On the other hand, consider the language of "centered" palindromes, as defined by the following grammar:

⟨*CenPal*⟩ ::= `!` | `0` ⟨*CenPal*⟩ `0` | `1` ⟨*CenPal*⟩ `1`

The first of the two problems with ⟨*palindrome*⟩ is avoided because ⟨*CenPal*⟩ does not generate the empty string. The second problem is avoided because there is only one production for ⟨*CenPal*⟩ whose first token is `0`, and similarly for `1`.

Exercise 11.2 Code a recursive-descent parser for centered palindromes.

11.3.5 Discussion

We have seen several examples of recursive-descent parsing in practice. In each case, it was necessary to ensure that, wherever there were several productions for a nonterminal, it would be possible to decide which production to choose using only the current token. In this section, we summarize the lessons of these examples in general terms. Such rules are important not only to be able to ensure that a recursive-descent parser is feasible, but also to verify that a grammar is not ambiguous, for if no alternative productions are available for expansion, the grammar must be unambiguous. It is for this reason that we may assert that *particular* grammars, for example, for expressions and statements, are *not* ambiguous, even though a *general* algorithm for testing ambiguity of context-free grammars cannot exist.

If a context-free grammar is to be implemented using recursive descent, it is necessary to determine, for each production, the *director set* of input tokens that will direct the parse to use *that* production, rather than other productions for the same nonterminal symbol. The constraint that must be satisfied if recursive-descent parsing is to be used is that the director sets for a nonterminal must be *pairwise disjoint*; i.e., if D and D' are the director sets associated with two productions for the same nonterminal symbol, $D \cap D' = \emptyset$. The director sets are determined as follows.

Let $\langle N \rangle ::= \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_n$ be all the productions for nonterminal symbol $\langle N \rangle$. The α_i are strings of terminal and nonterminal symbols. The director set associated with production $\langle N \rangle ::= \alpha_i$ then contains the following tokens:

- all initial tokens of non-empty strings derivable from α_i; this set is usually called $\mathit{first}(\alpha_i)$;
- in addition, if the empty string ε is derivable from α_i, all initial tokens of nonempty strings that may follow $\langle N \rangle$ in any derivation, possibly including the end-of-string pseudo-token `EOS`; this set is usually called $\mathit{follow}\langle N \rangle$.

Note that the definitions of *first* and *follow* are not completely symmetric; in particular, *follow* is defined *only* for single nonterminal symbols.

In simple examples, such as those we have been considering, it is easy to determine the *first* and *follow* sets "by hand." However, for more complex grammars, fairly sophisticated algorithms must be used to allow for mutual dependencies between nonterminals and for nonterminals that may generate the empty string. The details are beyond the scope of this presentation, but they may be found in many books on compilers.

If a grammar does *not* satisfy this criterion, it may still be possible to use recursive descent if common prefixes may be left factored, or if options or closure operations may be used to re-formulate the productions. The conditions that must then be verified may be determined by expressing the use of an option or closure operation in terms of conventional context-free productions and testing the condition stated earlier.

For example, a use of closure of the form

$$\langle N \rangle ::= \cdots \mid \alpha \left\{ \beta \right\}$$

may be re-expressed in conventional BNF as

$$\langle N \rangle ::= \cdots \mid \alpha \ \langle Ntail \rangle$$

$$\langle Ntail \rangle ::= \langle empty \rangle \mid \beta \ \langle Ntail \rangle$$

$$\langle empty \rangle ::=$$

where $\langle Ntail \rangle$ is a new nonterminal. The conditions on the $\langle Ntail \rangle$ productions would be that $first(\beta) \cap follow\langle N \rangle = \emptyset$, and that β not generate a language containing the empty string. The corresponding recognition code would have the form

```
parse α;
while (t ∈ first(β))
{ parse β;
}
```

Exercise 11.3 Why, in the preceding discussion, would β generating the empty string be problematic?

Similarly, use of an option of the following form

$$\langle N \rangle ::= \cdots \mid \alpha \left[\beta \right]$$

may be re-expressed in conventional BNF as