CHAPTER 10

Patterns, Automata, and Regular Expressions

A pattern is a set of objects with some recognizable property. One type of pattern is a set of character strings, such as the set of legal C identifiers, each of which is a string of letters, digits, and underscores, beginning with a letter or underscore. Another example would be the set of arrays of 0's and 1's of a given size that a character reader might interpret as representing the same symbol. Figure 10.1 shows three 7×7-arrays that might be interpreted as letter `A`'s. The set of all such arrays would constitute the pattern called "A."

```
0 0 0 1 0 0 0      0 0 0 0 0 0 0      0 0 0 1 0 0 0
0 0 1 1 1 0 0      0 0 1 0 0 0 0      0 0 1 0 1 0 0
0 0 1 0 1 0 0      0 0 1 1 0 0 0      0 1 1 0 1 0 0
0 1 1 0 1 1 0      0 1 0 1 0 0 0      0 1 1 1 1 1 0
0 1 1 1 1 1 0      0 1 1 1 0 0 0      1 1 0 0 0 1 1
1 1 0 0 0 1 1      1 0 0 1 1 0 0      1 0 0 0 0 0 1
1 0 0 0 0 0 1      1 0 0 0 1 0 0      0 0 0 0 0 0 0
```

Fig. 10.1. Three instances of the pattern "A."

The two fundamental problems associated with patterns are their definition and their recognition, subjects of this and the next chapter. Recognizing patterns is an integral part of tasks such as optical character recognition, an example of which was suggested by Fig. 10.1. In some applications, pattern recognition is a component of a larger problem. For example, recognizing patterns in programs is an essential part of compiling — that is, the translation of programs from one language, such as C, into another, such as machine language.

There are many other examples of pattern use in computer science. Patterns play a key role in the design of the electronic circuits used to build computers and other digital devices. They are used in text editors to allow us to search for instances of specific words or sets of character strings, such as "the letters `if` followed by any sequence of characters followed by `then`." Most operating systems allow us to use

patterns in commands; for example, the UNIX command "`ls *tex`" lists all files whose names end with the three-character sequence "`tex`".

An extensive body of knowledge has developed around the definition and recognition of patterns. This theory is called "automata theory" or "language theory," and its basic definitions and techniques are part of the core of computer science.

10.1 What This Chapter Is About

This chapter deals with patterns consisting of sets of strings. In it, we shall learn:

- The "finite automaton" is a graph-based way of specifying patterns. These come in two varieties, deterministic automata (Section 10.2) and nondeterministic automata (Section 10.3).
- A deterministic automaton is convertible in a simple way into a program that recognizes its pattern (Section 10.2).
- A nondeterministic automaton can be converted to a deterministic automaton recognizing the same pattern by use of the "subset construction" discussed in Section 10.4.
- Regular expressions are an algebra for describing the same kinds of patterns that can be described by automata (Sections 10.5 through 10.7).
- Regular expressions can be converted to automata (Section 10.8) and vice versa (Section 10.9).

We also discuss string patterns in the next chapter. There we introduce a recursive notation called "context-free grammars" for defining patterns. We shall see that this notation is able to describe patterns not expressible by automata or regular expressions. However, in many cases grammars are not convertible to programs in as simple manner as are automata or regular expressions.

10.2 State Machines and Automata

State

Programs that search for patterns often have a special structure. We can identify certain positions in the code at which we know something particular about the program's progress toward its goal of finding an instance of a pattern. We call these positions *states.* The overall behavior of the program can be viewed as moving from state to state as it reads its input.

To make these ideas more concrete, let us consider a specific pattern-matching problem: "What English words contain the five vowels in order?" To help answer this question, we can use a word list that is found with many operating systems. For example, in the UNIX system one can find such a list in the file `/usr/dict/words`, where the commonly used words of English appear one to a line. In this file, some of the words that contain the vowels in order are

```
abstemious
facetious
sacrilegious
```

Let us write a straightforward C program to examine a character string and decide whether all five vowels appear there in order. Starting at the beginning of the string, the program first searches for an `a`. We shall say it is in "state 0" until it sees an `a`, whereupon it goes into "state 1." In state 1, it looks for an `e`, and when it finds one, it goes into "state 2." It proceeds in this manner, until it reaches "state 4," in which it is looking for a `u`. If it finds a `u`, then the word has all five vowels, in order, and the program can go into an accepting "state 5." It is not necessary to scan the rest of the word, since it already knows that the word qualifies, regardless of what follows the `u`.

We can interpret state i as saying that the program has already encountered the first i vowels, in order, for $i = 0, 1, \ldots, 5$. These six states summarize all that the program needs to remember as it scans the input from left to right. For example, in state 0, while it is looking for an `a`, the program does not need to remember if it has seen an `e`. The reason is that such an `e` is not preceded by any `a`, and so cannot serve as the `e` in the subsequence `aeiou`.

The heart of this pattern-recognition algorithm is the function `findChar(pp,c)` in Fig. 10.2. This function's arguments are `pp` — the address of a pointer to a string of characters — and a desired character c. That is, `pp` is of type "pointer to pointer to character." Function `findChar` searches for the character c, and as a side effect it advances the pointer whose address it is given until that pointer either points past c or to the end of the string. It returns a value of type `BOOLEAN`, which we define to be a synonym for `int`. As discussed in Section 1.6, we expect that the only values for the type `BOOLEAN` will be `TRUE` and `FALSE`, which are defined to be 1 and 0, respectively.

At line (1), `findChar` examines the current character indicated by `pp`. If it is neither the desired character c nor the character `'\0'` that marks the end of a character string in C, then at line (2) we advance the pointer that is pointed to by `pp`. A test at line (3) determines whether we stopped because we exhausted the string. If we did, we return `FALSE`; otherwise, we advance the pointer and return `TRUE`.

Next in Fig. 10.2 is the function `testWord(p)` that tells whether a character string pointed to by `p` has all the vowels in order. The function starts out in state 0, just before line (7). In that state it calls `findChar` at line (7), with second argument `'a'`, to search for the letter `a`. If it finds an `a`, then `findChar` will return `TRUE`. Thus if `findChar` returns `TRUE` at line (7), the program moves to state 1, where at line (8) it makes a similar test for an `e`, scanning the string starting after the first `a`. It thus proceeds through the vowels, until at line (12), if it finds a `u` it returns `TRUE`. If any of the vowels are not found, then control goes to line (13), where `testWord` returns `FALSE`.

The main program of line (14) tests the particular string `"abstemious"`. In practice, we might use `testWord` repeatedly on all the words of a file to find those with all five vowels in order.

Graphs Representing State Machines

We can represent the behavior of a program such as Fig. 10.2 by a graph in which the nodes represent the states of the program. What is perhaps more important, we can design a program by first designing the graph, and then mechanically translating the graph into a program, either by hand, or by using one of a number of programming tools that have been written for that purpose.

```
      #include <stdio.h>

      #define TRUE 1
      #define FALSE 0
      typedef int BOOLEAN;

      BOOLEAN findChar(char **pp, char c)
      {
(1)       while (**pp != c && **pp != '\0')
(2)           (*pp)++;
(3)       if (**pp == '\0')
(4)           return FALSE;
          else {
(5)           (*pp)++;
(6)           return TRUE;
          }
      }

      BOOLEAN testWord(char *p)
      {
          /* state 0 */
(7)       if (findChar(&p, 'a'))
              /* state 1 */
(8)           if (findChar(&p, 'e'))
                  /* state 2 */
(9)               if (findChar(&p, 'i'))
                      /* state 3 */
(10)                  if (findChar(&p, 'o'))
                          /* state 4 */
(11)                      if (findChar(&p, 'u'))
                              /* state 5 */
(12)                          return TRUE;
(13)      return FALSE;
      }

      main()
      {
(14)      printf("%d\n", testWord("abstemious"));
      }
```

Fig. 10.2. Finding words with subsequence aeiou.

Transition

The picture representing a program's states is a directed graph, whose arcs are labeled by sets of characters. There is an arc from state s to state t, labeled by the set of characters C, if, when in state s, we go to state t exactly when we see one of the characters in set C. The arcs are called *transitions.* If x is one of the characters in set C, which labels the transition from state s to state t, then if we are in state s and receive an x as our next character, we say we "make a transition on x to state t." In the common case that set C is a singleton $\{x\}$, we shall label the arc by x, rather than $\{x\}$.

Accepting state and start state

We also label certain of the nodes *accepting states.* When we reach one of these

states, we have found our pattern and "accept." Conventionally, accepting states are represented by double circles. Finally, one of the nodes is designated the *start state,* the state in which we begin to recognize the pattern. We indicate the start state by an arrow entering from nowhere. Such a graph is called a *finite automaton* or just *automaton.* We see an example of an automaton in Fig. 10.3.

Automaton

The behavior of an automaton is conceptually simple. We imagine that an automaton receives a list of characters known as the *input sequence.* It begins in the start state, about to read the first character of the input sequence. Depending on that character, it makes a transition, perhaps to the same state, or perhaps to another state. The transition is dictated by the graph of the automaton. The automaton then reads the second character and makes the proper transition, and so on.

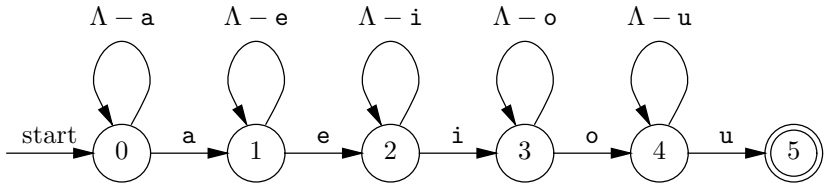


Fig. 10.3. Automaton to recognize sequences of letters that have subsequence `aeiou`.

✦ **Example 10.1.** The automaton corresponding to the function `testWord` of Fig. 10.2 is shown in Fig. 10.3. In this graph, we use a convention that will be followed subsequently; the Greek letter Λ (Lambda) stands for the set of all upper- and lower-case letters. We also use shorthands like Λ − `a` to represent the set of all letters except `a`.

Node 0 is the start state. On any letter but `a`, we stay in state 0, but on an `a`, we go to state 1. Similarly, once we reach state 1, we stay there unless we see an `e`, in which case we go to state 2. The next states, 3 and 4, similarly are reached when we see an `i` and then an `o`. We remain in state 4 unless we see a `u`, in which case we go to state 5, the lone accepting state. There are no transitions out of state 5, since we do not examine any more of the word being tested, but rather announce success by returning `TRUE`.

It is also worth noting that if we encounter a blank (or any other nonletter) in states 0 through 4, we have no transition. In that case, processing stops, and, since we are not now in an accepting state, we have *rejected* the input. ✦

✦ **Example 10.2.** Our next example is from signal processing. Instead of regarding all characters as potential inputs for an automaton, we shall allow only inputs 0 and 1. The particular automaton we shall design, sometimes called a *bounce filter,* takes a sequence of 0's and 1's as inputs. The object is to "smooth" the sequence by regarding a single 0 surrounded by 1's as "noise," and replacing the 0 by 1. Similarly, one 1 surrounded by 0's will be regarded as noise and replaced by 0.

Bounce filter

As an example of how a bounce filter might be used, we could be scanning a digitized black-white image, line by line. Each line of the image is, in fact, a

sequence of 0's and 1's. Since pictures sometimes do have small spots of the wrong color, due, for example, to imperfections in the film or the photography process, it is useful to get rid of such spots, in order to reduce the number of distinct regions in the image and allow us to concentrate on "real" features, rather than spurious ones.

Figure 10.4 is the automaton for our bounce filter. The interpretations of the four states are as follows:

a) We have just seen a sequence of 0's, at least two in a row.
b) We have just seen a sequence of 0's followed by a single 1.
c) We have just seen a sequence of at least two 1's.
d) We have just seen a sequence of 1's followed by a single 0.

State *a* is designated the start state, which implies that our automaton will behave as if there were an unseen prefix of 0's prior to the input.

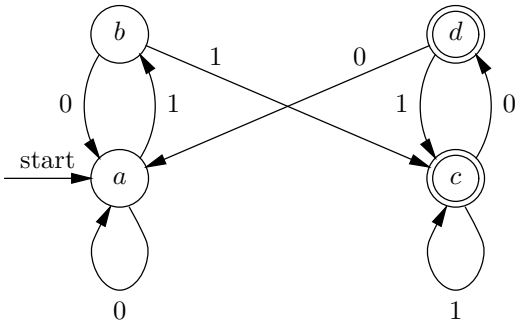


Fig. 10.4. Automaton to eliminate spurious 0's and 1's.

The accepting states are *c* and *d*. For this automaton, acceptance has a somewhat different significance from that of the automaton of Fig. 10.3. There, when we reached the accepting state, we said that the whole input was accepted, including characters the automaton had not even read yet.[1] Here, we want an accepting state to say "output a 1," and a nonaccepting state to say "output a 0." Under this interpretation, we shall translate each bit in the input to a bit of the output. Usually, the output will be the same as the input, but sometimes it will differ. For instance, Fig. 10.5 shows the input, states, and their outputs when the input is 0101101.

Input:	0	1	0	1	1	0	1
State:	*a*	*a*	*b*	*a*	*b*	*c*	*d*	*c*
Output:	0	0	0	0	0	1	1	1

Fig. 10.5. Simulation of the automaton in Fig. 10.4 on input 0101101.

[1] However, we could have modified the automaton to read all letters following the `u`, by adding a transition from state 5 to itself on all letters.

The Difference Between Automata and Their Programs

Automata are abstractions. As will be clear from Section 10.3, automata render an accept/reject decision on any sequence of input characters by seeing whether there is a path from the start state to some accepting state labeled by that sequence. Thus for example, the action indicated in Fig. 10.5 for the bounce-filter automaton of Fig. 10.4 tells us that the automaton rejects the prefixes ϵ, `0`, `01`, `010`, and `0101`, while it accepts `01011`, `010110`, and `0101101`. The automaton of Fig. 10.3 accepts strings like `abstemiou`, but rejects `abstemious`, because there is nowhere to go on the final `s` from state 5.

On the other hand, programs created from automata can use the accept/reject decision in various ways. For example, the program of Fig. 10.2 used the automaton of Fig. 10.3 not to approve of the string that labels the path to the accepting state, but to approve of the entire line of input, that is, `abstemious` instead of `abstemiou`. That is perfectly reasonable, and reflects the way we'd write a program to test for all five vowels in order, regardless of whether we used automata or any other approach. Presumably, as soon as we got to the `u`, our program would print the entire word without examining it further.

The automaton of Fig. 10.4 is used in a more straightforward way. We shall see in Fig. 10.7 a program for the bounce filter that simply translates each accepting state into an action to print a `1`, and translates each rejecting state into an action to print a `0`.

We begin in state a, and since a is nonaccepting, we output 0. It is important to notice that this initial output is not in response to any input, but represents the condition of the automaton when we first turn the device on.

The transition out of state a labeled by input 0 in Fig. 10.4 is to state a itself. Thus the second output is also 0. The second input is 1, and from state a we make a transition on 1 to state b. That state "remembers" that we've seen a single 1, but since b is a nonaccepting state, the output is still 0. On the third input, another 0, we go from state b back to a, and we continue to emit the output 0.

The next two inputs are 1's, which take the automaton first to state b, then to state c. On the first of the two 1's, we find ourselves in state b, which causes output 0; that output turns out to be wrong, because we have in fact started a run of 1's, but we don't know that after reading the fourth input. The effect of our simple design is that all runs, whether of 0's or 1's, are shifted one position to the right, since it takes two bits in a row before the automaton realizes it has seen the beginning of a new run, rather than a "noise" bit. When the fifth input is received, we follow the transition on input 1 from state b to state c. At that point, we make our first 1 output, because c is an accepting state.

The last two inputs are 0 and 1. The 0 takes us from state c to d, so that we can remember that we have seen a single 0. The output from state d is still 1, since that state is accepting. The final 1 takes us back to state c and produces a 1 output. ❖

EXERCISES

10.2.1: Design automata to read strings of 0's and 1's, and

a) Determine if the sequence read so far has even parity (i.e., there have been an even number of 1's). Specifically, the automaton accepts if the string so far has even parity and rejects if it has odd parity.

b) Check that there are no more than two consecutive 1's. That is, accept unless 111 is a substring of the input string read so far.

What is the intuitive meaning of each of your states?

10.2.2: Indicate the sequence of states and the outputs when your automata from Exercise 10.2.1 are given the input 101001101110.

10.2.3*: Design an automaton that reads a word (character string) and tells whether the letters of the word are in sorted order. For example, `adept` and `chilly` have their letters in sorted order; `baby` does not, because an `a` follows the first `b`. The word must be terminated by a blank, so that the automaton will know when it has read all the characters. (Unlike Example 10.1, here we must not accept until we have seen all the characters, that is, until we reach the blank at the end of the word.) How many states do you need? What are their intuitive meanings? How many transitions are there out of each state? How many accepting states are there?

10.2.4: Design an automaton that tells whether a character string is a legal C identifier (letter followed by letters, digits, or underscore) followed by a blank.

10.2.5: Write C programs to implement each of the automata of Exercises 10.2.1 through 10.2.4.

10.2.6: Design an automaton that tells whether a given character string is one of the third-person singular pronouns, `he`, `his`, `him`, `she`, `her`, or `hers`, followed by a blank.

10.2.7*: Convert your automaton from Exercise 10.2.6 into a C function and use it in a program to find all places where the third-person singular pronouns appear as substrings of a given string.

10.3 Deterministic and Nondeterministic Automata

One of the most basic operations using an automaton is to take a sequence of symbols $a_1a_2\cdots a_k$ and follow from the start state a path whose arcs have labels that include these symbols in order. That is, for $i = 1, 2, \ldots, k$, a_i is a member of the set S_i that labels the ith arc of the path. Constructing this path and its sequence of states is called *simulating* the automaton on the input sequence $a_1a_2\cdots a_k$. This path is said to have the *label* $a_1a_2\cdots a_k$; it may also have other labels, of course, since the sets S_i labeling the arcs along the path may each include many characters.

Label of a path

Example 10.3. We did one such simulation in Fig. 10.5, where we followed the automaton of Fig. 10.4 on input sequence 0101101. For another example, consider the automaton of Fig. 10.3, which we used to recognize words with subsequence `aeiou`. Consider the character string `adept`.

We start in state 0. There are two transitions out of state 0, one on the set of characters $\Lambda - \mathtt{a}$, and the other on `a` alone. Since the first character in `adept` is `a`,

Terminology for Automaton Inputs

In the examples we shall discuss here, the inputs to an automaton are characters, such as letters and digits, and it is convenient to think of inputs as characters and input sequences as character strings. We shall generally use that terminology here, but we shorten "character string" to just "string" on occasion. However, there are some applications where the inputs on which an automaton makes a transition are chosen from a set more general than the ASCII character set. For instance, a compiler may want to consider a keyword such as `while` to be a single input symbol, which we shall represent by the boldface string **while**. Thus we shall sometimes refer to the individual inputs as "symbols" rather than "characters."

we follow the latter transition, which takes us to state 1. Out of state 1, there are transitions on $\Lambda - \texttt{e}$ and `e`. Since the second character is `d`, we must take the former transition, because all letters but `e` are included in the set $\Lambda - \texttt{e}$. That leaves us in state 1 again. As the third character is `e`, we follow the second transition out of state 1, which takes us to state 2. The final two letters of `adept` are both included in the set $\Lambda - \texttt{i}$, and so our next two transitions are from state 2 to state 2. We thus finish our processing of `adept` in state 2. The sequence of state transitions is shown in Fig. 10.6. Since state 2 is not an accepting state, we do not accept the input `adept`. ✦

Input:		`a`	`d`	`e`	`p`	`t`
State:	0	1	1	2	2	2

Fig. 10.6. Simulation of the automaton in Fig. 10.3 on input `adept`.

Deterministic Automata

The automata discussed in the previous section have an important property. For any state s and any input character x, there is at most one transition out of state s whose label includes x. Such an automaton is said to be *deterministic.*

Simulating a deterministic automaton on a given input sequence is straightforward. In any state s, given the next input character x, we consider each of the labels of the transitions out of s. If we find a transition whose label includes x, then that transition points to the proper next state. If none includes x, then the automaton "dies," and cannot process any more input, just as the automaton of Fig. 10.3 dies after it reaches state 5, because it knows it already has found the subsequence `aeiou`.

It is easy to convert a deterministic automaton into a program. We create a piece of code for each state. The code for state s examines its input and decides which of the transitions out of s, if any, should be followed. If a transition from state s to state t is selected, then the code for state s must arrange for the code of state t to be executed next, perhaps by using a goto-statement.

✦ **Example 10.4.** Let us write a function `bounce()` corresponding to the automaton in Fig. 10.4, the bounce filter. There is a variable `x` used to read characters from the input. States a, b, c, and d will be represented by labels `a`, `b`, `c`, and `d`, respectively, and we use label `finis` for the end of the program, which we reach when we encounter a character other than 0 or 1 on the input.

```
void bounce()
{
          char x;

          /* state a */
a:        putchar('0');
          x = getchar();
          if (x == '0') goto a; /* transition to state a */
          if (x == '1') goto b; /* transition to state b */
          goto finis;

          /* state b */
b:        putchar('0');
          x = getchar();
          if (x == '0') goto a; /* transition to state a */
          if (x == '1') goto c; /* transition to state c */
          goto finis;

          /* state c */
c:        putchar('1');
          x = getchar();
          if (x == '0') goto d; /* transition to state d */
          if (x == '1') goto c; /* transition to state c */
          goto finis;

          /* state d */
d:        putchar('1');
          x = getchar();
          if (x == '0') goto a; /* transition to state a */
          if (x == '1') goto c; /* transition to state c */
          goto finis;

finis:    ;

}
```

Fig. 10.7. Function implementing the deterministic automaton of Fig. 10.4.

The code is shown in Fig. 10.7. For instance, in state a we print the character 0, because a is a nonaccepting state. If the input character is 0, we stay in state a, and if the input character is 1, we go to state b. ✦

Disjoint sets

There is nothing in the definition of "automaton" that requires the labels of the transitions out of a given state to be disjoint (sets are *disjoint* if they have no members in common; i.e., their intersection is the empty set). If we have the sort of graph suggested in Fig. 10.8, where on input x there are transitions from state s to states t and u, it is not clear how this automaton could be implemented by a program. That is, when executing the code for state s, if x is found to be the next input character, we are told we must next go to the beginning of the code for state t and also to the beginning of the code for state u. Since the program cannot go to two places at once, it is far from clear how one simulates an automaton with nondisjoint labels on the transitions out of a state.

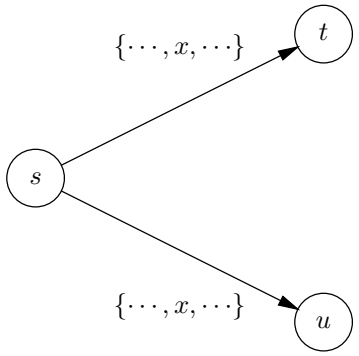


Fig. 10.8. Nondeterministic transition from state s on input x.

Nondeterministic Automata

Nondeterministic automata are allowed (but not required) to have two or more transitions containing the same symbol out of one state. Note that a deterministic automaton is technically a nondeterministic automaton as well, one that happens not to have multiple transitions on one symbol. An "automaton" is in general nondeterministic, but we shall use "nondeterministic automaton" when we want to emphasize that the automaton need not be a deterministic automaton.

Nondeterministic automata are not directly implementable by programs, as we mentioned, but they are useful conceptual tools for a number of applications that we shall discuss. Moreover, by using the "subset construction," to be covered in the next section, it is possible to convert any nondeterministic automaton to a deterministic automaton that accepts the same set of character strings.

Acceptance by Nondeterministic Automata

When we try to simulate a nondeterministic automaton on an input string of characters $a_1a_2\cdots a_k$, we may find that this same string labels many paths. It is conventional to say that the nondeterministic automaton *accepts* this input string if at least one of the paths it labels leads to acceptance. Even a single path ending in an accepting state outweighs any number of paths that end at a nonaccepting state.

Nondeterminism and Guessing

A useful way to look at nondeterminism is that it allows an automaton to "guess." If we don't know what to do in a given state on a given input character, we may make several choices of next state. Since any path labeled by a character string leading to an accepting state is interpreted as acceptance, the nondeterministic automaton in effect is given the credit for a right guess, no matter how many wrong guesses it also makes.

✦ **Example 10.5.** The League Against Sexist Speech (LASS) wishes to catch sexist writing that contains the word "man." They not only want to catch constructs such as "ombudsman," but more subtle forms of discrimination such as "maniac" or "emancipate." LASS plans to design a program using an automaton; that program will scan character strings and "accept" when it finds the character string `man` anywhere within the input.

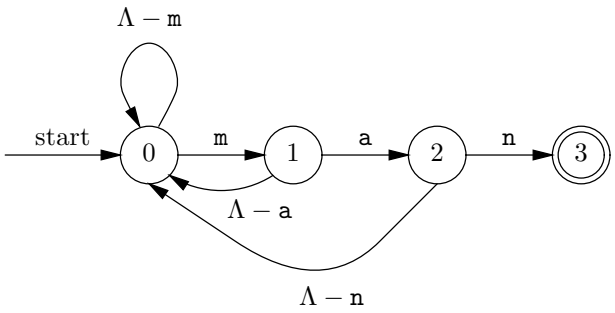


Fig. 10.9. Deterministic automaton that recognizes most, but not all, strings ending in `man`.

One might first try a deterministic automaton like that shown in Fig. 10.9. In this automaton, state 0, the start state, represents the case in which we have not begun to see the letters of "man." State 1 is intended to represent the situation in which we have seen an `m`; in state 2 we have recognized `ma`, and in state 3 we have seen `man`. In states 0, 1, and 2, if we fail to see the hoped-for letter, we go back to state 0 and try again.

However, Fig. 10.9 does not quite work correctly. On an input string such as `command`, it stays in state 0 while reading the `c` and `o`. It goes to state 1 on reading the first `m`, but the second `m` takes it back to state 0, which it does not subsequently leave.

A nondeterministic automaton that correctly recognizes character strings with an embedded `man` is shown in Fig. 10.10. The key innovation is that in state 0 we guess whether an `m` marks the beginning of `man` or not. Since the automaton is nondeterministic, it is allowed to guess both "yes" (represented by the transition from state 0 to state 1) and "no" (represented by the fact that the transition from state 0 to state 0 can be performed on all letters, including `m`) at the same time.

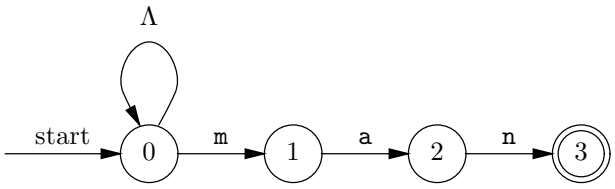


Fig. 10.10. Nondeterministic automaton that recognizes all strings ending in `man`.

Because acceptance by a nondeterministic automaton requires no more than one path to an accepting state, we get the benefit of both guesses.

Figure 10.11 shows the action of the nondeterministic automaton of Fig. 10.10 on input string `command`. In response to the `c` and `o`, the automaton can only stay in state 0. When the first `m` is input, the automaton has the choice of going to state 0 or to state 1, and so it does both. With the second `m`, there is nowhere to go from state 1, and so that branch "dies." However, from state 0, we can again go to either state 0 or 1, and so we do both. When the `a` is input, we can go from state 0 to 0 and from state 1 to 2. Similarly, when `n` is input, we can go from 0 to 0 and from 2 to 3.

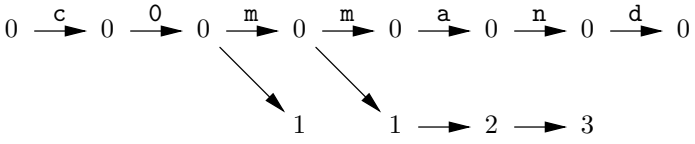


Fig. 10.11. Simulation of nondeterministic automaton of Fig. 10.10 on input string `command`.

Since state 3 is an accepting state, we accept at that point.[2] The fact that we are also in state 0 after seeing `comman` is irrelevant as far as acceptance is concerned. The final transition is on input `d`, from state 0 to 0. Note that state 3 goes nowhere on any input, and so that branch dies.

Also note that the transitions back to state 0, which were present in Fig. 10.9 to handle the case in which the next character of the word `man` was not received, are unnecessary in Fig. 10.10, because in Fig. 10.10 we are not compelled to follow the sequence from state 0 to 1 to 2 to 3 when we see input `man`. Thus, although state 3 looks like it is "dead" and ends computation when we see `man`, we are also in state 0 upon seeing `man`. That state allows us to accept inputs like `manoman` by staying in state 0 during the first `man` and going through states 1, 2, and 3 when the second `man` is read. ✥

[2] Notice that the automaton of Fig. 10.10, like that of Fig. 10.3, accepts when it sees the pattern it is looking for, not at the end of the word. When we eventually convert Fig. 10.10 to a deterministic automaton, we can design from it a program that prints the entire word, like the program of Fig. 10.2.

Of course, the design of Fig. 10.10, while appealing, cannot be turned into a program directly. We shall see in the next section how it is possible to turn Fig. 10.10 into a deterministic automaton with only four states. This deterministic automaton, unlike that of Fig. 10.9, will correctly recognize all occurrences of `man`.

While we can convert any nondeterministic automaton into a deterministic one, we are not always as fortunate as we are in the case of Fig. 10.10. In that case, the corresponding deterministic automaton will be seen to have no more states than the nondeterministic automaton, four states for each. There are other nondeterministic automata whose corresponding deterministic automata have many more states. A nondeterministic automaton with n states might be convertible only into a deterministic automaton with 2^n states. The next example happens to be one in which the deterministic automaton has many more states than the nondeterministic one. Consequently, a nondeterministic automaton may be considerably easier to design than a deterministic automaton for the same problem.

✦ **Example 10.6.** When Peter Ullman, the son of one of the authors, was in fourth grade, he had a teacher who tried to build the students' vocabularies by assigning them "partial anagram" problems. Each week they would be given a word and were asked to find all the words that could be made using one or more of its letters.

Partial anagram

One week, when the word was "Washington," the two authors of this book got together and decided to do an exhaustive search to see how many words were possible. Using the file `/usr/dict/words` and a three-step procedure, we found 269 words. Among them were five 7-letter words:

```
agonist
goatish
showing
washing
wasting
```

Since the case of a letter is not significant for this problem, our first step was to translate all upper-case letters in the dictionary into lower case. A program to carry out this task is straightforward.

Our second step was to select the words that contain only characters from the set $S = \{$`a,g,h,i,n,o,s,t,w`$\}$, the letters in `washington`. A simple, deterministic automaton can do this task; one is shown in Fig 10.12. The **newline** character is the character that marks the ends of lines in `/usr/dict/words`. In Fig. 10.12, we stay in state 0 as long as we see letters that appear in `washington`. If we encounter any other character besides **newline**, there is no transition, and the automaton can never reach the accepting state 1. If we encounter **newline** after reading only letters in `washington`, then we make the transition from state 0 to state 1, and accept.

The automaton in Fig. 10.12 accepts words such as `hash` that have more occurrences of some letter than are found in the word `washington` itself. Our third and final step, therefore, was to eliminate those words that contain three or more `n`'s or two or more of another of the characters in set S. This task can also be done by an automaton. For example, the automaton in Fig. 10.13 accepts words that have at least two `a`'s. We stay in state 0 until we see an `a`, whereupon we go to state 1. We stay there until we see a second `a`; at that point we go to state 2 and accept. This automaton accepts those words that fail to be partial anagrams of `washington`

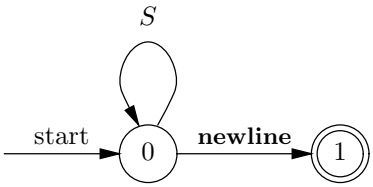


Fig. 10.12. Deterministic automaton that detects words consisting of letters that appear in `washington`.

because they have too many `a`'s. In this case, the words we want are exactly those that never cause the automaton to enter the accepting state 2 at any time during their processing.

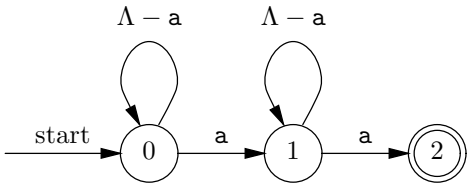


Fig. 10.13. Automaton that accepts if there are two *a*'s.

The automaton of Fig. 10.13 is deterministic. However, it represents only one of nine reasons why a word that is accepted by the automaton of Fig. 10.12 might still not be a partial anagram of `washington`. To accept all the words that have too many instances of a letter in `washington`, we can use the nondeterministic automaton of Fig. 10.14.

Figure 10.14 starts in state 0, and one of its choices on any letter is to remain in state 0. If the input character is any letter in `washington`, then there is another choice; the automaton also guesses that it should transfer to a state whose function it is to remember that one occurrence of this letter has been seen. For instance, on letter `i` we have the choice of going also to state 7. We then remain in state 7 until we see another `i`, whereupon we go to state 8, which is one of the accepting states. Recall that in this automaton, acceptance means that the input string is *not* a partial anagram of `washington`, in this case because it has two `i`'s.

Because there are two `n`'s in `washington`, letter `n` is treated somewhat differently. The automaton can go to state 9 on seeing an `n`, then to state 10 on seeing a second `n`, and then to state 11 on seeing a third `n`, which is where it accepts.

For instance, Fig. 10.15 shows all the states we can enter after reading the input string `shining`. Since we enter accepting state 8 after reading the second `i`, the word `shining` is not a partial anagram of `washington`, even though it is accepted by the automaton of Fig. 10.12 for having only letters that are found in `washington`.

To summarize, our algorithm consisted of three steps:

1. We first translated all upper-case letters in the dictionary into lower-caser letters.

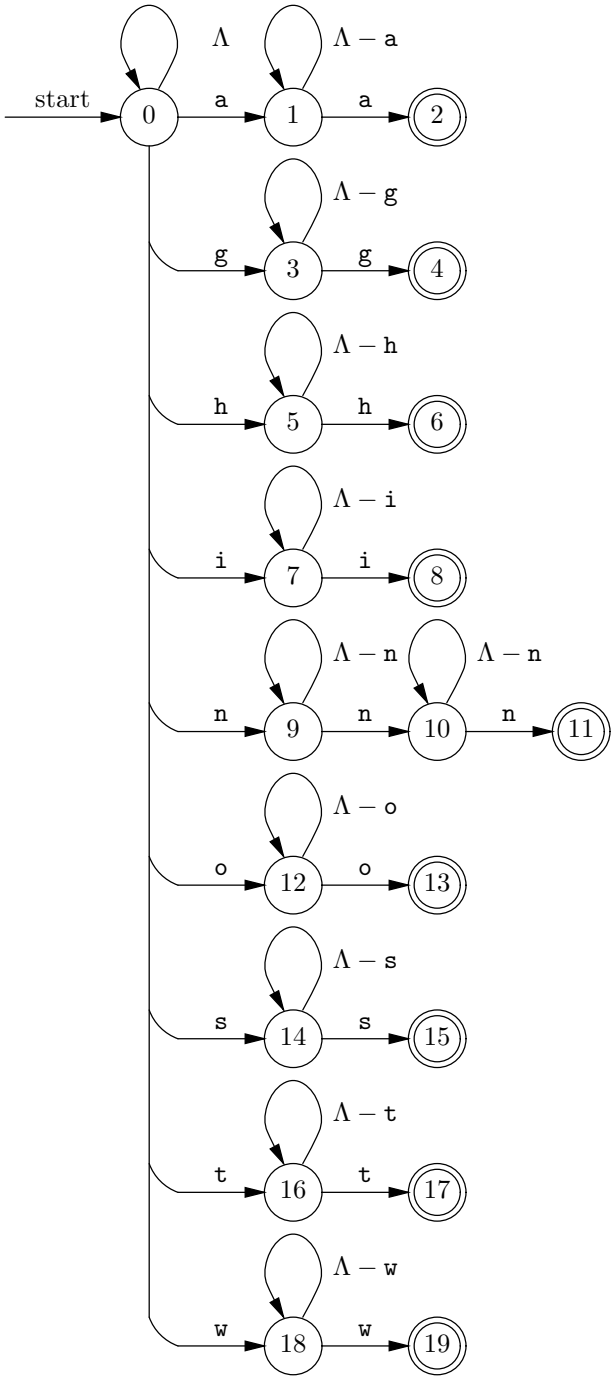


Fig. 10.14. Nondeterministic automaton that detects words with more than one `a`, `g`, `h`, `i`, `o`, `s`, `t`, or `w`, or more than two `n`'s.

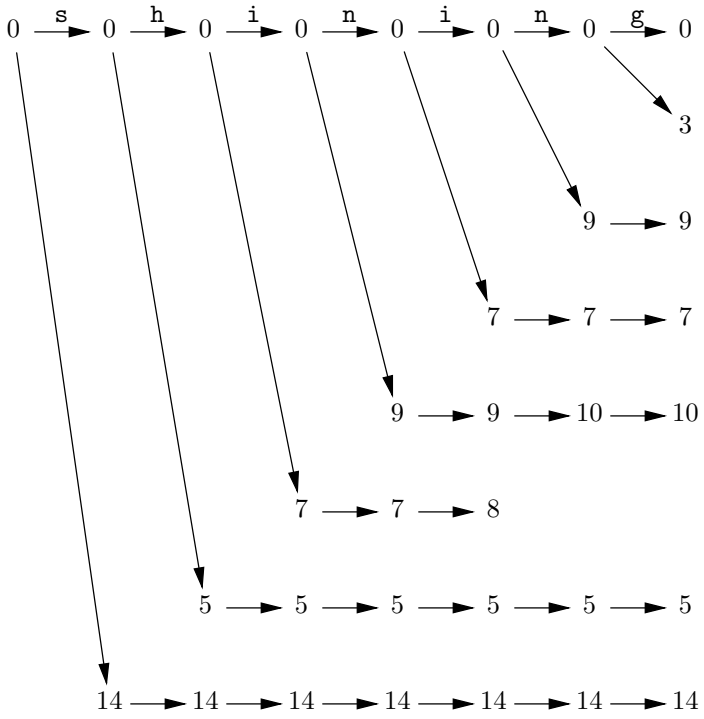


Fig. 10.15. States entered by the nondeterministic automaton of Fig. 10.14 on input string `shining`.

2. We found all resulting words that are accepted by the automaton of Fig. 10.12 and therefore consist only of letters in `washington`.
3. We removed from the list created in (2) all those words that are accepted by the nondeterministic automaton of Fig. 10.14.

This algorithm is a straightforward way to find all the partial anagrams of "Washington" in the file `/usr/dict/words`. Of course, we must find some reasonable way to simulate the nondeterministic automaton of Fig. 10.14, and we shall discuss how that can be done in the next sections. ❖

EXERCISES

10.3.1: Write a C program to implement the algorithm of the deterministic automaton of Fig. 10.9.

10.3.2: Design a deterministic automaton that correctly finds all occurrences of `man` in a character string. Implement this automaton as a program.

10.3.3: LASS wishes to detect all occurrences of the strings `man`, `son`, and `father`. Design a nondeterministic automaton that will accept whenever it finds the end of any of these three character strings.

10.3.4*: Design a deterministic automaton that solves the same problem as the previous exercise.

Finding Partial Anagrams Without Programming

As an aside, we were able to implement the three-step algorithm in Example 10.6 with almost no programming, using the commands of the UNIX system. For step (1) we used the UNIX command

```
tr A-Z a-z </usr/dict/words                                    (10.1)
```

to translate from upper to lower case. For step (2) we used the command

```
egrep '^[aghinostw]*$'                                         (10.2)
```

which, roughly, defines an automaton like that of Fig. 10.12. For step (3) we used the command

```
egrep -v 'a.*a|g.*g|h.*h|i.*i|n.*n.*n|o.*o|s.*s|t.*t|w.*w'     (10.3)
```

which specifies something like the automaton of Fig. 10.14. The entire task was done using the three-element pipe

```
(10.1) | (10.2) | (10.3)
```

That is, the entire command is formed by substituting the text represented by each of the indicated lines. The vertical bar, or "pipe" symbol, makes the output of the command on its left be the input to the command on its right. The `egrep` command is discussed in Section 10.6.

10.3.5: Simulate the automata of Figs. 10.9 and 10.10 on the input string `summand`.

10.3.6: Simulate the automaton of Fig. 10.14 on input strings

a) `saint`
b) `antagonist`
c) `hashish`

Which are accepted?

10.3.7: We can represent an automaton by a relation with attributes State, Input, and Next. The intent is that if (s, x, t) is a tuple, then input symbol x is a label of the transition from state s to state t. If the automaton is deterministic, what is a suitable key for the relation? What if the automaton is nondeterministic?

10.3.8: What data structures would you suggest for representing the relation of the previous exercise, if we only wanted to find the next state(s), given a state and an input symbol?

10.3.9: Represent the automata of

a) Fig. 10.10
b) Fig. 10.9
c) Fig. 10.14

as relations. You may use ellipses to represent the transitions on large sets of letters such as $\Lambda - \mathtt{m}$.

10.4 From Nondeterminism to Determinism

In this section we shall see that every nondeterministic automaton can be replaced by a deterministic one. As we have seen, it is sometimes easier to think of a nondeterministic automaton to perform a certain task. However, because we cannot write programs from nondeterministic automata as readily as from deterministic machines, it is quite important that there is an algorithm to transform a nondeterministic automaton into an equivalent deterministic one.

Equivalence of Automata

In the previous sections, we have seen two views of acceptance. In some examples, such as Example 10.1 (words containing the subsequence `aeiou`), we took acceptance to mean that the entire word was accepted, even though we may not have scanned the entire word yet. In others, like the bounce filter of Example 10.2, or the automaton of Fig. 10.12 (words whose letters are all in `washington`), we accepted only when we wanted to signal approval of the exact input that we had seen since we started the automaton. Thus in Example 10.2 we accepted all sequences of inputs that result in a 1 output. In Fig. 10.12, we accepted only when we had seen the **newline** character, and thus knew that the entire word had been seen.

When we talk about the formal behavior of automata, we require only the second interpretation (the input so far is accepted). Formally, suppose A and B are two automata (deterministic or not). We say A and B are *equivalent* if they accept the same set of input strings. Put another way, if $a_1a_2\cdots a_k$ is any string of symbols, then the following two conditions hold:

1. If there is a path labeled $a_1a_2\cdots a_k$ from the start state of A to some accepting state of A, then there is also a path labeled $a_1a_2\cdots a_k$ from the start state of B to some accepting state of B, and
2. If there is a path labeled $a_1a_2\cdots a_k$ from the start state of B to some accepting state of B, then there is also a path labeled $a_1a_2\cdots a_k$ from the start state of A to some accepting state of A.

◆ **Example 10.7.** Consider the automata of Figs. 10.9 and 10.10. As we noted from Fig. 10.11, the automaton of Fig. 10.10 accepts the input string `comman`, because this sequence of characters labels the path $0 \to 0 \to 0 \to 0 \to 1 \to 2 \to 3$ in Fig. 10.10, and this path goes from the start state to an accepting state. However, in the automaton of Fig. 10.9, which is deterministic, we can check that the only path labeled `comman` is $0 \to 0 \to 0 \to 1 \to 0 \to 0 \to 0$. Thus if Fig. 10.9 is automaton A, and Fig. 10.10 is automaton B, we have a violation of point (2) above, which tells us that these two automata are not equivalent. ◆

The Subset Construction

We shall now see how to "wring the nondeterminism out of an automaton" by constructing an equivalent deterministic automaton. The technique is called the *subset construction,* and its essence is suggested by Figs. 10.11 and 10.15, in which we simulated nondeterministic automata on particular inputs. We notice from these pictures that at any given time, the nondeterministic automaton is in some set of states, and that these states appear in one column of the simulation diagram. That

is, after reading some input list $a_1a_2\cdots a_k$, the nondeterministic automaton is "in" those states that are reached from the start state along paths labeled $a_1a_2\cdots a_k$.

✦ **Example 10.8.** After reading input string `shin`, the automaton illustrated in Fig. 10.15 is in the set of states $\{0, 5, 7, 9, 14\}$. These are the states that appear in the column just after the first `n`. After reading the next `i`, it is in the set of states $\{0, 5, 7, 8, 9, 14\}$, and after reading the following `n`, it is in set of states $\{0, 5, 7, 9, 10, 14\}$. ✦

We now have a clue as to how we can turn a nondeterministic automaton N into a deterministic automaton D. The states of D will each be a set of N's states, and the transitions among D's states will be determined by the transitions of N. To see how the transitions of D are constructed, let S be a state of D and x an input symbol. Since S is a state of D, it consists of states of N. Define the set T to be those states t of automaton N such that there is a state s in S and a transition of N from s to t on a set containing input symbol x. Then in automaton D we put a transition from S to T on symbol x.

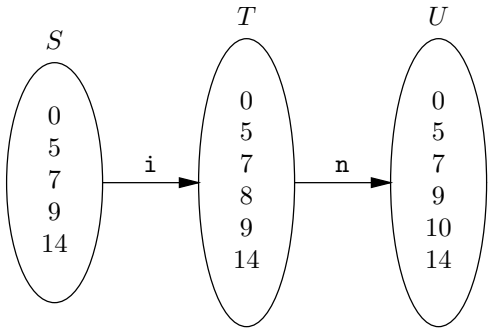


Fig. 10.16. Transitions among deterministic states S, T, and U.

Example 10.8 illustrated transitions from one deterministic state to another on various input symbols. When the current deterministic state is $\{0, 5, 7, 9, 14\}$, and the input symbol is the letter `i`, we saw in that example that the set of next nondeterministic states, according to the nondeterministic automaton of Fig. 10.14, is $T = \{0, 5, 7, 8, 9, 14\}$. From this deterministic state on input symbol `n`, the set of next nondeterministic states U is $\{0, 5, 7, 9, 10, 14\}$. These two deterministic transitions are depicted in Fig. 10.16.

Now we know how to construct a transition between two states of the deterministic automaton D, but we need to determine the exact set of states of D, the start state of D, and the accepting states of D. We construct the states of D by an induction.

BASIS. If the start state of nondeterministic automaton N is s_0, then the start state of deterministic automaton D is $\{s_0\}$, that is, the set containing only s_0.

INDUCTION. Suppose we have established that S, a set of N's states, is a state of D. We consider each possible input character x, in turn. For a given x, we let T be the set of N's states t such that for some state s in S, there is a transition from s to t with a label that includes x. Then set T is a state of D, and there is a transition from S to T on input x.

The accepting states of D are those sets of N's states that include at least one accepting state of N. That makes intuitive sense. If S is a state of D and a set of N's states, then the inputs $a_1a_2\cdots a_k$ that take D from its start state to state S also take N from its start state to all of the states in S. If S includes an accepting state, then $a_1a_2\cdots a_k$ is accepted by N, and D must also accept. Since D enters only state S on receiving input $a_1a_2\cdots a_k$, S must be an accepting state of D.

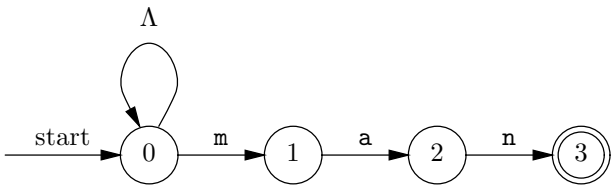


Fig. 10.17. Nondeterministic automaton that recognizes the strings ending in `man`.

◆ **Example 10.9.** Let us convert the nondeterministic automaton of Fig. 10.10, which we reproduce here as Fig. 10.17, to a deterministic automaton D. We begin with $\{0\}$, which is the start state of D.

The inductive part of the construction requires us to look at each state of D and determine its transitions. For $\{0\}$, we have only to ask where state 0 goes. The answer, which we get by examining Fig. 10.17, is that on any letter except `m`, state 0 goes only to 0, while on input `m`, it goes to both 0 and 1. Automaton D therefore needs state $\{0\}$, which it already has, and state $\{0,1\}$, which we must add. The transitions and states constructed so far for D are shown in Fig. 10.18.

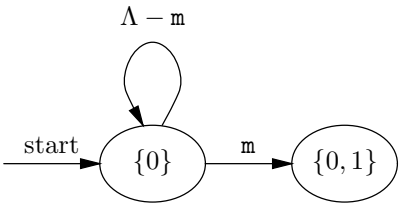


Fig. 10.18. State $\{0\}$ and its transitions.

Next, we must consider the transitions out of $\{0,1\}$. Examining Fig. 10.17 again, we see that on all inputs except `m` and `a`, state 0 goes only to 0, and state 1 goes nowhere. Thus there is a transition from state $\{0,1\}$ to state $\{0\}$, labeled by all the letters except `m` and `a`. On input `m`, state 1 again goes nowhere, but 0 goes to

both 0 and 1. Hence there is a transition from $\{0,1\}$ to itself, labeled by `m`. Finally, on input `a`, state 0 goes only to itself, but state 1 goes to state 2. Thus there is a transition labeled `a` from state $\{0,1\}$ to state $\{0,2\}$. The portion of D constructed so far is shown in Fig. 10.19.

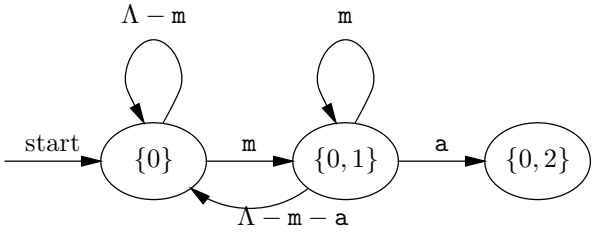


Fig. 10.19. States $\{0\}$ and $\{0,1\}$ and their transitions.

Now we need to construct the transitions out of state $\{0,2\}$. On all inputs except `m` and `n`, state 0 goes only to 0, and state 2 goes nowhere, and so there is a transition from $\{0,2\}$ to $\{0\}$, labeled by all the letters except `m` and `n`. On input `m`, state 2 goes nowhere, and 0 goes to both 0 and 1, and so there is a transition from $\{0,2\}$ to $\{0,1\}$ labeled `m`. On input `n`, state 0 goes only to itself and 2 goes to 3. Thus there is a transition labeled `n` from $\{0,2\}$ to $\{0,3\}$. This state of D is accepting, since it includes the accepting state of Fig. 10.17, state 3.

Finally, we must supply the transitions out of $\{0,3\}$. Since state 3 goes nowhere on any input, the transitions out of $\{0,3\}$ will reflect the transitions out of 0 only, and thus will go to the same states as state $\{0\}$. As the transitions out of $\{0,3\}$ do not take us to any state of D that we have not already seen, the construction of D is finished. The complete deterministic automaton is shown in Fig. 10.20.

Notice that this deterministic automaton correctly accepts all and only the strings of letters that end in `man`. Intuitively, the automaton will be in state $\{0\}$ whenever the character string so far does not end with any prefix of `man` except the empty string. State $\{0,1\}$ means that the string seen so far ends in `m`, $\{0,2\}$ means that it ends in `ma`, and $\{0,3\}$ means that it ends in `man`. ✥

✥ **Example 10.10.** The nondeterministic automaton of Fig. 10.17 has four states, and its equivalent deterministic automaton of Fig. 10.20 has four states also. It would be nice if all nondeterministic automata converted to small deterministic automata, and many common examples used in compiling of programming languages, for example, do in fact convert to relatively small deterministic automata. Yet there is no guarantee that the deterministic automaton will be small, and a k-state nondeterministic automaton could wind up being converted to a deterministic automaton with as many as 2^k states. That is, the deterministic automaton could have one state for every member of the power set of the set of states of the nondeterministic automaton.

As an example where we get many states, consider the automaton of Fig. 10.14, from the previous section. Since this nondeterministic automaton has 20 states, conceivably the deterministic automaton constructed by the subset construction

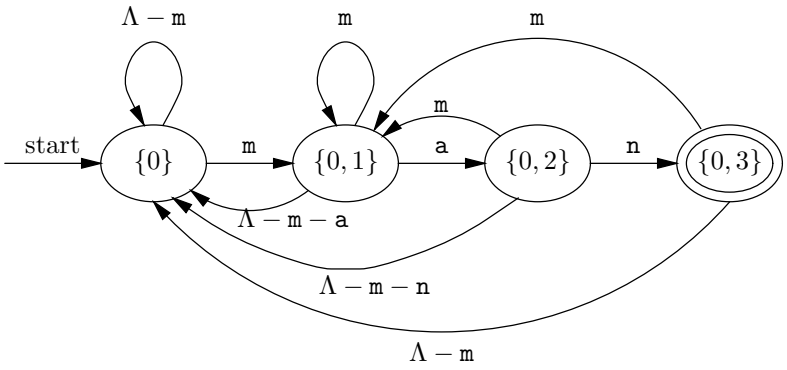


Fig. 10.20. The deterministic automaton D.

A Thought About the Subset Construction

The subset construction is rather tricky to understand. Especially, the idea that states (of the deterministic automaton) can be sets of states (of the nondeterministic automaton) may require some thought and patience to think through. However, this blending of the structured object (the set of states) and the atomic object (a state of the deterministic automaton) into one and the same object is an important concept of computer science. We have seen, and frequently must use, the idea in programs. For example, an argument of a function, say `L`, apparently atomic, may turn out on closer examination also to be an object with complex structure, for example, a record with fields that connect to other records, thus forming a list. Similarly, the state of D that we called $\{0, 2\}$, for instance, could just as well have been replaced by a simple name, such as "5" or "a," in Fig. 10.20.

could have as many as 2^{20} states or over a million states; these states would be all members of the power set of $\{0, 1, \ldots, 19\}$. It turns out not to be that bad, but there are quite a few states.

We shall not attempt to draw the equivalent deterministic automaton for Fig. 10.14. Rather, let us think about what sets of states we actually need. First, since there is a transition from state 0 to itself on every letter, all sets of states that we actually see will include 0. If the letter `a` has not yet been input, then we cannot get to state 1. However, if we have seen exactly one `a`, we shall be in state 1, no matter what else we have seen. We can make an analogous observation about any of the other letters in `washington`.

If we start Fig. 10.14 in state 0 and feed it a sequence of letters that are a subset of the letters appearing in `washington`, then in addition to being in state 0, we shall also be in some subset of the states 1, 3, 5, 7, 9, 12, 14, 16, and 18. By choosing the input letters properly, we can arrange to be in any of these sets of states. As there are $2^9 = 512$ such sets, there are at least that number of states in the deterministic automaton equivalent to Fig. 10.14.

However, there are even more states, because letter `n` is treated specially in

Fig. 10.14. If we are in state 9, we can also be in state 10, and in fact we shall be in both 9 and 10 if we have seen two `n`'s. Thus, while for the other eight letters we have two choices (e.g., for letter `a`, either include state 1 or don't), for letter `n` we have three choices (include neither of 9 and 10, include 9 only, or include both 9 and 10). Thus there are at least $3 \times 2^8 = 768$ states.

But that is not all. If the input so far ends in one of the letters of `washington`, and we previously saw enough of that letter, then we shall also be in the accepting state corresponding to that letter (e.g., state 2 for `a`). However, we cannot be in two accepting states after the same input. Counting the number of additional sets of states becomes trickier.

Suppose accepting state 2 is a member of the set. Then we know 1 is a member of the set, and of course 0 is a member, but we still have all our options for the states corresponding to the letters other than `a`; that number of sets is 3×2^7, or 384. The same applies if our set includes accepting state 4, 6, 8, 13, 15, 17, or 19; in each case there are 384 sets including that accepting state. The only exception is when accepting state 11 is included (and therefore 9 and 10 are also present). Then, there are only $2^8 = 256$ options. The total number of states in the equivalent deterministic automaton is thus

$$768 + 8 \times 384 + 256 = 4864$$

The first term, 768, counts the sets that have no accepting state. The next term counts the eight cases in which the set includes the accepting state for one of the eight letters other than `n`, and the third term, 256, counts the sets that include state 11. ✥

Why the Subset Construction Works

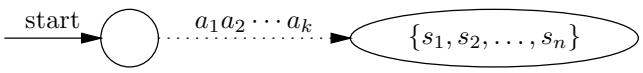
Clearly, if D is constructed from a nondeterministic automaton N using the subset construction, then D is a deterministic automaton. The reason is that for each input symbol x and each state S of D, we defined a specific state T of D such that the label of the transition from S to T includes x. But how do we know that the automata N and D are equivalent? That is, we need to know that for any input sequence $a_1a_2 \cdots a_k$, the state S that the automaton D reaches when we

1. Begin at the start state and
2. Follow the path labeled $a_1a_2 \cdots a_k$

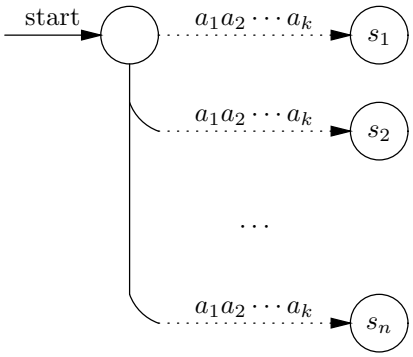
is an accepting state if and only if N accepts $a_1a_2 \cdots a_k$. Remember that N accepts $a_1a_2 \cdots a_k$ if and only if there is some path from N's start state, labeled $a_1a_2 \cdots a_k$, leading to an accepting state of N.

The connection between what D does and what N does is even stronger. If D has a path from its start state to state S labeled $a_1a_2 \cdots a_k$, then set S, thought of as a set of states of N, is exactly the set of states reached from the start state of N, along some path labeled $a_1a_2 \cdots a_k$. The relationship is suggested by Fig. 10.21. Since we have defined S to be an accepting state of D exactly when one of the members of S is an accepting state of N, the relationship suggested by Fig. 10.21 is all we need to conclude that either both or neither of D and N accept $a_1a_2 \cdots a_k$; that is, D and N are equivalent.

We need to prove the relationship of Fig. 10.21; the proof is an induction on k, the length of the input string. The formal statement to be proved by induction on k is that the state $\{s_1, s_2, \ldots, s_n\}$ reached in D by following the path labeled



(a) In automaton D.



(b) In automaton N.

Fig. 10.21. Relationship between the operation of nondeterministic automaton N and its deterministic counterpart D.

$a_1a_2\cdots a_k$ from the start state of D is exactly the set of states of N that are reached from N's start state by following some path labeled $a_1a_2\cdots a_k$.

BASIS. Let $k = 0$. A path of length 0 leaves us where we started, that is, in the start state of both automata D and N. Recall that if s_0 is the start state of N, then the start state of D is $\{s_0\}$. Thus the inductive statement holds for $k = 0$.

INDUCTION. Suppose the statement holds for k, and consider an input string

$$a_1a_2\cdots a_ka_{k+1}$$

Then the path from the start state of D to state T labeled by $a_1a_2\cdots a_ka_{k+1}$ appears as shown in Fig. 10.22; that is, it goes through some state S just before making the last transition to T on input a_{k+1}.

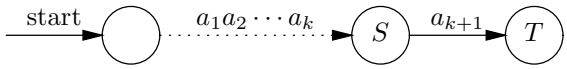


Fig. 10.22. S is the state reached by D just before reaching state T.

We may assume, by the inductive hypothesis, that S is exactly the set of states that automaton N reaches from its start state along paths labeled $a_1a_2\cdots a_k$, and we must prove that T is exactly the set of states N reaches from its start state along paths labeled $a_1a_2\cdots a_ka_{k+1}$. There are two parts to the proof of this inductive step.

1. We must prove that T does not contain too much; that is, if t is a state of N that is in T, then t is reached by a path labeled $a_1a_2\cdots a_ka_{k+1}$ from N's start state.

2. We must prove that T contains enough; that is, if t is a state of N reached from the start state along a path labeled $a_1a_2\cdots a_ka_{k+1}$, then t is in T.

For (1), let t be in T. Then, as suggested by Fig. 10.23, there must be a state s in S that justifies t being in T. That is, there is in N a transition from s to t, and its label includes a_{k+1}. By the inductive hypothesis, since s is in S, there must be a path from the start state of N to s, labeled $a_1a_2\cdots a_k$. Thus there is a path from the start state of N to t, labeled $a_1a_2\cdots a_ka_{k+1}$.

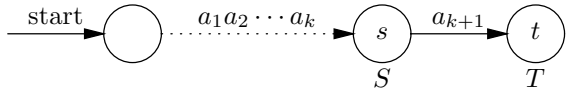


Fig. 10.23. State s in S explains why we put t in T.

Now we must show (2), that if there is a path from the start state of N to t, labeled $a_1a_2\cdots a_ka_{k+1}$, then t is in T. This path must go through some state s just before making a transition to t on input a_{k+1}. Thus there is a path from the start state of N to s, labeled $a_1a_2\cdots a_k$. By the inductive hypothesis, s is in set of states S. Since N has a transition from s to t, with a label that includes a_{k+1}, the subset construction applied to set of states S and input symbol a_{k+1}, demands that t be placed in T. Thus t is in T.

Given the inductive hypothesis, we have now shown that T consists of exactly the states of N that are reachable from the start state of N along some path labeled $a_1a_2\cdots a_ka_{k+1}$. That is the inductive step, and we conclude that the state of the deterministic automaton D reached along the path labeled $a_1a_2\cdots a_k$ is always the set of N's states reachable along some path with that label. Since the accepting states of D are those that include an accepting state of N, we conclude that D and N accept the same strings; that is, D and N are equivalent, and the subset construction "works."

EXERCISES

10.4.1: Convert your nondeterministic automaton of Exercise 10.3.3 to a deterministic automaton, using the subset construction.

10.4.2: What patterns do the nondeterministic automata of Fig. 10.24(a) to (d) recognize?

10.4.3: Convert the nondeterministic automata of Fig. 10.24(a) to (d) to deterministic finite automata.

Minimization of Automata

One of the issues concerning automata, especially when they are used to design circuits, is how few states are needed to perform a given task. That is, we may ask, given an automaton, whether there is an equivalent automaton with fewer states, and if so, what is the least number of states of any equivalent automaton?

It turns out that if we restrict ourselves to deterministic automata, there is a unique minimum-state deterministic automaton equivalent to any given automaton, and it is fairly easy to find it. The key is to define when two states s and t of a deterministic automaton are *equivalent,* that is, for any input sequence, the paths from s and t labeled by that sequence either both lead to accepting states or neither does. If states s and t are equivalent, then there is no way to tell them apart by feeding inputs to the automaton, and so we can merge s and t into a single state. Actually, we can more easily define when states are not equivalent, as follows.

Equivalent states

BASIS. If s is an accepting state and t is not accepting, or vice versa, then s and t are not equivalent.

INDUCTION. If there is some input symbol x such that there are transitions from states s and t on input x to two states that are known not to be equivalent, then s and t are not equivalent.

There are some additional details necessary to make this test work; in particular, we may have to add a "dead state," which is not accepting and has transitions to itself on every input. As a deterministic automaton may have no transition out of a given state on a given symbol, before performing this minimization procedure, we need to add transitions to the dead state from any state, on all inputs for which no other transition exists. We note that there is no similar theory for minimizing nondeterministic automata.

Dead state

10.4.4*: Some automata have state-input combinations for which there is no transition at all. If state s has no transition on symbol x, we can add a transition from s to a special "dead state" on input x. The dead state is not accepting, and has a transition to itself on every input symbol. Show that adding a "dead state" produces an automaton equivalent to the one with which we started.

10.4.5: Show that if we add a dead state to a deterministic automaton, we can get an equivalent automaton that has paths from the start state labeled by every possible string.

10.4.6*: Show that if we apply the subset construction to a deterministic automaton, we either get the same automaton, with each state s renamed $\{s\}$, or we add a dead state (corresponding to the empty set of states).

10.4.7**: Suppose that we take a deterministic automaton and change every accepting state to nonaccepting and every nonaccepting state to accepting.

a) How would you describe the language accepted by the new automaton in terms of the language of the old automaton?

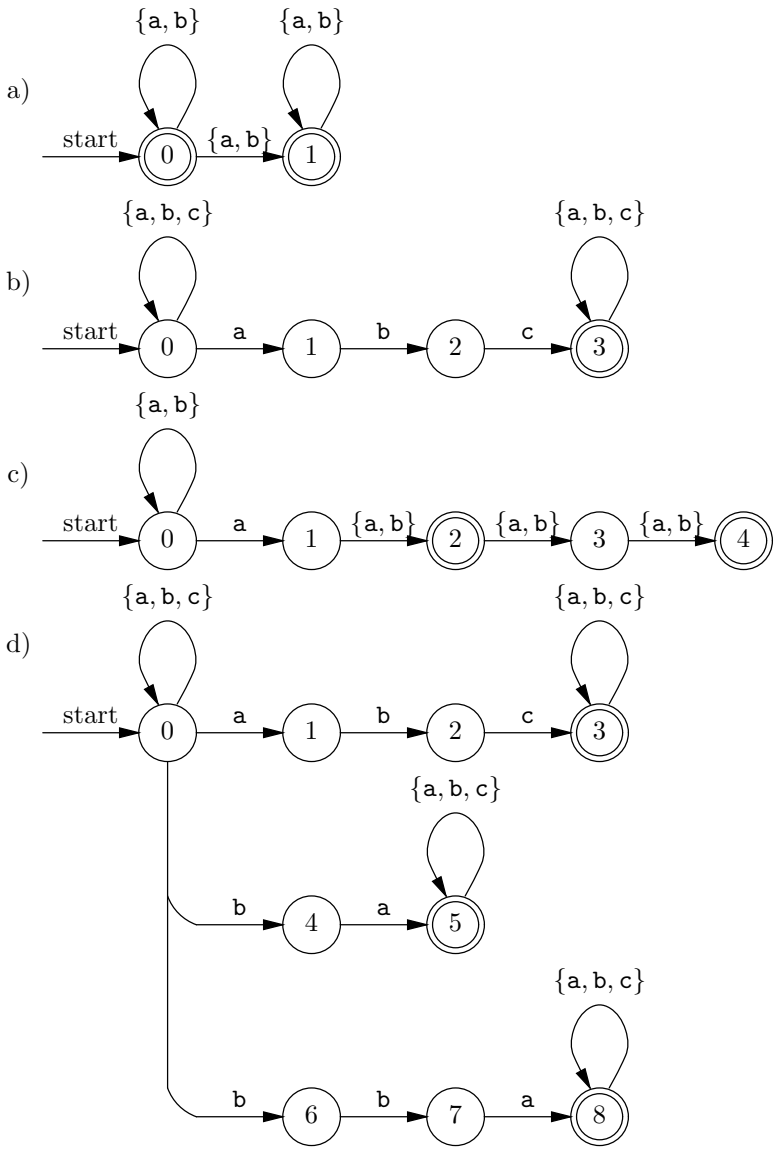


Fig. 10.24. Nondeterministic automata.

b) Repeat (a) if we first add a dead state to the original automaton.
c) Repeat (a) if the original automaton is nondeterministic.

10.5 Regular Expressions

An automaton defines a pattern, namely the set of strings labeling paths from the initial state to some accepting state in the graph of the automaton. In this section, we meet *regular expressions,* which are an algebraic way to define patterns. Regular expressions are analogous to the algebra of arithmetic expressions with which we

Notational Convention

We shall continue to use typewriter font for the characters that appear in strings. The regular expression atomic operand for a given character will be denoted by that character in boldface. For instance, **a** is the regular expression corresponding to character `a`. When we need to use a variable, we shall write it in italics. Variables are used to stand for complicated expressions. For instance, we shall use the variable *letter* to stand for "any letter," a set whose regular expression we shall soon meet.

are all familiar, and the relational algebra that we met in Chapter 8. Interestingly, the set of patterns that can be expressed in the regular-expression algebra is exactly the same set of patterns that can be described by automata.

Operands of Regular Expressions

Like all algebras, regular expressions have certain kinds of atomic operands. In the algebra of arithmetic, atomic operands are constants (e.g., integers or reals) or variables whose possible values are constants, and for relational algebra, atomic operands are either fixed relations or variables whose possible values are relations. In the algebra of regular expressions, an atomic operand is one of the following:

1. A character,
2. The symbol ϵ,
3. The symbol $\emptyset$, or
4. A variable whose value can be any pattern defined by a regular expression.

Values of Regular Expressions

Expressions in any algebra have values of a certain type. For arithmetic expressions, values are integers, reals, or whatever type of numbers with which we are working. For relational algebra, the value of an expression is a relation.

Language of a regular expression

For regular expressions, the value of each expression is a pattern consisting of a set of strings often called a *language*. The language denoted by a regular expression E will be referred to as $L(E)$, or the "language of E." The languages of the atomic operands are defined as follows.

1. If `x` is any character, then the regular expression **x** stands for the language $\{\mathtt{x}\}$; that is, $L(\mathbf{x}) = \{\mathtt{x}\}$. Note that this language is a set that contains one string; the string has length 1, and the lone position of that string has the character `x`.
2. $L(\epsilon) = \{\epsilon\}$. The special symbol ϵ as a regular expression denotes the set whose only string is the empty string, or string of length 0.
3. $L(\emptyset) = \emptyset$. The special symbol $\emptyset$ as a regular expression denotes the empty set of strings.

Note that we do not define a value for an atomic operand that is a variable. Such an operand only takes on a value when we replace the variable by a concrete expression, and its value is then whatever value that expression has.

Operators of Regular Expressions

There are three operators used in regular expressions. These can be grouped using parentheses, as in the algebras with which we are familiar. There is an order of precedence and associativity laws that allow us to omit some pairs of parentheses, just as we do in arithmetic expressions. We shall describe the rules regarding parentheses after we examine the operators.

Union

The first, and most familiar, is the *union* operator, which we shall denote $\mid$.[3] The rule for union is that if R and S are two regular expressions, then $R \mid S$ denotes the union of the languages that R and S denote. That is, $L(R \mid S) = L(R) \cup L(S)$. Recall that $L(R)$ and $L(S)$ are each sets of strings, so the notion of taking their union makes sense.

✦ **Example 10.11.** We know that **a** is a regular expression denoting {a}, and **b** is a regular expression denoting {b}. Thus **a** | **b** is a regular expression denoting {a, b}. That is the set containing the two strings a and b, each string being of length 1.

Similarly, we can write an expression such as (**a** | **b**) | **c** to denote the set {a, b, c}. Since union is an associative operator, that is, it does not matter in what order we group operands when we take the union of three sets, we can omit the parentheses and just write **a** | **b** | **c**. ✦

Concatenation

The second operator for the algebra of regular expressions is called *concatenation*. It is represented by no operator symbol at all, just as multiplication is sometimes written without an operator; for instance, in arithmetic ab denotes the product of a and b. Like union, concatenation is a binary, infix operator. If R and S are regular expressions, then RS is the concatenation of R and S.[4]

$L(RS)$, the language denoted by RS, is formed from the languages $L(R)$ and $L(S)$, as follows. For each string r in $L(R)$ and each string s in $L(S)$, the string rs, the concatenation of strings r and s, is in $L(RS)$. Recall the concatenation of two lists, such as character strings, is formed by taking the elements of the first list, in order, and following them by the elements of the second list, in order.

---

[3] The plus sign + is also commonly used for the union operator in regular expressions; we shall not do so, however.

[4] Technically, we should write RS as $(R)(S)$, to emphasize the fact that R and S are separate expressions and that their parts must not be blended because of precedence rules. The situation is analogous to the fact that if we multiply the arithmetic expression $w + x$ by the arithmetic expression $y + z$, we must write the product as $(w + x)(y + z)$. Note that, because multiplication takes precedence over addition, the product with parentheses omitted, $w + xy + z$, would not be interpreted as the product of $w + x$ and $y + z$. As we shall see, concatenation and union have precedences that make them similar to multiplication and addition, respectively.

Some Tricky Distinctions Among Types

The reader should not be confused by the multiplicity of objects that seem similar, but are actually quite different. For instance, the empty string ϵ is not the same as the empty set $\emptyset$, nor is either the same as the set containing the empty string $\{\epsilon\}$. The first is of type "string" or "list of characters," while the second and third are of type "set of strings."

We should also remember to distinguish between the character `a`, which is of type "character," the string `a` of length 1, which is of type "string," and {`a`}, which as the value of the regular expression **a** is of type "set of strings." Also note that in another context, {`a`} could be the set containing the character `a`, and we have no notational convention to distinguish the two meanings of {`a`}. However, in the balance of this chapter, {`a`} will normally have the former interpretation, that is, "set of strings" rather than "set of characters."

✥ **Example 10.12.** Let R be the regular expression **a**, and so $L(R)$ is the set {`a`}. Also let S be the regular expression **b**, so $L(S) = \{\texttt{b}\}$. Then RS is the expression **ab**. To form $L(RS)$, we need to take every string in $L(R)$ and concatenate it with every string in $L(S)$. In this simple case, both languages $L(R)$ and $L(S)$ are singletons, so we have only one choice from each. We pick `a` from $L(R)$ and `b` from $L(S)$, and concatenate these lists of length 1 to get the string `ab`. Thus $L(RS)$ is {`ab`}. ✥

Example 10.12 can be generalized, in the sense that any string, written in boldface, is a regular expression that denotes the language consisting of one string, the corresponding list of characters. For instance, **then** is a regular expression whose language is {`then`}. We shall see that concatenation is an associative operator, so it doesn't matter how the characters in the regular expression are grouped, and we do not need to use any parentheses.

✥ **Example 10.13.** Now let us look at the concatenation of two regular expressions whose languages are not singleton sets. Let R be the regular expression **a** | (**ab**).[5] The language $L(R)$ is the union of $L(\mathbf{a})$ and $L(\mathbf{ab})$, that is $\{\texttt{a}, \texttt{ab}\}$. Let S be the regular expression **c** | (**bc**). Similarly, $L(S) = \{\texttt{c}, \texttt{bc}\}$. The regular expression RS is $\big(\mathbf{a} \mid (\mathbf{ab})\big)\big(\mathbf{c} \mid (\mathbf{bc})\big)$. Note that the parentheses around R and S are required, because of precedence.

	c	bc
a	ac	abc
ab	abc	abbc

Fig. 10.25. Forming the concatenation of $\{a, ab\}$ with $\{c, bc\}$.

[5] As we shall see, concatenation takes precedence over union, so the parentheses are redundant.

To discover the strings in $L(RS)$, we pair each of the two strings from $L(R)$ with each of the two strings from $L(S)$. This pairing is suggested in Fig. 10.25. From `a` in $L(R)$ and `c` in $L(S)$, we get the string `ac`. The string `abc` is obtained in two different ways, either as (`a`)(`bc`) or as (`ab`)(`c`). Finally, the string `abbc` is obtained as the concatenation of `ab` from $L(R)$ and `bc` from $L(S)$. Thus $L(RS)$ is $\{$`ac`, `abc`, `abbc`$\}$. ❖

Note that the number of strings in the language $L(RS)$ cannot be greater than the product of the number of strings in $L(R)$ times the number of strings in $L(S)$. In fact, the number of strings in $L(RS)$ is exactly this product, unless there are "coincidences," in which the same string is formed in two or more different ways. Example 10.13 was an instance where the string `abc` was produced in two ways, and therefore the number of strings in $L(RS)$, which was 3, was one less than the product of the number of strings in the languages of R and S. Similarly, the number of strings in the language $L(R \mid S)$ is no greater than the sum of the number of strings in the languages $L(R)$ and $L(S)$, and can be less only when there are strings in common to $L(R)$ and $L(S)$. As we shall see when we discuss algebraic laws for these operators, there is a close, although not exact, analogy between union and concatenation on one hand, and the arithmetic operators + and × on the other.

Closure

The third operator is called *Kleene closure* or just *closure*.[6] It is a unary, postfix operator; that is, it takes one operand and it appears after that operand. Closure is denoted by a star, so R^* is the closure of regular expression R. Because the closure operator is of highest precedence, it is often necessary to put parentheses around the R, and write $(R)^*$.

The effect of the closure operator is to say "zero or more occurrences of strings in R." That is, $L(R^*)$ consists of

1. The empty string ϵ, which we can think of as zero occurrences of strings in R.
2. All the strings in $L(R)$; these represent one occurrence of a string in $L(R)$.
3. All the strings in $L(RR)$, the concatenation of $L(R)$ with itself; these represent two occurrences of strings in $L(R)$.
4. All the strings in $L(RRR)$, $L(RRRR)$, and so on, representing three, four, and more occurrences of strings in $L(R)$.

We can informally write

$$R^* = \epsilon \mid R \mid RR \mid RRR \mid \cdots$$

However, we must understand that the expression on the right side of the equals sign is not a regular expression, because it contains an infinite number of occurrences of the union operator. All regular expressions are built from a finite number of occurrences of the three operators.

---

[6] Steven C. Kleene wrote the original paper describing the algebra of regular expressions.

✦ **Example 10.14.** Let $R = \mathbf{a}$. What is $L(R^*)$? Surely ϵ is in this language, as it must be in any closure. The string a, which is the only string in $L(R)$, is in the language, as is aa from $L(RR)$, aaa from $L(RRR)$, and so on. That is, $L(\mathbf{a}^*)$ is the set of strings of zero or more a's, or $\{\epsilon, \mathtt{a}, \mathtt{aa}, \mathtt{aaa}, \ldots\}$. ✦

✦ **Example 10.15.** Now let R be the regular expression $\mathbf{a} \mid \mathbf{b}$, so $L(R) = \{\mathtt{a}, \mathtt{b}\}$, and consider what $L(R^*)$ is. Again, this language contains ϵ, representing zero occurrences of strings from $L(R)$. One occurrence of a string from R gives us $\{\mathtt{a}, \mathtt{b}\}$ for $L(R^*)$. Two occurrences give us the four strings $\{\mathtt{aa}, \mathtt{ab}, \mathtt{ba}, \mathtt{bb}\}$, three occurrences give us the eight strings of length three that consist of a's and/or b's, and so on. Thus $L(R^*)$ is all strings of a's and b's of any finite length whatsoever. ✦

Precedence of Regular Expression Operators

As we have mentioned informally above, there is a conventional order of precedence for the three operators union, concatenation, and closure. This order is

1. Closure (highest), then
2. Concatenation, then
3. Union (lowest).

Thus when interpreting any expression, we first group the closure operators by finding the shortest expression immediately to the left of a given * that has the form of an expression (i.e., parentheses, if any, are balanced). We may place parentheses around this expression and the *.

Next, we may consider the concatenation operators, from the left. For each, we find the smallest expression immediately to the left and the smallest expression immediately to the right, and we put parentheses around this pair of expressions. Finally, we consider union operators from the left. We find the smallest expression immediately to the left and right of each union, and we place parentheses around this pair of expressions with the union symbol in the middle.

✦ **Example 10.16.** Consider the expression $\mathbf{a} \mid \mathbf{bc}^*\mathbf{d}$. We first consider the *'s. There is only one, and to its left, the smallest expression is **c**. We may thus group this * with its operand, as $\mathbf{a} \mid \mathbf{b}(\mathbf{c}^*)\mathbf{d}$.

Next, we consider the concatenations in the above expression. There are two, one between the **b** and the left parenthesis, and the second between the right parenthesis and the **d**. Considering the first, we find the expression **b** immediately to the left, but to the right we must go until we include the right parenthesis, since expressions must have balanced parentheses. Thus the operands of the first concatenation are **b** and $(\mathbf{c}^*)$. We place parentheses around these to get the expression

$$\mathbf{a} \mid \big(\mathbf{b}(\mathbf{c}^*)\big)\mathbf{d}$$

For the second concatenation, the shortest expression immediately to the left is now $\big(\mathbf{b}(\mathbf{c}^*)\big)$, and the shortest expression immediately to the right is **d**. With parentheses added to group the operands of this concatenation, the expression becomes

$$\mathbf{a} \mid \Big(\big(\mathbf{b}(\mathbf{c}^*)\big)\mathbf{d}\Big)$$

Finally, we must consider the unions. There is only one; its left operand is **a**, and its right operand is the rest of the expression above. Technically, we must place parentheses around the entire expression, yielding

$$\Big(\mathbf{a} \mid \big((\mathbf{b}(\mathbf{c}^*))\mathbf{d}\big)\Big)$$

but the outer parentheses are redundant. ✥

Additional Examples of Regular Expressions

We close the section with some more complex examples of regular expressions.

✥ **Example 10.17.** We can extend the idea from Example 10.15 to say "strings of any length consisting of symbols $\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_n$" with the regular expression

$$(\mathbf{a}_1 \mid \mathbf{a}_2 \mid \cdots \mid \mathbf{a}_n)^*$$

For instance, we can describe C identifiers as follows. First, define the regular expression

$$\mathit{letter} = \mathbf{A} \mid \mathbf{B} \mid \cdots \mid \mathbf{Z} \mid \mathbf{a} \mid \mathbf{b} \mid \cdots \mid \mathbf{z} \mid \_$$

That is, the "letters" in C are the upper- and lowercase letters and the underscore. Similarly define

$$\mathit{digit} = \mathbf{0} \mid \mathbf{1} \mid \cdots \mid \mathbf{9}$$

Then the regular expression

$$\mathit{letter}(\mathit{letter} \mid \mathit{digit})^*$$

represents all strings of letters, digits, and underscores not beginning with a digit. ✥

✥ **Example 10.18.** Now let us consider a regular expression that is somewhat harder to write: the bounce filter problem discussed in Example 10.2. Recall that we described an automaton that made output 1 whenever the input ended in a sequence of 1's, loosely interpreted. That is, we decided we were in a sequence of 1's as soon as we saw two 1's in a row, but after deciding we were seeing 1's, a single 0 would not dissuade us from that conclusion. Thus the output of the automaton of Example 10.2 is 1 whenever the input ends in a sequence of two 1's, followed by anything, as long as each 0 is either followed immediately by a 1 or is the last input character seen so far. We express this condition by the regular expression

$$(\mathbf{0} \mid \mathbf{1})^*\mathbf{11}(\mathbf{1} \mid \mathbf{01})^*(\epsilon \mid \mathbf{0})$$

To understand this expression, first note that $(\mathbf{0} \mid \mathbf{1})^*$ represents any string of 0's and 1's. These strings must be followed by two 1's, as represented by the expression **11**. Thus the language of $(\mathbf{0} \mid \mathbf{1})^*\mathbf{11}$ is all strings of 0's and 1's that end (at least) in two 1's.

Next, the expression $(\mathbf{1} \mid \mathbf{01})^*$ represents all strings of 0's and 1's in which all 0's are followed by 1's. That is, the strings in the language for this expression are built by concatenating strings `1` and `01` in any order and in any quantity. While **1** lets us add a 1 to the string being formed at any time, **01** forces us to follow any

0 by a 1. Thus the expression (**0** | **1**)*__11__(**1** | **01**)* represents all strings of 0's and 1's that end in two 1's followed by any sequence in which 0's, if any, are followed immediately by 1's. The final factor, (ϵ | **0**), says "an optional 0," that is, the strings just described may be followed by a 0, or not, as we choose. ✦

EXERCISES

10.5.1: In Example 10.13 we considered the regular expression (**a** | **ab**)(**c** | **bc**), and saw that its language consisted of the three strings `ac`, `abc`, and `abbc`, that is, an `a` and a `c`, separated by from zero to two `b`'s. Write two other regular expressions that define the same language.

10.5.2: Write regular expressions that define the following languages.

a) The strings for the six C comparison operators, `=`, `<=`, `<`, `>=`, `>`, and `!=`.
b) All strings of 0's and 1's that end in 0.
c) All strings of 0's and 1's with at least one 1.
d) All strings of 0's and 1's with at most one 1.
e) All strings of 0's and 1's such that the third position from the right end is 1.
f) All strings of lower-case letters that are in sorted order.

10.5.3*: Write regular expressions that define the following languages.

a) All strings of `a`'s and `b`'s such that all runs of `a`'s are of even length. That is, strings such as `bbbaabaaaa`, `aaaabb`, and ϵ are in the language; `abbabaa` and `aaa` are not.

b) Strings that represent numbers of type `float` in C.

c) Strings of 0's and 1's having even parity, that is, an even number of 1's. *Hint*: Think of even-parity strings as the concatenation of elementary strings with even parity, either a single 0 or a pair of 1's separated only by 0's.

10.5.4**: Write regular expressions that define the following languages.

a) The set of all C identifiers that are not keywords. If you forget some of the keywords, it is not serious. The point of the exercise is to express strings that are *not* in some reasonably large set of strings.

b) All strings of `a`'s, `b`'s, and `c`'s such that no two consecutive positions are the same character.

c) The set of all strings of two lower-case letters that are not the same. *Hint*: You can "brute-force" this one, but there are 650 pairs of distinct letters. A better idea is to do some grouping. For example, the relatively short expression

$$(\texttt{a} \mid \texttt{b} \mid \cdots \mid \texttt{m})(\texttt{n} \mid \texttt{o} \mid \cdots \mid \texttt{z})$$

covers 169 of the 650 pairs.

d) All strings of 0's and 1's that, as a binary number, represent an integer that is a multiple of 3.

10.5.5: Put parentheses in the following regular expressions to indicate the proper grouping of operands according to the precedence of operators union, concatenation, and closure.

a) $\mathbf{a} \mid \mathbf{bc} \mid \mathbf{de}$
b) $\mathbf{a} \mid \mathbf{b}^* \mid (\mathbf{a} \mid \mathbf{b})^*\mathbf{a}$

10.5.6: Remove redundant parentheses from the following expressions, that is, remove parentheses whose grouping would be implied by the precedence of operators and the fact that union and concatenation are each associative (and therefore, the grouping of adjacent unions or adjacent concatenations is irrelevant).

a) $(\mathbf{ab})(\mathbf{cd})$
b) $\Big(\mathbf{a} \mid \big(\mathbf{b}(\mathbf{c})^*\big)\Big)$
c) $\Big(\big((\mathbf{a}) \mid \mathbf{b}\big)(\mathbf{c} \mid \mathbf{d})\Big)$

10.5.7*: Describe the languages defined by the following regular expressions.
a) $\emptyset \mid \epsilon$
b) $\epsilon\mathbf{a}$
c) $(\mathbf{a} \mid \mathbf{b})^*$
d) $(\mathbf{a}^*\mathbf{b}^*)^*$
e) $(\mathbf{a}^*\mathbf{ba}^*\mathbf{b})^*\mathbf{a}^*$
f) ϵ^*
g) R^{**}, where R is any regular expression.

10.6 The UNIX Extensions to Regular Expressions

The UNIX operating system has several commands that use a regular-expression like notation to describe patterns. Even if the reader is not familiar with UNIX or with most of these commands, these notations are useful to know. We find regular expressions used in at least three kinds of commands.

1. *Editors.* The UNIX editors `ed` and `vi`, as well as most modern text editors, allow the user to scan text for a place where an instance of a given pattern is found. The pattern is specified by a regular expression, although there is no general union operator, just "character classes," which we shall discuss below.

2. *The pattern-matching program* `grep` *and its cousins.* The UNIX command `grep` scans a file and examines each line. If the line contains a substring that matches the pattern specified by a regular expression, then the line is printed (`grep` stands for "*g*lobally search for *r*egular *e*xpression and *p*rint"). The command `grep` itself allows only a subset of the regular expressions, but the extended command `egrep` allows the full regular expression notation, including some other extensions. The command `awk` allows full regular expression searching, and also treats lines of text as if they were tuples of a relation, thus allowing operations of relational algebra like selection and projection to be performed on files.

3. *Lexical analysis.* The UNIX command `lex` is useful for writing a piece of a compiler and for many similar tasks. The first thing a compiler must do is partition a program into *tokens,* which are substrings that fit together logically. Examples are identifiers, constants, keywords such as `then`, and operators such as `+` or `<=`. Each token type can be specified as a regular expression; for instance, Example 10.17 showed us how to specify the token class "identifier."

Token

The `lex` command allows the user to specify the token classes by regular expressions. It then produces a program that serves as a *lexical analyzer,* that is, a program that partitions its input into tokens.

Character Classes

Often, we need to write a regular expression that denotes a set of characters, or strictly speaking, a set of character strings of length one, each string consisting of a different character in the set. Thus in Example 10.17 we defined the expression *letter* to denote any of the strings consisting of one upper- or lower-case letter, and we defined the expression *digit* to denote any of the strings consisting of a single digit. These expressions tend to be rather long, and UNIX provides some important shorthands.

First, we can enclose any list of characters in square brackets, to stand for the regular expression that is the union of these letters. Such an expression is called a *character class.* For example, the expression [**aghinostw**] denotes the set of letters appearing in the word `washington`, and [**aghinostw**]* denotes the set of strings composed of those letters only.

Second, we do not always have to list all the characters explicitly. Recall that characters are almost invariably coded in ASCII. This code assigns bit strings, which are naturally interpreted as integers, to the various characters, and it does so in a rational way. For instance, the capital letters are assigned consecutive integers. Likewise, the lower-case letters are assigned consecutive integers, and the digits are assigned consecutive integers.

If we put a dash between two characters, we denote not only those characters, but also all the characters whose codes lie between their codes.

✦ **Example 10.19.** We can define the upper- and lower-case letters by [**A-Za-z**]. The first three characters, **A-Z**, represent all the characters whose codes lie between those for `A` and `Z`, that is, all the upper-case letters. The next three characters, **a-z**, similarly denote all the lower-case letters.

Incidentally, because the dash has this special meaning, we must be careful if we want to define a character class including `-`. We must place the dash either first or last in the list. For instance, we could specify the set of four arithmetic operators by `[-+*/]`, but it would be an error to write `[+-*/]`, because then the range `+-*` would denote all the characters whose codes are between the codes for `+` and `*`. ✦

Line Beginning and Ending

Because UNIX commands so frequently deal with single lines of text, the UNIX regular expression notation includes special symbols that denote the beginning and end of a line. The symbol `^` denotes the beginning of a line, and `$` denotes the end of a line.

✦ **Example 10.20.** The automaton of Fig. 10.12 in Section 10.3, started at the beginning of a line, will accept that line exactly when the line consists only of letters in the word `washington`. We can express this pattern as a UNIX regular expression: `^`[**aghinostw**]*`$`. In words, the pattern is "the beginning of the line, followed by any sequence of letters from the word `washington`, followed by the end of the line."

Giving Characters Their Literal Meaning

Escape character

Incidentally, since the characters `^` and `$` are given a special meaning in regular expressions, it would seem that we do not have any way to specify these characters themselves in UNIX regular expressions. However, UNIX uses the backslash, \, as an *escape character.* If we precede `^` or `$` by a backslash, then the combination of two characters is interpreted as the second character's literal meaning, rather than its special meaning. For instance, `\$` represents the character `$` within a UNIX regular expression. Likewise, two backslashes are interpreted as a single backslash, without the special meaning of an escape character. The string `\\$` in a UNIX regular expression denotes the character backslash followed by the end of the line.

There are a number of other characters that are given special meanings by UNIX in certain situations, and these characters can always be referred to literally, that is, without their special meaning, by preceding them by a backslash. For example, square brackets must be treated this way in regular expressions to avoid interpreting them as character-class delimiters.

As an example of how this regular expression is used, the UNIX command line

```
grep '^[aghinostw]*$' /usr/dict/words
```

will print all the words in the dictionary that consist only of letters from `washington`. UNIX requires, in this case, that the regular expression be written as a quoted string. The effect of the command is that each line of the specified file `/usr/dict/words` is examined. If it has any substring that is in the set of strings denoted by the regular expression, then the line is printed; otherwise, the line is not printed. Note that the line beginning and ending symbols are essential here. Suppose they were missing. Since the empty string is in the language denoted by the regular expression [**aghinostw**]*, we would find that every line has a substring (namely ϵ) that is in the language of the regular expression, and thus every line would be printed. ✥

The Wild Card Symbol

The character `.` in UNIX regular expressions stands for "any character but the newline character."

✥ **Example 10.21.** The regular expression

.***a**.***e**.***i**.***o**.***u**.*

denotes all strings that contain the vowels, in order. We could use `grep` with this regular expression to scan the dictionary for all words with the vowels in increasing order. However, it is more efficient to omit the .* from the beginning and end, because `grep` searches for the specified pattern as a substring, rather than as the whole line, unless we include the line beginning and ending symbols explicitly. Thus the command

```
grep 'a.*e.*i.*o.*u' /usr/dict/words
```

will find and print all the words that have `aeiou` as a subsequence.

The fact that the dots will match characters other than letters is unimportant, since there are no other characters besides letters and the newline character in the file `/usr/dict/words`. However, if the dot could match the newline character, then this regular expression could allow `grep` to use several lines together to find one occurrence of the vowels in order. It is for examples like this one that the dot is defined not to match the newline character. ✦

Additional Operators

The regular expressions in the UNIX commands `awk` and `egrep` also include some additional operators.

1. Unlike `grep`, the commands `awk` and `egrep` also permit the union operator | in their regular expressions.

2. The unary postfix operators ? and $^+$ do not allow us to define additional languages, but they often make it easier to express languages. If R is a regular expression, then $R?$ stands for $\epsilon \mid R$, that is, an optional R. Thus $L(R?)$ is $L(R) \cup \{\epsilon\}$. R^+ stands for RR^*, or equivalently, "one or more occurrences of words from R." Thus,

$$L(R^+) = L(R) \cup L(RR) \cup L(RRR) \cdots$$

In particular, if ϵ is in $L(R)$, then $L(R^+)$ and $L(R^*)$ denote the same language. If ϵ is not in $L(R)$, then $L(R^+)$ denotes $L(R^*) - \{\epsilon\}$. The operators $^+$ and ? have the same associativity and precedence as *.

✦ **Example 10.22.** Suppose we want to specify by a regular expression real numbers consisting of a nonempty string of digits with one decimal point. It would not be correct to write this expression as [**0-9**]*\\.[**0-9**]*, because then the string consisting of a dot alone would be considered a real number. One way to write the expression using `egrep` is

$$[\mathbf{0\text{-}9}]^+\backslash.[\mathbf{0\text{-}9}]^* \mid \backslash.[\mathbf{0\text{-}9}]^+$$

Here, the first term of the union covers those numbers that have at least one digit to the left of the decimal point, and the second term covers those numbers that begin with the decimal point, and that therefore must have at least one digit following the decimal point. Note that a backslash is put before the dot so that the dot does not acquire the conventional "wild card" meaning. ✦

✦ **Example 10.23.** We can scan input for all lines whose letters are in strictly increasing alphabetical order with the `egrep` command

```
egrep '^a?b?c?d?e?f?g?h?i?j?k?l?m?n?o?p?q?r?s?t?u?v?w?x?y?z?$'
```

That is, we scan each line to see if between the beginning and end of the line there is an optional `a`, and optional `b`, and so on. A line containing the word `adept`, for instance, matches this expression, because the ?'s after `a`, `d`, `e`, `p`, and `t` can be interpreted as "one occurrence," while the other ?'s can be interpreted as "zero occurrences," that is, ϵ. ✦

EXERCISES

10.6.1: Write expressions for the following character classes.

a) All characters that are operators or punctuation symbols in C. Examples are `+` and parentheses.

b) All the lower-case vowels.

c) All the lower-case consonants.

10.6.2*: If you have UNIX available, write `egrep` programs to examine the file

```
/usr/dict/words
```

and find

a) All words that end in `dous`.

b) All words that have only one vowel.

c) All words that have alternating consonants and vowels.

d) All words that have four or more consecutive consonants.

10.7 Algebraic Laws for Regular Expressions

It is possible for two regular expressions to denote the same language, just as two arithmetic expressions can denote the same function of their operands. As an example, the arithmetic expressions $x + y$ and $y + x$ each denote the same function of x and y, because addition is commutative. Similarly, the regular expressions $R \mid S$ and $S \mid R$ denote the same languages, no matter what regular expressions we substitute for R and S; the justification is that union is also a commutative operation.

Often, it is useful to simplify regular expressions. We shall see shortly that, when we construct regular expressions from automata, we often construct a regular expression that is unnecessarily complex. A repertoire of algebraic equivalences may allow us to "simplify" expressions, that is, replace one regular expression by another that involves fewer operands and/or operators, yet that denotes the same language. The process is analogous to what we go through when we manipulate arithmetic expressions to simplify an unwieldy expression. For example, we might multiply two large polynomials and then simplify the result by grouping similar terms. As another example, we simplified expressions of relational algebra in Section 8.9 to allow faster evaluation.

Equivalent regular expressions

Two regular expressions R and S are *equivalent,* written $R \equiv S$, if $L(R) = L(S)$. If so, we say that $R \equiv S$ is an *equivalence.* In what follows, we shall assume that R, S, and T are arbitrary regular expressions, and state our equivalences with these operands.

Proving Equivalences

In this section we prove a number of equivalences involving regular expressions. Recall that an equivalence between two regular expressions is a claim that the languages of these two expressions are equal, no matter what languages we substitute for their variables. We thus prove an equivalence by showing the equality of two languages, that is, two sets of strings. In general, we prove that set S_1 equals set S_2 by proving containment in both directions. That is, we prove $S_1 \subseteq S_2$, and we also prove $S_2 \subseteq S_1$. Both directions are necessary to prove equality of sets.

Ways Union and Concatenation Resemble Plus and Times

In this section, we shall enumerate the most important equivalences involving the regular expression operators union, concatenation, and closure. We begin with the analogy between union and concatenation, on one hand, and addition and multiplication on the other. This analogy is not exact, as we shall see, primarily because concatenation is not commutative, while multiplication is, of course. However, there are many similarities between the two pairs of operations.

To begin, both union and concatenation have identities. The identity for union is $\emptyset$, and for concatenation the identity is ϵ.

1. *Identity for union.* $(\emptyset \mid R) \equiv (R \mid \emptyset) \equiv R$.

2. *Identity for concatenation.* $\epsilon R \equiv R\epsilon \equiv R$.

It should be obvious from the definition of the empty set and union why (1) is true. To see why (2) holds true, if string x is in $L(\epsilon R)$, then x is the concatenation of a string in $L(\epsilon)$ and a string r that is in $L(R)$. But the only string in $L(\epsilon)$ is ϵ itself, and so we know that $x = \epsilon r$. However, the empty string concatenated with any string r is just r itself, and so $x = r$. That is, x is in $L(R)$. Similarly, we can see that if x is in $L(R\epsilon)$, then x is in $L(R)$.

To prove the pair of equivalences (2) we not only have to show that everything in $L(\epsilon R)$ or $L(R\epsilon)$ is in $L(R)$, but we must also show the converse, that everything in $L(R)$ is in $L(\epsilon R)$ and in $L(R\epsilon)$. If r is in $L(R)$, then ϵr is in $L(\epsilon R)$. But $\epsilon r = r$, and so r is in $L(\epsilon R)$. The same reasoning tells us that r is also in $L(R\epsilon)$. We have thus shown that $L(R)$ and $L(\epsilon R)$ are the same language and that $L(R)$ and $L(R\epsilon)$ are the same language, which are the equivalences of (2).

Thus $\emptyset$ is analogous to 0 in arithmetic, and ϵ is analogous to 1. There is another way the analogy holds. $\emptyset$ is an *annihilator* for concatenation; that is,

3. *Annihilator for concatenation.* $\emptyset R \equiv R\emptyset \equiv \emptyset$. In other words, when we concatenate the empty set with anything, we get the empty set. Analogously, 0 is an annihilator for multiplication, since $0 \times x = x \times 0 = 0$.

We can see why (3) is true as follows. In order for there to be some string x in $L(\emptyset R)$, we would have to form x by concatenating a string from $L(\emptyset)$ with a string from $L(R)$. Since there is no string in $L(\emptyset)$, there is no way we can form x. A similar argument shows that $L(R\emptyset)$ must be empty.

The next equivalences are the commutative and associative laws for union that we discussed in Chapter 7.

4. *Commutativity of union.* $(R \mid S) \equiv (S \mid R)$.

5. *Associativity of union.* $\big((R \mid S) \mid T\big) \equiv \big(R \mid (S \mid T)\big)$.

As we mentioned, concatenation is also associative. That is,

6. *Associativity of concatenation.* $\big((RS)T\big) \equiv \big(R(ST)\big)$.

To see why (6) holds true, suppose string x is in $L\big((RS)T\big)$. Then x is the concatenation of some string y in $L(RS)$ and some string t in $L(T)$. Also, y must be the concatenation of some r in $L(R)$ and some s in $L(S)$. Thus $x = yt = rst$. Now consider $L\big(R(ST)\big)$. The string st must be in $L(ST)$, and so rst, which is x, is in $L\big(R(ST)\big)$. Thus every string x in $L\big((RS)T\big)$ is also in $L\big(R(ST)\big)$. A similar argument tells us that every string in $L\big(R(ST)\big)$ must also be in $L\big((RS)T\big)$. Thus these two languages are the same, and the equivalence (6) holds.

Next, we have the distributive laws of concatenation over union, that is,

7. *Left distributivity of concatenation over union.* $\big(R(S \mid T)\big) \equiv (RS \mid RT)$.

8. *Right distributivity of concatenation over union.* $\big((S \mid T)R\big) \equiv (SR \mid TR)$.

Let us see why (7) holds; (8) is similar and is left for an exercise. If x is in

$$L\big(R(S \mid T)\big)$$

then $x = ry$, where r is in $L(R)$ and y is either in $L(S)$ or in $L(T)$, or both. If y is in $L(S)$, then x is in $L(RS)$, and if y is in $L(T)$, then x is in $L(RT)$. In either case, x is in $L(RS \mid RT)$. Thus everything in $L\big(R(S \mid T)\big)$ is in $L(RS \mid RT)$.

We must also prove the converse, that everything in $L(RS \mid RT)$ is in

$$L\big(R(S \mid T)\big)$$

If x is in the former language, then x is either in $L(RS)$ or in $L(RT)$. Suppose x is in $L(RS)$. Then $x = rs$, where r is in $L(R)$ and s is in $L(S)$. Thus s is in $L(S \mid T)$, and therefore x is in $L\big(R(S \mid T)\big)$. Similarly, if x is in $L(RT)$, we can show that x must be in $L\big(R(S \mid T)\big)$. We have now shown containment in both directions, which proves the equivalence (7).

Ways Union and Concatenation Differ from Plus and Times

One reason that union fails to be analogous to addition is the idempotent law. That is, union is idempotent, but addition is not.

9. *Idempotence of union.* $(R \mid R) \equiv R$.

Concatenation also deviates from multiplication in an important way, since concatenation is not commutative, while multiplication of reals or integers is commutative. To see why RS is not in general equivalent to SR, take a simple example such as $R = \mathbf{a}$ and $S = \mathbf{b}$. Then $L(RS) = \{\texttt{ab}\}$, while $L(SR) = \{\texttt{ba}\}$, a different set.

Equivalences Involving Closure

There are a number of useful equivalences involving the closure operator.

10. $\emptyset^* \equiv \epsilon$. You may check that both sides denote the language $\{\epsilon\}$.

11. $RR^* \equiv R^*R$. Note that both sides are equivalent to R^+ in the extended notation of Section 10.6.

12. $(RR^* \mid \epsilon) \equiv R^*$. That is, the union of R^+ and the empty string is equivalent to R^*.

EXERCISES

10.7.1: Prove that the right distributive law of concatenation over union, equivalence (8), holds.

10.7.2: The equivalences $\emptyset\emptyset \equiv \emptyset$ and $\epsilon\epsilon \equiv \epsilon$ follow from equivalences already stated, by substitution for variables. Which equivalences do we use?

10.7.3: Prove equivalences (10) through (12).

10.7.4: Prove that

a) $(R \mid R^*) \equiv R^*$
b) $(\epsilon \mid R^*) \equiv R^*$

10.7.5*: Are there examples of particular regular expressions R and S that are "commutative," in the sense that $RS = SR$ for these particular expressions? Give a proof if not, or some examples if so.

$\emptyset$-free regular expression

10.7.6*: The operand $\emptyset$ is not needed in regular expressions, except that without it, we could not find a regular expression whose language is the empty set. Call a regular expression *$\emptyset$-free* if it has no occurrences of $\emptyset$. Prove by induction on the number of operator occurrences in a $\emptyset$-free regular expression R, that $L(R)$ is not the empty set. *Hint*: The next section gives an example of an induction on the number of operator occurrences of a regular expression.

10.7.7**: Show by induction on the number of operator occurrences in a regular expression R, that R is equivalent to either the regular expression $\emptyset$, or some $\emptyset$-free regular expression.

10.8 From Regular Expressions to Automata

Remember our initial discussion of automata in Section 10.2, where we observed a close relationship between deterministic automata and programs that used the concept of "state" to distinguish the roles played by different parts of the program. We said then that designing deterministic automata is often a good way to design such programs. However, we also saw that deterministic automata could be hard to design. We saw in Section 10.3 that sometimes nondeterministic automata were easier to design, and that the subset construction allows us to turn any nondeterministic automaton into a deterministic one. Now that we have met regular expressions, we see that often it is even easier to write a regular expression than it is to design a nondeterministic automaton.

Thus, it is good news that there is a way to convert any regular expression into a nondeterministic automaton, and from there we can use the subset construction to convert to a deterministic automaton. In fact, we shall see in the next section that it is also possible to convert any automaton into a regular expression whose

Not All Languages Are Described by Automata

While we have seen many languages that can be described by automata or regular expressions, there are languages that cannot be so described. The intuition is that "automata cannot count." That is, if we feed an automaton with n states a sequence of n of the same symbol, it must twice enter the same state. It then cannot remember exactly how many symbols it has seen. Thus it is not possible, for example, for an automaton to recognize all and only the strings of balanced parentheses. Since regular expressions and automata define the same languages, there is likewise no regular expression whose language is exactly the strings of balanced parentheses. We shall consider the matter of what languages are not definable by automata in the next chapter.

language is exactly the set of strings that the automaton accepts. Thus automata and regular expressions have exactly the same capability to describe languages.

In this section, we need to do a number of things to show how regular expressions are converted to automata.

1. We introduce automata with ϵ-transitions, that is, with arcs labeled ϵ. These arcs are used in paths but do not contribute to the labels of paths. This form of automaton is an intermediate between regular expressions and the automata discussed earlier in this chapter.

2. We show how to convert any regular expression to an automaton with ϵ-transitions that defines the same language.

3. We show how to convert any automaton with ϵ-transitions to an automaton without ϵ-transitions that accepts the same language.

Automata with Epsilon-Transitions

We first extend our notion of automata to allow arcs labeled ϵ. Such automata still accept a string s if and only if there is a path labeled s from the start state to an accepting state. However, note that ϵ, the empty string, is "invisible" in strings, and so when constructing the label for a path we in effect delete all the ϵ's and use only the "real" characters.

✦ **Example 10.24.** Consider the automaton with ϵ-transitions shown in Fig. 10.26. Here, state 0 is the start state, and state 3 is the lone accepting state. One path from state 0 to state 3 is

0, 4, 5, 6, 7, 8, 7, 8, 9, 3

The labels of the arcs form the sequence

ϵ b ϵ ϵ c ϵ c ϵ ϵ

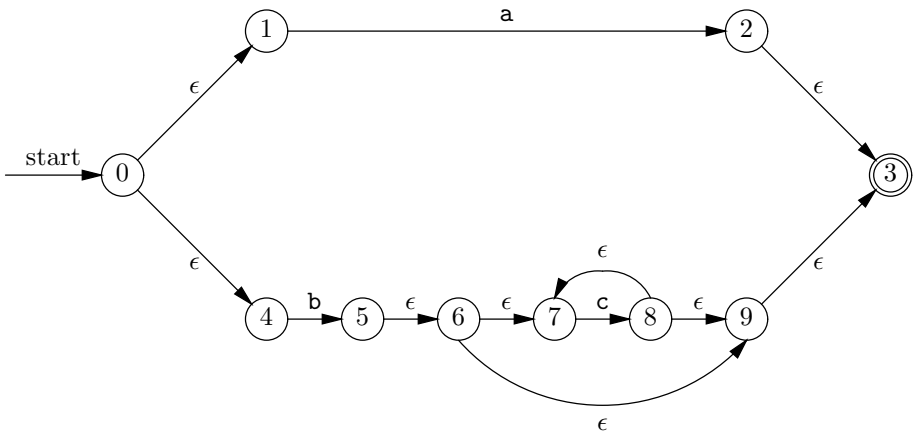


Fig. 10.26. An automaton with ϵ-transitions for **a** | **bc***.

When we remember that ϵ concatenated with any other string is that other string, we see we can "drop out" the ϵ's to get the string `bcc`, which is the label of the path in question.

You can probably discover that the paths from state 0 to state 3 are labeled by all and only the strings `a`, `b`, `bc`, `bcc`, `bccc`, and so on. A regular expression for this set is **a** | **bc***, and we shall see that the automaton of Fig. 10.26 is constructed naturally from that expression. ✥

From Expressions to Automata with Epsilon-Transitions

We convert a regular expression to an automaton using an algorithm derived from a complete induction on the number of operator occurrences in the regular expression. The idea is similar to a structural induction on trees that we introduced in Section 5.5, and the correspondence becomes clearer if we represent regular expressions by their expression trees, with atomic operands at the leaves and operators at interior nodes. The statement that we shall prove is:

STATEMENT $S(n)$: If R is a regular expression with n occurrences of operators and no variables as atomic operands, then there is an automaton A with ϵ-transitions that accepts those strings in $L(R)$ and no others. Moreover, A has

1. Only one accepting state,
2. No arcs into its start state, and
3. No arcs out of its accepting state.

BASIS. If $n = 0$, then R must be an atomic operand, which is either $\emptyset$, ϵ, or **x** for some symbol `x`. In these three cases, we can design a 2-state automaton that meets the requirements of the statement $S(0)$. These automata are shown in Fig. 10.27.

It is important to understand that we create a new automaton, with states distinct from those of any other automaton, for each occurrence of an operand in the regular expression. For instance, if there were three occurrences of **a** in the

expression, we would create three different automata, with six states in all, each similar to Fig. 10.27(c), but with `a` in place of `x`.

The automaton of Fig. 10.27(a) evidently accepts no strings, since you cannot get from the start state to the accepting state; thus its language is $\emptyset$. Figure 10.27(b) is suitable for ϵ, since it accepts the empty string but no other. Figure 10.27(c) is an automaton for accepting only the string `x`. We can create new automata with different values of the symbol `x` as we choose. Note that each of these automata satisfies the three requirements stated above; there is one accepting state, no arcs into the start state and no arcs out of the accepting state.

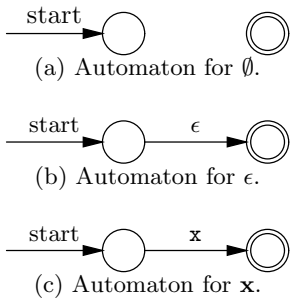


Fig. 10.27. Automata for basis cases.

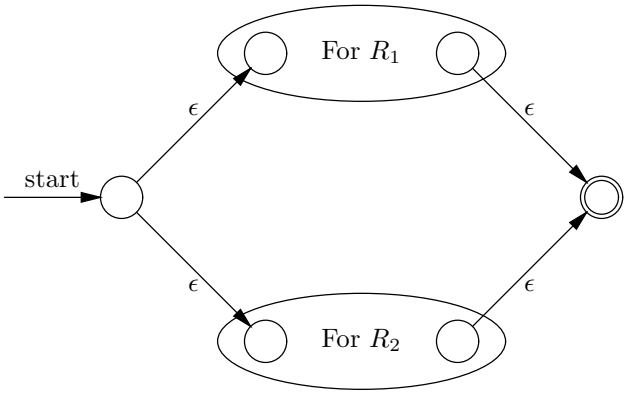
INDUCTION. Now suppose that $S(i)$ is true for all $i \leq n$; that is, for any regular expression R with up to n operator occurrences, there is an automaton satisfying the conditions of the inductive hypothesis and accepting all and only the strings in $L(R)$. Now, let R be a regular expression with $n+1$ operator occurrences. We focus on the "outermost" operator in R; that is, R can only be of the form $R_1 \mid R_2$, R_1R_2, or $R_1{*}$, depending on whether union, concatenation, or closure was the last operator used when R was formed.

In any of these three cases, R_1 and R_2 cannot have more than n operators, because there is one operator of R that is not part of either.[7] Thus the inductive hypothesis applies to R_1 and R_2 in all three cases. We can prove $S(n+1)$ by consideration of these cases in turn.

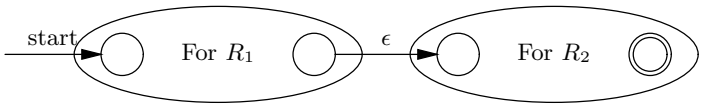
Case 1. If $R = R_1 \mid R_2$, then we construct the automaton of Fig. 10.28(a). We do so by taking the automata for R_1 and R_2 and adding two new states, one the start state and the other the accepting state. The start state of the automaton for R has ϵ-transitions to the start states of the automata for R_1 and R_2. The accepting states of those two automata have ϵ-transitions to the accepting state of the automaton for R. However, the start and accepting states of the automata for R_1 and R_2 are not start or accepting states in the constructed automaton.

The reason that this construction works is that the only ways to get from the start state to the accepting state in the automaton for R is to follow an ϵ-labeled arc to the start state of the automaton for R_1 or that for R_2. Then, we must follow a

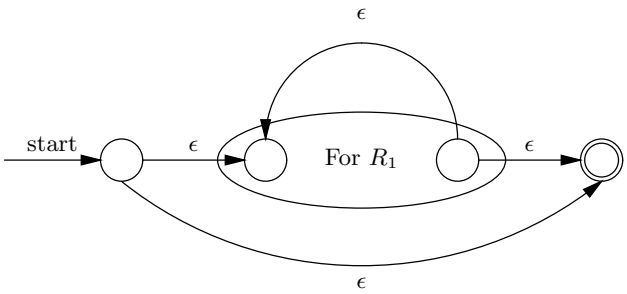
[7] Let us not forget that even though concatenation is represented by juxtaposition, rather than a visible operator symbol, uses of concatenation still count as operator occurrences when deciding how many operator occurrences R has.



(a) Constructing the automaton for the union of two regular expressions.



(b) Constructing the automaton for the concatenation of two regular expressions.



(c) Constructing the automaton for the closure of a regular expression.

Fig. 10.28. Inductive part of the construction of an automaton from a regular expression.

path in the chosen automaton to get to its accepting state, and then an ϵ-transition to the accepting state of the automaton for R. This path is labeled by some string s that the automaton we traveled through accepts, because we go from the start state to the accepting state of that automaton. Therefore, s is either in $L(R_1)$ or $L(R_2)$, depending on which automaton we traveled through. Since we only add ϵ's to that path's labels, the automaton of Fig. 10.28(a) also accepts s. Thus the strings accepted are those in $L(R_1) \cup L(R_2)$, which is $L(R_1 \mid R_2)$, or $L(R)$.

Case 2. If $R = R_1R_2$, then we construct the automaton of Fig. 10.28(b). This automaton has as its start state the start state of the automaton for R_1, and as its accepting state the accepting state of the automaton for R_2. We add an ϵ-transition from the accepting state of the automaton for R_1 to the start state of the automaton for R_2. The accepting state of the first automaton is no longer accepting, and the start state of the second automaton is no longer the start state in the constructed

automaton.

The only way to get from the start to the accepting state of Fig. 10.28(b) is

1. Along a path labeled by a string s in $L(R_1)$, to get from the start state to the accepting state of the automaton for R_1, then
2. Along the arc labeled ϵ to the start state of the automaton for R_2, and then
3. Along a path labeled by some string t in $L(R_2)$ to get to the accepting state.

The label of this path is st. Thus the automaton of Fig. 10.28(b) accepts exactly the strings in $L(R_1R_2)$, or $L(R)$.

Case 3. If $R = R_1{}^*$, we construct the automaton of Fig. 10.28(c). We add to the automaton for R_1 a new start and accepting state. The start state has an ϵ-transition to the accepting state (so string ϵ is accepted), and to the start state of the automaton for R_1. The accepting state of the automaton for R_1 is given an ϵ-transition back to its start state, and one to the accepting state of the automaton for R. The start and accepting states of the automaton for R_1 are not start or accepting in the constructed automaton.

The paths from start to accepting state in Fig. 10.28(c) are either labeled ϵ (if we go directly) or labeled by the concatenation of one or more strings from $L(R_1)$, as we go through the automaton for R_1 and, optionally, around to the start as many times as we like. Note that we do not have to follow the same path through the automaton for R_1 each time around. Thus the labels of the paths through Fig. 10.28(c) are exactly the strings in $L(R_1{}^*)$, which is $L(R)$.

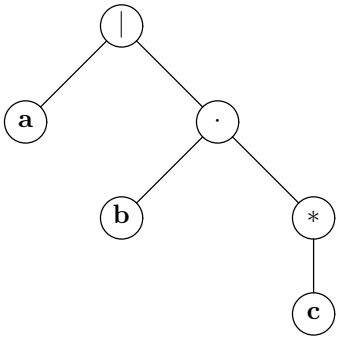
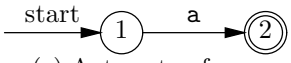


Fig. 10.29. Expression tree for the regular expression **a** | **bc***.

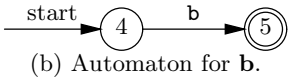
◆ **Example 10.25.** Let us construct the automaton for the regular expression **a** | **bc***. An expression tree for this regular expression is shown in Fig. 10.29; it is analogous to the expression trees we discussed in Section 5.2, and it helps us see the order in which the operators are applied to the operands.

There are three leaves, and for each, we construct an instance of the automaton of Fig. 10.27(c). These automata are shown in Fig. 10.30, and we have used the states that are consistent with the automaton of Fig. 10.26, which as we mentioned, is the automaton we shall eventually construct for our regular expression. It should be understood, however, that it is essential for the automata corresponding to the

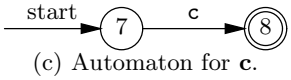
various occurrences of operands to have distinct states. In our example, since each operand is different, we would expect to use different states for each, but even if there were several occurrences of **a**, for example, in the expression, we would create distinct automata for each occurrence.



(a) Automaton for **a**.



(b) Automaton for **b**.



(c) Automaton for **c**.

Fig. 10.30. Automata for **a**, **b**, and **c**.

Now we must work up the tree of Fig. 10.29, applying operators and constructing larger automata as we go. The first operator applied is the closure operator, which is applied to operand **c**. We use the construction of Fig. 10.28(c) for the closure. The new states introduced are called 6 and 9, again to be consistent with Fig. 10.26. Fig. 10.31 shows the automaton for the regular expression **c***.

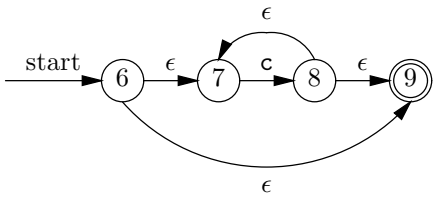


Fig. 10.31. Automaton for **c***.

Next, we apply the concatenation operator to **b** and **c***. We use the construction of Fig. 10.28(b), and the resulting automaton is shown in Fig. 10.32.

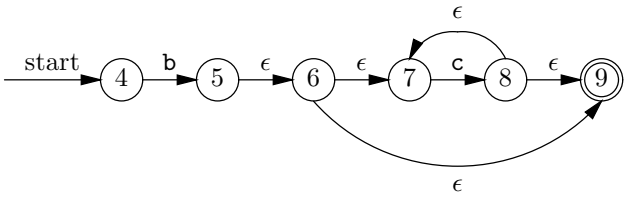


Fig. 10.32. Automaton for **bc***.

Finally, we apply the union operator to **a** and **bc***. The construction used is that of Fig. 10.28(a), and we call the new states introduced 0 and 3. The resulting automaton appeared in Fig. 10.26. ✦

Eliminating Epsilon-Transitions

If we are in any state s of an automaton with ϵ-transitions, we in effect are also in any state that we can get to from s by following a path of arcs labeled ϵ. The reason is that whatever string labels the path we've taken to get to state s, the same string will be the label of the path extended with ϵ-transitions.

✦ **Example 10.26.** In Fig. 10.26, we can get to the state 5 by following a path labeled `b`. From state 5, we can get to states 6, 7, 9, and 3 by following paths of ϵ-labeled arcs. Thus if we are in state 5, we are, in effect, also in these four other states. For instance, since 3 is an accepting state, we can think of 5 as an accepting state as well, since every input string that gets us to state 5 will also get us to state 3, and thus be accepted. ✦

Thus the first question we need to ask is, from each state, what other states can we reach following only ϵ-transitions? We gave an algorithm to answer this question in Section 9.7, when we studied reachability as an application of depth-first search. For the problem at hand, we have only to modify the graph of the finite automaton by removing transitions on anything but ϵ. That is, for each real symbol `x`, we remove all arcs labeled by `x`. Then, we perform a depth-first search of the remaining graph from each node. The nodes visited during the depth-first search from node v is exactly the set of node reachable from v using ϵ-transitions only.

Recall that one depth-first search takes $O(m)$ time, where m is the larger of the number of nodes and arcs of the graph. In this case, there are n depth-first searches to do, if the graph has n nodes, for a total of $O(mn)$ time. However, there are at most two arcs out of any one node in the automata constructed from regular expressions by the algorithm described previously in this section. Thus $m \leq 2n$, and $O(mn)$ is $O(n^2)$ time.

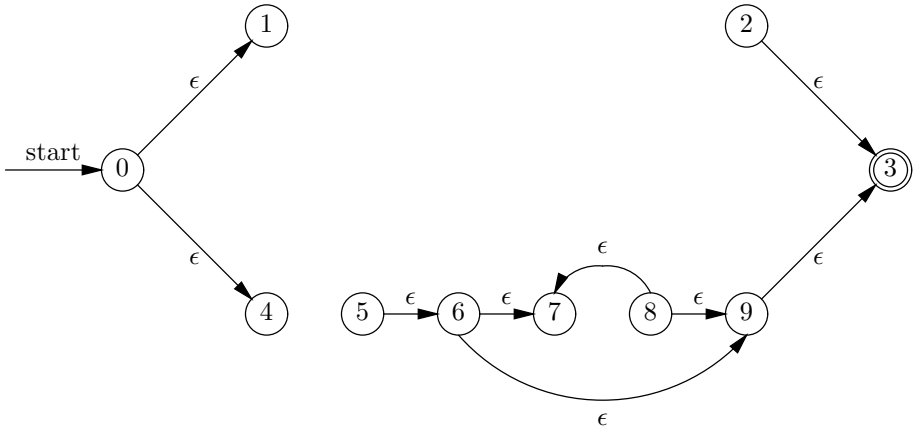


Fig. 10.33. The ϵ-transitions from Fig. 10.26.

✦ **Example 10.27.** In Fig. 10.33 we see the arcs that remain from Fig. 10.26 when the three arcs labeled by a real symbol, `a`, `b`, or `c`, are deleted. Figure 10.34 is a table giving the reachability information for Fig. 10.33; that is, a 1 in row i and column j means that there is a path of length 0 or more from node i to node j. ✦

	0	1	2	3	4	5	6	7	8	9
0	1	1			1					
1		1								
2			1	1						
3				1						
4					1					
5				1		1	1	1		1
6				1			1	1		1
7								1		
8				1				1	1	1
9				1						1

Fig. 10.34. Table of reachability for Fig. 10.33.

Armed with the reachability information, we can construct our equivalent automaton that has no ϵ-transitions. The idea is to bundle into one transition of the new automaton a path of zero or more ϵ-transitions of the old automaton followed by one transition of the old automaton on a real symbol. Every such transition takes us to the second state of one of the automata that were introduced by the basis rule of Fig. 10.27(c), the rule for operands that are real symbols. The reason is that only these states are entered by arcs with real symbols as labels. Thus our new automaton needs only these states and the start state for its own set of states. Let us call these states the *important* states.

Important state

In the new automaton, there is a transition from important state i to important state j with symbol `x` among its labels if there is some state k such that

1. State k is reachable from state i along a path of zero or more ϵ-transitions. Note that $k = i$ is always permitted.
2. In the old automaton, there is a transition from state k to state j, labeled `x`.

We also must decide which states are accepting states in the new automaton. As we mentioned, when we are in a state, we are effectively in any state it can reach along ϵ-labeled arcs, and so in the new automaton, we shall make state i accepting if there is, in the old automaton, a path of ϵ-labeled arcs from state i to the accepting state of the old automaton. Note that i may itself be the accepting state of the old automaton, which therefore remains accepting in the new automaton.

✦ **Example 10.28.** Let us convert the automaton of Fig. 10.26 to an automaton without ϵ-transitions, accepting the same language. First, the important states are

state 0, which is the initial state, and states 2, 5, and 8, because these are entered by arcs labeled by a real symbol.

We shall begin by discovering the transitions for state 0. According to Fig. 10.34, from state 0 we can reach states 0, 1, and 4 along paths of ϵ-labeled arcs. We find a transition on `a` from state 1 to 2 and a transition on `b` from 4 to 5. Thus in the new automaton, there is a transition from 0 to 2 labeled `a` and from 0 to 5 labeled `b`. Notice that we have collapsed the paths $0 \rightarrow 1 \rightarrow 2$ and $0 \rightarrow 4 \rightarrow 5$ of Fig. 10.26 into single transitions with the label of the non-ϵ transition along those paths. As neither state 0, nor the states 1 and 4 that it reaches along ϵ-labeled paths are accepting states, in the new automaton, state 0 is not accepting.

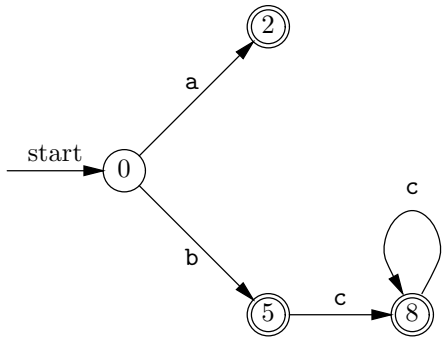


Fig. 10.35. Automaton constructed from Fig. 10.26 by eliminating ϵ-transitions. Note that this automaton accepts all and only the strings in $L(\mathbf{a} \mid \mathbf{bc}^*)$.

Next, consider the transitions out of state 2. Figure 10.34 tells us that from 2 we can reach only itself and 3 via ϵ-transitions, and so we must look for transitions out of states 2 or 3 on real symbols. Finding none, we know that there are no transitions out of state 2 in the new automaton. However, since 3 is accepting, and 2 reaches 3 by ϵ-transitions, we make 2 accepting in the new automaton.

When we consider state 5, Fig. 10.34 tells us to look at states 3, 5, 6, 7, and 9. Among these, only state 7 has a non-ϵ transition out; it is labeled `c` and goes to state 8. Thus in the new automaton, the only transition out of state 5 is a transition on `c` to state 8. We make state 5 accepting in the new automaton, since it reaches accepting state 3 following ϵ-labeled arcs.

Finally, we must look at the transitions out of state 8. By reasoning similar to that for state 5, we conclude that in the new automaton, the only transition out of state 8 is to itself and is labeled `c`. Also, state 8 is accepting in the new automaton.

Figure 10.35 shows the new automaton. Notice that the set of strings it accepts is exactly those strings in $L(\mathbf{a} \mid \mathbf{bc}^*)$, that is, the string `a` (which takes us to state 2), the string `b` (which takes us to state 5), and the strings `bc`, `bcc`, `bccc`, and so on, all of which take us to state 8. The automaton of Fig. 10.35 happens to be deterministic. If it were not, we would have to use the subset construction to convert it to a deterministic automaton, should we wish to design a program that would recognize the strings of the original regular expression. ✦

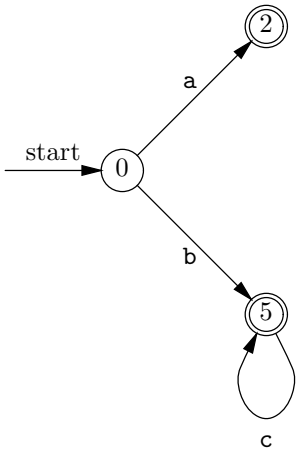


Fig. 10.36. Simpler automaton for the language $L(\mathbf{a} \mid \mathbf{bc}^*)$.

Incidentally, there is a simpler deterministic automaton that accepts the same language as Fig. 10.35. This automaton is shown in Fig. 10.36. In effect, we get the improved automaton by recognizing that states 5 and 8 are equivalent and can be merged. The resulting state is called 5 in Fig. 10.36.

EXERCISES

10.8.1: Construct automata with ϵ-transitions for the following regular expressions.

a) **aaa** *Hint*: Remember to create a new automaton for each occurrence of operand **a**.
b) $(\mathbf{ab} \mid \mathbf{ac})^*$
c) $(\mathbf{0} \mid \mathbf{1} \mid \mathbf{1}^*)^*$

10.8.2: For each of the automata constructed in Exercise 10.8.1, find the reachable sets of nodes for the graph formed from the ϵ-labeled arcs. Note that you need only construct the reachable states for the start state and the states that have non-ϵ transitions in, when you construct the automaton without ϵ-transitions.

10.8.3: For each of the automata of Exercise 10.8.1, construct an equivalent automaton without ϵ-transitions.

10.8.4: Which of the automata in Exercise 10.8.3 are deterministic? For those that are not, construct an equivalent deterministic automaton.

10.8.5*: For the deterministic automata constructed from Exercises 10.8.3 or Exercise 10.8.4, are there equivalent deterministic automata with fewer states? If so, find minimal ones.

10.8.6*: We can generalize our construction from a regular expression to an automaton with ϵ-transitions to include expressions that use the extended operators of Section 10.7. That statement is true in principle, since each of those extensions is a shorthand for an "ordinary" regular expression, by which we could replace the extended operator. However, we can also incorporate the extended operators directly into our construction. Show how to modify the construction to cover

a) the ? operator (zero or one occurrences)
b) the $^+$ operator (one or more occurrences)
c) character classes.

10.8.7: We can modify the case for concatenation in our algorithm to convert a regular expression into an automaton. In Fig. 10.28(b), we introduced an ϵ-transition from the accepting state of the automaton for R_1 to the initial state of the automaton for R_2. An alternative is to merge the accepting state of R_1 with the initial state of R_2 as shown in Fig. 10.37. Construct an automaton for the regular expression **ab*****c** using both the old and the modified algorithms.

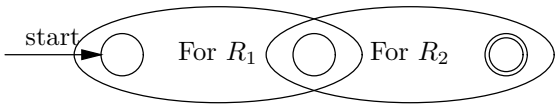


Fig. 10.37. Alternative automaton for the concatenation of two regular expressions.

10.9 From Automata to Regular Expressions

In this section, we shall demonstrate the other half of the equivalence between automata and regular expressions, by showing that for every automaton A there is a regular expression whose language is exactly the set of strings accepted by A. While we generally use the construction of the last section, where we convert "designs" in the form of regular expressions into programs, in the form of deterministic automata, this construction is also interesting and instructive. It completes the proof of the equivalence, in expressive power, of two radically different notations for describing patterns.

Our construction involves the elimination of states, one by one, from an automaton. As we proceed, we replace the labels on the arcs, which are initially sets of characters, by more complicated regular expressions. Initially, if we have label $\{\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n\}$ on an arc, we replace the label by the regular expression $\mathbf{x}_1 \mid \mathbf{x}_2 \mid \cdots \mid \mathbf{x}_n$, which represents essentially the same set of symbols, although technically the regular expression represents strings of length 1.

In general, we can think of the label of a path as the concatenation of the regular expressions along that path, or as the language defined by the concatenation of those expressions. That view is consistent with our notion of a path labeled by a string. That is, if the arcs of a path are labeled by the regular expressions $R_1, R_2, \ldots, R_n$, in that order, then the path is labeled by `w`, if and only if string `w` is in the language $L(R_1 R_2 \cdots R_n)$.

Example 10.29. Consider the path $0 \to 1 \to 2$ in Fig. 10.38. The regular expressions labeling the arcs are **a** | **b** and **a** | **b** | **c**, in that order. Thus the set of strings labeling this path are those in the language defined by the regular expression

$$(\mathbf{a} \mid \mathbf{b})(\mathbf{a} \mid \mathbf{b} \mid \mathbf{c}),$$

namely, {`aa`, `ab`, `ac`, `ba`, `bb`, `bc`}.

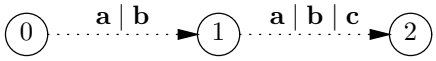


Fig. 10.38. Path with regular expressions as labels. The label of the path is the language of the concatenated regular expressions.

The State-Elimination Construction

The key step in conversion from an automaton to a regular expression is the elimination of states, which is illustrated in Fig. 10.39. We wish to eliminate state u, but we must keep the regular expression labels of the arcs so that the set of labels of the paths between any two of the remaining states does not change. In Fig. 10.39 the predecessors of state u are $s_1, s_2, \ldots, s_n$, and the successors of u are $t_1, t_2, \ldots, t_m$. Although we have shown the s's and the t's to be disjoint sets of states, there could in fact be some states common to the two groups.

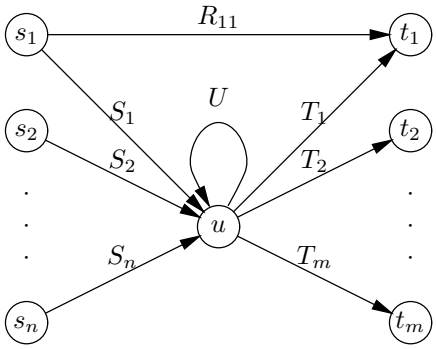


Fig. 10.39. We wish to eliminate state u.

However, if u is a successor of itself, we represent that fact explicitly by an arc labeled U. Should there be no such loop at state u, we can introduce it and give it label $\emptyset$. An arc with label $\emptyset$ is "not there," because any label of a path using that arc will be the concatenation of regular expressions including $\emptyset$. Since $\emptyset$ is the annihilator for concatenation, every such concatenation defines the empty language.

We have also shown explicitly an arc from s_1 to t_1, labeled R_{11}. In general, we suppose that for every $i = 1, 2, \ldots, n$, and for every $j = 1, 2, \ldots, m$, there is an arc from s_i to t_j, labeled by some regular expression R_{ij}. If the arc $s_i \to t_j$ is actually not present, we can introduce it and give it label $\emptyset$.

Finally, in Fig. 10.39 there is an arc from each s_i to u, labeled by regular expression S_i, and there is an arc from u to each t_j, labeled by regular expression T_j. If we eliminate node u, then these arcs and the arc labeled U in Fig. 10.39 will go away. To preserve the set of strings labeling paths, we must consider each pair s_i and t_j and add to the label of the arc $s_i \to t_j$ a regular expression that accounts for what is lost.

Before eliminating u, the set of strings labeling the paths from s_i to u, including those going around the loop $u \to u$ several times, and then from u to t_j, is described by the regular expression $S_i U^* T_j$. That is, a string in $L(S_i)$ gets us from s_i to u; a string in $L(U^*)$ gets us from u to u, following the loop zero, one, or more times. Finally, a string in $L(T_j)$ gets us from u to t_j.

Hence, after eliminating u and all arcs into or out of u, we must replace R_{ij}, the label of the arc $s_i \to t_j$, by

$$R_{ij} \mid S_i U^* T_j$$

There are a number of useful special cases. First, if $U = \emptyset$, that is, the loop on u is not really there, then $U^* = \emptyset^* = \epsilon$. Since ϵ is the identity under concatenation, $(S_i\epsilon)T_j = S_iT_j$; that is, U has effectively disappeared as it should. Similarly, if $R_{ij} = \emptyset$, meaning that there was formerly no arc from s_i to t_j, then we introduce this arc and give it label $S_iU^*T_j$, or just S_iT_j, if $U = \emptyset$. The reason we can do so is that $\emptyset$ is the identity for union, and so $\emptyset \mid S_iU^*T_j = S_iU^*T_j$.

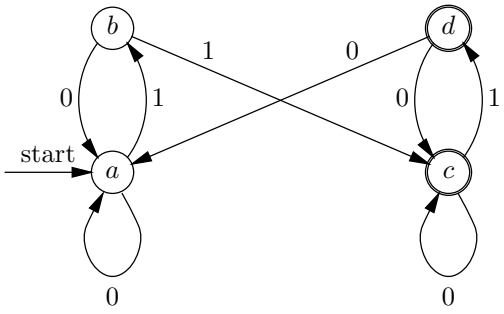


Fig. 10.40. Finite automaton for bounce filter.

◆ **Example 10.30.** Let us consider the bounce filter automaton of Fig. 10.4, which we reproduce here as Fig. 10.40. Suppose we wish to eliminate state b, which thus plays the role of state u in Fig. 10.39. State b has one predecessor, a, and two successors, a and c. There is no loop on b, and so we introduce one, labeled $\emptyset$. There is an arc from a to itself, labeled **0**. Since a is both a predecessor and a successor of b, this arc is needed in the transformation. The only other predecessor-successor pair is a and c. Since there is no arc $a \to c$, we add one with label $\emptyset$. The diagram of relevant states and arcs is as shown in Fig. 10.41.

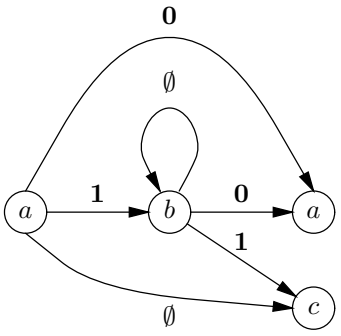


Fig. 10.41. State b, its predecessors, and successors.

For the a–a pair, we replace the label of the arc $a \rightarrow a$ by $\mathbf{0} \mid \mathbf{1}\emptyset^*\mathbf{0}$. The term $\mathbf{0}$ represents the original label of the arc, the $\mathbf{1}$ is the label of the arc $a \rightarrow b$, $\emptyset$ is the label of the loop $b \rightarrow b$, and the second $\mathbf{0}$ is the label of the arc $b \rightarrow a$. We can simplify, as described above, to eliminate $\emptyset^*$, leaving us with the expression $\mathbf{0} \mid \mathbf{10}$. That makes sense. In Fig. 10.40, the paths from a to a, going through b zero or more times, but no other state, have the set of labels $\{\mathtt{0}, \mathtt{10}\}$.

The pair a–c is handled similarly. We replace the label $\emptyset$ on the arc $a \rightarrow c$ by $\emptyset \mid \mathbf{1}\emptyset^*\mathbf{1}$, which simplifies to $\mathbf{11}$. That again makes sense, since in Fig. 10.40, the only path from a to c, via b has label $\mathtt{11}$. When we eliminate node b and change the arc labels, Fig. 10.40 becomes Fig. 10.42. Note that in this automaton, some of the arcs have labels that are regular expressions whose languages have strings of length greater than 1. However, the sets of path labels for the paths among states a, c, and d has not changed from what they were in Fig. 10.40. ✦

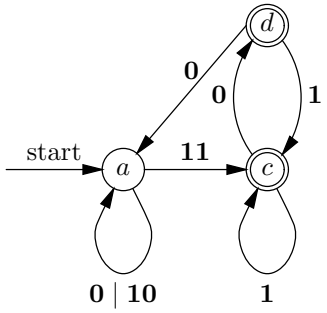


Fig. 10.42. Bounce filter automaton after eliminating state b.

Complete Reduction of Automata

To obtain a regular expression that denotes all and only the strings accepted by an automaton A we consider each accepting state t of A in turn. Every string accepted by A is accepted because it labels a path from the start state, s, to some accepting state t. We can develop a regular expression for the strings that get us from s to a particular accepting state t, as follows.

We repeatedly eliminate states of A until only s and t remain. Then, the automaton looks like Fig. 10.43. We have shown all four possible arcs, each with a regular expression as its label. If one or more of the possible arcs does not exist, we may introduce it and label it $\emptyset$.

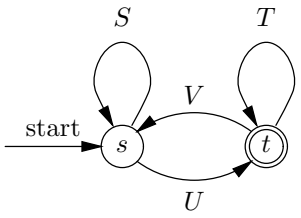


Fig. 10.43. An automaton reduced to two states.

We need to discover what regular expression describes the set of labels of paths that start at s and end at t. One way to express this set of strings is to recognize that each such path gets to t for the first time, and then goes from t to itself, zero or more times, possibly passing through s as it goes. The set of strings that take us to t for the first time is $L(S^*U)$. That is, we use strings in $L(S)$ zero or more times, staying in state s as we do, and then we follow a string from $L(U)$. We can stay in state t either by following a string in $L(T)$, which takes us from t to t, or by following a string in VS^*U, which takes us to state s, keeps us at s for a while, and then takes us back to t. We can follow zero or more strings from these two groups, in any order, which we can express as $(T \mid VS^*U)^*$. Thus a regular expression for the set of strings that get us from state s to state t is

$$S^*U(T \mid VS^*U)^* \tag{10.4}$$

There is one special case, when the start state s is itself an accepting state. Then, there are strings that are accepted because they take the automaton A from s to s. We eliminate all states but s, leaving an automaton that looks like Fig. 10.44. The set of strings that take A from s to s is $L(S^*)$. Thus we may use S^* as a regular expression to account for the contribution of accepting state s.

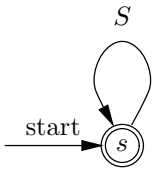


Fig. 10.44. Automaton with only the start state.

The complete algorithm for converting an automaton A with start state s to an equivalent regular expression is as follows. For each accepting state t, start with the automaton A and eliminate states until only s and t remain. Of course, we start anew with the original automaton A for each accepting state t.

If $s \neq t$, use formula (10.4) to get a regular expression whose language is the set of strings that take A from s to t. If $s = t$, use S^*, where S is the label of the arc $s \to s$, as the regular expression. Then, take the union of the regular expressions for each accepting state t. The language of that expression is exactly the set of strings accepted by A.

◆ **Example 10.31.** Let us develop a regular expression for the bounce-filter automaton of Fig. 10.40. As c and d are the accepting states, we need to

1. Eliminate states b and d from Fig. 10.40 to get an automaton involving only a and c.
2. Eliminate states b and c from Fig. 10.40 to get an automaton involving only a and d.

Since in each case we must eliminate state b, Fig. 10.42 has gotten us half way toward both goals. For (1), let us eliminate state d from Fig. 10.42. There is a path labeled **00** from c to a via d, so we need to introduce an arc labeled **00** from c to a. There is a path labeled **01** from c to itself, via d, so we need to add label **01** to the label of the loop at c, and that label becomes **1** | **01**. The resulting automaton is shown in Fig. 10.45.

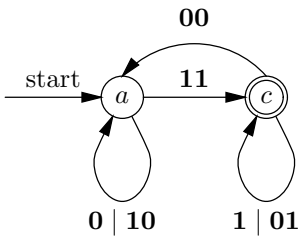


Fig. 10.45. Automaton of Fig. 10.40 reduced to states a and c.

Now for goal (2) we shall again start at Fig. 10.42 and eliminate state c this time. In Fig. 10.42 we can go from state a to state d via c, and the regular expression describing the possible strings is **111*0**.[8] That is, **11** takes us from a to c, **1*** allows us to loop at c zero or more times, and finally **0** takes us from c to d. Thus we introduce an arc labeled **111*0** from a to d. Similarly, in Fig. 10.42 we can go from d to itself, via c, by following strings in **11*0**. Thus this expression becomes the label of a loop at d. The reduced automaton is shown in Fig. 10.46.

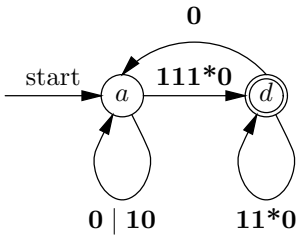


Fig. 10.46. Automaton of Fig. 10.40 reduced to states a and d.

Now we may apply the formula developed in (10.4) to the automata in Figs. 10.45 and 10.46. For Fig. 10.45, we have $S =$ **0** | **10**, $U =$ **11**, $V =$ **00**, and $T =$ **1** | **01**. Thus a regular expression denoting the set of strings that take the automaton of Fig. 10.40 from start state a to accepting state c is

$$(\mathbf{0} \mid \mathbf{10})^*\mathbf{11}\big((\mathbf{1} \mid \mathbf{01}) \mid \mathbf{00}(\mathbf{0} \mid \mathbf{10})^*\mathbf{11}\big)^* \tag{10.5}$$

and the expression denoting the strings that take state a to accepting state d is

$$(\mathbf{0} \mid \mathbf{10})^*\mathbf{111}^*\mathbf{0}\big(\mathbf{11}^*\mathbf{0} \mid \mathbf{0}(\mathbf{0} \mid \mathbf{10})^*\mathbf{111}^*\mathbf{0}\big)^* \tag{10.6}$$

---

[8] Remember that because * takes precedence over concatenation, **111*0** is parsed as **11**(**1***)**0**, and represents the strings consisting of two or more `1`'s followed by a `0`.

The expression that denotes the strings accepted by the bounce filter automaton is the union of (10.5) and (10.6), or

$$\Big((\mathbf{0} \mid \mathbf{10})^*\mathbf{11}\big((\mathbf{1} \mid \mathbf{01}) \mid \mathbf{00}(\mathbf{0} \mid \mathbf{10})^*\mathbf{11}\big)^*\Big) \mid \Big((\mathbf{0} \mid \mathbf{10})^*\mathbf{111}^*\mathbf{0}\big(\mathbf{11}^*\mathbf{0} \mid \mathbf{0}(\mathbf{0} \mid \mathbf{10})^*\mathbf{111}^*\mathbf{0}\big)^*\Big)$$

There is not much we can do to simplify this expression. There is a common initial factor $(\mathbf{0} \mid \mathbf{10})^*\mathbf{11}$, but little else in common. We can also remove the parentheses around the factor $(\mathbf{1} \mid \mathbf{01})$ in (10.5), because union is associative. The resulting expression is

$$(\mathbf{0} \mid \mathbf{10})^*\mathbf{11}\Big(\big(\mathbf{1} \mid \mathbf{01} \mid \mathbf{00}(\mathbf{0} \mid \mathbf{10})^*\mathbf{11}\big)^* \mid \mathbf{1}^*\mathbf{0}\big(\mathbf{11}^*\mathbf{0} \mid \mathbf{0}(\mathbf{0} \mid \mathbf{10})^*\mathbf{111}^*\mathbf{0}\big)^*\Big)$$

You may recall that we suggested a much simpler regular expression for the same language,

$$(\mathbf{0} \mid \mathbf{1})^*\mathbf{11}(\mathbf{1} \mid \mathbf{01})^*(\epsilon \mid \mathbf{0})$$

This difference should remind us that there can be more than one regular expression for the same language, and that the expression we get by converting an automaton to a regular expression is not necessarily the simplest expression for that language. ✥

EXERCISES

10.9.1: Find regular expressions for the automata of

a) Fig. 10.3
b) Fig. 10.9
c) Fig. 10.10
d) Fig. 10.12
e) Fig. 10.13
f) Fig. 10.17
g) Fig. 10.20

You may wish to use the shorthands of Section 10.6.

10.9.2: Convert the automata of Exercise 10.4.1 to regular expressions.

10.9.3*: Show that another regular expression we could use for the set of strings that get us from state s to state t in Fig. 10.43 is $(S \mid UT^*V)^*UT^*$.

10.9.4: How can you modify the construction of this section so that regular expressions can be generated from automata with ϵ-transitions?

✥ 10.10 Summary of Chapter 10

The subset construction of Section 10.4, together with the conversions of Sections 10.8 and 10.9, tell us that three ways to express languages have exactly the same expressive power. That is, the following three statements about a language L are either all true or all false.

1. There is some deterministic automaton that accepts all and only the strings in L.

2. There is some (possibly nondeterministic) automaton that accepts all and only the strings in L.

3. L is $L(R)$ for some regular expression R.

The subset construction shows that (2) implies (1). Evidently (1) implies (2), since a deterministic automaton is a special kind of nondeterministic automaton. We showed that (3) implies (2) in Section 10.8, and we showed (2) implies (3) in Section 10.9. Thus, all of (1), (2), and (3) are equivalent.

In addition to these equivalences, we should take away a number of important ideas from Chapter 10.

- Deterministic automata can be used as the core of programs that recognize many different kinds of patterns in strings.
- Regular expressions are often a convenient notation for describing patterns.
- There are algebraic laws for regular expressions that make union and concatenation behave in ways similar to + and ×, but with some differences.

10.11 Bibliographic Notes for Chapter 10

The reader can learn more about the theory of automata and languages in Hopcroft and Ullman [1979].

The automaton model for processing strings was first expressed in roughly the form described here by Huffman [1954], although there were a number of similar models discussed earlier and concurrently; the history can be found in Hopcroft and Ullman [1979]. Regular expressions and their equivalence to automata are from Kleene [1956]. Nondeterministic automata and the subset construction are from Rabin and Scott [1959]. The construction of nondeterministic automata from regular expressions that we used in Section 10.8 is from McNaughton and Yamada [1960], while the construction in the opposite direction, in Section 10.9, is from Kleene's paper.

The use of regular expressions as a way to describe patterns in strings first appeared in Ken Thompson's QED system (Thompson [1968]), and the same ideas later influenced many commands in his UNIX system. There are a number of other applications of regular expressions in system software, much of which is described in Aho, Sethi, and Ullman [1986].

Aho, A. V., R. Sethi, and J. D. Ullman [1986]. *Compiler Design: Principles, Techniques, and Tools*, Addison-Wesley, Reading, Mass.

Hopcroft, J. E. and J. D. Ullman [1979]. *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Reading, Mass.

Huffman, D. A. [1954]. "The synthesis of sequential switching machines," *Journal of the Franklin Institute* **257**:3-4, pp. 161–190 and 275–303.

Kleene, S. C. [1956]. "Representation of events in nerve nets and finite automata," in *Automata Studies* (C. E. Shannon and J. McCarthy, eds.), Princeton University Press.

McNaughton, R. and H. Yamada [1960]. "Regular expressions and state graphs for automata," *IEEE Trans. on Computers* **9**:1, pp. 39–47.

Rabin, M. O. and D. Scott [1959]. "Finite automata and their decision problems," *IBM J. Research and Development* **3**:2, pp. 115-125.

Thompson, K. [1968]. "Regular expression search algorithm," *Comm. ACM* **11**:6, pp. 419–422.