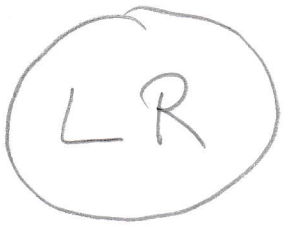
LR

that if the marker in the string is preceded by α and is followed by β, the construction must be replaced by $\gamma\Delta\delta$. The rules are tried in order starting from the top and the first one to match is applied; processing then resumes on the resulting string, starting from the top of the list, and the process is repeated until no rule matches.

Although Floyd productions were not primarily designed as a parsing tool but rather as a general string manipulation language, the identification of the Δ in the string with the gap in a bottom-up parser suggests itself and was already made in Floyd's original article [113]. Floyd productions for the grammar of Figure 9.2 are given in Figure 9.11. The parser is started with the Δ at the left of the input.

```
Δ n      =>  n Δ
Δ (      =>  ( Δ
n Δ      =>  F Δ
T Δ ×    =>  T× Δ
T×F Δ    =>  T Δ
F Δ      =>  T Δ
E+T Δ    =>  E Δ
T Δ      =>  E Δ
(E) Δ    =>  F Δ
Δ +      =>  + Δ
Δ )      =>  ) Δ
Δ #      =>  # Δ
#E# Δ    =>  S Δ
```

Fig. 9.11. Floyd productions for the grammar of Figure 9.2

The apparent convenience and conciseness of Floyd productions makes it very tempting to write parsers in them by hand, but Floyd productions are very sensitive to the order in which the rules are listed and a small inaccuracy in the order can have a devastating effect.

9.4 LR Methods

The LR methods are based on the combination of two ideas that have already been touched upon in previous sections. To reiterate, the problem is to find the handle in a sentential form as efficiently as possible, for as large a class of grammars as possible. Such a handle is searched for from left to right. Now, from Section 5.10 we recall that a very efficient way to find a string in a left-to-right search is by constructing a finite-state automaton. Just doing this is, however, not good enough. It is quite easy to construct an FS automaton that would recognize any of the right-hand sides in the grammar efficiently, but it would just find the leftmost reducible substring in the sentential form. This substring, however, often does not identify the correct handle.

The idea can be made practical by applying the same trick that was used in the Earley parser to drastically reduce the fan-out of the breadth-first search (see Section 7.2): start the automaton with the start rule of the grammar and only consider,

in any position, right-hand sides that could be derived from the start symbol. This top-down restriction device served in the Earley parser to reduce the cost to $O(n^3)$, here we require the grammar to be such that it reduces the cost to $O(n)$. The resulting automaton is started in its initial state at the left end of the sentential form and allowed to run to the right. It has the property that it stops at the right end of the handle segment and that its accepting state tells us how to reduce the handle; if it ends in an error state the sentential form was incorrect. Note that this accepting state is an accepting state of the handle-finding automaton, not of the LR parser; the latter accepts the input only when it has been completely reduced to the start symbol.

Once we have found the handle, we follow the standard procedure for bottom-up parsers: we reduce the handle to its parent non-terminal as described at the beginning of Chapter 7. This gives us a new "improved" sentential form, which, in principle should be scanned anew by the automaton from the left, to find the next handle. But since nothing has changed in the sentential form between its left end and the point of reduction, the automaton will go through the same movements as before, and we can save it the trouble by remembering its states and storing them between the tokens on the stack. This leads us to the standard setup for an LR parser, shown in Figure 9.12 (compare Figure 7.1). Here $\mathbf{s}_1$ is the initial state, $\mathbf{s}_g \cdots \mathbf{s}_b$ are the states from

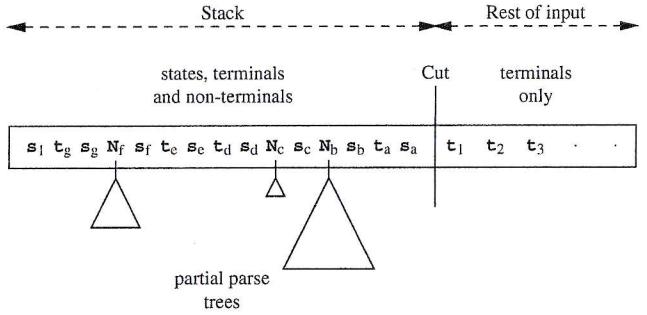

Fig. 9.12. The structure of an LR parse

previous scans, and $\mathbf{s}_a$ is the top, deciding, state.

By far the most important component in an LR parser is the handle-finding automaton, and there are many methods to construct one. The most basic one is LR(0) (Section 9.5); the most powerful one is LR(1) (Section 9.6); and the most practical one is LALR(1) (Section 9.7). In its decision process the LR automaton makes a very modest use of the rest of the input (none at all for LR(0) and a one-token look-ahead for LR(1) and LALR(1)); several extensions of LR parsing exist that involve the rest of the input to a much larger extent (Sections 9.13.2 and 10.2).

Deterministic handle-finding automata can be constructed for any CF grammar, which sounds promising, but the problem is that an accepting state may allow the automaton to continue searching in addition to identifying a handle (in which case we have a shift/reduce conflict), or identify more than one handle (and we have a reduce/reduce conflict). (Both types of conflicts are explained in Section 9.5.3.) In

other words, the automaton is deterministic; the attached semantics is not. If that happens the LR method used is not strong enough for the grammar. It is easy to see that there are grammars for which no LR method will be strong enough; the grammar of Figure 9.13 produces strings consisting of an odd number of **a**s, the middle of which is the handle. But finding the middle of a string is not a feature of

$$S_S \rightarrow \texttt{a}\ S\ \texttt{a} \mid \texttt{a}$$

Fig. 9.13. An unambiguous non-deterministic grammar

LR parsers, not even of the extended and improved versions.

As with the Earley parser, LR parsers can be improved by using look-ahead, and almost all of them are. An LR parser with a look-ahead of k tokens is called LR(k). Just as the Earley parser, it requires k end-of-input markers to be appended to the input; this implies that an LR(0) parser does not need end-of-input markers.

9.5 LR(0)

Since practical handle-finding FS automata easily get so big that their states cannot be displayed on a single page of a book, we shall use the grammar of Figure 9.14 for our examples. It describes very simple arithmetic expressions, terminated with a **$**.

1. $S_S \rightarrow$ E $
2. E → E - T
3. E → T
4. T → n
5. T → (E)

Fig. 9.14. A very simple grammar for differences of numbers

An example of a string in the language is **n- (n-n) $**; the **n** stands for any number. The only arithmetic operator in the grammar is the **-**; it serves to remind us that the proper parse tree must be derived, since **(n-n) -n$** is not the same as **n- (n-n) $**.

9.5.1 The LR(0) Automaton

We set out to construct a top-down-restricted handle-recognizing FS automaton for the grammar of Figure 9.14, and start by constructing a non-deterministic version. We recall that a non-deterministic automaton can be drawn as a set of states connected by arrows (transitions), each marked with one symbol or with ε. Each state will contain one *item*. Like in the Earley parser an item consists of a grammar rule with a dot • embedded in its right-hand side. An item $X \rightarrow \cdots Y \bullet Z \cdots$ in a state

means that the NFA bets on $X \rightarrow \cdots YZ \cdots$ being the handle and that it has already recognized $\cdots Y$. Unlike the Earley parser there are no back-pointers.

To simplify the explanation of the transitions involved, we introduce a second kind of state, which we call a *station*. It has only ε-arrows incoming and outgoing, contains something of the form $\bullet X$ and is drawn in a rectangle rather than in an ellipse. When the automaton is in such a station at some point in the sentential form, it assumes that at this point a handle starts which reduces to X. Consequently each $\bullet X$ station has ε-transitions to items for all rules for X, each with the dot at the left end, since no part of the rule has yet been recognized; see Figure 9.15. Equally reasonably, each state holding an item $X \rightarrow \cdots \bullet Z \cdots$ has an ε-transition to the station $\bullet Z$, since the bet on an X may be over-optimistic and the automaton may have to settle for a Z. The third and last source of arrows in the NFA is straightforward. From each state containing $X \rightarrow \cdots \bullet P \cdots$ there is a P-transition to the state containing $X \rightarrow \cdots P \bullet \cdots$, for P a terminal or a non-terminal. This corresponds to the move the automaton makes when it really meets a P. Note that the sentential form may contain non-terminals, so transitions on non-terminals should also be defined.

With this knowledge we refer to Figure 9.15. The stations for **S**, **E** and **T** are

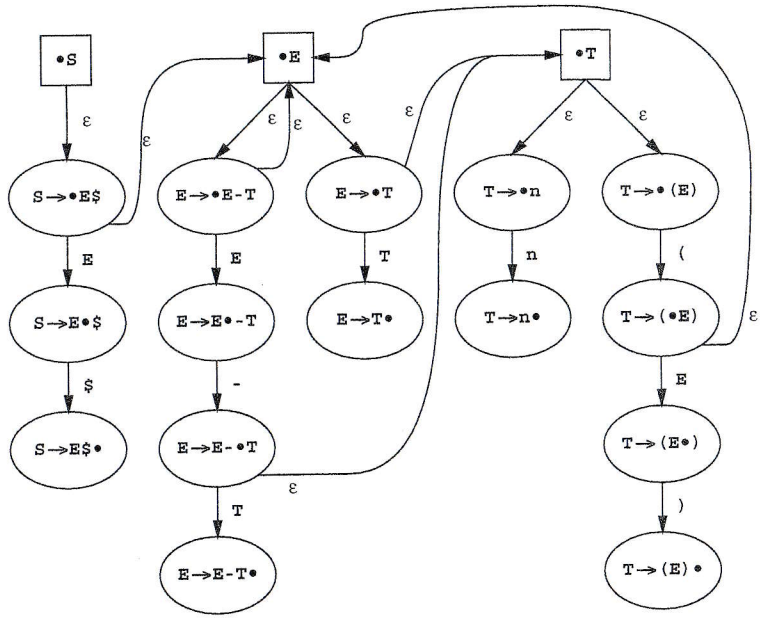

Fig. 9.15. A non-deterministic handle recognizer for the grammar of Figure 9.14

drawn at the top of the picture, to show how they lead to all possible items for **S**, **E** and **T**, respectively. From each station ε-arrows fan out to all states containing items with the dot at the left, one for each rule for the non-terminal in that station; from each such state non-ε-arrows lead down to further states. Now the picture is almost

complete. All that needs to be done is to scan the items for a dot followed by a non-terminal (readily discernible from the outgoing arrow marked with it) and to connect each such item to the corresponding station through an ε-arrow. This completes the picture.

There are three things to be noted about this picture. First, for each grammar rule with a right-hand side of length l there are $l+1$ items and they are easily found in the picture. Moreover, for a grammar with r different non-terminals, there are r stations. So the number of states is roughly proportional to the size of the grammar, which assures us that the automaton will have a modest number of states. For the average grammar of a hundred rules something like 300 states is usual. The second thing to note is that all states have outgoing arrows except the ones which contain a reduce item, an item with the dot at the right end. These are accepting states of the automaton and indicate that a handle has been found; the item in the state tells us how to reduce the handle. The third thing to note about Figure 9.15 is its similarity to the recursive transition network representation of Section 2.8.

We shall now run this NFA on the sentential form `E-n-n$`, to see how it works. As in the FS case we can do so if we are willing to go through the trouble of resolving the non-determinism on the fly. The automaton starts at the station `•S` and can immediately make ε-moves to `S→•E$`, `•E`, `E→•E-T`, `E→•T`, `•T`, `T→•n` and `T→•(E)`. Moving over the `E` reduces the set of items to `S→E•$` and `E→E•-T`; moving over the next `-` brings us at `E→E-•T` from which ε-moves lead to `•T`, `T→•n` and `T→•(E)`. Now the move over `n` leaves only one item: `T→n•`. Since this is a reduce item, we have found a handle segment, `n`, and we should reduce it to `T` using `T→n`. See Figure 9.16. This reduction gives us a new sentential form, `E-T-n$`, on which we can repeat the process.

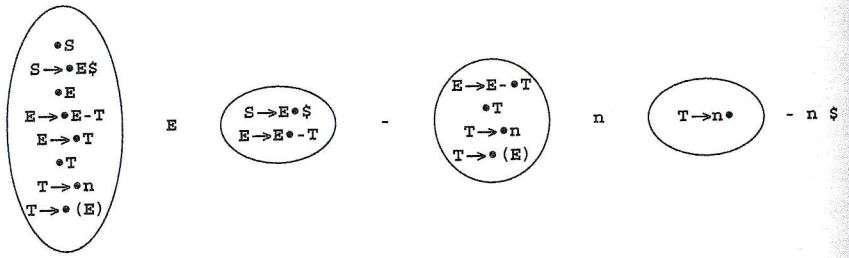

Fig. 9.16. The sets of NFA states while analysing `E-n-n$`

We see that there are two ways in which new items are produced: through ε-moves and through moving over a symbol. The first way yields items of the form $A \rightarrow \bullet\alpha$, and such an item derives from an item of the form $X \rightarrow \beta\bullet A\gamma$ in the same state. The second way yields items of the form $A \rightarrow \alpha\sigma\bullet\beta$ where σ is the token we moved over; such an item derives from an item of the form $A \rightarrow \alpha\bullet\sigma\beta$ in the parent state.

Just as in the [illegible] case (page 144) we have described an interpreter for the non-deterministic handle recognizer, and the first thing we need to do is to make the NFA deterministic, if we are to use this parsing method in earnest. We use the subset construction of Section 5.3.1 to construct a deterministic automaton with the sets of the items of Figure 9.15 as its states. The result is shown in Figure 9.17, where we have

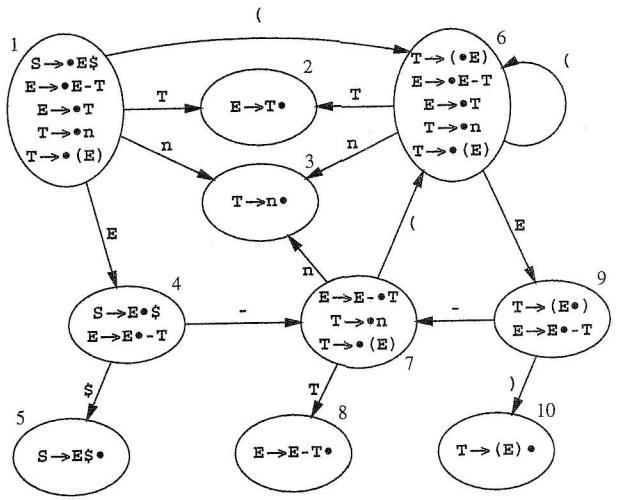

Fig. 9.17. The corresponding deterministic handle recognizer

left out the stations to avoid clutter and because they are evident from the other items. We see that the deterministic automaton looks a lot less understandable than Figure 9.15; this is the price one has to pay for having determinism. Yet we see that the subset construction has correctly identified the subsets we had already constructed by hand in Figure 9.16. This type of automaton is called an *LR(0) automaton.*

9.5.2 Using the LR(0) Automaton

It is customary to number the states of the deterministic automaton, as has already been done in Figure 9.17 (the order of the numbers is arbitrary; they serve identification purposes only). Now it has become much easier to represent the sentential form with its state information, both in a program and in a drawing:

① `E` ④ `-` ⑦ `n` ③ `- n $`

The sequence ① ④ ⑦ ③ can be read from Figure 9.17 using the path `E-n`. We start with state ① on the stack and shift in symbols from the sentential form, all the while assessing the new states. As soon as an accepting state shows up on the top of the stack (and it cannot show up elsewhere on the stack) the shifting stops and a reduce is called for; the accepting state indicates how to reduce. Accepting state ③ calls for a reduction `T→n`, so our new sentential form will be `E-T-n$`.

Repeating the handle-finding process on this new form we obtain the configuration

① E ④ - ⑦ T ⑧ - n $

which shows us two things. First, the automaton has landed in state ⑧ and thus identified a new reduction, E→E-T, which is correct. Second, we see here the effect already hinted at in Figure 9.12: by restarting the automaton at the beginning of the sentential form we have done superfluous work. Up to state ⑦, that is, up to the left end of the handle T→n, nothing had changed, so we could have saved work if we had remembered the old states ①, ④, and ⑦ between the symbols in the sentential form.

This leads to the following LR(0) parsing algorithm:

1. Consult the state s on top of the stack; it is either an accepting state specifying a reduction $X \rightarrow \alpha$ or it is a non-accepting state.
   a) If s is an accepting state, unstack $|\alpha|$ pairs of symbols and states from the stack, where $|\alpha|$ is the length of the right-hand side α. The unstacked symbols constitute the children in the parse tree for X; see Figure 9.12. Next we push X onto the stack. We have now reduced α to X.
   b) If s is a non-accepting state, shift the next token from the input onto the stack.
2. The top of the stack is now a non-terminal (1a.) or terminal (1b.) symbol T, with a state u under it. Find state u in the LR(0) automaton and follow the path marked T starting from that state.
   a) If this leads to a state v, push v onto the stack.
   b) Otherwise the input is erroneous.

Two things are important about this algorithm. The first is that if we start with a consistent stack configuration (each triple of state, symbol, and state on the stack corresponds to a transition in the LR(0) automaton) the stack configuration will again be consistent afterwards. And the second is that it does the proper reductions, and thus does the parsing we were looking for.

Note that the state u exposed after a reduction can never call for another reduction: if it did, that reduction would already have been performed earlier.

We see that LR(0) parsing is performed in two steps: 1. the top state indicates an action, shift or reduce with a given rule, which is then performed; 2. a new top state is computed by going from one state through a transition to another state. It is convenient to represent an LR(0) automaton in an *ACTION table* and a *GOTO table*, both indexed by states. The GOTO table has columns indexed by symbols; the ACTION table has just one column. In step 1 we consult the ACTION table based on the state; in step 2 we index the GOTO table with a given symbol and a given state to find the new state. The LR(0) ACTION and GOTO tables for the automaton of Figure 9.17 are given in Figure 9.18.

Suppose we find state 6 on top of the stack and the next input token is n. The ACTION table tells us to shift, and then the GOTO table, at the intersection of 6 and n, tells us to stack the state 3. And the ACTION table for state 3 tells us to reduce

	ACTION
1	shift
2	E → T
3	T → n
4	shift
5	S → E $
6	shift
7	shift
8	E → E - T
9	shift
10	T → (E)

	GOTO						
	n	-	(	)	$	E	T
1	3	e	6	e	e	4	2
2							
3							
4	e	7	e	e	5		
5							
6	3	e	6	e	e	9	2
7	3	e	6	e	e		8
8							
9	e	7	e	10	e		
10							

Fig. 9.18. LR(0) ACTION and GOTO tables for the grammar of Figure 9.14

using T→n. An entry "e" means that an error has been found: the corresponding symbol cannot legally appear in that position. A blank entry will never even be consulted: either the state calls for a reduction or the corresponding symbol will never at all appear in that position, regardless of the form of the input. In state 4, for example, we will never meet an E: the E would have originated from a previous reduction, but no reduction would do that in that position. Since non-terminals are only put on the stack in legal places no empty entry on a non-terminal will ever be consulted.

In practice the ACTION entries for reductions do not directly refer to the rules to be used, but to the numbers of these rules. These numbers are then used to index an array of routines that have built-in knowledge of the rules, that know how many entries to unstack and that perform the semantic actions associated with the recognition of the rule in question. Parts of these routines will be generated by a parser generator. Also, the reduce and shift information is combined in one table, the *ACTION/GOTO table*, with entries of the forms "sN", "rN" or "e". An entry "sN" means "shift the input symbol onto the stack and go to state N", which is often abbreviated to "shift to N". An entry "rN" means "reduce by rule number N"; the shift over the resulting non-terminal has to be performed afterwards. And "e" means error, as above. The ACTION/GOTO table for the automaton of Figure 9.17 is given in Figure 9.19.

Tables like in Figures 9.18 and 9.19 contain much empty space and are also quite repetitious. As grammars get bigger, the parsing tables get larger and they contain progressively more empty space and redundancy. Both can be exploited by data compression techniques and it is not uncommon that a table can be reduced to 15% of its original size by the appropriate compression technique. See, for example, Al-Hussaini and Stone [67] and Dencker, Dürre and Heuft [338].

The advantages of LR(0) over precedence and bounded-right-context are clear. Unlike precedence, LR(0) immediately identifies the rule to be used for reduction, and unlike bounded-right-context, LR(0) bases its conclusions on the entire left context rather than on the last m symbols of it. In fact, LR(0) can be seen as a clever implementation of BRC(∞,0), i.e., bounded-right-context with unrestricted left context and zero right context.

	n	-	(	)	$	E	T
1	s3	e	s6	e	e	s4	s2
2	r3	r3	r3	r3	r3	r3	r3
3	r4	r4	r4	r4	r4	r4	r4
4	e	s7	e	e	s5		
5	r1	r1	r1	r1	r1	r1	r1
6	s3	e	s6	e	e	s9	s2
7	s3	e	s6	e	e		s8
8	r2	r2	r2	r2	r2	r2	r2
9	e	s7	e	s10	e		
10	r5	r5	r5	r5	r5	r5	r5

Fig. 9.19. The ACTION/GOTO table for the grammar of Figure 9.14

9.5.3 LR(0) Conflicts

By now the reader may have the vague impression that something is wrong. On the one hand we claim that there is no known method to make a linear-time parser for an arbitrary grammar; on the other we have demonstrated above a method that seems to work for an arbitrary grammar. An NFA as in Figure 9.15 can certainly be constructed for any grammar, and the subset construction will certainly turn it into a deterministic one, which will definitely not require more than linear time. Voilà, a linear-time parser.

The problem lies in the accepting states of the deterministic automaton. An accepting state may still have an outgoing arrow, say on a symbol +, and if the next symbol is indeed a +, the state calls for both a reduction and for a shift: the combination of automaton and interpretation of the accepting states is not really deterministic after all. Or an accepting state may be an honest accepting state but call for two different reductions. The first problem is called a *shift/reduce conflict* and the second a *reduce/reduce conflict*. Figure 9.20 shows examples (which derive from a slightly different grammar than in Figure 9.14).

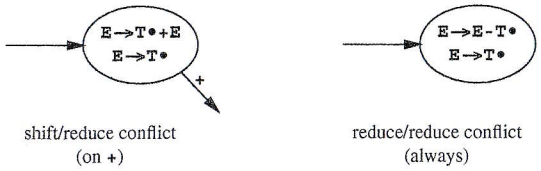

Fig. 9.20. Two types of conflict

Note that there cannot be a shift/shift conflict. A shift/shift conflict would imply that two different arrows leaving the same state would carry the same symbol. This is, however, prevented by the subset algorithm (which would have made into one the two states the arrows point to).

A state that contains a conflict is called an *inadequate state*. A grammar that leads to a deterministic LR(0) automaton with no inadequate states is called *LR(0)*. The absence of inadequate states in Figure 9.17 proves that the grammar of Figure 9.14 is LR(0).

9.5.4 ε-LR(0) Parsing

Many grammars would be LR(0) if they did not have ε-rules. The reason is that a grammar with a rule $A \to \varepsilon$ cannot be LR(0): from any station $P \to \cdots \bullet A \cdots$ an ε-arrow leads to a state $A \to \bullet$ in the non-deterministic automaton, which causes a DFA state containing both the shift item $P \to \cdots \bullet A \cdots$ and the reduce item $A \to \bullet$. And this state is inadequate, since it exhibits a shift/reduce conflict. We shall now look at a partial solution to this obstacle: ε-LR(0) parsing.

The idea is to do the ε-reductions required by the reduce part of the shift/reduce conflict already while constructing the DFA. Normally reduces cannot be precomputed since they require the first few top elements of the parsing stack, but obviously that problem does not exist for ε-reductions.

The grammar of Figure 9.21, a variant of the one in Figure 7.17, contains an ε-rule and hence is not LR(0). (The ε-rule is intended to represent multiplication.) The

$$
\begin{array}{lcl}
S_S & \to & \mathtt{E\ \$} \\
\mathtt{E} & \to & \mathtt{E\ Q\ F} \\
\mathtt{E} & \to & \mathtt{F} \\
\mathtt{F} & \to & \mathtt{a} \\
\mathtt{Q} & \to & \mathtt{/} \\
\mathtt{Q} & \to & \varepsilon
\end{array}
$$

Fig. 9.21. An ε-LR(0) grammar

start item S→•E$ leads to E→•EQF by an ε-move, and from there to E→E•QF by a move over E. This item has two ε-moves, to Q→•/ and to Q→•; the second causes a shift/reduce conflict. Following the above plan, we apply the offending rule to the item E→E•QF, but the resulting item cannot be E→EQ•F, for two reasons. First, the same item would result from finding a / in the input; and second, there is no corresponding Q on the parsing stack. So we mark the Q in the new item with a stroke on top: $\bar{\mathtt{Q}}$, to indicate that it does not correspond to a Q on the parse stack, a kind of non-Q.

We can now remove the item Q→• since it has played its part; the shift/reduce conflict is gone, and a deterministic handle recognizer results. This means that the grammar is ε-*LR(0)*; the deterministic handle recognizer is shown in Figure 9.22. The endangered state is state 4; the state that would result in "normal" LR(0) parsing is also shown, marked 4✘. We see that the immediate reduction Q→ε and the subsequent shift over Q have resulted in an item F→•a that is not present in the pure LR(0) state 4✘.

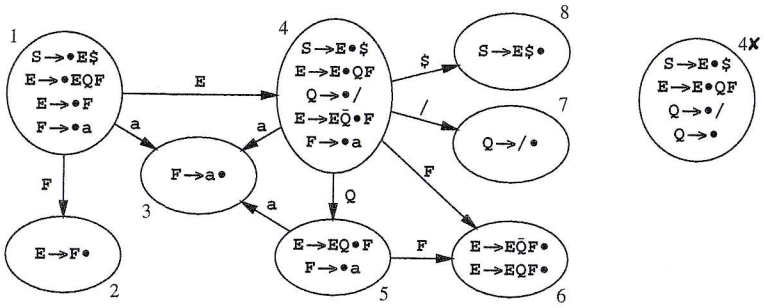

Fig. 9.22. Deterministic ε-LR(0) automaton for the grammar of Figure 9.21

In addition to the ε-reductions during parser table construction, ε-LR(0) parsing has another feature: when constructing the states of the deterministic handle recognizer, items that differ only in the presence or absence of bars over non-terminals are considered equal. So while the transition over **F** from state 4 yields an item **E→EQ̄F•** and that from state 5 yields **E→EQF•**, both transitions lead to state 6, which contains both items.

This feature has two advantages and one problem. The first advantage is that with this feature more grammars are ε-LR(0) than without it, although this plays no role in our example. The second is that the semantics of a single rule, the **E→EQF** in our example, is not split up over several items.

The problem is of course that we now have a reduce/reduce conflict. This problem is solved dynamically — during parsing — by checking the parse stack. If it contains

① **E** ④ **Q** ⑤ **F** ⑥ ...

we know the **Q** was there; we unstack 6 elements, perform the semantics of **E→EQF**, and push an **E**. If the parse stack contains

① **E** ④ **F** ⑥ ...

we know the **Q** was not there; we unstack 2 elements, create a node for **Q→**ε, unstack 2 more elements, perform the semantics of **E→EQF**, and push an **E**. Note that this modifies the basic behavior of the LR automaton, and it could thus be argued that ε-LR(0) parsing actually is not an LR technique.

Besides allowing grammars to be handled that would otherwise require much more complicated methods, ε-LR(0) parsing has the property that the non-terminals on the stack all correspond to non-empty segments of the input. This is obviously good for efficiency, but also very important in some more advanced parsing methods, for example generalized LR parsing (Section 11.1.4).

For more details on ε-LR(0) parsing and the related subject of hidden left recursion see Nederhof [156, Chapter 4], and Nederhof and Sarbo [94]. These also supply examples of grammars for which combining items with different bar properties is beneficial.

9.5.5 Practical LR Parse Table Construction

Above we explained the construction of the deterministic LR automaton (for example Figure 9.17) as an application of the subset algorithm to the non-deterministic LR automaton (Figure 9.15), but most LR parser generators (and many textbooks and papers) follow more closely the process indicated in Figure 9.16. This process combines the creation of the non-deterministic automaton with the subset algorithm: each step of the algorithm creates a transition $u \xrightarrow{t} v$, where u is an existing state and v is a new or old state. For example, the first step in Figure 9.16 created the transition ① $\xrightarrow{\mathbf{E}}$ ④. In addition the algorithm must do some bookkeeping to catch duplicate states. The LR(0) version works as follows; other LR parse table construction algorithms differ only in details.

The algorithm maintains a data structure representing the deterministic LR handle recognizer. Several implementations are possible, for example a graph like the one in Figure 9.17. Here we will assume it to consist of a list of pairs of states (item sets) and numbers, called S, and a set of transitions T. S represents the bubbles in the graph, with their contents and numbers; T represents the arrows. The algorithm also maintains a list U of numbers of new, unprocessed LR states. Since there is a one-to-one correspondence between states and state numbers we will use them interchangeably.

The algorithm starts off by creating a station $\bullet A$, where A is the start symbol of the grammar. This station is expanded, the resulting items are wrapped into a state numbered 1, the state is inserted into S, and its number is inserted in U. An item or a station I is expanded as follows:

1. If the dot is in front of a non-terminal A in I, create items of the form $A \rightarrow \bullet \cdots$ for all grammar rules $A \rightarrow \cdots$; then expand these items recursively until no more new items are created. The result of expanding I is the resulting item set; note that this is a set, so there are no duplicates. (This implements the ε-transitions in the non-deterministic LR automaton.)
2. If the dot is not in front of a non-terminal in I, the result of expanding I is just I.

The LR automaton construction algorithm repeatedly removes a state u from the list U and processes it by performing the following actions on it for all symbols (terminals and non-terminals) t in the grammar:

1. An empty item set v is created.
2. The algorithm finds items of the form $A \rightarrow \alpha \bullet t \beta$ in u. For each such item a new item $A \rightarrow \alpha t \bullet \beta$ is created, the kernel items. (This implements the vertical transitions in the non-deterministic LR automaton.) The created items are expanded as described above and the resulting items are inserted in v.
3. If state v is not already present in S, it is new and the algorithm adds it to U. Then v is added to S and the transition $u \xrightarrow{t} v$ is added to T. Here u was already present in S; the transition is certainly new to T; and v may or may not be new to S. Note that v may be empty; it is then the error state.

Since the above algorithm constructs all transitions, even those to error states, it builds a complete automaton (page 152).

The algorithm terminates because the work to be done is extracted from the list U, but only states not processed before are inserted in U. Since there are only a finite number of states, there must come a moment that there are no new states any more, after which the list U will become empty. And since the algorithm only creates states that are reachable and since only a very small fraction of all states are reachable, that moment usually arrives very soon.

9.6 LR(1)

Our initial enthusiasm about the clever and efficient LR(0) parsing technique will soon be damped considerably when we find out that very few grammars are in fact LR(0). If we drop the `$` from rule 1 in the grammar of Figure 9.14 since it does not really belong in arithmetic expressions, we find that the grammar is no longer LR(0). The new grammar is given in Figure 9.23, the non-deterministic automaton in Figure 9.24, and the deterministic one in Figure 9.25. State 5 has disappeared, since it was reached by a transition on `$`, but we have left the state numbering intact to facilitate comparison; a parser generator would of course number the states consecutively.

```
1. S → E
2. E → E - T
3. E → T
4. T → n
5. T → ( E )
```

Fig. 9.23. A non-LR(0) grammar for differences of numbers

When we inspect the new LR(0) automaton, we observe to our dismay that state 4 (marked ✘) is now inadequate, exhibiting a shift/reduce conflict on `-`, and the grammar is not LR(0). This is all the more vexing as this is a rather stupid inadequacy: `S→E•` can never occur in front of a `-` but only in front of a `#`, the end-of-input marker, so there is no real problem at all. If we had developed the parser by hand, we could easily test in state 4 if the symbol ahead was a `-` or a `#` and act accordingly (or else there was an error in the input). Since, however, practical parsers have hundreds of states, such manual intervention is not acceptable and we have to find algorithmic ways to look at the symbol ahead.

Taking our cue from the explanation of the Earley parser,[1] we attach to each dotted item a look-ahead symbol. We shall separate the look-ahead symbol from the item by a space rather than enclose it between `[]`s as we did before, to avoid visual

[1] Actually LR parsing was invented (Knuth [52, 1965]) before Earley parsing (Earley [14, 1970]).

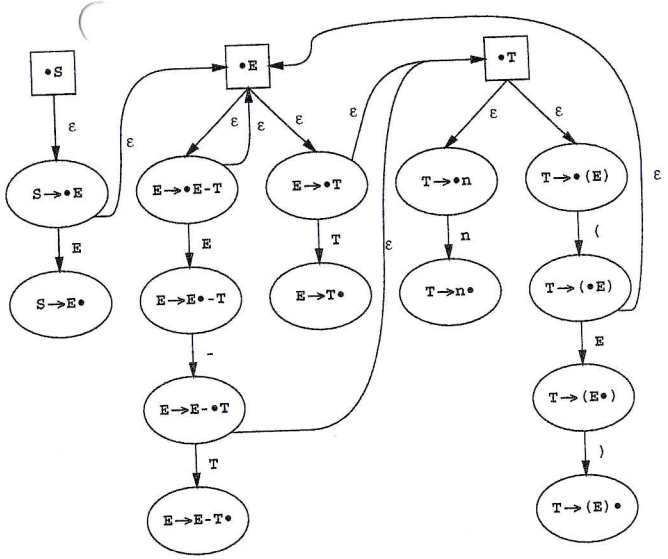

Fig. 9.24. NFA for the grammar in Figure 9.23

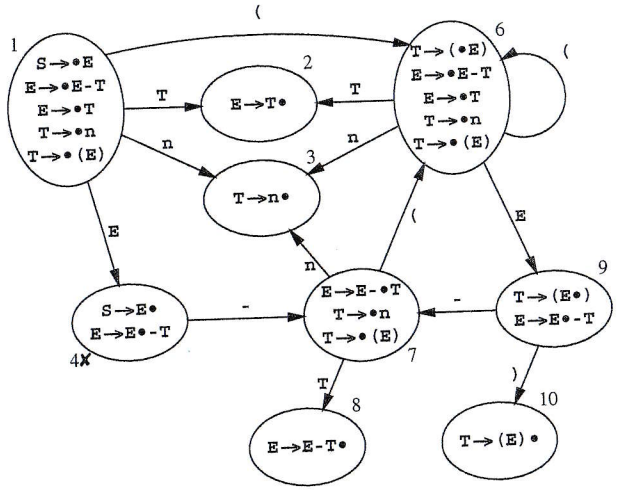

Fig. 9.25. Inadequate LR(0) automaton for the grammar in Figure 9.23

clutter. The construction of a non-deterministic handle-finding automaton using this kind of item, and the subsequent subset construction yield an LR(1) parser.

We shall now examine Figure 9.26, the NFA. Like the items, the stations have to carry a look-ahead symbol too. Actually, a look-ahead symbol in a station is more natural than that in an item: a station like `•E #` just means hoping to see an `E` followed by a `#`. The parser starts at station `•S #`, which has the end marker `#` as its look-ahead. From it we have ε-moves to all production rules for `S`, of which

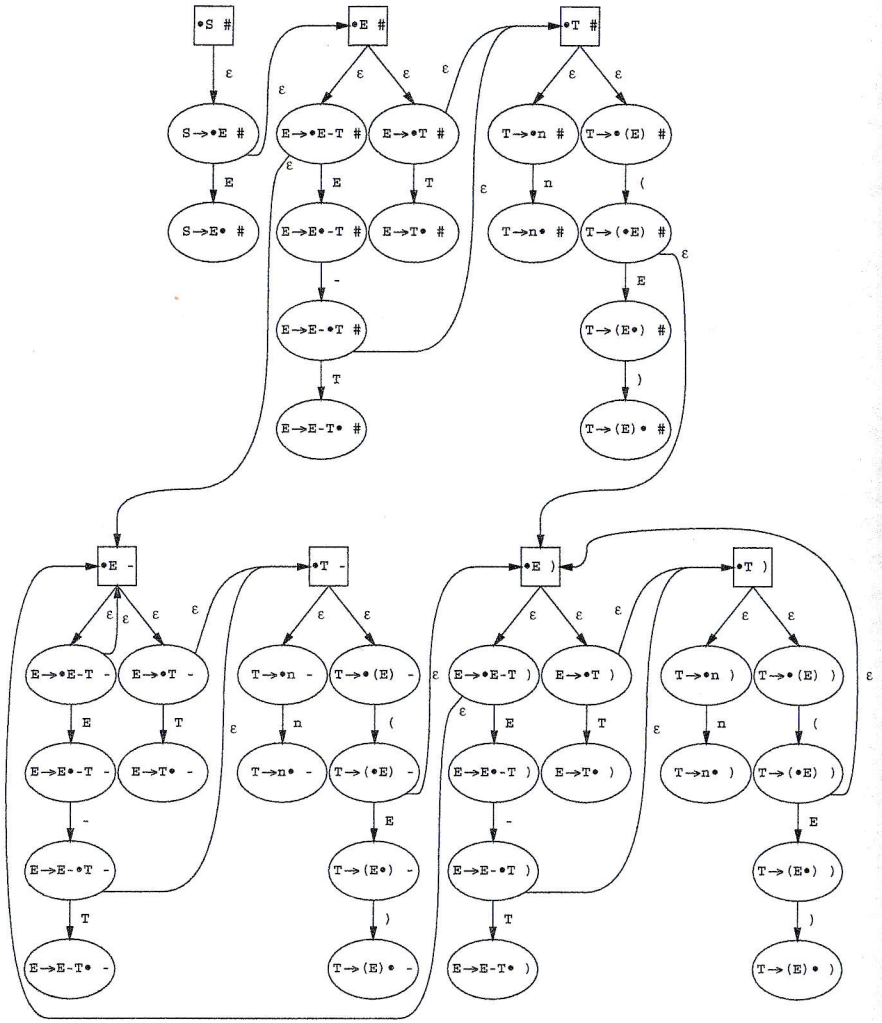

Fig. 9.26. Non-deterministic LR(1) automaton for the grammar in Figure 9.23

there is only one; this yields the item **S→•E #**. This item necessitates the station **•E #**; note that we do not automatically construct all possible stations as we did for the LR(0) automaton, but only those to which there are actual moves from elsewhere in the automaton. The station **•E #** produces two items by ε-transitions, **E→•E-T #** and **E→•T #**. It is easy to see how the look-ahead propagates. The item **E→•E-T #** in turn necessitates the station **•E -**, since now the automaton can be in the state "hoping to find an **E** followed by a **-**". The rest of the automaton will hold no surprises.

Look-aheads of items are directly copied from the items or stations they derive from; Figure 9.26 holds many examples. The look-ahead of a station derives either from the symbol following the originating non-terminal:

the item **E→•E-T** leads to station **•E -**

or from the previous look-ahead if the originating non-terminal is the last symbol in the item:

the item **S→•E #** leads to station **•E #**

There is a complication which does not occur in our example. When a non-terminal is followed by another non-terminal:

$P \rightarrow \bullet QR$

there will be ε-moves from this item to all stations $\bullet Q\ y$, where for y we have to fill in all terminals in FIRST(R). This is reasonable since all these and only these symbols can follow Q in this particular item. It will be clear that this is a rich source of stations. More complications arise when the grammar contains ε-rules, for example when R can produce ε; these are treated in Section 9.6.1.

The next step is to run the subset algorithm of page 145 on this automaton to obtain the deterministic automaton; if the automaton has no inadequate states, the grammar was *LR(1)* and we have obtained an LR(1) parser. The result is given in Figure 9.27. As was to be expected, it contains many more states than the LR(0)

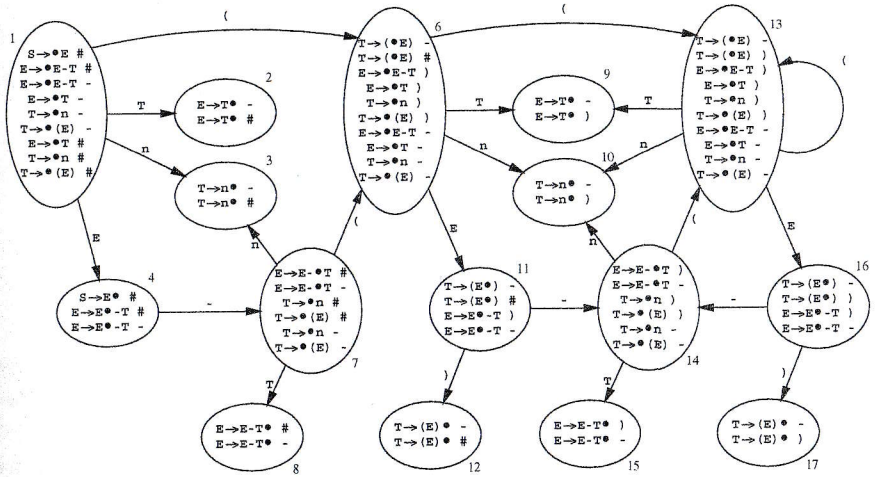

Fig. 9.27. Deterministic LR(1) automaton for the grammar in Figure 9.23

automaton although the 60% increase is very modest, due to the simplicity of the grammar. An increase of a factor of 10 or more is more likely in practice. (Although Figure 9.27 was constructed by hand, LR automata are normally created by a parser generator exclusively.)

We are glad but not really surprised to see that the problem of state 4 in Figure 9.25 has been resolved in Figure 9.27: on # reduce using S→E, on - shift to state 7 and on any other symbol give an error message.

It is again useful to represent the LR(1) automaton in an ACTION and a GOTO table; they are shown in Figure 9.28 (state 5 is missing, as explained on page 290). The combined ACTION/GOTO table can be obtained by superimposing both tables; this results in the LR(1) parsing table as it is used in practice.

	ACTION					GOTO							
	n	-	(	)	#		n	-	(	)	#	S	E	T
1	s	e	s	e	e	1	3		6			accept	4	2
2	e	r3	e	e	r3	2								
3	e	r4	e	e	r4	3								
4	e	s	e	e	r1	4		7						
6	s	e	s	e	e	6	10		13				11	9
7	s	e	s	e	e	7	3		6					8
8	e	r2	e	e	r2	8								
9	e	r3	e	r3	e	9								
10	e	r4	e	r4	e	10								
11	e	s	e	s	e	11		14		12				
12	e	r5	e	e	r5	12								
13	s	e	s	e	e	13	10		13				16	9
14	s	e	s	e	e	14	10		13					15
15	e	r2	e	r2	e	15								
16	e	s	e	s	e	16		14		17				
17	e	r5	e	r5	e	17								

Fig. 9.28. LR(1) ACTION and GOTO tables for the grammar of Figure 9.23

The sentential form E-n-n# leads to the following configuration:

① E ④ - ⑦ n ③ - n #

and since the look-ahead is -, the correct reduction T→n is indicated.

All stages of the LR(1) parsing of the string n-n-n are given in Figure 9.29. Note that state ④ in *h* causes a shift (look-ahead -) while in *l* it causes a reduce (look-ahead #).

When we compare the ACTION and GOTO tables in Figures 9.28 and 9.18, we find two striking differences. First, the ACTION table now has several columns and is indexed with the look-ahead token in addition to the state; this is as expected. What is less expected is that, second, all the error entries have moved to the ACTION table. The reason is simple. Since the look-ahead was taken into account when constructing the ACTION table, that table orders a shift only when the shift can indeed be performed, and the GOTO step of the LR parsing algorithm does not need to do checks any more: the blank entries in the GOTO table will never be accessed.

a	①	n-n-n#	shift
b	① n ③	-n-n#	reduce 4
c	① T ②	-n-n#	reduce 3
d	① E ④	-n-n#	shift
e	① E ④ - ⑦	n-n#	shift
f	① E ④ - ⑦ n ③	-n#	reduce 4
g	① E ④ - ⑦ T ⑧	-n#	reduce 2
h	① E ④	-n#	shift
i	① E ④ - ⑦	n#	shift
j	① E ④ - ⑦ n ③	#	reduce 4
k	① E ④ - ⑦ T ⑧	#	reduce 2
l	① E ④	#	reduce 1
m	① S	#	accept

Fig. 9.29. LR(1) parsing of the string n-n-n

It is instructive to see how the LR(0) and LR(1) parsers react to incorrect input, for example E-nn···. The LR(1) parser of Figure 9.28 finds the error as soon as the second n appears as a look-ahead:

① E ④ - ⑦ n ③ n···

since the pair (3,n) in the ACTION table yields "e"; the GOTO table is not even consulted. The LR(0) parser of Figure 9.18 behaves differently. After reading E-n it is in the configuration

① E ④ - ⑦ n ③ n···

where entry 3 in the ACTION table tells it to reduce by T→n:

① E ④ - ⑦ T ⑧ n···

and now entry 8 in the ACTION table tells it to reduce again, by E→E-T this time:

① E ④ n···

Only now is the error found, since the pair (4,n) in the GOTO table in Figure 9.18 yields "e".

Since the LR(0) automaton has fewer states than the LR(1) automaton, it retains less information about the input to the left of the handle; since it does not use look-ahead it uses less information about the input to the right of the handle. So it is not surprising that the LR(0) automaton is less alert than the LR(1) automaton.

9.6.1 LR(1) with ε-Rules

In Section 3.2.2 we have seen that one has to be careful with ε-rules in bottom-up parsers: they are hard to recognize bottom-up. Fortunately LR(1) parsers are strong enough to handle them without problems. In the NFA, an ε-rule is nothing special; it is just an exceptionally short list of moves starting from a station (see station •Bc in Figure 9.31(*a*)). In the deterministic automaton, the ε-reduction is possible in all

states of which the ε-rule is a member, but hopefully its look-ahead sets it apart from all other rules in those states. Otherwise a shift/reduce or reduce/reduce conflict results, and indeed the presence of ε-rules in a grammar raises the risks of such conflicts and reduces the likelihood of the grammar being LR(1).

```
S  ->  A B c
A  ->  a
B  ->  b
B  ->  ε
```

Fig. 9.30. A simple grammar with an ε-rule

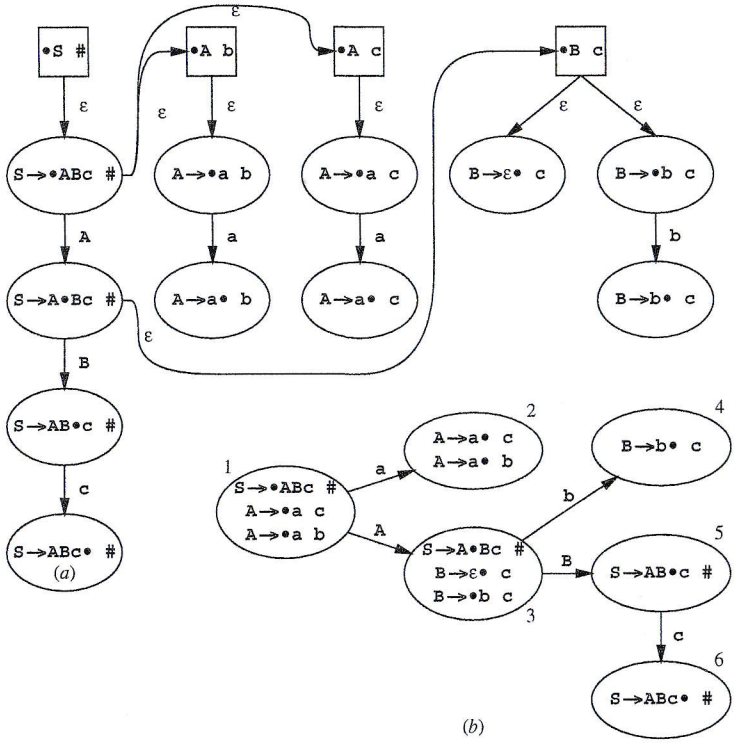

Fig. 9.31. Non-deterministic and deterministic LR(1) automata for Figure 9.30

To avoid page-filling drawings, we demonstrate the effect using the trivial grammar of Figure 9.30. Figure 9.31(*a*) shows the non-deterministic automaton, Figure 9.31(*b*) the resulting deterministic one. Note that no special actions were necessary to handle the rule `B->ε`.

The only complication occurs again in determining the look-ahead sets in rules in which a non-terminal is followed by another non-terminal; here we meet the same phenomenon as in an LL(1) parser (Section 8.2.2.1). Given an item, for example, $P \rightarrow \bullet ABC\ d$ where d is the look-ahead, we are required to produce the look-ahead set for the station $\bullet A \cdots$. If B had been a terminal, it would have been the look-ahead. Now we take the FIRST set of B, and if B produces ε (is nullable) we add the FIRST set of C since B can be transparent and allow us to see the first token of C. If C is also nullable, we may even see d, so in that case we also add d to the look-ahead set. The result of these operations can be written as FIRST(BCd). The new look-ahead set cannot turn out to be empty: the sequence of symbols from which it is derived (the BCd above) always ends in the original look-ahead set, and that was not empty.

9.6.2 LR($k > 1$) Parsing

Instead of a one-token look-ahead k tokens can be used, with $k > 1$. Surprisingly, this is not a straightforward extension of LR(1). The reason is that for $k > 1$ we also need to compute look-ahead sets for shift items. That this is so can be seen from the LR(2) grammar of Figure 9.32. It is clear that the grammar is not LR(1): the input must start

```
1.  Ss  ->  Aa | Bb | Cec | Ded
2.  A   ->  qE
3.  B   ->  qE
4.  C   ->  q
5.  D   ->  q
6.  E   ->  e
```

Fig. 9.32. An LR(2) Grammar

with a `q` but the parser cannot see if it should reduce by `C->q` (look-ahead `e`), reduce by `D->q` (look-ahead `e`), or shift over `e`. But each choice has a different two-token look-ahead set (`ec`, `ed` and {`ea`, `eb`}, respectively), so LR(2) should work.

The initial state, state 1, in the LR(2) parser for this grammar is

```
S->•Aa ##
S->•Bb ##
S->•Cec ##
S->•Ded ##
A->•qE a#
B->•qE b#
C->•q ec
D->•q ed
```

which calls for a shift over the `q`. After this shift the parser reaches a state

```
A→q•E a#
B→q•E b#
C→q• ec
D→q• ed
E→•e a#
E→•e b#
```

where we still have the same shift/reduce conflict: there are two reduce items, `C→q•` and `D→q•` with look-aheads `ec` and `ed`, and one shift item, `E→•e`, which shifts on an `e`.

The conflict goes away when we realize that for each item I two kinds of look-aheads are involved: the *item look-ahead*, the set of strings that can follow the end of I; and the *dot look-ahead*, the set of strings that can follow the dot in I. For parsing decisions it is the dot look-ahead that counts, since the dot position corresponds with the gap in an LR parser, so the dot look-ahead corresponds to the first k tokens of the rest of the input. Note that for reductions the item look-ahead seems to be the deciding factor, but since the dot is at the end in reduce items, the item look-ahead coincides with the dot look-ahead. In an LR(1) parser the dot look-ahead of a shift item I coincides with the set of tokens on which there is a shift from the state I resides in, so there is no need to compute it separately, but as we have seen above, this is not true for an LR(2) parser.

So we compute the full two-token dot look-aheads for the shift items to obtain state 2:

item with item look-ahead	dot look-ahead
`A→q•E a#`	`ea`
`B→q•E b#`	`eb`
`C→q• ec`	`ec`
`D→q• ed`	`ed`
`E→•e a#`	`ea`
`E→•e b#`	`eb`

Now the conflict is resolved since the two reduce actions and the shift action all have different dot look-aheads: shift on `ea` and `eb`, reduce to `C` on `ec`, and reduce to `D` on `ed`.

More in general, the dot look-ahead of an item $A \rightarrow \alpha \bullet \beta\ \gamma$, where γ is the item look-ahead, can be computed as $\text{FIRST}_k(\beta\gamma)$.

Parts of the ACTION and GOTO tables for the LR(2) parser for the grammar in Figure 9.32 are given in Figure 9.33. The ACTION table is now indexed by look-ahead strings of length 2 rather than by single tokens, but the GOTO table is still indexed by single symbols, since each entry in a GOTO table represents a transition in the handle-finding automaton, and transitions consume just one symbol. As a result, superimposing the two tables into one ACTION/GOTO table is no longer possible; combined ACTION/GOTO tables are a feature of LR(1) parsing only (and, with some handwaving, of LR(0)). Again all the error detection is done in the ACTION table.

ACTION

	qe	ea	eb	ec	ed	···
1	s	e	e	e	e	···
2	e	s	s	r4	r5	···
3	···					
4	···					
⋮	⋮					

GOTO

	q	a	b	c	d	e	E	···
1	2							···
2						3	4	···
3	···							
4	···							
⋮	⋮							

Fig. 9.33. Partial LR(2) ACTION and GOTO tables for the grammar of Figure 9.32

It is interesting to compare this to LR(0), where there is no look-ahead at all. There the ACTION table offers no protection against impossible shifts, and the GOTO table has to contain error entries. So we see that the LR(0), LR(1), and LR($k > 1$) table construction algorithms differ in more than just the value of k: LR(0) needs a check upon shift; LR($k > 1$) needs the computation of dot look-ahead; and LR(1) needs either but not both. It is of course possible to design a combined algorithm, but for all values of k part of it would not be activated.

However interesting LR($k > 1$) parsing may be, its practical value is quite limited: the required tables can assume gargantuan size (see, e.g., Ukkonen [66]), and it does not really help much. Although an LR(2) parser is more powerful than an LR(1) parser, in that it can handle some grammars that the other cannot, the emphasis is on "some". If a common-or-garden variety grammar is not LR(1), chances are minimal that it is LR(2) or higher.

9.6.3 Some Properties of LR(k) Parsing

Some theoretically interesting properties of varying practical significance are briefly mentioned here. It can be proved that any LR(k) grammar with $k > 1$ can be transformed into an LR($k-1$) grammar (and so to LR(1), but not always to LR(0)), often at the expense of an enormous increase in size; see for example Mickunas, et al. [407]. It can be proved that if a language allows parsing with a pushdown automaton as described in Section 3.3, it has an LR(1) grammar; such languages are called *deterministic languages*. It can be proved that if a grammar can be handled by any of the deterministic methods of Chapters 8 and 9, it can be handled by an LR(k) parser (that is, all deterministic methods are weaker than or equally strong as LR(k)). It can be proved that any LR(k) language can be obtained as a regular expression, the elements of which are LR(0) languages; see Bertsch and Nederhof [96].

LR($k \geq 1$) parsers have the immediate error detection property: they will stop at the first incorrect token in the input and not even perform another shift or reduce. This is important because this early error detection property allows a maximum amount of context to be preserved for error recovery; see Section 16.2.6. We have seen that LR(0) parsers do not have this property.

In summary, LR(k) parsers are the strongest deterministic parsers possible and they are the strongest linear-time parsers known, with the exception of some non-

canonical parsers; see Section 10. They react to errors immediately, are paragons of virtue and beyond compare, but even after 40 years they are not widely used.

9.7 LALR(1)

The reader will have sensed that our journey has not yet come to an end; the goal of a practical, powerful, linear-time parser has still not been attained completely. At their inception by Knuth in 1965 [52], it was realized that LR(1) parsers would be impractical in that the space required for their deterministic automata would be prohibitive. A modest grammar might already require hundreds of thousands or even millions of states, numbers that were totally incompatible with the computer memories of those days.

In the face of this difficulty, development of this line of parsers came to a standstill, partially interrupted by Korenjak's invention of a method to partition the grammar, build LR(1) parsers for each of the parts and combine these into a single over-all parser (Korenjak [53]). This helped, but not much, in view of the added complexity.

The problem was finally solved by using an unlikely and discouraging-looking method. Consider the LR(1) automaton in Figure 9.27 and imagine boldly discarding all look-ahead information from it. Then we see that each state in the LR(1) automaton reverts to a specific state in the LR(0) automaton; for example, LR(1) states 6 and 13 collapse into LR(0) state 6 and LR(1) states 2 and 9 collapse into LR(0) state 2. We say that LR(1) states 6 and 13 have the same *core*, the items in the LR(0) state 6, and similarly for LR(1) states 2 and 9.

There is not a single state in the LR(1) automaton that was not already present in a rudimentary form in the LR(0) automaton. Also, the transitions remain intact during the collapse: both LR(1) states 6 and 13 have a transition to state 9 on `T`, but so has LR(0) state 6 to 2. By striking out the look-ahead information from an LR(1) automaton, it collapses into an LR(0) automaton for the same grammar, with a great gain as to memory requirements but also at the expense of the look-ahead power. This will probably not surprise the reader too much, although a formal proof of this phenomenon is not trivial.

The idea is now to collapse the automaton but to keep the look-ahead information, as follows. The LR(1) state 2 (Figure 9.27) contains the items

```
E→T• -
E→T• #
```

and LR(1) state 9 contains

```
E→T• -
E→T• )
```

where the LR(0) core is

```
E→T•
```


They collapse into an LALR(1) state which corresponds to the LR(0) state 2 in Figure 9.25, but now with look-ahead:

```
E→T• #
E→T• -
E→T• )
```

The surprising thing is that this procedure preserves almost all the original look-ahead power and still saves an enormous amount of memory. The resulting automaton is called an *LALR(1) automaton*, for "Look Ahead LR(0) with a look-ahead of 1 token."

The LALR(1) automaton for our grammar of Figure 9.23 is given in Figure 9.34. The look-aheads are sets now and are shown between `[` and `]`, so state 2 is represented as `E→T• [#-)]`. We see that the original conflict in state 4 is indeed still resolved, as it was in the LR(1) automaton, but that its size is equal to that of the LR(0) automaton. Now that is a very fortunate state of affairs!

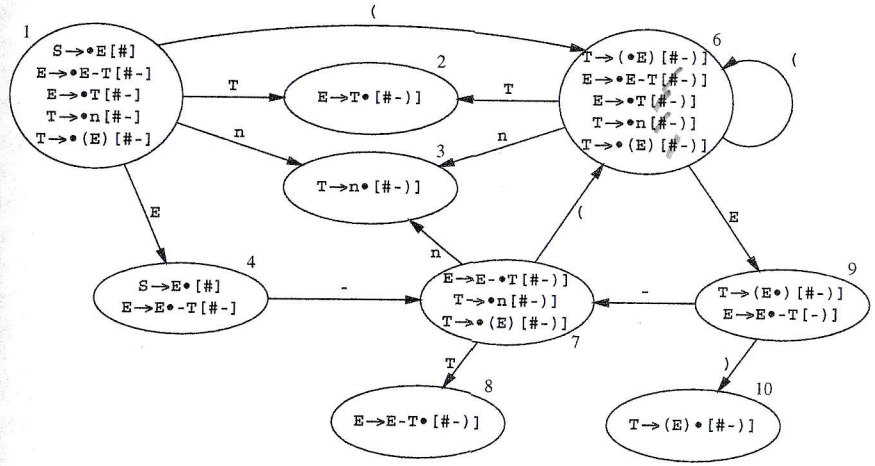

Fig. 9.34. The LALR(1) automaton for the grammar of Figure 9.23

We have finally reached our goal. LALR(1) parsers are powerful, almost as powerful as LR(1) parsers, they have fairly modest memory requirements, only slightly inferior to (= larger than) those of LR(0) parsers,[2] and they are time-efficient. LALR(1) parsing may very well be the most-used parsing method in the world today. Probably the most famous LALR(1) parser generators are *yacc* and its GNU version *bison*.

LALR(k) also exists and is LR(0) with an add-on look-ahead of k tokens. LALR(k) combines LR(0) information about the left context (in the LR(0) automa-

[2] Since the LALR(1) tables contain more information than the LR(0) tables (although they have the same size), they lend themselves slightly less well to data compression. So practical LALR(1) parsers will be bigger than LR(0) parsers.

ton) with LR(k) information about the right context (in the k look-aheads). Actually there is a complete family of *LA(k)LR(j)* parsers out there, which combines LR(j) information about the left context with LR(k) information about the right context. Like LALR(1), they can be derived from LR($j+k$) parsers in which all states with identical cores and identical first k tokens of the $j+k$-token look-ahead have coincided. So LALR(1) is actually LA(1)LR(0), Look-ahead Augmented (1) LR (0). See Anderson [55].

9.7.1 Constructing the LALR(1) Parsing Tables

When we have sufficiently drunk in the beauty of the vista that spreads before us on these heights, and start thinking about returning home and actually building such a parser, it will come to us that there is a small but annoying problem left. We have understood how the desired parser should look and also seen how to construct it, but during that construction we used the unacceptably large LR(1) parser as an intermediate step.

So the problem is to find a shortcut by which we can produce the LALR(1) parse table without having to construct the one for LR(1). This particular problem has fascinated scores of computer scientists for many years (see the references in (Web)Section 18.1.4), and several good (and some very clever) algorithms are known. On the other hand, several deficient algorithms have appeared in publications, as DeRemer and Pennello [63] and Kannapinn [99] have pointed out. (These algorithms are deficient in the sense that they do not work for some grammars for which the straightforward LR(1) collapsing algorithm does work, rather than in the sense that they would lead to incorrect parsers.)

Since LALR(1) is clearly a difficult concept; since we hope that each new LALR algorithm contributes to its understandability; and since we think some algorithms are just too interesting to skip, we have allowed ourselves to discuss four LALR(1) parsing table construction algorithms, in addition to the one above. We present 1. a very simple algorithm, which shows that constructing an LALR(1) parsing table is not so difficult after all; 2. the algorithm used in the well-known parser generator *yacc*; 3. an algorithm which creates LALR(1) by upgrading LR(0); and 4. one that does it by converting the grammar to SLR(1). This is also the order in which the algorithms were discovered.

9.7.1.1 A Simple LALR(1) Algorithm

The easiest way to keep the LALR(1) parse table small is to never let it get big. We achieve this by collapsing the states the moment they are created, rather than first creating all states and then collapsing them. We start as if we are making a full LR(1) parser, propagating look-aheads as described in Section 9.6, and we use the table building technique of Section 9.5.5. In this technique we create new states by performing transitions from existing unprocessed states obtained from a list U, and if the created state v is not already present in the list of processed states S, the algorithm adds it to U so it can be the source of new transitions.

For our LALR(1) algorithm we refine this step as follows. We check v to see if there is already a state w in S with the same core. If so, we merge v into w; if this modifies w, we put w back in U as an unprocessed state: since it has changed, it may lead to new and different states. If w was not modified, no new information has come to light and we can just extract the next unprocessed state from U; v itself is discarded in both cases. The state w keeps its number and its transitions; it is important to note that when w is processed again, its transitions are guaranteed to lead to states whose cores are already present in S and T.

The merging makes sure that the cores of all states in S are always different, as they should be in an LALR(1) parser; so never during the process will the table be larger than the final LALR(1) table. And by putting all modified states back into the list to be processed we have ensured that all states with their proper LALR(1) look-aheads will be found eventually. This surprisingly simple algorithm was first described by Anderson et al. [56] in 1973.

The algorithm is not ideal. Although it solves the main problem of LALR(1) parse table generation, excessive memory use, it still generates almost all LR(1) states, of which there are many more than LALR(1) states. The only situation in which we gain time over LR(1) parse table generation is when merging the created state v into an existing state w does not modify w. But usually v will bring new look-aheads, so usually w will change and will then be reprocessed. Computer scientists, especially compiler writers, felt the need for a faster LALR(1) algorithm, which led to the techniques described in the following three sections.

9.7.1.2 The Channel Algorithm

The well-known parser generator *yacc* uses an algorithm that is both intuitively relatively clear and reasonably efficient (Johnson [361]); it is described in more detail by Aho, Sethi and Ullman in [340]. The algorithm does not seem to have a name; we shall call it the *channel algorithm*.

We again use the grammar of Figure 9.23, which we now know is LALR(1) (but not LR(0)). Since we want to do look-ahead but do not yet know what to look for, we use LR(0) items extended with a yet unknown look-ahead field, indicated by an empty square; an example of an item would be `A→bC•De` □. Using such items, we construct the non-deterministic LR(0) automaton in the usual fashion; see Figure 9.35. Now suppose that we were told by some oracle what the look-ahead set of the item `S→•E` □ is (first column, second row in Figure 9.35); call this look-ahead set L. Then we could draw a number of conclusions. The first is that the item `S→E•` □ also has L. The next is that the look-ahead set of the station `•E`□ is also L, and from there L spreads to `E→•E-T`, `E→E•-T`, `E→E-•T`, `E→E-T•`, `E→•T` and `E→T•`. From `E→E-•T` and `E→•T` it flows to the station `•T` and from there it again spreads on.

The flow possibilities of look-ahead information from item to item once it is known constitute "channels" which connect items. Each channel connects two items and is one-directional. There are two kinds of channels. From each station channels run down to each item that derives from it; these channels propagate input from

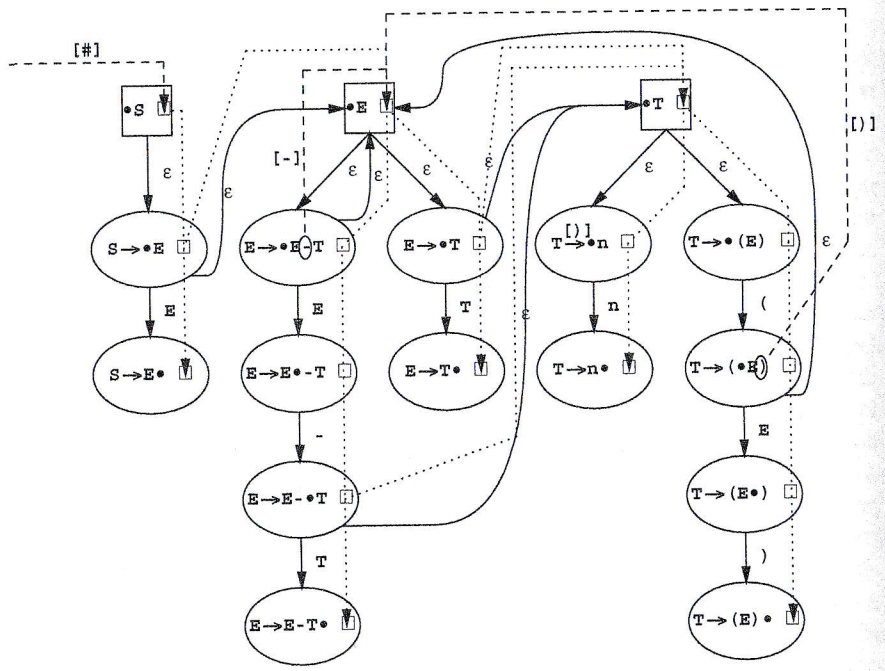

Fig. 9.35. Non-deterministic automaton with channels

elsewhere. From each item that has the dot in front of a non-terminal A, a channel runs parallel to the ε-arrow to the station $\bullet A\square$. If A is the last symbol in the right-hand side, the channel propagates the look-ahead of the item it starts from. If A is not the last symbol, but is followed by, for example, CDe (so the entire item would be something like $P \rightarrow B\bullet ACDe\ \square$), the input to the channel is FIRST(CDe); such input is said to be "generated spontaneously", as opposed to "propagated" input.

Figure 9.35 shows the full set of channels: those carrying propagated input as dotted lines, and those carrying spontaneous input as dashed lines, with their spontaneous input sets. A channel from outside introduces the spontaneous look-ahead #, the end-of-input marker, to the station(s) of the start symbol. The channel set can be represented in a computer as a list of input and output ends of channels:

Input end	leads to	output end	Remarks
[#]	==>	•S□	spontaneous
•S□	==>	S→•E □	propagated
S→•E □	==>	S→E• □	propagated
S→•E □	==>	•E □	propagated
	...		
[-]	==>	•E □	spontaneous
	...		

Next we run the subset algorithm on this (channeled) NFA in slow motion and watch carefully where the channels go. This procedure severely taxes the human

brain; a more practical way is to just construct the deterministic automaton without concern for channels and then use the above list (in its complete form) to re-establish the channels. This is easily done by finding the input and output end items and stations in the states of the deterministic automaton and constructing the corresponding channels. Note that a single channel in the NFA can occur many times in the deterministic automaton, since items can (and will) be duplicated by the subset construction. The result can best be likened to a bowl of mixed spaghetti and tagliatelli (the channels and the transitions) with occasional chunks of ham (the item sets) and will not be printed in this book.

Now we are close to home. For each channel we pump its input to the channel's end. First this will only have effect for channels that have spontaneous input: a # will flow in state 1 from item `S→•E[□]` to station `•E[□]`, which will then read `•E[#]`; a - from `E→•E-T[□]` flows to the `•E[□]`, which changes to `•E[-]`; etc. We go on pumping until all look-ahead sets are stable and nothing changes any more. We have now obtained the LALR(1) automaton and can discard the channels; of course we keep the transitions. This is an example of a transitive closure algorithm.

It is interesting to look more closely at state 4 (see Figure 9.34) and to see how `S→E•[#]` gets its look-ahead which excludes the `-`, although the `-` is present in the look-ahead set of `E→E•-T[#-]` in state 4. To this end, a magnified view of the top left corner of the full channeled LALR(1) automaton is presented in Figure 9.36; it comprises the states 1 to 4. Again channels with propagated input are dotted, those with spontaneous input are dashed and transitions are drawn. We can now see more clearly that `S→E•[#]` derives its look-ahead from `S→•E[#]` in 1, while `E→E•-T[#-]` derives its look-ahead (indirectly) from `•E[-]` in state 1. This item has a look-ahead `-` generated spontaneously in `E→•E-T[□]` in state 1. The channel from `S→•E[#]` to `•E[#-]` only works "downstream", which prevents the `-` from flowing back. LALR(1) parsers often give one the feeling that they succeed by a narrow margin!

If the grammar contains ε-rules, the same complications arise as in Section 9.6.1 in the determination of the FIRST set of the rest of the right-hand side: when a non-terminal is nullable we have to also include the FIRST set of what comes after it, and so on. We meet a special complication if the entire rest of the right-hand side can be empty: then we may see the look-ahead □, which we do not know yet. In fact this creates a third kind of channel that has to be watched in the subset algorithm. We shall not be so hypocritical as to suggest the construction of the LALR(1) automaton for the grammar of Figure 9.30 as an exercise to the reader, but we hope the general principles are clear. Let a parser generator do the rest.

9.7.1.3 LALR(1) by Upgrading LR(0)

The above techniques basically start from an LR(1) parse table, explicit or implicit, and then shrink it until the items are LR(0): they downgrade the LR(1) automaton to LALR(1). It is also possible to start from the LR(0) automaton, find the conflicts in it, and upgrade from there. This leads to a complicated but very efficient algorithm,

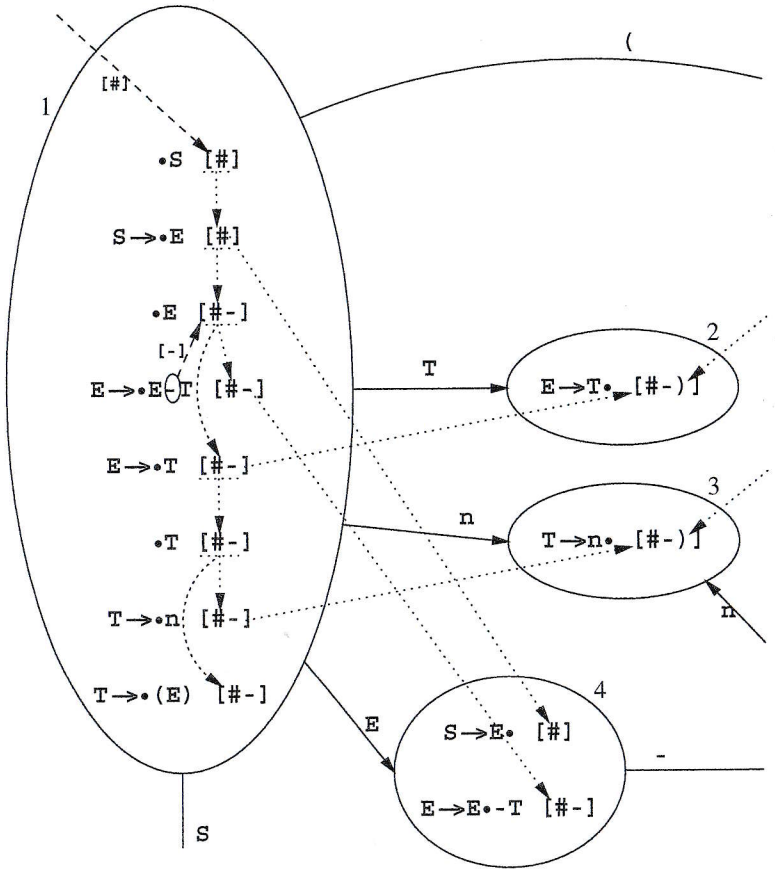

Fig. 9.36. Part of the deterministic automaton with channels (magnified cut)

designed by DeRemer and Pennello [63]. Again it has no name; we shall call it the *relations algorithm*, for reasons that will become clear.

Upgrading the inadequate LR(0) automaton in Figure 9.25 is not too difficult. We need to find the look-ahead(s) we are looking at in the input when we are in state 4 and reducing by **S→E** is the correct action. That means that the stack must look like

··· **E** ④

Looking back through the automaton, we can see that we can have come from one state only: state 1:

① **E** ④

Now we do the reduction because we want to see what happens when that is the correct action:

① **S**

and we see that we have reduced to **S**, which has only one look-ahead, **#**, the end-of-input token. So the reduce look-ahead of the item **S→E•** in state 4 is **#**, which differs from the shift look-ahead **-** for **E→E•-T**, so the conflict is resolved.

This is an example of a more general technique: to find the look-ahead(s) of an inadequate reduce item in an LR(0) automaton, we take the following steps:

- we assume that the implied reduction R is the proper action and simulate its effects on an imaginary stack;
- we simulate all possible further movements of the LR(0) automaton until the automaton is about to shift in the next token, t;
- we add t to the look-ahead set, since it has the property that it will be shifted and accepted if we do the reduction R when we see it as the next token in the input.

It will be clear that this is a very reasonable method of collecting good look-ahead sets. It is much less clear that it produces the same LALR look-ahead sets as the LALR algorithms above, and for a proof of that fact we refer the reader to DeRemer and Pennello's paper.

Turning the above ideas into an algorithm requires some serious effort. We will follow DeRemer and Pennello's explanation closely, using the same formalism and terminology as much as is convenient. The explanation uses an unspecified grammar of which only two rules are important: $A \rightarrow \omega$ and $B \rightarrow \beta A \gamma$, for some, possibly empty sequences of non-terminals and terminals ω, β, and γ. Refer to Figure 9.37.

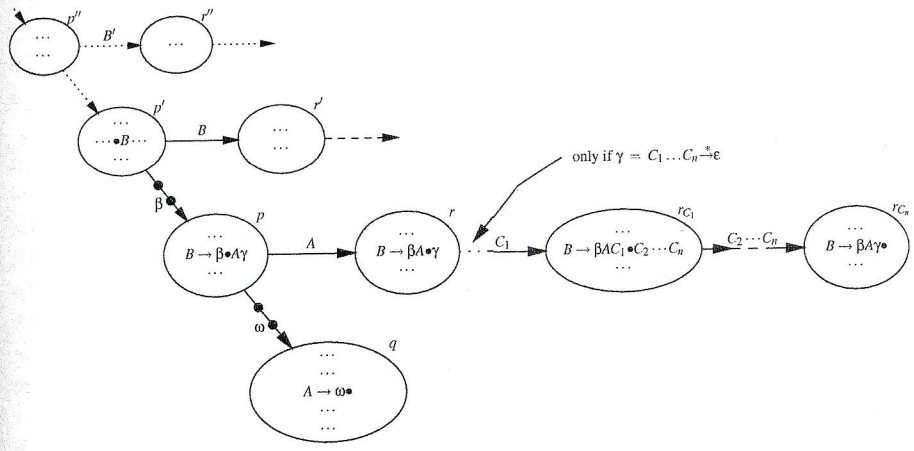

Fig. 9.37. Hunting for LALR(1) look-aheads in an LR(0) automaton — the **lookback** and **includes** relations

Suppose the LR(0) automaton has an inadequate state q with a reduce item $A \rightarrow \omega\bullet$, and we want to know the LALR look-ahead of this item. If state q is on the top of the stack, there must be a path through the LR(0) automaton from the start state 1 to q (or we would not have ended up in q), and the last part of this path spells ω (or we would not be able to reduce by $A \rightarrow \omega$). We can follow this path back to the

beginning of ω; this leads us to the state p, where the present item $A \to \omega\bullet$ originated. There are two things to note here: there may be several different paths back that spell ω, leading to several different ps; and ω may be ε, in which case p is equal to q. For simplicity Figure 9.37 shows one p only.

We have now established that the top segment of the stack is $p\ \omega_1 \cdots \omega_n\ q$, where p is one of the ps identified above and $\omega_1 \cdots \omega_n$ are the components of ω. We can now do the simulated reduction, as we did above. This shortens the stack to $p\,A$, and we have to shift over the A, arriving at a state r.

More formally, a reduce item $A \to \omega\bullet$ in an LR(0) state q identifies a set of transitions $\{p_1 \xrightarrow{A} r_1, \ldots, p_n \xrightarrow{A} r_n\}$, where for all p_i we have $p_i \xrightarrow{\omega} q$. This defines the so-called **lookback** relation between a pair (state, reduce item) and a transition. One writes $(q,\ A \to \omega\bullet)$ **lookback** $(p_i \xrightarrow{A} r_i)$ for $1 \le i \le n$. This is step 1 of the simulation. Note that this is a relation, not an algorithm to compute the transition(s); it just says that given a pair (state, reduce item) and a transition, we can check if the "lookback" relation holds between them. (DeRemer and Pennello write a transition $(p_i \xrightarrow{A} r_i)$ as (p_i,A), since the r follows directly from the LR(0) automaton, which is deterministic.)

The shift from p over A is guaranteed to succeed, basically because the presence of an item $A \to \omega\bullet$ in q combined with the existence of a path ω from q leading back to p proves that p contains an item that has a dot in front of an A. That $\bullet A$ causes both the ω path and the transition $p \xrightarrow{A} r$ (except when A is the start symbol, in which case we are done and the look-ahead is #). The general form of such an item is $B \to \beta\bullet A\gamma$, as shown in Figure 9.37. Here we have the first opportunity to see some look-ahead tokens: any terminal in FIRST(γ) will be an LALR look-ahead token for the reduce item $A \to \omega\bullet$ in state q. But the simulation is not finished yet, since γ may be or produce ε, in which case we will also have to look past the item $B \to \beta A\bullet\gamma$.

If γ produces ε, it has to consist of a sequence of non-terminals $C_1 \cdots C_n$, each capable of producing ε. This means that state r contains an item $C_1 \to \bullet$, which is immediately a reduce item; see a similar phenomenon in state 3 in Figure 9.31. Its presence will certainly make r an inadequate state, but, if the grammar is LALR(1), that problem will be solved when the algorithm treats the item $C_1 \to \bullet$ in r. For the moment we assume the problem is solved; we do the reduction, push C_1 on the simulated stack, and shift over it to state r_{C_1}. We repeat this process until we have processed all $C_1 \cdots C_n$, and by doing so reach a state r_{C_n} which contains a reduce item $B \to \beta A\gamma\bullet$.

Now it is tempting to say that any look-ahead of this item will also figure in the look-ahead that we are looking for, but that is not true. At this point in our simulation the stack contains $p\ A\ r\ C_1\ r_{C_1}\ \cdots\ C_n\ r_{C_n}$, so we see only the look-aheads of those items $B \to \beta A\gamma\bullet$ in state r_{C_n} that have reached that state through p! State r_{C_n} may be reachable through other paths, which may quite well bring in other look-aheads for the reduce item $B \to \beta A\gamma\bullet$ which do not belong in the look-ahead set of $A \to \omega$. So to simulate the reduction $B \to \beta A\gamma$ we walk the path γ back through the LR(0) automaton to state p, all the while removing C_is (components of γ) from the stack. Then from state p backwards we can freely find all paths that spell β, to reach all

states p'_i that contain the item $B \to \bullet\beta A\gamma$. Each of these states p' has a transition on B, for the same reasons p had a transition on A (again except when B is the start symbol). The transition over B leads to a state r', which brings us back to a situation similar to the one at p.

This process defines the so-called **includes** relation: $(p \xrightarrow{A} r)$ **includes** $(p' \xrightarrow{B} r')$ if and only if the grammar contains a rule $B \to \beta A\gamma$, and $\gamma \xrightarrow{*} \varepsilon$, and $p' \xrightarrow{\beta} p$. Note that one $(p \xrightarrow{A} r)$ can include several $(p' \xrightarrow{B} r')$s, when several paths β are possible.

To simulate all possible movements of the LR(0) automaton and find all the transitions that lead to states that contribute to the look-ahead of $A \to \omega\bullet$ in state q, we have to repeat the step from p to p' for successive p'', p''', ..., until we find no new ones any more or until we are stopped by reaching a reduction of the start symbol. This is step 2 of the simulation.

Any token t that can be shifted over in any of the states r, r', ... thus reached, belongs in the look-ahead of $A \to \omega\bullet$ in state q, since we have just shown that after the reduction $A \to \omega$ and possibly several other reductions, we arrive at a state in which a shift over t is possible. And no other tokens belong in the look-ahead set, since they will not allow a subsequent shift, and would get the parser stuck.

So we are interested in the terminal transitions of the states r, r', ... To describe them in a framework similar to the one used so far, we define a relation **directly-reads** as follows; refer to Figure 9.38. A transition $(p \xrightarrow{A} r)$ **directly-reads** t if r has

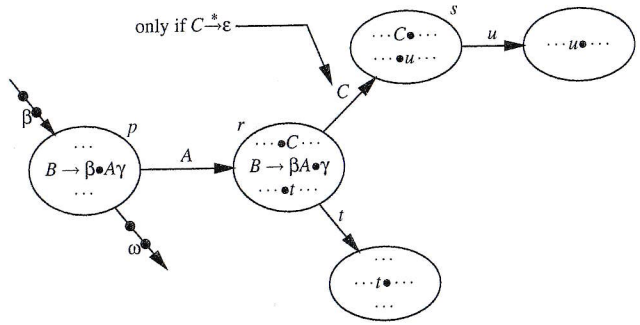

Fig. 9.38. Hunting for LALR(1) look-aheads in an LR(0) automaton — the **directly-reads** and **reads** relations

an outgoing arrow on the terminal symbol t. Actually, neither p nor A is used in this definition, but we start from the transition $p \xrightarrow{A} r$ rather than from the state r because the **lookback** and **includes** relations use transitions rather than states.

Again nullable non-terminals complicate the situation. If r happens to have an outgoing arrow marked with a non-terminal C that produces ε, we can reduce ε to C in our simulation, stack it, shift over it and reach another state, say s. Then anything we are looking at after the transition $r \xrightarrow{C} s$ must also be added to the look-ahead set of $A \to \omega\bullet$. Note that this C need not be the C_1 in Figure 9.38; it can be any

nullable non-terminal marking an outgoing arrow from state r. This defines the **reads** relation: $(p \xrightarrow{A} r)$ **reads** $(r \xrightarrow{C} s)$ if and only if both transitions exist and $C \xrightarrow{*} \varepsilon$. And then all tokens u that fulfill $(r \xrightarrow{C} s)$ **directly-reads** u belong in the look-ahead set of $A \rightarrow \omega\bullet$ in state q. Of course state s can again have transitions on nullable non-terminals, which necessitate repeated application of the "**reads** and **directly-reads**" operation. This is step 3 of the simulation.

We are now in a position to formulate the LALR look-ahead construction algorithm in one single formula. It uses the final relation in our explanation, **in-LALR-lookahead**, which ties together a reduce item in a state and a token: t **in-LALR-lookahead** $(q, A \rightarrow \omega\bullet)$, with the obvious meaning. The relations algorithm can now be written as:

$$
\begin{aligned}
&t \textbf{ in-LALR-lookahead } (q, A \rightarrow \omega\bullet) \;=\\
&\quad (q, A \rightarrow \omega\bullet) \textbf{ lookback } (p \xrightarrow{A} r) \textbf{ includes } (p' \xrightarrow{B} r') \cdots\\
&\quad \cdots \textbf{ includes } (p'' \xrightarrow{B'} r'') \textbf{ reads } (r'' \xrightarrow{C} s) \cdots\\
&\quad \cdots \textbf{ reads } (r''' \xrightarrow{C'} s') \textbf{ directly-reads } t
\end{aligned}
$$

This is not a formula in the arithmetic sense of the word: one cannot put in parentheses to show the precedences, as one can in $a + b \times c$; it is rather a linked sequence of relations, comparable to $a < b \leq c < d$, in which each pair of values must obey the relational operator between them. It means that a token t is in the LALR lookahead set of reduce item $A \rightarrow \omega\bullet$ in state q if and only if we can find values for p, p', ..., B, B', ..., r, r', ..., C, C', ..., and s, s', ..., so that all the relations are obeyed.

In summary, when you do a reduction using a reduce item, the resulting non-terminal either is at the end of another item, in which case you have to include that item in your computations, or it has something in front of it, in which case your look-ahead set contains everything you can read from there, directly or through nullable non-terminals.

The question remains how to utilize the sequence of relations to actually compute the LALR look-ahead sets. Two techniques suggest themselves. We can start from the pair $(q, A \rightarrow \omega\bullet)$, follow the definitions of the relations until we reach a token t, record it, backtrack and exhaustively search all possibilities: the top-down approach. We can also make a database of relation triples, insert the initially known triples and apply the relation definitions until nothing changes any more: the transitive closure approach. Both have their problems. The top-down method has to be careful to prevent being caught in loops, and will often recompute relations. The transitive closure sweep will have to be performed an indefinite number of times, and will compute triples that do not contribute to the solution.

Fortunately there is a better way. It is not immediately evident, but the above algorithm has a remarkable property: it only uses the grammar and the LR(0) transitions over non-terminals (except for both ends of the relation sequence); it never looks inside the LR(0) states. The reasonings that show the validity of the various definitions use the presence of certain items, but the final definitions do not. This makes it particularly easy to express the relations as arcs in a directed graph in which the non-terminal transitions are the nodes.

The relations graph corresponding to Figures 9.37 and 9.38 is shown in Figure 9.39. We see that it is quite different from the transition graphs in Figures 9.37 and

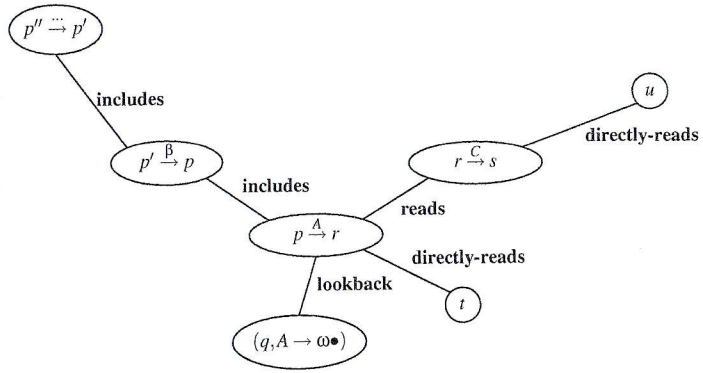

Fig. 9.39. Hunting for LALR(1) look-aheads in an LR(0) automaton — the relations graph

9.38: the transition arcs in those graphs have become nodes in the new graph, and the relations, not present in the old graphs, are the arcs in the new one. To emphasize this fact, the transition nodes in Figure 9.39 have been drawn in the same relative positions as the corresponding arcs in Figures 9.37 and 9.38; this is the cause of the strange proportions of Figure 9.39.

The LALR look-ahead sets can now be found by doing a transitive closure on this graph, to find all leaves connected to the $(q, A \rightarrow \omega\bullet)$ node. The point is that there exists a very efficient algorithm for doing transitive closure on a graph, the "SCCs algorithm". This algorithm successively isolates and condenses "strongly connected components" of the graph; hence its name. The algorithm was invented by Tarjan [334] in 1972, and is discussed extensively in books on algorithms and on the Internet.

DeRemer and Pennello describe the details required to cast the sequence of relations into a graph suitable for the SCCs algorithm. This leads to one of the most efficient LALR parse table construction algorithms known. It is linear in the number of relations involved in the computation, and in practice it is linear in the number of non-terminal transitions in the LR(0) automaton. It is several times faster than the channel algorithm used in *yacc*. Several optimizations can be found (Web)Section 18.1.4. Bermudez and Schimpf [76] extend the algorithm to LALR(k).

When reaching state r_{C_n} in Figure 9.37 we properly backtracked over all components of γ back to state p, to make sure that all look-aheads found could indeed be shifted when we perform the reduction $A \rightarrow \omega$. If we omit this step and just accept any look-ahead at r_{C_n} as look-ahead of $A \rightarrow \omega$, we obtain an *NQLALR(1)* parser, for "Not Quite LALR(1)". NQLALR(1) grammars are strange in that they do not fit in the usual hierarchy ((Bermudez and Schimpf [75]); but then, that can be expected from an incorrect algorithm.

9.7.1.4 LALR(1) by Converting to SLR(1)

When we look at the non-LR(0) automaton in Figure 9.25 with an eye to upgrading it to LALR(1), we realize that, for example, the **E** along the arrow from state 1 to state 4 is in fact a different **E** from that along the arrow from state 6 to state 9, in that it arises from a different station •**E**, the one in state 1, and it is the station that gets the look-ahead. So to distinguish it we can call it ①**E**④, so now the item **S**→•**E** reads **S**→•①**E**④, where ①**E**④ is just a non-terminal name, in spite of its appearance. This leads to the creation of a station •①**E**④ (not shown) which produces two items based on the two rules **E**→**T** and **E**→**E-T**. We can even give the non-terminals in these rules more specific names:

①**E**④ → ①**T**②
①**E**④ → ①**E**④ ④-⑦ ⑦**T**⑧

where we obtained the other state numbers by following the rules through the LR(0) automaton.

Continuing this way we can construct an "LR(0)-enhanced" version of the grammar of Figure 9.23; it is shown in Figure 9.40. A grammar rule $A \rightarrow BcD$ is trans-

①**S**◊ → ①**E**④
①**E**④ → ①**E**④ ④-⑦ ⑦**T**⑧ | ①**T**②
⑥**E**⑨ → ⑥**E**⑨ ⑨-⑦ ⑦**T**⑧ | ⑥**T**②
①**T**② → ①**n**③
⑥**T**② → ⑥(⑥ ⑥**E**⑨ ⑨)⑩ | ⑥**n**③
⑦**T**⑧ → ⑦(⑥ ⑥**E**⑨ ⑨)⑩ | ⑦**n**③

Fig. 9.40. An LR(0)-enhanced version of the grammar of Figure 9.23

formed into a new grammar rule $(s_1)A(s_x) \rightarrow (s_1)B(s_2)\ (s_2)c(s_3)\ (s_3)D(s_4)$, where (s_x) is the state shifted to by the non-terminal, and $(s_1)\cdots(s_4)$ is the sequence of states met when traveling down the right-hand side of the rule in the LR(0) automaton.

We see that the rules for **E** have been split into two versions, one starting at ① and the other at ⑥, and likewise the rules for **T**. It is clear that the look-aheads of the station •①**E**④ all end up in the look-ahead set of the item **E**→**E-T**• reached at the end of the sequence ①**E**④ ④-⑦ ⑦**T**⑧, so it is interesting to find out what the look-ahead set of the •①**E**④ in state 1 is, or rather just what the look-ahead set of •①**E**④ is, since there is only one •①**E**④ and it is in state 1.

Bermudez and Logothetis [79] have given a surprisingly simple answer to that question: the look-ahead set of •①**E**④ is the FOLLOW set of ①**E**④ in the LR(0)-enhanced grammar, and likewise for all the other LR(0)-enhanced non-terminals. Normally FOLLOW sets are not very fine tools, since they combine the tokens that can follow a non-terminal N from all over the grammar, regardless of the context in which the production N occurs. But here the LR(0) enhancement takes care of the context, and makes sure that terminal productions of •**E** in state 1 are recognized

only if they really derive from ①**E**④. That all this leads precisely to an LALR(1) parser is less clear; for a proof see the above paper.

To resolve the inadequacy of the automaton in Figure 9.25 we want to know the look-ahead set of the item **S**→**E**• in state 4, which is the FOLLOW set of ①**S**◊. The FOLLOW sets of the non-terminals in the LR(0)-enhanced grammar are as follows:

FOLLOW(①**S**◊) = [#]
FOLLOW(①**E**④) = [#-]
FOLLOW(⑥**E**⑨) = [-)]
FOLLOW(①**T**②) = [#-]
FOLLOW(⑥**T**②) = [-)]
FOLLOW(⑦**T**⑧) = [#-)]

so the desired LALR look-ahead set is #, in conformance with the "real" LALR automaton in Figure 9.34. Since state 4 was the only inadequate state, no more look-aheads sets need to be computed.

Actually, the reasoning in the previous paragraph is an oversimplification: a reduce item in a state may derive from more than one station and import look-aheads from each of them. To demonstrate this we compute the look-aheads of **E**→**E-T**• in state 8. The sequence ends in state 8, so we select from the LR(0)-enhanced grammar those rules of the form **E**→**E-T** that end in state 8:

①**E**④ → ①**E**④ ④-⑦ ⑦**T**⑧
⑥**E**⑨ → ⑥**E**⑨ ⑨-⑦ ⑦**T**⑧

We see that the look-aheads of both stations •①**E**④ and •⑥**E**⑨ end up in state 8, and so the LALR look-ahead set of **E**→**E-T**• in that state is

FOLLOW(①**E**④) ∪ FOLLOW(⑥**E**⑨) = [#-] ∪ [-)] = [#-)]

Since this is the same way as look-aheads are computed in an SLR parser for a normal — not LR(0)-enhanced — grammar (Section 9.8), the technique is often referred to as "converting to SLR".

The *LALR-by-SLR* technique is algorithmically very simple:

- deriving the LR(0)-enhanced grammar from the original grammar and the LR(0) automaton is straightforward;
- computing the FOLLOW sets is done by a standard algorithm;
- selecting the appropriate rules from the LR(0)-enhanced grammar is simple;
- uniting the results is trivial.

And, as said before, only the look-ahead sets of reduce items in inadequate states need to be computed.

9.7.1.5 Discussion

LALR(1) tables can be computed by at least five techniques: collapsing and downgrading the LR(1) tables; Anderson's simple algorithm; the channel algorithm; by upgrading the LR(0) automaton; and by converting to SLR(1). Of these, Anderson's algorithm [56] (Section 9.7.1.1) is probably the easiest to program, and its

non-optimal efficiency should only seldom be a problem on present-day machines. DeRemer and Pennello [63]'s relations algorithm (Section 9.7.1.3) and its relatives discussed in (Web)Section 18.1.4 are among the fastest. Much technical and experimental data on several LALR algorithms is given by Charles [88].

Vilares Ferro and Alonso Pardo [372] describe a remarkable implementation of an LALR parser in Prolog.

9.7.2 Identifying LALR(1) Conflicts

When a grammar is not LR(1), the constructed LR(1) automaton will have conflicts, and the user of the parser generator will have to be notified. Such notification often takes such forms as:

```
Reduce/reduce conflict
in state 213 on look-ahead ';'
    S->E versus A->T+E
```

This may seem cryptic but the user soon learns to interpret such messages and to reach the conclusion that indeed "the computer can't see this". This is because LR(1) parsers can handle all deterministic grammars and our idea of "what a computer can see" coincides reasonably well with what is deterministic.

The situation is worse for those (relatively rare) grammars that are LR(1) but not LALR(1). The user never really understands what is wrong with the grammar: the computer should be able to make the right parsing decisions, but it complains that it cannot. Of course there is nothing wrong with the grammar; the LALR(1) method is just marginally too weak to handle it.

To alleviate the problem, some research has gone into methods to elicit from the faulty automaton a possible input string that would bring it into the conflict state. See DeRemer and Pennello [63, Sect. 7]. The parser generator can then display such input with its multiple partial parse trees.

9.8 SLR(1)

There is a simpler way to proceed with the NFA of Figure 9.35 than using the channel algorithm: first pump around the look-ahead sets until they are all known and then apply the subset algorithm, rather than vice versa. This gives us the so called *SLR(1)* automaton (for Simple LR(1)); see DeRemer [54]. The same automaton can be obtained without using channels at all: construct the LR(0) automaton and then add to each item $A \rightarrow \cdots$ a look-ahead set that is equal to FOLLOW(A). Pumping around the look-ahead sets in the NFA effectively computes the FOLLOW sets of each non-terminal and spreads these over each item derived from it.

The SLR(1) automaton is shown in Figure 9.41. Since FOLLOW(S)={#}, FOLLOW(E)={#,-,)} and FOLLOW(T)={#,-,)}, only states 1 and 4 differ from those in the LALR(1) automaton of Figure 9.34. The increased look-ahead sets do not spoil the adequateness of any states: the grammar is also SLR(1).

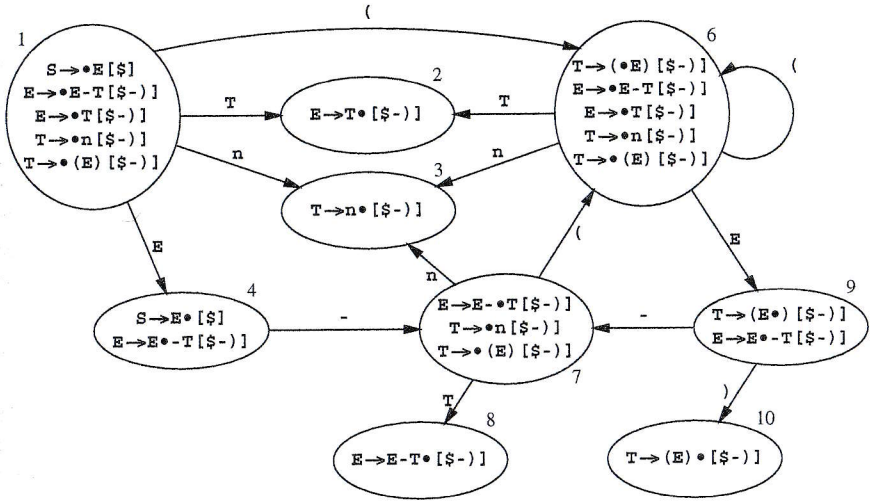

Fig. 9.41. SLR(1) automaton for the grammar of Figure 9.23

SLR(1) parsers are intermediate in power between LR(0) and LALR(1). Since SLR(1) parsers have the same size as LALR(1) parsers but are considerably less powerful, LALR(1) parsers are generally preferred.

FOLLOW$_k$ sets with $k > 1$ can also be used, leading to *SLR($k > 1$)* parsers. As with LA(k)LR(j), an LR(j) parser can be extended with additional FOLLOW$_k$ look-ahead, leading to *S(k)LR(j)* parsers. So SLR(1) is actually S(1)LR(0), and is just the most prominent member of the S(k)LR(j) parser family. To top things off, Bermudez and Schimpf [76] show that there exist NQSLR($k > 1$) parsers, thereby proving that "Simple LR" parsers are not really that simple for $k > 1$.

9.9 Conflict Resolvers

When states in an automaton have conflicts and no stronger method is available, the automaton can still be useful, provided we can find other ways to resolve the conflicts. Most LR parser generators have built-in conflict resolvers that will make sure that a deterministic automaton results, whatever properties the input grammar may have. Such a system will just enumerate the problems it has encountered and indicate how it has solved them.

Two useful and popular rules of thumb to solve LR conflicts are:

- on a shift/reduce conflict, shift (only on those look-aheads for which the conflict occurs);
- on a reduce/reduce conflict, reduce using the longest rule.

Both rules implement the same idea: take the largest bite possible. If you find that there is a production of A somewhere, make it as long as possible, including as much material on both sides as possible. This is very often what the grammar writer wants.