9

Deterministic Bottom-Up Parsing

There is a great variety of deterministic bottom-up parsing methods. The first deterministic parsers (Wolpe [110], Adams and Schlesinger [109]) were bottom-up parsers and interest has only increased since. The full bibliography of this book on its web site contains about 280 entries on deterministic bottom-up parsing against some 85 on deterministic top-down parsing. These figures may not directly reflect the relative importance of the methods, but they are certainly indicative of the fascination and complexity of the subject of this chapter.

There are two families of deterministic bottom-up parsers:

- Pure bottom-up parsers. This family comprises the precedence and bounded-(right)-context techniques, and are treated in Sections 9.1 to 9.3.
- Bottom-up parsers with an additional top-down component. This family, which is both more powerful and more complicated than the pure bottom-up parsers, consists of the LR techniques and is treated in Sections 9.4 to 9.10.

There are two main ways in which deterministic bottom-up methods are extended to allow more grammars to be handled:

- Remaining non-determinism is resolved by breadth-first search. This leads to Generalized LR parsing, which is covered in Section 11.1.
- The requirement that the bottom-up parser does the reductions in reverse rightmost production order (see below and Section 3.4.3.2) is dropped. This leads to non-canonical parsing, which is covered in Chapter 10.

The proper setting for the subject at hand can best be obtained by summarizing a number of relevant facts from previous chapters.

- A rightmost production expands the rightmost non-terminal in a sentential form, by replacing it by one of its right-hand sides, as explained in Section 2.4.3. A sentence is then produced by repeated rightmost production until no non-terminal remains. See Figure 9.1(*a*), where the sentential forms are right-aligned to show how the production process creeps to the left, where it terminates. The grammar used is that of Figure 7.8.

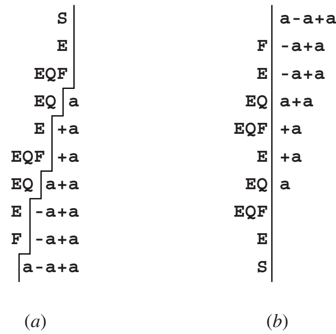


Fig. 9.1. Rightmost production (*a*) and rightmost reduction (*b*)

- Each step of a bottom-up parser, working on a sentential form, identifies the latest rightmost production in it and undoes it by reducing a segment of the input to the non-terminal it derived from. The identified segment and the production rule are called the "handle" (Section 3.4.3.2).

  Since the parser starts with the final sentential form of the production process (that is, the input) it finds its first reduction somewhere near to the left end, which is convenient for stream-based input. A bottom-up parser identifies rightmost productions in reverse order. See Figure 9.1(*b*) where the handles are left-aligned to show how the reduction process condenses the input.
- To obtain an efficient parser we need an efficient method to identify handles, without considering alternative choices. So the handle search must either yield *one* handle, in which case it must be the proper one, or *no* handle, in which case we have found an error in the input.

Although this chapter is called "Deterministic Bottom-Up *Parsing*", it is almost exclusively concerned with methods for finding handles. Once the handle is found, parsing is (almost always) trivial. The exceptions will be treated separately.

Unlike top-down parsing, which identifies productions before any of its constituents have been identified, bottom-up parsing identifies a production only at its very end, when all its constituents have already been identified. A top-down parser allows semantic actions to be performed at the beginning of a production and these actions can help in determining the semantics of the constituents. In a bottom-up parser, semantic actions are only performed during a reduction, which occurs at the end of a production, and the semantics of the constituents have to be determined without the benefit of knowing in which production they occur. We see that the increased power of bottom-up parsing compared to top-down parsing comes at a price: since the decision what production applies is postponed to the last moment, that decision can be based upon the fullest possible information, but it also means that the actions that depend on this decision come very late.

9.1 Simple Handle-Finding Techniques

There is a situation in daily life in which the average citizen is called upon to identify a handle. If one sees a formula like

$$4 + 5 \times 6 + 8$$

one immediately identifies the handle and evaluates it:

$$4 + \underline{5 \times 6} + 8$$

$$4 + \quad 30 \quad + 8$$

The next handle is

$$\underline{4 + \quad 30} \quad + 8$$

$$34 \qquad + 8$$

and then

$$\underline{34 \qquad + 8}$$

$$42$$

If we look closely, we can discern shifts and reduces in this process. People doing the arithmetic shift symbols until they reach the situation

$$4 + 5 \times 6 \qquad\qquad + 8$$

in which the control mechanism in their heads tells them that this is the right moment to do a reduce. If asked why, they might answer something like: “Ah, well, I was taught in school that multiplication comes before addition”. Before we formalize this notion and turn it into a parsing method, we consider an even simpler case.

Meanwhile we note that formulas like the one above are called “arithmetic expressions” and are produced by the grammar of Figure 9.2. **S** is the start symbol, **E**

```
Ss  -->  E
 E  -->  E + T
 E  -->  T
 T  -->  T × F
 T  -->  F
 F  -->  n
 F  -->  ( E )
```

Fig. 9.2. A grammar for simple arithmetic expressions

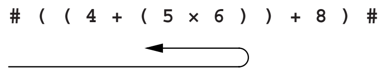
stands for “expression”, **T** for “term”, **F** for “factor” and **n** for any number. Having **n** rather than an explicit number causes no problems, since the exact value is immaterial to the parsing process. We have demarcated the beginning and the end of the

expression with # marks; the blank space that normally surrounds a formula is not good enough for automatic processing. The parser accepts the input as correct and stops when the input has been reduced to $\#S_s\#$.

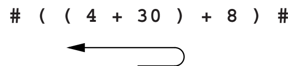
An arithmetic expression is *fully parenthesized* if each operator together with its operands has parentheses around it:

$$
\begin{aligned}
S_s &\rightarrow E \\
E &\rightarrow (E + T) \\
E &\rightarrow T \\
T &\rightarrow (T \times F) \\
T &\rightarrow F \\
F &\rightarrow n
\end{aligned}
$$

Our example expression would have the form



Now finding the handle is trivial: go to the first closing parenthesis and then back to the nearest opening parenthesis. The segment between and including the parentheses is the handle and the operator identifies the production rule. Reduce it and repeat the process as often as required. Note that after the reduction there is no need to start all over again, looking for the first closing parenthesis: there cannot be any closing parenthesis on the left of the reduction spot. So we can start searching right where we are. In the above example we find the next right parenthesis immediately and do the next reduction:



9.2 Precedence Parsing

Of course, grammars normally do not have these convenient begin- and end markers to each compound right-hand side, and the above parsing method has little practical value (as far as we know it does not even have a name). Yet, suppose we had a method for inserting the proper parentheses into an expression that was lacking them. At a first glance this seems trivial to do: when we see **+n×** we know we can replace this by **+(n×** and we can replace **×n+** by **×n)+**. There is a slight problem with **+n+**, but since the first **+** has to be performed first, we replace this by **+n)+**. The **#**s are easy; we can replace **#n** by **#(n** and **n#** by **n)#**. For our example we get:

```
# ( 4 + ( 5 × 6 ) + 8 ) #
```

This is, however, not quite correct — it should have been **#((4+(5×6))+8)#** — and for **4+5×6** we get the obviously incorrect form **#(4+(5×6)#**.

9.2.1 Parenthesis Generators

The problem is that we do not know how many parentheses to insert in, for example, `+n×`: in `4+5×6` we should replace it by `+(n×` to obtain `#(4+(5×6))#`, but `4+5×6×76×8` would require it to be replaced by `+(((n×`, etc. We solve this problem by inserting *parenthesis generators* rather than parentheses. A generator for open parentheses is traditionally written as $\lessdot$, one for closing parentheses as $\gtrdot$; we shall also use a "non-parenthesis", $\doteq$. These symbols look confusingly like $<$, $>$ and $=$, to which they are only remotely related. Now our tentatively inserted parentheses become firmly inserted parenthesis generators; see Figure 9.3. We have left out the

$$
\begin{array}{lcl}
+\ \times & \Rightarrow & +\ \lessdot\ \times \\
\times\ + & \Rightarrow & \times\ \gtrdot\ + \\
+\ + & \Rightarrow & +\ \gtrdot\ + \\
\#\cdots & \Rightarrow & \#\ \lessdot \cdots \\
\cdots\# & \Rightarrow & \cdots \gtrdot\ \#
\end{array}
$$

Fig. 9.3. Preliminary table of precedence relations

`n` since the parenthesis generator is dependent on the left and right operators only.

The table in Figure 9.3 is incomplete: the pattern `× ×` is missing, as are all patterns involving parentheses. In principle there should be a pattern for each combination of two operators (where we count the genuine parentheses as operators), and only the generator to be inserted is relevant for each combination. This generator is called the *precedence relation* between the two operators. It is convenient to collect all combinations of operators in a table, the *precedence table*. The precedence table for the grammar of Figure 9.2 is given in Figure 9.4; the leftmost column contains the left-hand symbols and the top-most row the right-hand symbols.

	#	+	×	(	)
#	$\doteq$	$\lessdot$	$\lessdot$	$\lessdot$	
+	$\gtrdot$	$\gtrdot$	$\lessdot$	$\lessdot$	$\gtrdot$
×	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\lessdot$	$\gtrdot$
(		$\lessdot$	$\lessdot$	$\lessdot$	$\doteq$
)	$\gtrdot$	$\gtrdot$	$\gtrdot$		$\gtrdot$

Fig. 9.4. Operator-precedence table to the grammar of Figure 9.2

There are three remarks to be made about this precedence table. First, we have added a number of $\lessdot$ and $\gtrdot$ tokens not covered above (for example, $\times\gtrdot\times$). Second, there is $\#\doteq\#$ and $(\doteq)$ — but there is no $)\doteq($! We shall shortly see what they mean. And third, there are three empty entries. When we find these combinations in the input, it contains an error.

Such a table is called a precedence table because for symbols that are normally regarded as operators it gives their relative precedence. An entry like $+\lessdot\times$ indicates

that in the combination `+×`, the `×` has a higher precedence than the `+`. We shall first show how the precedence table is used in parsing and then how such a precedence table can be constructed systematically for a given grammar, if the grammar allows it.

The stack in an operator-precedence parser differs from the normal bottom-up parser stack in that it contains "important" symbols, the operators, between which relations are defined, and "unimportant" symbols, the numbers, which are only consulted to determine the value of a handle and which do not influence the parsing. Moreover, we need locations on the stack to hold the parenthesis generators between the operators (although one could, in principle, do without these locations, by reevaluating the parenthesis generators again whenever necessary). Since there is a parenthesis generator between each pair of operators and there is also (almost) always a value between such a pair, we shall indicate both in the same position on the stack, with the parenthesis generator in line and the value below it; see Figure 9.5.

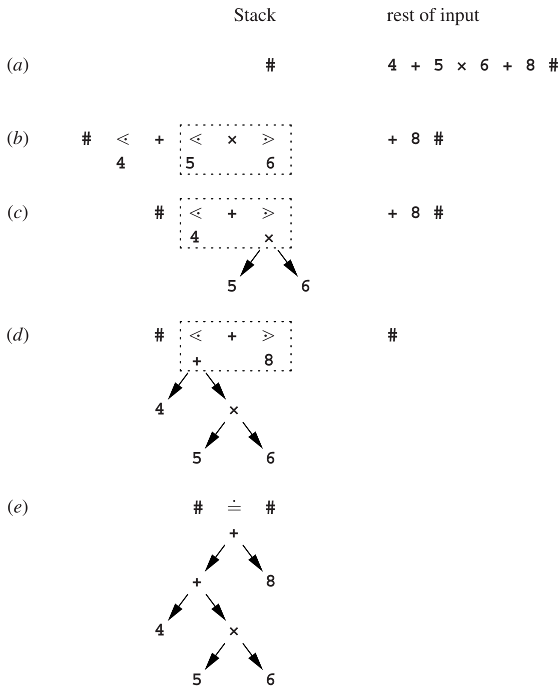


Fig. 9.5. Operator-precedence parsing of `4+5×6+8`

To show that, contrary to what is sometimes thought, operator-precedence can do more than just compute a value (and since we have seen too often now that `4+5×6+8=42`), we shall have the parser construct the parse tree rather than the value. The stack starts with a `#`. Values and operators are shifted onto it, interspersed with parenthesis generators, until a ⋗ generator is met; the following operator is not

shifted and is left in the input (Figure 9.5(*b*)). It is now easy to identify the handle segment, which is demarcated by a dotted rectangle in the figure. The operator **×** identifies the type of node to be created, and the handle is now reduced to a tree; see (*c*), in which also the next $\gtrdot$ has already appeared between the **+** on the stack and the **+** in the input. We see that the tree and the new generator have come in the position of the $\lessdot$ of the handle. A further reduction brings us to (*d*) in which the **+** and the **8** have already been shifted, and then to the final state of the operator-precedence parser, in which the stack holds **#**$\doteq$**#** and the parse tree dangles from the value position.

We see that the stack only holds $\lessdot$ markers and values, plus a $\gtrdot$ on the top each time a handle is found. The meaning of the $\doteq$ becomes clearer when we parse an input text which includes parentheses, like `4×(5+6)`; see Figure 9.6. We see that the

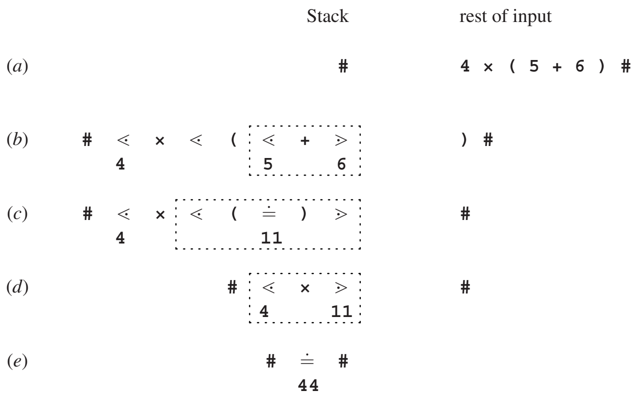


Fig. 9.6. An operator-precedence parsing involving $\doteq$

$\doteq$ is used to build handles consisting of more than one operator and two operands; the handle in (*c*) has two operators, the **(** and the **)** and one operand, the **11**. Where the $\lessdot$ generates open parentheses and the $\gtrdot$ generates close parentheses, both of which cause level differences in the parse tree, the $\doteq$ generates no parentheses and allows the operands to exist on the same level in the parse tree.

As already indicated on page 200, the set of stack configurations of a bottom-up parser can be described by a regular expression. For precedence parsers the expression is easy to see:

$$\texttt{\#} \mid \texttt{\#}\lessdot\texttt{q}\ \texttt{(}[\lessdot\doteq]\texttt{q)}^{*}\ \gtrdot^{?} \mid \texttt{\#}\doteq\texttt{\#}$$

where **q** is any operator; the first alternative is the start situation and the third alternative is the end situation. (Section 9.12.2 will show more complicated regular expressions for other bottom-up parsers.)

9.2.2 Constructing the Operator-Precedence Table

The above hinges on the difference between operators, which are terminal symbols and between which precedence relations are defined, and operands, which are non-

terminals. This distinction is captured in the following definition of an operator grammar:

> A CF grammar is an *operator grammar* if (and only if) each right-hand side contains at least one terminal or non-terminal and no right-hand side contains two consecutive non-terminals.

So each pair of non-terminals is separated by at least one terminal; all the terminals except those carrying values (**n** in our case) are called operators.

For such grammars, setting up the precedence table is relatively easy. First we compute for each non-terminal A the set $FIRST_{OP}(A)$, which is the set of all operators that can occur as the first operator in sentential forms deriving from A. Note that such a first operator can be preceded by at most one non-terminal in an operator grammar. The FIRST_{OP}s of all non-terminals are constructed simultaneously as follows:

1. For each non-terminal A, find all right-hand sides of all rules for A; now for each right-hand side R we insert the first operator in R (if any) into $\text{FIRST}_{\text{OP}}(A)$. This gives us the initial values of all FIRST_{OP}s.
2. For each non-terminal A, find all right-hand sides of all rules for A; now for each right-hand side R that starts with a non-terminal, say B, we add the elements of $\text{FIRST}_{\text{OP}}(B)$ to $\text{FIRST}_{\text{OP}}(A)$. This is reasonable, since a sentential form of A may start with B, so all operators in $\text{FIRST}_{\text{OP}}(B)$ should also be in $\text{FIRST}_{\text{OP}}(A)$.
3. Repeat step 2 above until no FIRST_{OP} changes any more. We have now found the FIRST_{OP} of all non-terminals.

We will also need the set $LAST_{OP}(A)$, which is defined similarly, and a similar algorithm, using the *last* operator in R in step 1 and a B which *ends* A in step 2 provides it. The sets for the grammar of Figure 9.2 are shown in Figure 9.7.

FIRST_{OP}(**S**) = {**#**}	LAST_{OP}(**S**) = {**#**}
FIRST_{OP}(**E**) = {**+**, **×**, **(**}	LAST_{OP}(**E**) = {**+**, **×**, **)**}
FIRST_{OP}(**T**) = {**×**, **(**}	LAST_{OP}(**T**) = {**×**, **)**}
FIRST_{OP}(**F**) = {**(**}	LAST_{OP}(**F**) = {**)**}

Fig. 9.7. FIRST_{OP} and LAST_{OP} sets for the grammar of Figure 9.2

Now we can fill the precedence table using the following rules, in which q, q_1 and q_2 are operators and A is a non-terminal.

- For each occurrence in a right-hand side of the form $q_1\ q_2$ or $q_1\ A\ q_2$, set $q_1 \doteq q_2$. This keeps operators from the same handle together.
- For each occurrence $q_1\ A$, set $q_1 \lessdot q_2$ for each q_2 in $\text{FIRST}_{\text{OP}}(A)$. This demarcates the left end of a handle.
- For each occurrence Aq_1, set $q_2 \gtrdot q_1$ for each q_2 in $\text{LAST}_{\text{OP}}(A)$. This demarcates the right end of a handle.

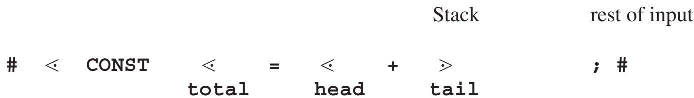
If we obtain a table without conflicts this way, that is, if we never find two different relations between two operators, then we call the grammar *operator-precedence*.

It will now be clear why `(`$\doteq$`)` and not `)`$\doteq$`(` in our grammar of Figure 9.2, and why `+`$\gtrdot$`+`: because `E+` occurs in `E-->E+T` and `+` is in LAST_{OP}(`E`).

In this way, the table can be derived from the grammar by a program and be passed on to the operator-precedence parser. A very efficient linear-time parser results. There is, however, one small problem we have glossed over: Although the method properly identifies the handle segment, it often does not identify the non-terminal to which to reduce it. Also, it does not show any unit rule reductions; nowhere in the examples did we see reductions of the form `E-->T` or `T-->F`. In short, operator-precedence parsing generates only *skeleton parse trees*.

Operator-precedence parsers are very easy to construct (often even by hand) and very efficient to use; operator-precedence is the method of choice for all parsing problems that are simple enough to allow it. That only a skeleton parse tree is obtained, is often not an obstacle, since operator grammars often have the property that the semantics is attached to the operators rather than to the right-hand sides; the operators are identified correctly.

It is surprising how many grammars are (almost) operator-precedence. Almost all formula-like computer input is operator-precedence. Also, large parts of the grammars of many computer languages are operator-precedence. An example is a construction like `CONST total = head + tail;` from a Pascal-like language, which is easily rendered as:



Ignoring the non-terminals has other bad consequences besides producing a skeleton parse tree. Since non-terminals are ignored, a missing non-terminal is not noticed. As a result, the parser will accept incorrect input without warning and will produce an incomplete parse tree for it. A parser using the table of Figure 9.4 will blithely accept the empty string, since it immediately leads to the stack configuration `#`$\doteq$`#`. It produces a parse tree consisting of one empty node.

The theoretical analysis of this phenomenon turns out to be inordinately difficult; see Levy [125], Williams [128, 129, 131] and many others in (Web)Section 18.1.6. In practice it is less of a problem than one would expect; it is easy to check for the presence of required non-terminals, either while the parse tree is being constructed or afterwards — but such a check would not follow from the parsing technique.

9.2.3 Precedence Functions

Although precedence tables require room for only a modest $|V_T|^2$ entries, where $|V_T|$ is the number of terminals in the grammar, they can often be represented much more frugally by so-called *precedence functions*, and it is usual to do so. The idea is the following. Rather than having a *table* T such that for any two operators q_1 and q_2, $T[q_1,q_2]$ yields the relation between q_1 and q_2, we have two integer *functions* f and g such that $f(q_1) < g(q_2)$ means that $q_1 \lessdot q_2$, $f(q_1) = g(q_2)$ means $q_1 \doteq q_2$

and $f(q_1) > g(q_2)$ means $q_1 \gtrdot q_2$. $f(q)$ is called the *left priority* of q, $g(q)$ the *right priority*; they would probably be better indicated by l and r, but the use of f and g is traditional. It will be clear that *two* functions are required: with just one function one cannot express, for example, $+ \gtrdot +$. Precedence functions take much less room than precedence tables: $2|V_T|$ entries versus $|V_T|^2$ for the table. Not all tables allow a representation with two precedence functions, but many do.

Finding the proper f and g for a given table seems simple enough and can indeed often be done by hand. The fact, however, that there are two functions rather than one, the size of the tables and the occurrence of the $\doteq$ complicate things. An algorithm to construct the two functions was given by Bell [120]. There is always a way to represent a precedence table with more than two functions; Bertsch [127] shows how to construct such functions.

Finding two precedence functions is equivalent to reordering the rows and columns of the precedence table so that the latter can be divided into three regions: a $\gtrdot$ region on the lower left, a $\lessdot$ region on the upper right and a $\doteq$ border between them; see Figure 9.8. The process is similar but not equivalent to doing a topological sort on f_q and g_q.

	#	)	+	×	(
#	$\doteq$		$\lessdot$	$\lessdot$	$\lessdot$
(		$\doteq$	$\lessdot$	$\lessdot$	$\lessdot$
+	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\lessdot$	$\lessdot$
×	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\lessdot$
)	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$	

Fig. 9.8. The precedence table of Figure 9.4 reordered

Precedence parsing recognizes that many languages have tokens that define the structure and tokens that carry the information; the first are the operators, the second the operands. That raises the question whether that difference can be formalized; see Gray and Harrison [124] for a partial answer, but usually the question is left to the user.

Some operators are actually composite; the C and Java programming language conditional expression, which is formed by two parts: `x>0?x:0` yields `x` if `x` is greater than 0; otherwise it yields 0. Such distributed operators are called *distfix operators*. They can be handled by precedence-like techniques; see, for example Peyton Jones [132] and Aasa [133].

9.2.4 Further Precedence Methods

Operator precedence structures the input in terms of operators only: it yields skeleton parse trees — correctly structured trees with the terminals as leaves but with unlabeled nodes — rather than parse trees. As such it is quite powerful, and serves in many useful programs to this day. In some sense it is even stronger than the more

famous LR techniques: operator precedence can easily handle ambiguous grammars, as long as the ambiguity remains restricted to the labeling of the tree. We could add a rule **E-->n** to the grammar of Figure 9.2 and it would be ambiguous but still operator-precedence. It achieves its partial superiority over LR by not fulfilling the complete task of parsing: getting a completely labeled parse tree.

There is a series of more advanced precedence parsers, which do properly label the parse tree with non-terminals. They were very useful at the time they were invented, but today their usefulness has been eclipsed by the LALR and LR parsers, which we will treat further on in this chapter (Sections 9.4 through 9.14). We will therefore only briefly touch upon them here, and refer the reader to the many publications in (Web)Section 18.1.6.

The most direct way to bring back the non-terminals in the parse tree is to involve them like the terminals in the precedence relations. This idea leads to *simple precedence* parsing (Wirth and Weber [118]). A grammar is simple precedence if and only if:

- it has a conflict-free precedence table over all its symbols, terminals and non-terminals alike;
- none of its right-hand sides is ε;
- all of its right-hand sides are different.

For example, we immediately have the precedence relations **(**$\doteq$**E** and **E**$\doteq$**)** from the rule **F-->(E)**.

The construction of the simple-precedence table is again based upon two sets, $FIRST_{ALL}(A)$ and $LAST_{ALL}(A)$. $\text{FIRST}_{\text{ALL}}(A)$ is similar to the set FIRST(A) from Section 8.2.1.1, and differs from it in that it also contains all non-terminals that can start a sentential form derived from A, whereas FIRST(A) contains terminals only. A similar definition applies to $\text{LAST}_{\text{ALL}}(A)$.

Unfortunately almost no grammar is simple-precedence, not even the simple grammar of Figure 9.2, since we have **(**$\lessdot$**E** in addition to **(**$\doteq$**E**, due to the occurrence of **(E** in **F-->(E)**, and **E** being in $\text{FIRST}_{\text{ALL}}$(**E**) from **E-->E+T**. A few other conflicts also occur. On the bright side, this kind of conflict can often be solved by inserting extra levels around the troublesome non-terminals, as done in Figure 9.9, but this brings us farther away from our goal, producing a correct parse tree.

It turns out that most of the simple-precedence conflicts are $\lessdot/\doteq$ conflicts. Now the difference between $\lessdot$ and $\doteq$ is in a sense less important than that between either of them and $\gtrdot$. Both $\lessdot$ and $\doteq$ result in a shift and only $\gtrdot$ asks for a reduce. Only when a reduce is found will the difference between $\lessdot$ and $\doteq$ become significant for finding the left end of the handle. Now suppose we drop the difference between $\lessdot$ and $\doteq$ and combine them into $\leq$; then we need a different means of identifying the handle segment. This can be done by requiring not only that all right-hand sides be different, but also that no right-hand side be equal to the tail of another right-hand side. A grammar that conforms to this and has a conflict-free $\leq/\gtrdot$ precedence table is called *weak precedence* (Ichbiah and Morse [121]).

```
Ss  -->  E'
E'  -->  E
 E  -->  E + T'
 E  -->  T'
T'  -->  T
 T  -->  T × F
 T  -->  F
 F  -->  n
 F  -->  ( E )
```

FIRST$_{ALL}$(E′) = {E, T′, T, F, n, (}   LAST$_{ALL}$(E′) = {T′, T, F, n,)}
FIRST$_{ALL}$(E) = {E, T′, T, F, n, (}   LAST$_{ALL}$(E) = {T, F, n,)}
FIRST$_{ALL}$(T′) = {T, F, n, (}   LAST$_{ALL}$(T′) = {F, n,)}
FIRST$_{ALL}$(T) = {T, F, n, (}   LAST$_{ALL}$(T) = {F, n,)}
FIRST$_{ALL}$(F) = {n, (}   LAST$_{ALL}$(F) = {n,)}

	#	E′	E	T′	T	F	n	+	×	(	)
#		$\doteq$	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$			$\lessdot$	
E′	$\doteq$										
E	$\gtrdot$							$\doteq$			$\doteq$
T′	$\gtrdot$							$\gtrdot$			$\gtrdot$
T	$\gtrdot$							$\gtrdot$	$\doteq$		$\gtrdot$
F	$\gtrdot$							$\gtrdot$	$\gtrdot$		$\gtrdot$
n	$\gtrdot$							$\gtrdot$	$\gtrdot$		$\gtrdot$
+				$\doteq$	$\lessdot$	$\lessdot$	$\lessdot$			$\lessdot$	
×						$\doteq$	$\lessdot$			$\lessdot$	
(			$\doteq$	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$			$\lessdot$	
)	$\gtrdot$							$\gtrdot$	$\gtrdot$		$\gtrdot$

Fig. 9.9. Modifying the grammar from Figure 9.2, to obtain a conflict-free simple-precedence table

Unfortunately the simple grammar of Figure 9.2 is not weak-precedence either. The right-hand side of **E→T** is the tail of the right-hand side of **E→E+T**, and upon finding the stack

$$\cdots \lessdot \mathbf{E} \doteq + \lessdot \mathbf{T} \gtrdot$$

we do not know whether to reduce with **E→T** or with **E→E+T**. Several tricks are possible: taking the longest reduce, looking deeper on the stack, etc.

The above methods determine the precedence relations by looking at 1 symbol on the stack and 1 token in the input. Once this has been said, the idea suggests itself to generalize this and to determine the precedence relations from the topmost m symbols on the stack and the first n tokens in the input. This is called *(m,n)-extended precedence* (Wirth and Weber [118]). For many entries in the table checking the full length on the stack and in the input is overkill, and ways have been found to use just

enough information, thus greatly reducing the table sizes. This technique is called *mixed-strategy precedence* (McKeeman [123]).

9.3 Bounded-Right-Context Parsing

There is a different way to solve the annoying problem of the identification of the right-hand side: let the identity of the rule be part of the precedence relation. This means that for each combination of, say, m symbols on the stack and n tokens in the input there should be a unique parsing decision which is either “shift” ($\lessdot$) or “reduce using rule X” ($\gtrdot_X$), as obtained by a variant of the rules for extended precedence. The parser is then a form of *bounded-right-context*. Figure 9.10 gives such tables for $m = 2$ and $n = 1$ for the grammar of Figure 9.2; these tables were constructed by hand. The rows correspond to stack symbol pairs; the entry Accept means that

	`#`	`+`	`×`	`n`	`(`	`)`
`#S`	Accept					
`#E`	$\gtrdot_{S\to E}$	$\lessdot$				Error
`#T`	$\gtrdot_{E\to T}$	$\gtrdot_{E\to T}$	$\lessdot$			Error
`#F`	$\gtrdot_{T\to F}$	$\gtrdot_{T\to F}$	$\gtrdot_{T\to F}$			Error
`#n`	$\gtrdot_{F\to n}$	$\gtrdot_{F\to n}$	$\gtrdot_{F\to n}$	Error	Error	Error
`#(`	Error	Error	Error	$\lessdot$	$\lessdot$	Error
`E+`	Error	Error	Error	$\lessdot$	$\lessdot$	Error
`E)`	$\gtrdot_{F\to(E)}$	$\gtrdot_{F\to(E)}$	$\gtrdot_{F\to(E)}$	Error	Error	$\gtrdot_{F\to(E)}$
`T×`	Error	Error	Error	$\lessdot$	$\lessdot$	Error
`+T`	$\gtrdot_{E\to E+T}$	$\gtrdot_{E\to E+T}$	$\lessdot$			$\gtrdot_{E\to E+T}$
`+F`	$\gtrdot_{T\to F}$	$\gtrdot_{T\to F}$	$\gtrdot_{T\to F}$			$\gtrdot_{T\to F}$
`+n`	$\gtrdot_{F\to n}$	$\gtrdot_{F\to n}$	$\gtrdot_{F\to n}$	Error	Error	$\gtrdot_{F\to n}$
`+(`	Error	Error	Error	$\lessdot$	$\lessdot$	Error
`×F`	$\gtrdot_{T\to T\times F}$	$\gtrdot_{T\to T\times F}$	$\gtrdot_{T\to T\times F}$			$\gtrdot_{T\to T\times F}$
`×n`	$\gtrdot_{F\to n}$	$\gtrdot_{F\to n}$	$\gtrdot_{F\to n}$	Error	Error	$\gtrdot_{F\to n}$
`×(`	Error	Error	Error	$\lessdot$	$\lessdot$	Error
`(E`	Error	$\lessdot$				$\lessdot$
`(T`	Error	$\gtrdot_{E\to T}$	$\lessdot$			$\gtrdot_{E\to T}$
`(F`	Error	$\gtrdot_{T\to F}$	$\gtrdot_{T\to F}$			$\gtrdot_{T\to F}$
`(n`	Error	$\gtrdot_{F\to n}$	$\gtrdot_{F\to n}$	Error	Error	$\gtrdot_{F\to n}$
`((`	Error	Error	Error	$\lessdot$	$\lessdot$	Error

Fig. 9.10. BC(2,1) table for the grammar of Figure 9.2

the input has been parsed and Error means that a syntax error has been found. Blank entries will never be accessed; all-blank rows have been left out. See, for example, Loeckx [122] for an algorithm for the construction of such tables.

9.3.1 Bounded-Context Techniques

The table of Figure 9.10 represents a variant of bounded-context, or more precisely, a particular implementation of bounded-right-context. To understand the bounded-context idea we have to go back to the basic bottom-up parsing algorithm explained in Section 3.2.2: find a right-hand side anywhere in the sentential form and reduce it. But we have already seen that often such a reduction creates a node that is not a node of the final parse tree, and backtracking is needed. For example, when we reduce the second **n** in **#n×n#** to **F** we have created a node that belongs to the parse tree. We obtain **#n×F#**, but if we now reduce the **F** to **T**, obtaining **#n×T#**, we have gone one step too far, and will no longer get a parsing. So why is the first reduction OK, and the second is not?

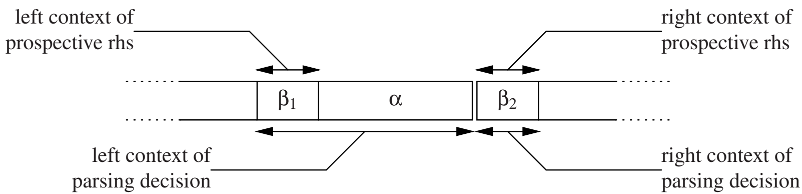
In bounded-context parsing the proposed reductions are restricted by context conditions. A right-hand side α of a rule $A \rightarrow \alpha$ found in a sentential form can only be reduced to A if it appears in the right context, $\beta_1\alpha\beta_2$. Here β_1 is the left context, β_2 the right one. Both contexts must be of bounded length, hence "bounded context"; either or both can be ε.

Using these contexts, it is easy to see from the grammar that **n** in the context **×**$\cdots$**#** can be reduced to **F**, but **F** in the context **×**$\cdots$**#** cannot be reduced to **T**, although in the context **+**$\cdots$**#** it could. Turning this intuition into an algorithm is very difficult. A grammar is *bounded-context* if no segment $\beta_1\alpha\beta_2$ that results from a production $A \rightarrow \alpha$ in a sentential form can result in any other way. If that condition holds, we can, upon seeing the context pattern $\beta_1\alpha\beta_2$, safely reduce to $\beta_1 A\beta_2$. If the maximum length of β_1 is m and that of β_2 is n, the grammar is *BC(m,n)*.

Finding sufficient and non-conflicting contexts is a difficult affair, which is sketched by Floyd [117]. Because of this difficulty, bounded-context is of no consequence as a parsing method; but bounded-context grammars are important in error recovery (Richter [313], Ruckert [324]) and substring parsing (Cormack [211], Ruckert [217]), since they allow parsing to be resumed in arbitrary positions. This property is treated in Section 16.5.2.

If all right contexts in a bounded-context grammar contain terminals only, the grammar and its parser are *bounded-right-context*, or *BRC(m,n)*. Much more is known about bounded-right-context than about general bounded-context, and extensive table construction algorithms are given by Eickel et al. [115] and Loeckx [122]. Table construction is marginally easier for BRC than for BC, but it can handle fewer grammars.

The implementation of BRC parsing as sketched above is awkward: to try a reduction $A \rightarrow \alpha$ in the context $\beta_1 \cdots \beta_2$ the top of the stack must be tested for the presence of α, which is of variable length, and then β_1 on the stack and β_2 in the input must be verified; repeat for all rules and all contexts. It is much more convenient to represent all triplets $(\beta_1\alpha\beta_2)$ as pairs $(\beta_1\alpha, \beta_2)$ in a matrix, like the one in Figure 9.10; in this way $\beta_1\alpha$ and β_2 are basically the left and right contexts of the parsing decision at the gap between stack and rest of input:



As a final step the left contexts are cut to equal lengths, in such a way that enough information remains. This is usually easily done; see Figure 9.10. This brings BRC parsing in line with the other table-driven bottom-up parsing algorithms.

Although some publications do not allow it, BC and BRC parsers can handle nullable non-terminals. If we add the rule `E`$\rightarrow\varepsilon$ to the grammar of Figure 9.2, the context `(,)` is strong enough to conclude that reducing with `E`$\rightarrow\varepsilon$ is correct.

Bounded-right-context is much more prominent than bounded-context, but since it is more difficult to pronounce, it is often just called "bounded-context"; this sometimes leads to considerable confusion. BRC(2,1) is quite powerful and was once very popular, usually under the name "BC(2,1)", but has been superseded almost completely by LALR(1) (Section 9.7).

It should be pointed out that bounded-context can identify reductions in non-canonical order, since a context reduction may be applied anywhere in the sentential form. Such a reduction can then result in a non-terminal which is part of the right context of another reduction pattern. So bounded context actually belongs in Chapter 10, but is easier to understand here.

If in bounded-*right*-context we repeatedly apply the first context reduction we find in a left-to-right sweep, we identify the reductions in canonical order, since the right context is free from non-terminals all the time, so no non-canonical reductions are needed.

If during table construction for a bounded-context parser we find that a segment $\beta_1\alpha\beta_2$ produced from $\beta_1 A\beta_2$ can also be produced otherwise, we can do two things: we can decide that the grammar is not BC and give up, or we can decide not to include the segment in our table of reduction contexts and continue. In doing so we now run the risk of losing some parsings, unless we can prove that for any sentential form there is at least one reduction context left. If that is the case, the grammar is *bounded-context parsable* or *BCP*. Constructing parse tables for BCP(m,n) is even more difficult than for BC or BRC, but the method can handle substantially more grammars than either; Williams [193] has the details.

Note that the parsing method is the same for BRC, BC and BCP; just the parse table construction methods differ.

9.3.2 Floyd Productions

Bounded-context parsing steps can be summarized conveniently by using Floyd productions. *Floyd productions* are rules for rewriting a string that contains a marker, Δ, on which the rules focus. A Floyd production has the form $\alpha\Delta\beta$ => $\gamma\Delta\delta$ and means

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 $\mathtt{s}_1$ is the initial state, $\mathtt{s}_g \cdots \mathtt{s}_b$ are the states from

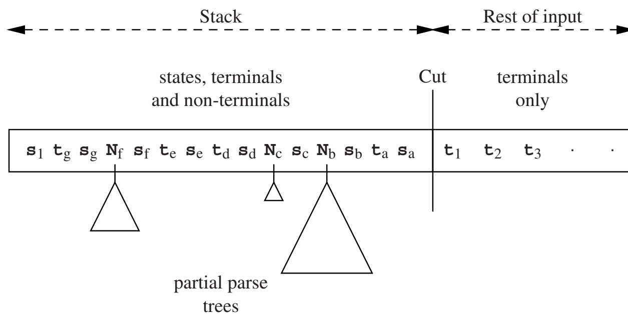


Fig. 9.12. The structure of an LR parse

previous scans, and $\mathtt{s}_a$ 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

```
Ss  -->  a S a | 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. Ss  -->  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 $X \to \cdots Y \bullet Z \cdots$ 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

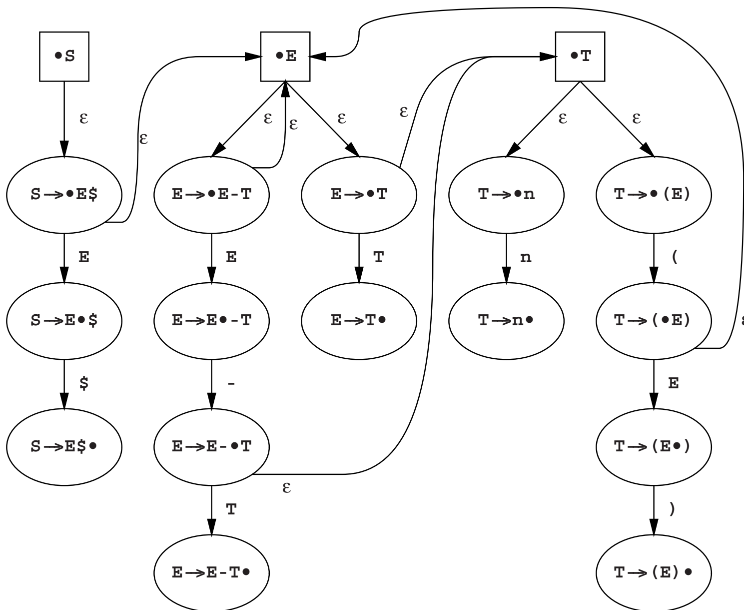


Fig. 9.15. A non-deterministic handle recognizer for the grammar of Figure 9.14

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.

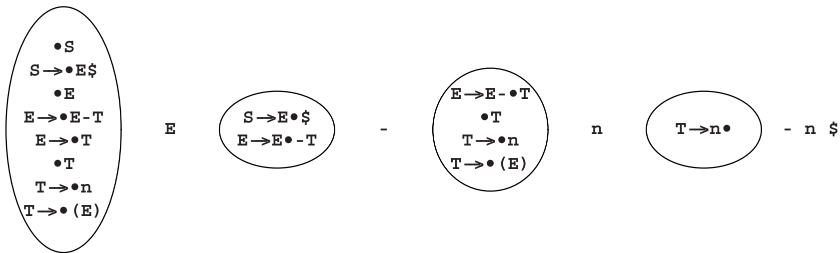


Fig. 9.16. The sets of NFA states while analysing **E-n-n$**

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 FS 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 $X \to \alpha$ 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 "s*N*", "r*N*" or "e". An entry "s*N*" means "shift the input symbol onto the stack and go to state *N*", which is often abbreviated to "shift to *N*". An entry "r*N*" 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).

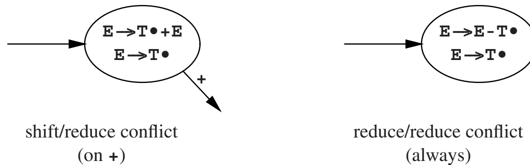


Fig. 9.20. Two types of conflict

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

```
Ss  -->  E $
E   -->  E Q F
E   -->  F
F   -->  a
Q   -->  /
Q   -->  ε
```

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: $\bar{\text{Q}}$, 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✘.

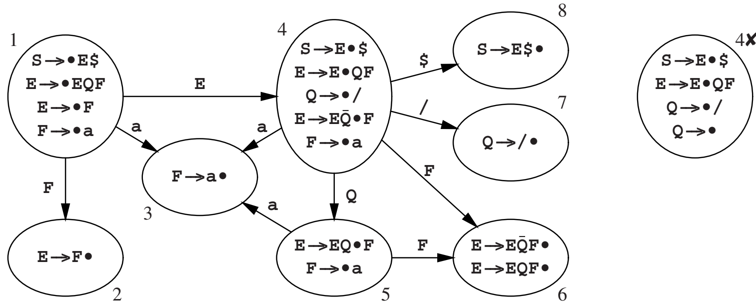


Fig. 9.22. Deterministic ε-LR(0) automaton for the grammar of Figure 9.21

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 $\xrightarrow{\mathtt{E}}$ 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 $A \to \bullet\cdots$ for all grammar rules $A \to \cdots$; 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 $A \to \alpha\bullet t\beta$ in u. For each such item a new item $A \to \alpha t\bullet\beta$ 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]).

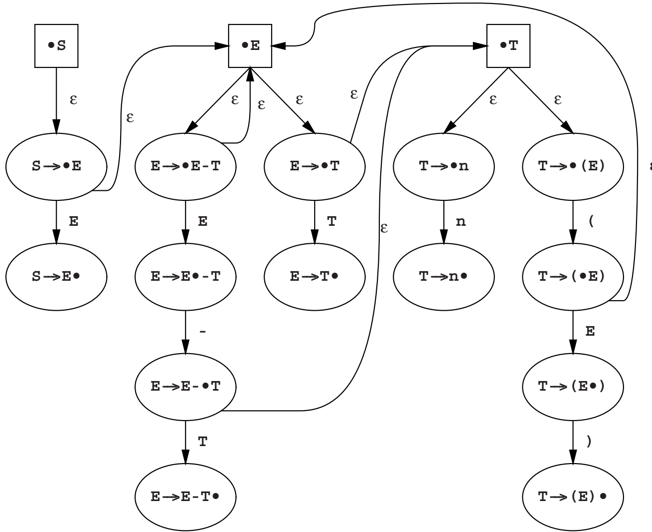


Fig. 9.24. NFA for the grammar in Figure 9.23

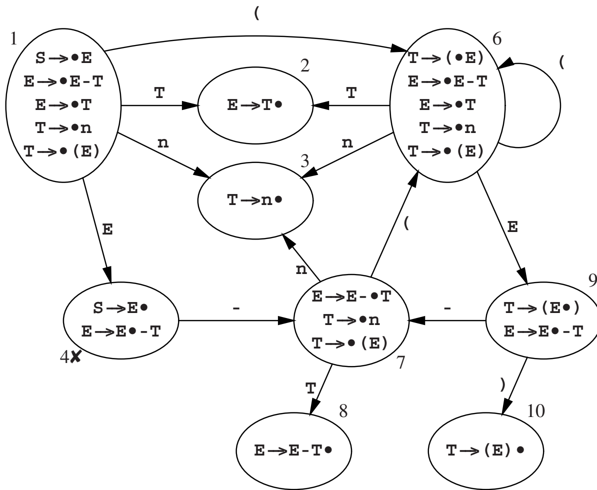


Fig. 9.25. Inadequate LR(0) automaton for the grammar in Figure 9.23

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

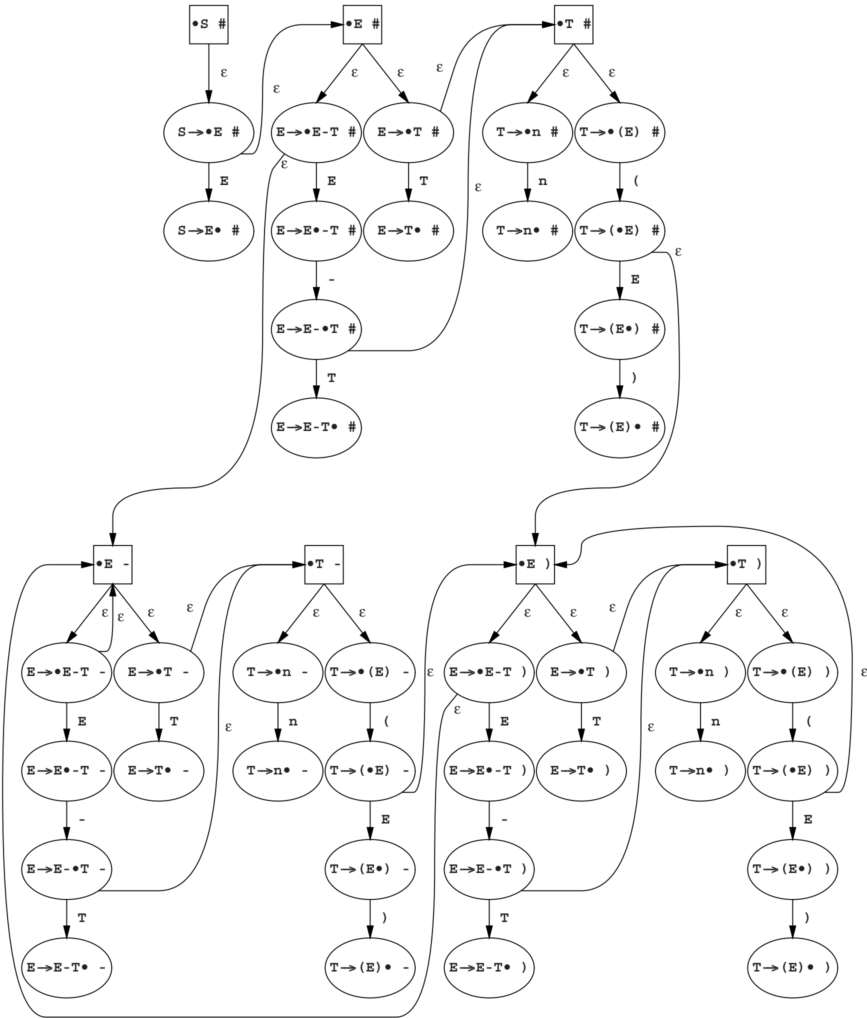


Fig. 9.26. Non-deterministic LR(1) automaton for the grammar in Figure 9.23

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→•E #**. 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:

$P \to \bullet QR$

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)

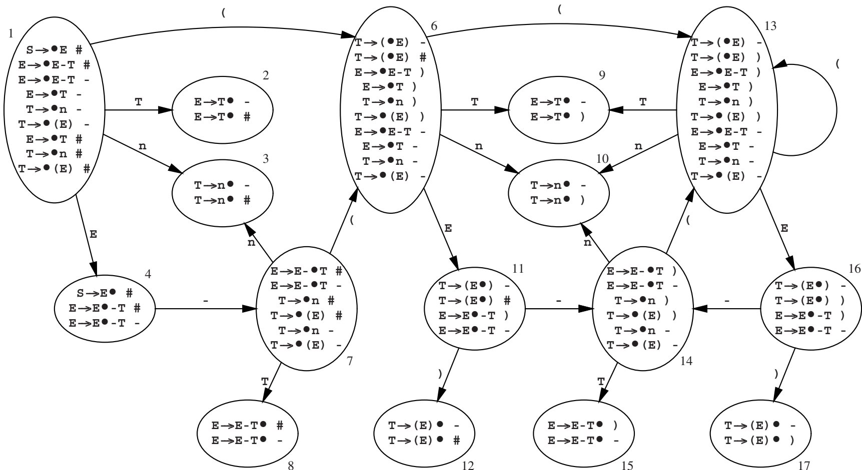


Fig. 9.27. Deterministic LR(1) automaton for the grammar in Figure 9.23

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

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

GOTO

	n	-	(	)	#	S	E	T
1	3		6			accept	4	2
2								
3								
4		7						
6	10		13				11	9
7	3		6					8
8								
9								
10								
11		14		12				
12								
13	10		13				16	9
14	10		13					15
15								
16		14		17				
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**$\rightarrow$**n**:

① **E** ④ **-** ⑦ **T** ⑧ **n**···

and now entry 8 in the ACTION table tells it to reduce again, by **E**$\rightarrow$**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

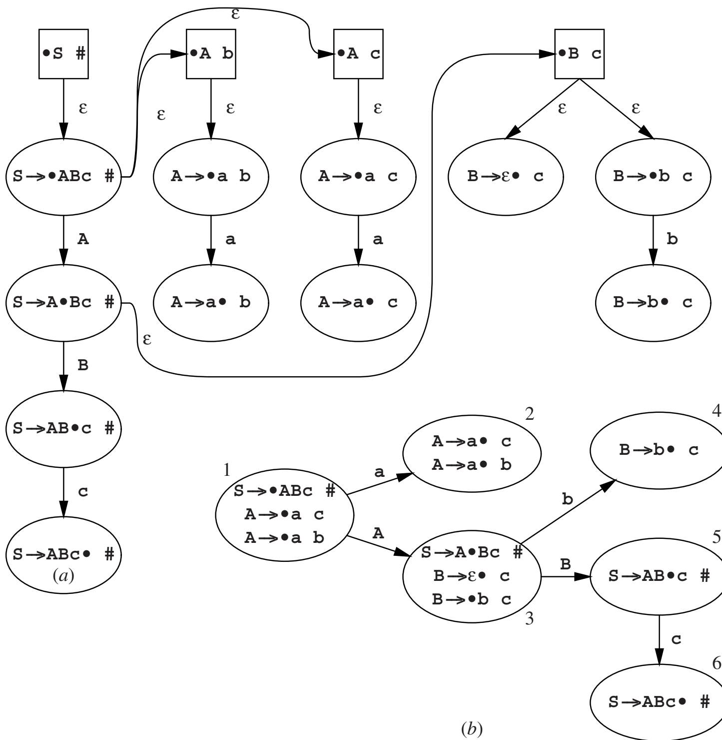


Fig. 9.31. Non-deterministic and deterministic LR(1) automata for Figure 9.30

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 $A \to \alpha \bullet \beta\ \gamma$, 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•
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 repre-

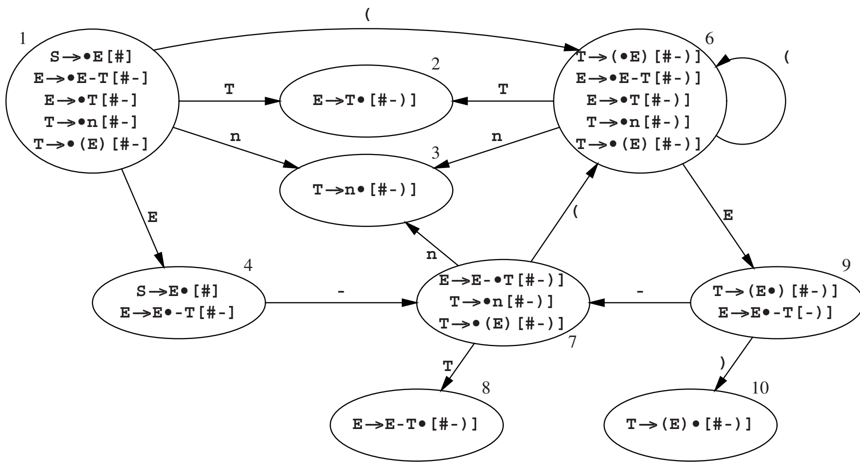


Fig. 9.34. The LALR(1) automaton for the grammar of Figure 9.23

sented 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!

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

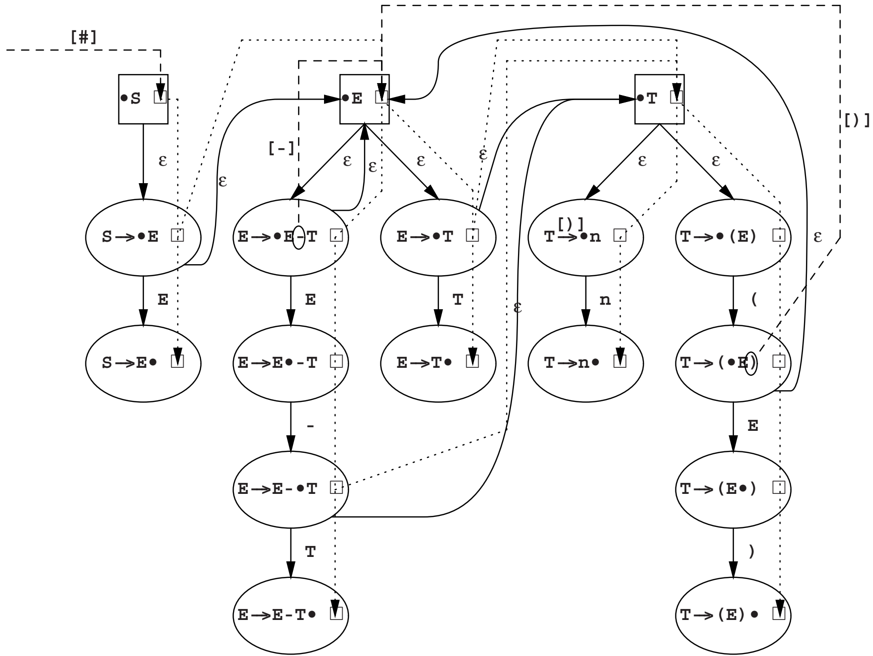


Fig. 9.35. Non-deterministic automaton with channels

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 •*A*□. 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 $P \rightarrow B\bullet ACDe$ □), 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,

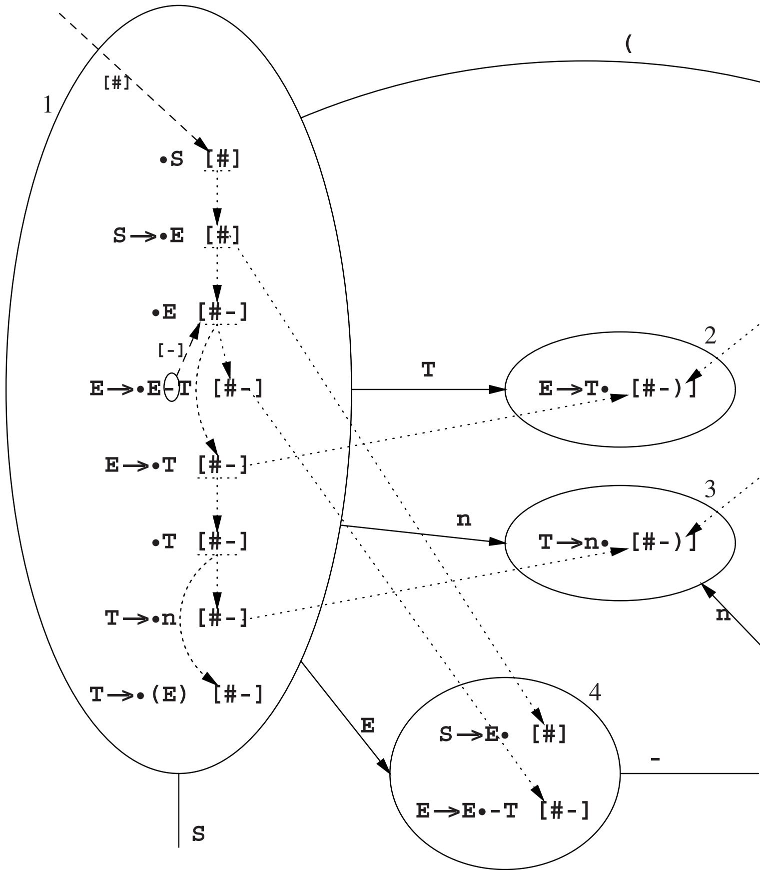


Fig. 9.36. Part of the deterministic automaton with channels (magnified cut)

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: $A \to \omega$ and $B \to \beta A\gamma$, for some, possibly empty sequences of non-terminals and terminals ω, β, and γ. Refer to Figure 9.37.

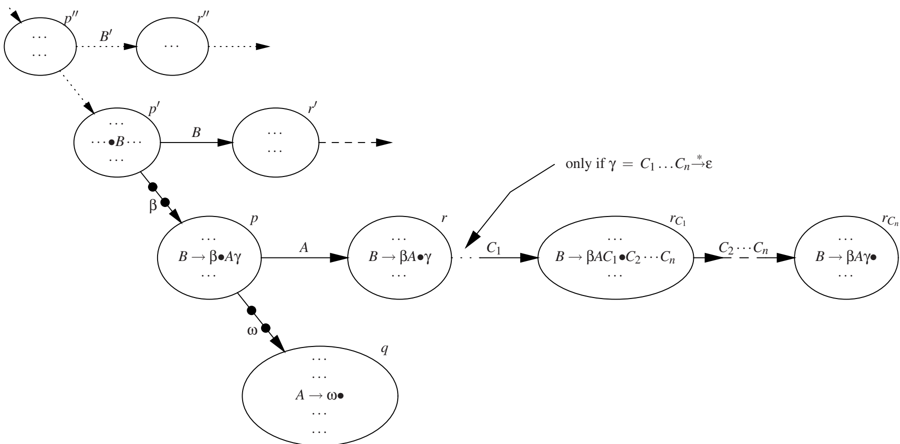


Fig. 9.37. Hunting for LALR(1) look-aheads in an LR(0) automaton — the **lookback** and **includes** relations

Suppose the LR(0) automaton has an inadequate state q with a reduce item $A \to \omega\bullet$, 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 $A \to \omega$). 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 $\{p_1 \stackrel{A}{\to} r_1, \ldots, p_n \stackrel{A}{\to} r_n\}$, where for all p_i we have $p_i \stackrel{\omega}{\to} q$. This defines the so-called **lookback** relation between a pair (state, reduce item) and a transition. One writes $(q,\ A \to \omega\bullet)$ **lookback** $(p_i \stackrel{A}{\to} r_i)$ 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 $(p_i \stackrel{A}{\to} r_i)$ 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 $p \stackrel{A}{\to} r$ (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

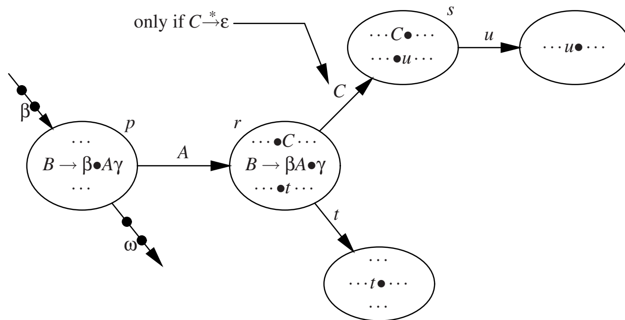


Fig. 9.38. Hunting for LALR(1) look-aheads in an LR(0) automaton — the **directly-reads** and **reads** relations

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:

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

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 $p, p', \ldots,$ $B, B', \ldots, r, r', \ldots, C, C', \ldots,$ and $s, s', \ldots,$ 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

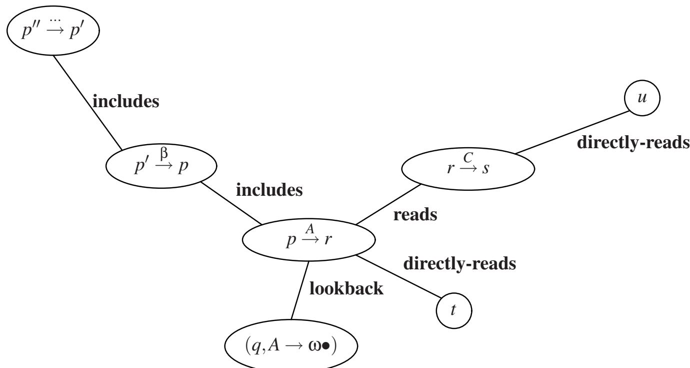


Fig. 9.39. Hunting for LALR(1) look-aheads in an LR(0) automaton — the relations graph

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:

```
①E④  →  ①T②
①E④  →  ①E④ ④-⑦ ⑦T⑧
```

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 $A \to BcD$ is trans-

```
①S◊  →  ①E④
①E④  →  ①E④ ④-⑦ ⑦T⑧  |  ①T②
⑥E⑨  →  ⑥E⑨ ⑨-⑦ ⑦T⑧  |  ⑥T②
①T②  →  ①n③
⑥T②  →  ⑥(⑥ ⑥E⑨ ⑨)⑩  |  ⑥n③
⑦T⑧  →  ⑦(⑥ ⑥E⑨ ⑨)⑩  |  ⑦n③
```

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

formed into a new grammar rule $(s_1)A(s_x) \to (s_1)B(s_2)\ (s_2)c(s_3)\ (s_3)D(s_4)$, 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).

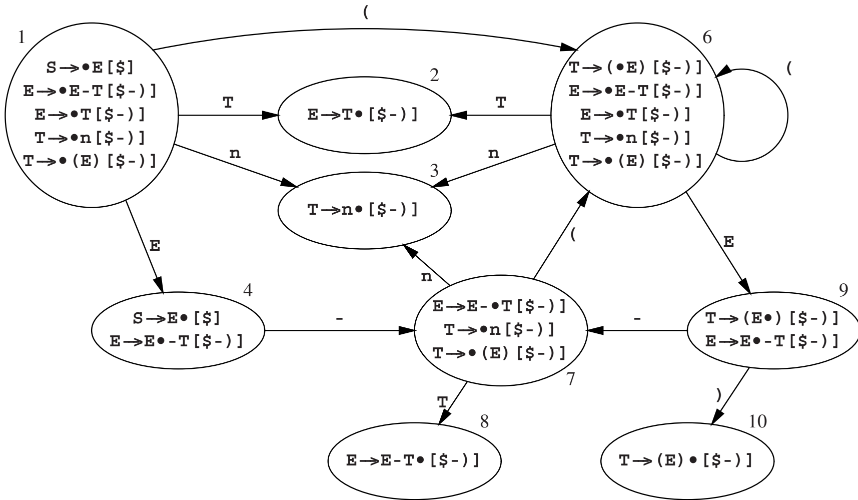


Fig. 9.41. SLR(1) automaton for the grammar of Figure 9.23

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.

Systems with built-in conflict resolvers are a mixed blessing. On the one hand they allow very weak or even ambiguous grammars to be used (see for example, Aho, Johnson and Ullman [335]). This can be a great help in formulating grammars for difficult and complex analysis jobs; see, for example, Kernighan and Cherry [364], who make profitable use of automatic conflict resolution for the specification of typesetter input.

On the other hand a system with built-in conflict resolvers may impose a structure on the input where there is none. Such a system no longer corresponds to any grammar-like sentence-generating mechanism, and it may be very difficult to specify exactly what strings will be accepted and with what structure. How severe a drawback this is depends on the application and of course on the capabilities of the parser generator user.

It is to a limited extent possible to have dynamic (parse-time) conflict resolvers, as in the LL case (Section 8.2.5.3). Such a conflict resolver is called in a context that is still under construction, which complicates its use, but in simple cases its working can be understood and predicted. McKenzie [86] describes an extension of *yacc* that supports dynamic conflict resolvers, among other things.

Some experiments have been made with interactive conflict resolvers, which consult the user of the parser when a conflict actually arises: a large chunk of text around the conflict point is displayed and the user is asked to resolve the conflict. This is useful in, for example, document conversion; see Share [365].

9.10 Further Developments of LR Methods

Although the LALR(1) method as explained in Section 9.7 is quite satisfactory for most applications, a number of extensions to and improvements of the LR methods have been studied. The most important of these will be briefly explained in this section; for details see the literature, (Web)Section 18.1.4 and the original references.

For methods to speed up LR parsing by producing executable parser code see Section 17.2.2.

9.10.1 Elimination of Unit Rules

Many rules in practical grammars are of the form $A \rightarrow B$; examples can be found in Figures 2.10, 4.6, 5.3, 7.8, 8.7, 9.42 and many others. Such rules are called unit

```
Metre  -->  Iambic | Trochaic | Dactylic | Anapestic
```

Fig. 9.42. A (multiple) unit rule

rules, single rules, or chain rules. They generally serve naming purposes only and have no semantics attached to them. Consequently, their reduction is a matter of stack manipulation and state transition only, to no visible purpose for the user. Such

"administrative reductions" can take a considerable part of the parsing time (50% is not unusual). Simple methods to short-cut such reductions are easily found (for example, removal by systematic substitution) but may result in an exponential increase in table size. Better methods were found but turned out to be complicated and to impair the error detection properties of the parser. That problem can again be corrected, at the expense of more complication. See Heilbrunner [64] for a thorough treatment and Chapman [71] for much practical information.

Note that the term "elimination of unit rules" in this case is actually a misnomer: the unit rules themselves are not removed from the grammar, but rather their effect from the parser tables. Compare this to the actual elimination of unit rules in Section 4.2.3.2.

Actually unit rule elimination is a special case of stack activity reduction, which is discussed in the next section. But it was recognized earlier, and a separate body of literature exists for it.

9.10.2 Reducing the Stack Activity

Consider the stack of an LR parser, and call the state on top of the stack s_t. Now we continue the parser one step with proper input and we suppose this step stacks a token X and another state s_u, and we suppose that $s_u \neq s_t$, as will normally happen. Now, rather than being satisfied with the usual top stack segment $s_t\ X\ s_u$, we collapse this into one new state, $s_t + s_u$, which now replaces the original s_t. This means two things. First, we have lost the symbol X, and with it the possibility to construct a parse tree, so we are back to constructing a recognizer. But second, and more importantly, we have replaced an expensive stacking operation by a cheap state transition.

We can repeat this process of appending new states to the top state until one of two things happens: a state already in it is appended for the second time, or the original state s_t gets popped and we are left with an empty state. Only at that moment do we resume the normal stacking and unstacking operation of an LR parser.

When doing so for all acceptable inputs, we meet all kinds of compound states, all with s_t on the left, and many pairs are connected by transitions on symbols, terminal and non-terminal ones. Together they form a finite-state automaton. When we are forced to resume normal LR operation, it is very likely that we will find a state different from s_t on top, say s_x. We can then repeat the process for s_x and obtain another FSA.

Continuing this way we obtain a set of FSAs connected by stacking and unstacking LR operations. Using these FSAs instead of doing all the stack manipulation hidden in them greatly reduces the stack activity of the parser. Such a parser is called *reduction-incorporated* (*RI*).

In a traditional LR parser the gain in speed will almost certainly be outweighed by the disadvantage of not being able to construct a parse tree. Its great advantage lies in situations in which stack activity is expensive. Examples are the use of an LR parser as a subparser in a GLR parser (Chapter 11), where stack activity involves graph manipulation, and in parallel parsing (Chapter 14), where stack activity may require process communication.

The details of the algorithm are pretty complicated; descriptions are given by Aycock and Horspool [176] and Scott and Johnstone [100]. The resulting tables can be very large, even for every-day grammars.

9.10.3 Regular Right Part Grammars

As shown in Section 2.3.2.4, there are two interpretations of a regular right-hand side of a rule: the recursive and the iterative interpretation. The recursive interpretation is no problem: for a form like A^+ anonymous non-terminals are introduced, the reduction of which entails no semantic actions. The burden of constructing a list of the recognized *A*s lies entirely on the semantic routines attached to the *A*s.

The iterative interpretation causes more problems. When an A^+ has been recognized and is about to be reduced, the stack holds an indeterminate number of *A*s:

$$\cdots A \cdots AAA \quad |$$

The right end of the handle has been found, but the left end is doubtful. Scooping up all *A*s from the right may be incorrect since some may belong to another rule; after all, the top of the stack may derive from a rule $P \rightarrow QAAA^+$. A possible solution is to have for each reducing state and look-ahead a FS automaton which scans the stack backwards while examining states in the stack to determine the left end and the actual rule to reduce to. The part to be reduced (the handle) can then be shown to a semantic routine which can, for example, construct a list of *A*s, thereby relieving the *A*s from a task that is not structurally theirs. The resulting tables can be enormous and clever algorithms have been designed for their construction and reduction. See for example, LaLonde [62], Nakata and Sassa [69, 74], Shin and Choe [90], Fortes Gálvez, [91], and Morimoto and Sassa [97]. Kannapinn [99] has given a critical analysis of many algorithms for LR and LALR parse table creation for EBNF grammars (in German).

9.10.4 Incremental Parsing

In incremental parsing, the structured input (a program text, a structured document, etc.) is kept in linear form together with a parse tree. When the input is (incrementally) modified by the user, for example by typing or deleting a character, it is the task of the incremental parser to update the corresponding parse tree, preferably at minimum cost. This requires serious measures inside the parser, to quickly determine the extent of the damage done to the parse tree, localize its effect, and take remedial steps. Formal requirements for the grammar to make this easier have been found. See for example, Degano, Mannucci and Mojana [330] and many others in (Web)Section 18.2.8.

9.10.5 Incremental Parser Generation

In incremental parser generation, the parser generator keeps the grammar together with its parsing table(s) and has to respond quickly to user-made changes in the grammar, by updating and checking the tables. See Horspool [80], Heering, Klint and Rekers [83], Horspool [84] and Rekers [347].

9.10.6 Recursive Ascent

In Sections 8.2.6 and 8.5 we have seen that an LL parser can be implemented conveniently using recursive descent. Analogously, an LR parser can be implemented using *recursive ascent*, but the required technique is not nearly as obvious as in the LL case. The key idea is to have the recursion stack mimic the LR parsing stack. To this end there is a procedure for each state; when a token is to be shifted to the stack, the procedure corresponding to the resulting state is called instead. This indeed constructs the correct recursion stack, but causes problems when a reduction has to take place: a dynamically determined number of procedures has to return in order to unstack the right-hand side. A simple technique to achieve this is to have two global variables, one, Nt, holding the non-terminal recognized and the second, l, holding the length of the right-hand side. All procedures will check l and if it is non-zero, they will decrease l by one and return immediately. Once l is zero, the procedure that finds that situation will call the appropriate state procedure based on Nt. For details see Roberts [78, 81, 87] and Kruseman Aretz [77]. The advantage of recursive ascent over table-driven is its potential for high-speed parsing.

9.10.7 Regular Expressions of LR Languages

In Section 9.6.3 we mentioned that any LR(k) language can be obtained as a regular expression, the elements of which are LR(0) languages. The opposite is even stronger: regular expressions over LR(0) languages can describe more than the LR(k) languages. An immediate example is the inherently ambiguous language $\mathtt{a}^m\mathtt{b}^n\mathtt{c}^n \cup \mathtt{a}^p\mathtt{b}^p\mathtt{c}^q$ discussed on page 64. It is produced by the regular expression

$$\mathcal{L}_{\mathtt{a}}^{*}\mathcal{L}_{\mathtt{bc}} | \mathcal{L}_{\mathtt{ab}}\mathcal{L}_{\mathtt{c}}^{*}$$

where the language $\mathcal{L}_{\mathtt{a}}$ is produced by the simplest grammar in this book, $\mathtt{S}\rightarrow\mathtt{a}$, $\mathcal{L}_{\mathtt{bc}}$ by $\mathtt{S}\rightarrow\mathtt{bSc}\,|\,\varepsilon$, and similarly for $\mathcal{L}_{\mathtt{ab}}$ and $\mathcal{L}_{\mathtt{c}}$. It is easy to see that each of these grammars is LR(0).

Bertsch and Nederhof [96] show that a linear-time parser can be constructed for regular expressions over LR(k) languages. Unfortunately the algorithm is based on descriptions of the languages by pushdown automata rather than CF grammars, and a transformation back to CF grammars would be very complicated. Some details are provided in Section 12.3.3.2, where a similar technique is used for linear-time substring parsing of LR languages.

9.11 Getting a Parse Tree Grammar from LR Parsing

Getting a parse tree grammar from LR parsing is similar to getting one from LL parsing (Section 8.4): each time one makes a “serious” decision (prediction, reduction) one generates a grammar rule for it. As in the LL case, LR parsing produces a parse tree grammar rather than a parse forest grammar.

We consider parsing `n-n` with the table of Figure 9.18. All non-terminals are numbered using the same increasing counter. After a single shift we have the configuration

```
① n ③                                  - n $
```

The moment we reduce `n` to `T`, we produce a rule `T_1`$\rightarrow$`n`, and push `T_1` on the stack:

```
① T_1 ②                                - n $
```

The next step reduces the `T` to `E`; this produces a rule `E_2`$\rightarrow$`T_1` and the configuration

```
① E_2 ②                                - n $
```

Continuing this process we obtain the parse tree grammar

```
T_1  ->  n
E_2  ->  T_1
T_3  ->  n
E_4  ->  E_2 - T_3
S_5  ->  E_4 $
```

and the final datum yielded by the parsing process is that `S_5` is the start symbol of the parse tree grammar.

Note that it is possible to number the input tokens with their positions and to follow where they go in the parse tree grammar:

$$\begin{array}{lll} \texttt{T_1} & \rightarrow & \texttt{n}_1 \\ \texttt{E_2} & \rightarrow & \texttt{T_1} \\ \texttt{T_3} & \rightarrow & \texttt{n}_3 \\ \texttt{E_4} & \rightarrow & \texttt{E_2}\ \texttt{-}_2\ \texttt{T_3} \\ \texttt{S_5} & \rightarrow & \texttt{E_4}\ \texttt{\$}_4 \end{array}$$

This is useful when semantics is attached to the input tokens.

Again the grammar is clean. It has no undefined non-terminals: each non-terminal included in a right-hand side during a reduction comes from the stack, and was defined in a previous reduction. It has no unreachable non-terminals either: each left-hand side non-terminal created in a reduction is put on the stack, and will later be included in some right-hand side during a subsequent reduction, except for the start symbol, which is reachable by definition.

9.12 Left and Right Contexts of Parsing Decisions

At the beginning of Chapter 7 we indicated that stacks in bottom-up parsing can be described by regular grammars, and we are now in a position to examine this phenomenon in more detail, by considering two non-obvious properties of an LR automaton: the left context of a state and the right context of an item.

9.12.1 The Left Context of a State

The left context of a state is easy to understand: it is the set of all sequences of symbols, terminals and non-terminals, that lead to that state. Although this set is usually infinitely large, it can be represented by a regular expression. It is easy to see that, for example, the left context of state 4 in the LR automaton in Figure 9.17 is **E**, but more work is needed to obtain the left context of, say, state 9. To find all paths that end in state 9 we proceed as follows. We can create the path to state 9 if we know the path(s) to state 6 and then append an **E**. This gives us one rule in a left-regular grammar: `P_9→P_6 E`, where `P_6` and `P_9` are the paths to states 6 and 9, respectively. Now there are three ways to get to state 6: from 1, from 6 and from 7, all through a `(`. This gives us three rules: `P_6→P_1(`, `P_6→P_6(`, `P_6→P_7(`. Continuing in this way we can construct the entire left-context grammar of the LR automaton in Figure 9.17. It is shown in Figure 9.43, and we see that it is left-regular.

```
P_1 → ε        P_4 → P_1 E   P_7 → P_4 -
P_2 → P_1 T    P_5 → P_4 $   P_7 → P_9 -
P_2 → P_6 T    P_6 → P_1 (   P_8 → P_7 T
P_3 → P_1 n    P_6 → P_6 (   P_9 → P_6 E
P_3 → P_6 n    P_6 → P_7 (   P_10 → P_9 )
P_3 → P_7 n
```

Fig. 9.43. Left-context grammar for the LR(0) automaton in Figure 9.17

We can now apply the transformations shown in Section 5.4.2 and Section 5.6 to obtain regular expressions for the non-terminals. This way we find that indeed the left context of state 4 is **E** and that that of state 9 is `[(|E-(][(|E-(]`*`E`. This expression simplifies to `[(|E-(]`$^+$`E`, which makes sense: it describes a sequence of one or more `(` or `E-(`, followed by an **E**. The first `(` or `E-(` brings us to state 6, any subsequent `(`s and `E-(`s bring us back to state 6, and the final **E** brings us to state 9.

Now the connection with the stack in an LR parser becomes clear. Such a stack can only consist of a sequence which leads to a state in the LR automaton; for example, it could not be `(-`, since that leads nowhere in Figure 9.17, though it could be `(E-` (which leads to state 7). In short, the union of all left contexts of all states describes the complete set of stack configurations of the LR parser.

All stack configurations in a given *P_s* end in state *s* and thus lead to the same parsing decision. LR(1) automata have more states than LR(0) automata, and thus more left context sets. For example, the LR(1) automaton in Figure 9.27 remembers whether it is working on the outermost expression (in which case a **#** may follow) or on a nested expression; the LR(0) automaton in Figure 9.17 does not. But the set of all stack configurations P_* is the same for LR(0) and LR(1), because they represent all open parts in a rightmost production, as explained in Section 5.1.1.

9.12.2 The Right Context of an Item

The right context of a state is less easy to understand: intuitively it is the set of all strings that are acceptable to an LR parser in that state, but that set differs considerably from the left context sketched above.

First it is a context-free language rather than a regular one. This is easy to see when we consider an LR parser for a CF language: any string in that language is acceptable as the right context of the initial state.

Second, it contains terminals only; there are no non-terminals in the rest of the input, to the right of the gap. Yet it is clear that the right context of an item is not just an unrestricted set of strings, but follows precisely from the CF grammar C and the state S, and we would like to capture these restrictions in a grammar. This is achieved by constructing a regular grammar G_S for the right context which still contains undeveloped non-terminals from C, similar to the left context grammar. The set of terminal strings acceptable after a state is then obtained by replacing these non-terminals by their terminal productions in C; this introduces the CF component. More precisely: each (regular) terminal production T_S of the grammar G_S is a start sentential form for grammar C; each combination of T_S and C produces a (CF) set of strings that can figure as rest of input at S.

There is another, perhaps more surprising, difference between left and right contexts: although the left contexts of all items in an LR state are the same, their right contexts can differ. The reason is that the same LR state can be reached by quite different paths through the grammar. Each such path can result in a different item in that state and can carry a different prediction of what will happen on the other side of the item. A trivial example occurs in the grammar

```
Ss  -->  a B c
S   -->  a D e
B   -->  ε
D   -->  ε
```

The state reached after shifting over an **a** contains

```
S-->a•Bc
S-->a•De
B-->•
D-->•
```

and it is clear that the right context of the item **B-->•** is **c** and that of **D-->•** is **e**. This example already alerts us to the relationship between right contexts and look-ahead symbols. Like the latter (Section 9.6.2) right contexts exist in an item and a dot variety. The item right context of **S-->a•Bc** is ε; its dot right context is **Bc**. Item right contexts are easier to compute but dot right contexts are more important in parsing.

We shall start by constructing the regular grammar for item right contexts for the automaton in Figure 9.17, and then derive dot right contexts from it. Since the right contexts are item-specific we include the names of the items in the names of the non-terminals that describe them. We use names of the form **F**_s{I} for the set of strings that can follow item I in state s in sentential forms during rightmost derivation.

As we have seen in Section 9.5, items can derive from items in the same state or from a parent state. An example of the first type is E->•E-T in state 6. It derives through the ε-moves T->(•E) $\xrightarrow{\varepsilon}$ •E $\xrightarrow{\varepsilon}$ E->•E-T and E->•E-T $\xrightarrow{\varepsilon}$ •E $\xrightarrow{\varepsilon}$ E->•E-T in the non-deterministic automaton of Figure 9.15 from both T->(•E) and E->•E-T in state 6. An example of the second type is T->(•E) in state 6, deriving in three ways from the T->•(E) in states 1, 6 and 7, through the transition T->•(E) $\xrightarrow{(}$ T->(•E) in Figure 9.15.

If the item E->•E-T originates from T->(•E), its right context consists of the) which follows the E in T->(•E); this gives one rule for F_6{E->•E-T}:

```
F_6{E->•E-T} -> ) F_6{T->(•E)}
```

If the item originates from T->•E-T, its right context consists of the -T which follows the E in T->•E-T; this gives the second rule for F_6{E->•E-T}:

```
F_6{E->•E-T} -> -T F_6{E->•E-T}
```

The general rule is: F_$s$$\{A \to \bullet\alpha\}$ -> γ F_$s$$\{X \to \beta\bullet A\gamma\}$ for an ε-transition $\{X \to \beta\bullet A\gamma\} \xrightarrow{\varepsilon} \{\bullet A\} \xrightarrow{\varepsilon} \{A \to \bullet\alpha\}$, for each state s in which the item $\{X \to \beta\bullet A\gamma\}$ occurs.

A shift over a token does not change the right context of an item: during a shift over a (from state 1 to state 6, the item T->•(E) changes into T->(•E), but its right context remains unaffected. This is expressed in the rule

```
F_6{T->(•E)} -> F_1{T->•(E)}
```

The general rule is: F_$r$$\{A \to \alpha t\bullet\beta\}$ -> F_$s$$\{A \to \alpha\bullet t\beta\}$ for a transition $\{A \to \alpha\bullet t\beta\} \xrightarrow{t} \{A \to \alpha t\bullet\beta\}$.

Repeating this procedure for all ε-moves and shifts in Figure 9.17 gives us the (right-regular) grammar for the right contexts; it is shown in Figure 9.44. Note that the two ways of propagating right context correspond with the two ways of propagating the one-token look-ahead in LR(1) parsing, as explained on page 293.

Again applying the transformations from Section 5.4.2 we can obtain regular expressions for the non-terminals. For example, the item right context of E->•E-T in state 6 is [)* [-T|)]]*)*)[-T]*$ which simplifies to [-T|)]*)[-T]*$. Again this makes sense: the prediction after that item is a sequence of -Ts and)s, with at least one), since to arrive at state 6, the input had to contain at least one (.

Finding dot right contexts is now simple: the dot right context of E->•E-T in state 6, D_6{E->•E-T}, is of course just E-T F_6{E->•E-T}. The general rule is: D_$s$$\{A \to \alpha\bullet\beta\}$ -> β F_$s$$\{A \to \alpha\bullet\beta\}$ for all items.

For a thorough and formal analysis of right contexts see Seyfarth and Bermudez [93].

9.13 Exploiting the Left and Right Contexts

There are many ways to exploit the left and right contexts as determined above. We will discuss here three techniques. The first, DR(k) parsing, uses knowledge of the

```
F_1{S->•E$} -> ε
F_1{E->•E-T} -> -T F_1{E->•E-T}
F_1{E->•E-T} -> $ F_1{S->•E$}
F_1{E->•T} -> -T F_1{E->•E-T}
F_1{E->•T} -> $ F_1{S->•E$}
F_1{T->•(E)} -> F_1{E->•T}
F_1{T->•n} -> F_1{E->•T}
F_2{E->T•} -> F_1{E->•T}
F_2{E->T•} -> F_6{E->•T}
F_3{T->n•} -> F_1{T->•n}
F_3{T->n•} -> F_6{T->•n}
F_3{T->n•} -> F_7{T->•n}
F_4{S->E•$} -> F_1{S->•E$}
F_4{E->E•-T} -> F_1{E->•E-T}
F_5{S->E$•} -> F_4{S->E•$}
F_6{E->•E-T} -> ) F_6{T->(•E)}
F_6{E->•E-T} -> -T F_6{E->•E-T}
F_6{E->•T} -> -T F_6{E->•E-T}
F_6{E->•T} -> ) F_6{T->(•E)}
F_6{T->(•E)} -> F_1{T->•(E)}
F_6{T->(•E)} -> F_6{T->•(E)}
F_6{T->(•E)} -> F_7{T->•(E)}
F_6{T->•(E)} -> F_6{E->•T}
F_6{T->•n} -> F_6{E->•T}
F_7{E->E-•T} -> F_4{E->E•-T}
F_7{E->E-•T} -> F_9{E->E•-T}
F_7{T->•(E)} -> F_7{E->E-•T}
F_7{T->•n} -> F_7{E->E-•T}
F_8{E->E-T•} -> F_7{E->E-•T}
F_9{E->E•-T} -> F_6{E->•E-T}
F_9{T->(E•)} -> F_6{T->(•E)}
F_10{T->(E)•} -> F_9{T->(E•)}
```

Fig. 9.44. Right-regular right-context grammar for the LR(0) automaton in Figure 9.17

left context to reduce the required table size drastically, while preserving full LR(k) parsing power. The second, LR-regular, uses the full right context to provide optimal parsing power, but the technique does not lead to an algorithm, and its implementation requires heuristics and/or handwaving. The third, LAR(k) parsing, is a tamed version of LR-regular, which yields good parsers for a large class of unambiguous grammars. An even more extensive application of the contexts is found in the chapter on non-canonical parsing, Chapter 10, where the right context is explicitly improved by doing reductions in it. And there is no reason to assume that this exhausts the possibilities.

9.13.1 Discriminating-Reverse (DR) Parsing

As Figure 9.12 shows, an LR parser keeps states alternatingly between the stacked symbols. Actually, this is an optimization; we could omit the states, but that would force us to rescan the stack after each parse action, to reestablish the top state, which would be inefficient. Or would it? Consider the sample parser configuration on page 283, which was based on the grammar of Figure 9.14 and the handle recognizer of Figure 9.17, and which we repeat here without the states:

```
# E - n ❍                    - n $
```

We have also added a bottom-of-stack marker, **#**, which can be thought of as caused by stacking the beginning of the input. There can be no confusion with the end-of-input marker **#**, since the latter will never be put on the stack.

When we look at the above configuration, we find it quite easy to guess what the top state, indicated by ❍, must be. The last shift was over an **n**, and although there are three arrows marked **n** in Figure 9.17, they all point to the same state, ③, so we can be certain we are looking at

```
# E - n ③                          - n $
```

Since ③ is a reduce state, we can confidently reduce the **n** to **T**. So the loss of the left context did not do any harm here; but, as the reader might expect, that is not going to last.

After the reduction we have the configuration

```
# E - T ❍                          - n $
```

and now we have a problem. There are again three arrows marked **T** in Figure 9.17, but they do not all point to the same state; two point to ② and one points to ⑧, so we seem none the wiser. But we know how to handle situations in which there are only a finite number of possibilities: we put them all in a state, and progress with that state as our knowledge. The state is

```
①: ②
⑥: ②
⑦: ⑧
```

and it represents our knowledge between the **-** and the **T** in the configuration; it says: if we are now in LR state ① or ⑥, the top state was ②, and if we are now in state ⑦, the top state was ⑧. Such a state is called a *DR* state, for *Discriminating Reverse* state (Fortes Gálvez [89]).

When we now look backwards on the stack, we see a **-**; the LR states ① and ⑥ do not have incoming arrows marked **-**, so we cannot be in one of these, but state ⑦ has, coming from ④ and ⑨. So our DR state between the **E** and the **-** is

```
④: ⑧
⑨: ⑧
```

which says if we are now in LR state ④, the top state was ⑧, and if we are now in state ⑨, the top state was ⑧. Here something surprising has happened: even though we do not know in which LR state we are, we now know that the top of the stack was ⑧, which is the answer we were looking for! This gives us the configuration

```
# E - T ⑧                          - n $
```

in which we reduced the **E-T** to **E**. We have now reproduced the parsing example of page 283 without states on the stack and without rescanning the entire stack for each parsing action. Whether that is something worth striving for is less than clear for the moment, but we will see that the technique has other benefits.

More generally, suppose we are in a DR state d

l_1 **:** t_1
...
l_k **:** t_k

where $l_1 \cdots l_k$ are LR states and the $t_1 \cdots t_k$ are the top-of-stack states implied by them, and suppose we have the stack symbol s on our left. Now we want to compute the DR state to the left of s, one step back on the stack. To do so we go through all transitions of the form $p_1 \xrightarrow{s} p_2$ in the LR handle recognizer, and for each transition

that has an p_2 equal to an l_j, we insert the form p_1: t_j in a new DR state e. This is reasonable because if we were in LR state p_1 to the left of the s, then moving over the s would bring us in LR state p_2, and that would imply that the top of the stack is t_j. In this way we obtain a transition in a *DR automaton*: $d \xrightarrow{s} e$, or more graphically, $e \xleftarrow{s} d$. This transition carries our knowledge about the top of the stack in our present position over the symbol s to the left.

We can compute the complete DR automaton by performing this step for all possible stack symbols, starting from the initial state of the DR automaton

t_1: t_1
···
t_k: t_k

which of course says that if we are in LR state t_j on the top of the stack, then the top-of-stack state is t_j. It is always good to see a difficult concept reduced to a triviality. States in which all $t_1 \cdots t_k$ are equal are final states, since they unequivocally tell us the top-of-stack state. The DR automaton generation process is guaranteed to terminate because there are only a finite number of DR states possible. The DR automaton for the LR automaton of Figure 9.17 is shown in Figure 9.45.

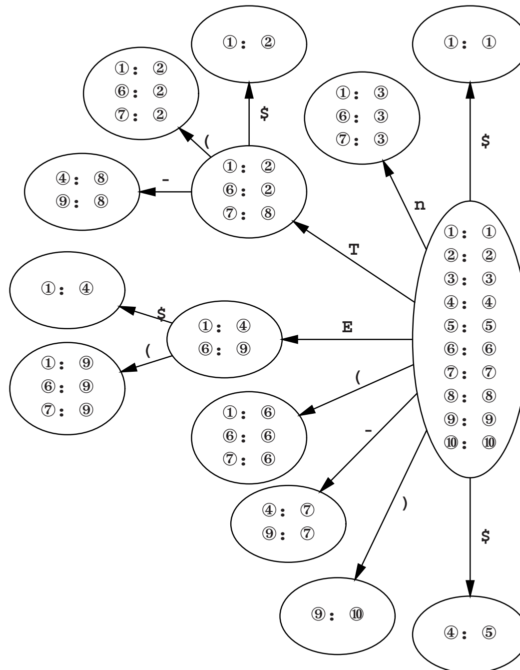


Fig. 9.45. DR automaton for the LR automaton of Figure 9.17

One thing we immediately notice when looking at the graph in Figure 9.45 is that it has no loops: at most two steps backwards suffice to find out which parsing action is called for. We have just shown that the grammar of Figure 9.14 is BRC(2,0)!

But there are more important things to notice: now that we have the transition diagram in Figure 9.45 we can discard the GOTO table of the LR parser (but of course we have to keep the ACTION table). That looks like a meager advantage: the DR automaton has 14 states and the LR(0) automaton only 10. But DR automata have an interesting property, already showing up in Figure 9.45: the first fan-out is equal to the number of symbols, the second fan-out is usually a modest number, the third fan-out a very modest number and in many DR tables there is no fourth fan-out. This is understandable, since each step to the left tends to reduce the uncertainty. Of course it is possible that some DR parser will occasionally dig unboundedly deep back in the stack, but such operations are usually controlled by a simple loop in the DR automaton (see Problem 9.21), involving only a few DR states.

Compared to GOTO tables the DR automata are very compact, and, even better, that property holds more or less independently of the type of LR table used: going from LR(0) to LR(1) to LR(2) tables, each one or more orders of magnitude larger than the previous, the corresponding DR automaton only grows minimally. So we can afford to use full LR(k) tables and still get a very small replacement for the GOTO table! We still need to worry a bit about the ACTION table, but almost all of its entries are “shift” or “error”, and it yields readily to table compression techniques. DR parsing has been used to create LR(1) parsers that are substantially smaller than the corresponding LALR(1) tables. The price paid for these smaller tables is an increased parse time caused by the stack scanning, but the increase is very moderate.

The reader may have noticed that we have swept two problems under the rug in the above explanation: we needed the large LR table to obtain the small DR table, a problem similar to the construction of LALR(1) parsers without generating the full LR(1) tables; and we ignored look-aheads. Solving these problems is the mainstay of DR parser generation; detailed solutions are described by Fortes Gálvez [92, 95]. The author also proves that parse time is linear in the length of the input, even if the parser sometimes has to scan the entire stack [95, Section 7.5.1], but the proof is daunting. A generalized version of DR parsing is reported by Fortes Gálvez et al. [179] and a non-canonical version by Farré and Fortes Gálvez [207, 209].

Kannapinn [99] describes a similar system, which produces even more compact parsers by first reducing the information contents of the LR(1) parser, using various techniques. This reduction is, however, at the expense of the expressive power, and for stronger reductions the technique produces smaller parsers but can handle fewer grammars, thus defining a number of subclasses of LR(1).

9.13.2 LR-Regular

The right context can be viewed as a kind of super-look-ahead, which suggests that it could be a great help in resolving inadequate states; but it is not particularly easy to make this plan work. The basic idea is simple enough: whenever we meet an inadequate state in the parse table construction process, compute the right contexts of

the offending items as described in the previous section. If the two contexts describe disjunct sets, they can serve to resolve the conflict at parse time by finding out to which of the two sets the rest of the input belongs. If the two contexts do not exclude each other, the plan does not work for the given grammar. (See Figure 9.13 for a simple unambiguous grammar for which this technique clearly will not work.)

This requires us to solve two problems: deciding whether the two dot right contexts are disjunct, and checking the rest of the input against both contexts. Both are serious problems, since the right contexts are CF languages. It can be proved that it is undecidable whether two CF languages have a terminal production in common, so finding out if the two right contexts really are sufficient to distinguish between the two items seems impossible (but see Problem 9.29). And checking the rest of the input against a CF language amounts to parsing it, the very problem we are trying to solve.

Both problems are solved by the same trick: we replace the CF grammars of the right contexts by regular grammars. As we have seen in Section 5.5 we can check if two regular languages are disjunct (take the intersection of the automata of both languages and see if the resulting automaton still accepts some string; if it does, the automata are not disjunct). And it is simple to test the rest of the input against both regular languages; below we will show that we can even do that efficiently. But this solution brings in a new problem: how to replace CF grammars by regular ones.

Of course a regular grammar R cannot be equivalent to a CF grammar C, so replacing one by the other involves an approximation "from above": R should at least produce all strings C produces or it will fail to identify an item as applicable when it is. But the overproduction should be minimal, or the set of string may no longer be disjunct from that of the other item, and the parser construction would fail unnecessarily. So R will have to *envelop* C as tightly as possible. If mutually disjunct regular envelopes for all right contexts in inadequate states exist, the grammar G is *LR-regular* (Čulik, II and Cohen [57]), but we can make a parser for G only if we can also actually find the envelopes.

It is actually not necessary to find regular envelopes of the right contexts of each of the items in an inadequate state. It suffices to find regular envelopes for the non-terminals of the grammar; these can then be substituted into the regular expressions for the right contexts.

Finding regular envelopes of non-terminals in a context-free grammar requires heuristics. It is possible to approximate non-terminals better and better with increasingly more complicated regular grammars, but it is undecidable if there exist regular envelopes for the right contexts that are good enough for a given grammar. So when we find that our approximations (regular envelopes) are not disjunct, we cannot know if better heuristics would help. We shall therefore restrict ourselves to the simple heuristic demonstrated in Section 9.13.2.3.

9.13.2.1 LR-Regular Parse Tables

Consider the grammar in Figure 9.46(*a*), which produces $\mathtt{d}^*\mathtt{a}$ and $\mathtt{d}^*\mathtt{b}$. It could, for example, represent a language of integer numbers, with the `d`s standing for digits,

```
Ss  -> A a        S->•Aa
S   -> B b        S->•Bb
A   -> A C        A->•AC
A   -> C          A->•C          C->d•
C   -> d          C->•d          D->d•
B   -> B D        B->•BD          (c)
B   -> D          B->•D
D   -> d          D->•d
     (a)            (b)
```

Fig. 9.46. An LR-regular grammar (*a*), with initial state 1 (*b*) and inadequate state 2 (c)

and the `a` and `b` for indications of the numeric base; examples could then be `123a` for a decimal number, and `123b` for a hexadecimal one. For another motivation of this grammar see Section 10.2.2 and Figure 10.12.

It is easy to see that the grammar of Figure 9.46(*a*) is not LR(*k*): to get past the first `d`, it has to be reduced to either `C` or `D`, but no fixed amount of look-ahead can reach the deciding `a` of `b` at the end of the input. The figure also shows the initial state 1 of an LR parser for the grammar, and the state reached by shifting over a `d`, the one that has the reduce/reduce conflict. The full LR automaton is shown in Figure 9.49.

To resolve that conflict we construct the right contexts of both items, `F_2{C->d•}` and `F_2{D->d•}`. The regular grammar for `F_2{C->d•}` is

```
F_1{S->•Aa}  ->  #
F_1{A->•AC}  ->  a F_1{S->•Aa}
 F_1{A->•C}  ->  a F_1{S->•Aa}
F_1{A->•AC}  ->  C F_1{A->•AC}
 F_1{A->•C}  ->  C F_1{A->•AC}
 F_1{C->•d}  ->  F_1{A->•C}
 F_2{C->d•}  ->  F_1{C->•d}
```

Unsurprisingly this resolves into C^*`a#`. A similar reasoning gives D^*`b#` for `F_2{D->d•}`. Next we have to replace the CF non-terminals `C` and `D` by their regular envelopes. In our example this is trivial, since both are already regular; so the two LR-regular contexts are d^*`a#` and d^*`b#`. And indeed the two sets are disjunct: the grammar of Figure 9.46(*a*) is LR-regular, and the LR-regular contexts can be used as LR-regular look-aheads. Right contexts always end in a `#` symbol, since each item eventually derives from the start symbol, and it has a look-ahead `#`.

So the entry for a state *p* in the ACTION table of an LR-regular parser can contain one of five things: "shift", "reduce", "error", "accept", or a pointer to an LR-regular look-ahead automaton; the latter occurs when the LR state corresponding to *p* was inadequate. The GOTO table of an LR-regular parser in identical to that of the LR parser it derives from.

9.13.2.2 Efficient LR-Regular Parsing

We now turn to the actual parsing, where we meet our second problem: how to determine which of the right contexts the rest of the input is in. The naive way is to just construct an FSA for each LR-regular look-ahead and send it off into the input to see if it stops in an accepting state. This has two drawbacks: 1. the input is rescanned by each FSA F, and there can be many of them; 2. the whole process is repeated after each shift, which may cause the parsing to require $O(n^2)$ time.

The second drawback can be removed by replacing the FSA F by a new FSA $\overleftarrow{F}$, which accepts the reverse of the strings that F accepts; basically such an FSA can be made by reversing all arrows, swapping the initial and accepting states, and making the result deterministic again. We start $\overleftarrow{F}$ at the right of the added end-of input token **#**, and run it backwards over the input. It marks each position in which it is in an accepting state with a marker F_1, the start state of the original, forward, automaton F. This costs $O(n)$ steps. Now, when during parsing we want to know if the rest of the input conforms to F, we can just check if the present position is marked F_1, at constant cost.

We can of course repeat the backward scan of the input for every reversed look-ahead FSA, but it is much more efficient to combine all of them in one big FSA $\overleftarrow{\mathcal{F}}$ by creating a new start state ⊛ with ε-transitions to the start states of all reversed automata for the dot right contexts, as shown in Figure 9.47. The clouds represent

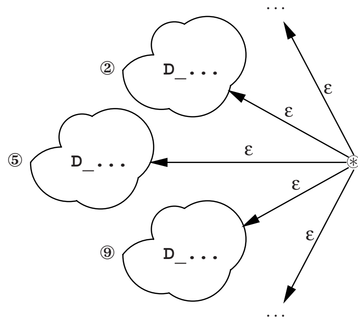
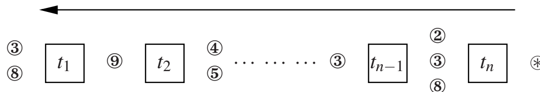


Fig. 9.47. Combined backwards-scanning automaton $\overleftarrow{\mathcal{F}}$ for LR-regular parsing

the various reversed automata, with their accepting states ②, ⑤, ⑨, etc. Using this combined automaton we need now scan backwards only once:



The backwards scan marks each position with the accepting states of all reversed FSAs in $\overleftarrow{\mathcal{F}}$ that apply at that position. These are the start states of the forward automata. A left-to-right LR parsing scan can then use these states as summaries of the look-aheads. This removes the first drawback mentioned above.

We have now achieved a linear-time algorithm: we first read the entire input (at cost $O(n)$); then we scan backwards, using one single FSA recording start states of right contexts (again $O(n)$); and finally we run the LR-regular parser forward, using the recorded states rather than the tokens as look-aheads (also $O(n)$).

9.13.2.3 Finding a Regular Envelope of a Context-Free Grammar

The fundamental difference between regular and context-free is the ability to nest. This nesting is implemented using a stack, both during production and parsing, for LL, LR and pushdown automaton alike. This observation immediately leads to a heuristic for "reducing" a CF language to regular: limit the stack depth. A stack of fixed depth can assume only a finite number of values, which then correspond to the states of a finite state automaton. The idea can be applied naturally to an LR parser with a stack limited to the top state only (but several other variations are possible).

The heuristic can best be explained using a non-deterministic LR automaton, for example the one in Figure 9.15. Assume the input is **n($**. Initially we work the system as an interpreter of the NFA, as in Figure 9.16, so we start in the leftmost state in that figure. Shifting over the **n** brings us to a state that contains only **T→n•** (actually state 2 in Figure 9.17), and since we remember only the top of the stack, we forget the initial state. State 2 orders us to reduce, but since we have lost the **n** *and* the initial state, we know that we have to shift over a **T** but we have no idea from what state to shift. We solve this by introducing a state containing all possible items, thus acknowledging our total ignorance; we then shift over **T** from that state. The result is the item set:

```
E→E-T•
E→T•
```

Note that this item set is not present in the deterministic LR(0) automaton, and cannot occur as a state in the CF parsing. The item set tells us to reduce to **E**, but again without any previous information. We act as above, now obtaining the item set

```
S→E•$
E→E•-T
T→(E•)
```

which is again not an LR(0) state. This item set allows shifts on **$**, **-** and **)**, but not on **(**; so the input **n($** is rejected, even by the regular envelope constructed here. Note that the input **n)$** *is* accepted; indeed it does not contain blatant impossibilities.

A closer look at the above discussion makes it clear what happens when we have to reduce to a non-terminal A: we continue with all items of the form $P \to \alpha A \bullet \beta$. These items can be found in the non-deterministic LR automaton as the items that have an incoming arrow on A. This gives us a way to convert such an automaton

into an FSA for a regular envelope: we connect by ε-transitions all reduce states for each non-terminal A to all states with incoming arrows marked A; next we remove all arrows marked with non-terminals.

This procedure converts the non-deterministic LR(0) automaton of Figure 9.15 into the non-deterministic finite-state automaton of Figure 9.48, in which the unmarked arrows represent ε-transitions, and the accepting state is again marked with a ◊. Rather than connecting all reduce items of a non-terminal A to all items of the

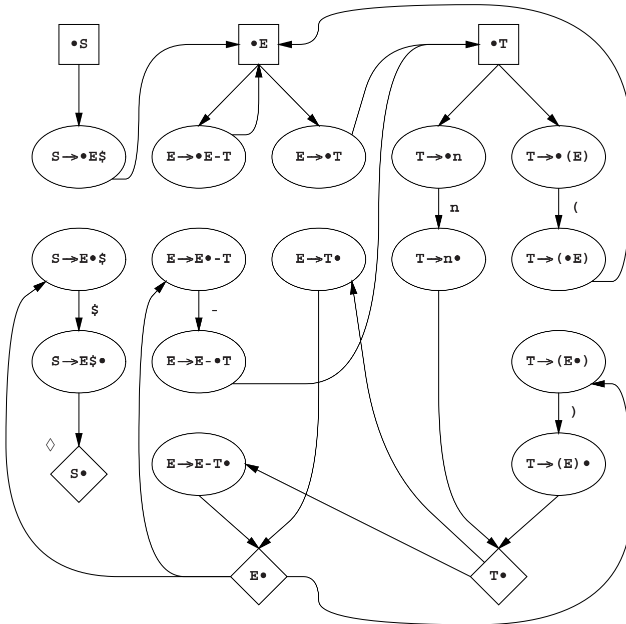


Fig. 9.48. A possible regular envelope for the grammar of Figure 9.14

form $P \to \alpha A \bullet \beta$, we first connect the reduce items to a "terminal station", which is the dual to the "departure" station shown in Figure 9.15, and connect from there to the destination states. Although Figure 9.48 could be drawn neater and without crossing lines, we have kept it as close as possible to Figure 9.15 to show the relationship.

A specific deterministic finite-state automaton for a given non-terminal P can be derived from it by marking the station of P as the start state, and making the automaton deterministic using the subset algorithm. This FSA — or rather the regular expression it corresponds to — can then be used in the expressions derived for item and dot right contexts in Section 9.12.2. See Problem 9.27.

If the resulting regular sets are too coarse and do not sufficiently separate the actions on various items, a better approximation could be obtained by remembering k states rather than 1, but the algorithm to do so is quite complicated. It is usually much easier to duplicate part of the grammar, for example as follows:

```
 S  --> E $
 E  --> E - T' | T
 T  --> n | ( E )
T'  --> n | ( E )
```

This trick increases the number of states in the FSA and so the tightness of the fit. But finding exactly which part to duplicate will always remain an art, since the basic problem is unsolvable.

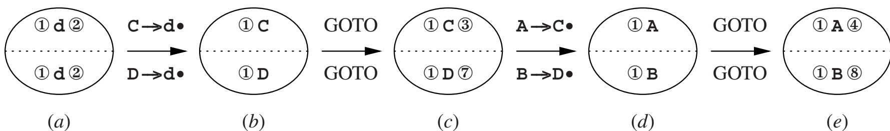
The grammar of Figure 9.46(*a*) shows that LR-regular parsing can handle some non-deterministic grammars. Čulik, II and Cohen [57] prove that the same is true for languages: LR-regular can handle some languages for which there are no deterministic grammars. For the dismal error detection properties of LR-regular, see Problem 9.28.

The above approximation algorithm is from Nederhof [402]. There are many other algorithms for approximating the right context, for example Farré and Fortes Gálvez [98]. See also Yli-Jyrä [403], Pereira and Wright [404], and other papers from (Web)Section 18.4.2. Nederhof's paper [401] includes a survey of regular approximating algorithms.

9.13.3 LAR(*m*) Parsing

Bermudez and Schimpf [82] show a rather different way of exploring and exploiting the right context. At first sight their method seems less than promising: when faced with two possible decisions in an inadequate state, parse ahead with both options and see which one survives. But it is easy to show that, at least occasionally, the method works quite well.

We apply the idea to the grammar of Figure 9.46. Its LR(0) automaton is shown in Figure 9.49; indeed state ② is inadequate, has a reduce/reduce conflict. Suppose the input is **dddb**, which almost immediately lands us in the inadequate state. Rather than first trying the reduction **C-->d•** and seeing where it gets us, and then **D-->d•**, we try both of them simultaneously, one step at a time. In both cases the parser starts in state ①, a **d** is stacked, and state ② is stacked on top, as in frame *a*:



The top level in the bubble is reduced using **C-->d•** and the bottom level with **D-->d•**, as shown in frame *b*. GOTOs over the resulting **C** and **D** give frame *c*. The new states ③ and ⑦ are OK and ask for more reductions, leading to frame *d*. Two more GOTOs put the states ④ and ⑧ on top; both require shifts, so our simultaneous parser is now ready for the next input token. The way we have drawn the combined simulated stacks in transition bubbles already shows that we intend to use them as states in a look-ahead automaton, the *LAR automaton*.

When we now process the next **d** in the input:

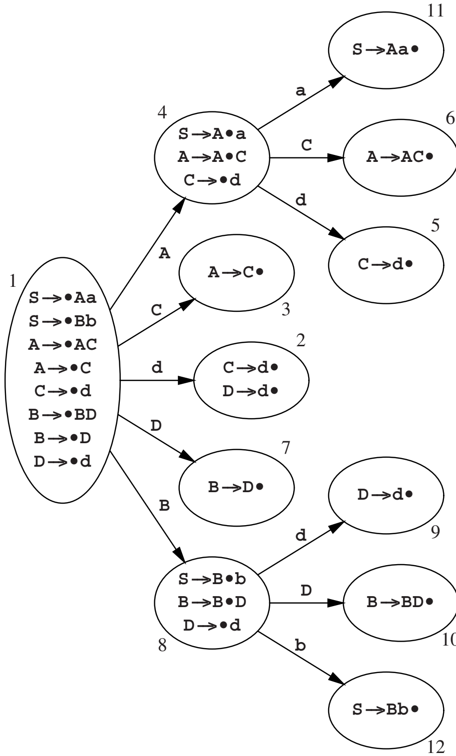
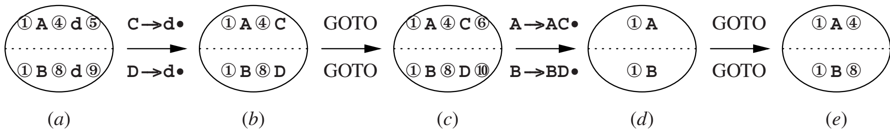
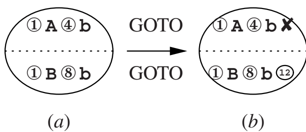


Fig. 9.49. The LR(0) automaton of the grammar of Figure 9.46



we are pleasantly surprised: the LAR state after the second **d** is the same as after the first one! This means that any further number of **d**s will just bring us back to this same state; we can skip explaining these and immediately proceed to the final **b**. We stack the **b** and immediately see that one of the GOTOs fails ((*b*)):



That is all we need to know: as soon as there is only one choice left we can stop our search since we know which decision to take in the inadequate state.

It would be inconvenient to repeat this simulation every time the inadequate state occurs during parsing, so we want to derive from it a finite-state look-ahead automaton that can be computed during parser generation time and can be consulted during parsing. To this end we perform the simulated look-ahead process during parser generation, for all input tokens. This results in a complete FS look-ahead automaton for the given inadequate state. Figure 9.50 shows the LAR automaton for the inadequate state ②, as derived above. Note that it is exactly the FS automaton a programmer

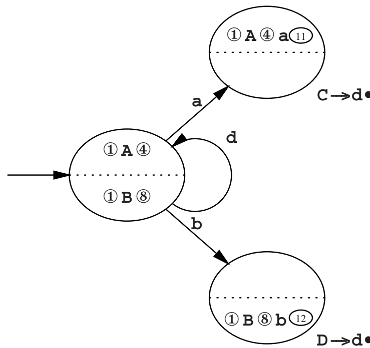


Fig. 9.50. The LAR automaton for the inadequate state ② in Figure 9.49

would have written for the problem: skip `d`s until you find the answer.

The above example allowed us to demonstrate the basic principles and the power of LAR parsing, but not its fine points, of which there are three. The inadequate state can have more than one conflict; we can run into more inadequate states while constructing the LAR automaton; and one or more simulated stacks may grow indefinitely, so the FS look-ahead automaton construction process may not terminate, generating more and more states.

The first two problems are easily solved. If the inadequate LR state has more than one conflict, we start a separate level in our initial LAR state for each possible action. Again states in which all levels but one are empty are terminal states (of the LAR automaton). And if we encounter an inadequate state p_x in the simulated stack of level l, we just copy that stack for all actions that p_x allows, keeping all copies in level l. Again states in which all levels but one are empty are terminal states; we do not need to find out which of the stacks in that level is the correct one.

The problem of the unbounded stack growth is more interesting. Consider the grammar of Figure 9.51; it produces the same language as that of Figure 9.46, but is right-recursive rather than left. The pertinent part of the LR(0) automaton is shown in Figure 9.52.

We process the first two `d`s just as above:

```
Ss  -->  A | B
 A  -->  C A | a
 C  -->  d
 B  -->  D B | b
 D  -->  d
```

Fig. 9.51. An LAR(1) grammar

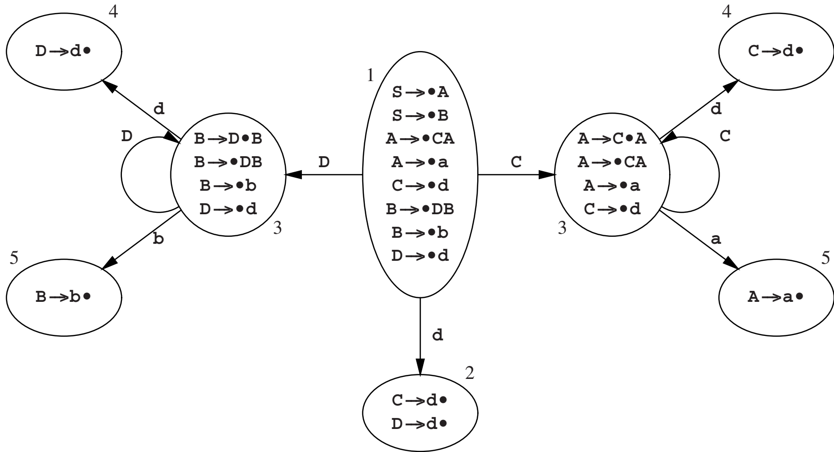
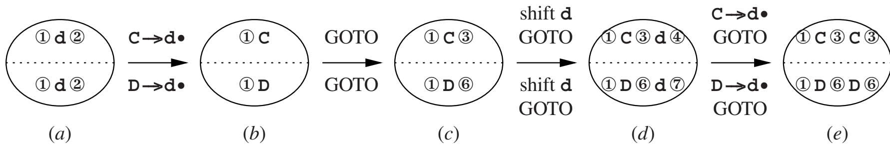
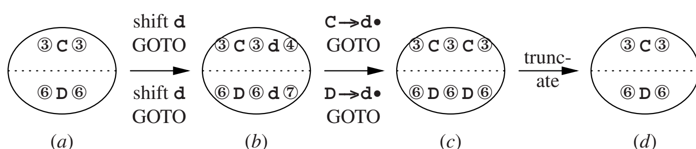


Fig. 9.52. Part of the LR(0) automaton for the grammar of Figure 9.51



but to our dismay we see that the miracle of the identical states does not repeat itself. In fact, it is easy to see that for each subsequent **d** the stacks will grow longer, creating more and more different LAR states, preventing us from constructing a *finite*-state look-ahead automaton at parser generation time. Bermudez and Schimpf's solution to this problem is simple: keep the top-most m symbols of the stack only. This leads to *LAR(m) parsing*. Note that, although we are constructing look-ahead automata, the m is not the length of the look-ahead, but rather the amount of left context maintained while doing the look-ahead. If the resulting LAR automaton has loops in it, the look-ahead itself is unbounded, unrelated to the value of m.

Using this technique with $m = 1$ truncates the stacks of frame e above to those in frame a below:



Proceeding as before, we shift in the **d**s, perform reductions and GOTOs, and finally truncate again to $m = 1$, and we are happy to see that this leads us back to the previous state. Since there are only a finite number of stacks of maximum length m, there are only a finite number of possible states in our LAR automaton, so the construction process is guaranteed to terminate. The result for our grammar is shown in Figure 9.53.

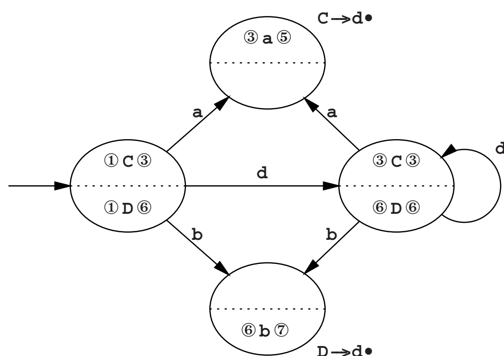


Fig. 9.53. The LAR(1) automaton for the inadequate state 2 in Figure 9.52

This technique seems a sure-fire way to resolve any problems with inadequate states, but of course it isn't. The snag is that when reducing a simulated stack we may have to reduce more symbols than are available on that stack. If that happens, the grammar is not LAR(m) — so the fact that our above attempt with $m = 1$ succeeded proved that the grammar of Figure 9.51 is LAR(1). Making m larger than the length of the longest right-hand side does not always help since successive reduces may still shorten the stack too much.

The above procedure can be summarized as follows:

- For each inadequate state p_x we construct an LAR look-ahead automaton, which starts in a provisional LAR state, which has as many levels as there are possible actions in p_x.
- In each provisional state we continue to perform reduce and GOTO actions until each stack has an LR state which allows shifting on top, all the while truncating the stack to m symbols.
  - If we run into an inadequate state p_y in this process, we duplicate the stack inside the level and continue with all actions p_y allows.

  - If we have to reduce more symbols from a stack than it contains, the grammar is not LAR(m).
  - If we shift over the end marker `#` in this process, the grammar is ambiguous and is not LAR(m).

  If there is now only one non-empty level left in the LAR state, it is a terminal LAR state. Otherwise the result is either a new LAR state, which we process, or a known LAR state.
- For each new LAR state p we create transitions $p \xrightarrow{t} p_t$ for all tokens t that p allows, where the p_t are new provisional states.
- We continue the above process until there are no more new LAR states or we find that the grammar is not LAR(m).

We regret to say that we have again left a couple of complications out of the discussion. When working with $m > 1$, the initial LAR state for an inadequate LR state must contain stacks that derive from the left context of that state. And the number of LAR automata can be reduced by taking traditional LALR look-ahead into account. These complications and more are discussed by Bermudez and Schimpf [82], who also provide advice about obtaining reasonable values for m.

9.14 LR(k) as an Ambiguity Test

It is often important to be sure that a grammar is not ambiguous, but unfortunately that property is undecidable: it can be proved that there cannot be an algorithm that can, for every CF grammar, decide whether it is ambiguous or unambiguous. This is comparable to the situation described in Section 3.4.2, where the fundamental impossibility of a recognizer for Type 0 grammars was discussed. (See Hopcroft and Ullman [391, p. 200]). The most effective ambiguity test for a CF grammar we have at present is the construction of the corresponding LR(k) automaton, but it is not a perfect test: if the construction succeeds, the grammar is guaranteed to be unambiguous; if it fails, in principle nothing is known. In practice, however, the reported conflicts will often point to genuine ambiguities.

The construction of an LR-regular parser (Section 9.13.2) is an even stronger, but more complicated test; see Heilbrunner [392] for a precise algorithm. Schmitz and Farré [398] describe a different very strong ambiguity test that can be made arbitrarily strong at arbitrary expense, but it is experimental.

9.15 Conclusion

The basis of bottom-up parsing is reducing the input, through a series of sentential forms, to the start symbol, all the while constructing the parse tree(s). The basis of deterministic bottom-up parsing is finding, with certainty, in each sentential form a segment α equal to the right-hand side of a rule $A \rightarrow \alpha$ such that the reduction using that rule will create a node A that is guaranteed to be part of the parse tree. The basis

of left-to-right deterministic bottom-up parsing is finding, preferably efficiently, the leftmost segment with that property, the *handle*.

Many plans have been devised to find the handle. Precedence parsing inserts three types of marker in the sentential form: $\lessdot$ for the left end of a handle; $\doteq$ for use in the middle of a handle; and $\gtrdot$ for the right end of the handle. The decision which marker to place in a given position depends on one or a few tokens on the left and on the right of the position. Bounded context identifies the handle by a left and right context, each a few tokens long. LR summarizes the entire left context into a single state of an FSA, which state then identifies the reduction rule, in combination with zero, one, or a few tokens of the right context. LR-regular summarizes the entire right context into a single state of a second FSA, which state in combination with the left context state then identifies the reduction rule. Many different FSAs have been proposed for this purpose.

Problems

Problem 9.1: Arguably the simplest deterministic bottom-up parser is one in which the shortest leftmost substring in the sentential form that matches a right-hand side in the grammar is the handle. Determine conditions for which this parser works. See also Problem 10.9.

Problem 9.2: Precedence parsing was explained as "inserting parenthesis generators". Sheridan [111] sketches an algorithm that inserts sufficient numbers of parentheses. Determine conditions for which this works.

Problem 9.3: There is an easy approach to LR(0) automata with shift/reduce conflicts only: shift if you can, reduce otherwise. Work out the consequences.

Problem 9.4: Extend the tables in Figure 9.18 for the case that the input consists of sentential forms containing both terminal and non-terminal symbols rather than strings of terminals. Same question for Figure 9.28.

Problem 9.5: Complete the LR(2) ACTION and GOTO tables of Figure 9.33.

Problem 9.6: Design the combined LR($k = 0, 1, > 1$) algorithm hinted at on page 299.

Problem 9.7: Devise an efficient table structure for an LR(k) parser where k is fairly large, say between 5 and 20. (Such grammars may arise in grammatical data compression, Section 17.5.1.)

Problem 9.8: An LR(1) grammar is converted to CNF, as in Section 4.2.3. Is it still LR(1)?

Problem 9.9: In an LR(1) grammar in CNF all non-terminals that are used only once in the grammar are substituted out. Is the resulting grammar still LR(1)?

Problem 9.10: Is it possible for two items in the LALR(1) channel algorithm to be connected both by propagation channels and by spontaneous channels?

Problem 9.11: Apply the algorithm of Section 9.7.1.3 to the grammar of Figure 9.30.

Problem 9.12: The **reads** and **directly-reads** relations in Section 9.7.1.3 seem to compute the FIRST sets of some tails of right-hand sides. Explore the exact relationship between **reads** and **directly-reads** and FIRST sets.

Problem 9.13: *Project for Prolog fans:* The relations in the algorithm of Section 9.7.1.3 immediately suggest Prolog. Program the algorithm in Prolog, keeping the single-formula formulation of page 310 as a single Prolog clause, if possible.

Problem 9.14: Although the LALR-by-SLR algorithm as described by Bermudez and Logothetis [79] can compute look-ahead sets of reduce items only, a very simple modification allows it to compute the LALR look-aheads of any item. Use it to compute the LALR look-ahead sets of `E-->E•-T` in states 4 and 9 of Figure 9.25.

Problem 9.15: *Project:* The channels in the channel algorithm in Section 9.7.1.2 and the relations in the relations algorithm in Section 9.7.1.3 bear some resemblance. Work out this resemblance and construct a unified algorithm, if possible.

Problem 9.16: *Project:* It is not obvious that starting the FSA construction process in Section 9.10.2 from state s_0 yields the best possible set, either in size or in amount of stack activity saved. Research the possibility that a different order produces a better set of FSAs, or even that a different or better set exists that does not derive from some order.

Problem 9.17: Derive left-context regular expressions for the states in Figure 9.17 as explained in Section 9.12.

Problem 9.18: Write a program to construct the regular grammar for the left contexts of a given grammar.

Problem 9.19: Write a program to construct the regular grammar for the right contexts of a given grammar.

Problem 9.20: It seems reasonable to assume that when the dot right contexts in a given inadequate state have a non-empty intersection even on the regular expression level, the grammar must be ambiguous: there is at least one continuation that will satisfy both choices, right up to the end of the input, and thus lead to two successful parses. The grammar `S-->aSa | a`, which is unambiguous, proves that this is not true: `D_2S-->•a=aaa*$` and `D_2S-->a•=aa*$`, and they have any string in `aaa*$` in common. What is wrong with the reasoning?

Problem 9.21: Construct a grammar that has a DR automaton with a loop in it.

Problem 9.22: Since the regular envelope in LR-regular parsing is too wide, it can happen that the rest of the input is inside the regular envelope but outside the CF right context grammar it envelopes. What happens in this case?

Problem 9.23: Show that the naive implementation of the LR-regular parser in Section 9.13.2 indeed has a time requirement of $O(n^2)$.

Problem 9.24: Work out the details of building a reverse FSA $\overleftarrow{F}$ from a given FSA F, both when F is non-deterministic and when it is already deterministic. ($\overleftarrow{F}$ should recognize the reverse of the strings F recognizes.)

Problem 9.25: Derive a deterministic automaton (or a regular expression) for `T` from the automaton in Figure 9.48.

Problem 9.26: Devise a way to do the transformation to a regular envelope on the deterministic LR(0) automaton (for example Figure 9.17) rather than on the non-deterministic one.

Problem 9.27: 1. Make the NFA in Figure 9.48 deterministic for `T`. 2. Derive a regular expression for `T` and use it in the expression `[-T|)]*) [-T]*$` derived for the item right context of `E-->•E-T` in state 6 in Section 9.12.2.

Problem 9.28: *Project*: Design reasonable error reporting for an LR-regular parser. (Background: If the backward scan of an LR-regular parser is performed on incorrect input, chances are that the automaton gets stuck somewhere, say at a position *P*, which means that no look-aheads will be attached to any positions left of *P*, which in turn means that parsing cannot even start. Giving an error message about position *P* is unattractive because 1) it may not be the leftmost error, which is awkward if there is more than one error, and 2) no reasonable error message can be given since there is finite-state information only.)

Problem 9.29: *Project Formal Languages:* The argument on page 328 suggesting that it is undecidable whether a grammar is LR-regular or not works the wrong way: it reduces our problem to an undecidable problem, but it should reduce an undecidable problem to ours. Correct.

Problem 9.30: On page 337 we write that the grammar is not LAR(*m*) if during reducing a simulated stack we have to reduce more symbols than are available on that stack. But why is that a problem? We know which reduction to do, so we could just do it. Or can we?