7. FSAs AND REGULAR LANGUAGES

§ 7.1. Regular Languages

We've seen how Finite State Acceptors can be created to accept certain regular languages (that is, languages described by regular expressions). We'll now prove that this can always be done. The converse, that every language accepted by a FSA is regular, is also true, though you'll have to look elsewhere for a proof.

Theorem 1: A language is regular if and only if it can be accepted by an FSA.

Outline of the Proof: Suppose that the language L is regular. Consider a regular expression for L. This expression can be built up from the languages 0, 1, λ and $\varnothing$ by the three operations of product (concatenation), sum and Kleene star. We'll prove in the following sections that:

(1) the languages 0, 1, λ and $\varnothing$ are accepted by FSAs;

(2) if L_1 and L_2 are languages accepted by FSAs M_1 and M_2 respectively then we can construct an FSA, which we shall call M_1M_2, that accepts the language L_1L_2;

(3) if L_1 and L_2 are languages accepted by FSAs M_1 and M_2 respectively then we can construct an FSA, which we shall call $M_1 + M_2$, that accepts the language $L_1 + L_2$;

(4) if L is a language accepted by an FSA M then we can construct an FSA, which we shall call M*, that accepts the language L*.

Once these have been demonstrated it follows that every regular language is accepted by an FSA. As explained before, we omit the proof of the converse.

§ 7.2. The Languages 0, 1, λ and ∅

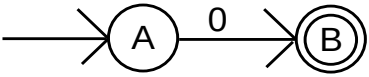
FSAs that accept the four primitive regular languages are:

∅:

Any FSA with no accepting states will accept the empty set. This one is the simplest one. It consists of just a single black hole.

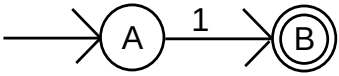
λ:

This FSA will accept the null string but as soon as any character is read the machine moves to a black-hole. (Remember the convention that if an arrow is not shown it is assumed to be going to a black hole.)

0: 

Here the string is accepted only if it consists of a single 0.

Similarly an FSA for the language 1, consisting only of the single character string 1, is:

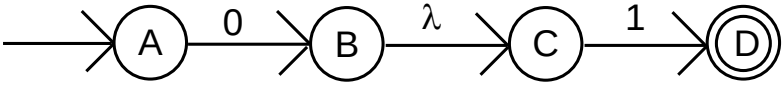
1: 

§ 7.3. Product of Two Languages

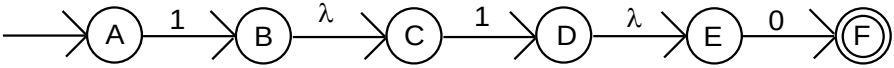
Suppose that L_1 and L_2 are languages accepted by FSAs M_1 and M_2 respectively. We'll describe the construction of an FSA, called M_1M_2, that accepts the product language L_1L_2.

STATES: The states of M_1M_2 are precisely those of M_1 and M_2 combined.
ARROWS: In addition to the arrows of the separate machines we construct additional arrows to connect the two machines. For every accepting state in M_1 we construct a null arrow to the initial state of M_2.
INITIAL STATES: The initial states of M_1M_2 are those of M_1.
ACCEPTING STATES: The accepting states of M_1M_2 are those of M_2.

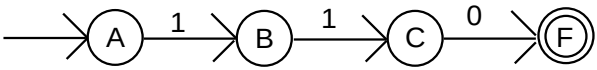
Example 1:
An FSA that accepts the regular language 01 is:



Example 2:
An FSA that accepts the regular language 110 is:

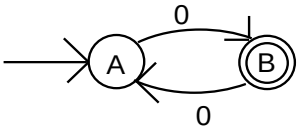


It's easy to see that these null transitions can be removed. So to accept the language 110 we need only 4 states:

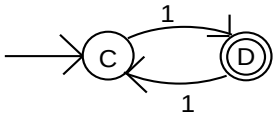


You may wonder why we needed the null transitions. Certainly in such simple cases they're not needed. But there are cases where they're needed to prevent 'reflux'. This is a medical term which refers to what happens when some of the contents of the stomach travel back up the oesophagus. Something similar happens when you wrongly omit null transitions in certain situations, allowing the flow of the machine to return to an earlier part of its anatomy. Have a look at the following example to see what this means.

Example 3:
Suppose M_1 and M_2 are the FSAs:

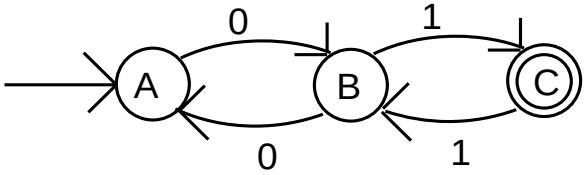


M_1



M_2

Then M_1 accepts the language $L_1 = (00)^*0$ and
M_2 accepts the language $L_2 = (11)^*1$.
If we were to plug the two machines into each other we'd get:

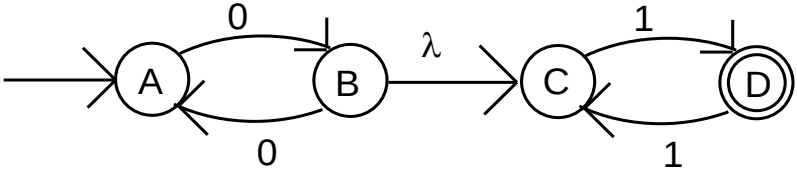


But this accepts more than it should. It accepts the string 011001 which is not in the language

$$L_1L_2 = (00)^*0(11)^*1.$$

The problem is that this machine can return to the M_1 part after it has left it, while what is supposed to happen is that having left M_1 we must never return. The null string is used as a one-way valve that allows the machine to move out of M_1 into M_2 while preventing it from returning.

The correct machine to accept L_1L_2 in this case is:



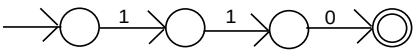
We can remove this null transition, but not simply by overlapping states B and C.

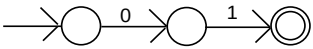
§7.4. Sum of Two Languages

Suppose that L_1 and L_2 are languages accepted by FSAs M_1 and M_2 respectively. We'll describe the construction of an FSA, which we'll call $M_1 + M_2$, that accepts the sum of these languages $L_1 + L_2$. Remember that $M_1 + M_2$ will have to accept any string that is accepted by either machine, and no other.

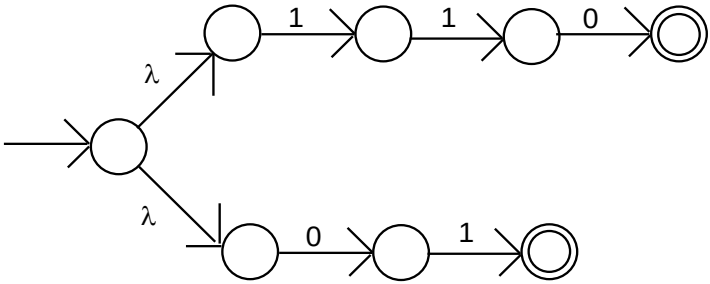
STATES: The states of $M_1 + M_2$ are those of the separate machines.
ARROWS: The arrows of $M_1 + M_2$ are those of the separate machines.
INITIAL STATES: The initial states of $M_1 + M_2$ are those of the separate machines.
ACCEPTING STATES: The accepting states of $M_1 + M_2$ are those of the separate machines.
In other words we simply consider the pair of machines as a single machine with more than one initial state.

Example 4:
A FSA that accepts the regular language 110 + 01 is:





That is, we simply consider the separate FSAs as a single one. If we wish, we could use the technique for combining multiple initial states to obtain:



However this is not necessary since our procedure for converting a non-deterministic FSA to a deterministic one automatically takes care of multiple initial states.

§7.5. Kleene Star of a Language

Suppose that L is a language accepted by an FSA M.
We'll describe the construction of an FSA, which we call M*, that accepts the language L*.

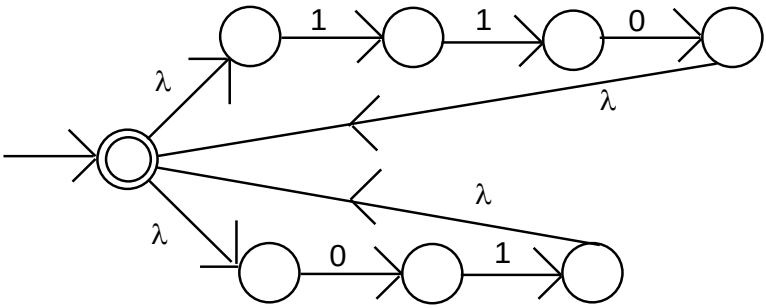
STATES: The states of M* are the same as those of M.

ARROWS: In addition to the arrows of M we construct additional arrows. These are null arrows leading from each accepting state of M to each initial state.

INITIAL STATES: The initial states of M* are the same as those of M.

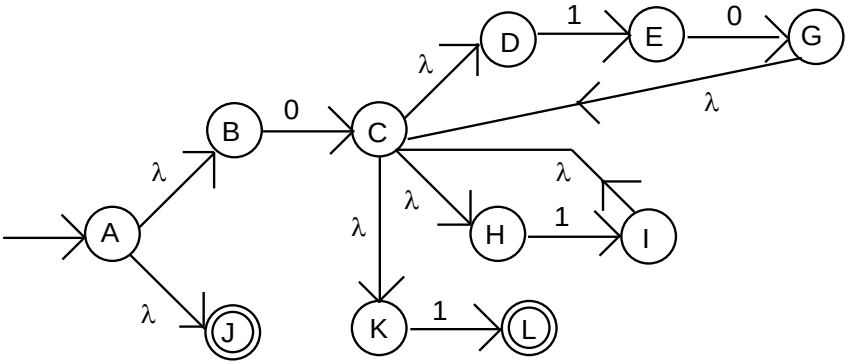
ACCEPTING STATES: The accepting states of M* are the initial states of M. These are both the initial state and the accepting state for M*. The accepting states of M remain (implicitly) accepting because there are null arrows leading from them to an accepting state, but we don't need to label them explicitly at this stage.

Example 5:
An FSA that accepts the regular language (110 + 01)* is:

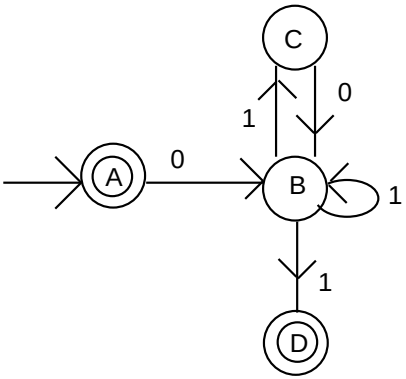


§7.6. Additional Examples

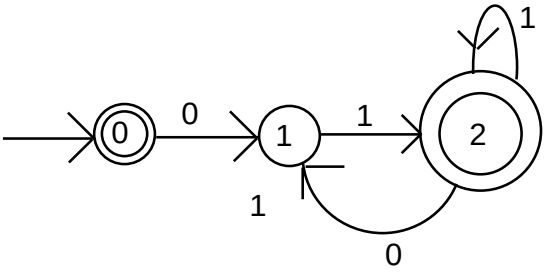
Example 6: An FSA that accepts the regular language 0((10)* + 1)*1 + λ is:



With a little experience you might be able to draw it simply as:



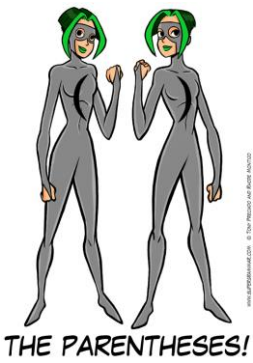
Now converting either of these to a deterministic machine we get:



We can then check that this is reduced and we can easily put it in standard form. Note that a regular expression that describes the language accepted by this final machine is:
$0(11^*0)^*1 + \lambda$. We've therefore shown that $0((10)^* + 1)^*1 + \lambda$ can be simplified to $0(11^*0)^*1 + \lambda$.

Feel free to remove nulls as you draw the original diagram but beware of 'reflux'.

§7.7. Example of a Non-Regular Language



The parenthesis language is a language whose alphabet consists of the left and right parentheses. It consists of those expressions that can arise from a valid arithmetic expression by eliminating all but the parentheses, such as (( ) (( ) ( ))). Strings such as ((( )() or ( )) (are not in the language. Note that there is more to it than simply having the same number of left parentheses as right ones.

If this language was regular it could be accepted by an FSA. However, since parentheses can be nested to arbitrary depths an accepting FSA would need to be able to count arbitrarily far and, with only finitely many

states, this would be impossible. This is only an informal argument. A more rigorous proof follows.

Theorem 2: The parenthesis language is not regular.
Proof: Suppose there's an FSA with N states that accepts the parenthesis language. Consider the expression (((..()...))) with N+1 left parentheses followed by an equal number of right ones. This string is in the language and so must be accepted by the FSA.

Since there are N states and N+1 left parentheses the machine must repeat some state in the course of reading the left parentheses. Suppose the state of the machine after k left parentheses is the same as when it has read a further r parentheses (where $r > 0$). Then adding an extra r left parentheses at the beginning of the expression will make no difference to whether or not a string is accepted.

The state after k or $k + r$ or $k + 2r$ left parentheses must be the same. In terms of the behaviour of the FSA the only difference in adding those extra r left parentheses is that the machine will go round the loop one extra time. But it must terminate in the same final state for each of these expressions.

However the original expression is accepted while the modified expression has too many left parentheses and so should be rejected. This is a contradiction and hence no such FSA can exist.

Theorem 3: The language on the alphabet {1} consisting of all strings of 1's whose length is a perfect square, is not a regular language.

Proof: Suppose there exists an FSA with N states which accepts this language.

Clearly $N > 1$.

Consider the expression 111...1 consisting of N^2 1's. It must be accepted by the FSA.

Now since there are more 1's than there are states, in the course of reading this expression the machine must repeat a state. Suppose that after k 1's it is in the same state as after $k + r$ 1's, where $r > 0$. Then adding an extra r 1's to the string will not affect the outcome. The only difference will be that the FSA will go round this loop an extra time. This can happen any number of times.

So for all positive integers m, $N^2 + mr$ must be accepted and so must be a perfect square. In particular this must be true for $m = N^2r$, that is, $N^2 + (N^2r)r$ is a perfect square.

But $N^2 + (N^2r)r = N^2(1 + r^2)$ and so $1 + r^2$ is a perfect square which is clearly impossible for $r > 0$.

§7.8. Set Operations and Regular Languages

We now show that the set of regular languages is closed under the set operations of union, intersection and

complement. That is, if L and M are regular languages then so are
$L \cup M$, $L \cap M$ and $-L$. Our proof is constructive. That is, if we have FSAs for L and M we show how to construct FSAs for their union, intersection and complements.

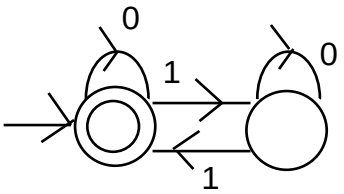
Union:

L + M is easy. As we saw earlier we simply take the machines for the two languages and consider them as a single FSA.

Example 7: Construct an FSA to accept the language of all binary strings which either have even parity or contain 011.

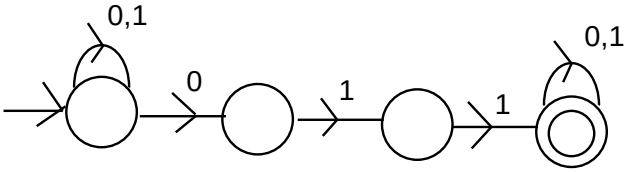
Solution:

The FSA



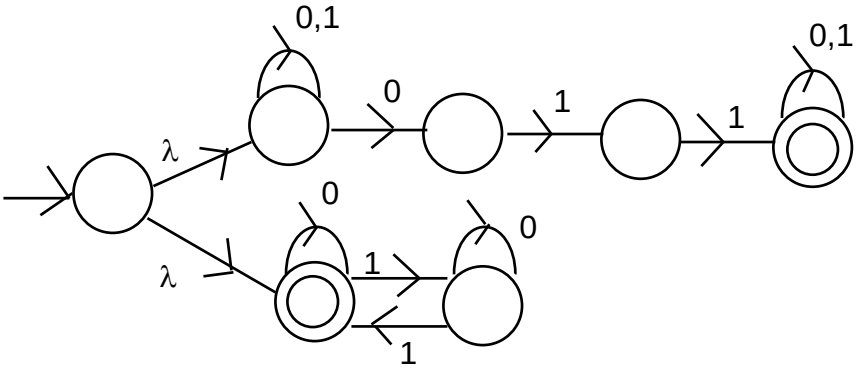
accepts all strings of even parity.

The FSA



accepts all strings containing 011.

Hence the FSA



accepts the language of all binary strings which either have even parity or contain 011.

Complements:

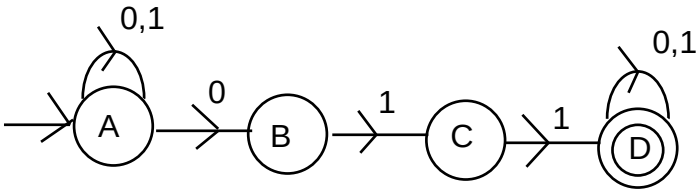
Suppose we have a *deterministic* FSA that accepts the language L. Then we obtain a deterministic FSA for the complement by simply swapping accepting states with non-accepting states. A state in the new machine is accepting if and only if it is non-accepting in the original machine. But this only holds for deterministic FSAs.

However for a non-deterministic FSA we can't simply swap accepting and non-accepting states. This is because a string might be able to reach both accepting and non-accepting states and so the string would be accepted by both M and –M, which certainly is not possible. The solution to this problem is to always make an FSA deterministic before swapping accepting and non-accepting states.

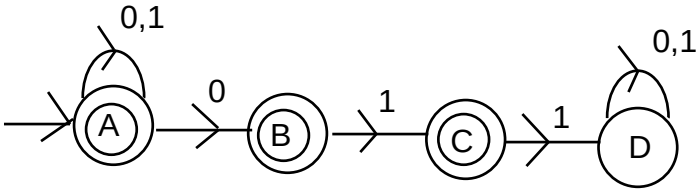
Example 8: Construct an FSA to accept the language of all binary strings which do not contain 011.

Solution:

The following FSA accepts those binary strings which *do* contain 011.



If we swapped accepting with non-accepting states at this stage we'd get



But this would accept 011 (just keep going through state A) and so is not correct. We must first make the FSA deterministic before swapping. Expressing the original FSA as a table we get:

	0	**1**	
→**A**	AB	A	
B		C	
C		D	
D	D	D	*

Making this deterministic we get:

	0	1		
→**0**	1	0		A
1	1	2		AB
2	1	3		AC
3	4	3	*	AD
4	4	5	*	ABD
5	4	3	*	ACD

The FSA for strings not containing 011 is thus:

	0	1	
→**0**	1	0	*
1	1	2	*
2	1	3	*
3	4	3	
4	4	5	
5	4	3	

Intersection:

Recall that if S, T are sets then:

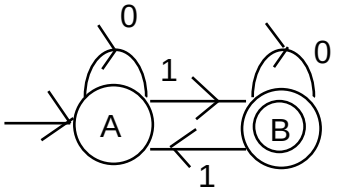
$$S \cap T = -(-S \cup -T).$$

So we can find an FSA to accept the intersection of two languages as follows (where M, N are the respective FSA's):

(1) Make M deterministic;
(2) Construct the negative of this (swap accepting and non-accepting states);

(3) Construct the sum of these machine, to get an FSA that accepts $-S \cup -T$
(by considering the two machines as one);
(4) Construct the negative of this (swap accepting and non-accepting states).

Example 9: Construct an FSA to accept all binary strings of even parity that contain 011.
Solution: This is the complement of the union of the language of all strings which are either of odd parity or which don't contain 011.
An FSA for the language of strings of even parity is:



Note that this is already deterministic.
As a table it is

	0	**1**	
→A	A	B	
B	B	A	*

A deterministic FSA for strings that do not contain 011 was given in example 8. An FSA for the union of these complements is thus:

	0	1	
→**0**	1	0	*
1	1	2	*
2	1	3	*
3	4	3	
4	4	5	
5	4	3	
→**A**	A	B	
B	B	A	*

We need to make this deterministic first before swapping accepting and non-accepting states.

	0	1		
→**0**	1	2	*	0A
1	1	3	*	1A
2	4	0	*	0B
3	4	5	*	2B
4	4	6	*	1B
5	7	8		3A
6	1	8	*	2A
7	7	9		4A
8	10	5	*	3B
9	10	5	*	5B
10	10	11	*	4B
11	7	8		5A

So the required FSA accepting binary strings which both have even parity and contain 011 is:

	0	1		
→**0**	1	2		0A
1	1	3		1A
2	4	0		0B
3	4	5		2B
4	4	6		1B
5	7	8	*	3A
6	1	8		2A
7	7	9	*	4A
8	10	5		3B
9	10	5		5B
10	10	11		4B
11	7	8	*	5A

Example 10: Construct an FSA to accept the language of all binary strings which contain the string 011 and the string 110.

Solution: Proceeding as above we get an FSA for strings not containing 011:

	0	1	
→**0**	1	0	*
1	1	2	*
2	1	3	*
3	4	3	
4	4	5	
5	4	3	

An FSA for strings not containing 110 is:

	0	**1**	
→**0**	0	1	*
1	0	2	*
2	3	2	*
3	3	4	
4	3	5	
5	3	5	

Since the names of our states overlap we must rename one lot. Suppose we rename the states for the second FSA by adding 6 to every state name. This will give us states 0 to 11 for the union.

	0	**1**	
→**0**	1	0	*
1	1	2	*
2	1	3	*
3	4	3	
4	4	5	
5	4	3	
→**6**	6	7	*
7	6	8	*
8	9	8	*
9	9	10	
10	9	11	
11	9	11	

Before we take the complement we must make it deterministic:

	0	**1**		
→**0**	1	2	*	0, 6
1	1	3	*	1, 6
2	1	4	*	0, 7
3	1	5	*	2, 7
4	6	4	*	0, 8
5	7	5	*	3, 8
6	6	8	*	1, 9
7	7	9		4, 9
8	6	10	*	2, 10
9	7	10		5, 10
10	7	10		3, 11

Finally we can now take the complement:

	0	**1**	
→**0**	1	2	
1	1	3	
2	1	4	
3	1	5	
4	6	4	
5	7	5	
6	6	8	
7	7	9	*
8	6	10	
9	7	10	*
10	7	10	*

EXERCISES FOR CHAPTER 7

EXERCISES 7A (Constructing FSAs from Regular Expressions)

Ex 7A1: Construct a *non-deterministic* Finite State Acceptor for the language described by the regular expression (1 + 10)*0 . Give your answer as a state diagram.

Ex 7A2: Construct a *non-deterministic* Finite State Acceptor for the language described by the regular expression 10(1 + 10)*0 + λ . Give your answer as a state diagram.

Ex 7A3: Construct *non-deterministic* FSAs for the following regular languages and convert them to reduced deterministic FSAs in standard form.

(a) 1110;
(b) (10)*;
(c) 1110 + (10)*;
(d) 0(1110 + (10)*)1;
(e) (0(1110 + (10)*)1)*;
(f) (0(1110 + (10)*)1)* + λ

Ex 7A4: Construct a *reduced*, *deterministic* FSA in *standard form* that accepts the language described by the regular expression (0 + 01)*010 + 01* + λ

Ex 7A5: Construct a *reduced*, *deterministic* FSA in *standard form* that accepts the language described by the regular expression $((1 + 11)^*1)^*1 + (11 + \lambda)^*$

Ex 7A6: Construct a *reduced*, *deterministic* FSA in *standard form* that accepts the language described by the regular expression $0^*((1 + 11)^*1)^*1 + (11 + \lambda)^*$

Ex 7A7: Construct a *non-deterministic* Finite State Acceptor for the regular language consisting of all strings on the set {E, S, Y} which contain the substring YES.
Give your answer as a state diagram.

Ex 7A8: Construct a state diagram for a non-deterministic FSA to accept the regular language
$10((10)^* + 11) + \lambda$.

Ex 7A9: (a) Construct a state diagram for a non-deterministic FSA to accept the regular language $(011 + \lambda)(01 + 11 + 011)^*0 + 0111$.
(b) Convert your non-deterministic FSA to a deterministic one, giving it as a state table.
(c) Reduce your deterministic FSA in (b) to an equivalent one on the smallest number of states.
(d) Put your minimised FSA into standard form.

EXERCISES 7B (Non-Regular Languages)

Ex 7B1: Prove that the set of all binary strings that are palindromic (that is, they read the same forwards as backwards) is not a regular language.

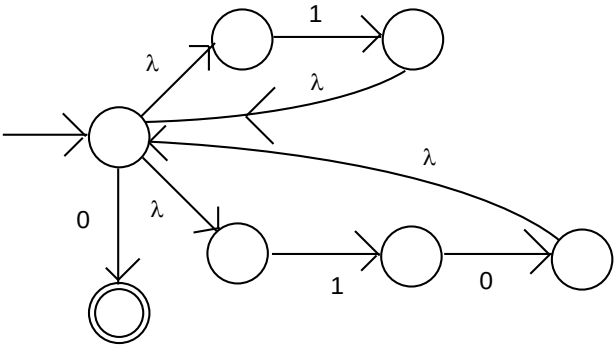
[**HINT:** Adapt the proof given for the non-regularity of the parenthesis language in §7.7.]

Ex 7B2: A *square binary string* is one that can be expressed in the form $\alpha\alpha$ for some α (that is, two copies of the same string). Prove that the language of all square binary strings is not a regular language.

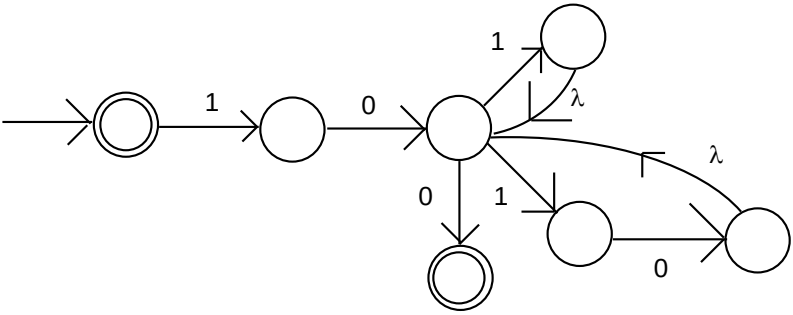
Ex 7B3: A string of +'s and −'s is defined to be *optimistic* if preceding any character there are at least as many +'s as −'s. Prove that the language of all optimistic strings is not a regular language.

SOLUTIONS FOR CHAPTER 7

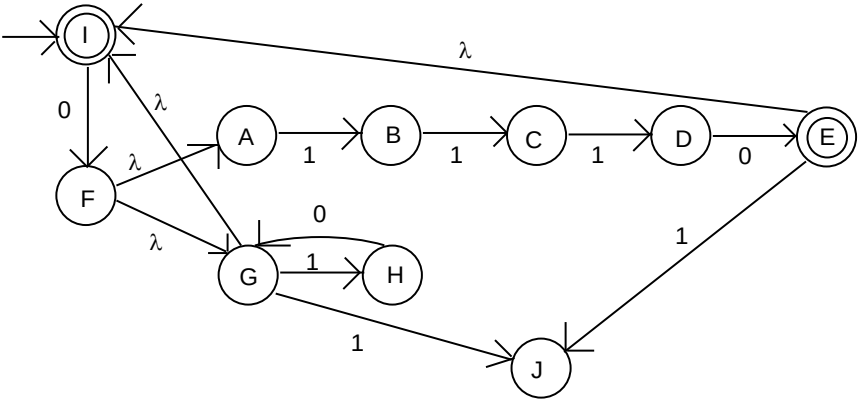
Ex 7A1:



Ex 7A2:



Ex 7A3: (f) is as follows. The earlier parts are building blocks of the final machine.



Ex 7A4:

	0	**1**	
→0	1	2	*
1	3	4	*
2	2	2	
3	3	5	
4	6	7	*

5	6	2	
6	3	5	*
7	2	7	*

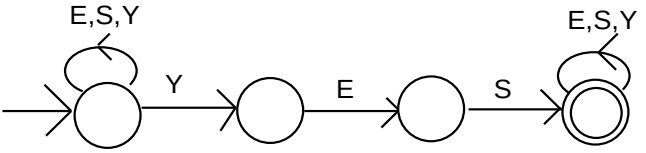
Ex 7A5: Observe that ((1 + 11)*1)*1 + (11 + λ)*
= (1*1)*1 + (11)* = 1*1 + (11)*
= 1*1 + λ = 1*.
The reduced deterministic FSA in standard form that accepts 1* is:

Ex 7A6:
Observe that 0*((1 + 11)*1)*1 + (11 + λ)*
= 0*(1*1)*1 + (11)*
= 0*1*1 + (11)*.

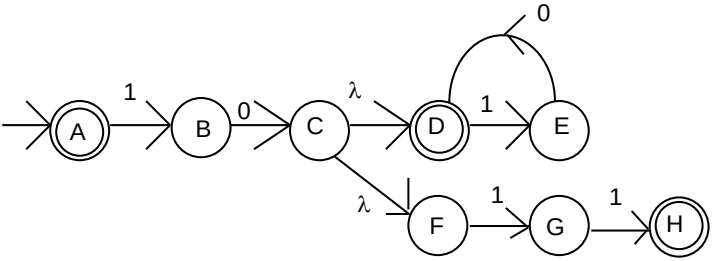
The reduced deterministic FSA in standard form that accepts 0*1*1 + (11)* is:

	0	**1**	
→**0**	1	2	*
1	1	2	
2	3	2	*
3	3	3	

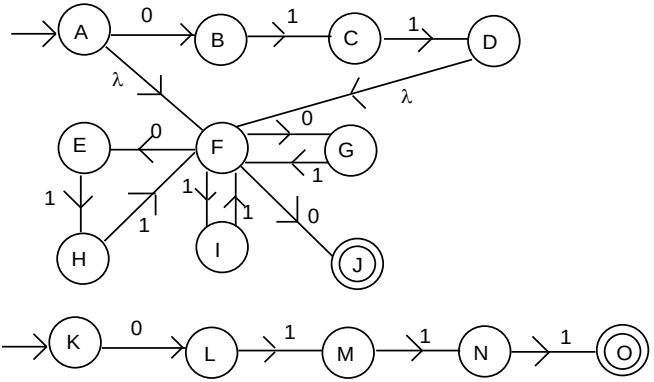
Ex 7A7:



Ex 7A8:



Ex 7A9:



	0	1	
→A	BEGJ	I	
B		C	
C		D	
D	EGJ	I	
E		H	
F	EGJ	I	
G		F	
H		F	
I		F	
J			*
→K	L		
L		M	
M		N	
N		O	
O			*

	0	1	$\equiv_0$	0	1	$\equiv_1$	0	1	$\equiv_2$
→0	1	2	0	1	0	0	1	2	0
1	3	4	1	0	0	1	2	0	1
2	3	5	0	0	0	2	2	0	2
3	3	3	0	0	0	2	2	2	3
4	6	7	0	1	0	0	1	3	4
5	6	2	0	1	0	0	1	2	0
6	3	8	1	0	0	1	2	3	5
7	6	9	0	1	1	3	1	4	6
8	6	10	0	1	1	3	1	3	7
9	6	10	1	1	1	4	1	3	8
10	6	10	0	1	1	3	1	3	7

	0	**1**	**≡$_2$**	**0**	**1**	**≡$_3$**	**0**	**1**	**≡$_4$**
→0	**1**	**2**	0	1	2	**0**	1	2	**0**
1	**3**	**4**	1	3	4	**1**	3	4	**1**
2	**3**	**5**	2	3	0	**2**	3	5	**2**
3	**3**	**3**	3	3	3	**3**	3	3	**3**
4	**6**	**7**	4	5	6	**4**	6	7	**4**
5	**6**	**2**	0	5	2	**5**	6	2	**5**
6	**3**	**8**	5	3	7	**6**	3	8	**6**
7	**6**	**9**	6	5	8	**7**	6	9	**7**
8	**6**	**10**	7	5	7	**8**	6	8	**8**
9	**6**	**10**	8	5	7	**9**	6	8	**9**
10	**6**	**10**	7	5	7	**8**	6	8	**8**

So 10 ≡ 8. We omit state 10 redirecting all references to it to a reference to 8.

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

This is already in Standard Form.

Ex 7B1: Suppose the language of palindromic binary strings is regular. Then there exists an FSA which accepts this language. Suppose it has N states. The string $0^{N+1}\ 10^{N+1}$ belongs to the language and so is accepted by the FSA. In the course of reading the first N + 1 0's the FSA must repeat a state, say after *k* steps and after *k* + *h* steps the machine is in the same state *s*. Introducing a further *h* 0's in this portion of the string will not affect the behaviour of the machine, except that it will cycle around from state *s* back to state *s* one extra time. The ultimate behaviour must be the same for the original and the modified strings. Hence the FSA must accept $0^{N+1+h}10^{N+1}$. But this string is not palindromic and so does not belong to the language, contradicting our original assumption.

Ex 7B2: Suppose that the language of square binary strings is regular. Then there exists an FSA which accepts it. Suppose it has *N* states. The string $0^{N+1}10^{N+1}1$ belongs to the language and so is accepted by the FSA. In the course of reading the first N + 1 0's the FSA must repeat a state, say after *k* steps and after *k* + *h* steps the machine is in the same state *s*. Introducing a further *h* 0's in this portion of the string will not affect the behaviour of the machine, except that it will cycle around from state *s* back to state *s* one extra time. The ultimate behaviour must be the same for the original and the

modified string. Hence the FSA must accept $0^{N+1+h}10^{N+1}1$. But this string is not square and so doesn't belong to the language, contradicting our original assumption.

Ex 7B3:

Proof:

Suppose there is an FSA with N states that accepts the parenthesis language. Consider the expression:

$$+ + + .. - - - ...$$

with N+1 +'s followed by an equal number of −'s. This string is in the language and so must be accepted by the FSA.

Since there are N states and N+1 −'s the machine must repeat some state in the course of reading the −'s. Suppose the state of the machine after k left −'s is the same as when it has read a further r −'s (where $r > 0$). Then adding an extra r −'s at the end of the expression will therefore make no difference to the outcome.

The state after k or $k + r$ or $k + 2r$ −'s must be the same. In terms of the behaviour of the FSA the only difference in adding those extra r −'s is that the machine will go round the loop one extra time. But it must terminate in the same final state for each of these expressions. However the original expression is accepted while the modified expression has too many −'s and so is not optimistic. It should therefore be rejected, a contradiction. Hence no such FSA can exist.