II B. Tech. – II Semester
(20BT41501) THEORY OF COMPUTATION

(Common to CSE, CSSE and IT)

Int. Marks	Ext. Marks	Total Marks	L	T	P	C
30	70	100	3	-	-	3

PRE-REQUISITES: A Course on –Discrete Mathematical Structures‖

COURSE DESCRIPTION: Finite automata; Nondeterministic Finite automata; Regular expressions; Applications of the pumping lemma; Context-Free Grammars; Normal forms for context-free grammars; pushdown automata; Chomsky hierarchy of languages; Turing machines.

COURSE OUTCOMES: *After successful completion of this course, the students will be able to:*

CO1. Design finite state machines to recognize formal languages.

CO2. Analyze formal languages using automata.

CO3. Identify different types of grammars in formal languages.

CO4. Construct context free grammars for context free languages

CO5. Develop Turing machine for different computational problems.

CO6. Validate formal languages of automata by applying closure properties.

DETAILED SYLLABUS:

UNIT I– FINITE AUTOMATA (10 periods)

Introduction to Finite automata, the central concepts of automata theory, Deterministic finite automata, Nondeterministic Finite automata, the equivalence of DFA and NDFA, Finite automata with epsilon-transitions, Conversion of epsilon-NFA to NFA and DFA, Mealy and Moore models.

UNIT II– REGULAR EXPRESSIONS AND LANGUAGES (9 periods)

Regular expressions, Identity rules, Finite automata and Regular expressions, Applications of regular expressions, Pumping lemma for regular languages, Applications of the pumping lemma, Closure properties of regular languages, Equivalence of two regular expressions, Equivalence of two finite automata and minimization of automata.

UNIT III –CONTEXT-FREE GRAMMARS (9 periods)

Context-Free Grammars, Parse trees, Applications of context free grammars, Ambiguity in grammars and languages, Normal forms for context-free grammars, the pumping lemma for context-free languages.

UNIT IV – PUSH DOWN AUTOMATA (8 periods)

Definition of the pushdown automaton, the languages of a PDA, Equivalence of PDA‘s and CFG‘s, Deterministic pushdown automata, Chomsky hierarchy of languages, Undecidability.

UNIT V –TURING MACHINE (9 periods)

Turing machine model, Representation of Turing machine, Language acceptability by Turing machine, Design of Turing machine, Techniques for Turing machine construction, Variants of Turing machines, Universal Turing machine, Recursive and recursively enumerable languages (REL), properties of recursive and recursively enumerable languages, the model of linear bounded automaton.

Total Periods: 45

Topics for self-study are provided in the lesson plan

TEXT BOOK:

T1. John E. Hopcroft, Rajeev Motwani and Jeffrey D Ullman, *Introduction to Automata Theory, Languages and Computation*, Pearson Education, 3rd Edition, 2011.

REFERENCE BOOKS:

R1. K.L.P. Mishra and N. Chandrasekaran, *Theory of Computer Science: Automata Languages and Computation*, PHI Learning, 3rd Edition, 2009.

R2. John C Martin, *Introduction to Languages and the Theory of Computation*, TMH, 4th Edition, 2010.

ADDITIONAL LEARNING RESOURCES:

https://nptel.ac.in/courses/106/104/106104148/

SREE VIDYANIKETHAN ENGINEERING COLLEGE

(Autonomous)

SreeSainath Nagar, A. Rangampet-517 102

Department of Computer Science and Engineering

Lesson Plan

Name of the Course : Theory of Computation

Class & Semester :III B.Tech I Semester(20BT41501)

Name(s) of the faculty Member(s) **:**Mr.V.SIVA PRASAD

Section :CSE

S. No	Topic	No. of periods	Book(s) followed	Date(s)	Topics for self-study
UNIT – I: FINITE AUTOMATA					
1.	Introduction to Finite automata,	1	T1		Formal Languages
2.	the central concepts of automata theory	1	T1		
3.	Deterministic finite automata,	1	T1		
4.	Nondeterministic Finite automata	1	T1		
5.	the equivalence of DFA and NDFA	1	T1		
6.	Finite automata with epsilon-transitions	1	T1		
7.	Conversion of epsilon-NFA to NFA	1	T1		
8.	Conversion of epsilon-NFA to DFA	1	T1		
9.	Mealy and Moore models	1	T1		
10	Formative Test	1			
Total periods required:		10			
UNIT-II: REGULAR EXPRESSIONS AND LANGUAGES					
11.	Regular expressions,	1	T1		Alphabet ,String and Language
12.	Identity rules	1	T1		
13.	Finite automata and Regular expressions,	1	T1		
14.	Applications of regular expressions	1	T1		
15.	Pumping lemma for regular languages,	1	T1		
16.	Applications of the pumping lemma	1	T1		
17.	Closure properties of regular languages	1	T1		
18.	Equivalence of two regular expressions,	1	T1		
19.	Equivalence of two finite automata and minimization of automata.	1	T1		
Total periods required:		**09**			
UNIT – III: CONTEXT-FREE GRAMMARS					
20.	Context-Free Grammars,	1	T1		Applications of Finite Automata
21.	Parse trees,	1	T1		
22.	Applications of context free grammars,	1	T1		
23.	Ambiguity in grammars,	1	T1		
24.	Ambiguity in languages,	1	T1		
25	Normal forms for context-free grammars,	1	T1		
26	pumping lemma for context-free languages.	1	T1		
27	Formative Test 3	1	T1		
Total periods required:		**09**			
UNIT- IV : PUSH DOWN AUTOMATA					

S. No	Topic	No. of periods	Book(s) followed	Date(s)	Topics for self-study
28	Definition of the pushdown automaton	1	T1		Regular Languages
29	the languages of a PDA	1	T1		
30	Equivalence of PDA‘s	1	T1		
31	Equivalence of CFG‘s	1	T1		
32	Deterministic pushdown automata,	1	T1		
33	Chomsky hierarchy of languages,	1	T1		
34	Un decidability	1	T1		
35	Formative Test 4				
Total periods required:		**8**			
UNIT-V: TURING MACHINE					
36	Turing machine model and Representation	1	T1		Two Stack PDA
37	Language acceptability by Turing machine,	1	T1		
38	Design of Turing machine,	1	T1		
39	Techniques for Turing machine construction,	1	T1		
38	Variants of Turing machines, Universal	1	T1		
39	Recursive and REL	1	T1		
40	properties of recursive and REL	1	T1		
41	the model of linear bounded automaton.	1	T1		
42	Formative Test 5				
Total periods required:		**9**			
Grand total periods required:		**45**			

TEXT BOOK:

T1. John E. Hopcroft, Rajeev Motwani and Jeffrey D Ullman, Introduction to Automata Theory, Languages and Computation, Pearson Education, 3rd Edition, 2011.

REFERENCE BOOKS:

R1. K.L.P. Mishra and N. Chandrasekaran, Theory of Computer Science: Automata Languages and Computation, PHI Learning, 3rd Edition, 2009.

R2. John C Martin, Introduction to Languages and the Theory of Computation, TMH, 4 th Edition, 2010

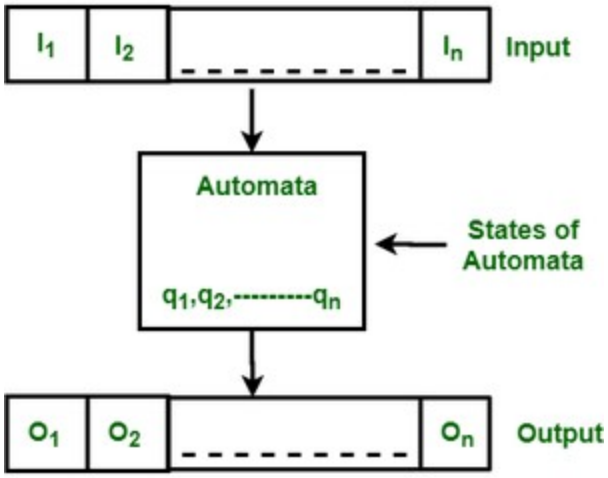
Signature of the Faculty **Signature of the Course Coordinator** **Signature of the HOD**

UNIT-1

FINITE AUTOMATA

Introduction to Finite Automata:

Finite Automata(FA) is the simplest machine to recognize patterns. The finite automata or finite state machine is an abstract machine that has five elements or tuples. It has a set of states and rules for moving from one state to another but it depends upon the applied input symbol. Basically, it is an abstract model of a digital computer. The following figure shows some essential features of general automation.



Finite Automata: It is used to recognize patterns of specific type input. It is the most restricted type of automata which can accept only regular languages (languages which can be expressed by regular expression using OR (+), Concatenation (.), Kleene Closure(*) like a*b*, (a+b) etc.)

Deterministic FA and Non-Deterministic FA: In deterministic FA, there is only one move from every state on every input symbol but in Non-Deterministic FA, there can be zero or more than one move from one state for an input symbol.

Note:

- Language accepted by NDFA and DFA are same.
- Power of NDFA and DFA is same.
- No. of states in NDFA is less than or equal to no. of states in equivalent DFA.
- For NFA with n-states, in worst case, the maximum states possible in DFA is 2n
- Every NFA can be converted to corresponding DFA.

The above figure shows the following features of automata:

Input

Output

States of automata

State relation

Output relation

A Finite Automata consists of the following:

Q : Finite set of states.

Σ : set of Input Symbols.

q : Initial state.

F : set of Final States.

δ : Transition Function.

1) Deterministic Finite Automata (DFA):

DFA consists of 5 tuples {Q, Σ, q, F, δ}.

Q : set of all states.

Σ : set of input symbols. (Symbols which machine takes as input)

q : Initial state. (Starting state of a machine)

F : set of final state.

δ : Transition Function, defined as δ : Q X Σ --> Q.

In a DFA, for a particular input character, the machine goes to one state only.

A transition function is defined on every state for every input symbol. Also in DFA null (or ε) move is not allowed, i.e., DFA cannot change state without any input character.

For example, below DFA with Σ = {0, 1} accepts all strings ending with 0.

One important thing to note is, **there can be many possible DFAs for a pattern**. A DFA with a minimum number of states is generally preferred.

2) Nondeterministic Finite Automata(NFA):

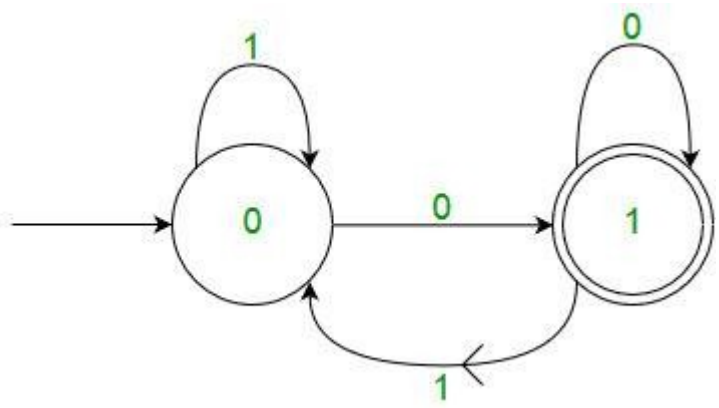
NFA is similar to DFA except following additional features:

Null (or ε) move is allowed i.e., it can move forward without reading symbols.

Ability to transmit to any number of states for a particular input.

However, these above features don't add any power to NFA. If we compare both in terms of power, both are equivalent.

Due to the above additional features, NFA has a different transition function, the rest is the same as DFA.

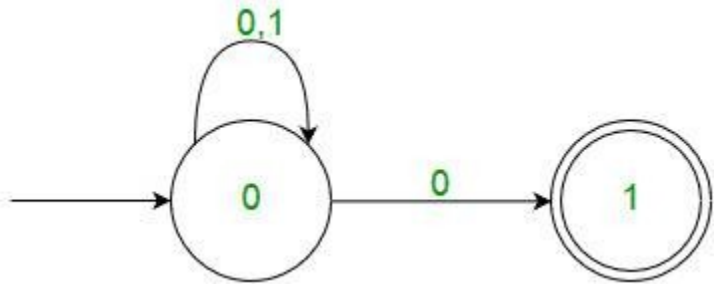


One important thing to note is, there can be many possible DFAs for a pattern. A DFA with a minimum number of states is generally preferred.

δ: Transition Function

δ: Q X (Σ U ε) --> 2 ^ Q.

As you can see in the transition function is for any input including null (or ε), NFA can go to any state number of states. For example, below is an NFA for the above problem.



One important thing to note is, in NFA, if any path for an input string leads to a final state, then the input string is accepted. For example, in the above NFA, there are multiple paths for the input string "00". Since one of the paths leads to a final state, "00" is accepted by the above NFA.

Some Important Points:

Justification:

Since all the tuples in DFA and NFA are the same except for one of the tuples, which is Transition Function (δ)

In case of DFA

δ : Q X Σ --> Q

Now if you observe you'll find out Q X Σ –> Q is part of Q X Σ –> 2Q.

On the RHS side, Q is the subset of 2Q which indicates Q is contained in 2Q or Q is a part of 2Q, however, the reverse isn't true. So mathematically, we can conclude that every DFA is NFA but not vice-versa. Yet there is a way to convert an NFA to DFA, so there exists an equivalent DFA for every NFA.

Both NFA and DFA have the same power and each NFA can be translated into a DFA.

There can be multiple final states in both DFA and NFA.

NFA is more of a theoretical concept.

DFA is used in Lexical Analysis in Compiler.

If the number of states in the NFA is N then, its DFA can have maximum 2N number of states.

Moore Machines: Moore machines are finite state machines with output value and its output depends only on present state. It can be defined as (Q, q0, ∑, O, δ, λ) where:

Q is finite set of states.

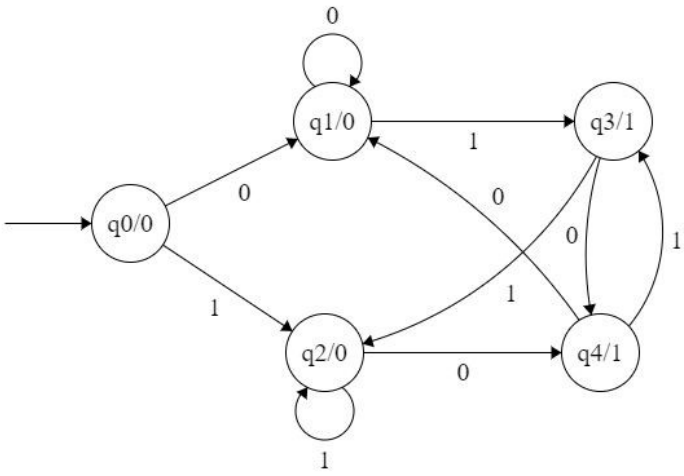
q0 is the initial state.

∑ is the input alphabet.

O is the output alphabet.

δ is transition function which maps Q×∑ → Q.

λ is the output function which maps Q → O.



In the moore machine shown in Figure 1, the output is represented with each input sta te separated by /. The length of output for a moore machine is greater than input by 1.

Input: 11

Transition: δ (q0,11)=> δ(q2,1)=>q2

Output: 000 (0 for q0, 0 for q2 and again 0 for q2)

Mealy Machines: Mealy machines are also finite state machines with output value and its output depends on
present state and current input symbol. It can be defined as (Q, q0, ∑, O, δ, λ') where:

Q is finite set of states.

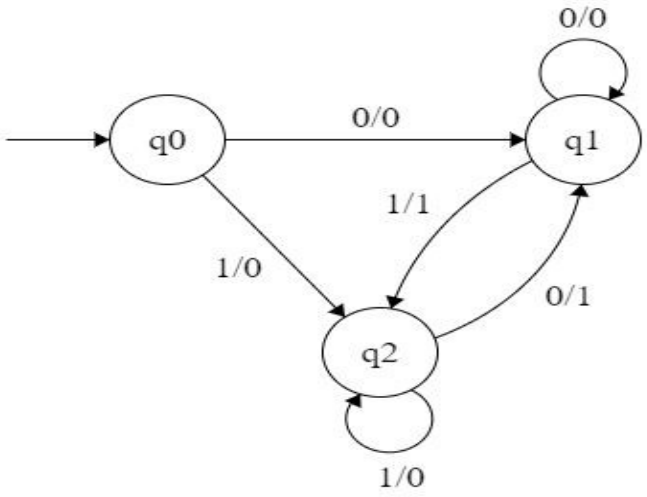
q0 is the initial state.

∑ is the input alphabet.

O is the output alphabet.

δ is transition function which maps Q×∑ → Q.

'λ' is the output function which maps Q×∑→ O.



In the mealy machine shown in Figure 1, the output is represented with each input sy mbol for each state separated
by /. The length of output for a mealy machine is equal to the length of input.

Input:1 1

Transition: δ (q0,11)=> δ(q2,1)=>q2

Output: 00 (q0 to q2 transition has Output 0 and q2 to q2 transition also has Output 0)

NOTE :

If there are n inputs in the Mealy machine then it generates n outputs while if there are n inputs in the Moore machine then it generates n + 1 outputs.

Conversion from Mealy to Moore Machine

Let us take the transition table of mealy machine shown in Figure 2.

	Input=0		Input=1	
Present State	Next State	Output	Next State	Output
q0	q1	0	q2	0
q1	q1	0	q2	1
q2	q1	1	q2	0

Table 1

Step 1. First find out those states which have more than 1 outputs associated with th em. q1 and q2 are the states which have both output 0 and 1 associated with them.

Step 2. Create two states for these states. For q1, two states will be q10 (state with output 0) and q11 (state with
output 1). Similarly for q2, two states will be q20 and q21.

Step 3. Create an empty moore machine with new generated state. For moore machi ne, Output will be associated to each state irrespective of inputs.

Step 4. Fill the entries of next state using mealy machine transition table shown in Ta ble 1. For q0 on input 0,next
state is q10 (q1 with output 0). Similarly, for q0 on input 1, next state is q20 (q2 with o utput 0). For q1 (both q10 and q11) on input 0, next state is q10. Similarly, for q1(both q10 and q11), next state is q21. For q10, output will be 0 and for q11, output will be 1

	Input=0	Input=1	
Present State	Next State	Next State	Output
q0			
q10			
q11			
q20			
q21			

Table 2

. Similarly, other entries can be fill ed.

	Input=0	Input=1	
Present State	Next State	Next State	Output
q0	q10	q20	0
q10	q10	q21	0
q11	q10	q21	1
q20	q11	q20	0

q21	q11	q20	1

Table 3

This is the transition table of moore machine shown in Figure 1.

Conversion from moore machine to mealy machine

Let us take the moore machine of Figure 1 and its transition table is shown in Table 3.
Step 1. Construct an empty mealy machine using all states of moore machine as show n in Table 4.

	Input=0		Input=1	
Present State	Next State	Output	Next State	Output
q0				
q10				
q11				
q20				
q21				

Table 4

Step 2: Next state for each state can also be directly found from moore machine transi tion Table as:

	Input=0		Input=1	
Present State	Next State	Output	Next State	Output
q0	q10		q20	
q10	q10		q21	
q11	q10		q21	

q20	q11		q20	
q21	q11		q20	

Table 5

Step 3: As we can see output corresponding to each
input in moore machine transition table. Use this to fill the Output entries.

e.g.; Output corresponding to q10, q11, q20 and q21 are 0, 1, 0 and 1 respectively.

	Input=0		Input=1	
Present State	Next State	Output	Next State	Output
q0	q10	0	q20	0
q10	q10	0	q21	1
q11	q10	0	q21	1
q20	q11	1	q20	0
q21	q11	1	q20	0

Table 6

Step 4: As we can see from table 6, q10 and q11 are similar to each other (same val ue of next state and Output for
different Input). Similarly, q20 and q21 are also similar. So, q11 and q21 can be elimin ated.

	Input=0		Input=1	
Present State	Next State	Output	Next State	Output
q0	q10	0	q20	0
q10	q10	0	q21	1

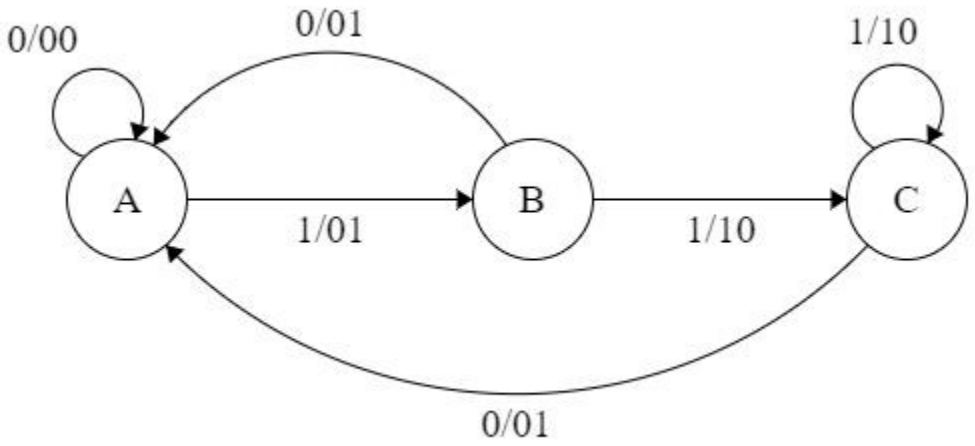
q20	q11	1	q20	0

Table 7

This is the same mealy machine shown in Table 1. So we have converted mealy to mo ore machine and converted back moore to mealy.

Note: Number of states in mealy machine can't be greater than number of states in moore machine.

Example: The Finite state machine described by the following state diagram with A as starting state, where an arc label is x / y and x stands for 1-bit input and y stands for 2- bit output?



Outputs the sum of the present and the previous bits of the input.

Outputs 01 whenever the input sequence contains 11.

Outputs 00 whenever the input sequence contains 10.

None of these.

Solution: Let us take different inputs and its output and check which option works:

Input: 01

Output: 00 01 (For 0, Output is 00 and state is A. Then, for 1, Output is 01 and state will be B)

Input: 11

Output: 01 10 (For 1, Output is 01 and state is B. Then, for 1, Output is 10 and state is C)

As we can see, it is giving the binary sum of present and previous bit. For first bit, prev ious bit is taken as 0.

This article is contributed by Sonal Tuteja. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Conversion of Epsilon-NFA to NFA

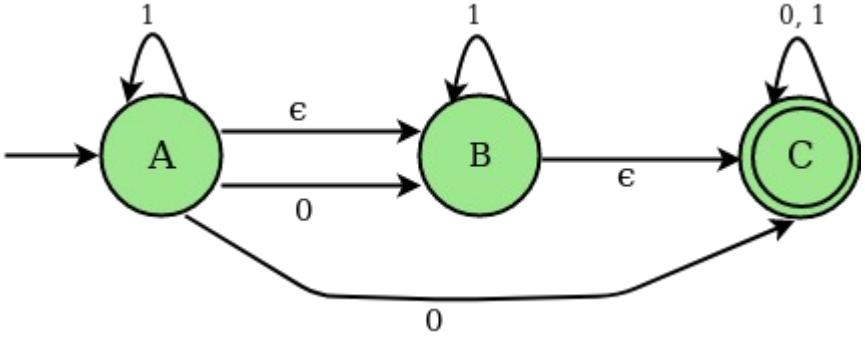
Non-deterministic Finite Automata (NFA) : NFA is a finite automaton where for some cases when a single input is given to a single state, the machine goes to more than 1 states, i.e. some of the moves cannot be uniquely determined by the present state and the present input symbol.

An NFA can be represented as **M = { Q, ∑, ∂, q0, F}**

Q → Finite non-empty set of states. ∑ → Finite non-empty set of input symbols. ∂ → Transitional Function. q0 → Beginning state. F → Final State

NFA with (null) or ∈ move : If any finite automata contains ε (null) move or transaction, then that finite automata is called NFA with ∈ moves

Example : Consider the following figure of NFA with ∈



move :

Transition state table for the above NFA

STATES	0	1	epsilon
A	B, C	A	B
B	–	B	C
C	C	C	–

Epsilon – closure : Epsilon closure for a given state X is a set of states which can be reached from the states X with only (null) or ε moves including the state X itself. In

other words, ε-closure for a state can be obtained by union operation of the ε-closure of the states which can be reached from X with a single ε move in recursive manner. For the above example ∈ closure are as follows :

Epsilon closure(A) : {A, B, C}

Epsilon closure(B) : {B, C}

Epsilon closure(C) : {C}

Deterministic Finite Automata (DFA) : DFA is a finite automata where, for all cases, when a single input is given to a single state, the machine goes to a single state, i.e., all the moves of the machine can be uniquely determined by the present state and the present input symbol.

Steps to Convert NFA with ε-move to DFA :

Step 1 : Take ∈ closure for the beginning state of NFA as beginning state of DFA.

Step 2 : Find the states that can be traversed from the present for each input symbol (union of transition value and their closures for each states of NFA present in current state of DFA).

Step 3 : If any new state is found take it as current state and repeat step 2.

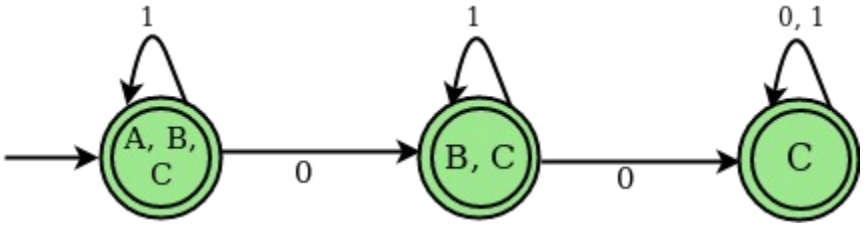
Step 4 : Do repeat Step 2 and Step 3 until no new state present in DFA transition table.

Step 5 : Mark the states of DFA which contains final state of NFA as final states of DFA.

Transition State Table for DFA corresponding to above NFA

STATES	0	1
A, B, C	B, C	A, B, C
B, C	C	B, C
C	C	C

DFA STATE DIAGRAM



Examples :

Input : 6 2

FC - BF

- C -

- - D

E A -

A - BF

- - -

Output :

STATES OF NFA : A, B, C, D, E, F,

GIVEN SYMBOLS FOR NFA: 0, 1, eps

NFA STATE TRANSITION TABLE

STATES	0	1	eps
A	FC	-	BF
B	-	C	-
C	-	-	D
D	E	A	-
E	A	-	BF
F	-	-	-

e-Closure (A) : ABF

e-Closure (B) : B

e-Closure (C) : CD

e-Closure (D) : D
e-Closure (E) : BEF
e-Closure (F) : F
****

DFA TRANSITION STATE TABLE
STATES OF DFA : ABF, CDF, CD, BEF,
GIVEN SYMBOLS FOR DFA: 0, 1,

STATES	0	1
ABF	CDF	CD
CDF	BEF	ABF
CD	BEF	ABF
BEF	ABF	CD

Input :9 2- - BH- - CE D - - - G- F -- - G- - BHI - -- - -

Output :

STATES OF NFA : A, B, C, D, E, F, G, H, I,
GIVEN SYMBOLS FOR NFA: 0, 1, eps
NFA STATE TRANSITION TABLE

STATES	0	1	eps
A	-	-	BH
B	-	-	CE
C	D	-	-
D	-	-	G
E	-	F	-
F	-	-	G
G	-	-	BH
H	I	-	-
I	-	-	-

e-Closure (A) : ABCEH

e-Closure (B) : BCE

e-Closure (C) : C

e-Closure (D) : BCDEGH

e-Closure (E) : E

e-Closure (F) : BCEFGH

e-Closure (G) : BCEGH

e-Closure (H) : H

e-Closure (I) : I

****

DFA TRANSITION STATE TABLE

STATES OF DFA : ABCEH, BCDEGHI, BCEFGH,

GIVEN SYMBOLS FOR DFA: 0, 1,

STATES	0	1
ABCEH	BCDEGHI	BCEFGH
BCDEGHI	BCDEGHI	BCEFGH
BCEFGH	BCDEGHI	BCEFGH

UNIT-II

Regular Expressions, Grammar and Languages

As discussed in Chomsky Hierarchy, Regular Languages are the most restricted types of languages and are accepted by finite automata.

Regular Expressions

Regular Expressions are used to denote regular languages. An expression is regular if:

- ϕ is a regular expression for regular language ϕ.
- ε is a regular expression for regular language {ε}.
- If a ∈ Σ (Σ represents the input alphabet), a is regular expression with language {a}.
- If a and b are regular expression, a + b is also a regular expression with language {a,b}.
- If a and b are regular expression, ab (concatenation of a and b) is also regular.
- If a is regular expression, a* (0 or more times a) is also regular.

Identities for regular expression –

There are many identities for the regular expression. Let p, q and r are regular expressions.

- ∅ + r = r
- ∅.r= r.∅ = ∅
- ∈.r = r.∈ =r
- ∈* = ∈ and ∅* = ∈
- r + r = r
- r*.r* = r*
- r.r* = r*.r = r^+.
- (r*)* = r*
- ∈ +r.r* = r* = ∈ + r.r*
- (p.q)*.p = p.(q.p)*
- (p + q)* = (p*.q*)* = (p* + q*)*
- (p+ q).r= p.r+ q.r and r.(p+q) = r.p + r.q

Regular Expressions

Regular Expressions are used to denote regular languages. An expression is regular if:

- ϕ is a regular expression for regular language ϕ.
- ε is a regular expression for regular language {ε}.
- If a ∈ Σ (Σ represents the input alphabet), a is regular expression with language {a}.
- If a and b are regular expression, a + b is also a regular expression with language {a,b}.
- If a and b are regular expression, ab (concatenation of a and b) is also regular.
- If a is regular expression, a* (0 or more times a) is also regular.

Closure Properties of Regular Languages

Union : If L1 and If L2 are two regular languages, their union L1 ∪ L2 will also be regular. For example, L1 = $\{a^n \mid n \geq 0\}$ and L2 = $\{b^n \mid n \geq 0\}$
L3 = L1 ∪ L2 = $\{a^n \cup b^n \mid n \geq 0\}$ is also regular.

Intersection : If L1 and If L2 are two regular languages, their intersection L1 ∩ L2 will also be regular. For example,
L1= $\{a^m b^n \mid n \geq 0$ and $m \geq 0\}$ and L2= $\{a^m b^n \cup b^n a^m \mid n \geq 0$ and $m \geq 0\}$
L3 = L1 ∩ L2 = $\{a^m b^n \mid n \geq 0$ and $m \geq 0\}$ is also regular.

Concatenation : If L1 and If L2 are two regular languages, their concatenation L1.L2 will also be regular. For example,
L1 = $\{a^n \mid n \geq 0\}$ and L2 = $\{b^n \mid n \geq 0\}$
L3 = L1.L2 = $\{a^m . b^n \mid m \geq 0$ and $n \geq 0\}$ is also regular.

Kleene Closure : If L1 is a regular language, its Kleene closure L1* will also be regular. For example,
L1 = (a ∪ b)
L1* = (a ∪ b)*

Complement : If L(G) is regular language, its complement L'(G) will also be regular. Complement of a language can be found by subtracting strings which are in L(G) from all possible strings. For example,
L(G) = $\{a^n \mid n > 3\}$
L'(G) = $\{a^n \mid n <= 3\}$

Note : Two regular expressions are equivalent if languages generated by them are same. For example, (a+b*)* and (a+b)* generate same language. Every string which is generated by (a+b*)* is also generated by (a+b)* and vice versa.

Pumping Lemma in Theory of Computation

There are two Pumping Lemmas, which are defined for
1. Regular Languages, and

2. Context – Free Languages

Pumping Lemma for Regular Languages

For any regular language L, there exists an integer n, such that for all x ∈ L with |x| ≥ n, there exists u, v, w ∈ Σ∗, such that x = uvw, and
(1) |uv| ≤ n
(2) |v| ≥ 1
(3) for all i ≥ 0: uviw ∈ L

In simple terms, this means that if a string v is ‘pumped’, i.e., if v is inserted any number of times, the resultant string still remains in L.

Pumping Lemma is used as a proof for irregularity of a language. Thus, if a language is regular, it always satisfies pumping lemma. If there exists at least one string made from pumping which is not in L, then L is surely not regular.
The opposite of this may not always be true. That is, if Pumping Lemma holds, it does not mean that the language is regular.
There are two Pumping Lemmas, which are defined for
1. Regular Languages, and
2. Context – Free Languages

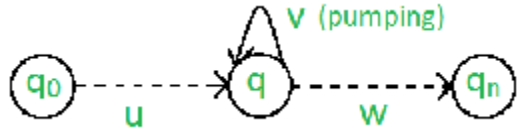
Pumping Lemma for Regular Languages

For any regular language L, there exists an integer n, such that for all x ∈ L with |x| ≥ n, there exists u, v, w ∈ Σ∗, such that x = uvw, and
(1) |uv| ≤ n
(2) |v| ≥ 1
(3) for all i ≥ 0: uviw ∈ L

In simple terms, this means that if a string v is ‘pumped’, i.e., if v is inserted any number of times, the resultant string still remains in L.

Pumping Lemma is used as a proof for irregularity of a language. Thus, if a language is regular, it always satisfies pumping lemma. If there exists at least one string made from pumping which is not in L, then L is surely not regular.
The opposite of this may not always be true. That is, if Pumping Lemma

holds, it does not mean that the language is regular.



For example, let us prove L01 = {0n1n | n ≥ 0} is irregular.
Let us assume that L is regular, then by Pumping Lemma the above given rules follow.
Now, let x ∈ L and |x| ≥ n. So, by Pumping Lemma, there exists u, v, w such that (1) – (3) hold.

We show that for all u, v, w, (1) – (3) does not hold.
If (1) and (2) hold then x = 0n1n = uvw with |uv| ≤ n and |v| ≥ 1.
So, u = 0a, v = 0b, w = 0c1n where : a + b ≤ n, b ≥ 1, c ≥ 0, a + b + c = n
But, then (3) fails for i = 0
uv0w = uw = 0a0c1n = 0a + c1n ∉ L, since a + c ≠ n.

Minimization of DFA

DFA minimization stands for converting a given DFA to its equivalent DFA with minimum number of states. DFA minimization is also called as Optimization of DFA and uses partitioning algorithm.

Minimization of DFA

Suppose there is a DFA D < Q, Σ, q0, δ, F > which recognizes a language L. Then the minimized DFA D < Q', Σ, q0, δ', F' > can be constructed for language L as:

Step 1: We will divide Q (set of states) into two sets. One set will contain all final states and other set will contain non-final states. This partition is called P0.

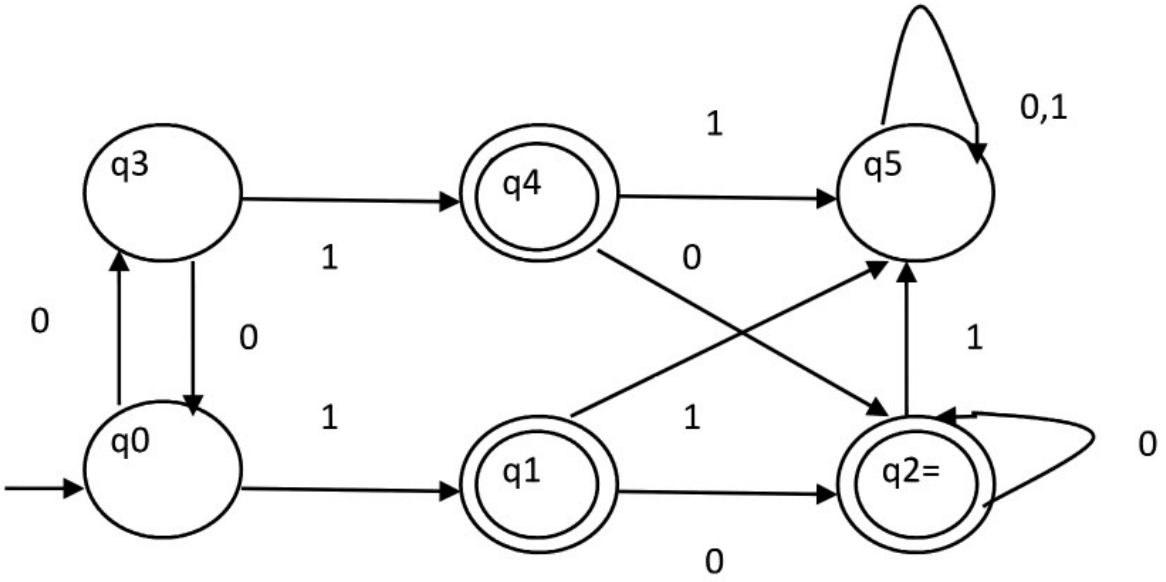
Step 2: Initialize k = 1

Step 3: Find Pk by partitioning the different sets of Pk-1. In each set of Pk-1, we will take all possible pair of states. If two states of a set are distinguishable, we will split the sets into different sets in Pk.

Step 4: Stop when Pk = Pk-1 (No change in partition)

Step 5: All states of one set are merged into one. No. of states in minimized DFA will be equal to no. of sets in Pk.

How to find whether two states in partition Pk are distinguishable ?
Two states (qi, qj) are distinguishable in partition Pk if for any input symbol a, δ (qi, a) and δ (qj, a) are in different sets in partition Pk-1.
Example
Consider the following DFA shown in figure.



Step 1. P0 will have two sets of states. One set will contain q1, q2, q4 which are final states of DFA and another set will contain remaining states. So P0 = { { q1, q2, q4 }, { q0, q3, q5 } }.
Step 2. To calculate P1, we will check whether sets of partition P0 can be partitioned or not:
i) For set { q1, q2, q4 } :
δ (q1, 0) = δ (q2, 0) = q2 and δ (q1, 1) = δ (q2, 1) = q5, So q1 and q2 are not distinguishable.
Similarly, δ (q1, 0) = δ (q4, 0) = q2 and δ (q1, 1) = δ (q4, 1) = q5, So q1 and q4 are not distinguishable.
Since, q1 and q2 are not distinguishable and q1 and q4 are also not distinguishable, So q2 and q4 are not distinguishable. So, { q1, q2, q4 } set will not be partitioned in P1.
ii) For set { q0, q3, q5 } :
δ (q0, 0) = q3 and δ (q3, 0) = q0
δ (q0, 1) = q1 and δ(q3, 1) = q4
Moves of q0 and q3 on input symbol 0 are q3 and q0 respectively which are in same set in partition P0. Similarly, Moves of q0 and q3 on input symbol 1 are q1 and q4 which are in same set in partition P0. So, q0 and q3 are not distinguishable.
δ (q0, 0) = q3 and δ (q5, 0) = q5 and δ (q0, 1) = q1 and δ (q5, 1) = q5
Moves of q0 and q5 on input symbol 1 are q1 and q5 respectively which are

in different set in partition P0. So, q0 and q5 are distinguishable. So, set { q0, q3, q5 } will be partitioned into { q0, q3 } and { q5 }. So,
P1 = { { q1, q2, q4 }, { q0, q3}, { q5 } }

To calculate P2, we will check whether sets of partition P1 can be partitioned or not:

iii)For set { q1, q2, q4 } :

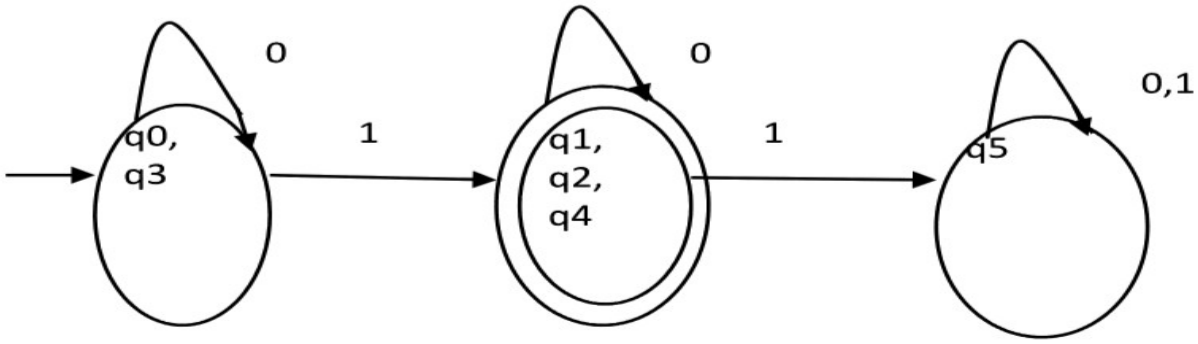
δ (q1, 0) = δ (q2, 0) = q2 and δ (q1, 1) = δ (q2, 1) = q5, So q1 and q2 are not distinguishable.
Similarly, δ (q1, 0) = δ (q4, 0) = q2 and δ (q1, 1) = δ (q4, 1) = q5, So q1 and q4 are not distinguishable.
Since, q1 and q2 are not distinguishable and q1 and q4 are also not distinguishable, So q2 and q4 are not distinguishable. So, { q1, q2, q4 } set will not be partitioned in P2.

iv)For set { q0, q3 } :

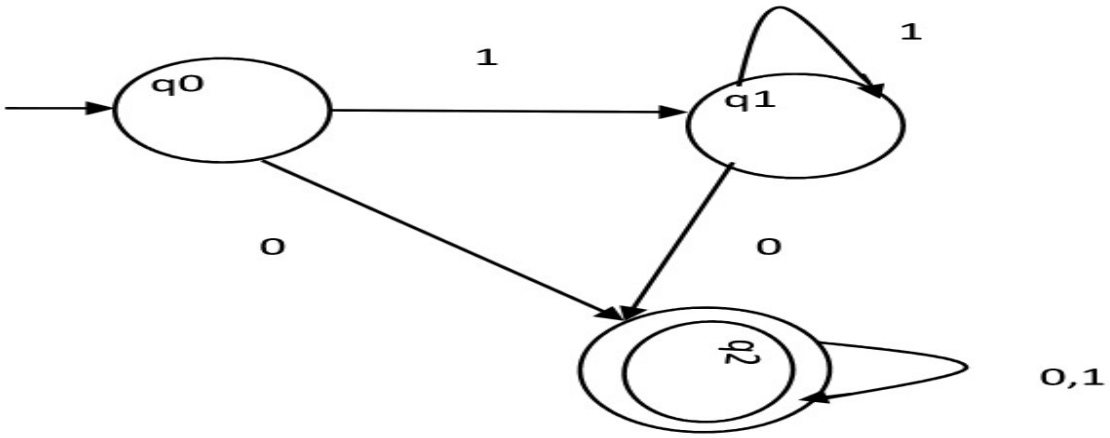
δ (q0, 0) = q3 and δ (q3, 0) = q0
δ (q0, 1) = q1 and δ (q3, 1) = q4
Moves of q0 and q3 on input symbol 0 are q3 and q0 respectively which are in same set in partition P1. Similarly, Moves of q0 and q3 on input symbol 1 are q1 and q4 which are in same set in partition P1. So, q0 and q3 are not distinguishable.

v) For set { q5 }:

Since we have only one state in this set, it can't be further partitioned. So,
P2 = { { q1, q2, q4 }, { q0, q3 }, { q5 } }
Since, P1=P2. So, this is the final partition. Partition P2 means that q1, q2 and q4 states are merged into one. Similarly, q0 and q3 are merged into one. Minimized DFA corresponding to DFA of Figure 1 is shown in Figure 2 as:



Question : Consider the given DFA. Which of the following is false?
1. Complement of L(A) is context-free.
2. L(A) = L ((11 * 0 + 0) (0 + 1)* 0* 1*)
3. For the language accepted by A, A is the minimal DFA.
4. A accepts all strings over { 0, 1 } of length atleast two.



A. 1 and 3 only
B. 2 and 4 only
C. 2 and 3 only
D. 3 and 4 only

Solution : Statement 4 says, it will accept all strings of length atleast 2. But it accepts 0 which is of length 1. So, 4 is false.
Statement 3 says that the DFA is minimal. We will check using the algorithm discussed above.
P0 = { { q2 }, { q0, q1 } }
P1 = { q2 }, { q0, q1 } }. Since, P0 = P1, P1 is the final DFA. q0 and q1 can be merged. So minimal DFA will have two states. Therefore, statement 3 is also false.
So correct option is (D).

This article has been contributed by Sonal Tuteja.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

UNIT-3

Context Free Grammars

The definition of context free grammars (CFGs) allows us to develop a wide variety of grammars. Most of the time, some of the productions of CFGs are not useful and are redundant. This happens because the definition of CFGs does not restrict us from making these redundant productions.

By simplifying CFGs we remove all these redundant productions from a grammar , while keeping the transformed grammar equivalent to the original grammar. Two grammars are called equivalent if they produce the same language. Simplifying CFGs is necessary to later convert them into Normal forms.

Types of redundant productions and the procedure of removing them are mentioned below.

1. Useless productions – The productions that can never take part in derivation of any string , are called useless productions. Similarly , a variable that can never take part in derivation of any string is called a useless variable. For eg.

S -> abS | abA | abB

A -> cd

B -> aB

C -> dc

In the example above , production ‘C -> dc’ is useless because the variable ‘C’ will never occur in derivation of any string. The other productions are written in such a way that variable ‘C’ can never reached from the starting variable ‘S’.

Production ‘B ->aB’ is also useless because there is no way it will ever terminate . If it never terminates , then it can never produce a string. Hence the production can never take part in any derivation.

To remove useless productions , we first find all the variables which will never lead to a terminal string such as variable ‘B’. We then remove all the productions in which variable ‘B’ occurs.

So the modified grammar becomes –

S -> abS | abA

A -> cd

C -> dc

We then try to identify all the variables that can never be reached from the starting variable such as variable ‘C’. We then remove all the productions in which variable ‘C’ occurs.
The grammar below is now free of useless productions –

S -> abS | abA

A -> cd

2. λ productions – The productions of type ‘A -> λ’ are called λ productions (also called lambda productions and null productions) . These productions can only be removed from those grammars that do not generate λ (an empty string). It is possible for a grammar to contain null productions and yet not produce an empty string.

To remove null productions , we first have to find all the nullable variables. A variable ‘A’ is called nullable if λ can be derived from ‘A’. For all the productions of type ‘A -> λ’ , ‘A’ is a nullable variable. For all the productions of type ‘B -> A1A2…An ‘ , where all ’Ai’s are nullable variables , ‘B’ is also a nullable variable.

After finding all the nullable variables, we can now start to construct the null production free grammar. For all the productions in the original grammar , we add the original production as well as all the combinations of the production that can be formed by replacing the nullable variables in the production by λ. If all the variables on the RHS of the production are nullable , then we do not add ‘A -> λ’ to the new grammar. An example will make the point clear.

Consider the grammar –

S -> ABCd (1)

A -> BC (2)

B -> bB | λ (3)

C -> cC | λ (4)

Lets first find all the nullable variables. Variables ‘B’ and ‘C’ are clearly nullable because they contain ‘λ’ on the RHS of their production. Variable ‘A’ is also nullable because in (2) , both variables on the RHS are also nullable. Similarly , variable ‘S’ is also nullable. So variables ‘S’ , ‘A’ , ‘B’ and ‘C’ are nullable variables.

Lets create the new grammar. We start with the first production. Add the first production as it is. Then we create all the possible combinations that can be formed by replacing the nullable variables with λ. Therefore line (1) now becomes ‘S -> ABCd | ABd | ACd | BCd | Ad | Bd |Cd | d ’.We apply the same rule to line (2) but we do not add ‘A -> λ’ even though it is a possible combination. We remove all the productions of type ‘V -> λ’. The new grammar now becomes –

S -> ABCd | ABd | ACd | BCd | Ad | Bd |Cd | d

A -> BC | B | C

B -> bB | b

C -> cC | c

3. Unit productions – The productions of type ‘A -> B’ are called unit productions.
To create a unit production free grammar ‘Guf’ from the original grammar ‘G’ , we follow the procedure mentioned below.

First add all the non-unit productions of ‘G’ in ‘Guf’. Then for each variable ‘A’ in grammar ‘G’ , find all the variables ‘B’ such that ‘A *=> B’. Now , for all variables like ‘A ’ and ‘B’, add ‘A -> x1 | x2 | …xn’ to ‘Guf’ where ‘B -> x1 | x2 | …xn ‘ is in ‘Guf’ . None of the x1 , x2 … xn are single variables because we only added non-unit productions in ‘Guf’. Hence the resultant grammar is unit production free. For eg.

S -> Aa | B

A -> b | B

B -> A | a

Lets add all the non-unit productions of ‘G’ in ‘Guf’. ‘Guf’ now becomes –

S -> Aa

A -> b

B -> a

Now we find all the variables that satisfy ‘X *=> Z’. These are ‘S*=>B’, ‘A *=> B’ and ‘B *=> A’. For ‘A *=> B’ , we add ‘A -> a’ because ‘B ->a’

exists in ‘Guf’. ‘Guf’ now becomes

S -> Aa

A -> b | a

B -> a

For ‘B *=> A’ , we add ‘B -> b’ because ‘A -> b’ exists in ‘Guf’. The new grammar now becomes

S -> Aa

A -> b | a

B -> a | b

We follow the same step for ‘S*=>B’ and finally get the following grammar –

S -> Aa | b | a

A -> b | a

B -> a | b

Now remove B -> a|b , since it doesnt occur in the production ‘S’, then the following grammar becomes,

S->Aa|b|a

A->b|a

Note: To remove all kinds of productions mentioned above, first remove the null productions, then the unit productions and finally , remove the useless productions. Following this order is very important to get the correct result.

This article is contributed by Nitish Joshi. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Ambiguity in Grammars and Languages

Suppose we have a context free grammar G with production rules:

S->aSb|bSa|SS|Ɛ

Left most derivation (LMD) and Derivation Tree: Leftmost derivation of a string from starting symbol S is done by replacing leftmost non-terminal symbol by RHS of corresponding production rule. For example: The leftmost derivation of string abab from grammar G above is done as:

S => a**S**b => ab**S**ab => abab

The symbols underlined are replaced using production rules.

Derivation tree: It tells how string is derived using production rules from S and has been shown in Figure 1.

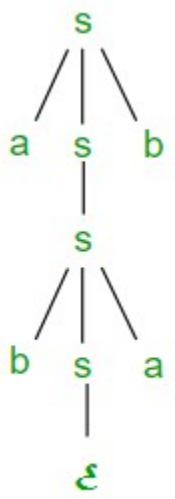
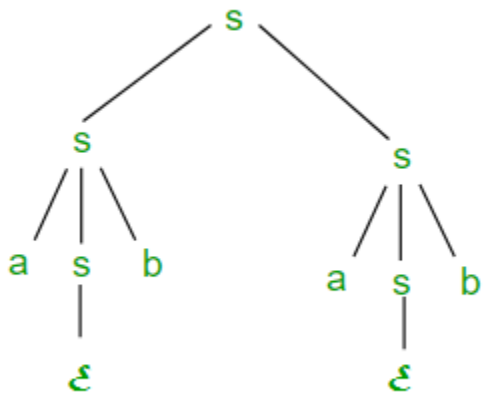


Figure 1

Right most derivation (RMD): Rightmost derivation of a string from starting symbol S is done by replacing rightmost non-terminal symbol by RHS of corresponding production rule. For Example: The rightmost derivation of string abab from grammar G above is done as:
S => S**S** => Sa**S**b => **S**ab => a**S**bab => abab

The symbols underlined are replaced using production rules. The derivation tree for abab using rightmost derivation has been shown in Figure 2.



A derivation can be either LMD or RMD or both or none. For Example:
S => aSb => abSab => abab is LMD as well as RMD

but S => SS => SaSb => Sab => aSbab => abab is RMD but not LMD.

Ambiguous Context Free Grammar: A context free grammar is called ambiguous if there exists more than one LMD or more than one RMD for a string which is generated by grammar. There will also be more than one derivation tree for a string in ambiguous grammar. The grammar described above is ambiguous because there are two derivation trees (Figure 1 and Figure 2). There can be more than one RMD for string abab which are:

S => SS => SaSb => Sab => aSbab => abab

S => aSb => abSab => abab

Ambiguous Context Free Languages: A context free language is called ambiguous if there is no unambiguous grammar to define that language and it is also called inherently ambiguous Context Free Languages.

Note:
If a context free grammar G is ambiguous, language generated by grammar L(G) may or may not be ambiguous

It is not always possible to convert ambiguous CFG to unambiguous CFG. Only some ambiguous CFG can be converted to unambiguous CFG.

There is no algorithm to convert ambiguous CFG to unambiguous CFG.

There always exist a unambiguous CFG corresponding to unambiguous CFL.

Deterministic CFL are always unambiguous.

Normal Form for Context Free Grammar

A context free grammar (CFG) is in Chomsky Normal Form (CNF) if all production rules satisfy one of the following conditions:

A non-terminal generating a terminal (e.g.; X->x)

A non-terminal generating two non-terminals (e.g.; X->YZ)

Start symbol generating ε. (e.g.; S-> ε)

Consider the following grammars,

G1 = {S->a, S->AZ, A->a, Z->z}

G2 = {S->a, S->aZ, Z->a}

The grammar G1 is in CNF as production rules satisfy the rules specified for CNF. However, the grammar G2 is not in CNF as the production rule S->aZ contains terminal followed by non-terminal which does not satisfy the rules specified for CNF.

Note –

For a given grammar, there can be more than one CNF.

CNF produces the same language as generated by CFG.

CNF is used as a preprocessing step for many algorithms for CFG like CYK(membership algo), bottom-up parsers etc.

For generating string w of length ‘n’ requires ‘2n-1’ production or steps in CNF.

Any Context free Grammar that do not have ε in it’s language has an equivalent CNF.

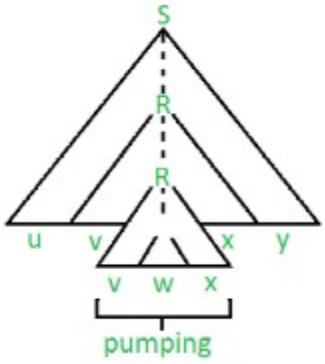
Pumping Lemma for Context-free Languages (CFL)

Pumping Lemma for CFL states that for any Context Free Language L, it is possible to find two substrings that can be ‘pumped’ any number of times and still be in the same language. For any language L, we break its strings into five parts and pump second and fourth substring.

Pumping Lemma, here also, is used as a tool to prove that a language is not CFL. Because, if any one string does not satisfy its conditions, then the language is not CFL.

Thus, if L is a CFL, there exists an integer n, such that for all $x \in L$ with $|x| \geq n$, there exists $u, v, w, x, y \in \Sigma*$, such that $x = uvwxy$, and

(1) |vwx| ≤ n
(2) |vx| ≥ 1
(3) for all i ≥ 0: uviwxiy ∈ L



For above example, 0n1n is CFL, as any string can be the result of pumping at two places, one for 0 and other for 1.
Let us prove, L012 = {0n1n2n | n ≥ 0} is not Context-free.
Let us assume that L is Context-free, then by Pumping Lemma, the above given rules follow.
Now, let x ∈ L and |x| ≥ n. So, by Pumping Lemma, there exists u, v, w, x, y such that (1) – (3) hold.
We show that for all u, v, w, x, y (1) – (3) do not hold.

If (1) and (2) hold then x = 0n1n2n = uvwxy with |vwx| ≤ n and |vx| ≥ 1.
(1) tells us that vwx does not contain both 0 and 2. Thus, either vwx has no 0’s, or vwx has no 2’s. Thus, we have two cases to consider.
Suppose vwx has no 0’s. By (2), vx contains a 1 or a 2. Thus uwy has ‘n’ 0’s and uwy either has less than ‘n’ 1’s or has less than ‘n’ 2’s.
But (3) tells us that uwy = uv0wx0y ∈ L.
So, uwy has an equal number of 0’s, 1’s and 2’s gives us a contradiction.
The case where vwx has no 2’s is similar and also gives us a contradiction.
Thus L is not context-free.

UNIT-IV

Pushdown Automata

A Pushdown Automata (PDA) can be defined as :

- Q is the set of states
- ∑is the set of input symbols
- Γ is the set of pushdown symbols (which can be pushed and popped from stack)
- q0 is the initial state
- Z is the initial pushdown symbol (which is initially present in stack)
- F is the set of final states
- δ is a transition function which maps Q x {Σ ∪ ∈} x Γ into Q x Γ*. In a given state, PDA will read input symbol and stack symbol (top of the stack) and move to a new state and change the symbol of stack.

Instantaneous Description (ID)

Instantaneous Description (ID) is an informal notation of how a PDA "computes" a input string and make a decision that string is accepted or rejected.

A ID is a triple (q, w, α), where:
1. q is the current state.
2. w is the remaining input.
3.α is the stack contents, top at the left.

Turnstile notation

⊢ sign is called a "turnstile notation" and represents one move.
⊢* sign represents a sequence of moves.
Eg- (p, b, T) ⊢ (q, w, α)
This implies that while taking a transition from state p to state q, the input symbol 'b' is consumed, and the top of the stack 'T' is replaced by a new string 'α'

Example : Define the pushdown automata for language $\{a^n b^n \mid n > 0\}$
Solution : M = where Q = { q0, q1 } and Σ = { a, b } and Γ = { A, Z } and δ is given by :

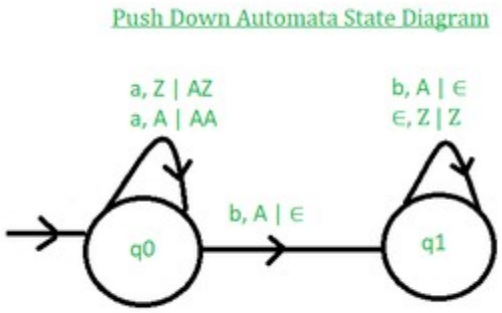
δ(q0, a, Z) = { (q0, AZ) }
δ(q0, a, A) = { (q0, AA) }
δ(q0, b, A) = { (q1, ∈) }
δ(q1, b, A) = { (q1, ∈) }
δ(q1, ∈, Z) = { (q1, ∈) }

Let us see how this automata works for aaabbb.

Row	State	Input	δ(transition function used)	Stack(Leftmost symbol represents top of stack)	State after move
1	q0	aaabbb		Z	q0
2	q0	**a**aabbb	δ(q0,a,Z)={(q0,AZ)}	AZ	q0
3	q0	a**a**abbb	δ(q0,a,A)={(q0,AA)}	AAZ	q0
4	q0	aa**a**bbb	δ(q0,a,A)={(q0,AA)}	AAAZ	q0
5	q0	aaa**b**bb	δ(q0,b,A)={(q1,∈)}	AAZ	q1
6	q1	aaab**b**b	δ(q1,b,A)= {(q1,∈)}	AZ	q1
7	q1	aaabb**b**	δ(q1,b,A)= {(q1,∈)}	Z	q1
8	q1	∈	δ(q1, ∈,Z)= {(q1,∈)}	∈	q1

Explanation : Initially, the state of automata is q0 and symbol on stack is Z and the input is aaabbb as shown in row 1. On reading ‘a’ (shown in bold in row 2), the state will remain q0 and it will push symbol A on stack. On next ‘a’ (shown in row 3), it will push another symbol A on stack. After reading 3 a’s, the stack will be AAAZ with A on the top. After reading ‘b’ (as shown in row 5), it will pop A and move to state q1 and stack will be AAZ. When all b’s are read, the state will be q1 and stack will be Z. In row 8, on input symbol ‘∈’ and Z on stack, it will pop Z and stack will be empty. This type of acceptance is known as **acceptance by empty stack.**

Push Down Automata State Diagram:



Note :

- The above pushdown automaton is deterministic in nature because there is only one move from a state on an input symbol and stack symbol.
- The non-deterministic pushdown automata can have more than one move from a state on an input symbol and stack symbol.
- It is not always possible to convert non-deterministic pushdown automata to deterministic pushdown automata.
- The expressive power of non-deterministic PDA is more as compared to expressive deterministic PDA as some languages are accepted by NPDA but not by deterministic PDA which will be discussed in the next article.
- The pushdown automata can either be implemented using acceptance by empty stack or acceptance by final state and one can be converted to another.

We have discussed Pushdown Automata (PDA) and its acceptance by empty stack article. Now, in this article, we will discuss how PDA can accept a CFL based on the final state. Given a PDA P as:

P = (Q, Σ, Γ, δ, q0, Z, F)

The language accepted by P is the set of all strings consuming which PDA can move from initial state to final state irrespective of any symbol left on the stack which can be depicted as:

L(P) = {w |(q0, w, Z) =>(qf, ε, s)}

Here, from start state q0 and stack symbol Z, the final state qf ε F is reached when input w is consumed. The stack can contain a string s which is irrelevant as the final state is reached and w will be accepted.

Example: Define the pushdown automata for language {a^nb^n | n > 0} using final state.

Solution: M = where Q = {q0, q1, q2, q3} and ∑ = {a, b} and Γ = { A, Z } and F={q3} and δ is given by:

δ(q0, a, Z) = { (q1, AZ) }

δ(q1, a, A) = { (q1, AA) }

δ(q1, b, A) = { (q2, ε) }

δ(q2, b, A) = { (q2, ε) }

δ(q2, ε, Z) = { (q3, Z) }

Let us see how this automaton works for aaabbb:

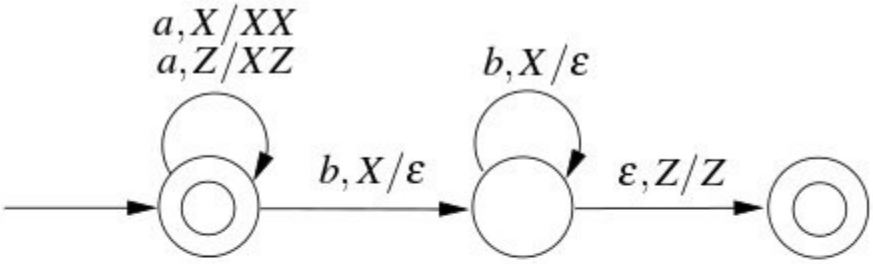
Row	State	Input	δ (transition function) used	Stack (Leftmost symbol represents top of stack)	State after move
0	q0	aaabbb		Z	
1	q0	**a**aabbb	δ(q0,a,Z)={(q1,AZ)}	AZ	q1
2	q1	a**a**abbb	δ(q1,a,A)={(q1,AA)}	AAZ	q1
3	q1	aa**a**bbb	δ(q1,a,A)={(q1,AA)}	AAAZ	q1
4	q1	aaa**b**bb	δ(q1,b,A)={(q2, ε)}	AAZ	q2
5	q2	aaab**b**b	δ(q2,b,A)={(q2,ε)}	AZ	q2
6	q2	aaabb**b**	δ(q2,b,A)={(q2,ε)}	Z	q2
7	q2	ε	δ(q2,ε,Z)={(q3,ε)}	Z	q3

Explanation: Initially, the state of automata is q0 and symbol on the stack is Z and the input is aaabbb as shown in row 0. On reading a (shown in bold in row 1), the state will be changed to q1 and it will push symbol A on the stack. On next a (shown in row 2), it will push another symbol A on the stack and remain in state q1. After reading 3 a's, the stack will be AAAZ with A on the top. After reading b (as shown in row 4), it will pop A and move to state q2 and the stack will be AAZ. When all b's are read, the state will be q2 and the stack will be Z. In row 7, on input symbol ε and Z on the stack, it will move to q3. As the final state q3 has been reached after processing input, the string will be accepted. This type of acceptance is known as *acceptance by the final state*. Next, we will see how this automata works for aab:

Row	State	Input	δ (transition function) used	Stack (Leftmost symbol represents top of stack)	State after move
0	q0	aab		Z	
1	q0	**a**ab	δ(q0,a,Z)={(q1,AZ)}	AZ	q1
2	q1	a**a**b	δ(q1,a,A)={(q1,AA)}	AAZ	q1
3	q1	aa**b**	δ(q1,b,A)={(q2,ε)}	AZ	q2
4	q2	ε		AZ	

As we can see in row 4, the input has been processed and PDA is in state q2 which is a non-final state, the string aab will not be accepted. Let us discuss the question based on this:

Que-1. Consider the transition diagram of a PDA given below with input alphabet ∑ = {a, b}and stack alphabet Γ = {X, Z}. Z is the initial stack symbol. Let L denote the language accepted by the PDA. (GATE-CS-2016)



Which one of the following is **TRUE**?

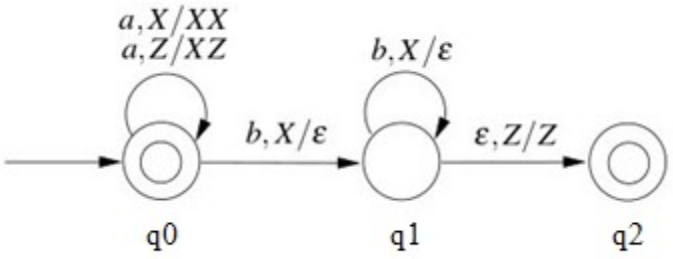
(A) $L = \{a^n b^n | n \geq 0\}$ and is not accepted by any finite automata

(B) $L = \{a^n | n \geq 0\} \cup \{a^n b^n | n \geq 0\}$ and is not accepted by any deterministic PD

(C) L is not accepted by any Turing machine that halts on every input

(D) $L = \{a^n | n \geq 0\} \cup \{a^n b^n | n \geq 0\}$ and is deterministic context-free

Solution: We first label the state of the given PDA as:



Next, the given PDA P can be written as:

Q = {q0, q1, q2} and ∑ = {a, b}

And Γ = {X, Z} and F={q0,q2} and δ is given by :

δ(q0, a, Z) = {(q0, XZ)}

δ(q0, a, X) = {(q0, XX)}

δ(q0, b, X) = {(q1, ε)}

δ(q1, b, X) = {(q1, ε)}

δ(q1, ε, Z) = {(q2, Z)}

As we can see, q0 is the initial as well as the final state, ε will be accepted. For every a, X is pushed onto the stack and PDA remains in the final state. Therefore, any number of a's can be accepted by PDA. If the input contains b, X is popped from the stack for every b. Then PDA is moved to the final state if the stack becomes empty after processing input (δ(q1, ε, Z) = {(q2, Z)}). Therefore, a number of b must be equal to the number of an if they exist. As there is only one move for a given state and input, the PDA is deterministic. So, the correct option is (D).

Construct Pushdown Automata for given languages

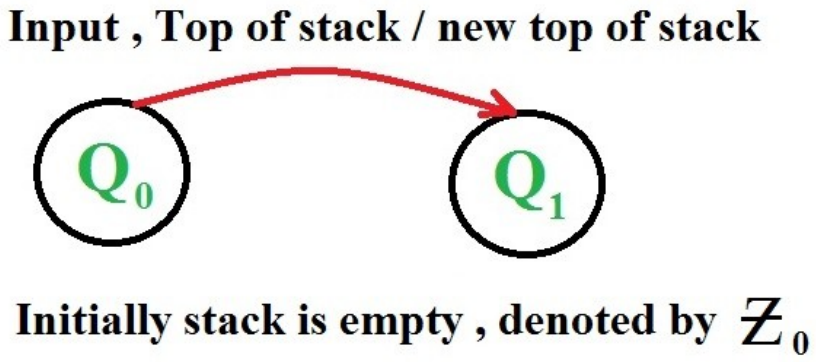
A push down automata is similar to deterministic finite automata except that it has a few more properties than a DFA.The data structure used for implementing a PDA is stack. A PDA has an output associated with every input. All the inputs are either pushed into a stack or just ignored. User can perform the basic push and pop operations on the stack which is use for PDA. One of the problems associated with DFAs was that could not make a count of number of characters which were given input to the machine. This problem is avoided by PDA as it uses a stack which provides us this facility also.

A Pushdown Automata (PDA) can be defined as –

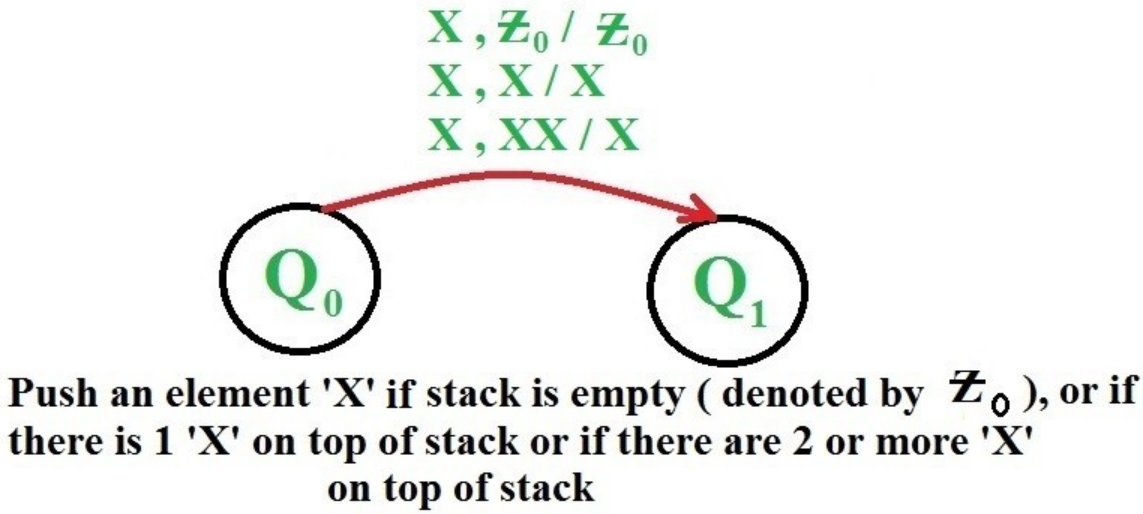
M = (Q, Σ, Γ, δ, q0, Z, F) where

- Q is a finite set of states
- Σ is a finite set which is called the input alphabet
- Γ is a finite set which is called the stack alphabet
- δ is a finite subset of Q X (Σ ∪ {ε} X Γ X Q X Γ^*) the transition relation.
- $q_0 \in Q$ is the start state
- Z ∈ Γ is the initial stack symbol
- F ⊆ Q is the set of accepting states

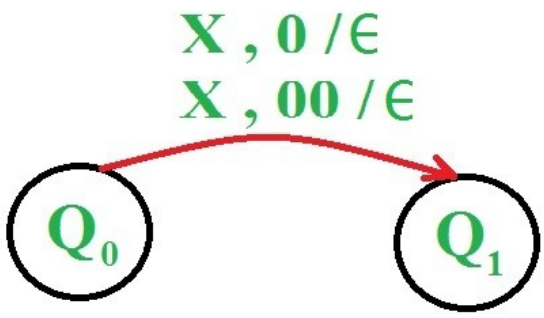
Representation of State Transition –



Representation of Push in a PDA –



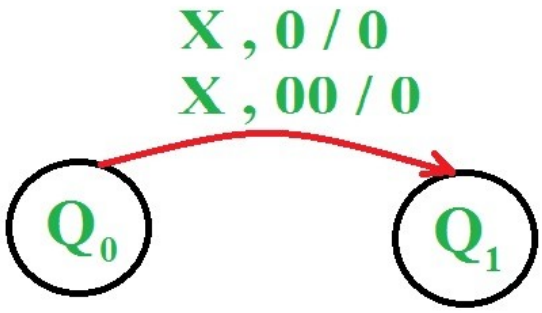
Representation of Pop in a PDA –



Pop an element 'X' if we have 1 or more 0's on top of stack

∈ shows deletion or pop

Representation of Ignore in a PDA –



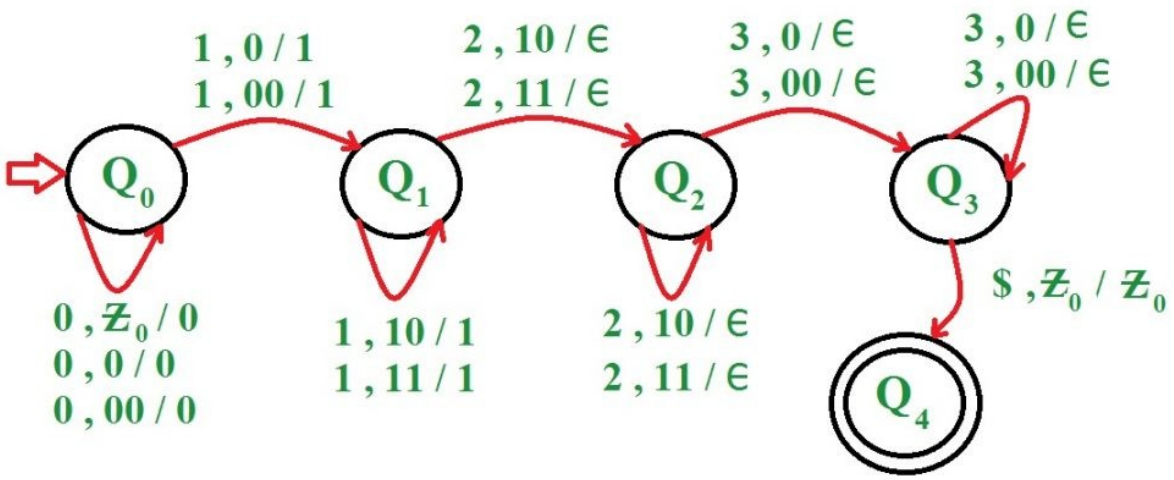
Ignore an element 'X' if we have 1 or more 0's on top of stack

Q) Construct a PDA for language L = {$0^n1^m2^m3^n$ | n>=1, m>=1}

Approach used in this PDA –
First 0's are pushed into stack. Then 1's are pushed into stack.
Then for every 2 as input a 1 is popped out of stack. If some 2's are still left and top of stack is a 0 then string is not accepted by the PDA. Thereafter if 2's are finished and top of stack is a 0 then for every 3 as input equal number of 0's are popped out of stack. If string is finished and stack is empty then string is accepted by the PDA otherwise not accepted.

- **Step-1:** On receiving 0 push it onto stack. On receiving 1, push it onto stack and goto next state
- **Step-2:** On receiving 1 push it onto stack. On receiving 2, pop 1 from stack and goto next state

- **Step-3:** On receiving 2 pop 1 from stack. If all the 1's have been popped out of stack and now receive 3 then pop a 0 from stack and goto next state
- **Step-4:** On receiving 3 pop 0 from stack. If input is finished and stack is empty then goto last state and string is accepted



Examples:

Input : 0 0 1 1 1 2 2 2 3 3

Result : ACCEPTED

Input : 0 0 0 1 1 2 2 2 3 3

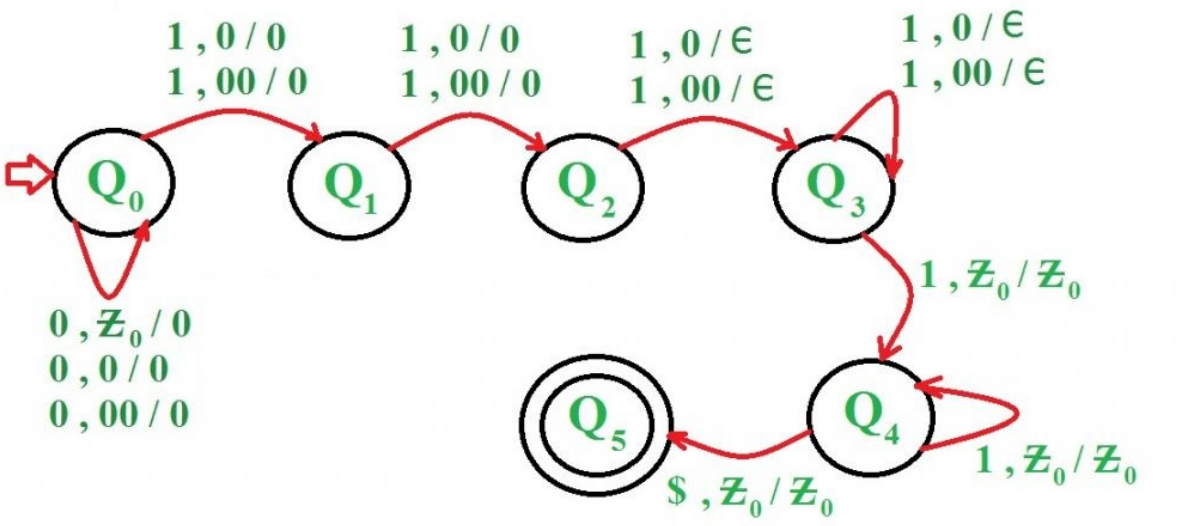
Result : NOT ACCEPTED

Construct a PDA for language L = {0^n1^m | n >= 1, m >= 1, m > n+2}

Approach used in this PDA –
First 0's are pushed into stack.When 0's are finished, two 1's are ignored. Thereafter for every 1 as input a 0 is popped out of stack. When stack is empty and still some 1's are left then all of them are ignored.

- **Step-1:** On receiving 0 push it onto stack. On receiving 1, ignore it and goto next state
- **Step-2:** On receiving 1, ignore it and goto next state
- **Step-3:** On receiving 1, pop a 0 from top of stack and go to next state

- **Step-4:** On receiving 1, pop a 0 from top of stack. If stack is empty, on receiving 1 ignore it and goto next state
- **Step-5:** On receiving 1 ignore it. If input is finished then goto last state



Examples:

Input : 0 0 0 1 1 1 1 1 1

Result : ACCEPTED

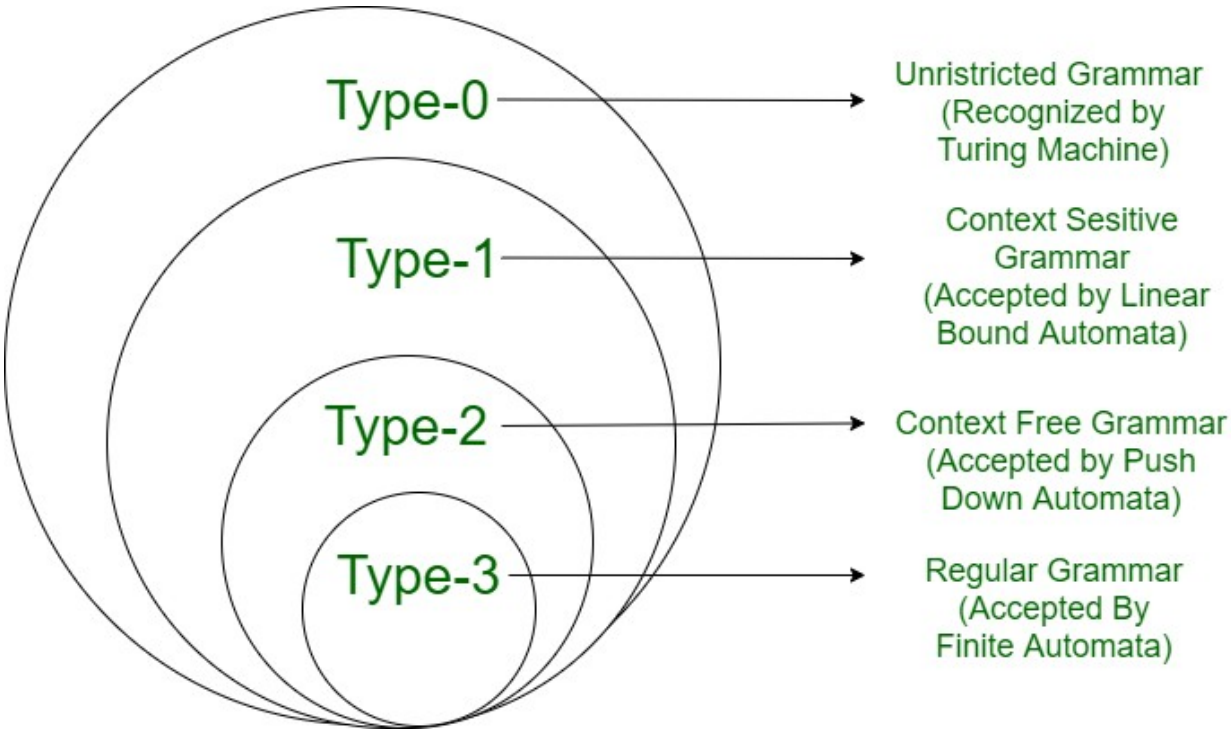
Input : 0 0 0 0 1 1 1 1

Result : NOT ACCEPTED

Chomsky Hierarchy in Theory of Computation

According to Chomsky hierarchy, grammar is divided into 4 types as follows:

1. Type 0 is known as unrestricted grammar.
2. Type 1 is known as context-sensitive grammar.
3. Type 2 is known as a context-free grammar.
4. Type 3 Regular Grammar.



Type 0: Unrestricted Grammar:

Type-0 grammars include all formal grammar. Type 0 grammar languages are recognized by turing machine. These languages are also known as the Recursively Enumerable languages.

Grammar Production in the form

of where

\alpha is (V + T)* V (V + T)*

V : Variables

T : Terminals.

is (V + T)*.

In type 0 there must be at least one variable on the Left side of production.

For example:

Sab --> ba

A --> S

Here, Variables are S, A, and Terminals a, b.

Type 1: Context-Sensitive Grammar

Type-1 grammars generate context-sensitive languages. The language generated by the grammar is recognized by the Linear Bound Automata

In Type 1

- First of all Type 1 grammar should be Type 0.
- Grammar Production in the form of

|\alpha |<=|\beta |

That is the count of symbol in is less than or equal to

Also $\beta \in (V + T)^+$

i.e. β can not be ε

For Example:

S --> AB

AB --> abc

B --> b

Type 2: Context-Free Grammar: Type-2 grammars generate context-free languages. The language generated by the grammar is recognized by a Pushdown automata. In Type 2:

- First of all, it should be Type 1.
- The left-hand side of production can have only one variable and there is no restriction on |\alpha | = 1.

For example:

S --> AB

A --> a

B --> b

Type 3: Regular Grammar: Type-3 grammars generate regular languages. These languages are exactly all languages that can be accepted by a finite-state automaton. Type 3 is the most restricted form of grammar.

Type 3 should be in the given form only :

V --> VT / T (left-regular grammar)

(or)

V --> TV /T (right-regular grammar)

For example:

S --> a

The above form is called strictly regular grammar.

There is another form of regular grammar called extended regular grammar. In this form:

V --> VT* / T*. (extended left-regular grammar)

(or)

V --> T*V /T* (extended right-regular grammar)

For example :

S --> ab.

Please write comments if you find anything incorrect, or if you want to share more information about the topic discussed above.

Undecidability

Decidable Problems
A problem is decidable if we can construct a Turing machine which will halt in finite amount of time for every input and give answer as ‘yes’ or ‘no’. A decidable problem has an algorithm to determine the answer for a given input.

Examples

- **Equivalence of two regular languages:** Given two regular languages, there is an algorithm and Turing machine to decide whether two regular languages are equal or not.
- **Finiteness of regular language:** Given a regular language, there is an algorithm and Turing machine to decide whether regular language is finite or not.
- **Emptiness of context free language:** Given a context free language, there is an algorithm whether CFL is empty or not.

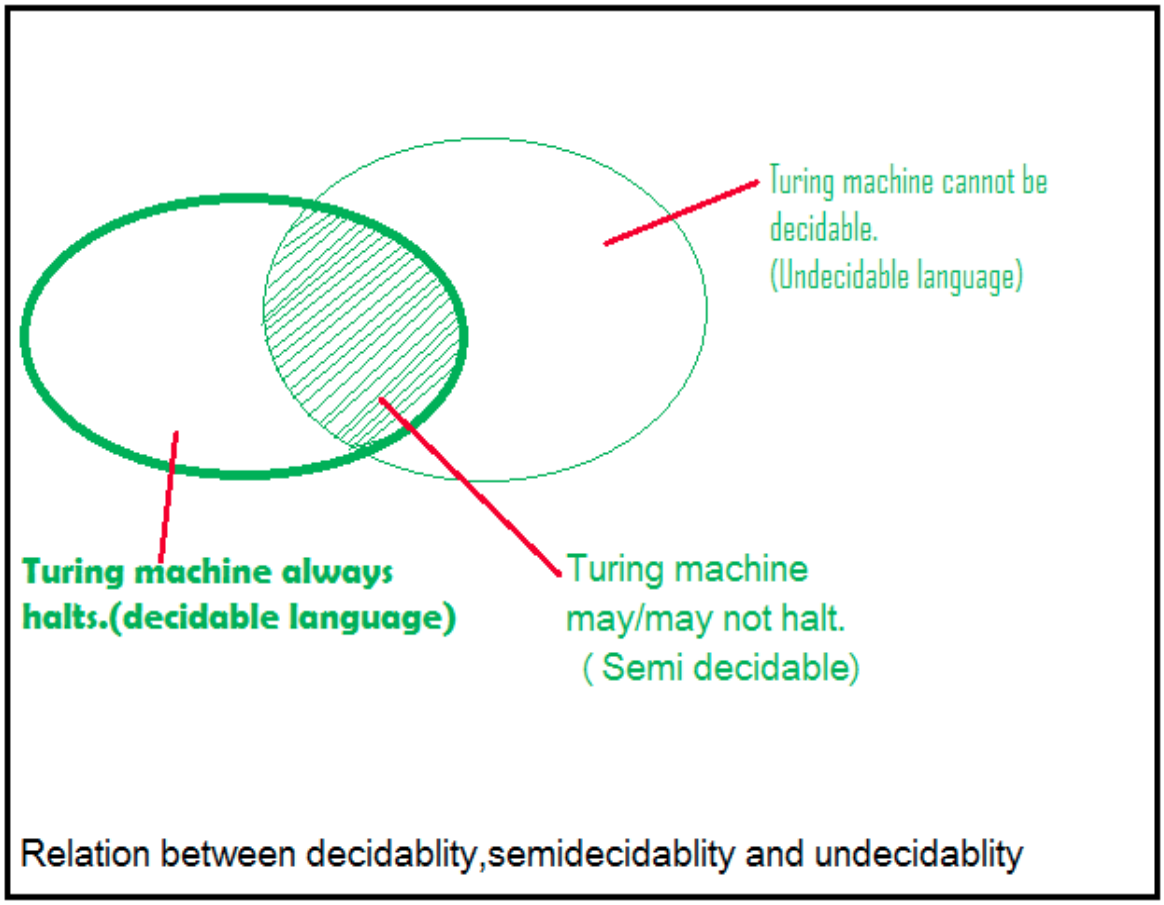
Undecidable Problems
A problem is undecidable if there is no Turing machine which will always halt in finite amount of time to give answer as ‘yes’ or ‘no’. An undecidable problem has no algorithm to determine the answer for a given input.

Examples

- **Ambiguity of context-free languages:** Given a context-free language, there is no Turing machine which will always halt in finite amount of time and give answer whether language is ambiguous or not.

- **Equivalence of two context-free languages:** Given two context-free languages, there is no Turing machine which will always halt in finite amount of time and give answer whether two context free languages are equal or not.
- **Everything or completeness of CFG:** Given a CFG and input alphabet, whether CFG will generate all possible strings of input alphabet (Σ^*)is undecidable.
- **Regularity of CFL, CSL, REC and REC:** Given a CFL, CSL, REC or REC, determining whether this language is regular is undecidable.

Note: Two popular undecidable problems are halting problem of TM and PCP (Post Correspondence Problem). Semi-decidable Problems
A semi-decidable problem is subset of undecidable problems for which Turing machine will always halt in finite amount of time for answer as 'yes' and may or may not halt for answer as 'no'.
Relationship between semi-decidable and decidable problem has been shown in Figure 1 as:



Relation between decidablity,semidecidablity and undecidablity

Rice's Theorem
Every non-trivial (answer is not known) problem on Recursive Enumerable languages is undecidable.e.g.; If a language is Recursive Enumerable, its complement will be recursive enumerable or not is undecidable.

Reducibility and Undecidability

Language A is reducible to language B (represented as A≤B) if there exists a function f which will convert strings in A to strings in B as:

$$w \varepsilon A <=> f(w) \varepsilon B$$

Theorem 1: If A≤B and B is decidable then A is also decidable.
Theorem 2: If A≤B and A is undecidable then B is also undecidable.

Question: Which of the following is/are undecidable?

1. G is a CFG. Is $L(G)=\phi$?
2. G is a CFG. Is $L(G)=\sum^*$?
3. M is a Turing machine. Is L(M) regular?
4. A is a DFA and N is an NFA. Is L(A)=L(N)?

A. 3 only
B. 3 and 4 only
C. 1, 2 and 3 only
D. 2 and 3 only

Explanation:

- Option 1 is whether a CFG is empty or not, this problem is decidable.
- Option 2 is whether a CFG will generate all possible strings (everything or completeness of CFG), this problem is undecidable.
- Option 3 is whether language generated by TM is regular is undecidable.
- Option 4 is whether language generated by DFA and NFA are same is decidable. So option D is correct.

Question: Which of the following problems are decidable?

1. Does a given program ever produce an output?
2. If L is context free language then L' is also context free?
3. If L is regular language then L' is also regular?
4. If L is recursive language then L' is also recursive?

A. 1,2,3,4
B. 1,2
C. 2,3,4
D. 3,4

Explanation:

- As regular and recursive languages are closed under complementation, option 3 and 4 are decidable problems.

- Context free languages are not closed under complementation, option 2 is undecidable.
- Option 1 is also undecidable as there is no TM to determine whether a given program will produce an output. **So, option D is correct.**

Question: Consider three decision problems P1, P2 and P3. It is known that P1 is decidable and P2 is undecidable. Which one of the following is TRUE?

A. P3 is undecidable if P2 is reducible to P3
B. P3 is decidable if P3 is reducible to P2's complement
C. P3 is undecidable if P3 is reducible to P2
D. P3 is decidable if P1 is reducible to P3

Explanation:

- Option A says P2≤P3. According to theorem 2 discussed, if P2 is undecidable then P3 is undecidable. It is given that P2 is undecidable, so P3 will also be undecidable. So option **(A) is correct**.
- Option C says P3≤P2. According to theorem 2 discussed, if P3 is undecidable then P2 is undecidable. But it is not given in question about undecidability of P3. So option **(C) is not correct.**
- Option D says P1≤P3. According to theorem 1 discussed, if P3 is decidable then P1 is also decidable. But it is not given in question about decidability of P3. So option **(D) is not correct**.
- Option (B) says P3≤P2'. According to theorem 2 discussed, if P3 is undecidable then P2' is undecidable. But it is not given in question about undecidability of P3. So option **(B) is not correct**.

Quiz on Undecidability

This article is contributed by **Sonal Tuteja**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

UNIT-5

Turing Machine in TOC

Turing Machine was invented by Alan Turing in 1936 and it is used to accept Recursive Enumerable Languages (generated by Type-0 Grammar).

A turing machine consists of a tape of infinite length on which read and writes operation can be performed. The tape consists of infinite cells on which each cell either contains input symbol or a special symbol called blank. It also consists of a head pointer which points to cell currently being read and it can move in both directions.

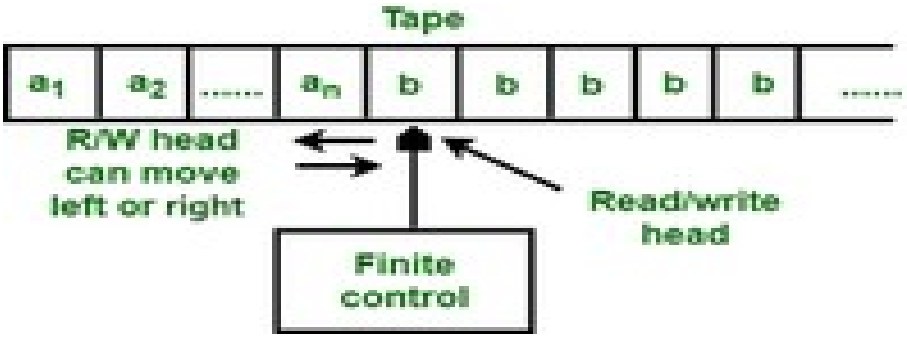


Figure: Turing Machine

A TM is expressed as a 7-tuple (Q, T, B, ∑, δ, q0, F) where:

- **Q** is a finite set of states
- **T** is the tape alphabet (symbols which can be written on Tape)
- **B** is blank symbol (every cell is filled with B except input alphabet initially)
- **∑** is the input alphabet (symbols which are part of input alphabet)
- **δ** is a transition function which maps Q × T → Q × T × {L,R}. Depending on its present state and present tape alphabet (pointed by head pointer), it will move to new state, change the tape symbol (may or may not) and move head pointer to either left or right.
- **q0** is the initial state
- **F** is the set of final states. If any state of F is reached, input string is accepted.

Let us construct a turing machine for L={0^n1^n|n>=1}

- Q = {q0,q1,q2,q3} where q0 is initial state.
- T = {0,1,X,Y,B} where B represents blank.
- ∑ = {0,1}
- F = {q3}

Transition function δ is given in Table 1 as:

	0	1	X	Y	B
q0	(q1,X,R)			(q3,Y,R)	
q1	(q1,0,R)	(q2,Y,L)		(q1,Y,R)	
q2	(q2,0,L)		(q0,X,R)	(q2,Y,L)	
q3				(q3,Y,R)	**Halt**

Illustration

Let us see how this turing machine works for 0011. Initially head points to 0 which is underlined and state is q0 as:

B	0	0	1	1	B

The move will be δ(q0, 0) = (q1, X, R). It means, it will go to state q1, replace 0 by X and head will move to right as:

B	X	0	1	1	B

The move will be δ(q1, 0) = (q1, 0, R) which means it will remain in same state and without changing any symbol, it will move to right as:

The move will be δ(q1, 1) = (q2, Y, L) which means it will move to q2 state and changing 1 to Y, it will move to left as:

Working on it in the same way, the machine will reach state q3 and head will point to B as shown:

Using move δ(q3, B) = halt, it will stop and accepted.

Note:

- In non-deterministic turing machine, there can be more than one possible move for a given state and tape symbol, but non-deterministic TM does not add any power.
- Every non-deterministic TM can be converted into deterministic TM.
- In multi-tape turing machine, there can be more than one tape and corresponding head pointers, but it does not add any power to turing machine.
- Every multi-tape TM can be converted into single tape TM.

Question: A single tape Turing Machine M has two states q0 and q1, of which q0 is the starting state. The tape alphabet of M is {0, 1, B} and its input alphabet is {0, 1}. The symbol B is the blank symbol used to indicate

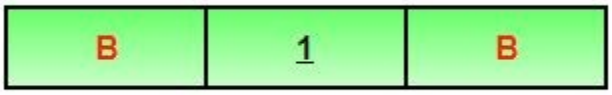
end of an input string. The transition function of M is described in the following table.

	0	1	B
q0	q1,1,R	q1,1,R	Halt
q1	q1,1,R	q0,L,R	q0,B,L

The table is interpreted as illustrated below. The entry (q1, 1, R) in row q0 and column 1 signifies that if M is in state q0 and reads 1 on the current tape square, then it writes 1 on the same tape square, moves its tape head one position to the right and transitions to state q1. Which of the following statements is true about M?

1. M does not halt on any string in (0 + 1)+
2. M does not halt on any string in (00 + 1)*
3. M halts on all string ending in a 0
4. M halts on all string ending in a 1

Solution: Let us see whether machine halts on string '1'. Initially state will be q0, head will point to 1 as:



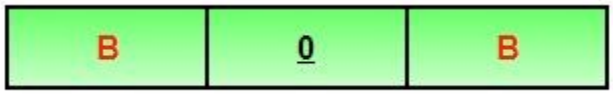
Using δ(q0, 1) = (q1, 1, R), it will move to state q1 and head will move to right as:

Using δ(q1, B) = (q0, B, L), it will move to state q0 and head will move to left as:

It will run in the same way again and again and not halt.

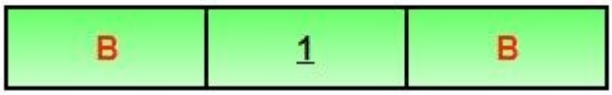
Option D says M halts on all string ending with 1, but it is not halting for 1. So, option D is incorrect.

Let us see whether machine halts on string '0'. Initially state will be q0, head will point to 1 as:



Using δ(q0, 0) = (q1, 1, R), it will move to state q1 and head will move to right as:

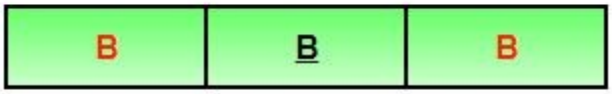
Using δ(q1,B)=(q0,B,L), it will move to state q0 and head will move to left as:



It will run in the same way again and again and not halt.

Option C says M halts on all string ending with 0, but it is not halting for 0. So, option C is incorrect.

Option B says that TM does not halt for any string (00 + 1)*. But NULL string is a part of (00 + 1)* and TM will halt for NULL string. For NULL string, tape will be,



Using δ(q0, B) = halt, TM will halt. As TM is halting for NULL, this option is also incorrect.
So, option (A) is correct.

This article is contributed by **Sonal Tuteja**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Recursive and Recursive Enumerable Languages in TOC

Recursive Enumerable (RE) or Type -0 Language

RE languages or type-0 languages are generated by type-0 grammars. An RE language can be accepted or recognized by Turing machine which means it will enter into final state for the strings of language and may or may not enter into rejecting state for the strings which are not part of the language. It means TM can loop forever for the strings which are not a part of the language. RE languages are also called as Turing recognizable languages.

Recursive Language (REC)

A recursive language (subset of RE) can be decided by Turing machine which means it will enter into final state for the strings of language and rejecting state for the strings which are not part of the language. e.g.; L= $\{a^nb^nc^n|n>=1\}$ is recursive because we can construct a turing machine which will move to final state if the string is of the form $a^nb^nc^n$ else move to non-final state. So the TM will always halt in this case. REC languages are also called as Turing decidable languages.

The relationship between RE and REC languages can be shown in Figure 1.

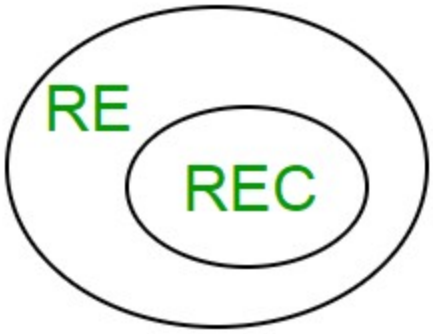


Figure 1

Closure Properties of Recursive Languages

- **Union**: If L1 and If L2 are two recursive languages, their union L1∪L2 will also be recursive because if TM halts for L1 and halts for L2, it will also halt for L1∪L2.
- **Concatenation:** If L1 and If L2 are two recursive languages, their concatenation L1.L2 will also be recursive. For Example:

  L1= $\{a^n b^n c^n | n>=0\}$

  L2= $\{d^m e^m f^m | m>=0\}$

  L3= L1.L2

  = $\{a^n b^n c^n d^m\ e^m f^m | m>=0$ and $n>=0\}$ is also recursive.
- L1 says n no. of a's followed by n no. of b's followed by n no. of c's. L2 says m no. of d's followed by m no. of e's followed by m no. of f's. Their concatenation first matches no. of a's, b's and c's and then matches no. of d's, e's and f's. So it can be decided by TM.
- **Kleene Closure:** If L1is recursive, its kleene closure L1* will also be recursive. For Example:

  L1= $\{a^n b^n c^n | n>=0\}$

  L1*= $\{ a^n b^n c^n || n>=0\}^*$ is also recursive.
- **Intersection and complement**: If L1 and If L2 are two recursive languages, their intersection L1 ∩ L2 will also be recursive. For

Example:

L1= {$a^n b^n c^n dm | n>=0$ and $m>=0$}

L2= {$a^n b^n c^n d^n | n>=0$ and $m>=0$}

L3=L1 ∩ L2

= { $a^n b^n c^n d^n$ |n>=0} will be recursive.

L1 says n no. of a's followed by n no. of b's followed by n no. of c's and then any no. of d's. L2 says any no. of a's followed by n no. of b's followed by n no. of c's followed by n no. of d's. Their intersection says n no. of a's followed by n no. of b's followed by n no. of c's followed by n no. of d's. So it can be decided by turing machine, hence recursive. Similarly, complement of recursive language L1 which is ∑*-L1, will also be recursive.

Note: As opposed to REC languages, RE languages are not closed under complementation which means complement of RE language need not be RE.

Turing Machine Construction

Turing Machines can broadly be classified into two types, the Acceptors and the Transducers. Acceptor Turing Machine is an automaton used to define Turing-acceptable languages. Such a machine can be used to check whether a given string belongs to a language or not. It is defined as a 7-tuple machine.

Coming to Transducers: In general, transducers are the devices used to convert one form of signal into another. The same can be told about Turing Machine Transducers.

A transducer is a type of Turing Machine that is used to convert the given input into the output after the machine performs various read-writes. It doesn't accept or reject an input but performs series of operations to obtain the output right in the same tape and halts when finished.

Few examples of Turing Machine transducers are:

- Turing Machine for addition
- Turing Machine for subtraction
- Turing Machine for multiplication
- Turing Machine for 1's and 2's complement

Implementation:

Now we will be proposing a Java program that was written to simulate the construction and execution performed by turing machine transducers. There are two inputs that must be given while executing it: A .txt file to define the automaton (an example for unary multiplication machine is given after the code), and a string to be entered via the console window which will be the input on tape for the automaton to execute.

The .txt file's path must be given as input. This was done so that the same program can be used for various types of machines instead of hard-coding the automaton. We just need to write a different txt file for generating a different automaton.

Java was chosen specifically because of the OOP structure, using which a class was defined for State, Transition, Machine, etc, to be able to encapsulate various aspects of an entity within an object. For example, a transition is defined as a class whose members are three characters – read, write and shift which store read symbol, write a symbol, and shift direction respectively, along with the index of the next state to which the machine should transition to. The same applies to a State object, which stores a list of possible outgoing transitions.

Variation of Turing Machine

1. Multiple track Turing Machine:

- A k-track Turing machine(for some k>0) has k-tracks and one R/W head that reads and writes all of them one by one.
- A k-track Turing Machine can be simulated by a single track Turing machine

2. Two-way infinite Tape Turing Machine:

- Infinite tape of two-way infinite tape Turing machine is unbounded in both directions left and right.
- Two-way infinite tape Turing machine can be simulated by one-way infinite Turing machine(standard Turing machine).

3. Multi-tape Turing Machine:

- It has multiple tapes and is controlled by a single head.
- The Multi-tape Turing machine is different from k-track Turing machine but expressive power is the same.
- Multi-tape Turing machine can be simulated by single-tape Turing machine.

4. Multi-tape Multi-head Turing Machine:

- The multi-tape Turing machine has multiple tapes and multiple heads

- Each tape is controlled by a separate head
- Multi-Tape Multi-head Turing machine can be simulated by a standard Turing machine.

5. Multi-dimensional Tape Turing Machine:

- It has multi-dimensional tape where the head can move in any direction that is left, right, up or down.
- Multi dimensional tape Turing machine can be simulated by one-dimensional Turing machine

6. Multi-head Turing Machine:

- A multi-head Turing machine contains two or more heads to read the symbols on the same tape.
- In one step all the heads sense the scanned symbols and move or write independently.
- Multi-head Turing machine can be simulated by a single head Turing machine.

7. Non-deterministic Turing Machine:

- A non-deterministic Turing machine has a single, one-way infinite tape.
- For a given state and input symbol has at least one choice to move (finite number of choices for the next move), each choice has several choices of the path that it might follow for a given input string.
- A non-deterministic Turing machine is equivalent to the deterministic Turing machine.

Introduction to Linear Bounded Automata :
A Linear Bounded Automaton (LBA) is similar to Turing Machine with some properties stated below:

- Turing Machine with Non-deterministic logic,
- Turing Machine with Multi-track, and
- Turing Machine with a bounded finite length of the tape.

Tuples Used in LBA :
LBA can be defined with eight tuples (elements that help to design automata) as:

$M = (Q , T , E , q0 , M_L , M_R , S , F)$,

where,

Q -> A finite set of transition states

T -> Tape alphabet

E -> Input alphabet

q0 -> Initial state

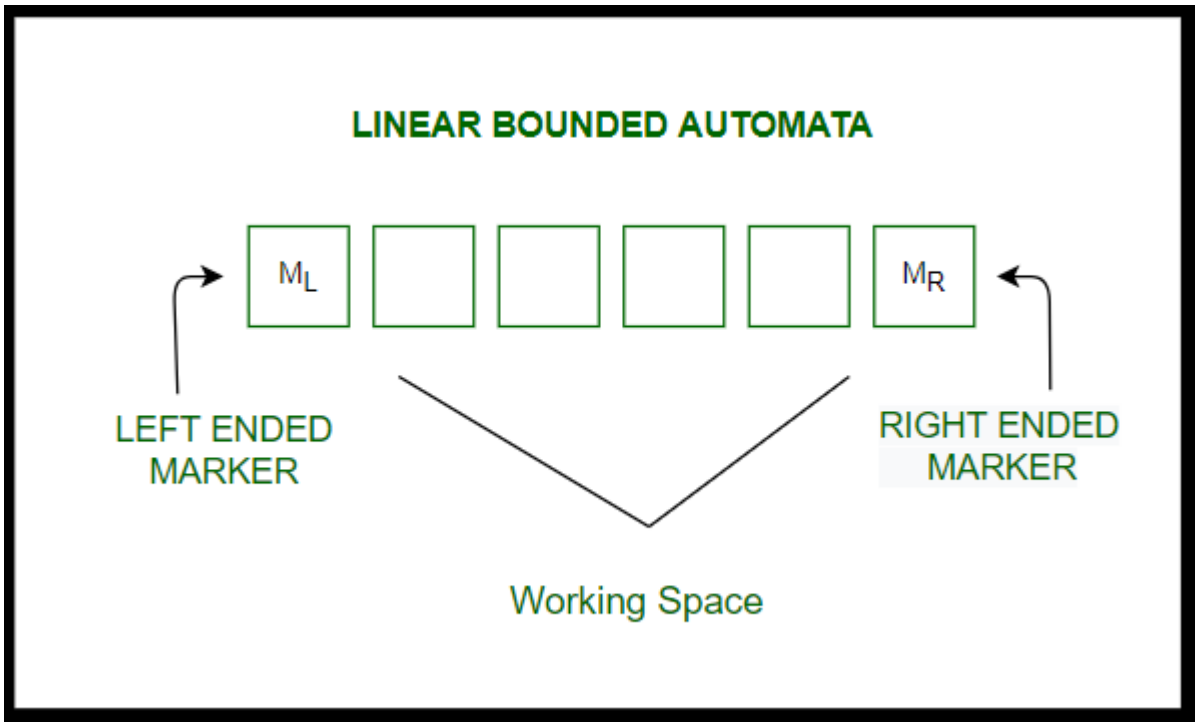
M_L -> Left bound of tape

M_R -> Right bound of tape

S -> Transition Function

F -> A finite set of final states

Diagrammatic Representation of LBA :



Examples:

Languages that form LBA with tape as shown above,

- L = {$a^{n!}$ | n >= 0}
- L = {wn | w from {a, b}+, n >= 1}
- L = {wwwR | w from {a, b}+}

Facts :

Suppose that a given LBA M has

--> q states,

--> m characters within the tape alphabet, and

--> the input length is n

1. Then M can be in at most f(n) = q * n * mn configurations i.e. a tape of n cells and m symbols, we are able to have solely mn totally different tapes.
2. The tape head is typically on any of the n cells which we have a tendency to are typically death penalty in any of the q states.