An LR parser does not have to scan the entire stack to know when the handle appears on top. Rather, the state symbol on top of the stack contains all the information it needs. It is a remarkable fact that if it is possible to recognize a handle knowing only the grammar symbols on the stack, then there is a finite automaton that can, by reading the grammar symbols on the stack from top to bottom, determine what handle, if any, is on top of the stack. The goto function of an LR parsing table is essentially such a finite automaton. The automaton need not, however, read the stack on every move. The state symbol stored on top of the stack is the state the handle-recognizing finite automaton would be in if it had read the grammar symbols of the stack from bottom to top. Thus, the LR parser can determine from the state on top of the stack everything that it needs to know about what is in the stack.

Another source of information that an LR parser can use to help make its shift-reduce decisions is the next k input symbols. The cases $k=0$ or $k=1$ are of practical interest, and we shall only consider LR parsers with $k \leq 1$ here. For example, the action table in Fig. 4.31 uses one symbol of lookahead. A grammar that can be parsed by an LR parser examining up to k input symbols on each move is called an *LR(k) grammar*.

There is a significant difference between LL and LR grammars. For a grammar to be LR(k), we must be able to recognize the occurrence of the right side of a production, having seen all of what is derived from that right side with k input symbols of lookahead. This requirement is far less stringent than that for LL(k) grammars where we must be able to recognize the use of a production seeing only the first k symbols of what its right side derives. Thus, LR grammars can describe more languages than LL grammars.

Constructing SLR Parsing Tables

We now show how to construct from a grammar an LR parsing table. We shall give three methods, varying in their power and ease of implementation. The first, called "simple LR" or *SLR* for short, is the weakest of the three in terms of the number of grammars for which it succeeds, but is the easiest to implement. We shall refer to the parsing table constructed by this method as an SLR table, and to an LR parser using an SLR parsing table as an SLR parser. A grammar for which an SLR parser can be constructed is said to be an SLR grammar. The other two methods augment the SLR method with lookahead information, so the SLR method is a good starting point for studying LR parsing.

An *LR(0) item* (*item* for short) of a grammar G is a production of G with a dot at some position of the right side. Thus, production $A \rightarrow XYZ$ yields the four items

$$\begin{aligned} A &\rightarrow \cdot XYZ \\ A &\rightarrow X \cdot YZ \\ A &\rightarrow XY \cdot Z \\ A &\rightarrow XYZ \cdot \end{aligned}$$

The production $A \to \epsilon$ generates only one item, $A \to \cdot$. An item can be represented by a pair of integers, the first giving the number of the production and the second the position of the dot. Intuitively, an item indicates how much of a production we have seen at a given point in the parsing process. For example, the first item above indicates that we hope to see a string derivable from XYZ next on the input. The second item indicates that we have just seen on the input a string derivable from X and that we hope next to see a string derivable from YZ.

The central idea in the SLR method is first to construct from the grammar a deterministic finite automaton to recognize viable prefixes. We group items together into sets, which give rise to the states of the SLR parser. The items can be viewed as the states of an NFA recognizing viable prefixes, and the "grouping together" is really the subset construction discussed in Section 3.6.

One collection of sets of LR(0) items, which we call the *canonical* LR(0) collection, provides the basis for constructing SLR parsers. To construct the canonical LR(0) collection for a grammar, we define an augmented grammar and two functions, *closure* and *goto*.

If G is a grammar with start symbol S, then G', the *augmented grammar* for G, is G with a new start symbol S' and production $S' \to S$. The purpose of this new starting production is to indicate to the parser when it should stop parsing and announce acceptance of the input. That is, acceptance occurs when and only when the parser is about to reduce by $S' \to S$.

The Closure Operation

If I is a set of items for a grammar G, then *closure*(I) is the set of items constructed from I by the two rules:

1. Initially, every item in I is added to *closure*(I).

2. If $A \to \alpha \cdot B\beta$ is in *closure*(I) and $B \to \gamma$ is a production, then add the item $B \to \cdot\gamma$ to I, if it is not already there. We apply this rule until no more new items can be added to *closure*(I).

Intuitively, $A \to \alpha \cdot B\beta$ in *closure*(I) indicates that, at some point in the parsing process, we think we might next see a substring derivable from $B\beta$ as input. If $B \to \gamma$ is a production, we also expect we might see a substring derivable from γ at this point. For this reason we also include $B \to \cdot\gamma$ in *closure*(I).

Example 4.34. Consider the augmented expression grammar:

$$\begin{aligned} E' &\to E \\ E &\to E + T \mid T \\ T &\to T * F \mid F \\ F &\to (E) \mid \mathbf{id} \end{aligned} \tag{4.19}$$

If I is the set of one item $\{[E' \to \cdot E]\}$, then *closure*(I) contains the items

$$
\begin{aligned}
E' &\to \cdot E \\
E &\to \cdot E+T \\
E &\to \cdot T \\
T &\to \cdot T*F \\
T &\to \cdot F \\
F &\to \cdot(E) \\
F &\to \cdot\mathbf{id}
\end{aligned}
$$

Here, $E' \to \cdot E$ is put in *closure*(I) by rule (1). Since there is an E immediately to the right of a dot, by rule (2) we add the E-productions with dots at the left end, that is, $E \to \cdot E + T$ and $E \to \cdot T$. Now there is a T immediately to the right of a dot, so we add $T \to \cdot T*F$ and $T \to \cdot F$. Next, the F to the right of a dot forces $F \to \cdot(E)$ and $F \to \cdot\mathbf{id}$ to be added. No other items are put into *closure*(I) by rule (2). □

The function *closure* can be computed as in Fig. 4.33. A convenient way to implement the function *closure* is to keep a boolean array *added*, indexed by the nonterminals of G, such that *added*[B] is set to **true** if and when we add the items $B \to \cdot\gamma$ for each B-production $B \to \gamma$.

```
function closure ( $I$ );
begin
    $J := I$;
    repeat
        for each item $A \to \alpha \cdot B\beta$ in $J$ and each production
            $B \to \gamma$ of $G$ such that $B \to \cdot\gamma$ is not in $J$ do
                add $B \to \cdot\gamma$ to $J$
    until no more items can be added to $J$;
    return $J$
end
```

Fig. 4.33. Computation of *closure*.

Note that if one B-production is added to the closure of I with the dot at the left end, then all B-productions will be similarly added to the closure. In fact, it is not necessary in some circumstances actually to list the items $B \to \cdot\gamma$ added to I by *closure*. A list of the nonterminals B whose productions were so added will suffice. In fact, it turns out that we can divide all the sets of items we are interested in into two classes of items.

1. *Kernel items*, which include the initial item, $S' \to \cdot S$, and all items whose dots are not at the left end.

2. *Nonkernel items*, which have their dots at the left end.

Moreover, each set of items we are interested in is formed by taking the closure of a set of kernel items; the items added in the closure can never be kernel items, of course. Thus, we can represent the sets of items we are really interested in with very little storage if we throw away all nonkernel items, knowing that they could be regenerated by the closure process.

The Goto Operation

The second useful function is $goto(I, X)$ where I is a set of items and X is a grammar symbol. $goto(I, X)$ is defined to be the closure of the set of all items $[A \to \alpha X \cdot \beta]$ such that $[A \to \alpha \cdot X \beta]$ is in I. Intuitively, if I is the set of items that are valid for some viable prefix γ, then $goto(I, X)$ is the set of items that are valid for the viable prefix γX.

Example 4.35. If I is the set of two items $\{[E' \to E \cdot], [E \to E \cdot + T]\}$, then $goto(I, +)$ consists of

$$\begin{aligned} E &\to E + \cdot T \\ T &\to \cdot T * F \\ T &\to \cdot F \\ F &\to \cdot (E) \\ F &\to \cdot \mathbf{id} \end{aligned}$$

We computed $goto(I, +)$ by examining I for items with $+$ immediately to the right of the dot. $E' \to E \cdot$ is not such an item, but $E \to E \cdot + T$ is. We moved the dot over the $+$ to get $\{E \to E + \cdot T\}$ and then took the closure of this set. □

The Sets-of-Items Construction

We are now ready to give the algorithm to construct C, the canonical collection of sets of LR(0) items for an augmented grammar G'; the algorithm is shown in Fig. 4.34.

```
procedure items(G');
begin
    C := {closure({[S' → ·S]})};
    repeat
        for each set of items I in C and each grammar symbol X
                such that goto(I, X) is not empty and not in C do
                    add goto(I, X) to C
    until no more sets of items can be added to C
end
```

Fig. 4.34. The sets-of-items construction.

Example 4.36. The canonical collection of sets of LR(0) items for grammar (4.19) of Example 4.34 is shown in Fig. 4.35. The *goto* function for this set of items is shown as the transition diagram of a deterministic finite automaton D in Fig. 4.36. □

$$\begin{array}{ll}
I_0: & E' \to \cdot E \\
 & E \to \cdot E + T \\
 & E \to \cdot T \\
 & T \to \cdot T * F \\
 & T \to \cdot F \\
 & F \to \cdot (E) \\
 & F \to \cdot \mathbf{id} \\
\\
I_1: & E' \to E \cdot \\
 & E \to E \cdot + T \\
\\
I_2: & E \to T \cdot \\
 & T \to T \cdot * F \\
\\
I_3: & T \to F \cdot \\
\\
I_4: & F \to (\cdot E) \\
 & E \to \cdot E + T \\
 & E \to \cdot T \\
 & T \to \cdot T * F \\
 & T \to \cdot F \\
 & F \to \cdot (E) \\
 & F \to \cdot \mathbf{id}
\end{array}
\qquad
\begin{array}{ll}
I_5: & F \to \mathbf{id} \cdot \\
\\
I_6: & E \to E + \cdot T \\
 & T \to \cdot T * F \\
 & T \to \cdot F \\
 & F \to \cdot (E) \\
 & F \to \cdot \mathbf{id} \\
\\
I_7: & T \to T * \cdot F \\
 & F \to \cdot (E) \\
 & F \to \cdot \mathbf{id} \\
\\
I_8: & F \to (E \cdot) \\
 & E \to E \cdot + T \\
\\
I_9: & E \to E + T \cdot \\
 & T \to T \cdot * F \\
\\
I_{10}: & T \to T * F \cdot \\
\\
I_{11}: & F \to (E) \cdot
\end{array}$$

Fig. 4.35. Canonical LR(0) collection for grammar (4.19).

If each state of D in Fig. 4.36 is a final state and I_0 is the initial state, then D recognizes exactly the viable prefixes of grammar (4.19). This is no accident. For every grammar G, the *goto* function of the canonical collection of sets of items defines a deterministic finite automaton that recognizes the viable prefixes of G. In fact, one can visualize a nondeterministic finite automaton N whose states are the items themselves. There is a transition from $A \to \alpha \cdot X \beta$ to $A \to \alpha X \cdot \beta$ labeled X, and there is a transition from $A \to \alpha \cdot B \beta$ to $B \to \cdot \gamma$ labeled ϵ. Then *closure*(I) for set of items (states of N) I is exactly the ϵ-*closure* of a set of NFA states defined in Section 3.6. Thus, *goto*(I, X) gives the transition from I on symbol X in the DFA constructed from N by the subset construction. Viewed in this way, the procedure *items*(G') in Fig. 4.34 is just the subset construction itself applied to the NFA N constructed from G' as we have described.

Valid items. We say item $A \to \beta_1 \cdot \beta_2$ is *valid* for a viable prefix $\alpha\beta_1$ if there is a derivation $S' \overset{*}{\underset{rm}{\Rightarrow}} \alpha A w \overset{*}{\underset{rm}{\Rightarrow}} \alpha\beta_1\beta_2 w$. In general, an item will be valid for many viable prefixes. The fact that $A \to \beta_1 \cdot \beta_2$ is valid for $\alpha\beta_1$ tells us a lot about whether to shift or reduce when we find $\alpha\beta_1$ on the parsing stack. In

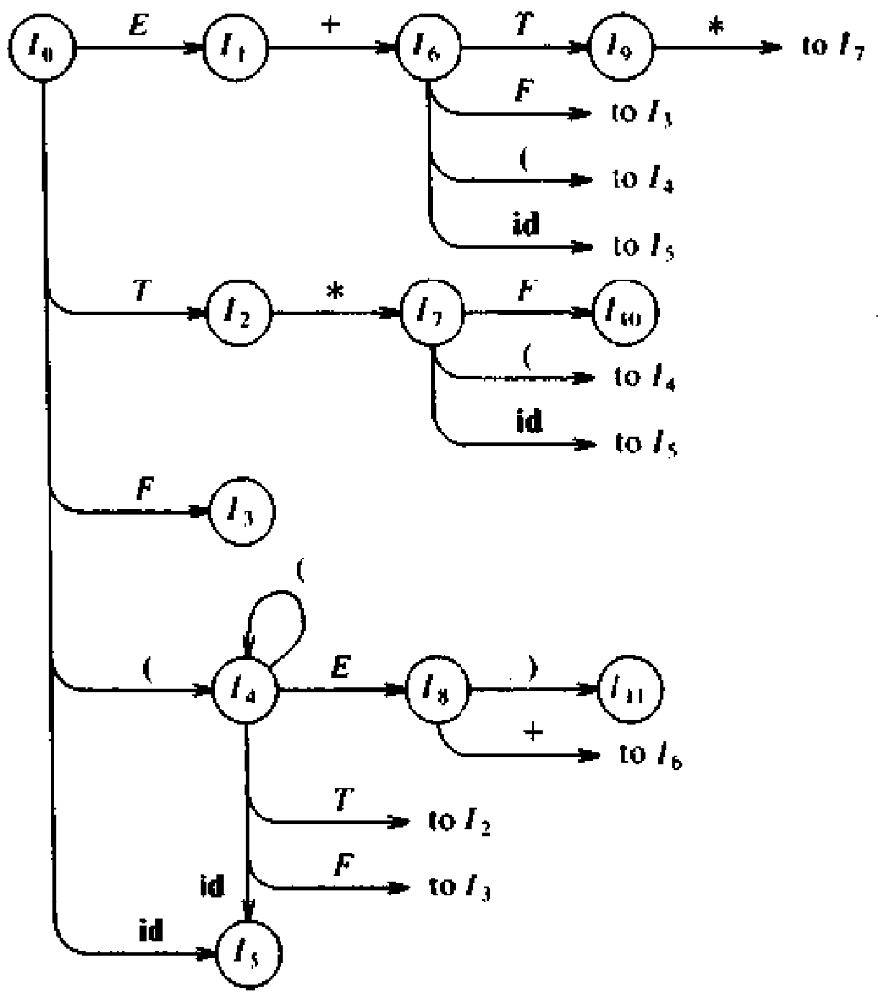

Fig. 4.36. Transition diagram of DFA D for viable prefixes.

particular, if $\beta_2 \neq \epsilon$, then it suggests that we have not yet shifted the handle onto the stack, so shift is our move. If $\beta_2 = \epsilon$, then it looks as if $A \rightarrow \beta_1$ is the handle, and we should reduce by this production. Of course, two valid items may tell us to do different things for the same viable prefix. Some of these conflicts can be resolved by looking at the next input symbol, and others can be resolved by the methods of the next section, but we should not suppose that all parsing action conflicts can be resolved if the LR method is used to construct a parsing table for an arbitrary grammar.

We can easily compute the set of valid items for each viable prefix that can appear on the stack of an LR parser. In fact, it is a central theorem of LR parsing theory that the set of valid items for a viable prefix γ is exactly the set of items reached from the initial state along a path labeled γ in the DFA constructed from the canonical collection of sets of items with transitions given by *goto*. In essence, the set of valid items embodies all the useful information that can be gleaned from the stack. While we shall not prove this theorem here, we shall give an example.

Example 4.37. Let us consider the grammar (4.19) again, whose sets of items and *goto* function are exhibited in Fig. 4.35 and 4.36. Clearly, the string $E + T *$ is a viable prefix of (4.19). The automaton of Fig. 4.36 will be in state I_7 after having read $E + T *$. State I_7 contains the items

$$
\begin{aligned}
&T \to T * \cdot F\\
&F \to \cdot(E)\\
&F \to \cdot\mathbf{id}
\end{aligned}
$$

which are precisely the items valid for $E + T *$. To see this, consider the following three rightmost derivations

$$
\begin{array}{lll}
E' \Rightarrow E & E' \Rightarrow E & E' \Rightarrow E\\
\quad \Rightarrow E+T & \quad \Rightarrow E+T & \quad \Rightarrow E+T\\
\quad \Rightarrow E+T*F & \quad \Rightarrow E+T*F & \quad \Rightarrow E+T*F\\
 & \quad \Rightarrow E+T*(E) & \quad \Rightarrow E+T*\mathbf{id}
\end{array}
$$

The first derivation shows the validity of $T \to T * \cdot F$, the second the validity of $F \to \cdot(E)$, and the third the validity of $F \to \cdot\mathbf{id}$ for the viable prefix $E + T *$. It can be shown that there are no other valid items for $E + T *$, and we leave a proof to the interested reader. □

SLR Parsing Tables

Now we shall show how to construct the SLR parsing action and goto functions from the deterministic finite automaton that recognizes viable prefixes. Our algorithm will not produce uniquely defined parsing action tables for all grammars, but it does succeed on many grammars for programming languages. Given a grammar, G, we augment G to produce G', and from G' we construct C, the canonical collection of sets of items for G'. We construct *action*, the parsing action function, and *goto*, the goto function, from C using the following algorithm. It requires us to know FOLLOW(A) for each nonterminal A of a grammar (see Section 4.4).

Algorithm 4.8. Constructing an SLR parsing table.

Input. An augmented grammar G'.

Output. The SLR parsing table functions *action* and *goto* for G'.

Method.

1. Construct $C = \{I_0, I_1, \ldots, I_n\}$, the collection of sets of LR(0) items for G'.

2. State i is constructed from I_i. The parsing actions for state i are determined as follows:

   a) If $[A \to \alpha \cdot a\beta]$ is in I_i and $goto(I_i, a) = I_j$, then set *action*$[i, a]$ to "shift j." Here a must be a terminal.

   b) If $[A \to \alpha\cdot]$ is in I_i, then set *action*$[i, a]$ to "reduce $A \to \alpha$" for all a

in FOLLOW(A); here A may not be S'.

c) If $[S' \to S\cdot]$ is in I_i, then set *action*$[i, \$]$ to "accept."

If any conflicting actions are generated by the above rules, we say the grammar is not SLR(1). The algorithm fails to produce a parser in this case.

3. The goto transitions for state i are constructed for all nonterminals A using the rule: If $goto(I_i, A) = I_j$, then $goto[i, A] = j$.

4. All entries not defined by rules (2) and (3) are made "error."

5. The initial state of the parser is the one constructed from the set of items containing $[S' \to \cdot S]$. □

The parsing table consisting of the parsing action and goto functions determined by Algorithm 4.8 is called the *SLR(1) table for G*. An LR parser using the SLR(1) table for G is called the SLR(1) parser for G, and a grammar having an SLR(1) parsing table is said to be *SLR(1)*. We usually omit the "(1)" after the "SLR," since we shall not deal here with parsers having more than one symbol of lookahead.

Example 4.38. Let us construct the SLR table for grammar (4.19). The canonical collection of sets of LR(0) items for (4.19) was shown in Fig. 4.35. First consider the set of items I_0:

$$\begin{aligned}
E' &\to \cdot E \\
E &\to \cdot E+T \\
E &\to \cdot T \\
T &\to \cdot T*F \\
T &\to \cdot F \\
F &\to \cdot(E) \\
F &\to \cdot\textbf{id}
\end{aligned}$$

The item $F \to \cdot(E)$ gives rise to the entry *action*[0, (] = shift 4, the item $F \to \cdot$**id** to the entry *action*[0, **id**] = shift 5. Other items in I_0 yield no actions. Now consider I_1:

$$\begin{aligned}
E' &\to E\cdot \\
E &\to E\cdot+T
\end{aligned}$$

The first item yields *action*[1, \$] = accept, the second yields *action*[1, +] = shift 6. Next consider I_2:

$$\begin{aligned}
E &\to T\cdot \\
T &\to T\cdot*F
\end{aligned}$$

Since FOLLOW(E) = {\$, +,)}, the first item makes *action*[2, \$] = *action*[2, +] = *action*[2,)] = reduce $E \to T$. The second item makes *action*[2, *] = shift 7. Continuing in this fashion we obtain the parsing action and goto tables that were shown in Fig. 4.31. In that figure, the numbers of productions in reduce actions are the same as the order in which they appear in the original grammar (4.18). That is, $E \to E+T$ is number 1, $E \to T$ is 2, and so on. □

Example 4.39. Every SLR(1) grammar is unambiguous, but there are many unambiguous grammars that are not SLR(1). Consider the grammar with productions

$$\begin{array}{l} S \to L = R \\ S \to R \\ L \to * R \\ L \to \textbf{id} \\ R \to L \end{array} \tag{4.20}$$

We may think of L and R as standing for l-value and r-value, respectively, and $*$ as an operator indicating "contents of."[3] The canonical collection of sets of LR(0) items for grammar (4.20) is shown in Fig. 4.37.

$$\begin{array}{ll}
I_0: & S' \to \cdot S \\
 & S \to \cdot L = R \\
 & S \to \cdot R \\
 & L \to \cdot * R \\
 & L \to \cdot \textbf{id} \\
 & R \to \cdot L \\
 & \\
I_1: & S' \to S \cdot \\
 & \\
I_2: & S \to L \cdot = R \\
 & R \to L \cdot \\
 & \\
I_3: & S \to R \cdot \\
 & \\
I_4: & L \to * \cdot R \\
 & R \to \cdot L \\
 & L \to \cdot * R \\
 & L \to \cdot \textbf{id}
\end{array}
\qquad
\begin{array}{ll}
I_5: & L \to \textbf{id} \cdot \\
 & \\
I_6: & S \to L = \cdot R \\
 & R \to \cdot L \\
 & L \to \cdot * R \\
 & L \to \cdot \textbf{id} \\
 & \\
I_7: & L \to * R \cdot \\
 & \\
I_8: & R \to L \cdot \\
 & \\
I_9: & S \to L = R \cdot
\end{array}$$

Fig. 4.37. Canonical LR(0) collection for grammar (4.20).

Consider the set of items I_2. The first item in this set makes *action*[2, =] be "shift 6." Since FOLLOW(R) contains =, (to see why, consider $S \Rightarrow L = R \Rightarrow * R = R$), the second item sets *action*[2, =] to "reduce $R \to L$." Thus entry *action*[2, =] is multiply defined. Since there is both a shift and a reduce entry in *action*[2, =], state 2 has a shift/reduce conflict on

[3] As in Section 2.8, an l-value designates a location and an r-value is a value that can be stored in a location.

input symbol =.

Grammar (4.20) is not ambiguous. This shift/reduce conflict arises from the fact that the SLR parser construction method is not powerful enough to remember enough left context to decide what action the parser should take on input = having seen a string reducible to L. The canonical and LALR methods, to be discussed next, will succeed on a larger collection of grammars, including grammar (4.20). It should be pointed out, however, that there are unambiguous grammars for which every LR parser construction method will produce a parsing action table with parsing action conflicts. Fortunately, such grammars can generally be avoided in programming language applications. □

Constructing Canonical LR Parsing Tables

We shall now present the most general technique for constructing an LR parsing table from a grammar. Recall that in the SLR method, state i calls for reduction by $A \rightarrow \alpha$ if the set of items I_i contains item $[A \rightarrow \alpha \cdot]$ and a is in FOLLOW(A). In some situations, however, when state i appears on top of the stack, the viable prefix $\beta\alpha$ on the stack is such that βA cannot be followed by a in a right-sentential form. Thus, the reduction by $A \rightarrow \alpha$ would be invalid on input a.

Example 4.40. Let us reconsider Example 4.39, where in state 2 we had item $R \rightarrow L\cdot$, which could correspond to $A \rightarrow \alpha$ above, and a could be the = sign, which is in FOLLOW(R). Thus, the SLR parser calls for reduction by $R \rightarrow L$ in state 2 with = as the next input (the shift action is also called for because of item $S \rightarrow L \cdot = R$ in state 2). However, there is no right-sentential form of the grammar in Example 4.39 that begins $R = \cdots$. Thus state 2, which is the state corresponding to viable prefix L only, should not really call for reduction of that L to R. □

It is possible to carry more information in the state that will allow us to rule out some of these invalid reductions by $A \rightarrow \alpha$. By splitting states when necessary, we can arrange to have each state of an LR parser indicate exactly which input symbols can follow a handle α for which there is a possible reduction to A.

The extra information is incorporated into the state by redefining items to include a terminal symbol as a second component. The general form of an item becomes $[A \rightarrow \alpha \cdot \beta, a]$, where $A \rightarrow \alpha\beta$ is a production and a is a terminal or the right endmarker \$. We call such an object an *LR(1) item*. The 1 refers to the length of the second component, called the *lookahead* of the item.[4] The lookahead has no effect in an item of the form $[A \rightarrow \alpha \cdot \beta, a]$, where β is not ϵ, but an item of the form $[A \rightarrow \alpha \cdot, a]$ calls for a reduction by $A \rightarrow \alpha$ only if

[4] Lookaheads that are strings of length greater than one are possible, of course, but we shall not consider such lookaheads here.