# 1

# STRINGS, LANGUAGES, AND COMPILERS



## 1.1 INTRODUCTION

Compiler construction is truly an engineering science. With this science, we can methodically—almost routinely—design and implement fast, reliable, and powerful compilers.

You should study compiler construction for several reasons:

- Compiler construction techniques have very broad applicability. The usefulness of these techniques is not limited to compilers.
- To program most effectively, you need to understand the compiling process.
- Language and language translation are at the very heart of computing. You should be familiar with their theory and practice.
- Unlike some areas of computer science, you do not typically pick up compiler construction techniques "on the job." Thus, the formal study of these techniques is essential.

To be fair, you should also consider reasons for *not* studying compiler construction. Only one comes to mind: Your doctor has ordered you to avoid excitement.

## 1.2 BASIC LANGUAGE CONCEPTS

In our study of compiler design theory, we begin with several important definitions. An *alphabet* is the finite set of characters used in the writing of a language. For example, the alphabet of the Java programming language consists of all the characters that can appear in a program: the upper- and lower-case letters, the digits, whitespace (space, tab, newline, and carriage return), and all the special symbols, such as =, +, and {. For most of the examples in this book, we will use very small alphabets, such as {b, c} and {b, c, d}. We

*Compiler Construction Using Java, JavaCC, and Yacc*, First Edition. Anthony J. Dos Reis

will avoid using the letter "a" in our alphabets because of the potential confusion with the English article "a".

A *string over an alphabet* is a finite sequence of characters selected from that alphabet. For example, suppose our alphabet is {b, c, d}. Then

```
cbd
cbcc
c
```

are examples of strings over our alphabet. Notice that in a string over an alphabet, each character in the alphabet can appear any number of times (including zero times) and in any order. For example, in the string cbcc (a string over the three-letter alphabet {b, c, d}), the character b appears once, c appears three times, and d does not appear.

The *length of a string* is the number of characters the string contains. We will enclose a string with vertical bars to designate its length. For example, |cbcc| designates the length of the string cbcc. Thus, |cbcc| = 4.

A *language* is a set of strings over some alphabet. For example, the set containing just the three strings cbd, cbcc, and c is a language. This set is not a very interesting language, but it is, nevertheless, a language according to our definition.

Let us see how our definitions apply to a "real" language—the programming language Java. Consider a Java program all written on a single line:

```
class C { public static void main(String[] args) {} }
```

Clearly, such a program is a single string over the alphabet of Java. We can also view a multiple-line program as a single string—namely, the string that is formed by connecting successive lines with a line separator, such as a newline character or a carriage return/newline sequence. Indeed, a multiline program stored in a computer file is represented by just such a string. Thus, the multiple-line program

```
class C
{
  public static void main(String[] args)
  {
  }
}
```

is the single string

```
class C□{ □  public static void main(String[] args)□  {□  }□}
```

where □ represents the line separator. The *Java language* is the set of all strings over the Java alphabet that are valid Java programs.

A language can be either finite or infinite and may or may not have a meaning associated with each string. The Java language is infinite and has a meaning associated with each string. The meaning of each string in the Java language is what it tells the computer to do. In contrast, the language {cbd, cbcc, c} is finite and has no meaning associated with each string. Nevertheless, we still consider it a language. A language is simply a set, finite or infinite, of strings, each of which may or may not have an associated meaning.

*Syntax rules* are rules that define the form of the language, that is, they specify which strings are in a language. *Semantic rules* are rules that associate a meaning to each string in a language, and are optional under our definition of language.

Occasionally, we will want to represent a string with a single symbol very much like $x$ is used to represent a number in algebra. For this purpose, we will use the small letters at the end of the English alphabet. For example, we might use $x$ to represent the string `cbd` and $y$ to represent the string `cbcc`.

## 1.3 BASIC COMPILER CONCEPTS

A *compiler* is a translator. It typically translates a program (the *source program*) written in one language to an equivalent program (the *target program*) written in another language (see Figure 1.1). We call the languages in which the source and target programs are written the *source* and *target languages*, respectively.

Typically, the source language is a high-level language in which humans can program comfortably (such as Java or C++), whereas the target language is the language the computer hardware can directly handle (*machine language*) or a symbolic form of it (*assembly language*).

If the source program violates a syntax rule of the source language, we say it has a *syntax error*. For example, the following Java method has one syntax error (a right brace instead of a left brace on the second line):

```
public void greetings()
}                                  // syntax error
  System.out.println("hello");
}
```

A *logic error* is an error that does not violate a syntax rule but results in the computer performing incorrectly when we run the program. For example, suppose we write the following Java method to compute and return the sum of 2 and 3:

```
public int sum()
{
  return 2 + 30;     // logic error
}
```

This method is a valid Java method but it tells the computer to do the wrong thing—to compute 2 + 30 instead 2 + 3. Thus, the error here is a logic error.

A compiler in its simplest form consists of three parts: the token manager, the parser, and the code generator (see Fig. 1.2).

The source program that the compiler inputs is a stream of characters. The *token manager* breaks up this stream into meaningful units, called *tokens*. For example, if a token manager reads


**Figure 1.1.**

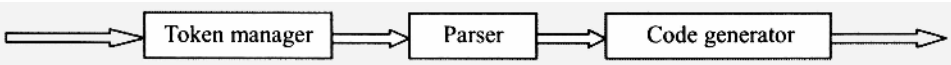


**Figure 1.2.**

```
int x;          // important example
x = 55;
```

it would output the following sequence of tokens:

```
int
x
;
x
=
55
;
```

The token manager does not produce tokens for white space (i.e., space, tab, newline, and carriage return) and comments because the parser does need these components of the source program. A token manager is sometimes called a *lexical analyzer, lexer, scanner,* or *tokenizer.*

A *parser* in its simplest form has three functions:

1. It analyzes the structure of the token sequence produced by the token manager. If it detects a syntax error, it takes the appropriate action (such as generating an error message and terminating the compile).
2. It derives and accumulates information from the token sequence that will be needed by the code generator.
3. It invokes the code generator, passing it the information it has accumulated.

The *code generator,* the last module of a compiler, outputs the target program based on the information provided by the parser.

In the compilers we will build, the parser acts as the controller. As it executes, it calls the token manager whenever it needs a token, and it calls the code generator at various points during the parse, passing the code generator the information the code generator needs. Thus, the three parts of the compiler operate concurrently. An alternate approach is to organize the compiling process into a sequence of *passes.* Each pass reads an input file and creates an output file that becomes the input file for the next pass. For example, we can organize our simple compiler into three passes. In the first pass, the token manager reads the source program and creates a file containing the tokens corresponding to the source program. In the second pass, the parser reads the file of tokens and outputs a file containing information required by the code generator. In the third pass, the code generator reads this file and outputs a file containing the target program.

## 1.4 BASIC SET THEORY

Since languages are sets of strings, it is appropriate at this point to review some basic set theory. One method of representing a set is simply to list its elements in any order. Typi-

cally, we use the left and right braces, "{" and "}", to delimit the beginning and end, respectively, of the list of elements. For example, we represent the set consisting of the integers 3 and 421 with

$$\{3, 421\} \quad \text{or} \quad \{421, 3\}$$

Similarly, we represent the set consisting of the two strings `b` and `bc` with

```
{b, bc}  or  {bc, b}
```

This approach cannot work for an infinite set because it is, of course, impossible to list all the elements of an infinite set. If, however, the elements of an infinite set follow some obvious pattern, we can represent the set by listing just the first few elements, followed by the ellipsis (. . .). For example, the set

```
{b, bb, bbb, ...}
```

represents the infinite set of strings containing one or more `b`'s and no other characters. Representing infinite sets this way, however, is somewhat imprecise because it requires the reader to figure out the pattern represented by the first few elements.

Another method for representing a set—one that works for both finite and infinite sets—is to give a rule for determining its elements. In this method, a set definition has the form

$$\{E : \textit{defining rule}\}$$

where $E$ is an expression containing one or more variables, and the defining rule generally specifies the allowable ranges of the variables in $E$. The colon means "such that." We call this representation the *set-builder notation*. For example, we can represent the set containing the integers 1 to 100 with

$$\{x : x \text{ is an integer and } 1 \leq x \leq 100\}$$

Read this definition as "the set of all $x$ such that $x$ is an integer and $x$ is greater than or equal to 1 and less than or equal to 100." A slightly more complicated example is

$$\{n^2 : n \text{ is an integer and } n \geq 1\}$$

Notice that the expression preceding the colon is not a single variable as in the preceding example. The defining rule indicates that $n$ can be 1, 2, 3, 4, and so on. The corresponding values of $n^2$ are the elements of the set—namely, 1, 4, 9, 16, etc. Thus, this is the infinite set of integer squares:

$$\{1, 4, 9, 16, \ldots\}$$

In set notation, the mathematical symbol $\in$ means "is an element of." A superimposed slash on a symbol negates the condition represented. Thus, $\notin$ means "is not a element of." For example, if $P = \{2, 3, 4\}$, then $3 \in P$, but $5 \notin P$.

The *empty set* [denoted by either $\{\}$ or $\phi$] is the set that contains no elements. The *universal set* (denoted by $U$) is the set of all elements under consideration. For example, if we are working with sets of integers, then the set of all integers is our universe. If we are

working with strings over the alphabet {b, c}, then the set of all strings over {b, c} is our universe.

The set operations *union, intersection,* and *complement,* form new sets from given sets. The union operator is most often denoted by the special symbol $\cup$. We, however, use the vertical bar | to denote the union operator. The advantage of | is that it is available on standard keyboards. We will use $\cap$ and ~ to denote the intersection and complement operators, respectively. $\cap$, the standard symbol for set intersection, unfortunately is not available on keyboards. However, we will use set intersection so infrequently that it will not be necessary to substitute a keyboard character for $\cap$.

Set union, intersection, and complement are defined as follows:

Union of $P$ and $Q$: $P \mid Q = \{x : x \in P \text{ or } x \in Q\}$
Intersection of $P$ and $Q$: $P \cap Q = \{x : x \in P \text{ and } x \in Q\}$
Complement of $P$: $\sim P = \{x : x \in U \text{ and } x \notin P\}$

Here are the definitions in words of these operators:

$P \mid Q$ is the set of all elements that are in either $P$ or $Q$ or both.
$P \cap Q$ is the set of elements that are in both $P$ and $Q$.
$\sim P$ is the set of all elements in the universe $U$ that are not in $P$.

For example, if $P$ = {b, bb}, $Q$ = {bb, bbb}, and our universe $U$ = {b, bb, bbb, . . .}, then

$P \mid Q$ = {b, bb, bbb}
$P \cap Q$ = {bb}
$\sim P$ = {bbb, bbbb, bbbbb, . . .}
$\sim Q$ = {b, bbbb, bbbbb, . . .}

A collection of sets is *disjoint* if the intersection of every pair of sets from the collection is the empty set (i.e., they have no elements in common). For example, the sets {b}, {bb, bbb}, and {bbbb} are disjoint since no two have any elements in common.

The set $P$ is a subset of $Q$ (denoted $P \subseteq Q$) if every element of $P$ is also in $Q$. The set $P$ is a proper subset of the set $Q$ (denoted $P \subset Q$) if $P$ is a subset of $Q$, and $Q$ has at least one element not in $P$. For example, if $P$ = {b, bb}, $Q$ = {b, bb, bbb}, and $R$ = {b, bb}, then $P$ is proper subset of $Q$, but $P$ is not a proper subset of $R$. However, $P$ is a subset of $R$. Two sets are equal if each is the subset of the other. With $P$ and $R$ given as above, $P \subseteq R$ and $R \subseteq P$. So we can conclude that $P = R$. Note that the empty set is a subset of any set; that is, $\{\} \subseteq S$ for any set $S$.

We can apply the set operations union, intersection, and complement to any sets. We will soon see some additional set operations specifically for sets of strings.

## 1.5 NULL STRING

When prehistoric humans started using numbers, they used the natural numbers 1, 2, 3, . . . . It was easy to grasp the idea of oneness, twoness, threeness, and so on. Therefore, it was natural to have symbols designating these concepts. In contrast, the number 0 is hardly a natural concept. After all, how could something (the symbol 0) designate nothing? Today,

of course, we are all quite comfortable with the number 0 and put it to good use every day. A similar situation applies to strings. It is natural to think of a string as a sequence of one or more characters. But, just as the concept zero is useful to arithmetic, so is the concept of a *null string*—the string whose length is zero—useful to language theory. The null string is the string that does not contain any characters.

How do we designate the null string? Normally, we designate strings by writing them down on a piece of paper. For example, to designate a string consisting of the first three small letters of the English alphabet, we write `abc`. A null string, however, does not have any characters, so there is nothing to write down. We need some symbol, preferably one that does not appear in the alphabets we use, to represent the null string. Some writers of compiler books use the Greek letter $\epsilon$ for the null string. However, since $\epsilon$ is easily confused with the symbol for set membership, we will use the small Greek letter $\lambda$ (lambda) to represent the null string.

One common misconception about the null string is that a string consisting of a single space is the null string. A space is a character whose length is one; the null string has length zero. They are not the same. Another misconception has to do with the empty set. The null string is a string. Thus, the set $\{\lambda\}$ contains exactly one string—namely the null string. The empty set $\{\}$, on the other hand, does not contain any string.

## 1.6 CONCATENATION

We call the operation of taking one string and placing it next to another string in the order given to form a new string *concatenation*. For example, if we concatenate `bcd` and `efg`, we get the string `bcdefg`. Note that the concatenation of any string $x$ with the null string $\lambda$ yields $x$. That is,

$$x\lambda = \lambda x = x$$

## 1.7 EXPONENT NOTATION

A nonnegative exponent applied to a character or a sequence of characters in a string specifies the replication of that character or sequence of characters. For example $\mathtt{b}^4$ is a shorthand representation of `bbbb`. We use parentheses if the scope of the replication is more than one character. Hence, $\mathtt{b(cd)}^2\mathtt{e}$ represents `bcdcde`. A string replicated zero times is by definition the null string; that is, for any string $x$, $x^0 = \lambda$.

We can use exponent notation along with set-builder notation to define sets of strings. For example, the set

$$\{\mathtt{b}^i : 1 \leq i \leq 3\}$$

is the set

$$\{\mathtt{b}^1, \mathtt{b}^2, \mathtt{b}^3\} = \{\mathtt{b}, \mathtt{bb}, \mathtt{bbb}\}$$

The exponent in exponent notation can never be less than zero. If we do not specify its lower bound in a set definition, assume it is zero. For example, the set

$$\{\mathtt{b}^i : i \leq 3\}$$

should be interpreted as

$$\{b^i : 0 \leq i \leq 3\} = \{b^0, b^1, b^2, b^3\} = \{\lambda, b, bb, bbb\}$$

**Exercise 1.1**

Describe in English the language defined by $\{b^i c^{2i} : i \geq 0\}$.

*Answer:*

The set of all strings consisting of b's followed by c's in which the number of c's is twice the number of b's. This set is $\{\lambda, bcc, bbcccc, bbbcccccc, \ldots\}$.

■

## 1.8 STAR OPERATOR (ALSO KNOWN AS THE ZERO-OR-MORE OPERATOR)

We have just seen that an exponent following a character represents a single string (for example, $b^3$ represents bbb). In contrast, the star operator, *, following a character (for example, b*) represents a set of strings. The set contains every possible replication (including zero replications) of the starred character. For example,

$$b^* = \{b^0, b^1, b^2, b^3, \ldots\} = \{b^n : n \geq 0\} = \{\lambda, b, bb, bbb, \ldots\}$$

Think of the star operator as meaning "zero or more."

The star operator always applies to the item immediately preceding it. If a parenthesized expression precedes the star operator, then the star applies to whatever is inside the parentheses. For example, in (bcd)*, the parentheses indicate that the star operation applies to the entire string bcd. That is,

$$(bcd)^* = \{\lambda, bcd, bcdbcd, bcdbcdbcd, \ldots\}$$

The star operator can also be applied to sets of strings. If $A$ is a set of strings, then $A^*$ is the set of strings that can be formed from the strings of $A$ using concatenation, allowing any string in $A$ to be replicated any number of times (including zero times) and used in any order. By definition, the null string is always in $A^*$.

Here are several examples of starred sets:

$\{b\}^* = \{\lambda, b, bb, bbb, \ldots\} = b^*$
$\{b, c\}^* = \{\lambda, b, c, bb, bc, cb, cc, bbb, \ldots\}$
$\{\lambda\}^* = \{\lambda\}$
$\{\}^* = \{\lambda\}$
$\{bb, cc\}^* = \{\lambda, bb, cc, bbbb, bbcc, ccbb, cccc, \ldots\}$
$\{b, cc\}^* = \{\lambda, b, bb, cc, bbb, bcc, ccb, bbbb \ldots\}$

Notice that $\{b\}^* = b^*$. That is, starring a set that contains just one string yields the same set as starring just that string.

Here is how to determine if a given string is in $A^*$, where $A$ is an arbitrary set of strings: If the given string is the null string, then it is in $A^*$ by definition. If the given string is nonnull, and it can be divided into substrings such that each substring is in $A$,

then the given string is in $A^*$. Otherwise, the string is not in $A^*$. For example, suppose $A$ = {b, cc}. We can divide the string bccbb into four parts: b, cc, b, and b, each of which is in $A$. Therefore, bccbb $\in A^*$. On the other hand, for the string bccc the required subdivision is impossible. If we divide bccc into b, cc, and c, the first two strings are in $A$ but the last is not. All other subdivisions of bccc similarly fail. Therefore, bccc $\notin A^*$.

We call the set that results from the application of the star operator to a string or set of strings the *Kleene closure,* in honor of Stephen C. Kleene, a pioneer in theoretical computer science.

Let us now use the star operator to restate two important definitions that we gave earlier. Let the capital Greek letter $\Sigma$ (sigma) represent an arbitrary alphabet. A *string over the alphabet* $\Sigma$ is any string in $\Sigma^*$. For example, suppose $\Sigma$ = {b, c}. Then

$$\Sigma^* = \{\lambda, \text{b}, \text{c}, \text{bb}, \text{bc}, \text{cb}, \text{cc}, \text{bbb}, \ldots\}$$

Thus, $\lambda$, b, c, bb, bc, cb, cc, bbb,. . . are strings over $\Sigma$. It may appear strange to view $\lambda$ as a string over the alphabet $\Sigma$ = {b, c}. Actually, this view is quite reasonable since $\lambda$ has no characters *not* in {b, c}. $\lambda$ is always a string over $\Sigma$ regardless of the content of $\Sigma$ because, by definition, $\lambda$ is always in $\Sigma^*$. A *language over the alphabet* $\Sigma$ is any subset of $\Sigma^*$. For example {$\lambda$}, {b} , and {b, cc} are each languages over $\Sigma$ = {b, c}. Even the empty set is a language over $\Sigma$ because it is a subset of $\Sigma^*$.

**Exercise 1.2**

a) List all the strings of length 3 in {b, cc}*.

b) Is ccbcc $\in$ {b, cc}*?

*Answer:*

a) bbb, bcc, ccb.

b) Yes. To confirm this, subdivide ccbcc into cc, b, and cc, all of which are elements of {b, cc}.

■

## 1.9 CONCATENATION OF SETS OF STRINGS

Concatenation can be applied to sets of strings as well as individual strings. If we let $A$ and $B$ be two sets of strings, then $AB$, the *concatenation of the sets* $A$ and $B$, is

$$\{xy : x \in A \text{ and } y \in B\}$$

That is, $AB$ is the set of all strings that can be formed by concatenating a string $A$ with a string $B$. For example, if $A$ = {b, cc} and $B$ = {d, dd}, then

$AB$ = {bd, bdd, ccd, ccdd}
$BA$ = {db, dcc, ddb, ddcc}

As an example of concatenation, consider the set b*c*, the concatenation of the sets b* and c*. Each string in b*c* consists of some string from b* concatenated to some

string in c*. That is, each string consists of zero or more b's followed by zero or more c's. The number of b's does not have to equal the number of c's, but all b's must precede all c's. Thus, b*c* = {λ, b, c, bb, bc, cc, bbb, bbc, bcc, ccc, . . .}. In exponent notation, $b^*c^* = \{b^i c^j : i \geq 0 \text{ and } j \geq 0\}$.

A string can also be concatenated with a set. If $x$ is a string and $A$ is a set of strings, then $xA$, the concatenation of $x$ with $A$ is

$$\{xy : y \in A\}$$

Similarly, $Ax$ is

$$\{yx : y \in A\}$$

For example, bbc*, the concatenation of the string bb and the set c*, is the set of all strings consisting of bb followed by a string in c*. Thus,

bbc* = {bbλ = bb, bbc, bbcc, bbccc, . . .}

Notice that it follows from our definitions that $xA = \{x\}A$, where $x$ is an arbitrary string and $A$ is a set of strings. That is, we get the same result whether we concatenate $x$ (the string) or $\{x\}$ (the set containing just $x$) to a set $A$.

**Exercise 1.3**

a) List all strings in b*cb* of length less than 3.

b) Write an expression using the star operator which defines the same set as $\{b^p c^q d^r : p \geq 0, q \geq 1, r \geq 2\}$.

*Answer:*

a) c, bc, cb.

b) b*cc*ddd*.

■

The union operator implies a choice with respect to the makeup of the strings in the language specified. For example, we can interpret

{b} ({c} | {d})

as the set of strings consisting of a b followed by a choice of c or d. That is, the set consists of the strings bc and bd.

**Exercise 1.4**

Describe in English the set defined by b*({c} | {d})e*.

*Answer:*

The set of all strings consisting of zero or more b's, followed by either c or d, followed by zero or more e's.

■

## 1.10 PLUS OPERATOR (ALSO KNOWN AS THE ONE-OR-MORE OPERATOR)

The *plus operator* is like the star operator, except that the former means "one or more" instead of "zero or more." We can apply it either to an individual string or a set of strings. It appears as a + following the item to which it applies. For example,

$$\mathtt{b}+ = \{\mathtt{b}^1, \mathtt{b}^2, \mathtt{b}^3, \ldots\} = \{\mathtt{b}^i : i \geq 1\}$$
$$\{\mathtt{b}, \mathtt{c}\}+ = \{\mathtt{b}, \mathtt{c}, \mathtt{bb}, \mathtt{bc}, \mathtt{cb}, \mathtt{cc}, \mathtt{bbb}, \ldots\}$$

$A+$, where $A$ is a set of strings, contains the null string only if the set $A$ itself contains the null string. $A^*$, on the other hand, always contains the null string for any set $A$.

Consider the set bb*. Each string in bb* consists of a single b followed a string in b*. Because the shortest string in b* is $\lambda$, the shortest string in bb* is b$\lambda$ = b. Thus, every string in bb* contains one or more b's. That is, bb* = b+. In general, for a string $x$ and a set of strings $A$,

$$xx^* = x^*x = x+$$

and

$$AA^* = A^*A = A+$$

We call the set that results from the application of the plus operator to a string or a set of strings the *positive closure*.

**Exercise 1.5**

Show that $\{\lambda\} \mid \mathtt{b}+ = \mathtt{b}^*$.

*Answer:*

$\{\lambda\} \mid \mathtt{b}+ = \{\lambda\} \mid \{\mathtt{b}, \mathtt{bb}, \mathtt{bbb}, \ldots\} = \{\lambda, \mathtt{b}, \mathtt{bb}, \mathtt{bbb}, \ldots\} = \mathtt{b}^*$.

■

## 1.11 QUESTION MARK OPERATOR (ALSO KNOWN AS ZERO-OR-ONE OPERATOR)

The question mark operator specifies an optional item. We can apply it to either an individual string or a set of strings. It appears as a ? following the item to which it applies. For example, bc? specifies a b followed by an optional c—that is, a b followed by zero or one c. Thus bc? is the set {b, bc}. Think of c? as representing the set $\{\lambda\} \mid \{\mathtt{c}\} = \{\lambda, \mathtt{c}\}$. Thus, bc? = b{$\lambda$, c} = {b, bc}.

If $A$ is a set of strings, then b$A$? specifies a b optionally followed by any single string in $A$, which is the set {b} |b$A$. For example, b{b, c}? is the set {b, bb, bc}.

**Exercise 1.6**

Show that (b+)? = b*.

*Answer:*

(b+)? = {λ} | b+ = {λ} | {b, bb, bbb, . . .} = {λ, b, bb, bbb, . . .} = b*.

■

## 1.12 SHORTHAND NOTATION FOR A SET CONTAINING A SINGLE STRING

{b} and b are not the same. The former is a set containing one string; the latter is a string—not a set containing a string. In spite of this distinction, it is common practice to represent the former (the set) by writing the latter (the string). We follow this practice only when the context clearly implies the correct interpretation, or where it does not make a difference. For example, instead of writing {c} | bd*, we can write c | bd* (recall we are using | to represent the set union). The union operator clearly implies that the c to its left must represent the set {c} and not the string c. In some expressions, it does not matter which interpretation we use. For example, whether we interpret b as a string or as a set makes no difference in the Kleene closure b*. Similarly, in b+ and b$A$, our interpretation makes no difference.

## 1.13 OPERATOR PRECEDENCE

If an expression contains more than one kind of operator, then the operations are performed in an order determined by their *precedence*. Specifically, operations with higher precedence are performed before operations with lower precedence. Our string operations, ordered from highest to lowest precedence, with equal precedence operations listed on the same line, are

Complementation
Star, Plus, Question Mark
Concatenation
Intersection
Union

For example, in c | bd*, we first apply the star to the d, then we concatenate b and d*, and last, we take the union of c and bd*. We can override this order by using parentheses. For example, in ((c | b)d)*, we perform the union first, then the concatenation, and the star operation last.

If the star, plus, or question mark operators appear consecutively, we perform their corresponding operations left to right. For example, in (bb)?+, we perform the question mark operation first, then the plus operation. Thus, (bb)?+ = {λ, bb}+ = (bb)*.

### Exercise 1.7

Write an expression without using ~ that defines the same set as ~(b*). Assume complementation is with respect to the set Σ*, where Σ = {b, c}.

*Answer:*

b* is the set of all strings over the alphabet with no c's. Therefore, its complement is the set of strings that have at least one c. Thus, every string in ~(b*) must be of the form $xcy$,

where $x$ and $y$ are arbitrary strings over {b, c}. Thus, ~(b*) = (b | c)*c(b | c)*. Another expression that defines the same set is b*c(b | c)*.

■

## 1.14 REGULAR EXPRESSIONS

Let us use our convention of designating a set containing a single string by writing just the string. For example, let us write b to represent the set {b} . Then each of these following expressions designates a set of strings:

$\phi$
$\lambda$
b
c
$\lambda$|b
bbc
b*c*
b|(cc)*
(b | c)*

For example, $\phi$, $\lambda$, b, c, and $\lambda$|b designate, respectively, the sets {}, {$\lambda$}, {b} , {c}, and {$\lambda$, b}. In every expression above, the only operations that appear, if any, are union, concatenation, and star. We call such expressions *regular expressions.* We can use regular expressions to define languages—that is, to define sets of strings.

Let us look at a precise definition of a regular expression: A *regular expression over the alphabet* $\Sigma$ is any of the following:

$\phi$
$\lambda$
any single symbol in $\Sigma$

These expressions are the *base regular expressions.* In addition, we can construct additional regular expressions using the following *construction rule:*

If $r$ and $s$ are arbitrary regular expressions, then the following expressions are also regular:

($r$)
$r|s$
$rs$
$r$*

Our construction rule allows us to construct new regular expressions, using union, concatenation, star, and parenthesis, from our base regular expressions or expressions previously constructed using the construction rule. For example, since b and c are regular expressions, so are (b), b | c, bc, and b* by our construction rule. We can continue applying our construction rule, producing ever more complex regular expressions. For example,

using our previously constructed regular expressions bc and b*, we can now, in turn, construct bc | b* with our construction rule. We are not allowed to apply our construction rule an infinite number of times when building a regular expression. This restriction implies that any regular expression must be of finite length.

Every regular expression defines (i.e., represents) a language. For example, b | c defines the language {b, c}. We call any language that can be defined with a regular expression a *regular language*.

Every regular language has more than one regular expression that defines it. For example, bb*, λbb*, and b*b all define the language consisting of one or more b's.

An assumption we have made about regular expressions over an alphabet Σ is that Σ does not contain |, *, (, or ). Thus, when we see b | c, we know the vertical bar is the union operator. However, if the vertical bar were in Σ, then b | c would be ambiguous. It could represent either the set {b, c} (if we regard | as the union operator) or the set containing the single string "b | c" (if we regard | as a symbol from Σ). A simple way to disambiguate an expression like b | c is to quote the symbols in a regular expression that come from Σ. Accordingly, we would write "b | c" or "b" "|" "c" to represent the single string "b | c". Here, the vertical bar is a symbol from Σ. We know this because the vertical bar is in quotes. But we would write "b" | "c" to represent the set {b, c}. Here, the vertical bar is the union operator. We know this because here the vertical bar is not in quotes.

**Exercise 1.8**

Give the strings in the set specified by "b" | "c" "|" "d".

*Answer:*

Two strings: "b" and "c | d".

■

Let us look at some expressions that are not regular expressions:

| | |
|---|---|
| bb \| bbb \| bbbb \| . . . | (the ellipsis ". . . " is not allowed) |
| (cc)+ | (the plus operator is not allowed) |
| (cc)? | (the question mark operator is not allowed) |
| $\{b^i : i \geq 0\}$ | (exponent and set-builder notation is not allowed) |

Because we do not allow the ellipsis, the plus operator, the exponent notation, or the set builder notation in regular expressions, the expressions above are not regular expressions. The languages they represent, however, are regular because we can represent them with regular expressions, namely bbb*, cc(cc)*, λ|cc, and b*, respectively.

When we analyze regular expressions, it is often helpful to think of the union operator as indicating a choice. For example, we can think of the regular expression (b | c) d as representing the set of strings consisting of the choice b or c followed by d—that is, as the set consisting of bd and cd.

If we allow our regular expressions to include the ~, +, and ? operators in addition to |, *, and concatenation, we get the class of expressions called *extended regular expressions*. Any language defined by an extended regular language can also be defined by a nonextended regular language. In other words, extended regular expressions are no more powerful than nonextended regular expressions in defining languages. Thus, every extended regular expression necessarily has a nonextended equivalent. For example, the extended regular expression (b | c)? has the nonextended equivalent λ|b | c.

Although extended regular expressions are no more powerful than nonextended regular expressions, they are, nevertheless, useful because they are often easier to use and understand than their nonextended equivalents.

## 1.15 LIMITATIONS OF REGULAR EXPRESSIONS

Regular expressions have limitations. They cannot represent every language. For example, consider the following language that we call *PAIRED:*

$$PAIRED = \{b^i c^i : i \geq 0\} = \{\lambda, bc, bbcc, bbbccc, bbbbcccc, \ldots\}$$

Each string in *PAIRED* consists of some number of b's followed by the *same* number of c's. With a regular expression, it is possible to capture the condition that all b's precede all c's (as in b*c*). But there is no way to capture the condition that the number of b's equals the number of c's unless we limit the length of the string (we give a proof for this in Chapter 17). The language represented by b*c* includes strings in which the number of b's is equal to the number of c's (for example, bbcc). But it also includes strings in which the number of b's is not equal to the number of c's (for example, bcc). *PAIRED,* on the other hand, contains only strings in which the number of b's is equal to the number of c's. *PAIRED* is not equal to b*c* but is, in fact, a proper subset of it.

Another attempt at a regular expression for *PAIRED* is the infinite-length expression

$$\lambda|bc|bbcc|bbbccc|bbbbcccc|\ldots$$

Although this expression does represent *PAIRED,* it is not a regular expression because regular expressions cannot be of infinite length.

Let us place an upper bound on the exponent $i$ in the preceding definition of *PAIRED* . We then get a new language that is, in fact, regular. For example,

$$\{b^i c^i : 0 \leq i \leq 2\}$$

is a regular language represented by the regular expression

$$\lambda|bc|bbcc$$

That a regular expression cannot represent the language *PAIRED* is a serious limitation since similar constructs frequently appear in programming languages. For example, in Java, arithmetic expressions may be nested with parentheses to an arbitrary depth:

((( . . .)))

Similarly, blocks of code may be nested to an arbitrary depth with braces:

{{{ . . .}}}

We cannot describe either of these constructs with regular expressions unless we place an upper limit on the depth of nesting.

When we assert that the two regular expressions are equal, we are asserting that the languages defined by those regular expressions are equal. For example, when we write

$\lambda|bb^* = b^*$

we are asserting that the language defined by $\lambda|bb^*$ is equal to the language defined by $b^*$.

Although regular expressions are too limited to fully describe the typical programming language, they are still quite useful to the compiler designer. Their usefulness appears in the design of the token manager. In particular, we can use regular expressions to describe the various tokens that the token manager provides to the parser. For example, if we let $D$ represent any digit 0 through 9, then the regular expression $DD^*$ represents an unsigned integer token. We will see in Chapter 13 how regular expressions in conjunction with the software tool JavaCC can automate the implementation of the token manager.

## PROBLEMS

1. How long is the shortest possible Java program?
2. What is the advantage of organizing a compiler into a sequence of passes?
3. Describe in words the set $\{b, c\}^*$.
4. What does the set $\{\}^*$ contain?
5. Is it true that $x^* = \{x\}^*$ for any string $x$?
6. Under what circumstances are $P$ and $\sim P$ disjoint?
7. What does $P \mid Q = P$ imply?
8. What does $P \cap Q = P$ imply?
9. If $P = \{b\}$ and $Q = \{bb, c\}$, then what does $P^* \cap Q^*$ equal?
10. If $A = \{\lambda, b\}$, how many distinct strings are in $AA$? List them.
11. Is $x^*$ always an infinite set? If not, give an example for which it is not infinite.
12. Does $b^*c^* = \{b, c\}^*$? Justify your answer.
13. Represent the set $\phi|\{\lambda\}|bbbc(bbbc)^*$ with a regular expression that does not use the | operator.
14. Using exponent notation, represent the set $b^*c^*b^*$.
15. Write a regular expression for the set of all strings over the alphabet $\{b, c\}$ containing exactly one b.
16. Write a regular expression for the set of all strings over the alphabet $\{b, c\}$ containing at least one b.
17. Write an expression using exponent notation for the set $\{b^i c^j d^j : i \geq 0, j \geq 0\} \cap \{b^p c^q d^q : p \geq 0 \text{ and } q \geq 0\}$ without using the $\cap$ operator.
18. Is $(b^*c^*)^* = \{b, c\}^*$? If not, provide a counterexample.
19. List all the strings in $\{b, cc\}^*$ that are of length 3.
20. Does $(b^*|b^*ccc)^* = \{b, ccc\}^*$? Justify your answer.
21. Is concatenation distributive over a union. That is, for all sets of strings $A$, $B$, $C$, does $A(B \mid C) = AB \mid AC$?
22. Is the star operation distributive over a union. That is, for all sets of strings $A$, $B$, does $(A|B)^* = A^*|B^*$?
23. Suppose $X$, $A$, and $B$ are sets of strings, $\lambda \notin B$, and $X = A|XB$. What can be concluded about $X$? Hint: does $X$ contain $AB$?
24. Does $xA$ always equal $\{x\}A$, where $x$ is a string and $A$ is a set of strings?

25. The parser in a compiler does not need the tokens corresponding to white space and comments. Yet, syntax errors may occur if white space and comments are removed from the source program. Explain this apparent contradiction.
26. Write a regular expression that defines the same language as b*c* ∩ c*d*.
27. Write a regular expression that defines the same language as {b, cc}* ∩ c*.
28. Write a regular expression that defines the same language as (bb)* ∩ (bbb)*.
29. Write a regular expression for the set of all strings over the alphabet {b, c} containing an even number of b's.
30. Write a nonextended regular expression that defines the same language as (~b)*, where the universe is (b | c | d)*.
31. Write a nonextended regular expression that defines the same language as ~({b, c}*), where the universe is (b | c | d)*.
32. Prove that any finite language is regular.
33. Describe in English the strings in (bb | cc|((bc | cb)(bb | cc)*(bc | cb)))*.
34. Is (((b))) a regular expression over the alphabet {b, c}?
35. Is () a regular expression over the alphabet {b, c}?
36. Give three regular expressions that define the empty set.
37. Suppose the alphabet for regular expressions consists of the symbols b, c, the backslash, the vertical bar, the single quote, and the double quote. Give an unambiguous regular expression that specifies the set consisting of b, c, the backslash, the vertical bar, the single quote, and the double quote.
38. Convert (b | c?)+ to an equivalent nonextended regular expression.
39. Show that extended regular expressions are not more powerful than regular expressions. That is, show that any language that can be defined by an extended regular expression can also be defined by a nonextended regular expression.