4

Grammars and Parsing

Formally a **language** is a set of finite-length strings over a finite alphabet. Because most interesting languages are infinite sets, we cannot define such languages by enumerating their elements. A **grammar** is a compact, finite representation of a language. In Chapter Chapter:global:two, we discuss the rudiments of **context-free grammars** (CFGs) and define a simple language using a CFG. Due to their efficiency and precision, CFGs are commonly used for defining the syntax of actual programming languages. In Chapter 4, we formalize the definition and notation for CFGs and present algorithms that analyze such grammars in preparation for the parsing techniques covered in Chapters Chapter:global:five and Chapter:global:six.

4.1 Context-Free Grammars: Concepts and Notation

A CFG is defined by the following four components.

1. A finite **terminal alphabet** Σ. This is the set of tokens produced by the scanner. We always augment this set with the token \$, which signifies end-of-input.

2. A finite **nonterminal alphabet** N. Symbols in this alphabet are *variables* of the grammar.

3. A **start symbol** $\mathsf{S} \in N$ that initiates all *derivations*. S is also called the **goal symbol**.

4. P, a finite set of productions (sometimes called **rewriting rules**) of the form $\mathsf{A} \rightarrow \mathcal{X}_1 \ldots \mathcal{X}_m$, where $\mathsf{A} \in N$, $\mathcal{X}_i \in N \cup \Sigma$, $1 \leq i \leq m$, and $m \geq 0$. The

only valid production with $m = 0$ is of the form $\mathsf{A} \rightarrow \lambda$, where λ denotes the **empty string**.

These components are often grouped into a four-tuple, $\mathsf{G} = (N, \Sigma, P, \mathsf{S})$, which is the formal definition of a CFG. The terminal and nonterminal alphabets must be disjoint (*i.e.*, $\Sigma \cap N = \emptyset$). The **vocabulary** V of a CFG is the set of terminal and nonterminal symbols (*i.e.*, $V = \Sigma \cup N$).

A CFG is essentially a recipe for creating strings. Starting with S, nonterminals are rewritten using the grammar's productions until only terminals remain. A rewrite using the production $\mathsf{A} \rightarrow \alpha$ replaces the nonterminal A with the vocabulary symbols in α. As a special case, a rewrite using the production $\mathsf{A} \rightarrow \lambda$ causes A to be erased. Each rewrite is a step in a **derivation** of the resulting string. The set of terminal strings derivable from S comprises the **context-free language** of grammar G, denoted L(G).

In describing parsers, algorithms, and grammars, consistency is useful in denoting symbols and strings of symbols. We follow the notation set out in the following chart.

Names Beginning With	Represent Symbols In	Examples
Uppercase	N	A, B, C, Prefix
Lowercase and punctuation	Σ	a, b, c, if, then, (, ;
$\mathcal{X}$, $\mathcal{Y}$	$N \cup \Sigma$	$\mathcal{X}_i$, $\mathcal{Y}_3$
Other Greek letters	$(N \cup \Sigma)^\star$	α, γ

Using this notation, we write a production as $\mathsf{A} \rightarrow \alpha$ or $\mathsf{A} \rightarrow \mathcal{X}_1 \ldots \mathcal{X}_m$, depending on whether the detail of the production's **right-hand side** (RHS) is of interest. This format emphasizes that a production's **left-hand side** (LHS) must be a single nonterminal but the RHS is a string of zero or more vocabulary symbols.

There is often more than one way to rewrite a given nonterminal; in such cases, multiple productions share the same LHS symbol. Instead of repeating the LHS symbol, an "or notation" is used.

$$\begin{array}{lcl} \mathsf{A} & \rightarrow & \alpha \\ & | & \beta \\ & \ldots & \\ & | & \zeta \end{array}$$

This is an abbreviation for the following sequence of productions.

$$\begin{array}{lcl} \mathsf{A} & \rightarrow & \alpha \\ \mathsf{A} & \rightarrow & \beta \\ & \ldots & \\ \mathsf{A} & \rightarrow & \zeta \end{array}$$

If $\mathsf{A} \rightarrow \gamma$ is a production, then $\alpha \mathsf{A} \beta \Rightarrow \alpha \gamma \beta$ denotes one step of a derivation using this production. We extend $\Rightarrow$ to $\Rightarrow^+$ (derived in one or more steps) and $\Rightarrow^\star$ (derived in zero or more steps). If $\mathsf{S} \Rightarrow^\star \beta$, then β is said to be a *sentential*

1	E	$\rightarrow$	Prefix (E)
2		|	v Tail
3	Prefix	$\rightarrow$	f
4		|	λ
5	Tail	$\rightarrow$	+ E
6		|	λ

Figure 4.1: A simple expression grammar.

form of the CFG. SF(G) denotes the set of sentential forms of grammar G. Thus $L(G) = \{ w \in \Sigma^\star \mid S \Rightarrow^+ w \}$. Also, $L(G) = SF(G) \cap \Sigma^\star$. That is, the language of G is simply those sentential forms of G that are terminal strings.

Throughout a derivation, if more than one nonterminal is present in a sentential form, then there is a choice as to which nonterminal should be expanded in the next step. Thus to characterize a derivation sequence, we need to specify, at each step, which nonterminal is expanded and which production is applied. We can simplify this characterization by adopting a convention such that nonterminals are rewritten in some systematic order. Two such conventions are the

- leftmost derivation and
- rightmost derivation.

4.1.1 Leftmost Derivations

A derivation that always chooses the *leftmost* possible nonterminal at each step is called a **leftmost derivation**. If we know that a derivation is leftmost, we need only specify the productions in order of their application; the expanded nonterminal is implicit. To denote derivations that are leftmost, we use $\Rightarrow_{lm}$, $\Rightarrow^+_{lm}$, and $\Rightarrow^\star_{lm}$. A sentential form produced via a leftmost derivation is called a **left sentential form**. The production sequence discovered by a large class of parsers—the top-down parsers—is a leftmost derivation. Hence, these parsers are said to produce a *leftmost parse*.

As an example, consider the grammar shown in Figure 4.1, which generates simple expressions (v represents a variable and f represents a function). A leftmost derivation of f (v + v) is as follows.

E	$\Rightarrow_{lm}$	Prefix (E)
	$\Rightarrow_{lm}$	f (E)
	$\Rightarrow_{lm}$	f (v Tail)
	$\Rightarrow_{lm}$	f (v + E)
	$\Rightarrow_{lm}$	f (v + v Tail)
	$\Rightarrow_{lm}$	f (v + v)

4.1.2 Rightmost Derivations

An alternative to a leftmost derivation is a **rightmost derivation** (sometimes called a **canonical derivation**). In such derivations the rightmost possible nonterminal is always expanded. This derivation sequence may seem less intuitive given the English convention of processing information left-to-right, but such derivations are produced by an important class of parsers, namely the bottom-up parsers discussed in Chapter Chapter:global:six.

As a bottom-up parser discovers the productions that derive a given token sequence, it traces a rightmost derivation, but the productions are applied in *reverse order*. That is, the last step taken in a rightmost derivation is the first production applied by the bottom-up parser; the first step involving the start symbol is the parer's final production. The sequence of productions applied by a bottom-up parser is called a **rightmost** or **canonical** parse. For derivations that are rightmost, the notation $\Rightarrow_{\mathrm{rm}}$, $\Rightarrow^{+}_{\mathrm{rm}}$, and $\Rightarrow^{\star}_{\mathrm{rm}}$ is used. A sentential form produced via a rightmost derivation is called a **right sentential form**. A rightmost derivation of the grammar shown in Figure 4.1 is as follows.

$$
\begin{aligned}
\mathsf{E} &\Rightarrow_{\mathrm{rm}} \mathsf{Prefix\ (\ E\ )} \\
&\Rightarrow_{\mathrm{rm}} \mathsf{Prefix\ (\ v\ Tail\ )} \\
&\Rightarrow_{\mathrm{rm}} \mathsf{Prefix\ (\ v\ +\ E\ )} \\
&\Rightarrow_{\mathrm{rm}} \mathsf{Prefix\ (\ v\ +\ v\ Tail\ )} \\
&\Rightarrow_{\mathrm{rm}} \mathsf{Prefix\ (\ v\ +\ v\ )} \\
&\Rightarrow_{\mathrm{rm}} \mathsf{f\ (\ v\ +\ v\ )}
\end{aligned}
$$

4.1.3 Parse Trees

A derivation is often represented by a *parse tree* (sometimes called a *derivation tree*). A **parse tree** has the following characteristics.

- It is rooted by the grammar's start symbol S.
- Each node is either a grammar symbol or λ.
- Its interior nodes are nonterminals. An interior node and its children represent the application of a production. That is, a node representing a nonterminal A can have offspring $\mathcal{X}_1, \mathcal{X}_2, \ldots, \mathcal{X}_m$ if, and only if, there exists a grammar production $\mathsf{A} \rightarrow \mathcal{X}_1\, \mathcal{X}_2 \ldots \mathcal{X}_m$. When a derivation is complete, each leaf of the corresponding parse tree is either a terminal symbol or λ.

Figure 4.2 shows the parse tree for f (v + v) using the grammar from Figure 4.1. Parse trees serve nicely to visualize how a string is structured by a grammar. A leftmost or rightmost derivation is essentially a textual representation of a parse tree, but the derivation conveys also the order in which the productions are applied.

A sentential form is derivable from a grammar's start symbol. Hence, a parse tree must exist for every sentential form. Given a sentential form and its parse tree, a **phrase** of the sentential form is a sequence of symbols descended from

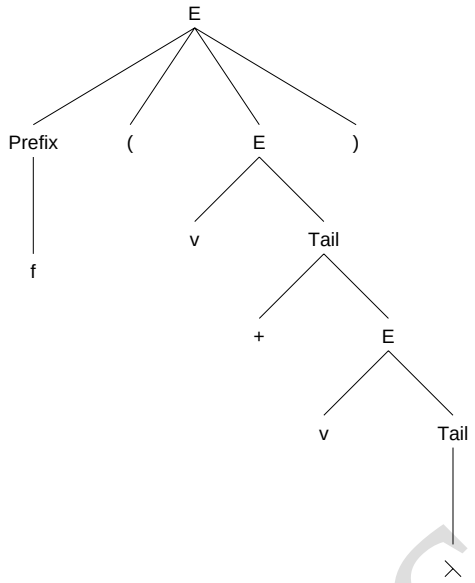

Figure 4.2: The parse tree for f (v + v) .

a single nonterminal in the parse tree. A **simple** or **prime phrase** is a phrase that contains no smaller phrase. That is, it is a sequence of symbols directly derived from a nonterminal. The **handle** of a sentential form is the leftmost simple phrase. (Simple phrases cannot overlap, so "leftmost" is unambiguous.) Consider the parse tree in Figure 4.2 and the sentential form f (v Tail). f and v Tail are simple phrases and f is the handle. Handles are important because they represent individual derivation steps, which can be recognized by various parsing techniques.

4.1.4 Other Types of Grammars

Although CFGs serve well to characterize syntax, most programming languages contain rules that are not expressible using CFGs. For example, the rule that variables must be declared before they are used cannot be expressed, because a CFG provides no mechanism for transmitting to the body of a program the exact set of variables that has been declared. In practice, syntactic details that cannot be represented in a CFG are considered part of the static semantics and are checked by semantic routines (along with scope and type rules). The non-CFGs that are relevant to programming language translation are the

- regular grammars, which are less powerful than CFGs, and the
- context-sensitive and unrestricted grammars, which are more powerful.

Regular Grammars

A CFG that is limited to productions of the form $\mathsf{A} \rightarrow \mathsf{a\ B}$ or $\mathsf{C} \rightarrow \mathsf{d}$ is a **regular grammar**. Each rule's RHS consists of either a symbol from $\Sigma \cup \{ \lambda \}$ followed by a nonterminal symbol or just a symbol from $\Sigma \cup \{ \lambda \}$. As the name suggests, a regular grammar defines a regular set (see Exercise 13.) We observed in Chapter Chapter:global:three that the language $\{ [^i]^i \mid i \geq 1 \}$ is not regular; this language is generated by the following CFG.

1	S	$\rightarrow$	T
2	T	$\rightarrow$	[T]
3		|	λ

This grammar establishes that the languages definable by regular grammars (regular sets) are a *proper* subset of the context-free languages.

Beyond Context-Free Grammars

CFGs can be generalized to create richer definitional mechanisms. A **context-sensitive grammar** requires that nonterminals be rewritten only when they appear in a particular context (for example, $\alpha \mathsf{A} \beta \rightarrow \alpha \delta \beta$), provided the rule never causes the sentential form to contract in length. An **unrestricted** or **type-0 grammar** is the most general. It allows arbitrary patterns to be rewritten.

Although context-sensitive and unrestricted grammars are more powerful than CFGs, they also are far less useful.

- Efficient parsers for such grammars do not exist. Without a parser, a grammar definition cannot participate in the automatic construction of compiler components.
- It is difficult to prove properties about such grammars. For example, it would be daunting to prove that a given type-0 grammar generates the C programming language.

Efficient parsers for many classes of CFGs do exist. Hence, CFGs present a nice balance between generality and practicality. Throughout this text, we focus on CFGs. Whenever we mention a grammar (without saying which kind), you should assume that the grammar is context-free.

4.2 Properties of CFGs

CFGs are a definitional mechanism for specifying languages. Just as there are many programs that compute the same result, so there are many grammars that generate the same language. Some of these grammars may have properties that complicate parser construction. The most common of these properties are

- the inclusion of useless nonterminals,

- allowing a derived string to have two or more different parse trees, and
- generating the wrong language.

In this section, we discuss these properties and their implication for language processing.

4.2.1 Reduced Grammars

A grammar is **reduced** if each of its nonterminals and productions participates in the derivation of some string in the grammar's language. Nonterminals that can be safely removed are called **useless**.

```
1  S  →  A
2     |  B
3  A  →  a
4  B  →  B b
5  C  →  c
```

The above grammar contains two kinds of nonterminals that cannot participate in any derived string.

- With S as the start symbol, the nonterminal C cannot appear in any phrase.
- Any phrase that mentions B cannot be rewritten to contain only terminals.

Exercises 14 and 15 consider how to detect both forms of useless nonterminals. When B, C, and their associated productions are removed, the following reduced grammar is obtained.

```
1  S  →  A
2  A  →  a
```

Many parser generators verify that a grammar is reduced. An unreduced grammar probably contains errors that result from mistyping of grammar specifications.

4.2.2 Ambiguity

Some grammars allow a derived string to have two or more different parse trees (and thus a nonunique structure). Consider the following grammar, which generates expressions using the infix operator for subtraction.

```
1  Expr  →  Expr - Expr
2        |  id
```

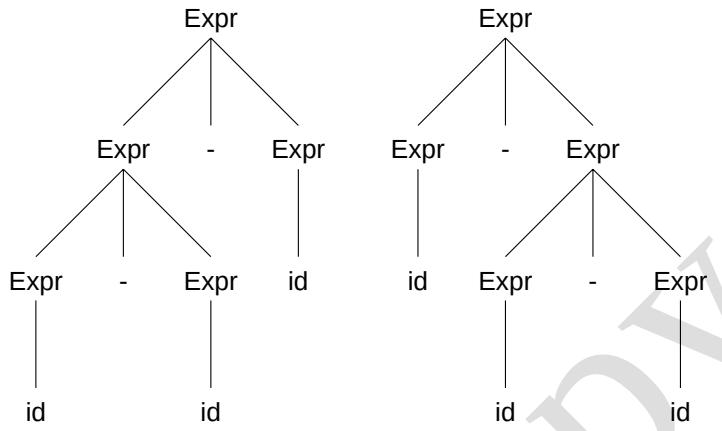

Figure 4.3: Two parse trees for id - id - id.

This grammar allows two different parse trees for id - id - id, as illustrated in Figure 4.3.

Grammars that allow different parse trees for the same terminal string are called **ambiguous**. They are rarely used because a unique structure (*i.e.*, parse tree) cannot be guaranteed for all inputs. Hence, a unique translation, guided by the parse tree structure, may not be obtained.

It seems we need an algorithm that checks an arbitrary CFG for ambiguity. Unfortunately, no algorithm is possible for this in the general case [HU79, Mar91]. Fortunately, for certain grammar classes, successful parser construction by the algorithms we discuss in Chapters Chapter:global:five and Chapter:global:six proves a grammar to be unambiguous.

4.2.3 Faulty Language Definition

The most potentially serious flaw that a grammar might have is that it generates the "wrong" language. That is, the terminal strings derivable by the grammar do not correspond *exactly* to the strings present in the desired language. This is a subtle point, because a grammar typically serves as the very definition of a language's syntax.

The correctness of a grammar is usually tested informally by attempting to parse a set of inputs, some of which are supposed to be in the language and some of which are not. One might try to compare for equality the languages defined by a pair of grammars (considering one a standard), but this is rarely done. For some grammar classes, such verification is possible; for others, no comparison algorithm is known. A general comparison algorithm applicable to all CFGs is known to be impossible to construct [Mar91].

foreach $p \in Prods$ of the form " $\mathsf{A} \rightarrow \alpha$ [$\mathcal{X}_1 \ldots \mathcal{X}_n$] β " **do**
  $N \leftarrow$ NEWNONTERM( )
  $p \leftarrow$ " $\mathsf{A} \rightarrow \alpha\ N\ \beta$ "
  $Prods \leftarrow Prods \cup \{$ " $N \rightarrow \mathcal{X}_1 \ldots \mathcal{X}_n$ " $\}$
  $Prods \leftarrow Prods \cup \{$ " $N \rightarrow \lambda$ " $\}$
foreach $p \in Prods$ of the form " $\mathsf{B} \rightarrow \gamma$ { $\mathcal{X}_1 \ldots \mathcal{X}_m$ } δ " **do**
  $M \leftarrow$ NEWNONTERM( )
  $p \leftarrow$ " $\mathsf{B} \rightarrow \gamma\ M\ \delta$ "
  $Prods \leftarrow Prods \cup \{$ " $M \rightarrow \mathcal{X}_1 \ldots \mathcal{X}_n\ M$ " $\}$
  $Prods \leftarrow Prods \cup \{$ " $M \rightarrow \lambda$ " $\}$

Figure 4.4: Algorithm to transform a BNF grammar into standard form.

4.3 Transforming Extended Grammars

Backus-Naur form (BNF) extends the grammar notation defined in Section 4.1 with syntax for defining optional and repeated symbols.

- Optional symbols are enclosed in square brackets. In the production

  $$\mathsf{A} \rightarrow \alpha\ [\ \mathcal{X}_1 \ldots \mathcal{X}_n\ ]\ \beta$$

  the symbols $\mathcal{X}_1 \ldots \mathcal{X}_n$ are entirely present or absent between the symbols of α and β.

- Repeated symbols are enclosed in braces. In the production

  $$\mathsf{B} \rightarrow \gamma\ \{\ \mathcal{X}_1 \ldots \mathcal{X}_m\ \}\ \delta$$

  the entire sequence of symbols $\mathcal{X}_1 \ldots \mathcal{X}_m$ can be repeated zero or more times.

These extensions are useful in representing many programming language constructs. In Java, declarations can optionally include modifiers such as `final`, `static`, and `const`. Each declaration can include a *list* of identifiers. A production specifying a Java-like declaration could be as follows.

Declaration → [final] [static] [const] Type identifier { , identifier }

This declaration insists that the modifiers be ordered as shown. Exercises 11 and 12 consider how to specify the optional modifiers in any order.

Although BNF can be useful, algorithms for analyzing grammars and building parsers assume the standard grammar notation as introduced in Section 4.1. The algorithm in Figure 4.4 transforms extended BNF grammars into standard form. For the BNF syntax involving braces, the transformation uses *right* recursion on M to allow zero or more occurrences of the symbols enclosed within braces. This transformation also works using left recursion—the resulting grammar would have generated the same language.

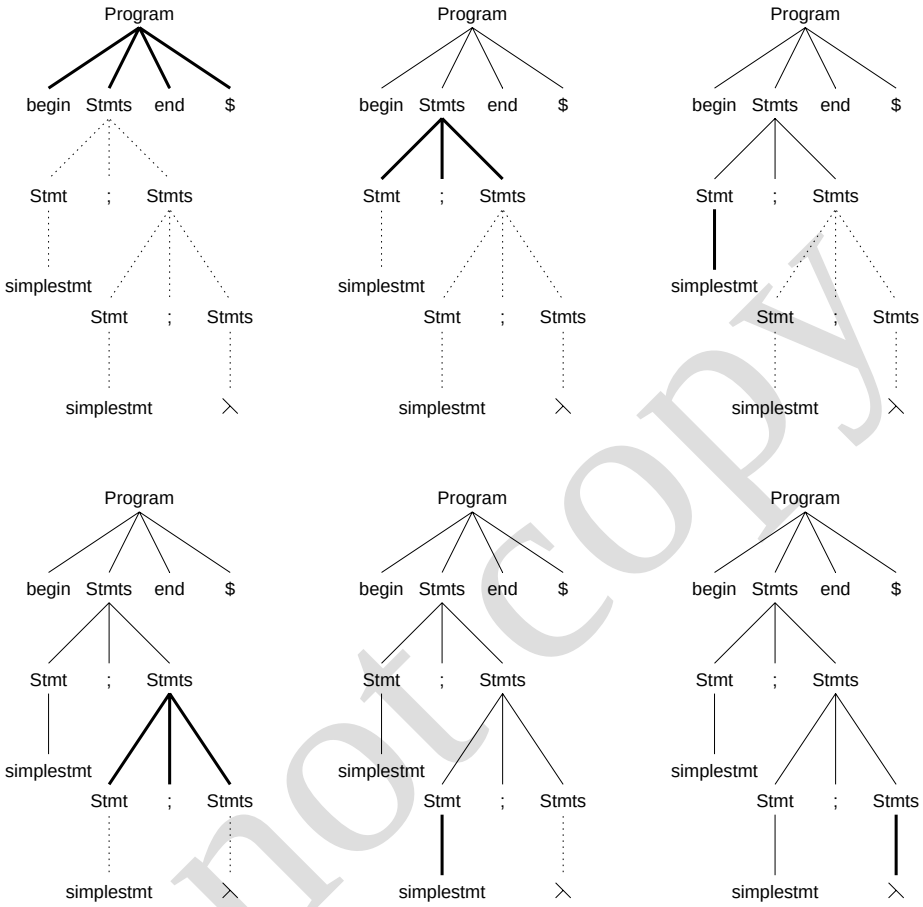

Figure 4.5: A top-down parse of begin simplestmt ; simplestmt ; end $.

As discussed in Section 4.1, a particular derivation (*e.g.*, leftmost or rightmost) depends on the structure of the grammar. It turns out that right-recursive rules are more appropriate for top-down parsers, which produce leftmost derivations. Similarly, left-recursive rules are more suitable for bottom-up parsers, which produce rightmost derivations.

4.4 Parsers and Recognizers

Compilers are expected to verify the syntactic validity of their inputs with respect to a grammar that defines the programming language's syntax. Given a grammar G and an input string x, the compiler must determine if $x \in L(G)$. An algorithm that performs this test is called a **recognizer**.

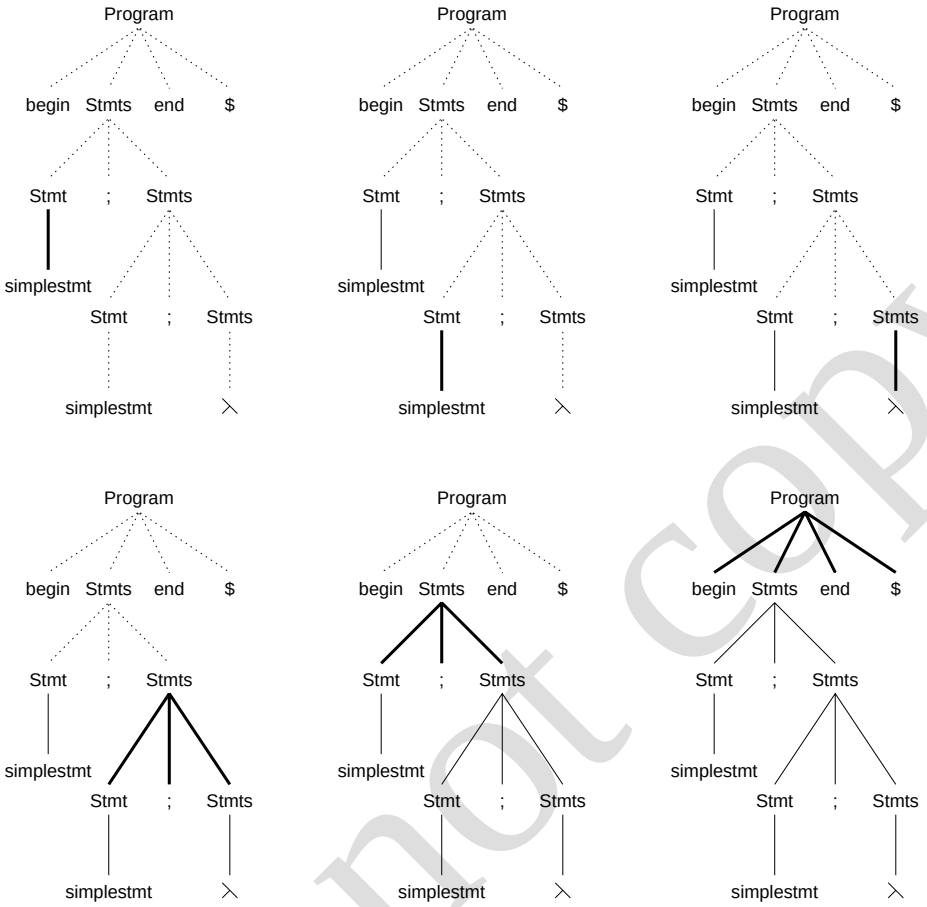

Figure 4.6: A bottom-up parse of begin simplestmt ; simplestmt ; end $.

For language translation, we must determine not only the string's validity, but also its structure, or **parse tree**. An algorithm for this task is called a **parser**. Generally, there are two approaches to parsing.

- A parser is considered **top-down** if it generates a parse tree by starting at the root of the tree (the start symbol), expanding the tree by applying productions in a depth-first manner. A top-down parse corresponds to a preorder traversal of the parse tree. Top-down parsing techniques are *predictive* in nature because they always predict the production that is to be matched before matching actually begins. The top-down approach includes the recursive-descent parser discussed in Chapter Chapter:global:two.

- The **bottom-up** parsers generate a parse tree by starting at the tree's leaves and working toward its root. A node is inserted in the tree only after its

children have been inserted. A bottom-up parse corresponds to a postorder traversal of the parse tree.

The following grammar generates the skeletal block structure of a programming language.

1	Program	$\rightarrow$	begin Stmts end $
2	Stmts	$\rightarrow$	Stmt ; Stmts
3		$\mid$	λ
4	Stmt	$\rightarrow$	simplestmt
5		$\mid$	begin Stmts end

Using this grammar, Figures 4.5 and 4.6 illustrate a top-down and bottom-up parse of the string begin simplestmt ; simplestmt ; end $.

When specifying a parsing technique, we must state whether a leftmost or rightmost parse will be produced. The best-known and most widely used top-down and bottom-up parsing strategies are called LL and LR, respectively. These names seem rather arcane, but they reflect how the input is processed and which 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, Σ 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 λ is not represented explicitly as a symbol. Instead, a production $\mathsf{A} \rightarrow \lambda$ 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.

ISTERMINAL($\mathcal{X}$): Returns **true** if $\mathcal{X}$ 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.

PRODUCTIONSFOR(A): Returns an iterator that visits each production for nonterminal A.

```
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 λ.

OCCURRENCES($\mathcal{X}$): Returns an iterator that visits each occurrence of $\mathcal{X}$ in the RHS of all rules.

PRODUCTION(y): Returns a descriptor for the production $A \rightarrow \alpha$ where α 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 β.

4.5.2 Deriving the Empty String

One of the most common grammar computations is determining which nonterminals can derive λ. This information is important because such nonterminals may disappear during a parse and hence must be carefully handled. Determining if a nonterminal can derive λ is not entirely trivial because the derivation can take more than one step:

$$\mathsf{A} \Rightarrow \mathsf{BCD} \Rightarrow \mathsf{BC} \Rightarrow \mathsf{B} \Rightarrow \lambda.$$

An algorithm to compute the productions and symbols that can derive λ 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 λ. 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 $\mathsf{A} \rightarrow \lambda$ is 0. Once a production is known to derive λ, 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 λ, as follows.

- RuleDerivesEmpty(p) indicates whether production p can derive λ. When every symbol in rule p's RHS can derive λ, Step **5** establishes that p can derive λ.
- SymbolDerivesEmpty(A) indicates whether the nonterminal A can derive λ. When any production for A can derive λ, Step **6** establishes that A can derive λ.

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

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

Some texts include λ in $\mathsf{First}(\alpha)$ if $\alpha \Rightarrow^{\star} \lambda$. The resulting algorithms require frequent subtraction of λ from symbol sets. We adopt the convention of *never*

```
function FIRST(α) : Set
    foreach A ∈ NONTERMINALS( ) do VisitedFirst(A) ← false          8
    ans ← INTERNALFIRST(α)
    return (ans)
end
function INTERNALFIRST(Xβ) : Set
    if Xβ = ⊥                                                        9
    then return (∅)
    if X ∈ Σ                                                        10
    then return ({ X })
    /*                                                          */
    /*                    X is a nonterminal.                  */  11
    /*                                                          */
    ans ← ∅
    if not VisitedFirst(X)
    then
        VisitedFirst(X) ← true                                      12
        foreach rhs ∈ ProductionsFor(X) do
            ans ← ans ∪ INTERNALFIRST(rhs)                          13
    if SymbolDerivesEmpty(X)                                        14
    then ans ← ans ∪ INTERNALFIRST(β)
    return (ans)                                                    15
end
```

Figure 4.8: Algorithm for computing First(α).

including λ in First(α). Testing whether a given string of symbols α derives λ is easily accomplished—when the results from the algorithm of Figure 4.7 are available.

First(α) is computed by scanning α left-to-right. If α begins with a terminal symbol a, then clearly First(α) = { a }. If a nonterminal symbol A is encountered, then the grammar productions for A must be consulted. Nonterminals that can derive λ 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 λ-production, which contributes nothing. The solutions are as follows.

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

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→ Prefix (E) requires com-

putation of First(Prefix). Because Prefix $\Rightarrow^{\star} \lambda$, First((E)) must also be included. The resulting set is as follows.

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

Termination of First(A) must be handled properly in grammars where the computation of First(A) appears to depend on First(A), as follows.

$$\begin{array}{lcl} \text{A} & \rightarrow & \text{B} \\ & \cdots & \\ \text{B} & \rightarrow & \text{C} \\ & \cdots & \\ \text{C} & \rightarrow & \text{A} \end{array}$$

In this grammar, First(A) depends on First(B), which depends on First(C), which depends on First(A). In computing 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.

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

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

- The empty set is returned if $\mathcal{X}\beta$ 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 $\mathcal{X}$ is a terminal, then First($\mathcal{X}\beta$) is $\{ \mathcal{X} \}$ at Step **10**.
- The only remaining possibility is that $\mathcal{X}$ is a nonterminal. If *VisitedFirst*($\mathcal{X}$) is **false**, then the productions for $\mathcal{X}$ are recursively examined for inclusion. Otherwise, $\mathcal{X}$'s productions already participate in the current computation.
- If $\mathcal{X}$ can derive λ at Step **14**—this fact has been previously computed by the algorithm in Figure 4.7—then we must include all symbols in First(β).

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

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

Figure 4.9 shows the progress of 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 FIRST($\mathcal{X}\beta$) is shown with nonblank entries in the $\mathcal{X}$ and β columns. A "$\star$" 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 INTERNALFIRST on null strings are omitted.

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 ex-

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

Level	First $\mathcal{X}$	First β	*ans*	Step	Done?	Comment
				COMPUTEFIRST(B)		
0	B	$\perp$	{ }	Step **11**		
1	b	$\perp$	{ b }	Step **10**	$\star$	B$\rightarrow$ b
1	$\perp$	$\perp$	{ }	Step **9**	$\star$	B$\rightarrow \lambda$
0			{ b }	Step **14**	$\star$	Final answer
				COMPUTEFIRST(A)		
0	A	$\perp$	{ }	Step **11**		
1	a	$\perp$	{ a }	Step **10**	$\star$	A$\rightarrow$ a
1	$\perp$	$\perp$	{ }	Step **9**	$\star$	A$\rightarrow \lambda$
0			{ a }	Step **14**	$\star$	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**	$\star$	Because B $\Rightarrow^* \lambda$
2			{ b,c }	Step **14**	$\star$	
1			{ a,b,c }	Step **14**	$\star$	
0			{ a,b,c }	Step **14**	$\star$	

Figure 4.10: A grammar and its First sets.

```
function FOLLOW(A) : Set
    foreach A ∈ NONTERMINALS( ) do
        VisitedFollow(A) ← false                                   16
    ans ← INTERNALFOLLOW(A)
    return (ans)
end
function INTERNALFOLLOW(A) : Set
    ans ← ∅
    if not VisitedFolow(A)                                         17
    then
        VisitedFollow(A) ← true                                    18
        foreach a ∈ OCCURRENCES(A) do                              19
            ans ← ans ∪ FIRST(TAIL(a))                             20
            if ALLDERIVEEMPTY(TAIL(a))                             21
            then
                targ ← LHS(PRODUCTION(a))
                ans ← ans ∪ INTERNALFOLLOW(targ)                   22
    return (ans)                                                   23
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).

cept the goal symbol *must* be followed by some terminal. Formally, for $A \in N$,

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

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(α) algorithm given in Figure 4.8.

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

Level	Rule	Step	Result	Comment
		COMPUTEFOLLOW(Prefix)		
0		FOLLOW(Prefix)		
0	E → Prefix (E)	Step **20**	{ (}	
		COMPUTEFOLLOW(E)		
0		FOLLOW(E)		
0	E → Prefix (E)	Step **20**	{) }	
0	Tail → + E	Step **22**	{ }	
1		FOLLOW(Tail)		
1	E → v Tail	Step **22**	{ }	
2		FOLLOW(E)		
		Step **17**	{ }	Recursion avoided
1		Step **23**	{ }	Returns
0		Step **23**	{) }	Returns
		COMPUTEFOLLOW(Tail)		
0		FOLLOW(Tail)		
0	E → v Tail	Step **22**	{ }	
1		FOLLOW(E)		
1	E → Prefix (E)	Step **20**	{) }	
1	Tail → + E	Step **22**	{ }	
2		FOLLOW(Tail)		
		Step **17**	{ }	Recursion avoided
1		Step **23**	{) }	Returns
0		Step **23**	{) }	Returns

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

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 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.

Level	Rule	Step	Result	Comment
		COMPUTEFOLLOW(B)		
0			FOLLOW(B)	
0	S→A B c	Step **20**	{ c }	
0		Step **23**	{ c }	Returns
		COMPUTEFOLLOW(A)		
0			FOLLOW(A)	
0	S→ A B c	Step **20**	{ b,c }	
0		Step **23**	{ b,c }	Returns
		COMPUTEFOLLOW(S)		
0			FOLLOW(S)	
0		Step **23**	{ }	Returns

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.

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 α. Similarly, $\text{Follow}_k(\text{A})$ 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 $\text{First}_1(\alpha)$ and $\text{Follow}_1(\text{A})$ can be generalized to compute $\text{First}_k(\alpha)$ and $\text{Follow}_k(\text{A})$ sets (see Exercise 24).

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.

Exercises

1. Transform the following grammar into a standard CFG using the algorithm in Figure 4.4.

1	S	→	Number
2	Number	→	[Sign] [Digs period] Digs
3	Sign	→	plus
4		|	minus
5	Digs	→	digit { digit }

2. Design a language and context-free grammar to represent the following languages.
   (a) The set of strings of base-8 numbers.
   (b) The set of strings of base-16 numbers.
   (c) The set of strings of base-1 numbers.
   (d) A language that offers base-8, base-16, and base-1 numbers.

3. Describe the language denoted by each of the following grammars.
   (a) $(\{A, B, C\}, \{a, b, c\}, \emptyset, A)$
   (b) $(\{A, B, C\}, \{a, b, c\}, \{A \rightarrow B\ C\}, A)$
   (c) $(\{A, B, C\}, \{a, b, c\}, \{A \rightarrow A\ a, A \rightarrow b\}, A)$
   (d) $(\{A, B, C\}, \{a, b, c\}, \{A \rightarrow B\ B, B \rightarrow a, B \rightarrow b, B \rightarrow c\}, A)$

4. What are the difficulties associated with constructing a grammar whose generated strings are decimal representations of irrational numbers?

5. A grammar for infix expressions follows.

1	Start	→	E $
2	E	→	T plus E
3		|	T
4	T	→	T times F
5		|	F
6	F	→	(E)
7		|	num

   (a) Show the leftmost derivation of the following string.

   num plus num times num plus num $

   (b) Show the rightmost derivation of the following string.

   num times num plus num times num $

   (c) Describe how this grammar structures expressions, in terms of the precedence and left- or right- associativity of operators.

6. Consider the following two grammars.

   (a)

1	Start	→	E $
2	E	→	(E plus E
3		|	num

   (b)

1	Start	→	E $
2	E	→	E (plus E
3		|	num

   Which of these grammars, if any, is ambiguous? Prove your answer by showing two distinct derivations of some input string for the ambiguous grammar(s).

7. Compute First and Follow sets for the nonterminals of the following grammar.

1	S	→	a S e
2		|	B
3	B	→	b B e
4		|	C
5	C	→	c C e
6		|	d

8. Compute First and Follow sets for each nonterminal in ac grammar from Chapter Chapter:global:two, reprised as follows.

1	Prog	→	Dcls Stmts $
2	Dcls	→	Dcl Dcls
3		|	λ
4	Dcl	→	floatdcl id
5		|	intdcl id
6	Stmts	→	Stmt Stmts
7		|	λ
8	Stmt	→	id assign Val ExprTail
9		|	print id
10	ExprTail	→	plus Val ExprTail
11		|	minus Val ExprTail
12		|	λ
13	Val	→	id
14		|	num

9. Compute First and Follow sets for each nonterminal in Exercise 1.

10. As discussed in Section 4.3, the algorithm in Figure 4.4 could use left- or right-recursion to transform a repeated sequence of symbols into standard grammar form. A production of the form $\mathsf{A} \rightarrow \mathsf{A}\ \alpha$ is said to be **left recursive**. Similarly, a production of the form $\mathsf{A} \rightarrow \beta\ \mathsf{A}$ is said to be **right recursive**. Show that any grammar that contains left- and right-recursive rules for the same LHS nonterminal must be ambiguous.

11. Section 4.3 describes extended BNF notation for optional and repeated symbol sequences. Suppose the n grammar symbols $\mathcal{X}_1 \ldots \mathcal{X}_n$ represent a set of n options. What is the effect of the following grammar with regard to how the options can appear?

$$\begin{array}{lcl} \mathsf{Options} & \rightarrow & \mathsf{Options\ Option} \\ & | & \lambda \\ \mathsf{Option} & \rightarrow & \mathcal{X}_1 \\ & | & \mathcal{X}_2 \\ & & \cdots \\ & | & \mathcal{X}_n \end{array}$$

12. Consider n optional symbols $\mathcal{X}_1 \ldots \mathcal{X}_n$ as described in Exercise 11.

    (a) Devise a CFG that generates any subset of these options. That is, the symbols can occur in any order, any symbol can be missing, and no symbol is repeated.

    (b) What is the relation between the size of your grammar and n, the number of options?

    (c) How is your solution affected if symbols $\mathcal{X}_i$ and $\mathcal{X}_j$ are present only if $i < j$?

13. Show that regular grammars and finite automata have equivalent definitional power by developing

    (a) an algorithm that translates regular grammars into finite automata and

    (b) an algorithm that translates finite automata into regular grammars.

14. Devise an algorithm to detect nonterminals that cannot be reached from a CFG's goal symbol.

15. Devise an algorithm to detect nonterminals that cannot derive any terminal string in a CFG.

16. A CFG is reduced by removing useless terminals and productions. Consider the following two tasks.

    (a) Nonterminals not reachable from the grammar's goal symbol are removed.

    (b) Nonterminals that derive no terminal string are removed.

Does the order of the above tasks matter? If so, which order is preferred?

17. The algorithm presented in Figure 4.8 retains no information between invocations of FIRST. As a result, the solution for a given nonterminal might be computed multiple times.

    (a) Modify the algorithm so it remembers and references valid previous computations of First(A), $A \in N$

    (b) Frequently an algorithm needs First sets computed for *all* $\mathcal{X} \in N$. Devise an algorithm that efficiently computes First sets for all nonterminals in a grammar. Analyze the efficiency of your algorithm.

    Hint: Consider constructing a directed graph whose vertices represent nonterminals. Let an edge (A, B) represent that First(B) depends on First(A).

    (c) Repeat this exercise for the Follow sets.

18. Prove that COMPUTEFIRST(A) correctly computes First(A) for any $A \in N$.

19. Prove that COMPUTEFOLLOW(A) correctly computes Follow(A) for any $A \in N$.

20. Let G be any CFG and assume $\lambda \notin L(G)$. Show that G can be transformed into a language-equivalent CFG that uses no λ-productions.

21. A **unit production** is a rule of the form $A \rightarrow B$. Show that any CFG that contains unit productions can be transformed into a language-equivalent CFG that uses no unit productions.

22. Some CFGs denote a language with an infinite number of strings; others denote finite languages. Devise an algorithm that determines whether a given CFG generates an infinite language.

    Hint: Use the results of Exercises 20 and 21 to simplify the analysis.

23. Let G be an unambiguous CFG without λ-productions.

    (a) If $x \in L(G)$, show that the number of steps needed to derive x is linear in the length of x.

    (b) Does this linearity result hold if λ-productions are included?

    (c) Does this linearity result hold if G is ambiguous?

24. The algorithms in Figures 4.8 and 4.11 compute First(α) and Follow(A).

    (a) Modify the algorithm in Figure 4.8 to compute $\text{First}_k(\alpha)$.

    Hint: Consider formulating the algorithm so that when $\text{First}_i(\alpha)$ is computed, enough information is retained to compute $\text{First}_{i+1}(\alpha)$.

    (b) Modify the algorithm in Figure 4.11 to compute $\text{Follow}_k(A)$.

Bibliography

[HU79] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[Mar91] John C. Martin. *Introduction to Langauges and the Theory of Computation*. McGraw-Hill, 1991.