kind of parse is produced. In both cases, the first character (L) states that the
token sequence is processed from left to right. The second letter (L or R) indicates
whether a leftmost or rightmost parse is produced. The parsing technique can be
further characterized by the number of lookahead symbols \(i.e.,\) symbols beyond
the current token) that the parser may consult to make parsing choices. LL(1) and
LR(1) parsers are the most common, requiring only one symbol of lookahead.

4.5 Grammar Analysis Algorithms

It is often necessary to analyze a grammar to determine if it is suitable for parsing
and, if so, to construct tables that can drive a parsing algorithm. In this section,
we discuss a number of important analysis algorithms, and so strengthen the
basic concepts of grammars and derivations. These algorithms are central to the
automatic construction of parsers, as discussed in Chapters Chapter:global:five
and Chapter:global:six.

4.5.1 Grammar Representation

The algorithms presented in this chapter refer to a collection of utilities for accessing
and modifying representations of a CFG. The efficiency of these algorithms
is affected by the data structures upon which these utilities are built. In this
section, we examine how to represent CFGs efficiently. We assume that the
implementation programming language offers the following constructs directly or
by augmentation.

- A **set** is an unordered collection of distinct objects.
- A **list** is an ordered collection of objects. An object can appear multiple
times in a list.
- An **iterator** is a construct that enumerates the contents of a set or list.

As discussed in Section 4.1, a grammar formally contains two disjoint sets of
symbols, \(\Sigma\) and \(N\), which contain the grammar’s terminals and nonterminals,
respectively. Grammars also contain a designated start symbol and a set of
productions. The following observations are relevant to obtaining an efficient
representation for grammars.

- Symbols are rarely deleted from a grammar.
- Transformations such as those shown in Figure 4.4 can add symbols and
productions to a grammar.
- Grammar-based algorithms typically visit all rules for a given nonterminal
or visit all occurrences of a given symbol in the productions.
- Most algorithms process a production’s RHS one symbol at a time.
Based on these observations, we represent a production by its LHS and a list of
the symbols on its RHS. The empty string $\lambda$ is not represented explicitly as a
symbol. Instead, a production $A \rightarrow \lambda$ has an empty list of symbols for its RHS.
The collection of grammar utilities is as follows.

**Grammar(S):** Creates a new grammar with start symbol $S$ and no productions.

**Production(A, rhs):** Creates a new production for nonterminal $A$ and returns a
descriptor for the production. The iterator $\text{rhs}$ supplies the symbols for the
production’s RHS.

**Productions():** Returns an iterator that visits each production in the grammar.

**Nonterminal(A):** Adds $A$ to the set of nonterminals. An error occurs if $A$
is already a terminal symbol. The function returns a descriptor for the
nonterminal.

**Terminal(x):** Adds $x$ to the set of terminals. An error occurs if $x$ is already a
nonterminal symbol. The function returns a descriptor for the terminal.

**NonTerminals():** Returns an iterator for the set of nonterminals.

**Terminals():** Returns an iterator for the set of terminal symbols.

**IsTerminal(x):** Returns $\text{true}$ if $x$ is a terminal; otherwise, returns $\text{false}$.

**RHS(p):** Returns an iterator for the symbols on the RHS of production $p$.

**LHS(p):** Returns the nonterminal defined by production $p$.

**ProductionsFor(A):** Returns an iterator that visits each production for nonter-
minal $A$.

**Occurrences(x):** Returns an iterator that visits each occurrence of $X$ in the
RHS of all rules.

**Production(y):** Returns a descriptor for the production $A \rightarrow a$ where $a$ contains
the occurrence $y$ of some vocabulary symbol.

**Tail(y):** Accesses the symbols appearing after an occurrence. Given a symbol
occurrence $y$ in the rule $A \rightarrow a \ y \ \beta$, $\text{Tail}(y)$ returns an iterator for the symbols in $\beta$.

### 4.5.2 Deriving the Empty String

One of the most common grammar computations is determining which nonter-
minals can derive $\lambda$. This information is important because such nonterminals
may disappear during a parse and hence must be carefully handled. Determining
4.5. Grammar Analysis Algorithms

procedure DerivesEmptyString()
    foreach A ∈ NonTerminals() do
        SymbolDerivesEmpty(A) ← false
    foreach p ∈ Productions() do
        RuleDerivesEmpty(p) ← false
        call CountSymbols(p) 1
        call CheckForEmpty(p)
    foreach X ∈ WorkList do 2
        WorkList ← WorkList − {X}
    foreach x ∈ Occurrences(X) do 4
        p ← Production(x)
        Count(p) ← Count(p) − 1
        call CheckForEmpty(p)
end

procedure CountSymbols(p)
    Count(p) ← 0
    foreach X ∈ RHS(p) do Count(p) ← Count(p) + 1
end

procedure CheckForEmpty(p)
    if Count(p) = 0
        then
            RuleDerivesEmpty(p) ← true 5
            A ← LHS(p)
            if not SymbolDerivesEmpty(A)
            then
                SymbolDerivesEmpty(A) ← true 6
                WorkList ← WorkList ∪ {A}
            end
        end
end

Figure 4.7: Algorithm for determining nonterminals and productions that can derive λ.
if a nonterminal can derive \( \lambda \) is not entirely trivial because the derivation can take more than one step:

\[
A \Rightarrow BCD \Rightarrow BC \Rightarrow B \Rightarrow \lambda.
\]

An algorithm to compute the productions and symbols that can derive \( \lambda \) is shown in Figure 4.7. The computation utilizes a worklist at Step 2. A worklist is a set that is augmented and diminished as the algorithm progresses. The algorithm is finished when the worklist is empty. Thus the loop at Step 2 must account for changes to the set WorkList. To prove termination of algorithms that utilize worklists, it must be shown that all worklist elements appear a finite number of times.

In the algorithm of Figure 4.7, the worklist contains nonterminals that are discovered to derive \( \lambda \). The integer Count\((p)\) is initialized at Step 1 to the number of symbols on \( p \)'s RHS. The count for any production of the form \( A \rightarrow \lambda \) is 0. Once a production is known to derive \( \lambda \), its LHS is placed on the worklist at Step 7. When a symbol is taken from the worklist at Step 3, each occurrence of the symbol is visited at Step 4 and the count of the associated production is decremented by 1. This process continues until the worklist is exhausted. The algorithm establishes two structures related to derivations of \( \lambda \), as follows.

- RuleDerivesEmpty\((p)\) indicates whether production \( p \) can derive \( \lambda \). When every symbol in rule \( p \)'s RHS can derive \( \lambda \), Step 5 establishes that \( p \) can derive \( \lambda \).

- SymbolDerivesEmpty\((A)\) indicates whether the nonterminal \( A \) can derive \( \lambda \). When any production for \( A \) can derive \( \lambda \), Step 6 establishes that \( A \) can derive \( \lambda \).

Both forms of information are useful in the grammar analysis and parsing algorithms discussed in Chapters 4, Chapter:global:five, and Chapter:global:six.

### 4.5.3 First Sets

A set commonly consulted by parser generators is First\((a)\). This is the set of all terminal symbols that can begin a sentential form derivable from the string of grammar symbols in \( a \). Formally,

\[
\text{First}(a) = \{ a \in \Sigma \mid a \Rightarrow^* \lambda \}.
\]

Some texts include \( \lambda \) in First\((a)\) if \( a \Rightarrow^* \lambda \). The resulting algorithms require frequent subtraction of \( \lambda \) from symbol sets. We adopt the convention of never including \( \lambda \) in First\((a)\). Testing whether a given string of symbols \( a \) derives \( \lambda \) is easily accomplished—when the results from the algorithm of Figure 4.7 are available.

First\((a)\) is computed by scanning \( a \) left-to-right. If \( a \) begins with a terminal symbol \( a \), then clearly First\((a) = \{ a \} \). If a nonterminal symbol \( A \) is encountered,
4.5. Grammar Analysis Algorithms

function First(α) : Set
     foreach A ∈ NonTerminals() do
          VisitedFirst(A) ← false
          ans ← InternalFirst(α)
     end
     return (ans)

function InternalFirst(Xβ) : Set
     if Xβ = ⊥ then
          return (ф)
     end
     if X ∈ Σ then
          return ([X])
     end
     ans ← ф
     if not VisitedFirst(X) then
          VisitedFirst(X) ← true
          foreach rhs ∈ ProductionsFor(X) do
               ans ← ans ∪ InternalFirst(rhs)
          end
          if SymbolDerivesEmpty(X) then
               ans ← ans ∪ InternalFirst(β)
          end
     end
     return (ans)

Figure 4.8: Algorithm for computing First(α).

then the grammar productions for A must be consulted. Nonterminals that can derive \( \lambda \) potentially disappear during a derivation, so the computation must account for this as well.

As an example, consider the nonterminals Tail and Prefix from the grammar in Figure 4.1. Each nonterminal has one production that contributes information directly to the nonterminal’s First set. Each nonterminal also has a \( \lambda \)-production, which contributes nothing. The solutions are as follows.

First(Tail) = \{+\}
First(Prefix) = \{f\}

In some situations, the First set of one symbol can depend on the First sets of other symbols. To compute First(E), the production \( E \rightarrow \text{Prefix} (\ E \) ) requires computation of First(Prefix). Because \( \text{Prefix} \Rightarrow^* \lambda \), First( ( \ E \ )) must also be included. The resulting set is as follows.

First(E) = \{v, f, (\}
Termination of $\text{First}(A)$ must be handled properly in grammars where the computation of $\text{First}(A)$ appears to depend on $\text{First}(A)$, as follows.

$$
\begin{align*}
    A & \rightarrow B \\
    & \vdots \\
    B & \rightarrow C \\
    & \vdots \\
    C & \rightarrow A
\end{align*}
$$

In this grammar, $\text{First}(A)$ depends on $\text{First}(B)$, which depends on $\text{First}(C)$, which depends on $\text{First}(A)$. In computing $\text{First}(A)$, we must avoid endless iteration or recursion. A sophisticated algorithm could preprocess the grammar to determine such cycles of dependence. We leave this as Exercise 17 and present a clearer but slightly less efficient algorithm in Figure 4.8. This algorithm avoids endless computation by remembering which nonterminals have already been visited, as follows.

- $\text{First}(\alpha)$ is computed by invoking $\text{First}(\alpha)$.
- Before any sets are computed, Step 8 resets $\text{VisitedFirst}(A)$ for each nonterminal $A$.
- $\text{VisitedFirst}(X)$ is set at Step 12 to indicate that the productions of $A$ already participate in the computation of $\text{First}(\alpha)$.

The primary computation is carried out by the function $\text{InternalFirst}$, whose input argument is the string $X\beta$. If $X\beta$ is not empty, then $X$ is the string’s first symbol and $\beta$ is the rest of the string. $\text{InternalFirst}$ then computes its answer as follows.

- The empty set is returned if $X\beta$ is empty at Step 9. We denote this condition by $\bot$ to emphasize that the empty set is represented by a null list of symbols.
- If $X$ is a terminal, then $\text{First}(X\beta)$ is $\{X\}$ at Step 10.
- The only remaining possibility is that $X$ is a nonterminal. If $\text{VisitedFirst}(X)$ is false, then the productions for $X$ are recursively examined for inclusion. Otherwise, $X$’s productions already participate in the current computation.
- If $X$ can derive $\lambda$ at Step 14—this fact has been previously computed by the algorithm in Figure 4.7—then we must include all symbols in $\text{First}(\beta)$.

Figure 4.9 shows the progress of $\text{ComputeFirst}$ as it is invoked on the nonterminals of Figure 4.1. The level of recursion is shown in the leftmost column. Each call to $\text{First}(X\beta)$ is shown with nonblank entries in the $X$ and $\beta$ columns. A “*” indicates that the call does not recurse further. Figure 4.10 shows another grammar and the computation of its $\text{First}$ sets; for brevity, recursive calls to $\text{InternalFirst}$ on null strings are omitted.
4.5. Grammar Analysis Algorithms

<table>
<thead>
<tr>
<th>Level</th>
<th>First</th>
<th>ans</th>
<th>Step</th>
<th>Done?</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\chi$</td>
<td>$\beta$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**ComputeFirst(Tail)**

<table>
<thead>
<tr>
<th>Level</th>
<th>First</th>
<th>ans</th>
<th>Step</th>
<th>Done?</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Tail</td>
<td>⊥</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>+</td>
<td>⊥</td>
<td></td>
<td>Step 11</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>⊥</td>
<td>E</td>
<td>Step 10</td>
<td>★ Tail $\rightarrow$ E</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>⊥</td>
<td>⊥</td>
<td>Step 9</td>
<td>★ Tail $\rightarrow$ λ</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>⊥</td>
<td>⊥</td>
<td>Step 13</td>
<td>After all rules for Tail</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>⊥</td>
<td>⊥</td>
<td>Step 9</td>
<td>★ Since $\beta = ⊥$</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>⊥</td>
<td>⊥</td>
<td>Step 14</td>
<td>★ Final answer</td>
<td></td>
</tr>
</tbody>
</table>

**ComputeFirst(Prefix)**

<table>
<thead>
<tr>
<th>Level</th>
<th>First</th>
<th>ans</th>
<th>Step</th>
<th>Done?</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Prefix</td>
<td>⊥</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>f</td>
<td>⊥</td>
<td></td>
<td>Step 10</td>
<td>★ Prefix $\rightarrow$ f</td>
</tr>
<tr>
<td>1</td>
<td>⊥</td>
<td>⊥</td>
<td>Step 9</td>
<td>★ Prefix $\rightarrow$ λ</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>⊥</td>
<td>⊥</td>
<td>Step 13</td>
<td>After all rules for Prefix</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>⊥</td>
<td>⊥</td>
<td>Step 9</td>
<td>★ Since $\beta = ⊥$</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>⊥</td>
<td>⊥</td>
<td>Step 14</td>
<td>★ Final answer</td>
<td></td>
</tr>
</tbody>
</table>

**ComputeFirst(E)**

<table>
<thead>
<tr>
<th>Level</th>
<th>First</th>
<th>ans</th>
<th>Step</th>
<th>Done?</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>E</td>
<td>⊥</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Prefix</td>
<td>(E)</td>
<td></td>
<td>Step 11</td>
<td>E $\rightarrow$ Prefix (E)</td>
</tr>
<tr>
<td>1</td>
<td>⊥</td>
<td>⊥</td>
<td>Step 15</td>
<td>Computation shown above</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>⊥</td>
<td>(E)</td>
<td>Step 10</td>
<td>★ Since Prefix $\Rightarrow^* λ$</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>⊥</td>
<td>⊥</td>
<td>Step 14</td>
<td>★ Results due to E $\Rightarrow$ Prefix (E)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>⊥</td>
<td>⊥</td>
<td>Step 10</td>
<td>★ E $\rightarrow$ v Tail</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>⊥</td>
<td>⊥</td>
<td>Step 9</td>
<td>★ Since $\beta = ⊥$</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>⊥</td>
<td>⊥</td>
<td>Step 14</td>
<td>★ Final answer</td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.9: First sets for the nonterminals of Figure 4.1.

4.5.4 Follow Sets

Parser-construction algorithms often require the computation of the set of terminals that can follow a nonterminal A in some sentential form. Because we augment grammars to contain an end-of-input token ($\$, every nonterminal except the goal symbol must be followed by some terminal. Formally, for $A \in N$,

Follow($A$) = \{ $b \in \Sigma$ | $S \Rightarrow^+ A \ b \ \beta$ \}.

Follow($A$) provides the right context associated with nonterminal $A$. For example, only those terminals in Follow($A$) can occur after a production for $A$ is applied.

The algorithm shown in Figure 4.11 computes Follow($A$). Many aspects of this algorithm are similar to the First($a$) algorithm given in Figure 4.8.
Figure 4.10: A grammar and its First sets.
4.5. Grammar Analysis Algorithms

function Follow(A) : Set
    foreach A ∈ NonTerminals() do
        VisitedFollow(A) ← false 16
        ans ← InternalFollow(A)
    return (ans)
end

function InternalFollow(A) : Set
    ans ← ∅ 17
    if not VisitedFollow(A) then
        VisitedFollow(A) ← true 18
        foreach a ∈ Occurrences(A) do
            ans ← ans ∪ First(Tail(a)) 19
            if AllDeriveEmpty(Tail(a)) then
                tar ← LHS(Production(a))
                ans ← ans ∪ InternalFollow(tar) 20
            end
        return (ans)
    end
end

function AllDeriveEmpty(γ) : Boolean
    foreach X ∈ γ do
        if not SymbolDerivesEmpty(X) or X ∈ Σ then return (false)
    return (true)
end

Figure 4.11: Algorithm for computing Follow(A).

- Follow(A) is computed by invoking Follow(A).
- Before any sets are computed, Step 16 resets VisitedFollow(A) for each non-terminal A.
- VisitedFollow(A) is set at Step 18 to indicate that the symbols following A are already participating in this computation.

The primary computation is performed by InternalFollow(A). Each occurrence a of A is visited by the loop at Step 19. Tail(a) is the list of symbols immediately following the occurrence of A.

- Any symbol in First(Tail(a)) can follow A. Step 20 includes such symbols in the returned set.
- Step 21 detects if the symbols in Tail(a) can derive λ. This situation arises
<table>
<thead>
<tr>
<th>Level</th>
<th>Rule</th>
<th>Step</th>
<th>Result</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>E → Prefix (E)</td>
<td>20</td>
<td>{}</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Follow(Prefix)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**ComputeFollow(E)**

<table>
<thead>
<tr>
<th>Level</th>
<th>Rule</th>
<th>Step</th>
<th>Result</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>E → Prefix (E)</td>
<td>20</td>
<td>{}</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Follow(E)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Tail→+E</td>
<td>22</td>
<td>{}</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>E→ν Tail</td>
<td>22</td>
<td>{}</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Follow(E)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**ComputeFollow(Tail)**

<table>
<thead>
<tr>
<th>Level</th>
<th>Rule</th>
<th>Step</th>
<th>Result</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>E→ν Tail</td>
<td>22</td>
<td>{}</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Follow(E)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>E→Prefix (E)</td>
<td>20</td>
<td>{}</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Tail→+E</td>
<td>22</td>
<td>{}</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Follow(Tail)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Follow**

<table>
<thead>
<tr>
<th>Level</th>
<th>Rule</th>
<th>Step</th>
<th>Result</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>E→ν Tail</td>
<td>22</td>
<td>{}</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Follow(E)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>E→Prefix (E)</td>
<td>20</td>
<td>{}</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Tail→+E</td>
<td>22</td>
<td>{}</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Follow(Tail)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.12: Follow sets for the nonterminals of Figure 4.1.

when there are no symbols appearing after this occurrence of A or when
the symbols appearing after A can each derive λ. In either case, Step 22
includes the Follow set of the current production’s LHS.

Figure 4.12 shows the progress of ComputeFollow as it is invoked on the nonterminals of Figure 4.1. As another example, Figure 4.13 shows the computation of Follow sets for the grammar in Figure 4.10.

First and Follow sets can be generalized to include strings of length k rather than length 1. First_k(α) is the set of k-symbol terminal prefixes derivable from α. Similarly, Follow_k(A) is the set of k-symbol terminal strings that can follow A in some sentential form. First_k and Follow_k are used in the definition of parsing techniques that use k-symbol lookaheads (for example, LL(k) and LR(k)). The algorithms that compute First_k(α) and Follow_k(A) can be generalized to compute First_k(α) and Follow_k(A) sets (see Exercise 24).
4.5. Grammar Analysis Algorithms

<table>
<thead>
<tr>
<th>Level</th>
<th>Rule</th>
<th>Step</th>
<th>Result</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ComputeFollow(B)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Follow(B)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>S → A B c</td>
<td>Step 20</td>
<td>{c}</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>S → A B c</td>
<td>Step 23</td>
<td>{c}</td>
<td>Returns</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Level</th>
<th>Rule</th>
<th>Step</th>
<th>Result</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ComputeFollow(A)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Follow(A)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>S → A B c</td>
<td>Step 20</td>
<td>{b,c}</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>S → A B c</td>
<td>Step 23</td>
<td>{b,c}</td>
<td>Returns</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Level</th>
<th>Rule</th>
<th>Step</th>
<th>Result</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ComputeFollow(S)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Follow(S)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>S → B B c</td>
<td>Step 23</td>
<td>{}</td>
<td>Returns</td>
</tr>
</tbody>
</table>

Figure 4.13: Follow sets for the grammar in Figure 4.10. Note that Follow(S) = {} because S does not appear on the RHS of any production.

This ends our discussion of CFGs and grammar-analysis algorithms. The First and Follow sets introduced in this chapter play an important role in the automatic construction of LL and LR parsers, as discussed in Chapters Chapter:global:five and Chapter:global:six, respectively.