成 果 报 奖

附件材料

Introduction to Automata Theory, Languages, and Computation

Solutions for Chapter 8

Solutions for Section 8.1

Exercise 8.1.1(a)

We need to take a program *P* and modify it so it:

1. Never halts unless we explicitly want it to, and
2. Halts whenever it prints `hello, world.`

For (1), we can add a loop such as `while(1){x=x;}` to the end of `main`, and also at any point where `main` returns. That change catches the normal ways a program can halt, although it doesn't address the problem of a program that halts because some exception such as division by 0 or an attempt to read an unavailable device. Technically, we'd have to replace all of the exception handlers in the run-time environment to cause a loop whenever an exception occurred.

For (2), we modify *P* to record in an array the first 12 characters printed. If we find that they are `hello, world.`, we halt by going to the end of `main` (past the point where the while-loop has been installed).

Solutions for Section 8.2

Exercise 8.2.1(a)

To make the ID's clearer in HTML, we'll use *[q0]* for *q_0*, and similarly for the other states.

[q0]00 |- X[q1]0 |- X0[q1]

The TM halts at the above ID.

Exercise 8.2.2(a)

Here is the transition table for the TM:

state	0	1	B	X	Y

q0	(q2,X,R)	(q1,X,R)	(qf,B,R)	-	(q0,Y,R)
q1	(q3,Y,L)	(q1,1,R)	-	-	(q1,Y,R)
q2	(q2,0,R)	(q3,Y,L)	-	-	(q2,Y,R)
q3	(q3,0,L)	(q3,1,L)	-	(q0,X,R)	(q3,Y,L)
qf	-	-	-	-	-

In explanation, the TM makes repeated excursions back and forth along the tape. The symbols *X* and *Y* are used to replace 0's and 1's that have been cancelled one against another. The difference is that an *X* guarantees that there are no unmatched 0's and 1's to its left (so the head never moves left of an *X*), while a *Y* may have 0's or 1's to its left.

Initially in state *q0*, the TM picks up a 0 or 1, remembering it in its state (*q1* = found a 1; *q2* = found a 0), and cancels what it found with an *X*. As an exception, if the TM sees the blank in state *q0*, then all 0's and 1's have matched, so the input is accepted by going to state *qf*.

In state *q1*, the TM moves right, looking for a 0. If it finds it, the 0 is replaced by *Y*, and the TM enters state *q3* to move left an look for an *X*. Similarly, state *q2* looks for a 1 to match against a 0.

In state *q3*, the TM moves left until it finds the rightmost *X*. At that point, it enters state *q0* again, moving right over *Y*'s until it finds a 0, 1, or blank, and the cycle begins again.

Exercise 8.2.4

These constructions, while they can be carried out using the basic model of a TM are much clearer if we use some of the tricks of Sect. 8.3.

For part (a), given an input *[x,y]* use a second track to simulate the TM for *f* on the input *x*. When the TM halts, compare what it has written with *y*, to see if *y* is indeed *f(x)*. Accept if so.

For part (b), given *x* on the tape, we need to simulate the TM *M* that recognizes the graph of *f*. However, since this TM may not halt on some inputs, we cannot simply try all *[x,i}* to see which value of *i* leads to acceptance by *M*. The reason is that, should we work on some value of *i* for which *M* does not halt, we'll never advance to the correct value of *f(x)*. Rather, we consider, for various combinations of *i* and *j*, whether *M* accepts *[x,i]* in *j* steps. If we consider *(i,j)* pairs in order of their sum (i.e., (0,1), (1,0), (0,2), (1,1), (2,0), (0,3),...) then eventually we shall simulate *M* on *[x,f(x)]* for a sufficient number of steps that *M* reaches acceptance. We need only wait until we consider pairs whose sum is *f(x)* plus however many steps it takes *M* to accept *[x,f(x)]*. In this manner, we can discover what *f(x)* is, write it on the tape of the TM that we have designed to compute *f(x)*, and halt.

Now let us consider what happens if *f* is not defined for some arguments. Part (b) does not change, although the constructed TM will fail to discover *f(x)* and thus will continue searching forever. For part (a), if we are given *[x,y]*, and *f* is not defined on *x*, then the TM for *f* will never halt on *x*. However, there is nothing wrong with that. Since *f(x)* is undefined, surely *y* is not *f(x)*. Thus, we do not want the TM for the graph of *f* to accept *[x,y]* anyway.

Exercise 8.2.5(a)

This TM only moves right on its input. Moreover, it can only move right if it sees alternating 010101... on the input tape. Further, it alternates between states *q_0* and *q_1* and only accepts if it sees a blank in state *q_1*. That in turn occurs if it has just seen 0 and moved right, so the input must end in a 0. That is, the language is that of regular expression **(01)*0**.

Solutions for Section 8.3

Exercise 8.3.3

Here is the subroutine. Note that because of the technical requirements of the subroutine, and the fact that a TM is not allowed to keep its head stationary, when we see a non-0, we must enter state *q3*, move right, and then come back left in state *q4*, which is the ending state for the subroutine.

state	0	1	B
q1	(q2,0,R)	-	-
q2	(q2,0,R)	(q3,1,R)	(q3,B,R)
q3	(q4,0,L)	(q4,1,L)	(q4,B,L)

Now, we can use this subroutine in a TM that starts in state *q0*. If this TM ever sees the blank, it accepts in state *qf*. However, whenever it is in state *q0*, it knows only that it has not seen a 1 immediately to its right. If it is scanning a 0, it must check (in state *q5*) that it does not have a blank immediately to its right; if it does, it accepts. If it sees 0 in state *q5*, it comes back to the previous 0 and calls the subroutine to skip to the next non-0. If it sees 1 in state *q5*, then it has seen 01, and uses state *q6* to check that it doesn't have another 1 to the right.

In addition, the TM in state *q4* (the final state of the subroutine), accepts if it has reached a blank, and if it has reached a 1 enters state *q6* to make sure there is a 0 or blank following. Note that states *q4* and *q5* are really the same, except that in *q4* we are certain we are not scanning a 0. They could be combined into one state. Notice also that the subroutine is not a perfect match for what is needed, and there is some unnecessary jumping back and forth on the tape. Here is the remainder of the transition table.

state	0	1	B
q0	(q5,0,R)	(q5,1,R)	(qf,B,R)
q5	(q1,0,L)	(q6,1,R)	(qf,B,R)
q6	(q0,0,R)	-	(qf,B,R)
q4	-	(q6,1,R)	(qf,B,R)

Solutions for Section 8.4

Exercise 8.4.2(a)

For clarity, we put the state in square brackets below. Notice that in this example, we never branch. *[q_0]01 |- 1[q_0]1 |- 10[q_1] |- 10B[q_2]*

Exercise 8.4.3(a)

We'll use a second tape, on which the guess *x* is stored. Scan the input from left yo right, and at each cell, guess whether to stay in the initial state (which does the scanning) or go to a new state that copies the next 100 symbols onto the second tape. The copying is done by a sequence of 100 state, so exactly 100 symbols can be placed on tape 2.

Once the copying is done, retract the head of tape 2 to the left end of the 100 symbols. Then, continue moving right on tape 1, and at each cell guess either to continue moving right or to guess that the second copy of *x* begins. In the latter case, compare the next 100 symbols on tape 1 with the 100 symbols on tape 2. If they all match, then move right on tape 1 and accept as soon as a blank is seen.

Exercise 8.4.5

For part (a), guess whether to move left or right, entering one of two different states, each responsible for moving in one direction. Each of these states proceeds in its direction, left or right, and if it sees a $ it enters state *p*. Technically, the head has to move off the $, entering another state, and then move back to the $, entering state *p* as it does so.

Part (b), doing the same thing deterministically, is trickier, since we might start off in the wrong direction and travel forever, never seeing the $. Thus, we have to oscillate, using left and right endmarkers *X* and *Y*, respectively, to mark how far we have traveled on a second track. Start moving one cell left and leave the *X*. Then, move two cells right and leave the *Y*. Repeatedly move left to the *X*, move the *X* one more cell left, go right to the *Y*, move it once cell right, and repeat.

Eventually, we shall see the $. At this time, we can move left or right to the other endmarker, erase it, move back to the end where the $ was found, erase the other endmarker, and wind up at the $.

Exercise 8.4.8(a)

There would be 10 tracks. Five of the tracks hold one of the symbols from the tape alphabet, so there are 7^5 ways to select these tracks. The other five tracks hold either *X* or blank, so these tracks can be selected in 2^5 ways. The total number of symbols is thus 7^5 * 2^5 = 537,824.

Exercise 8.4.8(b)

The number of symbols is the same. The five tracks with tape symbols can still be chosen in 5^7 ways. The sixth track has to tell which subset of the five tapes have their head at that position. There are 2^5 possible subsets, and therefore 32 symbols are needed for the 6th track. Again the number of symbols is 7^5 * 2^5.

Solutions for Section 8.5

Exercise 8.5.1(c)

In principle, any language that is recursively enumerable can be recognized by a 2-counter machine, but how do we design a comprehensible answer for a particular case? As the *a*'s are read, count them with both counters. Then, when *b*'s enter, compare them with one counter, and accept if they are the same. Continue accepting as long as *c*'s enter. If the numbers of *a*'s and *b*'s differ, then compare the second counter with the number of *c*'s, and accept if they match.

Introduction to Automata Theory, Languages, and Computation

Solutions for Chapter 6

Solutions for Section 6.1

Exercise 6.1.1(a)

In what follows, *e* stands for epsilon, the empty string, and *Z* stands for the initial symbol, *Z_0*.

```
(q,01,Z) |- (q,1,XZ) |- (q,e,XZ) |- (p,e,Z)
                     |- (p,1,Z)  |- (p,e,e)
```

Solutions for Section 6.2

Exercise 6.2.1(a)

We shall accept by empty stack. Symbol *X* will be used to count the 0's on the input. In state *q*, the start state, where we have seen no 1's, we add an *X* to the stack for each 0 seen. The first *X* replaces *Z_0*, the start symbol. When we see a 1, we go to state *p*, and then only pop the stack, one *X* for each input 1. Formally, the PDA is *({q,p},{0,1},{X,Z_0},delta,q,Z_0)*. The rules:

1. *delta(q,0,Z_0) = {(q,X)}*
2. *delta(q,0,X) = {(q,XX)}*
3. *delta(q,1,X) = {(p,epsilon)}*
4. *delta(p,1,X) = {(p,epsilon)}*

Exercise 6.2.2(a)

Revised 6/20/02.

Begin in start state *q0*, with start symbol *Z_0*, and immediately guess whether to check for:

1. *i=j=0* (state *q1*).
2. *i=j>0* (state *q2*).
3. *j=k* (state *q3*).

We shall accept by final state; as seen below, the accepting states are *q1* and *q3*. The rules, and their explanations (again, *e* stands for epsilon):

- *delta(q0,e,Z_0) = {(q1,Z_0), (q2,Z_0), (q3,Z_0)}*, the initial guess.
- *delta(q1,c,Z_0) = {(q1,Z_0)}*. In case (1), we assume there are no *a*'s or *b*'s, and we consume all *c*'s. State *q1* will be one of our accepting states.
- *delta(q2,a,Z_0) = {(q2,XZ_0)}*, and *delta(q2,a,X) = {(q2,XX)}*. These rules begin case (2). We use *X* to count the number of *a*'s read from the input, staying in state *q2*.
- *delta(q2,b,X) = delta(q4,b,X) = {(q4,e)}*. When *b*'s are seen, we go to state *q4* and pop *X*'s against the *b*'s.
- *delta(q4,e,Z_0) = {(q1,Z_0)}*. If we reach the bottom-of-stack marker in state *q4*, we have seen an equal number of *a*'s and *b*'s. We go spontaneously to state *q1*, which will accept and consume all *c*'s, while continuing to accept.
- *delta(q3,a,Z_0) = {(q3,Z_0)}*. This rule begins case (3). We consume all *a*'s from the input. Since *j=k=0* is possible, state *q3* must be an accepting state.
- *delta(q3,b,Z_0) = {(q5,XZ_0)}*. When *b*'s arrive, we start counting them and go to state *q5*, which is not an accepting state.
- *delta(q5,b,X) = {(q5,XX)}*. We continue counting *b*'s.
- *delta(q5,c,X) = delta(q6,c,X) = {(q6,e)}*. When *c*'s arrive, we go to state *q6* and match the *c*'s against the *b*'s.
- *delta(q6,e,Z_0) = {(q3,e)}*. When the bottom-of-stack marker is exposed in state *q6*, we have seen an equal number of *b*'s and *c*'s. We spontaneously accept in state *q3*, but we pop the stack so we cannot accept after reading more *a*'s.

Exercise 6.2.4

Introduce a new state *q*, which becomes the initial state. On input epsilon and the start symbol of *P*, the new PDA has a choice of popping the stack (thus accepting epsilon), or going to the start state of *P*.

Exercise 6.2.5(a)

We again use *e* to represent epsilon.

(q0,bab,Z_0) |- (q2,ab,BZ_0) |- (q3,b,Z_0) |- (q1,b,AZ_0) |- (q1,e,Z_0) |- (q0,e,Z_0) |- (f,e,e)

Exercise 6.2.8

Suppose that there is a rule that *(p,X1X2...Xk)* is a choice in *delta(q,a,Z)*. We create *k-2* new states *r1,r2,...,r{k-2}* that simulate this rule but do so by adding one symbol at a time to the stack. That is, replace *(p,X1X2...Xk)* in the rule by *(r{k-2},X{k-1}Xk*. Then create new rules *delta(r{k-2},e,X{k-1}) = {(r{k-1},X{k-2}X{k-1})}*, and so on, down to *delta(r2,e,X3) = {(r1,X2X3)}* and *delta(r1,X2) = {(p,X1X2)}*.

Solutions for Section 6.3

Exercise 6.3.1

({q},{0,1),{0,1,A,S},delta,q,S) where *delta* is defined by:

1. *delta(q,e,S) = {(q,0S1), (q,A)}*
2. *delta(q,e,A) = {(q,1A0), (q,S), (q,e)}*
3. *delta(q,0,0) = {(q,e)}*
4. *delta(q,1,1) = {(q,e)}*

In the above, *e* represents the empty string.

Exercise 6.3.3

In the following, *S* is the start symbol, *e* stands for the empty string, and *Z* is used in place of *Z_0*.

1. *S -> [qZq] | [qZp]*

   The following four productions come from rule (1).

2. *[qZq] -> 1[qXq][qZq]*
3. *[qZq] -> 1[qXp][pZq]*
4. *[qZp] -> 1[qXq][qZp]*
5. *[qZp] -> 1[qXp][pZp]*

   The following four productions come from rule (2).

6. *[qXq] -> 1[qXq][qXq]*
7. *[qXq] -> 1[qXp][pXq]*
8. *[qXp] -> 1[qXq][qXp]*
9. *[qXp] -> 1[qXp][pXp]*

   The following two productions come from rule (3).

10. *[qXq] -> 0[pXq]*
11. *[qXp] -> 0[pXp]*

    The following production comes from rule (4).

12. *[qXq] -> e*

    The following production comes from rule (5).

13. *[pXp] -> 1*

    The following two productions come from rule (6).

14. *[pZq] -> 0[qZq]*
15. *[pZp] -> 0[qZp]*

Exercise 6.3.6

Convert *P* to a CFG, and then convert the CFG to a PDA, using the two constructions given in Section 6.3. The result is a one-state PDA equivalent to *P*.

Solutions for Section 6.4

Exercise 6.4.1(b)

Not a DPDA. For example, rules (3) and (4) give a choice, when in state *q*, with 1 as the next input symbol, and with *X* on top of the stack, of either using the 1 (making no other change) or making a move on epsilon input that pops the stack and going to state *p*.

Exercise 6.4.3(a)

Suppose a DPDA *P* accepts both *w* and *wx* by empty stack, where *x* is not epsilon (i.e., *N(P)* does not have the prefix property). Then *(q_0,wxZ_0) |-* (q,x,epsilon)* for some state *q*, where *q_0* and *Z_0* are the start state and symbol of *P*. It is not possible that *(q,x,epsilon) |-* (p,epsilon,epsilon)* for some state *p*, because we know *x* is not epsilon, and a PDA cannot have a move with an empty stack. This observation contradicts the assumption that *wx* is in *N(P)*.

Exercise 6.4.3(c)

Modify *P′* in the following ways to create DPDA *P*:

1. Add a new start state and a new start symbol. *P*, with this state and symbol, pushes the start symbol of *P′* on top of the stack and goes to the start state of *P′*. The purpose of the new start symbol is to make sure *P* doesn't accidentally accept by empty stack.
2. Add a new ``popping state" to *P*. In this state, *P* pops every symbol it sees on the stack, using epsilon input.
3. If *P′* enters an accepting state, *P* enters the popping state instead.

As long as *L(P′)* has the prefix property, then any string that *P′* accepts by final state, *P* will accept by empty stack.

Introduction to Automata Theory, Languages, and Computation

Solutions for Chapter 2

Revised 9/6/01.

Solutions for Section 2.2

Exercise 2.2.1(a)

States correspond to the eight combinations of switch positions, and also must indicate whether the previous roll came out at *D*, i.e., whether the previous input was accepted. Let 0 represent a position to the left (as in the diagram) and 1 a position to the right. Each state can be represented by a sequence of three 0's or 1's, representing the directions of the three switches, in order from left to right. We follow these three bits by either *a* indicating it is an accepting state or *r*, indicating rejection. Of the 16 possible states, it turns out that only 13 are accessible from the initial state, 000r. Here is the transition table:

	A	B
->000r	100r	011r
*000a	100r	011r
*001a	101r	000a
010r	110r	001a
*010a	110r	001a
011r	111r	010a
100r	010r	111r
*100a	010r	111r
101r	011r	100a
*101a	011r	100a
110r	000a	101a
*110a	000a	101a
111r	001a	110a

Exercise 2.2.2

In what follows, we use *dhat* for ``delta-hat." The statement to be proved is *dhat(q,xy) = dhat(dhat(q,x),y)*, and we proceed by induction on the length of *y*.

Basis: If *y = epsilon*, then the statement is *dhat(q,x) = dhat(dhat(q,x),epsilon)*. This statement follows from the basis in the definition of *dhat*. Note that in applying this definition, we must treat *dhat(q,x)* as if it were just a state, say *p*. Then, the statement to be proved is *p = dhat(p,epsilon)*, which is easy to recognize as the basis in the definition of *dhat*.

Induction: Assume the statement for strings shorter than *y*, and break *y = za*, where *a* is the last symbol of *y*. The steps converting *dhat(dhat(q,x),y)* to *dhat(q,xy)* are summarized in the following table:

Expression	Reason
dhat(dhat(q,x),y)	Start
dhat(dhat(q,x),za)	*y=za* by assumption
delta(dhat(dhat(q,x),z),a)	Definition of *dhat*, treating *dhat(q,x)* as a state
delta(dhat(q,xz),a)	Inductive hypothesis
dhat(q,xza)	Definition of *dhat*
dhat(q,xy)	*y=za*

Exercise 2.2.4(a)

The intuitive meanings of states *A, B*, and *C* are that the string seen so far ends in 0, 1, or at least 2 zeros.

	0	**1**
->A	B	A
B	C	A
*C	C	A

Exercise 2.2.6(a)

The trick is to realize that reading another bit either multiplies the number seen so far by 2 (if it is a 0), or multiplies by 2 and then adds 1 (if it is a 1). We don't need to remember the entire number seen --- just its remainder when divided by 5. That is, if we have any number of the form *5a+b*, where *b* is the remainder, between 0 and 4, then *2(5a+b) = 10a+2b*. Since 10*a* is surely divisible by 5, the remainder of *10a+2b* is the same as the remainder of 2b when divided by 5. Since *b*, is 0, 1, 2, 3, or 4, we can tabulate the answers easily. The same idea holds if we want to consider what happens to *5a+b* if we multiply by 2 and add 1.

The table below shows this automaton. State *qi* means that the input seen so far has remainder *i* when divided by 5.

	0	**1**
->*q0	q0	q1
q1	q2	q3
q2	q4	q0
q3	q1	q2
q4	q3	q4

There is a small matter, however, that this automaton accepts strings with leading 0's. Since the problem calls for accepting only those strings that begin with 1, we need an additional state *s*, the start state, and an additional ``dead state'' *d*. If, in state *s*, we see a 1 first, we act like *q0*; i.e., we go to state *q1*. However, if the first input is 0, we should never accept, so we go to state *d*, which we never leave. The complete automaton is:

	0	**1**
->s	d	q1
*q0	q0	q1
q1	q2	q3
q2	q4	q0
q3	q1	q2
q4	q3	q4
d	d	d

Exercise 2.2.9

Part (a) is an easy induction on the length of *w*, starting at length 1.

Basis: |*w*| = 1. Then *dhat(q_0,w) = dhat(q_f,w)*, because *w* is a single symbol, and *dhat* agrees with *delta* on single symbols.

Induction: Let *w = za*, so the inductive hypothesis applies to *z*. Then *dhat(q_0,w) = dhat(q_0,za) = delta(dhat(q_0,z),a) = delta(dhat(q_f,z),a)* [by the inductive hypothesis] = *dhat(q_f,za) = dhat(q_f,w)*.

For part (b), we know that *dhat(q_0,x) = q_f*. Since *xepsilon, we know by part (a) that dhat(q_f,x) = q_f. It is then a simple induction on k to show that dhat(q_0,x^k) = q_f.*

Basis: For k=1 the statement is given.

Induction: Assume the statement for k-1; i.e., dhat(q_0,x^{k-1}) = q_f. Using Exercise 2.2.2, dhat(q_0,x^k) = dhat(dhat(q_0,x^{k-1}),x) = dhat(q_f,x) [by the inductive hypothesis] = q_f [by (a)].

Exercise 2.2.10

The automaton tells whether the number of 1's seen is even (state A) or odd (state B), accepting in the latter case. It is an easy induction on |w| to show that dh(A,w) = A if and only if w has an even number of 1's.

Basis: |w| = 0. Then w, the empty string surely has an even number of 1's, namely zero 1's, and dhat(A,w) = A.

Induction: Assume the statement for strings shorter than w. Then w = za, where a is either 0 or 1.

Case 1: a = 0. If w has an even number of 1's, so does z. By the inductive hypothesis, dhat(A,z) = A. The transitions of the DFA tell us dhat(A,w) = A. If w has an odd number of 1's, then so does z. By the inductive hypothesis, dhat(A,z) = B, and the transitions of the DFA tell us dhat(A,w) = B. Thus, in this case, dhat(A,w) = A if and only if w has an even number of 1's.

Case 2: a = 1. If w has an even number of 1's, then z has an odd number of 1's. By the inductive hypothesis, dhat(A,z) = B. The transitions of the DFA tell us dhat(A,w) = A. If w has an odd number of 1's, then z has an even number of 1's. By the inductive hypothesis, dhat(A,z) = A, and the transitions of the DFA tell us dhat(A,w) = B. Thus, in this case as well, dhat(A,w) = A if and only if w has an even number of 1's.

Solutions for Section 2.3

Exercise 2.3.1

Here are the sets of NFA states represented by each of the DFA states A through H: A = {p}; B = {p,q}; C = {p,r}; D = {p,q,r}; E = {p,q,s}; F = {p,q,r,s}; G = {p,r,s}; H = {p,s}.

	0	**1**
->A	B	A
B	D	C
C	E	A
D	F	C
*E	F	G
*F	F	G
*G	E	H

*H	E	H

Exercise 2.3.4(a)

The idea is to use a state qi, for i = 0,1,...,9 to represent the idea that we have seen an input i and guessed that this is the repeated digit at the end. We also have state qs, the initial state, and qf, the final state. We stay in state qs all the time; it represents no guess having been made. The transition table:

	0	**1**	**...**	**9**
->qs	{qs,q0}	{qs,q1}	...	{qs,q9}
q0	{qf}	{q0}	...	{q0}
q1	{q1}	{qf}	...	{q1}
...	...	...	...	...
q9	{q9}	{q9}	...	{qf}
*qf	{}	{}	...	{}

Solutions for Section 2.4

Exercise 2.4.1(a)

We'll use q0 as the start state. q1, q2, and q3 will recognize abc; q4, q5, and q6 will recognize abd, and q7 through q10 will recognize aacd. The transition table is:

	a	**b**	**c**	**d**
->q0	{q0,q1,q4,q7}	{q0}	{q0}	{q0}
q1	{}	{q2}	{}	{}
q2	{}	{}	{q3}	{}
*q3	{}	{}	{}	{}
q4	{}	{q5}	{}	{}
q5	{}	{}	{}	{q6}
*q6	{}	{}	{}	{}
q7	{q8}	{}	{}	{}
q8	{}	{}	{q9}	{}
q9	{}	{}	{}	{q10}
*q10	{}	{}	{}	{}

Exercise 2.4.2(a)

The subset construction gives us the following states, each representing the subset of the NFA states indicated: A = {q0}; B = {q0,q1,q4,q7}; C = {q0,q1,q4,q7,q8}; D = {q0,q2,q5}; E = {q0,q9}; F = {q0,q3}; G = {q0,q6}; H = {q0,q10}. Note that F, G and H can be combined into one accepting state, or we can use these three state to signal the recognition of abc, abd, and aacd, respectively.

	a	**b**	**c**	**d**
->A	B	A	A	A
B	C	D	A	A
C	C	D	E	A
D	B	A	F	G
E	B	A	A	H
*F	B	A	A	A
*G	B	A	A	A
*H	B	A	A	A

Solutions for Section 2.5

Exercise 2.5.1

For part (a): the closure of p is just {p}; for q it is {p,q}, and for r it is {p,q,r}.

For (b), begin by noticing that a always leaves the state unchanged. Thus, we can think of the effect of strings of b's and c's only. To begin, notice that the only ways to get from p to r for the first time, using only b, c, and epsilon-transitions are bb, bc, and c. After getting to r, we can return to r reading either b or c. Thus, every string of length 3 or less, consisting of b's and c's only, is accepted, with the exception of the string b. However, we have to allow a's as well. When we try to insert a's in these strings, yet keeping the length to 3 or less, we find that every string of a's b's, and c's with at most one a is accepted. Also, the strings consisting of one c and up to 2 a's are accepted; other strings are rejected.

There are three DFA states accessible from the initial state, which is the epsilon closure of p, or {p}. Let A = {p}, B = {p,q}, and C = {p,q,r}. Then the transition table is:

	a	**b**	**c**
->A	A	B	C
B	B	C	C
*C	C	C	C