A description of the early FORTRAN compilers can be found in Backus [1957] and Backus [1981]. A description of an early Algol60 compiler can be found in Randell and Russell [1964]. Pascal compilers are described in Barron [1981], where a description of the Pascal P-system can also be found (Nori [1981]).

The Ratfor preprocessor mentioned in Section 1.2 is described in Kernighan [1975]. The T-diagrams of Section 1.6 were introduced by Bratman [1961].

This text focuses on standard translation techniques useful for the translation of most languages. Additional techniques may be needed for efficient translation of languages outside the main tradition of Algol-based imperative languages. In particular, the translation of functional languages such as ML and Haskell has been the source of many new techniques, some of which may become important general techniques in the future. Descriptions of these techniques can be found in Appel [1992], Peyton Jones [1992], and Peyton Jones [1987]. The latter contains a description of Hindley-Milner type checking (mentioned in Section 1.1).

Chapter 2

Scanning

The scanning, or **lexical analysis**, phase of a compiler has the task of reading the source program as a file of characters and dividing it up into tokens. Tokens are like the words of a natural language: each token is a sequence of characters that represents a unit of information in the source program. Typical examples are **keywords**, such as `if` and `while`, which are fixed strings of letters; **identifiers**, which are user-defined strings, usually consisting of letters and numbers and beginning with a letter; **special symbols**, such as the arithmetic symbols + and *; as well as a few multicharacter symbols, such as >= and <>. In each case a token represents a certain pattern of characters that is recognized, or matched, by the scanner from the beginning of the remaining input characters.

Since the task performed by the scanner is a special case of pattern matching, we need to study methods of pattern specification and recognition as they apply to the scanning process. These methods are primarily those of **regular expressions** and **finite automata**. However, a scanner is also the part of the compiler that handles the input of the source code, and since this input often involves a significant time overhead, the scanner must operate as efficiently as possible. Thus, we need also to pay close attention to the practical details of the scanner structure.

We divide the study of scanner issues as follows. First, we give an overview of the operation of a scanner and the structures and concepts involved. Then, we study regular expressions, a standard notation for representing the patterns in strings that form the lexical structure of a programming language. Following that, we study finite-state machines, or finite automata, which represent algorithms for recognizing string patterns given by regular expressions. We also study the process of constructing

finite automata out of regular expressions. We then turn to practical methods for writing programs that implement the recognition processes represented by finite automata, and we study a complete implementation of a scanner for the TINY language. Finally, we study the way the process of producing a scanner program can be automated through the use of a scanner generator, and we repeat the implementation of a scanner for TINY using Lex, which is a standard scanner generator available for use on Unix and other systems.

2.1 THE SCANNING PROCESS

It is the job of the scanner to read characters from the source code and form them into logical units to be dealt with by further parts of the compiler (usually the parser). The logical units the scanner generates are called **tokens**, and forming characters into tokens is much like forming characters into words in an English sentence and deciding which word is meant. In this it resembles the task of spelling.

Tokens are logical entities that are usually defined as an enumerated type. For example, tokens might be defined in C as[1]

```
typedef enum
    {IF,THEN,ELSE,PLUS,MINUS,NUM,ID,...}
    TokenType;
```

Tokens fall into several categories. These include the **reserved words**, such as `IF` and `THEN`, which represent the strings of characters "if" and "then." A second category is that of **special symbols**, such as the arithmetic symbols `PLUS` and `MINUS`, which represent the characters "+" and "−." Finally, there are tokens that can represent multiple strings. Examples are `NUM` and `ID`, which represent numbers and identifiers.

Tokens as logical entities must be clearly distinguished from the strings of characters that they represent. For example, the reserved word token `IF` must be distinguished from the string of two characters "if" that it represents. To make this distinction clearer, the string of characters represented by a token is sometimes called its **string value** or its **lexeme**. Some tokens have only one lexeme: reserved words have this property. A token may represent potentially infinitely many lexemes, however. Identifiers, for example, are all represented by the single token `ID`, but they have many different string values representing their individual names. These names cannot be ignored, since a compiler must keep track of them in a symbol table. Thus, a scanner must also construct the string values of at least some of the tokens. Any value associated to a token is called

1. In a language without enumerated types we would have to define tokens directly as symbolic numeric values. Thus, in old-style C one sometimes sees the following:

```
#define IF 256
#define THEN 257
#define ELSE 258
...
```

(These numbers begin at 256 to avoid confusion with numeric ASCII values.)

an **attribute** of the token, and the string value is an example of an attribute. Tokens may also have other attributes. For example, a `NUM` token may have a string value attribute such as "32767," which consists of five numeric characters, but it will also have a numeric value attribute that consists of the actual value 32767 computed from its string value. In the case of a special symbol token such as `PLUS`, there is not only the string value "+" but also the actual arithmetic operation + that is associated with it. Indeed, the token symbol itself may be viewed as simply another attribute, and the token viewed as the collection of all of its attributes.

A scanner needs to compute at least as many attributes of a token as necessary to allow further processing. For example, the string value of a `NUM` token needs to be computed, but its numeric value need not be computed immediately, since it is computable from its string value. On the other hand, if its numeric value is computed, then its string value may be discarded. Sometimes the scanner itself may perform the operations necessary to record an attribute in the appropriate place, or it may simply pass on the attribute to a later phase of the compiler. For example, a scanner could use the string value of an identifier to enter it into the symbol table, or it could pass it along to be entered at a later stage.

Since the scanner will have to compute possibly several attributes for each token, it is often helpful to collect all the attributes into a single structured data type, which we could call a **token record**. Such a record could be declared in C as

```
typedef struct
   { TokenType tokenval;
     char * stringval;
     int numval;
   } TokenRecord;
```

or possibly as a union

```
typedef struct
   { TokenType tokenval;
     union
     { char * stringval;
       int numval;
     } attribute;
   } TokenRecord;
```

(which assumes that the string value attribute is needed only for identifiers and the numeric value attribute only for numbers). A more common arrangement is for the scanner to return the token value only and place the other attributes in variables where they can be accessed by other parts of the compiler.

Although the task of the scanner is to convert the entire source program into a sequence of tokens, the scanner will rarely do this all at once. Instead, the scanner will operate under the control of the parser, returning the single next token from the input on demand via a function that will have a declaration similar to the C declaration

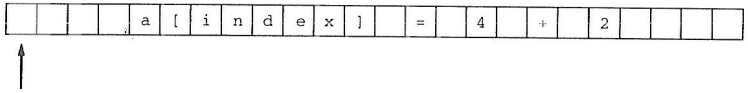
```
TokenType getToken(void);
```

The **getToken** function declared in this manner will, when called, return the next token from the input, as well as compute additional attributes, such as the string value of the token. The string of input characters is usually not made a parameter to this function, but is kept in a buffer or provided by the system input facilities.

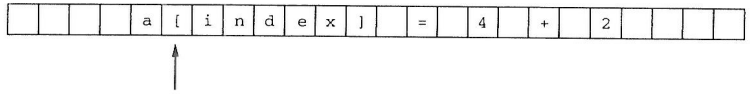
As an example of the operation of **getToken**, consider the following line of C source code, which we used as an example in Chapter 1:

```
a[index] = 4 + 2
```

Suppose that this line of code is stored in an input buffer as follows, with the next input character indicated by the arrow:



A call to **getToken** will now need to skip the next four blanks, recognize the string "a" consisting of the single character *a* as the next token, and return the token value **ID** as the next token, leaving the input buffer as follows:



Thus, a subsequent call to **getToken** will begin the recognition process again with the left bracket character.

We turn now to the study of methods for defining and recognizing patterns in strings of characters.

2.2 REGULAR EXPRESSIONS

Regular expressions represent patterns of strings of characters. A regular expression *r* is completely defined by the set of strings that it matches. This set is called the **language generated by the regular expression** and is written as $L(r)$. Here the word *language* is used only to mean "set of strings" and has (at least at this stage) no specific relationship to a programming language. This language depends, first, on the character set that is available. Generally, this will be the set of ASCII characters or some subset of it. Sometimes the set will be more general than the ASCII character set, in which case the set elements are referred to as **symbols**. This set of legal symbols is called the **alphabet** and is usually written as the Greek symbol Σ (sigma).

A regular expression *r* will also contain characters from the alphabet, but such characters have a different meaning: in a regular expression, all symbols indicate *patterns*. In this chapter, we will distinguish the use of a character as a pattern by writing all patterns in boldface. Thus, **a** is the character *a* used as a pattern.

Last, a regular expression *r* may contain characters that have special meanings. Such characters are called **metacharacters** or **metasymbols**. These generally may not be legal characters in the alphabet, or we could not distinguish their use as metacharacter from their use as a member of the alphabet. Often, however, it is not possible to require such an exclusion, and a convention must be used to differentiate the two possible uses of a metacharacter. In many situations this is done by using an **escape character** that "turns off" the special meaning of a metacharacter. Common escape characters are the backslash and quotes. Note that escape characters are themselves metacharacters, if they are also legal characters in the alphabet.

2.2.1 Definition of Regular Expressions

We are now in a position to describe the meaning of regular expressions by stating which languages are generated by each pattern. We do this in several stages. First, we describe the set of basic regular expressions, which consist of individual symbols. Then, we describe operations that generate new regular expressions from existing ones. This is similar to the way arithmetic expressions are constructed: the basic arithmetic expressions are the numbers, such as 43 and 2.5. Then arithmetic operations, such as addition and multiplication, can be used to form new expressions from existing ones, as in 43 * 2.5 and 43 * 2.5 + 1.4.

The group of regular expressions that we describe here is minimal in the sense that it contains only the essential operations and metasymbols. Later we will consider extensions to this minimal set.

Basic Regular Expressions These are just the single characters from the alphabet, which match themselves. Given any character *a* from the alphabet Σ, we indicate that the regular expression **a** matches the character *a* by writing $L(\mathbf{a}) = \{a\}$. There are two additional symbols that we will need in special situations. We need to be able to indicate a match of the **empty string**, that is, the string that contains no characters at all. We use the symbol **ε** (epsilon) to denote the empty string, and we define the metasymbol ε (boldface ε) by setting $L(\varepsilon) = \{\varepsilon\}$. We also occasionally need to be able to write a symbol that matches no string at all, that is, whose language is the **empty set**, which we write as $\{\}$. We use the symbol **ϕ** for this, and we write $L(\phi) = \{\}$. Note the difference between $\{\}$ and $\{\varepsilon\}$: the set $\{\}$ contains no strings at all, while the set $\{\varepsilon\}$ contains the single string consisting of no characters.

Regular Expression Operations There are three basic operations in regular expressions: (1) choice among alternatives, which is indicated by the metacharacter | (vertical bar); (2) concatenation, which is indicated by juxtaposition (without a metacharacter); and (3) repetition or "closure," which is indicated by the metacharacter *. We discuss each of these in turn, giving the corresponding set construction for the languages of matched strings.

Choice Among Alternatives If *r* and *s* are regular expressions, then *r*|*s* is a regular expression which matches any string that is matched either by *r* or by *s*. In terms of languages, the language of *r*|*s* is the **union** of the languages of *r* and *s*, or $L(r|s) = L(r) \cup L(s)$. As

a simple example, consider the regular expression `a|b`: it matches either of the characters *a* or *b*, that is, $L(\texttt{a|b}) = L(\texttt{a}) \cup L(\texttt{b}) = \{a\} \cup \{b\} = \{a, b\}$. As a second example, the regular expression `a|ε` matches either the single character *a* or the empty string (consisting of no characters). In other words, $L(\texttt{a|}\varepsilon) = \{a, \varepsilon\}$.

Choice can be extended to more than one alternative, so that, for example, $L(\texttt{a|b|c|d}) = \{a, b, c, d\}$. We also sometimes write long sequences of choices with dots, as in `a|b|...|z`, which matches any of the lowercase letters *a* through *z*.

Concatenation The concatenation of two regular expressions *r* and *s* is written as *rs*, and it matches any string that is the concatenation of two strings, the first of which matches *r* and the second of which matches *s*. For example, the regular expression `ab` matches only the string *ab*, while the regular expression `(a|b)c` matches the strings *ac* and *bc*. (The use of parentheses as metacharacters in this regular expression will be explained shortly.)

We can describe the effect of concatenation in terms of generated languages by defining the concatenation of two sets of strings. Given two sets of strings S_1 and S_2, the concatenated set of strings S_1S_2 is the set of strings of S_1 appended by all the strings of S_2. For example, if $S_1 = \{aa, b\}$ and $S_2 = \{a, bb\}$, then $S_1S_2 = \{aaa, aabb, ba, bbb\}$. Now the concatenation operation for regular expressions can be defined as follows: $L(rs) = L(r)L(s)$. Thus (using our previous example), $L(\texttt{(a|b)c}) = L(\texttt{a|b})L(\texttt{c}) = \{a, b\}\{c\} = \{ac, bc\}$.

Concatenation can also be extended to more than two regular expressions: $L(r_1r_2 \ldots r_n) = L(r_1)L(r_2) \ldots L(r_n)$ = the set of strings formed by concatenating all strings from each of $L(r_1), \ldots, L(r_n)$.

Repetition The repetition operation of a regular expression, sometimes also called **(Kleene) closure**, is written *r**, where *r* is a regular expression. The regular expression *r** matches any finite concatenation of strings, each of which matches *r*. For example, `a*` matches the strings ε, *a*, *aa*, *aaa*, ... (It matches ε because ε is the concatenation of *no* strings that match `a`.) We can define the repetition operation in terms of generated languages by defining a similar operation * for sets of strings. Given a set *S* of strings, let

$$S^* = \{\varepsilon\} \cup S \cup SS \cup SSS \cup \ldots$$

This is an infinite set union, but each element in it is a finite concatenation of strings from *S*. Sometimes the set *S** is written as follows:

$$S^* = \bigcup_{n=0}^{\infty} S^n$$

where $S^n = S \ldots S$ is the concatenation of *S* *n*-times. ($S^0 = \{\varepsilon\}$.)

Now we can define the repetition operation for regular expressions as follows:

$$L(r^*) = L(r)^*$$

As an example, consider the regular expression `(a|bb)*`. (Again, the reason for the parentheses as metacharacters will be explained later.) This regular expression matches any of the following strings: ε, *a*, *bb*, *aa*, *abb*, *bba*, *bbbb*, *aaa*, *aabb*, and so on. In terms of languages, $L(\texttt{(a|bb)*}) = L(\texttt{a|bb})^* = \{a, bb\}^* = \{\varepsilon, a, bb, aa, abb, bba, bbbb, aaa, aabb, abba, abbbb, bbaa, \ldots\}$.

Precedence of Operations and Use of Parentheses The foregoing description neglected the question of the precedence of the choice, concatenation, and repetition operations. For example, given the regular expression `a|b*`, should we interpret this as `(a|b)*` or as `a|(b*)`? (There is a significant difference, since $L(\texttt{(a|b)*}) = \{\varepsilon, a, b, aa, ab, ba, bb, \ldots\}$, while $L(\texttt{a|(b*)}) = \{\varepsilon, a, b, bb, bbb, \ldots\}$.) The standard convention is that repetition should have higher precedence, so that the second interpretation is the correct one. Indeed, among the three operations, * is given the highest precedence, concatenation is given the next highest, and | is given the lowest. Thus, for example, `a|bc*` is interpreted as `a|(b(c*))`, and `ab|c*d` is interpreted as `(ab)|((c*)d)`.

When we wish to indicate a different precedence, we must use parentheses to do so. This is the reason we had to write `(a|b)c` to indicate that the choice operation should be given higher precedence than concatenation, for otherwise `a|bc` is interpreted as matching either *a* or *bc*. Similarly, without parentheses `(a|bb)*` would be interpreted as `a|bb*`, which matches *a*, *b*, *bb*, *bbb*, ... This use of parentheses is entirely analogous to their use in arithmetic, where (3 + 4) * 5 = 35, but 3 + 4 * 5 = 23, since * is assumed to have higher precedence than +.

Names for Regular Expressions Often, it is helpful as a notational simplification to give a name to a long regular expression, so that we do not have to write the expression itself each time we wish to use it. As an example, if we want to develop a regular expression for a sequence of one or more numeric digits, then we could write

```
(0|1|2|...|9)(0|1|2|...|9)*
```

or we could write

```
digit digit*
```

where

```
digit = 0|1|2|...|9
```

is a **regular definition** of the name `digit`.

The use of a regular definition is a great convenience, but it does introduce the added complication that the name itself then becomes a metasymbol and a means must be found to distinguish the name from the concatenation of its characters. In our case, we have made that distinction by using italics for the name. Note that a name cannot be used in its own definition (i.e., recursively)—we must be able to remove names by successively replacing them with the regular expressions for which they stand.

Before considering a series of examples to elaborate our definition of regular expressions, we collect all the pieces of the definition of a regular expression together.

Definition

A **regular expression** is one of the following:

1. A **basic** regular expression, consisting of a single character **a**, where a is from an alphabet Σ of legal characters; the metacharacter ε; or the metacharacter ϕ. In the first case, $L(\mathbf{a}) = \{a\}$; in the second, $L(\varepsilon) = \{\varepsilon\}$; in the third, $L(\phi) = \{\}$.
2. An expression of the form $r|s$, where r and s are regular expressions. In this case, $L(r|s) = L(r) \cup L(s)$.
3. An expression of the form rs, where r and s are regular expressions. In this case, $L(rs) = L(r)L(s)$.
4. An expression of the form $r*$, where r is a regular expression. In this case, $L(r*) = L(r)*$.
5. An expression of the form (r), where r is a regular expression. In this case, $L((r)) = L(r)$. Thus, parentheses do not change the language. They are used only to adjust the precedence of the operations.

We note that, in this definition, the precedence of the operations in (2), (3), and (4) is in reverse order of their listing; that is, | has lower precedence than concatenation and concatenation has lower precedence than *. We also note that this definition gives a metacharacter meaning to the six symbols ϕ, ε, |, *, (,).

In the remainder of this section, we consider a series of examples designed to elaborate on the definition we have just given. These are somewhat artificial in that they do not usually appear as token descriptions in a programming language. In Section 2.2.3, we consider some common regular expressions that often appear as tokens in programming languages.

In the following examples, there generally is an English description of the strings to be matched, and the task is to translate the description into a regular expression. This situation, where a language manual contains descriptions of the tokens, is the most common one facing compiler writers. Occasionally, it may be necessary to reverse the direction, that is, move from a regular expression to an English description, so we also include a few exercises of this kind.

Example 2.1

Consider the simple alphabet consisting of just three alphabetic characters: $\Sigma = \{a, b, c\}$. Consider the set of all strings over this alphabet that contain exactly one b. This set is generated by the regular expression

```
(a|c)*b(a|c)*
```

Note that, even though **b** appears in the center of the regular expression, the letter b need not be in the center of the string being matched. Indeed, the repetition of a or c before and after the b may occur different numbers of times. Thus, all the following strings are matched by the above regular expression: b, abc, $abaca$, $baaaac$, $ccbaca$, $cccccb$. §

Example 2.2

With the same alphabet as before, consider the set of all strings that contain at most one b. A regular expression for this set can be obtained by using the solution to the previous example as one alternative (matching those strings with exactly one b) and the regular expression `(a|c)*` as the other alternative (matching no b's at all). Thus, we have the following solution:

```
(a|c)*|(a|c)*b(a|c)*
```

An alternative solution would allow either b or the empty string to appear between the two repetitions of a or c:

```
(a|c)*(b|ε)(a|c)*
```

This example brings up an important point about regular expressions: the same language may be generated by many different regular expressions. Usually, we try to find as simple a regular expression as possible to describe a set of strings, though we will never attempt to prove that we have in fact found the "simplest"—for example, the shortest. There are two reasons for this. First, it rarely comes up in practical situations, where there is usually one standard "simplest" solution. Second, when we study methods for recognizing regular expressions, the algorithms there will be able to simplify the recognition process without bothering to simplify the regular expression first. §

Example 2.3

Consider the set of strings S over the alphabet $\Sigma = \{a, b\}$ consisting of a single b surrounded by the same number of a's:

$$S = \{b, aba, aabaa, aaabaaa, \ldots\} = \{a^n b a^n | n \neq 0\}$$

This set cannot be described by a regular expression. The reason is that the only repetition operation we have is the closure operation *, which allows any number of repetitions. So if we write the expression `a*ba*` (about as close as we can get to a regular expression for S), then there is no guarantee that the number of a's before and after the b will be the same. We express this by saying that "regular expressions can't count." To give a mathematical proof of this fact, however, would require the use of a famous theorem about regular expressions called the **pumping lemma**, which is studied in automata theory, but which we will not mention further here.

Clearly, not all sets of strings that we can describe in simple terms can be generated by regular expressions. A set of strings that *is* the language for a regular expression is, therefore, distinguished from other sets by calling it a **regular set**. Occasionally, nonregular sets appear as strings in programming languages that need to be recognized by a scanner. These are usually dealt with when they arise, and we will return to this matter again briefly in the section on practical scanner considerations. §

Example 2.4

Consider the strings over the alphabet $\Sigma = \{a, b, c\}$ that contain no two consecutive b's. Thus, between any two b's there must be at least one a or c. We build up a regular

expression for this set in several stages. First, we can force an a or c to come *after* every b by writing

```
(b(a|c))*
```

We can combine this with the expression `(a|c)*`, which matches strings that have no b's at all, and write

```
((a|c)*|(b(a|c))*)*
```

or, noting that $(r^*|s^*)^*$ matches the same strings as $(r|s)^*$

```
((a|c)|(b(a|c)))*
```

or

```
(a|c|ba|bc)*
```

(Warning! This is not yet the correct answer.)

The language generated by this regular expression does, indeed, have the property we seek, namely, that there are no two consecutive b's (but isn't quite correct yet). Occasionally, we should prove such assertions, so we sketch a proof that all strings in $L($`(a|c|ba|bc)*`$)$ contain no two consecutive b's. The proof is by induction on the length of the string (i.e., the number of characters in the string). Clearly, it is true for all strings of length 0, 1, or 2: these strings are precisely the strings ε, a, c, aa, ac, ca, cc, ba, bc. Now, assume it is true for all strings in the language of length $i < n$, and let s be a string in the language of length $n > 2$. Then, s contains more than one of the non-ε strings just listed, so $s = s_1s_2$, where s_1 and s_2 are also in the language and are not ε. Hence, by the induction assumption, both s_1 and s_2 have no two consecutive b's. Thus, the only way s itself could have two consecutive b's would be for s_1 to end with a b and for s_2 to begin with a b. But this is impossible, since no string in the language can end with a b.

This last fact that we used in the proof sketch—that no string generated by the preceding regular expression can end with a b—also shows why our solution is not yet quite correct: it does not generate the strings b, ab, and cb, which contain no two consecutive b's. We fix this by adding an optional trailing b, as follows:

```
(a|c|ba|bc)*(b|ε)
```

Note that the mirror image of this regular expression also generates the given language

```
(b|ε)(a|c|ab|cb)*
```

We could also generate this same language by writing

```
(notb|b notb)*(b|ε)
```

where `notb = a|c`. This is an example of the use of a name for a subexpression. This solution is in fact preferable in cases where the alphabet is large, since the definition of `notb` can be adjusted to include all characters except b, without complicating the original expression. §

Example 2.5

This example is one where we are given the regular expression and are asked to determine a concise English description of the language it generates. Consider the alphabet $\Sigma = \{a, b, c\}$ and the regular expression

```
((b|c)*a(b|c)*a)*(b|c)*
```

This generates the language of all strings containing an even number of a's. To see this, consider the expression inside the outer left repetition:

```
(b|c)*a(b|c)*a
```

This generates those strings ending in a that contain exactly two a's (any number of b's and c's can appear before or between the two a's). Repeating these strings gives all strings ending in a whose number of a's is a multiple of 2 (i.e., even). Tacking on the repetition `(b|c)*` at the end (as in the previous example) gives the desired result.

We note that this regular expression could also be written as

```
(nota* a nota* a)* nota*
```

§

2.2.2 Extensions to Regular Expressions

We have given a definition of regular expressions that uses a minimal set of operations common to all applications, and we could limit ourselves to using only the three basic operations (together with parentheses) in all our examples. However, we have already seen in the examples so far that writing regular expressions using only these operators is sometimes unwieldy, creating regular expressions that are more complicated than they would be if a more expressive set of operations were available. For example, it would be useful to have a notation for a match of any character (we now have to list every character in the alphabet in a long alternative). In addition, it would help to have a regular expression for a range of characters and a regular expression for all characters except one character.

In the following paragraphs we will describe some extensions to the standard regular expressions already discussed, with corresponding new metasymbols, that cover these and similar common situations. In most of these cases no common terminology exists, so we will use a notation similar to that used by the scanner generator Lex, which is described later in this chapter. Indeed, many of the situations we are about to describe will appear again in our description of Lex. Not all applications that use regular expressions will include these operations, however, and even when they do, a different notation may be used.

We now proceed to our list of new operations.

ONE OR MORE REPETITIONS

Given a regular expression r, repetition of r is described using the standard closure operation, written r^*. This allows r to be repeated 0 or more times. A typical situation that arises is the need for *one* or more repetitions instead of none, which guarantees that at least one string matching r appears, disallowing the empty string ε. An example is that of a natural number, where we want a sequence of digits, but we want at least one digit to appear. For example, if we want to match binary numbers,

we could write `(0|1)*`, but this will also match the empty string, which is not a number. We could, of course, write

```
(0|1)(0|1)*
```

but this situation occurs often enough that a relatively standard notation has been developed for it that uses `+` instead of `*`: *r*+ indicates one or more repetitions of *r*. Thus, our previous regular expression for binary numbers can now be written

```
(0|1)+
```

ANY CHARACTER

A common situation is the need to match any character in the alphabet. Without a special operation this requires that every character in the alphabet be listed in an alternative. A typical metacharacter that is used to express a match of any character is the period "`.`", which does not require that the alphabet actually be written out. Using this metacharacter, we can write a regular expression for all strings that contain at least one *b* as follows:

```
.*b.*
```

A RANGE OF CHARACTERS

Often, we need to write a range of characters, such as all lowercase letters or all digits. We have done this up to now by using the notation `a|b|...|z` for the lowercase letters or `0|1|...|9` for the digits. An alternative is to have a special notation for this situation, and a common one is to use square brackets and a hyphen, as in `[a-z]` for the lowercase letters and `[0-9]` for the digits. This can also be used for individual alternatives, so that `a|b|c` can be written as `[abc]`. Multiple ranges can be included as well, so that `[a-zA-Z]` represents all lowercase and uppercase letters. This general notation is referred to as **character classes**. Note that this notation may depend on the underlying order of the character set. For example, writing `[A-Z]` assumes that the characters *B*, *C*, and so on come between the characters *A* and *Z* (a reasonable assumption) and that *only* the uppercase characters are between *A* and *Z* (true for the ASCII character set). Writing `[A-z]` will *not* match the same characters as `[A-Za-z]`, however, even in the ASCII character set.

ANY CHARACTER NOT IN A GIVEN SET

As we have seen, it is often helpful to be able to exclude a single character from the set of characters to be matched. This can be achieved by designating a metacharacter to indicate the "not" or complement operation on a set of alternatives. For examples, a standard character representing "not" in logic is the tilde character `~`, and we could write a regular expression for a character in the alphabet that is not *a* as `~a` and a character that is not either *a* or *b* or *c* as

```
~(a|b|c)
```

An alternative to this notation is used in Lex, where the carat character `^` is used in conjunction with the character classes just described to form complements. For

example, any character that is not *a* is written as `[^a]`, and any character that is not *a* or *b* or *c* is written as

```
[^abc]
```

OPTIONAL SUBEXPRESSIONS

A final common occurrence is for strings to contain optional parts that may or may not appear in any particular string. For example, a number may or may not have a leading sign, such as `+` or `-`. We can use alternatives to express this, as in the regular definitions

```
natural = [0-9]+
signedNatural = natural | + natural | - natural
```

This can quickly become cumbersome, however, and we introduce the question mark metacharacter *r*`?` to indicate that strings matched by *r* are optional (or that 0 or 1 copies of *r* are present). Thus, the leading sign example becomes

```
natural = [0-9]+
signedNatural = (+|-)? natural
```

2.2.3 Regular Expressions for Programming Language Tokens

Programming language tokens tend to fall into several limited categories that are fairly standard across many different programming languages. One category is that of **reserved words**, sometimes also called **keywords**, which are fixed strings of alphabetic characters that have special meaning in the language. Examples include `if`, `while`, and `do` in such languages as Pascal, C, and Ada. Another category consists of the **special symbols**, including arithmetic operators, assignment, and equality. These can be a single character, such as `=`, or multiple characters, such as `:=` or `++`. A third category consists of **identifiers**, which commonly are defined to be sequences of letters and digits beginning with a letter. A final category consists of **literals** or **constants**, which can include numeric constants such as 42 and 3.14159, string literals such as "hello, world," and characters such as "a" and "b." We describe typical regular expressions for some of these here and discuss a few other issues related to the recognition of tokens. More detail on practical recognition issues appears later in the chapter.

Numbers Numbers can be just sequences of digits (natural numbers), or decimal numbers, or numbers with an exponent (indicated by an e or E). For example, 2.71E-2 represents the number .0271. We can write regular definitions for these numbers as follows:

```
nat = [0-9]+
signedNat = (+|-)? nat
number = signedNat("." nat)?(E signedNat)?
```

Here we have written the decimal point inside quotes to emphasize that it should be matched directly and not be interpreted as a metacharacter.

Reserved Words and Identifiers Reserved words are the simplest to write as regular expressions: they are represented by their fixed sequences of characters. If we wanted to collect all the reserved words into one definition, we could write something like

```
reserved = if | while | do | ...
```

Identifiers, on the other hand, are strings of characters that are not fixed. Typically, an identifier must begin with a letter and contain only letters and digits. We can express this in terms of regular definitions as follows:

```
letter = [a-zA-Z]
digit = [0-9]
identifier = letter(letter|digit)*
```

Comments Comments typically are ignored during the scanning process.[2] Nevertheless, a scanner must recognize comments and discard them. Thus, we will need to write regular expressions for comments, even though a scanner may have no explicit constant token (we could call these **pseudotokens**). Comments can have a number of different forms. Typically, they are either free format and surrounded by delimiters such as

```
{this is a Pascal comment}
/* this is a C comment */
```

or they begin with a specified character or characters and continue to the end of the line, as in

```
; this is a Scheme comment
-- this is an Ada comment
```

It is not hard to write a regular expression for comments that have single-character delimiters, such as the Pascal comment, or for those that reach from some specified character(s) to the end of the line. For example, the Pascal comment case can be written as

```
{(~})*}
```

where we have written `~}` to indicate "not `}`" and where we have assumed that the character `}` has no meaning as a metacharacter. (A different expression must be written for Lex, which we discuss later in this chapter.) Similarly, an Ada comment can be matched by the regular expression

```
--(~newline)*
```

2. Sometimes they can contain compiler directives.

in which we assume that ***newline*** matches the end of a line (writable as `\n` on many systems), that the "`-`" character has no meaning as a metacharacter, and that the trailing end of the line is not included in the comment itself. (We will see how to write this in Lex in Section 2.6.)

It is much more difficult to write down a regular expression for the case of delimiters that are more than one character in length, such as C comments. To see this, consider the set of strings *ba. . .*(no appearances of *ab*). . *.ab* (we use *ba. . .ab* instead of the C delimiters /*. . .*/, since the asterisk, and sometimes the forward slash, is a metacharacter that requires special handling). We cannot simply write

```
ba(~(ab))*ab
```

because the "not" operator is usually restricted to single characters rather than strings of characters. We can try to write out a definition for `~(ab)` using `~a`, `~b`, and `~(a|b)`, but this is not trivial. One solution is

```
b*(a*~(a|b)b*)*a*
```

but this is difficult to read (and to prove correct). Thus, a regular expression for C comments is so complicated that it is almost never written in practice. In fact, this case is usually handled by ad hoc methods in actual scanners, which we will see later in this chapter.

Finally, another complication in recognizing comments is that, in some programming languages, comments can be nested. For example, Modula-2 allows comments of the form

```
(* this is (* a Modula-2 *) comment *)
```

Comment delimiters must be paired exactly in such nested comments, so that the following is not a legal Modula-2 comment:

```
(* this is (* illegal in Modula-2 *)
```

Nesting of comments requires the scanner to count the numbers of delimiters. But we have noted in Example 2.3 (Section 2.2.1) that regular expressions cannot express counting operations. In practice, we use a simple counter scheme as an ad hoc solution to this problem (see the exercises).

Ambiguity, White Space, and Lookahead Frequently, in the description of programming language tokens using regular expressions, some strings can be matched by several different regular expressions. For example, strings such as `if` and `while` could be either identifiers or keywords. Similarly, the string `<>` might be interpreted as representing either two tokens ("less than" and "greater than") or a single token ("not equal to"). A programming language definition must state which interpretation is to be observed, and the regular expressions themselves cannot do this. Instead, a language definition must give **disambiguating rules** that will imply which meaning is meant for each such case.

Two typical rules that handle the examples just given are the following. First, when a string can be either an identifier or a keyword, keyword interpretation is generally preferred. This is implied by using the term **reserved word**, which means simply a key-