| 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 $i$th 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

placed at the end of the list, and in place of a terminal we may use the notation "**any**," meaning that if the current input symbol has not been found so far on the list, we should do that action no matter what the input is. Moreover, error entries can safely be replaced by reduce actions, for further uniformity along a row. The errors will be detected later, before a shift move.

**Example 4.65:** Consider the parsing table of Fig. 4.37. First, note that the actions for states 0, 4, 6, and 7 agree. We can represent them all by the list

| SYMBOL | ACTION |
|---|---|
| **id** | s5 |
| ( | s4 |
| **any** | error |

State 1 has a similar list:

| | |
|---|---|
| + | s6 |
| $ | acc |
| **any** | error |

In state 2, we can replace the error entries by r2, so reduction by production 2 will occur on any input but $*$. Thus the list for state 2 is

| | |
|---|---|
| $*$ | s7 |
| **any** | r2 |

State 3 has only error and r4 entries. We can replace the former by the latter, so the list for state 3 consists of only the pair (**any**, r4). States 5, 10, and 11 can be treated similarly. The list for state 8 is

| | |
|---|---|
| + | s6 |
| ) | s11 |
| **any** | error |

and for state 9

| | |
|---|---|
| $*$ | s7 |
| **any** | r1 |

□

We can also encode the GOTO table by a list, but here it appears more efficient to make a list of pairs for each nonterminal $A$. Each pair on the list for $A$ is of the form $(currentState, nextState)$, indicating

$$\text{GOTO}[currentState, A] \;=\; nextState$$

This technique is useful because there tend to be rather few states in any one column of the GOTO table. The reason is that the GOTO on nonterminal $A$ can only be a state derivable from a set of items in which some items have $A$ immediately to the left of a dot. No set has items with $X$ and $Y$ immediately to the left of a dot if $X \neq Y$. Thus, each state appears in at most one GOTO column.

For more space reduction, we note that the error entries in the goto table are never consulted. We can therefore replace each error entry by the most common non-error entry in its column. This entry becomes the default; it is represented in the list for each column by one pair with **any** in place of *currentState*.

**Example 4.66:** Consider Fig. 4.37 again. The column for $F$ has entry 10 for state 7, and all other entries are either 3 or error. We may replace error by 3 and create for column $F$ the list

| CURRENTSTATE | NEXTSTATE |
|---|---|
| 7 | 10 |
| **any** | 3 |

Similarly, a suitable list for column $T$ is

| | |
|---|---|
| 6 | 9 |
| **any** | 2 |

For column $E$ we may choose either 1 or 8 to be the default; two entries are necessary in either case. For example, we might create for column $E$ the list

| | |
|---|---|
| 4 | 8 |
| **any** | 1 |

□

This space savings in these small examples may be misleading, because the total number of entries in the lists created in this example and the previous one together with the pointers from states to action lists and from nonterminals to next-state lists, result in unimpressive space savings over the matrix implementation of Fig. 4.37. For practical grammars, the space needed for the list representation is typically less than ten percent of that needed for the matrix representation. The table-compression methods for finite automata that were discussed in Section 3.9.8 can also be used to represent LR parsing tables.

### 4.7.7 Exercises for Section 4.7

**Exercise 4.7.1:** Construct the

a) canonical LR, and

b) LALR

sets of items for the grammar $S \rightarrow S\ S\ +\ |\ S\ S\ *\ |\ a$ of Exercise 4.2.1.

**Exercise 4.7.2:** Repeat Exercise 4.7.1 for each of the (augmented) grammars of Exercise 4.2.2(a)–(g).

! **Exercise 4.7.3:** For the grammar of Exercise 4.7.1, use Algorithm 4.63 to compute the collection of LALR sets of items from the kernels of the LR(0) sets of items.

! **Exercise 4.7.4:** Show that the following grammar

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

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

! **Exercise 4.7.5:** Show that the following grammar

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

is LR(1) but not LALR(1).

## 4.8 Using Ambiguous Grammars

It is a fact that every ambiguous grammar fails to be LR and thus is not in any of the classes of grammars discussed in the previous two sections. However, certain types of ambiguous grammars are quite useful in the specification and implementation of languages. For language constructs like expressions, an ambiguous grammar provides a shorter, more natural specification than any equivalent unambiguous grammar. Another use of ambiguous grammars is in isolating commonly occurring syntactic constructs for special-case optimization. With an ambiguous grammar, we can specify the special-case constructs by carefully adding new productions to the grammar.

Although the grammars we use are ambiguous, in all cases we specify disambiguating rules that allow only one parse tree for each sentence. In this way, the overall language specification becomes unambiguous, and sometimes it becomes possible to design an LR parser that follows the same ambiguity-resolving choices. We stress that ambiguous constructs should be used sparingly and in a strictly controlled fashion; otherwise, there can be no guarantee as to what language is recognized by a parser.