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.

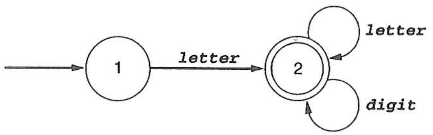


Figure 2.1
A finite automaton for identifiers

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

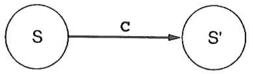
Diagrams such as the one we have discussed are useful descriptions of finite automata, since they allow us to visualize easily the actions of the algorithm. Occasionally, however, it is necessary to have a more formal description of a finite automaton, and so we proceed now to give a mathematical definition. 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.

partial ?

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).

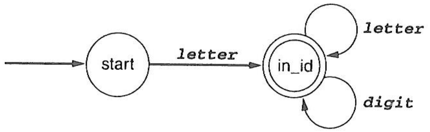
We make the following notes about this definition. $S \times \Sigma$ refers to the **Cartesian or cross product of S and Σ**: the set of pairs (s, c), where $s \in S$ and $c \in \Sigma$. The function T records the transitions: $T(s, c) = s'$ if there is a transition from state s to state s' labeled by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

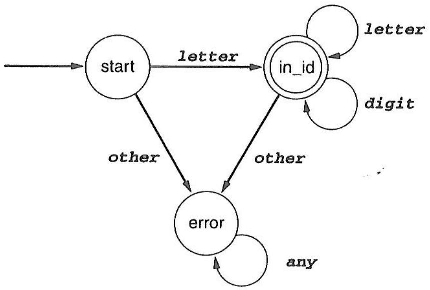
For instance, the name **`letter`** represents any letter of the alphabet according to the following regular definition:

```
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

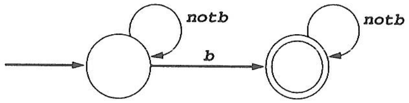
```
other = ~(letter|digit)
```

[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.

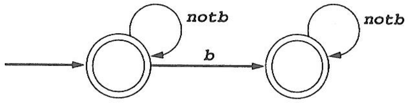
We now turn to a series of examples of DFAs, paralleling the examples of the previous section.

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. §

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. §

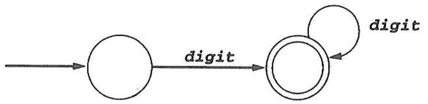
2.8 In the previous section we gave regular definitions for numeric constants in scientific notation as follows:

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

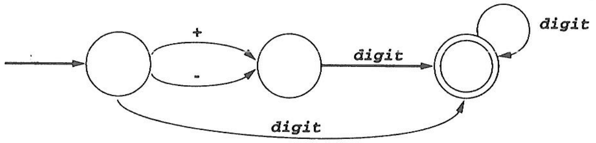
We would like to write down DFAs for the strings matched by these definitions, but it is helpful to first rewrite them as follows:

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

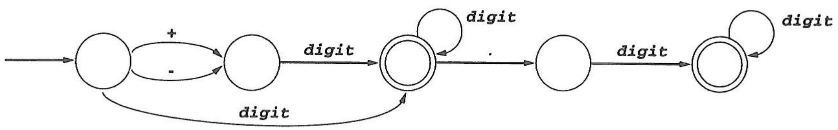
It is easy to write down a DFA for **nat** as follows (recall that **a+ = aa*** for any **a**):



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:



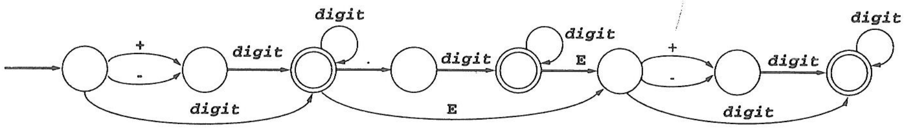
It is also easy to add the optional fractional part, as follows:



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

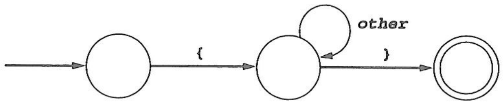
Finally, we need to add the optional exponential part. To do this, we note that the exponential part must begin with the letter *E* and can occur only after we have reached either of the previous accepting states. The final diagram is given in Figure 2.3.

Figure 2.3 A finite automaton for floating-point numbers



§

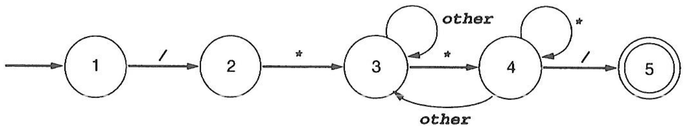
Example 2.9 Unnested comments can be described using DFAs. For example, comments surrounded by curly brackets are accepted by the following DFA:



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

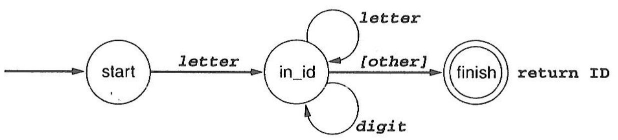
We have studied DFAs as a way of representing algorithms that accept character strings according to a pattern. As the reader may have already guessed, there is a strong relationship between a regular expression for a pattern and a DFA that accepts strings according to the pattern. We will explore this 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:

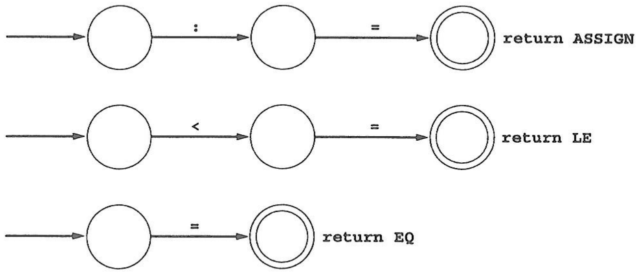
Figure 2.5
Finite automaton for an identifier with delimiter and return value



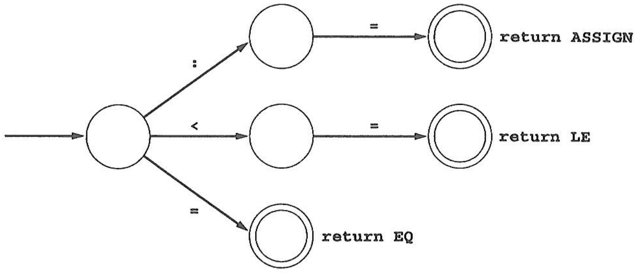
In the diagram we have surrounded the ***other*** transition with square brackets to indicate that the delimiting character should be considered lookahead, that is, that it should be returned to the input string and not consumed. In addition, the error state has become the accepting state in this diagram 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.

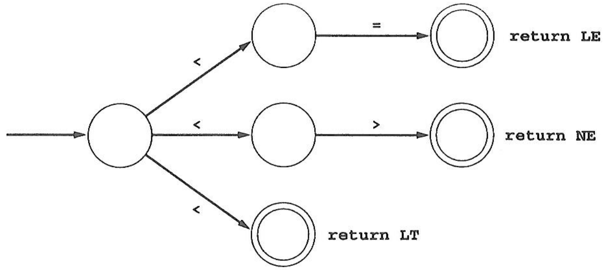
We turn our attention now to the question of how to arrive at the start state in the first place. In a typical programming language there are many tokens, and each token will be recognized by its own DFA. If each of these tokens begins with a different character, then it is easy to tie them together 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:



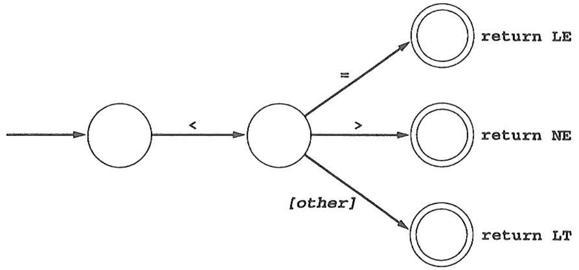
Since each of these tokens begins with a different character, we can simply identify their start states to get the following DFA:



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:

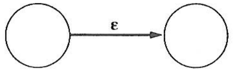


In principle, we should be able to combine all the tokens into one giant DFA in this fashion. However, the complexity of such a task becomes enormous, especially if it is done in an unsystematic way.

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.

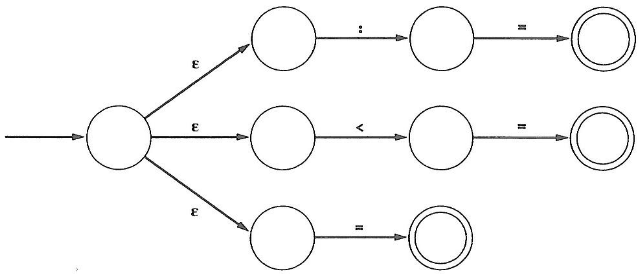
The new kind of finite automaton is called a **nondeterministic finite automaton,** or **NFA** for short. Before we define it, we need one more generalization that will be useful in applying finite automata to scanners: the concept of the ε-transition.

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:

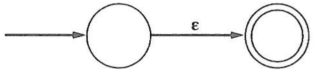


This should not be confused with a match of the character ε in the input: if the alphabet includes such a character, it must be distinguished from the use of ε as a metacharacter to represent an ε-transition.

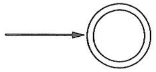
ε-transitions are somewhat counterintuitive, since they may occur "spontaneously," that is, without lookahead and without change to the input string, but they are useful in two ways. First, they can express a choice of alternatives in a way that does not involve combining states. For example, the choice 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:

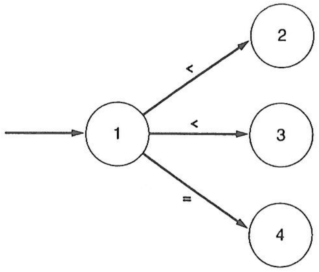


Of course, this is equivalent to the following DFA, which expresses that acceptance should occur without matching any characters:



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 $c_1c_2. . .c_n$ 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.

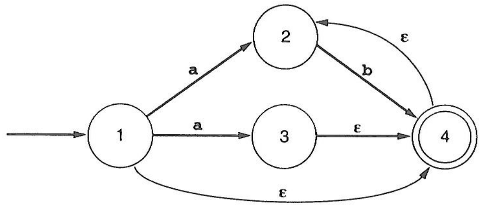
Again, we need to note a few things about this definition. Any of the c_i in $c_1c_2. . .c_n$ may be ε, and the string that is actually accepted is the string $c_1c_2. . .c_n$ with the ε's removed (since the concatenation of s with ε is s itself). Thus, the string

$c_1c_2. . .c_n$ may actually have fewer than n characters in it. Also, the sequence of states $s_1, . . . , s_n$ are chosen from the *sets* of states $T(s_0, c_1), . . . , T(s_{n-1}, c_n)$, 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.

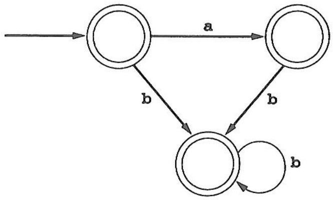


The string **abb** can be accepted by either of the following sequences of transitions:

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

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

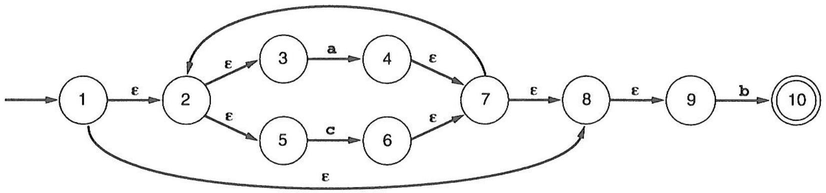
Indeed the transitions from state 1 to state 2 on a, and from state 2 to state 4 on b, allow the machine to accept the string ab, and then, using the ε-transition from state 4 to state 2, all strings matching the regular expression **ab+**. Similarly, the transitions from state 1 to state 3 on a, and 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:

$$\rightarrow 1 \xrightarrow{\varepsilon} 2 \xrightarrow{\varepsilon} 3 \xrightarrow{\mathtt{a}} 4 \xrightarrow{\varepsilon} 7 \xrightarrow{\varepsilon} 2 \xrightarrow{\varepsilon} 5 \xrightarrow{\mathtt{c}} 6 \xrightarrow{\varepsilon}$$

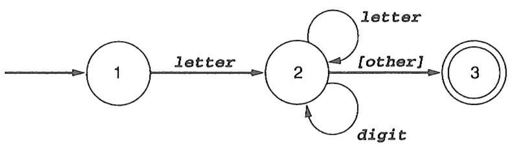
$$\rightarrow 7 \xrightarrow{\varepsilon} 2 \xrightarrow{\varepsilon} 3 \xrightarrow{\mathtt{a}} 4 \xrightarrow{\varepsilon} 7 \xrightarrow{\varepsilon} 8 \xrightarrow{\varepsilon} 9 \xrightarrow{\mathtt{b}} 10$$

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 "nonconsuming" 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:

input char / state	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:

input char state	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):

input char state	/	*	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;
```

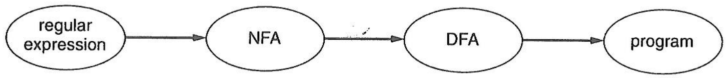
Algorithmic methods such as we have just described are called **table driven**, since they use tables to direct the progress of the algorithm. Table-driven methods have certain advantages: the size of the code is reduced, the same code will work for many different problems, and the code is easier to 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 inefficient, 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.

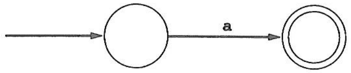
The simplest algorithm for translating a regular expression into a DFA proceeds via an intermediate construction, in which an NFA is derived from the regular expression, and then the NFA is used to construct an equivalent DFA. There exist algorithms that can translate a regular expression directly into 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:



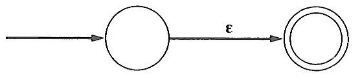
2.4.1 From a Regular Expression to an NFA

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

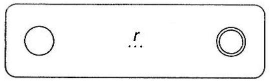


Similarly, an NFA that is equivalent to ε is



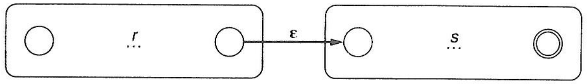
The case of the regular expression ϕ (which never occurs in practice in a compiler) is left as an exercise.

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 $r \mid s$ under the same assumptions as before. We do this as follows: