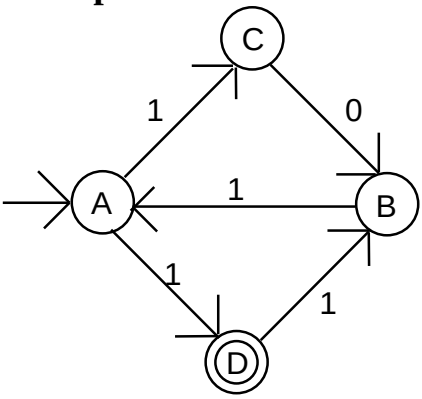
6. NON-DETERMINISTIC FINITE STATE MACHINES

§6.1. Multiple Transitions

Do you notice anything peculiar about the following FSA?

Example 1:



There seems to be a mistake because there are two transitions labelled ‘1’ coming out of state A. If we are in state A and we read a ‘1’, do we go to state C or to state D? We appear to have a choice. Does this mean we have to toss a coin so that sometimes we go to state C but on other occasions, with the same input string, we go to D?

There’s no mistake. This is an example of what is known as a **non-deterministic** FSA. It appears to incorporate the element of chance. But what possible use could there be for a machine whose behaviour is unpredictable? Well, let’s ask what strings can be accepted by this FSA. First we have to modify our definition of ‘accepted’.

A string is **accepted** by a non-deterministic FSA if *it is possible* to terminate in an accepting state.

The string 1011 could terminate in the non-accepting state C by going around the top triangle. On other occasions it will go straight out to D on the first '1' and then disappear into the black hole. But sometimes it will go once round the loop ACB and then go to the accepting state D. The fact that it can end up in an accepting state means that this string is accepted by this machine.

The string 1111 is also accepted if it goes around the bottom loop. In fact going around this loop any number of times, a sequence of 1's represented by the regular expression (111)*1 will be accepted. But going around the top loop an arbitrary number of times and then going from A to D we get all strings that can be described by (101)*1. So the language accepted by this FSA includes all strings of the form (111)*1 + (101)*1.

A string might be accepted by going around both loops many times in any order. For example 1011111011011 is accepted by going around the top loop once, then around the bottom loop once, then around the top loop twice more, then finally moving to the accepting state D. This non-deterministic FSA accepts the language (101 + 111)*1.

The point of considering non-deterministic machines is that in many cases they are much easier to design than deterministic ones. Try to construct a deterministic FSA to accept the language (101 + 111)*1. It isn't so easy.

So, the advantage of non-deterministic FSAs over deterministic ones is that they are easier to design. But how do we implement them?

We could use a random number generator to simulate the rolling of dice whenever we have a choice to make. But if we wanted to use this machine to decide whether or not the string 1011011 is to be accepted we'd have to run the machine a large number of times, large enough for the possibility to arise of going twice around the top loop before coming out to D. This seems quite clumsy. And even if we ran the machine hundreds of times it is just possible, even if quite unlikely, that the machine will never choose this accepting path.

Non-deterministic FSA's are much simpler to design but deterministic ones are more satisfactory to implement. To get the best of both worlds we need a routine method of converting non-deterministic FSA's to deterministic ones.

Another way of thinking of non-deterministic machines is to imagine that different possibilities are

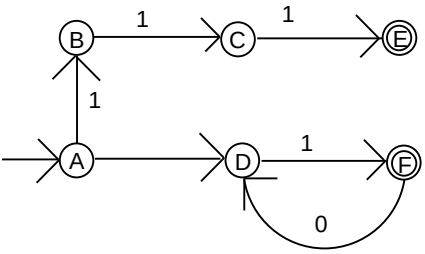
investigated in parallel. A team of people, communicating with each other by mobile phones, can search for a treasure in a maze by using the parallel technique. Faced with a choice, instead of tossing a coin to decide which way to go, they simply split into two groups and say to each other, “you go this way, we'll go the other”. These groups may split again as further choices confront them. As soon as one party finds the treasure the others are notified.

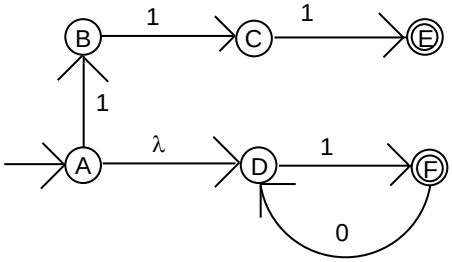
§6.2. Null Transitions

Having multiple transitions, for the same input, is not the only way an FSA can be non-deterministic. It can have null transitions. Consider the following machine:

Example 2:

Note the two transitions coming out of state A. No, I haven’t forgotten to write the input characters that produce the transition from A to D. It is a ‘null transition’. It can be made spontaneously, without reading any input.





However the danger of using an arrow without a label to represent a null transition is that if we forget to label an arrow it will be interpreted wrongly. To avoid this we use the null string symbol λ to represent a null transition. Then, if we come across an unlabelled arrow we'll know that somebody has simply forgotten to label it. So the first diagram is incomplete and should be drawn as in the second diagram.

Let's run this machine for the input string 1010. As indicated, we start at A. Then we toss a coin to decide whether to move straight to D or read the first input character.

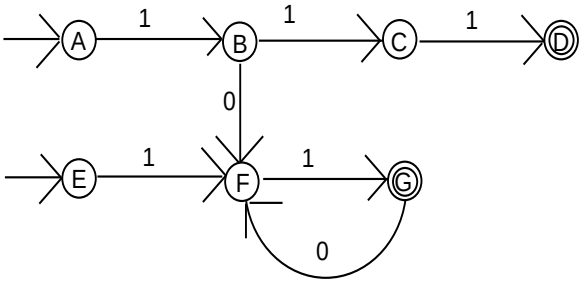
Perhaps we choose to read the first character. It's a '1' so we go to state B. Reading the '0' sends us into the black hole. The string has been rejected. But with a non-deterministic machine we have to give it another chance. Maybe the next time the same thing will happen. But sooner or later the choice of moving spontaneously to state D will arise. This time the string will be accepted in state F.

So 1010 is accepted by the machine, not because this will happen every time, or because it is likely that running it a certain number of times means it will end in

an accepting state. It is accepted because there is the *possibility* for it to end in an accepting state.

§6.3. Multiple Initial States

Example 3:



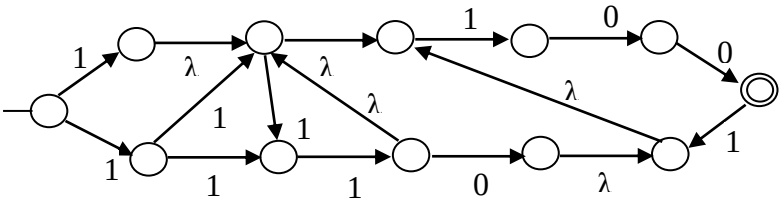
This non-deterministic machine offers a choice of where to start. It accepts the strings 111 and those such as 101, 10101, 1010101, .by starting at A. It also accepts the strings such as 11, 1101, 110101, ... by starting at E. The language accepted by this machine is:

$$111 + 10(10)^*1 + 1(10)^*1.$$

Having seen that there may be some merit in being allowed to design non-deterministic machines, provided there is a routine way of converting them to deterministic ones, let's start thinking about this conversion process.

§6.4. Completing the Nulls

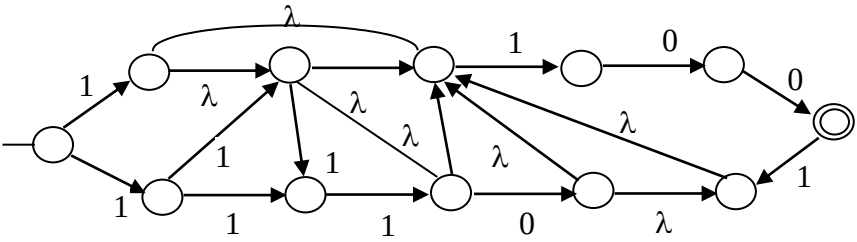
Example 4: In the following NDFSA it is possible to move spontaneously along several null transitions one after the other.



Completing the Nulls:

If there'is a sequence of null transitions from state A to state B add a direct null transition from A to B (unless there's already one). We then say that the NDFSA is **complete**.

Strictly speaking there's a sequence of nulls from each state to itself (a sequence of length zero) and so we ought to introduce a null from each state to itself. However this would clutter up the diagram too much and so we merely remember that there is an invisible null from each state to itself. Completing the nulls in the above NDFSA we get:



Working with a table is much more reliable than using a diagram as it's not so easy to miss a null in a table. We use the same system to complete the nulls as to find accessible states.

Example 5:
We need to complete the nulls for the following NDFSA:

	0	**1**	**λ**
→A	B	A	BD
B	F	C	F
C		F	
D	AC		A
E		ABC	
F	C		BE

(Here I've shaded the non-null portion of the table as that information is not needed for this stage.)

Write down A. Read off the fact that there are nulls from A to B and D so write down BD after the A and underline the A to record the fact that we have dealt with it.

ABD

Now consider B (the first non-underlined letter). There's a null leading from B to F so adjoin C to the list and underline the B.

ABDF

There's a null from D to A, but we already have that in our list so we merely underline the D.

ABDF

There's a null from F to both B and E. Since B is already there we merely add E to the list and underline the F.

ABDFE

There are no nulls from E so we have nothing to add. We merely mark off E as having been considered.

ABDFE

We have now nothing further to consider so move onto B.

So A→ABD→ABDF→ABDF→ABDFE→ABDFE;
B→BF→BFE→BFE;
C→C;
D→DA→DAB→DABF→DABFE→DABFE;
E→E;
F→FBE→FBE→FBE

This is exactly the same procedure that we used to find accessible states. The only differences are that:
(1) we only follow the null transitions and
(2) we begin at each state.

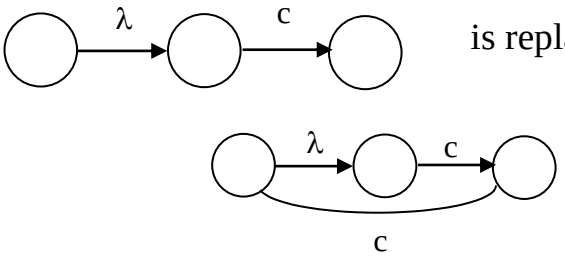
We record the information in the λ-column of the transition table. However, as explained above, we omit the state itself. So instead of recording FBE for F we record BE.

	0	1	λ
→**A**	B	A	BDFE
B	F	C	FE
C		F	
D	AC		ABFE
E		ABC	
F	C		BE

§6.5. Removing the Nulls

Suppose we've a complete NDFSA. We now remove any null transitions as follows.

Removing nulls:
If there is a null transition from A to B and a non-null transition from B to C then add a non-null transition from A to C and associate it with the same input character as the non-null transition.



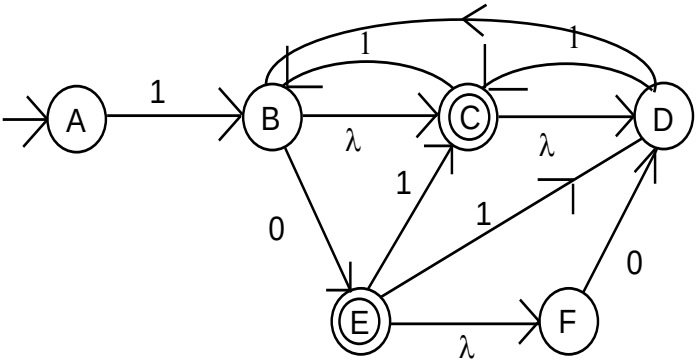
The nulls are now largely superfluous. We don't need to use them because we've provided these alternative paths. Moreover we don't have to worry

about nulls that follow a non-null because these will be processed before a subsequent non-null.

The only exceptions are terminating nulls. To allow for these we have to convert any state that leads to an accepting state by a null, into an accepting state (if it's not one already).

The null transitions are now completely redundant are so are now removed.

Example 6:
Consider the following complete NDFSA:



To start with, let us record this information in a table.

	0	1	λ	
→**A**		B		
B	E		CD	
C		B	D	*
D		C		
E		CD	F	*
F	D			

We look across the table from B to the λ-column where we find CD. If we're in state B, reading a '1' we could use the '1' directly from B (though in this case there's no appropriate 1-transition) or we could first move by a λ to either C or D and then use the '1'.

So the set of states we could move to is got by taking the union of the three sets in the 1-column of the B, C and D rows combined. This is $\varnothing \cup \{B, C\} \cup \{B \cup C\} = \{B, C\}$, which we're writing as BC. Similarly for the 0-columns where we get $\{E\} \cup \varnothing \cup \cup \varnothing = \{E\}$. So, taking the union of the BCD rows together (because B→CD in the λ-column):

	0	1	λ	
B	E		CD	
C		B	D	*
D		C		

and taking the union of the entries in the 0- and 1-columns we get:

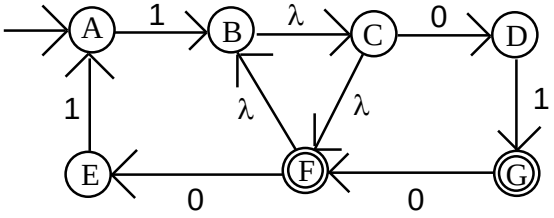
B	E	BC		*

This becomes the new B row. We put an * because there was one in at least one of the BCD rows. If none of states B, C, D had been accepting we'd have recorded a blank (non-accepting). Note too that we don't bother with combining the entries in the λ-column, because we're ready to dispense with them.

Proceeding in this way we get the following state table:

	0	**1**	
→**A**		B	
B	E	BC	*
C		BC	*
D		C	
E	D	CD	*
F	D		

Example 7:
Consider the following non-deterministic FSA:



The state table for this FSA is:

	0	1	λ	
→**A**		B		
B			C	
C	D		F	
D		G		
E		A		
F	E		B	*
G	F			*

Completing the nulls we get:

	0	1	λ	
→**A**		B		
B			CF	*
C	D		BF	*
D		G		
E		A		
F	E		BC	*
G	F			*

Removing the nulls we get:

	0	1	
→**A**		B	
B	DE		*
C	DE		*
D		G	
E		A	
F	DE		*
G	F		*

State B is an accepting state because there are null transitions leading from it to F. Similarly C must be considered accepting.

We now have multiple transitions. In the next section we'll see how to combine these.

§6.6. Combining Multiple Transitions

Suppose now we have a non-deterministic FSA, M, with all its null transitions removed, but having multiple transitions from a state for the same input character. We transform this non-deterministic one to a deterministic one (we shall call this M^{Δ} with the capital Greek delta representing 'deterministic;) by considering *sets* of states as the states of the deterministic machine.

Suppose M has a set of states S, a transition function T and a set of accepting states, A. Then this new machine, M^{Δ}, is defined to have $\wp(S)$, the power set of S, as its set of states. Its transition function is T^{Δ} where $T^{\Delta}(X, c) = \cup\{T(x, c) \mid x \in X$.
For example, $T(\{3, 5\}, 1) = T(3, 1) \cup T(5, 1)$.

The set of accepting states for M^{Δ} is $A^{\Delta} = \{X \mid X \cap A \neq \varnothing\}$. So, if the set of accepting states is $A = \{2, 3\}$ then $\{1, 2, 5\} \in A^{\Delta}$ since $\{1, 2, 5\} \cap A = \{2\}$, which is non-empty.
But $\{1, 4, 5\}$ is not accepting since it is disjoint from A.

So the states in M^{Δ} are the sets of states in M. To get the value of $T^{\Delta}(X, c)$ we just combine the $T(x, c)$'s for all the $x \in X$. Finally we mark as accepting, any set of states for the original non-deterministic machine that contains an accepting state in M.

The resulting FSA will clearly be deterministic because from each set of states of the original machine there'll only be one set of states that we can reach for a given input character. So now we'll have a deterministic FSA that's equivalent to M, in the sense that it accepts precisely the same language as M.

Example 8:
Convert the following non-deterministic FSA to a deterministic one:

	0	**1**	
→**A**	CD	D	*
B	AD	B	
C	AC	C	
D	A		

Solution: There are four states and so 16 sets of states. But rather than process all of them we only consider them as they arise. In this way we automatically omit any inaccessible combinations and this can save a lot of work.

We label the sets of states as 0, 1, 2, ... and annotate our table with a description of which states each label represents.

We begin by assigning the label 0 to the set of initial states, in this case just A. Notice that this automatically deals with multiple initial states.

The first set to arise in the table is CD so this we label as 1, and label 2 is assigned to D. We replace CD and D in the table by their codes 1 and 2. Remember that CD is shorthand for {C, D}.

So far we have:

	0	**1**	
→**0**	1	2	A
1			CD
2			D

To process CD we simply take the union of the C and D rows of the non-deterministic table, giving AC and C respectively. Since we don't yet have labels for these we assign to them the labels 3 and 4.

	0	**1**	
→**0**	1	2	A
1	3	4	CD
2			D
3			AC
4			C

State D is just as it is in the original table: A and blank. State A has already been assigned the label 0, but blank, meaning the empty set, has yet to be given a label. It's now time to do this and so it's labelled 5. This, of course, is our black hole from which we never escape.

So far we have:

	0	1	
→**0**	1	2	A
1	3	4	CD
2	0	5	D
3			AC
4			C
5			∅

AC leads to ACD and CD. We've already labelled CD as '1' but ACD is new and so is labelled 6. C leads to AC and C, ∅ leads to ∅ under both characters and ACD leads to ACD and CD. This gives us no new combinations so we finish at that point.

	0	1	
→**0**	1	2	A
1	3	4	CD
2	0	5	D
3	6	1	AC
4	3	4	C
5	5	5	∅
6	6	1	ACD

All that remains is to determine which of these states are accepting states. Remember that a set of states is accepting if it contains at least one accepting state. So here, any set containing A is accepting.

	0	1		
→**0**	1	2	*	A
1	3	4		CD
2	0	5		D
3	6	1	*	AC
4	3	4		C
5	5	5		∅
6	6	1	*	ACD

This is now a deterministic machine that is equivalent to the original non-deterministic one. We can now omit the last column (it simply connects the states of these two machines but is no longer needed).

NOTES: (1) This process automatically omits any inaccessible states or combinations of states. There are 16 possible subsets, but only these 7 are accessible.

(2) This process automatically puts the FSA in standard form. The only further processing that we may need is to identify equivalent states.

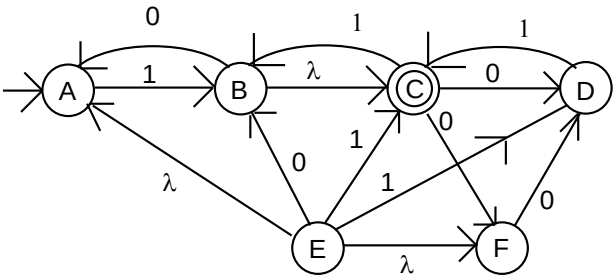
§6.7. A Worked Example

Given the state diagram for a non-deterministic FSA there are five stages in converting it to a reduced deterministic FSA in standard form:

(1) Construct the state table;
(2) Complete the nulls;
(3) Remove the nulls;
(4) Remove multiple transitions;
(5) Combine equivalent states;
(6) Put into standard form.

The next example works through all these processes.

Example 9: Convert the following non-deterministic FSA to a deterministic one in standard form. Give your answer as a state table.



(1) Construct a state table

	0	**1**	**λ**	
→**A**		B		
B	A		C	
C	DF	B		*
D		C		
E	B	CD	AF	
F	D			

(2) Complete the nulls:

	0	1	λ	
→**A**		B	A	
B	A		BC	
C	DF	B	C	*
D		C	D	
E	B	CD	AEF	
F	D		F	

(3) Remove the nulls:

	0	1	
→**A**		B	
B	ADF	B	*
C	DF	B	*
D		C	
E	BD	BCD	
F	D		

(4) Remove multiple transitions:

	0	1		
→**0**	1	2		A
1	1	1		∅
2	3	2	*	B
3	4	5		ADF
4	1	6		D
5	3	2	*	BC
6	7	2	*	C
7	4	6		DF

(5) Combine equivalent states:

	0	**1**	**≡$_0$**	**0**	**1**	**≡$_1$**	**0**	**1**	**≡$_2$**	**0**	**1**	**≡$_3$**
→0	1	2	0	0	1	0	1	2	0	1	2	0
1	1	1	0	0	0	1	1	1	1	1	1	1
2	3	2	1	0	1	2	0	2	2	3	2	2
3	4	5	0	0	1	0	0	2	3	0	2	3
4	1	6	0	0	1	0	1	2	0	1	2	0
5	3	2	1	0	1	2	0	2	2	3	2	2
6	7	2	1	0	1	2	0	2	2	3	2	2
7	4	6	0	0	1	0	0	2	3	0	2	3

Identify 4 with 0, identify 5 and 6 with 2 and identify 7 with 3.

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

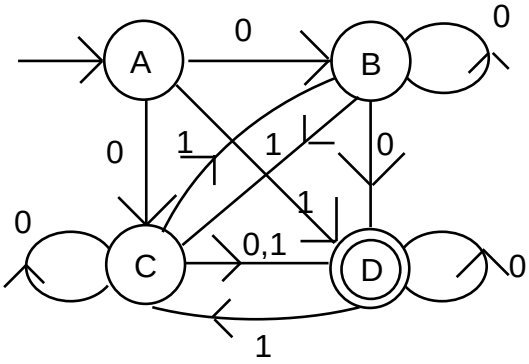
(6) Put into standard form:

It is already in Standard Form.

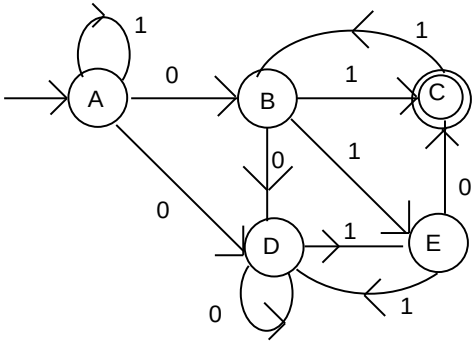
EXERCISES FOR CHAPTER 6

EXERCISES 6A (Combining Multiple Transitions)

Ex 6A1: Convert the following non-deterministic FSA to a deterministic one. Give your answer as a state table.

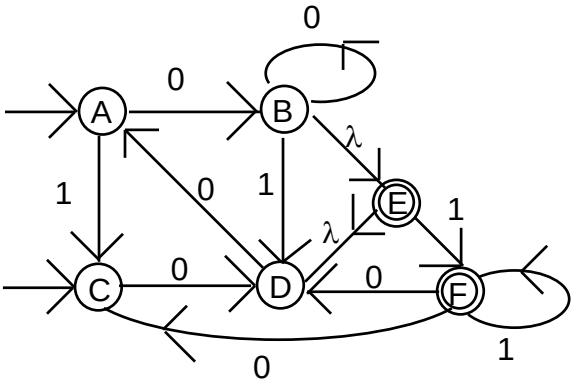


Ex 6A2: Convert the following non-deterministic Finite State Acceptor into a deterministic one. Give your answer as a state table.

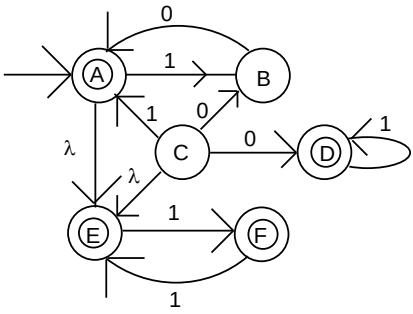


EXERCISES 6B (Removing Null Transitions)

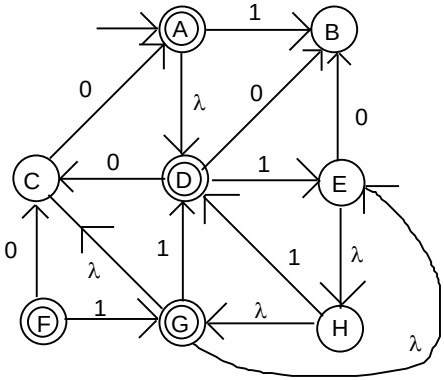
Ex 6B1: Convert the following non-deterministic FSA to a deterministic one. Express you machine as a table in standard form, giving the transition table, the accepting states and a final column in which you describe which set of states of the non-deterministic machine corresponds to each of the states in the deterministic one. Be patient as there are 15 states. *You do not need to minimise your machine.*



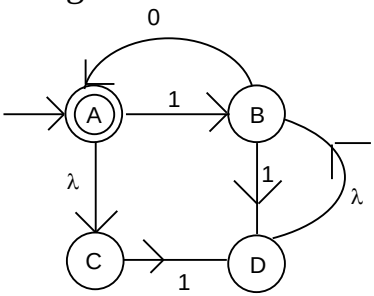
Ex 76B2: Convert the following non-deterministic FSA to a deterministic one. Then reduce it and put it in standard form.



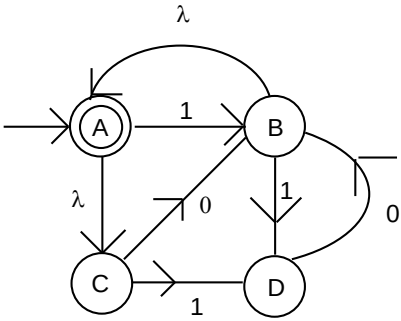
Ex 6B3: Convert the following non-deterministic FSA to a deterministic one. Then reduce it and put it in standard form.



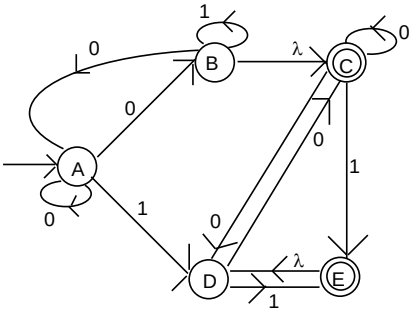
Ex 6B4: Remove the null transitions from the following non-deterministic Finite State Acceptor. (You are not required to make it fully deterministic.) Give your answer as a state diagram.



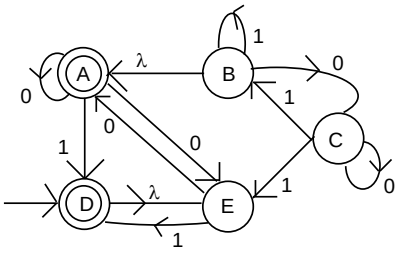
Ex 6B5: Remove the null transitions from the following non-deterministic Finite State Acceptor. (You are not required to make it fully deterministic.) Give your answer as a state table.



Ex 6B6: Convert the following non-deterministic Finite State Acceptor to a deterministic one, expressing your answer as a state table in standard form.



Ex 6B7: Convert the following non-deterministic Finite State Acceptor to a deterministic one, expressing your answer as a state table in standard form.



SOLUTIONS FOR CHAPTER 6

Ex 6A1: The state table for the non-deterministic machine is:

	0	1	
→**A**	BC	D	
B	BD	C	
C	CD	BD	
D	D	C	*

Converting this to a deterministic machine we get:

	0	1		
→**0**	1	2		A
1	3	3		BC
2	2	4	*	D
3	3	3	*	BCD
4	5	6		C
5	5	3	*	CD
6	6	4	*	BD

Ex 6A2: The state table for the non-deterministic machine is:

	0	1	
→**A**	BD	A	
B	D	CE	
C		B	*
D	D	E	
E	C	D	

Converting this to a deterministic machine we get:

	0	1		
→**0**	1	0		A
1	2	3		BD
2	2	4		D
3	5	1	*	CE
4	5	2		E
5	6	7	*	C
6	6	6		∅
7	2	3		B

Ex 6B1: Removing the null transitions we get:

	0	1	
→**A**	B	C	
B	BA	DF	*
→**C**	D		
D	A		
E	A	F	*
F	CD	F	*

Making it deterministic we get:

	0	1		
→**0**	1	2		AC
1	3	4	*	BD
2	5	6		C
3	3	7	*	AB
4	8	9	*	DF

5	10	6		D
6	6	6		∅
7	8	9	*	CDF
8	11	2		ACD
9	12	9	*	F
10	13	2		A
11	3	7	*	ABD
12	14	6		CD
13	3	4	*	B
14	3	2		AD

Ex 6B2: The state table for the non-deterministic FSA (after removing the null transitions) and the corresponding deterministic one are:

	0	**1**	
→**A**		BF	*
B	A		
C	BD	AF	*
D		D	*
E		F	*
F		E	*

Our process automatically removes inaccessible states. On checking for equivalents states we find that 4 ≡ 3. Removing state 4 we get a reduced deterministic FSA in standard form equivalent to the given one:

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

Ex 6B3: The state table for the non-deterministic FSA (after removing the null transitions) and the corresponding deterministic FSA is:

	0	1	
→A	BC	BE	*
B			
C	A		
D	BC	E	*
E	AB	D	*
F	C	G	*
G	AB	D	*
H	AB	D	*

Making it deterministic we get:

	0	1		
→**0**	1	2	*	A
1	0	3		BC
2	4	5	*	BE
3	3	3		∅

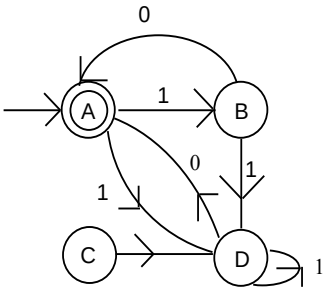
4	1	2	*	AB
5	1	6	*	D
6	4	5	*	E

Now 5 ≡ 4 ≡ 0 and 6 ≡ 2 (details omitted). Removing states 4, 5, 6 we get a reduced deterministic FSA in standard form equivalent to the given one.

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

Ex 6B4:

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



Ex 6B5:

	0	**1**	
→**A**	AB	D	
B	ACD	BE	*
C	CD	E	*

	0	**1**	
D	C	E	
E	C	E	*

Ex 6B6:

	0	1		
→**0**	1	2		A
1	3	4	*	AB
2	5	6		D
3	3	4	*	ABCD
4	7	8	*	BDE
5	9	6	*	C
6	5	6	*	E
7	3	10	*	ACD
8	7	8	*	BE
9	9	6	*	CD
10	5	6	*	DE

Ex 6B7: Removing the nulls:

	0	1	
→**A**	AE	D	*
B	ACE	BD	*
C	C	BE	
D	A	D	*
E	A	D	

Making it deterministic:

	0	1		
→**0**	1	0	*	D
1	2	0	*	A
2	2	0	*	AE

It isn’t necessary to go through the formal process of minimization because clearly this accepts all strings, since every state is accepting. So it can be reduced to a single state.

	0	1	
→0	0	0	*