Syllabus

01 July 2019 06:17 AM

FORMAL LANGUAGES AND AUTOMATA THEORY (FLAT)

V Semester: B.Tech-CSE						Scheme: 2017		
Course Code	Category	Hours/Week			Credits	Maximum Marks		
CS303	Program Core	L	T	P	C	Continuous Internal Assessment	End Exam	TOTAL
		3	0	0	3	40	60	100
Sessional Exam Duration:2 Hrs						End Exam Duration:3 Hrs		

Course Out comes: At the end of the course students will be able to

CO1: Design the finite automata for a given regular language.

CO2: Understand the regular expressions and pumping lemma of regular languages.

CO3: Understand the regular grammar, context free grammar and pumping lemma for CFL.

CO4: Design push down automata and context free grammar for a given context free language.

CO5: Design the Turing machine for the given formal language.

UNIT– I

Finite Automata preliminaries: Strings, Alphabet, Language Operations, Finite State Machine definitions, Finite Automation Model, Acceptance of strings and languages, Non-deterministic Finite Automation, Equivalence between NFA and DFA, conversion of NFA into DFA, Equivalence between two FSM's, Minimization of FSM, Moore and Mealy machines, Applications of FA's.

UNIT– II

Regular Expressions and Regular Sets: Regular sets, Regular expressions, Identity rules, Manipulation of regular expression, Equivalence between RE and FA, Inter conversion, Pumping lemma for RE, Closure properties of regular sets.

UNIT– III

Grammar Formalism: Regular Grammar-Right linear grammar and left linear grammar, Equivalence between regular linear grammar and FA, inter-conversion between RE and RG, Derivation trees, Right most and left most derivation of strings.

Context Free Grammar: Context Free Grammar, Ambiguity in CFG, minimization of CFG, Chomsky Normal Form, Griebach Normal Form, pumping lemma of CFL.

UNIT– IV

Push Down Automata: Definition of the Pushdown Automaton, A Graphical Notation for PDA's, Instantaneous Descriptions of a PDA, The Languages of a PDA, Acceptance by Final State, Acceptance by Empty Stack, Equivalence of PDA's and CFG's, Properties of Context Free Languages.

UNIT– V

Turing Machines: Introduction to Turing Machines, Notation for the Turing Machine, Instantaneous Descriptions for the Turing Machines, Transition Diagrams for Turing Machines, The Language of a Turing Machine, Universal Turing machine, Halting problem of Turing Machine.

Text Books:

1. J.E.Hopcroft, Rajeev Motwani and J.D.Ullman, Introduction to Automata Theory Languages and Computation, Third edition, 2007, Pearson Education.
2. Mishra and Chandrashakaran, [2008], [Third Edition], Theory of computer sciences: Automata languages and computation, Third Edition, 2008, PHI.

Reference Books:

1. John C Martin, Introduction to languages and the theory of computation, Third edition, 2007, TMH.
2. Peter Linz, An Introduction to Formal Languages and Automata, Fourth edition, 2010, Narosa Book Distributors Pvt. Ltd.
3. Michael Sipser, Introduction to Theory of Computation, 3rd Edition, 2012, Cengage Learning.
4. Bernar M Moret, The Theory of Computation, First edition, 2002, Pearson Education.

Web References:

1. https://nptel.ac.in/courses/111103016/
2. https://www.tutorialspoint.com/automata_theory/

Question Paper Pattern:

Sessional Exam:

The question paper for Sessional examination is for 30 marks, covering half of the syllabus for first Sessional and remaining half for second Sessional exam. Question No 1, which carries 6 marks, contains three short answer questions of two marks each. The remaining three questions shall be EITHER/OR type questions carrying 8 marks each.

End Exam:

Question Paper Contains Six Questions. Question 1 contains 5 short Answer questions each of 2 marks. (Total 10 marks) covering one question from each unit. The remaining five questions shall be EITHER/OR type questions carrying 10 marks each. Each of these questions is from one unit and may contain sub-questions. i.e. there will be two questions from each unit and the student should answer any one question.

Note: JFLAP software is used to design the models of DFA, NFA, Moore machine, Mealy machine, PDA and TM.

UNIT - I

30 June 2017 07:14 PM

Basic Terms and Definitions

05 July 2017 06:23 AM

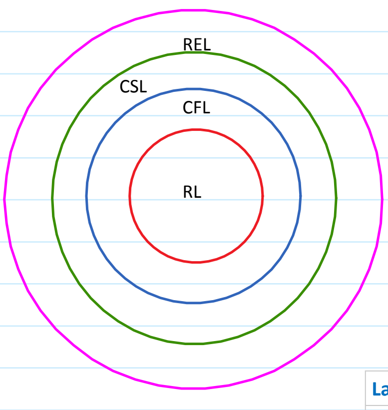
Symbol: a, b, c, 1, 2, 3 , etc. are called symbols.
Alphabet: finite and non-empty set of symbols.
Denoted by Σ.
Examples: Σ = {1, 2}
Σ = { a, b}
Word or String: finite sequence of symbols chosen from Σ.
Denoted by u, v, w, x, y, z
Ex: For Σ = {0, 1} , strings are λ, 0, 1, 00, 01, 10, 11, 000, 001, ….
Null String or Empty String is λ

Σ^1 = { 0, 1 }	(Strings of length 1)
Σ^2 = { 00, 01, 10, 11}	(Strings of length 2)
Σ^3 = { 000, 001, 010, 011, 100, 101, 110, 111 }	(Strings of length 3)
Σ^k = { x | x is string of length k }	(strings of length exactly k)
Σ^* = { x | x is string of any length ≥ 0 }	(including null string)
Σ^+ = { x | x is string of any length > 0 }	(no null string)

SET THEORY:
SET: unordered collection of definite and distinct objects.
A = {1, 2, 3, …..} or A = { x | x is a natural number}
Empty Set vs Non-empty Set
Finite Set vs Infinite Set
{1, 2, 3} = {2, 1, 3} = {1, 2, 2, 3}

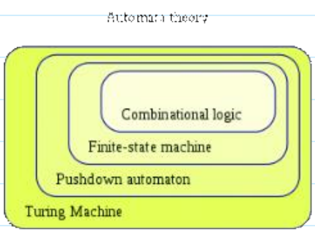
Language : L is a subset of Σ*
For Σ = {0, 1},
L1 = {all strings of 0's and 1's ending with 01}
L2 = { all strings of 0's and 1's with substring 101}
L3 = { all strings of 0's and 1's with even length}
L4 = {all strings of 0's and 1's starting with 1 and ending with 011} etc.

Formal Language: A language that has a well-defined set of syntax rules. (grammar)
The mathematician "Noam Chomsky" gave a classification of formal languages as follows:



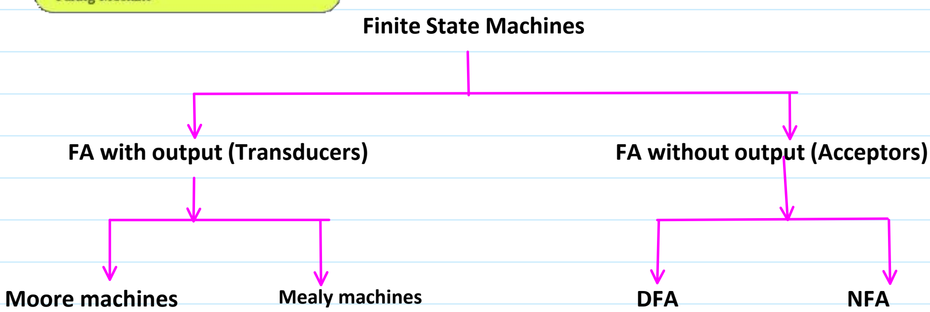
RL - Regular Languages
CFL - Context Free Languages
CSL - Context Sensitive Languages
REL - Recursively Enumerable Languages

Each regular language is also a context free language. But each CFL is not a RL.
Each context free language is also a context sensitive language. But each CSL is not a CFL.
Each context sensitive language is also a recursively enumerable language. But each REL is not a CSL.



Language	Language Recognizer
RL	DFA or NFA
CFL	PDA
CSL	LBA
REL	TM

DFA = Deterministic Finite Automata
NFA = Non-deterministic Finite Automata
PDA = Push Down Automata
LBA = Linear Bounded Automata
TM= Turing Machine



FA models the computer hardware consisting of a processor with finite amount of memory.

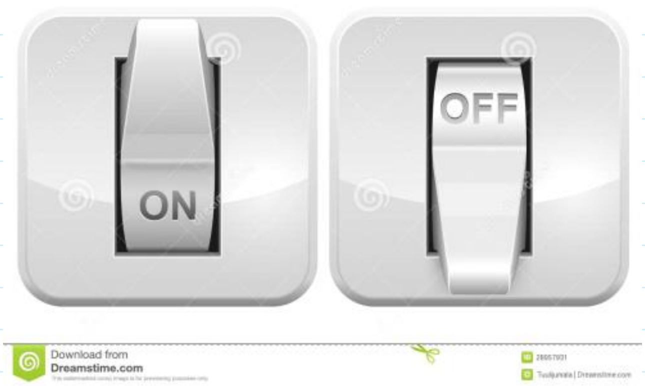
So far we have seen computations in high level (using programming languages). Now we look at the computations at machine level.

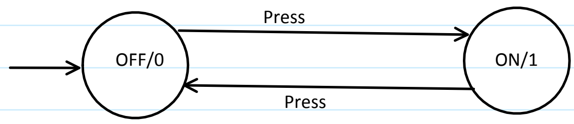
A digital computer takes input data and produces output data. How the input data is mapped with output data? We see the mapping of input data to output at the lowest level.

Before learning DFA formally, see DFA in real life.
Example: Electric Switch
Switch has two states: ON (1) and OFF (0).
By default, switch is in the initial state of OFF.
When we press the switch, state transition occurs between ON and OFF as shown below:

DFA is represented using directed graph where each node is a state and edge represents state transition.

state and edge represents state transition.





Deterministic Finite Automata

Given the state and input,
You must be able to determine
The next output state

Finite no.of states
Finite amount of memory

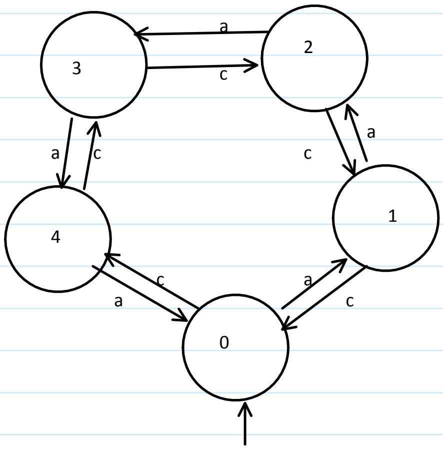
Singular - "automaton"

Given the switch state as ON, and
Asked to determine the next state for
Pressing 5 times as 01001 or 11100 etc.

ON OFF

Here, the switch must remember current state out of two possible states.
Therefore, memory of two bits is enough.

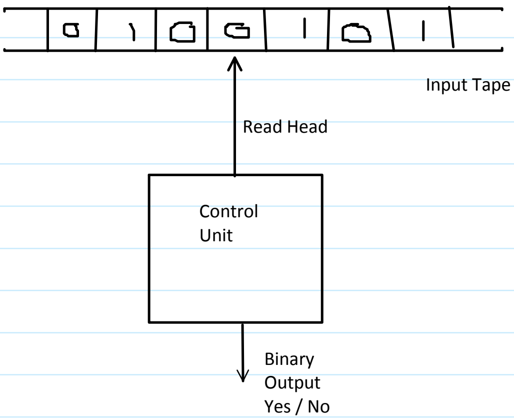
Another example of DFA is FAN Regulator.



c - clockwise rotation
a - anticlockwise rotation

Here, Fan has to remember the current state out of five possible states. Therefore, $\log_2(5)=3$ bits are enough for memory

DFA Model



DFA has memory in the form of tape to store input string to be scanned.
Tape is divided into cells so that each cell stores a symbol of input string.
Control unit is responsible for state transitions.
Using read head, the control unit scans the input string stored in the tape one symbol at a time.
After reading the last symbol of input string, DFA halts in one state which is considered as the state yielding output "yes" or "no".

In general, if DFA has 'n' states then it requires $\log_2(n)$ bits for memory

Formal Definition of DFA:
M = (Q, Σ, δ, q0, F) where
Q is set of states
Σ is set of symbols
δ is state-transition function defined as Q X Σ → Q
q0 is initial state
F is set of final states , F ⊂ Q

For the electric switch example:
Q = {OFF, ON}
Σ = { 0, 1}
q0 = OFF
F={0,1}

For the fan regulator example:

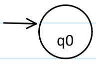
Q = { 0, 1, 2, 3, 4}
Σ = { c, a}
q0 = 0
F={0,1,2,3,4}

DFA Design

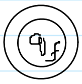
10 July 2017 05:53 AM

While designing DFA, it is represented using a directed graph in which each state is a node and the edge between two nodes shows the state transition between the two states. Each edge is labelled with an input symbol to denote the input symbol upon which state transition occurs.

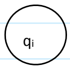
Graph Notation:



Initial state



Final State

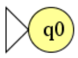


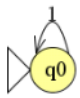
Intermediate state

When a DFA is represented as a graph, each state in DFA is shown as a node in the graph and state transition is shown as an edge between nodes with the symbol on which transition occurs.

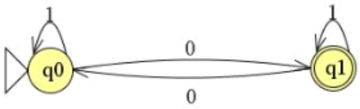
On the other hand, if DFA is shown as table, then the number of rows is equal to the number of states in DFA and the number of columns is equal to the number of input symbols in the alphabet.

Design DFA to accept strings of 0's and 1's containing odd number of 0's.



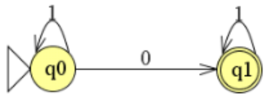


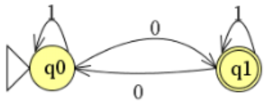
DFA as a Graph:



DFA as a Table:

State	0	1
→ q0	q1	q0
q1	q0	q1





State-Transition Table

State-Transition Graph

DFA Design Constraints:

1. From each state and for every input symbol, draw exactly one transition. (m states, n symbols → mn transitions).
2. In DFA, state transition occurs only after reading the input symbol. (no null transitions).
3. DFA must contain only one initial state. (Deterministic property). (Ex: C program contains only one main() function.)

1. **Pattern Recognition Problems**
2. **Divisible by k Problems**
3. **Modulo-k-counter problems**

Pattern Recognition Problems

1. ***Draw a DFA to accept string of a's having at least one a.***
2. ***Draw a DFA to accept strings of a's and b's having at least one a.***
3. ***Draw a DFA to accept strings of a's and b's having at most one b.***
4. ***Draw a DFA to accept strings of a's and b's having exactly one a.***
5. ***Obtain a DFA to accept strings of a's and b's starting with the string ab.***
6. ***Draw a DFA to accept strings of a's and b's ending with the string abb.***
7. ***Draw a DFA to accept strings of a's and b's having substring aab.***
8. ***Design a DFA to accept strings of a's and b's that do not start with string ab.***
9. ***Design a DFA to accept strings of a's and b's that do not end with string abb.***
10. ***Design a DFA to accept strings of a's and b's that do not have substring aab.***
11. ***Draw a DFA to accept string of a's and b's such that*** $L = \{ awa \mid w \in (a + b)^* \text{ where } n >= 0 \}$
12. ***Draw a DFA to accept strings of a's and b's starting with ab or ba.***
13. ***Draw a DFA to accept strings of a's and b's ending with ab or ba.***
14. ***Design a DFA to accept strings of a's and b's having substring aab or bba.***
15. ***Obtain a DFA to accept strings of a's and b's having four a's.***
16. ***Obtain a DFA to accept strings of 0's,1's and 2's beginning with a '0' followed by odd number of 1's and ending with a '2'.***
17. ***Obtain a DFA to accept strings of a's and b's with at most two consecutive b's.***
18. ***Obtain a DFA to accept strings of a's and b's starting with at least two a's and ending with at least two b's.***
19. ***Draw a DFA to accept strings of a's and b's having not more than three a's.***
20. ***Draw a DFA to accept set of all strings on the alphabet*** $\Sigma = \{ 0, 1 \}$ ***that either begins or ends with the substring 01.***
21. ***Draw a DFA to accept the language*** $L = \{ w : n_a(w) >=1, n_b(w) = 2\}$
22. ***Draw a DFA to accept the language*** $L = \{ w : n_a(w) = 2, n_b(w) >= 3\}$

Divisible by k Problems

1. ***Obtain a DFA that accepts binary integers divisible by 2,3,4 and 5.***
2. ***Draw a DFA to accept decimal strings divisible by 2,3,4 and 5.***
3. ***Draw a DFA to accept decimal strings divisible by 2 and 3.***
4. ***Draw a DFA to accept binary strings divisible by 2 and 3***
5. ***Draw a DFA to accept decimal strings divisible by 2 or 3.***
6. ***Draw a DFA to accept binary strings divisible by 2 or 3.***

Modulo k Counter Problems

1. ***Obtain a DFA to accept strings of even number of a's.***
2. ***Obtain a DFA to accept strings of odd number of b's.***
3. ***Obtain a DFA to accept strings of even number of a's and odd number of b's.***
4. ***Obtain a DFA to accept strings of odd number of a's and even number of b's.***
5. ***Obtain a DFA to accept strings of odd number of a's and odd number of b's.***
6. ***Obtain a DFA to accept strings of even number of a's and even number of b's.***
7. ***Obtain a DFA to accept the language*** $L = \{ w : |w| \bmod 3 = 0 \}$ ***where*** $\Sigma = \{ a \}$.
8. ***Obtain a DFA to accept the language*** $L = \{ w : |w| \bmod 3 = 0 \}$ ***where*** $\Sigma = \{ a,b \}$.
9. ***Obtain a DFA to accept the following language*** $L = \{$ ***w such that***

a) $|w| \bmod 3 >= |w| \bmod 2$ ***where*** $w \in \Sigma^*$ ***and*** $\Sigma = \{a\}$.

b) $|w| \bmod 3 \neq |w| \bmod 2$ ***where*** $w \in \Sigma^*$ ***and*** $\Sigma = \{a\}$

10. ***Obtain a DFA to accept the following language*** $L = \{$ ***w such that***

a) $|w| \bmod 3 >= |w| \bmod 2$ ***where*** $w \in \Sigma^*$ ***and*** $\Sigma = \{a, b\}$.

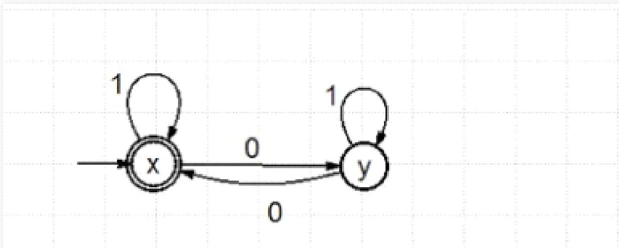
b) $|w| \bmod 3 \neq |w| \bmod 2$ ***where*** $w \in \Sigma^*$ ***and*** $\Sigma = \{a, b\}$

11. ***Obtain a DFA to accept the language*** $L = \{ w : |w| \bmod 5 \neq 0 \}$ ***on*** $\Sigma = \{a\}$.
12. ***Obtain a DFA to accept the language*** $L = \{ w : |w| \bmod 5 \neq 0 \}$ ***on*** $\Sigma = \{a, b\}$.
13. ***Obtain a DFA to accept strings of a's and b's such that*** $L = \{ w \mid w \in (a + b)^*$ ***such that*** $N_a(w) \bmod 3 = 0$ ***and*** $N_b(w) \bmod 2 = 0 \}$

C Program on DFA1

16 July 2018 08:52 PM

Design a DFA which will accept all the strings containing even number of 0's over an alphabet {0, 1} and write a program to implement the DFA.



Start here | dfa_even_zeros.c

```c
#include<stdio.h>
#include<string.h>
#define max 100
int main()
{
 char str[max],f='x',c;
 printf("do you want to check for epsilon string's case? (y/n) : ");
 scanf("%c",&c);
 if(c=='y'|| c=='Y')
 goto flag;

 printf("enter the string to be checked: ");
 scanf("%s",str);
 int i;
 for(i=0;i<strlen(str);i++)
 {
  switch(f)
  {
   case 'x': if(str[i]=='0') f='y';
    else if(str[i]=='1') f='x';
   break;
   case 'y': if(str[i]=='0') f='x';
    else if(str[i]=='1') f='y';
   break;
  }
 }
flag: if(f=='x') printf("\nString is accepted as it reaches the final state that is %c",f);
 else printf("\nString is not accepted as it reaches a state %c which is not the final state",f);

return 0;
}
```

"C:\Users\GVK\Documents\C programs\dfa_even_zeros.exe"

```
do you want to check for epsilon string's case? (y/n) : y

String is accepted as it reaches the final state that is x
Process returned 0 (0x0)   execution time : 3.959 s
Press any key to continue.
```

"C:\Users\GVK\Documents\C programs\dfa_even_zeros.exe"

```
do you want to check for epsilon string's case? (y/n) : n
enter the string to be checked: 10101

String is accepted as it reaches the final state that is x
Process returned 0 (0x0)   execution time : 16.912 s
Press any key to continue.
```

"C:\Users\GVK\Documents\C programs\dfa_even_zeros.exe"

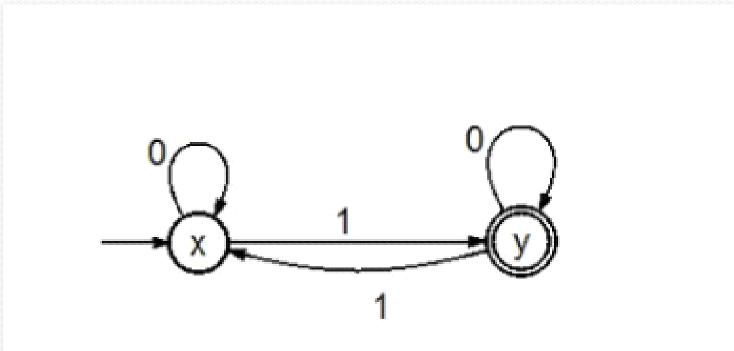
```
do you want to check for epsilon string's case? (y/n) : n
enter the string to be checked: 101010

String is not accepted as it reaches a state y which is not the final state
Process returned 0 (0x0)   execution time : 10.938 s
Press any key to continue.
```

C Program on DFA2

16 July 2018   09:22 PM

Design a DFA which will accept all the strings containing odd number of 1's over an alphabet {0, 1} and write a program to implement the DFA.



Start here | dfa_even_zeros.c | dfa2.c | dfa3.c

```c
#define max 100
main()
{
 char str[max],f='x';
 int i;
 printf("enter the string to be checked: ");
 scanf("%s",str);
 for(i=0;i<strlen(str);i++)
 {
  switch(f)
  {
   case 'x': if(str[i]=='0') f='x';
    else if(str[i]=='1') f='y';
   break;
   case 'y': if(str[i]=='0') f='y';
    else if(str[i]=='1') f='x';
   break;
  }
 }
 if(f=='y') printf("\nString accepted. State reached is: %c",f);
 else printf("\nstring not accepted. State reached is: %c",f);
}
```

"C:\Users\GVK\Documents\C programs\dfa3.exe"

```
enter the string to be checked: 010011

String accepted. State reached is: y
Process returned 37 (0x25)   execution time : 7.961 s
Press any key to continue.
```

"C:\Users\GVK\Documents\C programs\dfa3.exe"

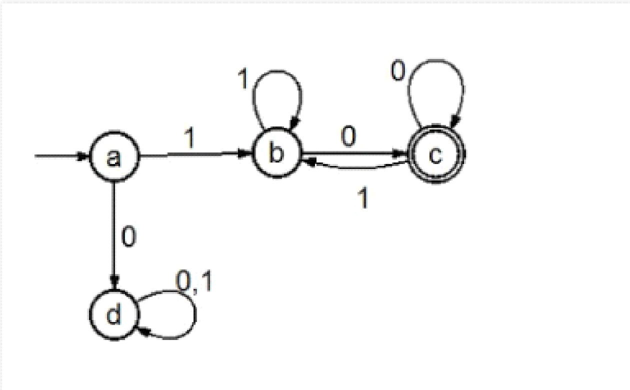
```
enter the string to be checked: 01001

string not accepted. State reached is: x
Process returned 41 (0x29)   execution time : 5.365 s
Press any key to continue.
```

C Program on DFA3

16 July 2018 09:14 PM

Design a DFA which will accept all the strings starting with 1 and ending with 0 over an alphabet {0, 1} and write a program to implement the DFA.



Start here | dfa_even_zeros.c | dfa2.c

```c
#define max 20
main()
{
 char str[max],f='a';
 int i;
 printf("Enter the string to be checked: ");
 scanf("%s",str);
 for(i=0;str[i]!='\0';i++)
 {
  switch(f)
  {
   case 'a': if(str[i]=='0') f='d';
    else if(str[i]=='1') f='b';
   break;

   case 'b': if(str[i]=='0') f='c';
    else if(str[i]=='1') f='b';
   break;
   case 'c': if(str[i]=='0') f='c';
    else if(str[i]=='1') f='b';
   break;
   case 'd': if(str[i]=='0') f='d';
    else if(str[i]=='1') f='d';
   break;
  }
 }
 if(f=='c') printf("\nEntered string is accepted as it reached the final state i.e., %c",f);
 else printf("\nEntered string is not accepted as it reached %c which not the final state",f);
}
```

"C:\Users\GVK\Documents\C programs\dfa2.exe"

```
Enter the string to be checked: 01010

Entered string is not accepted as it reached d which not the final state
Process returned 73 (0x49)   execution time : 6.511 s
Press any key to continue.
```

"C:\Users\GVK\Documents\C programs\dfa2.exe"

```
Enter the string to be checked: 010101

Entered string is not accepted as it reached d which not the final state
Process returned 73 (0x49)   execution time : 5.588 s
Press any key to continue.
```

"C:\Users\GVK\Documents\C programs\dfa2.exe"

```
Enter the string to be checked: 101010

Entered string is accepted as it reached the final state i.e., c
Process returned 65 (0x41)   execution time : 6.324 s
Press any key to continue.
```

"C:\Users\GVK\Documents\C programs\dfa2.exe"

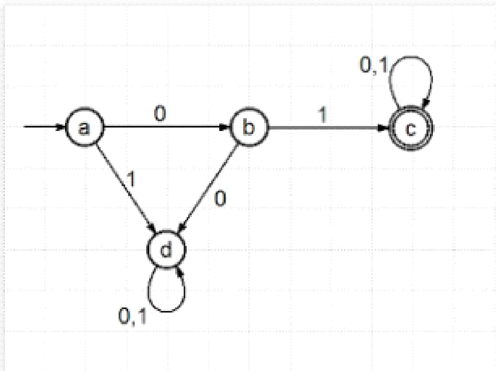
```
Enter the string to be checked: 100101

Entered string is not accepted as it reached b which not the final state
Process returned 73 (0x49)   execution time : 5.198 s
Press any key to continue.
```

C Program on DFA4

16 July 2018 09:32 PM

Design a DFA which will accept all the strings starting with 01 over an alphabet {0, 1} and write a program to implement the DFA.



Start here | dfa4.c

```c
#include<stdio.h>
#define max 100
main()
{
 char str[max],f='a';
 int i;
 printf("enter the string to be checked: ");
 scanf("%s",str);
 for(i=0;str[i]!=NULL;i++)
 {
  switch(f)
  {
   case 'a': if(str[i]=='0') f='b';
    else if(str[i]=='1') f='d';
   break;
   case 'b': if(str[i]=='0') f='d';
    else if(str[i]=='1') f='c';
   break;
   case 'c': if(str[i]=='0') f='c';
    else if(str[i]=='1') f='c';
   break;
   case 'd': if(str[i]=='0') f='d';
    else if(str[i]=='1') f='d';
   break;
  }
 }
 if(f=='c') printf("String is accepted as it reaches the final state that is %c.",f);
 else printf("String is not accepted as it reaches the state %c which is not a final state.",f);
}
```

"C:\Users\GVK\Documents\C programs\dfa4.exe"

```
enter the string to be checked: 000101
String is not accepted as it reaches the state d which is not a final state.
Process returned 76 (0x4C)   execution time : 5.508 s
Press any key to continue.
```

"C:\Users\GVK\Documents\C programs\dfa4.exe"

```
enter the string to be checked: 0101010
String is accepted as it reaches the final state that is c.
Process returned 59 (0x3B)   execution time : 4.400 s
Press any key to continue.
```

"C:\Users\GVK\Documents\C programs\dfa4.exe"

```
enter the string to be checked: 1001010
String is not accepted as it reaches the state d which is not a final state.
Process returned 76 (0x4C)   execution time : 4.453 s
Press any key to continue.
```

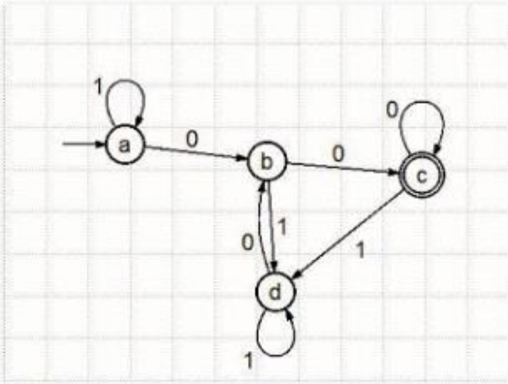
"C:\Users\GVK\Documents\C programs\dfa4.exe"

```
enter the string to be checked: 11010101
String is not accepted as it reaches the state d which is not a final state.
Process returned 76 (0x4C)   execution time : 4.622 s
Press any key to continue.
```

C Program on DFA5

16 July 2018 09:40 PM

Design a DFA which will accept all the strings ending with 00 over an alphabet {0, 1} and write a program to implement the DFA.



Start here | dfa5.c

```c
#define max 100
main()
{
 char str[max],f='a';
 int i;
 printf("enter the string to be checked: ");
 scanf("%s",str);
 for(i=0;str[i]!='\0';i++)
 {
  switch(f)
  {
   case 'a': if(str[i]=='0') f='b';
    else if(str[i]=='1') f='a';
   break;
   case 'b': if(str[i]=='0') f='c';
    else if(str[i]=='1') f='d';
   break;
   case 'c': if(str[i]=='0') f='c';
    else if(str[i]=='1') f='d';
   break;
   case 'd': if(str[i]=='0') f='b';
    else if(str[i]=='1') f='d';
   break;
  }
 }
 if(f=='c') printf("\nString is accepted as it reached the final state %c at the end.",f);
 else printf("\nString is not accepted as it reached %c which is not the final state.",f);
}
```

"C:\Users\GVK\Documents\C programs\dfa5.exe"

```
enter the string to be checked: 10100

String is accepted as it reached the final state c at the end.
Process returned 63 (0x3F)   execution time : 8.479 s
Press any key to continue.
```

"C:\Users\GVK\Documents\C programs\dfa5.exe"

```
enter the string to be checked: 010101

String is not accepted as it reached d which is not the final state.
Process returned 69 (0x45)   execution time : 6.098 s
Press any key to continue.
```

"C:\Users\GVK\Documents\C programs\dfa5.exe"

```
enter the string to be checked: 101010

String is not accepted as it reached b which is not the final state.
Process returned 69 (0x45)   execution time : 4.585 s
Press any key to continue.
```

"C:\Users\GVK\Documents\C programs\dfa5.exe"

```
enter the string to be checked: 10101011

String is not accepted as it reached d which is not the final state.
Process returned 69 (0x45)   execution time : 5.239 s
Press any key to continue.
```

NFA Introduction

21 July 2017 05:23 AM

Non-deterministic Finite Automata (NFA) :

It is not always possible to solve the problems using deterministic procedures. There are few problems that can be solved using non-deterministic approaches. In other words, guessing the solutions for the problem.

For example, if a person is missing and a group of people searching for him/her. Usually the people searches for the missing person each one taking a different route. Someone searches in bus station, someone in railway station, someone in friends place so on. Finally, if any one person finds the missing person, the mission will be completed.

In computer science applications, there are few problems that can be solved using non-deterministic algorithms. For example, in MANET, if a source node has data to send to a destination node, DSR algorithm establishes a routing path using RREQ and RREP packets. The source broadcasts RREQ and it is received by all its neighbour nodes. Then the packet is forwarded by neighbour nodes until it reaches the destination node. The RREP packet is then given by destination node.

Formal Definition:

M = (Q, Σ, δ, q0, F) where

Q is set of states
Σ is set of input symbols
q0 is set of initial states
F is set of final states and
δ is state-transition function defined as $Q \times \Sigma \rightarrow 2^Q$

Sometimes it is not easy to draw DFA for few problems. There NFA helps you. Draw NFA for the problem and later you may convert it into DFA.

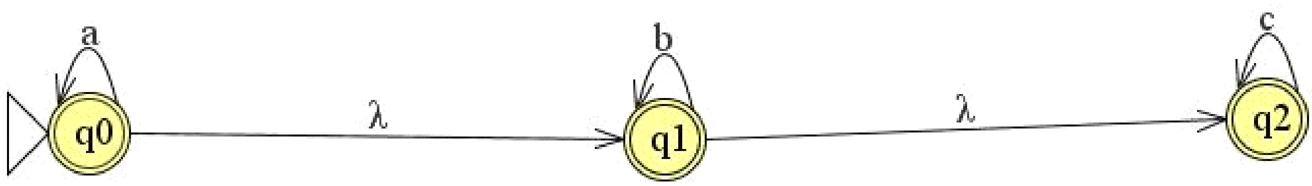
While designing DFA, three rules must be followed:

1. Regarding no. of transactions: For m states and n input symbols, DFA must have mn transitions.
2. Regarding null transitions: DFA has no null transitions. (state-transition occurs only after reading input symbol).
3. Regarding no. of initial states: DFA has only one initial state.

NFA design can omit the above three rules i.e.

1. NFA can have any no. of transitions (from any state you can draw any no. of transitions)
2. NFA can have null transitions.
3. NFA can have multiple initial states.

Example: Design NFA to accept the strings containing any no. of a's followed by any no. of b's followed by any no. of c's.



From state q0, there is no transition defined for the symbol b. Also, λ-transitions are present.

State	a	b	c
q0	{q0, q1, q2}	{q1,q2}	{q2}
q1	ϕ	{q1,q2}	{q2}
q2	Φ	Φ	{q2}

- In the table, we can see multiple states from a given state and for given symbol. Also empty set (no transitions are defined) for certain symbols from a state.

NFA Design

21 July 2017 06:19 AM

1. Design NFA to accept all the strings of a's and b's starting with ab.
2. Design NFA to accept all the strings of a's and b's ending with bb.
3. Design NFA to accept all the strings of a's and b's with substring aba.
4. Design NFA to accept all the strings of a's and b's starting with either ab or ba.
5. Design NFA to accept all the strings of a's and b's ending with either ab or ba.
6. Design NFA to accept all the strings of a's and b's containing substring either abb or aab
7. Find NFA for L = { x ϵ { a, b } * : x ends with a or x contain ab }
8. Find NFA for L = $\{a^i b^j : i \geq 1 , j \geq 1 \} \cup \{ \lambda, a \}$
9. Obtain NFA for L = $\{ 10^n : n \geq 0 \} \cup \{10^n 10^m : n, m \geq 0 \}$
10. Draw NFA for L = { x ϵ { a, b, c } * : x contains exactly one b immediately following c }
11. Find NFA for L = { x ϵ { 0, 1 } * : x is starting with 1 and |x| is divisible by 3 }
12. Find NFA for L = { x ϵ { a, b } * : x contains any number of a's followed by at least one b }
13.

Design an NFA with no more than five states for the set $\{abab^n : n \geq 0\} \cup \{aba^n : n \geq 0\}$.

The language $\{ w \in \Sigma^* \mid w$ contains at least two 0s, or exactly two 1s $\}$ with six states.

Design NFA to accept strings with atleast two consecutive 0's or 1's.
Design NFA to accept all strings of 0's and 1's in which third symbol from the right end is always 1.
Design NFA to accept all strings of 0's and 1's in which second leftmost symbol is always 1.
Design DFA and NFA to accept all the strings of 0's and 1's whose tenth symbol from the right end is 1.

Using ten-tuples to represent states give the design of a DFA that recognizes the language L = $\{x \in \{0,1\}^* \mid$ the tenth symbol from the right end of x is a '1'$\}$ To define the DFA we need only to specify all components:

States $Q = \{(x_{10}, x_9, x_8 \cdots x_2, x_1) \mid x_i \in \{0,1\}\}$ - note there are 2^{10} states.

Alphabet $\Sigma = \{0,1\}$

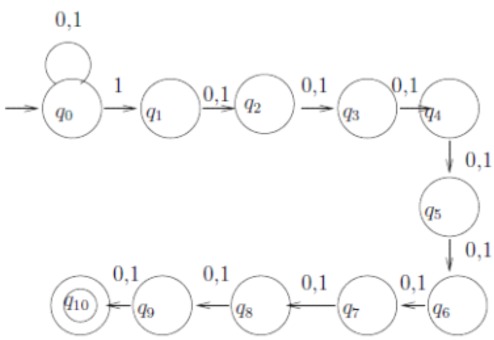
Start state $q_0 = (0,0,0,0,0,0,0,0,0,0)$

Final states $F = \{(1, x_9, x_8 \cdots x_2, x_1) \mid x_i \in \{0,1\}\}$ - note there are $2^9 = 512$ final or accepting states.

Transition Function For each element $a \in \Sigma$ $\delta(s,a)$ is defined by

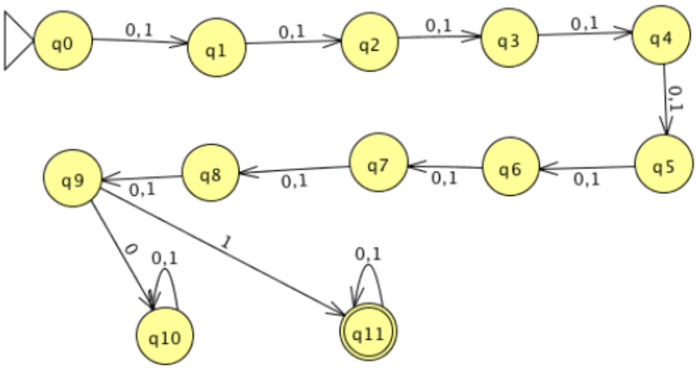
$$\delta((x_{10}, x_9, x_8 \cdots x_2, x_1),\ a) = (x_9, x_8, x_7 \cdots x_1, a)$$

The idea is to maintain in a list the last ten symbols and when the next charcter is considered shift everything one position and add the new character as the most recent symbol.

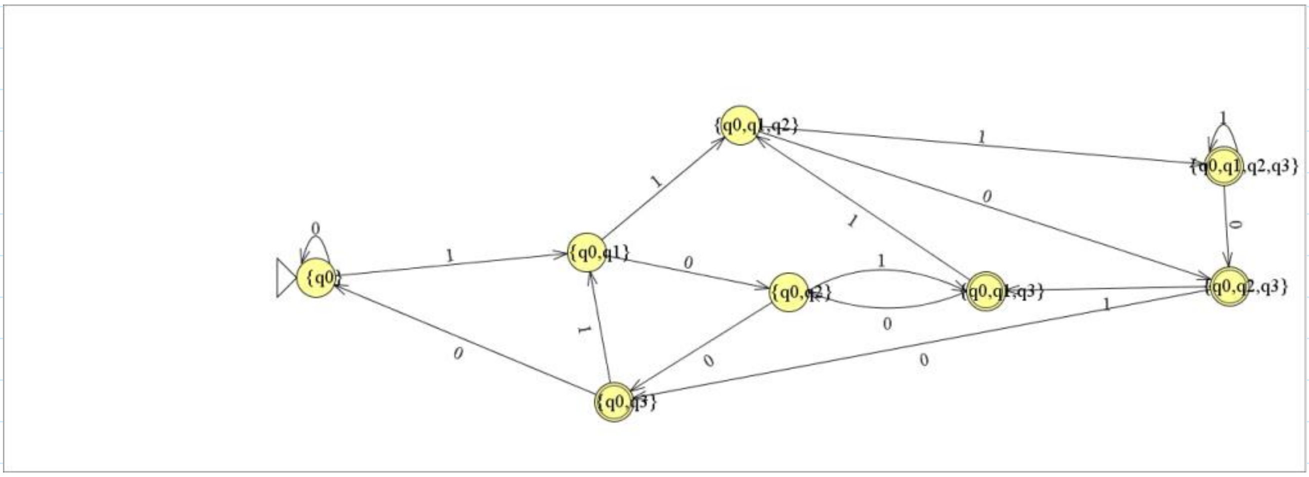


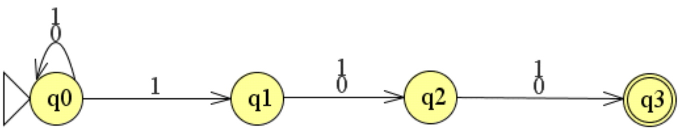
An NFA that accepts strings such that the tenth symbol from the right end is a 1

The set of all strings whose tenth symbol from the left end is a 1.



Design DFA accepting all binary strings in which third symbol from the right end is always a 1.





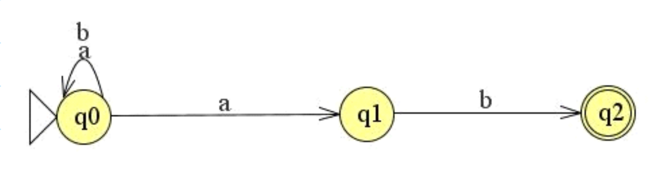
NFA to DFA Conversion

24 July 2017 04:50 AM

For any given NFA M, there exists an equivalent DFA M' such that L(M) = L(M').

Proof: Let M = (Q, Σ, δ, q_0, F) be given NFA accepting L(M).
We can define DFA M' = (Q', Σ, δ', q_0', F') as follows:
Σ is same for both DFA and NFA.
$q_0' = q_0$
For each input symbol 'a' if $\delta(q_i, a) = \{q_j, q_k, q_l\}$ then
$\delta'(q_i,a) = [q_j,q_k,q_l]$ (single state)
$Q' = 2^Q$
For each input symbol 'a' if $\delta(q_i, a) = \{q_j, q_k, q_l\} \in F$ then
$\delta'(q_i,a) = [q_j,q_k,q_l] \in F'$ (single state)

Example: Obtain the DFA for the following NFA.

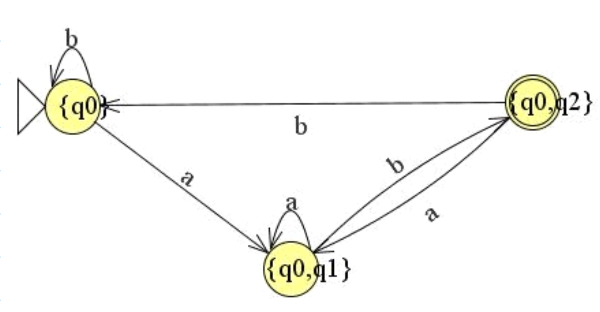


NFA Table:

δ	a	b
q0	{q0, q1}	q0
q1	ϕ	q2
q2	Φ	ϕ

DFA Table:

δ'	a	b
[q_0']	[q0,q1]	[q0']
[q0,q1]	[q0,q1]	[q0,q2]
[q0,q2]	[q0,q1]	[q0]



To prove the equivalence relationship between DFA and NFA, we use the principle of mathematical induction.
Induction principle is used to prove that something is always true.
For e.g., Real-life example
The Sun came yesterday, The Sun came today and the Sun will come tomorrow.
After one year, Sun will come
After 10 years, Sun will come
i.e., The Sun will come as long as the Universe exists and nothing wrong happens.
Programming example:
Recursive definition of factorial of a number
n! = n * (n-1)!
0!=1
1!=1
2!=2*(2-1)!
3!=3*(3-1)!

n!=n*(n-1)!

(n+1)!=(n+1) * n!

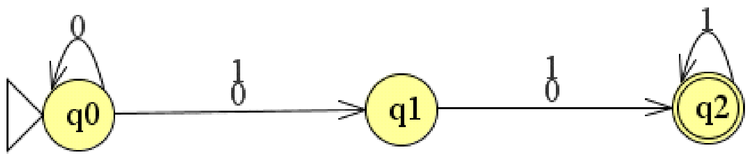
If we find factorial upto number n, then we can also find factorial for number (n+1)

Another example: if we run a program calculating the sum of two numbers, and if we run the program 10 times successfully, and if we want to run the program 11th time, we expect it will run successfully.

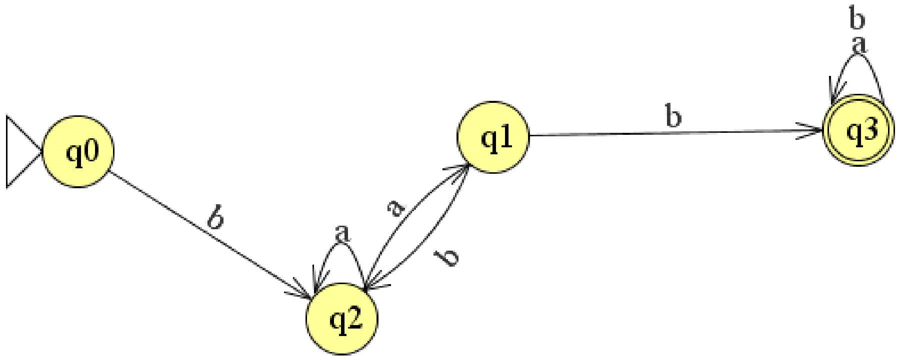
Similarly, it is always true that for any given NFA, there exists an equivalent DFA.

NFA to DFA Problems

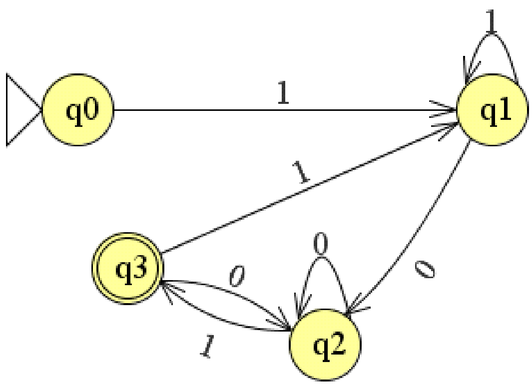
24 July 2017 05:37 AM

1. 

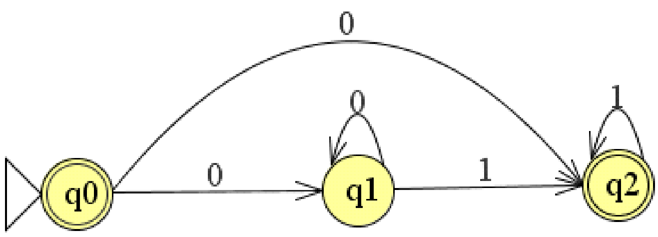
1.



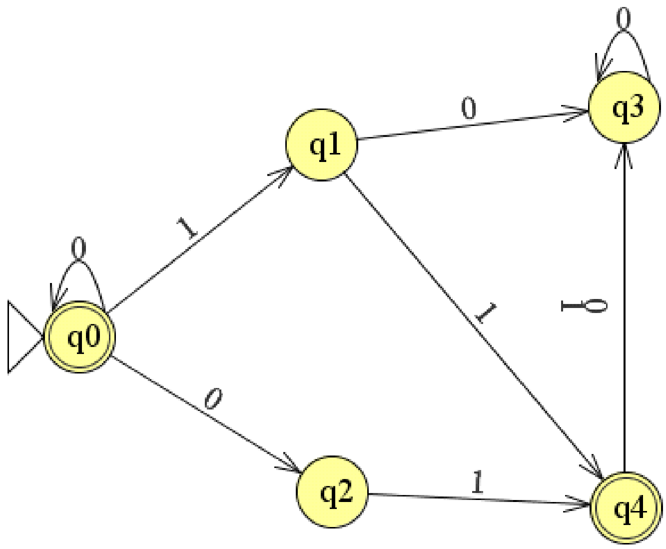
2.



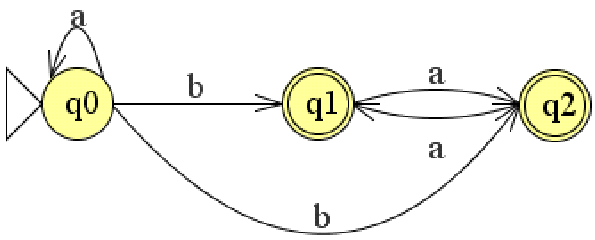
3.



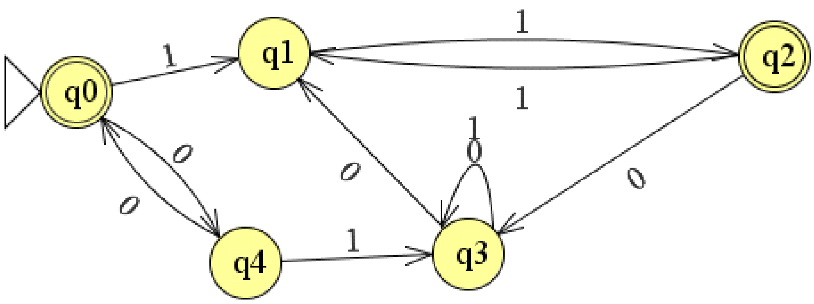
4.


0
0
1
q1
0
q3
1
0
q0
0
q2
1
q4

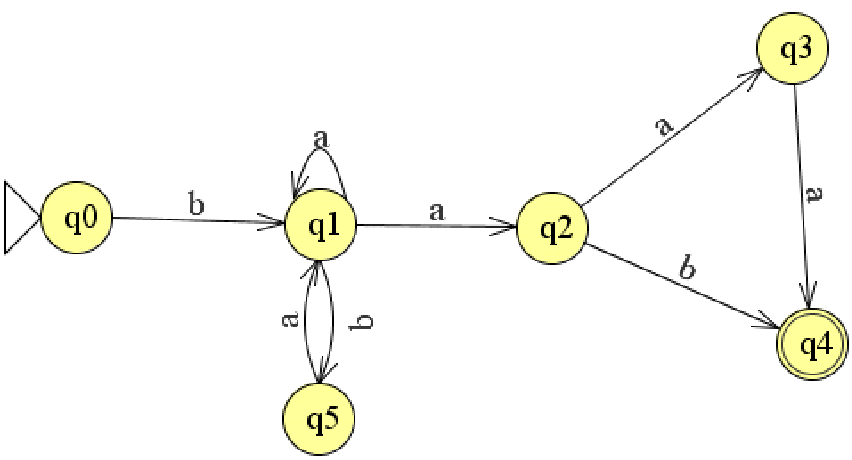
5.


a
q0
b
q1
a
q2
a
b

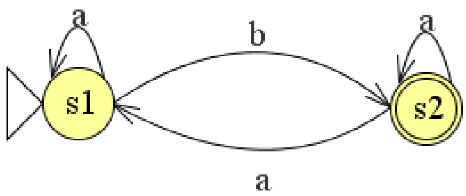
6.


1
q1
1
q2
q0
1
0
1
0
0
0
0
q4
1
q3

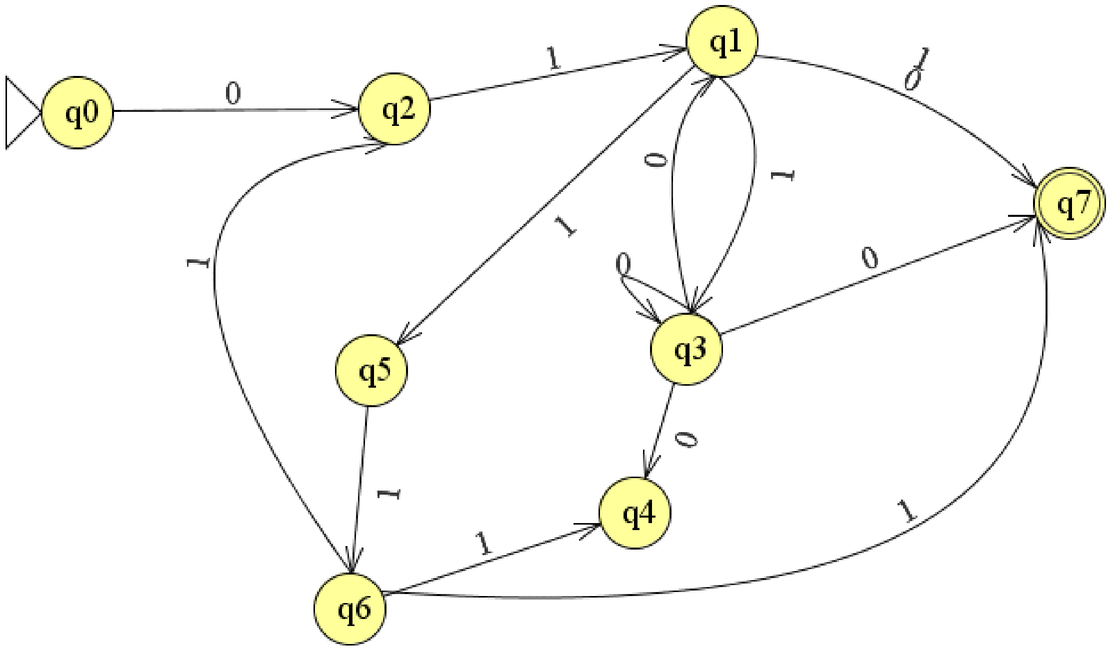
7.


q3
a
a
a
q0
b
q1
a
q2
b
a
b
q4
q5

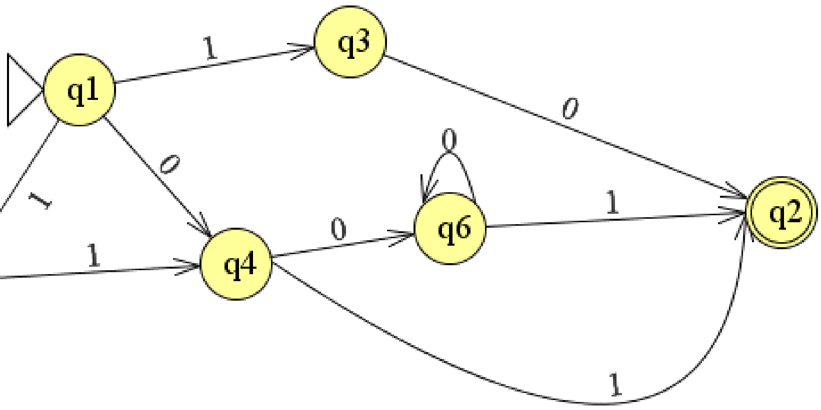
8.



9.



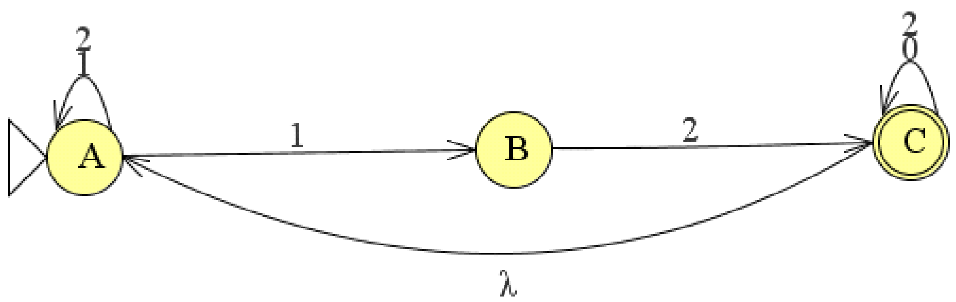
10.



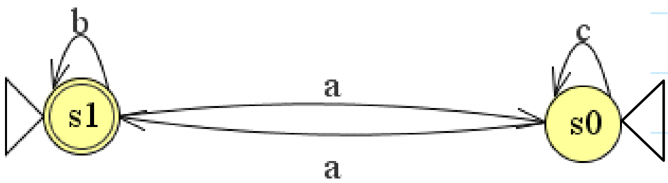
NFA-λ to DFA Problems

24 July 2017 03:18 PM

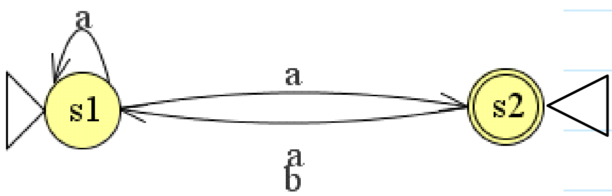
1.



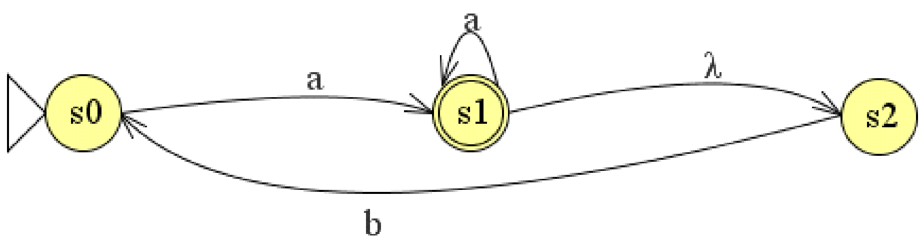
2.



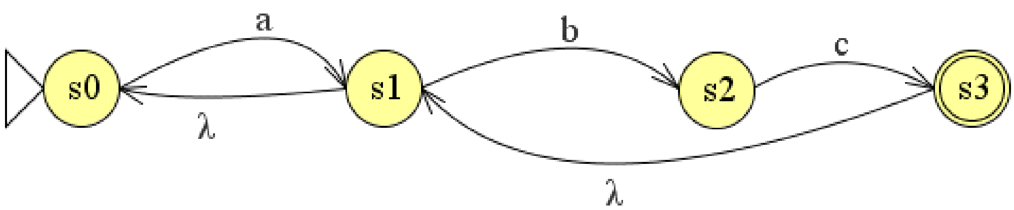
3.



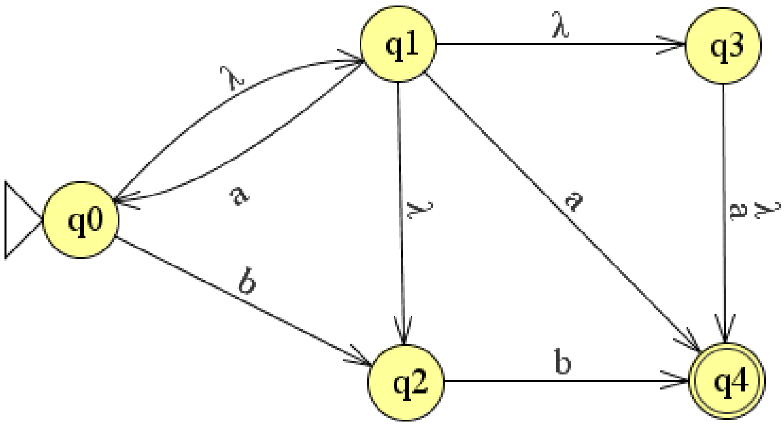
4.



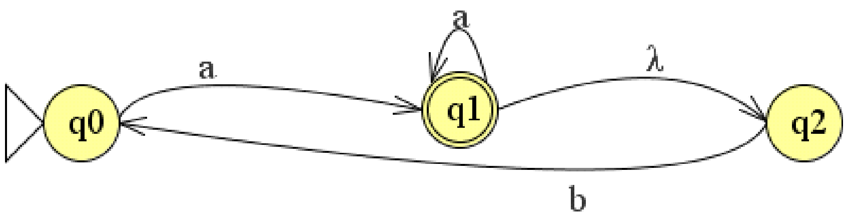
5.



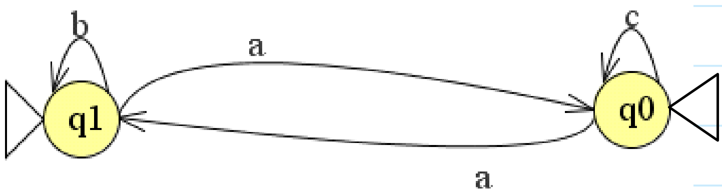
6.



7.



8.

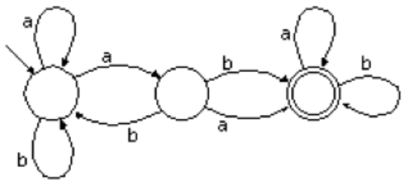


NFA Implementation

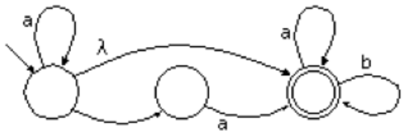
16 July 2018 10:46 PM

Nondeterministic Finite State Automata

A finite-state automaton can be *nondeterministic* in either or both of two ways:



A state may have two or more arcs emanating from it labeled with the same symbol. When the symbol occurs in the input, either arc may be followed.

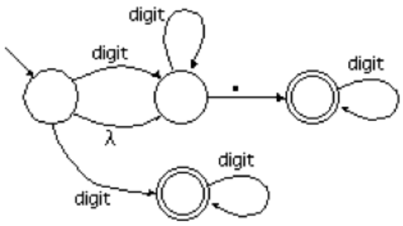


A state may have one or more arcs emanating from it labeled with λ (the empty string) . These arcs may optionally be followed without looking at the input or consuming an input symbol.

Due to nondeterminism, the same string may cause an nfa to end up in one of several different states, some of which may be final while others are not. The string is accepted if **any** possible ending state is a final state.

Example NFAs

Integer with optional sign.



Integer or real number.

Implementing an NFA

If you think of an automaton as a computer, how does it handle nondeterminism? There are two ways that this could, in theory, be done:

1. When the automaton is faced with a choice, it always (magically) chooses correctly. We sometimes think of of the automaton as consulting an *oracle* which advises it as to the correct choice.
2. When the automaton is faced with a choice, it spawns a new process, so that all possible paths are followed simultaneously.

The first of these alternatives, using an oracle, is sometimes attractive mathematically. But if we want to write a program to implement an nfa, that isn't feasible.

There are three ways, two feasible and one not yet feasible, to simulate the second alternative:

1. Use a recursive backtracking algorithm. Whenever the automaton has to make a choice, cycle through all the alternatives and make a recursive call to determine whether any of the alternatives leads to a solution (final state).
2. Maintain a state set or a state vector, keeping track of *all* the states that the nfa could be in at any given point in the string.
3. Use a *quantum computer*. Quantum computers explore literally all possibilities simultaneously. They are theoretically possible, but are at the cutting edge of physics. It may (or may not) be feasible to build such a device.

Recursive Implementation of NFAs

An nfa can be implemented by means of a recursive search from the start state for a path (directed by the symbols of the input string) to a final state.

Here is a rough outline of such an implementation:

```
function nfa (state A) returns Boolean:
    local state B, symbol x;
    for each λ transition from state A to some state B do
        if nfa (B) then return True;
    if there is a next symbol then
        { read next symbol (x);
          for each x transition from state A to
            some state B do
                if nfa (B) then
                    return True;
          return False;
        }
    else
        { if A is a final state then return True;
          else return False;
        }
```

One problem with this implementation is that it could get into an infinite loop if there is a cycle of λ transitions. This could be prevented by maintaining a simple counter (How?).

State-Set Implementation of NFAs

Another way to implement an NFA is to keep either a *state set* or a *bit vector* of all the states that the NFA could be in at any given time. Implementation is easier if you use a bit-vector approach (v[i] is True iff state i is a possible state), since most languages provide vectors, but not sets, as a built-in datatype. However, it's a bit easier to describe the algorithm if you use a state-set approach, so that's what we will do. The logic is the same in either case.

```
function nfa (state set A) returns Boolean:
   local state set B, state a, state b, state c, symbol x;

   for each a in A do
      for each λ transition from a
        to some state b do
           add b to B;
   while there is a next symbol do
      { read next symbol (x);
        B := ϕ;
        for each a in A do
          { for each λ transition from a to some state b do
                 add b to B;
            for each x transition from a to some state b do
```

```
            add b to B;
          }
        for each λ transition from
           some state b in B to some state c not in B do
              add c to B;
        A := B;
      }
   if any element of A is a final state then
      return True;
   else
      return False;
```

Formal Definition of NFAs

The extension of our notation to NFAs is somewhat strained.

A *nondeterministic finite acceptor* or *nfa* is defined by the quintuple

$$M = (Q, \Sigma, \delta, q0, F)$$

where

- Q is a finite set of *states,*
- Σ is a finite set of symbols, the *input alphabet,*
- $\delta: Q \times (\Sigma \cup \{ \lambda \}) \to 2^Q$ is a *transition function,*
- $q0 \in Q$ is the *initial state,*
- $F \subseteq Q$ is a set of *final states.*

These are all the same as for a dfa except for the definition of δ:

- Transitions on λ are allowed in addition to transitions on elements of Σ, and
- The range of δ is 2^Q rather than Q. This means that the values of δ are not elements of Q, but rather are sets of elements of Q.

The language defined by nfa M is defined as

$$L(M) = \{w \in \Sigma^*: \delta^*(q0, w) \cap F \neq \phi\}$$

DFA = NFA

Two acceptors are *equivalent* if the accept the same language.

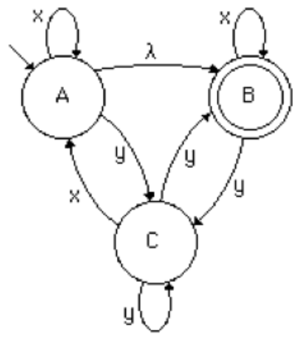
A DFA is just a special case of an NFA that happens not to have any null transitions or multiple transitions on the same symbol. So DFAs are not more powerful than NFAs.

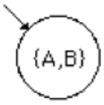
For any NFA, we can construct an equivalent DFA (see below). So NFAs are not more powerful than DFAs. DFAs and NFAs define the same class of languages -- the *regular* languages.

To translate an NFA into a DFA, the trick is to label each state in the DFA with a *set of states* from the NFA. Each state in the DFA summarizes all the states that the NFA might be in. If the NFA contains |Q| states, the resultant DFA could contain as many as $|2^Q|$ states. (Usually far fewer states will be needed.)

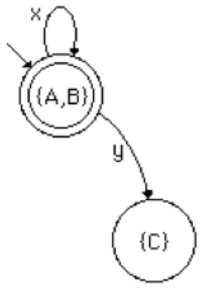
From NFA to DFA

Consider the following NFA:



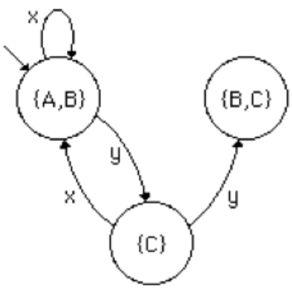


What states can we be in (in the NFA) before reading any input? Obviously, the start state, A. But there is a λ transition from A to B, so we could also be in state B. For the DFA, we construct the composite state {A, B}.



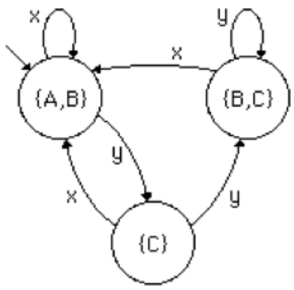
State {A,B} lacks a transition for x. From A, x takes us to A (in the NFA), and the null transition might take us to B; from B, x takes us to B. So in the DFA, x takes us from {A,B} to {A,B}.

State {A,B} also needs a transition for y. In the NFA, δ(A,y)=C and δ(B,y)=C, so we need to add a state {C} and an arc y from {A,B} to {C}.



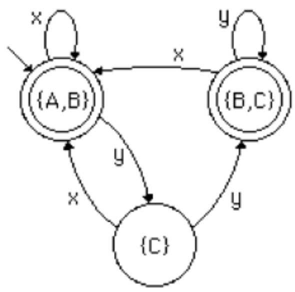
In the NFA, δ(C,x)=A, but then a null transition might or might not take us to B, so we need to add an arc x from {C} to {A,B}.

Also, there are two arcs from C labeled y, going to states B and C. So in the DFA we need to add the state {B,C} and the arc y from {C} to this new state.



In the NFA, δ(B,x)=B and δ(C,x)=A (and by a λ transition we might get back to B), so we need an x arc from {B,C} to {A,B}.

δ(B,y)=C, while δ(C,y) is either B or C, so we have an arc labeled y from {B,C} to {B,C}.



We now have a transition from every state for every symbol in Σ. The only remaining chore is to mark all the final states. In the original NFA, B was a final state, so in the DFA, every state containing B is a final state.

DFA Minimization

25 July 2017 03:04 PM

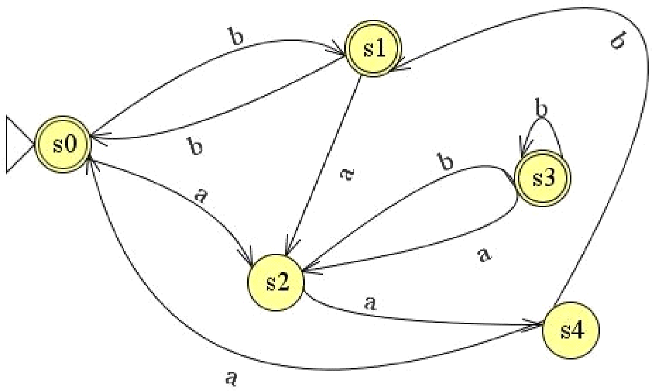
While implementing DFA, the amount of memory required is directly proportional to number of states in DFA.
To save memory space, it is important to minimize DFA (reduce no. of states in DFA).

- DFA minimization is based on the property of equivalence of states.
  Two states, say s1 and s2 of an finite automaton M are equivalent if for any x ϵ Σ*,
  δ*(s1,x) = δ*(s2,x) = t ϵ Q
  That is , for any input string both the states must reach to the same state t.
- Two states S1 and S2 are 0-equivalent if they have the same output, that is, either both are accepting states or both are non-accepting states.
- Two states S1 and S2 are 1-equivalent if they have the same output (i.e., they are 0-equivalent) and for each input symbol, their succeeding states are also 0-equivalent.
- Two states S1 and S2 are k-equivalent if for any x ϵ Σ*, where x has no more than k symbols, δ*(s1,x) = δ*(s2,x)

0-equivalent states:

{s0, s1, s3} (set of final states)	{s2, s4} (set of non-final states)

Example:



DFA
minimizat...

δ	a	b
S0	S2	S1
S1	S2	S0
S3	S2	S3

All the states S0, s1, s3 are Going to same Group of Non-final states

All the states S0, s1, s3 are Going to same Final states Group of

Therefore, States s0, s1 and s3 are 1-equivalent.

δ	a	b
S2	S4	S3
S4	S0	S1

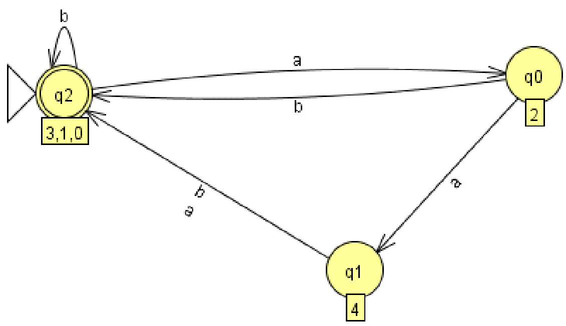
Different groups

Same group

Threfore, states s2 and s4 are not 1-equivalent.

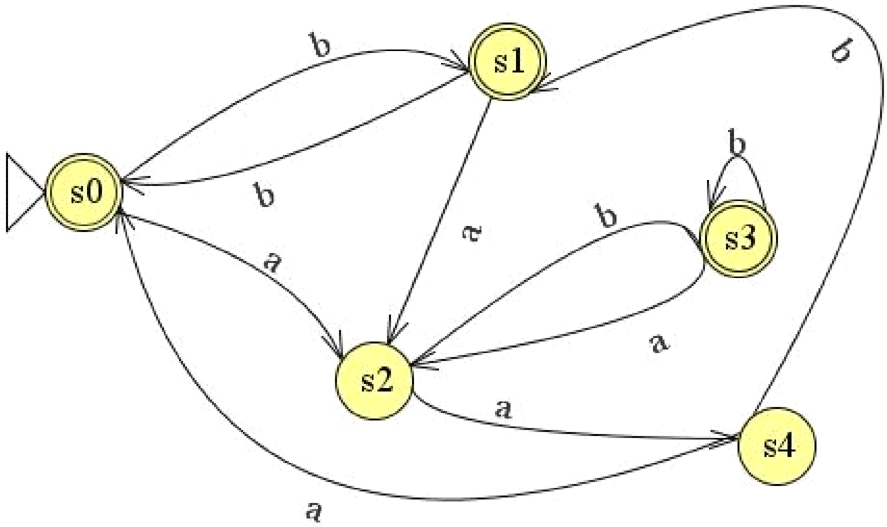
The final groups of states are:

{s0, s1, s3} {s2} {s4}

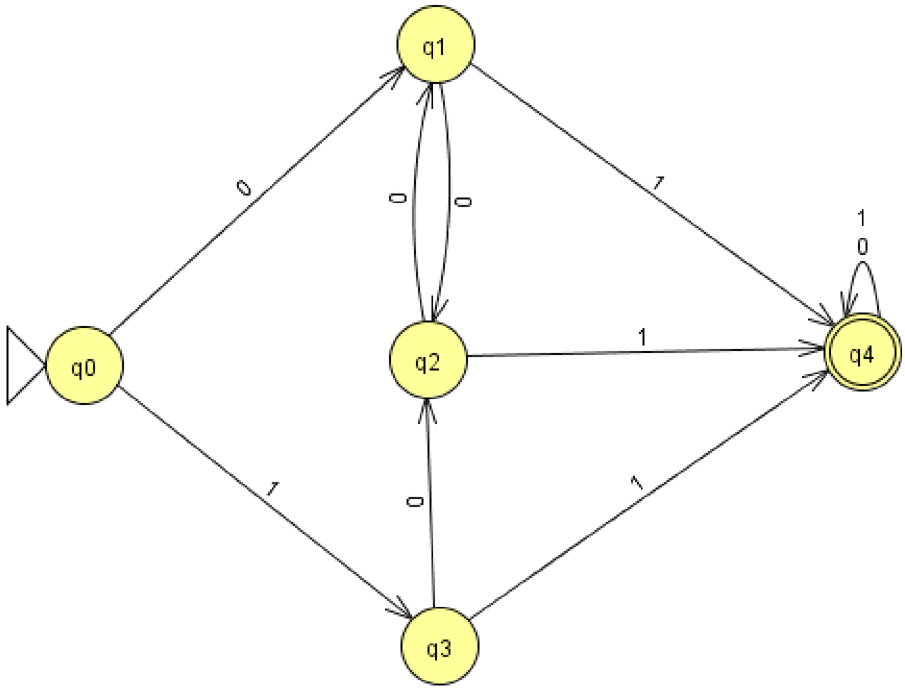


DFA minimization problems

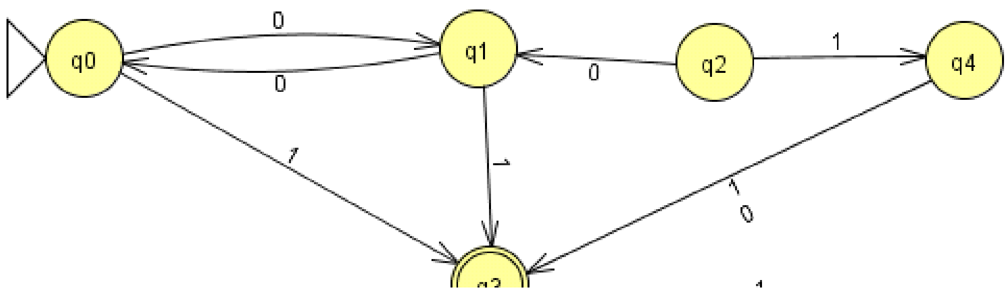
27 July 2017 12:39 PM

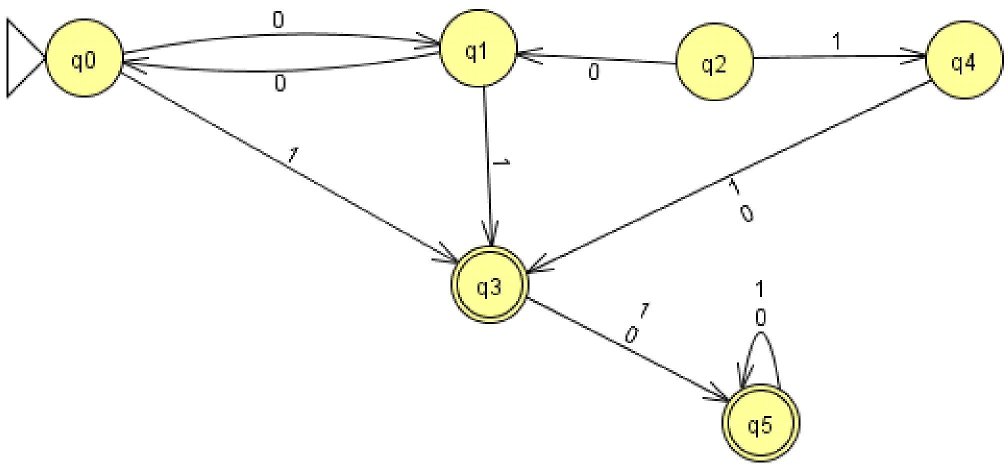


1.

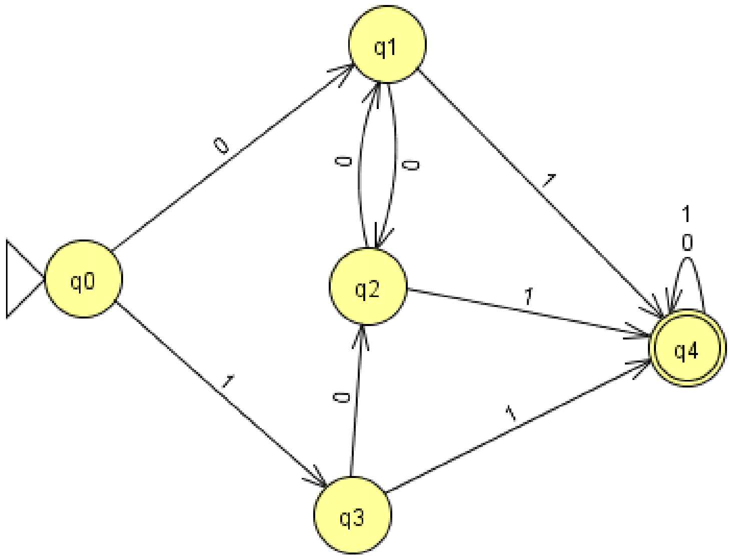


2.

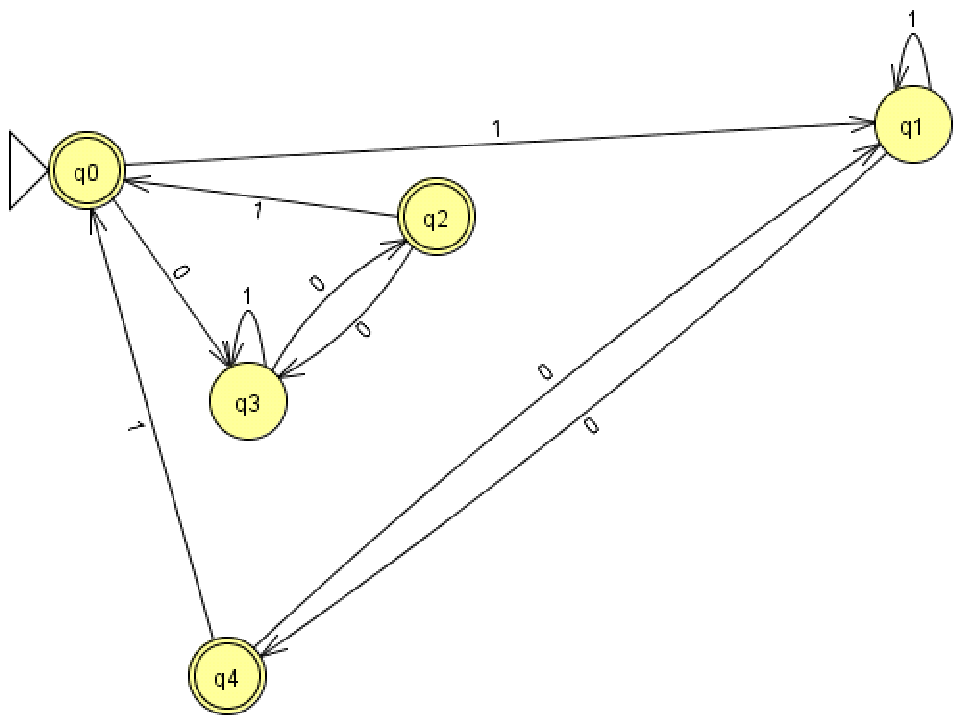




3.



4.



5.

δ	a	b	Output
→A	B	C	0
B	F	D	0
C	G	E	0
D	H	B	0
E	B	F	1
F	D	H	0
G	E	B	0
H	B	C	1

6.

δ	a	b
→A	B	E
B	C	F
C	D	H
D	E	H
E	F	I
F	G	B
G	H	B
H	I	C
I	A	E

Moore & Mealy machines

27 July 2017 06:33 PM

Finite State Machines

- **FA with output (Transducers)**
  - **Moore machines**
  - **Mealy machines**
- **FA without output (Acceptors)**
  - **DFA**
  - **NFA**

Machines producing binary output are not much significant. The machine is considered as efficient if it produces output other than binary output.

★ All the problems do not have the answer as "yes" or "no". There are few problems that have answer other than "yes" or "no".

Ex: Do you come to movie? (Answer is either yes or no)
What is your name? (Vijay; answer is other than yes or no)

Given L={binary strings ending with 01}
Does w=010101101 belong to L? (Answer is yes)
Does x=01110 belong to L? (Answer is no)

What is the 1's complement of 0100111? (Answer is 1011000; other than yes or no)

Moore machine formal definition: M = (Q, Σ, δ, q0, Δ, Γ) where

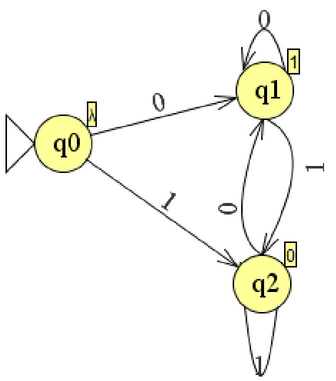
Q is set of states,
Σ is set of input symbols,
δ is state-transition function defined as δ: Q X Σ → Q
q0 is initial state
Δ is set of output symbols and
Γ is output function mapping Q into Δ

Mealy machine formal definition: M = (Q, Σ, δ, q0, Δ, Γ) where

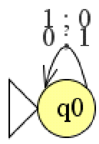
Q is set of states,
Σ is set of input symbols,
δ is state-transition function defined as δ: Q X Σ → Q
q0 is initial state
Δ is set of output symbols and
Γ is output function mapping Q X Σ into Δ

In Moore machine, output is associated with state.
In Mealy machine, output is associated with transition.

Example: Moore machine to calculate 1's complement of binary string.



Example: Mealy machine to calculate 1's complement of binary string.



For an input string of "n" symbols,
Moore machine produce output string of "n+1" symbols → (because of output symbol associated with initial state)
Mealy machine produce output string of "n" symbols.

Moore & Mealy machines problems

27 July 2017 07:45 PM

1. Design Moore and Mealy machines to get 1's complement of binary number.
2. Design Mealy machine for the following table and also find the output for the string "abbabaaa".

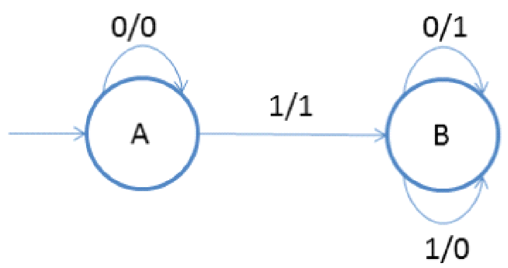
δ	a	b	o/p
q0	q1	q2	1
q1	q1	q1	0
q2	q1	q0	1

3. Design Moore and Mealy machines that give an output '1' if input of binary sequence a '1' is preceded by exactly two zero's.
4. Design a Moore machine such that it produces output A if the string ends with 10, B if the string ends with 11 and C otherwise.

5. Design a Moore and Mealy machines for a binary input sequence, if it ends in 101, output is 'A', if it ends in '110' output is B, otherwise 'C'.
6. Design Moore and Mealy machines that replace each occurrence of substring 100 by 101.
7. Design Moore and Mealy machines that print residue modulo of 2,3,4, and 5 for the given binary number.
8. Design Moore and Mealy machines that print residue modulo of 2,3,4, and 5 for the given decimal number.
9. Design a Mealy machine which can give output Even, Odd according to the total number of 1's encountered is even or odd. The input symbols are 0 and 1.
10. Design Mealy machine to count how many times the substring 'aab' occurs in a string.
11. Design a mealy machine to print two's complement of binary number. (Assume input and output are taken from right to left).
12. Design a Mealy machine to perform a 3-bit odd parity check on the input string. If the total number of 1-bits in the input string is even, the total number of 1-bits of the string will be odd.

Design a mealy machine for 2's complement

We will design mealy machine for 2's complement.



Generally we take 2's complement as follows:

1. Take 1's complement of the input
2. Add 1 to step 1

But here we are taking 2's complement in a different manner to design mealy machine.

The approach goes as follows:

1. Start from **right to left**
2. Ignore all 0's
3. When 1 comes ignore it and then take 1's complement of every digit

Example

1. Lets take 001 and we know that its 2's complement is (110+1 = 111)
2. So scan from right to left
3. On state A '1' came first to go to stage B and in output write 1
4. On state B replace '0' with '1' and vice-versa
5. So finally we got 111 as output
6. Be aware that the output is also printed in **right to left order**

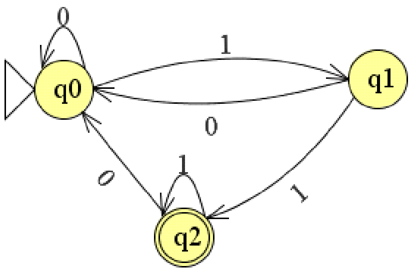
UNIT - II

30 June 2017 07:14 PM

Regular Expressions

03 August 2017 12:12 PM

A Regular Language has different notations: Set Notation, Graph Notation, Tabular Notation.
Example: Language of all strings of 0's and 1's ending with 11.
Set Notation: L = {x ϵ {0, 1}* | x ends with 11}
Graph Notation:



Tabular Notation:

δ	0	1
→q0	q0	q1
q1	q0	q2
q2	q0	q2

Now we see another notation: Regular Expression r = (0 + 1)*11 (Among four notations, which one is better and why?)

Regular Expression is the short and practical notation to describe the regular language.

Regular expression

Definition: A regular expression is recursively defined as follows.

1. ϕ is a regular expression denoting an empty language.
2. ε-(epsilon) is a regular expression indicates the language containing an empty string.
3. *a* is a regular expression which indicates the language containing only {a}
4. If R is a regular expression denoting the language L_R and S is a regular expression denoting the language L_S, then
   a. R+S is a regular expression corresponding to the language $L_R U L_S$.
   b. R.S is a regular expression corresponding to the language $L_R.L_S$.
   c. R* is a regular expression corresponding to the language L_R^*.
5. The expressions obtained by applying any of the rules from 1-4 are regular expressions.

Regular	Meaning

expressions	
(a+b)*	Set of strings of a's and b's of any length including the NULL string.
(a+b)*abb	Set of strings of a's and b's ending with the string abb
ab(a+b)*	Set of strings of a's and b's starting with the string ab.
(a+b)*aa(a+b)*	Set of strings of a's and b's having a sub string aa.
a*b*c*	Set of string consisting of any number of a's(may be empty string also) followed by any number of b's(may include empty string) followed by any number of c's(may include empty string).
$a^+b^+c^+$	Set of string consisting of at least one 'a' followed by string consisting of at least one 'b' followed by string consisting of at least one 'c'.
aa*bb*cc*	Set of string consisting of at least one 'a' followed by string consisting of at least one 'b' followed by string consisting of at least one 'c'.
(a+b)* (a + bb)	Set of strings of a's and b's ending with either *a* or *bb*
(aa)*(bb)*b	Set of strings consisting of even number of a's followed by odd number of b's
(0+1)*000	Set of strings of 0's and 1's ending with three consecutive zeros(or ending with 000)
(11)*	Set consisting of even number of 1's

Regular Expression	Meaning
λ or ϵ	Empty string
a + b	String of length one (exactly)
(a + b) (a + b) or $(a + b)^2$	Strings of a's and b's of length 2 (exactly)
(a + b) (a + b) (a + b) or $(a + b)^3$	Strings of a's and b's of length 3 (exactly)
$(a + b)^{10}$	Strings of a's and b's of length 10 (exactly)
(λ + a + b) (λ + a + b) or $(λ + a + b)^2$	Strings of a's and b's of length atmost 2
$(λ + a + b)^{10}$	Strings of a's and b's of length atmost 10
(a + b)*	Strings of a's and b's of any length n $\geq$ 0
$(a + b)^+$	Strings of a's and b's of any length n $\geq$ 1
(a + b)*abb	Strings of a's and b's ending with abb
ab(a + b)*	Strings of a's and b's starting with ab
(a + b)*aab(a + b)*	Strings of a's and b's containing substring aab
a*b*c*	Any no.of a's followed by any no. of b's followed by any no. of c's
$a^+b^+c^+$	Atleast one a followerd by atleast one b followed by atleast one c
aa*bb*cc*	Atleast one a followerd by atleast one b followed by atleast one c
(a + bb) (a + b)*	Starting with either a or bb
(a + b)* (a + bb)	Ending with either a or bb
(a + b)* (a + bb) (a + b)*	Containing substring either a or bb
((a + b)(a + b))*	Even length strings
((a + b)(a + b) (a + b))*	String length divisible by 3

((a + b)(a + b))* (a + b)	Odd length strings
(a + b)* aaa (a + b)*	Strings with three consecutive a's
(b + ab)* (a + λ)	No two consecutive a's
a (a + b) b	Starting with a and ending with b
(a + b)*a(a + b)	Second symbol from right end is a
(a + b)*a$(a + b)^9$	Tenth symbol from right end is a
(aa)*(bb)*b	Even a's followed by odd b's

Simplification of Regular Expressions

05 August 2017 05:32 AM

Different regular expressions can be written for the same language.
For example, the language of all strings of a's and b's with atleast one a followed by atleast one b, there are two regular expressions such as aa*bb* and a^+b^+

As the regular expressions are used practically, it is important to write them with few symbols as possible. In other words, it is required to simplify the regular expressions.
To simplify the regular expressions, there are few identity rules to be followed:

Identity Rules of Regular Expressions.

(a) $r_1 + \phi = r_1$ (b) $r_1 . \lambda = r_1 = \lambda . r_1$

(c) $r_1 . \phi = \phi = \phi . r_1$ (d) $r_1 + r_2 = r_2 + r_1$

(e) $r_1 + r_1 = r_1$ (f) $r_1 + (r_2 + r_3) = (r_1 + r_2) + r_3$

(g) $r_1 . (r_2 . r_3) = (r_1 . r_2) . r_3$ (h) $r_1 . (r_2 + r_3) = r_1 . r_2 + r_1 . r_3$

(i) $\lambda^* = \lambda$ (j) $\phi^* = \lambda$

(k) $(r_1 + r_2)^* = (r_1^* + r_2^*)^*$ (l) $(r_1 + r_2)^* = (r_1^* r_2^*)^*$

(m) $(r_1^*)^* = r_1^*$ (n) $r_1^* . r_1^* = r_1^*$ (u) $RR^* = R^*R$
$= R^+$

(o) $r_1 + \lambda \neq r_1$ (p) $r_1 . r_2 \neq r_2 . r_1$ (v) $\lambda + RR^* = R^*$
$R^* + \lambda$

(q) $r_1 . r_2 \neq r_1$ (r) $r_1 + (r_2 . r_3) \neq (r_1 + r_2) . (r_1 + r_3)$

(s) $(r_1 . r_2)^* \neq (r_1^* . r_2^*)^*$ (t) $(r_1 . r_2)^* \neq (r_1^* + r_2^*)^*$

——— x ———

Arden's Theorem: Let P and Q be two regular expressions over Σ and if P does not contain λ then $R = Q + RP$ has a unique solution
$R = QP^*$

Proof:

The following theorem is very much useful in simplifying regular expressions (i.e. replacing a given regular expression **P** by a simpler regular expression equivalent to **P**).

Theorem 5.1 (Arden's theorem) Let **P** and **Q** be two regular expressions over Σ. If **P** does not contain Λ, then the following equation in **R**, namely

$$\mathbf{R} = \mathbf{Q} + \mathbf{RP} \tag{5.1}$$

has a unique solution (i.e. one and only one solution) given by $\mathbf{R} = \mathbf{QP}^*$.

Proof $\mathbf{Q} + (\mathbf{QP}^*)\mathbf{P} = \mathbf{Q}(\Lambda + \mathbf{P}^*\mathbf{P}) = \mathbf{QP}^*$ by I_9

Hence (5.1) is satisfied when $\mathbf{R} = \mathbf{QP}^*$. This means $\mathbf{R} = \mathbf{QP}^*$ is a solution

The following theorem is very much useful in simplifying regular expressions (i.e. replacing a given regular expression **P** by a simpler regular expression equivalent to **P**).

Theorem 5.1 (Arden's theorem) Let **P** and **Q** be two regular expressions over Σ. If **P** does not contain Λ, then the following equation in **R**, namely

$$\mathbf{R} = \mathbf{Q} + \mathbf{RP} \tag{5.1}$$

has a unique solution (i.e. one and only one solution) given by $\mathbf{R} = \mathbf{QP}^*$.

Proof $\mathbf{Q} + (\mathbf{QP}^*)\mathbf{P} = \mathbf{Q}(\Lambda + \mathbf{P}^*\mathbf{P}) = \mathbf{QP}^*$ by I_9

Hence (5.1) is satisfied when $\mathbf{R} = \mathbf{QP}^*$. This means $\mathbf{R} = \mathbf{QP}^*$ is a solution of (5.1).

To prove uniqueness, consider (5.1). Here, replacing **R** by **Q + RP** on the R.H.S., we get the equation

$$\begin{aligned}
\mathbf{Q} + \mathbf{RP} &= \mathbf{Q} + (\mathbf{Q} + \mathbf{RP})\mathbf{P}\\
&= \mathbf{Q} + \mathbf{QP} + \mathbf{RPP}\\
&= \mathbf{Q} + \mathbf{QP} + \mathbf{RP}^2\\
&= \mathbf{Q} + \mathbf{QP} + \mathbf{QP}^2 + \cdots + \mathbf{QP}^i + \mathbf{RP}^{i+1}\\
&= \mathbf{Q}(\Lambda + \mathbf{P} + \mathbf{P}^2 + \cdots + \mathbf{P}^i) + \mathbf{RP}^{i+1}
\end{aligned}$$

From (5.1),

$$\mathbf{R} = \mathbf{Q}(\Lambda + \mathbf{P} + \mathbf{P}^2 + \cdots + \mathbf{P}^i) + \mathbf{RP}^{i+1} \quad \text{for } i \geq 0 \tag{5.2}$$

We now show that any solution of (5.1) is equivalent to $\mathbf{QP}^*$. Suppose **R** satisfies (5.1), then it satisfies (5.2). Let w be a string of length i in the set **R**. Then w belongs to the set $Q(\Lambda + \mathbf{P} + \mathbf{P}^2 + \ldots + \mathbf{P}^i) + \mathbf{RP}^{i+1}$. As **P** does not contain Λ, $\mathbf{RP}^{i+1}$ has no string of length less than $i + 1$ and so w is not in the set $\mathbf{RP}^{i+1}$. This means that w belongs to the set $Q(\Lambda + \mathbf{P} + \mathbf{P}^2 + \ldots + \mathbf{P}^i)$, and hence to $\mathbf{QP}^*$.

Consider a string w in the set $\mathbf{QP}^*$. Then w is in the set $\mathbf{QP}^k$ for some $k \geq 0$, and hence in $Q(\Lambda + \mathbf{P} + \mathbf{P}^2 + \cdots + \mathbf{P}^k)$. So w is on the R.H.S. of (5.2). Therefore, w is in **R** (L.H.S. of (5.2)). Thus **R** and $\mathbf{QP}^*$ represent the same set. This proves the uniqueness of the solution of (5.1). ∎

Example 5.3

(a) Give an r.e. for representing the set L of strings in which every 0 is immediately followed by at least two 1's.

(b) Prove that the regular expression $\mathbf{R} = \Lambda + \mathbf{1}^*(\mathbf{011})^*(\mathbf{1}^*\,(\mathbf{011})^*)^*$ also describes the same set of strings.

Solution

(a) If w is in L, then either (a) w does not contain any 0, or (b) it contains a 0 preceded by 1 and followed by 11. So w can be written as $w_1w_2 \ldots w_n$, where each w_i is either 1 or 011. So L is represented by the r.e. $(\mathbf{1} + \mathbf{011})^*$.

(b) $\mathbf{R} = \Lambda + \mathbf{P}_1\mathbf{P}_1^*$, where $\mathbf{P}_1 = \mathbf{1}^*(\mathbf{011})^*$

$$\begin{aligned}
&= \mathbf{P}_1^* && \text{using } I_9\\
&= (\mathbf{1}^*(\mathbf{011})^*)^*\\
&= (\mathbf{P}_2^*\mathbf{P}_3^*)^* && \text{letting } \mathbf{P}_2 = \mathbf{1},\ \mathbf{P}_3 = \mathbf{011}\\
&= (\mathbf{P}_2 + \mathbf{P}_3)^* && \text{using } I_{11}\\
&= (\mathbf{1} + \mathbf{011})^*
\end{aligned}$$

EXAMPLE 5.4

Prove $(\mathbf{1 + 00^*1}) + (\mathbf{1 + 00^*1})(\mathbf{0 + 10^*1})^* (\mathbf{0 + 10^*1}) = \mathbf{0^*1}(\mathbf{0 + 10^*1})^*$.

Solution

L.H.S. $= (\mathbf{1 + 00^*1})(\Lambda + (\mathbf{0 + 10^*1})^* (\mathbf{0 + 10^*1})\Lambda$ using I_{12}

$= (\mathbf{1 + 00^*1})(\mathbf{0 + 10^*1})^*$ using I_9

$= (\Lambda + \mathbf{00^*})\mathbf{1}(\mathbf{0 + 10^*1})^*$ using I_{12} for $\mathbf{1 + 00^*1}$

$= \mathbf{0^*1}(\mathbf{0 + 10^*1})^*$ using I_9

$=$ R.H.S.

① Prove $(1+00^*1)+(1+00^*1)(0+10^*1)^*(0+10^*1) = 0^*1(0+10^*1)^*$

② $\lambda + 1^*(011)^*(1^*(011)^*)^* = (1+011)^*$

③ $b+ab^*+aa^*b+aa^*ab^* = a^*(b+ab^*)$

④ $ab^*a(a+bb^*a)^*b = a(b+aa^*b)^*aa^*b$

$\boxed{r_1^* = r_1^* r_1^* = (r_1^*)^* = r_1 + r_1^*}$

$\rightarrow R^* = R^*R^* = (R^*)^* = R+R^* = \lambda + RR^*$

$\rightarrow R^*R = RR^*$

$\rightarrow (R+S)^* = (R^*+S^*)^* = (R^*S^*)^* = (R^*S)^*R^* = R^*(SR^*)^*$

$\rightarrow R(SR)^* = (RS)^*R$

$\rightarrow (R^*S)^* = \lambda + (R+S)^*S$

$(RS^*)^* = \lambda + R(R+S)^*$

Simplify $aa(b^*+a) + a(ab^*+aa)$

$\Rightarrow aab^*+aaa+aab^*+aaa$

$\Rightarrow (aab^*+aaa)$

$\Rightarrow aa(b^*+a)$

Prove $b+ab^*+aa^*b+aa^*ab^* = a^*(b+ab^*)$ already done

$\Rightarrow (b+ab^*)+aa^*(b+ab^*)$

$\Rightarrow (\lambda+aa^*)(b+ab^*)$

$= a^*(b+ab^*)$

Prove $a^*(b+ab^*) = b+aa^*b^*$

LHS. $a^*(b+ab^*) = a^*b + a^*ab^*$

$= (\lambda+aa^*)b + a^*ab^*$ $\quad (R^* = \lambda + RR^*)$

$= b+aa^*b+a^*ab^*$

$= b+aa^*b+aa^*b^*$

$= b+aa^*(b+b^*)$

$= b+aa^*b^*$

④ $ab^*a(a+bb^*a)^*b = a(b+aa^*b)^*aa^*b$

$\Rightarrow ab^*a(a^*bb^*a)^*a^*b$ $\quad [\because (R+S)^* = (R^*S)^*R^*]$

$\Rightarrow ab^*(aa^*bb^*)^*aa^*b$ $\quad [R(SR)^* = (RS)^*R]$

$\Rightarrow a(b+aa^*b)^*aa^*b$ $\quad [R^*(SR^*)^* = (R+S)^*]$

$\Rightarrow a(b+aa^*b)^* aa^*b \quad [R^*(SR^*)^* = (R+S)^*$

Equivalence between RE and FA

08 August 2017 04:06 AM

Regular expression (RE) and Finite Automata (FA) are representations of regular languages.

A regular language can be described using both RE and FA.

When RE is given, how to find equivalent FA?

When FA is given, how to write equivalent RE?

RE to FA conversion

08 August 2017 04:10 AM

Theorem: Let R be a regular expression. Then there exists a finite automaton M = (Q, Σ, δ, q_0, A) which accepts L(R).

Proof: By definition, ϕ, ε and *a* are regular expressions. So, the corresponding machines to recognize these expressions are shown in figure 3.1.a, 3.1.b and 3.1.c respectively.

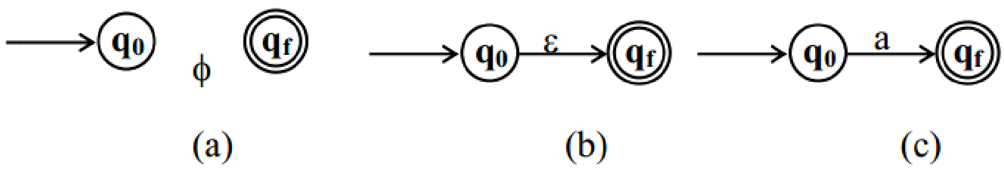


Fig 3.1 NFAs to accept ϕ, ε and a

The schematic representation of a regular expression R to accept the language L(R) is shown in figure 3.2. where q is the start state and f is the final state of machine M.

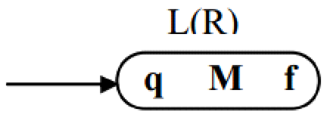


Fig 3.2 Schematic representation of FA accepting L(R)

In the definition of a regular expression it is clear that if R and S are regular expression, then R+S and R.S and R* are regular expressions which clearly uses three operators '+', '-' and '.'. Let us take each case separately and construct equivalent machine. Let $M_1 = (Q_1, \Sigma_1, \delta_1, q_1, f_1)$ be a machine which accepts the language $L(R_1)$ corresponding to the regular expression R_1. Let $M_2 = (Q_2, \Sigma_2, \delta_2, q_2, f_2)$ be a machine which accepts the language $L(R_2)$ corresponding to the regular expression R_2.

Case 1: $R = R_1 + R_2$. We can construct an NFA which accepts either $L(R_1)$ or $L(R_2)$ which can be represented as $L(R_1 + R_2)$ as shown in figure 3.3.

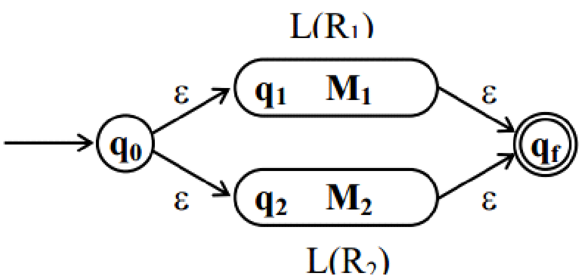


Fig. 3.3 To accept the language L(R1 + R2)

It is clear from figure 3.3 that the machine can either accept $L(R_1)$ or $L(R_2)$. Here, q_0 is the start state of the combined machine and q_f is the final state of combined machine M.

Case 2: $R = R_1 . R_2$. We can construct an NFA which accepts $L(R_1)$ followed by $L(R_2)$ which can be represented as $L(R_1 . R_2)$ as shown in figure 3.4.

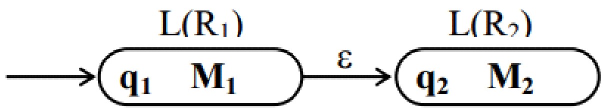
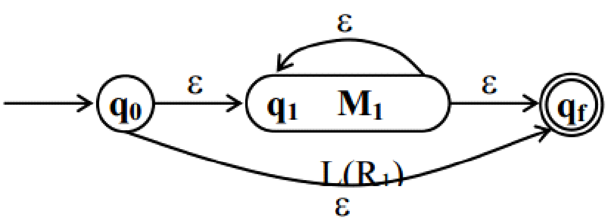


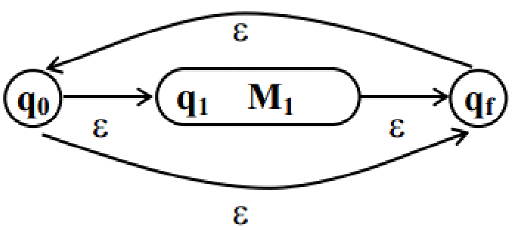
Fig. 3.4To accept the language L(R1 . R2)

It is clear from figure 3.4 that the machine after accepting $L(R_1)$ moves from state q_1 to f_1. Since there is a ε-transition, without any input there will be a transition from state f_1 to state q_2. In state q_2, upon accepting $L(R_2)$, the machine moves to f_2 which is the final state. Thus, q_1 which is the start state of machine M_1 becomes the start state of the combined machine M and f_2 which is the final state of machine M_2, becomes the final state of machine M and accepts the language $L(R_1.R_2)$.

Case 3: $R = (R_1)^*$. We can construct an NFA which accepts either $L(R_1)^*$) as shown in figure 3.5.a. It can also be represented as shown in figure 3.5.b.



(a)



(b)

Fig. 3.5 To accept the language $L(R1)^*$

It is clear from figure 3.5 that the machine can either accept ε or any number of $L(R_1)$s thus accepting the language $L(R_1)^*$. Here, q_0 is the start state q_f is the final state.

3.5.2 Direct Method for Conversion of r.e. to FA

This method is a direct method for obtaining FA from given regular expression. This is called a **subset** method. The method is given as below -

Step 1 : Design a transition diagram for given regular expression, using NFA with ε moves.

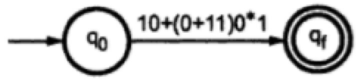
Step 2 : Convert this NFA with ε to NFA without ε.

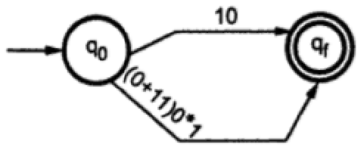
Step 3 : Convert the obtained NFA to equivalent DFA.

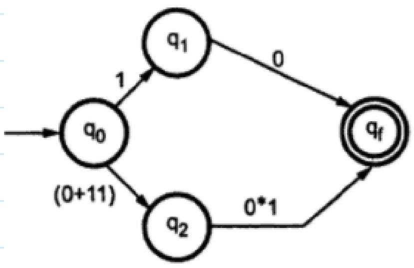
Let us understand this method with the help of some example.

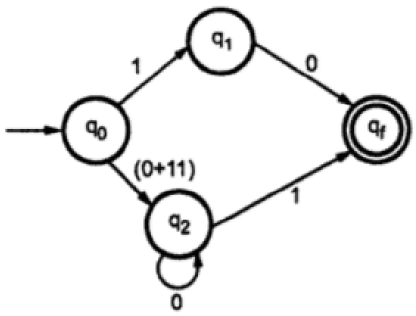
Example 3.34 : *Design a FA from given regular expression* $10 + (0 + 11)0^* 1$.

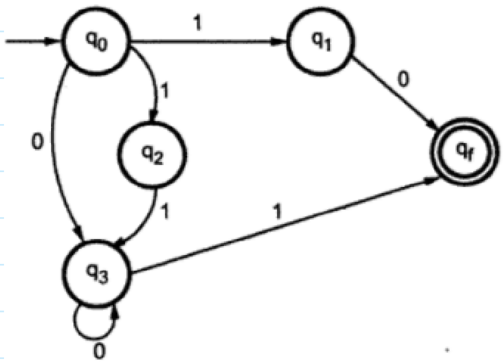
Solution : First we will construct the transition diagram for given regular expression.











Now we have got NFA without ε. Now we will convert it to required DFA for that, we will first write a transition table for this NFA.

State \ Input	0	1
q_0	q_3	$\{q_1, q_2\}$
q_1	q_f	ϕ
q_2	ϕ	q_3
q_3	q_3	q_f
q_f	ϕ	ϕ

The equivalent DFA will be

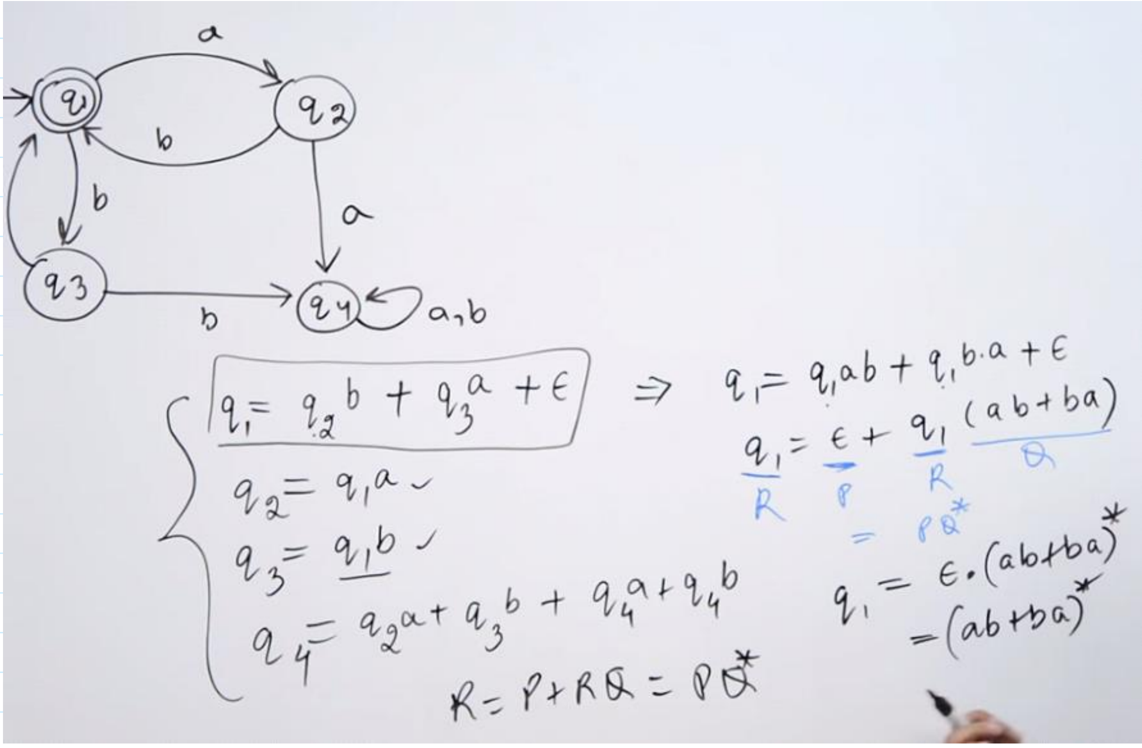
State \ Input	0	1
$[q_0]$	$[q_3]$	$[q_1, q_2]$
$[q_1]$	$[q_f]$	ϕ
$[q_2]$	ϕ	$[q_3]$
$[q_3]$	$[q_3]$	$[q_f]$
$[q_1, q_2]$	$[q_f]$	$[q_3]$
$[q_f]$	ϕ	ϕ

Practice Problems:

1. (0 + 1)* (00 + 11) (0 + 1)*
2. R = ba + (a + bb) a*b
3. r=(0 + 1)* (011)
4. r=10 + (00 + 11) 0*10
5. r= 0 + 11 + 101*0
6. r=(01 + 2*)* 1
7. r=0* + (01 + 0)*
8. r=(01 + 0)* (00 + 11)

FA to RE Conversion

17 August 2017 09:30 AM



Process of Constructing RE from FA:

There are some assumptions:

- In the transitional graph, there must be no epsilon moves.
- In the FA, there is only one initial state.

Now, we have to construct equations for all the states. There are n number of equations

If there are n states.

For any FA, these equations are constructed in the following way:

<state name> = Σ [< state name from which inputs are coming>, <input>]

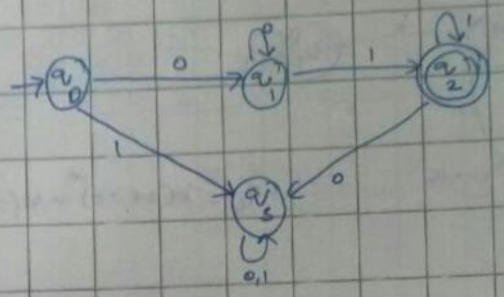
For the beginning state, there is an arrow at the beginning coming from no state. So, a λ is added with the equation of the beginning state.

Then, these equations have to be solved by the identities of RE. The expression obtained for the final state and consists of only the input symbol (Σ) is the RE for the FA.

Practice Problems:

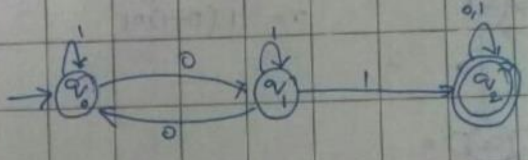
Arden's Theorem - Problems

(3)



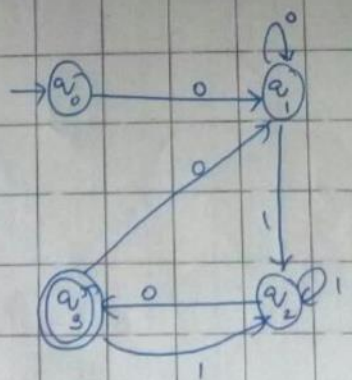
$\boxed{r = 00^*11^*}$

(4)

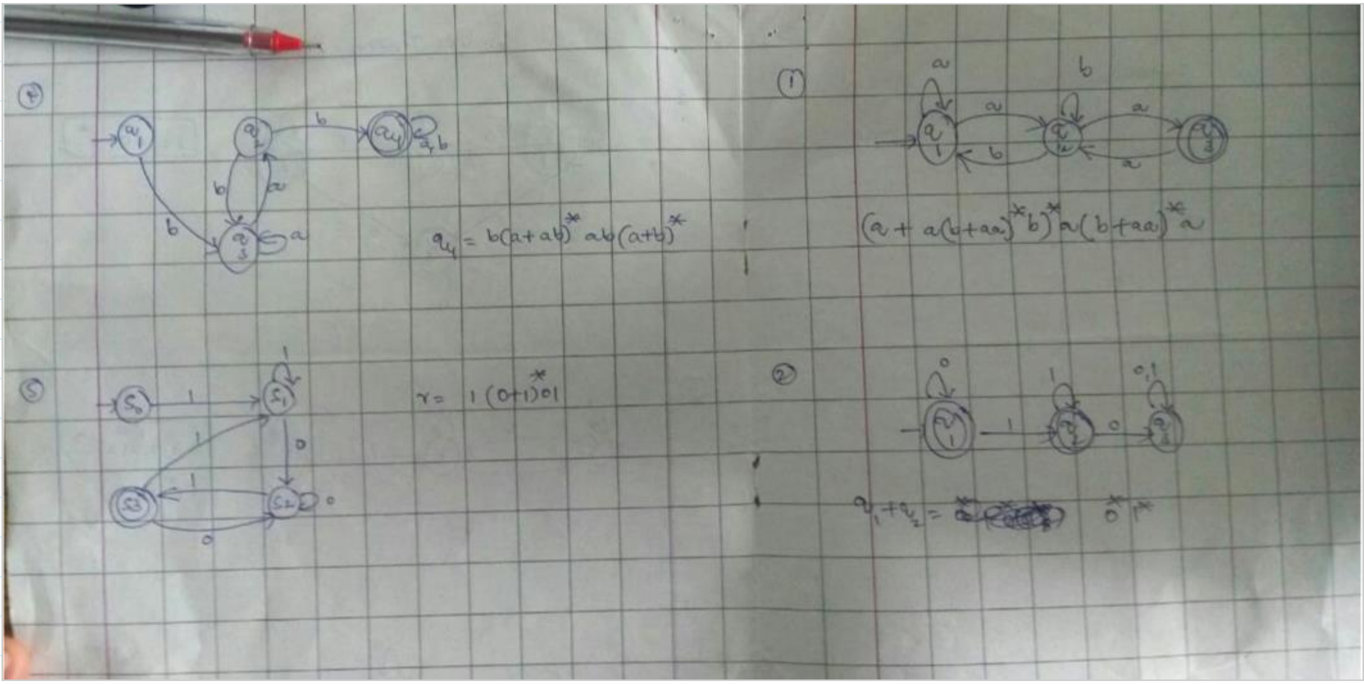


$\boxed{\begin{array}{l} r = (1+01^*0)^* \\ 01^*1(0+1)^* \end{array}}$

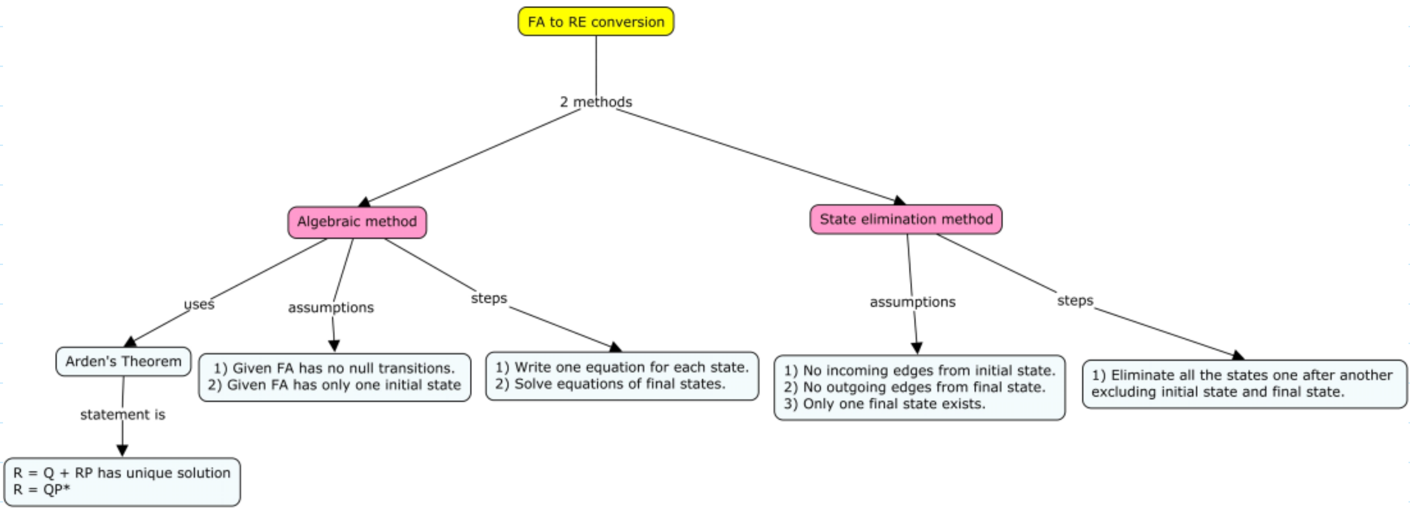
(5)



$\boxed{r = 0\{0+1(1+01)^*00\}^*\,1(1+01)^*0}$


q4 = b(a+ab)* ab(a+b)*
r = 1(0+1)*01
(a + a(b+aa)*b)*a(b+aa)*a

RE to FA


FA to RE conversion
2 methods
Algebraic method
State elimination method
uses
assumptions
steps
Arden's Theorem
1) Given FA has no null transitions.
2) Given FA has only one initial state
1) Write one equation for each state.
2) Solve equations of final states.
statement is
R = Q + RP has unique solution
R = QP*
assumptions
steps
1) No incoming edges from initial state.
2) No outgoing edges from final state.
3) Only one final state exists.
1) Eliminate all the states one after another excluding initial state and final state.

Pumping Lemma for Regular Languages

18 August 2017 06:05 AM

Proposition:Fish is a brain food Some upper cast XYZ community is brilliant because it eats Fish.

Me: If that's so,What about Fisherman's family members ? Because their main food is Fish. all the Fishermen's all sons/daughters should be brilliant scientists, Engineers/doctors/IAS/IFS/Software scientists may be in NASA, ISRO, Microsoft,Google,IISc etc. Which is obviously not. Many of them are still catching fish.
So, proposition is wrong. Fish is not brain food.

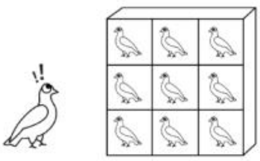
Let's assume i committed the crime.
But i was also out of town with my friends at the same same time.
Hence, i was at two locations at the same time.
This is absurd.
So, our original assumption is wrong.
I never committed the crime.

Pumping Lemma is based on Pigeonhole Principle.

The Pigeonhole Principle is one of the most obvious fundamentals in mathematics. It is so obvious that you may be surprised that there is even a name for it. It states that:

"If n items are put into m containers, with n > m, then at least one container must contain more than one item."

For those who prefer visuals and really hate math:



Even though the principle is simple it has been used to prove many complex mathematical theorems and lemmas. Here is one I find quite interesting:

"Incompressible strings of every length exist."

Alternatively,

"There is a file of every size that your favorite zip program can't compress."

The solution is left to the reader as an exercise.

Example 1: Among 367 people, there must be at least two with the same birthday.

BTW, what if there are 368 people?

Example 2: How many students must be in our class to guarantee that at least two of them receive the same score on the final exam?

Answer: Since there 101 possible scores, the class should have at least 102 students.

At least how many students in our class were born on the same day of the week?

The ***generalized pigeonhole principle***: If N objects are placed into k boxes, then there is at least one box containing at least $\lceil N/k \rceil$ objects.

Proof: Suppose none of the boxes contains $\lceil N/k \rceil$ or more objects.
Then every box contains at most $\lceil N/k \rceil - 1$ objects.
So, the total number of objects is at most $k(\lceil N/k \rceil - 1)$.
But $\lceil N/k \rceil - 1 < N/k$
Thus, the total number of objects is less than $k(N/k)$,
i.e. less than N.
This is a contradiction. End of proof.

How many students should be in our class to guarantee that at least 4 of them were born on the same day of the week?
$\lceil N/7 \rceil$ should be at least 4. So, N should be 22 or more.

Pumping Lemma (PL) for Regular Languages

Theorem:

Let L be a regular language. Then there exists a constant 'n' (which depends on L) such that for every string w in L such that $|w| \geq n$, we can break w into three strings, w=xyz, such that:

1. $|y| > 0$
2. $|xy| \leq n$
3. For all $k \geq 0$, the string xy^kz is also in L.

PROOF:

Let L be regular defined by an FA having 'n' states. Let w= a_1, a_2, a_3----a_n and is in L.

Pumping Lemma (PL) for Regular Languages

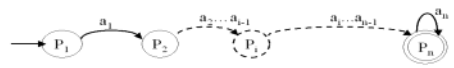
Theorem:

Let L be a regular language. Then there exists a constant 'n' (which depends on L) such that for every string w in L such that $|w| \geq n$, we can break w into three strings, w=xyz, such that:

1. $|y| > 0$
2. $|xy| \leq n$
3. For all $k \geq 0$, the string xy^kz is also in L.

PROOF:

Let L be regular defined by an FA having 'n' states. Let $w= a_1,a_2,a_3$----a_n and is in L.
$|w| = n \geq n$. Let the start state be P_1. Let w = xyz where $x= a_1,a_2,a_3$ -----a_{n-1}, $y=a_n$ and $z = \varepsilon$.



$\delta(P_1,a_1)=P_2$
$\delta(P_2,a_2)=P_3$
⋮
$\delta(P_n,a_n)=P_{n+1}$

But there are only n states. => there must be a loop. Let there be a loop in P_n State.

Let $x=a_1,\ldots\ldots a_{n-1}$
$y=a_n$
$z=\varepsilon$

Therefore $xy^kz = a_1$ ------ $a_{n-1}\,(a_n)^k\,\varepsilon$

k=0 a_1 ------ a_{n-1} is accepted

k=1 a_1 ------ a_n is accepted

k=2 a_1 ------ a_{n+1} is accepted

k=10 a_1 ------ a_{n+9} is accepted and so on.

Uses of Pumping Lemma: - This is to be used to show that, certain languages are not regular. It should never be used to show that some language is regular. If you want to show that language is regular, write separate expression, DFA or NFA.

General Method of proof: -

(i) Select w such that $|w| \geq n$
(ii) Select y such that $|y| \geq 1$
(iii) Select x such that $|xy| \leq n$
(iv) Assign remaining string to z
(v) Select k suitably to show that, resulting string is not in L.

Example 1.

To prove that $L=\{w|w \in a^nb^n$, where $n \geq 1\}$ is not regular

Proof:

Let L be regular. Let n is the constant (PL Definition). Consider a word w in L.
Let $w = a^nb^n$, such that $|w|=2n$. Since $2n > n$ and L is regular it must satisfy PL.

Consider $w= \overbrace{aa\text{-----}a}^{n}\ \overbrace{bb\text{-----}b}^{n}$

$\longleftarrow xy \longrightarrow \longleftarrow z \longrightarrow$

xy contain only a's. (Because $|xy| \leq n$).
Let $|y|=l$, where $l > 0$ (Because $|y| > 0$).

Then, the break up of x. y and z can be as follows

$w = \overbrace{a^{n-l}}^{x}\ \overbrace{a^l}^{y}\ \overbrace{b^n}^{z}$

from the definition of PL , $w=xy^kz$, where k=0,1,2,------∞, should belong to L.

That is $a^{n-l}\,(a^l)^k\, b^n \in L$, for all k=0,1,2,------ ∞

Put k=0. we get $a^{n-l}\, b^n \notin L$.

Contradiction. Hence the Language is not regular.

Example 2.

To prove that L={w|w is a palindrome on {a,b}*} is not regular. i.e., L={aabaa, aba, abbbba,…}

Proof:

Let L be regular. Let n is the constant (PL Definition). Consider a word w in L. Let $w = a^nba^n$ such that $|w|=2n+1$. Since $2n+1 > n$ and L is regular it must satisfy PL.

Proof: Let L be regular. Let $w = 1^p$ where p is prime and $|p| = n+2$

Let y = m.

by PL $xy^kz \in L$

$|xy^kz| = |xz| + |y^k|$ Let k = p-m

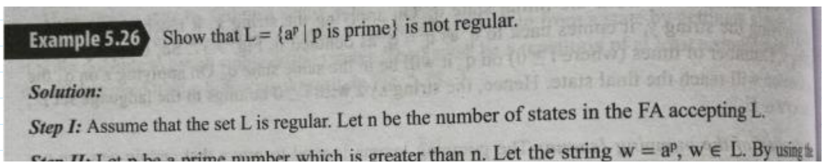
= (p-m) + m (p-m)

= (p-m) (1+m) ----- this can not be prime if $p\text{-}m \geq 2$ or $1+m \geq 2$

1. $(1+m) \geq 2$ because $m \geq 1$
2. Limiting case p=n+2
   $(p\text{-}m) \geq 2$ since $m \leq n$

Example 4.

Prove-LipPis-prime-n...


Example 5.26 Show that $L= \{a^p \mid p \text{ is prime}\}$ is not regular.

Solution:

Step I: Assume that the set L is regular. Let n be the number of states in the FA accepting L.

which is greater than n. Let the string $w = a^p$, $w \in L$. By using the if $p-m \geq 2$ or $1+m \geq 2$

1. $(1+m) \geq 2$ because $m \geq 1$
2. Limiting case $p=n+2$
   $(p-m) \geq 2$ since $m \leq n$

Example 4.

To prove that $L=\{ 0^{i^2} \mid i$ is integer and $i>0\}$ is not regular. i.e., $L=\{0^2, 0^4, 0^9, 0^{16}, 0^{25}, ----\}$

Proof: Let L be regular. Let $w = 0^{n^2}$ where $|w| = n^2 \geq n$

by PL $xy^kz \in L$, for all $k = 0,1,---$

Select $k = 2$

$| xy^2z | = | xyz | + | y |$

$= n^2 +$ Min 1 and Max n

44

Solution:

Step I: Assume that the set L is regular. Let n be the number of states in the FA accepting L.

Step II: Let p be a prime number which is greater than n. Let the string $w = a^p$, $w \in L$. By using the pumping lemma, we can write $w = xyz$ with $| xy | \leq n$ and $| y | > 0$. As the string w consists of only 'a', x, y, and z are also a string of 'a's. Let us assume that $y = a^m$ for some m with $1 \leq m \leq n$.

Step III: Let us take $i = p + 1$. $| xy^iz |$ will be $| xyz | + | y^{i-1} |$

$$
\begin{aligned}
| xy^iz | &= | xyz | + | y^{i-1} | \\
&= p + (i-1) \, | y | && [xyz = a^p] \\
&= p + (i-1)m && [y = a^m] \\
&= p + pm && [i = p + 1] \\
&= p(1 + m).
\end{aligned}
$$

$p(1 + m)$ is not a prime number as it has factors p and $(1 + m)$ including 1 and $p(1 + m)$. So, $xy^iz \notin$ L. This is a contradiction.

Therefore, $L = \{a^p \mid p \text{ is prime}\}$ is not regular.

Therefore $n^2 < | xy^2z | \leq n^2 + n$

$n^2 < | xy^2z | < n^2 + n + 1+n$ adding $1 + n$ (Note that less than or equal to is replaced by less than sign)

$n^2 < | xy^2z | < (n + 1)^2$

Say $n = 5$ this implies that string can have length > 25 and < 36 which is not of the form 0^{i^2}.

Exercises for students: -

a) Show that following languages are not regular
   (i) $L=\{a^nb^m \mid n, m \geq 0 \text{ and } n<m \}$
   (ii) $L=\{a^nb^m \mid n, m \geq 0 \text{ and } n>m \}$
   (iii) $L=\{a^nb^mc^md^n \mid n, m \geq 1 \}$
   (iv) $L=\{a^n \mid n \text{ is a perfect square} \}$
   (v) $L=\{a^n \mid n \text{ is a perfect cube} \}$
b) Apply pumping lemma to following languages and understand why we cannot complete proof
   (i) $L=\{a^naba \mid n \geq 0 \}$
   (ii) $L=\{a^nb^m \mid n, m \geq 0 \}$

45

Closure Properties of Regular Languages

22 August 2017 02:38 PM

Regular Languages Properties : Decision Properties and Closure Properties

Decision Properties of Regular Languages

1. Is the language described empty?
2. Is a particular string w in the described language?
3. Do two descriptions of a language actually describe the same language?

   This question is often called "equivalence" of languages.

Closure Properties of Regular Languages

1. The union of two regular languages is regular.
2. The intersection of two regular languages is regular.
3. The complement of a regular language is regular.
4. The difference of two regular languages is regular.
5. The reversal of a regular language is regular.
6. The closure (star) of a regular language is regular.
7. The concatenation of regular languages is regular.
8. A homomorphism (substitution of strings for symbols) of a regular language is regular.
9. The inverse homomorphism of a regular language is regular

UNIT - III

30 June 2017 07:14 PM

Regular Grammar

28 August 2017 11:01 AM

Grammar: A grammar G is a quadruple G = < V, T, P, S > where
V is a finite set of variables
T is a finite set of terminals
P is a finite set of productions
S is a special variable called start variable.

Phrase-Structure Grammar: A Pharse-Structure grammar is a grammar G = < V, T, P, S > where P consists of productions of the form $x \rightarrow y$ where $x \in (V \cup T)^+$ and $y \in (V \cup T)^*$

Chomsky Hierarchy of Phrase-Structure Grammars:
Type-0 grammar
Type-1 grammar
Type-2 grammar
Type-3 grammar

Type-0 grammar (Unrestricted grammar) : $u \rightarrow v$ where $u,v \in (V \cup T)^*$
Type-1 grammar (Context-sensitive grammar) : $x \rightarrow y$ where $x,y \in (V \cup T)^+$ and $|x| \leq |y|$
Type-2 grammar (Context Free Grammar): $A \rightarrow x$ where $A \in V$ and $x \in (V \cup T)^*$
Type-3 grammar (Regular Grammar) : $A \rightarrow xB$, $A \rightarrow x$ or $A \rightarrow Bx$, $A \rightarrow x$
where $A,B \in V$ and $x \in (V \cup T)^*$

Regular Grammar: Regular grammar is either right linear grammar or left linear grammar.
Every regular grammar is linear but every linear grammar is not regular.

2.1.3 Right-Linear Grammar

In general productions have the form:

$$(V \cup T)^+ \rightarrow (V \cup T)^*.$$

In right-linear grammar, all productions have one of the two forms:

$$V \rightarrow T^*V$$

or

$$V \rightarrow T^*$$

i.e., the left hand side should have a single variable and the right hand side consists of any number of terminals (members of T) optionally followed by a single variable.

2.1.4 Right-Linear Grammars and NFAs

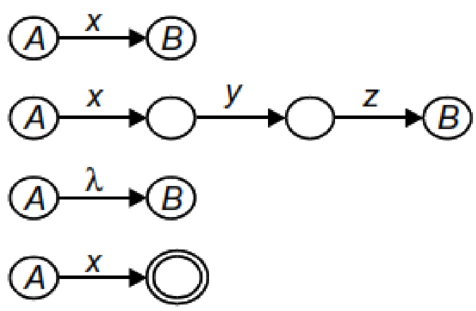
There is a simple connection between right-linear grammars and NFAs, as shown in the following illustration.

$A \to xB$

$A \to xyzB$

$A \to B$

$A \to x$



As an example of the correspondence between an NFA and a right linear grammar, the following automaton and grammar both recognize the set of set of strings consisting of an even number of 0's and an even number of 1's.

$S \to \lambda$

$S \to 0B$

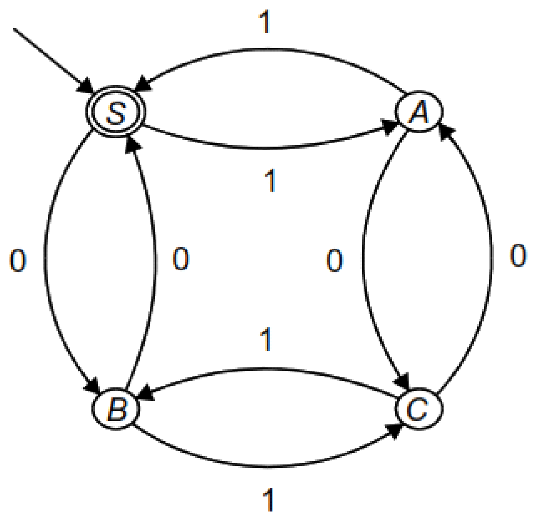
$S \to 0A$

$A \to 0C$

$A \to 1S$

$B \to 0S$

$B \to 1C$



2.1.5 Left-Linear Grammar

In a left-linear grammar, all productions have one of the two forms:

$$V \to VT^*$$

or

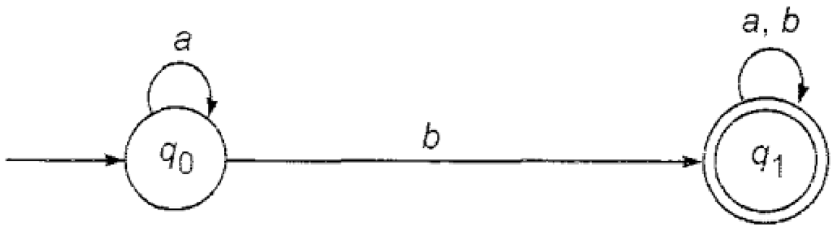
$$V \to T^*$$

i.e., the left hand side must consist of a single varibale, and the right-hand side consists of an optional single variable followed by one number of terminals.

2.1.6 Conversion of Left-linear Grammar into Right-Linear Grammar

Step	Method
(a) Construct a right-linear grammar for the different languages L^R.	Replace each production $A \to x$ of L with a production $A \to x^R$ and replace each production $A \to Bx$ with a production $A \to x^R B$
(b) Construct an NFA for L^R from the right-linear grammar. This NFA should have just one final state.	Refer to section 2.1.4 for deriving an NFA from a right-linear grammar.
(c) Reverse the NFA for L^R to obtain an NFA for L.	(i) Construct an NFA to recognize the language L. (ii) Ensure the NFA has only a single final state (iii) Reverse the direction of arcs (iv) Make the initial state final and final state initial
(d) Construct a right-linear grammar for L from the NFA for L.	This is the technique described in the previous section.

Construct a regular grammar G generating the regular set represented by $\mathbf{P} = \mathbf{a^*b(a + b)^*}$.



Let $G = (\{A0, A_1\}, \{a, b\}, P, A_0)$, where P is given by

$$A_0 \to aA_0, \quad A_0 \to bA_1, \quad A_0 \to b$$

$$A_1 \to aA_1, \quad A_1 \to bA_1, \quad A_1 \to a, \quad A_1 \to b$$

G is the required regular grammar.

Example 2.1.2: Construct right-and left-linear grammars for the language $L = \{a^n b^m : n \geq 2,\ m \geq 3\}$.

Solution

Right-Linear Grammar:

$$S \to aS$$
$$S \to aaA$$
$$A \to bA$$
$$A \to bbb$$

Left-Linear Grammar:

$$S \to Abbb$$
$$S \to Sb$$
$$A \to Aa$$
$$A \to aa$$

Context Free Grammar

18 September 2017 05:55 AM

<u>Definition:</u> A grammar G = (V,T,P,S) is said to be CFG iff all productions are in the form $A \rightarrow x$ where $A \in V$ and $x \in (V \cup T)^*$

Write CFGs for the following languages:

(1) $L = \{a^n b^n : n \geq 0\}$ $\quad S \to aSb \mid \lambda$

(2) $L = \{a^n b^n : n \geq 1\}$ $\quad S \to aSb \mid ab$

(3) $L = \{a^{n+1} b^n : n \geq 0\}$ $\quad S \to aSb \mid a$

(4) $L = \{a^n b^{n+1} : n \geq 0\}$ $\quad S \to aSb \mid b$

(5) $L = \{a^{n+2} b^n : n \geq 0\}$ $\quad S \to aSb \mid aa$

(6) $L = \{a^n b^{n+2} : n \geq 0\}$ $\quad S \to aSb \mid bb$

(7) $L = \{a^{2n} b^n : n \geq 0\}$ $\quad S \to aaSb \mid \lambda$

(8) $L = \{a^n b^{2n} : n \geq 0\}$ $\quad S \to aSbb \mid \lambda$

(9) palindromes of a's & b's $\quad S \to \lambda \mid a \mid b \mid aSa \mid bSb$

(10) $L = \{ww^R : w \in \{a,b\}^*\}$ $\quad S \to \lambda \mid aSa \mid bSb$

(11) $L = \{wcw^R : w \in \{a,b\}^*\}$ $\quad S \to c \mid aSa \mid bSb$

(12) $L = \{\underbrace{0^m 1^m}_{A} \underbrace{2^n}_{B} : m \geq 1 \ \& \ n \geq 0\}$

$S \to AB$

$A \to 01 \mid 0A1$

$B \to \lambda \mid 2B$

(13) $L = \{w \mid n_a(w) = n_b(w)\}$

$S \to \lambda \mid aSb \mid bSa$

① Find Language: $S \to 0A \mid \epsilon$
$A \to 1S$

$S \to \epsilon$ $\quad$ $S \Rightarrow 0A$

$L = \{\epsilon\}$ $\quad$ $\Rightarrow 01S \Rightarrow 010A \Rightarrow$

$\Rightarrow 01$ $\quad$ $\Rightarrow 0101S$ $\quad$ $\Rightarrow 01010A$

$\Rightarrow 0101$ $\quad$ $\Rightarrow 010101S \Rightarrow 010101$

$L = \{\epsilon, 01\}$ $\quad$ $L = \{\epsilon, 01, 0101\}$ $\quad$ $L = \{\epsilon, 01, 0101, 010101, \ldots\}$

$\therefore L = \{(01)^*\}$ or $L = \{(01)^n : n \geq 0\}$

② Write CFG to generate balanced parantheses.

$S \to (S) \mid \epsilon$

③ $L = \{a^{n+2} b^m : n \geq 0 \;\&\; m > n\}$

$n=0$ $\quad$ $aabb^*$

$n=1$ $\quad$ $aaabbb^*$

$\therefore L = \{aabb^*, aaabbb^*, \ldots\}$

$S \to aAB$

$A \to ab \mid aAb$

$B \to \lambda \mid b \mid bB$

③ $L = \{a^n b^m : n \geq 0, m > n\}$

$n=0$ $\quad$ $n=1$ $\quad$ $n=2$

$m>0$ $\quad$ $m>1$ $\quad$ $m>2$

$L = \{\epsilon bb^*, \; abbb^*, \; aabbbb^* \; \text{---}\}$

$S \to aSb \mid b$ $\quad \Rightarrow \quad$ $S \to AB$

$A \to aSb \mid b$

$B \to \lambda \mid bB$

④ $L = \{a^n b^{n-3} : n \geq 3\}$

$n=3$ $\quad$ $n=4$ $\quad$ $n=5$ $\quad$ $n=6$

$L = \{aaa, \; aaaab, \; aaaaabb, \; aaaaaabbb, \; \text{---}\}$

$S \to aaaA$

$A \to \cancel{ab}\; aAb \mid \epsilon$

LMD, RMD & Derivation Trees

13 September 2018 08:29 AM

2.2 DERIVATION TREES

A 'derivation tree' is an ordered tree which the the nodes are labeled with the left sides of productions and in which the children of a node represent its corresponding right sides.

2.2.1 Definition of a Derivation Tree

Let $G = (V, T, S, P)$ be a CFG. An ordered tree is a derivation tree for G iff it has the following properties:

(i) The root of the derivation tree is S.

(ii) Each and every leaf in the tree has a label from $T \cup \{\lambda\}$.

(iii) Each and every interior vertex (a vertex which is no a leaf) has a label from V.

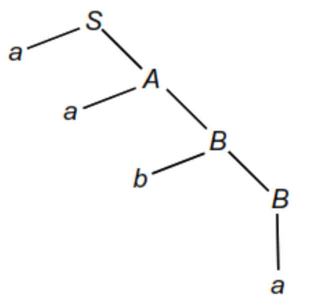
(iv) If a vertex has label $A \in V$, and its children are labeled (from left to right) $a_1, a_2, \ldots\ldots a_n$, then P must contain a production of the form

$$A \to a_1,\ a_2,\ \ldots\ \ldots\ a_n$$

(v) A leaf labeled λ has no siblings, that is, a vertex with a child labeled λ can have no other children.

2.2.2 Sentential Form

For a given CFG with productions $S \to aA, A \to aB, B \to bB, B \to a$. The derivation tree is as shown below.



$$S \Rightarrow aA \Rightarrow aaB \Rightarrow aabB \Rightarrow aaba$$

The resultant of the derivation tree is the word $w = aaba$.
This is said to be in "Sentential Form".

2.2.3 Partial Derivation Tree

In the definition of derivation tree given, if every leaf has a label from $V \cup T \cup \{\lambda\}$ it is said to be "partial derivation tree".

2.2.4 Right Most/Left Most/Mixed Derivation

Consider the grammar G with production

$$\begin{Bmatrix} 1.\ S \to aSS \\ 2.\ S \to b \end{Bmatrix}$$

Now, we have

$$\begin{aligned} S &\overset{1}{\Rightarrow} aSS \\ &\overset{1}{\Rightarrow} aaSSS \\ &\overset{2}{\Rightarrow} aabSS \qquad\qquad \text{(Left Most Derivation)} \\ &\overset{1}{\Rightarrow} aabaSSS \\ &\overset{2}{\Rightarrow} aababSS \\ &\overset{2}{\Rightarrow} aababbS \\ &\overset{2}{\Rightarrow} aababbb \end{aligned}$$

The sequence followed is "left most derivation", following "1121222", giving "*aababbb*".

$$
\begin{aligned}
S &\overset{1}{\Rightarrow} aSS \\
&\overset{2}{\Rightarrow} aSb \\
&\overset{1}{\Rightarrow} aaSSb \qquad \text{(Mixed Derivation)} \\
&\overset{2}{\Rightarrow} aabSb \\
&\overset{1}{\Rightarrow} aabaSSb \\
&\overset{2}{\Rightarrow} aabaSbb \\
&\overset{2}{\Rightarrow} aababbb
\end{aligned}
$$

The sequence 1212122 represents a "Mixed Derivation", giving "*aababbb*".

$$
\begin{aligned}
S &\overset{1}{\Rightarrow} aSS \\
&\overset{2}{\Rightarrow} aSb \\
&\overset{1}{\Rightarrow} aaSSb \\
&\overset{1}{\Rightarrow} aaSaSSb \qquad \text{(Right Most Derivation)} \\
&\overset{2}{\Rightarrow} aaSaSbb \\
&\overset{2}{\Rightarrow} aaSabbb \\
&\overset{2}{\Rightarrow} aababbb
\end{aligned}
$$

The sequence 1211222 represents a "Right Most Derivation", giving "*aababbb*".

Example 2.2.1: A grammar G which is context-free has the productions

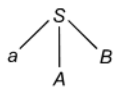
$$
\begin{aligned}
&S \to aAB \\
&A \to Bba \\
&B \to bB \\
&B \to c.
\end{aligned}
$$

(The word $w = acbabc$ is derived as follows)

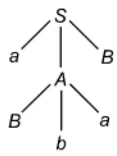
$$S \Rightarrow aAB \to a(Bba)B \Rightarrow acbaB \Rightarrow acba(bB) \Rightarrow acbabc.$$

Obtain the derivation tree.

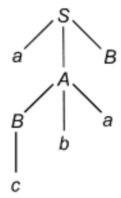
Solution



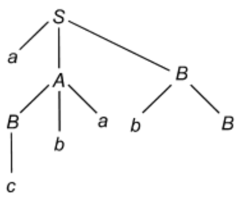
(a) $S \to aAB$



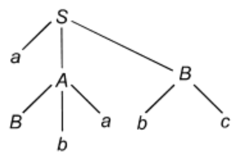
(b) $A \to Bba$



(c) $B \to c$



(d) $B \to bB$



(e) $B \to c$

Example 2.2.2: A CFG given by productions is

$$S \to a,$$
$$S \to aAS,$$
$$\text{and} \quad A \to bS$$

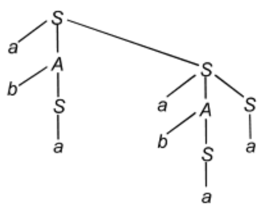
Obtain the derivation tree of the word $w = abaabaa$.

Solution

$w = abaabaa$ is derived from S as

$$\begin{aligned} S \Rightarrow aAS \Rightarrow a(bS)S \Rightarrow abaS &\Rightarrow aba(aAS) \\ &\Rightarrow abaa(bs)S \\ &\Rightarrow abaabaS \\ &\Rightarrow abaabaa \end{aligned}$$

The derivation tree is sketched below.



Example 2.2.3: Given a CFG given by $G = (N, T, P, S)$

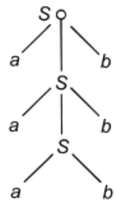
with $N = \{S\}$, $T = \{a, b\}$, $P = \begin{Bmatrix} (1)\, S \to aSb \\ (2)\, S \to ab \end{Bmatrix}$.

Obtain the derivation tree and the language generated $L(G)$.

Solution

$$\begin{aligned} S &\Rightarrow ab && \text{i.e., } ab \in L(G) \\ S &\Rightarrow aSb \\ &\Rightarrow aabb && \text{i.e., } a^2b^2 \in L(G) \\ S &\Rightarrow aSb \\ &\Rightarrow aaSbb \\ &\Rightarrow aaabbb && \text{i.e., } a^3b^3 \in L(G), \\ &\Rightarrow a^3b^3 && \text{and so on} \end{aligned}$$

Derivation tree is as follows.



Language generated $L(G) = \{a^n b^n \mid n \geq 1\}$.

Example 2.2.4: Given a CFG $G = (N, T, P, S)$

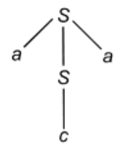
with $N = \{S\}$, $T = \{a, b, c\}$ and $P = \begin{Bmatrix} (1)\; S \to aSa \\ (2)\; S \to bSb \\ (3)\; S \to c \end{Bmatrix}$.

Obtain the derivation tree and language generated $L(G)$.

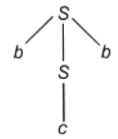
Solution

(i) $S \Rightarrow c, \quad c \in L(G)$

(ii) $S \Rightarrow aSa \Rightarrow aca \in L(G)$



(iii) $S \Rightarrow bSb \Rightarrow bcb \in L(G)$

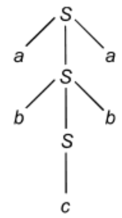


(iv) $S \Rightarrow aSa$

$\Rightarrow abSba$

$\Rightarrow abcba \in L(G)$

and so on.



Hence the language generated $L(G)$ is given by

$$L(G) = \{wcw^R \mid w \in \{a, b\}^*\}$$

where w^R = reversal of w

i.e., if $w = a_1 a_2 \ldots\ldots a_{n-1} a_n$

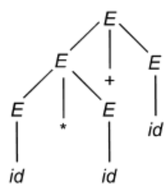
then $w^R = a_n a_{n-1} \ldots\ldots a_2 a_1$.

Example 2.2.5: Given $G = (N, T, P, S)$ with $N = \{E\}$, $S = E$, $T = \{id, +, *, c\}$

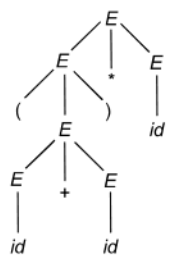
and P: 1. $E \to E + E$
2. $E \to E * E$.
3. $E \to (E)$
4. $E \to id$

Obtain the derivation tree.

Solution



$\Rightarrow$ $\boxed{id * id + id}$



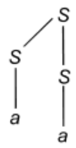
$\Rightarrow$ $\boxed{(id + id) * id}$

Example 2.2.6: Obtain the language generated $L(G)$ for a CFG given $G(N, T, P, S)$ with $N = \{S\}$, $T\{a\}$, $P: \begin{Bmatrix} 1.\ S \to SS \\ 2.\ S \to a \end{Bmatrix}$

Solution

$S \Rightarrow a$

$$\begin{aligned} S &\Rightarrow SS \\ &\Rightarrow aS \\ &\Rightarrow aa \end{aligned}$$



$$S \Rightarrow SS \Rightarrow aS \Rightarrow aSS \Rightarrow aaS \Rightarrow aaa \text{ and so on....}$$

Therefore the language generated is

$$L(G) = \{a^n \mid n \geq 1\}$$

Example 2.2.7: Obtain the language generated by each of the following production rules.

(a) $A \to a$, $A \to aB$, $A \to \in$

(b) $S \to aS$, $S \to \in$

(c) $A \to a$, $A \to aB$, $A \to \in$

(d) $A \to aS$, $S \to bS$, $S \to \in$

(e) $S \to aS$, $S \to bS$, $S \to a$

(f) $S \to ab$, $S \to bs$, $S \to a$, $S \to b$

Solution

(a) The language generated is a "type-3 language" or "regular set".

(b) $S \Rightarrow \in$
$S \Rightarrow aS \Rightarrow a$
$S \Rightarrow as \Rightarrow aaS \Rightarrow aa$
and so on.
Hence the language generated is

$$L(G) = \{a^n \mid n \geq 0\}$$

(c) $A \Rightarrow \in$
$A \Rightarrow a$ $\quad L(G) = \{ww^R \mid w \in \{a,b\}^+\}$
$A \Rightarrow aB$

(d) $S \rightarrow aS$
$S \rightarrow bS$ $\quad L(G) = \{a,b\}^*$
$S \rightarrow \in$ $\quad$ Language generated of any string of a,b

(e) $S \rightarrow aS$
$S \rightarrow bS$ $\quad L(G) = \{a,b\}^* a$
$S \rightarrow a$

(f) $S \rightarrow ab$
$S \rightarrow bS$ $\quad L(G) = \{a,b\}^+$.
$S \rightarrow a$
$S \rightarrow b$

Example 2.2.8: Given a CFG $G = (N, T, P, S)$

with $N = \{S, A\}$, $T = \{a, b\}$ and $P = \begin{Bmatrix} 1. & S \rightarrow aS \\ 2. & S \rightarrow aA \\ 3. & A \rightarrow bA \\ 4. & A \rightarrow b \end{Bmatrix}$

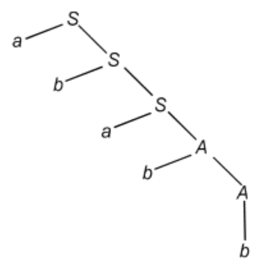
Obtain the derivation tree and $L(G)$.

Solution

$$S \Rightarrow aA \Rightarrow ab$$
$$S \Rightarrow aS \Rightarrow aaA \Rightarrow aab$$
$$S \Rightarrow aS \Rightarrow aaS \Rightarrow aaaA \Rightarrow aaabA \Rightarrow aaabb$$

and so on ...

The derivation tree has been shown here in fig.



The language generated is

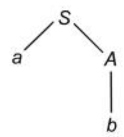
$$L(G) = \{a^n b^m \;\; n \geq 1, \; m \geq 1\}$$

Example 2.2.9: Given a CFG with

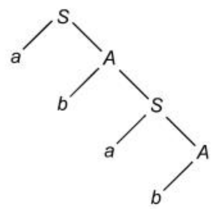
$$P = \left\{\begin{array}{l} 1.\ S \rightarrow aA \\ 2.\ A \rightarrow bS \\ 3.\ A \rightarrow b \end{array}\right\}.$$ Obtain the derivation tree and $L(G)$.

Solution

$$S \Rightarrow aA \Rightarrow ab$$



$$S \Rightarrow aA \Rightarrow abS \Rightarrow abaA \Rightarrow abab\ldots\ \ldots$$



The derivation trees suggest ab, $abab$, ...
Therefore the language generated

$$L(G) = \{(ab)^n \mid n \geq 1\}$$

Example 2.2.10: Obtain the production rules for CFG given the language generated as

(a) $L(G) = \{w \mid w \in \{a, b\}^*,\ \times_a(w) = \times_b(w)\}$

(b) $L(G) = \{w \mid w \in \{a, b\}^*,\ \times_a(w) = 2\times_b(w)\}$

(c) $L(G) = \{w \mid w \in \{a, b\}^*,\ \times_a(w) = 3\times_b(w)\}$

Solution

(a) $S \rightarrow SaSbS$
$S \rightarrow \in$
$S \rightarrow SbSaS$

(b) $S \rightarrow SaSaSbS$
$S \rightarrow SaSbSaS$
$S \rightarrow SbSaSaS$
$S \rightarrow \in$

(c) $S \to SaSaSaSbS$
$S \to SaSaSbSaS$
$S \to SaSbSaSaS$
$S \to SbSaSaSaS$
$S \to \in$

Example 2.2.11: Given a grammar G with production rules

$$S \to aB$$
$$S \to bA$$
$$A \to aS$$
$$A \to bAA$$
$$A \to a$$
$$B \to bS$$
$$B \to aBB$$
$$B \to b$$

Obtain the (i) leftmost derivation, and (ii) rightmost derivation for the string "*aaabbabbba*".

Solution

(i) *Leftmost derivation:*

$$S \Rightarrow aB \Rightarrow aaBB \Rightarrow aaaBBB \Rightarrow aaabBB \Rightarrow aaabbB$$
$$\Rightarrow aaabbabB \Rightarrow aaabbabbB \Rightarrow aaabbabbbS \Rightarrow aaabbabbba$$
$$\Rightarrow aaabbabb$$

(ii) *Rightmost derivation:*

$$S \Rightarrow aB \Rightarrow aaBB \Rightarrow aaBbS \Rightarrow aaBbbA \Rightarrow aaaBBbba$$
$$\Rightarrow aaabBbba \Rightarrow aaabbSbba \Rightarrow aaabbaBbba \Rightarrow aaabbabbba$$

EXAMPLE 6.3

Let G be the grammar $S \to 0B \mid 1A$, $A \to 0 \mid 0S \mid 1AA$, $B \to 1 \mid 1S \mid 0BB$. For the string 00110101, find (a) the leftmost derivation, (b) the rightmost derivation, and (c) the derivation tree.

Solution

(a) $S \Rightarrow 0B \Rightarrow 00BB \Rightarrow 001B \Rightarrow 0011S$
$\Rightarrow 0^21^20B \Rightarrow 0^21^201S \Rightarrow 0^21^2010B \Rightarrow 0^21^20101$

(b) $S \Rightarrow 0B \Rightarrow 00BB \Rightarrow 00B1S \Rightarrow 00B10B$
$\Rightarrow 0^2B101S \Rightarrow 0^2B1010B \Rightarrow 0^2B10101 \Rightarrow 0^2110101.$

(c) The derivation tree is given in Fig. 6.9.

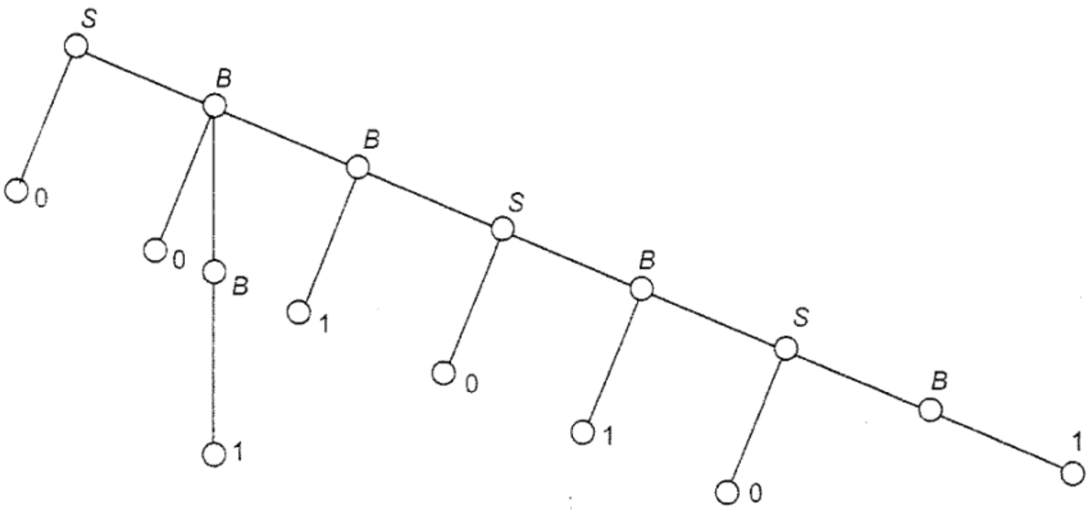


Fig. 6.9 The derivation tree with yield 00110101 for Example 6.3.

Ambiguous Grammar

03 October 2017 10:06 AM

Derivation of a String:
Left most derivation of a string:
Right most derivation of a string:
Derivation tree of a string:
Ambiguous Grammar:

2.3 PARSING AND AMBIGUITY

2.3.1 Parsing

A grammar can be used in two ways:

(a) Using the grammar to generate strings of the language.
(b) Using the grammar to recognize the strings.

"Parsing" a string is finding a derivation (or a derivation tree) for that string.

Parsing a string is like recognizing a string. The only realistic way to recognize a string of a context-free grammar is to parse it.

2.3.2 Exhaustive Search Parsing

The basic idea of the "Exhaustive Search Parsing" is to parse a string w, generate all strings in L and check if w is among them.

Problem arises when L is an infinite language. Therefore a systematic approach is needed to achieve this, as it is required to know that no strings are overlooked. And also it is necessary so as to stop after a finite number of steps.

The idea of exhaustive search parsing for a string is to generate all strings of length no greater than $|w|$, and see if w is among them.

The restrictions that are placed on the grammar will allow us to generate any string $w \in L$ in at most $2|w|-1$ derivation steps.

Exhaustive search parsing is inefficient. It requires time exponential in $|w|$.

There are ways to further restrict context free grammar so that strings may be parsed in linear or non-linear time (which methods are beyond the scope of this book).

There is no known linear or non-linear algorithm for parsing strings of a general context free grammar.

2.3.3 Topdown/Bottomup Parsing

Sequence of rules are applied in a leftmost derivation in Topdown parsing. (Refer to section 2.2.4.)

Sequence of rules are applied in a rightmost derivation in Bottomup parsing.

This is illustrated below.

Consider the grammar G with production

$$1.\ S \to aSS$$
$$2.\ S \to b.$$

The parse trees are as follows.

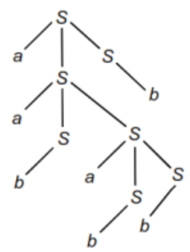


Fig. Topdown parsing.

$aababbb \to$ Left parse of the string with the sequence 1121222.
This is known as "Topdown Parsing."

"Right Parse" is the reversal of sequence of rules applied in a rightmost derivation.

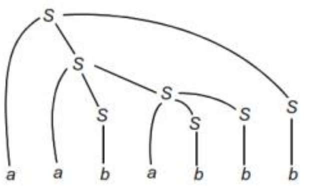


Fig. Bottom-up parsing.

aababbb $\rightarrow$ Right parse of the string with the sequence 2221121.
This is known as "Bottom-up Parsing."

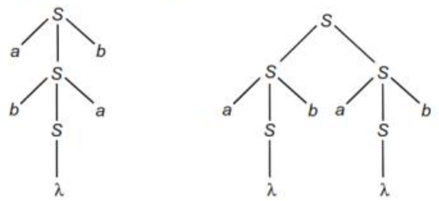
2.3.4 Ambiguity

The grammar given by

$$G = (\{S\}, \{a, b\}, S, S \rightarrow aSb \mid bSa \mid SS \mid \lambda)$$

generates strings having an equal number of a's and b's.

The string "*abab*" can be generated from this grammar in two distinct ways, as shown in the following derivation trees:



Similarly, "*abab*" has two distinct leftmost derivations:

$$S \Rightarrow aSb \Rightarrow abSab \Rightarrow abab$$
$$S \Rightarrow SS \Rightarrow aSbS \Rightarrow abS \Rightarrow abaSb \Rightarrow abab.$$

Also, "*abab*" has two distinct rightmost derivations:

$$S \Rightarrow aSb \Rightarrow abSab \Rightarrow abab$$
$$S \Rightarrow SS \Rightarrow SaSb \Rightarrow Sab \Rightarrow aSbab \Rightarrow abab$$

Each of the above derivation trees can be turned into a unique rightmost derivation, or into a unique leftmost derivation. Each leftmost or rightmost derivation can be turned into a unique derivation tree. These representations are largely interchangeable.

2.3.5 Ambiguous Grammars/Ambiguous Languages

Since derivation trees, leftmost derivations, and rightmost derivations are equivalent rotations, the following definitions are equivalent:

Definition: Let $G = (N, T, P, S)$ be a CFG.

A string $w \in L(G)$ is said to be "ambiguously derivable "if there are two or more different derivation trees for that string in G.

Definition: A CFG given by $G = (N, T, P, S)$ is said to be "ambiguous" if there exists at least one string in $L(G)$ which is ambiguously derivable. Otherwise it is unambiguous.

Ambiguity is a property of a grammar, and it is usually, but not always possible to find an equivalent unambiguous grammar.

An "inherantly ambiguous language" is a language for which no unambiguous grammar exists.

Example 2.3.1: Prove that the grammar

$$S \rightarrow aB \mid ab,$$
$$A \rightarrow aAB \mid a,$$
$$B \rightarrow ABb \mid b$$

is ambiguous.

Solution

It is easy to see that "*ab*" has two different derivations as shown below.

Given the grammar G with production

1. $S \rightarrow aB$
2. $S \rightarrow ab$
3. $A \rightarrow aAB$
4. $A \rightarrow a$
5. $B \rightarrow ABb$
6. $B \rightarrow b$

$$\left.\begin{array}{ll} \text{Using (2),} & S \Rightarrow ab \\ \text{Using (1),} & S \Rightarrow aB \Rightarrow ab \\ \text{and then (6).} & \end{array}\right\}$$

Sometimes we come across ambiguous sentences in the language we are using. Consider the following sentence in English: "In books selected information is given." The word 'selected' may refer to books or information. So the sentence may be parsed in two different ways. The same situation may arise in context-free languages. The same terminal string may be the yield of two derivation trees. So there may be two different leftmost derivations of w by Theorem 6.2. This leads to the definition of ambiguous sentences in a context-free language.

Definition 6.6 A terminal string $w \in L(G)$ is ambiguous if there exist two or more derivation trees for w (or there exist two or more leftmost derivations of w).

Consider, for example, $G = (\{S\}, \{a, b, +, *\}, P, S)$, where P consists of $S \rightarrow S + S \mid S * S \mid a \mid b$. We have two derivation trees for $a + a * b$ given in Fig. 6.10.

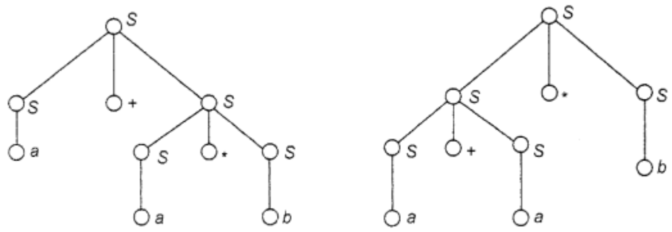


Fig. 6.10 Two derivation trees for $a + a * b$.

The leftmost derivations of $a + a * b$ induced by the two derivation trees are

$$S \Rightarrow S + S \Rightarrow a + S \Rightarrow a + S * S \Rightarrow a + a * S \Rightarrow a + a * b$$
$$S \Rightarrow S * S \Rightarrow S + S * S \Rightarrow a + S * S \Rightarrow a + a * S \Rightarrow a + a * b$$

Therefore, $a + a * b$ is ambiguous.

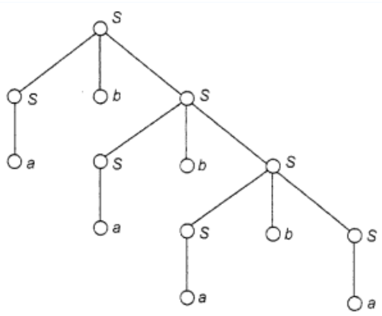
Definition 6.7 A context-free grammar G is ambiguous if there exists some $w \in L(G)$, which is ambiguous.

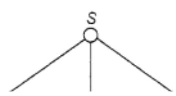
EXAMPLE 6.4

If G is the grammar $S \rightarrow SbS \mid a$, show that G is ambiguous.

Solution

To prove that G is ambiguous, we have to find a $w \in L(G)$, which is ambiguous. Consider $w = abababa \in L(G)$. Then we get two derivation trees for w (see Fig. 6.11). Thus, G is ambiguous.





Using (2), $S \Rightarrow ab$
Using (1), $S \Rightarrow aB \Rightarrow ab$
and then (6).

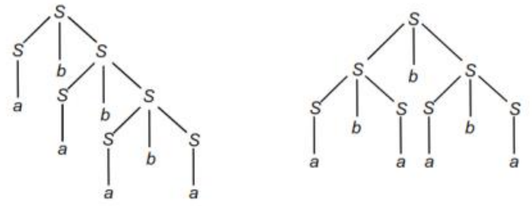
Example 2.3.2: Show that the grammar $S \rightarrow S \mid S,\ S \rightarrow a$ is ambiguous.

Solution

In order to show that G is ambiguous, we need to find a $w \in L(G)$, which is ambiguous.

Assume $\quad w = abababa.$

The two derivation trees for $w = abababa$ is shown below in Fig. (a) and (b).



(a) (b)

Therefore, the grammar G is ambiguous.

Example 2.3.3: Show that the grammar G with production

$$S \rightarrow a \mid aAb \mid abSb$$
$$A \rightarrow aAAb \mid bS$$

is ambiguous.

Solution

$$S \Rightarrow abSb \qquad (\because S \rightarrow abSb)$$
$$\Rightarrow abab \qquad (\because S \rightarrow a)$$

Similarly,

$$S \Rightarrow aAb \qquad (\because S \rightarrow aAb)$$
$$\Rightarrow abSb \qquad (\because A \rightarrow bS)$$
$$\Rightarrow abab$$

Since '*abab*' has two different derivations, the grammar G is ambiguous.

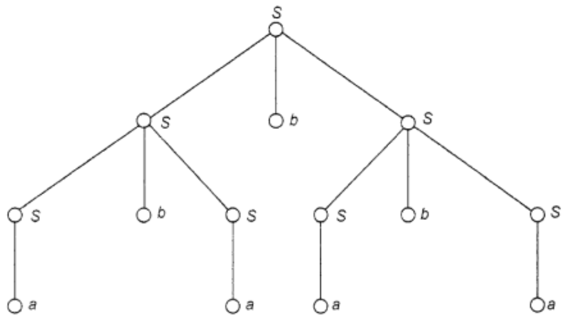


Fig. 6.11 Two derivation trees of *abababa* for Example 6.4.

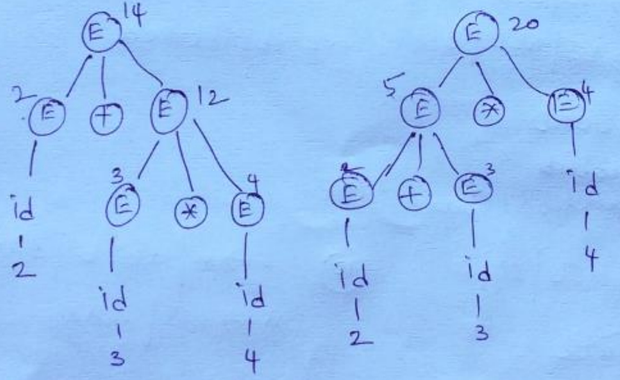
Ambiguous Grammar

① $E \rightarrow E+E \mid E*E \mid id$

$w = id + id * id$

LMD1: $E \Rightarrow E+E$
$\Rightarrow id+E$
$\Rightarrow id+E*E$
$\Rightarrow id+id*E$
$\Rightarrow id+id*id$

LMD2: $E \Rightarrow E*E$
$\Rightarrow E+E*E$
$\Rightarrow id+E*E$
$\Rightarrow id+id*E$
$\Rightarrow id+id*id$



② $S \rightarrow XY \mid aaY,\ X \rightarrow Xa \mid a,\ Y \rightarrow b$

$w = aab$

③ $S \rightarrow aSbS \mid bSaS \mid \lambda$

$w = abab$

④ $S \rightarrow bAc \mid baC,\ A \rightarrow a,\ C \rightarrow c$

$w = bac$

(5) $S \to XY \mid Z$

$X \to aXb \mid ab$

$Y \to aYd \mid cd$

$Z \to aZd \mid aWd$

$W \to bWc \mid bc$

$w = aabbccdd$

(6) $S \to aS \mid X$

$X \to aX \mid a$

$w = aaaa$

(7) $S \to aB \mid bA$

$A \to aS \mid bAA \mid a$

$B \to bS \mid aBB \mid b$

$w = aabbab$

~~w = aabbab~~

$w = aaabbabbba$

(8) $S \to iCtS \mid iCtSeS \mid a$

$C \to b$

$w = ibtibtaea$

(9) $S \to aSb \mid SS \mid \lambda$

$w = aabb$

(10) $S \to a \mid abSb \mid aAb$

$A \to bS \mid aAAb$

$w = abab$

(11) $S \to aB \mid ab$

$A \to aAB \mid a$

$B \to ABb \mid b$

$w = ab$

CFG minimization

03 October 2017 10:08 AM

CFG minimization algorithm:

Repeat the following steps 1, 2 and 3 until there are no useless/null/unit productions in the given grammar.

1. Remove useless productions.

a) Eliminate variables that do not derive any terminal string.

b) Eliminate variables that are not reachable from the start variable.

2. Remove null productions.
3. Remove unit productions.

Minimize the following CFGs.

1. S → aS | A | C
   A → a
   B → aa
   C → acb

2. S → a | aA | B | C
   A → aB | λ
   B → Aa
   C → cCD
   D → ddd

3. S → aAa
   A → Sb | bcc | DaA
   C → abb | DD
   D → aDA
   E → ac

4 S → XY
X → 0
Y → Z | 1
Z → W
W → C
C → 0

5 S → A
A → B
B → C
C → D
D → a

6 S → aA | bB
A → aA | a
B → bB
D → ab | Ea
E → aC | d

7 S → aA | a | Bb | cC
A → aB
B → a | Aa
C → cCD
D → ddd

8 S → ABCa | bD
A → BC | b
B → b | λ
C → c | λ
D → d

9 S → BAAB
A → 0A2 | 2A0 | λ
B → AB | 1B | λ

10 S → AB
A → a
B → C | b
C → D
D → E | Bc
E → d | Ab

11 S → A0 | B
B → A | 11
A → 0 | 12 | B

12 S → Aa | B | Ca
B → aB |b
C → Dd | D
D → E | d
E → ab

13 S → aSa | bSb | A
A → aBb | bBa
B → aB | bB | λ

Chomsky Normal Form (CNF)

03 October 2017 04:43 PM

A CFG G = (V, T, P, S) is said to be in CNF notation iff all the productions are in the form

A → BC
A → a

Where A, B, C ϵ V and a ϵ T

By definition, the right hand side of a production in CFG contains any no.of variables and terminals.
CNF reduces the length of right hand side of production to one or two symbols only.
If there are two symbols then both must be variables.
If there is only one symbol then it must be a terminal.

Note: Any CFG can be converted into CNF notation but the given grammar must not contain null productions and unit productions . If so, eliminate them.

1 S → 0A | 1B
A → 0AA | 1S | 1
B → 1BB | 0S |0

2 S → Aba
A → aab
B → Ac

S → bA | aB
A → bAA | aS | a
B → aBB | bS | b

S → aSa | SSa | a

Griebach Normal Form (GNF)

05 October 2017 09:39 AM

MA513: Formal Languages and Automata Theory
Topic: Properties of Context-free Languages
Lecture Number 29 Date: October 18, 2011

1 Greibach Normal Form (GNF)

A CFG $G = (V, T, R, S)$ is said to be in GNF if every production is of the form $A \to a\alpha$, where $a \in T$ and $\alpha \in V^*$, i.e., α is a string of zero or more variables.

Definition: A production $\mathcal{U} \in R$ is said to be in the form **left recursion**, if $\mathcal{U} : A \to A\alpha$ for some $A \in V$.

Left recursion in R can be eliminated by the following scheme:
• If $A \to A\alpha_1|A\alpha_2|\ldots|A\alpha_r|\beta_1|\beta_2|\ldots|\beta_s$, then replace the above rules by
(i) $Z \to \alpha_i|\alpha_i Z, 1 \le i \le r$ and (ii) $A \to \beta_i|\beta_i Z, 1 \le i \le s$

• If $G = (V, T, R, S)$ is a CFG, then we can construct another CFG $G_1 = (V_1, T, R_1, S)$ in **Greibach Normal Form (GNF)** such that $L(G_1) = L(G) - \{\epsilon\}$.

The stepwise algorithm is as follows:

1. Eliminate null productions, unit productions and useless symbols from the grammar G and then construct a $G' = (V', T, R', S)$ in **Chomsky Normal Form (CNF)** generating the language $L(G') = L(G) - \{\epsilon\}$.
2. Rename the variables like $A_1, A_2, \ldots A_n$ starting with $S = A_1$.
3. Modify the rules in R' so that if $A_i \to A_j\gamma \in R'$ then $j > i$
4. Starting with A_1 and proceeding to A_n this is done as follows:
   (a) Assume that productions have been modified so that for $1 \le i \le k$, $A_i \to A_j\gamma \in R'$ only if $j > i$
   (b) If $A_k \to A_j\gamma$ is a production with $j < k$, generate a new set of productions substituting for the A_j the body of each A_j production.
   (c) Repeating (b) at most $k - 1$ times we obtain rules of the form $A_k \to A_p\gamma, p \ge k$
   (d) Replace rules $A_k \to A_k\gamma$ by removing left-recursion as stated above.
5. Modify the $A_i \to A_j\gamma$ for $i = n-1, n-2, ., 1$ in desired form at the same time change the Z production rules.

1

GNF was given by Sheila A. Griebach in 1965. This normal form not only put restrictions on the length of the body of a production (like CNF) but also put restrictions on the positions in which terminals and non-terminals appear in the body of production.

Using CNF, a string of length w can be derived using 2w-1 steps.
Using GNF, a string of length w can be derived using w steps because each step produces a terminal symbol of string.
Moreover, GNF is used to construct PDA.

Any CFG can be converted into CNF as well as GNF grammar.

Brief Procedure:
1. Minimize CFG. (useless, null and unit productions deleted)
2. Convert into CNF.
3. Rename variables as A1, A2, .. Starting with S = A1.
4. Apply substitution rule for all productions of the form:]
   Ai ---> Aj where i > j until i = j
5. For all productions i = j , eliminate left recursion.
6. Again apply substitution rule, to convert productions in GNF.

Example: Convert the following grammar G into Greibach Normal Form (GNF).

$$\boxed{\begin{array}{l} S \to XA|BB \\ B \to b|SB \\ X \to b \\ A \to a \end{array}}$$

To write the above grammar G into GNF, we shall follow the following steps:

1. Rewrite G in Chomsky Normal Form (CNF)

   It is already in CNF.

2. Re-label the variables

   S with A_1

   X with A_2

   A with A_3

   B with A_4

   After re-labeling the grammar looks like:

   $A_1 \to A_2A_3|A_4A_4$

   $A_4 \to b|A_1A_4$

   $A_2 \to b$

   $A_3 \to a$

3. Identify all productions which do not conform to any of the types listed below:

   $A_i \to A_jx_k$ such that $j > i$

   $Z_i \to A_jx_k$ such that $j \leq n$

   $A_i \to ax_k$ such that $x_k \in V^*$ and $a \in T$

4. $A_4 \to A_1A_4$ .... identified

5. $A_4 \to A_1A_4|b$.

   To eliminate A_1 we will use the substitution rule $A_1 \to A_2A_3|A_4A_4$.

   Therefore, we have $A_4 \to A_2A_3A_4|A_4A_4A_4|b$

   The above two productions still do not conform to any of the types in step 3.

   Substituting for $A_2 \to b$

   $A_4 \to bA_3A_4|A_4A_4A_4|b$

   Now we have to remove left recursive production $A_4 \to A_4A_4A_4$

$A_4 \rightarrow bA_3A_4|b|bA_3A_4Z|bZ$

$Z \rightarrow A_4A_4|A_4A_4Z$

6. At this stage our grammar now looks like

$A_1 \rightarrow A_2A_3|A_4A_4$

$A_4 \rightarrow bA_3A_4|b|bA_3A_4Z|bZ$

$Z \rightarrow A_4A_4|A_4A_4Z$

$A_2 \rightarrow b$

$A_3 \rightarrow a$

All rules now conform to one of the types in step 3.
But the grammar is still not in Greibach Normal Form!

7. All productions for A_2, A_3 and A_4 are in GNF

<u>for $A_1 \rightarrow A_2A_3|A_4A_4$</u>

Substitute for A_2 and A_4 to convert it to GNF

$A_1 \rightarrow bA_3|bA_3A_4A_4|bA_4|bA_3A_4ZA_4|bZA_4$

<u>for $Z \rightarrow A_4A_4|A_4A_4Z$</u>

Substitute for A_4 to convert it to GNF

$Z \rightarrow bA_3A_4A_4|bA_4|bA_3A_4ZA_4|bZA_4|bA_3A_4A_4Z|bA_4Z|bA_3A_4ZA_4Z|bZA_4Z$

8. Finally the grammar in GNF is

$A_1 \rightarrow bA_3|bA_3A_4A_4|bA_4|bA_3A_4ZA_4|bZA_4$

$A_4 \rightarrow bA_3A_4|b|bA_3A_4Z|bZ$

$Z \rightarrow bA_3A_4A_4|bA_4|bA_3A_4ZA_4|bZA_4|bA_3A_4A_4Z|bA_4Z|bA_3A_4ZA_4Z|bZA_4Z$

$A_2 \rightarrow b$

$A_3 \rightarrow a$

3

GNF Practice Problems:

1. S ----> aSb | aA, A ----> Aa | Sa |a
2. S ----> XY1 | 0, X ----> 00X | Y, Y ----> 1X1
3. S ----> 01 | 0S | 00S

4.5.2 Greibach Normal Form (G.N.F.)

A Context free grammar is said to be in Greibach-normal form (G.N.F.) if all production have the form

$$\boxed{A \to aX}$$

where, $a \in T$, $x \in N^*$ means "X-can be null".

Example *Convert the following grammar in "Greibach normal form".*

$$S \to AB$$
$$A \to aA|bB|b$$
$$B \to b$$

Answer Given grammar is not G.N.F.

By using the substitution

$\because$ $\quad S \to AB$

$\because$ $\quad A \to aA|bB|b$

Put the all values of 'A'

$$S \to aAB|bBB|bB$$

$$A \to aA|bB|b$$
$$B \to a$$

($\because$ X may be null) This is in G.N.F.

Example *Convert the grammar*

$$S \to abSb|aa \text{ into G.N.F.}$$

Answer $S \to aBSB|aA$

$$A \to a$$
$$B \to b \text{ which is in G.N.F.}$$

Example *Convert the following grammar into G.N.F. (Greibach normal form)*

$$S \to aSb \mid ab$$

Answer $\because$ $S \to a\,S\,b \mid ab$

Let $B \to b$

then $S \to a\,S\,B \mid a\,B$

$$B \to a$$

which is in G.N.F.

Example *Convert the grammar*

$$S \to ab \mid aS \mid aaS$$

into G.N.F.

Answer $\because$ $S \to ab \mid as \mid aaS$

Let $B \to b$

$$A \to a$$

$\therefore$ $S \to a\,B \mid a\,S \mid a\,AS$

$$A \to a$$
$$B \to b$$

which is in G.N.F.

Example *Convert the grammar*

$$S \to AB\,b \mid a$$
$$A \to aa\,A \mid B$$
$$B \to b\,A\,b$$

into G.N.F.

Answer $S \to AB\,b \mid a$

We can put the value of

$$A \to aa\,A$$

then

$$S \to aaA\,Bb \mid a$$

Let $P \to b$

$$Q \to a$$

then

$$S \to aQ\,ABP \mid a$$

$$P \to b$$
$$Q \to a$$

$\because$ $A \to aaA \mid B$

$$P \to b$$
$$Q \to a$$
$\because$ $$A \to aaA \mid B$$
$\because$ $$B \to bAb$$
Therefore

$$A \to aaA \mid bAb$$
$$A \to aQA \mid bAP$$
$$(\because P \to b,\ Q \to a)$$
$$B \to bAP$$

Final G.N.F. is

$$S \to aQABP \mid a$$
$$P \to b$$
$$Q \to a$$
$$A \to aQA \mid bAP$$
$$B \to bAP$$

Example *Convert thc following grammar into G.N.F.*

$$S \to aAS$$
$$S \to a$$
$$A \to SbA$$
$$A \to SS$$
$$A \to ba$$

Answer
$$S \to aAS$$
$$S \to a$$

both are in G.N.F.

$$A \to SbA$$
$$A \to SS$$
$$A \to ba$$

are not in G.N.F.

we will put the productions of 'S'.

Therefore,

$$A \to aASbA$$
$$A \to abA$$
$$A \to aASS$$
$$A \to aS$$
$$A \to ba$$

Let $$B \to b$$

then $$A \to aASBA$$
$$A \to aBA$$

$$A \to aASS$$
$$A \to aS$$
$$A \to ba$$
$$C \to a$$
$$A \to aASBA$$
$$A \to aBA$$
$$A \to aASS$$
$$A \to aS$$
$$A \to bC$$
$$B \to b$$
$$C \to a$$

al G.N.F. is

$$S \to aAS$$
$$S \to a$$
$$A \to aASBA$$
$$A \to aBA$$
$$A \to aASS$$
$$A \to aS$$
$$A \to bC$$
$$B \to b$$
$$C \to a$$

Note: Generally we see that $A \to A\alpha \mid \beta$ type production in C.F.G., we can replace it by following two production.

$$A \to \beta A' \mid \beta$$
$$A' \to \alpha A' \mid \alpha$$

Here, A' is a new non-terminal by this way we actually remove left recursion from the grammar.

Example 6.41 Convert the following grammar into GNF.

$$S \rightarrow AA/a$$
$$A \rightarrow SS/b$$

Solution:

Step I: There are no unit productions and no null production in the grammar. The given grammar is in CNF.

Step II: In the grammar, there are two non-terminals S and A. Rename the non-terminals as A_1 and A_2, respectively. The modified grammar will be

$$A_1 \rightarrow A_2A_2/a$$
$$A_2 \rightarrow A_1A_1/b$$

Step III: In the grammar, $A_2 \rightarrow A_1A_1$ is not in the format $A_i \rightarrow A_jV$ where $i \leq j$. Replace the leftmost A_1 at the RHS of the production $A_2 \rightarrow A_1A_1$. After replacing the modified A_2, production will be

$$A_2 \rightarrow A_2A_2A_1/aA_1/b$$

The production $A_2 \rightarrow aA_1/b$ is in the format $A \rightarrow \beta_i$ and the production $A_2 \rightarrow A_2A_2A_1$ is in the format of $A \rightarrow A\alpha_j$ So, we can introduce a new non-terminal B_2 and the modified A_2 production will be (according to Lemma II)

$$A_2 \rightarrow aA_1/b$$
$$A_2 \rightarrow aA_1B_2$$
$$A_2 \rightarrow bB_2$$

And the B_2 productions will be

$$B_2 \rightarrow A_2A_1$$
$$B_2 \rightarrow A_2A_1B_2$$

Step IV: All A_2 productions are in the format of GNF. In the production $A_1 \rightarrow A_2A_2/a$, $A \rightarrow a$ is in the prescribed format. But the production $A_1 \rightarrow A_2A_2$ is not in the format of GNF. Replace the leftmost A_2 at the RHS of the production by the previous A_2 productions. The modified A_1 productions will be

$$A_1 \rightarrow aA_1A_2/bA_2/aA_1B_2A_2/bB_2A_2$$

The B_2 productions are not in GNF. Replace the leftmost A_2 at the RHS of the two productions by the A_2 productions. The modified B_2 productions will be

$$B_2 \rightarrow aA_1A_1/bA_1/aA_1B_2A_1/bB_2A_1$$
$$B_2 \rightarrow aA_1A_1B_2/bA_1B_2/aA_1B_2A_1B_2/bB_2A_1B_2$$

For the given CFG, the GNF will be

$$A1 \rightarrow aA_1A_2/bA_2/aA_1B_2A_2/bB_2A_2/a$$
$$A2 \rightarrow aA_1/b/aA_1B_2/bB_2$$
$$B2 \rightarrow aA_1A_1/bA_1/aA_1B_2A_1/bB_2A_1$$
$$B2 \rightarrow aA_1A_1B_2/bA_1B_2/aA_1B_2A_1B_2/bB_2A_1B_2$$

Example 6.42 Convert the following CFG into GNF.

$$S \rightarrow XY$$
$$X \rightarrow YS/b$$
$$Y \rightarrow SX/a$$

Solution:

Step I: In the grammar, there is no null production and no unit production. The grammar also is in CNF.

Step II: In the grammar, there are three non-terminals S, X, and Y. Rename the non-terminals as A_1, A_2, and A_3, respectively. After renaming, the modified grammar will be

$$A_1 \rightarrow A_2A_3$$
$$A_2 \rightarrow A_3A_1/b$$
$$A_3 \rightarrow A_1A_2/a$$

Step III: In the grammar, the production $A_3 \rightarrow A_1A_2$ is not in the format $A_i \rightarrow A_jV$ where $i \leq j$.

Replace the leftmost A_1 at the RHS of the production $A_3 \rightarrow A_1A_2$ by the production $A_1 \rightarrow A_2A_3$. The production will become $A_3 \rightarrow A_2A_3A_2$, which is again not in the format of $A_i \rightarrow A_jV$ where $i \leq j$. Replace the leftmost A_2 at the RHS of the production $A_3 \rightarrow A_2A_3A_2$ by the production $A_2 \rightarrow A_3A_1/b$. The modified A_3 production will be

$$A_3 \rightarrow A_3A_1A_3A_2/bA_3A_2/a$$

The production $A_3 \rightarrow bA_3A_2/a$ is in the format of $A \rightarrow \beta_i$ and the production $A_3 \rightarrow A_3A_1A_3A_2$ is in the format of $A \rightarrow A\alpha_j$. So, we can introduce a new non-terminal B and the modified A_3 production will be (according to Lemma II)

$$A_3 \rightarrow bA_3A_2$$
$$A_3 \rightarrow a$$
$$A_3 \rightarrow bA_3A_2B$$
$$A_3 \rightarrow aB$$

And B productions will be

$$B \rightarrow A_1A_3A_2$$
$$B \rightarrow A_1A_3A_2B$$

Step IV: All the A_3 productions are in the specified format of GNF.

The A_2 production is not in the specified format of GNF. Replacing A_3 productions in A_2 productions, the modified A_2 production becomes

$$A_2 \to bA_3A_2A_1/aA_1/bA_3A_2BA_1/aBA_1/b$$

Now, all the A_2 productions are in the prescribed format of GNF.

The A_1 production is not in the prescribed format of GNF. Replacing A_2 productions in A_1, the modified A_1 productions will be

$$A_1 \to bA_3A_2A_1A_3/aA_1A_3/bA_3A_2BA_1A_3/aBA_1A_3/bA_3$$

All the A_1 productions are in the prescribed format of GNF.

But the B productions are still not in the prescribed format of GNF. By replacing the leftmost A_1 at the RHS of the B productions by A_1 productions, the modified B productions will be

$$B \to bA_3A_2A_1A_3A_3A_2/aA_1A_3A_3A_2/bA_3A_2BA_1A_3A_3A_2/aBA_1A_3A_3A_2/bA_3A_3A_2$$
$$B \to bA_3A_2A_1A_3A_3A_2B/aA_1A_3A_3A_2B/bA_3A_2BA_1A_3A_3A_2B/aBA_1A_3A_3A_2B/bA_3A_3A_2B.$$

Now, all the B productions of the grammar are in the prescribed format of GNF.

So, for the given CFG, the GNF will be

$$A_1 \to bA_3A_2A_1A_3/aA_1A_3/bA_3A_2BA_1A_3/aBA_1A_3/bA_3$$
$$A_2 \to bA_3A_2A_1/aA_1/bA_3A_2BA_1/aBA_1/b$$
$$B \to bA_3A_2A_1A_3A_3A_2/aA_1A_3A_3A_2/bA_3A_2BA_1A_3A_3A_2/aBA_1A_3A_3A_2/bA_3A_3A_2$$
$$B \to bA_3A_2A_1A_3A_3A_2B/aA_1A_3A_3A_2B/bA_3A_2BA_1A_3A_3A_2B/aBA_1A_3A_3A_2B/bA_3A_3A_2B$$

Example 6.43 Convert the following CFG into GNF.

$$S \to AB/BC$$
$$A \to aB/bA/a$$
$$B \to bB/cC/b$$
$$C \to c$$

Solution:

Step I: In the previous grammar, there is no unit production and no null production. But all productions are not in CNF. Let us take two non-terminals D_a and D_b which will be placed in the place of 'a' and 'b', respectively. So, two new productions $D_a \to a$ and $D_b \to b$ will be added to the grammar

$$S \to AB/BC$$
$$A \to D_aB/D_bA/a$$
$$B \to D_bB/CC/b$$
$$C \to c$$
$$D_a \to a$$
$$D_b \to b$$

Now all the productions are in CNF.

Step II: There are six non-terminals in the grammar. Rename the non-terminals as A_1, A_2 ..A_6. After replacing, the modified productions will be

$$A_1 \to A_2A_3/A_3A_4$$
$$A_2 \to A_5A_3/A_6A_2/a$$
$$A_3 \to A_6A_3/A_4A_4/b$$
$$A_4 \to c$$
$$A_5 \to a$$
$$A_6 \to b$$

The productions for A.., A.. and A are all in the format A. → A.V where $i \le j$. Replace A_6 and A_4

Step II: There are six non-terminals in the grammar. Rename the non-terminals as $A_1, A_2 .. A_6$. After replacing, the modified productions will be

$$A_1 \to A_2A_3/A_3A_4$$
$$A_2 \to A_5A_3/A_6A_2/a$$
$$A_3 \to A_6A_3/A_4A_4/b$$
$$A_4 \to c$$
$$A_5 \to a$$
$$A_6 \to b$$

The productions for A_1, A_2, and A_3 are all in the format $A_i \to A_jV$ where $i \le j$. Replace A_6 and A_4 in the productions $A_3 \to A_6A_3$ and $A_3 \to A_4A_4$ by $A_6 \to b$ and $A_4 \to c$, respectively. The modified A_3 productions will be

$$A_3 \to bA_3/cA_4/b$$

All the productions are now in the format of GNF.
Replace A_5 and A_6 in the productions $A_2 \to A_5A_3$ and $A_2 \to A_6A_2$ by $A_5 \to a$ and $A_6 \to b$, respectively. The modified A_2 productions will be

$$A_2 \to aA_3/bA_2/a$$

All the productions are now in the format of GNF.
The A_1 productions $A_1 \to A_2A_3/A_3A_4$ are not in the format of GNF. Replace A_2 at the RHS of the production $A_1 \to A_2A_3$. The modified production will be

$$A_1 \to aA_3A_3/bA_2A_3/aA_3$$

Replace A_3 at the RHS of the production $A_1 \to A_3A_4$. The modified production will be

$$A_1 \to bA_3A_4/cA_4A_4/bA_4$$

So, for the given CFG, the GNF will be

$$A_1 \to aA_3A_3/bA_2A_3/aA_3/bA_3A_4/cA_4A_4/bA_4$$
$$A_2 \to aA_3/bA_2/a$$
$$A_3 \to bA_3/cA_4/b$$
$$A_4 \to c$$
$$A_5 \to a$$
$$A_6 \to b$$

Pumping Lemma of CFL

15 September 2018 06:47 PM

3.3 PROPERTIES OF CONTEXT FREE LANGUAGES

3.3.1 Pumping Lemma for CFG

A "Pumping Lemma" is a theorem used to show that, if certain strings belong to a language, then certain other strings must also belong to the language.

Let us discuss a Pumping Lemma for CFL.

We will show that , if L is a context-free language, then strings of L that are at least 'm' symbols long can be "pumped" to produce additional strings in L. The value of 'm' depends on the particular language.

Let L be an infinite context-free language. Then there is some positive integer 'm' such that, if S is a string of L of Length at least 'm', then

(i) $S = uvwxy$ (for some u, v, w, x, y)
(ii) $|vwx| \leq m$
(iii) $|vx| \geq 1$
(iv) $uv^i wx^i y \in L$.

for all non-negative values of i.

It should be understood that

(i) If S is sufficiently long string, then there are two substrings, v and x, somewhere in S. There is stuff (u) before v, stuff (w) between v and x, and stuff (y), after x.
(ii) The stuff between v and x won't be too long, because $|vwx|$ can't be larger than m.
(iii) Substrings v and x won't both be empty, though either one could be.
(iv) If we duplicate substring v, some number (i) of times, and duplicate x the same number of times, the resultant string will also be in L.

3.3.2 Definitions

A variable is useful if it occurs in the derivation of some string. This requires that

(a) the variable occurs in some sentential form (you can get to the variable if you start from S), and
(b) a string of terminals can be derived from the sentential form (the variable is not a "dead end").

A variable is "recursive" if it can generate a string containing itself. For example, variable A is recursive if

$$S \overset{*}{\Rightarrow} uAy$$

for some values of u and y.

A recursive variable A can be either

(i) "Directly Recursive", i.e., there is a production

$$A \rightarrow x_1 A x_2$$

for some strings $x_1, x_2 \in (T \cup V)^*$, or

(ii) "Indirectly Recursive", i.e., there are variables x_i and productions

$$\begin{aligned} &A \rightarrow X_1 \ldots \\ &X_1 \rightarrow \ldots X_2 \ldots \\ &X_2 \rightarrow \ldots X_3 \ldots \\ &X_N \rightarrow \ldots A \ldots \end{aligned}$$

3.3.3 Proof of Pumping Lemma

(a) Suppose we have a CFL given by L. Then there is some context-free Grammar G that generates L. Suppose

(i) L is infinite, hence there is no proper upper bound on the length of strings belonging to L.

(ii) L does not contain λ.

(iii) G has no productions or λ-productions.

There are only a finite number of variables in a grammar and the productions for each variable have finite lengths. The only way that a grammar can generate arbitrarily long strings is if one or more variables is both useful and recursive.

Suppose no variable is recursive.

Since the start symbol is nonrecursive, it must be defined only in terms of terminals and other variables. Then since those variabls are non recursive, they have to be defined in terms of terminals and still other variables and so on. After a while we run out of "other variables" while the generated string is still finite. Therefore there is an upperbond on the length of the string which can be generated from the start symbol. This contradicts our statement that the language is finite.

Hence, our assumption that no variable is recursive must be incorrect.

(b) Let us consider a string X belonging to L.

If X is sufficiently long, then the derivation of X must have involved recursive use of some variable A.

Since A was used in the derivation, the derivation should have started as

$$S \overset{*}{\Rightarrow} uAy$$

for some values of u and y. Since A was used recursively the derivation must have continued as

$$S \overset{*}{\Rightarrow} uAy \overset{*}{\Rightarrow} uvAxy$$

Finally the derivation must have eliminated all variables to reach a string X in the language.

$$S \overset{*}{\Rightarrow} uAy \overset{*}{\Rightarrow} uvAxy \overset{*}{\Rightarrow} uvwxy = x$$

This shows that derivation steps

$$A \overset{*}{\Rightarrow} vAx$$

and

$$A \overset{*}{\Rightarrow} w$$

are possible. Hence the derivation

$$A \overset{*}{\Rightarrow} vwx$$

must also be possible.

It should be noted here that the above does not imply that a was used recursively only once. The * of $\overset{*}{\Rightarrow}$ could cover many uses of A, as well as other recursive variables.

There has to be some "last" recursive step. Consider the longest strings that can be derived for v, w and x without the use of recursion. Then there is a number 'm' such that $|vwx| < m$.

Since the grammar does not contain any λ-productions or unit productions, every derivation step either introduces a terminal or increases the length of the sentential form. Since $A \overset{*}{\Rightarrow} vAx$, it follows that $|vx| > 0$.

Finally, since $uvAxy$ occurs in the derivation, and $A \overset{*}{\Rightarrow} vAx$ and $A \overset{*}{\Rightarrow} w$ are both possible, it follows that $uv^i wx^i y$ also belongs to L.

This completes the proof of all parts of Lemma.

3.3.4 Usage of Pumping Lemma

The Pumping Lemma can be used to show that certain languages are not context free.

Let us show that the language

$$L = \{a^i b^i c^i \mid i > 0\}$$

is not context-free.

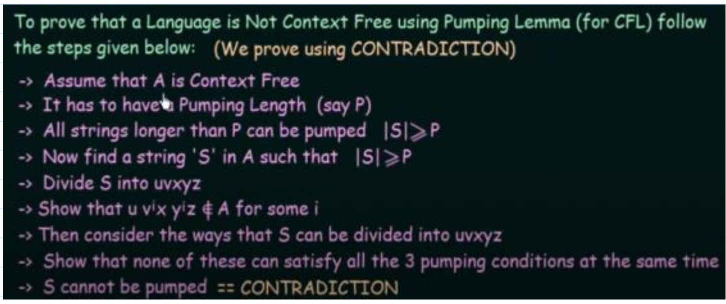
Proof: Suppose L is a context-free language.

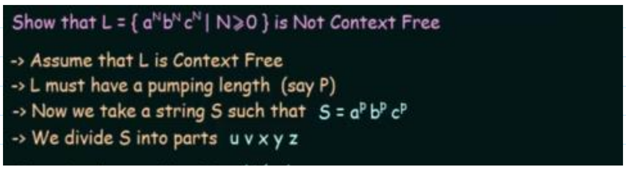
If string $X \in L$, where $|X| > m$, it follows that $X = uvwxy$, where $|vwx| \le m$.

Choose a value i that is greater than m. Then, wherever vwx occurs in the string $a^i b^i c^i$, it cannot contain more than two distinct letters it can be all a's, all b's, all c's, or it can be a's and b's, or it can be b's and c's.

Therefore the string vx cannot contain more than two distinct letters; but by the "Pumping Lemma" it cannot be empty, either, so it must contain at least one letter.

Now we are ready to "pump".





Therefore the string vx cannot contain more than two distinct letters; but by the "Pumping Lemma" it cannot be empty, either, so it must contain at least one letter.

Now we are ready to "pump".

Since $uvwxy$ is in L, uv^2wx^2y must also be in L. Since v and x can't both be empty,

$$|uv^2wx^2y| > |uvwxy|,$$

so we have added letters.

Both since vx does not contain all three distinct letters, we cannot have added the same number of each letter.

Therefore, uv^2wx^2y cannot be in L.

Thus we have arrived at a "contradiction".

Hence our original assumption, that L is context free should be false.
Hence the language L is not context-free. □

Example 3.3.1: Check whether the language given by

$$L = \{a^m b^m c^n \; : m \le n \le 2m\}$$

is a CFL or not.

Solution

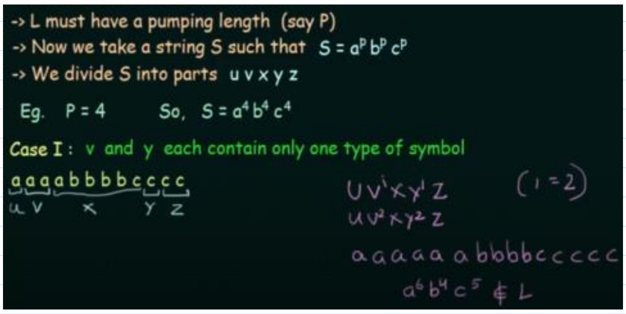
Let $s = a^n b^n c^{2n}$, n being obtained from Pumping Lemma.

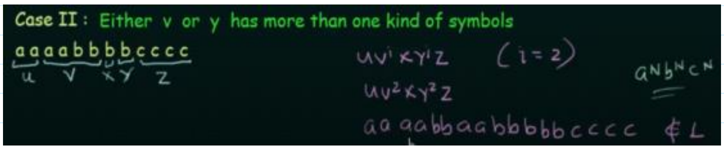
Then $s = uvwxy$, where $1 \le |vx| \le n$.

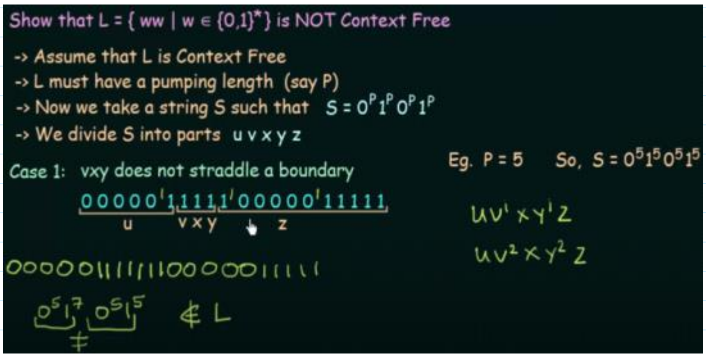
Therefore, vx cannot have all the three symbols a, b, c.

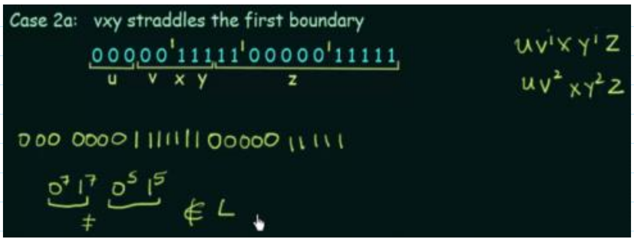
If you assume that vx has only a's and b's then we can shoose i such that uv^iwx^iy has more than $2n$ occurrence of a or b and exactly $2n$ occurences of c.

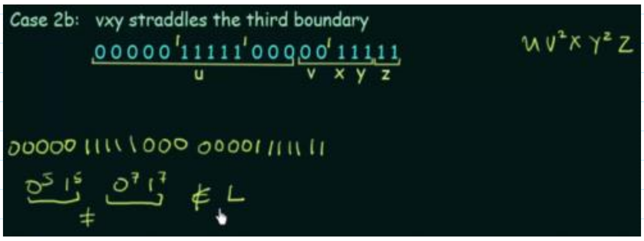
Hence $uv^iwx^iy \notin L$, which is a contradiction. Hence L is not a CFL.

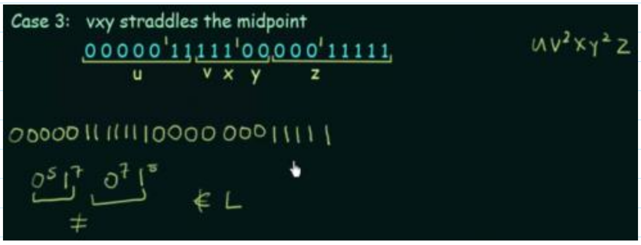












Example 3.3.4: Check whether the language given by

$$L = \{w \in \{a, b, c\}^* \mid n_a(w) = n_b(w) = n_c(w)\}$$

is not context-free.

Proof: If L is assumed to be context-free, then

$$L \cap L(a^* b^* c^*) = \{a^n b^n c^n \mid n \geq 0\}.$$

which is also context-free.

But it is a fact that the latter is not context-free.

Therefore we conclude that

$$L = \{w \in \{a, b, c\}^* \mid n_a(w) = n_b(w) = n_c(w)\}$$

is not context-free. □

Example 3.3.5: Determine whether the language given by $L = \{a^{n^2} \mid n \geq 1\}$ is context-free or not.

Solution

Let us assume that

$$s = a^{n^2}.$$

$s = uvwxy$, where $1 \leq |vx| \leq n$. which is true

since, $|vwx| \leq n$ (by Pumping Lemma)

Let $|vx| = m,\ m \leq n.$

By Pumping Lemma, uv^2wx^2y is in L.

Since
$$|uv^2wx^2y| > n^2,$$
$$|uv^2wx^2y| = k^2.$$

where $k \geq n+1$.

But $|uv^2wx^2y| = n^2 + m < n^2 + 2n + 1$.

Therefore, $|uv^2wx^2y|$ lies between n^2 and $(n+1)^2$.

Hence, $uv^2wx^2y \notin L$, which is a contradiction.

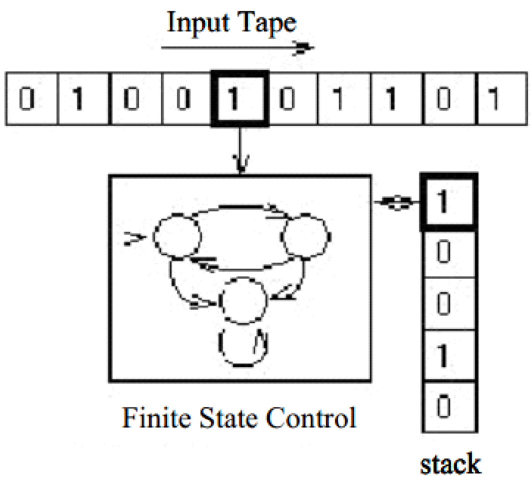
Therefore, $\{a^{n^2} : n \geq 1\}$ is not context-free.

UNIT - IV

30 June 2017 07:14 PM

PDA model

09 October 2017 06:06 AM



Formal Definition:

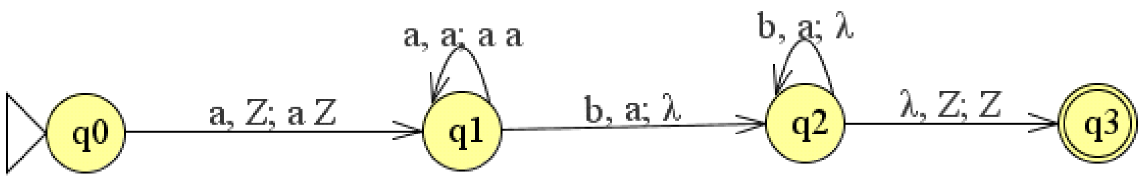
A *nondeterministic pushdown automaton* or *npda* is a 7-tuple M = ($Q, \Sigma, \Gamma, \delta, q_0, z, F$), where

- Q is a finite set of *states,*
- Σ is a the *input alphabet,*
- Γ is the *stack alphabet,*
- δ is a *transition function,* has the form
- δ: Q X ($\Sigma \cup \{\in\}$) X $\Gamma \rightarrow$ finite subsets of Q X Γ^*
- $q_0 \in Q$ is the *initial state,*
- $z \in \Gamma$ is the *stack start symbol,* and
- $F \subseteq Q$ is a set of *final states.*

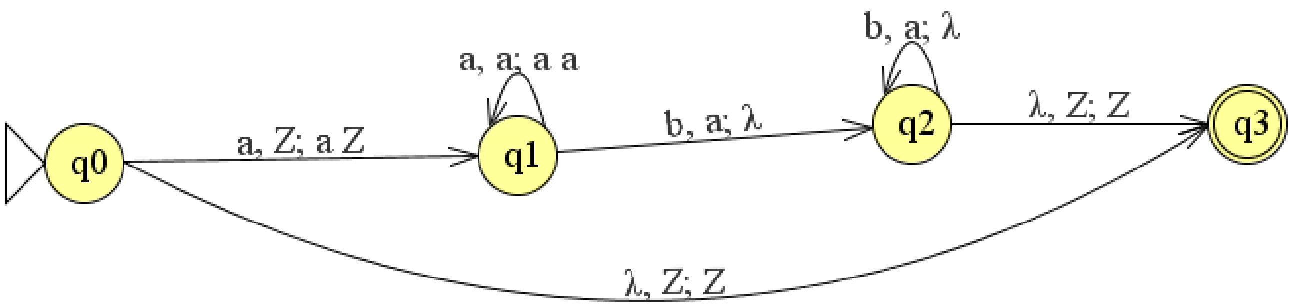
PDA Design

10 October 2017 11:59 AM

1 L = {$a^n b^n$: n >= 1}

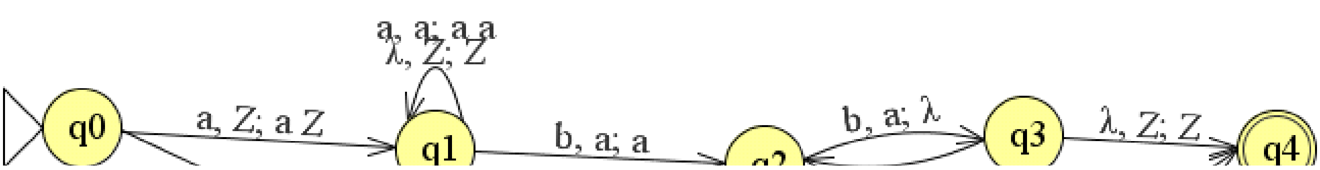


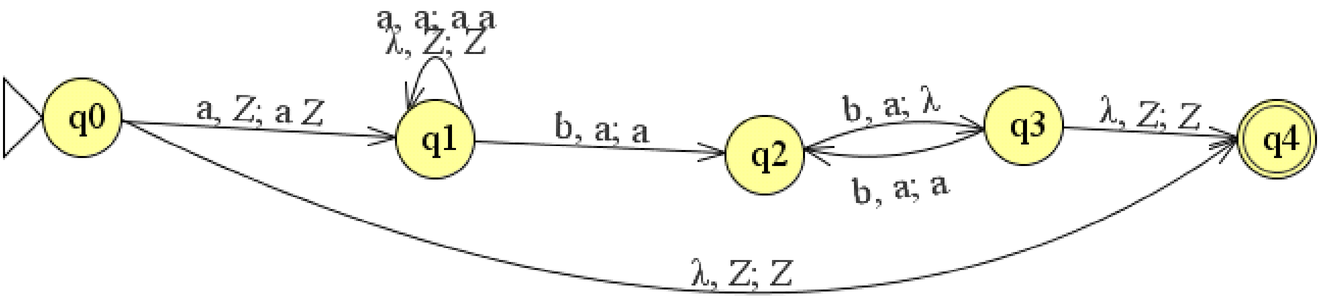
2 L = { $a^n b^n$: n>=0}



Input	Result
a a b b b b	Reject
a a b b b b b b	Reject
λ	Accept
a a a a b b b b	Accept

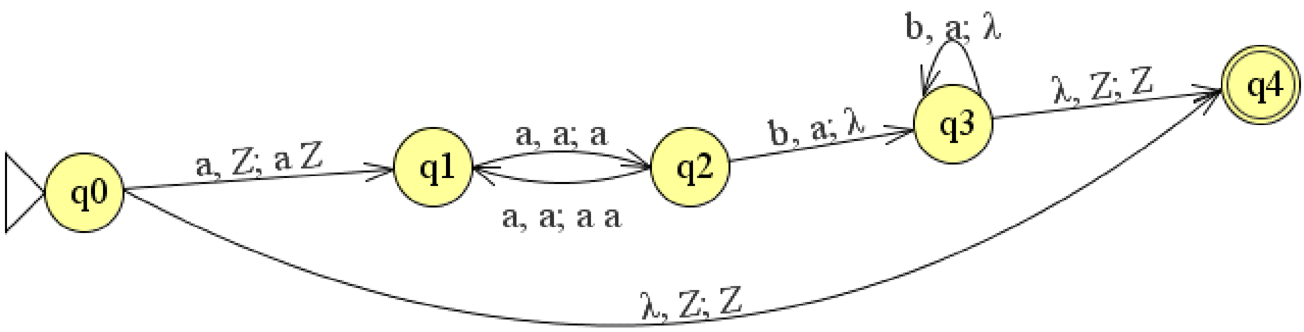
3 L = {$a^n b^{2n}$: n >=0}





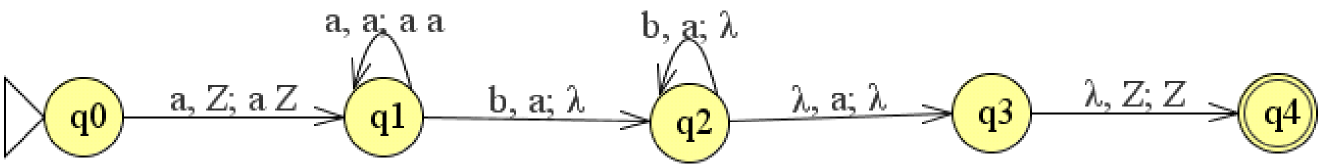
Input	Result
a a b b b b	Accept
a a b b b b b b	Reject
λ	Accept

4 L = $\{a^{2n}b^n : n >= 0\}$

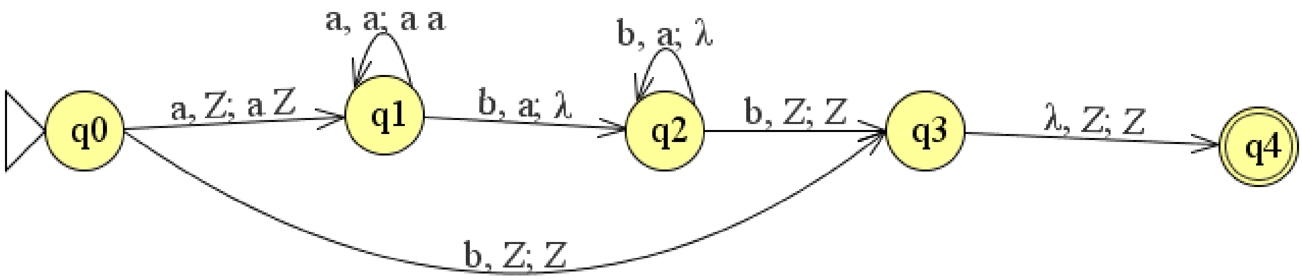


Input	Result
a a a a b b	Accept
a a b	Accept
λ	Accept
a a a b b	Reject

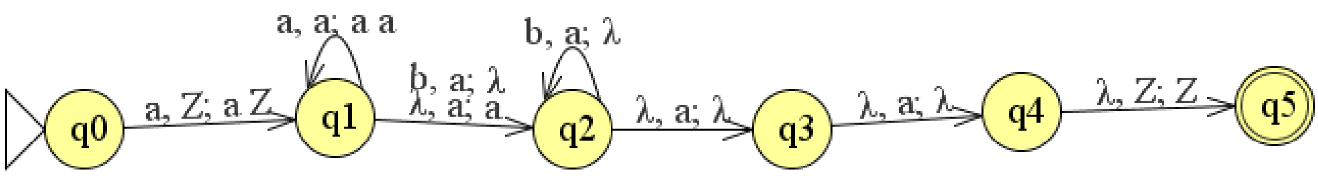
5 L = $\{a^{n+1}b^n : n >=0\}$



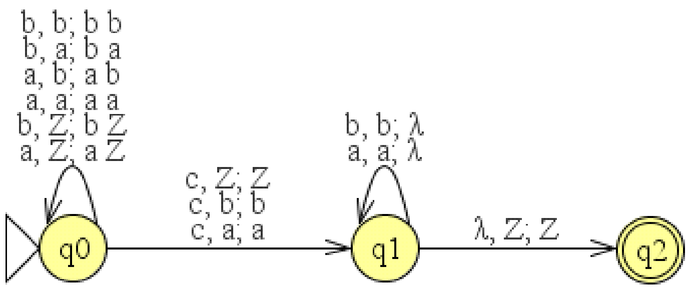
6 L = $\{a^n b^{n+1} : n >= 0\}$



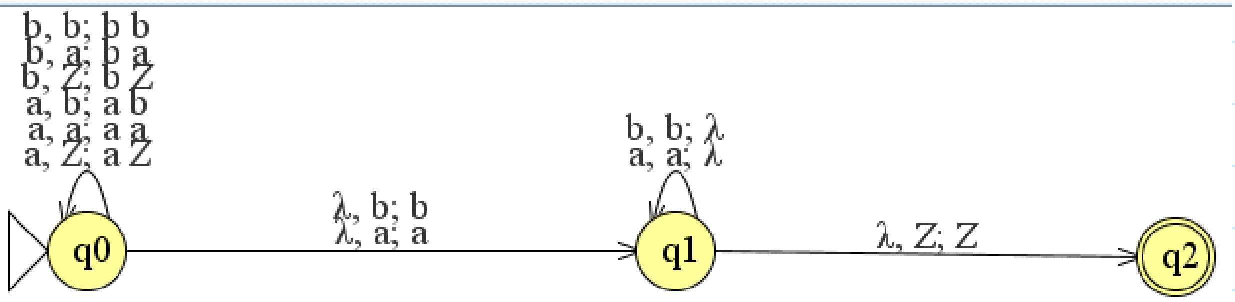
7 L = $\{a^{n+2} b^n : n >=0\}$



8 L = $\{wcw^R : w \in \{0,1\}^*\}$

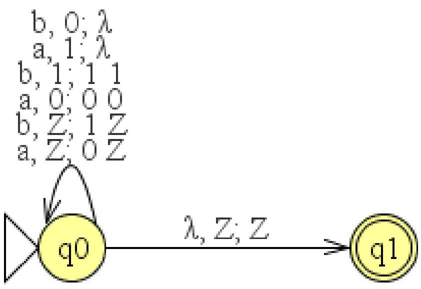


9 L = $\{ww^R : w \in \{0,1\}^*\}$



Input	Result
a b b a	Accept
a a b b a a	Accept
a b a a a a b a	Accept

10 L = {$n_a(w) = n_b(w)$: w ϵ {a,b}*}



CFG to PDA Conversion

15 October 2017 07:21 AM

1. Convert the given grammar into Griebach Normal Form (GNF).
2. We construct PDA with 3 states (q0, q1 and qf) as follows: (Assuming q0 is initial state and qf is final state) (Also assume that Z is initial symbol on stack)
3. Push Start variable (S) into stack without reading input symbol and change state from q0 to q1. δ(q0, λ, Z) = (q1, SZ)
4. For each production of the form: A → aα write the following PDA moves: δ(q1, a, A) = (q1, α)
5. Finally, make a transition from state q1 to final state qf as : δ(q1, λ, Z) = (qf, Z)

CFG to NPDA

For any context-free grammar in GNF, it is easy to build an equivalent nondeterministic pushdown automaton (NPDA).

Any string of a context-free language has a leftmost derivation. We set up the NPDA so that the stack contents "corresponds" to this sentential form: every move of the NPDA represents one derivation step.

The sentential form is

(The characters already read) + (symbols on the stack)
– (Final z (initial stack symbol)

In the NPDA, we will construct, the states that are not of much importance. All the real work is done on the stack. We will use only the following three states, irrespective of the complexity of the grammar.

(i) start state q_0 just gets things initialized. We use the transition from q_0 to q_1 to put the grammar's start symbol on the stack.

$$\delta(q_0, \lambda, Z) \to \{(q_1, Sz)\}$$

(ii) State q_1 does the bulk of the work. We represent every derivation step as a move from q_1 to q_1.

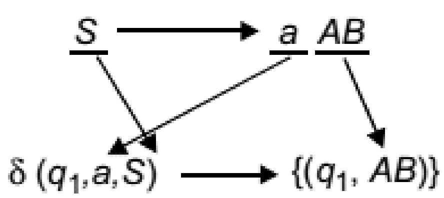
(iii) We use the transition from q_1 to q_f to accept the string

$$\delta(q_1, \lambda, z) \to \{(q_f, z)\}$$

Example Consider the grammar $G = (\{S, A, B\}, \{a, b\}, S, P)$, where

$$P = \{S \to a, S \to aAB, A \to aA, A \to a, B \to bB, B \to b\}$$

These productions can be turned into transition functions by rearranging the components.



Thus we obtain the following table:

Thus we obtain the following table:

(Start)	$\delta(q_0, \lambda, z) \to \{(q_1, Sz)\}$
$S \to a$	$\delta(q_1, a, S) \to \{(q_1, \lambda)\}$
$S \to aAB$	$\delta(q_1, a, S) \to \{(q_1, AB)\}$
$A \to aA$	$\delta(q_1, a, A) \to \{(q_1, A)\}$
$A \to a$	$\delta(q_1, a, A) \to \{(q_1, \lambda)\}$
$B \to bB$	$\delta(q_1, b, B) \to \{(q_1, B)\}$
$B \to b$	$\delta(q_1, b, B) \to \{(q_1, \lambda)\}$
(finish)	$\delta(q_1, \lambda, z) \to \{(q_f, z)\}$

For example, the derivation

$$S \Rightarrow aAB \Rightarrow aaB \Rightarrow aabB \Rightarrow aabb$$

maps into the sequence of moves

$$\begin{aligned}(q_0, aabb, z) &\vdash (q_1, aabb, Sz)\\ &\vdash (q_1, abb, ABz)\\ &\vdash (q_1, bb, Bz)\\ &\vdash (q_1, b, Bz)\\ &\vdash (q_1, \lambda, z)\\ &\vdash (q_2, \lambda, \lambda)\end{aligned}$$

Construct a pda that accepts the language generated by a grammar with productions

$$S \rightarrow aSbb|a.$$

We first transform the grammar into Greibach normal form, changing the productions to

$$\begin{aligned} S &\rightarrow aSA|a, \\ A &\rightarrow bB, \\ B &\rightarrow b. \end{aligned}$$

The corresponding automaton will have three states $\{_0, q_1, q_2\}$, with initial state q_0 and final state q_2. First, the start symbol S is put on the stack by

$$\delta(q_0, \lambda, z) = \{(q_1, Sz)\}.$$

The production $S \rightarrow aSA$ will be simulated in the pda by removing S from the stack and replacing it with SA, while reading a from the input. Similarly, the rule $S \rightarrow a$ should cause the pda to read an a while simply removing S. Thus, the two productions are represented in the pda by

$$\delta(q_1, a, S) = \{(q_1, SA), (q_1, \lambda)\}.$$

In an analogous manner, the other productions give

$$\begin{aligned} \delta(q_1, b, A) &= \{(q_1, B)\}, \\ \delta(q_1, b, B) &= \{(q_1, \lambda)\}. \end{aligned}$$

The appearance of the stack start symbol on top of the stack signals the completion of the derivation and the pda is put into its final state by

$$\delta(q_1, \lambda, z) = \{(q_2, \lambda)\}.$$

The construction of this example can be adapted to other cases, leading toa general result.

Example 7.7

Consider the grammar

$$\begin{aligned} S &\rightarrow aA, \\ A &\rightarrow aABC\,|bB|\,a, \\ B &\rightarrow b, \\ C &\rightarrow c. \end{aligned}$$

Since the grammar is already in Greibach normal form, we can use the construction in the previous theorem immediately. In addition to rules

$$\delta(q_0, \lambda, z) = \{(q_1, Sz)\}$$

and

$$\delta(q_1, \lambda, z) = \{(q_f, z)\},$$

the pda will also have transition rules

$$\begin{aligned}
\delta(q_1, a, S) &= \{(q_1, A)\}, \\
\delta(q_1, a, A) &= \{(q_1, ABC), (q_1, \lambda)\}, \\
\delta(q_1, b, A) &= \{(q_1, B)\}, \\
\delta(q_1, b, B) &= \{(q_1, \lambda)\}, \\
\delta(q_1, c, C) &= \{(q_1, \lambda)\}.
\end{aligned}$$

The sequence of moves made by M in processing *aaabc* is

$$\begin{aligned}
(q_0, aaabc, z) &\vdash (q_1, aaabc, Sz) \\
&\vdash (q_1, aabc, Az) \\
&\vdash (q_1, abc, ABCz) \\
&\vdash (q_1, bc, BCz) \\
&\vdash (q_1, c, Cz) \\
&\vdash (q_1, \lambda, z) \\
&\vdash (q_f, \lambda, z).
\end{aligned}$$

This corresponds to the derivation

$$S \Rightarrow aA \Rightarrow aaABC \Rightarrow aaaBC \Rightarrow aaabC \Rightarrow aaabc.$$

Practice Problems

Construct an npda that accepts the language generated by the grammar

$$S \to aSbb|aab.$$

Construct an npda that accepts the language generated by the grammar $S \to aSSS|ab$.

Construct an npda corresponding to the grammar

$$\begin{aligned}
S &\to aABB|aAA, \\
A &\to aBB|a, \\
B &\to bBB|A.
\end{aligned}$$

Construct an npda that will accept the language generated by the grammar $G = (\{S, A\}, \{a, b\}, S, P)$, with productions $S \to AA\,|a$, $A \to SA|\,b$.

PDA to CFG conversion

15 October 2017 07:21 AM

1. Given PDA, M = (Q, Σ, δ, q0, F, Γ, Z) we have to find G = (V, T, P, S) as follows:
2. T= Σ (all input symbols of PDA become terminal symbols of CFG)
3. Variables are triplet form: if there exists a PDA move as δ(qi, a, Z) = (qj, AZ) then variable corresponding to this move is (qiZqj).
4. Start variable: if q0, qf are initial and final states respectively, and Z is the initial symbol of stack then start variable is (q0Zqf)
5. To write productions, PDA moves must perform stack operation, either PUSH or POP. Otherwise, rewrite the PDA move. For example, δ(qi, a, A) = (qj, A), (stack content is not modified after transition)
   δ(qi, a, A) = (qk, λ)
   δ(qk, λ, Z) = (qj, AZ)
6. PDA moves for pop operations: δ(qi, a, A) = (qj, λ)
   (qiAqj) → a
7. PDA moves for push operations: δ(qi, a, A) = (qj, BC)
   (qiAqk) → a (qjBql) (qlCqk) for all values of qk and ql.

PDA to CFG

As we have converted CFG to PDA, we can convert a given PDA to CFG. The general procedure for this conversion is shown below:

1. The input symbols of PDA will be the terminals of CFG.
2. If the PDA moves from state to q_i to state q_j on consuming the input $a \in \Sigma$ when Z is the top of the stack, then the non-terminals of CFG are the triplets of the form (q_iZq_j).
3. If q_0 is the start state and q_f is the final state then (q_0Zq_f) is the start symbol of CFG.
4. The productions of CFG can be obtained from the transitions of PDA as shown below:
   a. For each transition of the form

   $$\delta(q_i, a, Z) = (q_j, AB)$$

   introduce the productions of the form

   $$(q_iZq_k) \rightarrow a\ (q_jAq_l)(q_lBq_k)$$

   where q_k and q_l will take all possible values from Q.

   b. For each transition of the form

   $$\delta(q_i, a, Z) \quad = \quad (q_j, \varepsilon)$$

   introduce the production

   $$(q_iZq_j) \rightarrow a$$

Note: Using this procedure, we may introduce lot of useless symbols, which in any way can be eliminated.

Example 5.22: Obtain a CFG for the PDA shown below:

$$\delta(q_0, a, Z) = (q_0, AZ)$$
$$\delta(q_0, a, A) = (q_0, A)$$
$$\delta(q_0, b, A) = (q_1, \varepsilon)$$
$$\delta(q_1, \varepsilon, Z) = (q_2, \varepsilon)$$

Note: To obtain a CFG from the PDA, all the transitions should be of the form

$$\delta(q_i, a, Z) = (q_j, AB)$$

or

$$\delta(q_i, a, Z) = (q_j, \varepsilon)$$

In the given transitions except the second transition, all transitions are in the required form. So, let us take the second transition

$$\delta(q_0, a, A) \quad = \quad (q_0, A)$$

and convert it into the required form. This can be achieved if we have understood what the transition indicates. It is clear from the transition that when input symbol *a* is encountered and top of the stack is A, the PDA remains in state q_0 and contents of the stack are not altered. This can be interpreted as delete A from the stack and insert A onto the stack.

So, once A is deleted from the stack we enter into new state q_3. But, in state q_3 without consuming any input we add A on to the stack. The corresponding transitions are:

$$\delta(q_0, a, A) \quad = \quad (q_3, \varepsilon)$$
$$\delta(q_3, \varepsilon, Z) \quad = \quad (q_0, AZ)$$

So, the given PDA can be written using the following transitions:

$$\delta(q_0, a, Z) \quad = \quad (q_0, AZ)$$
$$\delta(q_0, a, A) \quad = \quad (q_3, \varepsilon)$$
$$\delta(q_3, \varepsilon, Z) \quad = \quad (q_0, AZ)$$
$$\delta(q_0, b, A) \quad = \quad (q_1, \varepsilon)$$
$$\delta(q_1, \varepsilon, Z) \quad = \quad (q_2, \varepsilon)$$

Now, the transitions

$$\delta(q_0, a, A) \quad = \quad (q_3, \varepsilon)$$
$$\delta(q_0, b, A) \quad = \quad (q_1, \varepsilon)$$
$$\delta(q_1, \varepsilon, Z) \quad = \quad (q_2, \varepsilon)$$

can be converted into productions as shown below:

For δ of the form $\delta(q_i, a, Z) = (q_j, \varepsilon)$	Resulting Productions $(q_iZq_j) \rightarrow a$
$\delta(q_0, a, A) = (q_3, \varepsilon)$	$(q_0Aq_3) \rightarrow a$
$\delta(q_0, b, A) = (q_1, \varepsilon)$	$(q_0Aq_1) \rightarrow b$
$\delta(q_1, \varepsilon, Z) = (q_2, \varepsilon)$	$(q_1Zq_2) \rightarrow \varepsilon$

Now, the transitions

$$\delta(q_0, a, Z) = (q_0, AZ)$$
$$\delta(q_3, \varepsilon, Z) = (q_0, AZ)$$

can be converted into productions using rule 4.a as shown below:

For δ of the form $\delta(q_i, a, Z) = (q_j, AB)$	Resulting Productions $(q_iZq_k) \rightarrow a\, (q_jAq_l)(q_lBq_k)$
$\delta(q_0, a, Z) = (q_0, AZ)$	$(q_0Zq_0) \rightarrow a\,(q_0Aq_0)(q_0Zq_0) \mid a\,(q_0Aq_1)(q_1Zq_0) \mid a\,(q_0Aq_2)(q_2Zq_0) \mid a\,(q_0Aq_3)(q_3Zq_0)$ $(q_0Zq_1) \rightarrow a\,(q_0Aq_0)(q_0Zq_1) \mid a\,(q_0Aq_1)(q_1Zq_1) \mid a\,(q_0Aq_2)(q_2Zq_1) \mid a\,(q_0Aq_3)(q_3Zq_1)$ $(q_0Zq_2) \rightarrow a\,(q_0Aq_0)(q_0Zq_2) \mid a\,(q_0Aq_1)(q_1Zq_2) \mid a\,(q_0Aq_2)(q_2Zq_2) \mid a\,(q_0Aq_3)(q_3Zq_2)$ $(q_0Zq_3) \rightarrow a\,(q_0Aq_0)(q_0Zq_3) \mid a\,(q_0Aq_1)(q_1Zq_3) \mid a\,(q_0Aq_2)(q_2Zq_3) \mid a\,(q_0Aq_3)(q_3Zq_3)$
$\delta(q_3, \varepsilon, Z) = (q_0, AZ)$	$(q_3Zq_0) \rightarrow (q_0Aq_0)(q_0Zq_0) \mid (q_0Aq_1)(q_1Zq_0) \mid (q_0Aq_2)(q_2Zq_0) \mid (q_0Aq_3)(q_3Zq_0)$ $(q_3Zq_1) \rightarrow (q_0Aq_0)(q_0Zq_1) \mid (q_0Aq_1)(q_1Zq_1) \mid (q_0Aq_2)(q_2Zq_1) \mid (q_0Aq_3)(q_3Zq_1)$ $(q_3Zq_2) \rightarrow (q_0Aq_0)(q_0Zq_2) \mid (q_0Aq_1)(q_1Zq_2) \mid (q_0Aq_2)(q_2Zq_2) \mid (q_0Aq_3)(q_3Zq_2)$ $(q_3Zq_3) \rightarrow (q_0Aq_0)(q_0Zq_3) \mid (q_0Aq_1)(q_1Zq_3) \mid (q_0Aq_2)(q_2Zq_3) \mid (q_0Aq_3)(q_3Zq_3)$

The start symbol of the grammar will be q_0Zq_2.

Example 5.23: Obtain a CFG that generates the language accepted by PDA M = ($\{q_0, q_1\}$, $\{a, b\}$, $\{A, Z\}$, δ, q_0, Z, $\{q_1\}$), with the transitions

$$\delta(q_0, a, Z) = (q_0, AZ)$$
$$\delta(q_0, b, A) = (q_0, AA)$$
$$\delta(q_0, a, A) = (q_1, \varepsilon)$$

Now, the transition

$$\delta(q_0, a, A) = (q_1, \varepsilon)$$

can be converted into production as shown below:

For δ of the form $\delta(q_i, a, Z) = (q_j, \varepsilon)$	Resulting Productions $(q_iZq_j) \rightarrow a$
$\delta(q_0, a, A) = (q_1, \varepsilon)$	$(q_0Aq_1) \rightarrow a$

can be converted into production as shown below:

For δ of the form $\delta(q_i, a, Z) = (q_j, \varepsilon)$	Resulting Productions $(q_iZq_j) \rightarrow a$
$\delta(q_0, a, A) = (q_1, \varepsilon)$	$(q_0Aq_1) \rightarrow a$

Now, the transitions

$$\delta(q_0, a, Z) = (q_0, AZ)$$
$$\delta(q_0, b, A) = (q_0, AA)$$

can be converted into productions using rule 4.a as shown below:

For δ of the form $\delta(q_i, a, Z) = (q_j, AB)$	Resulting Productions $(q_iZq_k) \rightarrow a\,(q_jAq_l)(q_lBq_k)$
$\delta(q_0, a, Z) = (q_0, AZ)$	$(q_0Zq_0) \rightarrow a\,(q_0Aq_0)(q_0Zq_0) \mid a\,(q_0Aq_1)(q_1Zq_0)$ $(q_0Zq_1) \rightarrow a\,(q_0Aq_0)(q_0Zq_1) \mid a\,(q_0Aq_1)(q_1Zq_1)$
$\delta(q_0, b, A) = (q_0, AA)$	$(q_0Aq_0) \rightarrow b(q_0Aq_0)(q_0Aq_0) \mid b(q_0Aq_1)(q_1Aq_0)$ $(q_0Aq_1) \rightarrow b(q_0Aq_0)(q_0Aq_1) \mid b(q_0Aq_1)(q_1Aq_1)$

The start symbol of the grammar will be q_0Zq_1.

Example:

Construct PDA to accept if-else of a C program and convert it to CFG. (This does not accept if –if –else-else statements).

Let the PDA P = ({q}, {i, e}, {X,Z}, δ, q, Z), where δ is given by:

$$\delta(q, i, Z) = \{(q, XZ)\},\ \ \delta(q, e, X) = \{(q, \varepsilon)\} \text{ and } \delta(q, \varepsilon, Z) = \{(q, \varepsilon)\}$$

Solution:

Equivalent productions are:

$S \rightarrow [qZq]$
$[qZq] \rightarrow i[qXq][qZq]$

$[qXq] \rightarrow e$
$[qZq] \rightarrow \varepsilon$

If [qZq] is renamed to A and [qXq] is renamed to B, then the CFG can be defined by:

$G = (\{S, A, B\}, \{i, e\}, \{S \rightarrow A,\ A \rightarrow iBA \mid \varepsilon,\ B \rightarrow e\}, S)$

Closure Properties of CFL

Many operations on Context Free Languages (CFL) guarantee to produce CFL. A few do not produce CFL. *Closure properties* consider operations on CFL that are guaranteed to produce a CFL. The CFL's are closed under *substitution*, *union*, *concatenation*, *closure (star)*, *reversal*, *homomorphism* and *inverse homomorphism*. CFL's are not closed under *intersection* (but the intersection of a CFL and a regular language is always a CFL), *complementation*, and *set-difference*.

Theorem 8.3

The family of context-free languages is closed under union, concatenation, and star-closure.

Proof: Let L_1 and L_2 be two context-free languages generated by the context-free grammars $G_1 = (V_1, T_1, S_1, P_1)$ and $G_2 = (V_2, T_2, S_2, P_2)$, respectively. We can assume without loss of generality that the sets V_1 and V_2 are disjoint.

Consider now the language L (G3), generated by the grammar

$$G_3 = (V_1 \cup V_2 \cup \{S_3\}, T_1 \cup T_2, S_3, P_3),$$

where S_3 is a variable not in $V1 \cup V2$. The productions of G_3 are all the productions of G_1 and G_2, together with an alternative starting production that allows us to use one or the other grammars. More precisely

$$P_3 = P_1 \cup P_2 \cup \{S_3 \rightarrow S_1 | S_2\}.$$

Example

Let $L_1 = \{ a^n b^n , n > 0\}$. Corresponding grammar G_1 will have P: $S1 \rightarrow aAb|ab$

Let $L_2 = \{ c^m d^m , m \geq 0\}$. Corresponding grammar G_2 will have P: $S2 \rightarrow cBb| \varepsilon$

Union of L_1 and L_2, $L = L_1 \cup L_2 = \{ a^n b^n \} \cup \{ c^m d^m \}$

The corresponding grammar G will have the additional production $S \rightarrow S1 \mid S2$

b. *Closure under concatenation of CFL's L_1 and L_2:*

Let L={ab}, s(a)=L_1 and s(b)=L_2. Then s(L)=L_1L_2

How to get grammar for L_1L_2?

Add new start symbol and rule $S \rightarrow S_1S_2$

The grammar for L_1L_2 is G = (V, T, P, S) where $V = V_1 \cup V_2 \cup \{S\}$, $S \notin V_1 \cup V_2$ and $P = P_1 \cup P_2 \cup \{S \rightarrow S_1S_2\}$

Example:

$L_1 = \{a^nb^n \mid n \geq 0\}$, $L_2 = \{b^na^n \mid n \geq 0\}$ then $L_1L_2 = \{a^nb^{\{n+m\}}a^m \mid n, m \geq 0\}$

Their corresponding grammars are

G_1: $S_1 \rightarrow aS_1b \mid \varepsilon$, G_2 : $S_2 \rightarrow bS_2a \mid \varepsilon$

The grammar for L_1L_2 is

$G = (\{S, S_1, S_2\}, \{a, b\}, \{S \rightarrow S_1S_2, S_1 \rightarrow aS_1b \mid \varepsilon, S_2 \rightarrow bS_2a\}, S)$.

Example: L1 = $\{a^nb^n \mid n >= 0\}$ L2 = $\{c^md^m \mid n >= 0\}$ then L3 = L1.L2 = $\{a^nb^nc^md^m \mid m,n >= 0\}$ is CFL.

c. *Closure under Kleene's star (closure * and positive closure $^+$) of CFL's L_1:*

Let $L = \{a\}^*$ (or $L = \{a\}^+$) and $s(a) = L_1$. Then $s(L) = L_1^*$ (or $s(L) = L_1^+$).

Example:

$L_1 = \{a^nb^n \mid n \geq 0\}$ $(L_1)^* = \{a^{\{n1\}}b^{\{n1\}} \ldots a^{\{nk\}}b^{\{nk\}} \mid k \geq 0$ and $ni \geq 0$ for all $i\}$

$L_2 = \{a^{\{n^2\}} \mid n \geq 1\}$, $(L_2)^* = a^*$

How to get grammar for $(L_1)^*$:

Add new start symbol S and rules $S \rightarrow SS_1 \mid \varepsilon$.

The grammar for $(L_1)^*$ is

$G = (V, T, P, S)$, where $V = V_1 \cup \{S\}$, $S \notin V_1$, $P= P_1 \cup \{S \rightarrow SS_1 \mid \varepsilon\}$

The family of context-free languages is not closed under intersection and complementation.

Proof: Consider the two languages

$$L_1 = \{a^n b^n c^m : n \geq 0, m \geq 0\}$$

or

and

$$L_2 = \{a^n b^m c^m : n \geq 0, m \geq 0\}.$$

There are several ways one can show that L_1 and L_2 are context-free. For instance, a grammar for L_1 is

$$\begin{aligned} S &\rightarrow S_1 S_2, \\ S_1 &\rightarrow a S_1 b | \lambda, \\ S_2 &\rightarrow c S_2 | \lambda. \end{aligned}$$

Alternatively, we note that L_1 is the concatenation of two context-free languages, so it is context-free by Theorem 8.3. But

$$L_1 \cap L_2 = \{a^n b^n c^n : n \geq 0\},$$

which we have already shown not to be context-free. Thus, the family of context-free languages is not closed under intersection.

The second part of the theorem follows from Theorem 8.3 and the set identity

$$L_1 \cap L_2 = \overline{\overline{L}_1 \cup \overline{L}_2}.$$

If the family of context-free languages were closed under complementation, then the right side of the above expression would be a context-free language for any context-free L_1 and L_2. But this contradicts what we have just shown, that the intersection of two context-free languages is not necessarily context-free. Consequently, the family of context-free languages is not closed under complementation.■

6.3 Closure Properties of CFL

In chapter 3, we have seen that regular languages are closed under union, concatenation and kleen closure. Now we will discuss closure properties of context free languages. In the sense, we will, check which are those properties for them CFLs are closed under. The context free languages are closed under some operation means after performing that particular operation on those CFLs the resultant language is context free language. These properties are as below.

1. The context free languages are closed under union.
2. The context free languages are closed under concatenation.
3. The context free languages are closed under kleen closure.
4. The context free languages are not closed under intersection.
5. The context free languages are not closed under complement.

We will discuss the above mentioned closure properties of CFL with the help of proofs and examples.

Theorem 1 : If L_1 and L_2 are context free languages then $L = L_1 \cup L_2$ is also context free. That is, the CFLs are closed under union.

Proof : We will consider two languages L_1 and L_2 which are context free languages. We can give these languages using context free grammars G_1 and G_2 such that $G_1 \in L_1$ and $G_2 \in L_2$. The G_1 can be given as $G_1 = \{V_1, \Sigma, P_1, S_1\}$ where P_1 can be given as

$P_1 = \{$

$S_1 \rightarrow A_1 S_1 A_1 \mid B_1 S B_1 \mid \varepsilon$

$A_1 \rightarrow a$

$B_1 \rightarrow b$

$\}$

Here $V_1 = \{S_1, A_1, B_1\}$ and S_1 is a start symbol.

Similarly, we can write $G_2 = \{V_2, \Sigma, P_2, S_2\}$

$N_2 = \{S_2, A_2, B_2\}$ and S_2 is a start symbol.

P_2 can be given as :

$P_2 = \{$

$S_2 \rightarrow a A_2 A_2 \mid b B_2 B_2$

$A_2 \rightarrow b$

$B_2 \rightarrow a$

$\}$

Now $L = L_1 \cup L_2$ gives $G \in L$. This G can be written as

$G = \{V, \Sigma, P, S\}$

$V = \{S_1, A_1, B_1, S_2, A_2, B_2\}$

$P = \{P_1 \cup P_2\}$

S is a start symbol.

$P = \{ \quad S \rightarrow S_1 \mid S_2$

$S_1 \rightarrow A_1 S_1 A_1 \mid B_1 S B_1 \mid \varepsilon$

$A_1 \rightarrow a$

$B_1 \rightarrow b$

$S_2 \rightarrow aA_2A_2 \mid bB_2B_2$

$A_2 \rightarrow b$

$B_2 \rightarrow a$

$\}$

Thus grammar G is a context free grammar which produces languages L which is context free language.

Theorem 2 : If L_1 and L_2 are two context free languages then L_1L_2 is CFG. That means context free languages are closed under concatenation.

Proof : Let L_1 is a context free language which can be represented by a context free grammar G_1, such that $G_1 \in L_1$ and

$G_1 = \{V_1, \Sigma, P_1, S_1\}$

$V_1 = \{S_1, A_1, B_1\}$

$\Sigma = \{a, b\}$

S_1 is a start symbol and P_1 is a set of production rules.

$P_1 = \{ \quad S_1 \rightarrow A_1 S_1 A_1 \mid B_1 S_1 B_1 \mid \varepsilon$

$A_1 \rightarrow a$

$B_1 \rightarrow b$

$\}$

Similarly, L_2 is a context free language which can be represented by a context free grammar G_2, such that $G_2 \in L_2$ and

$$G_2 = \{V_2, \Sigma, P_2, S_2\}$$
$$V_2 = \{S_2, A_2, B_2\}$$
$$\Sigma = \{a, b\}$$

S_2 is a start symbol and P_2 is a set of production rules.

$P_2 = \{$ $\quad S_2 \to aA_2A_2 \mid bB_2B_2$

$A_2 \to b$

$B_2 \to a$

$\}$

Now $L = L_1 L_2$ can be obtained by G such that $G = G_1 \cdot G_2$. Therefore

$$G = \{V, \Sigma, P, S\}$$
$$V = \{S, S_1 \cdot A_1, B_1, S_2, A_2, B_2\}$$

where S is a start symbol. The production rules, P can be given as,

$P = \{$ $\quad S \to S_1 \mid S_2$

$S_1 \to A_1\ S_1\ A_1 \mid B_1\ S_1\ B_1 \mid \varepsilon$

$A_1 \to a$

$B_1 \to b$

$S_2 \to aA_2A_2 \mid bB_2B_2$

$A_2 \to b$

$A_2 \to b$

$B_2 \to a$

$\}$

As grammar G is context free grammar the language L produced by G is also context free language. Hence context free languages are closed under concatenation.

Theorem 3 : If L_1 is context free language then L_1^* is also context free. That means CFL is closed under kleen closure.

Proof : Let, L_1 be a context free language represented by G_1 such that $G_1 \to \varepsilon\ L_1$.

The CFG G_1 can be given as,

$G_1 = \{V_1, \Sigma, P_1, S_1\}$ where S_1 is a start symbol.

$P_1 = \{$ $\quad S_1 \to A_1S_1A_1 \mid B_1S_1B_1 \mid \varepsilon$

$A_1 \to a$

$B_1 \to b$

$\}$

Now $L = L_1^*$ can be represented by a grammar G such that

$G = \{$ $\quad (V, \Sigma, P, S)$

$V = \{S, S_1, A_1\ B_1\}$

and $\quad P = S \to S_1\ S \mid \varepsilon$

$S_1 \to A_1\ S_1\ A_1 \mid B_1\ S_1\ B_1$

$A_1 \to a$

$B_1 \to b$

$\}$

Thus grammar G is a context free grammar and language L produced by G is also context free language. Hence context free language are closed under kleen closure.

Theorem 4 : If L_1 and L_2 are two CFLs then $L = L_1 \cap L_2$ may be CFL or may not be CFL. That means L is not closed under intersection.

Proof : Let, $L_1 = \{0^n 1^n 2^i \mid n \geq 1 , i \geq 1\}$

$L_2 = \{0^n 1^n 2^n \mid n \geq , i \geq 1\}$

The grammar for L_1 is

$S \rightarrow AB$

$A \rightarrow 0A1 \mid 01$

$B \rightarrow 2B \mid 2$

Similarly L_2 can be represented by grammar.

$S \rightarrow AB$

$A \rightarrow 0A \mid 0$

$B \rightarrow 1B2 \mid 12$

Now if we try to obtain

$L = L_1 \cap L_2$ then we get sometimes context free languages and sometimes non context free languages. Thus we can say that CFLs are not closed under intersection.

Theorem 5 : If L_1 is a CFL then L_1' may or may not be CFL. That means CFL is not closed under complement.

Proof : Let L_1 and L_2 are two CFLs. We will assume that complement of a context free language is a CFL itself. Hence L_1' and L_2' both are CFLs. We can also state that $(L_1' \cup L_2')$ is context free (since CFLs are closed under union). But $(L_1' \cup L_2') = L_1 \cap L_2$ i.e. $L = L_1 \cap L_2$ may or may not be CFL. The L_1 and L_2 are arbitrary CFLs, there may exist L_1' and L_2' which are not CFL. Hence complement of certain language may be context free or may not be. Therefore we can say that CFL is not closed under complement operation.

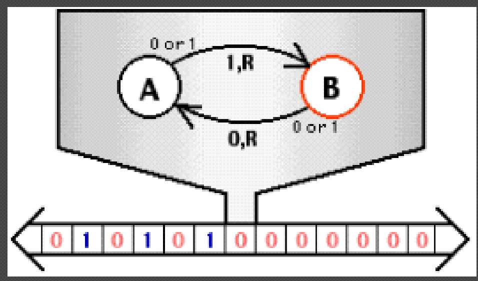
UNIT - V

30 June 2017 07:14 PM

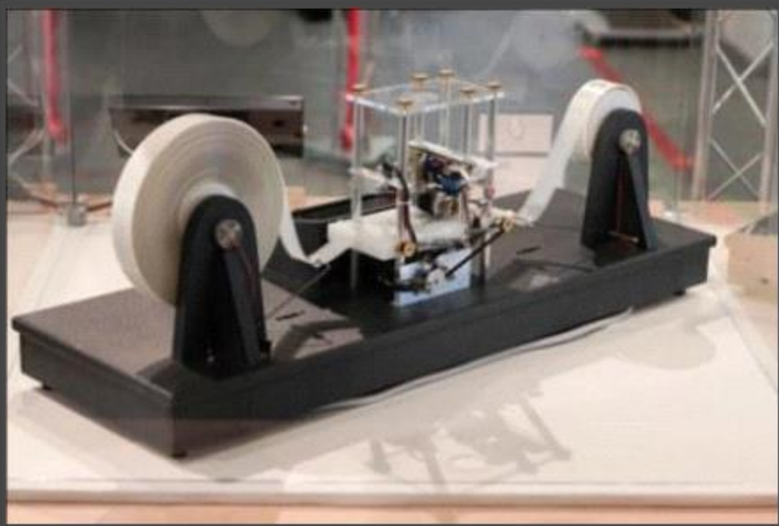
Turing Machines

16 July 2018 09:56 PM

Turing Machine: Acceptor & Transducer



We have studied two types of languages from the Chomsky hierarchy: regular languages and context-free languages. These languages can describe many practically important systems and so they are heavily used in practice. They are, however, of limited capability and there are many languages that they can not process. Here we are going to study the most general of the languages in Chomsky hierarchy, the phrase structure languages (also called Type 0 languages), and the machines that can process them: Turing machines.



Turing machines were invented of by the English mathematician Alan Turing as a model of human "computation". Later Alonzo Church says that any computation done by humans or computers can be carried out by some Turing machine. This statement is known as Church's thesis and today it is generally accepted as true. Computers we use today are as powerful as Turing machines except that computers have finite memory while Turing machines have infinite memory.

Turing Machines can be represented using a 7-tuple:

$$T = (Q, \Sigma, \Gamma, \delta, q_0, \beta, F)$$

Where :

Q = Set of states

Σ = Input alphabet

Γ = Tape alphabet ($\Sigma \subseteq \Gamma$)

δ = Transition function

$\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$

q_0 = Start state

Turing Machines can be represented using a 7-tuple:

$$T = (Q, \Sigma, \Gamma, \delta, q_0, \beta, F)$$

Where :
Q = Set of states
Σ = Input alphabet
Γ = Tape alphabet ($\Sigma \subseteq \Gamma$)
δ = Transition function
$\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$
q_0 = Start state
β = Blank symbol
F = Set of accepting states ($F \subseteq Q$)

TM as ACCEPTOR

A Turing machine *halts* when it no longer has any available moves. If it halts in a final state, it accepts its input; otherwise, it rejects its input.

Turing machine accepts its input if it halts in a final state. There are two ways of rejecting the input string in case of TM:

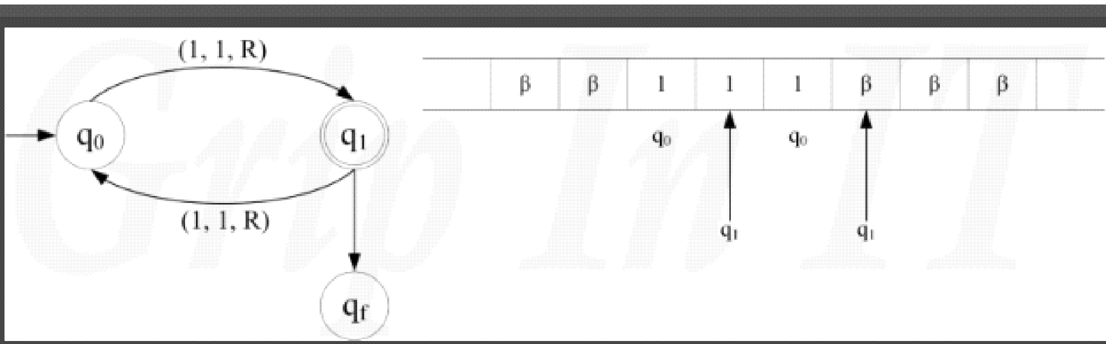
1. The Turing machine could halt in a nonfinal state, or
2. The Turing machine could never stop i.e hang (in which case we say it is in an *infinite loop.*)

L = $\{1^m \mid m \text{ is odd}\}$

The Transition $\delta(q_0, 1) = \delta(q_1, 1, R)$ means Initially q_0 will read the input 1 from Tape, move to state q_1,1 is replaced by 1 and read/write head moves to right.

$\delta(q_0, 1) = \delta(q_1, 1, R)$
$\delta(q_1, 1) = \delta(q_0, 1, R)$
$\delta(q_1, \beta) = \delta(qf, \beta, L/R)$

If Blank is reached in q_1 state then input string contain odd no. of 1's. because on reading odd 1's machine moves to q_1 state.

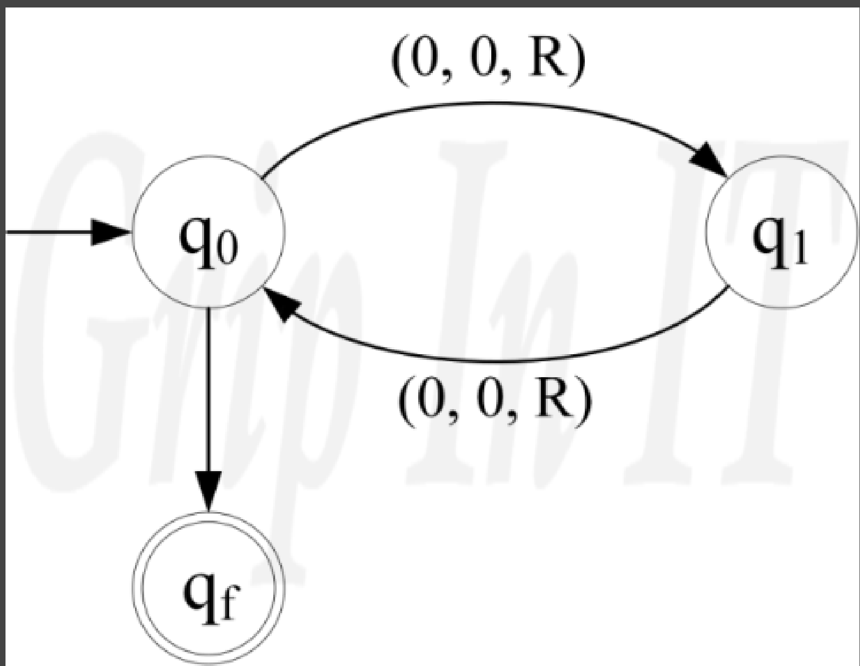


L = {0^m | m is even}

$\delta(q_0, 0) = \delta(q_1, 0, R)$

$\delta(q_1, 0) = \delta(q_0, 0, R)$

$\delta(q_0, \beta) = \delta(q_f, \beta, L/R)$



$L = \{a^n b^n \mid n > 0\}$

There is no concept of minimal TM, change state only if needed. Do not think about ∈ in TM.

Strings Accepted by language = {ab, aabb, aaabbb, ...}

$\delta(q_0, a) = (q_1, x, R)$

$\delta(q_1, a) = (q_1, a, R)$

$\delta(q_1, b) = (q_2, y, L)$

$\delta(q_2, a) = (q_2, a, L)$

$\delta(q_2, x) = (q_0, x, R)$

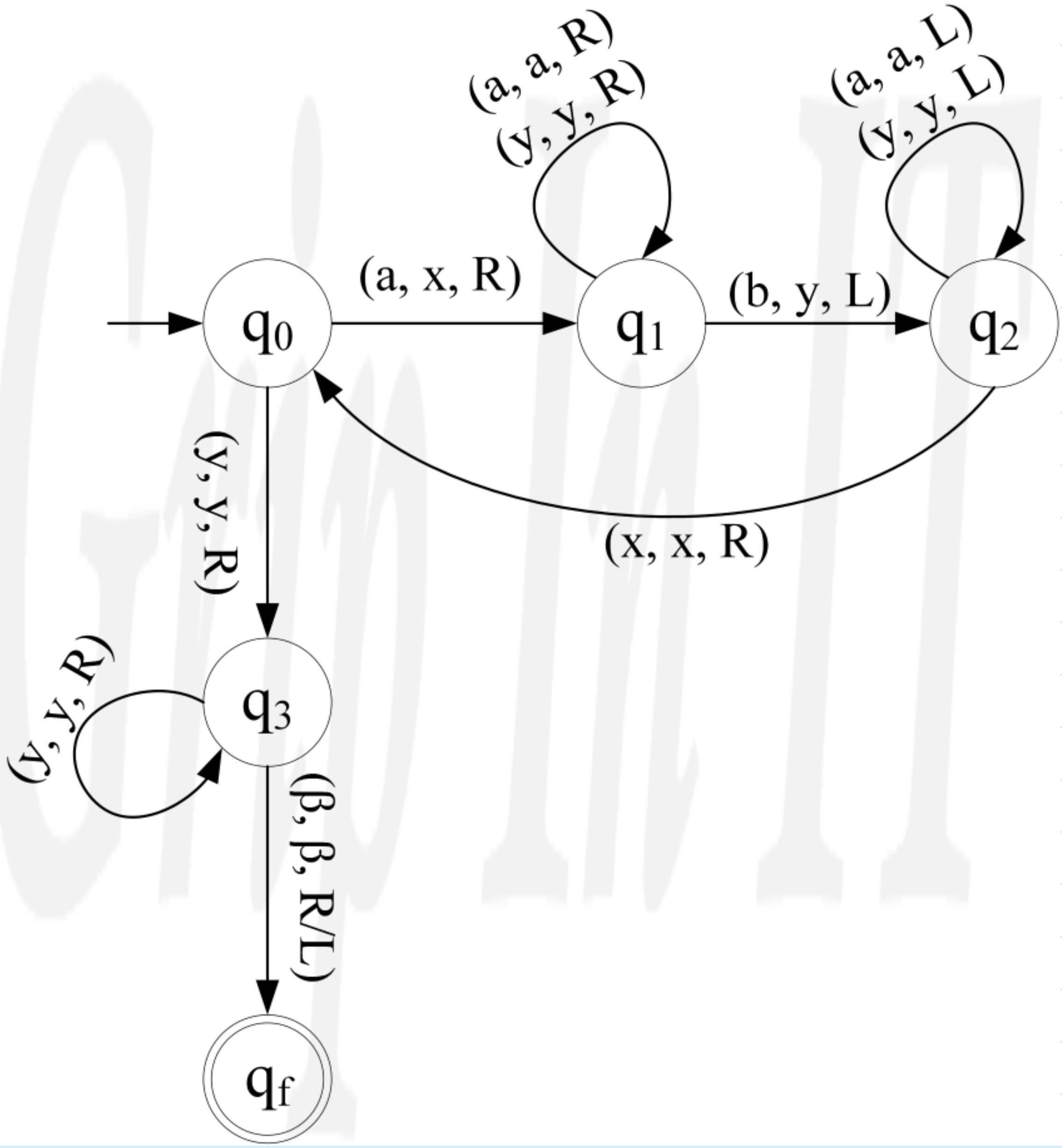
$\delta(q_1, y) = (q_1, y, R)$

$\delta(q_2, y) = (q_2, y, L)$

$\delta(q_0, y) = (q_3, y, R)$

$\delta(q_3, y) = (q_3, y, R)$

$\delta(q_3, \beta) = (q_f, \beta, R/L)$



TM as TRANSDUCER

A Turing machine can be used as a transducer. The most obvious way to do this is to treat the entire non-blank portion of the initial tape as input, and to treat the entire non-blank portion of the tape when the machine halts as output.

Qus 1: Design TM to calculate m – n, where m, n are positive integer and m > n.

f(m,n) = m – n

Here suppose m is 5 and n is 3 then we can represent m by 5 o's and n by 3 o's. Both m and n are separated by 1.

5 – 3 = 2

<u>000001000</u> = <u>00</u>

The transition's for this TM is as below –

$\delta(q_0, 0) = (q_1, \beta, R)$

$\delta(q_1, 0) = (q_1, 0, R)$

$\delta(q_1, 1) = (q_2, 1, R)$

$\delta(q_2, 0) = (q_2, 0, R)$

$\delta(q_2, \beta) = (q_3, \beta, L)$

$\delta(q_3, 0) = (q_4, \beta, L)$ // when all 0's are over at q_3, it receive 1 at end.

$\delta(q_4, 0) = (q_4, 0, L)$

$\delta(q_4, 1) = (q_5, 1, L)$

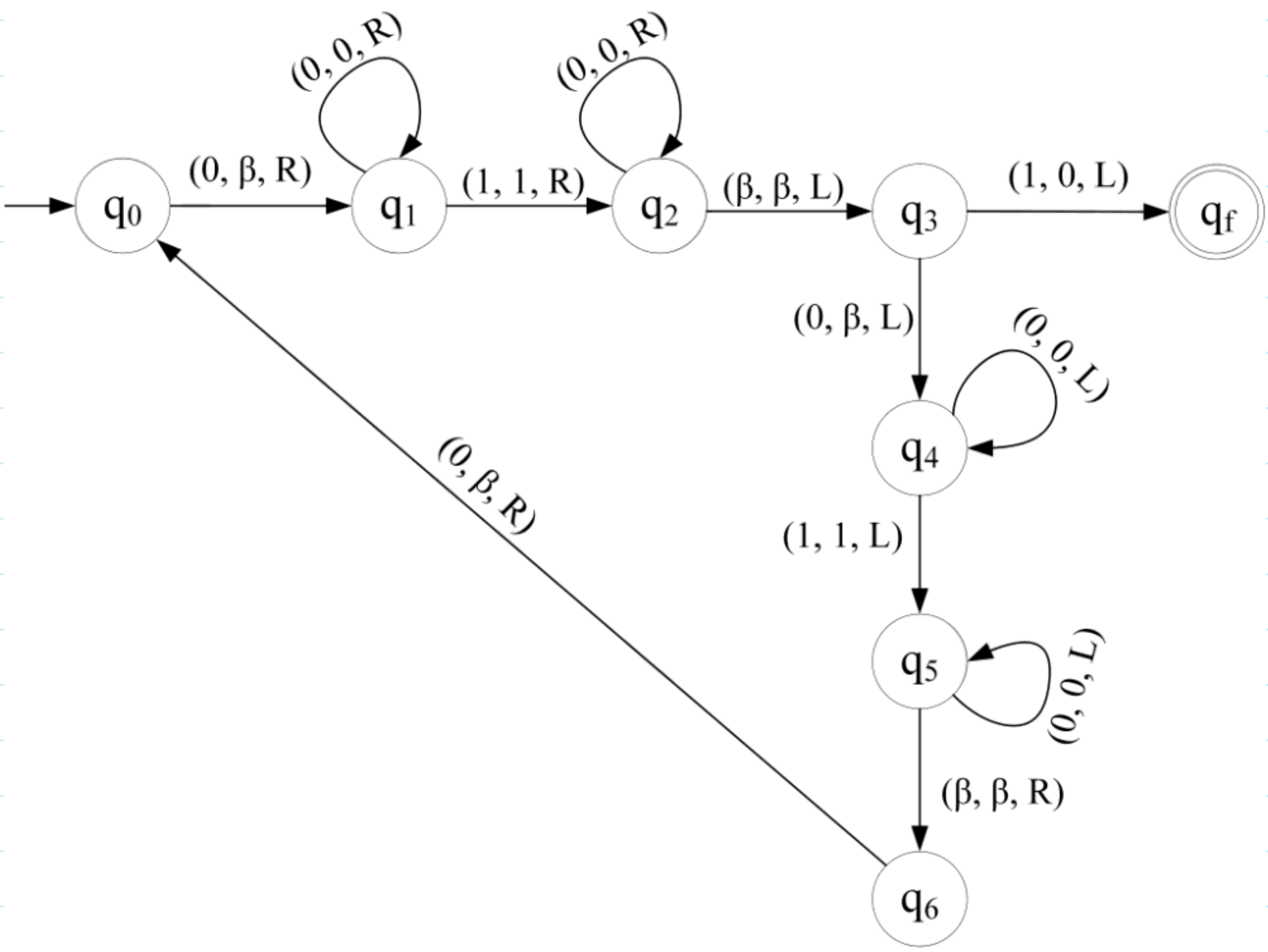
$\delta(q_5, 0) = (q_5, 0, L)$

$\delta(q_5, \beta) = (q_6, \beta, R)$

$\delta(q_6, 0) = (q_1, \beta, R)$ // See Note 1 below

$\delta(q_3, 1) = (q_f, 0, L)$

Note 1: Beginning of next cycle for cancellation of 0 by 0. This process continues till all 0's in n are cancelled by an equal no of 0's in m. In the next cycle a 0 is cancelled in m but n has no more 0's so read/write head receives the separator 1 in q_3 state and replace it with 0 and enters in final state q_f.



Qus 2: Design TM to calculate m + n, where m, n are positive integer

f(m,n) = m + n

Here suppose m is 3 and n is 4 then we can represent m by 3 1's and n by 4 1's. Both m and n are separated by 0.

3 + 4 = 7

1110 1111 = 1111111

The transition's for this TM is as below –

Logic: When 0 is encountered, it is changed to 1 and last 1 is replaced by β.

$\delta(q_0, 1) = (q_0, 1, R)$

$\delta(q_0, 1) = (q_1, 1, R)$

$\delta(q_1, 1) = (q_1, 1, R)$

$\delta(q_1, \beta) = (q_2, \beta, L)$

$\delta(q_2, 1) = (q_f, \beta, L)$

Turing Machines Introduction

10 October 2018 05:53 AM

In the early 1930s, mathematicians were trying to define effective computation. Alan Turing in 1936, Alanzo Church in 1933, S.C. Kleene in 1935, Schonfinkel in 1965 gave various models using the concept of Turing machines, λ-calculus, combinatory logic, post-systems and μ-recursive functions. It is interesting to note that these were formulated much before the electro-mechanical/electronic computers were devised. Although these formalisms, describing effective computations, are dissimilar, they turn to be equivalent.

Among these formalisms, the Turing's formulation is accepted as a model of algorithm or computation. The Church–Turing thesis states that any algorithmic procedure that can be carried out by human beings/computer can be carried out by a Turing machine. It has been universally accepted by computer scientists that the Turing machine provides an ideal theoretical model of a computer.

Turing machines are useful in several ways. As an automaton, the Turing machine is the most general model. It accepts type-0 languages. It can also be used for computing functions. It turns out to be a mathematical model of partial recursive functions. Turing machines are also used for determining the undecidability of certain languages and measuring the space and time complexity of problems. These are the topics of discussion in this chapter and some of the subsequent chapters.

For formalizing computability, Turing assumed that, while computing, a person writes symbols on a one-dimensional paper (instead of a two-dimensional paper as is usually done) which can be viewed as a tape divided into cells.

One scans the cells one at a time and usually performs one of the three simple operations, namely (i) writing a new symbol in the cell being currently scanned, (ii) moving to the cell left of the present cell, and (iii) moving to the cell right of the present cell. With these observations in mind, Turing proposed his 'computing machine.'

Turing Machine Model

10 October 2018 05:57 AM

9.1 TURING MACHINE MODEL

The Turing machine can be thought of as finite control connected to a R/W (read/write) head. It has one tape which is divided into a number of cells. The block diagram of the basic model for the Turing machine is given in Fig. 9.1.

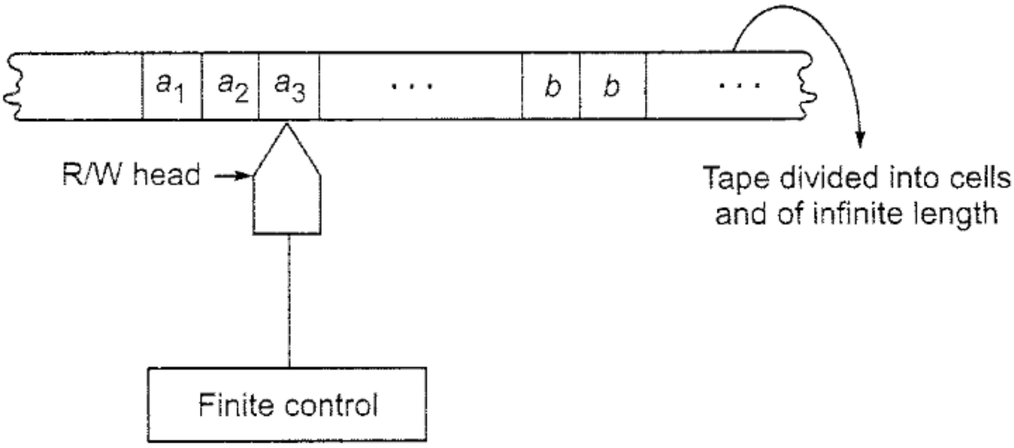


Fig. 9.1 Turing machine model.

Each cell can store only one symbol. The input to and the output from the finite state automaton are effected by the R/W head which can examine one cell at a time. In one move, the machine examines the present symbol under the R/W head on the tape and the present state of an automaton to determine

(i) a new symbol to be written on the tape in the cell under the R/W head,
(ii) a motion of the R/W head along the tape: either the head moves one cell left (L), or one cell right (R),
(iii) the next state of the automaton, and
(iv) whether to halt or not.

The above model can be rigorously defined as follows:

Definition 9.1 A Turing machine M is a 7-tuple, namely $(Q, \Sigma, \Gamma, \delta, q_0, b, F)$, where

1. Q is a finite nonempty set of states,
2. Γ is a finite nonempty set of tape symbols,
3. $b \in \Gamma$ is the blank,
4. Σ is a nonempty set of input symbols and is a subset of Γ and $b \notin \Sigma$,
5. δ is the transition function mapping (q, x) onto (q', y, D) where D denotes the direction of movement of R/W head; $D = L$ or R according as the movement is to the left or right.
6. $q_0 \in Q$ is the initial state, and
7. $F \subseteq Q$ is the set of final states.

Notes: (1) The acceptability of a string is decided by the reachability from the initial state to some final state. So the final states are also called the accepting states.

(2) δ may not be defined for some elements of $Q \times \Gamma$.

9.2.1 REPRESENTATION BY INSTANTANEOUS DESCRIPTIONS

'Snapshots' of a Turing machine in action can be used to describe a Turing machine. These give 'instantaneous descriptions' of a Turing machine. We have defined instantaneous descriptions of a pda in terms of the current state, the input string to be processed, and the topmost symbol of the pushdown store. But the input string to be processed is not sufficient to be defined as the ID of a Turing machine, for the R/W head can move to the left as well. So an ID of a Turing machine is defined in terms of the entire input string and the current state.

Definition 9.2 An ID of a Turing machine M is a string $\alpha\beta\gamma$, where β is the present state of M, the entire input string is split as $\alpha\gamma$, the first symbol of γ is the current symbol a under the R/W head and γ has all the subsequent symbols of the input string, and the string α is the substring of the input string formed by all the symbols to the left of a.

EXAMPLE 9.1

A snapshot of Turing machine is shown in Fig. 9.2. Obtain the instantaneous description.

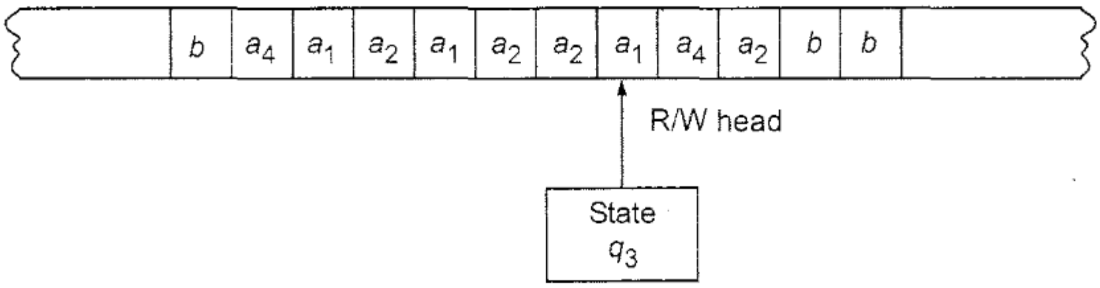
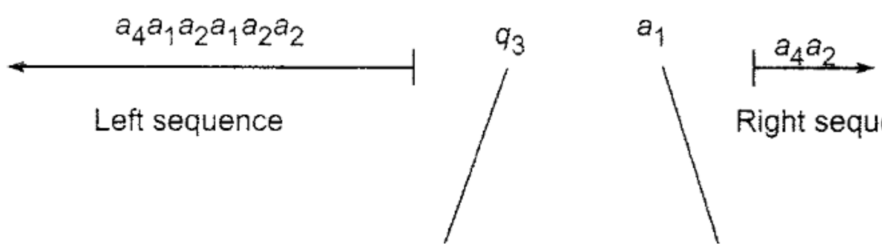


Fig. 9.2 A snapshot of Turing machine.

Solution

The present symbol under the R/W head is a_1. The present state is q_3. So a_1 is written to the right of q_3. The nonblank symbols to the left of a_1 form the string $a_4a_1a_2a_1a_2a_2$, which is written to the left of q_3. The sequence of nonblank symbols to the right of a_1 is a_4a_2. Thus the ID is as given in Fig. 9.3.



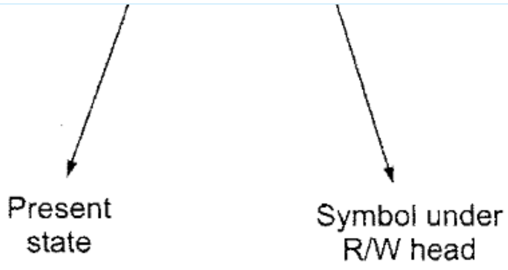


Fig. 9.3 Representation of ID.

Notes: (1) For constructing the ID, we simply insert the current state in the input string to the left of the symbol under the R/W head.

(2) We observe that the blank symbol may occur as part of the left or right substring.

Moves in a TM

As in the case of pushdown automata, $\delta(q, x)$ induces a change in ID of the Turing machine. We call this change in ID a move.

Suppose $\delta(q, x_i) = (p, y, L)$. The input string to be processed is $x_1x_2 \ldots x_n$, and the present symbol under the R/W head is x_i. So the ID before processing x_i is

$$x_1x_2 \ldots x_{i-1}qx_i \ldots x_n$$

After processing x_i, the resulting ID is

$$x_1 \ldots x_{i-2}\, px_{i-1}yx_{i+1} \ldots x_n$$

This change of ID is represented by

$$x_1x_2 \ldots x_{i-1}\, q\, x_i \ldots x_n \vdash x_i \ldots x_{i-2}\, px_{i-1}\, yx_{i+1} \ldots x_n$$

If $i = 1$, the resulting ID is $p\, y\, x_2x_3 \ldots x_n$.

If $\delta(q, x_i) = (p, y, R)$, then the change of ID is represented by

$$x_1x_2 \ldots x_{i-1}q\, x_i \ldots x_n \vdash x_1x_2 \ldots x_{i-1}\, y\, px_{i+1} \ldots x_n$$

If $i = n$, the resulting ID is $x_1x_2 \ldots x_{n-1}\, y\, p\, b$.

We can denote an ID by I_j for some j. $I_j \vdash I_k$ defines a relation among IDs. So the symbol $\overset{*}{\vdash}$ denotes the reflexive–transitive closure of the relation $\vdash$. In particular, $I_j \overset{*}{\vdash} I_j$. Also, if $I_1 \overset{*}{\vdash} I_n$, then we can split this as $I_1 \vdash I_2 \vdash I_3 \vdash \ldots \vdash I_n$ for some IDs, $I_2, \ldots, I_{n-1}$.

Note: The description of moves by IDs is very much useful to represent the processing of input strings.

9.2.2 Representation by Transition Table

We give the definition of δ in the form of a table called the transition table. If $\delta(q, a) = (\gamma, \alpha, \beta)$, we write $\alpha\beta\gamma$ under the α-column and in the q-row. So if

$\delta(q, a) = (\gamma, \alpha, \beta)$, we write $\alpha\beta\gamma$ under the α-column and in the q-row. So if we get $\alpha\beta\gamma$ in the table, it means that α is written in the current cell, β gives the movement of the head (L or R) and γ denotes the new state into which the Turing machine enters.

Consider, for example, a Turing machine with five states $q_1, \ldots, q_5$, where q_1 is the initial state and q_5 is the (only) final state. The tape symbols are 0, 1 and b. The transition table given in Table 9.1 describes δ.

TABLE 9.1 Transition Table of a Turing Machine

Present state	*Tape symbol*		
	b	0	1
$\rightarrow q_1$	$1Lq_2$	$0Rq_1$	
q_2	bRq_3	$0Lq_2$	$1Lq_2$
q_3		bRq_4	bRq_5
q_4	$0Rq_5$	$0Rq_4$	$1Rq_4$
q_5	$0Lq_2$		

As in Chapter 3, the initial state is marked with $\rightarrow$ and the final state with ○.

EXAMPLE 9.2

Consider the TM description given in Table 9.1. Draw the computation sequence of the input string 00.

Solution

We describe the computation sequence in terms of the contents of the tape and the current state. If the string in the tape is $a_1a_2 \ldots a_j a_{j+1} \ldots a_m$ and the TM in state q is to read a_{j+1}, then we write $a_1a_2 \ldots a_j\, q\, a_{j+1} \ldots a_m$.

For the input string $00b$, we get the following sequence:

$$q_100b \vdash 0q_10b \vdash 00q_1b \vdash 0q_201 \vdash q_2001$$
$$\vdash q_2b001 \vdash bq_3001 \vdash bbq_401 \vdash bb_0q41 \vdash bb_01q_4b$$
$$\vdash bb010q_5 \vdash bb01q_200 \vdash bb0q_2100 \vdash bbq_20100$$
$$\vdash bq_2b0100 \vdash bbq_30100 \vdash bbbq_4100 \vdash bbb_1q_400$$
$$\vdash bbb10q_40 \vdash bbb100q_4b \vdash bbb1000q_5b$$
$$\vdash bbb100q_200 \vdash bbb10q_2000 \vdash bbb1q_20000$$
$$\vdash bbbq_210000 \vdash bbq_2b10000 \vdash bbbq_310000 \vdash bbbbq_40000$$

$\vdash bbb100q_200 \vdash bbb10q_2000 \vdash bbb1q_20000$

$\vdash bbbq_210000 \vdash bbq_2b10000 \vdash bbbq_310000 \vdash bbbbq_50000$

9.2.3 Representation by Transition Diagram

We can use the transition systems introduced in Chapter 3 to represent Turing machines. The states are represented by vertices. Directed edges are used to

represent transition of states. The labels are triples of the form (α, β, γ), where $\alpha, \beta \in \Gamma$ and $\gamma \in \{L, R\}$. When there is a directed edge from q_i to q_j with label (α, β, γ), it means that

$$\delta(q_i, \alpha) = (q_j, \beta, \gamma)$$

During the processing of an input string, suppose the Turing machine enters q_i and the R/W head scans the (present) symbol α. As a result, the symbol β is written in the cell under the R/W head. The R/W head moves to the left or to the right, depending on γ, and the new state is q_j.

Every edge in the transition system can be represented by a 5-tuple $(q_i, \alpha, \beta, \gamma, q_j)$. So each Turing machine can be described by the sequence of 5-tuples representing all the directed edges. The initial state is indicated by $\rightarrow$ and any final state is marked with ○.

EXAMPLE 9.3

M is a Turing machine represented by the transition system in Fig. 9.4. Obtain the computation sequence of M for processing the input string 0011.

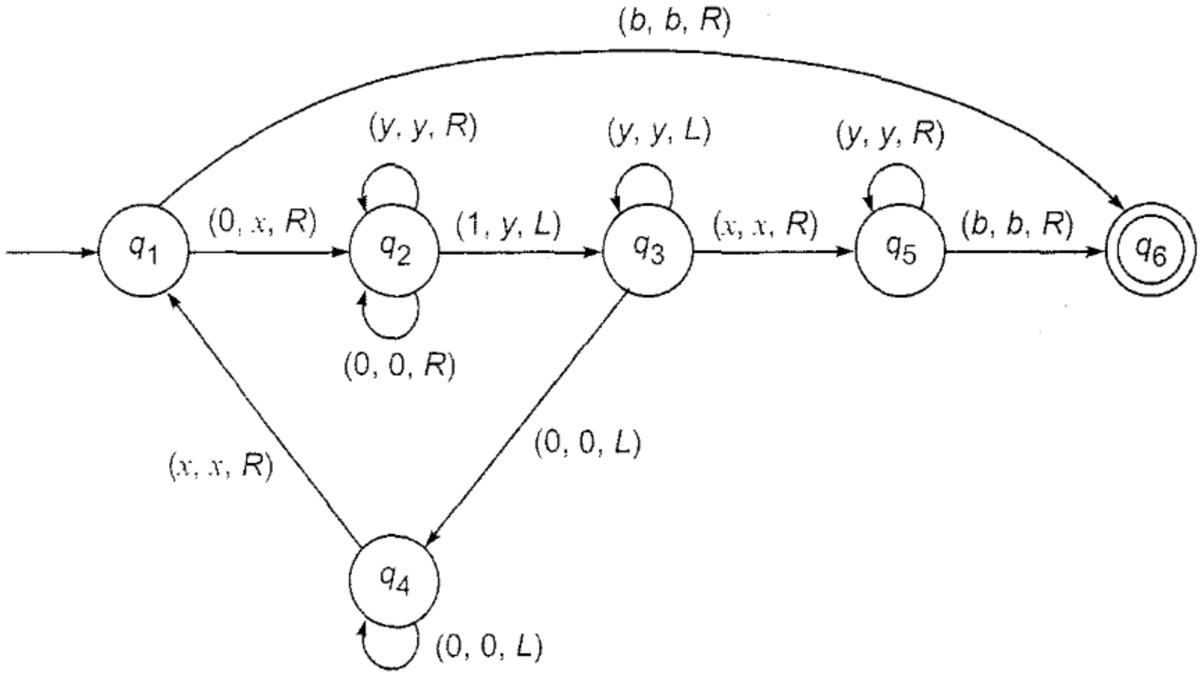


Fig. 9.4 Transition system for M.

Solution

Fig. 9.4 Transition system for *M*.

Solution

The initial tape input is $b0011b$. Let us assume that M is in state q_1 and the R/W head scans 0 (the first 0). We can represent this as in Fig. 9.5. The figure can be represented by

$$\begin{array}{c}\ \downarrow \\ b0011b \\ q_1\end{array}$$

From Fig. 9.4 we see that there is a directed edge from q_1 to q_2 with the label $(0, x, R)$. So the current symbol 0 is replaced by x and the head moves right. The new state is q_2. Thus, we get

$$\begin{array}{c}\ \ \downarrow \\ bx011b \\ q_2\end{array}$$

The change brought about by processing the symbol 0 can be represented as

$$\underset{q_1}{b\overset{\downarrow}{0}011b} \xrightarrow{(0,x,R)} \underset{q_2}{bx\overset{\downarrow}{0}11b}$$

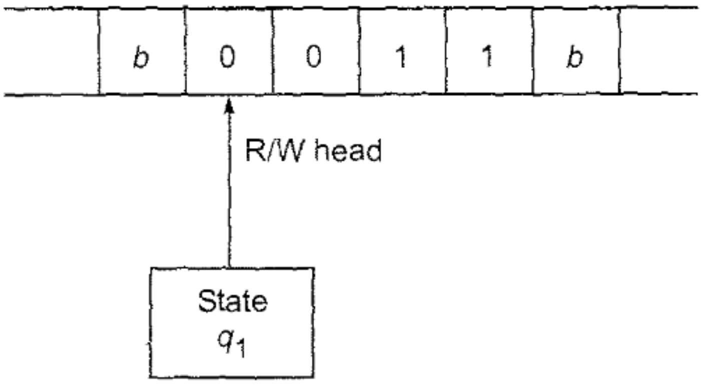


Fig. 9.5 TM processing 0011.

The entire computation sequence reads as follows:

$$\underset{q_1}{b\overset{\downarrow}{0}011b} \xrightarrow{(0,x,R)} \underset{q_2}{bx\overset{\downarrow}{0}11b} \xrightarrow{(0,0,R)} \underset{q_2}{bx0\overset{\downarrow}{1}1b}$$

$$\xrightarrow{(1,y,L)} \underset{q_3}{bx\overset{\downarrow}{0}y1b} \xrightarrow{(0,0,L)} \underset{q_4}{b\overset{\downarrow}{x}0y1b} \xrightarrow{(x,x,R)} \underset{q_1}{bx\overset{\downarrow}{0}y1b}$$

$$\xrightarrow{(0,x,R)} \underset{q_2}{bxx\overset{\downarrow}{y}1b} \xrightarrow{(y,y,R)} \underset{q_2}{bxxy\overset{\downarrow}{1}b} \xrightarrow{(1,y,L)} \underset{q_3}{bxx\overset{\downarrow}{y}yb}$$

$$\xrightarrow{(y,y,L)} \underset{q_3}{bx\overset{\downarrow}{x}yyb} \xrightarrow{(x,x,R)} \underset{q_5}{bxx\overset{\downarrow}{y}yb} \xrightarrow{(y,y,R)} \underset{q_5}{bxxy\overset{\downarrow}{y}b}$$

$$\xrightarrow{(y,y,R)} \underset{q_5}{bxxyy\overset{\downarrow}{b}} \xrightarrow{(b,b,R)} \underset{q_6}{bxxyyb\overset{\downarrow}{b}}$$

9.4 DESIGN OF TURING MACHINES

We now give the basic guidelines for designing a Turing machine.

(i) The fundamental objective in scanning a symbol by the R/W head is to 'know' what to do in the future. The machine must remember the past symbols scanned. The Turing machine can remember this by going to the next unique state.

(ii) The number of states must be minimized. This can be achieved by changing the states only when there is a change in the written symbol or when there is a change in the movement of the R/W head. We shall explain the design by a simple example.

EXAMPLE 9.5

Design a Turing machine to recognize all strings consisting of an even number of 1's.

Solution

The construction is made by defining moves in the following manner:

(a) q_1 is the initial state. M enters the state q_2 on scanning 1 and writes b.
(b) If M is in state q_2 and scans 1, it enters q_1 and writes b.
(c) q_1 is the only accepting state.

So M accepts a string if it exhausts all the input symbols and finally is in state q_1. Symbolically,

$$M = (\{q_1, q_2\}, \{1, b\}, \{1, b\}, \delta, q, b. \{q_1\})$$

where δ is defined by Table 9.3.

TABLE 9.3 Transition Table for Example 9.5

Present state	1
$\to (q_1)$	bq_2R
q_2	bq_1R

Let us obtain the computation sequence of 11. Thus, $q_1 11 \vdash bq_2 1 \vdash bbq_1$. As q_1 is an accepting state. 11 is accepted. $q_1 111 \vdash bq_2 11 \vdash bbq_1 1 \vdash bbbq_2$. M halts and as q_2 is not an accepting state, 111 is not accepted by M.

EXAMPLE 9.6

Design a Turing machine over $\{1, b\}$ which can compute a concatenation function over $\Sigma = \{1\}$. If a pair of words (w_1, w_2) is the input, the output has to be w_1w_2.

Solution

Let us assume that the two words w_1 and w_2 are written initially on the input tape separated by the symbol b. For example, if $w_1 = 11$, $w_2 = 111$, then the input and output tapes are as shown in Fig. 9.6.

b	1	1	*b*	1	1	1	*b*

b	1	1	1	1	1	*b*	*b*

Fig. 9.6 Input and output tapes.

We observe that the main task is to remove the symbol b. This can be done in the following manner:

(a) The separating symbol b is found and replaced by 1.

(b) The rightmost 1 is found and replaced by a blank b.
(c) The R/W head returns to the starting position.

A computation is illustrated in Table 9.4.

TABLE 9.4 Computation for $11b111$

$$q_0 11b111 \vdash 1q_0 1b111 \vdash 11q_0 b111 \vdash 111q_1 111$$
$$\vdash 1111q_1 11 \vdash 11111q_1 1 \vdash 111111q_1 b \vdash 11111q_2 1b$$
$$\vdash 1111q_3 1bb \vdash 111q_3 11bb \vdash 11q_3 111bb \vdash 1q_3 1111bb$$
$$\vdash q_3 11111bb \vdash q_3 b11111bb \vdash bq_f 11111bb$$

From the above computation sequence for the input string $11b111$, we can construct the transition table given in Table 9.5.

For the input string $1b1$, the computation sequence is given as

$$q_0 1b1 \vdash 1q_0 b1 \vdash 11q_1 1 \vdash 111q_1 b \vdash 11q_2 b \vdash 1q_3 1bb$$
$$\vdash q_3 11bb \vdash q_3 b11bb \vdash bq_f 11bb.$$

TABLE 9.5 Transition Table for Example 9.6

Present state	*Tape symbol*	
	1	b
$\to q_0$	$1Rq_0$	$1Rq_1$
q_1	$1Rq_1$	bLq_2
q_2	bLq_3	—
q_3	$1Lq_3$	bRq_f
q_f	—	—

EXAMPLE 9.7

Design a TM that accepts

$$\{0^n 1^n \mid n \geq 1\}.$$

Solution

We require the following moves:

(a) If the leftmost symbol in the given input string w is 0, replace it by x and move right till we encounter a leftmost 1 in w. Change it to y and move backwards.

(b) Repeat (a) with the leftmost 0. If we move back and forth and no 0 or 1 remains, move to a final state.

(c) For strings not in the form $0^n 1^n$, the resulting state has to be nonfinal.

Keeping these ideas in our mind, we construct a TM M as follows:

where

$$M = (Q, \Sigma, \Gamma, \delta, q_0, b, F)$$

$$Q = \{q_0, q_1, q_2, q_3, q_f\}$$

$$F = \{q_f\}$$

$$\Sigma = \{0, 1\}$$

$$\Gamma = \{0, 1, x, y, b\}$$

The transition diagram is given in Fig. 9.7. M accepts $\{0^n 1^n \mid n \geq 1\}$. The moves for 0011 and 010 are given below just to familiarize the moves of M to the reader.

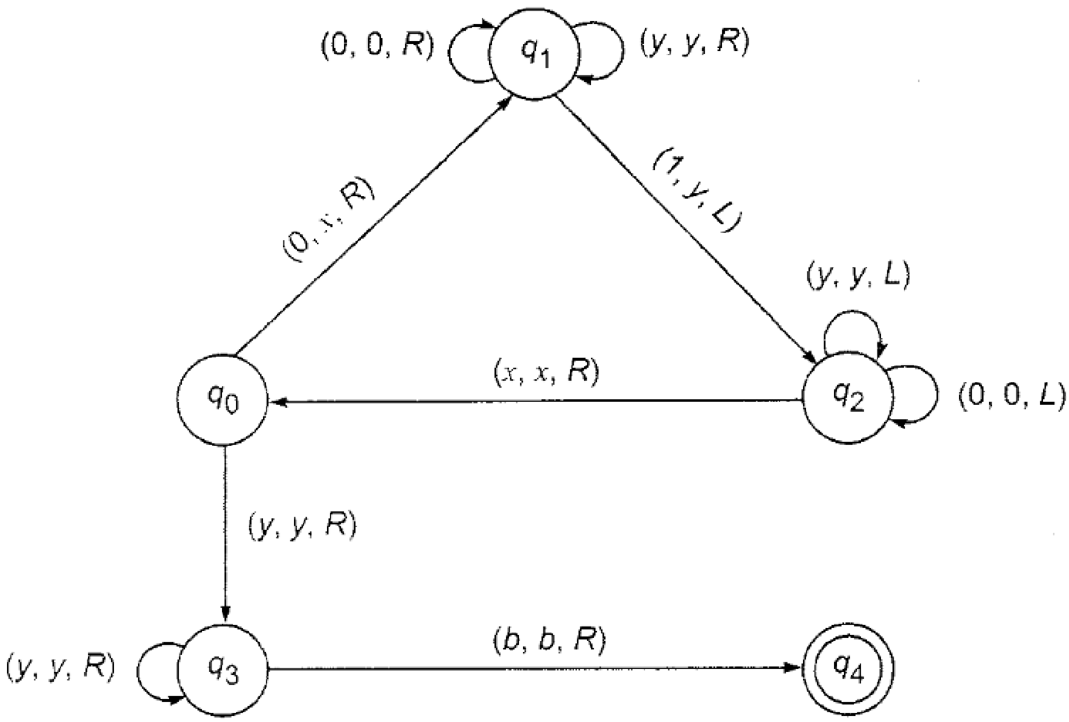


Fig. 9.7 Transition diagram for Example 9.7.

$$q_0 0011 \vdash xq_1 011 \vdash x0q_1 11 \vdash xq_2 0y1$$
$$\vdash q_2 x0y1 \vdash xq_0 0y1 \vdash xxq_1 y1 \vdash xxyq_1 1$$
$$\vdash xxq_2 yy \vdash xq_2 xyy \vdash xxq_0 yy \vdash xxyq_3 y$$
$$\vdash xxyyq_3 = xxyyq_3 b \vdash xxyybq_4 b$$

Hence 0011 is accepted by M.

$$q_0 010 \vdash xq_1 10 \vdash q_2 xy0 \vdash xq_0 y0 \vdash xyq_3 0$$

As $\delta(q_3, 0)$ is not defined, M halts. So 010 is not accepted by M.

EXAMPLE 9.8

Design a Turing machine M to recognize the language

$$\{1^n 2^n 3^n \mid n \geq 1\}.$$

Solution

Before designing the required Turing machine M, let us evolve a procedure for processing the input string 112233. After processing, we require the ID to be of the form $bbbbbbq_7$. The processing is done by using five steps:

Step 1 q_1 is the initial state. The R/W head scans the leftmost 1, replaces 1 by b, and moves to the right. M enters q_2.

Step 2 On scanning the leftmost 2, the R/W head replaces 2 by b and moves to the right. M enters q_3.

Step 3 On scanning the leftmost 3, the R/W head replaces 3 by b, and moves to the right. M enters q_4.

Step 4 After scanning the rightmost 3, the R/W heads moves to the left until it finds the leftmost 1. As a result, the leftmost 1, 2 and 3 are replaced by b.

Step 5 Steps 1–4 are repeated until all 1's, 2's and 3's are replaced by blanks.

The change of IDs due to processing of 112233 is given as

$$q_1112233 \vdash bq_212233 \vdash b1q_22233 \vdash b1bq_3233 \vdash b1b2q_333$$

$$\vdash b1b2bq_43 \vdash b1b_2q_5b3 \vdash b1bq_52b3 \vdash b1q_5b2b3 \vdash bq_51b2b3$$

$$\vdash q_6b1b2b3 \vdash bq_11b2b3 \vdash bbq_2b2b3 \vdash bbbq_22b3$$

$$\vdash bbbbq_3b3 \vdash bbbbbq_33 \vdash bbbbbbq_4b \vdash bbbbbq_7bb$$

Thus,

$$q_1112233 \overset{*}{\vdash} q_7bbbbbb$$

As q_7 is an accepting state, the input string 112233 is accepted.

Now we can construct the transition table for M. It is given in Table 9.6.

TABLE 9.6 Transition Table for Example 9.7

Present state	Input tape symbol			
	1	2	3	b
$\to q_1$	bRq_2			bRq_1
q_2	$1Rq_2$	bRq_3		bRq_2
q_3		$2Rq_3$	bRq_4	bRq_3

q_2	$1Rq_2$	bRq_3		bRq_2
q_3		$2Rq_3$	bRq_4	bRq_3
q_4			$3Lq_5$	bLq_7
q_5	$1Lq_6$	$2Lq_5$		bLq_5
q_6	$1Lq_6$			bRq_1
q_7				

It can be seen from the table that strings other than those of the form $0^n1^n2^n$ are not accepted. It is advisable to compute the computation sequence for strings like 1223, 1123, 1233 and then see that these strings are rejected by M.

9.6 TECHNIQUES FOR TM CONSTRUCTION

In this section we give some high-level conceptual tools to make the construction of TMs easier. The Turing machine defined in Section 9.1 is called the standard Turing machine.

9.6.1 Turing Machine with Stationary Head

In the definition of a TM we defined $\delta(q, a)$ as (q', y, D) where $D = L$ or R. So the head moves to the left or right after reading an input symbol. Suppose, we want to include the option that the head can continue to be in the same cell for some input symbol. Then we define $\delta(q, a)$ as (q', y, S). This means that the TM, on reading the input symbol a, changes the state to q' and writes y in the current cell in place of a and continues to remain in the same cell. In terms of IDs,

$$wqax \vdash wq'yx$$

Of course, this move can be simulated by the standard TM with two moves, namely

$$wqax \vdash wyq''x \vdash wq'yx$$

That is, $\delta(q, a) = (q', y, S)$ is replaced by $\delta(q, a) = (q'', y, R)$ and $\delta(q'', X) = (q', y, L)$ for any tape symbol X.

Thus in this model $\delta(q, a) = (q', y, D)$ where $D = L, R$ or S.

9.6.2 Storage in the State

We are using a state, whether it is of a FA or pda or TM, to 'remember' things. We can use a state to store a symbol as well. So the state becomes a pair (q, a) where q is the state (in the usual sense) and a is the tape symbol stored in (q, a). So the new set of states becomes $Q \times \Gamma$.

in (q, a). So the new set of states becomes $Q \times \Gamma$.

EXAMPLE 9.9

Construct a TM that accepts the language **0 1* + 1 0*.**

Solution

We have to construct a TM that remembers the first symbol and checks that it does not appear afterwards in the input string. So we require two states, q_0, q_1. The tape symbols are 0, 1 and b. So the TM, having the 'storage facility in state', is

$$M = (\{q_0, q_1\} \times \{0, 1, b\}, \{0, 1\}, \{0, 1, b\}, \delta, [q_0, b], \{[q_1, b]\})$$

We describe δ by its implementation description.

1. In the initial state, M is in q_0 and has b in its data portion. On seeing the first symbol of the input sting w, M moves right, enters the state q_1 and the first symbol, say a, it has seen.
2. M is now in $[q_1, a]$. (i) If its next symbol is b, M enters $[q_1, b]$, an accepting state. (ii) If the next symbol is a, M halts without reaching the final state (i.e. δ is not defined). (iii) If the next symbol is $\overline{a}$ ($\overline{a} = 0$ if $a = 1$ and $\overline{a} = 1$ if $a = 0$), M moves right without changing state.
3. Step 2 is repeated until M reaches $[q_1, b]$ or halts (δ is not defined for an input symbol in w).

9.6.3 MULTIPLE TRACK TURING MACHINE

In the case of TM defined earlier, a single tape was used. In a multiple track TM, a single tape is assumed to be divided into several tracks. Now the tape alphabet is required to consist of k-tuples of tape symbols, k being the number of tracks. Hence the only difference between the standard TM and the TM with multiple tracks is the set of tape symbols. In the case of the standard Turing machine, tape symbols are elements of Γ; in the case of TM with multiple track, it is Γ^k. The moves are defined in a similar way.

9.6.4 SUBROUTINES

We know that subroutines are used in computer languages, when some task has to be done repeatedly. We can implement this facility for TMs as well.

First, a TM program for the subroutine is written. This will have an initial state and a 'return' state. After reaching the return state, there is a temporary

First, a TM program for the subroutine is written. This will have an initial state and a 'return' state. After reaching the return state, there is a temporary halt. For using a subroutine, new states are introduced. When there is a need for calling the subroutine, moves are effected to enter the initial state for the subroutine (when the return state of the subroutine is reached) and to return to the main program of TM.

We use this concept to design a TM for performing multiplication of two positive integers.

EXAMPLE 9.10

Design a TM which can multiply two positive integers.

Solution

The input (m, n), m, n being given, the positive integers are represented by 0^m10^n. M starts with 0^m10^n in its tape. At the end of the computation, 0^{mn}(mn in unary representation) surrounded by b's is obtained as the ouput.

The major steps in the construction are as follows:

1. 0^m10^n1 is placed on the tape (the output will be written after the rightmost 1).
2. The leftmost 0 is erased.
3. A block of n 0's is copied onto the right end.
4. Steps 2 and 3 are repeated m times and 10^m10^{mn} is obtained on the tape.
5. The prefix 10^m1 of 10^m10^{mn} is erased, leaving the product mn as the output.

For every 0 in 0^m, 0^n is added onto the right end. This requires repetition of step 3. We define a subroutine called COPY for step 3.

For the subroutine COPY, the initial state is q_1 and the final state is q_5. δ is given by the transition table (see Table 9.7).

TABLE 9.7 Transition Table for Subroutine COPY

State	*Tape symbol*			
	0	1	2	b
q_1	q_22R	q_41L	—	—
q_2	q_20R	q_21R	—	q_30L
q_3	q_30L	q_31L	q_12R	—
q_4	—	q_51R	q_40L	—
q_5	—	—	—	—

The Turing machine M has the initial state q_0. The initial ID for M is

$q_0 0^m 10^n 1$. On seeing 0, the following moves take place (q_6 is a state of M). $q_0 0^m 10^n 1 \vdash b q_6 0^{m-1} 10^n 1 \overset{*}{\vdash} b 0^{m-1} q_6 10^n 1 \vdash b 0^{m-1} 1 q_1 0^n 1$. q_1 is the initial state

of COPY. The TM M_1 performs the subroutine COPY. The following moves take place for M_1: $q_1 0^n 1 \vdash 2 q_2 0^{n-1} 1 \overset{*}{\vdash} 2 0^{n-1} 1 q_3 b \vdash 2 0^{n-1} q_3 1 0 \overset{*}{\vdash} 2 q_1 0^{n-1} 1 0$. After exhausting 0's, q_1 encounters 1. M_1 moves to state q_4. All 2's are converted back to 0's and M_1 halts in q_5. The TM M picks up the computation by starting from q_5. The q_0 and q_6 are the states of M. Additional states are created to check whether each 0 in 0^m gives rise to 0^m at the end of the rightmost 1 in the input string. Once this is over, M erases 10^n1 and finds 0^{nm} in the input tape.

M can be defined by

$$M = (\{q_0, q_1, \ldots, q_{12}\}, \{0, 1\}, \{0, 1, 2, b\}, \delta, q_0, b, \{q_{12}\})$$

where δ is defined by Table 9.8.

TABLE 9.8 Transition Table for Example 9.10

	0	1	2	b
q_0	$q_6 bR$	—	—	—
q_6	$q_6 0R$	$q_1 1R$	—	—
q_5	$q_7 0L$	—	—	—
q_7	—	$q_8 1L$	—	—
q_8	$q_9 0L$	—	—	$q_{10} bR$
q_9	$q_9 0L$	—	—	$q_0 bR$
q_{10}	—	$q_{11} bR$	—	—
q_{11}	$q_{11} bR$	$q_{12} bR$	—	—

Thus M performs multiplication of two numbers in unary representation.

1. Increment a number //unary operators
2. Decrement a number
3. 1's Complement a number
4. 2's Complement a number (when you scan input string from right to left, copy the symbol as it is until you see first 1, and from then onwards 1's complement the symbol for eg. 10010100 for this 2's complement is 01101100)
5. Copy the string
6. Addition of two numbers // binary operators
7. Substraction of two numbers (Always assume that first number is greater than second number)
8. Multiplication of two numbers (Repeated Addition)
9. Division of two numbers (Repeated Substraction)
10. TM as acceptor
11. TM as transducer
12. TM as DPDA (L=wcwR)
13. TM as NPDA (L=wwR) (even length palindromes)
14. TM accepting non-CFL like L={anbncn:n>=1} etc.

Universal TM

10 October 2018 07:47 AM

10.4 A Universal Turing Machine

Consider the following argument against Turing's thesis: "A Turing machine as presented in Definition 9.1 is a special purpose computer. Once δ is defined, the machine is restricted to carrying out one particular type of computation. Digital computers, on the other hand, are general purpose machines that can be programmed to do different jobs at different times. Consequently, Turing machines cannot be considered equivalent to general purpose digital computers."

This objection can be overcome by designing a reprogrammable Turing machine, called a **universal Turing machine**. A universal Turing machine M_u is an automaton that, given as input the description of any Turing machine M and a string w, can simulate the computation of M on w. To construct such an M_u, we first choose a standard way of describing Turing machines. We may, without loss of generality, assume that

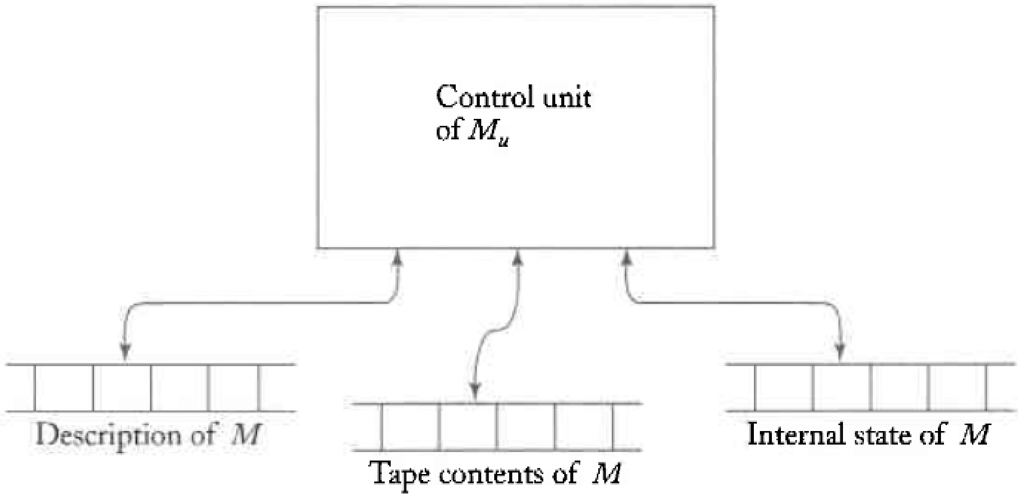
$$Q = \{q_1, q_2, \ldots, q_n\},$$

with q_1 the initial state, q_2 the single final state, and

$$\Gamma = \{a_1, a_2, \ldots, a_m\},$$

where a_1 represents the blank. We then select an encoding in which q_1 is represented by 1, q_2 is represented by 11, and so on. Similarly, a_1 is encoded as 1, a_2 as 11, etc. The symbol 0 will be used as a separator between the 1's. With the initial and final state and the blank defined by this convention, any Turing machine can be described completely with δ only. The transition function is encoded according to this scheme, with the arguments and result in some prescribed sequence. For example, $\delta(q_1, a_2) = (q_2, a_3, L)$ might appear as

$$\cdots 1\,0\,1\,1\,0\,1\,1\,0\,1\,1\,1\,0\,1\,0 \cdots.$$



It follows from this that any Turing machine has a finite encoding as a string on $\{0, 1\}^+$, and that, given any encoding of M, we can decode it uniquely. Some strings will not represent any Turing machine (e.g., the strong 00011), but we can easily spot these, so they are of no concern.

A universal Turing machine M_u then has an input alphabet that includes $\{0, 1\}$ and the structure of a multitape machine, as shown in Figure 10.16.

For any input M and w, tape 1 will keep an encoded definition of M. Tape 2 will contain the tape contents of M, and tape 3 the internal state of M. M_u looks first at the contents of tapes 2 and 3 to determine the configuration of M. It then consults tape 1 to see what M would do in this configuration. Finally, tapes 2 and 3 will be modified to reflect the result of the move.

It is within reason to construct an actual universal Turing machine (see, for example, Denning, Dennis, and Qualitz 1978), but the process is uninteresting. We prefer instead to appeal to Turing's hypothesis. The implementation clearly can be done using some programming language; in fact, the program suggested in Exercise 1, Section 9.1 is a realization of a universal Turing machine in a higher level language. Therefore, we expect that it can also be done by a standard Turing machine. We are then justified in claiming the existence of a Turing machine that, given any program, can carry out the computations specified by that program and that is therefore a proper model for a general purpose computer.

The observation that every Turing machine can be represented by a string of 0's and 1's has important implications. But before we explore these implications, we need to review some results from set theory.

Church-Turing Thesis

10 October 2018 02:10 PM

6.7 THE CHURCH TURING THESIS

In the earlier sections, we saw a mathematical model, namely, Turing machine that can carry out complex tasks such as acceptance of language, computing functions and general purpose computations. At the beginning of the 20th century, the mathematician D. Hilbert asked. "Whether there exists an algorithm that can prove any well-stated mathematical formula". In 1931, K.Gödel showed that such an algorithm cannot exist. In 1936, A.M. Turing proposed Turing machine as a computational model and suggested that the definition of an algorithm can be based on this model. The mathematician and logician, Alonzo church proposed an alternative formalization for the notion of algorithm in 1936, known as *Church-Turing thesis*. This conjecture is stated in number of ways by different writers. Some of the equivalent statements of Church-turing thesis are as follows :

1. "Any computation that can be carried out by mechanical means can be performed by some Turing machine."
2. "Anything that is inituitively computable can be computed by a Turing machine."
3. "The turing machine that halts on all inputs is the precise formal notion corresponding to the intuitive notion of an 'algorithm'."
4. "Given any problem which can be solved with an effective algorithm, there is a TM that can solve this problem."
5. "Any general way to compute is to compute only the partial-recursive functions or equivalently what TMs can compute."
6. "There is no formalism to model any mechanical calculus that is more powerful then TMs and equivalent formalisms."
7. "A number-theoretic function is computable by an algorithm if and only if it is Turing computable".

The word "thesis" is used instead of the word "theorem" as it is not a mathematical result. It is based on the intuitive notion of what "mechanical computations" are and equates it with a mathematical idea, *i.e.*, "algorithm". In fact, Church-Turing thesis is non-provable. It is supported only by previous experience and by intuitive evidences given as follows :

1. So far whatever alternative models have been proposed for mechanical computations are not found to be more powerful than the Turing machine.
2. There exists no problem which is solvable by an effective algorithm cannot be solved by TM.

machine.

2. There exists no problem which is solvable by an effective algorithm but cannot be solved by TM.
3. All known formalisms to model discrete computing devices have at most the power of TMs, (or anything that can be done on any existing digital computer can also be done by a TM).

The *Church-Turing thesis* is now universally accepted and thus, we have accepted the TM as the ultimate computational model.

The term *"mechanical calculus"* (*i.e.,* mechanical computation) used in statement 1 may be defined as a computation which can be performed by some TM. This statement suggests that *"Do not try to solve mechanically what cannot be solved by TMs"*. Statement 3 suggests that an "algorithm" must exclude TMs that may not halt on some inputs. The statement 4 uses the term *"effective algorithm"* which is informal and imprecise. However, any problem should be considered to be effectively solvable if it can be solved using a computer with a program and using an unlimited amount of memory. The term *"number-theoretic function"* of statement 7 refers to functions from a subset of N^k into N for any $k \geq 1$. Let T be a turing machine and α be a string of tape symbols. Let $T(\alpha) = \beta$, *i.e.,* T eventually halts with a string β in the tape. We can now think of T as computing number-theoretic function, by letting a string of 1's of length $n + 1$ as the unary representation of the non-negative integer n.