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

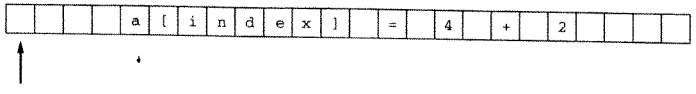
```
TokenType getToken(void);
```

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.

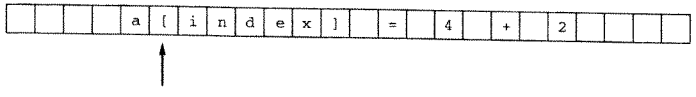
As an example of the operation of **getToken**, consider the following line of C source code, which we used as an example in Chapter 1:

```
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 $\boldsymbol{\varepsilon}$ (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 $\boldsymbol{\phi}$ for this, and we write $L(\boldsymbol{\phi}) = \{\}$. 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, $L(\mathbf{a}|\mathbf{b}) = L(\mathbf{a}) \cup L(\mathbf{b}) = \{a\} \cup \{b\} = \{a, b\}$. 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, $L(\mathbf{a}|\varepsilon) = \{a, \varepsilon\}$.

Choice can be extended to more than one alternative, so that, for example, $L(\mathbf{a}|\mathbf{b}|\mathbf{c}|\mathbf{d}) = \{a, b, c, d\}$. 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), $L(\mathbf{(a|b)c}) = L(\mathbf{a|b})L(\mathbf{c}) = \{a, b\}\{c\} = \{ac, bc\}$.

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, $L(\mathbf{(a|bb)^*}) = L(\mathbf{a|bb})^* = \{a, bb\}^* = \{\varepsilon, a, bb, aa, abb, bba, bbbb, aaa, aabb, abba, abbbb, bbaa, \ldots\}$.

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 $L(\mathbf{(a|b)^*}) = \{\varepsilon, a, b, aa, ab, ba, bb, \ldots\}$, while $L(\mathbf{a|(b^*)}) = \{\varepsilon, a, b, bb, bbb, \ldots\}$.) 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, $L(\boldsymbol{\varepsilon}) = \{\varepsilon\}$; in the third, $L(\boldsymbol{\phi}) = \{\}$.
2. An expression of the form $\boldsymbol{r}|\boldsymbol{s}$, where r and s are regular expressions. In this case, $L(\boldsymbol{r}|\boldsymbol{s}) = L(r) \cup L(s)$.
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-

word that cannot also be an identifier. Second, when a string can be a single token or a sequence of several tokens, the single-token interpretation is typically preferred. This preference is often referred to as the **principle of longest substring**: the longest string of characters that could constitute a single token at any point is assumed to represent the next token.[3]

An issue that arises with the use of the principle of longest substring is the question of **token delimiters**, or characters that imply that a longer string at the point where they appear cannot represent a token. Characters that are unambiguously part of other tokens are delimiters. For example, in the string `xtemp=ytemp`, the equal sign delimits the identifier `xtemp`, since = cannot appear as part of an identifier. Blanks, newlines, and tab characters are generally also assumed to be token delimiters: `while x ...` is thus interpreted as containing the two tokens representing the reserved word `while` and the identifier with name `x`, since a blank separates the two character strings. In this situation it is often helpful to define a white space pseudotoken, similar to the comment pseudotoken, which simply serves the scanner internally to distinguish other tokens. Indeed, comments themselves usually serve as delimiters, so that, for example, the C code fragment

```
do/**/if
```

represents the two reserved words `do` and `if` rather than the identifier `doif`.

A typical definition of the white space pseudotoken in a programming language is

```
whitespace = (newline|blank|tab|comment)+
```

where the identifiers on the right stand for the appropriate characters or strings. Note that, other than acting as a token delimiter, white space is usually ignored. Languages that specify this behavior are called **free format**. Alternatives to free format include the fixed format of a few languages like FORTRAN and various uses of indentation, such as the **offside rule** (see the Notes and References section). A scanner for a free-format language must discard white space after checking for any token delimiting effects.

Delimiters end token strings but they are not part of the token itself. Thus, a scanner must deal with the problem of **lookahead**: when it encounters a delimiter, it must arrange that the delimiter is not removed from the rest of the input, either by returning it to the input string ("backing up") or by looking ahead before removing the character from the input. In most cases, it is only necessary to do this for a single character ("single-character lookahead"). For example, in the string `xtemp=ytemp`, the end of the identifier `xtemp` is found when the = is encountered, and the = must remain in the input, since it represents the next token to be recognized. Note also that lookahead may not be necessary to recognize a token. For example, the equal sign may be the only token that begins with the = character, in which case it can be recognized immediately without consulting the next character.

Sometimes a language may require more than single-character lookahead, and the scanner must be prepared to back up possibly arbitrarily many characters. In that case, buffering of input characters and marking places for backtracking become issues in the design of a scanner. (Some of these questions are dealt with later on in this chapter.)

3. Sometimes this is called the principle of "maximal munch."

FORTRAN is a good example of a language that violates many of the principles we have just been discussing. FORTRAN is a fixed-format language in which white space is removed by a preprocessor before translation begins. Thus, the FORTRAN line

```
I F ( X 2 . EQ. 0) THE N
```

would appear to a compiler as

```
IF(X2.EQ.0)THEN
```

so white space no longer functions as a delimiter. Also, there are no reserved words in FORTRAN, so all keywords can also be identifiers, and the position of the character string in each line of input is important in determining the token to be recognized. For example, the following line of code is perfectly correct FORTRAN:

```
IF(IF.EQ.0)THENTHEN=1.0
```

The first `IF` and `THEN` are keywords, while the second `IF` and `THEN` are identifiers representing variables. The effect of this is that a FORTRAN scanner must be able to backtrack to arbitrary positions within a line of code. Consider, for concreteness, the following well-known example:

```
DO99I=1,10
```

This initiates a loop comprising the subsequent code up to the line whose number is 99, with the same effect as the Pascal `for i := 1 to 10`. On the other hand, changing the comma to a period

```
DO99I=1.10
```

changes the meaning of the code completely: this assigns the value 1.1 to the variable with name `DO99I`. Thus, a scanner cannot conclude that the initial `DO` is a keyword until it reaches the comma (or period), in which case it may be forced to backtrack to the beginning of the line and start over.

2.3 FINITE AUTOMATA

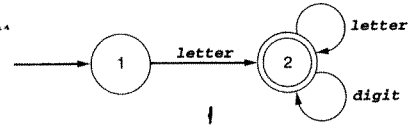
Finite automata, or finite-state machines, are a mathematical way of describing particular kinds of algorithms (or "machines"). In particular, finite automata can be used to describe the process of recognizing patterns in input strings, and so can be used to construct scanners. There is also, of course, a strong relationship between finite automata and regular expressions, and we will see in the next section how to construct a finite automaton from a regular expression. Before we begin the study of finite automata proper, however, let us consider an elucidating example.

The pattern for identifiers as commonly defined in programming languages is given by the following regular definition (we assume that `letter` and `digit` have been already defined):

```
identifier = letter(letter|digit)*
```

This represents a string that begins with a letter and continues with any sequence of letters and/or digits. The process of recognizing such a string can be described by the diagram of Figure 2.1.

Figure 2.1
A finite automaton for identifiers



In that diagram, the circles numbered 1 and 2 represent **states**, which are locations in the process of recognition that record how much of the pattern has already been seen. The arrowed lines represent **transitions** that record a change from one state to another upon a match of the character or characters by which they are labeled. In the sample diagram, state 1 is the **start state**, or the state at which the recognition process begins. By convention, the start state is indicated by drawing an unlabeled arrowed line to it coming "from nowhere." State 2 represents the point at which a single letter has been matched (indicated by the transition from state 1 to state 2 labeled **`letter`**). Once in state 2, any number of letters and/or digits may be seen, and a match of these returns us to state 2. States that represent the end of the recognition process, in which we can declare success, are called **accepting states**, and are indicated by drawing a double-line border around the state in the diagram. There may be more than one of these. In the sample diagram state 2 is an accepting state, indicating that, after a letter is seen, any subsequent sequence of letters and digits (including none at all) represents a legal identifier.

The process of recognizing an actual character string as an identifier can now be indicated by listing the sequence of states and transitions in the diagram that are used in the recognition process. For example, the process of recognizing **`xtemp`** as an identifier can be indicated as follows:

$$\rightarrow 1 \xrightarrow{\texttt{x}} 2 \xrightarrow{\texttt{t}} 2 \xrightarrow{\texttt{e}} 2 \xrightarrow{\texttt{m}} 2 \xrightarrow{\texttt{p}} 2$$

In this diagram, we have labeled each transition by the letter that is matched at each step.

2.3.1 Definition of Deterministic Finite Automata

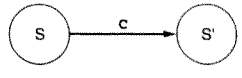
Diagrams such as the one we have discussed are useful descriptions of finite automata, since they allow us to visualize easily the actions of the algorithm. Occasionally, however, it is necessary to have a more formal description of a finite automaton, and so we proceed now to give a mathematical definition. Most of the time, however, we will not need so abstract a view as this, and we will describe most examples in terms of the diagram alone. Other descriptions of finite automata are also possible, particularly tables, and these will be useful for turning the algorithms into working code. We will describe them as the need arises.

We should also note that what we have been describing are **deterministic** finite automata: automata where the next state is uniquely given by the current state and the current input character. A useful generalization of this is the **nondeterministic finite automaton**, which will be studied later on in this section.

Definition

A **DFA** (deterministic finite automaton) M consists of an alphabet Σ, a set of states S, a transition function T: $S \times \Sigma \rightarrow S$, a start state $s_0 \in S$, and a set of accepting states $A \subset S$. The language accepted by M, written $L(M)$, is defined to be the set of strings of characters $c_1c_2 \ldots c_n$ with each $c_i \in \Sigma$ such that there exist states $s_1 = T(s_0, c_1)$, $s_2 = T(s_1, c_2), \ldots, s_n = T(s_{n-1}, c_n)$ with s_n an element of A (i.e., an accepting state).

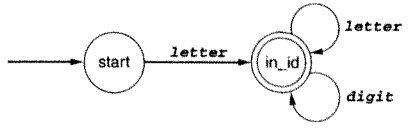
We make the following notes about this definition. $S \times \Sigma$ refers to the Cartesian or cross product of S and Σ: the set of pairs (s, c), where $s \in S$ and $c \in \Sigma$. The function T records the transitions: $T(s, c) = s'$ if there is a transition from state s to state s' labeled by c. The corresponding piece of the diagram for M looks as follows:



Acceptance as the existence of a sequence of states $s_1 = T(s_0, c_1)$, $s_2 = T(s_1, c_2), \ldots, s_n = T(s_{n-1}, c_n)$ with s_n an accepting state thus means the same thing as the diagram

$$\rightarrow s_0 \xrightarrow{c_1} s_1 \xrightarrow{c_2} s_2 \longrightarrow \cdots \longrightarrow s_{n-1} \xrightarrow{c_n} s_n$$

We note a number of differences between the definition of a DFA and the diagram of the identifier example. First, we used numbers for the states in the identifier diagram, while the definition does not restrict the set of states to numbers. Indeed, we can use any system of identification we want for the states, including names. For example, we could write an equivalent diagram to that of Figure 2.1 as



where we have now called the states start (since it is the start state) and in_id (since we have seen a letter and will be recognizing an identifier after any subsequent letters and numbers). The set of states for this diagram now becomes {start, in_id} instead of {1, 2}.

A second difference between the diagram and the definition is that we have not labeled the transitions with characters but with names representing a set of characters.

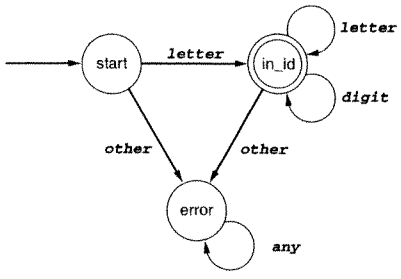
For instance, the name **letter** represents any letter of the alphabet according to the following regular definition:

```
letter = /[a-zA-Z]
```

This is a convenient extension of the definition, since it would be cumbersome to draw 52 separate transitions, one for each lowercase letter and one for each uppercase letter. We will continue to use this extension of the definition in the rest of the chapter.

A third and more essential difference between the definition and our diagram is that the definition represents transitions as a *function* T: $S \times \Sigma \rightarrow S$. This means that $T(s, c)$ must have a value *for every s and c*. But in the diagram we have T(start, c) defined only if c is a letter, and T(in_id, c) is defined only if c is a letter or a digit. Where are the missing transitions? The answer is that they represent errors—that is, in recognizing an identifier we cannot accept any characters other than letters from the start state and letters or numbers after that.[4] The convention is that these **error transitions** are not drawn in the diagram but are simply assumed to always exist. If we were to draw them, the diagram for an identifier would look as in Figure 2.2.

Figure 2.2
A finite automaton for identifiers with error transitions



In that figure, we have labeled the new state error (since it represents an erroneous occurrence), and we have labeled the error transitions **other**. By convention, **other** represents any character not appearing in any other transition from the state where it originates. Thus, the definition of **other** coming from the start state is

```
other = ~letter
```

and the definition of **other** coming from the state in_id is

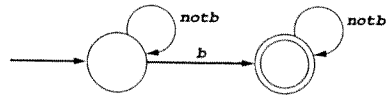
```
other = ~(letter|digit)
```

4. In reality, these nonalphanumeric characters mean either that we do not have an identifier at all (if we are in the start state) or that we have encountered a delimiter that ends the recognition of an identifier (if we are in an accepting state). We will see how to handle these situations later in this section.

Note also that all transitions from the error state go back to itself (we have labeled these transitions **any** to indicate that any character results in this transition). Also, the error state is nonaccepting. Thus, once an error has occurred, we cannot escape from the error state, and we will never accept the string.

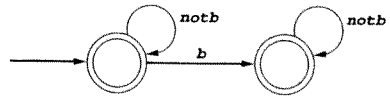
We now turn to a series of examples of DFAs, paralleling the examples of the previous section.

Example 2.6 The set of strings that contain exactly one b is accepted by the following DFA:



Note that we have not bothered to label the states. We will omit labels when it is not necessary to refer to the states by name. §

Example 2.7 The set of strings that contain at most one b is accepted by the following DFA:



Note how this DFA is a modification of the DFA of the previous example, obtained by making the start state into a second accepting state. §

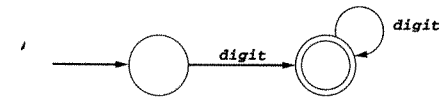
Example 2.8 In the previous section we gave regular definitions for numeric constants in scientific notation as follows:

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

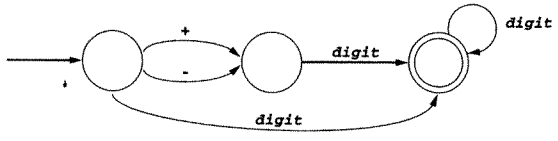
We would like to write down DFAs for the strings matched by these definitions, but it is helpful to first rewrite them as follows:

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

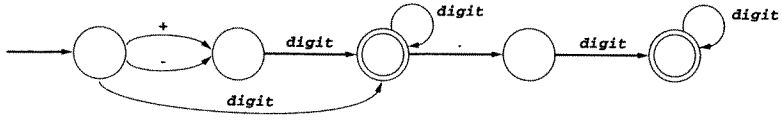
It is easy to write down a DFA for **nat** as follows (recall that **a+ = aa*** for any **a**):



A **signedNat** is a little more difficult because of the optional sign. However, we may note that a **signedNat** begins either with a digit or a sign and a digit and then write the following DFA:



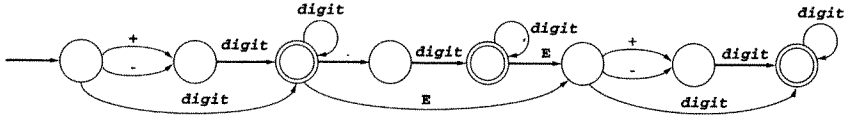
It is also easy to add the optional fractional part, as follows:



Note that we have kept both accepting states, reflecting the fact that the fractional part is optional.

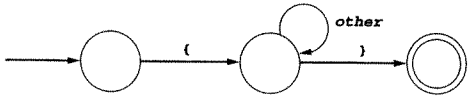
Finally, we need to add the optional exponential part. To do this, we note that the exponential part must begin with the letter *E* and can occur only after we have reached either of the previous accepting states. The final diagram is given in Figure 2.3.

Figure 2.3 A finite automaton for floating-point numbers



§

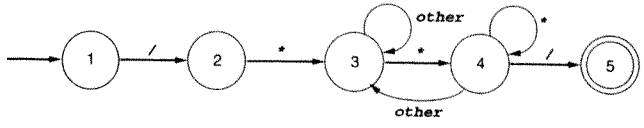
Example 2.9 Unnested comments can be described using DFAs. For example, comments surrounded by curly brackets are accepted by the following DFA:



In this case **other** means all characters except the right curly bracket. This DFA corresponds to the regular expression **{(~})*}**, which we wrote down previously in Section 2.2.4.

We noted in that section that it was difficult to write down a regular expression for comments that are delimited by a sequence of two characters, such as C comments, which are of the form **/*...(no */s)...*/**. It is actually easier to write down a DFA that accepts such comments than it is to write a regular expression for them. A DFA for such C comments is given in Figure 2.4.

Figure 2.4 A finite automaton for C-style comments



In that figure the **other** transition from state 3 to itself stands for all characters except *, while the **other** transition from state 4 to state 3 stands for all characters except * and /. We have numbered the states in this diagram for simplicity, but we could have given the states more meaningful names, such as the following (with the corresponding numbers in parentheses): start (1); entering_comment (2); in_comment (3); exiting_comment (4); and finish (5). §

2.3.2 Lookahead, Backtracking, and Nondeterministic Automata

We have studied DFAs as a way of representing algorithms that accept character strings according to a pattern. As the reader may have already guessed, there is a strong relationship between a regular expression for a pattern and a DFA that accepts strings according to the pattern. We will explore this relationship in the next section. But, first, we need to study more closely the precise algorithms that DFAs represent, since we want eventually to turn these algorithms into the code for a scanner.

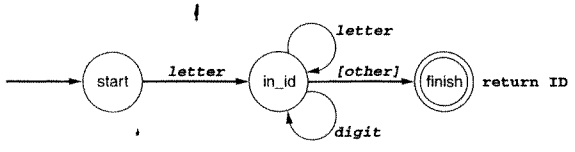
We have already noted that the diagram of a DFA does not represent everything a DFA needs but gives only an outline of its operation. Indeed, we saw that the mathematical definition implies that a DFA must have a transition for every state and character, and that those transitions that result in errors are usually left out of the diagram for the DFA. But even the mathematical definition does not describe every aspect of behavior of a DFA algorithm. For example, it does not specify what happens when an error does occur. It also does not specify the action that a program is to take upon reaching an accepting state, or even when matching a character during a transition.

A typical action that occurs when making a transition is to move the character from the input string to a string that accumulates the characters belonging to a single token (the token string value or lexeme of the token). A typical action when reaching an accepting state is to return the token just recognized, along with any associated attributes. A typical action when reaching an error state is to either back up in the input (backtracking) or to generate an error token.

Our original example of an identifier token exhibits much of the behavior that we wish to describe here, and so we return to the diagram of Figure 2.4. The DFA of that

figure does not exhibit the behavior we want from a scanner for several reasons. First, the error state is not really an error at all, but represents the fact that either an identifier is not to be recognized (if we came from the start state) or a delimiter has been seen and we should now accept and generate an identifier token. Let us assume for the moment (which will in fact be correct behavior) that there are other transitions representing the nonletter transitions from the start state. Then we can indicate that a delimiter has been seen from the state in_id, and that an identifier token should be generated, by the diagram of Figure 2.5:

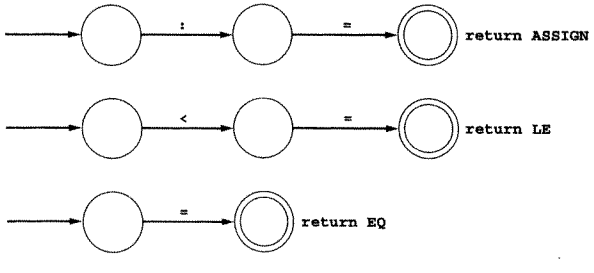
Figure 2.5
Finite automaton for an identifier with delimiter and return value



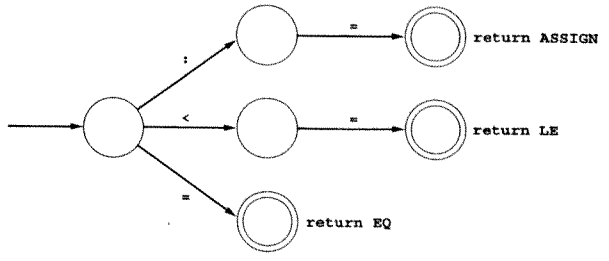
In the diagram we have surrounded the **other** transition with square brackets to indicate that the delimiting character should be considered lookahead, that is, that it should be returned to the input string and not consumed. In addition, the error state has become the accepting state in this diagram and there are no transitions out of the accepting state. This is what we want, since the scanner should recognize one token at a time and should begin again in its start state after each token is recognized.

This new diagram also expresses the principle of longest substring described in Section 2.2.4: the DFA continues to match letters and digits (in state in_id) until a delimiter is found. By contrast the old diagram allowed the DFA to accept at any point while reading an identifier string, something we certainly do not want to happen.

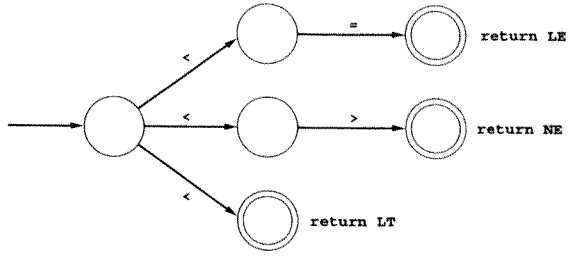
We turn our attention now to the question of how to arrive at the start state in the first place. In a typical programming language there are many tokens, and each token will be recognized by its own DFA. If each of these tokens begins with a different character, then it is easy to tie them together by simply uniting all of their start states into a single start state. For example, consider the tokens given by the strings :=, <=, and =. Each of these is a fixed string, and DFAs for them can be written as follows:



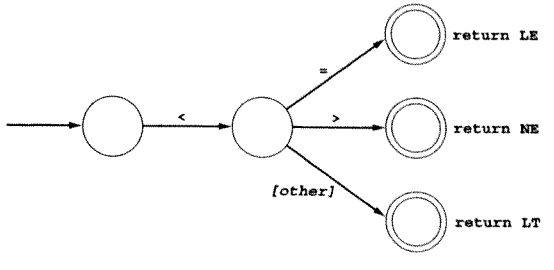
Since each of these tokens begins with a different character, we can simply identify their start states to get the following DFA:



However, suppose we had several tokens that begin with the same character, such as <, <=, and <>. Now we cannot simply write the following diagram, since it is not a DFA (given a state and a character, there must always be a unique transition to a single new state):



Instead, we must arrange it so that there is a unique transition to be made in each state, such as in the following diagram:

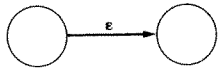


In principle, we should be able to combine all the tokens into one giant DFA in this fashion. However, the complexity of such a task becomes enormous, especially if it is done in an unsystematic way.

A solution to this problem is to expand the definition of a finite automaton to include the case where more than one transition from a state may exist for a particular character, while at the same time developing an algorithm for systematically turning these new, generalized finite automata into DFAs. We will describe these generalized automata here, while postponing the description of the translation algorithm until the next section.

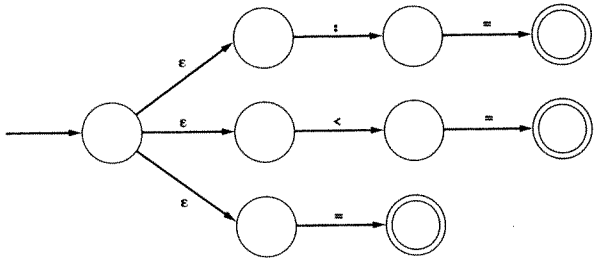
The new kind of finite automaton is called a **nondeterministic finite automaton**, or **NFA** for short. Before we define it, we need one more generalization that will be useful in applying finite automata to scanners: the concept of the ε-transition.

An **ε-transition** is a transition that may occur without consulting the input string (and without consuming any characters). It may be viewed as a "match" of the empty string, which we have previously written as ε. In a diagram an ε-transition is written as though ε were actually a character:

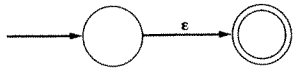


This should not be confused with a match of the character ε in the input: if the alphabet includes such a character, it must be distinguished from the use of ε as a metacharacter to represent an ε-transition.

ε-transitions are somewhat counterintuitive, since they may occur "spontaneously," that is, without lookahead and without change to the input string, but they are useful in two ways. First, they can express a choice of alternatives in a way that does not involve combining states. For example, the choice of the tokens :=, <=, and = can be expressed by combining the automata for each token as follows:



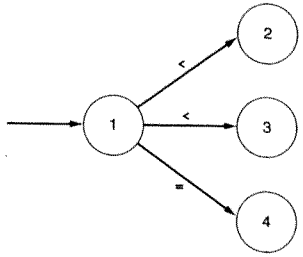
This has the advantage of keeping the original automata intact and only adding a new start state to connect them. The second advantage to ε-transitions is that they can explicitly describe a match of the empty string:



Of course, this is equivalent to the following DFA, which expresses that acceptance should occur without matching any characters:

But it is useful to have the previous, explicit notation.

We now proceed to a definition of a nondeterministic automaton. It is quite similar to that of a DFA, except that, according to the above discussion, we need to expand the alphabet Σ to include ε. We do this by writing $\Sigma \cup \{\varepsilon\}$ (the union of Σ and ε) where we used Σ before (this assumes that ε is not originally a member of Σ). We also need to expand the definition of T (the transition function) so that each character can lead to more than one state. We do this by letting the value of T be a *set* of states rather than a single state. For example, given the diagram



we have $T(1, <) = \{2, 3\}$. In other words, from state 1 we can move to either state 2 or state 3 on the input character <, and T becomes a function that maps state/symbol pairs to *sets of states*. Thus, the range of T is the **power set** of the set S of states (the set of all subsets of S); we write this as $\wp(S)$ (script p of S). We now state the definition.

Definition

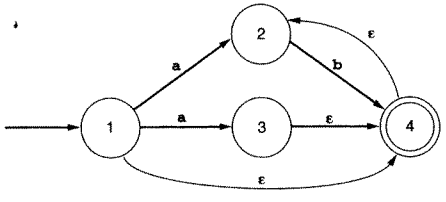
An **NFA** (nondeterministic finite automaton) M consists of an alphabet Σ, a set of states S, a transition function T: $S \times (\Sigma \cup \{\varepsilon\}) \rightarrow \wp(S)$, a start state s_0 from S, and a set of accepting states A from S. The language accepted by M, written $L(M)$, is defined to be the set of strings of characters $c_1c_2 \ldots c_n$ with each c_i from $\Sigma \cup \{\varepsilon\}$ such that there exist states s_1 in $T(s_0, c_1)$, s_2 in $T(s_1, c_2)$, . . . , s_n in $T(s_{n-1}, c_n)$ with s_n an element of A.

Again, we need to note a few things about this definition. Any of the c_i in $c_1c_2 \ldots c_n$ may be ε, and the string that is actually accepted is the string $c_1c_2 \ldots c_n$ with the ε's removed (since the concatenation of s with ε is s itself). Thus, the string

$c_1c_2 \ldots c_n$ may actually have fewer than n characters in it. Also, the sequence of states $s_1, \ldots, s_n$ are chosen from the *sets* of states $T(s_0, c_1), \ldots, T(s_{n-1}, c_n)$, and this choice will not always be uniquely determined. This is, in fact, why these automata are called *nondeterministic*: the sequence of transitions that accepts a particular string is not determined at each step by the state and the next input character. Indeed, arbitrary numbers of ε's can be introduced into the string at any point, corresponding to any number of ε-transitions in the NFA. Thus, an NFA does not represent an algorithm. However, it can be simulated by an algorithm that backtracks through every nondeterministic choice, as we will see later in this section.

First, however, we consider a couple of examples of NFAs.

Example 2.10 Consider the following diagram of an NFA.

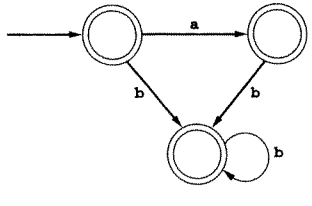


The string **abb** can be accepted by either of the following sequences of transitions:

$$\rightarrow 1 \xrightarrow{a} 2 \xrightarrow{b} 4 \xrightarrow{\varepsilon} 2 \xrightarrow{b} 4$$

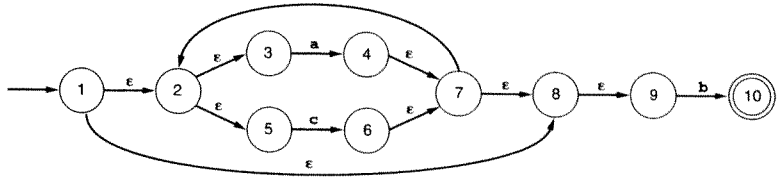
$$\rightarrow 1 \xrightarrow{a} 3 \xrightarrow{\varepsilon} 4 \xrightarrow{\varepsilon} 2 \xrightarrow{b} 4 \xrightarrow{\varepsilon} 2 \xrightarrow{b} 4$$

Indeed the transitions from state 1 to state 2 on *a*, and from state 2 to state 4 on *b*, allow the machine to accept the string *ab*, and then, using the ε-transition from state 4 to state 2, all strings matching the regular expression **ab+**. Similarly, the transitions from state 1 to state 3 on *a*, and from state 3 to state 4 on ε, enable the acceptance of all strings matching **ab***. Finally, following the ε-transition from state 1 to state 4 enables the acceptance of all strings matching **b***. Thus, this NFA accepts the same language as the regular expression **ab+|ab*|b***. A simpler regular expression that generates the same language is **(a|ε)b***. The following DFA also accepts this language:

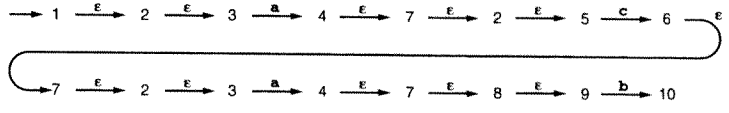


§

Example 2.11 Consider the following NFA:



It accepts the string *acab* by making the following transitions:

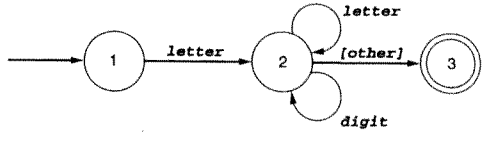


In fact, it is not hard to see that this NFA accepts the same language as that generated by the regular expression **(a|c)*b**. §

2.3.3 Implementation of Finite Automata in Code

There are several ways to translate either a DFA or an NFA into code, and we will survey these in this section. Not all these methods will be useful for a compiler scanner, however, and the last two sections of this chapter will demonstrate the coding aspects appropriate for scanners in more detail.

Consider, again, our original example of a DFA that accepts identifiers consisting of a letter followed by a sequence of letters and/or digits, in its amended form that includes lookahead and the principle of longest substring (see Figure 2.5):



The first and easiest way to simulate this DFA is to write code in the following form:

```
{ starting in state 1 }
if the next character is a letter then
  advance the input;
  { now in state 2 }
  while the next character is a letter or a digit do
    advance the input; { stay in state 2 }
```

(continued)

```
    end while;
    { go to state 3 without advancing the input }
    accept;
  else
    { error or other cases }
  end if;
```

Such code uses the position in the code (nested within tests) to maintain the state implicitly, as we have indicated by the comments. This is reasonable if there are not too many states (requiring many levels of nesting), and if loops in the DFA are small. Code like this has been used to write small scanners. But there are two drawbacks to this method. The first is that it is ad hoc—that is, each DFA has to be treated slightly differently, and it is difficult to state an algorithm that will translate every DFA to code in this way. The second is that the complexity of the code increases dramatically as the number of states rises or, more specifically, as the number of different states along arbitrary paths rises. As one simple example of these problems, we consider the DFA from Example 2.9 as given in Figure 2.4 (page 53) that accepts C comments, which could be implemented by code in the following form:

```
{ state 1 }
if the next character is "/" then
  advance the input; { state 2 }
  if the next character is "*" then
    advance the input; { state 3 }
    done := false;
    while not done do
      while the next input character is not "*" do
        advance the input;
      end while;
      advance the input; { state 4 }
      while the next input character is "*" do
        advance the input;
      end while;
      if the next input character is "/" then
        done := true;
      end if;
      advance the input;
    end while;
    accept; { state 5 }
  else { other processing }
  end if;
else { other processing }
end if;
```

Notice the considerable increase in complexity, and the need to deal with the loop involving states 3 and 4 by using the Boolean variable *done*.

A substantially better implementation method is obtained by using a variable to maintain the current state and writing the transitions as a doubly nested case statement inside a loop, where the first case statement tests the current state and the nested second level tests the input character, given the state. For example, the previous DFA for identifiers can be translated into the code scheme of Figure 2.6.

Figure 2.6
Implementation of identifier DFA using a state variable and nested case tests.

```
state := 1; { start }
while state = 1 or 2 do
  case state of
  1: case input character of
     letter : advance the input;
              state := 2;
     else state := ... { error or other };
     end case;
  2: case input character of
     letter, digit: advance the input;
                    state := 2; { actually unnecessary }
     else state := 3;
     end case;
  end case;
end while;
if state = 3 then accept else error ;
```

Notice how this code reflects the DFA directly: transitions correspond to assigning a new state to the *state* variable and advancing the input (except in the case of the "non-consuming" transition from state 2 to state 3).

Now the DFA for C comments (Figure 2.4) can be translated into the more readable code scheme of Figure 2.7. An alternative to this organization is to have the outer case based on the input character and the inner cases based on the current state (see the exercises).

In the examples we have just seen, the DFA has been "hardwired" right into the code. It is also possible to express the DFA as a data structure and then write "generic" code that will take its actions from the data structure. A simple data structure that is adequate for this purpose is a **transition table**, or two-dimensional array, indexed by state and input character that expresses the values of the transition function T:

	Characters in the alphabet c
States s	States representing transitions $T(s, c)$

As an example, the DFA for identifiers can be represented as the following transition table:

state \ input char	letter	digit	other
1	2		
2	2	2	3
3			

Figure 2.7
Implementation of DFA of Figure 2.4

```
state := 1; { start }
while state = 1, 2, 3 or 4 do
 case state of
 1: case input character of
     "/" : advance the input;
          state := 2;
     else state := . . . { error or other };
     end case;
 2: case input character of
     "*": advance the input;
          state := 3;
     else state := . . . { error or other };
     end case;
 3: case input character of
     "*": advance the input;
          state := 4;
     else advance the input { and stay in state 3 };
     end case;
 4: case input character of
     "/" advance the input;
          state := 5;
     "*": advance the input; { and stay in state 4 }
     else advance the input;
          state := 3;
     end case;
 end case;
end while;
if state = 5 then accept else error ;
```

In this table, blank entries represent transitions that are not shown in the DFA diagram (i.e., they represent transitions to error states or other processing). We also assume that the first state listed is the start state. However, this table does not indicate which states are accepting and which transitions do not consume their inputs. This information can be kept either in the same data structure representing the table or in a separate data structure. If we add this information to the above transition table (using a separate column to indicate accepting states and brackets to indicate "noninput-consuming" transitions), we obtain the following table:

state \ input char	letter	digit	other	*Accepting*
1	2			no
2	2	2	[3]	no
3				yes

As a second example of a transition table, we present the table for the DFA for C comments (our second example in the foregoing):

state \ input char	/	*	other	*Accepting*
1	2			no
2		3		no
3	3	4	3	no
4	5	4	3	no
5				yes

Now we can write code in a form that will implement any DFA, given the appropriate data structures and entries. The following code schema assumes that the transitions are kept in a transition array *T* indexed by states and input characters; that transitions that advance the input (i.e., those not marked with brackets in the table) are given by the Boolean array *Advance*, indexed also by states and input characters; and that accepting states are given by the Boolean array *Accept*, indexed by states. Here is the code scheme:

```
state := 1;
ch := next input character;
while not Accept[state] and not error(state) do
  newstate := T[state,ch];
  if Advance[state,ch] then ch := next input char;
  state := newstate;
end while;
if Accept[state] then accept;
```

Algorithmic methods such as we have just described are called **table driven**, since they use tables to direct the progress of the algorithm. Table-driven methods have certain advantages: the size of the code is reduced, the same code will work for many different problems, and the code is easier to change (maintain). The disadvantage is that the tables can become very large, causing a significant increase in the space used by the program. Indeed, much of the space in the arrays we have just described is wasted. Table-driven methods, therefore, often rely on table-compression methods such as sparse-array representations, although there is usually a time penalty to be paid for such compression, since table lookup becomes slower. Since scanners must be efficient, these methods are rarely used for them, though they may be used in scanner generator programs such as Lex. We will not study them further here.

Finally, we note that NFAs can be implemented in similar ways to DFAs, except that since NFAs are nondeterministic, there are potentially many different sequences of transitions that must be tried. Thus, a program that simulates an NFA must store up transitions that have not yet been tried and backtrack to them on failure. This is very similar to algorithms that attempt to find paths in directed graphs, except that the input string guides the search. Since algorithms that do a lot of backtracking tend to be inef-

ficient, and a scanner must be as efficient as possible, we will not describe such algorithms further. Instead, the problem of simulating an NFA can be solved by using the method we study in the next section that converts an NFA into a DFA. We thus proceed to that section, where we will return briefly to the question of simulating an NFA.

2.4 FROM REGULAR EXPRESSIONS TO DFAs

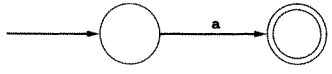
In this section we will study an algorithm for translating a regular expression into a DFA. There also exists an algorithm for translating a DFA into a regular expression, so that the two notions are equivalent. However, because of the compactness of regular expressions, they are usually preferred to DFAs as token descriptions, and so scanner generation commonly begins with regular expressions and proceeds through the construction of a DFA to a final scanner program. For this reason, our interest will be only in an algorithm that performs this direction of the equivalence.

The simplest algorithm for translating a regular expression into a DFA proceeds via an intermediate construction, in which an NFA is derived from the regular expression, and then the NFA is used to construct an equivalent DFA. There exist algorithms that can translate a regular expression directly into a DFA, but they are more complex, and the intermediate construction is also of some interest. Thus, we concentrate on describing two algorithms, one that translates a regular expression into an NFA and a second that translates an NFA into a DFA. Combined with one of the algorithms to translate a DFA into a program described in the previous section, the process of constructing a scanner can be automated in three steps, as illustrated by the following picture:

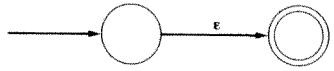
2.4.1 From a Regular Expression to an NFA

The construction we will describe is known as **Thompson's construction**, after its inventor. It uses ε-transitions to "glue together" the machines of each piece of a regular expression to form a machine that corresponds to the whole expression. Thus, the construction is inductive, and it follows the structure of the definition of a regular expression: we exhibit an NFA for each basic regular expression and then show how each regular expression operation can be achieved by connecting together the NFAs of the subexpressions (assuming these have already been constructed).

Basic Regular Expressions A basic regular expression is of the form **a**, ε, or ϕ, where **a** represents a match of a single character from the alphabet, ε represents a match of the empty string, and ϕ represents a match of no strings at all. An NFA that is equivalent to the regular expression **a** (i.e., accepts precisely those strings in its language) is



Similarly, an NFA that is equivalent to ε is

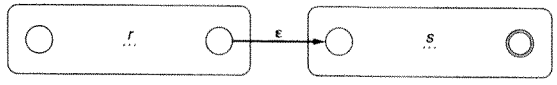


The case of the regular expression ϕ (which never occurs in practice in a compiler) is left as an exercise.

Concatenation We wish to construct an NFA equivalent to the regular expression rs, where r and s are regular expressions. We assume (inductively) that NFAs equivalent to r and s have already been constructed. We express this by writing

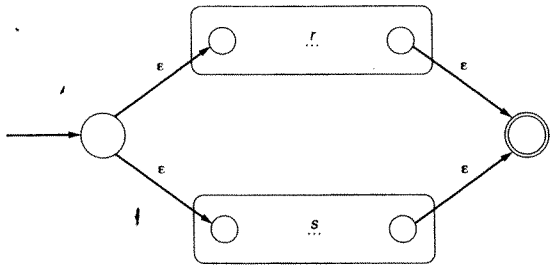
for the NFA corresponding to r, and similarly for s. In this drawing, the circle on the left inside the rounded rectangle indicates the start state, the double circle on the right indicates the accepting state, and the three dots indicate the states and transitions inside the NFA that are not shown. This picture assumes that the NFA corresponding to r has only one accepting state. This assumption will be justified if every NFA we construct has one accepting state. This is true for the NFAs of basic regular expressions, and it will be true for each of the following constructions.

We can now construct an NFA corresponding to rs as follows:



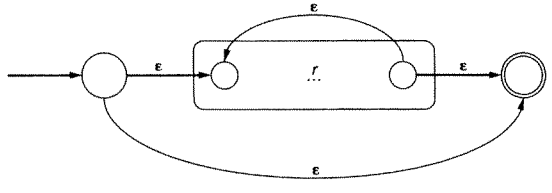
We have connected the accepting state of the machine of r to the start state of the machine of s by an ε-transition. The new machine has the start state of the machine of r as its start state and the accepting state of the machine of s as its accepting state. Clearly, this machine accepts $L(rs) = L(r)L(s)$ and so corresponds to the regular expression rs.

Choice Among Alternatives We wish to construct an NFA corresponding to $r|s$ under the same assumptions as before. We do this as follows:



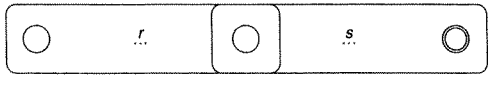
We have added a new start state and a new accepting state and connected them as shown using ε-transitions. Clearly, this machine accepts the language $L(r|s) = L(r) \cup L(s)$.

Repetition We want to construct a machine that corresponds to $r*$, given a machine that corresponds to r. We do this as follows:



Here again we have added two new states, a start state and an accepting state. The repetition in this machine is afforded by the new ε-transition from the accepting state of the machine of r to its start state. This permits the machine of r to be traversed one or more times. To ensure that the empty string is also accepted (corresponding to zero repetitions of r), we must also draw an ε-transition from the new start state to the new accepting state.

This completes the description of Thompson's construction. We note that this construction is not unique. In particular, other constructions are possible when translating regular expression operations into NFAs. For example, in expressing concatenation rs, we could have eliminated the ε-transition between the machines of r and s and instead identified the accepting state of the machine of r with the start state of the machine of s, as follows:

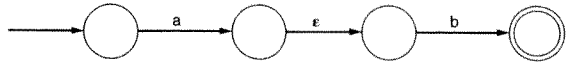


(This simplification depends, however, on the fact that in the other constructions, the accepting state has no transitions from it to other states—see the exercises.) Other simplifications are possible in the other cases. The reason we have expressed the translations as we have is that the machines are constructed according to very simple rules. First, each state has at most two transitions from it, and if there are two transitions, they must both be ε-transitions. Second, no states are deleted once they are constructed, and no transitions are changed except for the addition of transitions from the accepting state. These properties make it very easy to automate the process.

We conclude the discussion of Thompson's construction with a few examples.

Example 2.12 We translate the regular expression **ab|a** into an NFA according to Thompson's construction. We first form the machines for the basic regular expressions **a** and **b**:

We then form the machine for the concatenation **ab**:



Now we form another copy of the machine for **a** and use the construction for choice to get the complete NFA for **ab|a**, which is shown in Figure 2.8.

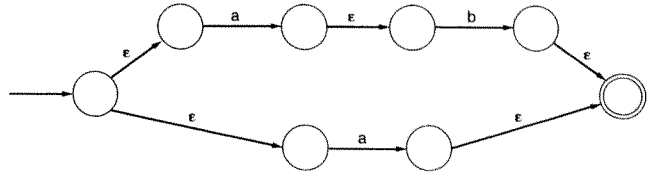
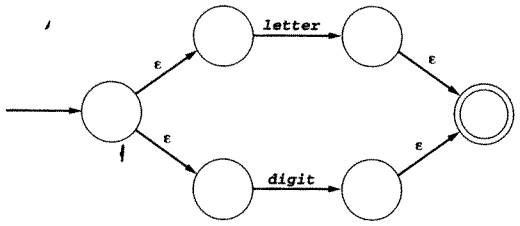


Figure 2.8
NFA for the regular expression **ab|a** using Thompson's construction

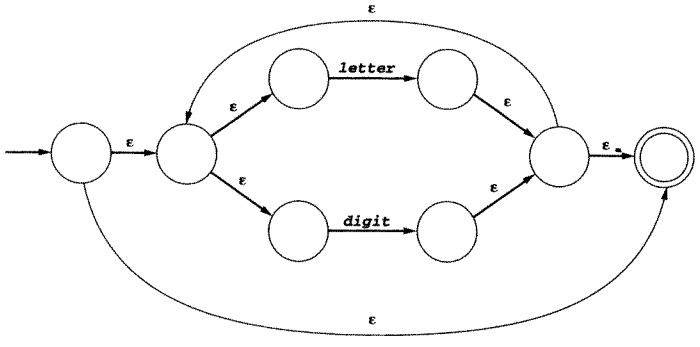
§

Example 2.13 We form the NFA of Thompson's construction for the regular expression ***letter*(*letter*|*digit*)***. As in the previous example, we form the machines for the regular expressions ***letter*** and ***digit***:

We then form the machine for the choice ***letter*|*digit***:

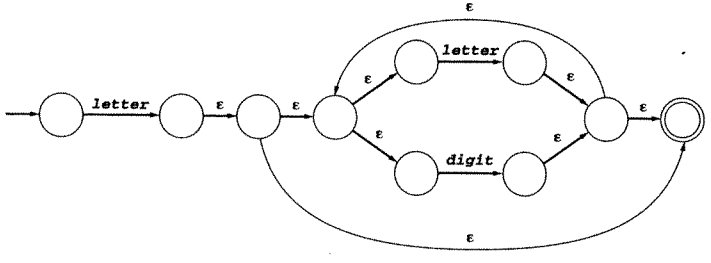


Now we form the NFA for the repetition **(*letter*|*digit*)*** as follows:



Finally, we construct the machine for the concatenation of ***letter*** with **(*letter*|*digit*)*** to get the complete NFA, as shown in Figure 2.9.

Figure 2.9
NFA for the regular expression `letter(letter|digit)*` using Thompson's construction



§

As a final example, we note that the NFA of Example 2.11 (Section 2.3.2) is exactly that corresponding to the regular expression `(a|c)*b` under Thompson's construction.

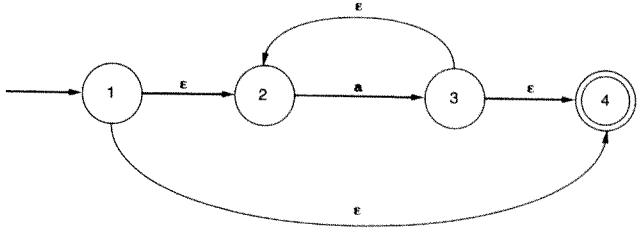
2.4.2 From an NFA to a DFA

We now wish to describe an algorithm that, given an arbitrary NFA, will construct an equivalent DFA (i.e., one that accepts precisely the same strings). To do this we will need some method for eliminating both ε-transitions and multiple transitions from a state on a single input character. Eliminating ε-transitions involves the construction of ε-closures, an **ε-closure** being the set of all states reachable by ε-transitions from a state or states. Eliminating multiple transitions on a single input character involves keeping track of the set of states that are reachable by matching a single character. Both these processes lead us to consider sets of states instead of single states. Thus, it is not surprising that the DFA we construct has as its states *sets of states* of the original NFA. Thus, this algorithm is called the **subset construction**. We first discuss the ε-closure in a little more detail and then proceed to a description of the subset construction.

The ε-Closure of a Set of States We define the ε-closure of a single state s as the set of states reachable by a series of zero or more ε-transitions, and we write this set as $\bar{s}$. We leave a more mathematical statement of this definition to an exercise and proceed directly to an example. Note, however, that the ε-closure of a state always contains the state itself.

Example 2.14

Consider the following NFA corresponding to the regular expression `a*` under Thompson's construction:



In this NFA, we have $\bar{1} = \{1, 2, 4\}$, $\bar{2} = \{2\}$, $\bar{3} = \{2, 3, 4\}$, and $\bar{4} = \{4\}$. §

We now define the ε-closure of a set of states to be the union of the ε-closures of each individual state. In symbols, if S is a set of states, then we have

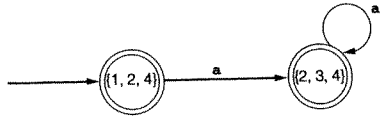
$$\bar{S} = \bigcup_{s \text{ in } S} \bar{s}$$

For instanee, in the NFA of Example 2.14, $\overline{\{1, 3\}} = \overline{1} \cup \overline{3} = \{1, 2, 4\} \cup \{2, 3, 4\} = \{1, 2, 3, 4\}$.

The Subset Construction We are now in a position to describe the algorithm for constructing a DFA from a given NFA M, which we will call $\overline{M}$. We first compute the ε-closure of the start state of M; this becomes the start state of $\overline{M}$. For this set, and for each subsequent set, we compute transitions on characters a as follows. Given a set S of states and a character a in the alphabet, compute the set $S'_a = \{\ t \mid \text{for some } s \text{ in } S \text{ there is a transition from } s \text{ to } t \text{ on } a\ \}$. Then, compute $\overline{S'_a}$, the ε-closure of S'_a. This defines a new state in the subset construction, together with a new transition $S \xrightarrow{a} \overline{S'_a}$. Continue with this process until no new states or transitions are created. Mark as accepting those states constructed in this manner that contain an accepting state of M. This is the DFA $\overline{M}$. It contains no ε-transitions because every state is constructed as an ε-closure. It contains at most one transition from a state on a character a because each new state is constructed from *all* states of M reachable by transitions from a state on a single character a.

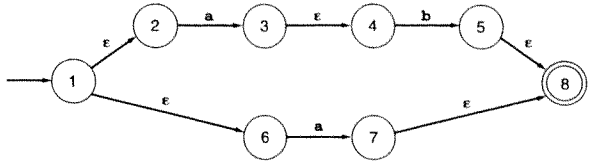
We illustrate the subset construction with a number of examples.

Example 2.15 Consider the NFA of Example 2.14. The start state of the corresponding DFA is $\overline{1} = \{1, 2, 4\}$. There is a transition from state 2 to state 3 on a, and no transitions from states 1 or 4 on a, so there is a transition on a from $\{1, 2, 4\}$ to $\overline{\{1, 2, 4\}_a} = \overline{\{3\}} = \{2, 3, 4\}$. Since there are no further transitions on a character from any of the states 1, 2, or 4, we turn our attention to the new state $\{2, 3, 4\}$. Again, there is a transition from 2 to 3 on a and no a-transitions from either 3 or 4, so there is a transition from $\{2, 3, 4\}$ to $\overline{\{2, 3, 4\}_a} = \overline{\{3\}} = \{2, 3, 4\}$. Thus, there is an a-transition from $\{2, 3, 4\}$ to itself. We have run out of states to consider, and so we have constructed the entire DFA. It only remains to note that state 4 of the NFA is accepting, and since both $\{1, 2, 4\}$ and $\{2, 3, 4\}$ contain 4, they are both accepting states of the corresponding DFA. We draw the DFA we have constructed as follows, where we name the states by their subsets:

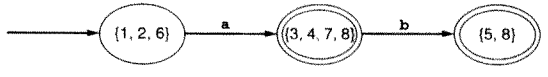


(Once the construction is complete, we could discard the subset terminology if we wished.) §

Example 2.16 Consider the NFA of Figure 2.8, to which we add state numbers:

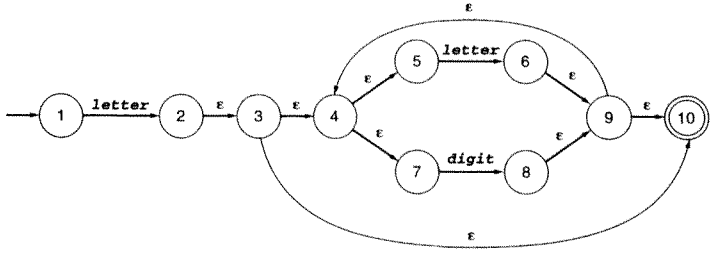


The DFA subset construction has as its start state $\overline{\{1\}} = \{1, 2, 6\}$. There is a transition on a from state 2 to state 3, and also from state 6 to state 7. Thus, $\overline{\{1, 2, 6\}_a} = \overline{\{3, 7\}} = \{3, 4, 7, 8\}$, and we have $\{1, 2, 6\} \xrightarrow{a} \{3, 4, 7, 8\}$. Since there are no other character transitions from 1, 2, or 6, we go on to $\{3, 4, 7, 8\}$. There is a transition on b from 4 to 5 and $\overline{\{3, 4, 7, 8\}_b} = \overline{\{5\}} = \{5, 8\}$, and we have the transition $\{3, 4, 7, 8\} \xrightarrow{b} \{5, 8\}$. There are no other transitions. Thus, the subset construction yields the following DFA equivalent to the previous NFA:

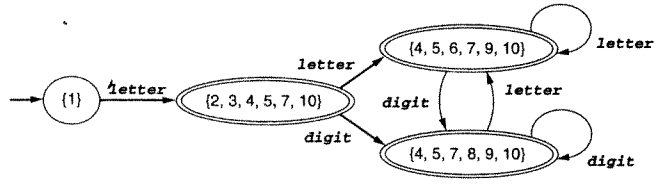


§

Example 2.17 Consider the NFA of Figure 2.9 (Thompson's construction for the regular expression **`letter(letter|digit)*`**):



The subset construction proceeds as follows. The start state is $\overline{\{1\}} = \{1\}$. There is a transition on ***letter*** to $\overline{\{2\}} = \{2, 3, 4, 5, 7, 10\}$. From this state there is a transition on ***letter*** to $\overline{\{6\}} = \{4, 5, 6, 7, 9, 10\}$ and a transition on ***digit*** to $\overline{\{8\}} = \{4, 5, 7, 8, 9, 10\}$. Finally, each of these states also has transitions on ***letter*** and ***digit***, either to itself or to the other. The complete DFA is given in the following picture:



2.4.3 Simulating an NFA Using the Subset Construction

In the last section we briefly discussed the possibility of writing a program to simulate an NFA, a question that requires dealing with the nondeterminacy, or nonalgorithmic nature, of the machine. One way of simulating an NFA is to use the subset construction, but instead of constructing all the states of the associated DFA, we construct only the state at each point that is indicated by the next input character. Thus, we construct only those sets of states that will actually occur in a path through the DFA that is taken on the given input string. The advantage to this is that we may not need to construct the entire DFA. The disadvantage is that a state may be constructed many times, if the path contains loops.

For instance, in Example 2.16, if we have the input string consisting of the single character *a*, we will construct the start state {1, 2, 6} and then the second state {3, 4, 7, 8} to which we move and match the *a*. Then, since there is no following *b*, we accept without ever generating the state {5, 8}.

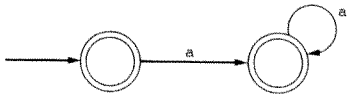
On the other hand, in Example 2.17, given the input string *r2d2*, we have the following sequence of states and transitions:

$$\{1\} \xrightarrow{r} \{2, 3, 4, 5, 7, 10\} \xrightarrow{2} \{4, 5, 7, 8, 9, 10\}$$
$$\xrightarrow{d} \{4, 5, 6, 7, 9, 10\} \xrightarrow{2} \{4, 5, 7, 8, 9, 10\}$$

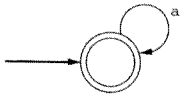
If these states are constructed as the transitions occur, then all the states of the DFA have been constructed and the state {4, 5, 7, 8, 9, 10} has even been constructed twice. Thus, this process is less efficient than constructing the entire DFA in the first place. For this reason, simulation of NFAs is not done in scanners. It does remain an option for pattern matching in editors and search programs, where regular expressions can be given dynamically by the user.

2.4.4 Minimizing the Number of States in a DFA

The process we have described of deriving a DFA algorithmically from a regular expression has the unfortunate property that the resulting DFA may be more complex than necessary. For instance, in Example 2.15 we derived the DFA



for the regular expression **a***, whereas the DFA



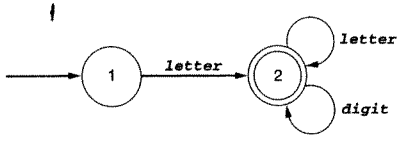
will do as well. Since efficiency is extremely important in a scanner, we would like to be able to construct, if possible, a DFA that is minimal in some sense. In fact, an important result from automata theory states that, given any DFA, there is an equivalent DFA containing a minimum number of states, and that this minimum-state DFA is unique (except for renaming of states). It is also possible to directly obtain this minimum-state DFA from any given DFA, and we will briefly describe the algorithm here, without proof that it does indeed construct the minimum-state equivalent DFA (it should be easy for the reader to be informally convinced of this by reading the algorithm).

The algorithm proceeds by creating sets of states to be unified into single states. It begins with the most optimistic assumption possible: it creates two sets, one consisting of all the accepting states and the other consisting of all the nonaccepting states. Given this partition of the states of the original DFA, consider the transitions on each character *a* of the alphabet. If all accepting states have transitions on *a* to accepting states, then this defines an *a*-transition from the new accepting state (the set of all the old accepting states) to itself. Similarly, if all accepting states have transitions on *a* to nonaccepting states, then this defines an *a*-transition from the new accepting state to the new nonaccepting state (the set of all the old nonaccepting states). On the other hand, if there are two accepting states *s* and *t* that have transitions on *a* that land in different sets, then no *a*-transition can be defined for this grouping of the states. We say that *a* **distinguishes** the states *s* and *t*. In this case, the set of states under consideration (i.e., the set of all accepting states) must be split according to where their *a*-transitions land. Similar statements hold, of course, for each of the other sets of states, and once we have considered all characters of the alphabet, we must move on to them. Of course, if any further sets are split, we must return and repeat the process from the beginning. We continue this process of refining the partition of the states of the original DFA into sets until either all sets contain only one element (in which case, we have shown the original DFA to be minimal) or until no further splitting of sets occurs.

For the process we have just described to work correctly, we must also consider error transitions to an error state that is nonaccepting. That is, if there are accepting states *s* and *t* such that *s* has an *a*-transition to another accepting state, while *t* has no *a*-transition at all (i.e., an error transition), then *a* distinguishes *s* and *t*. Similarly, if a nonaccepting state *s* has an *a*-transition to an accepting state, while another nonaccepting state *t* has no *a*-transition, then *a* distinguishes *s* and *t* in this case too.

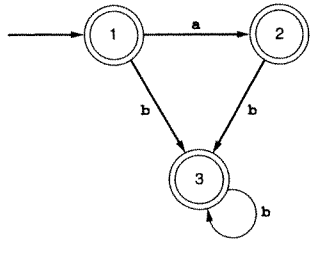
We conclude our discussion of state minimization with a couple of examples.

Example 2.18 Consider the DFA we constructed in the previous example, corresponding to the regular expression **letter(letter|digit)***. It had four states consisting of the start state and three accepting states. All three accepting states have transitions to other accepting states on both **letter** and **digit** and no other (nonerror) transitions. Thus, the three accepting states cannot be distinguished by any character, and the minimization algorithm results in combining the three accepting states into one, leaving the following minimum-state DFA (which we have already seen at the beginning of Section 2.3):

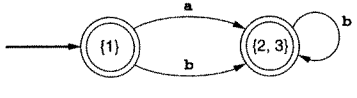


§

Example 2.19 Consider the following DFA, which we gave in Example 2.1 (Section 2.3.2) as equivalent to the regular expression **(a|ε)b***:



In this case, all the states (except the error state) are accepting. Consider now the character *b*. Each accepting state has a *b*-transition to another accepting state, so none of the states are distinguished by *b*. On the other hand, state 1 has an *a*-transition to an accepting state, while states 2 and 3 have no *a*-transition (or, rather, an error transition on *a* to the error nonaccepting state). Thus, *a* distinguishes state 1 from states 2 and 3, and we must repartition the states into the sets {1} and {2, 3}. Now we begin over. The set {1} cannot be split further, so we no longer consider it. Now the states 2 and 3 cannot be distinguished by either *a* or *b*. Thus, we obtain the minimum-state DFA:



§

2.5 IMPLEMENTATION OF A TINY SCANNER

We want now to develop the actual code for a scanner to illustrate the concepts studied so far in this chapter. We do this for the TINY language that we introduced informally in Chapter 1 (Section 1.7). We then discuss a number of practical implementation issues raised by this scanner.

2.5.1 Implementing a Scanner for the Sample Language TINY

In Chapter 1 we gave only the briefest informal introduction to the TINY language. Our task here is to specify completely the lexical structure of TINY, that is, to define the tokens and their attributes. The tokens and token classes of TINY are summarized in Table 2.1.

The tokens of TINY fall into three typical categories: reserved words, special symbols, and "other" tokens. There are eight reserved words, with familiar meanings (though we do not need to know their semantics until much later). There are 10 special symbols, giving the four basic arithmetic operations on integers, two comparison operations (equal and less than), parentheses, semicolon, and assignment. All special symbols are one character long, except for assignment, which is two.

Table 2.1
Tokens of the TINY language

Reserved Words	Special Symbols	Other
if	**+**	***number***
then	**-**	(1 or more
else	*****	digits)
end	**/**	
repeat	**=**	
until	**<**	***identifier***
read	**(**	(1 or more
write	**)**	letters)
	;	
	:=	

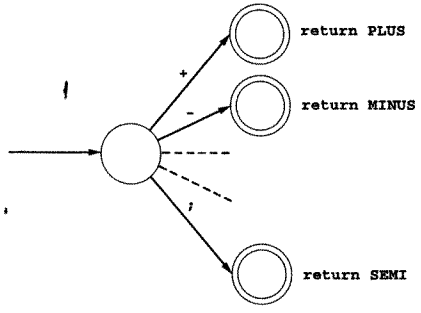
The other tokens are numbers, which are sequences of one or more digits, and identifiers, which (for simplicity) are sequences of one or more letters.

In addition to the tokens, TINY has the following lexical conventions. Comments are enclosed in curly brackets {...} and cannot be nested; the code is free format; white space consists of blanks, tabs, and newlines; and the principle of longest substring is followed in recognizing tokens.

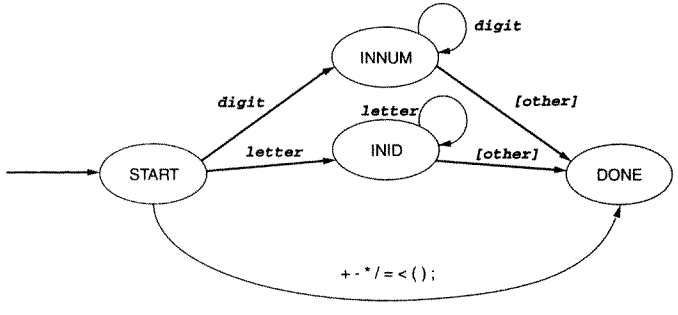
In designing a scanner for this language, we could begin with regular expressions and develop NFAs and DFAs according to the algorithms of the previous section. Indeed regular expressions have been given previously for numbers, identifiers, and comments (TINY has particularly simple versions of these). Regular expressions for the other tokens are trivial, since they are all fixed strings. Instead of following this route,

we will develop a DFA for the scanner directly, since the tokens are so simple. We do this in several steps.

First, we note that all the special symbols except assignment are distinct single characters, and a DFA for these symbols would look as follows:



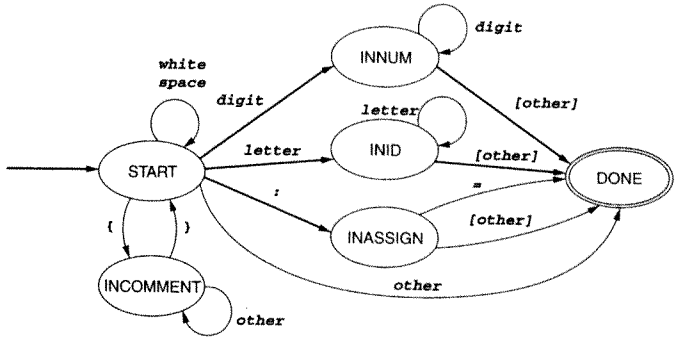
In this diagram, the different accepting states distinguish the token that is to be returned by the scanner. If we use some other indicator for the token to be returned (a variable in the code, say), then all the accepting states can be collapsed into one state that we will call **DONE**. If we combine this two-state DFA with DFAs that accept numbers and identifiers, we obtain the following DFA:



Note the use of the square brackets to indicate lookahead characters that should not be consumed.

We need to add comments, white space, and assignment to this DFA. White space is consumed by a simple loop from the start state to itself. Comments require an extra state, reached from the start state on left curly bracket and returning to it on right curly bracket. Assignment also requires an intermediate state, reached from the start state on semicolon. If an equal sign immediately follows, then an assignment token is generated. Otherwise, the next character should not be consumed, and an error token is generated.

Figure 2.10
DFA of the TINY scanner



In fact, all single characters that are not in the list of special symbols, are not white space or comments, and are not digits or letters, should be accepted as errors, and we lump these in with the single-character symbols. The final DFA for our scanner is given in Figure 2.10.

We have not included reserved words in our discussion or in the DFA of Figure 2.10. This is because it is easiest from the point of view of the DFA to consider reserved words to be the same as identifiers, and then to look up the identifiers in a table of reserved words after acceptance. Indeed, the principle of the longest substring guarantees that the only action of the scanner that needs changing is the token that is returned. Thus, reserved words are considered only after an identifier has been recognized.

We turn now to a discussion of the code to implement this DFA, which is contained in the **scan.h** and **scan.c** files (see Appendix B). The principal procedure is **getToken** (lines 674–793), which consumes input characters and returns the next token recognized according to the DFA of Figure 2.10. The implementation uses the doubly nested case analysis we have described in Section 2.3.3, with a large case list based on the state, within which are individual case lists based on the current input character. The tokens themselves are defined as an enumerated type in **globals.h** (lines 174–186), which include all the tokens listed in Table 2.1, together with the bookkeeping tokens **EOF** (when the end of the file is reached) and **ERROR** (when an erroneous character is encountered). The states of the scanner are also defined as an enumerated type, but within the scanner itself (lines 612–614).

A scanner also needs in general to compute the attributes, if any, of each token, and sometimes also take other actions (such as inserting identifiers into a symbol table). In the case of the TINY scanner, the only attribute that is computed is the lexeme, or string value of the token recognized, and this is placed in the variable **tokenString**. This variable, together with **getToken** are the only services offered to other parts of the compiler, and their definitions are collected in the header file **scan.h** (lines 550–571). Note that **tokenString** is declared with a fixed length of 41, so that identifiers, for example, cannot be more than 40 characters (plus the ending null character). This is a limitation that is discussed later.

The scanner makes use of three global variables: the file variables **source** and **listing**, and the integer variable **lineno**, which are declared in **globals.h**, and allocated and initialized in **main.c**.

Additional bookkeeping done by the **getToken** procedure is as follows. The table **reservedWords** (lines 649–656) and the procedure **reservedLookup** (lines 658–666) perform a lookup of reserved words after an identifier is recognized by the principal loop of **getToken**, and the value of **currentToken** is changed accordingly. A flag variable **save** is used to indicate whether a character is to be added to **tokenString**; this is necessary, since white space, comments, and nonconsumed lookaheads should not be included.

Character input to the scanner is provided by the **getNextChar** function (lines 627–642), which fetches characters from **lineBuf**, a 256-character buffer internal to the scanner. If the buffer is exhausted, **getNextChar** refreshes the buffer from the **source** file using the standard C procedure **fgets**, assuming each time that a new source code line is being fetched (and incrementing **lineno**). While this assumption allows for simpler code, a TINY program with lines greater than 255 characters will not be handled quite correctly. We leave the investigation of the behavior of **getNextChar** in this case (and improvements to its behavior) to the exercises.

Finally, the recognition of numbers and identifiers in TINY requires that the transitions to the final state from **INNUM** and **INID** be nonconsuming (see Figure 2.10). We implement this by providing an **ungetNextChar** procedure (lines 644–647) that backs up one character in the input buffer. Again, this does not quite work for programs having very long source lines, and alternatives are explored in the exercises.

As an illustration of the behavior of the TINY scanner, consider the TINY program **sample.tny** in Figure 2.11 (the same program that was given as an example in Chapter 1). Figure 2.12 shows the listing output of the scanner, given this program as input, when **TraceScan** and **EchoSource** are set.

The remainder of this section will be devoted to an elaboration of some of the implementation issues raised by this scanner implementation.

Figure 2.11
Sample program in the TINY language

```
{ Sample program
  in TINY language -
  computes factorial
}
read x; { input an integer }
if 0 < x then { don't compute if x <= 0 }
  fact := 1;
  repeat
    fact := fact * x;
    x := x - 1
  until x = 0;
  write fact { output factorial of x }
end
```


Figure 2.12
Output of scanner given the TINY program of Figure 2.11 as input

```
TINY COMPILATION: sample.tny
    1: { Sample program
    2:   in TINY language -
    3:   computes factorial
    4: }
    5: read x; { input an integer }
       5: reserved word: read
       5: ID, name= x
       5: ;
    6: if 0 < x then { don't compute if x <= 0 }
       6: reserved word: if
       6: NUM, val= 0
       6: <
       6: ID, name= x
       6: reserved word: then
    7:   fact := 1;
       7: ID, name= fact
       7: :=
       7: NUM, val= 1
       7: ;
    8:   repeat
       8: reserved word: repeat
    9:     fact := fact * x;
       9: ID, name= fact
       9: :=
       9: ID, name= fact
       9: *
       9: ID, name= x
       9: ;
   10:     x := x - 1
      10: ID, name= x
      10: :=
      10: ID, name= x
      10: -
      10: NUM, val= 1
   11:   until x = 0;
      11: reserved word: until
      11: ID, name= x
      11: =
      11: NUM, val= 0
      11: ;
   12:   write fact { output factorial of x }
      12: reserved word: write
      12: ID, name= fact
   13: end
      13: reserved word: end
      14: EOF
```

2.5.2 Reserved Words Versus Identifiers

Our TINY scanner recognizes reserved words by first considering them as identifiers and then looking them up in a table of reserved words. This is a common practice in scanners, but it means that the efficiency of the scanner depends on the efficiency of the lookup process in the reserved word table. In our scanner we have used a very simple method—linear search—in which the table is searched sequentially from beginning to end. This is not a problem for very small tables such as that for TINY, with only eight reserved words, but it becomes an unacceptable situation in scanners for real languages, which commonly have between 30 and 60 reserved words. In this case a faster lookup is required, and this can require the use of a better data structure than a linear list. One possibility is a binary search, which we could have applied had we written the list of reserved words in alphabetic order. Another possibility is to use a hash table. In this case we would like to use a hash function that has a very small number of collisions. Such a hash function can be developed in advance, since the reserved words are not going to change (at least not rapidly), and their places in the table will be fixed for every run of the compiler. Some research effort has gone into the determination of **minimal perfect hash functions** for various languages, that is, functions that distinguish each reserved word from the others, and that have the minimum number of values, so that a hash table no larger than the number of reserved words can be used. For instance, if there are only eight reserved words, then a minimal perfect hash function would always yield a value from 0 to 7, and each reserved word would yield a different value. (See the Notes and References section for more information.)

Another option in dealing with reserved words is to use the same table that stores identifiers, that is, the symbol table. Before processing is begun, all reserved words are entered into this table and are marked reserved (so that no redefinition is allowed). This has the advantage that only a single lookup table is required. In the TINY scanner, however, we do not construct the symbol table until after the scanning phase, so this solution is not appropriate for this particular design.

2.5.3 Allocating Space for Identifiers

A further flaw in the design of the TINY scanner is that token strings can only be a maximum of 40 characters. This is not a problem for most of the tokens, since their string sizes are fixed, but it is a problem for identifiers, since programming languages often require that arbitrarily long identifiers be allowed in programs. Even worse, if we allocate a 40-character array for each identifier, then much of the space is wasted, since most identifiers are short. This doesn't happen in the code of the TINY compiler, since token strings are copied using the utility function `copyString`, which dynamically allocates only the necessary space, as we will see in Chapter 4. A solution to the size limitation of `tokenString` would be similar: only allocate space on an as needed basis, possibly using the `realloc` standard C function. An alternative is to allocate an initial large array for all identifiers and then to perform do-it-yourself memory allocation within this array. (This is a special case of the standard dynamic memory management schemes discussed in Chapter 7.)

2.6 USE OF Lex TO GENERATE A SCANNER AUTOMATICALLY

In this section we repeat the development of a scanner for the TINY language carried out in the previous section, but now we will use the Lex scanner generator to generate a scanner from a description of the tokens of TINY as regular expressions. Since there are a number of different versions of Lex in existence, we confine our discussion to those features that are common to all or most of the versions. The most popular version of Lex is called **flex** (for Fast Lex). It is distributed as part of the **Gnu compiler package** produced by the Free Software Foundation, and is also freely available at many Internet sites.

Lex is a program that takes as its input a text file containing regular expressions, together with the actions to be taken when each expression is matched. Lex produces an output file that contains C source code defining a procedure `yylex` that is a table-driven implementation of a DFA corresponding to the regular expressions of the input file, and that operates like a `getToken` procedure. The Lex output file, usually called `lex.yy.c` or `lexyy.c`, is then compiled and linked to a main program to get a running program, just as the `scan.c` file was linked with the `tiny.c` file in the previous section.

In the following, we first discuss the Lex conventions for writing regular expressions and the format of a Lex input file. We then discuss the Lex input file for the TINY scanner given in Appendix B.

2.6.1 Lex Conventions for Regular Expressions

Lex conventions are very similar to those discussed in Section 2.2.3. Rather than list all of Lex's metacharacters and describe them individually, we will give an overview and then give the Lex conventions in a table.

Lex allows the matching of single characters, or strings of characters, simply by writing the characters in sequence, as we did in previous sections. Lex also allows metacharacters to be matched as actual characters by surrounding the characters in quotes. Quotes can also be written around characters that are not metacharacters, where they have no effect. Thus, it makes sense to write quotes around all characters that are to be matched directly, whether or not they are metacharacters. For example, we can write either `if` or `"if"` to match the reserved word **if** that begins an if-statement. On the other hand, to match a left parenthesis, we must write `"("`, since it is a metacharacter. An alternative is to use the backslash metacharacter \, but this works only for single metacharacters: to match the character sequence `(*` we would have to write `\(\*`, repeating the backslash. Clearly, writing `"(*"` is preferable. Also using the backslash with regular characters may have a special meaning. For example, `\n` matches a newline and `\t` matches a tab (these are typical C conventions, and most such conventions carry over into Lex).

Lex interprets the metacharacters `*`, `+`, `(`, `)`, and `|` in the usual way. Lex also uses the question mark as a metacharacter to indicate an optional part. As an example of the

Lex notation discussed so far, we can write a regular expression for the set of strings of *a*'s and *b*'s that begin with either *aa* or *bb* and have an optional *c* at the end as

```
(aa|bb)(a|b)*c?
```

or as

```
("aa"|"bb")("a"|"b")*"c"?
```

The Lex convention for character classes (sets of characters) is to write them between square brackets. For example, `[abxz]` means any one of the characters *a*, *b*, *x*, or *z*, and we could write the previous regular expression in Lex as

```
(aa|bb)[ab]*c?
```

Ranges of characters can also be written in this form using a hyphen. Thus, the expression `[0-9]` means in Lex any of the digits zero through nine. A period is a metacharacter that also represents a set of characters: it represents any character except a newline. Complementary sets—that is, sets that do *not* contain certain characters—can also be written in this notation, using the carat `^` as the first character inside the brackets. Thus, `[^0-9abc]` means any character that is not a digit and is not one of the letters *a*, *b*, or *c*.

As an example, we write a regular expression for the set of signed numbers that may contain a fractional part or an exponent beginning with the letter *E* (this expression was written in slightly different form in Section 2.2.4):

```
("+"|"-")?[0-9]+("."[0-9]+)?(E("+"|"-")?[0-9]+)?
```

One curious feature in Lex is that inside square brackets (representing a character class), most of the metacharacters lose their special status and do not need to be quoted. Even the hyphen can be written as a regular character if it is listed first. Thus, we could have written `[-+]` instead of `("+"|"-")` in the previous regular expression for numbers (but not `[+-]` because of the metacharacter use of `-` to express a range of characters). As another example, `[."?]` means any of the three characters period, quotation mark, or question mark (all three of these characters have lost their metacharacter meaning inside the brackets). Some characters, however, are still metacharacters even inside the square brackets, and to get the actual character, we must precede the character by a backslash (quotes cannot be used as they have lost their metacharacter meaning). Thus, `[\^\\]` means either of the actual characters ^ or \.

A further important metacharacter convention in Lex is the use of curly brackets to denote names of regular expressions. Recall that a regular expression can be given a name, and that these names can be used in other regular expressions as long as there are no recursive references. For example, we defined ***signedNat*** in Section 2.2.4 as follows:

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


In this and other examples, we used italics to distinguish names from ordinary sequences of characters. Lex files, however, are ordinary text files, so italics are not available. Instead, Lex uses the convention that previously defined names are surrounded by curly brackets. Thus, the previous example would appear as follows in Lex (Lex also dispenses with the equal sign in defining names):

```
nat [0-9]+
signedNat (+|-)?{nat}
```

Note that the curly brackets do not appear when a name is defined, only when it is used.

Table 2.2 contains a summary list of the metacharacter conventions of Lex that we have discussed. There are a number of other metacharacter conventions in Lex that we will not use and we do not discuss them here (see the references at the end of the chapter).

Table 2.2 Metacharacter conventions in Lex

Pattern	Meaning
`a`	the character *a*
`"a"`	the character *a*, even if *a* is a metacharacter
`\a`	the character *a* when *a* is a metacharacter
`a*`	zero or more repetitions of *a*
`a+`	one or more repetitions of *a*
`a?`	an optional *a*
`a|b`	*a* or *b*
`(a)`	*a* itself
`[abc]`	any of the characters *a*, *b*, or *c*
`[a-d]`	any of the characters *a*, *b*, *c*, or *d*
`[^ab]`	any character except *a* or *b*
`.`	any character except a newline
`{xxx}`	the regular expression that the name *xxx* represents

2.6.2 The Format of a Lex Input File

A Lex input file consists of three parts, a collection of **definitions**, a collection of **rules**, and a collection of **auxiliary routines** or **user routines**. The three sections are separated by double percent signs that appear on separate lines beginning in the first column. Thus, the layout of a Lex input file is as follows:

```
{definitions}
%%
{rules}
%%
{auxiliary routines}
```

To properly understand how Lex interprets such an input file, we must keep in mind that some parts of the file will be regular expression information that Lex uses to guide its construction of the C output code, while other parts of the file will be actual C code that we are supplying to Lex, and that Lex will insert verbatim in the output code at the appropriate location. The precise rules Lex uses for this will be given after we have discussed each of the three sections and given a few examples.

The definition section occurs before the first **%%**. It contains two things. First, any C code that must be inserted external to any function should appear in this section between the delimiters **%{** and **%}**. (Note the order of these characters!) Second, names for regular expressions must also be defined in this section. A name is defined by writing it on a separate line starting in the first column and following it (after one or more blanks) by the regular expression it represents.

The second section contains the rules. These consist of a sequence of regular expressions followed by the C code that is to be executed when the corresponding regular expression is matched.

The third and final section contains C code for any auxiliary routines that are called in the second section and not defined elsewhere. This section may also contain a main program, if we want to compile the Lex output as a standalone program. This section can also be missing, in which case the second **%%** need not be written. (The first **%%** is always necessary.)

We give a series of examples to illustrate the format of a Lex input file.

Example 2.20 The following Lex input specifies a scanner that adds line numbers to text, sending its output to the screen (or a file, if redirected):

```
%{
/* a Lex program that adds line numbers
   to lines of text, printing the new text
   to the standard output
*/
#include <stdio.h>
int lineno = 1;
%}
line .*\n
%%
{line} { printf("%5d %s",lineno++,yytext); }
%%
main()
{ yylex(); return 0; }
```

For example, running the program obtained from Lex on this input file itself gives the following output:

```
1 %{
2 /* a Lex program that adds line numbers
3    to lines of text, printing the new text
4    to the standard output
5 */
6 #include <stdio.h>
7 int lineno = 1;
8 %}
9 line .*\n
10 %%
11 {line} { printf("%5d %s",lineno++,yytext); }
12 %%
13 main()
14 { yylex(); return 0; }
```


We comment on this Lex input file using these line numbers. First, lines 1 through 8 are between the delimiters **%{** and **%}**. This causes these lines to be inserted directly into the C code produced by Lex, external to any procedure. In particular, the comment in lines 2 through 5 will be inserted near the beginning of the program, and the **#include** directive and the definition of the integer variable **lineno** on lines 6 and 7 will be inserted externally, so that **lineno** becomes a global variable and is initialized to the value 1. The other definition that appears before the first **%%** is the definition of the name **line** which is defined to be the regular expression **".*\n"**, which matches 0 or more characters (not including a newline), followed by a newline. In other words, the regular expression defined by **line** matches every line of input. Following the **%%** on line 10, line 11 comprises the action section of the Lex input file. In this case we have written a single action to be performed whenever a **line** is matched (**line** is surrounded by curly brackets to distinguish it as a name, according to the Lex convention). Following the regular expression is the **action**, that is, the C code that is to be executed whenever the regular expression is matched. In this example, the action consists of a single C statement, which is contained within the curly brackets of a C block. (Keep in mind that the curly brackets surrounding the name **line** have a completely different function from the curly brackets that form a block in the C code of the following action.) This C statement is to print the line number (in a five-space field, right justified) and the string **yytext**, after which **lineno** is incremented. The name **yytext** is the internal name Lex gives to the string matched by the regular expression, which in this case consists of each line of input (including the newline).[5] Finally, the C code after the second double percent (lines 13 and 14) is inserted as is at the end of the C code produced by Lex. In this example, the code consists of the definition of a **main** procedure that calls the function **yylex**. This allows the C code produced by Lex to be compiled into an executable program. (**yylex** is the name given to the procedure constructed by Lex that implements the DFA associated with the regular expressions and actions given in the action section of the input file.) §

5. We list the Lex internal names that are discussed in this section in a table at the end of the section.

Example 2.21 Consider the following Lex input file:

```
%{
/* a Lex program that changes all numbers
   from decimal to hexadecimal notation,
   printing a summary statistic to stderr
*/
#include <stdlib.h>
#include <stdio.h>
int count = 0;
%}
digit [0-9]
number {digit}+
%%
{number} { int n = atoi(yytext);
           printf("%x", n);
           if (n > 9) count++; }
%%
main()
{ yylex();
  fprintf(stderr,"number of replacements = %d",
                 count);
  return 0;
}
```

It is similar in structure to the previous example, except that the **main** procedure prints the count of the number of replacements to **stderr** after calling **yylex**. This example is also different in that not all text is matched. Indeed, only numbers are matched in the action section, where the C code for the action first converts the matched string (**yytext**) to an integer **n**, then prints it in hexadecimal form (**printf("%x",...)**), and finally increments **count** if the number is greater than 9. (If the number is smaller than or equal to 9, then it looks no different in hex.) Thus, the only action specified is for strings that are sequences of digits. Yet Lex generates a program that also matches all nonnumeric characters, and passes them through to the output. This is an example of a **default action** by Lex. If a character or string of characters matches none of the regular expressions in the action section, Lex will match it by default and echo it to the output. (Lex can also be forced to generate a runtime error, but we will not study this here.) The default action can also be specifically indicated through the Lex internally defined macro **ECHO**. (We study this use in the next example.) §

Example 2.22 Consider the following Lex input file:

```
%{
/* Selects only lines that end or
   begin with the letter 'a'.
   Deletes everything else.
*/
```



```
#include <stdio.h>
%}
ends_with_a .*a\n
begins_with_a a.*\n
%%
{ends_with_a} ECHO;
{begins_with_a} ECHO;
.*\n ;
%%
main()
{ yylex(); return 0; }
```

This Lex input causes all input lines that begin or end with the character *a* to be written to the output. All other input lines are suppressed. The suppression of the input is caused by the rule below the **ECHO** rules. In this rule the "empty" action is specified for the regular expression **.*\n** by writing a semicolon for the C action code.

There is an additional feature of this Lex input that is worth noting. The listed rules are **ambiguous** in that a string may match more than one rule. In fact, *any* input line matches the expression **.*\n**, regardless of whether it is part of a line that begins or ends with an *a*. Lex has a priority system for resolving such ambiguities. First, Lex always matches the longest possible substring (so Lex always generates a scanner that follows the longest substring principle). Then, if the longest substring still matches two or more rules, Lex picks the first rule in the order they are listed in the action section. It is for this reason that the above Lex input file lists the **ECHO** actions first. If we had listed the actions in the following order,

```
.*\n ;
{ends_with_a} ECHO;
{begins_with_a} ECHO;
```

then the program produced by Lex would generate no output at all for any file, since every line of input will be matched by the first rule. §

Example 2.23 In this example, Lex generates a program that will convert all uppercase letters to lowercase, except for letters inside C-style comments (that is, anything inside the delimiters **/*...*/**):

```
%{
/* Lex program to convert uppercase to
   lowercase except inside comments
*/
#include <stdio.h>
#ifndef FALSE
#define FALSE 0
#endif
#ifndef TRUE
#define TRUE 1
```

```
#endif
%}
%%
[A-Z] {putchar(tolower(yytext[0]));
       /* yytext[0] is the single
          uppercase char found */
      }
"/*"  { char c;
        int done = FALSE;
        ECHO;
        do
        { while ((c=input())!='*')
            putchar(c);
          putchar(c);
          while ((c=input())=='*')
            putchar(c);
          putchar(c);
          if (c == '/') done = TRUE;
        } while (!done);
      }
%%
void main(void)
{ yylex();}
```

This example shows how code can be written to sidestep difficult regular expressions and implement a small DFA directly as a Lex action. Recall from the discussion in Section 2.2.4 that a regular expression for a C comment is extremely complex to write down. Instead, we write down a regular expression only for the string that begins a C comment—that is, **"/*"**—and then supply action code that will look for the ending string **"*/"**, while providing the appropriate action for other characters within the comment (in this case to just echo them without further processing). We do this by imitating the DFA from Example 2.9 (see Figure 2.4, page 53). Once we have recognized the string **"/*"**, we are in state 3, so our code picks up the DFA there. The first thing we do is cycle through characters (echoing them to the output) until we see an asterisk (corresponding to the ***other*** loop in state 3), as follows:

```
while ((c=input())!='*') putchar(c);
```

Here we have used yet another Lex internal procedure called **input**. The use of this procedure, rather than a direct input using **getchar**, ensures that the Lex input buffer is used, and that the internal structure of the input string is preserved. (Note, however, that we do use a direct output procedure **putchar**. This will be discussed further in Section 2.6.4.)

The next step in our code for the DFA corresponds to state 4. We loop again until we do *not* see an asterisk, and then, if the character is a forward slash, we exit; otherwise, we return to state 3. §

We end this subsection with a summary of the Lex conventions we have introduced in the examples.

AMBIGUITY RESOLUTION

Lex's output will always first match the longest possible substring to a rule. If two or more rules cause substrings of equal length to be matched, then Lex's output will pick the rule listed first in the action section. If no rule matches any nonempty substring, then the default action copies the next character to the output and continues.

INSERTION OF C CODE

(1) Any text written between **%{** and **%}** in the definition section will be copied directly to the output program external to any procedure. (2) Any text in the auxiliary procedures section will be copied directly to the output program at the end of the Lex code. (3) Any code that follows a regular expression (by at least one space) in the action section (after the first **%%**) will be inserted at the appropriate place in the recognition procedure **yylex** and will be executed when a match of the corresponding regular expression occurs. The C code representing an action may be either a single C statement or a compound C statement consisting of any declarations and statements surrounded by curly brackets.

INTERNAL NAMES

Table 2.3 lists the Lex internal names that we discuss in this chapter. Most of these have been discussed in the previous examples.

Table 2.3 Some Lex internal names

Lex Internal Name	Meaning/Use
lex.yy.c or **lexyy.c**	Lex output file name
yylex	Lex scanning routine
yytext	string matched on current action
yyin	Lex input file (default: **stdin**)
yyout	Lex output file (default: **stdout**)
input	Lex buffered input routine
ECHO	Lex default action (print **yytext** to **yyout**)

We note one feature from this table not mentioned previously, which is that Lex has its own internal names for the files from which it takes input and to which it sends output: **yyin** and **yyout**. Using the standard Lex input routine **input** will automatically take input from the file **yyin**. However, in the foregoing examples, we have bypassed the internal output file **yyout** and just written to the standard output using **printf** and **putchar**. A better implementation, allowing the assignment of output to an arbitrary file, would replace these uses with **fprintf(yyout,...)** and **putc(...,yyout)**.

2.6.3 A TINY Scanner Using Lex

Appendix B gives a listing of a Lex input file **tiny.l** that will generate a scanner for the TINY language, whose tokens were described in Section 2.5 (see Table 2.1). In the following we make a few remarks about this input file (lines 3000–3072).

First, in the definitions section, the C code we insert directly into the Lex output consists of three **#include** directives (**globals.h**, **util.h**, and **scan.h**) and the definition of the **tokenString** attribute. This is necessary to provide the interface between the scanner and the rest of the TINY compiler.

The further contents of the definition section comprise the definitions of the names for the regular expressions that define the TINY tokens. Note that the definition of **number** uses the previously defined name **digit**, and the definition of **identifier** uses the previously defined **letter**. The definitions also distinguish between newlines and other white space (blanks and tabs, lines 3019 and 3020), since a newline will cause **lineno** to be incremented.

The action section of the Lex input consists of listing the various tokens, together with a **return** statement that returns the appropriate token as defined in **globals.h**. In this Lex definition we have listed the rules for reserved words before the rule for an identifier. Had we listed the identifier rule first, the ambiguity resolution rules of Lex would cause an identifier to always be recognized instead of a reserved word. We could also write code as in the scanner of the previous section, in which only identifiers are recognized, and then reserved words are looked up in a table. This would indeed be preferable in a real compiler, since separately recognized reserved words cause the size of the tables in the scanner code generated by Lex to grow enormously (and hence the size of memory used by the scanner).

One quirk of the Lex input is that we have to write code to recognize comments to ensure that **lineno** is updated correctly, even though the regular expression for TINY comments is easy to write. Indeed, the regular expression is

```
"{"[^\}]*"}"
```

(Note the use of the backslash inside the square brackets to remove the metacharacter meaning of right curly bracket—quotes will not work here.)[6]

We note also that there is no code written to return the **EOF** token on encountering the end of the input file. The Lex procedure **yylex** has a default behavior on encountering **EOF**—it returns the value 0. It is for this reason that the token **ENDFILE** was written first in the definition of **TokenType** in **globals.h** (line 179), so that it will have value 0.

Finally, the **tiny.l** file contains a definition of the **getToken** procedure in the auxiliary procedures section (lines 3056–3072). While this code contains some ad hoc initializations of Lex internals (such as **yyin** and **yyout**) that would be better performed directly in the main program, it does permit us to use the Lex-generated scanner directly, without changing any other files in the TINY compiler. Indeed, after generating the C scanner file **lex.yy.c** (or **lexyy.c**), this file can be compiled and linked directly with the other TINY source files to produce a Lex-based version of the compiler. However, this version of the compiler lacks one service of the earlier version, in that no source code echoing with line numbers is provided (see Exercise 2.35).

6. Some versions of Lex have an internally defined variable **yylineno** that is automatically updated. Use of this variable instead of **lineno** would make it possible to eliminate the special code.

EXERCISES

2.1 Write regular expressions for the following character sets, or give reasons why no regular expression can be written:
- **a.** All strings of lowercase letters that begin and end in *a*.
- **b.** All strings of lowercase letters that either begin or end in *a* (or both).
- **c.** All strings of digits that contain no leading zeros.
- **d.** All strings of digits that represent even numbers.
- **e.** All strings of digits such that all the 2's occur before all the 9's.
- **f.** All strings of *a*'s and *b*'s that contain no three consecutive *b*'s.
- **g.** All strings of *a*'s and *b*'s that contain an odd number of *a*'s or an odd number of *b*'s (or both).
- **h.** All strings of *a*'s and *b*'s that contain an even number of *a*'s and an even number of *b*'s.
- **i.** All strings of *a*'s and *b*'s that contain exactly as many *a*'s as *b*'s.

2.2 Write English descriptions for the languages generated by the following regular expressions:
- **a.** **(a|b)*a(a|b|ε)**
- **b.** **(A|B|...|Z)(a|b|...|z)***
- **c.** **(aa|b)*(a|bb)***
- **d.** **(0|1|...|9|A|B|C|D|E|F)+(x|X)**

2.3 **a.** Many systems contain a version of **grep** (global regular expression print), a regular expression search program originally written for Unix.[7] Find a document describing your local grep, and describe its metasymbol conventions.
- **b.** If your editor accepts some sort of regular expressions for its string searches, describe its metasymbol conventions.

2.4 In the definition of regular expressions, we described the precedence of the operations, but not their associativity. For example, we did not specify whether **a|b|c** meant **(a|b)|c** or **a|(b|c)** and similarly for concatenation. Why was this?

2.5 Prove that $L(r^{**}) = L(r^*)$ for any regular expression r.

2.6 In describing the tokens of a programming language using regular expressions, it is not necessary to have the metasymbols ϕ (for the empty set) or ε (for the empty string). Why is this?

2.7 Draw a DFA corresponding to the regular expression ϕ.

2.8 Draw DFAs for each of the sets of characters of **(a)**–**(i)** in Exercise 2.1, or state why no DFA exists.

2.9 Draw a DFA that accepts the four reserved words **case**, **char**, **const**, and **continue** from the C language.

7. There are actually three versions of grep available on most Unix systems: "regular" grep, egrep (extended grep), and fgrep (fast grep).