Lecture #

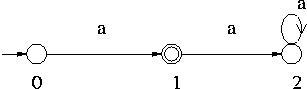
Module # \_1

Title: Designing finite automata

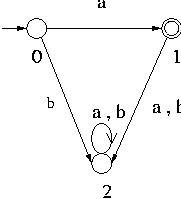
*Explanation:*

**Example 1:** Q = { 0, 1, 2 }, = { a }, A = { 1 }, the initial state is 0 and is as shown in the following table.

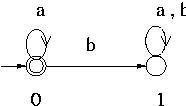
|  |  |  |
| --- | --- | --- |
| State (q) | Input (a) | Next State ( (q, a) ) |
| 0 | a | 1 |
| 1 | a | 2 |
| 2 | a | 2 |

A state transition diagram for this DFA is given below.   
  
             
If the alphabet of the Example 1 is changed to { a, b } in stead of { a }, then we need a DFA such as shown in the following example to accept the same string a. It is a little more complex DFA.   
  
**Example 2:** Q = { 0, 1, 2 }, = { a, b }, A = { 1 }, the initial state is 0 and is as shown in the following table.

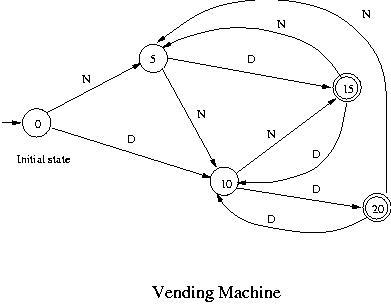
|  |  |  |
| --- | --- | --- |
| State (q) | Input (a) | Next State ( (q, a) ) |
| 0 | a | 1 |
| 0 | b | 2 |
| 1 | a | 2 |
| 1 | b | 2 |
| 2 | a | 2 |
| 2 | b | 2 |

Note that for each state there are two rows in the table for corresponding to the symbols a and b, while in the Example 1 there is only one row for each state.   
  
A state transition diagram for this DFA is given below.   
  
              
  
  
A DFA that accepts all strings consisting of only symbol a over the alphabet { a, b } is the next example.   
  
**Example 3:** Q = { 0, 1 }, = { a, b }, A = { 0 }, the initial state is 0 and is as shown in the following table.

|  |  |  |
| --- | --- | --- |
| State (q) | Input (a) | Next State ( (q, a) ) |
| 0 | a | 0 |
| 0 | b | 1 |
| 1 | a | 1 |
| 1 | b | 1 |

A state transition diagram for this DFA is given below.   
  
              
  
  
**Example 4:** For the example of vending machine of the previous section, Q = { 0, 5, 10, 15, 20 }, = { D, N }, A = { 15, 20 }, the initial state q0 = 0. If we make it a DFA, its transition function is as shown in the following table.

|  |  |  |
| --- | --- | --- |
| State (q) | Input (a) | Next State ( (q, a) ) |
| 0 | N | 5 |
| 0 | D | 10 |
| 5 | N | 10 |
| 5 | D | 15 |
| 10 | N | 15 |
| 10 | D | 20 |
| 15 | N | 5 |
| 15 | D | 10 |
| 20 | N | 5 |
| 20 | D | 10 |

              
Questions:

1. Find a DFA that accept the set consisting of all string.  
    a) with exactly one ‘a’ on an alphabet {a,b}  
    b) In which both number of 0’s and number of 1’s are even.

References:

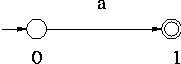
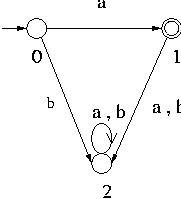
*Introduction to Automata Theory, Languages an Computation- John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman Page No-64*

Lecture # 4

Module # 1

Title: NFA

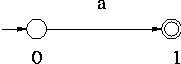
*Explanation:*

In the previous section we have seen DFAs that accept some simple languages such as , {} , and { a }. As you might have noticed, those DFAs have states and transitions which do not contribute to accepting strings and languages. For example all we need about an FA that accepts { a } is the following regardless of the alphabet (whether be it { a } , { a , b } or any other) .   
  
              
  
This is so to say the essence of such an FA. But it is not DFA. A DFA that accepts { a } would need more states and transitions as you can see below for example.   
  
             
Without those extra state and transitions it is not a DFA if the alphabet is { a , b } .   
To avoid those redundant states and transitions and to make modeling easier we use finite automata called nondeterministic finite automata (abbreviated as NFA) . Below we are going to formally define nondeterministic finite automata (abbreviated as NFS) and see some examples. As we are going to see later, for any NFA there is a DFA which accepts the same language and vice versa.   
  
NFAs are quite similar to DFAs. The only difference is in the transition function. NFAs do not necessarily go to a unique next state. An NFA may not go to any state from the current state on reading an input symbol or it may select one of several states nondeterministically (e.g. by throwing a die) as its next state.   
**Definition of nondeterministic finite automaton**   
  
Let Q be a finite set and let be a finite set of symbols. Also let be a function from Q to 2Q ,   let q0 be a state in Q and let A be a subset of Q. We call the elements of Q a **state**, the **transition function**, q0 the **initial state** and A the set of **accepting states**.   
Then a **nondeterministic finite automaton** is a 5-tuple < Q , , q0 , , A >   
  
**Notes on the definition**

1. As in the case of DFA the set Q in the above definition is simply a set with a finite number of elements. Its elements can be interpreted as a state that the system (automaton) is in.
2. The transition function is also called a **next state function** . Unlike DFAs an NFA moves into one of the states given by (q, a) if it receives the input symbol a while in state q. Which one of the states in (q, a) to select is determined nondeterministically.
3. Note that is a function. Thus for **each state** q of Q and for **each symbol** a of   (q, a) must be specified. But it can be the empty set, in which case the NFA aborts its operation.
4. As in the case of DFA the accepting states are used to distinguish sequences of inputs given to the finite automaton. If the finite automaton is in an accepting state when the input ends i.e. ceases to come, the sequence of input symbols given to the finite automaton is "accepted". Otherwise it is not accepted.
5. Note that any DFA is also a NFA.

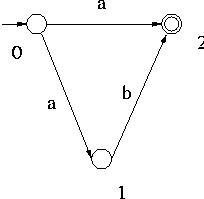
**Examples of NFA**   
**Example 1:** Q = { 0, 1 }, = { a }, A = { 1 }, the initial state is 0 and is as shown in the following table.

|  |  |  |
| --- | --- | --- |
| State (q) | Input (a) | Next State ( (q, a) ) |
| 0 | a | { 1 } |
| 1 | a |  |

A state transition diagram for this finite automaton is given below.   
  
  
  
              
  
  
  
If the alphabet is changed to { a, b } in stead of { a }, this is still an NFA that accepts { a } .

**Example 2:** Q = { 0, 1, 2 }, = { a, b }, A = { 2 }, the initial state is 0 and is as shown in the following table.

|  |  |  |
| --- | --- | --- |
| State (q) | Input (a) | Next State ( (q, a) ) |
| 0 | a | { 1 , 2 } |
| 0 | b |  |
| 1 | a |  |
| 1 | b | { 2 } |
| 2 | a |  |
| 2 | b |  |

Note that for each state there are two rows in the table for corresponding to the symbols a and b, while in the Example 1 there is only one row for each state.   
  
A state transition diagram for this finite automaton is given below.   
  
              
**Operation of NFA**   
  
Let us see how an automaton operates when some inputs are applied to it. As an example let us consider the automaton of Example 2 above.   
Initially it is in state 0. When it reads the symbol a, it moves to either state 1 or state 2. Since the state 2 is the accepting state, if it moves to state 2 and no more inputs are given, then it stays in the accepting state. We say that this automaton accepts the string a. If on the other hand it moves to state 1 after reading a, if the next input is b and if no more inputs are given, then it goes to state 2 and remains there. Thus the string ab is also accepted by this NFA. If any other strings are given to this NFA, it does not accept any of them.   
  
Let us now define the function \* and then formalize the concepts of acceptance of strings and languages by NFA.

Questions:

1. Find an NFA for the following language.  
   L= {x Є {0 , 1 }\* : the second symbol from the right is 1}
2. What is non deterministic finite state automaton? Give its formal definition.

References:

*Introduction to Automata Theory, Languages an Computation- John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman Page No-69*

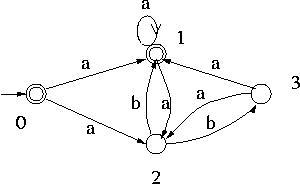
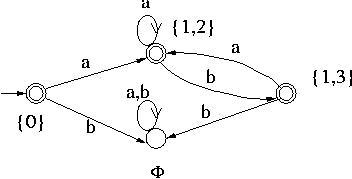
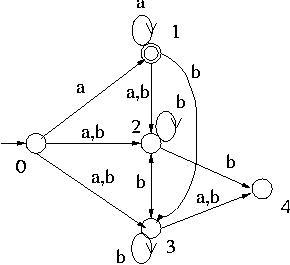
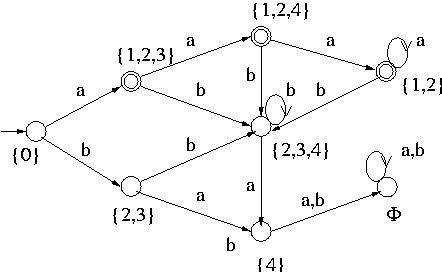
Lecture #5

Module # 1

Title: *Equivalence of NFA and DFA.*

*Explanation:*

**Conversion of NFA to DFA**   
  
Let M2 = < Q2 , , q2,0 , 2 , A2 > be an NFA that recognizes a language L. Then the DFA M = < Q, , q0 , , A > that satisfies the following conditions recognizes L:   
  
**Q** = 2Q2 , that is the set of all subsets of Q2 ,   
q0 = { q2,0 } ,

( q, a ) = for each state q in **Q** and each symbol a in   and   
A = { q є Q | q A2 ≠ Ф }   
  
To obtain a DFA M = < Q, , q0 , , A > which accepts the same language as the given NFA M2 = < Q2 , , q2,0 , 2 , A2 > does, you may proceed as follows:   
  
Initially **Q** = Ф.   
First put { q2,0 } into **Q**. { q2,0 } is the initial state of the DFA M.   
Then for each state q in **Q** do the following:   
add the set , where here is that of NFA M2, as a state to **Q** if it is not already in **Q** for each symbol a in .   
For this new state, add ( q, a ) = to , where the on the right hand side is that of NFA M2.   
  
When no more new states can be added to **Q**, the process terminates. All the states of **Q** that contain accepting states of M2 are accepting states of M.   
  
  
**Example 1:** Let us convert the following NFA to DFA.   
  
              
  
Initially Q is empty. Then since the initial state of the DFA is {0} , {0} is added to Q.   
Since 2( 0 , a ) = { 1 , 2 } , { 1 , 2 } is added to **Q** and ( { 0 } , a ) = { 1 , 2 } .   
Since 2( 0 , b ) =Ф, Ф is added to **Q** and ( { 0 } , b ) = Ф.   
At this point Q = { {0} , { 1 , 2 }, Ф } .   
Then since { 1 , 2 } is now in **Q**, the transitions from { 1 , 2 } on symbols a and b are computed. Since 2( 1 , a ) = { 1 , 2 } , and 2( 2 , a ) = Ф, ( { 1 , 2 } , a ) = { 1 , 2 } . Similarly ( { 1 , 2 } , b ) = { 1 , 3 } . Thus { 1 , 3 } is added to **Q** .   
Similarly ( { 1 , 3 } , a ) = { 1 , 2 } and ( { 1 , 3 } , b ) = Ф. Thus no new states are added to **Q** . Since the transitions from all states of **Q** have been computed and no more states are added to **Q**, the conversion process stops here.   
  
Note that there are no states of Q2 in Ф. Hence there are no states that M2 can go to from Ф. Hence ( Ф, a ) = ( Ф, b ) = Ф.   
  
For the accepting states of M, since states 0 and 1 are the accepting states of the NFA, all the states of **Q** that contain 0 and/or 1 are accepting states. Hence { 0 }, { 1 , 2 } and { 1 , 3 } are the accepting states of M.   
  
The DFA thus obtained is shown below.   
  
  
              
**Example 2:** Similarly the NFA   
  
              
  
is converted to the following DFA:   
  
  
   

Questions:

1. Convert an NFA into an equivalent DFA(may 2007- 12a)
2. State and prove the equivalence of NFA and DFA.

References:

*Introduction to Automata Theory, Languages an Computation- John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman Page No-74*

Lecture #6

Module # 1

Title: *Finite automata with epsilon moves*

*Explanation:*

One of the objectives of this chapter is to show that there is a one-to-one correspondence between regular languages and finite automata. We are going to do that by showing that a finite automaton can be constructed from a given regular expression by combining simpler FAs using union, concatenation and Kleene star operations. These operations on FAs can be described conveniently if -Transitions are used. Basically an NFA with -Transitions is an NFA but can respond to an empty string and move to the next state. Here we are going to formally define NFA with -Transitions (abbreviated as NFA-) and see some examples. As we are going to see later, for any NFA- there is a NFA (hence DFA) which accepts the same language and vice versa.   
  
**Definition of nondeterministic finite automaton with -Transitions**   
  
Let Q be a finite set and let be a finite set of symbols. Also let be a function from Q {} to 2Q ,   let q0 be a state in Q and let A be a subset of Q. We call the elements of Q a **state**, the **transition function**, q0 the **initial state** and A the set of **accepting states**.   
Then a **nondeterministic finite automaton with -Transitions** is a 5-tuple < Q , , q0 , , A >

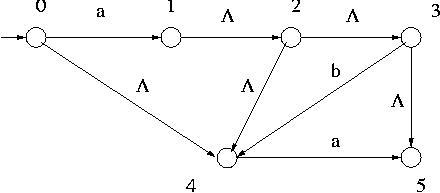
**Notes on the definition**

1. A transition on reading means that the NFA- makes the transition without reading any symbol in the input. Thus the tape head does not move when is read.
2. Note that any NFA is also a NFA-.

**Example of NFA-**

Q = { 0, 1, 2, 3, 4, 5 }, = { a, b }, A = , the initial state is 0 and is as shown in the following table.

|  |  |  |
| --- | --- | --- |
| State (q) | Input (a) | Next State ( (q, a) ) |
| 0 | a | { 1 } |
| 0 |  | { 4 } |
| 1 |  | { 2 } |
| 2 |  | { 3, 4 } |
| 3 |  | { 5 } |
| 3 | b | { 4 } |
| 4 | a | { 5 } |

Here the transitions to Ф are omitted from the table. A state transition diagram for this finite automaton is given below.   
  
  
              
  
  
When a symbol a is read at the initial state 0, for example, it can move to any of the states other than 0. For once you are in state 1, for example, you can go to state 2, 3, 4 and 5 without reading any symbol on the tape. If you read string ab, then you come to state 4. For though you go to states 1, 2, 3, 4 and 5 by reading a, there are no transitions on reading b except from state 3. Thus 4 is the only state you can go to from the initial state by reading ab.   
Questions:

1. Does e-transitions enhance the power of NFA? Explain.

References:

*Introduction to Automata Theory, Languages an Computation- John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman Page No-86*

Lecture #7

Module # 1

Title: *Regular sets and Regular grammars.*

*Explanation:*

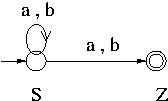
We have learned three ways of characterising regular languages: regular expressions, finite automata and construction from simple languages using simple operations. There is yet another way of characterizing them, that is by something called grammar. A grammar is a set of rewrite rules which are used to generarte strings by successively rewriting symbols. For example consider the language represented by a+, which is { a, aa, aaa, . . . } . One can generate the strings of this language by the following procedure: Let S be a symbol to start the process with. Rewrite S using one of the following two rules: S -> a , and S -> aS . These rules mean that S is rewritten as a or as aS. To generate the string aa for example, start with S and apply the second rule to replace S with the right hand side of the rule, i.e. aS, to obtain aS. Then apply the first rule to aS to rewrite S as a. That gives us aa. We write S => aS to express that aS is obtained from S by applying a single production. Thus the process of obtaining aa from S is written as S => aS => aa . If we are not interested in the intermediate steps, the fact that aa is obtained from S is written as S =>\* aa , In general if a string β is obtained from a string by applying productions of a grammar G, we write **=>\*G** βand say that β **is derived from **. If there is no ambiguity about the grammar G that is referred to, then we simply write **=>\*** β  
Formally a **grammar** consists of a set of nonterminals (or variables) V, a set of terminals (the alphabet of the language), a start symbol S, which ia a nonterminal, and a set of rewrite rules (productions) P. A production has in general the form γ -> , where γis a string of terminals and nonterminals with at least one nonterminal in it and is a string of terminals and nonterminals. A **grammar is regular** if and only if γ is a single nonterminal and is a single terminal or a single terminal followed by a single nonterminal, that is a production is of the form X -> a or X -> aY, where X and Y are nonterminals and a is a terminal.

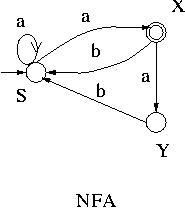
For example, = {a, b}, V = { S } and P = { S -> aS, S -> bS, S -> } is a regular grammar and it generates all the strings consisting of a's and b's including the empty string.

The following theorem holds for regular grammars.

**Theorem 3:** A language L is accepted by an FA i.e. regular, if L - {} can be generated by a regular grammar.

This can be proven by constructing an FA for the given grammar as follows: For each nonterminal create a state. S corresponds to the initial state. Add another state as the accepting state Z. Then for every production X -> aY, add the transition ( X, a ) = Y and for every production X -> a add the transition ( X, a ) = Z.   
  
For example = {a, b}, V = { S } and P = { S -> aS, S -> bS, S -> a, S -> b } form a regular grammar which generates the language ( a + b )+. An NFA that recognizes this language can be obtained by creating two states S and Z, and adding transitions ( S, a ) = { S, Z } and ( S, b ) = { S, Z } , where S is the initial state and Z is the accepting state of the NFA.

The NFA thus obtained is shown below.   
  
              
  
Thus L - {} is regular. If L contains as its member, then since { } is regular , L = ( L -{ } ) {} is also regular.

Conversely from any NFA < Q, , , q0, A > a regular grammar < Q, , P, q0 > is obtained as follows:   
For any a in , and nonterminals X and Y, X -> aY is in P if and only if (X, a) = Y , and for any a in and any nonterminal X, X -> a is in P if and only if (X, a) = Y for some accepting state Y.   
  
Thus the following converse of Theorem 3 is obtained.   
  
**Theorem 4 :** If L is regular i.e. accepted by an NFA, then L - {} is generated by a regular grammar.   
  
For example, a regular grammar corresponding to the NFA given below is < Q, { a, b }, P, S > , where Q = { S, X, Y } , P = { S -> aS, S -> aX, X -> bS, X -> aY, Y -> bS, S -> a } .   
            

In addition to regular languages there are three other types of languages in [**Chomsky hierarchy**](http://www.cs.odu.edu/~toida/nerzic/390teched/language/intro.html) : context-free languages, context-sensitive languages and phrase structure languages. They are characterized by context-free grammars, context-sensitive grammars and phrase structure grammars, respectively.

These grammars are distinguished by the kind of productions they have but they also form a hierarchy, that is the set of regular languages is a subset of the set of context-free languages which is in turn a subset of the set of context-sensitive languages and the set of context-sensitive languages is a subset of the set of phrase structure languages.   
A grammar is a **context-free grammar** if and only if its production is of the form X -> , where is a string of terminals and nonterminals, possibly the empty string.   
For example P = { S -> aSb, S -> ab } with = { a, b } and V = { S } is a contex-free grammar and it generates the language { anbn | n is a positive integer } . As we shall see later this is an example of context-free language which is not regular.   
  
A grammar is a **context-sensitive grammar** if and only if its production is of the form

1X2 -> 1 β2, where X is a nonterminal and 1 , 2 and β are strings of terminals and nonterminals, possibly empty except β.

Thus the nonterminal X can be rewritten as β only in the context of 1X2 .   
For example P = { S -> XYZS1, S -> XYZ, S1 -> XYZS1, S1 -> XYZ, YX -> XY, ZX -> XZ, ZY -> YZ, X -> a, aX -> aa, aY -> ab, BY -> bb, bZ -> bc, cZ -> cc } with = { a, b, c } and V = { X, Y, Z, S, S1 } is a context-sensitive grammar and it generates the language { anbncn | n is a positive integer } . It is an example of context-sensitive language which is not context-free.

Context-sensitive grammars are also characterized by productions whose left hand side is not longer than the right hand side, that is, for every production -> β, || ≤ | β | .   
  
For a **phrase structure grammar**, there is no restriction on the form of production, that is a production of a phrase structure grammar can take the form -> β, where and β can be any string.

Questions:

*All questions related to the topics introduced including previous year University questions*

References:

*Reference book & page nos*

Lecture #8

Module # 1

Title: *Regular Expressions.*

*Explanation:*

**Regular expression**   
  
Regular expressions are used to denote regular languages. They can represent regular languages and operations on them succinctly.   
The set of regular expressions over an alphabet is defined recursively as below. Any element of that set is a **regular expression**.

**Ex. 1:** Find the shortest string that is not in the language represented by the regular expression **a\*(ab)\*b\***.   
  
**Solution:** It can easily be seen that , a, b, which are strings in the language with length 1 or less. Of the strings with length 2 aa, bb and ab are in the language. However, ba is not in it. Thus the answer is ba.   
  
  
**Ex. 2:** For the two regular expressions given below,   
(a) find a string corresponding to **r2** but not to **r1** and   
(b) find a string corresponding to both **r1** and **r2**.   
  
        **r1 = a\* + b\***     **r2 = ab\* + ba\* + b\*a + (a\*b)\***   
  
**Solution:** (a) Any string consisting of only a's or only b's and the empty string are in **r1**. So we need to find strings of **r2** which contain at least one a and at least one b. For example ab and ba are such strings.   
(b) A string corresponding to **r1** consists of only a's or only b's or the empty string. The only strings corresponding to **r2** which consist of only a's or b's are a, b and the strings consiting of only b's (from **(a\*b)\***).   
  
  
**Ex. 3:** Let **r1** and **r2** be arbitrary regular expressions over some alphabet. Find a simple (the shortest and with the smallest nesting of \* and +) regular expression which is equal to each of the following regular expressions.   
  
(a) **(r1 + r2 + r1r2 + r2r1)\***   
(b) **(r1(r1 + r2)\*)+**   
  
**Solution:** One general strategy to approach this type of question is to try to see whether or not they are equal to simple regular expressions that are familiar to us such as **a**, **a\***, **a+**, **(a + b)\***, **(a + b)+** etc.   
(a) Since **(r1 + r2)\*** represents all strings consisting of strings of **r1** and/or **r2** , **r1r2 + r2r1** in the given regular expression is redundant, that is, they do not produce any strings that are not represented by **(r1 + r2)\***. Thus **(r1 + r2 + r1r2 + r2r1)\*** is reduced to **(r1 + r2)\***.   
(b) **(r1(r1 + r2)\*)+** means that all the strings represented by it must consist of one or more strings of **(r1(r1 + r2)\*)**. However, the strings of **(r1(r1 + r2)\*)** start with a string of **r1** followed by any number of strings taken arbitrarily from **r1** and/or **r2**. Thus anything that comes after the first **r1** in **(r1(r1 + r2)\*)+** is represented by **(r1 + r2)\***. Hence **(r1(r1 + r2)\*)** also represents the strings of **(r1(r1 + r2)\*)+**, and conversely **(r1(r1 + r2)\*)+** represents the strings represented by **(r1(r1 + r2)\*)**. Hence **(r1(r1 + r2)\*)+** is reduced to **(r1(r1 + r2)\*)**.   
  
  
**Ex. 4:** Find a regular expression corresponding to the language *L* over the alphabet { a , b } defined recursively as follows:   
  
Basis Clause:  є *L*   
Inductive Clause: If x є *L* , then aabx є *L* and xbb є *L* .   
Extremal Clause: Nothing is in *L* unless it can be obtained from the above two clauses.   
  
**Solution:** Let us see what kind of strings are in *L*. First of all  є *L* . Then starting with , strings of *L* are generated one by one by prepending aab or appending bb to any of the already generated strings. Hence a string of *L* consists of zero or more aab's in front and zero or more bb's following them. Thus **(aab)\*(bb)\*** is a regular expression for *L*.   
  
  
**Ex. 5:** Find a regular expression corresponding to the language *L* defined recursively as follows:   
  
Basis Clause:  є *L* and a є *L* .   
Inductive Clause: If x є *L* , then aabx є *L* and bbx є *L* .   
Extremal Clause: Nothing is in *L* unless it can be obtained from the above two clauses.   
  
**Solution:** Let us see what kind of strings are in *L*. First of all and a are in *L* . Then starting with or a, strings of *L* are generated one by one by prepending aab or bb to any of the already generated strings. Hence a string of *L* has zero or more of aab's and bb's in front possibly followed by a at the end. Thus **(aab + bb)\*(a + )** is a regular expression for *L*.   
  
  
**Ex. 6:** Find a regular expression corresponding to the language of all strings over the alphabet { a, b } that contain exactly two a's.   
  
**Solution:** A string in this language must have at least two a's. Since any string of b's can be placed in front of the first a, behind the second a and between the two a's, and since an arbitrasry string of b's can be represented by the regular expression **b\***, **b\***a **b\***a **b\*** is a regular expression for this language.   
  
  
**Ex. 7:** Find a regular expression corresponding to the language of all strings over the alphabet { a, b } that do not end with ab.   
  
**Solution:** Any string in a language over { a , b } must end in a or b. Hence if a string does not end with ab then it ends with a or if it ends with b the last b must be preceded by a symbol b. Since it can have any string in front of the last a or bb, **( a + b )\*( a + bb )** is a regular expression for the language.   
  
  
**Ex. 8:** Find a regular expression corresponding to the language of all strings over the alphabet { a, b } that contain no more than one occurence of the string aa.   
  
**Solution:** If there is one substring aa in a string of the language, then that aa can be followed by any number of b. If an a comes after that aa, then that a must be preceded by b because otherwise there are two occurences of aa. Hence any string that follows aa is represented by **( b + ba )\***. On the other hand if an a precedes the aa, then it must be followed by b. Hence a string preceding the aa can be represented by **( b + ab )\***. Hence if a string of the language contains aa then it corresponds to the regular expression **( b + ab )\*aa( b + ba )\*** .   
If there is no aa but at least one a exists in a string of the language, then applying the same argument as for aa to a, **( b + ab )\*a( b + ba )\*** is obtained as a regular expression corresponding to such strings.   
If there may not be any a in a string of the language, then applying the same argument as for aa to , **( b + ab )\*( b + ba )\*** is obtained as a regular expression corresponding to such strings.   
Altogether **( b + ab )\*( + a + aa )( b + ba )\*** is a regular expression for the language.   
  
  
**Ex. 9:** Find a regular expression corresponding to the language of strings of even lengths over the alphabet of { a, b }.   
  
**Solution:** Since any string of even length can be expressed as the concatenation of strings of length 2 and since the strings of length 2 are aa, ab, ba, bb, a regular expression corresponding to the language is **( aa + ab + ba + bb )\***. Note that 0 is an even number. Hence the string is in this language.   
  
  
**Ex. 10:** Describe as simply as possible in English the language corresponding to the regular expression **a\*b(a\*ba\*b)\*a\*** .   
  
**Solution:** A string in the language can start and end with a or b, it has at least one b, and after the first b all the b's in the string appear in pairs. Any numbe of a's can appear any place in the string. Thus simply put, it is the set of strings over the alphabet { a, b } that contain an odd number of b's   
  
  
**Ex. 11:** Describe as simply as possible in English the language corresponding to the regular expression **(( a + b )3)\*( + a + b )** .   
  
**Solution:** **(( a + b )3)** represents the strings of length 3. Hence **(( a + b )3)\*** represents the strings of length a multiple of 3. Since **(( a + b )3)\*( a + b )** represents the strings of length 3n + 1, where n is a natural number, the given regular expression represents the strings of length 3n and 3n + 1, where n is a natural number.   
  
  
**Ex. 12:** Describe as simply as possible in English the language corresponding to the regular expression **( b + ab )\*( a + ab )\***.   
  
**Solution:** **( b + ab )\*** represents strings which do not contain any substring aa and which end in b, and **( a + ab )\*** represents strings which do not contain any substring bb. Hence altogether it represents any string consisting of a substring with no aa followed by one b followed by a substring with no bb.   
  
Questions:

1. Give the full definition of Regular expressions.
2. Construct a finite state automata equivalent to the regular expression 10+ (0+11) 0 \* 1.

References:

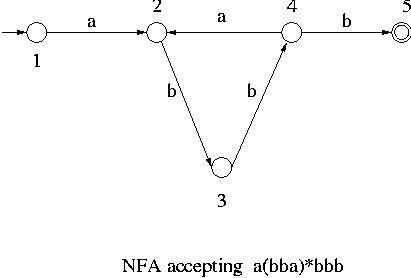
*Introduction to Automata Theory, Languages an Computation- John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman Page No-97*

Lecture #9

Module # \_1

Title: *Pumping Lemma for regular languages.*

*Explanation:*

**Pumping Lemma**   
  
Let us consider the NFA given below.   
  
  
  
              
  
  
This NFA accepts among others some strings of length greater than 5 such as abbabbb, abbabbabbb etc. Those strings which are accepted by this NFA and whose length is greater than 5 have a substring which can be repeated any number of times without being rejected by the NFA. For example the string abbabbb is accepted by the NFA and if one of its substrings bba is repeated any number of times in a***bba***bbb, the resultant strings such as abbb (bba repeated 0 times), abbabbabbb, abbabbabbabbb etc. are also accepted by the NFA.   
In general if a string w (such as abbabbb in the example above) is accepted by an NFA with *n* states and if its length is longer than *n*, then there must be a cycle in the NFA along some path from the initial state to some accepting state (such as the cycle 2-3-4-2 in the above example). Then the substring representing that cycle (bba in the example) can be repeated any number of times within the string w without being rejected by the NFA.   
The following theorem which is called Pumping Lemma is based on this observation. It states that if a language is regular, then any long enough string of the language has a substring which can be repeated any number of times with the resultant strings still in the language. It is stated without a proof here.   
  
**Pumping Lemma :** Suppose that a language L is regular. Then there is a positive integer n such that for any string x in L with |x| ≥ n, there are strings u, v and w which satisfy the following relationships:   
            x = uvw   
            |uv| ≤ n   
            |v| > 0   and   
            for every integer m ≥0, uvmw є L.   
  
Note that Pumping Lemma gives a necessity for regular languages and that it is not a sufficiency, that is, even if there is an integer n that satisfies the conditions of Pumping Lemma, the language is not necessarily regular. Thus Pumping Lemma can not be used to prove the regularity of a language. It can only show that a language is nonregular.   
  
**Example:** As an example to illustrate how Pumping Lemma might be used to prove that a language is nonregular, let us prove that the language L = akbk is nonregular, where k is a natural number.   
  
Suppose that L is regular. Then there is a positive integer n that satisfies the conditions of Pumping Lemma. Consider a string x = anbn for that n. Then there must be strings u, v, and w such that   
  
            x = uvw,   
            |uv| ≤ n   
            |v| > 0 , and   
            for every m ≥ 0, uvmw є L.   
  
Since |v| > 0 , v has at least one symbol. In string x = anbn, v can be inside an , inside bn or straddle an and bn .  
Hence there are the following three possibilities for v:   
        1. v = ap, p > 0 ,   
        2. v = bp, p > 0     or   
        3. v = apbq , p, q > 0 .   
Let us now consider the string uvmw for m = 2 for each of these three cases.   
In case 1, uv2w = an-pa2pbn = an+pbn . Since p > 0 , n + p ≠ n . Hence an+pbn can not be in the language L represented by akbk .   
Similarly in case 2, since uv2w = an bn + p , it can not be in L.   
In case 3, uv2w = an-p(apbq)2bn-q = an-p(apbqapbq)bn-q = an bqapbn , which can not be in L.   
Thus in all three cases uv2w is not in L. This violates the condition that for every m 0, uvmw L. Hence L is not a regular language.

Questions:

1. Show that L= { ap / p is a prime } is not regular.
2. Show that L= { anbn / n>0 } is not regular.
3. State and prove pumping lemma for regular languages.

*All questions related to the topics introduced including previous year University questions*

References:

*Introduction to Automata Theory, Languages an Computation- John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman Page No-140*

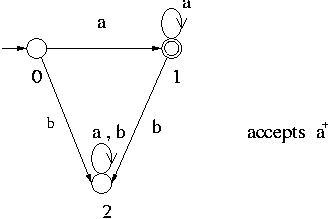
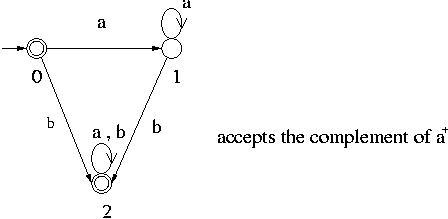
Lecture #10

Module # \_1

Title: *Closure properties of regular languages.*

*Explanation:*

We say a set of languages is **closed** under an operation if the result of applying the operation to any arbitrary language(s) of the set is a language in the set.   
For example a set of languages is closed under union if the union of any two languages of the set also belongs to the set.   
The following theorem is immediate from the Inductive Clause of the definition of the set of regular languages.   
  
**Theorem 1:** The set of regular languages over an alphabet is closed under operations union, concatenation and Kleene star.   
**Proof:** Let ***Lr*** and ***Ls*** be regular languages over an alphabet . Then by the definition of the set of regular languages , ***Lr*** ***Ls*** , ***LrLs*** and ***Lr*\*** are regular languages and they are obviously over the alphabet . Thus the set of regular languages is closed under those operations.   
  
**Note 1:** Later we shall see that the complement of a regular language and the intersection of regular laguages are also regular.   
**Note 2:** The union of infinitely many regular languages is not necessarily regular. For example while { akbk } is regular for any natural number k **, { anbn | n is a natural number } which is the union of all the languages { akbk } , is not regular** as we shall see later.   
  
The following theorem shows that any finite language is regular. We say a language is finite if it consists of a finite number of strings, that is, a **finite language** is a set of n strings for some natural number n.   
  
**Theorem 2:** A finite language is regular.   
  
**Proof:** Let us first assume that a language consisting of a single string is regular and prove the theorem by induction. We then prove that a language consisting of a single string is regular.   
  
**Claim 1:** A language consisting of n strings is regular for any natural number n (that is, a finite language is regular) if { w } is regular for any string w.   
  
**Proof of the Claim 1:** Proof by induction on the number of strings.   
  
**Basis Step:** Ф (corresponding to n = 0) is a regular language by the Basis Clause of the definition of regular language.   
**Inductive Step:** Assume that a language *L* consisting of n strings is a regular language (induction hypothesis). Then since { w } is a regular language as proven below, *L* { w } is a regular language by the definition of regular language.   
  
**End of proof of Claim 1**   
  
Thus if we can show that { w } is a regular language for any string w, then we have proven the theorem.   
  
**Claim 2:** Let w be a string over an alphabet . Then { w } is a regular language.   
  
**Proof of Claim 2:** Proof by induction on strings.   
  
**Basis Step:** By the Basis Clause of the definition of regular language, {} and { a } are regular languages for any arbitrary symbol a of .   
  
**Inductive Step:** Assume that { w } is a regular language for an arbitrary string w over . Then for any symbol a of , { a } is a regular language from the Basis Step. Hence by the Inductive Clause of the definition of regular language { a }{ w } is regular. Hence { aw } is regular.   
  
**End of proof for Claim 2**   
  
Note that Claim 2 can also be proven by induction on the length of string.   
  
**End of proof of Theorem 2**.

**Complement**   
Let M = < Q , , q0 , , A > be a DFA that accepts a language L. Then a DFA that accepts the complement of L, i.e. \* - L, can be obtained by swapping its accepting states with its non-accepting states, that is Mc = < Q , , q0 , , Q - A > is a DFA that accepts \* - L .   
For example the following DFA accepts the language a+ over = { a , b }.   
              
  
A DFA that accepts its complement is obtained from the above DFA by changing all single circles to double circles and vice versa as shown below.   
  
              
  
  
  
**Remark 1:** If we have NFA rather than DFA, we must first convert it to DFA before swapping states to get its complement.   
  
**Remark 2:** Since a language is regular if and only if it is accepted by some NFA, **the complement of a regular language is also regular.**   
  
  
**Intersection of Regular Languages**   
  
Langauges are sets. Therefore all the properties of sets are inherited by languages. In particular De Morgan's law also applies to languages. By Remark 2 above, if ***L***1 and ***L***2 are regular languages, then their complements are regular languages. Since ***L***1 ***L***2 = by De Morgan's law, ***L***1 ***L***2 is regular.   
  
Thus summing all this up we can say that the set of regular languages over an alphabet is closed with respect to union, intersection, difference, concatenation and Kleene star operations.

Questions:

1. Show that regular sets are closed under intersection.
2. Show that regular sets are closed under complementation and union.

References:

*Introduction to Automata Theory, Languages an Computation- John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman Page No-145*

Lecture #11

Module # \_1

Title: *Applications of finite automata..*

*Explanation:*

Applications

• software for finding patterns in large bodies of text (such as collection of web pages)

• software for designing circuits

• lexical analyser of compilers

• model-checking, modelling and verification of embedded systems, software model-checking

• model of real machines: watches, telephone, lock for cars, . . .

• application in genetics, regular pattern in the language of protein

• application in linguistic, building of large dictionnary, spell programs, search

Questions:

1. Write and explain any 2 applications of finite state automata.

References:

*Introduction to Automata Theory, Languages an Computation- John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman Page No-82*

Lecture #12

Module # \_1

Title: Decision algorithms for regular sets*.*

*Explanation:*

At this point we would like to find out answers to some important questions related to regular languages. The questions we consider here all have answers which may be either ?yes? or ?no?. These are known as decision problems since we used to decide whether the answer is ?yes? or ?no?. [The reason for considering decision problems is that a regular language is recognized by a FA, which, in response to an input string, either ?accepts' or ?rejects' the input string and can be considered as producing ?yes? or no ?answers?, respectively.]

Consider the following typical and important question:

***w* and a regular language *L* , is an element of *L* ?**

The answer is either yes or no.

While *w* is represent explicitly, we wonder how *L* given to us. Obviously, *L* cannot be given as an enumeration of strings (*L* may be infinite). *L* will be represented either by a *DFA* , *NFA* or regular expression.

The question presented above is called the membership problem for the corresponding regular language *L*.

If *L* is represented by a *DFA* , the problem has an easy solution-

* **Simulate the *DFA* on input *w***
* **If the DFA ends in an accepting state, the answer is yes. Otherwise, the answer is no.**

If *L* is given as an *NFA* , we can first convert it to an equivalent *DFA* and than use the above algorithm to find the answer. This is not efficient, since the conversion algorithm from *NFA* to *DFA* (by using subset constructions) is expensive.

Similarly, if *L* is expressed by using a regular expression, we can first convert it to an *NFA* and than use the above algorithm. We see that this is also an expensive method.

We will consider some more decision problems related to regular languages as given below.

* **Given a *FA M* , is *L(M)* empty?**
* **Given a *FAM*, is *L(M)* infinite?**
* **Given two *FA* s *M1* and M2 , do they accept the same language? That is, whether *L(M1)=L(M2)*?**

It is interesting to note that we can use the pumping lemma to determine whether the language accepted by a *DFA* is empty or infinite. The following theorem states this result.

Theorem : If *M* is a *DFA* with *n* states, then the language accepted by *M* (i.e. *L(M)*) is

1. non empty if, and only if, *M* accepts some string *w* with 
2. infinite if, and only if, *M* accepts some string *w* such that 

Proof:

1. If *M* accepts a string *w* with |w| < n, then *L(M)* is clearly non empty. Conversely, let *L(M)* be non empty, and let *w* be the shortest string accepted by *M*. Then it must be the case that |w| < n. Otherwise , according to the pumping lemma *w* can be decomposed as *w=xyz* satisfying all the three constraints of the pumping lemma. So,

For the case *i=0*, the string is a string which is shorter than *w* (since y ≠ є )

This contradicts that *w* is the shortest string accepted by *M*. Hence, |w| < n.

1. Let *M* accept a string *w* with . Then by pumping lemma *w* can be decomposed as *w=xyz* satisfying all the three constraints of the pumping lemma. Hence



Therefore, *L(M)* must be infinite.

Conversely, let *L(M)* be infinite, and let *w* be the shortest string accepted by *M* whose length is at least *n* i.e. w . (Note that such a string must exist, since *L(M)* is infinite and there are only a finite number of strings of length less than *n*). Then, it must be the case that, . Otherwise (i.e. if , by the pumping lemma we can decompose *w* as *w=xyz* satisfying all the constrains of the pumping lemma. So,

. For *i=0*, in particular, is a shorter string than *w* (since ), leading to a contradiction. Hence, .

This theorem gives us the following naive algorithm to determine the emptiness and finiteness of a language *L(M)* accepted by a *DFA M* .

**Algorithm to decide emptiness**

* Run *M* on all strings of length less than *n* , where *n* is the number of states.
* If *M* accepts any of these, than *L(M)* is nonempty. Otherwise, *L(M)* is empty. (From part (1) of the theorem).

**Algorithm to decide finiteness of *L(M)* .**

* Run *M* on all strings of length between *n* and *2n*

If *M* accepts any string of these, then *L(M)* is infinite. Otherwise, *L(M)* is finite.(From part (2) of the theorem)

But, efficient algorithms exists to decide these problem. We know that a *DFA* can be represented by a directed graph and for a *DFA* to accept a string there must exist a path from the start state to any final state. Using this fact, we have the following efficient algorithm to decide emptiness. (Assume, *DFA M* is given as a directed graph)

* Do *DFA* from the start state *q0*
* If any of the final state is reachable from the start state *q0* , than *L(M)* is nonempty. Otherwise, *L(M)* is empty

**We now consider an efficient algorithm to determine whether *L(M)* is infinite.**

We know that all the states which are not reachable from *q0* can be detected (along with the associated transition) without changing the accepted language.

Similarly, the accepted language does not change if all the states that cannot lead to an accepting state (also called ?trap' states) are detected.

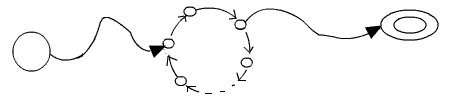
Claim1 : **If *L(M)* is infinite, then there must exist a cycle in the directed graph.**

Proof : Since *L(M)* is infinite, according to the previous theorem, there exists a string with where *n* is the number of states in the *DFA M* . Since the length of the accepted string *w* is greater than the number of states, there must exist a repeated state in the path from *q0* to the final state while processing the string *w*. His repetition of (at least one) state in the path implies the existence of a cycle.

Claim 2: If there is a cycle in the directed graph (for the *DFA M* ), then *L(M)* must be infinite.

Proof : We know that all states are reachable from the start state *q0*. Also, there can not be any cycle involving ?useless? states, because these have already been removed.

Hence if there exists a cycle, there must be a path from the start state *q0* to one of the states involved in the cycle and, also, there must be a path from on e of the states involved in the cycle to an accepting state. The situation is depicted in the following figure.



So, clearly, starting at , than following the cycle infinitely many times, the DFA can accept infinitely many strings.

Hence, *L(M)* is infinite.

It is a well-known fact that there exists efficient algorithm to detect a cycle in directed graph. From the above, we have the following efficient algorithm to decide infiniteness of *L(M)*.

* **Delete all states not reachable from the start state and delete all states that cannot lead to an accept state ( *DFS* can be used for this).**
* **If there is a cycle, then *L(M)* is infinite. Otherwise, *L(M)* is finite.**

It is observed that using the decision algorithm for emptiness and finiteness together with closure properties we can find more decision algorithms. Here is an example.

Example : **Given *DFA* s M1 and M2. Is *L(M1) = L(M2)*?**

Solution : Observe that and 

Thus *L(M1) = L(M2),* iff and This implies that

*L(M1) = L(M2) iff *

Since regular languages are closed under union, intersection and complement, we can construct a *DFA* M3 recognizing the language 

If *M3* accepts any string (i.e. ) then ) .Otherwise , *L(M1) = L(M2)*

Questions:

1. Prove that the set of strings accepted by an FA with n states is infinite iff the FA accepts some string of lenghth l where, n<=l<2n.

References:

*Introduction to Automata Theory, Languages an Computation- John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman Page No-59*

Lecture #13

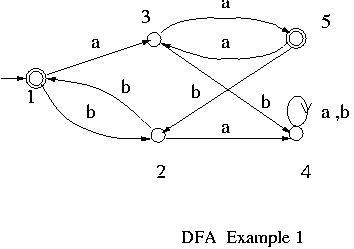
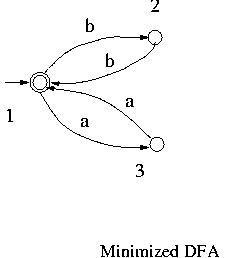
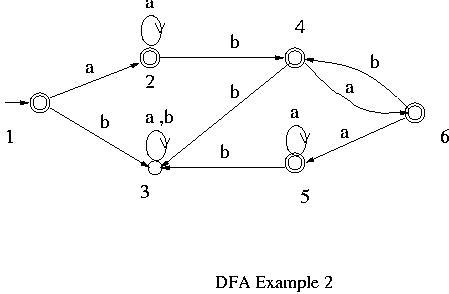
Module # \_1

Title: Minimization of FSA*.*

*Explanation:*

One important result on finite automata, both theoretically and practically, is that for any regular language there is a unique DFA having the smallest number of states that accepts it. Let M = < Q , , q0 , , A > be a DFA that accepts a language L. Then the following algorithm produces the DFA, denote it by M1, that has the smallest number of states amomg the DFAs that accept L.   
  
**Minimization Algorithm for DFA**   
  
Construct a partition = { A, Q - A } of the set of states Q ;   
new := new\_partition(} ;   
**while** (new ≠ )   
        := new ;   
        new := new\_partition()   
final := ;   
  
  
function **new\_partition**()   
**for** each set S of **do**   
        partition S into subsets such that two states p and q of S are in the same subset of S   
        if and only if for each input symbol, p and q make a transition to (states of) the same set of .   
  
        The subsets thus formed are sets of the output partition in place of S.   
        If S is not partitioned in this process, S remains in the output partition.   
**end**   
  
  
Minimum DFA M1 is constructed from final as follows:

* Select one state in each set of the partition final as the representative for the set. These representatives are states of minimum DFA M1.
* Let p and q be representatives i.e. states of minimum DFA M1. Let us also denote by p and q the sets of states of the original DFA M represented by p and q, respectively. Let s be a state in p and t a state in q. If a transition from s to t on symbol a exists in M, then the minimum DFA M1 has a transition from p to q on symbol a.
* The start state of M1 is the representative which contains the start state of M.
* The accepting states of M1 are representatives that are in A.   
  Note that the sets of final are either a subset of A or disjoint from A.

Remove from M1 the dead states and the states not reachable from the start state, if there are any. Any transitions to a dead state become undefined.   
A state is a **dead state** if it is not an accepting state and has no out-going transitions except to itself.   
  
**Example 1 :**   
Let us try to minimize the number of states of the following DFA.   
  
  
      
  
Initially = { { 1 , 5 } , { 2 , 3 , 4 } }.   
  
new\_partition is applied to .   
Since on b state 2 goes to state 1, state 3 goes to state 4 and 1 and 4 are in different sets in , states 2 and 3 are going to be separated from each other in new .   
Also since on a sate 4 goes to sate 4, state 3 goes to state 5 and 4 and 5 are in different sets in , states 3 and 4 are going to be separated from each other in new.   
Further, since on b 2 goes to 1, 4 goes to 4 and 1 and 4 are in different sets in , 2 and 4 are separated from each other in new.   
On the other hand 1 and 5 make the same transitions. So they are not going to be split.   
  
Thus the new partition is { { 1 , 5 } , { 2 } , { 3 } , { 4 ] }. This becomes the in the second iteration.   
  
When new\_partition is applied to this new , since 1 and 5 do the same transitions, remains unchanged.   
Thus final = { { 1 , 5 } , { 2 } , { 3 } , { 4 ] }.   
  
Select 1 as the representative for { 1 , 5 }. Since the rest are singletons, they have the obvious representatives.   
Note here that state 4 is a dead state because the only transitionout of it is to itself.   
Thus the set of **states** for the minimized DFA is { 1 , 2 , 3 }.   
For the **transitions**, since 1 goes to 3 on a, and to 2 on b in the original DFA, in the minimized DFA transitions are added from 1 to 3 on a, and 1 to 2 on b. Also since 2 goes to 1 on b, and 3 goes to 1 on a in the original DFA, in the minimized DFA transitions are added from 2 to 1 on b, and from 3 to 1 on a.   
Since the rest of the states are singletons, all transitions between them are inherited for the minimized DFA.   
Thus the minimized DFA is as given in the following figure:   
  
     
  
**Example 2 :**   
Let us try to minimize the number of states of the following DFA.   
  
  
  
  
Initially = { { 3 } , { 1 , 2 , 4 , 5 , 6 } }.   
By applying new\_partition to this , new = { { 3 } , { 1 , 4 , 5 } , { 2 , 6 } } is obtained.   
Applyting new\_partition to this , new = { { 3 } , { 1 , 4 } , { 5 } , { 2 } , { 6 } } is obtained.   
Applyting new\_partition again, new = { { 1 } , { 2 } , { 3 } , { 4 } , { 5 } , { 6 } } is obtained.   
Thus the number of states of the given DFA is already minimum and it can not be reduced any further.

Questions:

1. Find the minimal DFA for the following transition system.(July 2008- 12b)

References:

*Introduction to Automata Theory, Languages an Computation- John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman Page No-173*

Lecture #14

Module # \_1

Title: Moore machine

*Explanation:*

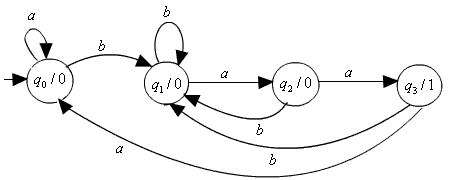
The definition of FA that we have already considered allows only two possible outputs is response to an input string, accept or reject. The definition can be extended so that the output can be chosen from some alphabet. Considering two different approaches to associate the output we have two different types of machines in the category- Moore machines and Mealy machines (They are named after the inventors). In a Moore machine the output is associated with the state, whereas in a Mealy machine the output is associated with the transition. Even though the two models look different, we can prove that they are equivalent.

**Moore machines** :

A Moore machine is a six-tuple

 where and *q0* are as in *DFA* . is the output alphabet and , is a mapping which gives the output associated with each state. Note that there is no final state and the input and output alphabet need not to be same.

Let the sequence of states the machine goes through in response to the input sequence is . Then the output produced by the machine in response to this input is defined as .Note that a Moore machine produces an output without taking any input on state *q0*. That is, is the output in response to input . Hence, the length of the output string is always one more than that of the input string.

**Example 1**: Suppose we wish to determine exactly low many times the sub string occurs in the input string. The Moore machine presented by the given transition diagram        

Keeps count of this number.

Every state outputs **0** except the state q3 which outputs *a* **1.** start state *q0*, following any path, we arrive at state *q3* the last three input symbols read must be b,a and a. As soon as we arrive at *q3*, it outputs 1(prior to that it outputs all 0s) indicating that it has read the sub string **baa** in the input. From *q3* we can arrive at *q1* on input *b* and then again arrive at *q3*(following some path) provided the last three input symbols read are b,a and a. Thus, the machine outputs 1 as soon as it read the sub string **baa**; otherwise, it outputs **0**s. So, the number of sub string **baa** in the input is given by the number of **1**s in the output string at the point when the machine finishes processing the input string.

For example, on input the machine will go through the states producing the output sequence **000000010000010** indicating that the sub string **baa** occurs twice in the input string as the number of 1s in the output string is **2**.

The Moore machine can also be represented by a table, where the table to represent the transition remains same as in *FA* , but there is a separate column (separated by a double line) to represent the output associated with each state. The tabular form of the Moore machine of the above example is given below.

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | *b* |  |
| *q0* | *q0* | *q1* | *0* |
| *q1* | *q2* | *q1* | *0* |
| *q2* | *q3* | *q1* | *0* |
| *q3* | *q0* | *q1* | *1* |

A Moore machine does not define a language of accepted strings, because in response to any input string it produces an output string and there is no concept of final states. The processing of the input string terminates when it outputs the symbol corresponding to the last input symbol.

For a given *FA M* , accepting the language *L(M)*, if we associate 0 to any nonaccepting state and 1 to each accept state, then the 1's in any output sequence (produced in response to some input sequence) mark the ending of all sub strings of the input starting from the first symbol that are in *L(M)*.

From this, we can consider *FA* to be a special case of a Moore machine where the output alphabet and a state *p* is ?accepting' if and only if .

So, a Moore machine can be said to recognize the language of all input strings whose outputs ends in a 1. In the example Moore machine given above if we make *q3* as final state and remove all outputs associated with the states, it will be a *DFA* accepting all string over that ends with .

Questions:

1. What are Moore machines? Explain.
2. Design a more machine which computes residue mod 4 for the binary input string.

References:

*Reference book & page nos*

Lecture #15

Module # \_1

Title: Mealy machine

*Explanation:*

Mealy machines : A Mealy machine is a six-tuple, , where all elements are as in Moore machine, except for which is defined as



This means that gives the output associated with the transition from state *q* on input .   
Let the sequence of states the machine goes through in response to the input string is .

Then the output produced by the machine in response to this input is defined by . The length of the output sequence unlike for the Moore machine.

|  |  |
| --- | --- |
| **Example:2** Consider the Mealy machine given below.  The machine outputs 1 in the output string in response to some input string to indicate two consecutive occurrences of in the input.  For example, the out put corresponding to the input is 00001100100. |  |

We can express the Mealy machine in tabular form as indicated below.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  | | --- | --- | --- | --- | |  | *q0* | *q1* | *q2* | | *q0* |  | /0 | *b/0* | | *q1* |  | /1 | *b/0* | | *q2* |  | /0 | *b/0* | | The entry *b/0* for the raw *q1* & column *q2*indicates that there is a transition from state *q1* to state *q2* on input *b* and the output associated with this transition is 0. For no transition defined from state *p* to state *q* the entry for raw *p* & column *q* will be |

**Equivalence of Moore and Mealy machines**

Since Moore and Mealy machines both produces output (instead of normal convention of accepting a language by a *FA* ). We can compare them in the sense that they are equivalent if they always produce the same output string in response to the same input string. But there can never be an exact match between the output strings produced by them since the length of the output string of a Moore machine is always one more than that of a Mealy machine in response to the same input string. However, if we ignore the response of a Moore machine for its initial state (i.e. response to input ), then we can define the equivalence of a Moore machine, and a Mealy machine by saying that if for all input string , where is the output of for its initial state and are outputs of and on *w* respectively. Then they are equivalent

Questions:

1. What are Mealy machines? Give its formal definition.

References:

*Reference book & page nos*