Chapter 7 Solution

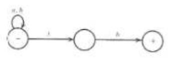
Automata Theory (University of the Punjab)

Chapter 7, Problem 1P

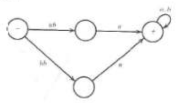
12 Bookmarks Show all steps: ON

Using the bypass algorithm in the proof of Theorem 6, Part 2, convert each of the following TGs into regular expressions:

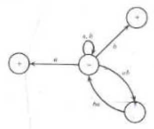
(i)



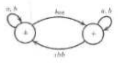
(ii)



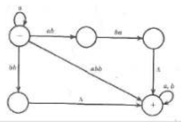
(iii)



(iv)



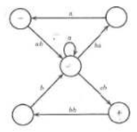
(v)



Chapter 7, Problem 1P

12 Bookmarks Show all steps: ON

vi)



THEOREM 6

Any language that can be defined by

regular expression, or

finite automaton, or

transition graph

can be defined by all three methods.

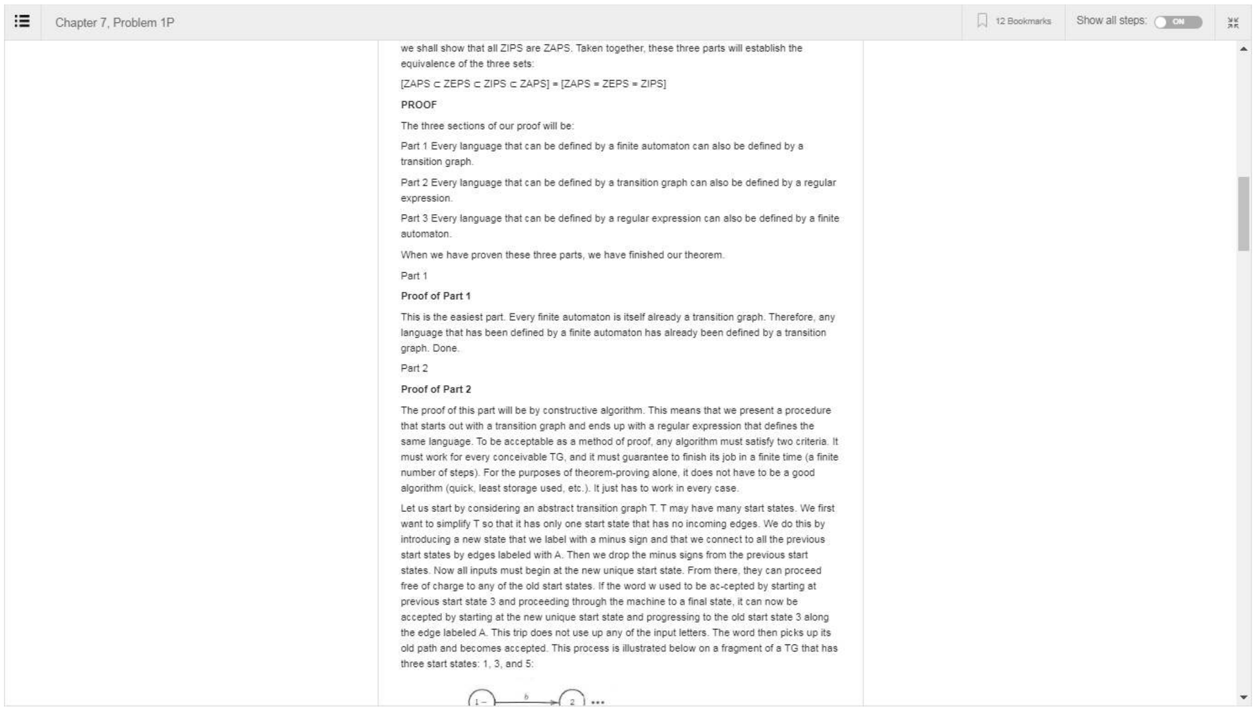
This theorem is the most important and fundamental result in the theory of finite automata. We are going to take extreme care with its proof. In the process, we shall introduce foui algorithms that have the practical value of enabling us actually to construct the corresponding machines and expressions. More than that, the importance of this chapter lies in its Value as an illustration of thorough theoretical thinking in this field.

The logic of this proof is a bit involved. If we were trying to prove the mathematical theorem that the set of all ZAPS (whatever they are) is the same as the set of all ZEPS, we could break the proof into two parts. In Part 1, we would show that all ZAPS are also ZEPS. In Part 2, we would show that all ZEPS are also ZAPS. Together, this would demonstrate the equivalence of the two sets.

Here, we have a more ambitious theorem. We wish to show that the set of ZAPS, the set of ZEPS, and the set ol ZIPS are all the same. To do this, we need three parts. In Part 1, we shall show that all ZAPS are ZEPS. In Part 2, we shall show that all ZEPS are ZIPS. Finally, in Part 3, we shall show that all ZIPS are ZAPS. Taken together, these three parts will establish the equivalence of the three sets:

[ZAPS ⊂ ZEPS ⊂ ZIPS ⊂ ZAPS] = [ZAPS = ZEPS = ZIPS]

PROOF



Chapter 7, Problem 1P

12 Bookmarks

Show all steps: ON

we shall show that all ZIPS are ZAPS. Taken together, these three parts will establish the equivalence of the three sets:

[ZAPS ⊂ ZEPS ⊂ ZIPS ⊂ ZAPS] ⇒ [ZAPS = ZEPS = ZIPS]

PROOF

The three sections of our proof will be:

Part 1 Every language that can be defined by a finite automaton can also be defined by a transition graph.

Part 2 Every language that can be defined by a transition graph can also be defined by a regular expression.

Part 3 Every language that can be defined by a regular expression can also be defined by a finite automaton.

When we have proven these three parts, we have finished our theorem.

Part 1

Proof of Part 1

This is the easiest part. Every finite automaton is itself already a transition graph. Therefore, any language that has been defined by a finite automaton has already been defined by a transition graph. Done.

Part 2

Proof of Part 2

The proof of this part will be by constructive algorithm. This means that we present a procedure that starts out with a transition graph and ends up with a regular expression that defines the same language. To be acceptable as a method of proof, any algorithm must satisfy two criteria. It must work for every conceivable TG, and it must guarantee to finish its job in a finite time (a finite number of steps). For the purposes of theorem-proving alone, it does not have to be a good algorithm (quick, least storage used, etc.). It just has to work in every case.

Let us start by considering an abstract transition graph T. T may have many start states. We first want to simplify T so that it has only one start state that has no incoming edges. We do this by introducing a new state that we label with a minus sign and that we connect to all the previous start states by edges labeled with A. Then we drop the minus signs from the previous start states. Now all inputs must begin at the new unique start state. From there, they can proceed free of charge to any of the old start states. If the word w used to be ac-cepted by starting at previous start state 3 and proceeding through the machine to a final state, it can now be accepted by starting at the new unique start state and progressing to the old start state 3 along the edge labeled A. This trip does not use up any of the input letters. The word then picks up its old path and becomes accepted. This process is illustrated below on a fragment of a TG that has three start states: 1, 3, and 5:

Chapter 7, Problem 1P

12 Bookmarks

Show all steps: ON

The three sections of our proof will be:

Part 1 Every language that can be defined by a finite automaton can also be defined by a transition graph.

Part 2 Every language that can be defined by a transition graph can also be defined by a regular expression.

Part 3 Every language that can be defined by a regular expression can also be defined by a finite automaton.

When we have proven these three parts, we have finished our theorem.

Part 1

Proof of Part 1

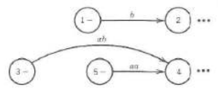
This is the easiest part. Every finite automaton is itself already a transition graph. Therefore, any language that has been defined by a finite automaton has already been defined by a transition graph. Done.

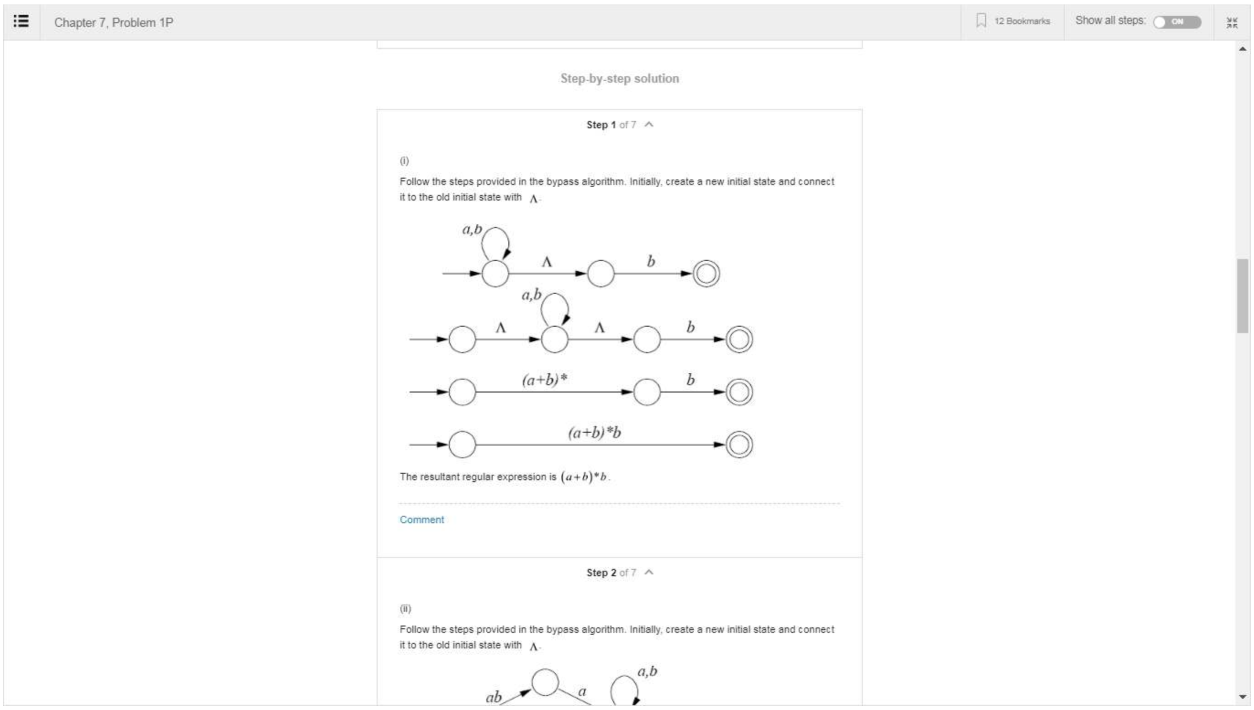
Part 2

Proof of Part 2

The proof of this part will be by constructive algorithm. This means that we present a procedure that starts out with a transition graph and ends up with a regular expression that defines the same language. To be acceptable as a method of proof, any algorithm must satisfy two criteria. It must work for every conceivable TG, and it must guarantee to finish its job in a finite time (a finite number of steps). For the purposes of theorem-proving alone, it does not have to be a good algorithm (quick, least storage used, etc.). It just has to work in every case.

Let us start by considering an abstract transition graph T. T may have many start states. We first want to simplify T so that it has only one start state that has no incoming edges. We do this by introducing a new state that we label with a minus sign and that we connect to all the previous start states by edges labeled with A. Then we drop the minus signs from the previous start states. Now all inputs must begin at the new unique start state. From there, they can proceed free of charge to any of the old start states. If the word w used to be ac-cepted by starting at previous start state 3 and proceeding through the machine to a final state, it can now be accepted by starting at the new unique start state and progressing to the old start state 3 along the edge labeled A. This trip does not use up any of the input letters. The word then picks up its old path and becomes accepted. This process is illustrated below on a fragment of a TG that has three start states: 1, 3, and 5:





Chapter 7, Problem 1P

12 Bookmarks Show all steps: ON

Step-by-step solution

Step 1 of 7

(i)

Follow the steps provided in the bypass algorithm. Initially, create a new initial state and connect it to the old initial state with Λ.

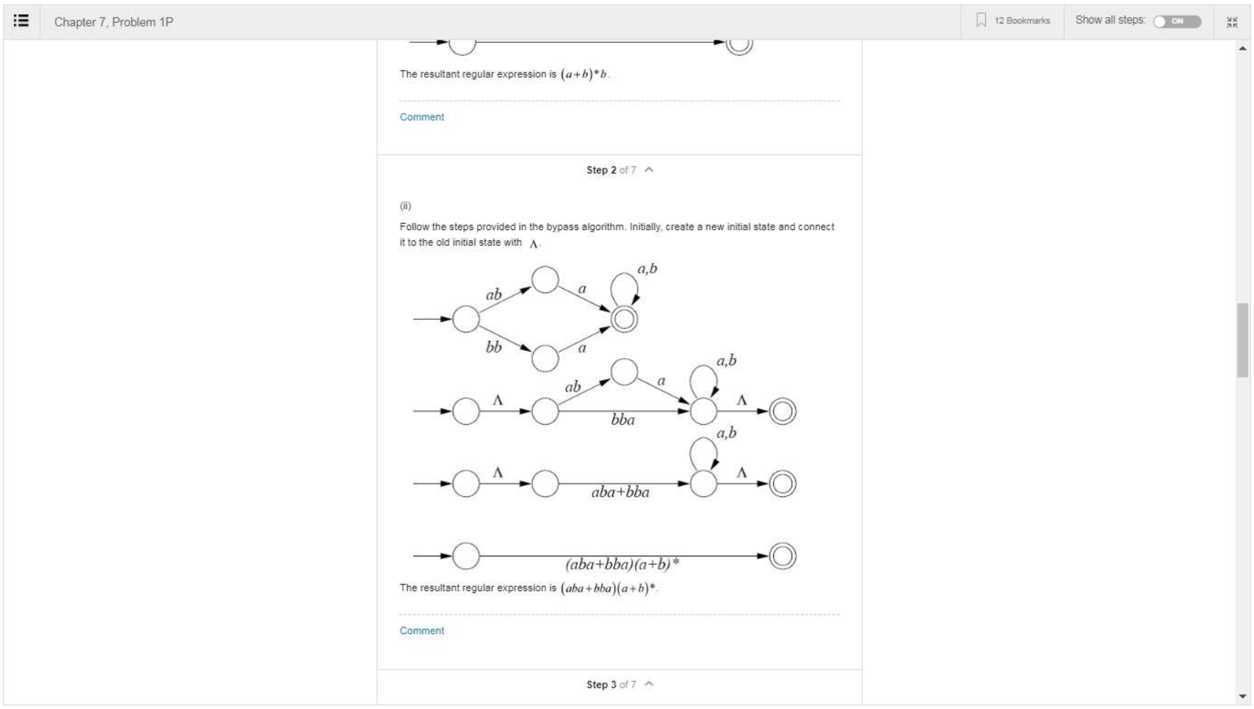
The resultant regular expression is $(a+b)^*b$.

Comment

Step 2 of 7

(ii)

Follow the steps provided in the bypass algorithm. Initially, create a new initial state and connect it to the old initial state with Λ.



Chapter 7, Problem 1P

12 Bookmarks Show all steps: ON

The resultant regular expression is $(a+b)^*b$.

Comment

Step 2 of 7

(ii)

Follow the steps provided in the bypass algorithm. Initially, create a new initial state and connect it to the old initial state with Λ.

The resultant regular expression is $(aba+bba)(a+b)^*$.

Comment

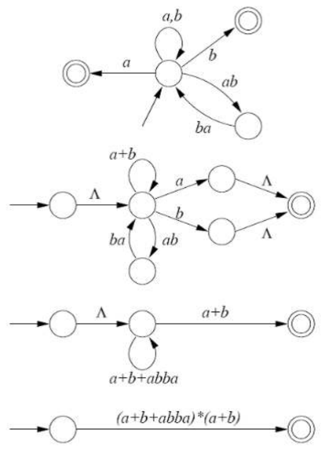
Step 3 of 7

Comment

Step 3 of 7

(iii)

Follow the steps provided in the bypass algorithm. Initially, create a new initial state and connect it to the old initial state with Λ.



The resultant regular expression is $(a+b+abba)^*(a+b)$.

Comment

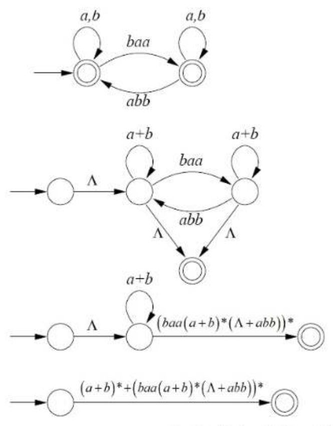
Step 4 of 7

(iv)

Follow the steps provided in the bypass algorithm. Initially, create a new initial state and connect it to the old initial state with Λ.

Comment

Step 5 of 7



The resultant regular expression is $(a+b)^*+(baa(a+b)^*(\Lambda+abb))^*$.

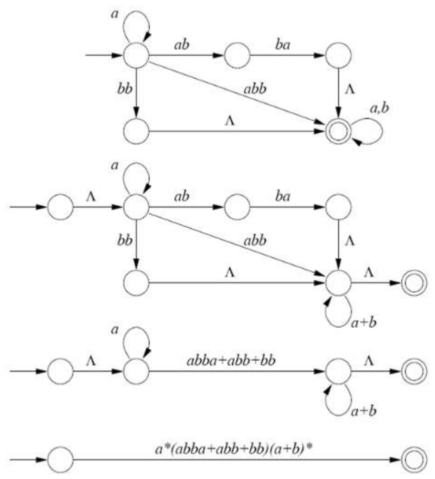
Chapter 7, Problem 1P

12 Bookmarks Show all steps: ON

Step 6 of 7

(v)

Follow the steps provided in the bypass algorithm. Initially, create a new initial state and connect it to the old initial state with Λ.



The resultant regular expression is $a^*(abba+abb+bb)(a+b)^*$.

Comment

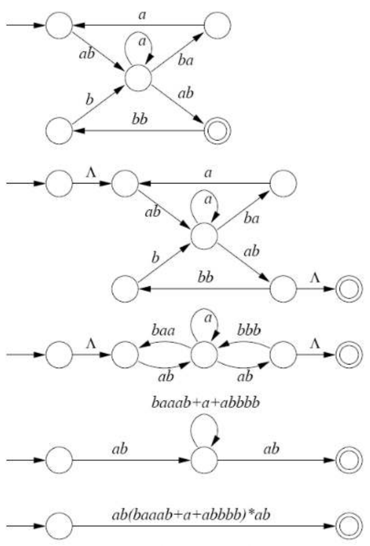
Chapter 7, Problem 1P

12 Bookmarks Show all steps: ON

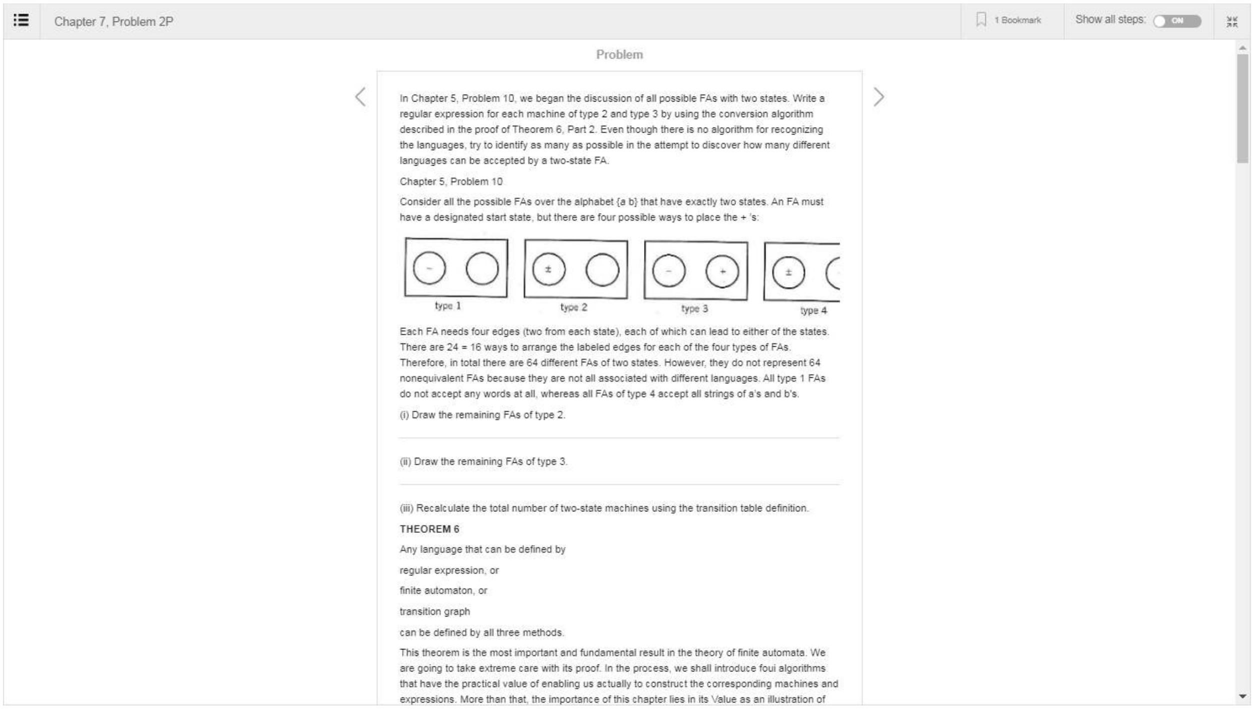
Step 7 of 7

(vi)

Follow the steps provided in the bypass algorithm. Initially, create a new initial state and connect it to the old initial state with Λ.



The resultant regular expression is $ab(baaab+a+abbbb)^*ab$.



Chapter 7, Problem 2P

1 Bookmark

Show all steps: ON

Problem

In Chapter 5, Problem 10, we began the discussion of all possible FAs with two states. Write a regular expression for each machine of type 2 and type 3 by using the conversion algorithm described in the proof of Theorem 6, Part 2. Even though there is no algorithm for recognizing the languages, try to identify as many as possible in the attempt to discover how many different languages can be accepted by a two-state FA.

Chapter 5, Problem 10

Consider all the possible FAs over the alphabet {a b} that have exactly two states. An FA must have a designated start state, but there are four possible ways to place the + 's:

Each FA needs four edges (two from each state), each of which can lead to either of the states. There are 24 = 16 ways to arrange the labeled edges for each of the four types of FAs. Therefore, in total there are 64 different FAs of two states. However, they do not represent 64 nonequivalent FAs because they are not all associated with different languages. All type 1 FAs do not accept any words at all, whereas all FAs of type 4 accept all strings of a's and b's.

(i) Draw the remaining FAs of type 2.

(ii) Draw the remaining FAs of type 3.

(iii) Recalculate the total number of two-state machines using the transition table definition.

THEOREM 6

Any language that can be defined by

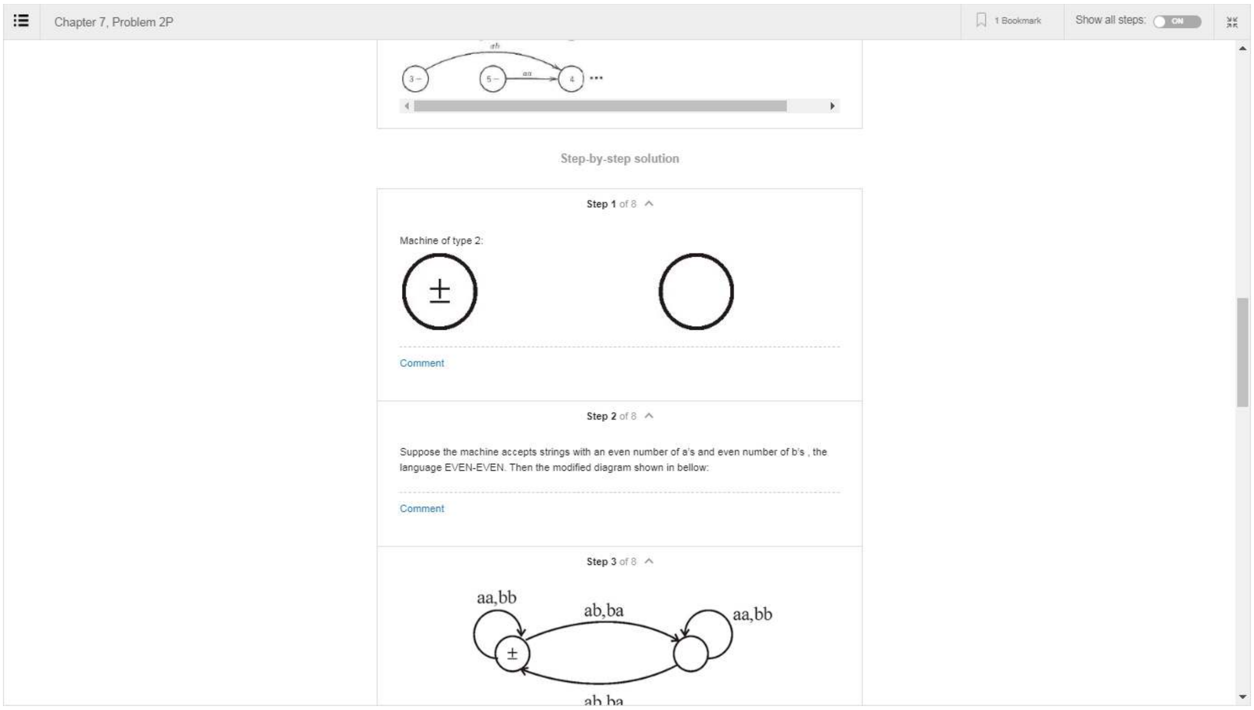
regular expression, or

finite automaton, or

transition graph

can be defined by all three methods.

This theorem is the most important and fundamental result in the theory of finite automata. We are going to take extreme care with its proof. In the process, we shall introduce foui algorithms that have the practical value of enabling us actually to construct the corresponding machines and expressions. More than that, the importance of this chapter lies in its Value as an illustration of



Chapter 7, Problem 2P

1 Bookmark

Show all steps: ON

Step-by-step solution

Step 1 of 8

Machine of type 2:

Comment

Step 2 of 8

Suppose the machine accepts strings with an even number of a's and even number of b's , the language EVEN-EVEN. Then the modified diagram shown in bellow:

Comment

Step 3 of 8

Chapter 7, Problem 2P

1 Bookmark

Show all steps: ON

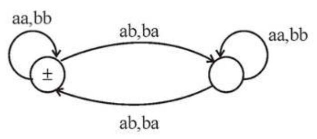
Comment

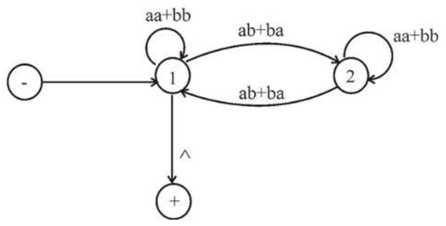
Step 2 of 8

Suppose the machine accepts strings with an even number of a's and even number of b's , the language EVEN-EVEN. Then the modified diagram shown in bellow:

Comment

Step 3 of 8





Comment

Chapter 7, Problem 2P

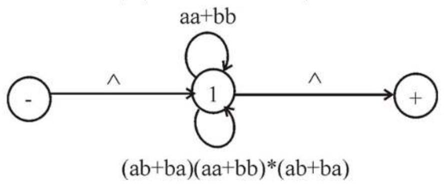
1 Bookmark

Show all steps: ON

Comment

Step 4 of 8

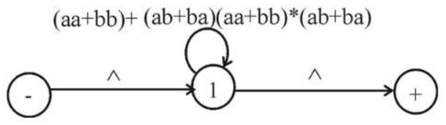
When we eliminate state2, the path from 1 to 2 to 1 becomes a loop at state1



Comment

Step 5 of 8

Which becomes



Comment

Chapter 7, Problem 2P

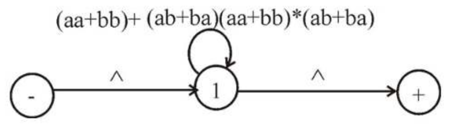
1 Bookmark Show all steps: ON

(ab+ba)(aa+bb)*(ab+ba)

Comment

Step 5 of 8

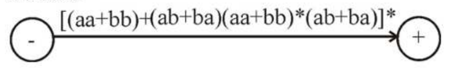
Which becomes



Comment

Step 6 of 8

Which becomes



Comment

Step 7 of 8

Which reduces to the regular expression:

Comment

Step 8 of 8

Chapter 7, Problem 2P

1 Bookmark Show all steps: ON

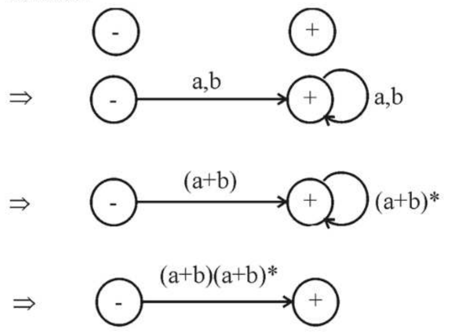
Step 7 of 8

Which reduces to the regular expression:

Comment

Step 8 of 8

Machine of type 3:



Comment

Was this solution helpful? 0 0