Chapter 3

Regular Expressions and Languages

We begin this chapter by introducing the notation called "regular expressions." These expressions are another type of language-defining notation, which we sampled briefly in Section 1.1.2. Regular expressions also may be thought of as a "programming language," in which we express some important applications, such as text-search applications or compiler components. Regular expressions are closely related to nondeterministic finite automata and can be thought of as a "user-friendly" alternative to the NFA notation for describing software components.

In this chapter, after defining regular expressions, we show that they are capable of defining all and only the regular languages. We discuss the way that regular expressions are used in several software systems. Then, we examine the algebraic laws that apply to regular expressions. They have significant resemblance to the algebraic laws of arithmetic, yet there are also some important differences between the algebras of regular expressions and arithmetic expressions.

3.1 Regular Expressions

Now, we switch our attention from machine-like descriptions of languages — deterministic and nondeterministic finite automata — to an algebraic description: the "regular expression." We shall find that regular expressions can define exactly the same languages that the various forms of automata describe: the regular languages. However, regular expressions offer something that automata do not: a declarative way to express the strings we want to accept. Thus, regular expressions serve as the input language for many systems that process strings. Examples include:

1. Search commands such as the UNIX `grep` or equivalent commands for finding strings that one sees in Web browsers or text-formatting systems. These systems use a regular-expression-like notation for describing patterns that the user wants to find in a file. Different search systems convert the regular expression into either a DFA or an NFA, and simulate that automaton on the file being searched.

2. Lexical-analyzer generators, such as Lex or Flex. Recall that a lexical analyzer is the component of a compiler that breaks the source program into logical units (called *tokens*) of one or more characters that have a shared significance. Examples of tokens include keywords (e.g., `while`), identifiers (e.g., any letter followed by zero or more letters and/or digits), and signs, such as `+` or `<=`. A lexical-analyzer generator accepts descriptions of the forms of tokens, which are essentially regular expressions, and produces a DFA that recognizes which token appears next on the input.

3.1.1 The Operators of Regular Expressions

Regular expressions denote languages. For a simple example, the regular expression $\mathbf{01}^* + \mathbf{10}^*$ denotes the language consisting of all strings that are either a single 0 followed by any number of 1's or a single 1 followed by any number of 0's. We do not expect you to know at this point how to interpret regular expressions, so our statement about the language of this expression must be accepted on faith for the moment. We shortly shall define all the symbols used in this expression, so you can see why our interpretation of this regular expression is the correct one. Before describing the regular-expression notation, we need to learn the three operations on languages that the operators of regular expressions represent. These operations are:

1. The *union* of two languages L and M, denoted $L \cup M$, is the set of strings that are in either L or M, or both. For example, if $L = \{001, 10, 111\}$ and $M = \{\epsilon, 001\}$, then $L \cup M = \{\epsilon, 10, 001, 111\}$.

2. The *concatenation* of languages L and M is the set of strings that can be formed by taking any string in L and concatenating it with any string in M. Recall Section 1.5.2, where we defined the concatenation of a pair of strings; one string is followed by the other to form the result of the concatenation. We denote concatenation of languages either with a dot or with no operator at all, although the concatenation operator is frequently called "dot." For example, if $L = \{001, 10, 111\}$ and $M = \{\epsilon, 001\}$, then $L.M$, or just LM, is $\{001, 10, 111, 001001, 10001, 111001\}$. The first three strings in LM are the strings in L concatenated with ϵ. Since ϵ is the identity for concatenation, the resulting strings are the same as the strings of L. However, the last three strings in LM are formed by taking each string in L and concatenating it with the second string in M, which is 001. For instance, 10 from L concatenated with 001 from M gives us 10001 for LM.

3. The *closure* (or *star*, or *Kleene closure*)[1] of a language L is denoted L^* and represents the set of those strings that can be formed by taking any number of strings from L, possibly with repetitions (i.e., the same string may be selected more than once) and concatenating all of them. For instance, if $L = \{0, 1\}$, then L^* is all strings of 0's and 1's. If $L = \{0, 11\}$, then L^* consists of those strings of 0's and 1's such that the 1's come in pairs, e.g., 011, 11110, and ϵ, but not 01011 or 101. More formally, L^* is the infinite union $\cup_{i \geq 0} L^i$, where $L^0 = \{\epsilon\}$, $L^1 = L$, and L^i, for $i > 1$ is $LL \cdots L$ (the concatenation of i copies of L).

Example 3.1 : Since the idea of the closure of a language is somewhat tricky, let us study a few examples. First, let $L = \{0, 11\}$. $L^0 = \{\epsilon\}$, independent of what language L is; the 0th power represents the selection of zero strings from L. $L^1 = L$, which represents the choice of one string from L. Thus, the first two terms in the expansion of L^* give us $\{\epsilon, 0, 11\}$.

Next, consider L^2. We pick two strings from L, with repetitions allowed, so there are four choices. These four selections give us $L^2 = \{00, 011, 110, 1111\}$. Similarly, L^3 is the set of strings that may be formed by making three choices of the two strings in L and gives us

$$\{000, 0011, 0110, 1100, 01111, 11011, 11110, 111111\}$$

To compute L^*, we must compute L^i for each i, and take the union of all these languages. L^i has 2^i members. Although each L^i is finite, the union of the infinite number of terms L^i is generally an infinite language, as it is in our example.

Now, let L be the set of all strings of 0's. Note that L is infinite, unlike our previous example, which is a finite language. However, it is not hard to discover what L^* is. $L^0 = \{\epsilon\}$, as always. $L^1 = L$. L^2 is the set of strings that can be formed by taking one string of 0's and concatenating it with another string of 0's. The result is still a string of 0's. In fact, every string of 0's can be written as the concatenation of two strings of 0's (don't forget that ϵ is a "string of 0's"; this string can always be one of the two strings that we concatenate). Thus, $L^2 = L$. Likewise, $L^3 = L$, and so on. Thus, the infinite union $L^* = L^0 \cup L^1 \cup L^2 \cup \cdots$ is L in the particular case that the language L is the set of all strings of 0's.

For a final example, $\emptyset^* = \{\epsilon\}$. Note that $\emptyset^0 = \{\epsilon\}$, while $\emptyset^i$, for any $i \geq 1$, is empty, since we can't select any strings from the empty set. In fact, $\emptyset$ is one of only two languages whose closure is *not* infinite. □

3.1.2 Building Regular Expressions

Algebras of all kinds start with some elementary expressions, usually constants and/or variables. Algebras then allow us to construct more expressions by

[1] The term "Kleene closure" refers to S. C. Kleene, who originated the regular expression notation and this operator.

Use of the Star Operator

We saw the star operator first in Section 1.5.2, where we applied it to an alphabet, e.g., Σ^*. That operator formed all strings whose symbols were chosen from alphabet Σ. The closure operator is essentially the same, although there is a subtle distinction of types.

Suppose L is the language containing strings of length 1, and for each symbol a in Σ there is a string a in L. Then, although L and Σ "look" the same, they are of different types; L is a set of strings, and Σ is a set of symbols. On the other hand, L^* denotes the same language as Σ^*.

applying a certain set of operators to these elementary expressions and to previously constructed expressions. Usually, some method of grouping operators with their operands, such as parentheses, is required as well. For instance, the familiar arithmetic algebra starts with constants such as integers and real numbers, plus variables, and builds more complex expressions with arithmetic operators such as $+$ and $\times$.

The algebra of regular expressions follows this pattern, using constants and variables that denote languages, and operators for the three operations of Section 3.1.1 —union, dot, and star. We can describe the regular expressions recursively, as follows. In this definition, we not only describe what the legal regular expressions are, but for each regular expression E, we describe the language it represents, which we denote $L(E)$.

BASIS: The basis consists of three parts:

1. The constants ϵ and $\emptyset$ are regular expressions, denoting the languages $\{\epsilon\}$ and $\emptyset$, respectively. That is, $L(\epsilon) = \{\epsilon\}$, and $L(\emptyset) = \emptyset$.

2. If a is any symbol, then $\mathbf{a}$ is a regular expression. This expression denotes the language $\{a\}$. That is, $L(\mathbf{a}) = \{a\}$. Note that we use boldface font to denote an expression corresponding to a symbol. The correspondence, e.g. that $\mathbf{a}$ refers to a, should be obvious.

3. A variable, usually capitalized and italic such as L, is a variable, representing any language.

INDUCTION: There are four parts to the inductive step, one for each of the three operators and one for the introduction of parentheses.

1. If E and F are regular expressions, then $E + F$ is a regular expression denoting the union of $L(E)$ and $L(F)$. That is, $L(E+F) = L(E) \cup L(F)$.

2. If E and F are regular expressions, then EF is a regular expression denoting the concatenation of $L(E)$ and $L(F)$. That is, $L(EF) = L(E)L(F)$.

Expressions and Their Languages

Strictly speaking, a regular expression E is just an expression, not a language. We should use $L(E)$ when we want to refer to the language that E denotes. However, it is common usage to refer to say "E" when we really mean "$L(E)$." We shall use this convention as long as it is clear we are talking about a language and not about a regular expression.

Note that the dot can optionally be used to denote the concatenation operator, either as an operation on languages or as the operator in a regular expression. For instance, $\mathbf{0.1}$ is a regular expression meaning the same as $\mathbf{01}$ and representing the language $\{01\}$. However, we shall avoid the dot as concatenation in regular expressions.[2]

3. If E is a regular expression, then E^* is a regular expression, denoting the closure of $L(E)$. That is, $L(E^*) = \big(L(E)\big)^*$.

4. If E is a regular expression, then (E), a parenthesized E, is also a regular expression, denoting the same language as E. Formally; $L\big((E)\big) = L(E)$.

Example 3.2: Let us write a regular expression for the set of strings that consist of alternating 0's and 1's. First, let us develop a regular expression for the language consisting of the single string 01. We can then use the star operator to get an expression for all strings of the form $0101\cdots01$.

The basis rule for regular expressions tells us that $\mathbf{0}$ and $\mathbf{1}$ are expressions denoting the languages $\{0\}$ and $\{1\}$, respectively. If we concatenate the two expressions, we get a regular expression for the language $\{01\}$; this expression is $\mathbf{01}$. As a general rule, if we want a regular expression for the language consisting of only the string w, we use w itself as the regular expression. Note that in the regular expression, the symbols of w will normally be written in boldface, but the change of font is only to help you distinguish expressions from strings and should not be taken as significant.

Now, to get all strings consisting of zero or more occurrences of 01, we use the regular expression $(\mathbf{01})^*$. Note that we first put parentheses around $\mathbf{01}$, to avoid confusing with the expression $\mathbf{01}^*$, whose language is all strings consisting of a 0 and any number of 1's. The reason for this interpretation is explained in Section 3.1.3, but briefly, star takes precedence over dot, and therefore the argument of the star is selected before performing any concatenations.

However, $L\big((\mathbf{01})^*\big)$ is not exactly the language that we want. It includes only those strings of alternating 0's and 1's that begin with 0 and end with 1. We also need to consider the possibility that there is a 1 at the beginning and/or

[2] In fact, UNIX regular expressions use the dot for an entirely different purpose: representing any ASCII character.

a 0 at the end. One approach is to construct three more regular expressions that handle the other three possibilities. That is, $(\mathbf{10})^*$ represents those alternating strings that begin with 1 and end with 0, while $\mathbf{0}(\mathbf{10})^*$ can be used for strings that both begin and end with 0 and $\mathbf{1}(\mathbf{01})^*$ serves for strings that begin and end with 1. The entire regular expression is

$$(\mathbf{01})^* + (\mathbf{10})^* + \mathbf{0}(\mathbf{10})^* + \mathbf{1}(\mathbf{01})^*$$

Notice that we use the + operator to take the union of the four languages that together give us all the strings with alternating 0's and 1's.

However, there is another approach that yields a regular expression that looks rather different and is also somewhat more succinct. Start again with the expression $(\mathbf{01})^*$. We can add an optional 1 at the beginning if we concatenate on the left with the expression $\epsilon + \mathbf{1}$. Likewise, we add an optional 0 at the end with the expression $\epsilon + \mathbf{0}$. For instance, using the definition of the + operator:

$$L(\epsilon + \mathbf{1}) = L(\epsilon) \cup L(\mathbf{1}) = \{\epsilon\} \cup \{1\} = \{\epsilon, 1\}$$

If we concatenate this language with any other language L, the ϵ choice gives us all the strings in L, while the 1 choice gives us $1w$ for every string w in L. Thus, another expression for the set of strings that alternate 0's and 1's is:

$$(\epsilon + \mathbf{1})(\mathbf{01})^*(\epsilon + \mathbf{0})$$

Note that we need parentheses around each of the added expressions, to make sure the operators group properly. □

3.1.3 Precedence of Regular-Expression Operators

Like other algebras, the regular-expression operators have an assumed order of "precedence," which means that operators are associated with their operands in a particular order. We are familiar with the notion of precedence from ordinary arithmetic expressions. For instance, we know that $xy+z$ groups the product xy before the sum, so it is equivalent to the parenthesized expression $(xy) + z$ and not to the expression $x(y + z)$. Similarly, we group two of the same operators from the left in arithmetic, so $x - y - z$ is equivalent to $(x - y) - z$, and not to $x - (y - z)$. For regular expressions, the following is the order of precedence for the operators:

1. The star operator is of highest precedence. That is, it applies only to the smallest sequence of symbols to its left that is a well-formed regular expression.

2. Next in precedence comes the concatenation or "dot" operator. After grouping all stars to their operands, we group concatenation operators to their operands. That is, all expressions that are *juxtaposed* (adjacent, with no intervening operator) are grouped together. Since concatenation

is an associative operator it does not matter in what order we group consecutive concatenations, although if there is a choice to be made, you should group them from the left. For instance, **012** is grouped (**01**)**2**.

3. Finally, all unions (+ operators) are grouped with their operands. Since union is also associative, it again matters little in which order consecutive unions are grouped, but we shall assume grouping from the left.

Of course, sometimes we do not want the grouping in a regular expression to be as required by the precedence of the operators. If so, we are free to use parentheses to group operands exactly as we choose. In addition, there is never anything wrong with putting parentheses around operands that you want to group, even if the desired grouping is implied by the rules of precedence.

Example 3.3: The expression $\mathbf{01}^* + \mathbf{1}$ is grouped $\big(\mathbf{0}(\mathbf{1}^*)\big) + \mathbf{1}$. The star operator is grouped first. Since the symbol **1** immediately to its left is a legal regular expression, that alone is the operand of the star. Next, we group the concatenation between **0** and $(\mathbf{1}^*)$, giving us the expression $\big(\mathbf{0}(\mathbf{1}^*)\big)$. Finally, the union operator connects the latter expression and the expression to its right, which is **1**.

Notice that the language of the given expression, grouped according to the precedence rules, is the string 1 plus all strings consisting of a 0 followed by any number of 1's (including none). Had we chosen to group the dot before the star, we could have used parentheses, as $(\mathbf{01})^* + \mathbf{1}$. The language of this expression is the string 1 and all strings that repeat 01, zero or more times. Had we wished to group the union first, we could have added parentheses around the union to make the expression $\mathbf{0}(\mathbf{1}^* + \mathbf{1})$. That expression's language is the set of strings that begin with 0 and have any number of 1's following. □

3.1.4 Exercises for Section 3.1

Exercise 3.1.1: Write regular expressions for the following languages:

***** a) The set of strings over alphabet $\{a, b, c\}$ containing at least one a and at least one b.

b) The set of strings of 0's and 1's whose tenth symbol from the right end is 1.

c) The set of strings of 0's and 1's with at most one pair of consecutive 1's.

! Exercise 3.1.2: Write regular expressions for the following languages:

***** a) The set of all strings of 0's and 1's such that every pair of adjacent 0's appears before any pair of adjacent 1's.

b) The set of strings of 0's and 1's whose number of 0's is divisible by five.

!! **Exercise 3.1.3:** Write regular expressions for the following languages:

a) The set of all strings of 0's and 1's not containing 101 as a substring.

b) The set of all strings with an equal number of 0's and 1's, such that no prefix has two more 0's than 1's, nor two more 1's than 0's.

c) The set of strings of 0's and 1's whose number of 0's is divisible by five and whose number of 1's is even.

! **Exercise 3.1.4:** Give English descriptions of the languages of the following regular expressions:

* a) $(\mathbf{1} + \epsilon)(\mathbf{00}^*\mathbf{1})^*\mathbf{0}^*$.

b) $(\mathbf{0}^*\mathbf{1}^*)^*\mathbf{000}(\mathbf{0} + \mathbf{1})^*$.

c) $(\mathbf{0} + \mathbf{10})^*\mathbf{1}^*$.

*! **Exercise 3.1.5:** In Example 3.1 we pointed out that $\emptyset$ is one of two languages whose closure is finite. What is the other?

Chapter 5

Context-Free Grammars and Languages

We now turn our attention away from the regular languages to a larger class of languages, called the "context-free languages." These languages have a natural, recursive notation, called "context-free grammars." Context-free grammars have played a central role in compiler technology since the 1960's; they turned the implementation of parsers (functions that discover the structure of a program) from a time-consuming, ad-hoc implementation task into a routine job that can be done in an afternoon. More recently, the context-free grammar has been used to describe document formats, via the so-called document-type definition (DTD) that is used in the XML (extensible markup language) community for information exchange on the Web.

In this chapter, we introduce the context-free grammar notation, and show how grammars define languages. We discuss the "parse tree," a picture of the structure that a grammar places on the strings of its language. The parse tree is the product of a parser for a programming language and is the way that the structure of programs is normally captured.

There is an automaton-like notation, called the "pushdown automaton," that also describes all and only the context-free languages; we introduce the pushdown automaton in Chapter 6. While less important than finite automata, we shall find the pushdown automaton, especially its equivalence to context-free grammars as a language-defining mechanism, to be quite useful when we explore the closure and decision properties of the context-free languages in Chapter 7.

5.1 Context-Free Grammars

We shall begin by introducing the context-free grammar notation informally. After seeing some of the important capabilities of these grammars, we offer formal definitions. We show how to define a grammar formally, and introduce

the process of "derivation," whereby it is determined which strings are in the language of the grammar.

5.1.1 An Informal Example

Let us consider the language of palindromes. A *palindrome* is a string that reads the same forward and backward, such as `otto` or `madamimadam` ("Madam, I'm Adam," allegedly the first thing Eve heard in the Garden of Eden). Put another way, string w is a palindrome if and only if $w = w^R$. To make things simple, we shall consider describing only the palindromes with alphabet $\{0, 1\}$. This language includes strings like 0110, 11011, and ϵ, but not 011 or 0101.

It is easy to verify that the language L_{pal} of palindromes of 0's and 1's is not a regular language. To do so, we use the pumping lemma. If L_{pal} is a regular language, let n be the associated constant, and consider the palindrome $w = 0^n10^n$. If L_{pal} is regular, then we can break w into $w = xyz$, such that y consists of one or more 0's from the first group. Thus, xz, which would also have to be in L_{pal} if L_{pal} were regular, would have fewer 0's to the left of the lone 1 than there are to the right of the 1. Therefore xz cannot be a palindrome. We have now contradicted the assumption that L_{pal} is a regular language.

There is a natural, recursive definition of when a string of 0's and 1's is in L_{pal}. It starts with a basis saying that a few obvious strings are in L_{pal}, and then exploits the idea that if a string is a palindrome, it must begin and end with the same symbol. Further, when the first and last symbols are removed, the resulting string must also be a palindrome. That is:

BASIS: ϵ, 0, and 1 are palindromes.

INDUCTION: If w is a palindrome, so are $0w0$ and $1w1$. No string is a palindrome of 0's and 1's, unless it follows from this basis and induction rule.

A context-free grammar is a formal notation for expressing such recursive definitions of languages. A grammar consists of one or more variables that represent classes of strings, i.e., languages. In this example we have need for only one variable P, which represents the set of palindromes; that is the class of strings forming the language L_{pal}. There are rules that say how the strings in each class are constructed. The construction can use symbols of the alphabet, strings that are already known to be in one of the classes, or both.

Example 5.1 : The rules that define the palindromes, expressed in the context-free grammar notation, are shown in Fig. 5.1. We shall see in Section 5.1.2 what the rules mean.

The first three rules form the basis. They tell us that the class of palindromes includes the strings ϵ, 0, and 1. None of the right sides of these rules (the portions following the arrows) contains a variable, which is why they form a basis for the definition.

The last two rules form the inductive part of the definition. For instance, rule 4 says that if we take any string w from the class P, then $0w0$ is also in class P. Rule 5 likewise tells us that $1w1$ is also in P. □

1. $P \rightarrow \epsilon$
2. $P \rightarrow 0$
3. $P \rightarrow 1$
4. $P \rightarrow 0P0$
5. $P \rightarrow 1P1$

Figure 5.1: A context-free grammar for palindromes

5.1.2 Definition of Context-Free Grammars

There are four important components in a grammatical description of a language:

1. There is a finite set of symbols that form the strings of the language being defined. This set was $\{0, 1\}$ in the palindrome example we just saw. We call this alphabet the *terminals*, or *terminal symbols*.

2. There is a finite set of *variables*, also called sometimes *nonterminals* or *syntactic categories*. Each variable represents a language; i.e., a set of strings. In our example above, there was only one variable, P, which we used to represent the class of palindromes over alphabet $\{0, 1\}$.

3. One of the variables represents the language being defined; it is called the *start symbol*. Other variables represent auxiliary classes of strings that are used to help define the language of the start symbol. In our example, P, the only variable, is the start symbol.

4. There is a finite set of *productions* or *rules* that represent the recursive definition of a language. Each production consists of:

   (a) A variable that is being (partially) defined by the production. This variable is often called the *head* of the production.

   (b) The production symbol $\rightarrow$.

   (c) A string of zero or more terminals and variables. This string, called the *body* of the production, represents one way to form strings in the language of the variable of the head. In so doing, we leave terminals unchanged and substitute for each variable of the body any string that is known to be in the language of that variable.

   We saw an example of productions in Fig. 5.1.

The four components just described form a *context-free grammar*, or just *grammar*, or *CFG*. We shall represent a CFG G by its four components, that is, $G = (V, T, P, S)$, where V is the set of variables, T the terminals, P the set of productions, and S the start symbol.

Example 5.2: The grammar G_{pal} for the palindromes is represented by

$$G_{pal} = (\{P\}, \{0,1\}, A, P)$$

where A represents the set of five productions that we saw in Fig. 5.1. □

Example 5.3: Let us explore a more complex CFG that represents (a simplification of) expressions in a typical programming language. First, we shall limit ourselves to the operators + and $*$, representing addition and multiplication. We shall allow arguments to be identifiers, but instead of allowing the full set of typical identifiers (letters followed by zero or more letters and digits), we shall allow only the letters a and b and the digits 0 and 1. Every identifier must begin with a or b, which may be followed by any string in $\{a, b, 0, 1\}^*$.

We need two variables in this grammar. One, which we call E, represents expressions. It is the start symbol and represents the language of expressions we are defining. The other variable, I, represents identifiers. Its language is actually regular; it is the language of the regular expression

$$(\mathbf{a} + \mathbf{b})(\mathbf{a} + \mathbf{b} + \mathbf{0} + \mathbf{1})^*$$

However, we shall not use regular expressions directly in grammars. Rather, we use a set of productions that say essentially the same thing as this regular expression.

1.	E	$\rightarrow$	I
2.	E	$\rightarrow$	$E + E$
3.	E	$\rightarrow$	$E * E$
4.	E	$\rightarrow$	(E)
5.	I	$\rightarrow$	a
6.	I	$\rightarrow$	b
7.	I	$\rightarrow$	Ia
8.	I	$\rightarrow$	Ib
9.	I	$\rightarrow$	$I0$
10.	I	$\rightarrow$	$I1$

Figure 5.2: A context-free grammar for simple expressions

The grammar for expressions is stated formally as $G = (\{E, I\}, T, P, E)$, where T is the set of symbols $\{+, *, (,), a, b, 0, 1\}$ and P is the set of productions shown in Fig. 5.2. We interpret the productions as follows.

Rule (1) is the basis rule for expressions. It says that an expression can be a single identifier. Rules (2) through (4) describe the inductive case for expressions. Rule (2) says that an expression can be two expressions connected by a plus sign; rule (3) says the same with a multiplication sign. Rule (4) says

Compact Notation for Productions

It is convenient to think of a production as "belonging" to the variable of its head. We shall often use remarks like "the productions for A" or "A-productions" to refer to the productions whose head is variable A. We may write the productions for a grammar by listing each variable once, and then listing all the bodies of the productions for that variable, separated by vertical bars. That is, the productions $A \to \alpha_1,\ A \to \alpha_2, \ldots, A \to \alpha_n$ can be replaced by the notation $A \to \alpha_1 | \alpha_2 | \cdots | \alpha_n$. For instance, the grammar for palindromes from Fig. 5.1 can be written as $P \to \epsilon \mid 0 \mid 1 \mid 0P0 \mid 1P1$.

that if we take any expression and put matching parentheses around it, the result is also an expression.

Rules (5) through (10) describe identifiers I. The basis is rules (5) and (6); they say that a and b are identifiers. The remaining four rules are the inductive case. They say that if we have any identifier, we can follow it by a, b, 0, or 1, and the result will be another identifier. □

5.1.3 Derivations Using a Grammar

We apply the productions of a CFG to infer that certain strings are in the language of a certain variable. There are two approaches to this inference. The more conventional approach is to use the rules from body to head. That is, we take strings known to be in the language of each of the variables of the body, concatenate them, in the proper order, with any terminals appearing in the body, and infer that the resulting string is in the language of the variable in the head. We shall refer to this procedure as *recursive inference*.

There is another approach to defining the language of a grammar, in which we use the productions from head to body. We expand the start symbol using one of its productions (i.e., using a production whose head is the start symbol). We further expand the resulting string by replacing one of the variables by the body of one of its productions, and so on, until we derive a string consisting entirely of terminals. The language of the grammar is all strings of terminals that we can obtain in this way. This use of grammars is called *derivation*.

We shall begin with an example of the first approach — recursive inference. However, it is often more natural to think of grammars as used in derivations, and we shall next develop the notation for describing these derivations.

Example 5.4: Let us consider some of the inferences we can make using the grammar for expressions in Fig. 5.2. Figure 5.3 summarizes these inferences. For example, line (i) says that we can infer string a is in the language for I by using production 5. Lines (ii) through (iv) say we can infer that $b00$

is an identifier by using production 6 once (to get the b) and then applying production 9 twice (to attach the two 0's).

	String Inferred	For language of	Production used	String(s) used
(i)	a	I	5	—
(ii)	b	I	6	—
(iii)	$b0$	I	9	(ii)
(iv)	$b00$	I	9	(iii)
(v)	a	E	1	(i)
(vi)	$b00$	E	1	(iv)
(vii)	$a + b00$	E	2	$(v), (vi)$
$(viii)$	$(a + b00)$	E	4	(vii)
(ix)	$a * (a + b00)$	E	3	$(v), (viii)$

Figure 5.3: Inferring strings using the grammar of Fig. 5.2

Lines (v) and (vi) exploit production 1 to infer that, since any identifier is an expression, the strings a and $b00$, which we inferred in lines (i) and (iv) to be identifiers, are also in the language of variable E. Line (vii) uses production 2 to infer that the sum of these identifiers is an expression; line $(viii)$ uses production 4 to infer that the same string with parentheses around it is also an expression, and line (ix) uses production 3 to multiply the identifier a by the expression we had discovered in line $(viii)$. □

The process of deriving strings by applying productions from head to body requires the definition of a new relation symbol $\Rightarrow$. Suppose $G = (V, T, P, S)$ is a CFG. Let $\alpha A \beta$ be a string of terminals and variables, with A a variable. That is, α and β are strings in $(V \cup T)^*$, and A is in V. Let $A \to \gamma$ be a production of G. Then we say $\alpha A \beta \underset{G}{\Rightarrow} \alpha \gamma \beta$. If G is understood, we just say $\alpha A \beta \Rightarrow \alpha \gamma \beta$. Notice that one derivation step replaces any variable anywhere in the string by the body of one of its productions.

We may extend the $\Rightarrow$ relationship to represent zero, one, or many derivation steps, much as the transition function δ of a finite automaton was extended to $\hat{\delta}$. For derivations, we use a * to denote "zero or more steps," as follows:

BASIS: For any string α of terminals and variables, we say $\alpha \overset{*}{\underset{G}{\Rightarrow}} \alpha$. That is, any string derives itself.

INDUCTION: If $\alpha \overset{*}{\underset{G}{\Rightarrow}} \beta$ and $\beta \underset{G}{\Rightarrow} \gamma$, then $\alpha \overset{*}{\underset{G}{\Rightarrow}} \gamma$. That is, if α can become β by zero or more steps, and one more step takes β to γ, then α can become γ. Put another way, $\alpha \overset{*}{\underset{G}{\Rightarrow}} \beta$ means that there is a sequence of strings $\gamma_1, \gamma_2, \ldots, \gamma_n$, for some $n \geq 1$, such that

1. $\alpha = \gamma_1$,

2. $\beta = \gamma_n$, and

3. For $i = 1, 2, \ldots, n-1$, we have $\gamma_i \Rightarrow \gamma_{i+1}$.

If grammar G is understood, then we use $\stackrel{*}{\Rightarrow}$ in place of $\stackrel{*}{\underset{G}{\Rightarrow}}$.

Example 5.5: The inference that $a*(a+b00)$ is in the language of variable E can be reflected in a derivation of that string, starting with the string E. Here is one such derivation:

$$E \Rightarrow E*E \Rightarrow I*E \Rightarrow a*E \Rightarrow$$

$$a*(E) \Rightarrow a*(E+E) \Rightarrow a*(I+E) \Rightarrow a*(a+E) \Rightarrow$$

$$a*(a+I) \Rightarrow a*(a+I0) \Rightarrow a*(a+I00) \Rightarrow a*(a+b00)$$

At the first step, E is replaced by the body of production 3 (from Fig. 5.2). At the second step, production 1 is used to replace the first E by I, and so on. Notice that we have systematically adopted the policy of always replacing the leftmost variable in the string. However, at each step we may choose which variable to replace, and we can use any of the productions for that variable. For instance, at the second step, we could have replaced the second E by (E), using production 4. In that case, we would say $E*E \Rightarrow E*(E)$. We could also have chosen to make a replacement that would fail to lead to the same string of terminals. A simple example would be if we used production 2 at the first step, and said $E \Rightarrow E+E$. No replacements for the two E's could ever turn $E+E$ into $a*(a+b00)$.

We can use the $\stackrel{*}{\Rightarrow}$ relationship to condense the derivation. We know $E \stackrel{*}{\Rightarrow} E$ by the basis. Repeated use of the inductive part gives us $E \stackrel{*}{\Rightarrow} E*E$, $E \stackrel{*}{\Rightarrow} I*E$, and so on, until finally $E \stackrel{*}{\Rightarrow} a*(a+b00)$.

The two viewpoints — recursive inference and derivation — are equivalent. That is, a string of terminals w is inferred to be in the language of some variable A if and only if $A \stackrel{*}{\Rightarrow} w$. However, the proof of this fact requires some work, and we leave it to Section 5.2. □

5.1.4 Leftmost and Rightmost Derivations

In order to restrict the number of choices we have in deriving a string, it is often useful to require that at each step we replace the leftmost variable by one of its production bodies. Such a derivation is called a *leftmost derivation*, and we indicate that a derivation is leftmost by using the relations $\underset{lm}{\Rightarrow}$ and $\stackrel{*}{\underset{lm}{\Rightarrow}}$, for one or many steps, respectively. If the grammar G that is being used is not obvious, we can place the name G below the arrow in either of these symbols.

Similarly, it is possible to require that at each step the rightmost variable is replaced by one of its bodies. If so, we call the derivation *rightmost* and use

Notation for CFG Derivations

There are a number of conventions in common use that help us remember the role of the symbols we use when discussing CFG's. Here are the conventions we shall use:

1. Lower-case letters near the beginning of the alphabet, a, b, and so on, are terminal symbols. We shall also assume that digits and other characters such as + or parentheses are terminals.
2. Upper-case letters near the beginning of the alphabet, A, B, and so on, are variables.
3. Lower-case letters near the end of the alphabet, such as w or z, are strings of terminals. This convention reminds us that the terminals are analogous to the input symbols of an automaton.
4. Upper-case letters near the end of the alphabet, such as X or Y, are either terminals or variables.
5. Lower-case Greek letters, such as α and β, are strings consisting of terminals and/or variables.

There is no special notation for strings that consist of variables only, since this concept plays no important role. However, a string named α or another Greek letter might happen to have only variables.

the symbols $\underset{rm}{\Rightarrow}$ and $\overset{*}{\underset{rm}{\Rightarrow}}$ to indicate one or many rightmost derivation steps, respectively. Again, the name of the grammar may appear below these symbols if it is not clear which grammar is being used.

Example 5.6: The derivation of Example 5.5 was actually a leftmost derivation. Thus, we can describe the same derivation by:

$$E \underset{lm}{\Rightarrow} E*E \underset{lm}{\Rightarrow} I*E \underset{lm}{\Rightarrow} a*E \underset{lm}{\Rightarrow}$$

$$a*(E) \underset{lm}{\Rightarrow} a*(E+E) \underset{lm}{\Rightarrow} a*(I+E) \underset{lm}{\Rightarrow} a*(a+E) \underset{lm}{\Rightarrow}$$

$$a*(a+I) \underset{lm}{\Rightarrow} a*(a+I0) \underset{lm}{\Rightarrow} a*(a+I00) \underset{lm}{\Rightarrow} a*(a+b00)$$

We can also summarize the leftmost derivation by saying $E \overset{*}{\underset{lm}{\Rightarrow}} a*(a+b00)$, or express several steps of the derivation by expressions such as $E*E \overset{*}{\underset{lm}{\Rightarrow}} a*(E)$.

There is a rightmost derivation that uses the same replacements for each variable, although it makes the replacements in different order. This rightmost derivation is:

$$E \underset{rm}{\Rightarrow} E*E \underset{rm}{\Rightarrow} E*(E) \underset{rm}{\Rightarrow} E*(E+E) \underset{rm}{\Rightarrow}$$

$$E*(E+I) \underset{rm}{\Rightarrow} E*(E+I0) \underset{rm}{\Rightarrow} E*(E+I00) \underset{rm}{\Rightarrow} E*(E+b00) \underset{rm}{\Rightarrow}$$

$$E*(I+b00) \underset{rm}{\Rightarrow} E*(a+b00) \underset{rm}{\Rightarrow} I*(a+b00) \underset{rm}{\Rightarrow} a*(a+b00)$$

This derivation allows us to conclude $E \overset{*}{\underset{rm}{\Rightarrow}} a*(a+b00)$. □

Any derivation has an equivalent leftmost and an equivalent rightmost derivation. That is, if w is a terminal string, and A a variable, then $A \overset{*}{\Rightarrow} w$ if and only if $A \overset{*}{\underset{lm}{\Rightarrow}} w$, and $A \overset{*}{\Rightarrow} w$ if and only if $A \overset{*}{\underset{rm}{\Rightarrow}} w$. We shall also prove these claims in Section 5.2.

5.1.5 The Language of a Grammar

If $G = (V, T, P, S)$ is a CFG, the *language* of G, denoted $L(G)$, is the set of terminal strings that have derivations from the start symbol. That is,

$$L(G) = \{w \text{ in } T^* \mid S \overset{*}{\underset{G}{\Rightarrow}} w\}$$

If a language L is the language of some context-free grammar, then L is said to be a *context-free language*, or CFL. For instance, we asserted that the grammar of Fig. 5.1 defined the language of palindromes over alphabet $\{0, 1\}$. Thus, the set of palindromes is a context-free language. We can prove that statement, as follows.

Theorem 5.7: $L(G_{pal})$, where G_{pal} is the grammar of Example 5.1, is the set of palindromes over $\{0, 1\}$.

PROOF: We shall prove that a string w in $\{0, 1\}^*$ is in $L(G_{pal})$ if and only if it is a palindrome; i.e., $w = w^R$.

(If) Suppose w is a palindrome. We show by induction on $|w|$ that w is in $L(G_{pal})$.

BASIS: We use lengths 0 and 1 as the basis. If $|w| = 0$ or $|w| = 1$, then w is ϵ, 0, or 1. Since there are productions $P \to \epsilon$, $P \to 0$, and $P \to 1$, we conclude that $P \overset{*}{\Rightarrow} w$ in any of these basis cases.

INDUCTION: Suppose $|w| \geq 2$. Since $w = w^R$, w must begin and end with the same symbol That is, $w = 0x0$ or $w = 1x1$. Moreover, x must be a palindrome; that is, $x = x^R$. Note that we need the fact that $|w| \geq 2$ to infer that there are two distinct 0's or 1's, at either end of w.

If $w = 0x0$, then we invoke the inductive hypothesis to claim that $P \stackrel{*}{\Rightarrow} x$. Then there is a derivation of w from P, namely $P \Rightarrow 0P0 \stackrel{*}{\Rightarrow} 0x0 = w$. If $w = 1x1$, the argument is the same, but we use the production $P \to 1P1$ at the first step. In either case, we conclude that w is in $L(G_{pal})$ and complete the proof.

(Only-if) Now, we assume that w is in $L(G_{pal})$; that is, $P \stackrel{*}{\Rightarrow} w$. We must conclude that w is a palindrome. The proof is an induction on the number of steps in a derivation of w from P.

BASIS: If the derivation is one step, then it must use one of the three productions that do not have P in the body. That is, the derivation is $P \Rightarrow \epsilon$, $P \Rightarrow 0$, or $P \Rightarrow 1$. Since ϵ, 0, and 1 are all palindromes, the basis is proven.

INDUCTION: Now, suppose that the derivation takes $n+1$ steps, where $n \geq 1$, and the statement is true for all derivations of n steps. That is, if $P \stackrel{*}{\Rightarrow} x$ in n steps, then x is a palindrome.

Consider an $(n+1)$-step derivation of w, which must be of the form

$$P \Rightarrow 0P0 \stackrel{*}{\Rightarrow} 0x0 = w$$

or $P \Rightarrow 1P1 \stackrel{*}{\Rightarrow} 1x1 = w$, since $n+1$ steps is at least two steps, and the productions $P \to 0P0$ and $P \to 1P1$ are the only productions whose use allows additional steps of a derivation. Note that in either case, $P \stackrel{*}{\Rightarrow} x$ in n steps.

By the inductive hypothesis, we know that x is a palindrome; that is, $x = x^R$. But if so, then $0x0$ and $1x1$ are also palindromes. For instance, $(0x0)^R = 0x^R0 = 0x0$. We conclude that w is a palindrome, which completes the proof. □

5.1.6 Sentential Forms

Derivations from the start symbol produce strings that have a special role. We call these "sentential forms." That is, if $G = (V, T, P, S)$ is a CFG, then any string α in $(V \cup T)^*$ such that $S \stackrel{*}{\Rightarrow} \alpha$ is a *sentential form*. If $S \stackrel{*}{\underset{lm}{\Rightarrow}} \alpha$, then α is a *left-sentential form*, and if $S \stackrel{*}{\underset{rm}{\Rightarrow}} \alpha$, then α is a *right-sentential form*. Note that the language $L(G)$ is those sentential forms that are in T^*; i.e., they consist solely of terminals.

Example 5.8 : Consider the grammar for expressions from Fig. 5.2. For example, $E * (I + E)$ is a sentential form, since there is a derivation

$$E \Rightarrow E * E \Rightarrow E * (E) \Rightarrow E * (E + E) \Rightarrow E * (I + E)$$

However this derivation is neither leftmost nor rightmost, since at the last step, the middle E is replaced.

As an example of a left-sentential form, consider $a * E$, with the leftmost derivation

The Form of Proofs About Grammars

Theorem 5.7 is typical of proofs that show a grammar defines a particular, informally defined language. We first develop an inductive hypothesis that states what properties the strings derived from each variable have. In this example, there was only one variable, P, so we had only to claim that its strings were palindromes.

We prove the "if" part: that if a string w satisfies the informal statement about the strings of one of the variables A, then $A \stackrel{*}{\Rightarrow} w$. In our example, since P is the start symbol, we stated "$P \stackrel{*}{\Rightarrow} w$" by saying that w is in the language of the grammar. Typically, we prove the "if" part by induction on the length of w. If there are k variables, then the inductive statement to be proved has k parts, which must be proved as a mutual induction.

We must also prove the "only-if" part, that if $A \stackrel{*}{\Rightarrow} w$, then w satisfies the informal statement about the strings derived from variable A. Again, in our example, since we had to deal only with the start symbol P, we assumed that w was in the language of G_{pal} as an equivalent to $P \stackrel{*}{\Rightarrow} w$. The proof of this part is typically by induction on the number of steps in the derivation. If the grammar has productions that allow two or more variables to appear in derived strings, then we shall have to break a derivation of n steps into several parts, one derivation from each of the variables. These derivations may have fewer than n steps, so we have to perform an induction assuming the statement for all values n or less, as discussed in Section 1.4.2.

$$E \underset{lm}{\Rightarrow} E * E \underset{lm}{\Rightarrow} I * E \underset{lm}{\Rightarrow} a * E$$

Additionally, the derivation

$$E \underset{rm}{\Rightarrow} E * E \underset{rm}{\Rightarrow} E * (E) \underset{rm}{\Rightarrow} E * (E + E)$$

shows that $E * (E + E)$ is a right-sentential form. $\square$