

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  $rhs$  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.

**IS\_TERMINAL(X)**: Returns **true** if  $X$  is a terminal; otherwise, returns **false**.

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

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

**PRODUCTIONS\_FOR(A)**: Returns an iterator that visits each production for nonterminal  $A$ .

**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  $A \rightarrow \alpha$  where  $\alpha$  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  $A \rightarrow \alpha y \beta$ , **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 nonterminals can derive  $\lambda$ . This information is important because such nonterminals may disappear during a parse and hence must be carefully handled. Determining

```

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} 3
    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} 7
  end

```

Figure 4.7: Algorithm for determining nonterminals and productions that can derive  $\lambda$ .

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  $First(\alpha)$ . This is the set of all terminal symbols that can begin a sentential form derivable from the string of grammar symbols in  $\alpha$ . Formally,

$$First(\alpha) = \{a \in \Sigma \mid \alpha \Rightarrow^* a \beta\}.$$

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

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

```

function FIRST( $\alpha$ ): Set
    foreach A  $\in$  NONTERMINALS() do VisitedFirst(A)  $\leftarrow$  false      8
    ans  $\leftarrow$  INTERNALFIRST( $\alpha$ )
    return (ans)
end
function INTERNALFIRST( $X\beta$ ): Set
    if  $X\beta = \perp$                                                     9
    then return ( $\emptyset$ )
    if  $X \in \Sigma$                                                   10
    then return ( $\{X\}$ )
    /*                                                                */
    /*          X is a nonterminal.                                */
    /*                                                                */
    ans  $\leftarrow$   $\emptyset$ 
    if not VisitedFirst(X)
    then
        VisitedFirst(X)  $\leftarrow$  true                                12
        foreach rhs  $\in$  ProductionsFor(X) do
            ans  $\leftarrow$  ans  $\cup$  INTERNALFIRST(rhs)                13
        if SymbolDerivesEmpty(X)                                    14
        then ans  $\leftarrow$  ans  $\cup$  INTERNALFIRST( $\beta$ )
        return (ans)                                              15
    end

```

Figure 4.8: Algorithm for computing First( $\alpha$ ).

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.

$$\begin{aligned} \text{First}(\text{Tail}) &= \{+\} \\ \text{First}(\text{Prefix}) &= \{f\} \end{aligned}$$

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.

$$\text{First}(E) = \{v, f, ()\}$$

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.

$$\begin{array}{l} A \rightarrow B \\ \dots \\ B \rightarrow C \\ \dots \\ C \rightarrow A \end{array}$$

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  $\perp$  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 First sets; for brevity, recursive calls to  $\text{INTERNALFIRST}$  on null strings are omitted.

Level	First $X$	$\beta$	$ans$	Step	Done?	Comment
COMPUTEFIRST(Tail)						
0	Tail	$\perp$	{ }	Step 11		
1	+	E	{+}	Step 10	★	Tail $\rightarrow$ +E
1	$\perp$	$\perp$	{ }	Step 9	★	Tail $\rightarrow$ $\lambda$
0			{+}	Step 13		After all rules for Tail
1	$\perp$	$\perp$	{ }	Step 9	★	Since $\beta = \perp$
0			{+}	Step 14	★	Final answer
COMPUTEFIRST(Prefix)						
0	Prefix	$\perp$	{ }	Step 11		
1	f	$\perp$	{f}	Step 10	★	Prefix $\rightarrow$ f
1	$\perp$	$\perp$	{ }	Step 9	★	Prefix $\rightarrow$ $\lambda$
0			{f}	Step 13		After all rules for Prefix
1	$\perp$	$\perp$	{ }	Step 9	★	Since $\beta = \perp$
0			{f}	Step 14	★	Final answer
COMPUTEFIRST(E)						
0	E	$\perp$	{ }	Step 11		
1	Prefix	( E )	{ }	Step 11		E $\rightarrow$ Prefix ( E )
1			{f}	Step 15		Computation shown above
2	(	E )	{( }	Step 10	★	Since Prefix $\Rightarrow^*$ $\lambda$
1			{f,( }	Step 14	★	Results due to E $\rightarrow$ Prefix ( E )
1	v	Tail	{v }	Step 10	★	E $\rightarrow$ v Tail
1	$\perp$	$\perp$	{ }	Step 9		Since $\beta = \perp$
0			{f,(,v }	Step 14	★	Final answer

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  $A \in N$ ,

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

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

The algorithm shown in Figure 4.11 computes  $\text{Follow}(A)$ . Many aspects of this algorithm are similar to the  $\text{First}(\alpha)$  algorithm given in Figure 4.8.

1  $S \rightarrow A B c$   
 2  $A \rightarrow a$   
 3  $\quad \mid \lambda$   
 4  $B \rightarrow b$   
 5  $\quad \mid \lambda$

Level	First $X$	$\beta$	$ans$	Step	Done?	Comment
COMPUTEFIRST(B)						
0	B	$\perp$	{ }	Step 11		
1	b	$\perp$	{b}	Step 10	★	$B \rightarrow b$
1	$\perp$	$\perp$	{ }	Step 9	★	$B \rightarrow \lambda$
0			{b}	Step 14	★	Final answer
COMPUTEFIRST(A)						
0	A	$\perp$	{ }	Step 11		
1	a	$\perp$	{a}	Step 10	★	$A \rightarrow a$
1	$\perp$	$\perp$	{ }	Step 9	★	$A \rightarrow \lambda$
0			{a}	Step 14	★	Final answer
COMPUTEFIRST(S)						
0	S	$\perp$	{ }	Step 11		
1	A	B c	{a}	Step 15		Computation shown above
2	B	c	{b}	Step 15		Because $A \Rightarrow^* \lambda$ ; computation shown above
3	c	$\perp$	{c}	Step 10	★	Because $B \Rightarrow^* \lambda$
2			{b,c}	Step 14	★	
1			{a,b,c}	Step 14	★	
0			{a,b,c}	Step 14	★	

Figure 4.10: A grammar and its First sets.

## 4.5. Grammar Analysis Algorithms

21

```

function FOLLOW(A) : Set
  foreach A ∈ NONTERMINALS() do
    VisitedFollow(A) ← false
  ans ← INTERNALFOLLOW(A)
  return (ans)
end
function INTERNALFOLLOW(A) : Set
  ans ← ∅
  if not VisitedFollow(A)
  then
    VisitedFollow(A) ← true
    foreach a ∈ OCCURRENCES(A) do
      ans ← ans ∪ FIRST(TAIL(a))
      if ALLDERIVEEMPTY(TAIL(a))
      then
        targ ← LHS(PRODUCTION(a))
        ans ← ans ∪ INTERNALFOLLOW(targ)
    return (ans)
  end
function ALLDERIVEEMPTY( $\gamma$ ) : Boolean
  foreach X ∈  $\gamma$  do
    if not SymbolDerivesEmpty(X) or X ∈  $\Sigma$ 
    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  $\lambda$ . This situation arises

Level	Rule	Step	Result	Comment
			COMPUTEFOLLOW(Prefix)	
0			FOLLOW(Prefix)	
0	$E \rightarrow \underline{\text{Prefix}} ( E )$	Step <b>20</b>	{ ( }	
			COMPUTEFOLLOW(E)	
0			FOLLOW(E)	
0	$E \rightarrow \text{Prefix} ( \underline{E} )$	Step <b>20</b>	{ ) }	
0	$\text{Tail} \rightarrow + \underline{E}$	Step <b>22</b>	{ }	
1			FOLLOW(Tail)	
1	$E \rightarrow v \underline{\text{Tail}}$	Step <b>22</b>	{ }	
2			FOLLOW(E)	
		Step <b>17</b>	{ }	Recursion avoided
1		Step <b>23</b>	{ }	Returns
0		Step <b>23</b>	{ ) }	Returns
			COMPUTEFOLLOW(Tail)	
0			FOLLOW(Tail)	
0	$E \rightarrow v \underline{\text{Tail}}$	Step <b>22</b>	{ }	
1			FOLLOW(E)	
1	$E \rightarrow \text{Prefix} ( \underline{E} )$	Step <b>20</b>	{ ) }	
1	$\text{Tail} \rightarrow + \underline{E}$	Step <b>22</b>	{ }	
2			FOLLOW(Tail)	
		Step <b>17</b>	{ }	Recursion avoided
1		Step <b>23</b>	{ ) }	Returns
0		Step <b>23</b>	{ ) }	Returns

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  $\lambda$ . 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.  $\text{First}_k(\alpha)$  is the set of  $k$ -symbol terminal prefixes derivable from  $\alpha$ . Similarly,  $\text{Follow}_k(A)$  is the set of  $k$ -symbol terminal strings that can follow  $A$  in some sentential form.  $\text{First}_k$  and  $\text{Follow}_k$  are used in the definition of parsing techniques that use  $k$ -symbol lookaheads (for example,  $\text{LL}(k)$  and  $\text{LR}(k)$ ). The algorithms that compute  $\text{First}_1(\alpha)$  and  $\text{Follow}_1(A)$  can be generalized to compute  $\text{First}_k(\alpha)$  and  $\text{Follow}_k(A)$  sets (see Exercise 24).

Level	Rule	Step	Result	Comment
			COMPU <b>T</b> E <b>F</b> OLLOW(B)	
0			FOLLOW(B)	
0	$S \rightarrow A \underline{B} c$	Step <b>20</b>	{c}	
0		Step <b>23</b>	{c}	Returns
			COMPU <b>T</b> E <b>F</b> OLLOW(A)	
0			FOLLOW(A)	
0	$S \rightarrow \underline{A} B c$	Step <b>20</b>	{b,c}	
0		Step <b>23</b>	{b,c}	Returns
			COMPU <b>T</b> E <b>F</b> OLLOW(S)	
0			FOLLOW(S)	
0		Step <b>23</b>	{ }	Returns

Figure 4.13: Follow sets for the grammar in Figure 4.10. Note that  $\text{Follow}(S) = \{ \}$  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 Chapter:global:five and Chapter:global:six, respectively.