.

□

Algorithm 4.59: An easy, but space-consuming LALR table construction.

INPUT: An augmented grammar G'.

OUTPUT: The LALR 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(1) items.

2. For each core present among the set of LR(1) items, find all sets having that core, and replace these sets by their union.

3. Let $C' = \{J_0, J_1, \ldots, J_m\}$ be the resulting sets of LR(1) items. The parsing actions for state i are constructed from J_i in the same manner as in Algorithm 4.56. If there is a parsing action conflict, the algorithm fails to produce a parser, and the grammar is said not to be LALR(1).

4. The GOTO table is constructed as follows. If J is the union of one or more sets of LR(1) items, that is, $J = I_1 \cup I_2 \cup \cdots \cup I_k$, then the cores of $\text{GOTO}(I_1, X)$, $\text{GOTO}(I_2, X), \ldots, \text{GOTO}(I_k, X)$ are the same, since $I_1, I_2, \ldots, I_k$ all have the same core. Let K be the union of all sets of items having the same core as $\text{GOTO}(I_1, X)$. Then $\text{GOTO}(J, X) = K$.

□