Chapter 4

Finite State Machines

4.1 Introduction

4.1.1 Wolf-Goat-Cabbage Puzzle

A shepherd arrives at a river bank with a wolf, a goat and a cabbage. There is a boat there that can carry them to the other bank. However, the boat can carry the shepherd and at most one other item. The shepherd's actions are limited by the following constraints: if the wolf and goat are left alone, the wolf will devour the goat, and if the goat and the cabbage are left alone, well, you can imagine...

You can get a solution quickly by rejecting certain obvious possibilities. But let us attack this problem more systematically. What is the state of affairs at any point during the passage: what is on the left bank, what is on the right bank, and where the boat is (we can deduce the contents of the boat by determining which items are absent from both banks). The state of the left bank is a subset of {w,g,c} —w for wolf, g for goat, and c for cabbage— and similarly for the right bank. The shepherd is assumed to be with the boat (the cabbage cannot steer the boat :-)), so the state of the boat is that it is: (1) positioned at the left bank, (2) positioned at the right bank, (3) in transit from left to right, or (4) in transit from right to left; let us represent these possibilities by the symbols, L, R, LR, RL, respectively.

Thus, we represent the initial state by a triple like ⟨{w,g,c}, L, {}⟩. Now what possible choices are there? The shepherd can row alone, or take one item with him in the boat, the wolf, the goat or the cabbage. These lead to the following states respectively.

⟨{w,g,c}, LR, {}⟩
⟨{g,c}, LR, {}⟩
⟨{w,c}, LR, {}⟩
⟨{w,g}, LR, {}⟩

Observe that all states except ⟨{w,c}, LR, {}⟩ are inadmissible, since someone will consume something. So, let us continue the exploration from ⟨{w,c}, LR, {}⟩.

When the shepherd reaches the other bank, the state changes from ⟨{w,c}, LR, {}⟩ to ⟨{w,c}, R, {g}⟩. Next, the shepherd has a choice: he can row back with the goat to the left bank (an obviously stupid move, because he will then be at the initial state), or he may row alone. In the first case, we get the state ⟨{w,c}, RL, {}⟩, and in the second case ⟨{w,c}, RL, {g}⟩. We may continue exploring from each of these possibilities, adding more states to the diagram. Figure 4.1 shows the initial parts of the exploration more succinctly.

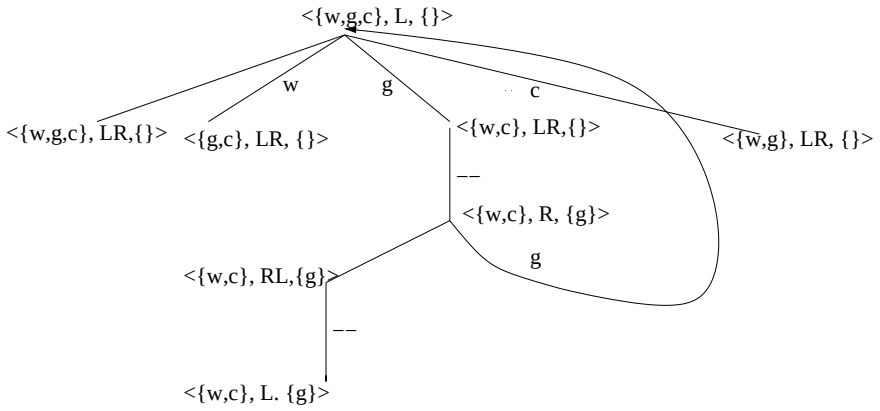


Figure 4.1: Partial State Space for the Wolf-Goat-Cabbage Problem

The important thing to note is that the number of states is finite (prove it). So, the exploration will terminate sometime.

Exercise 27

Complete the diagram. Show that a specific kind of path in the graph corresponds to a solution. How many solutions are there? Can you define states differently to derive a smaller diagram? □

Remark A beautiful treatment of this puzzle appears in Dijkstra [16]. He shows that with some systematic thinking you can practically eliminate the state-space search. You can play the game at `http://www.plastelina.net`; choose game 1. □

Exercise 28

Given is a 2×2 board which contains a tile in each of its cells; one is a blank tile (denoted by —), and the others are numbered 1 through 3. A *move* exchanges the blank tile with one of its neighbors, in its row or column. The tiles are initially placed as shown in Table 4.1.

1	3
2	—

Table 4.1: Easy version of Sam Loyd Puzzle; initial configuration

1	2
3	—

Table 4.2: Easy version of Sam Loyd Puzzle; final configuration

Show that it is impossible to reach the configuration (state) given in Table 4.2 from the initial state, given in Table 4.1.

Proof through enumeration: There are two possible moves from any state: either the blank tile moves horizontally or vertically. Enumerate the states, observe which states are reachable from which others and prove the result. It is best to treat the system as a finite state machine.

A non-enumerative proof: From a given configuration, construct a string by reading the numbers from left to right along the first row, dropping down to the next row and reading from right to left. For Table 4.1, we get 132 and for Table 4.2, we get 123. How does a move affect a string? More precisely, which property of a string is preserved by a move?

Exercise 29

(A puzzle due to Sam Loyd) This is the same puzzle as in the previous exercise played on a 4×4 board. The board contains a tile in each of its cells; one is a blank tile (denoted by —), and the others are numbered 1 through 15. A *move* exchanges the blank tile with one of its neighbors, in its row or column. The tiles are initially placed as shown in Table 4.3.

01	02	03	04
05	06	07	08
09	10	11	12
13	15	14	—

Table 4.3: Puzzle of Sam Loyd; initial configuration

Show that a sorted configuration, as shown in Table 4.4, can not be reached in a finite sequence of moves from the initial configuration.

You can do a computer search (calculate the search space size before you start), or prove this result. Create a string (a permutation) of 1 through 15 from each configuration, show that each move preserves a certain property of a permutation, that the initial configuration has the given property and the final configurations does not. Consider the number of *inversions* in a permutation.

01	02	03	04
05	06	07	08
09	10	11	12
13	14	15	—

Table 4.4: Puzzle of Sam Loyd; final configuration

4.1.2 A Traffic Light

A traffic light is in one of three states, green, yellow or red. The light changes from green to yellow to red; it cannot change from green to red, red to yellow or yellow to green. We may depict the permissible state transitions by the diagram shown in Figure 4.2.

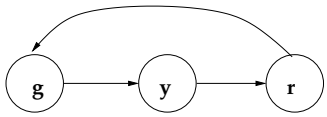


Figure 4.2: State Transitions in a Traffic Light

What causes the state transitions? It is usually the passage of time; let us say that the light changes every 30 seconds. We can imagine that an internal clock generates a pulse every 30 seconds that causes the light to change state. Let symbol p denote this pulse.

Suppose that an ambulance arrives along an intersecting road and remotely sets this light red (so that it may proceeed without interference from vehicles travelling along this road). Then, we have a new state transition, from green to red and from yellow to red, triggered by the signal from the ambulance; call this signal a. See Figure 4.3 for the full description.

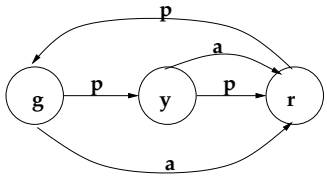


Figure 4.3: State Transitions in an Enhanced Traffic Light

4.1.3 A Pattern Matching Problem

You are given a list of English words, as in a dictionary. Find the words in which the five vowels —a,e,i,o,u— are in order. These are words like “abstemious”,

"facetious" and "sacrilegious". But not "tenacious", which contains all the vowels but not in order.

Let us design a program to solve this problem. Our program looks at each word in the dictionary in turn. For each word it scans it until it finds an "a", or fails to find it. In the latter case, it rejects the word and moves on to the next word. In the first case, it resumes its search from the point where it found "a" looking for "e". This process continues until all the vowels in order are found, or the word is rejected.

A programming hint: Sentinel How do you search for a symbol c in a string $S[0..N]$? Here is the typical strategy.

$i := 0$;
while $S[i] \neq$ "c" $\wedge\ i \leq N$ **do**
  $i := i + 1$
od ;
if $i \leq N$ **then** "success" **else** "failure" **fi**

A simpler strategy uses a "sentinel", an item at the end of the list which guarantees that the search will not fail. It simplifies AND speeds up the loop.

$S[N + 1] :=$ "c"; $i := 0$;
while $S[i] \neq$ "c" **do**
  $i := i + 1$
od ;
if $i \leq N$ **then** "success" **else** "failure" **fi** □

Bonus Programming Exercise Write a program for the pattern matching problem, and apply it to a dictionary of your choice. □

If you complete the program you will find that its structure is a mess. There are five loops, each looking for one vowel. They will be nested within a loop. A failure causes immediate exit from the corresponding loop. (Another possibility is to employ a procedure which is passed the word, the vowel and the position in the word where the search is to start.) Modification of this program is messy. Suppose we are interested in words in which exactly these vowels occur in order, so "sacrilegious" will be rejected. How will the program be modified? Suppose we don't care about the order, but we want all the vowels to be in the word; so, "tenacious" will make the cut. For each of these modifications, the program structure will change significantly.

What we are doing in all these cases is to match a pattern against a word. The pattern could be quite complex. Think about the meaning of pattern if you are searching a database of music, or a video for a particular scene. Here are some more examples of "mundane" patterns that arise in text processing.

begin	**end**	**while**	**do**	**od**	**if**	**then**	**fi**
106	107	100	101	102	103	104	105

Table 4.5: Translations of keywords

:=	;	<	≤	=	≠	>	≥	+
1000	1001	1002	1003	1004	1005	1006	1007	1008

Table 4.6: Translations of non-keywords

A longer pattern matching example Consider a language that has the following keywords:

begin end while do od if then fi

A lexical processor for the program may have to:

1. Convert every keyword to a number, as described in Table 4.5.
2. Convert every non-keyword to a distinct 2-digit number,
3. Convert every other symbol as described in Table 4.6, and
4. Ignore comments (the stuff that appears between braces) and extra white spaces.

Thus, a string like

while $i \neq n$ **do** {silly loop} $j := i + 1$ **od**

will be converted as shown in Table 4.7.

4.2 Finite State Machine

4.2.1 What is it?

Consider the problem of checking a word for vowels in order. We can describe a machine to do the checking as shown in Figure 4.4. The machine has six states, each shown as a circle. Each directed edge has a label, the name of a symbol (or set of symbols) from a specified *alphabet*.

while	i	$\neq$	n	**do**	{silly loop}	j	:=	i	+	1	**od**
100	10	1005	11	101		12	1000	10	1008	13	102

Table 4.7: Translation of a program

The machine operates as follows. Initially, the machine is in the state to which the "start" arrow points (the state labeled 1). It receives a stream of symbols. Depending on the symbol and its current state the machine determines its next state, which may be the same as the current state. Thus, in state 1, if it receives symbol "a" it transits to state 2, and otherwise (shown by the arrow looping back to the state) it stays in state 1. Any state shown by a double circle is called an *accepting state*; the remaining states are *rejecting states*. In this example, the only accepting state is 6.

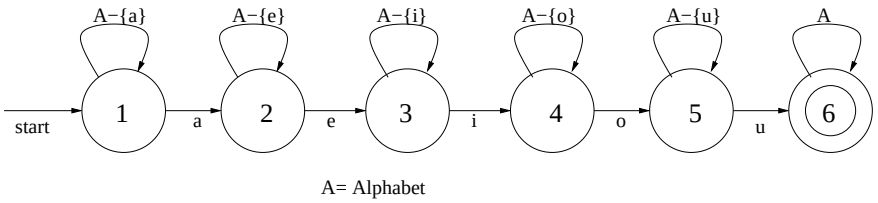


Figure 4.4: Machine to check for vowels in order

If the machine in Figure 4.4 receives the string "abstemious" then its successive states are: 1 2 2 2 2 3 3 4 5 6 6. Since its final state is an accepting state, we say that the string is *accepted* by the machine. A string that makes the machine end up in a rejecting state is said to be *rejected* by the machine.

Which state does the machine end up in for the following strings: aeio, tenacious, f, aaeeiioouu, ϵ (ϵ denotes the empty string)? Convince yourself that the machine accepts a string iff five vowels appear in order in that string.

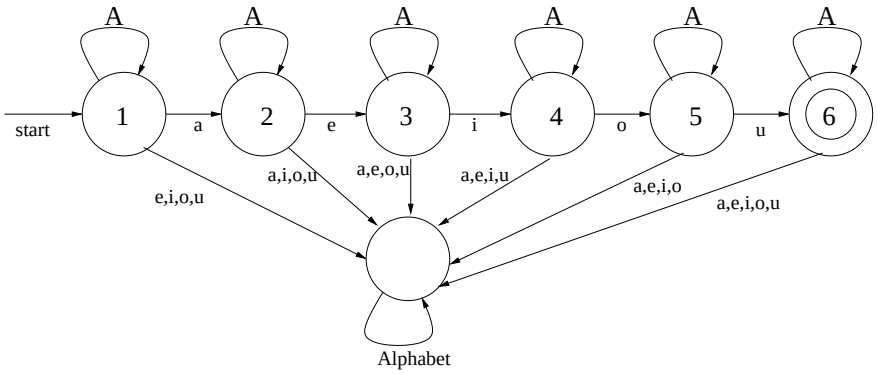
Convention Henceforth, if a transition from a state is not shown, assume that the machine transits to a permanently rejecting state (that rejecting state may not be shown either). A state is *permanently rejecting* if the state is rejecting and all transitions from this state loop back to the state.

Exercise 30

Draw a machine that accepts strings which contain the five vowels in order, and no other vowels. So, the machine will accept "abstemious", but not "sacrilegious". See Figure 4.5. □

Definitions A (deterministic) finite state machine over a given alphabet has a *finite* number of states, one state designated as the *initial* state, a subset of states designated as *accepting* and a *state transition function* that specifies the next state for each state and input symbol. The machine *accepts* or *rejects* every *finite* string over its alphabet.

Note There is a more general kind of finite state machine called a *nondeterministic* machine. The state transitions are not completely determined by the



A = Alphabet − {a,e,i,o,u}

Figure 4.5: Machine to check for exactly five vowels in order

current state and the input symbol as in the deterministic machines you have seen so far. The machine is given the power of clairvoyance so that it chooses the next state, out of a possible set of successor states, which is the "best" state for processing the remaining unseen portion of the string. □

In all cases, we deal with strings —a sequence of symbols— drawn from a fixed alphabet. A string may or may not satisfy a pattern: "abstemious" satisfies the pattern of having all five vowels in order. Here are some more examples of patterns.

Examples In solving these problems, D is the set of decimal digits, i.e., $D = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$.

1. An *unsigned integer* is a non-empty sequence of digits; see Figure 4.6.

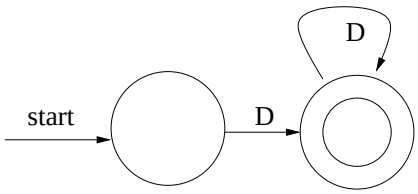


Figure 4.6: unsigned integer

2. A *signed integer* is an unsigned integer with a "+" or a "−" in the beginning; see Figure 4.7.

3. An *integer* is either an unsigned or signed integer; see Figure 4.8.

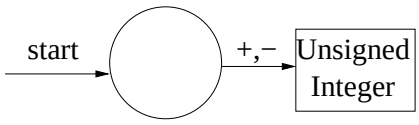


Figure 4.7: signed integer

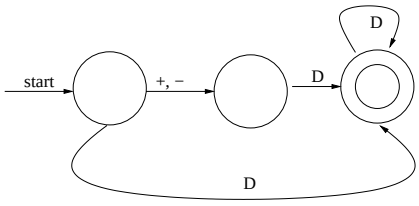


Figure 4.8: integer

4. A *fractional number* is an *integer* followed by a period followed by an unsigned integer. Note that the following are all fractional numbers: $0.30, 0.0, 000.0, -3.2$, and the following ones are not: $3, 3., .3, 3. - 2, 3.2.5$. See Figure 4.9. Here the final state of "integer" is made the starting state of "Unsigned Integer", and it is no longer an accepting state. The start edge of Integer is the start edge of Fractional Number.

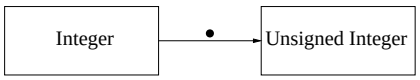


Figure 4.9: Fractional Number

5. A *number* is either a fractional number or a fractional number followed by the letter "E" followed by an integer. The following ones are all numbers: 3.2, -3.2, $3.2E5$, $0.3E1$, $3.2E + 5$. The following ones are not: $3E5$, $3.2E6.5$. In Figure 4.10, the final accepting state of "Fractional Number" remains accepting, and it is connected to the initial state of "Integer" with the edge labeled with E. The start edge of Fractional Number is the start edge of Number.

Exercise 31

Draw finite state machines that accept the strings in the following problems.

1. Any string ending in a white space. This is often called a *word*.
2. Any string in which "(" and ")" are balanced, the level of parentheses nesting is at most 3.

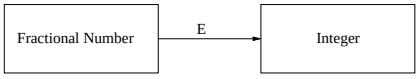


Figure 4.10: Number

3. Any string starting with "b" followed by any number of "a"s and then a "d". These strings are: "bd", "bad", "baad", "baaad", ... □

Exercise 32

1. Design finite state machines for the following problems. Assume that the alphabet is $\{0, 1\}$.
   (a) Accept all strings.
   (b) Reject all strings.
   (c) Accept if the string has an even number of 0s.
   (d) Accept if the string has an odd number of 1s.
   (e) Accept if the conditions in both (1c) and in (1d) apply. Can you find a general algorithm to construct a finite state machine from two given finite state machines, where the constructed machine accepts only if both component machines accept? What assumptions do you have to make about the component machines?
   (f) Accept if either of the conditions in (1c) and in (1d) apply. Can you find a general algorithm to construct a finite state machine from two given finite state machines, where the constructed machine accepts only if either component machine accepts? What assumptions do you have to make about the component machines?
   (g) Reject every string with an even number of 0s and odd number of 1s (that is this machine accepts exactly those strings that the machine in exercise (1e) rejects. Again, is there a general procedure to convert the machine in exercise (1e) to reject all the strings it accepts and vice-versa?
   (h) Convince yourself that you cannot design a finite state machine to accept a string that has an equal number of zeros and ones.
2. For the Wolf-Goat-Cabbage puzzle, design a suitable notation to represent each move of the shepherd using a symbol. Then, any strategy is a string. Design a finite state machine that accepts such a string and enters an accepting state if the whole party crosses over to the right bank, intact, and rejects the string otherwise.
3. For the following problems, the alphabet consists of letters (from the Roman alphabet) and digits (Arabic numerals).

(a) Accept if it contains a keyword, as given in Table 4.5.

(b) Accept if the string is a legal *identifier*: a letter followed by zero or more symbols (a letter or digit).

4. A computer has n 32-bit words of storage. What is the number of states? For a modern computer, n is around 2^{25}. Suppose each state transition takes a nanosecond (10^{-9} second). How long will it take the machine to go through all of its states?

5. Write a program (in C++ or Java) —without reference to finite state machines— that outputs "accept" if the input is a string with an even number of 0s and an odd number of 1s. Next, hand-translate the finite state machine you have designed for this problem in an earlier exercise into a program. Compare the two programs in terms of length, simplicity, design time, and execution efficiency. □

Exercise 33
Let F be a finite state machine.

1. Design a program that accepts a description of F and constructs a Java program J equivalent to F. That is, J accepts a string as input and prints "accept" or "reject". Assume that your alphabet is {0,1} , and a special symbol, say #, terminates the input string.

2. Design a program that accepts a description of F and a string s and prints "accept" or "reject" depending on whether F accepts or rejects s. □

4.2.2 Reasoning about Finite State Machines

Consider the finite state machine shown in Figure 4.11. We would like to show that the strings accepted by the machine have an even number of 0s and an odd number of 1s. The problem is complicated by the fact that there are loops.

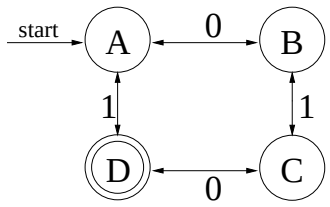


Figure 4.11: Accepts strings with even number of 0s and odd number of 1s

The strategy is to guess which string is accepted in each state, and attach that as a label to that state. This is similar to program proving. Let

$p \equiv$ this string has an even number of 0s,
$q \equiv$ this string has an even number of 1s

From A to B: if	$p \wedge q$	holds for x then	$\neg p \wedge q$	holds for $x0$
From A to D: if	$p \wedge q$	holds for x then	$p \wedge \neg q$	holds for $x1$
From B to A: if	$\neg p \wedge q$	holds for x then	$p \wedge q$	holds for $x0$
From B to C: if	$\neg p \wedge q$	holds for x then	$\neg p \wedge \neg q$	holds for $x1$
From C to B: if	$\neg p \wedge \neg q$	holds for x then	$\neg p \wedge q$	holds for $x1$
From C to D: if	$\neg p \wedge \neg q$	holds for x then	$p \wedge \neg q$	holds for $x0$
From D to C: if	$p \wedge \neg q$	holds for x then	$\neg p \wedge \neg q$	holds for $x0$
From D to A: if	$p \wedge \neg q$	holds for x then	$p \wedge q$	holds for $x1$

Table 4.8: Verifications of state transitions

A plausible annotation of the machine is shown in Figure 4.12. That is, we guess that any string for which the machine state becomes B has an odd number of 0s and an even number of 1s; similarly for the remaining state annotations.

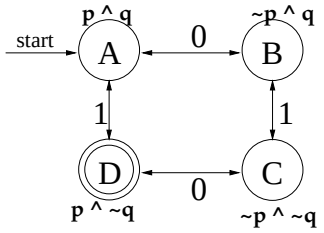


Figure 4.12: Annotation of the machine in Figure 4.11

Verification Procedure The verification procedure consists of three steps: (1) annotate each state with a predicate over finite strings (the predicate defines a set of strings, namely, the ones for which it is *true*), (2) show that the annotation on the initial state holds for the empty string, and (3) for each transition do the following verification: suppose the transition is labeled s and it is from a state annotated with b to one with c; then, show that if b holds for any string x, then c holds for xs.

For the machine in Figure 4.12, we have already done step (1). For step (2), we have to show that the empty string satisfies $p \wedge q$, that is, the empty string has an even number of 0s and 1s, which clearly holds. For step (3), we have to verify all eight transitions, as shown in Table 4.8. For example, it is straightforward to verify the transition from A to B by considering an arbitrary string x with an even number of 0s and 1s ($p \wedge q$) and proving that $x0$ has odd number of 0s and even number of 1s ($\neg p \wedge q$).

Why Does the Verification Procedure Work? It seems that we are using some sort of circular argument, but that is not so. In order to convince yourself

that the argument is not circular, construct a proof using induction. The theorem we need to prove is as follows: after processing any string x, the machine state is A, B, C or D iff x satisfies $p \wedge q$, $\neg p \wedge q$, $\neg p \wedge \neg q$ or $p \wedge \neg q$, respectively. The proof of this statement is by induction on the length of x.

For $|x| = 0$: x is an empty string and $p \wedge q$ holds for it. The machine state is A, so the theorem holds.

For $|x| = n + 1$, $n \geq 0$: use the induction hypothesis and the proofs from Table 4.8.

4.2.3 Finite State Transducers

The finite state machines we have seen so far simply accept or reject a string. So, they are useful for doing complicated tests, such as to determine if a string matches a given pattern. Such machines are called *acceptors*. Now, we will enhance the machine so that it also produces a string as output; such machines are called *transducers*. Transducers provide powerful string processing mechanism. Typically, acceptance or rejection of the input string is of no particular importance in transducers; only the construction of the appropriate output string matters.

Pictorially, we will depict a transition as shown in Figure 4.13. It denotes that on reading symbol s, the machine transits from A to B and outputs string t. The output alphabet of the machine —over which t is a string— may differ from its input alphabet.

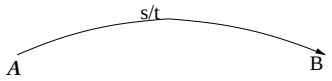


Figure 4.13: Transition Labeling in a Finite State Transducer

Example Accept any string of 0s and 1s. Squeeze each substring of 0s to a single 0 and similarly for the 1s. Thus,

000100110 becomes 01010

A solution is shown in Figure 4.14.

Verifications of Transducers How do we verify a transducer? We would like to show that the output is a function, f, of the input. For the transducer in Figure 4.14, function f is given by:

$$
\begin{array}{lll}
f(\epsilon) = \epsilon & f(0) = 0 & f(1) = 1 \\
f(x00) = f(x0) & f(x01) = f(x0)1 & \\
f(x10) = f(x1)0 & f(x11) = f(x1) &
\end{array}
$$

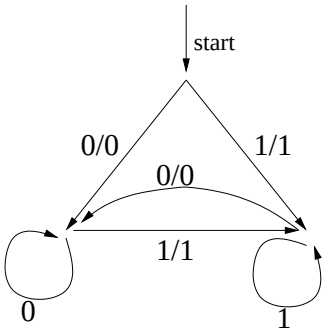


Figure 4.14: Transducer that squeezes each block to a single bit

The verification strategy for finite state acceptors is augmented as follows. As before, annotate each state by a predicate (denoting the set of strings for which the machine enters that state). Show for the initial state that the annotation is satisfied by the empty string and it outputs $f(\epsilon)$. For a transition of the form shown in Figure 4.13, if A is annotated with p and B with q, show that (1) if p holds for any string x, then q holds for xs, and (2) $f(xs) = f(x) +\!\!+ t$ (the symbol $+\!\!+$ denotes concatenation), i.e., the output in state B (which is the output in state A —assumed to be $f(x)$— concatenated with string t) is the desired output for any string for which this state is entered.

Exercise 34

Design a transducer which replaces each 0 by 01 and 1 by 10 in a string of 0s and 1s. □

Exercise 35

The input is a 0-1 string. A 0 that is both preceded and succeeded by at least three 1s is to be regarded as a 1. The first three symbols are to be reproduced exactly. The example below shows an input string and its transformation; the bit that is changed has an overline on it in the input and underline in the output.

$0110111\overline{0}11111000111$ becomes
$0110111\underline{1}11111000111$

Design a transducer for this problem and establish its correctness. □

Solution In Figure 4.15, the transitions pointing downward go to the initial state. Prove correctness by associating with each state a predicate which asserts that the string ending in that state has a certain suffix.

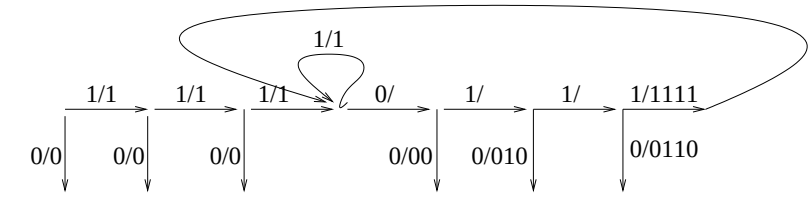


Figure 4.15: Replace 0 by 1 if it is preceded and succeeded by at least three 1s

4.2.4 Serial Binary Adder

Let us build an adding circuit (adder) that receives two binary operands and outputs their sum in binary. We will use the following numbers for illustration.

$$\begin{array}{r} 0\ 1\ 1\ 0\ 0 \\ +\ 0\ 1\ 1\ 1\ 0 \\ \hline 1\ 1\ 0\ 1\ 0 \end{array}$$

The input to the adder is a sequence of bit pairs, one bit from each operand, starting with their lowest bits. Thus the successive inputs for the given example are: (0 0) (0 1) (1 1) (1 1) (0 0). The adder outputs the sum as a sequence of bits, starting from the lowest bit; for this example, the output is 0 1 0 1 1. If there is a carry out of the highest bit it is not output, because the adder cannot be sure that it has seen all inputs. (How can we get the full sum out of this adder?)

We can design a transducer for this problem as shown in Figure 4.16. There are two states, the initial state is n and the carry state c; in state c, the current sum has a carry to the next position. The transitions are easy to justify. For instance, if the input bits are (0 1) in the n state, their sum is $0 + 1 + 0 = 1$; the last 0 in the sum represents the absence of carry in this state. Therefore, 1 is output and the machine remains in the n state. If the machine is in c state and it receives (0 0) as input, the sum is $0 + 0 + 1 = 1$; hence, it outputs 1 and transits to the n state. For input (1 1) in the c state, the sum is $1 + 1 + 1 = 3$, which is 11 in binary; hence 1 is output and the machine remains in the c state.

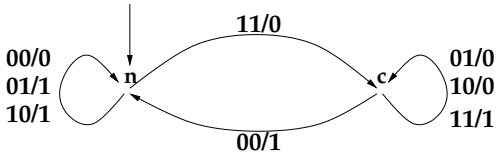


Figure 4.16: Serial Binary Adder

Exercise 36

Suppose that the input for either operand is terminated by a special symbol #. Thus, a possible input could be $(1,1)(\#,0)(\#,1)(\#,\#)$, representing the sum of 1 and 101. Redesign the serial adder to produce the complete sum.

4.2.5 Parity Generator

When a long string is transmitted over a communication channel, it is possible for some of the symbols to get corrupted. For a binary string, bits may get flipped, i.e., a 0 becomes a 1 and a 1 becomes a 0. There are many sophisticated ways for the receiver to detect such errors and request retransmissions of the relevant portions of the string. I will sketch a relatively simple technique to achieve this.

First, the sender breaks up the string into blocks of equal length. Below the block length is 3, and white spaces separate the blocks.

011 100 010 111

Next, the sender appends a bit at the end of each block so that each 4-bit block has an even number of 1s. This additional bit is called a *parity* bit, and each block is said to have *even parity*. The input string shown above becomes, after addition of parity bits,

0110 1001 0101 1111

This string is transmitted. Suppose two bits are flipped during transmission, as shown below; the flipped bits are underlined.

0110 100$\underline{0}$ 0101 $\underline{0}$111

Note that the flipped bit could be a parity bit or one of the original ones. Now each erroneous block has odd parity, and the receiver can identify all such blocks. It then asks for retransmission of those blocks. If two bits (or any even number) of bits of a block get flipped, the receiver cannot detect the error. In practice, the blocks are much longer (than 3, shown here) and many additional bits are used for error detection.

The logic at the receiver can be depicted by a finite state acceptor, see Figure 4.17. Here, a block is accepted iff it has even parity. The receiver will ask for retransmission of a block if it enters a reject state for that block (this is not part of the diagram).

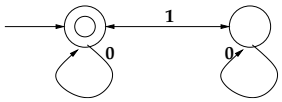


Figure 4.17: Checking the parity of a block of arbitrary length

The sender is a finite state transducer that inserts a bit after every three input bits; see figure 4.18. The start state is 0. The states have the following

meanings: in a state numbered $2i$, $0 \leq i \leq 2$, the machine has seen i bits of input of the current block (all blocks are 3 bits long) and the current block parity is even; in state $2i-1$, $1 \leq i \leq 2$, the machine has seen i bits of input of the current block and the current block parity is odd. From states 3 and 4, the machine reads one input bit and outputs the bit read and a parity bit (1 and 0, respectively).

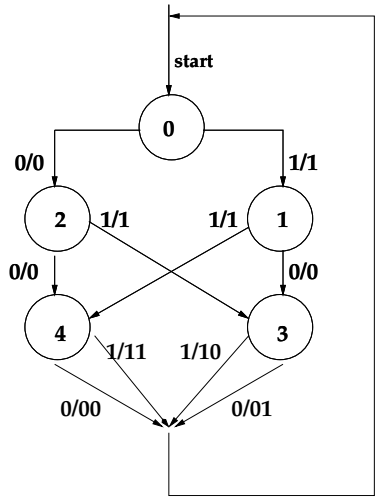


Figure 4.18: Append parity bit to get even parity; block length is 3

Exercise 37
Redesign the machine of Fig 4.17 to accept only a 4-bit string.

Exercise 38
Design a machine that accepts a string of symbols, and outputs the same string by (1) removing all white spaces in the beginning, (2) reducing all other blocks of white spaces (consecutive white spaces) to a single white space. Thus, the string (where - denotes a white space)

```
----Mary----had--a--little---lamb-
```

is output as

```
Mary-had-a-little-lamb-
```

Modify your design so that a trailing white space is not produced.

Exercise 39
A binary string is *valid* if all blocks of 0s are of even length and all blocks of 1s are of odd length. Design a machine that reads a string and outputs a Y or N for each bit. It outputs N if the current bit ends a block (a block is ended by a bit that differs from the bits in that block) and that block is not valid; otherwise the output is Y. See Table 4.9 for an example. □

0	0	1	0	0	0	1	1	0	0	1	1	0	0	1
Y	*Y*	*Y*	*Y*	*Y*	*Y*	*N*	*Y*	*N*	*Y*	*Y*	*Y*	*N*	*Y*	*Y*

Table 4.9: Checking for valid blocks

4.3 Specifying Control Logic Using Finite State Machines

4.3.1 The Game of Simon

A game that tests your memory —called *Simon*— was popular during the 80s. This is an electronic device that has a number of keys. Each key lights up on being pressed or on receiving an internal pulse.

The game is played as follows. The device lights up a random sequence of keys; call this sequence a *challenge*, and the player is expected to press the same sequence of keys. If the player's response matches the challenge, the device buzzes happily, otherwise sadly. Following a successful response, the device poses a longer challenge. The challenge for which the player loses (the player's response differs from the challenge) is a measure of the memory capability of the player.

We will represent the device by a finite state machine, ignoring the lights and buzzings. Also, we simplify the problem by having 2 keys, marked 0 and 1. Suppose the challenge is a 2-bit sequence (generated randomly within the machine). Figure 4.19 shows a finite state machine that accepts 4 bits of input (2 from the device and 2 from the player) and enters an accepting state only if the first two bits match the last two.

Exercise 40

The device expects the player to press the keys within 30 seconds. If no key is pressed in this time interval, the machine transits to the initial state (and rejects the response). Assume that 30 seconds after the last key press the device receives the symbol p (for pulse) from an internal clock. Modify the machine in Figure 4.19 to take care of this additional symbol. You may assume that p is never received during the input of the first 2 bits. □

Remark Finite state machines are used in many applications where the passage of time or exceeding a threshold level for temperature, pressure, humidity, carbon-monoxide, or similar analog measures, causes a state change. A sensor converts the analog signals to digital signals which are then processed by a finite state machine. A certain luxury car has rain sensors mounted in its windshield that detect rain and turn on the wipers. (Be careful when you go to a car wash with this car.) □

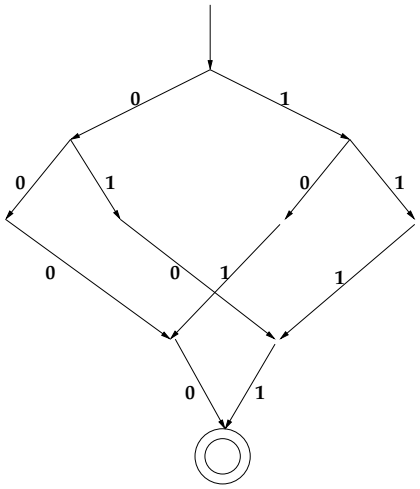


Figure 4.19: A Simplified game of Simon

4.3.2 Soda Machine

A soda machine interacts with a user to deliver a product. The user provides the input string to the machine by pushing certain buttons and depositing some coins. The machine dispenses the appropriate product provided adequate money has been deposited. Additionally, it may return some change and display warning messages.

We consider a simplified soda machine that dispenses two products, A and B. A costs 15¢ and B 20¢. The machine accepts only nickels and dimes. It operates according to the following rules.

1. If the user presses the appropriate button —a for A and b for B— after depositing at least the correct amount —15¢ for A and 20¢ for B— the machine dispenses the item and returns change, if any, in nickels.

2. If the user inserts additional coins after depositing 20¢ or more, the last coin is returned.

3. If the user asks for an item before depositing the appropriate amount, a warning light flashes for 2 seconds.

4. The user may cancel the transaction at any time. The deposit, if any, is returned in nickels.

The first step in solving the problem is to decide on the input and output alphabets. I propose the following input alphabet:

$\{n,\ d,\ a,\ b,\ c\}$.

Insertion of a nickel (resp., dime) is represented by n (resp., d), pressing the buttons for A (resp., B) is represented by a (resp., b), and pressing the cancellation button is represented by c.

The output alphabet of the machine is

$$\{n,\ d,\ A,\ B,\ w\}.$$

Returning a nickel (resp., dime) is represented by n (resp., d). A string like nnn represents the return of 3 nickels. Dispensing A (resp., B) is represented by A (resp., B). Flashing the warning light is represented by w.

The machine shown in Figure 4.20 has its states named after the multiples of 5, denoting the total deposit at any point. No other deposit amount is possible, no other number lower than 25 is divisible by 5 (a nickel's value) and no number higher than 25 will be accepted by the machine. (Why do we have a state 25 when the product prices do not exceed 20?) The initial state is 0. In Figure 4.20, all transitions of the form $c/nnn\ldots$ are directed to state 0.

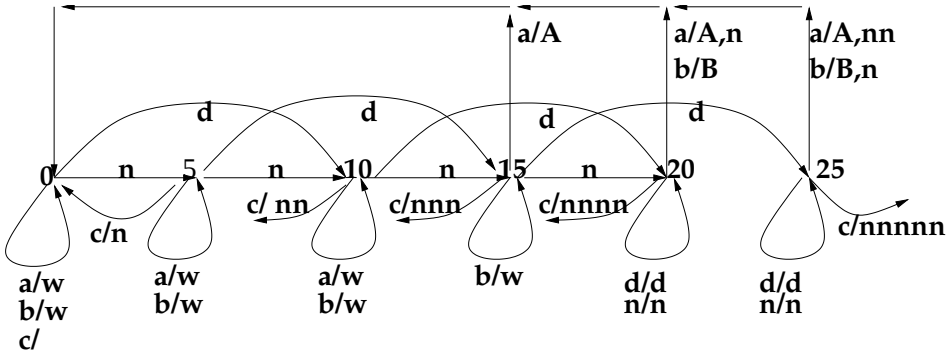


Figure 4.20: Soda Machine; transitions $c/nnn\ldots$ go to state 0

Exercise 41
Design a soda machine that dispenses three products costing 35¢, 55¢ and 75¢. It operates in the same way as the machine described here. □

4.4 Regular Expressions

We have seen so far that a finite state machine is a convenient way of defining certain patterns (but not all). We will study another way, *regular expressions*, of defining patterns that is exactly as powerful as finite state machines: the same set of patterns can be defined by finite state machines and regular expressions.

Suppose we want to search a file for all occurrences of *simple integer*, where a simple integer is either 0 or a non-zero digit followed by any number of digits. We

can define the pattern by a finite state machine. We can also write a definition using a regular expression:

$$0 \mid (1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9)(0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9)^*$$

4.4.1 What is a Regular Expression?

A regular expression is like an arithmetic expression. An arithmetic expression, such as $3 * (x + 5)$, has operands 3, x, 5 and operators $+$ and $*$. For a regular expression, we have an associated *alphabet* that plays the role of constants, like 3 and 5. A regular expression may have operands (like x in the arithmetic expression) that are names of other regular expressions. We have three operators: *concatenation* (denoted by a period or simple juxtaposition), *union* or *alternation* (denoted by $|$) and *closure* (denoted by *). The first two operators are binary infix operators like the arithmetic operators plus and times; the last one is a unary operator, like unary minus, which is written after its operand as a superscript. More formally, a regular expression defines a set of strings, and it has one of the following forms:

the symbol ϕ, denoting an empty set of strings, or
the symbol ϵ, denoting a set with an empty string, or
a symbol of the alphabet,
  denoting a set with only one string which is that symbol, or
pq, where p and q are regular expressions,
  denoting a set of strings obtained by *concatenation*
  of strings from p with those of q, or
$p \mid q$, where p and q are regular expressions,
  denoting the *union* of the sets corresponding to p and q, or
p^*, where p is a regular expression,
  denoting the *closure* of
  (zero or more concatenations of the strings in) the set corresponding to p.

Examples of Regular Expressions Let the alphabet be $\{\alpha, \beta, \gamma\}$.

ϵ, ϕ, α, β, γ
$\epsilon\alpha$, $\alpha\beta$, $\alpha\phi$, $((\epsilon\phi)\epsilon)\phi$
$(\alpha\beta \mid \alpha((\alpha\beta)\epsilon)) \mid (\alpha \mid \epsilon)$
$((\alpha\beta)^*(\alpha((\alpha\beta)\epsilon))^*)((\alpha \mid \epsilon) \mid \alpha\beta)^*$
$((\alpha(\alpha\beta)^* \mid (\beta\alpha)^*\beta)^*\alpha\beta) \mid ((\alpha\gamma)^*(\gamma\alpha)^*)$ □

Binding Power To avoid execessive number of parentheses, we impose a precedence order (binding power) over the operators; operators in order of increasing binding power are: alternation, concatenation and closure. So, $\alpha\beta^* \mid \alpha^*\beta$ is $(\alpha(\beta^*)) \mid ((\alpha^*)\beta)$. □

Each regular expression stands for a set of strings.

Name	Regular expression	Strings
$p =$	$\alpha \mid \alpha\beta \mid \alpha\alpha\beta$	$\{\alpha, \alpha\beta, \alpha\alpha\beta\}$
$q =$	$\beta \mid \beta\gamma \mid \beta\beta\gamma$	$\{\beta, \beta\gamma, \beta\beta\gamma\}$
	pq	$\{\alpha\beta, \alpha\beta\gamma, \alpha\beta\beta\gamma,$ $\alpha\beta\beta, \alpha\beta\beta\gamma, \alpha\beta\beta\beta\gamma,$ $\alpha\alpha\beta\beta, \alpha\alpha\beta\beta\gamma, \alpha\alpha\beta\beta\beta\gamma\}$
	$p \mid q$	$\{\alpha, \alpha\beta, \alpha\alpha\beta, \beta, \beta\gamma, \beta\beta\gamma\}$
	p^*	$\{\epsilon,$ $\alpha, \alpha\beta, \alpha\alpha\beta,$ $\alpha\alpha, \alpha\alpha\beta, \alpha\alpha\alpha\beta,$ $\alpha\beta\alpha, \alpha\beta\alpha\beta,$ $\alpha\beta\alpha\alpha\beta, \ldots\}$

Exercise 42

1. With the given alphabet what are the strings in $\epsilon\alpha$, $\alpha\epsilon$, $\phi\alpha$, $\phi\epsilon$, $\epsilon\phi$?
2. What is the set of strings $(\alpha\beta \mid \alpha\alpha\beta)(\beta\alpha \mid \epsilon\alpha\beta\epsilon)$?
3. What is the set of strings $(\alpha \mid \beta)^*$? □

Note on closure One way to think of closure is as follows:

$$p^* \;=\; \epsilon \mid p \mid pp \mid ppp \mid \ldots$$

The right side is not a legal regular expression because it has an infinite number of terms in it. The purpose of closure is to make the right side a regular expression. □

4.4.2 Examples of Regular Expressions

1. $a \mid bc^*d$ is $\{a, bd, bcd, bccd, bcccd, \ldots\}$.
2. All integers are defined by $(0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9)^*$. How would you avoid the "empty" integer?
3. Define a simple integer to be either a 0 or a nonzero digit followed by any number of digits:
   $0 \mid (1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9)(0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9)^*$.
   Show that 3400 is an integer, but 0034 and 00 are not.

   The definition of a simple integer can be simplified by naming certain subexpressions of the regular expression.

   $$\begin{aligned} \mathit{Digit} &= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\ \mathit{pDigit} &= 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\ \mathit{simple\_integer} &= 0 \mid (\mathit{pDigit}\ \mathit{Digit}^*) \end{aligned}$$

4. Legal identifiers in Java. Note that a single letter is an identifier.

$$\begin{aligned} &\mathit{Letter} = A \mid B \mid \ldots\ Z \mid a \mid b \mid \ldots\ z \\ &\mathit{Digit} = 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\ &\mathit{identifier} = \mathit{Letter}(\mathit{Letter} \mid \mathit{Digit})^* \end{aligned}$$

5. Words in which all the vowels appear in order:

   $$(\mathit{Letter}^*)a(\mathit{Letter}^*)e(\mathit{Letter}^*)i(\mathit{Letter}^*)o(\mathit{Letter}^*)u(\mathit{Letter}^*)$$

6. An *increasing integer* is a nonempty integer whose digits are strictly increasing. The following regular expression definition of increasing integer is due to Ben Finkel (class of Fall 2009).

   Let us define int_i, for $0 \leq i \leq 9$, to be an increasing integer whose first digit is greater than or equal to i. Then, an increasing integer is

   $$\mathit{IncInt} = int_0$$

   To define int_i, $0 \leq i \leq 9$, it is easier to start with the highest index, 9, and work downwards.

   $$\begin{aligned} int_9 &= 9 \\ int_8 &= 8 \mid 8\ int_9 \mid int_9 = 8 \mid (\epsilon \mid 8)int_9 \\ int_7 &= 7 \mid 7\ int_8 \mid int_8 = 7 \mid (\epsilon \mid 7)int_8 \\ int_i &= i \mid i\ int_{i+1} \mid int_{i+1} = i \mid (\epsilon \mid i)int_{i+1}, \text{ for } 0 \leq i < 9 \end{aligned}$$

Exercise 43

The following solution has been proposed for the *increasing integer* problem: 0*1*2*3*4*5*6*7*8*9*. What is wrong with it?

Solution The given expression generates non-decreasing strings, not just increasing strings. So, 11 is generated.

The following solution almost corrects the problem; each integer is generated at most once, but it generates the empty string. Write $[i]$ as a shorthand for $\epsilon|i$.

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

4.4.3 Algebraic Properties of Regular Expressions

We give some of the essential identities of the Regular Expression algebra.

1. Identity for Union $(\phi \mid R) = R$, $(R \mid \phi) = R$
2. Identity for Concatenation $(\epsilon R) = R$, $(R\epsilon) = R$
3. $(\phi R) = \phi$, $(R\phi) = \phi$
4. Commutativity of Union $(R \mid S) = (S \mid R)$

5. Associativity of Union $((R \mid S) \mid T) = (R \mid (S \mid T))$

6. Associativity of Concatenation $((RS)T) = (R(ST))$

7. Distributivity of Concatenation over Union
   $(R(S \mid T)) = (RS \mid RT)$
   $((S \mid T)R) = (SR \mid TR)$

8. Idempotence of Union $(R \mid R) = R$

9. Closure
   $\phi^* = \epsilon$
   $RR^* = R^*R$
   $R^* = (\epsilon \mid RR^*)$

Exercise 44

Write regular expressions for the following sets of binary strings.

1. Strings whose numerical values are even.

2. Strings whose numerical values are non-zero.

3. Strings that have at least one 0 and at most one 1.

4. Strings in which the 1s appear contiguously.

5. Strings in which every substring of 1s is of even length. □

Exercise 45

Define the language over the alphabet $\{0,\ 1,\ 2\}$ in which consecutive symbols are different. □

Exercise 46

What are the languages defined by

1. $(0^*1^*)^*$

2. $(0^* \mid 1^*)^*$

3. ϵ^*

4. $(0^*)^*$

5. $(\epsilon \mid 0^*)^*$ □

4.4.4 Solving Regular Expression Equations

We find it convenient to write a long definition, such as that of *IncInt* in page 93, by using a number of sub-definitions, such as int_9 through int_0. It is often required to eliminate all such variables from a regular expression and get a (long) expression in which only the symbols of the alphabet appear. We can do it easily for a term like int_8 that is defined to be $8 \mid (\epsilon \mid 8)int_9$: replace int_9 by its definition to get $8 \mid (\epsilon \mid 8)9$, which is $(8 \mid 9 \mid 89)$. However, in many cases the equations are recursive, so this trick will not work. For example, let p and q be sets of binary strings that have even number of 1s and odd number of 1s, respectively. Then,

$$\begin{aligned} p &= 0^* \mid 0^*1q \\ q &= 0^*1p \end{aligned}$$

Replace q in the definition of p to get

$$p = 0^* \mid 0^*10^*1p$$

This is a recursive equation. We see that p is of the form $p = 0^* \mid \alpha p$ where string α does not name p (here, $\alpha = 0^*10^*1$). Then $p = 0^*\alpha^*$, that is, $p = 0^*(0^*10^*1)^*$. Hence, $q = 0^*1(0^*(0^*10^*1)^*)$.

A more elaborate example It is required to define a binary string that is a multiple of 3 considered as a number. Thus, 000 and 011 are acceptable strings, but 010 is not. Let b_i, $0 \leq i \leq 2$, be a binary string that leaves a remainder of i after division by 3. We have:

$$b_0 = \epsilon \mid b_00 \mid b_11 \quad (1)$$
$$b_1 = b_01 \mid b_20 \quad (2)$$
$$b_2 = b_10 \mid b_21 \quad (3)$$

These equations can be understood by answering the following questions: on division by 3 if p leaves a remainder of i, $0 \leq i \leq 2$, then what are the remainders left by $p0$ and $p1$? Let $value(p)$ denote the value of string p as an integer; then, $value(p0) = 2 \times value(p)$ and $value(p1) = 2 \times value(p) + 1$.

We solve these equations to create a regular expression for b_0. We had noted previously that the solution to $p = \epsilon \mid \alpha p$, where string α does not name p, is $p = \alpha^*$. We generalize this observation.

Observation: Given that $p = a \mid \alpha p$, where strings a and α do not name p, we have $p = \alpha^*a$. Dually, given that $p = a \mid p\alpha$, where strings a and α do not name p, we have $p = a\alpha^*$.

We prove validity of the second observation. Substitute $a\alpha^*$ for p in the equation and show that $a \mid p\alpha = p$.

$$\begin{aligned} & a \mid p\alpha \\ = \quad & \{\text{replace } p \text{ by } a\alpha^*\} \end{aligned}$$

$$
\begin{array}{ll}
 & a \mid a\alpha^*\alpha \\
= & \{\text{apply (7) in Section 4.4.3}\} \\
 & a(\epsilon \mid \alpha^*\alpha) \\
= & \{\alpha^* = \epsilon \mid \alpha^*\alpha, \text{ see (9) in Section 4.4.3}\} \\
 & a\alpha^* \\
= & \{\text{replace } a\alpha^* \text{ by } p\} \\
 & p
\end{array}
$$

□

Apply this observation to (3) with p, a, α set to b_2, b_10, 1 to get

$$b_2 = b_101^*$$

In the RHS of (2), replace b_2 by b_101^*:

$$b_1 = b_01 \mid b_101^*0$$

Apply the observation on this equation with p, a, α set to b_1, b_01, 01^*0.

$$b_1 = b_01(01^*0)^*$$

Replace b_1 in the RHS of (1) by the RHS above.

$$
\begin{array}{ll}
b_0 & = \epsilon \mid b_00 \mid b_01(01^*0)^*1, \text{ or} \\
 & = \epsilon \mid b_0(0 \mid 1(01^*0)^*1)
\end{array}
$$

Apply the observation with p, a, α set to b_0, ϵ, $(0 \mid 1(01^*0)^*1)$.

$$
\begin{array}{ll}
b_0 & = \epsilon\ (0 \mid 1(01^*0)^*1)^*, \text{ or} \\
 & = (0 \mid 1(01^*0)^*1)^*
\end{array}
$$

Exercise 47

The definition of b_0 allows the empty string to be regarded as a number. Fix the definitions so that a number is a non-empty string. Make sure that your fix does not result in every number starting with 0.

Solution The simple fix is to modify equation (1).

$$b_0 = 0 \mid b_00 \mid b_11 \qquad (1')$$

But this has the effect of every number starting with 0. To avoid this problem, modify equation (2) to include 1 as a possibility for b_1. The equations now become

$$
\begin{array}{lll}
b_0 & = 0 \mid b_00 \mid b_11 & (1') \\
b_1 & = 1 \mid b_01 \mid b_20 & (2') \\
b_2 & = b_10 \mid b_21 & (3)
\end{array}
$$

Solve these equations.

4.4.5 From Regular Expressions to Machines

Regular expressions and finite state machines are equivalent: for each regular expression R there exists a finite state machine F such that the set of strings in R is the set of strings accepted by F. The converse also holds. I will not prove this result, but will instead give an informal argument.

First, let us construct a machine to recognize the single symbol b. The machine has a start state S, an accepting state F and a rejecting state G. There is a transition from S to F labeled b, a transition from S to G labeled with all other symbols and a transition from F to G labeled with all symbols. The machine remains in G forever (i.e., for all symbols the machine transits from G to G), see Figure 4.21. In this figure, *Alph* stands for the alphabet.

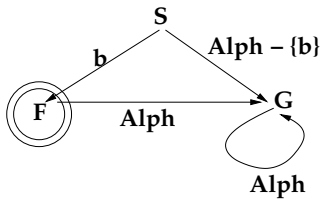


Figure 4.21: Machine that accepts b

Using our convention about permanently rejecting states, see page 77, we will simplify Figure 4.21 to Figure 4.22.

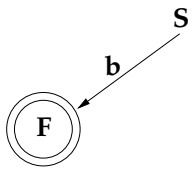


Figure 4.22: Machine that accepts b, simplified

How do we construct a machine to recognize concatenation? Suppose we have a machine that accepts a regular expression p and another machine that accepts q. Suppose p's machine has a single accepting state. Then we can merge the two machines by identifying the accepting state of the first machine with the start state of the second. We will see an example of this below. You may think about how to generalize this construction when the first machine has several accepting states.

Next, let us consider closure. Suppose we have to accept c^*. The machine in Figure 4.23 does the job.

Now, let us put some of these constructions together and build a machine to recognize bc^*. The machine is shown in Figure 4.24.

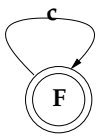


Figure 4.23: Machine that accepts c^*

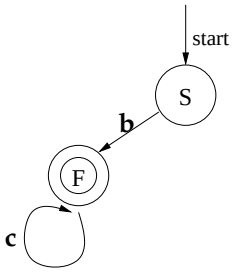


Figure 4.24: Machine that accepts bc^*

You can see that it is a concatenation of a machine that accepts b and one that accepts c^*. Next, let us construct a machine that accepts $bc^* \mid cb^*$. Clearly, we can build machines for both bc^* and cb^* separately. Building their union is easy, because bc^* and cb^* start out with different symbols, so we can decide which machine should scan the string, as shown in Figure 4.25.

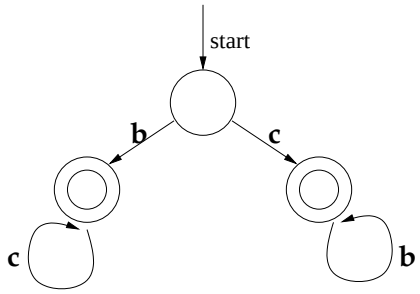


Figure 4.25: Machine that accepts $bc^* \mid cb^*$

Exercise 48

Construct a machine to accept $bc^* \mid bd^*$. □

4.4.6 Regular Expressions in Practice; from GNU Emacs

The material in this section is taken from the online GNU Emacs manual[1].

Regular expressions have a syntax in which a few characters are special constructs and the rest are "ordinary". An ordinary character is a simple regular expression which matches that same character and nothing else. The special characters are '$', '^', '.', '*', '+', '?', '[', ']' and '\'. Any other character appearing in a regular expression is ordinary, unless a '\' precedes it.

For example, 'f' is not a special character, so it is ordinary, and therefore 'f' is a regular expression that matches the string 'f' and no other string. (It does *not* match the string 'ff'.) Likewise, 'o' is a regular expression that matches only 'o'. (When case distinctions are being ignored, these regexps also match 'F' and 'O', but we consider this a generalization of "the same string", rather than an exception.)

Any two regular expressions A and B can be concatenated. The result is a regular expression which matches a string if A matches some amount of the beginning of that string and B matches the rest of the string.

As a simple example, we can concatenate the regular expressions 'f' and 'o' to get the regular expression 'fo', which matches only the string 'fo'. Still trivial. To do something nontrivial, you need to use one of the special characters. Here is a list of them.

'. (Period)' is a special character that matches any single character except a newline. Using concatenation, we can make regular expressions like 'a.b' which matches any three-character string which begins with 'a' and ends with 'b'.

'*' is not a construct by itself; it is a postfix operator, which means to match the preceding regular expression repetitively as many times as possible. Thus, 'o*' matches any number of 'o's (including no 'o's).

'*' always applies to the *smallest* possible preceding expression. Thus, 'fo*' has a repeating 'o', not a repeating 'fo'. It matches 'f', 'fo', 'foo', and so on.

The matcher processes a '*' construct by matching, immediately, as many repetitions as can be found. Then it continues with the rest of the pattern. If that fails, backtracking occurs, discarding some of the matches of the '*'-modified construct in case that makes it possible to match the rest of the pattern. For example, matching 'ca*ar' against the string 'caaar', the 'a*' first tries to match all three 'a's; but the rest of the pattern is 'ar' and there is only 'r' left to match, so this try fails. The next alternative is for 'a*' to match only two 'a's. With this choice, the rest of the regexp matches successfully.

'+' is a postfix character, similar to '*' except that it must match the preceding expression at least once. So, for example, 'ca+r' matches the strings 'car' and 'caaaar' but not the string 'cr', whereas 'ca*r' matches all three strings.

'?' is a postfix character, similar to '*' except that it can match the preceding expression either once or not at all. For example, 'ca?r' matches 'car' or 'cr'; nothing else.

[1] Copyright (C) 1989,1991 Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA

‘[...]’ is a "character set", which begins with ‘[’ and is terminated by a ‘]’. In the simplest case, the characters between the two brackets are what this set can match.

Thus, ‘[ad]’ matches either one ‘a’ or one ‘d’, and ‘[ad]*’ matches any string composed of just ‘a’s and ‘d’s (including the empty string), from which it follows that ‘c[ad]*r’ matches ‘cr’, ‘car’, ‘cdr’, ‘caddaar’, etc.

You can also include character ranges a character set, by writing two characters with a ‘-’ between them. Thus, ‘[a-z]’ matches any lower-case letter. Ranges may be intermixed freely with individual characters, as in ‘[a-z$%.]’, which matches any lower case letter or ‘$’, ‘%’ or period.

Note that the usual special characters are not special any more inside a character set. A completely different set of special characters exists inside character sets: ‘]’, ‘-’ and ‘^’.

To include a ‘]’ in a character set, you must make it the first character. For example, ‘[]a]’ matches ‘]’ or ‘a’. To include a ‘-’, write ‘-’ at the beginning or end of a range. To include ‘^’, make it other than the first character in the set.

‘[^ ...]’ ‘[^’ begins a "complemented character set", which matches any character except the ones specified. Thus, ‘[^a-z0-9A-Z]’ matches all characters *except* letters and digits.

‘^’ is not special in a character set unless it is the first character. The character following the ‘^’ is treated as if it were first (‘-’ and ‘]’ are not special there).

A complemented character set can match a newline, unless newline is mentioned as one of the characters not to match. This is in contrast to the handling of regexps in programs such as ‘grep’.

‘^’ is a special character that matches the empty string, but only at the beginning of a line in the text being matched. Otherwise it fails to match anything. Thus, ‘^foo’ matches a ‘foo’ which occurs at the beginning of a line.

‘$’ is similar to ‘^’ but matches only at the end of a line. Thus, ‘xx*$’ matches a string of one ‘x’ or more at the end of a line.

‘\’ has two functions: it quotes the special characters (including ‘\’), and it introduces additional special constructs.

Because ‘\’ quotes special characters, ‘\$’ is a regular expression which matches only ‘$’, and ‘\[’ is a regular expression which matches only ‘[’, etc.

For the most part, ‘\’ followed by any character matches only that character. However, there are several exceptions: two-character sequences starting with ‘\’ which have special meanings. The second character in the sequence is always an ordinary character on their own. Here is a table of ‘\’ constructs.

‘\|’ specifies an alternative. Two regular expressions A and B with ‘\|’ in between form an expression that matches anything that either A or B matches.

Thus, ‘foo\|bar’ matches either ‘foo’ or ‘bar’ but no other string.

‘\|’ applies to the largest possible surrounding expressions. Only a surrounding ‘\(... \)’ grouping can limit the scope of ‘\|’.

Full backtracking capability exists to handle multiple uses of ‘\|’.

‘\(... \)’ is a grouping construct that serves three purposes:

1. To enclose a set of '\|' alternatives for other operations. Thus, '\(foo\|bar\)x' matches either 'foox' or 'barx'.

2. To enclose a complicated expression for the postfix operators '*', '+' and '?' to operate on. Thus, 'ba\(na\)*' matches 'bananana', etc., with any (zero or more) number of 'na' strings.

3. To mark a matched substring for future reference.

This last application is not a consequence of the idea of a parenthetical grouping; it is a separate feature which is assigned as a second meaning to the same '\(... \)' construct. In practice there is no conflict between the two meanings. Here is an explanation of this feature:

'\D' after the end of a '\(... \)' construct, the matcher remembers the beginning and end of the text matched by that construct. Then, later on in the regular expression, you can use '\' followed by the digit D to mean "match the same text matched the Dth time by the '\(... \)' construct."

The strings matching the first nine '\(... \)' constructs appearing in a regular expression are assigned numbers 1 through 9 in order that the open-parentheses appear in the regular expression. '\1' through '\9' refer to the text previously matched by the corresponding '\(... \)' construct.

For example, '\(.*\)\1' matches any newline-free string that is composed of two identical halves. The '\(.*\)' matches the first half, which may be anything, but the '\1' that follows must match the same exact text.

If a particular '\(... \)' construct matches more than once (which can easily happen if it is followed by '*'), only the last match is recorded.

'\`' matches the empty string, provided it is at the beginning of the buffer.

'\'' matches the empty string, provided it is at the end of the buffer.

'\b' matches the empty string, provided it is at the beginning or end of a word. Thus, '\bfoo\b' matches any occurrence of 'foo' as a separate word. '\bballs?\b' matches 'ball' or 'balls' as a separate word.

`\B' matches the empty string, provided it is *not* at the beginning or end of a word.

'\<' matches the empty string, provided it is at the beginning of a word.

'\>' matches the empty string, provided it is at the end of a word.

'\w' matches any word-constituent character. The syntax table determines which characters these are.

'\W' matches any character that is not a word-constituent.

'\sC' matches any character whose syntax is C. Here C is a character which represents a syntax code: thus, 'w' for word constituent, '(' for open-parenthesis, etc. Represent a character of whitespace (which can be a newline) by either '-' or a space character.

'\SC' matches any character whose syntax is not C.

The constructs that pertain to words and syntax are controlled by the setting of the syntax table.

Here is a complicated regexp, used by Emacs to recognize the end of a sentence together with any whitespace that follows. It is given in Lisp syntax to enable you to distinguish the spaces from the tab characters. In Lisp syntax, the

string constant begins and ends with a double-quote. '`\"`' stands for a double-quote as part of the regexp, '`\\`' for a backslash as part of the regexp, '`\t`' for a tab and '`\n`' for a newline.

```
"[.?!][]\"')]*\\($\\|\t\\|  \\)[ \t\n]*"
```

This contains four parts in succession: a character set matching period, '?', or '!'; a character set matching close-brackets, quotes, or parentheses, repeated any number of times; an alternative in backslash-parentheses that matches end-of-line, a tab, or two spaces; and a character set matching whitespace characters, repeated any number of times.

To enter the same regexp interactively, you would type TAB to enter a tab, and 'C-q C-j' to enter a newline. You would also type single slashes as themselves, instead of doubling them for Lisp syntax.

4.5 Enhancements to Finite State Machines

Finite state machines may be enhanced —by adding structures to states and transitions— which make them effective in specifying a variety of hardware and software systems. They are particularly effective in specifying control systems. We will study a few enhancements in this section. Much of this material is inspired by Statecharts [20], introduced by David Harel in the mid 80s. Statecharts have been very influential in software specifications and designs, and they have inspired modeling systems such as UML [17]. We will cover a very small subset of the theory of statecharts, and adopt different notations, terminology and semantics.

Consider the finite state machine shown in Figure 4.26 (which is same as Figure 4.11 of Section 4.2.2, page 81). It accepts a string which has even number of 0s and odd number of 1s. The machine can be described more succinctly by employing two boolean variables, *zero* and *one*, which convey the parity of the number of zeroes and ones in the string. We can then express the logic using the machine in Figure 4.27.

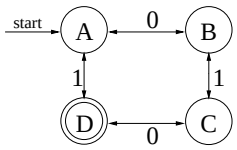


Figure 4.26: Accept strings with even number of 0s and odd number of 1s

In Figure 4.27, each transition has the form $x \rightarrow S$, where x is a symbol, 0 or 1, and S is an assignment to a variable (Unfortunately, I have to use $\tilde{}zero$ in the figure, instead of $\neg zero$, due to limitations of available fonts). The initial state has an associated transition that assigns the initial values of

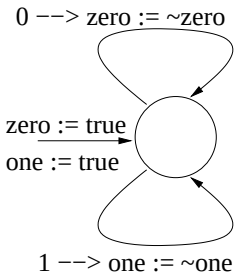


Figure 4.27: Count parity of 0s and 1s

the variables. There is no explicit accepting state; we have to specify that the string is accepted if $zero \wedge \neg one$ holds. As this example shows, we carry part of the state information in the variables.

Finite state machines embody both control and data. Using variables to encode data leaves us with the problem of encoding control alone, often a simpler task. In this section, we develop the notations and conventions for manipulating variable values. Additionally, we describe how a state may have internal structure; a state can itself be a finite state machine.

Note: We had earlier used the notation s/t in finite state transducers (see Section 4.2.3) to denote that on reading symbol s, the machine makes a state transition and outputs string t. We now employ a slightly different notation, replacing "/" by " $\longrightarrow$ ", which is a more common notation in software design.

Convention If no transition is specified out of state s for some condition c, then the machine remains in state s if c holds.

4.5.1 Adding Structures to Transitions

As we have remarked earlier, the state in a finite state machine is an aggregate of control and data states. It is not always clear what constitutes control and what is data, but it is often possible to reduce the size of a machine (the size is the number of states) by using variables. The previous example, of recognizing a string with even number of 0s and odd number of 1s, showed a small reduction in size. We motivate the technique with a more realistic example.

An *identifier* in some programming language is defined to be a string over letters and digits whose first symbol is a letter and length is at most 6. Using L for the set of letters and LD for letters and digits, the machine shown in Figure 4.28 enters an accepting state only for a valid identifier.

The number of states in the machine is related (linearly) to the maximum length of the identifier. If an identifier is allowed to have length m, there will be $m + 2$ states in the machine, greatly obscuring its intent. In Java, identifiers may be as long as $2^{16} - 1$.

Figure 4.28: Accept valid identifiers

In Figure 4.29, we use variable n for the length of the identifier seen so far. The size of the machine is 3, independent of the length of identifiers.

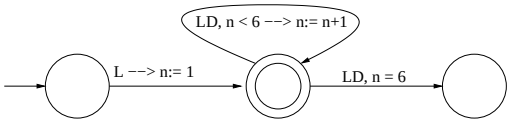


Figure 4.29: Accept valid identifiers, using variables

In this figure, we have introduced several ideas. First, there is variable n and assignment to it. Second, our transitions are more elaborate. The left side of the $\longrightarrow$ is called *guard*. The guard may have up to two parts, one for the symbol (such as LD) and the other to specify a condition (written as a predicate over the variables introduced). Thus $LD,\ n < 6 \longrightarrow n := n + 1$ has guard $LD,\ n < 6$, whose symbol part is LD and predicate part is $n < 6$. The right side of $\longrightarrow$, called the *command*, is a program that manipulates the variables. Typically, we have a few assignment statements in the command part, though statecharts [20] allow arbitrary programs. The left or the right side of the $\longrightarrow$ may be absent.

A transition takes place only if the guard is satisfied; i.e., the corresponding symbol is present in the input and the variable values satisfy the given predicate. A transition is accompanied by execution of the command part.

Introducing variables that can take on unbounded values (such as integer-valued variables) takes us out of the domain of finite state machines. We can solve problems that no finite state machine can, such as counting the number of zeroes in a string. Only the control aspect is embodied as a finite state machine.

4.5.2 Examples of Structured Transitions

4.5.2.1 Balanced Parentheses

Accept a string consisting of left and right parentheses, "(" and ")", only if it is completely balanced, as in (), (()) and ()(()). This problem can be solved using classical finite state machines only if there is a bound on the depth of nesting of the parentheses; we used depth 3 in an earlier example. The introduction of variables makes it possible to solve the general problem, though the resulting solution can not be translated to a classical finite state machine. In Figure 4.30, n is the number of unmatched "(" in the string seen so far. A string is accepted

iff $n = 0$ and the state is A. A transition with "else" guard is taken if no other guard is satisfied.

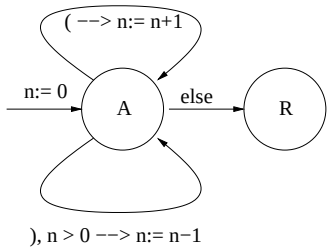


Figure 4.30: Accept balanced parentheses

A Variation Same as above but the string contains parentheses and brackets, "[" and "]", and they have to be balanced in the customary manner (as in an arithmetic expression). String ()[(())] is balanced whereas ([)] is not. Merely counting for each type of bracket is insufficient, because ([)] will then be accepted.

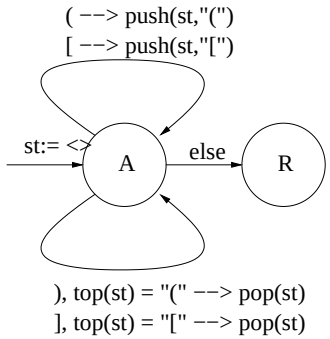


Figure 4.31: Accept balanced parentheses and brackets

In Figure 4.31, we use variable *st* which is a stack of unmatched symbols; the input string is accepted iff the state is A and *st* is empty (denoted by <> in the figure). We use *push* to push a symbol on the top of the stack, *pop* to remove the top symbol of a non- empty stack, and *top* to get the value of the top symbol.

4.5.2.2 Simple arithmetic Expression

Consider arithmetic expressions of the form $a \times b \times c + d \times e + \cdots$, where a, b, c, d, e are constants, the only operators are $+$ and $\times$ which alternate

with the constants, and there are no parentheses. Consider only non-empty expressions ended by the special symbol #. In Figure 4.32 we show a classical finite state machine that accepts such strings; here a denotes the next input constant. Observe that if the symbols + or × are seen in the initial state, the string is rejected.

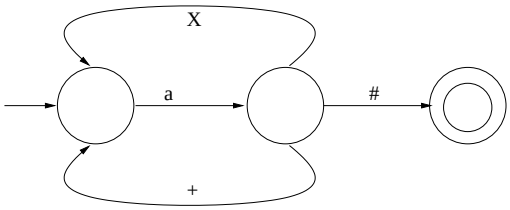


Figure 4.32: Accepts simple arithmetic expression

Next, we enhance the machine of Figure 4.32 in Figure 4.33 to compute the value of the arithmetic expression. For example, the machine will accept the string $3\times2+4\times1\times2\#$ and output 14. The machine that outputs the expression value is shown in Figure 4.33. There are two integer variables, s and p. Variable p holds the value of the current term and s the value of the expression excluding the current term. Thus, for $3\times2+4\times1\times2\#$, after we have scanned $3\times2+4\times$, values of p and s are 4 and $3\times2=6$, respectively. The transition marked with × denotes that the guard is × and the command part is empty (sometimes called *skip*).

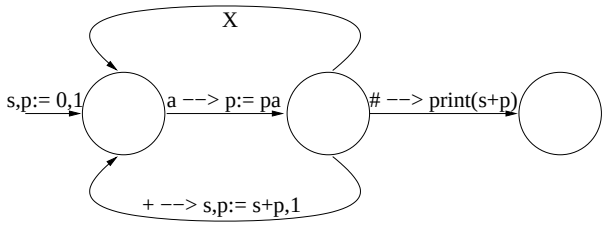


Figure 4.33: Computes value of simple arithmetic expression

A Variation We consider more general arithmetic expressions which are of the same form as described above, but also include parentheses; an example is $3+(4\times(2\times6+8)\times3)+7$. These are close to being general arithmetic expressions in typical programming languages, the only exclusions being subtraction and division operators (which could be added very simply). Our goal, as before, is to design a machine that accepts a valid expression ended with the symbol # and print the value of the expression. We will combine ideas from several machines developed so far. In particular, we will enhance the machine of Figure 4.33 to

handle parentheses, and we handle parentheses using the machine of Figure 4.30, though we have to store partially computed values in a stack, as in Figure 4.31.

The outline of the scheme is as follows. We start evaluating an expression like $3+(4\times(2\times6+8)\times3)+7$ as before, setting $s, p = 3, 1$ after scanning 3+. Then on scanning "(", we save the pair $\langle s, p\rangle$ on stack st, and start evaluating the expression within parentheses as a fresh expression, i.e., by setting $s, p := 0, 1$. Again, on encountering "(", we save $s, p = 0, 4$ on the stack, and start evaluating the inner parenthesized expression starting with $s, p := 0, 1$. On scanning ")", we know that evaluation of some parenthesized expression is complete, the value of the expression is $a = s + p$ and we should resume computation of its outer expression as if we have seen the constant a. We retrieve the top pair of values from the stack, assign them to s and p, and simulate the transition that handles a constant, i.e., we set $p := p \times a$. We store the length of the stack in n, incrementing it on encountering a "(" and decrementing it for a ")". A ")" is accepted only when $n > 0$ and "#" when $n = 0$.

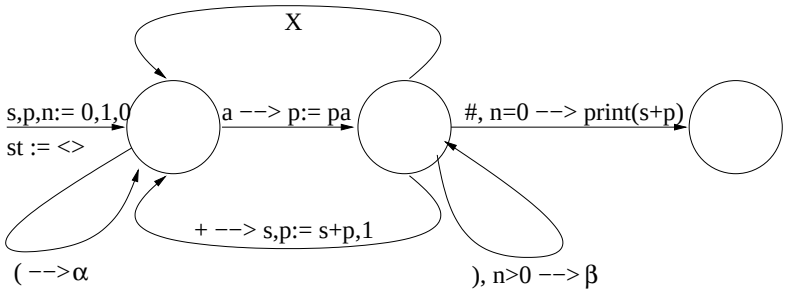


Figure 4.34: Computes value of arithmetic expression

Here, α and β are the following program fragments.

α : $push(st, \langle s, p\rangle)$; $s, p := 0, 1$; $n := n + 1$
β : $a := s + p$; $pop(st, \langle s, p\rangle)$; $p := p \times a$; $n := n - 1$

Exercise 49

Run the machine of Figure 4.34 on input string $3+(4\times(2\times6+8)\times3)+7\#$ and show the values of the variables (including contents of the stack) at each step. Choose several strings that are syntactically incorrect, and run the machine on each of them.

Exercise 50

(Research) Would it be simpler to specify the machine in Figure 4.34 if we could call a machine recursively? Explore the possibility of adding recursion to finite state machines.

4.5.2.3 Event-based Programming

Enhanced finite state machines are often used to specify and design event-based systems. An event happens in the external world and the machine has to react to the event. Consider the dome light in a car that is either "on" or "off". Initially, the light is off. The light comes on if a switch is pressed (denote the switch press by event *sw*) or if the door is opened (denote by *door*). The light goes off if the switch is pressed again (i.e., event *sw*), the door is closed (denote by *door′*) or 5 minutes elapse with the light being on (denote this time-out event by *tmout*).

The example just described denotes an elementary event-based system. The events are *sw*, *door*, *door′*, and *tmout*. The designer has no control over if or when the events happen (the time out event is special; it is guaranteed to happen at the designated time). He has to design a system that enters the correct state and carries out the appropriate actions when an event happens. The event may be likened to the occurrence of a symbol. Therefore, we treat each event as a symbol and use the diagrams as before to depict the state transitions. Figure 4.35 shows a possible specification of the dome light problem. Here, we have taken the liberty of writing compound events as boolean combinations of simpler events. Boolean connectives $\wedge$ and $\vee$ have the expected meanings, $\neg$ is problematic and should be avoided. The notions of accepting and rejecting states are irrelevant for specifications of event based systems.

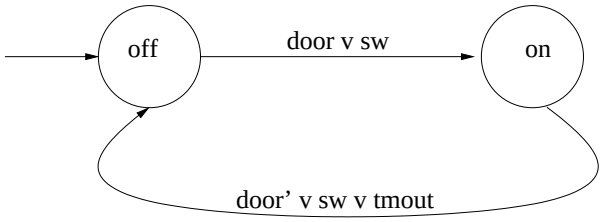


Figure 4.35: Control the dome light in a car

Our treatment of this example is inadequate. Suppose that the switch is pressed to turn on the dome light, the door is then opened and closed. We would expect the dome light to stay on. But our machine would enter the off state. Proper development of this example is left as an exercise in page 113.

4.5.3 Adding Structures to States

The classical theory of finite state machines treats a state as an indivisible unit, devoid of any structure. In this section, we show that we can often simplify a design by treating a set of states as a single state at a higher level of design, and exploiting the internal structure of this structured state at a lower level.

For motivation, consider *fractional number* as defined in Page 79. A *fractional number* is an *integer* followed by a period followed by an unsigned integer.

An acceptor of fractional numbers can be structured as in Figure 4.9 (Page 79), which may be thought of as having just 2 states: state I to accept an integer and state U to accept an unsigned integer, the transition from I to U is made on detecting a period. Next, we treat I and U as being finite state machines themselves. If the machine is in I, it is also in one of the substates of I.

4.5.3.1 Structured state

A structured state s consists of a set of states, and s is either an *or-state* or an *and-state*. If the machine is in an or-state s, it is also in *one* of the component states of s. If the machine is in an *and*-state s, it is also in *all* of the component states of s. The component states of s may themselves be structured states. A typical finite state machine can be regarded as a single or-state, because the machine is in one of the component states.

The hierarchy of states induces a tree structure on the states of the machine. The root corresponds to the whole machine, its component states are its children, and each non-leaf state is either an and-state or an or-state. We show an example of such a machine diagrammatically in Figure 4.36. Here the root state is an and-state, designated by a triangle, which consists of two component states. Each component is an or-state consisting of two component states. We have used boolean logical connectives to show the structure of non-leaf states.

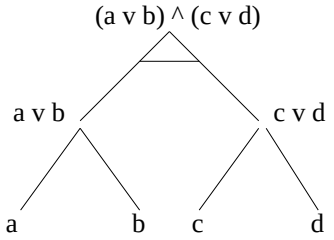


Figure 4.36: A Finite State Machine with tree structure over states

In Figure 4.36, the machine could be in states a and c simultaneously, or in b and d. In fact, we can enumerate all possible state compositions by taking the boolean formula that describes the root and writing it in disjunctive normal form. For the machine in Figure 4.36, the root is $(a \vee b) \wedge (c \vee d)$, which is same as $(a \wedge c) \vee (b \wedge c) \vee (a \wedge d) \vee (b \wedge d)$. The machine state could be given by any one of the disjuncts.

Exercise 51

Consider a machine whose root is labeled $(a \vee b) \wedge (c \wedge d) \vee h \vee (g \vee (e \wedge f))$. Draw the tree corresponding to this machine and enumerate all possible state combinations in which the machine could be at any time.

4.5.3.2 Diagrams of Structured Finite State Machines

We adopt a small number of conventions to depict structured finite state machines. An or-state is denoted by a region (circle or square) which contains all its component states. An and-state is similarly denoted, but the components are additionally separated by dotted lines; see Figure 4.37.

Figure 4.37: Diagram Conventions for Structured States

We apply these conventions to depict a machine in Figure 4.38 whose structure is given in Figure 4.36. Since the over-all machine is an and-state, there is an initial state for each component; we have chosen a and c for initial states in this example.

The transitions in Figure 4.38 are among the leaf nodes within a component. In practice, the transitions often cross state boundaries; there may be a transition from a to d, for example. The given machine starts in states a and c, makes transitions within the left component when a 0 is detected and within the right component for 1. The accepting state is (a, d). This machine accepts strings with even number of 0s and odd 1s. We have built the machine from two independent machines, one to keep track of the parity of 0s and the other for 1s. The two machines are similar and the overall machine is a composition of these two machines. Here, the treatment of each symbol, 0 and 1, is neatly delegated to a single component. Contrast this machine with the one in Figure 4.26 (page 102) for the same problem.

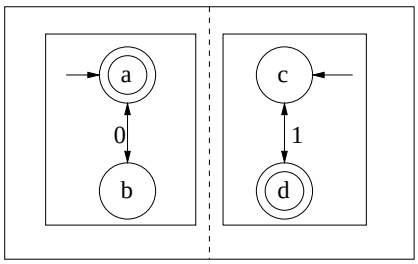


Figure 4.38: A Structured Machine, $(a \vee b) \wedge (c \vee d)$

Since a machine may be in several leaf states simultaneously, several transitions may possibly occur at any moment. The general rule for transition is as follows: on detecting a symbol, *all* transitions from the given states are under-

taken together. In Figure 4.38, each symbol occurs in exactly one component; so, there is never a simultaneous transition. We describe an example next, where the general transition rule is applied.

Consider opening a car door. This action has effects on several parts of the system. The dome light comes on, if the car is moving then a "door open" light comes on, if the headlights are on then warning chime sounds and if the key is in the ignition, a message is displayed in the dashboard. We can describe all these effects by having an and-state that includes each part —dome light, "door open" light, chime and dashboard— as a component. The event of opening a door causes simultaneous transitions in each component.

Semantics We have been very lax about specifying the behaviors of enhanced finite state machines. Unlike a classical machine where the moment of a transition ("when" the transition happens after a symbol is received) and its duration ("how long" it takes for the transition to complete) are irrelevant, enhanced machines have to take such factors into account. It will not do to say that the light comes on in response to a switch press after an arbitrary delay, if we have to consider time-out in the design.

A transition takes place instantaneously as soon its guard holds, and if several guards hold simultaneously for *transitions from a single state*, any one of these transitions may fire. Observe that transitions from different states may fire simultaneously if their guards hold. We assume that it takes no time at all to evaluate the guard or execute the command part.

There is no guarantee that a transition t will be executed at all if its guard holds, because another transition may fire and falsify the predicate in the guard of t. As an example, consider two machines that share a printer. Each machine may print if it is ready to print and the printer is free. Therefore, potentially two transitions are ready to fire at some moment. But as soon as one transition succeeds, i.e., starts printing, the printer is no longer free and the other machine's transition may not fire immediately, or ever. Semantics of concurrent behavior go beyond the scope of this course.

4.5.4 Examples of Structured States

Convention Henceforth, we omit the surrounding region for a structured state when there is no ambiguity. Figure 4.39 shows the same machine as in Figure 4.38.

4.5.4.1 Desk Fan

A desk fan has three switches: (1) a "power" switch to turn the fan off and on, (2) a "speed" switch that toggles the fan speed between levels 1 and 2, and (3) a "rotate" switch that causes the fan head to rotate or remain stationary. We describe the behavior of the fan in Figure 4.40. In Figure 4.40(a), the overall design of the fan is shown, which describes the function of the power switch. In Figure 4.40(b), the structure of "on" state is elaborated as an or-state. In

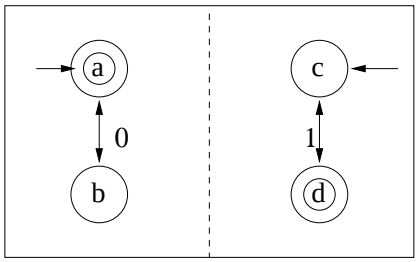


Figure 4.39: Redrawing the machine in Figure 4.38

Figure 4.40(c), a different design for "on" state is shown; it is an and-state. The design in Figure 4.40(c) is more modular; it clearly shows that speed and rotate switches control different aspects of the fan.

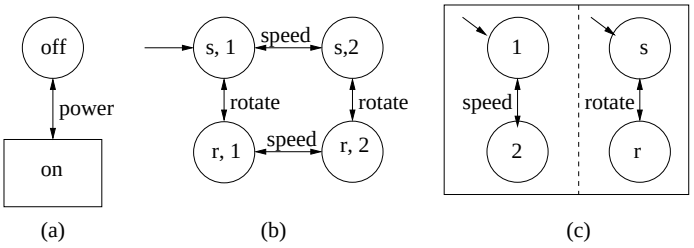


Figure 4.40: A fan with 2 speeds and rotate action

Next, we consider a design modification. Suppose we wish to allow three speed levels. The modification of Figure 4.40(b) is extensive. But, the modifications in Figure 4.40(c) can be made only in the left component, as shown in Figure 4.41; the right component is unaffected.

Next, suppose we wish to add a heater to the fan, which is controlled by a separate "heat" switch. The heater is either off or on and the heat switch toggles the state. Initially the heater is off. We simply add another component to "on" state. We show the entire fan, with 3 speed and the heater in Figure 4.42.

Preventing Certain Transitions An and-state, as shown in Figure 4.42 for example, allows all combinations of possible states of the components. Quite often, certain combinations are undesirable. For example, we my wish to prevent the fan from running at the highest speed (speed 3) while the heat is on. We accomplish this by preventing the transition from speed 2 to speed 3 if the heat is on (i.e., the state is h-on), and from h-off to h-on while the speed is at 3. For state s (leaf or non-leaf), write $in(s)$ as a predicate to denote that the machine is in state s. We use such predicates in the guards of the transitions to prevent

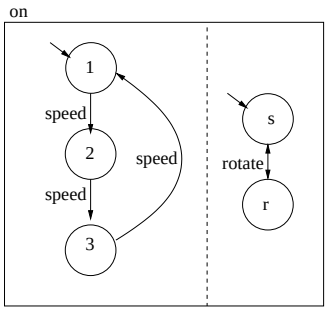


Figure 4.41: Modification of Figure 4.40(c) to allow 3 speeds in the fan

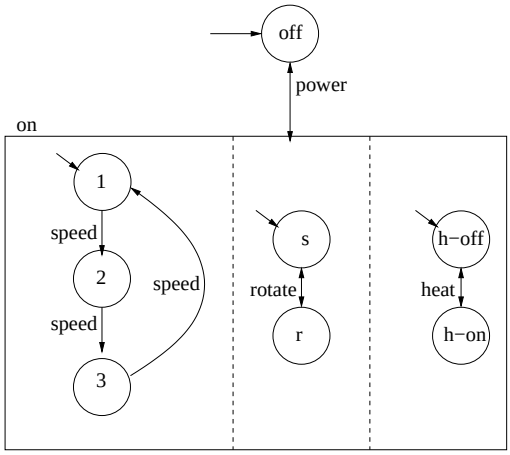


Figure 4.42: Modifications with 3 speeds and heater in the fan

certain transitions. A modified version of Figure 4.42 appears in Figure 4.43 which implements these constraints.

Observe that for and-state x whose component states are set S

$$y \in S \wedge in(y) \Rightarrow in(x) \wedge (\forall z : z \in S : in(z))$$

Similarly, for or-state x whose component states are set S

$$y \in S \wedge in(y) \Rightarrow in(x) \wedge (\forall z : z \in S \wedge z \neq y : \neg in(z))$$

Exercise 52

Reconsider the problem of the dome light from Section 4.5.2.3 in page 108. The car state is given by three components: the switch (the state is off or on), the door (shut or ajar) and the dome light's state (off or on). There are two possible events that can affect the state: (1) opening or closing the door (use event *door*

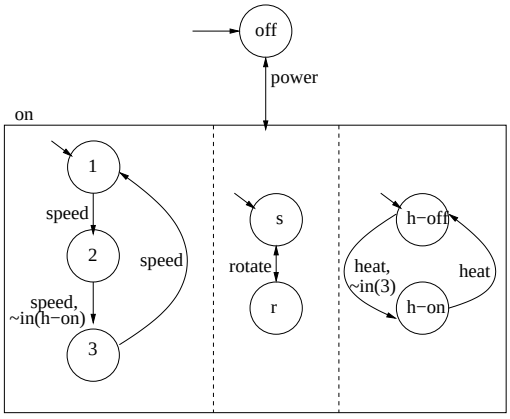


Figure 4.43: Restricting certain state transitions

to toggle between shut and ajar; that is, $door'$ is same as $door$), and (2) flipping the switch (event sw). Flipping the switch has a simple effect on the switch state, toggling it between off and on. However, the effect on the light state is more elaborate. If the door is shut, flipping the switch toggles the light state between off and on. If the door is ajar, the switch event has no effect. Similarly, if the switch is on then the light is on irrespective of the state of the door. First, describe the system using a simple finite state machine. Next, describe it using three components, for the switch, door and light.

4.5.4.2 Keeping Score in a Tennis Game

A tennis game involves two players, a *server* and a *receiver*. At any moment in a game, the score is a pair $(sp,\ rp)$, where sp is the server's score and rp the receiver's. A score, sp or rp, is one of $\{0,\ 15,\ 30,\ 40\}$ (don't ask me how they came up with these numbers). Thus, typical scores during a game may be $(0,\ 0)$, $(40,\ 15)$ and $(40,\ 40)$. Initial score is $(0,\ 0)$.

Each point in the game is won by the server or the receiver. Winning a point increases the score of the corresponding player (from 30 to 40, for example). A player wins a game if the his/her score is 40, the opponent's score is below 40 and the player wins the point. If both scores are 40, a player wins the game by winning the next two points; if the next two points are not won by a single player — the server wins a point and the receiver wins the other — the score reverts back to $(40,\ 40)$.

First, we show a simple finite state machine for score keeping. The score keeping machine at a game probably employs the same finite state machine. In Figure 4.44, we have a matrix of states, where a score $(sp,\ rp)$ is a matrix element. There are two states outside the matrix denoting a win by a different player, gs for "game to server", and, dually, gr. Each row of the matrix corresponds to a fixed score by the server and the column for the receiver. A

point won by the receiver causes a transition to the right neighbor along a row (provided there is a right neighbor); similarly a point won by the server causes transition downward in a column (provided there is a neighbor below). For the rightmost column, a receiver's point causes him/her to win the game provided the server's score is below 40; similarly, for the server.

The remaining question is to model the behavior at (40, 40). Here, we do a simple analysis to conclude that a winning point by the receiver merely decreases the score of the server, so that the score becomes (30, 40); dually, if the server wins the point the score becomes (40, 30).

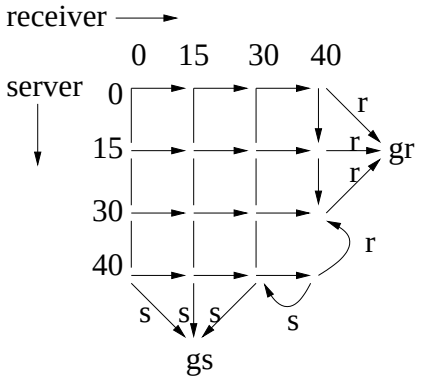


Figure 4.44: Scoring in a tennis game. s/r are points won by server/receiver.

Representation with Structured Transitions We can represent the machine more succinctly by using structured transitions. The transitions in Figure 4.45 are labeled with the statement numbers from the following program fragment.

We employ two variables, sp and rp for the scores of the server and receiver, respectively. Below, sp' is the next value higher than sp (i.e., if $sp = 30$ then $sp' = 40$); similarly, rp'. Variable s represents a point won by the server, and, similarly, r is a point won by the receiver. Thus, statement 0 represents the transition to the initial state where both player scores are 0, and 3 represents the transition from a deuce state where the server wins the point.

0: $sp, rp := 0, 0$
1: $s, sp < 40 \ \longrightarrow \ sp := sp'$
2: $r, rp < 40 \ \longrightarrow \ rp := rp'$
3: $s, sp = 40 \wedge rp = 40 \ \longrightarrow \ rp := 30$
4: $r, sp = 40 \wedge rp = 40 \ \longrightarrow \ sp := 30$
5: $s, sp = 40 \wedge rp < 40 \ \longrightarrow \ gs := true$
6: $r, sp < 40 \wedge rp = 40 \ \longrightarrow \ gr := true$

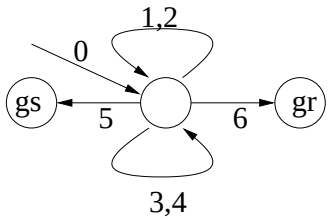


Figure 4.45: Scoring in a tennis game, using structured transitions

Representation with Structured States The state of the machine is a tuple; therefore, we can decompose the state into two components, and treat changes to each component by a separate machine. Such a design is shown in Figure 4.46. Variables *rp* and *sp* refer to the states of the receiver and server. Observe the transitions when the score is (40, 40).

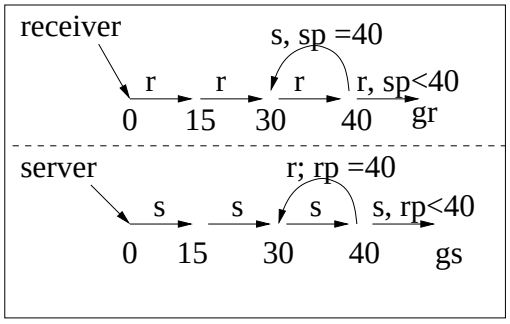


Figure 4.46: Scoring in a tennis game, using structured states

Representation with Structured Transitions and States We combine the ideas of Figures 4.45 and 4.46 to get Figure 4.47.

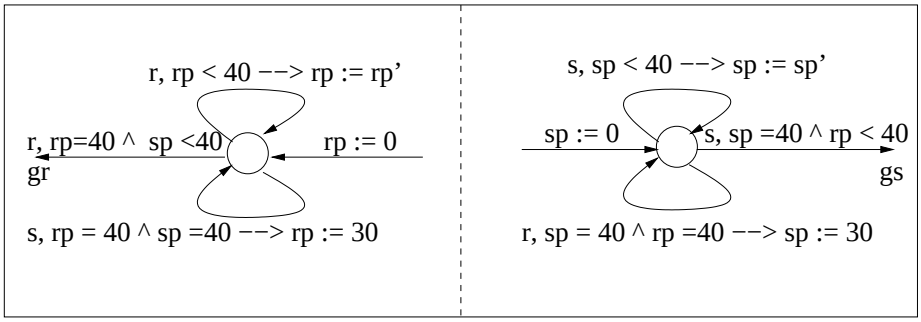


Figure 4.47: Scoring in a tennis game, using structured transitions and states