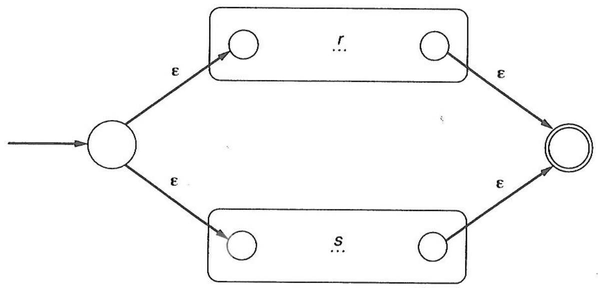
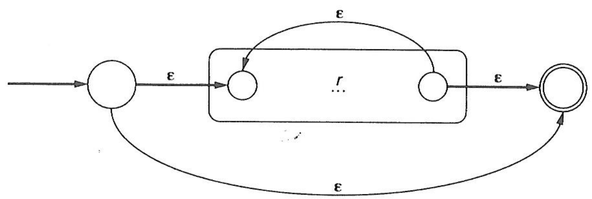


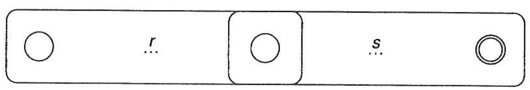
We have added a new start state and a new accepting state and connected them as shown using ε-transitions. Clearly, this machine accepts the language $L(r|s) = L(r) \cup L(s)$.

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:



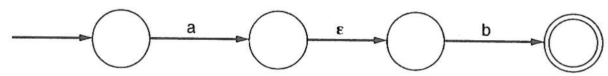
(This simplification depends, however, on the fact that in the other constructions, the accepting state has no transitions from it to other states—see the exercises.) Other simplifications are possible in the other cases. The reason we have expressed the translations as we have is that the machines are 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

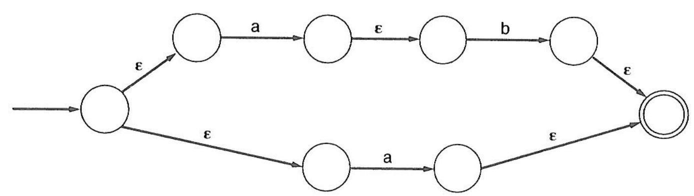
We translate the regular expression **ab|a** into an NFA according to Thompson's construction. We first form the machines for the basic regular expressions **a** and **b**:

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

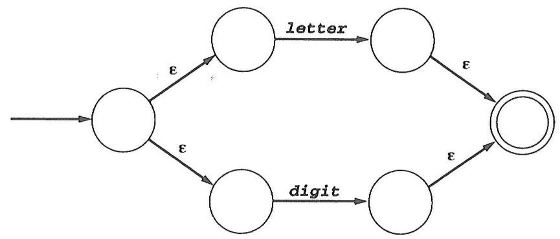


§

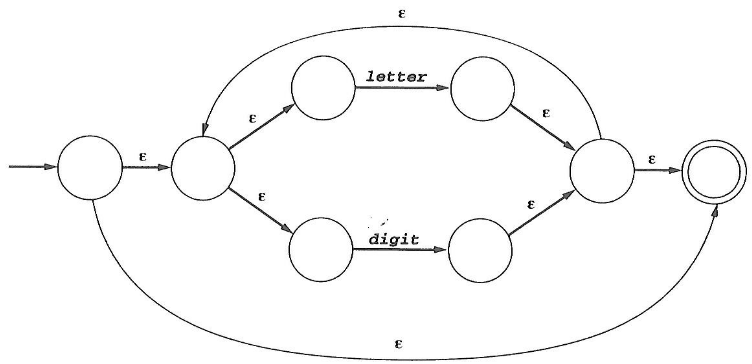
Example 2.13

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.

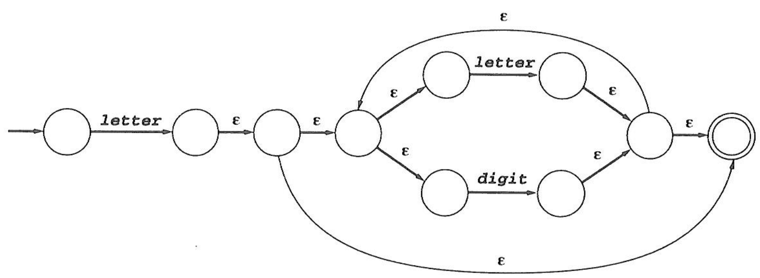


Figure 2.9
NFA for the regular expression `letter(letter|digit)*` using Thompson's construction

§

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.

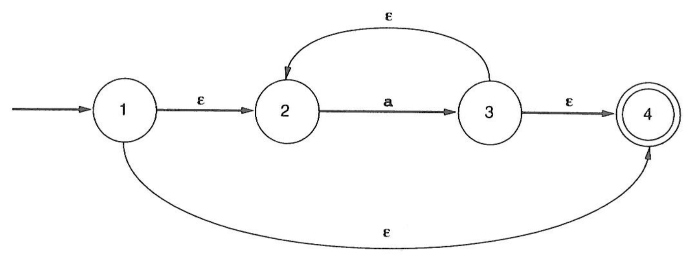
2.4.2 From an NFA to a DFA

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

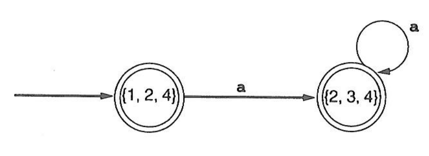
$$\bar{S} = \bigcup_{s \text{ in } S} \bar{s}$$

For instance, 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 $S'_a = \{ t \mid$ for some s in S there is a transition from s to t 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.

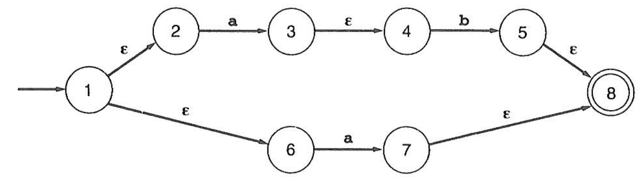
We illustrate the subset construction with a number of examples.

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:

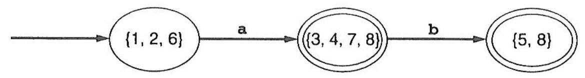


(Once the construction is complete, we could discard the subset terminology if we wished.) §

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

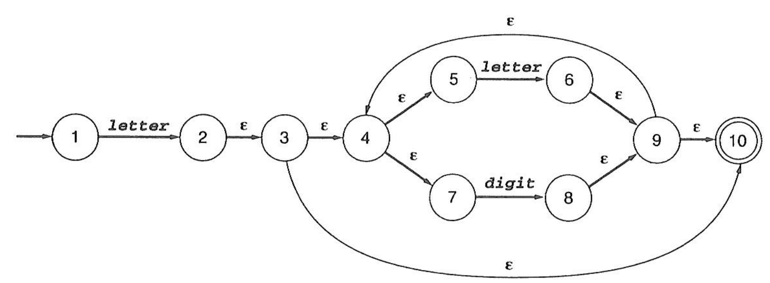


The DFA subset construction has as its start state $\overline{\{1\}} = \{1, 2, 6\}$. There is a transition on a from state 2 to state 3, and also from state 6 to state 7. Thus, $\overline{\{1, 2, 6\}_a} = \overline{\{3, 7\}} = \{3, 4, 7, 8\}$, and we have $\{1, 2, 6\} \xrightarrow{a} \{3, 4, 7, 8\}$. 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:

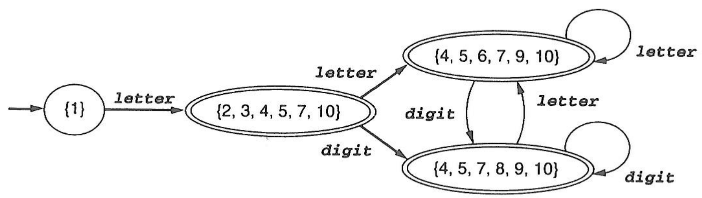


§

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}.

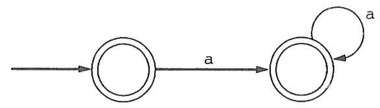
On the other hand, in Example 2.17, given the input string *r2d2*, we have the following sequence of states and transitions:

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

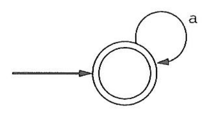
If these states are constructed as the transitions occur, then all the states of the DFA have been constructed and the state {4, 5, 7, 8, 9, 10} has even been constructed twice. Thus, this process is less efficient than constructing the entire DFA in the first place. For this reason, simulation of NFAs 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).

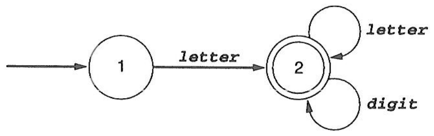
The algorithm proceeds by creating sets of states to be unified into single states. It begins with the most optimistic assumption possible: it creates two sets, one consisting of all the accepting states and the other consisting of all the nonaccepting states. Given this partition of the states of the 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

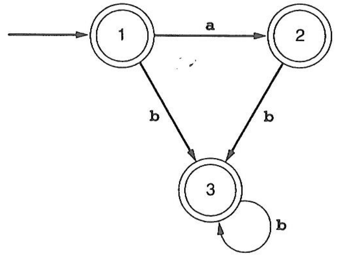
Consider the DFA we constructed in the previous example, corresponding to the regular expression **`letter(letter|digit)*`**. It had four states consisting of the start state and three accepting states. All three accepting states have transitions to other accepting states on both **`letter`** and **`digit`** 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):



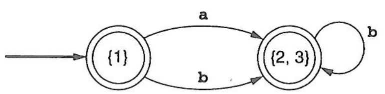
§

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

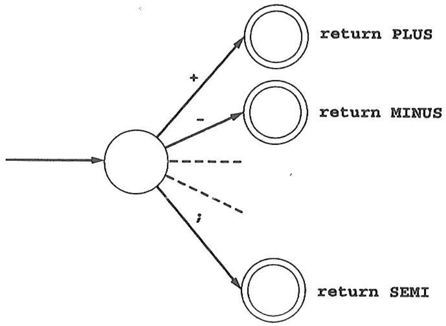
Reserved Words	Special Symbols	Other
if	+	***number***
then	-	(1 or more
else	*	digits)
end	/	
repeat	=	
until	<	***identifier***
read	(	(1 or more
write	)	letters)
	;	
	:=	

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.

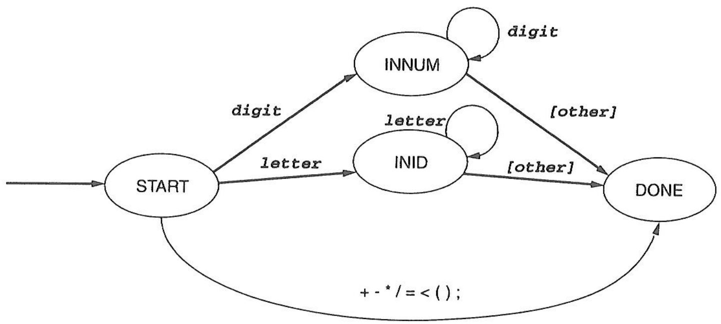
In addition to the tokens, TINY has the following lexical conventions. Comments are enclosed in curly brackets {. . .} and cannot be nested; the code is free format; white space consists of blanks, tabs, and newlines; and the principle of longest substring is followed in recognizing tokens.

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.

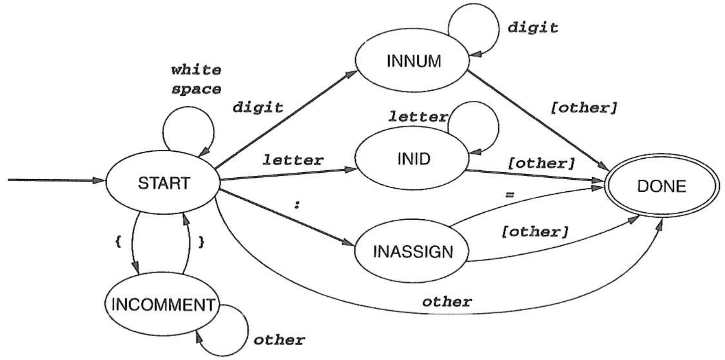


Figure 2.10
DFA of the TINY scanner

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 **ENDFILE** (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

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

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}
```