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 $C \to d\cdot$. 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 $C \to d$, 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 see \$ on the input, or it started with a string not of the form $\mathbf{c^*dc^*d}$. It thus makes sense that state 7 should reduce by $C \rightarrow d$ on input \$ and declare error on inputs c or d.

Let us now replace I_4 and I_7 by I_{47}, the union of I_4 and I_7, consisting of the set of three items represented by $[C \rightarrow d\cdot,\ c/d/\$]$. The goto's on d to I_4 or I_7 from I_0, I_2, I_3, and I_6 now enter I_{47}. The action of state 47 is to reduce on any input. The revised parser behaves essentially like the original, although it might reduce d to C in circumstances where the original would declare error, for example, on input like *ccd* or *cdcdc*. The error will eventually be caught; in fact, it will be caught before any more input symbols are shifted.

More generally, we can look for sets of LR(1) items having the same *core*, that is, set of first components, and we may merge these sets with common cores into one set of items. For example, in Fig. 4.41, I_4 and I_7 form such a pair, with core $\{C \rightarrow d\cdot\}$. Similarly, I_3 and I_6 form another pair, with core $\{C \rightarrow c\cdot C, C \rightarrow \cdot cC, C \rightarrow \cdot d\}$. There is one more pair, I_8 and I_9, with common core $\{C \rightarrow cC\cdot\}$. Note that, in general, a core is a set of LR(0) items for the grammar at hand, and that an LR(1) grammar may produce more than two sets of items with the same core.

Since the core of GOTO(I, X) depends only on the core of I, the goto's of merged sets can themselves be merged. Thus, there is no problem revising the goto function as we merge sets of items. The action functions are modified to reflect the non-error actions of all sets of items in the merger.

Suppose we have an LR(1) grammar, that is, one whose sets of LR(1) items produce no parsing-action conflicts. If we replace all states having the same core with their union, it is possible that the resulting union will have a conflict, but it is unlikely for the following reason: Suppose in the union there is a conflict on lookahead a because there is an item $[A \rightarrow \alpha\cdot, a]$ calling for a reduction by $A \rightarrow \alpha$, and there is another item $[B \rightarrow \beta\cdot a\gamma, b]$ calling for a shift. Then some set of items from which the union was formed has item $[A \rightarrow \alpha\cdot, a]$, and since the cores of all these states are the same, it must have an item $[B \rightarrow \beta\cdot a\gamma, c]$ for some c. But then this state has the same shift/reduce conflict on a, and the grammar was not LR(1) as we assumed. Thus, the merging of states with common cores can never produce a shift/reduce conflict that was not present in one of the original states, because shift actions depend only on the core, not the lookahead.

It is possible, however, that a merger will produce a reduce/reduce conflict, as the following example shows.

Example 4.58: Consider the grammar

$$\begin{array}{lcl} S' & \rightarrow & S \\ S & \rightarrow & a\ A\ d \mid b\ B\ d \mid a\ B\ e \mid b\ A\ e \\ A & \rightarrow & c \\ B & \rightarrow & c \end{array}$$

which generates the four strings *acd*, *ace*, *bcd*, and *bce*. The reader can check that the grammar is LR(1) by constructing the sets of items. Upon doing so, we find the set of items $\{[A \to c\cdot,\ d], [B \to c\cdot,\ e]\}$ valid for viable prefix ac and $\{[A \to c\cdot, e], [B \to c\cdot,\ d]\}$ valid for bc. Neither of these sets has a conflict, and their cores are the same. However, their union, which is

$$\begin{array}{l} A \to c\cdot,\ d/e \\ B \to c\cdot,\ d/e \end{array}$$

generates a reduce/reduce conflict, since reductions by both $A \to c$ and $B \to c$ are called for on inputs d and e. □

We are now prepared to give the first of two LALR table-construction algorithms. The general idea is to construct the sets of LR(1) items, and if no conflicts arise, merge sets with common cores. We then construct the parsing table from the collection of merged sets of items. The method we are about to describe serves primarily as a definition of LALR(1) grammars. Constructing the entire collection of LR(1) sets of items requires too much space and time to be useful in practice.

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, $J = I_1 \cap I_2 \cap \cdots \cap I_k$, then the cores of GOTO$(I_1, X)$, GOTO$(I_2, X), \ldots,$ 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 GOTO(I_1, X). Then GOTO$(J, X) = K$.

□

The table produced by Algorithm 4.59 is called the *LALR parsing table* for G. If there are no parsing action conflicts, then the given grammar is said to be an *LALR(1) grammar.* The collection of sets of items constructed in step (3) is called the *LALR(1) collection.*

Example 4.60: Again consider grammar (4.55) whose GOTO graph was shown in Fig. 4.41. As we mentioned, there are three pairs of sets of items that can be merged. I_3 and I_6 are replaced by their union:

$$\begin{array}{ll} I_{36}: & C \rightarrow c{\cdot}C,\ c/d/\$ \\ & C \rightarrow {\cdot}cC,\ c/d/\$ \\ & C \rightarrow {\cdot}d,\ c/d/\$ \end{array}$$

I_4 and I_7 are replaced by their union:

$$I_{47}: \quad C \rightarrow d{\cdot},\ c/d/\$$$

and I_8 and I_9 are replaced by their union:

$$I_{89}: \quad C \rightarrow cC{\cdot},\ c/d/\$$$

The LALR action and goto functions for the condensed sets of items are shown in Fig. 4.43.

STATE	ACTION			GOTO	
	c	d	\$	S	C
0	s36	s47		1	2
1			acc		
2	s36	s47			5
36	s36	s47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

Figure 4.43: LALR parsing table for the grammar of Example 4.54

To see how the GOTO's are computed, consider GOTO(I_{36}, C). In the original set of LR(1) items, GOTO(I_3, C) = I_8, and I_8 is now part of I_{89}, so we make GOTO(I_{36}, C) be I_{89}. We could have arrived at the same conclusion if we considered I_6, the other part of I_{36}. That is, GOTO(I_6, C) = I_9, and I_9 is now part of I_{89}. For another example, consider GOTO(I_2, c), an entry that is exercised after the shift action of I_2 on input c. In the original sets of LR(1) items, GOTO(I_2, c) = I_6. Since I_6 is now part of I_{36}, GOTO(I_2, c) becomes I_{36}. Thus, the entry in Fig. 4.43 for state 2 and input c is made s36, meaning shift and push state 36 onto the stack. □

When presented with a string from the language **c*dc*d**, both the LR parser of Fig. 4.42 and the LALR parser of Fig. 4.43 make exactly the same sequence of shifts and reductions, although the names of the states on the stack may differ. For instance, if the LR parser puts I_3 or I_6 on the stack, the LALR parser will put I_{36} on the stack. This relationship holds in general for an LALR grammar. The LR and LALR parsers will mimic one another on correct inputs.

When presented with erroneous input, the LALR parser may proceed to do some reductions after the LR parser has declared an error. However, the LALR parser will never shift another symbol after the LR parser declares an error. For example, on input *ccd* followed by \$, the LR parser of Fig. 4.42 will put

$$0\ 3\ 3\ 4$$

on the stack, and in state 4 will discover an error, because \$ is the next input symbol and state 4 has action error on \$. In contrast, the LALR parser of Fig. 4.43 will make the corresponding moves, putting

$$0\ 36\ 36\ 47$$

on the stack. But state 47 on input \$ has action reduce $C \to d$. The LALR parser will thus change its stack to

$$0\ 36\ 36\ 89$$

Now the action of state 89 on input \$ is reduce $C \to cC$. The stack becomes

$$0\ 36\ 89$$

whereupon a similar reduction is called for, obtaining stack

$$0\ 2$$

Finally, state 2 has action error on input \$, so the error is now discovered.

4.7.5 Efficient Construction of LALR Parsing Tables

There are several modifications we can make to Algorithm 4.59 to avoid constructing the full collection of sets of LR(1) items in the process of creating an LALR(1) parsing table.

- First, we can represent any set of LR(0) or LR(1) items I by its kernel, that is, by those items that are either the initial item — $[S' \to \cdot S]$ or $[S' \to \cdot S, \$]$ — or that have the dot somewhere other than at the beginning of the production body.

- We can construct the LALR(1)-item kernels from the LR(0)-item kernels by a process of propagation and spontaneous generation of lookaheads, that we shall describe shortly.

- If we have the LALR(1) kernels, we can generate the LALR(1) parsing table by closing each kernel, using the function CLOSURE of Fig. 4.40, and then computing table entries by Algorithm 4.56, as if the LALR(1) sets of items were canonical LR(1) sets of items.

Example 4.61: We shall use as an example of the efficient LALR(1) table-construction method the non-SLR grammar from Example 4.48, which we reproduce below in its augmented form:

$$\begin{array}{lcl} S' & \rightarrow & S \\ S & \rightarrow & L = R \mid R \\ L & \rightarrow & *R \mid \textbf{id} \\ R & \rightarrow & L \end{array}$$

The complete sets of LR(0) items for this grammar were shown in Fig. 4.39. The kernels of these items are shown in Fig. 4.44. □

I_0:	$S' \rightarrow \cdot S$	I_5:	$L \rightarrow \textbf{id}\cdot$
I_1:	$S' \rightarrow S\cdot$	I_6:	$S \rightarrow L = \cdot R$
I_2:	$S \rightarrow L\cdot = R$ $R \rightarrow L\cdot$	I_7:	$L \rightarrow *R\cdot$
I_3:	$S \rightarrow R\cdot$	I_8:	$R \rightarrow L\cdot$
I_4:	$L \rightarrow *\cdot R$	I_9:	$S \rightarrow L = R\cdot$

Figure 4.44: Kernels of the sets of LR(0) items for grammar (4.49)

Now we must attach the proper lookaheads to the LR(0) items in the kernels, to create the kernels of the sets of LALR(1) items. There are two ways a lookahead b can get attached to an LR(0) item $B \rightarrow \gamma\cdot\delta$ in some set of LALR(1) items J:

1. There is a set of items I, with a kernel item $A \rightarrow \alpha\cdot\beta, a$, and $J = \text{GOTO}(I, X)$, and the construction of

$$\text{GOTO}\big(\text{CLOSURE}(\{[A \rightarrow \alpha\cdot\beta, a]\}), X\big)$$

as given in Fig. 4.40, contains $[B \rightarrow \gamma\cdot\delta, b]$, regardless of a. Such a lookahead b is said to be generated *spontaneously* for $B \rightarrow \gamma\cdot\delta$.

2. As a special case, lookahead \$ is generated spontaneously for the item $S' \rightarrow \cdot S$ in the initial set of items.

3. All is as in (1), but $a = b$, and $\text{GOTO}\big(\text{CLOSURE}(\{[A \rightarrow \alpha\cdot\beta, b]\}), X\big)$, as given in Fig. 4.40, contains $[B \rightarrow \gamma\cdot\delta, b]$ only because $A \rightarrow \alpha\cdot\beta$ has b as one of its associated lookaheads. In such a case, we say that lookaheads *propagate* from $A \rightarrow \alpha\cdot\beta$ in the kernel of I to $B \rightarrow \gamma\cdot\delta$ in the kernel of J. Note that propagation does not depend on the particular lookahead symbol; either all lookaheads propagate from one item to another, or none do.

We need to determine the spontaneously generated lookaheads for each set of LR(0) items, and also to determine which items propagate lookaheads from which. The test is actually quite simple. Let # be a symbol not in the grammar at hand. Let $A \to \alpha{\cdot}\beta$ be a kernel LR(0) item in set I. Compute, for each X, $J = \text{GOTO}\big(\text{CLOSURE}(\{[A \to \alpha{\cdot}\beta, \#]\}), X\big)$. For each kernel item in J, we examine its set of lookaheads. If # is a lookahead, then lookaheads propagate to that item from $A \to \alpha{\cdot}\beta$. Any other lookahead is spontaneously generated. These ideas are made precise in the following algorithm, which also makes use of the fact that the only kernel items in J must have X immediately to the left of the dot; that is, they must be of the form $B \to \gamma X{\cdot}\delta$.

Algorithm 4.62: Determining lookaheads.

INPUT: The kernel K of a set of LR(0) items I and a grammar symbol X.

OUTPUT: The lookaheads spontaneously generated by items in I for kernel items in GOTO(I, X) and the items in I from which lookaheads are propagated to kernel items in GOTO(I, X).

METHOD: The algorithm is given in Fig. 4.45. □

```
for ( each item $A \to \alpha{\cdot}\beta$ in $K$ ) {
        $J$ := CLOSURE({[$A \to \alpha{\cdot}\beta$,#]} );
        if ( [$B \to \gamma{\cdot}X\delta, a$] is in $J$, and $a$ is not # )
                conclude that lookahead $a$ is generated spontaneously for item
                    $B \to \gamma X{\cdot}\delta$ in GOTO($I, X$);
        if ( [$B \to \gamma{\cdot}X\delta$, #] is in $J$ )
                conclude that lookaheads propagate from $A \to \alpha{\cdot}\beta$ in $I$ to
                    $B \to \gamma X{\cdot}\delta$ in GOTO($I, X$);
}
```

Figure 4.45: Discovering propagated and spontaneous lookaheads

We are now ready to attach lookaheads to the kernels of the sets of LR(0) items to form the sets of LALR(1) items. First, we know that \$ is a lookahead for $S' \to {\cdot}S$ in the initial set of LR(0) items. Algorithm 4.62 gives us all the lookaheads generated spontaneously. After listing all those lookaheads, we must allow them to propagate until no further propagation is possible. There are many different approaches, all of which in some sense keep track of "new" lookaheads that have propagated into an item but which have not yet propagated out. The next algorithm describes one technique to propagate lookaheads to all items.

Algorithm 4.63: Efficient computation of the kernels of the LALR(1) collection of sets of items.

INPUT: An augmented grammar G'.

OUTPUT: The kernels of the LALR(1) collection of sets of items for G'.

METHOD:

1. Construct the kernels of the sets of LR(0) items for G. If space is not at a premium, the simplest way is to construct the LR(0) sets of items, as in Section 4.6.2, and then remove the nonkernel items. If space is severely constrained, we may wish instead to store only the kernel items for each set, and compute GOTO for a set of items I by first computing the closure of I.

2. Apply Algorithm 4.62 to the kernel of each set of LR(0) items and grammar symbol X to determine which lookaheads are spontaneously generated for kernel items in GOTO(I, X), and from which items in I lookaheads are propagated to kernel items in GOTO(I, X).

3. Initialize a table that gives, for each kernel item in each set of items, the associated lookaheads. Initially, each item has associated with it only those lookaheads that we determined in step (2) were generated spontaneously.

4. Make repeated passes over the kernel items in all sets. When we visit an item i, we look up the kernel items to which i propagates its lookaheads, using information tabulated in step (2). The current set of lookaheads for i is added to those already associated with each of the items to which i propagates its lookaheads. We continue making passes over the kernel items until no more new lookaheads are propagated.

□

Example 4.64: Let us construct the kernels of the LALR(1) items for the grammar of Example 4.61. The kernels of the LR(0) items were shown in Fig. 4.44. When we apply Algorithm 4.62 to the kernel of set of items I_0, we first compute CLOSURE$(\{[S' \to \cdot S, \#]\})$, which is

$$\begin{array}{ll} S' \to \cdot S,\ \# & L \to \cdot * R,\ \#/= \\ S \to \cdot L = R,\ \# & L \to \cdot \mathbf{id},\ \#/= \\ S \to \cdot R,\ \# & R \to \cdot L,\ \# \end{array}$$

Among the items in the closure, we see two where the lookahead $=$ has been generated spontaneously. The first of these is $L \to \cdot * R$. This item, with $*$ to the right of the dot, gives rise to $[L \to *\cdot R, =]$. That is, $=$ is a spontaneously generated lookahead for $L \to *\cdot R$, which is in set of items I_4. Similarly, $[L \to \cdot\mathbf{id}, =]$ tells us that $=$ is a spontaneously generated lookahead for $L \to \mathbf{id}\cdot$ in I_5.

As $\#$ is a lookahead for all six items in the closure, we determine that the item $S' \to \cdot S$ in I_0 propagates lookaheads to the following six items:

$S' \to S\cdot$ in I_1	$L \to *\cdot R$ in I_4
$S \to L\cdot = R$ in I_2	$L \to \mathbf{id}\cdot$ in I_5
$S \to R\cdot$ in I_3	$R \to L\cdot$ in I_2

FROM		TO	
I_0:	$S' \to \cdot S$	I_1:	$S' \to S\cdot$
		I_2:	$S \to L\cdot = R$
		I_2:	$R \to L\cdot$
		I_3:	$S \to R\cdot$
		I_4:	$L \to *\cdot R$
		I_5:	$L \to \mathbf{id}\cdot$
I_2:	$S \to L\cdot = R$	I_6:	$S \to L = \cdot R$
I_4:	$L \to *\cdot R$	I_4:	$L \to *\cdot R$
		I_5:	$L \to \mathbf{id}\cdot$
		I_7:	$L \to *R\cdot$
		I_8:	$R \to L\cdot$
I_6:	$S \to L = \cdot R$	I_4:	$L \to *\cdot R$
		I_5:	$L \to \mathbf{id}\cdot$
		I_8:	$R \to L\cdot$
		I_9:	$S \to L = R\cdot$

Figure 4.46: Propagation of lookaheads

In Fig. 4.47, we show steps (3) and (4) of Algorithm 4.63. The column labeled INIT shows the spontaneously generated lookaheads for each kernel item. These are only the two occurrences of = discussed earlier, and the spontaneous lookahead \$ for the initial item $S' \to \cdot S$.

On the first pass, the lookahead \$ propagates from $S' \to S$ in I_0 to the six items listed in Fig. 4.46. The lookahead = propagates from $L \to *\cdot R$ in I_4 to items $L \to *R\cdot$ in I_7 and $R \to L\cdot$ in I_8. It also propagates to itself and to $L \to \mathbf{id}\cdot$ in I_5, but these lookaheads are already present. In the second and third passes, the only new lookahead propagated is \$, discovered for the successors of I_2 and I_4 on pass 2 and for the successor of I_6 on pass 3. No new lookaheads are propagated on pass 4, so the final set of lookaheads is shown in the rightmost column of Fig. 4.47.

Note that the shift/reduce conflict found in Example 4.48 using the SLR method has disappeared with the LALR technique. The reason is that only lookahead \$ is associated with $R \to L\cdot$ in I_2, so there is no conflict with the parsing action of shift on = generated by item $S \to L\cdot{=}R$ in I_2. □

SET	ITEM	LOOKAHEADS			
		INIT	PASS 1	PASS 2	PASS 3
I_0:	$S' \to \cdot S$	\$	\$	\$	\$
I_1:	$S' \to S\cdot$		\$	\$	\$
I_2:	$S \to L\cdot = R$		\$	\$	\$
	$R \to L\cdot$		\$	\$	\$
I_3:	$S \to R\cdot$		\$	\$	\$
I_4:	$L \to *\cdot R$	=	=/\$	=/\$	=/\$
I_5:	$L \to \mathbf{id}\cdot$	=	=/\$	=/\$	=/\$
I_6:	$S \to L = \cdot R$			\$	\$
I_7:	$L \to *R\cdot$		=	=/\$	=/\$
I_8:	$R \to L\cdot$		=	=/\$	=/\$
I_9:	$S \to L = R\cdot$				\$

Figure 4.47: Computation of lookaheads

4.7.6 Compaction of LR Parsing Tables

A typical programming language grammar with 50 to 100 terminals and 100 productions may have an LALR parsing table with several hundred states. The action function may easily have 20,000 entries, each requiring at least 8 bits to encode. On small devices, a more efficient encoding than a two-dimensional array may be important. We shall mention briefly a few techniques that have been used to compress the ACTION and GOTO fields of an LR parsing table.

One useful technique for compacting the action field is to recognize that usually many rows of the action table are identical. For example, in Fig. 4.42, states 0 and 3 have identical action entries, and so do 2 and 6. We can therefore save considerable space, at little cost in time, if we create a pointer for each state into a one-dimensional array. Pointers for states with the same actions point to the same location. To access information from this array, we assign each terminal a number from zero to one less than the number of terminals, and we use this integer as an offset from the pointer value for each state. In a given state, the parsing action for the ith terminal will be found i locations past the pointer value for that state.

Further space efficiency can be achieved at the expense of a somewhat slower parser by creating a list for the actions of each state. The list consists of (terminal-symbol, action) pairs. The most frequent action for a state can be