1. Define the following terms:

1. **Set:** Set is defined as collection of objects. These objects are called elements of the set. All the element are enclosed in a curly brackets '{' and '}' and every element is separated by commas.
2. **Subset:** The set A is called subset of set B if every element of set A is present in set B but reverse is not true.
3. **Complement:** A complement of a set A is the set A' of everything that is not an element of A. This makes sense only in the context of some "Universal Set" U containing all the elements. For Ex :- $A' = \{ x \in U \mid x \notin A \}$
4. **Set Difference**: Considering two sets A and B. The difference A – B is the set of everything in A but not in B. For Eg. :- $A - B = \{ x \mid x \in A \text{ and } x \notin B \}$
5. **Symmetric Difference**: For the two set A and B the symmetric difference can be given as the $A \oplus B$. For Ex :- $A \oplus B = (A - B) \cup (B - A)$
6. **Proposition:** A preposition is a declarative statement that is sufficiently objective, meaningful and precise to have a truth value.
   For Ex:- Fourteen is an even integer.
7. **Conjunction:** The conjunction $p \wedge q$ of p and q is read as "p and q". It is denoted by ∧.

P	Q	P ∧ Q
T	T	T
T	F	F
F	T	F
F	F	F

Table 1.1. Truth table of Conjunction

8. **Disjunction:** The disjunction $p \vee q$ of p and q is read as "p or q". It is denoted by ∨.

P	Q	P ∨ Q
T	T	T
T	F	T
F	T	T
F	F	F

Table 1.2. Truth table of Disjunction

9. **Negation:** The negation ¬p of p is read as the "not p". It is denoted by ¬

p	Q	¬p
T	-	F
F	-	T

Table 1.3. Truth table of Negation

10. **Conditional:** The proposition $p \rightarrow q$ is commonly read as "if p then q".

P	Q	P →Q
T	T	T
T	F	F
F	T	T
F	F	T

Table 1.4. Truth table of conditional

11. **Tautology:** If all the entries of the truth table for any compound proposition are true then the

proposition is termed as tautology.

12. **Onto function / subjective / surjection:** For the function f : A →B, if f(A) = B (the range and codomain of f are equal and every element of the codomain is actually one of the values of the functions), the function f is said to be a onto function.
13. **One to one function / injective / injunction:** For the function f : A →B, we say f is one-to-one if no single element y of B can be f(x) for more than one x in A.
14. **Bijection function:** For the function f: A →B, f is bijection if f is one-to-one as well onto function.
15. **Composite function:** Suppose we have function f : A → B and g : B → C. the function h : A → C defined by h(x) = g (f(x)) is called the composite of g and f.
16. **Inverse function:** For f: A → B, then for any y ϵB there is at least one x ϵ A with $f(x)$ = y and f is onto function and for any y ϵ B there is at most one x ϵA with f(x) = y where f is one-to-one. Therefore for any y ϵB it makes sense to speak of the element x ϵ A for which f(x = y)
    We denote it x by f^{-1} (y). For Eg.:- f : A → B f^{-1} : B → A
17. **Relation:** A relation R on a set A is a subset of A × A.
18. **Equivalence relation:** Assume that R is a relation on a set A; in other words, R⊆ A × A Then
    1. R is reflexive if for every a ∈ A, aRa.
    2. R is symmetric if for every a and b in A if aRb then bRa
    3. R is transitive if for every a, b and c in A if aRb and bRc then aRc.

    R is equivalence on A if R is reflective, symmetric and transitive

2. Prove that $f: R \rightarrow R$, $f(x) = x^2$ is not one-to-one and not onto function.

- The range and codomain of f(x) = x^2 are not equal or every element of codomain is not actually one of the value of function. So function f is not onto function.
- The function is not one to one because every element y of B is f(x) for more than one x in A. EX: f(-1) = f(1) =1.

3. Prove that $f: R \rightarrow R^+$, $f(x) = x^2$ is not one-to-one and onto function.

- The range and codomain of f(x) = x^2 are equal or every element of codomain is actually one of the value of the function. So function f is onto function.
- The function is not one to one because every element y of B is f(x) for more than one x in A. EX: f(-1) = f(1) =1.

4. Prove that $f: R^+ \rightarrow R$, $f(x) = x^2$ is one-to-one and not onto function.

- The range and codomain of f(x) = x^2 are not equal or every element of codomain is not actually one of the value of the function. So function f is not onto function.
- Given function is one to one because no single element of codomain can be f(x) for more than one element in domain.

5. Prove that $f: R^+ \rightarrow R^+$, $f(x) = x^2$ is one-to-one and onto function.

(bijection).

- The range and codomain of $f(x) = x^2$ are equal or every element of codomain is actually one of the value of the function. So function f is onto function.
- Given function is one to one because no single element of codomain can be f(x) for more than one element in domain.
- The function $f(x) = x^2$ is onto function as well as one-to-one function. So, it is called as bijection function.

6. Give quantified statement saying that p is prime.

- Let us consider that statement "P is prime" involving the free variable p over the universe N of natural number.
- We take our definition of a prime number, a number greater than 1 whose only divisor are itself and 1.
- The first statement is now to express the fact that one number is a divisor of another.
- The statement "k is a divisor of p" means that p is a multiple of k or there is an integer m with
  $p = m * k$
- Next saying that the only divisors of p are p and 1 is the same as saying that every divisor of p is either p or 1.
- Adapting the statement that "for every k, if k is a divisor of p, then k is either p or 1".
- Considering all these together we get that "p is prime".$\{ (p > 1) \wedge \forall k (\exists m (p = m * k)) \rightarrow (k = 1) \vee (k = p) \}$

7. What is Proof?

- A proof of a statement is essentially just a convincing argument that the statement is true.
- There are several methods for establishing a proof, some of them are :
  1. Direct Proof
  2. By Contradiction
  3. By Contra positive
  4. By Mathematical Induction

8. For any integers a and b, if a and b are odd, then ab is odd.

Proof:

- An integer n is odd if there exists an integer x so that n=2x+1.
- Now let a and b be any odd integers. Then according to this definition, there is an integer x so that a=2x+1, and there is an integer y so that b=2y+1.
- We wish to show that there is an integer z so that ab=2z+1. Let us therefore calculate ab:
  =(2x+1)(2y+1)
  =4xy+2x+2y+1
  =2(2xy+x+y)+1
- Since we have shown that is a z, namely, 2xy+x+y, so that ab=2z+1, the proof is complete.
- Example:

X=45 & y= 11
xy= 2(2xy+x+y)+1
= 2(2(45)(11)+45+11)+1
= 2(1046)+1
xy=2093
Hence proved.

9. To Prove: For every three positive integers i, j, and n, if i*j = n, then i ≤ $\sqrt{n}$ or j ≤ $\sqrt{n}$.

- The statement we wish to prove is of the general form "for every x, if p(x) then q(x)." For each x, the statement "if p(x) then q(x)" is logically equivalent to "if not p(x) then not q(x)." and therefore the statement we want to prove is equivalent to this: For any positive integer i, j, and n, if it is not the case that i ≤ $\sqrt{n}$ or j ≤ $\sqrt{n}$, then i*j≠n.
- If it is not true that i ≤ $\sqrt{n}$ or j ≤ $\sqrt{n}$, then i >$\sqrt{n}$ and j>$\sqrt{n}$. A generally accepted fact from mathematical is that if a and b are number with a> b, and c is a number >0, then ac>bc.
- Applying this to the inequality i>$\sqrt{n}$ with c=j, we obtain i*j >$\sqrt{n}$ * j. since n>0, we know that $\sqrt{n}$>0, and we may apply the same fact again to the inequality j>$\sqrt{n}$, this time letting c=$\sqrt{n}$, to obtain j$\sqrt{n}$>$\sqrt{n}\sqrt{n}$=n. We now have i*j > j$\sqrt{n}$>n, and it follow that i*j ≠ n.
- Hence, the proof is complete.

10. $\sqrt{2}$ (Root two) is irrational number. (Most IMP)

- Suppose for the sake of contradiction that $\sqrt{2}$ is rational. Then there are integers m′ and n′ with $\sqrt{2}$= m'/ n'.
- By dividing both m' and n' by all the factors that are common to both, we obtain $\sqrt{2}$=m/n, for some integer m and n having no common factors. Since m/n=$\sqrt{2}$, m= n$\sqrt{2}$. Squaring both sides of this equation, we obtain $m^2=2n^2$, and therefore m^2 is even.
- If a and b are odd, then ab is odd. Since a conditional statement is logically equivalent to its contra positive, we may conclude that for any a and b, if ab is not odd, then either a is not odd or b is not odd.
- However, an integer is not odd if and only if it is even, and so for any a and b, if ab is even, m must be even.
- This means that for some k, m=2k. Therefore, $(2k)^2=2n^2$.
- Simplifying this and canceling 2 from both side, we obtain $2k^2=n^2$. Therefore n^2 is even.
- The same argument that we have already used to show that n must be even, and so n=2j for some j.
- We have shown that m and n are both divisible by 2. This contradicts the previous statement that m and n have no common factor. The assumption that $\sqrt{2}$ is rational therefore it leads to a contradiction, and the conclusion is that $\sqrt{2}$ is irrational.

11. Explain principle of Mathematical Induction.

Suppose P(n) is a statement involving an integer n. Then to prove that P(n) is true for every n>=n_0, it is sufficient to show these two things:

1. P (n_0) is true.
2. For every k>= n_0, if P(k) is true, P(k+1) is true.

12. Prove $\sum_{i=1}^{n} i = n(n+1)/2$

Step-1: Basic step

We must show that p(0) is true.

0 = 0(0+1)/2

And, this is obviously true.

Step-2: Induction Hypothesis

k >= 0 and

$1+2+3+4\ldots..+k = \frac{k(k+1)}{2}$

Step-3: Proof of Induction

p(k+1) = 1+2+3+….+k+(k+1)

$= \frac{k(k+1)}{2} + (k+1)$ by induction hypothesis

$= \frac{k(k+1)+2(k+1)}{2}$

$= \frac{(k+1)(k+2)}{2}$

$P(k+1) = \frac{(k+1)((k+1)+1)}{2}$

Hence by principle of mathematical induction $\sum_{i=1}^{n} i = n(n+1)/2$ is true.

13. Prove $\sum_{i=0}^{n} i^2 = n(n+1)(2n+1)/6$

Step-1: Basic

We must show that p(0) is true.

$P(0)=\frac{0(0+1)(2(0)+1)}{6} = 0$

And, this is obviously true.

Step-2: Induction Hypothesis

k >= 0 and

$P(k) = 1+4+9+\ldots.k^2 = \frac{k(k+1)(2k+1)}{6}$

Step-3:Proof of Induction

$P(k+1) = \frac{k(k+1)(2k+1)}{6} + (k+1)^2$

$= \frac{k(k+1)(2k+1)+6(k+1)^2}{6}$

$= \frac{k+1}{6}[k(2k+1)+6(k+1)]$

$= \frac{k+1}{6}[2k^2 + k + 6k + 6]$

$$= \frac{k+1}{6}[2k^2 + 7k + 6]$$
$$= \frac{k+1}{6}[2k^2 + 4k + 3k + 6]$$
$$= \frac{k+1}{6}[2k(k+2) + 3(k+2)]$$
$$= \frac{k+1}{6}[(k+2)(2k+3)]$$
$$= \frac{(k+1)(k+2)(2k+3)}{6}$$
$$= \frac{(k+1)((k+1)+1)(2k+2+1)}{6}$$
$$= \frac{(k+1)((k+1)+1)(2(k+1)+1)}{6}$$

Hence by principle of mathematical induction $\sum_{i=0}^{n} i^2 = n(n+1)(2n+1)/6$ is true.

14. Prove $\sum_{i=1}^{n} \frac{1}{i(i+1)} = \frac{n}{n+1}$

Step-1: Basic

We must show that p(0) is true.

P(0) = $\frac{0}{0+1} = \frac{0}{1}$ =0

$$\frac{1}{i(i+1)} = \frac{1}{0(0+1)} = 0$$

And, this is obviously true.

Step-2: Induction Hypothesis

k >= 0 and

p(k)= $\frac{1}{2} + \frac{1}{6} + \frac{1}{12} + \frac{1}{20} + \cdots + \frac{1}{k(k+1)} = \frac{k}{k+1}$

Step-3: Proof of Induction

p(k+1) = $\frac{1}{2} + \frac{1}{6} + \frac{1}{12} + \frac{1}{20} + \cdots + \frac{1}{k(k+1)} + \frac{1}{(k+1)(k+1+1)}$

$$= \frac{1}{2} + \frac{1}{6} + \frac{1}{12} + \frac{1}{20} + \cdots + \frac{1}{k(k+1)} + \frac{1}{(k+1)(k+2)}$$
$$= \frac{k}{k+1} + \frac{1}{(k+1)(k+2)}$$
$$= \frac{k(k+2)+1}{(k+1)(k+2)}$$
$$= \frac{k((k+1)+1)+1}{(k+1)(k+2)}$$
$$= \frac{(k(k+1)+k+1)}{(k+1)(k+2)}$$
$$= (k^2 + k + k + 1)/(k+1)(k+2)$$

$=(k^2+2k+1)/(k+1)(k+2)$

$=\frac{(k+1)(k+1)}{(k+1)(k+2)}$

$=\frac{k+1}{k+2}$

$=\frac{k+1}{(k+1)+1}$

Hence by principle of mathematical induction $\sum_{i=1}^{n}\frac{1}{i(i+1)}=\frac{n}{n+1}$ is true.

15. Prove 7+ 13+19+…..+(6n+1)= n(3n+4)

Step-1: Basic

We must show that p(0) is true.

P(0)= 0(3(0)+4)=0

And, this is obviously true.

Step-2: Induction Hypothesis

k >= 0 and

p(k) = 7+13+19+…..+(6k+1)=k(3k+4)

Step-3: Proof of Induction

P(k+1) = 7+13+19+…..+(6k+1)+(6(k+1)+1)

= k(3k+4)+(6(k+1)+1)

= k(3k+4)+(6k+6+1)

= $3k^2$+4k+6k+7

= $3k^2$+10k+7

= $3k^2$+3k+7k+7

= 3k(k+1)+7(k+1)

= (k+1)(3k+7)

= (k+1)(3k+3+4)

= (k+1)(3(k+1)+4)

Hence by principle of mathematical induction $7+13+19+\cdots..+(6n+1)=n(3n+4)$ is true.

16. Prove 1+ $\sum_{i=1}^{n} i*i! = (n+1)$

Step-1: Basic

We must show that p(0) is true.

P(0)= (0+1)! =(1)! = 1

And, this is obviously true.

Step-2: Induction Hypothesis

k >= 0 and

p(k) = 1+(1+4+18+…..+(k*k!)) = (k+1)!

Step-3: Proof of Induction

P(k+1) = 1+(1+4+18+…..+(k*k!) + (k+1)*(k+1)!)
= (k+1)! + (k+1)(k+1)!)
= (k+1)! (1+(k+1))
= (k+1)! ((k+1)+1)
= ((k+1) + 1)!

Hence by principle of mathematical induction $1 + \sum_{i=1}^{n} i * i! = (n+1)$ is true.

17. Prove $\sum_{i=1}^{n}(2i-1) = 1+3+5+\cdots+(2n-1) = n^2$

Step-1: Basic

We must show that P(1) is true.

$P(1) = (1)^2 = 1$

And, this is obviously true.

Step-2: Induction Hypothesis

k >= 0 and

$p(k) = 1+3+5+\ldots..+(2k-1)=k^2$

Step-3: Proof of Induction

P(k+1) = 1+3+5+….+(2k-1)+(2(k+1)-1)
= k^2 + (2(k+1)-1)
= k^2 + (2k+2-1)
= k^2 + 2k+1
=$(k+1)^2$

Hence by principle of mathematical induction $\sum_{i=1}^{n}(2i-1) = 1+3+5+\cdots+(2n-1) = n^2$ is true.

18. Prove that $2^n > n^3$ where n>=10, Using Principle of Mathematical Induction.

Step-1: Basic step

We must show that p(10) is true.

$2^{10}=1024$ and $10^3=1000$

So,1024>1000

And, this is obviously true.

Step-2: Induction Hypothesis

For k>=10

$P(k)= 2^k > k^3$

Statement to be shown in Induction step is,

$P(k+1)=2^{k+1} > (k+1)^3$

Step-3: Proof of Induction

$2^{k+1} > (k+1)^3$

Where, $2^{k+1}=(2^k)(2) > 2^k (1.331)$

$> 2^k (1+\frac{1}{10})^3$

$>2^k(1+\frac{1}{k})^3$ for k>=10

$>(\frac{k+1}{k})^3 (2^k)$

$>(\frac{k+1}{k})^3 (k^3)$,because $2^k>k^3$

$(2^k)(2)>(k+1)^3$

$2^{k+1}>(k+1)^3$;k>=10

Hence by principle of mathematical induction $2^n>n^3$ is true.

19. Prove $n(n^2+5)$ is divisible by 6 for n>=0, Using Principle of Mathematical Induction.

Step-1: Basic step

We must show that p(0) is true.

$P(0)=0(0^2+5)=0*6/6=0$

And, this is obviously true.

Step-2: Induction Hypothesis

For k>=0 and

$P(k)=k(k^2+5)$ is divisible by 6.

Statement to be shown in Induction step is,

$P(k+1)=(k+1)[(k+1)^2+5]$ is divisible by 6.

Step-3: Proof of Induction

$(k+1)[(k+1)^2+5]$

$=k[(k+1)^2+5]+1[(k+1)^2+5]$

$=k(k^2+2k+1+5)+(k^2+2k+1+5)$

$=k^3+2k^2+k+5k+(k^2+2k+6)$

$=k(k^2+5)+k(2k+1)+(k^2+2k+6)$

$= k(k^2+5)+2k^2+k+k^2+2k+6$

$= k(k^2+5)+3k^2+3k+6$

$= k(k^2+5)+3k(k+1)+6$

- Here, $k(k^2+5)$ is divisible by 6 ,given in induction hypothesis.
- In Second term k and k+1 are consecutive. So, one number is even and one is odd. So, even number is always multiple of 2 and here 3 is also present .So, second term having (2*3) is also divisible by 6.
- Last term 6 is obviously divisible by 6.Hence proved.

20. Strong Principle of Mathematical Induction

Suppose p(n) is a statement involving on integer n then to prove that p(n) is true for every $n>= n_0$. It is sufficient to show these two condition

1) $P(n_0)$ is true
2) For any $k >=n_0$, if p(n) is true for every n satisfying $n_0<=n<=k$ then p(k+1) is true.

21. Prove that Integer Bigger than 2 have prime factorization using strong PMI

Basic step:

- P(2) is the statement that 2 is either a prime or a product of two or more primes. This is true because 2 is prime.

Induction hypothesis:

- $K \geq 2$, and for every n with $2 \leq n \leq k$,n is either prime or a product of two or more primes.

Statement to be shown in induction step:

- K+1 is either prime or a product of two or more primes.

Proof of induction step:

- We consider two cases. If k+1 is prime, the statement p(k+1) is true. Otherwise, by definition of a prime, k+1 = r*s, for some positive integer r and s, neither of which is 1 or k+1. It follows that $2 \leq r \leq k$ and $2 \leq s \leq k$. therefore, by the induction hypothesis; both r and s are either prime or the product of two or more primes.
- Therefore, their product k+1 is the product of two or more primes, and p(k+1) is true.
- Hence, integer bigger than 2 have prime factorization.

1. Regular language & Regular Expression.

Regular Language :-

A Regular Language over an alphabet ∑ is one that can be obtained from this basic language using the operation of Union, Concatenation and Kleene (*).

Regular Expression :-

- Regular Language can be described by an explicit formula.
- It is common to simplify the formula slightly by leaving out the set brackets {} or replacing them with parenthesis () and replace U by +. The result is called Regular Expression.

	Language	Regular Expression
1.	{^}	^
2.	{0}	0
3.	{001}	001
4.	{0,1}	0+1
5.	{0,10}	0+10
6.	{1,0}110	(1+0)110
7.	{110}*{0,1}	(110)*(0+1)
8.	{1}*{10}	(1)*(10)
9.	{10,111,11010}*	(10+111+11010)*
10.	{0,10}*({11}*U{001,^})	(0+10)*((11)*+(001+^))

Table 2.1. Regular Expression

More Examples of regular Expression

1. 0 or 1
   0+1
2. 0 or 11 or 111
   0+11+111
3. Regular expression over ∑={a,b,c} that represent all string of length 3.
   (a+b+c)(a+b+c)(a+b+c)
4. String having zero or more a.
   a*
5. String having one or more a.
   a^+
6. All binary string.
   (0+1)*
7. 0 or more occurrence of either a or b or both
   (a+b)*
8. 1 or more occurrence of either a or b or both
   $(a+b)^+$
9. Binary no. ends with 0
   (0+1)*0
10. Binary no. ends with 1
    (0+1)*1
11. Binary no. starts and ends with 1.

1(0+1)*1

12. String starts and ends with same character.
    0(0+1)*0 or a(a+b)*a
    1(0+1)*1 b(a+b)*b
13. All string of a and b starting with a
    a(a/b)*
14. String of 0 and 1 ends with 00.
    (0+1)*00
15. String ends with abb.
    (a+b)*abb
16. String starts with 1 and ends with 0.
    1(0+1)*0
17. All binary string with at least 3 characters and 3rd character should be zero.
    (0+1)(0+1)0(0+1)*
18. Language which consist of exactly two b's over the set ∑={a,b}
    a*ba*ba*
19. ∑={a,b} such that 3rd character from right end of the string is always a.
    (a+b)*a(a+b)(a+b)
20. Any no. of a followed by any no. of b followed by any no. of c.
    a*b*c*
21. String should contain at least 3 one.
    (0+1)*1(0+1)*1(0+1)*1(0+1)*
22. String should contain exactly two 1's
    0*10*10*
23. Length of string should be at least 1 and at most 3.
    (0+1) + (0+1) (0+1) + (0+1) (0+1) (0+1)
24. No.of zero should be multiple of 3
    (1*01*01*01*)*
25. ∑={a,b,c} where a should be multiple of 3.
    ((b+c)*a (b+c)*a (b+c)*a (b+c)*)*
26. Even no. of 0.
    (1*01*01*)*
27. Odd no. of 1.
    0*(10*10*)*10*
28. String should have odd length.
    (0+1)((0+1)(0+1))*
29. String should have even length.
    ((0+1)(0+1))*
30. String start with 0 and has odd length.
    0((0+1)(0+1))*
31. String start with 1 and has even length.
    1(0+1)((0+1)(0+1))*
32. Even no of 1

(0*10*10*)*

33. String of length 6 or less
    $(0+1+\wedge)^6$
34. String ending with 1 and not contain 00.
    $(1+01)^+$
35. All string begins or ends with 00 or 11.
    (00+11)(0+1)*+(0+1)*(00+11)
36. Language of all string containing both 11 and 00 as substring.
    ((0+1)*00(0+1)*11(0+1)*)+ ((0+1)*11(0+1)*00(0+1)*)
37. Language of C identifier.
    (_+L)(_+L+D)*

2. Definition of Finite Automata.

A finite Automata or finite state machine is a 5-tuple$(Q,\sum,q_0,A,\delta)$ where,

Q is finite set of states;
$\sum$ is finite alphabet of input symbols;
$q_0 \in Q$ (Initial state);
A (set of accepting states);
δ is a function from $Q\times\sum\rightarrow Q$(Transition function);

For any element q of Q and any symbol $a \in \sum$, we interpret δ (q, a) as the state to which the Finite Automata moves, if it is in state q and receives the input a.

Application of finite automata

A finite automaton is used to solve several common types of computer algorithm. Some of them are:

1. Design of digital circuit.
2. String matching.
3. Communication protocols for information exchange.
4. Lexical analysis phase of a compiler.

3. The Extended transition function δ* for FA.

Let M= $(Q,\sum,q_0,A,\delta)$ be an Finite Automata. We define the function $\delta^*: Q\times\sum^*\rightarrow Q$ as follow:

1) For any $q \in Q$, $\delta^*(q,\wedge)=q$
2) For any $q \in Q$, $y \in \sum^*$, and $a \in \sum$
   $\delta^*(q, ya)= \delta(\delta^*(q, y),a)$

Example:

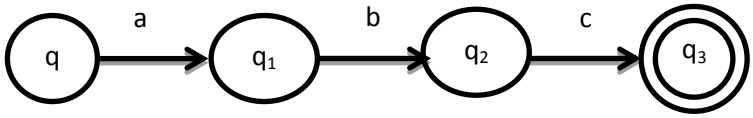


Fig 2.1 Finite Automata

$\delta^*(q,abc)$
$\delta(\delta^*(q,ab),c)$

δ(δ*(δ*(q,a),b),c)
δ(δ(δ* (q,^a),b),c)
δ(δ(δ(δ*(q,^),a),b),c)
δ(δ(δ(q,a),b),c)
δ(δ(q1,b),c)
δ(q_2,c)
q_3

4. Acceptance by an Finite Automata.

Let M= (Q_1, ∑, q_0, A, δ) be an FA. A string x ∈ ∑* is accepted by M if δ*(q_0, x) ∈ A. If string is not accepted, we can say it is rejected by M. The language accepted by M, or the language recognized by M, is the set L(M) = {x ∈ ∑*/x is accepted by M} If L is any Language over ∑, L is accepted or recognized by M if and only if L=L(M).

5. Draw Finite Automata for following:

1. **The string with next to last symbol as 0.**

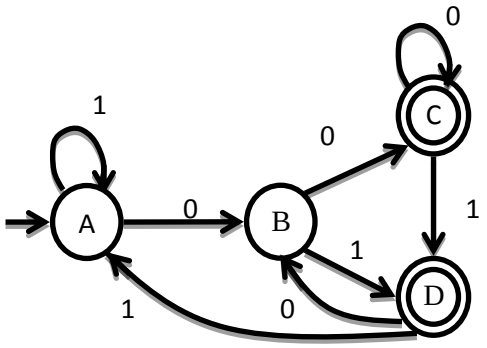


Fig 2.2 Finite Automata for next to last symbol as 0

2. **The string with number of 0s and number of 1s are odd**

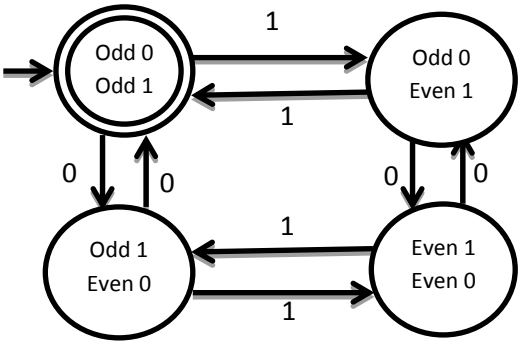


Fig 2.3 Finite Automata for number of 0s and number of 1s are odd

3. **The string ending in 10 or 11.**

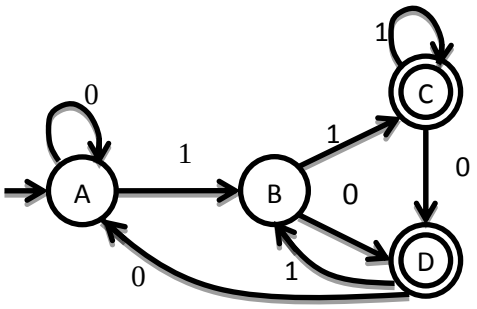


Fig 2.4 Finite Automata for string ending in 10 or 11

4. **The string corresponding to Regular expression {00}*{11}***

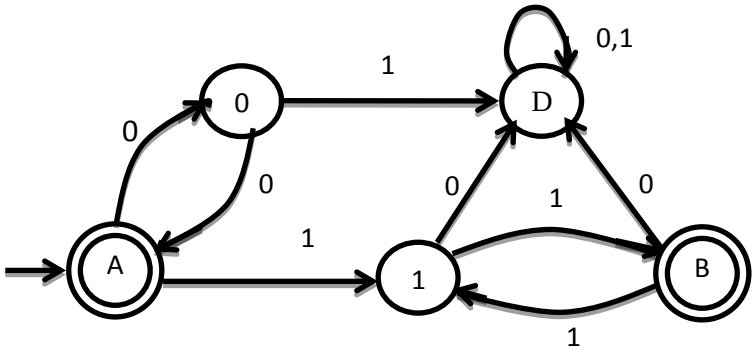


Fig 2.5 Finite Automata for {00}*{11}*

5. **(a+b)*baaa**

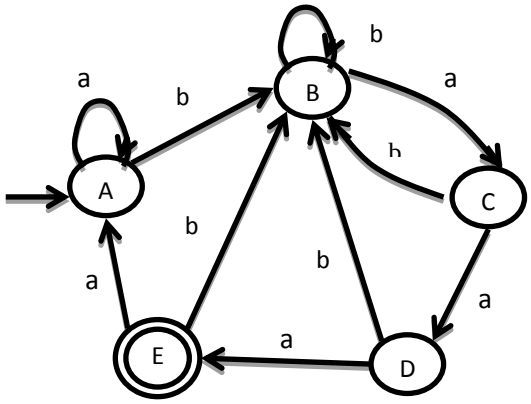


Fig 2.6 Finite Automata for (a+b)*baaa

6. Find a string of minimum length in {0,1}* not in the language corresponding to the regular expression: 1*(0+10)*1*

The smallest string = 0110

7. Define Dead end state.

Dead state are those non accepting states whose transitions for every input symbols terminate

on themselves.

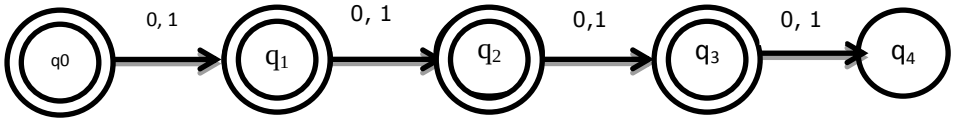


Fig 2.7 Dead end state

Ex: all strings of length at most 3. The string of length >3 should be rejected through a dead state or a failure state. In above example dead state is q_4.

8. Union, Intersection & Compliment operation on Finite Automata

Suppose M1=(Q_1,∑,q_1,A_1,δ_1)

M2=(Q_2,∑,q_2,A_2,δ_2)

Accept languages L1 and L2 respectively. Let M be an Finite Automata defined by

M=(Q,∑,q_0,A,δ) where,

Q=Q1×Q2

q_0=(q_1, q_2) and transition function δ is defined by the formula

δ((p,q),a)=(δ1(p,a), δ2(q,a)) for any p ∈ Q1 and q ∈ Q2 and a ∈ ∑ then

1) if A={(p,q) / p ∈ A1 or q ∈ A2}, M accept the language L1 U L2;
2) if A={(p,q) / p ∈ A1 and q ∈ A2}, M accept the language L1 ∩ L2;
3) if A={(p,q) / p ∈ A1 and q ∉ A2}, M accept the language L1 - L2.

9. Draw Finite Automata for following languages: L1={x/x 00 is not substring of x, x ∈ {0,1}*} L2={x/x ends with 01, x ∈ {0,1}*} Draw finite Automata for L1 U L2, L1∩ L2 and L1-L2.

M1

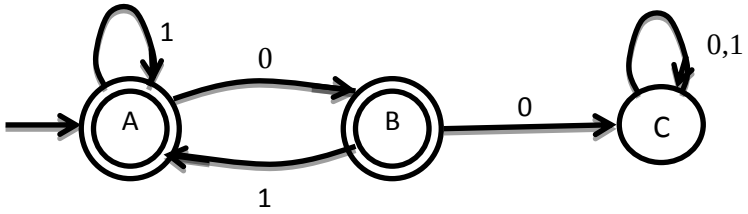


Fig 2.8 Finite Automata for 00 is not substring

M2

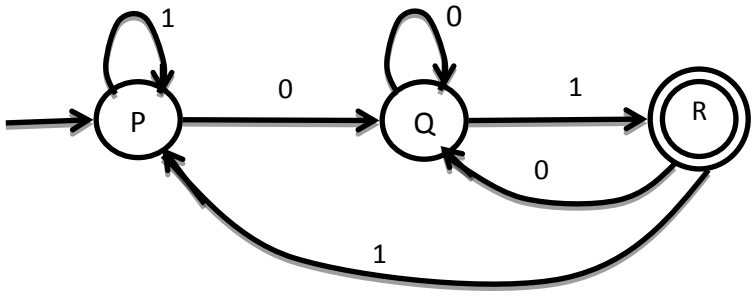


Fig 2.9 Finite Automata for string ending in 01

Here Q1={A,B,C} and Q2={P,Q,R}

So,Q = Q1×Q2 ={AP,AQ,AR,BP,BQ,BR,CP,CQ,CR}

$q_0=(q_1,q_2)=(A,P)$

δ((A,P),0) = (δ(A,0), δ(P,0))
= BQ

δ((A,P),1) = (δ(A,1), δ(P,1))
= AP

δ((B,Q),0) = (δ(B,0), δ(Q,0))
= CQ

δ((B,Q),1) = (δ(B,1), δ(Q,1))
= AR

δ((C,Q),0) = (δ(C,0), δ(Q,0))
= CQ

δ((C,Q),1) = (δ(C,1), δ(Q,1))
= CR

δ((A,R),0) = (δ(A,0), δ(R,0))
= BQ

δ((A,R),1) = (δ(A,1), δ(R,1))
= AP

δ((C,R),0) = (δ(C,0), δ(R,0))
= CQ

δ((C,R),1) = (δ(C,1), δ(R,1))
= CP

δ((C,P),0) = (δ(C,0), δ(P,0))
= CQ

δ((C,P),1) = (δ(C,1), δ(P,1))
= CP

L1-L2

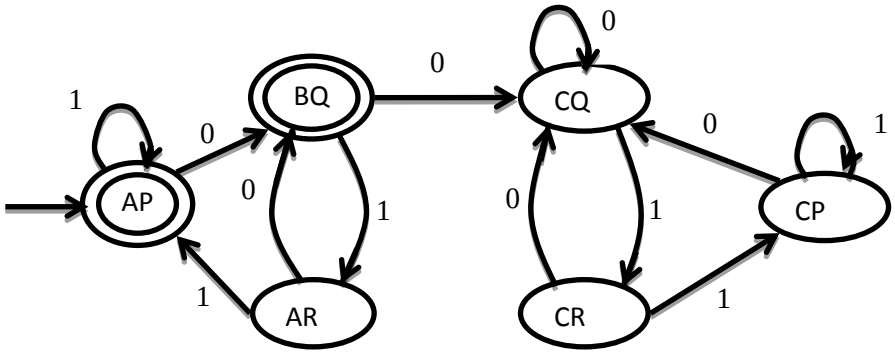


Fig 2.10 Finite Automata for L1-L2

L1 U L2

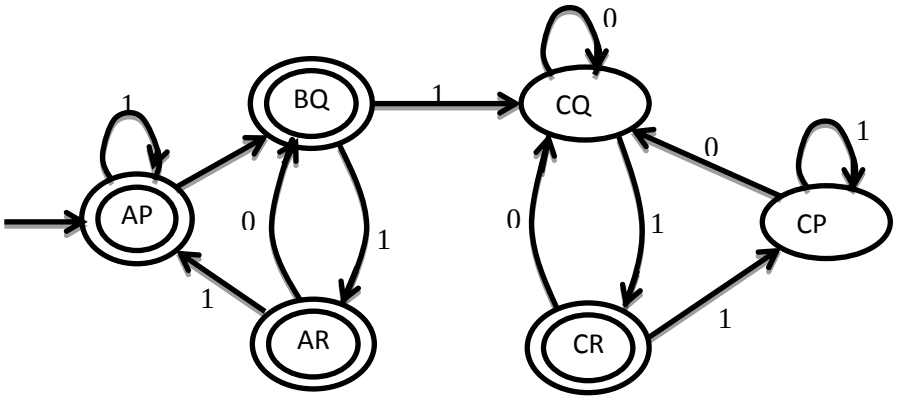


Fig. 2.11 Finite Automata for L1 U L2

L1∩ L2

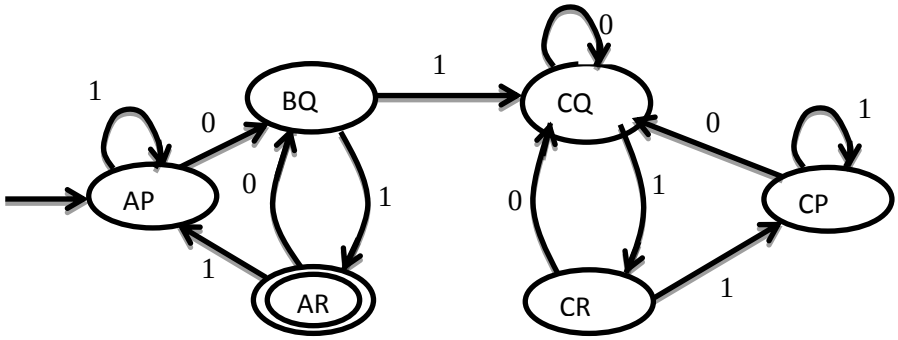


Fig. 2.12 Finite Automata for L1∩ L2

10. Draw Finite Automata for following languages: L1={x/x 11 is not substring of x, x ∈ {0,1}*} L2={x/x ends with 10, x ∈ {0,1}*} Draw finite Automata for L1∩ L2 and L1-L2. (Most IMP)

M1

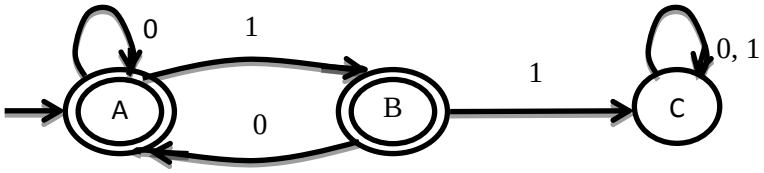


Fig. 2.13 Finite Automata for 11 is not substring

M2

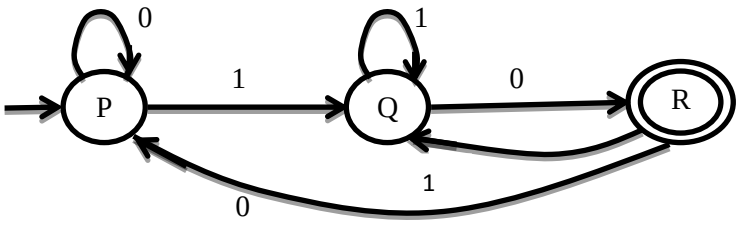


Fig. 2.14 Finite Automata for string ending in 10

Here Q1={A,B,C}, Q2={P,Q,R}

So, Q = Q1×Q2 ={AP,AQ,AR,BP,BQ,BR,CP,CQ,CR}

$q_0=(q_1,q_2)$

q_0=(A,P)

δ((A,P),0) = (δ(A,0), δ(P,0))
= AP

δ((A,P),1) = (δ(A,1), δ(P,1))
= BQ

δ((B,Q),0) = (δ(B,0), δ(Q,0))
= AR

δ((B,Q),1) = (δ(B,1), δ(Q,1))
= CQ

δ((A,R),0) = (δ(A,0), δ(R,0))
= AP

δ((A,R),1) = (δ(A,1), δ(R,1))
= BQ

δ((C,Q),0) = (δ(C,0), δ(Q,0))
= CR

δ((C,Q),1) = (δ(C,1), δ(Q,1))
= CQ

δ((C,R),0) = (δ(C,0), δ(R,0))
= CP

δ((C,R),1) = (δ(C,1), δ(R,1))
= CQ

δ((C,P),0) = (δ(C,0), δ(P,0))
= CP

δ((C,P),1) = (δ(C,1), δ(P,1))
= CQ

L1 - L2

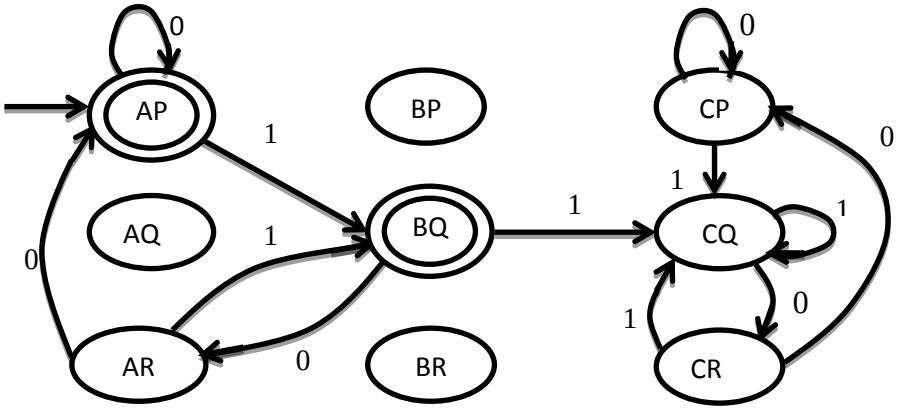


Fig. 2.15 Finite Automata for L1-L2

L1 ∩ L2

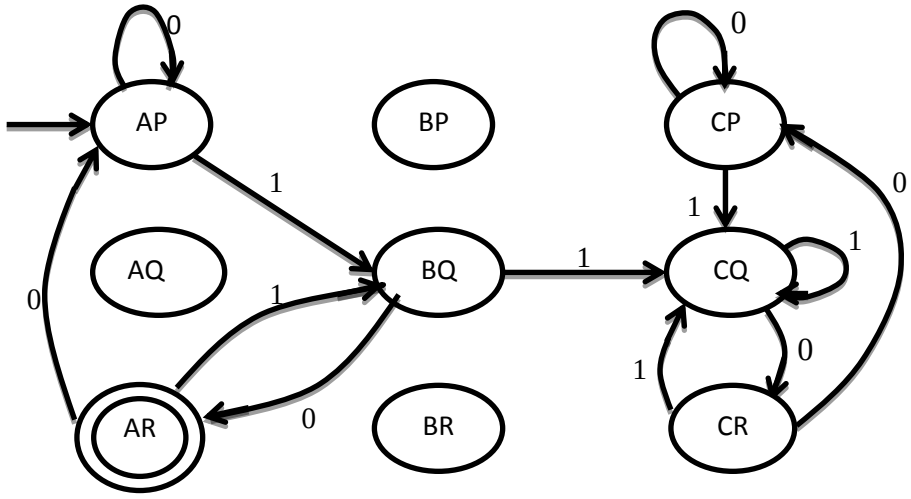


Fig. 2.16 Finite Automata for L1∩L2

11. Draw the Finite Automata recognizing the following language

- **L1 ∩ L2**
- **L1 - L2**

L1

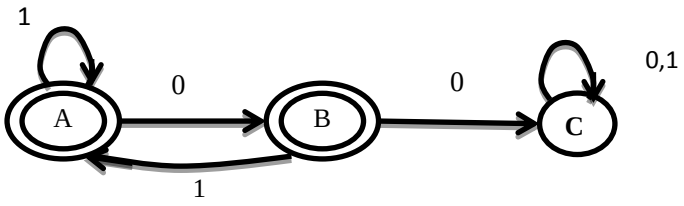


Fig. 2.17 Finite Automata

L2

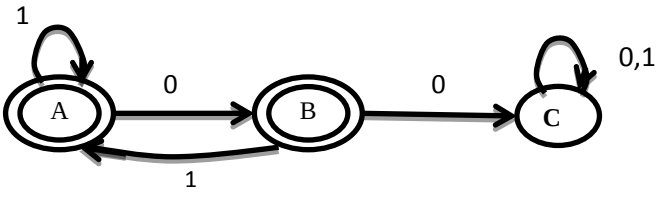


Fig. 2.18 Finite Automata

Q1={A,B,C},Q2={A,B,C}

Q=Q1×Q2= {AA,AB,AC,BA,BB,BC,CA,CB,CC}

$q_0=(q_1,q_2)$ $q_0=(A,A)$

δ((A,A),0) = (δ(A,0), δ(A,0))= BB
δ((A,A),1) = (δ(A,1), δ(A,1))= AA
δ((B,B),0) = (δ(B,0), δ(B,0))= CC
δ((B,B),1) = (δ(B,1), δ(B,1))= AA
δ((C,C),0) = (δ(C,0), δ(C,0))= CC
δ((C,C),1) = (δ(C,1), δ(C,1))= CC

L1 - L2

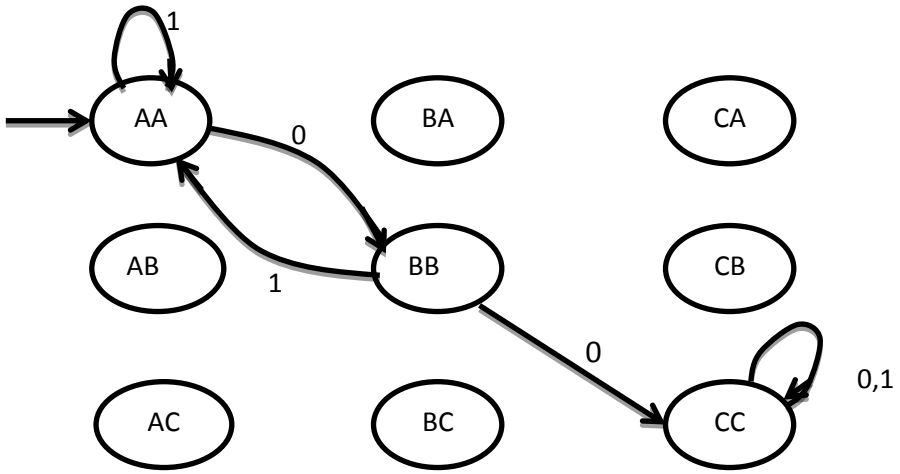


Fig. 2.19 Finite Automata for L1-L2

L1 ∩ L2

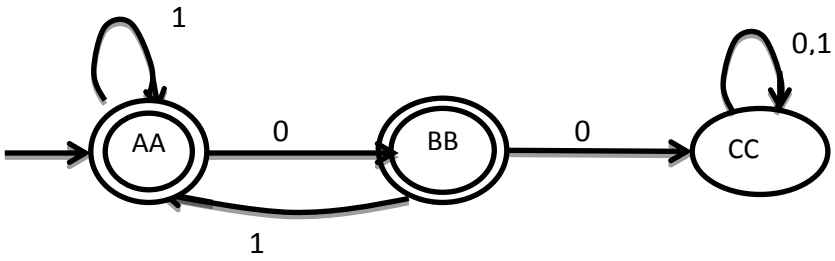


Fig. 2.20 Finite Automata for L1∩L2

12. Definition: Nondeterministic Finite Automata.

A nondeterministic finite automaton (NFA) is a 5-tuple (Q,Σ,q_0,A,δ) Where Q and Σ are nonempty finite sets, $q_0 \in Q$, $A \subseteq Q$, and $\delta : Q \times \Sigma \rightarrow 2^Q$

Q is a finite set of states;
Σ is a finite set of input alphabets;
$q_0 \in Q$ is the initial state;
$A \subseteq Q$ is the set of accepting states;
$\delta : Q \times (\Sigma \cup \{\Lambda\}) \rightarrow 2^Q$ is the transition function.

13. Definition: Non recursive Definition of δ* for an NFA.

For an NFA M=(Q,Σ, q_0,A, δ), and any p ∈ Q, δ*(p, Λ)= {p}. For any p ∈ Q and x= $a_1a_2….a_n$ ∈ Σ* (with n≥1), δ*(p ,x) is the set of all states q for which there is a sequence of states p=p_0,p_1,….p_{n-1},p_n = q satisfying

$P_i \in \delta(p_{i-1}, a_i)$ for each i with $1 \le i \le n$

14. Definition: Recursive Definition of δ* for an NFA

Let M=(Q,Σ, q0,A, δ), be an NFA. The function $\delta^*: Q \times \Sigma^* \rightarrow 2^Q$ is defined as follows.

1) For any q ∈ Q, δ*(q, Λ) = {q}.
2) For any q ∈ Q, y ∈Σ*, and a ∈Σ,
   $\delta^*(q, ya) = \bigcup_{r \in \delta^*(q, y)} \delta(r, a)$

15. Definition: Acceptance by an NFA

Let M= (Q,Σ,q_0,A,δ) be an NFA. The string x ∈Σ* is accepted by M if $\delta^*(q_0, x) \cap A \neq \emptyset$. The language recognized, or accepted, by M is the set L(M) of all string accepted by M. For any language L ⊆ Σ*, L is recognized by M if L= L(M).

16. Using the Recursive Definition of δ* in an NFA.

Let M=(Q,Σ, q_0,A, δ), where Q= {q_0, q_1, q_2, q_3}, Σ={0,1}, A= {q_3}, and δ is given by the following table:

q	δ(q,0)	δ(q,1)
q_0	{ q_0}	{ q_0, q_1}
q_1	{ q_2}	{ q_2}
q_2	{ q_3}	{ q_3}
q_3	∅	∅

Table 2.2 Transition Table

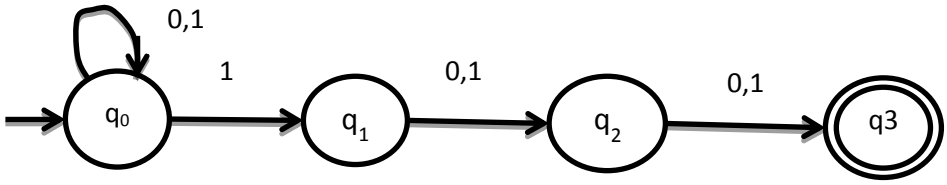


Fig. 2.21 NFA

Let us try to determine L(M) by calculating δ*(q_0,x) for a few string x of increasing length. First observe that form the non recursive definition of δ* it is almost obvious that δ and δ* agree for string of length 1. We see from the table that δ*(q_0,0)={ q_0} and δ*(q_0,1)={ q_0, q_1};

$\delta^*(q_0,11) = \bigcup_{r} \delta(r,1)$

$r \in \delta^*\{q_0,1\}$

$= \bigcup_{r \in \{q_0, q1\}} \delta(r,1)$

$= \delta\{q_0,1\} \cup \delta(q_1,1)$

$= \{q_0, q_1\} \cup \{q_2\}$

$= \{q_0, q_1, q_2\}$

$\delta^*(q_0,01) = \bigcup_{r \in \delta^*(q_0,0)} \delta(r,1)$

$= \bigcup_{r \in (q0)} \delta(r,1)$

$= \delta(q_0,1)$

$= \{q_0, q_1\}$

$\delta^*(q_0,111) = \bigcup_{r \in \delta^*(q_0,11)} \delta(r,1)$

$= \bigcup_{r \in (q_0, q_1, q_2)} \delta(r,1)$

$= \delta(q_0,1) \cup \delta(q_1,1) \cup \delta(q_2,1)$

$= \{q_0, q_1, q_2, q_3\}$

$\delta^*(q_0,011) = \bigcup_{r \in \delta^*(q_0,01)} \delta(r,1)$

$= \bigcup_{r \in (q_0, q_1)} \delta(r,1)$

$= \delta(q_0,1) \cup \delta(q_1,1)$

$= \{q_0, q_1, q_2\}$

17. Definition: A Nondeterministic Finite Automaton with Λ - Transition.

A nondeterministic finite automata with Λ - Transition is a 5-tuple (Q,Σ, q0,A, δ), where Q and Σ are finite sets, $q_0 \in Q$, $A \subseteq Q$, and $\delta : Q \times (\Sigma \cup \{\Lambda\}) \rightarrow 2^Q$

18. Definition: Nondeterministic Definition of δ* for an NFA- Λ.

For an NFA- Λ M= (Q,Σ,q_0,A, δ), states p,q∈ Q, and a string $x = a_1a_2\ldots a_n \in \Sigma^*$, we will say M moves from p to q by a sequence of transition corresponding to x if there exist an integer $m \geq n$, a sequence $b_1b_2\ldots b_m \in \Sigma \cup \{\Lambda\}$ satisfying $b_1b_2\ldots b_m = x$, and a sequence of states $p = p_0, p_1 \ldots p_m = q$ so that for each i with $1 \leq i \leq m$, $p_i \in (p_{i-1}, b_i)$.

For $x \in \Sigma^*$ and $p \in \delta^*(p, x)$ is the set of all states $q \in Q$ such that there is a sequence of transition corresponding to x by which M moves from p to q.

19. Definition: Λ- Closure of a Set of States.

Let $M=(Q,\Sigma, q_0,A, \delta)$, be an NFA – Λ, and let S ne any subset of Q. The Λ- closure of S is the set Λ(s) defined as follows.

1) Every element of S is an element of Λ(s);
2) For any $q \in \Lambda(s)$, every element of $\delta(q, \Lambda)$ is in Λ(s);
3) No other elements of Q are in Λ(s);

20. Definition: Recursive Definition of δ* for an NFA- Λ.

Let $M= (Q, \Sigma, q_0, A, \delta)$ be an NFA – Λ, The extended transition function $\delta^*: Q \times \Sigma^* \rightarrow 2^Q$ is defined as follows.

1) For any $q \in Q$, $\delta^*(q, \Lambda) = \Lambda(\{q\})$.
2) For any $q \in Q$, $y \in \Sigma^*$, and $a \in \Lambda$,

$$\delta^*(q,ya)= \Lambda\left(\bigcup_{r \in \delta^*(q,y)} \delta(r,a)\right)$$

A string x is accepted by M if $\delta^*(q_0, x) \cap A \neq \emptyset$. The language recognized by M is the set L(M) of all strings accepted by M.

21. Compare FA, NFA and NFA-^.

Parameter	NFA	FA	NFA-^
Transition	Non deterministic	Deterministic	Non deterministic
Power	NFA is as powerful as DFA	FA is powerful as an NFA	It's powerful as FA
Design	Easy to design due to non-determinism	More difficult to design	Allow flexibility in handling NFA problem.
Acceptance	It is difficult to find whether w ∈ L as there are several paths. Backtracking is required to explore several parallel paths.	It is easy to find whether w ∈ L as transition are deterministic.	^-transition is useful in constructing a composite FA with respect to union, concatenation, and star.

Table. 2.3 Difference between FA, NFA, NFA-^

22. Applying Definitions of ^(S) and δ*

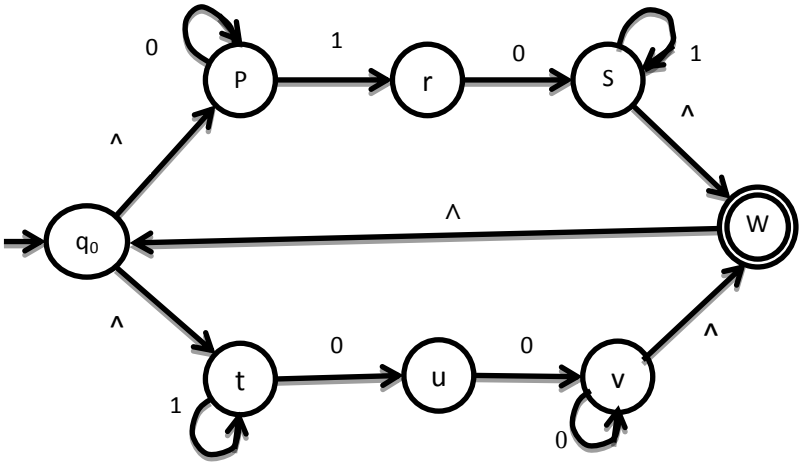


Fig. 2.22 NFA-^

$\delta^*(q_0,^) = ^(\{q_0\})$
$= \{ q_0,p,t\}$

$\delta^*(q_0,0) = ^ (\quad U \quad \delta(r,0))$
$r \in \delta^*(q0,^)$
$= ^ (\delta(q_0,0) U \delta(p,0) U \delta(t,0))$
$= ^ (Ø U \{p\} U \{u\})$
$= ^ (\{p,u\})$
$= \{p,u\}$

$\delta^*(q_0,01) = ^ (\quad U \quad \delta(p,1))$
$r \in \delta^*(q0,0)$
$= ^ (\delta(p,1) U \delta(u,1))$
$= ^ (\{r\})$
$= \{r\}$

$\delta^*(q_0,010) = ^ (\quad U \quad \delta(p,0))$
$r \in \delta^*(q0,01)$
$= ^ (\delta(r,0)$
$= ^(\{s\})$
$= \{s,w,q_0,p,t\}$

23. Conversion from NFA to FA.

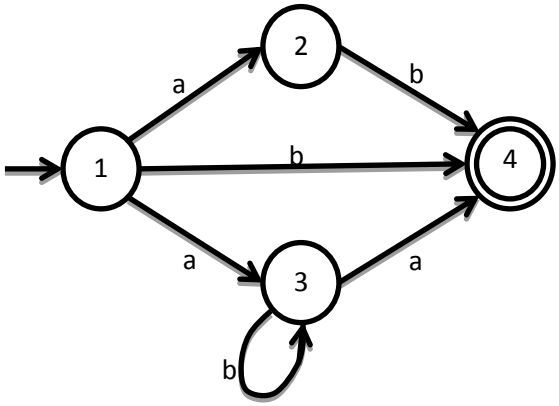


Fig. 2.23 NFA

δ*(1,a) = {2,3}
δ*(1,b) = {4}
δ*({2,3},a) = δ(2,a) U δ(3,a)
= {4}
δ*({2,3},b) = δ(2,b) U δ(3,b)
= {3,4}
δ*(4,a) = {Ø},
δ*(4,b) = {Ø}
δ*({3,4},a) = δ(3,a) U δ(4,a)
= {4}
δ*({3,4},b) = δ(3,b) U δ(4,b)
= {3}
δ*(3,a) = {4}
δ*(3,b) = {3}

q	δ(q,a)	δ(q,b)
1	{2,3}	{4}
2	{ Ø }	{4}
3	{4}	{3}
4	{Ø}	{Ø}

Table. 2.4 Transition Table

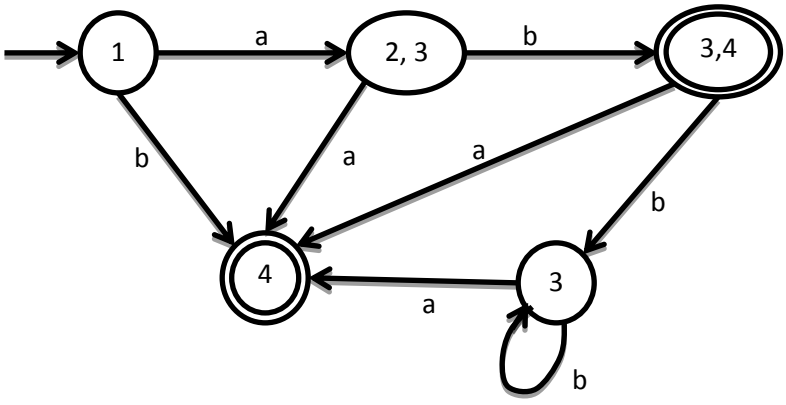


Fig. 2.24 Finite Automata

24. Convert NFA - ^ to FA

q	δ (q,^)	δ (q,0)	δ(q,1)
A	{B}	{A}	Ø
B	{D}	{C}	Ø
C	Ø	Ø	{B}
D	Ø	{D}	Ø

Table. 2.5 Transition Table

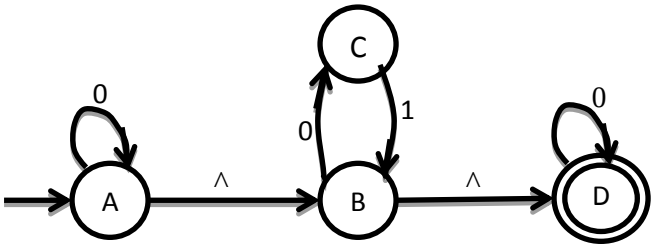


Fig. 2.25 NFA-^

δ*(A, ^) = {A,B,D}

δ*(A,0) = ^(U δ(r,0))
r∈ δ*(A,^)
= ^ (U δ(r,0))
r∈(A,B,D)
= ^(δ(A,0)U δ(B,0)U δ(D,0))
= ^{A,C,D}
={A,B,C,D}

δ*(A,1) = ^ (U δ(r,1))
r∈ δ*(A,^)
= ^ (U δ(r,1))
r∈(A,B,D)
= ^ (δ(A,1) U δ(B,1) U δ(D,1))
= Ø

δ*(B,^)= {B,D}

δ*(B,0)= ^ (U δ(r,0))
r∈δ*(B,^)
= ^ (U δ(r,0))
r∈δ(B,D)
= ^(δ(B,0)U δ(D,0))
= ^{ C,D}
= {C,D}

δ*(B,1) = ^ (U δ(r,1))
r∈δ*(B,^)

= ^ (U δ(r,1))
r∈(B,D)
= ^(δ(B,1)U δ(D,1))
= Ø

δ*(C,^) = {C}
δ*(C,0) = ^ (U δ(r,0))
r∈δ*(C,^)
= ^ (U δ(r,0))
r∈(C)
= ^(δ(C,0))
= Ø
δ*(C,1) = ^ (U δ(r,1))
r∈δ*(C,^)
= ^ (U δ(r,1))
r∈(C)
= ^(δ(C,1))
= ^{B}
= {B,D}

δ*(D,^) = {D}
δ*(D,0) = ^ (U δ(r,0))
r∈δ*(D,^)
= ^ (U δ(r,0))
r∈(D)
= ^(δ(D,0))
= {D}
δ*(D,1) = ^ (U δ(r,1))
r∈δ*(D,^)
= ^ (U δ(r,1))
r∈(D)
= ^(δ(D,1))
= Ø

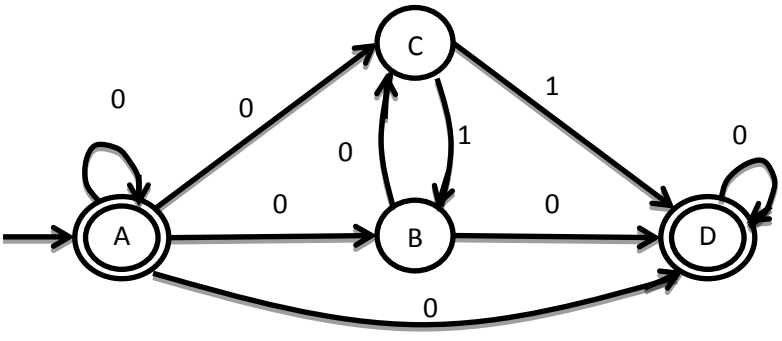


Fig. 2.26 NFA

δ({A},0) = {A,B,C,D}
δ({A},1) = Ø
δ({A,B,C,D},0) = (δ(A,0) U δ(B,0) U δ(C,0) U δ(D,0))
= {A,B,C,D}
δ({A,B,C,D},1) = (δ(A,1)U δ(B,1)U δ(C,1)U δ(D,1))
= {B,D}
δ({B,D},0) = (δ(B,0)U δ(D,0))
= {C,D}
δ({B,D},1) = (δ(B,1)U δ(D,1))
= Ø
δ({C,D},0) = (δ(C,0)U δ(D,0))
= {D}
δ({C,D},1) = (δ(C,1)U δ(D,1))
= {B,D}
δ({D},0) = {D}
δ({D},1) = Ø

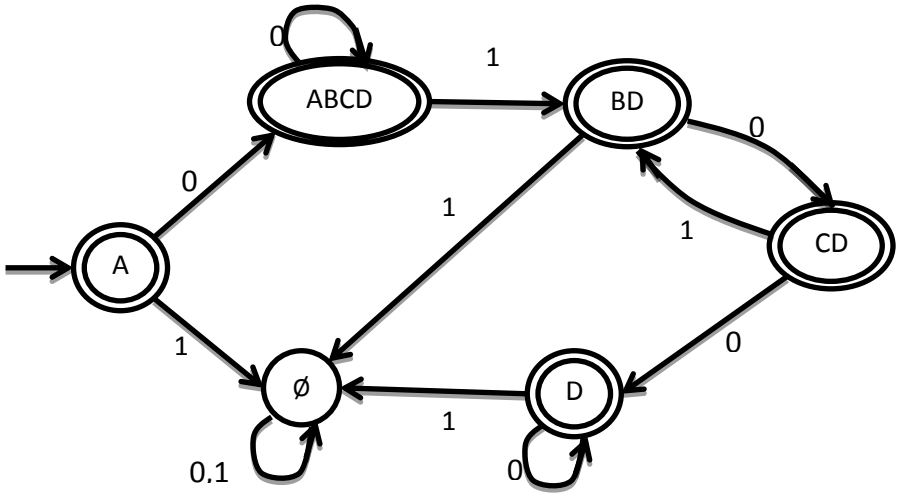


Fig. 1.27 Finite Automata

25. Convert NFA - ^ to FA

q	δ (q,^)	δ (q,0)	δ(q,1)
A	{B,D}	{A}	Ø
B	Ø	{C}	{E}
C	Ø	Ø	{B}
D	Ø	{E}	{D}
E	Ø	Ø	Ø

Table 2.6. Transition Table

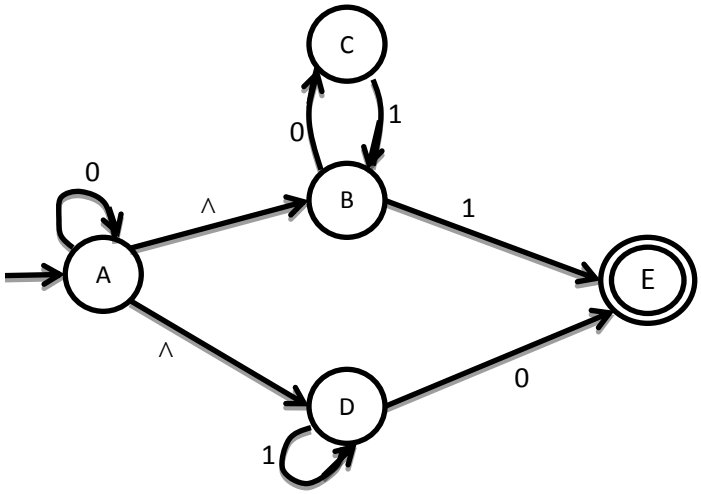


Fig. 2.28 NFA -^

$\delta^*(A,^) = \{A,B,D\}$

$\delta^*(A,0) = ^{\left(\bigcup_{r \in \delta^*(A,^)} \delta(r,0)\right)}$

$= ^{\left(\bigcup_{r \in \delta(A,B,D)} \delta(r,0)\right)}$

$= ^(\delta(A,0) \cup \delta(B,0) \cup \delta(D,0))$

$= ^\{A,C,E\}$

$= \{A,B,C,D,E\}$

$\delta^*(A,1) = ^{\left(\bigcup_{r \in \delta^*(A,^)} \delta(r,1)\right)}$

$= ^{\left(\bigcup_{r \in (A,B,D)} \delta(r,1)\right)}$

$= ^(\delta(A,1) \cup \delta(B,1) \cup \delta(D,1))$

$= ^(E,D)$

$= \{ ED\}$

$\delta^*(B,^) = \{B\}$

$\delta^*(B,0) = ^{\left(\bigcup_{r \in \delta^*(B,^)} \delta(r,0)\right)}$

$= ^{\left(\bigcup_{r \in \delta(B)} \delta(r,0)\right)}$

$= ^(\delta(B,0))$

$= ^\{ C\}$

$= \{ C\}$

$\delta^*(B,1) = ^{\left(\bigcup_{r \in \delta^*(B,^)} \delta(r,1)\right)}$

$= \wedge(\bigcup_{r \in (B)} \delta(r,1))$
$= \wedge(\delta(B,1))$
$= \{E\}$

$\delta^*(C,\wedge) = \{C\}$
$\delta^*(C,0) = \wedge(\bigcup_{r \in \delta^*(C,\wedge)} \delta(r,0))$
$= \wedge(\bigcup_{r \in (C)} \delta(r,0))$
$= \wedge(\delta(C,0))$
$= \emptyset$
$\delta^*(C,1) = \wedge(\bigcup_{r \in \delta^*(C,\wedge)} \delta(r,1))$
$= \wedge(\bigcup_{r \in (C)} \delta(r,1))$
$= \wedge(\delta(C,1))$
$= \wedge\{B\}$
$=\{B\}$
$\delta^*(D,\wedge) = \{D\}$
$\delta^*(D,0) = \wedge(\bigcup_{r \in \delta^*(D,\wedge)} \delta(r,0))$
$= \wedge(\bigcup_{r \in (D)} \delta(r,0))$
$= \wedge(\delta(D,0))$
$= \wedge\{E\}$
$= \{E\}$
$\delta^*(D,1) = \wedge(\bigcup_{r \in \delta^*(D,\wedge)} \delta(r,1))$
$= \wedge(\bigcup_{r \in (D)} \delta(r,1))$
$= \wedge(\delta(D,1))$
$= \{D\}$
$\delta^*(E,\wedge) =\{E\}$
$\delta^*(E,0) = \emptyset$
$\delta^*(E,1) = \emptyset$

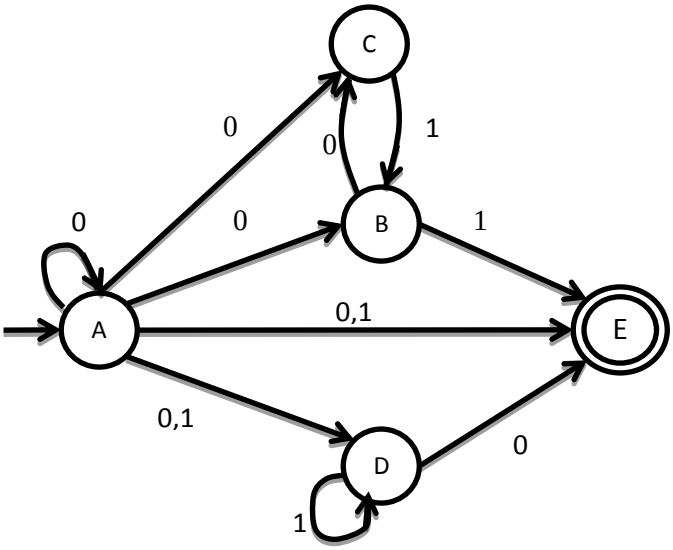


Fig. 2.29 NFA

δ({A},0)	= {A,B,C,D,E}
δ({A},1)	= {ED}
δ({A,B,C,D,E},0)	= (δ(A,0)U δ(B,0)U δ(C,0)U δ(D,0)U δ(E,0))
	= {A,B,C,D,E}
δ({A,B,C,D,E},1)	= (δ(A,1)U δ(B,1)U δ(C,1)U δ(D,1) U δ(E,0))
	= {E,B,D}
δ({E,D},0)	= (δ(E,0)U δ(D,0))
	= {Ø}U{E}
	= {E}
δ({E,D},1)	= (δ(E,1)U δ(D,1))
	= {Ø}U{D}
	= {D}
δ({B,E,D},0)	= (δ(B,0)Uδ(E,0)U δ(D,0))
	= {C,E}
δ({B,E,D},1)	= (δ(E,1)U δ(D,1))
	= {D,E}
δ({C,E},0)	= (δ(C,0)U δ(E,0))
	= Ø
δ({C,E},1)	= (δ(C,1)U δ(E,1))
	= {B}
δ({D},0)	= {E}
δ({D},1)	= {D}
δ({B},0)	= {C}
δ({B},1)	= {E}
δ({E},0)	= Ø
δ({E},1)	= Ø

δ({C},0) = Ø
δ({C},1) = {B}

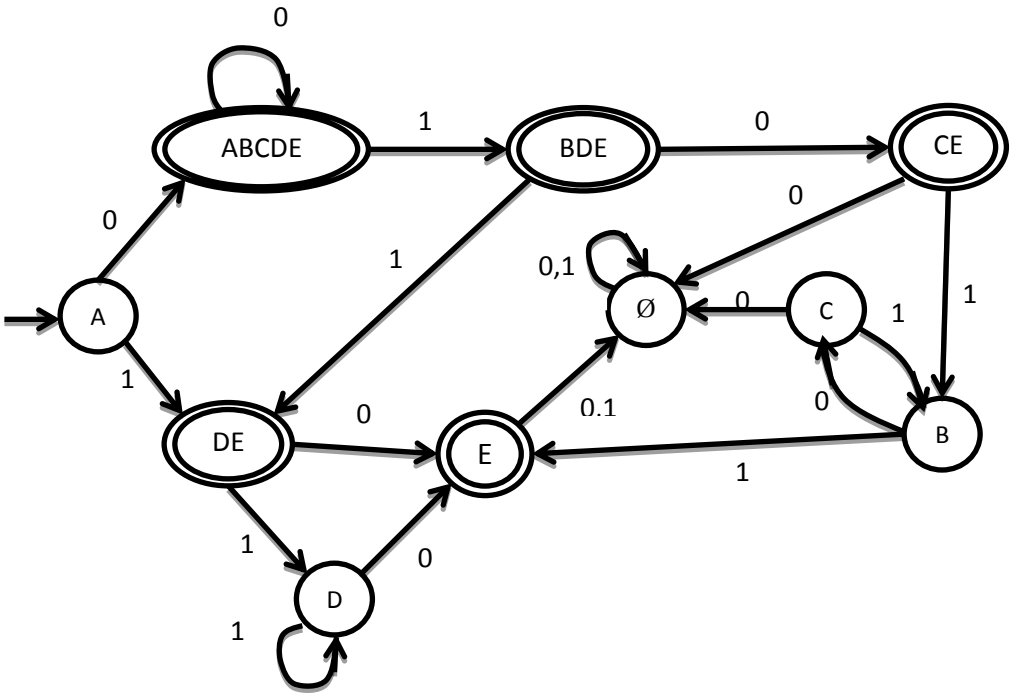


Fig.2.30 Finite Automata

Kleene's Theorem Part-1 or

26. Prove: Any Regular Language can be accepted by finite automata.

Proof:

- On the basis of statement L can be recognized by FA, NFA and NFA-^. It is sufficient to so that every regular language can be accepted by NFA- ^.
- Set of regular language over alphabet Σ contains the basic languages. Ø, {^} and {a} (a ∈ Σ) to be closed under operation of union, concatenation, and Kleene*.
- This allows us to prove using structural induction that every regular language over Σ can be accepted by an NFA-^.
- The basis step of the proof is to show that the three basic languages can be accepted by NFA-^s.
- The induction hypothesis is that L1 and L2 are languages that can be accepted by NFA-^s, and the induction step is to show that L1 U L2, L1L2, and L_1* can also be accepted by NFA-^s.
- NFA-^ for the three basic languages is shown below.

Ø **{^}** **{a}**

Fig. 2.31 Basic Languages for NFA-^

- Now, suppose that L1 and L2 are recognized by the NFA-^s M1 and M2, respectively, where for both i=1 and i=2,

  $M_i=(Q_i,\varepsilon,q_i,A_i,\delta_i)$
- By renaming state if necessary, we may assume that Q1 ∩ Q2 = Ø. We will construct NFA-^s M_u, M_c, and M_k recognizing the language L1 U L2, L1L2, and L_1*, respectively.

Construction Of M_u

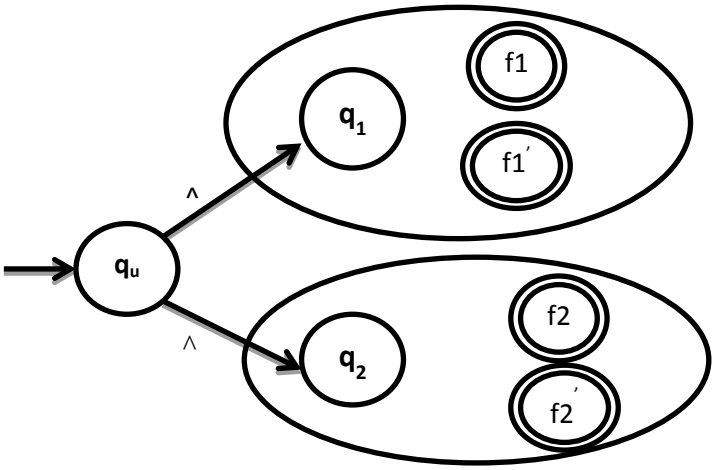


Fig. 2.32 Construction Of M_u

- Construction of $M_u = (Q_u, \varepsilon, q_u, A_u, \delta_u)$. Let q_u be a new state, not in either Q1 or Q2 and let

  $Q_u = Q1 \cup Q2 \cup \{ q_u \}$

  $A_u = A1 \cup A2$
- Now, we define δ_u so that M_u can move from its initial state to either q1 or q2 by a ^ transition, and then make exactly the same moves that the respective M_i would. Normally we define:

  $\delta_u(q_u,^) = \{q1,q2\}$

  $\delta_u(q_u,a) = \emptyset$ for every $a \in \varepsilon$

  And for each $q \in Q1 \cup Q2$ and $a \in \varepsilon \cup \{^\}$,

  $\delta_u(q_u,a)=\{ \delta_1(q,a)$ if $q \in Q1\}$ and $\{ \delta_2(q,a)$ if $q \in Q2\}$
- For either value of i, if $x \in L_i$, then M_u can process x by moving to q_i on a ^-transition and then executing the moves that cause M_i to accept x, on the other hand, if x is accepted by M_u, there is a sequence of transition corresponding to x, starting at q_u and ending at an element of A1 or A2. The first of these transition must be a ^-transition from q_u to either q1 or q2, since there are no other transition from q_u. therefore, since Q1 ∩ Q2 = Ø, either all the transition are between of Q1 or all are between elements of Q_2. It follow that x must be accepted by either M1 or M2.

Construction Of M_c

- Construction of $M_c = (Q_c, \varepsilon, q_c, A_c, \delta_c)$. In this case we do not need any new states, Let Q_c = Q1 U Q2, $q_c=q_1$, and $A_c = A_2$. The transition will include all those of M1 and M2 as well as a ε-transition from each state in A_1 to q_2.
- In other words, for any q not in A1, and $\alpha \in \varepsilon \cup \{^\}$, $\delta_c(q,a)$ is defined to be either $\delta_1(q,a)$ or $\delta_2(q,a)$, depending on whether q is in Q1 and Q2, for $q \in A1$.

$\delta_c(q,a)= \delta_1(q,a)$ for every a ∈ ε,
And $\delta_c(q,^)= \delta_1(q,^) \cup \{q2\}$

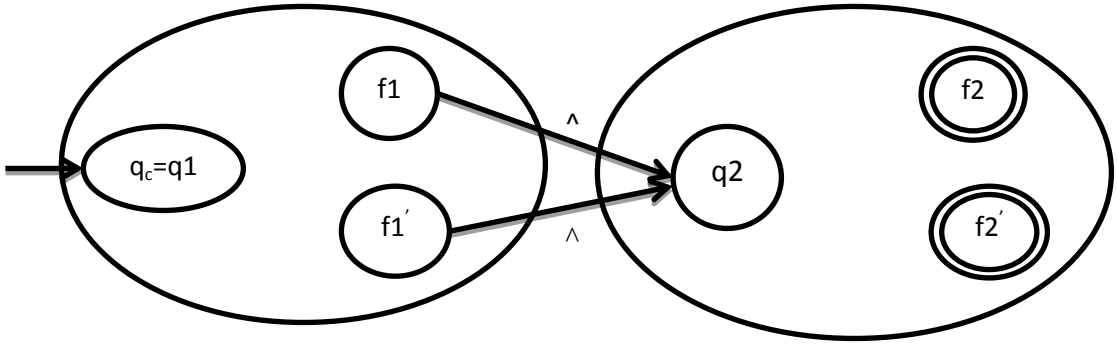


Fig. 2.33 Construction Of M_c

- On an input string x1x2, where $x_i \in L_i$ for both value of i, M_c can process x1, arriving at a state A1; jump from this state to q2 by a ^-transition; and then process x2 the way M2 would, So that x1x2 is accepted. Conversely, if x is accepted by M_c, there is a sequence of transition corresponding to x that begins at q1 and ends at an element of A2. One of them must therefore be from an element of Q1 to an element Q2, and according to the definition of δ_c, this can only be a ^- transition from an element of A1 to q2. Because Q1 ∩ Q2= Ø, all the previous transition are between elements of Q1 and all the subsequent ones are between elements of Q2. It follows that x=x1^x2=x1x2, where x1 is accepted by M1 and x2 is accepted by M2; in other words, x ∈ L1L2.

Construction Of M_k

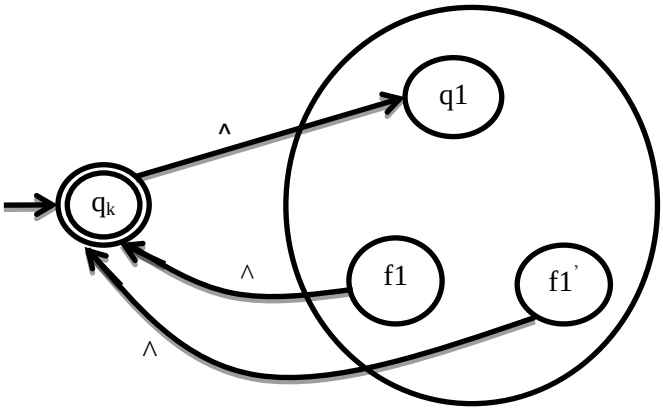


Fig. 2.34 Construction Of M_k

- Construction of $M_k = (Q_k, \Sigma, q_k, A_k, \delta_k)$. Let q_k be a new state not in Q1 and let Q_k=Q1U{q_k}. Once again all the transitions of M1 will be allowed in M_k, but in addition there is a ^-transition from q_k to q_1 and there is a ^-transition from each elements of A1 to q_k. More precisely,

  $\delta_k(q_k,^)=\{q1\}$ and $\delta_k(q_k,a)= \emptyset$ for a ∈ ε.
  for q∈A1, $\delta_k(q_k,^)= \delta_k(q_k,^) \cup \{q_k\}$.
- Suppose x∈L1*. if x=^ then clearly x is accepted by M_k. Otherwise, for some m≥1,x=x1x2….xm,

where xi ∈ L1 for each i, M_k can move from qk to q1 by a ^-transition; for each i, M_k moves from q1 to an element f_i of A1 by a sequence of transition corresponding to x_i; and for each i, M_k then moves from f_i back to qk by a ^-transition.

- It follows that (^x1^) (^x2^)…… (^xm^)=x is accepted by M_k. On the other hand, if x is accepted by M_k, there is a sequence of transition corresponding to x that begins and ends at q_k. Since the only transition from qk is a ^-transition to q1, and the only transition to qk are ^-transition from elements of A1, x can be decomposed in the form

  x= (^x1^) (^x2^)…… (^xm^)
- Where, for each i, there is a sequence of transition corresponding to x_i from q1 to an element of A1. Therefore, x∈ L1*.
- Since we have constructed an NFA-^ recognizing L in each of the three cases, the proof is complete.

27. Draw NFA-^ for following regular expression.

1. (00+1)*(10)*

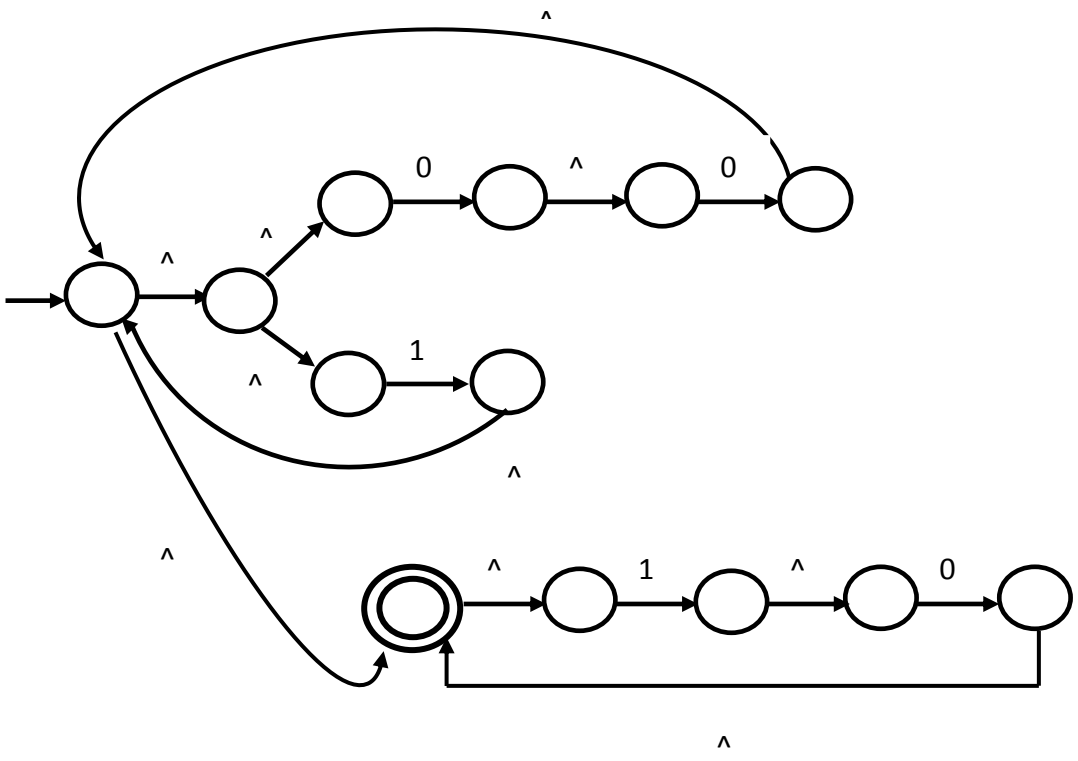


Fig. 2.35 NFA-^ for (00+1)*(10)*

2. (0 + 1)* (10+01)* 11

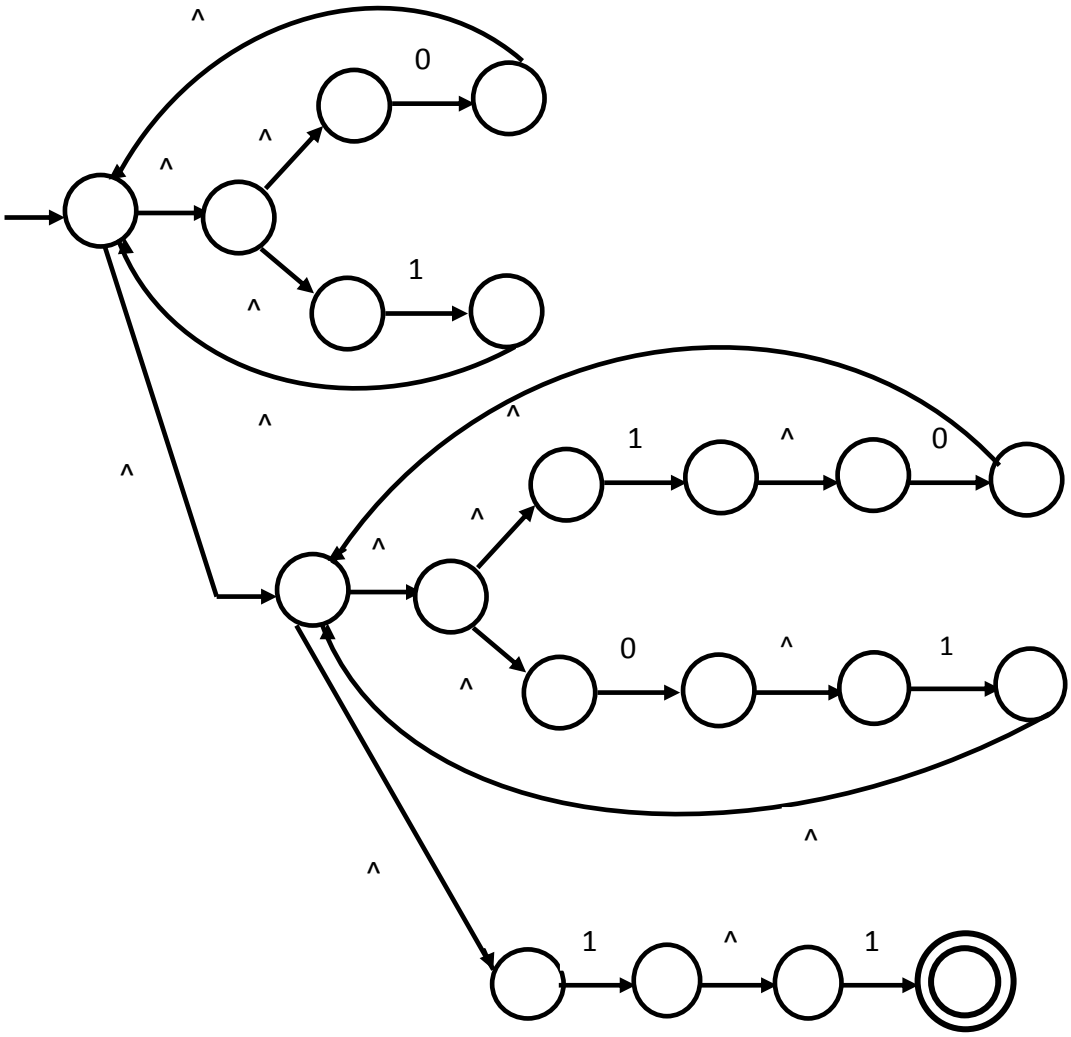


Fig. 2.36 NFA-^ for(0+1)*(10+01)*11

3. (0 + 1)* (10+110)* 1

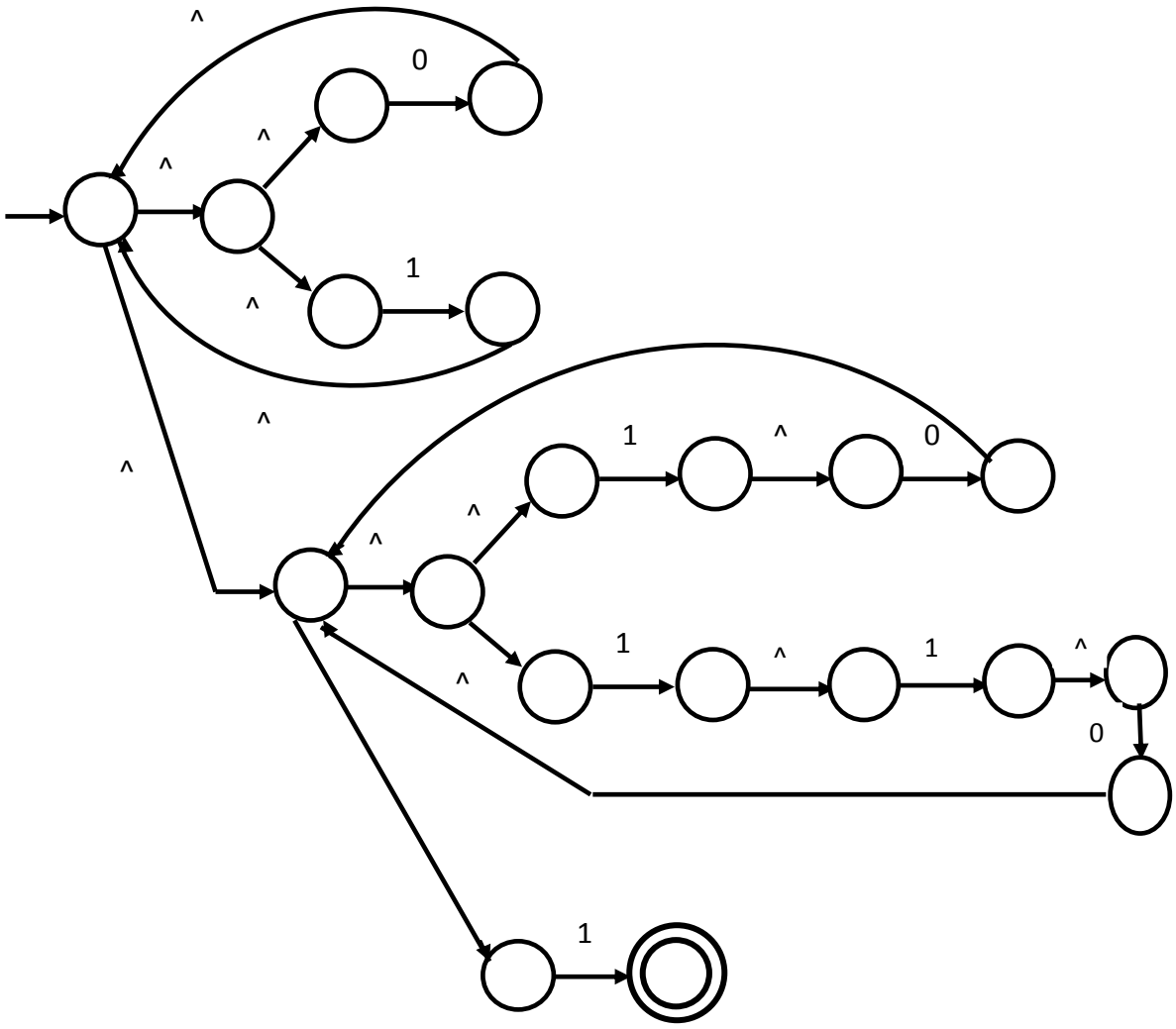


Fig. 2.37 NFA-^ for(0+1)*(10+110)*1

28. Finite Automata with Output.

- Finite automata has limited capability of either accepting a string or rejecting a string.
- Accepance of string was based on the reachability of a machine from starting state to final state. Finite automata can also be used as an output device.
- Such machines do not have a final state.
- Machine generates output on every input.
- There are two types of automata with outputs:
  1. Moore machine
  2. Mealy machine

29. Moore Machine

- Mathematiically moore machine is a six tuple machine and define as Mo=(Q,Σ, Δ, δ , ,\',q_0) where

  Q: A Nonempty finite set of state in Mo

  Σ: A Nonempty finite set of input symbols

  Δ: A Nonempty finite set of outputs

  δ: It is transition function which takes two arguments as in finite automata, one is input state and other is input symbol. The output of this function is a single state, so clearly δ is the function which is responsible for the transition of Mo.

  ,\': it is a mapping function which maps Q to Δ, giving the output associated with each state.

  q_0 : Is the initial state of Mo and $q_0 \in Q$.

Examples of Moore Machine

1. **Design a moore machine for the 1's compliment of binary number.**

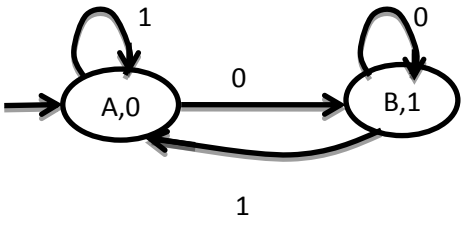


Fig. 2.38 Moore M/c for 1's compliment

2. **Design a moore machine to count occurance of "ab" as substring.**

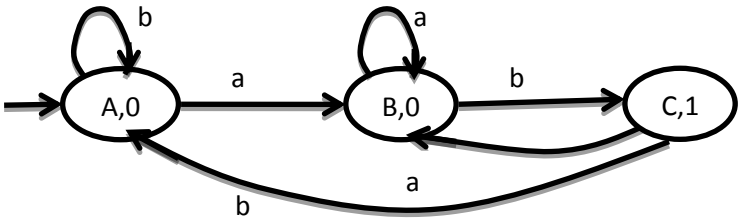


Fig. 2.39 Moore M/c to count occurances of ab

3. **Construct a moore machine that takes set of all strings over {0, 1} and produces 'A' if i/p ends witg '10' or produces 'B' if i/p ends with '11' otherwise produces 'C'.**

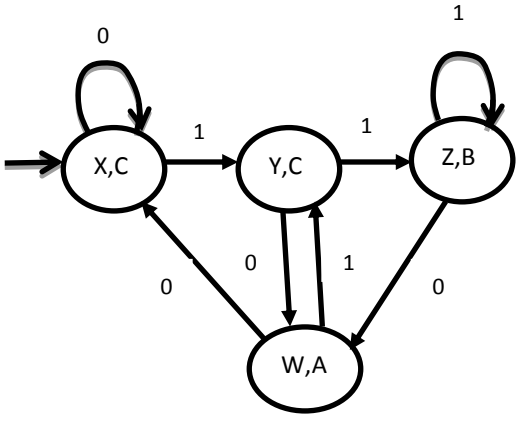


Fig. 2.40 Moore M/c for string ending in 10 or 11

4. **Construct a moore machine that takes binary number as an i/p and produces residue modulo '3' as an output.**

	0	1	Δ
q0	q0	q1	0
q1	q2	q0	1
q2	q1	q2	2

Table 2.7transition Table

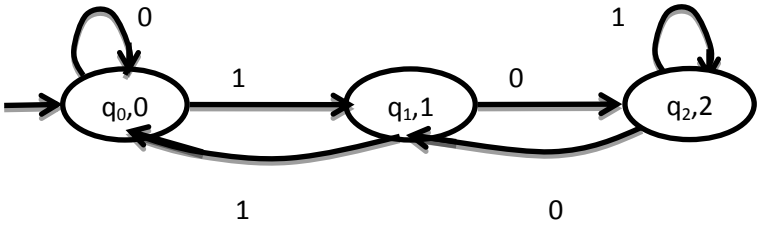


Fig. 2.41 Moore M/c to produces residue modulo '3'

30. Mealy Machine

- It is a finite automata in which output is associated with each transition.
- Mathematiically mealy machine is a six tuple machine and define as Me=(Q,Σ, Δ, δ , $_{/}\backslash$',q_0) where

  Q: A Nonempty finite set of state in Me

  Σ: A Nonempty finite set of input symbols

  Δ: A Nonempty finite set of outputs

  δ: It is transition function which takes two arguments as in moore machine, one is input state and other is input symbol. The output of this function is a single state, so clearly δ is the function which is responsible for the transition of Me.

  $_{/}\backslash$': it is a mapping function which maps Q x Σ to Δ, giving the output associated with each transition.

  q_0 : Is the initial state of Me and $q_0 \in Q$.

Examples of Mealy Machine

1. **Design a mealy machine for the 1's compliment of binary number.**

Fig. 2.42 Mealy M/c for 1's compliment of binary

2. **Design a mealy machine for regular expression (0+1)*(00+11).**

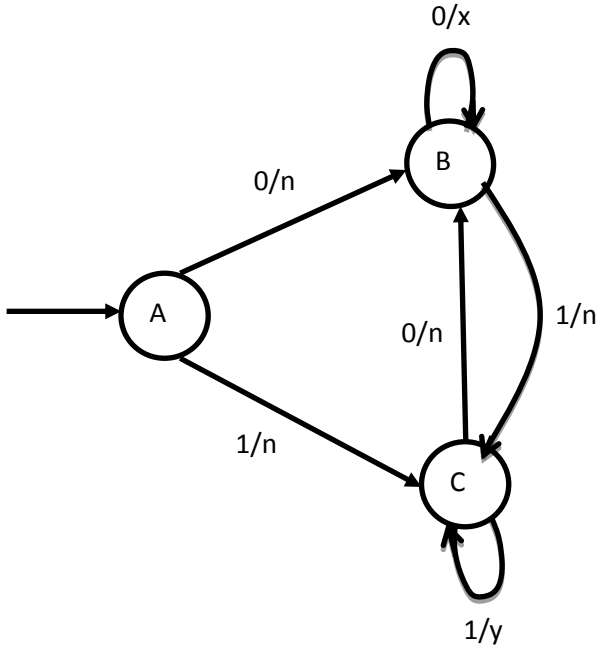


Fig. 2.43 Mealy M/c for (0+1)*(00+11)

3. **Design a mealy machine where Σ={0, 1, 2} print residue modulo 5 of input treated as ternary (base 3).**

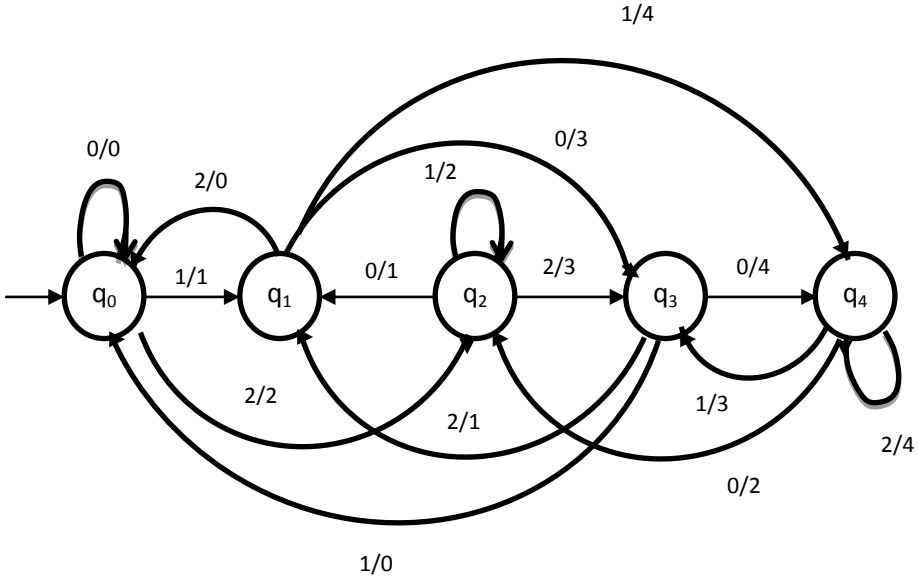


Fig. 2.44 Mealy M/c to produces residue modulo '5'

31. Explain procedure to minimize Finite Automata

S-1: Make final state and non-final state as distinguish.

S-2: Recursively interacting over the pairs of state for any transition for

$\delta(p,x) = r$

$\delta(q,x) = s$

and for $x \in \varepsilon$. If r and s are distinguishable make p and q as distinguish.

S-3: If any iteration over all possible state pairs one fails to find a new pair of states that are distinguishable terminate.

S-4: All the states that are not distinguished are equivalence.

32. For following FA find minimized FA accepting same language.

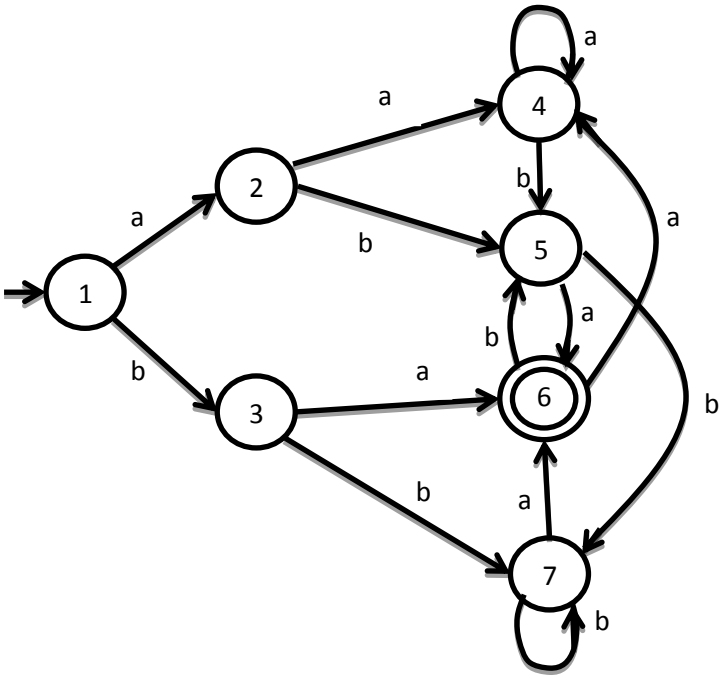


Fig. 2.45 Finite Automata

2						
3	X	X				
4			X			
5	X	X		X		
6	X	X	X	X	X	
7	X	X		X		X
	1	2	3	4	5	6

Final state is {6}

And, Non-Final state is {1,2,3,4,5,7}

(6, 1), (6,2), (6, 3), (6, 4), (6, 5), (6, 7) are distinguish pairs.

Consider pair (1,2)

$\delta(1,a)=2$ $\delta(1,b)=3$

δ(2,a)=4 δ(2,b)=5

Consider pair (1,3)

δ(1,a)=2 δ(1,b)=3

δ(3,a)=6 δ(3,b)=7

pair (2,6) is distinguish, so It's a distinguished pair.

Consider pair (1,4)

δ(1,a)=2 δ(1,b)=3

δ(4,a)=4 δ(4,b)=5

Consider pair (1,5)

δ(1,a)=2 δ(1,b)=3

δ(5,a)=6 δ(5,b)=7

pair (2,6) is distinguish, so It's a distinguished pair.

Consider pair (1,7)

δ(1,a)=2 δ(1,b)=3

δ(7,a)=6 δ(7,b)=7

pair (2,6) is distinguish, so It's a distinguished pair.

Consider pair (2,3)

δ(2,a)=4 δ(2,b)=5

δ3(,a)=6 δ(3,b)=7

pair (4,6) is distinguish, so It's a distinguished pair.

Consider pair (2,4)

δ(2,a)=4 δ(2,b)=5

δ(4,a)=4 δ(4,b)=5

Consider pair (2,5)

δ(2,a)=4 δ(2,b)=5

δ(5,a)=6 δ(5,b)=7

pair (4,6) is distinguish, so It's a distinguished pair.

Consider pair (2,7)

δ(2,a)=4 δ(2,b)=5

δ(7,a)=6 δ(7,b)=7

pair (4,6) is distinguish, so It's a distinguished pair.

Consider pair (3,4)

δ(3,a)=6 δ(3,b)=7

δ(4,a)=4 δ(4,b)=5

pair (6,4) is distinguish, so (3,4) is distinguish.

Consider pair (3,5)

δ(3,a)=6 δ(3,b)=7

δ(5,a)=6 δ(5,b)=7

Consider pair (3,7)

δ(3,a)=6 δ(3,b)=7

δ(7,a)=6 δ(7,b)=7

Consider pair (4,5)

δ(4,a)=4 δ(4,b)=5

δ(5,a)=6 δ(5,b)=7
pair (6,4) is distinguish, so (4,5) is distinguish.
Consider pair (4,7)
δ(4,a)=4 δ(4,b)=5
δ(7,a)=6 δ(7,b)=7
pair (6,4) is distinguish, so (4,7) is distinguish.
Consider pair (5,7)
δ(5,a)=6 δ(5,b)=7
δ(7,a)=6 δ(7,b)=7

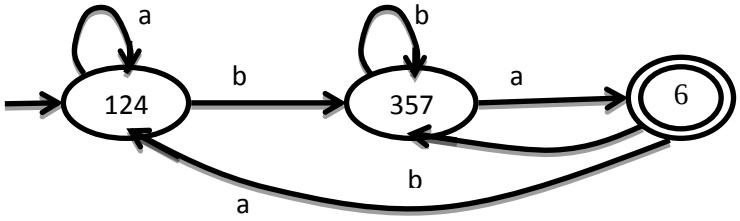


Fig. 2.46 Minimized Finite Automata

33. Define pumping lemma and its application.

Suppose L is a regular language. Then there is an integer n so that for any x∈ L with |x|>=n, there are strings u, v, and w so that

1. x=uvw
2. |uv|<=n
3. |v|>0
4. For any m>=0, $uv^m w \in L$

Application: (Explain the application of the Pumping Lemma to show a Language is Regular or Not)
The pumping lemma is extremely useful in proving that certain sets are non-regular. The general methodology followed during its applications is :

- Select a string z in the language L.
- Break the string z into x, y and z in accordance with the above conditions imposed by the pumping lemma.
- Now check if there is any contradiction to the pumping lemma for any value of i.

34. Use the pumping lemma to show that following language is not regular: L = {ww|w ϵ {0,1}*}

Step 1: Let us assume that L is regular and L is accepted by an FA with n states.
Step 2: Let us chose the string

$\psi = \underbrace{a^n b}_{\psi} \; \underbrace{a^n b}_{\psi}$

$|\psi| = 2n+2 >= n$

Let us write w as xyz with

|y|>0
And |xy|<=n
Since |xy|<=n, x must be of the form a^s.
Since |xy|<=n, y must be of the form a^r | r>0
Now $\psi=a^nba^nb=\underbrace{a^s}_{x}\underbrace{a^r}_{y}\underbrace{a^{n-s-r}ba^nb}_{z}$

Step 3: Let us check whether x y^i z for i=2 belongs to L.
$Xy^2z= a^s a^{2r} a^{n-s-r}ba^nb= a^{n+r}ba^nb$
Since, r>0, $a^{n+r}ba^nb$ is not of the form ψψ as the number of a's in the first half is n+r and second half is n. Therefore, $xy^2z \notin L$. Hence by contrdiction we can say language is not regular.

35. Prove that the language L = {0^n: n is a prime number} is not regular.

Step 1: Let us assume that L is regular and L is accepted by an FA with n states.
Step 2: Let us chose the string
$\psi=a^p$,where p is prime and p>n
$|\psi|=|a^p|=p>=n$
Let us write w as xyz with
|y|>0
And |xy|<=n
Since |xy|<=n, x must be of the form a^s.
We assume that $y= a^m$ for m>0.
Step 3: Length of x y^i z can be written as given below.
$Xy^iz= |xyz|+|y^{i-1}|=p+(i-1)m$
As $|y|=|a^m|=m$
Let us check whether P(i-1) m is prime for every i.
For i=p+1, $p+(i-1)m= P+ P_m =P(1+m)$
So $xy^{p+1}z \notin L$. Hence by contrdiction we can say language is not regular.

36. Use Pumping Lemma to show that following language is not regular. L = { ww^R / w ε {0,1}* }

Step 1: Let us assume that L is regular and L is accepted by an FA with n states.
Step 2: Let us chose the string
$\psi=\underbrace{a^nb}_{\psi}\ \underbrace{ba^n}_{\psi^R}$
$|\psi|=2n+2>=n$
Let us write w as xyz with
|y|>0
And |xy|<=n
Since |xy|<=n, x must be of the form a^s.
Since |xy|<=n, y must be of the form a^r | r>0
Now $\psi=a^nbba^n=\underbrace{a^s}_{x}\underbrace{a^r}_{y}\underbrace{a^{n-s-r}bba^n}_{z}$

Step 3: Let us check whether $x\,y^i\,z$ for i=2 belongs to L.

$Xy^2z = a^s a^{2r} \underline{a^{n-s-r}} \underline{bba^n} = a^{n+r}bba^n$

Since, r>0, $a^{n+r}bba^n$ is not of the form $\psi\psi^R$ as the string starts with (n+r) a's but ends in (n) a's. Therefore, $Xy^2z \notin L$. Hence by contrdiction we can say language is not regular.

1. Define: Context Free Grammar & Context Free Language.

Context Free Grammar:

A context free grammar is a 4-tuple G=(V,Σ,S,P) where,

V is finite set of non terminals,

Σ is disjoint finite set of terminals,

S is an element of V and it's a start symbol,

P is a finite set formulas of the form $A \rightarrow \alpha$ where $A \in V$ and $\alpha \in (V \cup \Sigma)^*$.

Application of Context Free Grammar(CFG):

1) CFG are extensively used to specify the syntax of programming language.
2) CFG is used to develop a parser.

Context Free Language:

Language generated by CFG is called context free language.

Let G= (V, Σ, S, P) be a CFG. The Language generated by G is

$L(G) : \{x \in \Sigma^* / S \Longrightarrow_G^* x\}$

A language L is a context free Language(CFL) if, there is a CFG G so that L=L(G)

2. Define: Regular Grammar.

A grammar G=(V,Σ,S,P) is regular if every production takes one of the two forms,

$B \rightarrow aC$

$B \rightarrow a$

Where B and C are Nonterminals and a is terminal.

3. Give Recursive Definitions for following.

I. Recursive Definition of {a,b}*

1. ^∈L.
2. For any S∈L, Sa∈L.
3. For any S∈L, Sb∈L.
4. No other strings are in L.

II. Recursive Definition of Palindrome (pal)

1. ^, a, b ∈ pal
2. For any S ∈ pal , aSa ∈ pal and bSb ∈ pal
3. No other string are in pal

III. The language $\{a^n b^n / n \geq 0\}$

1. ^∈ L
2. For every x ∈ L, axb ∈L
3. No other strings are in L

4. Write CFG for following.

1) Write CFG for ab*

$S \rightarrow aX$

$X \rightarrow ^ | bX$

2) Write CFG for a*b*

S→XY
X→aX|^
Y→bY|^

3) Write CFG for (011+1)*(01)*

S→AB
A→011A | 1A | ^
B→01B | ^

4) Write CFG which contains at least three times 1.

S→A1A1A1A
A→0A | 1A | ^

5) Write CFG that must start and end with same symbol.

S→0A0 | 1A1
A→0A | 1A | ^

6) The language of even & odd length palindrome string over {a,b}

S→aSa|bSb|a|b|^

7) No. of a and no. of b are same

S→aSb|bSa|^

8) Write CFG for regular expression (a+b)*a(a+b)*a(a+b)*

S→XaXaX
X→aX|bX|^

9) Write CFG for L={$a^ib^jc^k$ | i=j or j=k}

For i=j	for j=k
S->AB	S->CD
A->aAb | ab	C->aC | a
B->cB | c	D->bDc | bc

10) Write CFG for L={ $a^ib^jc^k$ | j>i+k}

S→ABC
A→aAb |^
B→bB | b
C→bCc |^

11) Write CFG for L={ $0^i1^j0^k$ | j>i+k}

S→ABC
A→0A1 |^
B→1B | 1
C→1C0 |^

5. Define: Types of Derivation & Ambiguity.

There are mainly two types of derivations.

1. Left most derivation
2. Right most derivation

Let Consider the CFG with the Production S→S+S | S-S | S*S | S/S |(S)| a

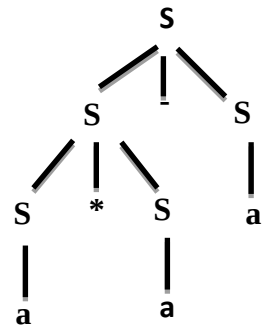
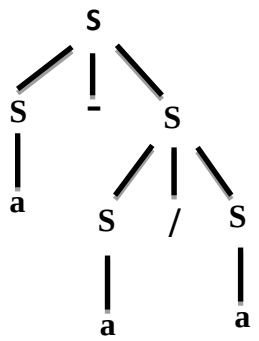
Left Most Derivation	Right Most Derivation
A derivation of a string W in a grammar G is a Left most derivation if at every step the left most nonterminal is replaced	A derivation of a string W in a grammar G is a Right most derivation if at every step the right most nonterminal is replaced
Consider string a*a-a S→S-S S*S-S a*S-S a*a-S a*a-a	Consider string: a-a/a S→S-S S-S/S S-S/a S-a/a a-a/a
Equivalent left most derivation tree 	Equivalent Right most derivation tree 

Table 3.1 Difference between left most & right most derivation

An Ambiguous CFG :

A context free grammar G is ambiguous if there is at least one string in L(G) having two or more distinct derivation tree. (or, equivalently two or more distinct leftmost derivation or rightmost derivation)

1) **Prove that given grammar is ambiguous. S→S+S | S-S | S*S | S/S | (S) |a**

String : a+a+a

S→S+S	S→S+S
a+S	S+S+S
a+S+S	a+S+S
a+a+S	a+a+S
a+a+a	a+a+a

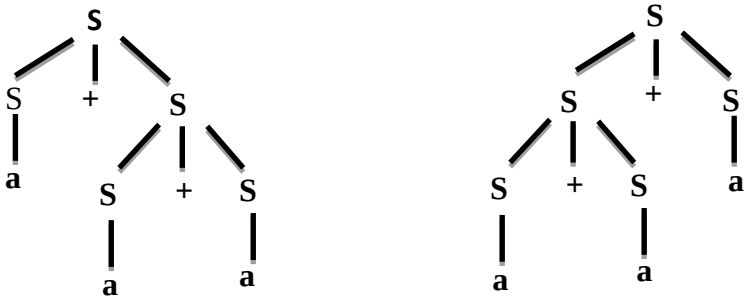


Fig. 3.1 Two left most derivation tree for string a+a+a

- Here, we have two left most derivation for string *a+a+a* hence, above grammar is ambiguous.

2) Prove that S->a | Sa | bSS | SSb | SbS is ambiguous

String: baaab

S→bSS	S→SSb
baS	bSSSb
baSSb	baSSb
baaSb	baaSb
baaab	baaab

- We have two left most derivation for string *baaab* hence, above grammar is ambiguous.

6. Conversion from Finite Automata to Grammar.

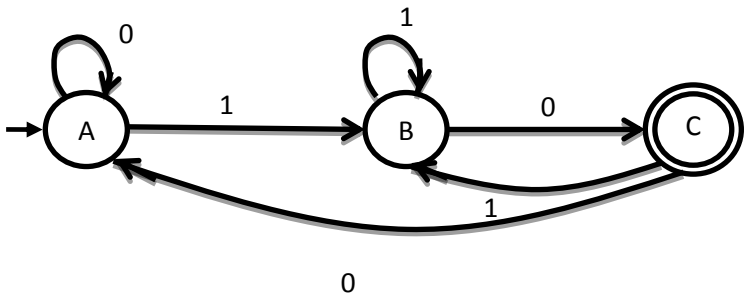


Fig. 3.2 Finite Automata

Equivalent CFG

A→0A
A→1B
B→1B
B→0C
B→0
C→0A
C→1B

7. Backus-Naur Form (BNF)

- BNF is one of the notation techniques for context free grammar.
- It is often used to describe syntax of the language used in computing.
- BNF is shorthand notation for CFG.
- Following grammar uses the notation known as Backus Naur Form. Here, variables written between <..> are non terminals and vertical bar '|' indicating a alternate choice. Apart from the familiar notation =, | and <..>, a new element here is […], which is used to enclosed an optional specification.

Example:

<exp>=<exp> + <term> | <term>
<term>=<term> * <factor> | <factor>

<factor>=<factor> ^ <primary> | <primary>
<primary>=<id> | <const>
<id>=<letter>
<const>=[+/-]<digit>
<letter>=a | b | c |……| z
<digit>=0 | 1 |…….| 9

8. Simplified forms & Normal forms.

Definition: Nullable Variable

A Nullable variable in a CFG G=(V,Σ,S,P) is defined as follows:

1) Any variable A for which P contains A→^ is nullable.
2) if P contains production
   A→B1B2…..Bn where B1B2…Bn are nullable variable, then A is nullable.
3) No other variables in V are nullable.

Eliminate ^ production :

1) **S→aX/Yb**
   X→S/^
   Y→bY/b
   Grammar after elimination of ^ production:
   S→aX/Yb/a
   X→S
   Y→bY/b
2) **S→XaX/bX/Y**
   X→XaX/XbX/^
   Y→ab
   Grammar after elimination of ^ production:
   S→XaX/bX/Y/aX/Xa/a/b
   X→XaX/aX/Xa/a/XbX/Xb/bX/b
   Y→ab

Definition: Unit Production

Unit productions are always in the form of A→B. Where A & B are single non terminals.

Eliminate Unit Production:

1) **S→ABA/BA/AA/AB/A/B**
   A→aA/a
   B→bB/b
   Grammar after elimination of unit production:
   Unit production are S→A and S→B
   S→ABA/BA/AA/AB/aA/a/bB/b
   S→aA/a
   S→bB/b
2) **S→Aa/B**
   A→a/bc/B
   B→A/bb

Grammar after elimination of unit production:
Unit production are S→B,A→B and B→A.
A→a/bc/B
A→a/bc/A/bb
A→a/bc/bb

B→A/bb
B→a/bc/bb

S→Aa/B
S→Aa/a/bc/bb

So CFG after removing unit production is:
S→Aa/a/bc/bb
A→a/bc/bb
B→a/bc/bb

Definition: Chomsky Normal Form

A context free grammar is in Chomsky normal form (CNF) is every production is one of these two forms:

A→BC
A→a

Where A, B, C are nonterminals and a is terminal.

Step to convert CFG into CNF:

1) Eliminate ^-Productions.
2) Eliminate Unit Productions.
3) Restricting the right side of productions to single terminal or string of two or more nonterminals.
   (Replace all mixed string with solid NTs)
4) Final step of CNF. (shorten the string of NT to length 2)

9. Convert following CFG to CNF:

S→aX/Yb
X→S/∧
Y→bY/b

Step-1: Eliminate ^-Production:

Nullable production is X→^, new CFG without ^-production is:
S→aX/a/Yb
X→S
Y→bY/b

Step-2: Eliminate Unit Production:

Unit Production is X→S, new CFG without Unit Production is:
S→aX/a/Yb

X→aX/a/Yb
Y→bY/b

Step-3: Replace all mixed string with solid NT:

S→AX/YB/a
X→AX/YB/a
Y→BY/b
A→a
B→b

Step-4 : Shorten the string of NT to length 2

All NT strings on the RHS in the above CFG are already the required length.
So, CFG is in CNF.

10. Convert following CFG to CNF

S→AACD
A→aAb/∧
C→aC/a
D→aDa/bDb/∧

Step-1: Eliminate ^-Production:

Nullable production is A→^ and D→^, new CFG without ^-production is:
*apply for A →^*
S→AACD/ACD/CD
A→ab/aAb
C→aC/a
D→aDa/bDb/^
*apply for D →^*
S→AACD/ACD/CD/AAC/AC/C
A→ab/aAb
C→aC/a
D→aDa/bDb/aa/bb

Step-2: Eliminate Unit Production:

Unit Production is S→C, new CFG without Unit Production is:
S→AACD/ACD/CD/AAC/AC/aC/a
A→ab/aAb
C→aC/a
D→aDa/bDb/aa/bb

Step-3:Replace all mixed string with solid NT:

S→AACD/ACD/CD/AAC/AC/PC/a
A→PQ/PAQ
C→PC/a
D→PDP/QDQ/PP/QQ
P→a

Q→b

Step-4 : Shorten the string of NT to length 2

S→AT1, T1→AT2 ,T2→CD
S→AU1,U1→CD
S→AV1,V1→AC
S→CD/AC/PC/a
A→PQ
A→PW1, W1→AQ
C→PC/a
D→PP/QQ D→PY1, Y1→DP
D→QZ1, Z1→DQ
P→a
Q→b

11. Convert following CFG to CNF

S→S(S)/∧

Step-1: Eliminate ^-Production:

Nullable production is S→^, new CFG without ^-production is:
S→S(S)/(S)/S()/()

Step-2: Eliminate Unit Production:

Here, there is no unit production,
S→S(S)/(S)/S()/()

Step-3:Replace all mixed string with solid NT:

S→SXSY/XSY/SXY/XY
X→(
Y→)

Step-4 : Shorten the string of NT to length 2

S→ST1, T1→XT2, T2→SY
S→XV1 V1→SY
S→SU1 U1→XY
S→XY
X→(
Y→)

12. Unions, Concatenations and Kleen's of Context free language.

Theorem:- If L_1 and L_2 are context - free languages, then the languages $L_1 \cup L_2$, L_1L_2 , and L_1^* are also CFLs.

The proof is constructive: Starting with CFGs

$G_1 = (V_1, \Sigma, S_1, P_1)$ and $G_2 = (V_2, \Sigma, S_2, P_2)$,

Generating L_1 and L_2, respectively, we show how to construct a new CFG for each of the three cases.

A grammar $G_u = (V_u, \Sigma, S_u, P_u)$ generating $L_1 \cup L_2$. First we rename the element of V_2 if necessary

so that $V_1 \cap V_2 = \emptyset$ and we define

$V_u = V_1 \cup V_2 \cup \{S_u\}$

Where S_u is a new symbol not in V_1 or V_2. Then we let

$P_u = P_1 \cup P_2 \cup \{ S_u \rightarrow S_1 \mid S_2 \}$

On the one hand, if x is in either L_1 or L_2, then $S_u \Rightarrow^* x$ in the grammar G_u, because we can start a derivation with either $S_u \rightarrow S_1$ or $S_u \rightarrow S_2$ and continue with the derivation of x in G_1 or G_2. Therefore,

$L_1 \cup L_2 \subseteq L(G_u)$

On the other hand, if x is derivable from S_u in G_u, the first step in any derivation must be

$S_u \Rightarrow S_1$ or $S_u \Rightarrow S_2$

In the first case, all subsequent productions used must be productions in G_1, because no variables in V_2 are involved, and thus $x \in L_1$; in the second case, $x \in L_2$. Therefore,

$L(G_u) \subseteq L_1 \cup L_2$

A grammar $G_c = (V_c, \Sigma, S_c, P_c)$ generating L_1L_2. Again we relabel variables if necessary so that $V_1 \cap V_2 = \emptyset$ and define

$Vc = V_1 \cup V_2 \cup \{S_c\}$

This time we let

$P_c = P_1 \cup P_2 \cup \{ S_c \rightarrow S_1S_2 \}$

If $x \in L_1L_2$ then $x = x_1x_2$, where $x_i \in L_i$ for each i. we may then derive x in G_c as follows:

$S_c \Rightarrow S_1 S_2 \Rightarrow {}^*x_1 S_2 \Rightarrow {}^* x_1x_2 = x$

Where the second step is the derivation of x_1 in G_1 and the third step is the derivation of x_2 in G_2. Conversely, if x can be derived from S_c, then since the first step in the derivation must be $S_c \Rightarrow S_1 S_2$, x must be derivable from $S_1 S_2$. Therefore, $x = x_1x_2$, where for each i, x_i can be derived from S_i in G_c. Since $V_1 \cap V_2 = \emptyset$, being derivable from S_i in G_c means being derivable from S_i in G_i, and so $x \in L_1L_2$.

A grammar $G^* = (V, \Sigma, S, P)$ generating $L_1{}^*$. Let

$V = V_1 \cup \{S\}$

Where $S \notin V_1$. The language $L_1{}^*$ contains strings of the form $x = x_1x_2 \ldots x_k$, where each $x_i \in L_1$. Since each x_i can be derived from S_1, then to derive x from S it is enough to be able to derive a string of k S_1'S. We can accomplish this by including the productions

$S \rightarrow S_1S \mid \wedge$

In P. Therefore, let

$P = P_1 \cup \{ S \rightarrow S_1S \mid \wedge \}$

The proof that $L_1{}^* \subseteq L(G^*)$ is straightforward. If $x \in L(G^*)$, on the other hand, then either $x = \wedge$ or x can be derived from some string of the form $S_1{}^k$ in G^*. In the second case, since the only production in G^* beginning with S_1 are those in G_1, we may conclude that

$x \in L(G_1)^k \subseteq L(G_1)^*$.

1. What is Pushdown Automata?

- Pushdown automata is a computational model equivalent to context free grammar.
- A pushdown Automata is essentially a finite Automata with a stack data structure as shown below.

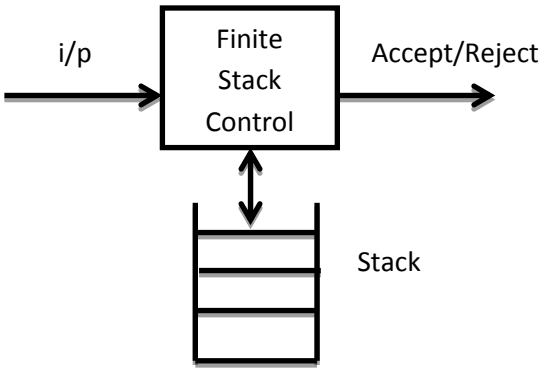


Fig. 4.1 Computational model of PDA

- A PDA can write an unbounded no. of stack symbols on the stack and read these symbols later.
- Writing a symbol on to the stack is called "PUSH" operation.
- Removing a symbol off the stack is called "POP" operation.
- PDA can accept only the stacks top most symbol.

Definition of Pushdown Automata :-

A pushdown Automata(PDA) is a 7-tuple $M= (Q,\Sigma, \Gamma, q_0, Z_0, A, \delta)$ where

Q is finite set of states.

Σ and Γ are finite sets, the i/p and stack alphabet respectively.

q_0 is initial state.

Z_0 is initial stack symbol, is an element of Γ.

A is a set of accepting states.

$\delta : Q \times (\Sigma \cup \{\wedge\}) \times \Gamma \rightarrow$ the set of finite subset of $Q \times \Gamma^*$.

The function δ is called transition function of M.

Acceptance by a PDA:-

if $M= (Q,\Sigma, \Gamma, q_0, Z_0, A, \delta)$ is a PDA and $x \in \Sigma^*$, x is accepted by M, If $(q_0,x,z_0) \vdash_m^* (q,\wedge,\alpha)$ for some $\alpha \in \Gamma^*$ and some $q \in A$. A language $L \subseteq \Sigma^*$ is said to be accepted by M if L is precisely the set of strings accepted by M, in this case, we write L=L(M)

2. Deterministic Pushdown Automata.

- $M= (Q,\Sigma, \Gamma, q_0, Z_0, A, \delta)$ be a pushdown Automata.
- M is deterministic if there is no configuration for which M has a choice of more than one move.
- In other words M is deterministic if it satisfies both of the following condition:
  1) For any $q \in Q$, $q \in \Sigma \cup \{\wedge\}$ and $X \in \Gamma$, the set $\delta(q,a,x)$ has at most one element.
  2) for any $q \in Q$ and $X \in \Gamma$, if $\delta(q,\wedge,x) \neq \emptyset$ then $\delta(q,a,x) = \emptyset$ for every $a \in \Sigma$.
- A language L is a deterministic context free language (DCFL) if there is deterministic PDA

accepting L.

3. Design a PDA for following CFG. S→aSa/bSb/c OR Design PDA for L={x∈x^r/x∈{a,b}*}. The string in L are odd length palindromes over {a,b} OR Design PDA for palindrome with middle symbol c.

Move No.	State	i/p	Stack Symbol	Move
1	q_0	a	Z_0	(q_0,aZ_0)
2	q_0	b	Z_0	(q_0,bZ_0)
3	q_0	a	a	(q_0,aa)
4	q_0	b	a	(q_0,ba)
5	q_0	a	b	(q_0,ab)
6	q_0	b	b	(q_0,bb)
7	q_0	c	Z_0	(q_1, Z_0)
8	q_0	c	a	(q_1, a)
9	q_0	c	b	(q_1, b)
10	q_1	a	a	(q_1, ^)
11	q_1	b	b	(q_1, ^)
12	q_1	^	Z_0	(q_2, Z_0)

Table 4.1 Transition table for xcx^r

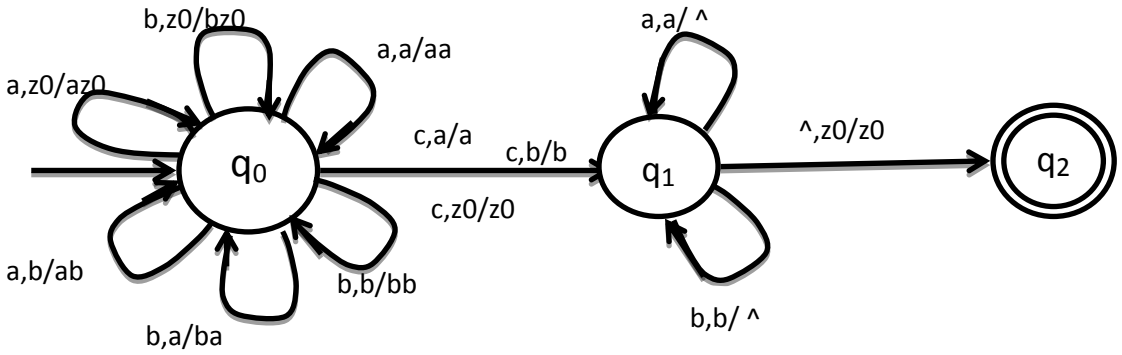


Fig. 4.2 PDA for xcx^r

String:abcba

Move No.	Resulting State	unread i/p	Stack
Initial	q_0	abcba	Z_0
1	q_0	bcba	aZ_0
4	q_0	cba	baZ_0
9	q_1	ba	baZ_0
11	q_1	a	aZ_0

10	q_1	^	Z_0
12	q_2	^	Z_0

4. Design PDA for even-odd length palindrome.

Move No.	State	i/p	Stack Symbol	Move
1	q_0	a	Z_0	(q_0,aZ_0) (q_1,Z_0)
2	q_0	b	Z_0	(q_0,bZ_0) (q_1,Z_0)
3	q_0	a	a	(q_0,aa) (q_1,a)
4	q_0	b	a	(q_0,ba) (q_1,a)
5	q_0	a	b	(q_0,ab) (q_1,b)
6	q_0	b	b	(q_0,bb) (q_1,b)
7	q_0	^	Z_0	(q_1, Z_0)
8	q_0	^	a	(q_1, a)
9	q_0	^	b	(q_1, b)
10	q_1	a	a	$(q_1, ^)$
11	q_1	b	b	$(q_1, ^)$
12	q_1	^	Z_0	(q_2, Z_0)

Table 4.2 Transition table for even-odd length palindrome

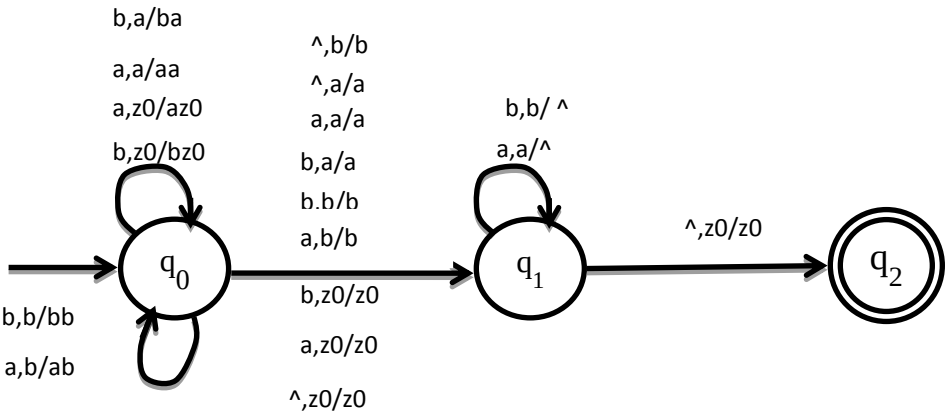


Fig. 4.3 PDA for even-odd length palindrome

String:aba

Move No.	Resulting State	unread i/p	Stack
Initial	q_0	aba	Z_0
4	q_0	ba	aZ_0
10	q_1	a	aZ_0
12	q_1	^	Z_0
	q_2	^	Z_0

5. Design PDA for $L=\{a^nb^n/a,b\in\Sigma,n\geq 0\}$ or PDA for Number of a's and b's are same.

Move No.	State	i/p	Stack Symbol	Move
1	q_0	^	Z_0	(q_2, z_0)
2	q_0	a	Z_0	(q_0,aZ_0)
3	q_0	a	a	(q_0,aa)
4	q_0	b	a	$(q_1, ^)$
5	q_1	b	a	$(q_1, ^)$
6	q_1	^	Z_0	(q_2, z_0)

Table 4.3 Transition table for Number of a's and b's are same.

String:aaabbb

Resulting State	unread i/p	Stack
q_0	aaabbb	Z_0
q_0	aabbb	aZ_0
q_0	abbb	aaZ_0
q_0	bbb	$aaaZ_0$
q_1	bb	aaZ_0
q_1	b	aZ_0
q_1	^	Z_0
q_2	^	Z_0
		accepted

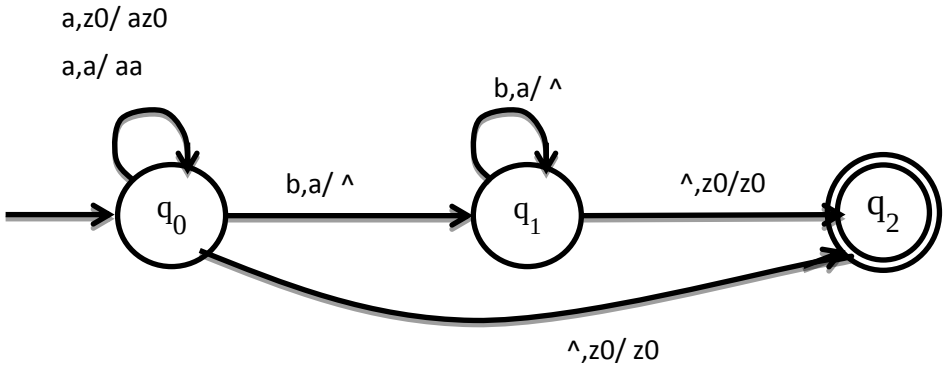


Fig. 4.4 PDA accepting Number of a's and b's are same.

6. Design and Draw a deterministic PDA Accepting "Balance string of brackets". OR Design and Draw a deterministic PDA for following grammar S→SS/[S]/{S}/∧

Move No.	State	i/p	Stack Symbol	Move
1	q_0	[	Z_0	$(q_1,[Z_0)$
2	q_0	{	Z_0	$(q_1,\{Z_0)$

3	q_1	{	{	$(q_1,\{\{)$
4	q_1	[	{	$(q_1,[\{)$
5	q_1	{	[	$(q_1,\{[)$
6	q_1	[	[	$(q_1,[[)$
7	q_1	]	[	$(q_1,^)$
8	q_1	}	{	$(q_1,^)$
9	q_1	^	Z_0	(q_0,z_0)

Table 4.4 Transition table for balanced string of parenthesis

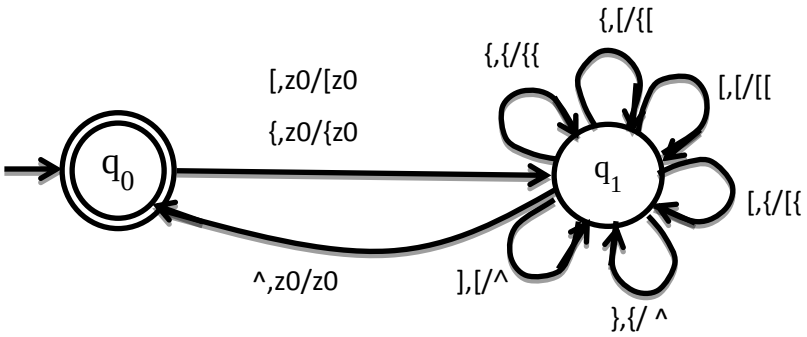


Fig. 4.5 PDA for balanced string of parenthesis

String: []{}

Resulting State	unread i/p	Stack
q_0	[]{}	Z_0
q_1	]{}	$[Z_0$
q_1	^{}	Z_0
q_0	{}	Z_0
q_1	}	$\{Z_0$
q_1	^	Z_0
q_0	^	Z_0

7. Design PDA to accept string with more a's than b's.

OR

Design PDA for given Language $L=\{x \in \{a,b\}^* / n_a(x) > n_b(x)\}$

Move No.	State	i/p	Stack Symbol	Move
1	q_0	a	Z_0	(q_1,aZ_0)
2	q_0	b	Z_0	(q_0,bZ_0)
3	q_0	a	b	$(q_0,^)$
4	q_0	b	b	(q_0,bb)
5	q_1	a	a	(q_1,aa)
6	q_1	b	a	$(q_0,^)$
7	q_0	^	a	(q_1,a)

Table 4.5 Transition table for more a's than b's

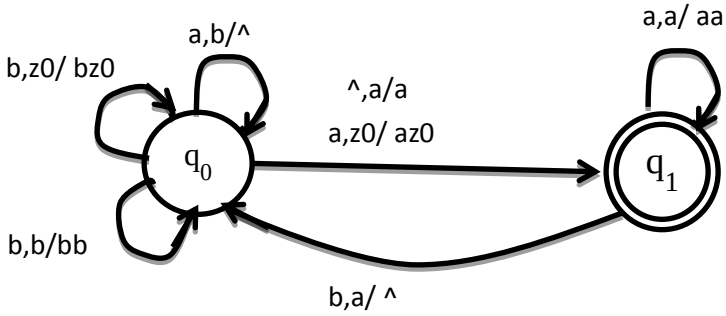


Fig. 4.6 PDA for more a's than b's

8. Design PDA for $\{a^n b^{n+m} c^m \mid n,m>=1\}$

Move No.	State	i/p	Stack Symbol	Move
1	q_0	a	Z_0	(q_1, aZ_0)
3	q_1	a	a	(q_1, aa)
4	q_1	b	a	$(q_2, ^)$
5	q_2	b	a	$(q_2, ^)$
6	q_2	b	Z_0	(q_3, bZ_0)
7	q_3	b	b	(q_3, bb)
8	q_3	c	b	$(q_4, ^)$
9	q_4	c	b	$(q_4, ^)$
11	q_4	^	Z_0	(q_5, Z_0)
All other combination				(none)

Table 4.6 Transition table for $a^n b^{n+m} c^m$

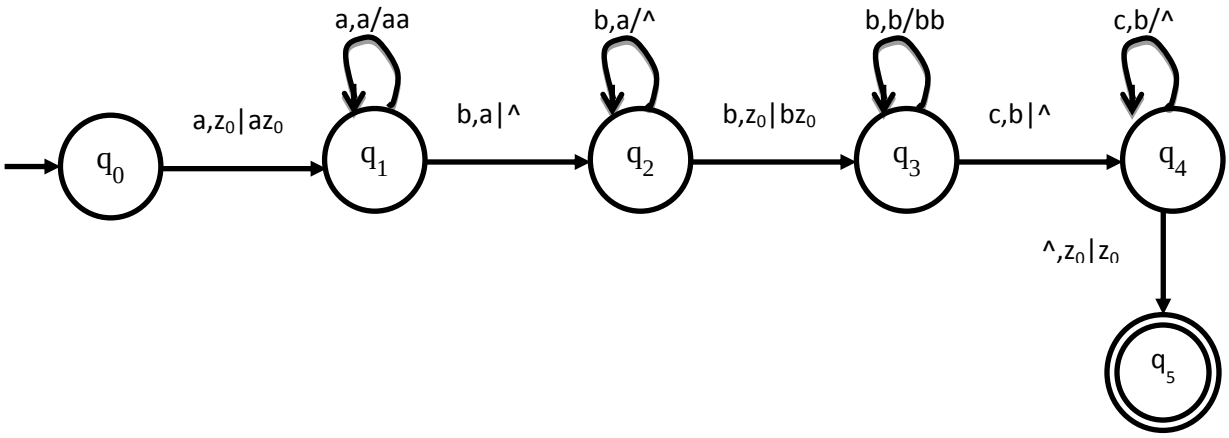


Fig. 4.7 PDA for $a^n b^{n+m} c^m$

9. Design PDA for $\{a^i b^j c^k \mid i,j,k>=1$ & j=i or j=k$\}$

Move No.	State	i/p	Stack Symbol	Move
1	q_0	a	Z_0	$(q_1, aZ_0)(q_4, z_0)$
2	q_1	a	a	(q1, aa)

3	q_1	b	a	(q_2, ^)
4	q_2	b	a	(q_2, ^)
5	q_2	c	Z_0	(q_3, Z_0)
6	q_3	c	Z_0	(q_3, Z_0)
7	q_3	^	Z_0	(q_7, Z_0)
8	q_4	a	Z_0	(q_4, Z_0)
9	q_4	b	Z_0	(q_5,bZ_0)
10	q_5	b	b	(q_5, bb)
11	q_5	c	b	(q_6, ^)
12	q_6	c	b	(q_6, ^)
13	q_6	^	Z_0	(q_7, Z_0)
All other combination				(none)

Table 4.7 Transition table for $a^ib^jc^k$,j=i or j=k

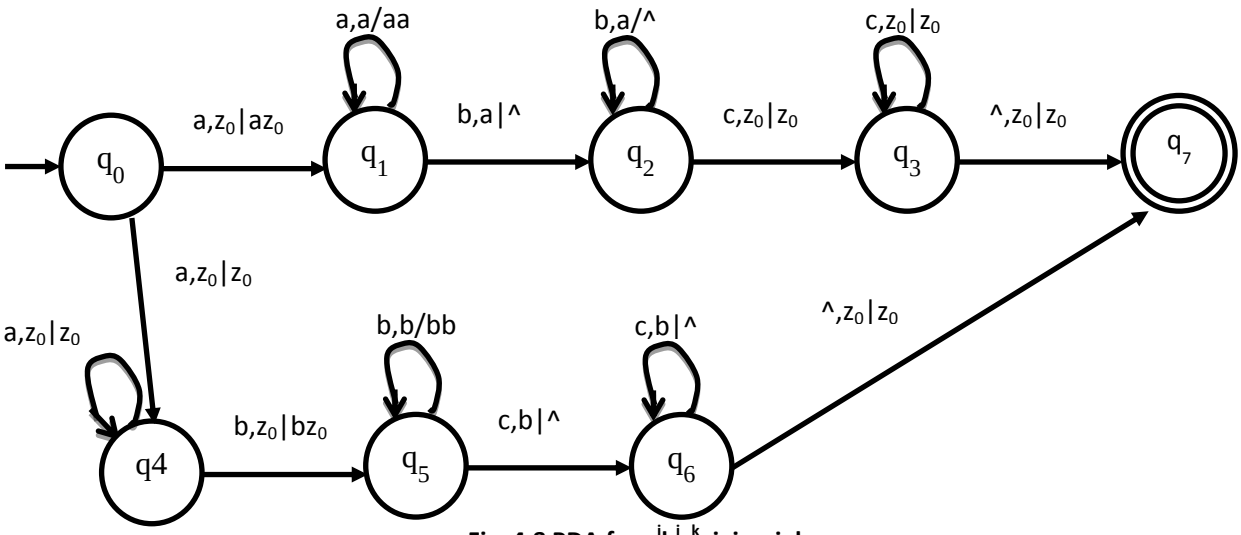


Fig. 4.8 PDA for $a^ib^jc^k$,j=i or j=k

10. Top-Down PDA corresponding to a CFG

- Let G=(V,Σ,S,P) be a CFG, we define M=(Q,Σ, ┌,q_0,Z_0,A, δ) as follow:
  Q={ q_0,q_1,q_2,} A={q_2} ┌=VUΣU{Z_0}
- The initial move of M is to place S on the stack and move to q_1:
  δ(q_0,^, Z_0)={(q_1, SZ_0)}
- The only move to the accepting state q_2 is fromq_1, when stack is empty except for Z_0: δ(q_1,^, Z_0)={ q_2, Z_0}
- Otherwise, the only moves of M are as follows
  1) for every A∈ V, δ(q_1,^,A)={ (q_1, α)/A→α is a production in G}
  2) foe every a∈Σ, δ(q_1,a,a)= (q_1,^)

Example:

S→a/aS/bSS/SSb/SbS

Move No.	State	i/p	Stack Symbol	Move
1	q_0	^	Z_0	(q_1,SZ_0)
2	q_1	^	S	(q_1,a) (q_1,aS) (q_1,bSS) (q_1,SSb) (q_1,SbS)
3	q_1	a	a	$(q_1,^)$
4	q_1	b	b	$(q_1,^)$
5	q_1	^	Z_0	(q_2, Z_0)

Table 4.8 Top down PDA equivalent to CFG

$(q_0,abbaaa, Z_0)$
$\vdash(q_1,abbaaa, SZ_0)$
$\vdash(q_1,abbaaa, SbSZ_0)$
$\vdash(q_1,abbaaa, abSZ_0)$
$\vdash(q_1,bbaaa, bSZ_0)$
$\vdash(q_1,baaa, SZ_0)$
$\vdash(q_1,baaa, bSSZ_0)$
$\vdash(q_1,aaa, SSZ_0)$
$\vdash(q_1,aaa, aSZ_0)$
$\vdash(q_1, aa,SZ_0)$
$\vdash(q_1,aa, aSZ_0)$
$\vdash(q_1, a, SZ_0)$
$\vdash(q_1, a, aZ_0)$
$\vdash(q_1,^, Z_0)$
$\vdash(q_2,^, Z_0)$

11. Bottom-Up PDA for simple Algebraic Expressions.

S→S+T
S→T
T→T*a
T→a

Move	Production	Stack	Unread i/p
		Z_0	a+a*a
shift	-	aZ_0	+a*a
reduce	T→a	TZ_0	+a*a
reduce	S→T	SZ_0	+a*a
shift	-	$+SZ_0$	a*a
shift	-	$a+SZ_0$	*a
reduce	T→a	$T+SZ_0$	*a
shift	-	$*T+SZ_0$	a
shift	-	$a*T+SZ_0$	-
reduce	T→a*T	$T+SZ_0$	-
reduce	S→S+T	SZ_0	-

(POP(S))		Z_0	-
accept			

Table 4.9 Processing a+a*a by Bottom up PDA

12. PDA to CFG

Set of production for equivalent CFG.

1. Add the following production for the start symbol S.
   $S \rightarrow [q_0\ z\ qi]$
   q_0 initial state
   z initial stake symbol
   $qi \in Q$
2. For each transition of the form
   $\delta(qi, a, B) \rightarrow (qj, C)$
   where $q_i, qj \in Q$
   $a \in (\Sigma \cup \wedge)$
   $B, C \in (\Gamma \cup \wedge)$
   Then for each $q \in Q$, add the production
   $[qi, B, q] \rightarrow a[qj, C, q]$
3. For transition of the form
   $\delta(qi, a, B) \rightarrow (qj, C1C2)$
   where $qi, qj \in Q$
   $a \in (\Sigma \cup \wedge)$
   $B, C1, C2 \in (\Gamma \cup \wedge)$
   Then for each $p1, p2 \in Q$, add the production
   $[qi, B, p1] \rightarrow a[qj\ C1\ p2][p2\ C2\ p1]$

13. Convert following PDA to CFG

state	input	stack	New state	stack
q	**1**	**z**	**q**	**xz**
q	**1**	**x**	**q**	**xx**
q	**^**	**x**	**q**	**^**
q	**0**	**x**	**p**	**x**
p	**1**	**x**	**p**	**^**
p	**0**	**z**	**q**	**z**

Solution:

Step 1: Add the production for the start symbol
$S \rightarrow [q\ z\ q]$
$S \rightarrow [q\ z\ p]$

Step 2: Add production for the $\delta(q, 1, z) \rightarrow (q, xz)$
$[q\ z\ q] \rightarrow 1[q\ x\ q][q\ z\ q]$
$[q\ z\ q] \rightarrow 1[q\ x\ p][p\ z\ q]$

[q z p]→1[q x q][q z p]
[q z p]→1[q x p][p z p]

Step 3: Add production for the δ(q, 1, x) →(q ,xx)

[q x q]→1[q x q][q x q]
[q x q]→1[q x p][p x q]
[q x p]→1[q x q][q x p]
[q x p]→1[q x p][p x p]

Step 4: Add the production for δ(q, ^, x) →(q ,^)

[q x q]→^

Step 5: Add the production for δ(q, 0, x) →(p ,x)

[q x q]→0 [p x q]
[q x p]→0 [p x p]

Step 6: Add the production for δ(p, 1, x) →(p ,^)

[p x p]→1

Step 7: Add the production for δ(p, 0, z) →(q ,z)

[p z q]→0 [q z q]
[p z p]→0 [q z p]

Step 8: Renaming variable name

Original name	New name
[q z q]	A
[q z p]	B
[p z q]	C
[p z p]	D
[q x q]	E
[q x p]	F
[p x q]	G
[p x p]	H

The set of production can be written as:

S→A|B
A→1EA|1FC
B→1EB|1FD
E→1EE|1FG
F→1EF|1FH
E→^
E→0G
F→0H
H→1
C→0A
D→0B

Step 9: simplification of grammar

Symbol G is not available on left side of grammar so, it can be eliminated.

S→A|B
A→1EA|1FC

B→1EB|1FD
E→1EE|^
F→1EF|1FH|0H
H→1
C→0A
D→0B

14. Intersection and complement of CFL (Non CFL)

Theorem:-There are CFLs L1 and L2 so that L1∩ L2 is not a CFL, and there is a CFL L so that L' is not a CFL.

we observed that

$L= \{ a^i b^j c^k | \ i < j \text{ and } i < k \}$

is not context-free. However, although no PDA can test both conditions i < j and i< k simultaneously, it is easy enough to build two PDAs that test the conditions separately. In other words, although the intersection L of the two languages

$$L_1 = \{a^i b^j c^k \mid i < j\}$$

and

$$L_2 = \{a^i b^j c^k \mid i < k\}$$

is not a CFL, both languages themselves are. Another way to verify that L_1 is a CFL is to check it is generated by the grammar with productions

S→ABC A→aAb|^ B→bB|b C→cC|^

Similarly, $L_2$is generated by the CFG with productions

S→AC A→aAc|B B→bB|^ C→cC|c

The second statement in the theorem follows from the first and the formula

$$L_1 \cap L_2 = (L_1' \cup L_2')'$$

If complements of CFLs were always CFLs, then for any CFLs L_1 and L_2, the language $L_1'$and L_2' would be CFLs, so would their union, and so would its complement. We know now that this is not the case.

15. Give the difference between Top Down & Bottom Up parsing.

Top Down Parsing	Bottom Up Parsing
• A parser is top-down if it discovers a parse tree top to bottom • A top down parser for a given grammar G tries to derive a string through a sequence of derivation starting with a start symbol. • Top down parsing methods are: ✓ Top down parsing(with backtracking/without backtracking) ✓ Recursive decent parser	• A parser is bottom up if it discovers a parse tree bottom to top • In bottom up parsing, the source string is reduced to the start symbol of the grammar. Bottom up parsing method is also called shift reduce parsing. • Bottom up parsing methods are: ✓ Naïve bottom up parsing ✓ Operator precedence parsing

✓ LL(1) parser	✓ LR parsing

Table 4.10 Difference between top down and bottom up parsing

16. Pumping lemma for CFG

Suppose L is a context-free language. Then there is an integer n so that for every u ∈ L with |u| ≥ n, thereare strings v, w, x, y, and z satisfying

1. u = vwxyz
2. |wy| > 0
3. |wxy| ≤ n

for every m ≥ 0, $vw^mxy^mz \in L$

1. What is Turing machine? Explain its capabilities.

A Turing machine is a 5-tuple T=(Q,∑, Γ,q_0, δ) where,

- Q is a finite set of states not to contain h_a and h_r.
- ∑ and Γ are finite sets, the input and tape alphabets respectively.
- q_0 is initial state.
- δ is a partial function.

Capabilities

- Turing machine is capable of performing computations on inputs and producing a new result. An abstract model of a Turing machine is shown below.

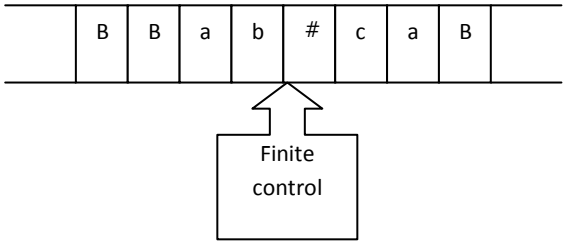


Fig. 5.1 Abstract model of Turing machine

- Input to Turing machine is provided through a long tape. Turing machine provided with read/write head.
- The tape is divided into square; each square holds a single symbol.
- The head is capable of performing 3 operations:
  1. Reading a symbol being scanned.
  2. Modifying a symbol being scanned.
  3. Shifting either to previous square or next square.

2. Define Universal Turing Machine

- A general purpose computer can be programmed to solve different types of problem.
- A TM can also behave like a general purpose computer.
- A universal Turing machine is a Turing machine *Tu* that works as follows.
- It is assumed to receive an input string of the form *e(T)e(z)*, where *T* is an arbitrary TM, *z* is a string over the input alphabet of *T* , and *e* is an encoding function whose values are strings in {0,1}*. The computation performed by *Tu* on this input string satisfies following two properties:
  1. *Tu* accepts the string *e(T)e(z)* if and only if *T* accepts *z*.
  2. If *T* accepts *z* and produces output *y*, then *Tu* produces output *e(y)*.
- UTM should be able to simulate every Turing machine. Simulation of a Turing will involve:
  1. Encoding behavior of a particular TM as a program.
  2. Execution of the above program by UTM.
- We can assume the UTM as a 3-tape Turing machine.
  1. Input is written on the first tape.
  2. Moves of the TM is encoded is written on the second tape.
  3. The current state of TM is written on third tape.

3. Explain Church Turing Thesis.

- Turing machine is a general model of computation means that any algorithmic procedure that can be carried out at all, by a human computer or a team of humans or an electronic computer, can be carried out by a TM. This statement usually referred to as Church's thesis, or the Church-Turing thesis.
- It is not a mathematically precise statement that can be proved, because we do not have a precise definition of the term *algorithmic procedure*. By now, however, there is enough evidence for the thesis to have been generally accepted. Here is an informal summary of some of the evidence.
  1. The nature of the model makes it seem likely that all the steps crucial to human computation can be carried out by a TM. Humans normally work with a two-dimensional sheet of paper. A TM tape could be organized so as to simulate two dimensions; one likely consequence would be that the TM would require more moves to do what a human could do in one.
  2. Various enhancements of the TM model have been suggested in order to make the operation more like that of a human computer, or more convenient, or more efficient. The multitape TM is an example.
  3. Other theoretical models includes abstract machines with two stacks or with a queue, as well as machines that are more like modern computers.
  4. Since the introduction of the Turing machine, no one has suggested any type of computation that ought to be included in the category of "algorithmic procedure" and cannot be implemented on a TM.

4. Design a TM Accepting {a,b}*{aba}

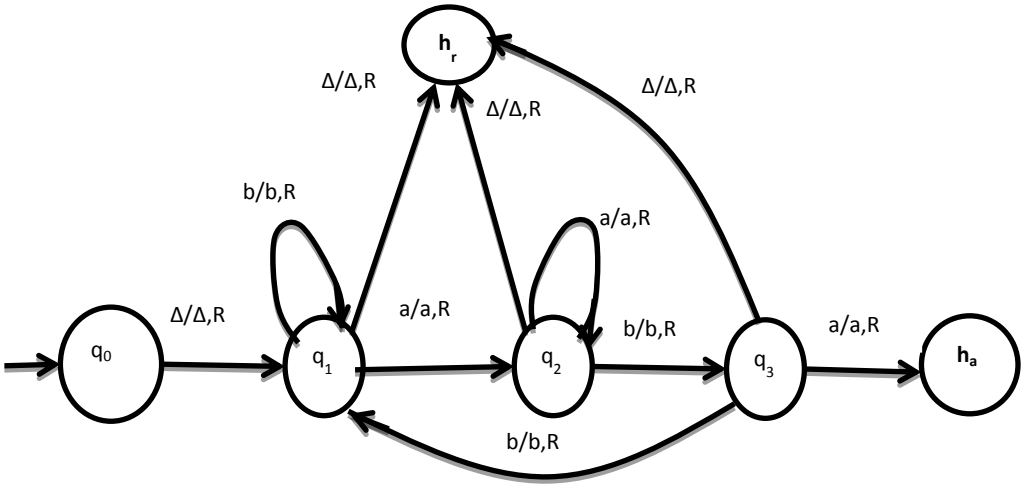


Fig. 5.2 TM accepting {a+b}*aba

5. Design a TM for accepting Palindromes for odd and even length.

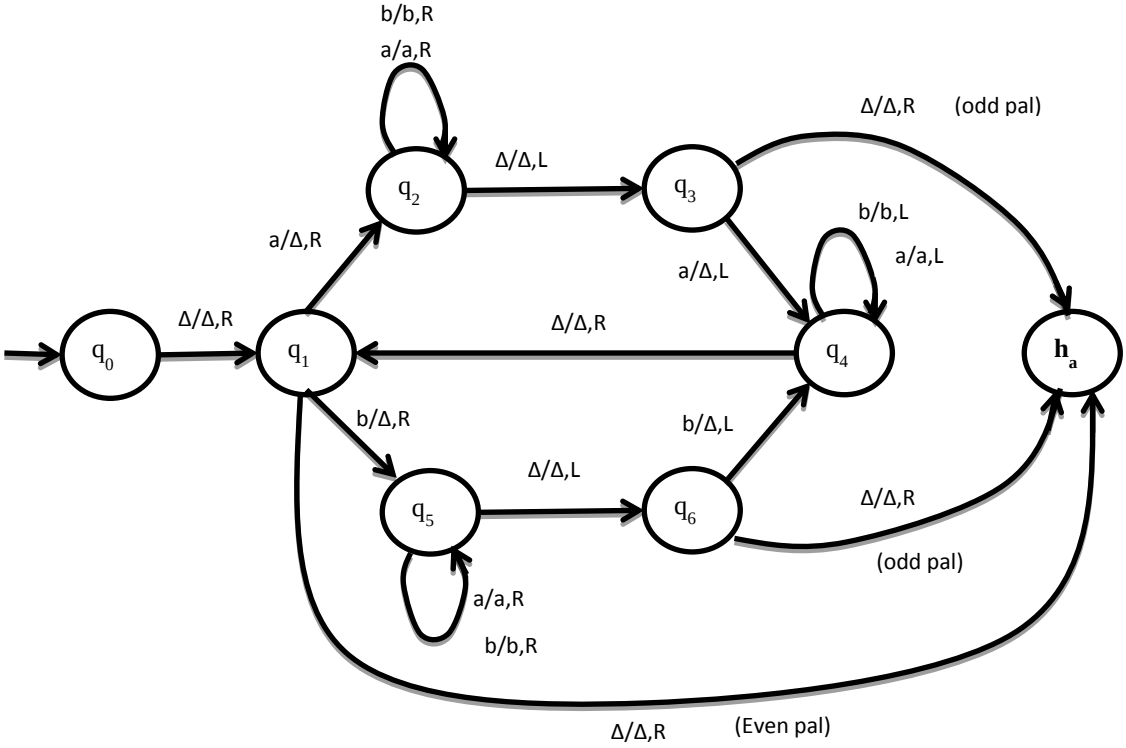


Fig. 5.3 TM accepting palindrome string

6. Design a Turing machine for accepting $\{a^n b^n c^n / n>=0\}$

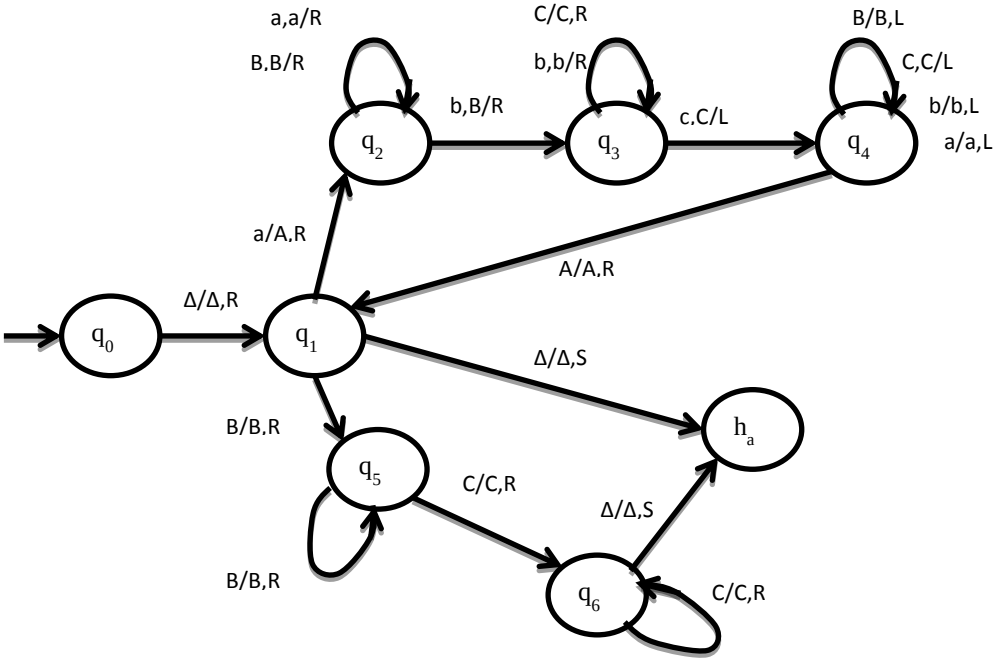


fig.5.4 TM accepting $a^n b^n c^n$

7. Design a TM for Accepting {SS | S ∈ {a,b}*}

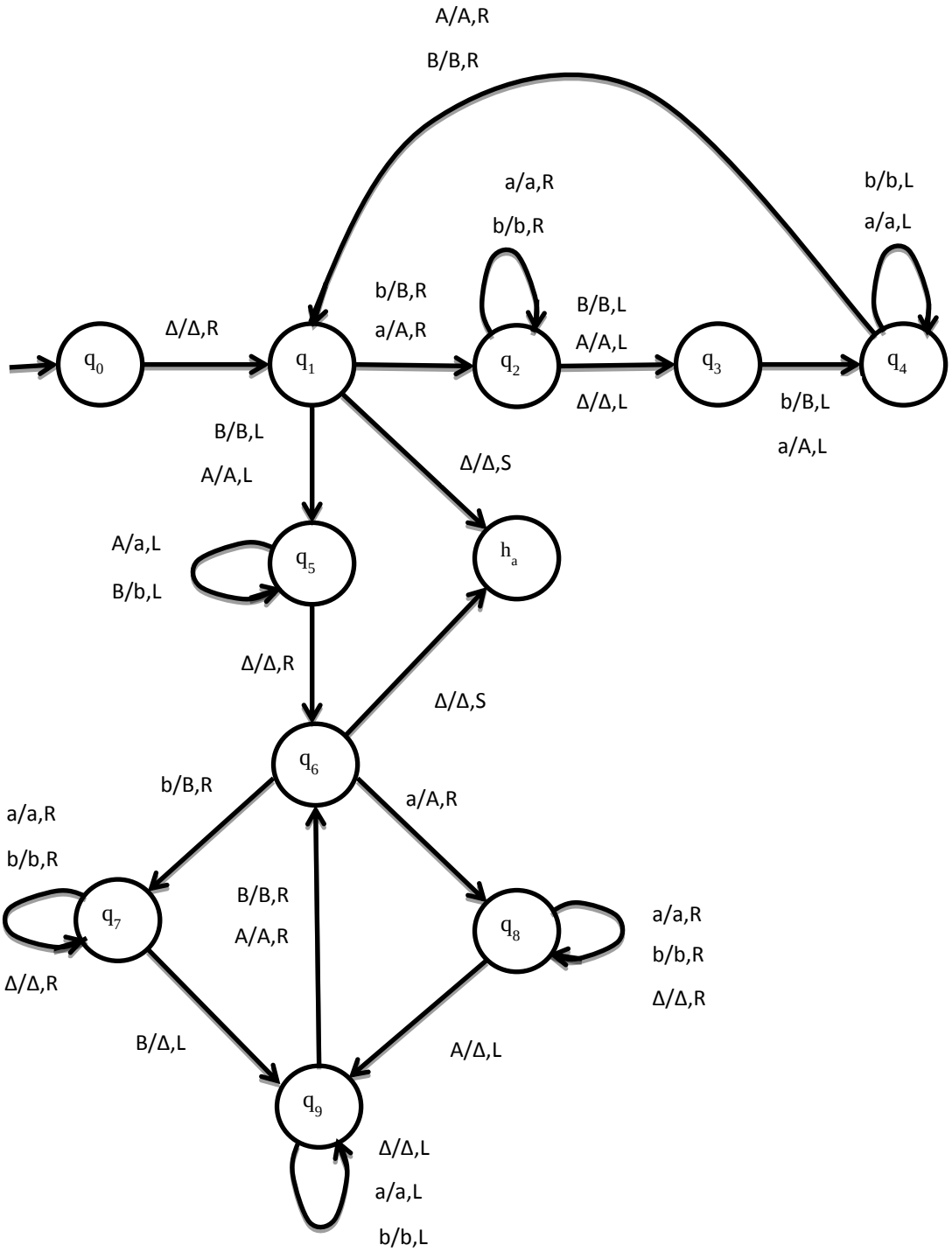


Fig. 5.5 TM accepting {ss | s ∈ {a,b}*}

8. Design a Turing machine to copy a string.

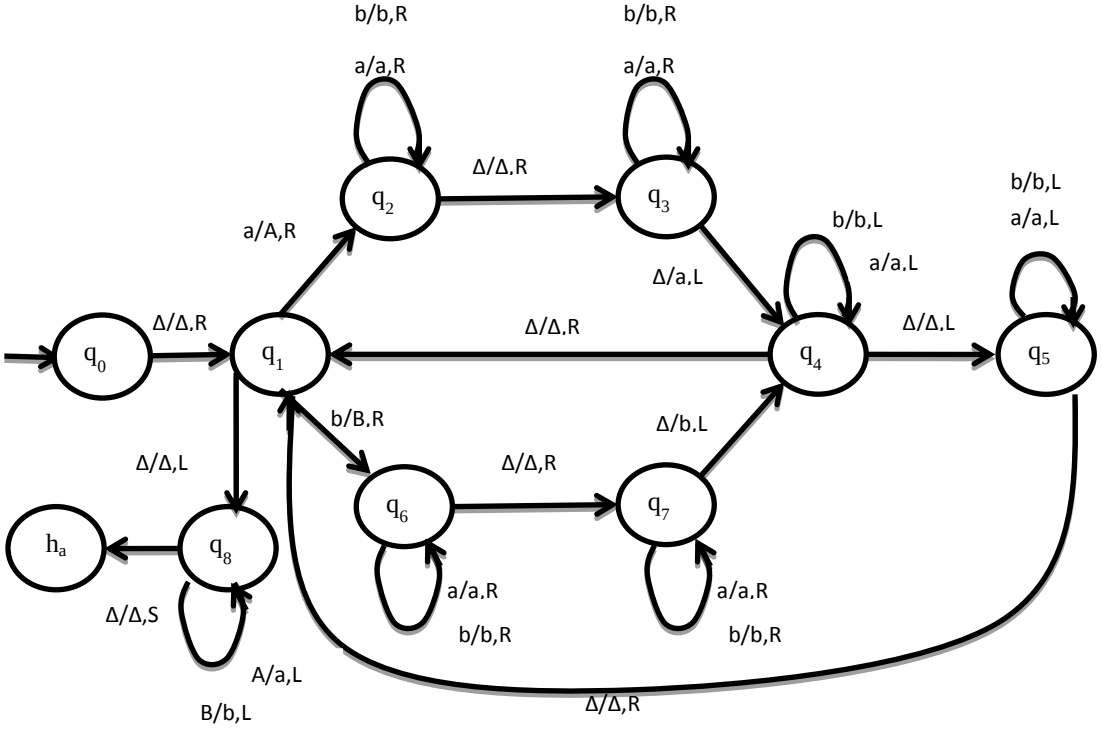


Fig.5.6 TM to copy strings

9. Design Turing Machine for Accepting {x∈{a,b,c}* / n a(x)=n b(x)=nc(x)}

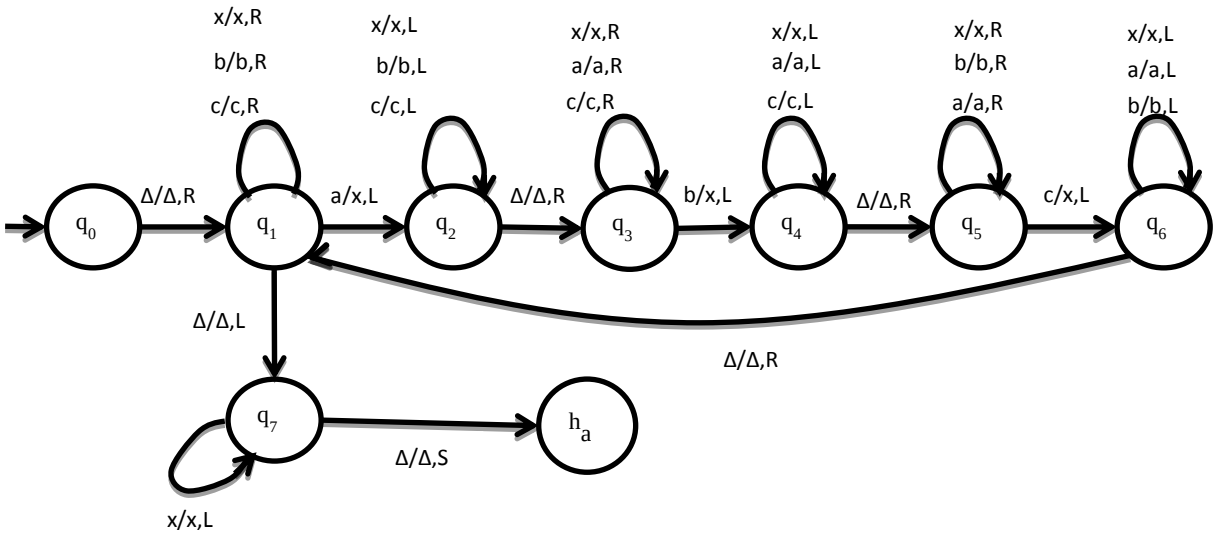


Fig.5.7 TM to accept na(x)=n b(x)=nc(x)

10. Design a Turing machine to delete a symbol

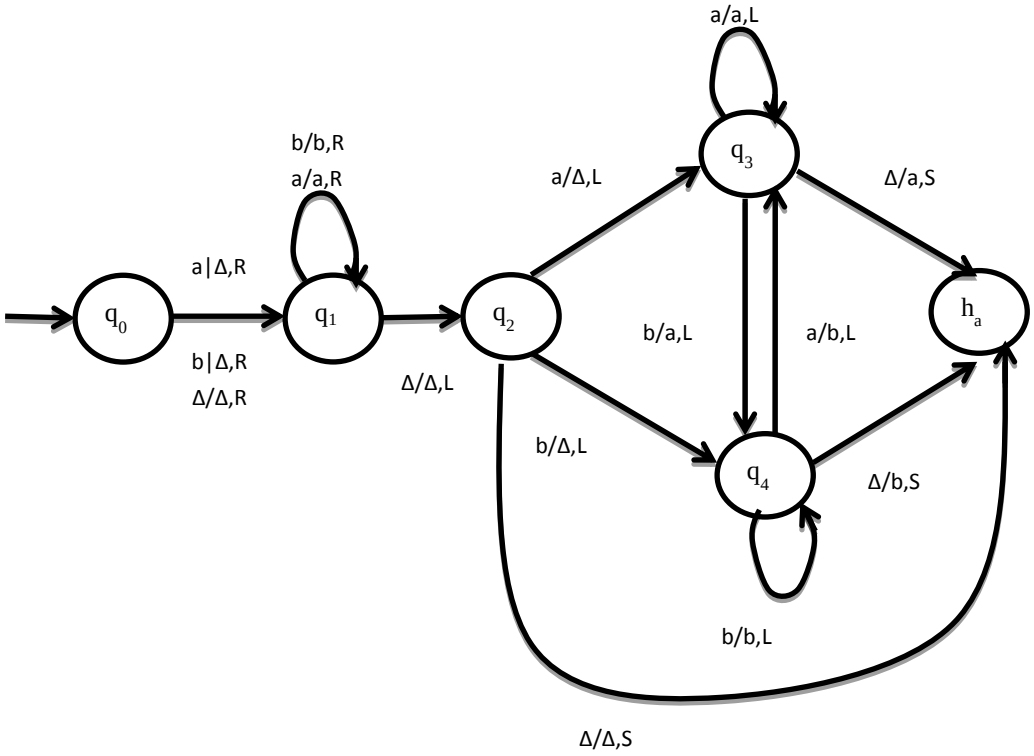


Fig. 5.8 TM to delete a symbol

11. Define: TM Enumerating a Language

Let T be a k-tape Turing machine for some $k \geq 1$, and let $L \subseteq \sum$. We say T enumerates L if it operates such that the following conditions are satisfied.

1. The tape head on the first tape never moves to the left, and no nonblank symbol printed on tape 1 is subsequently modified or erased.
2. For every $x \in L$, there is some point during the operation of T when tape 1 has contents

   $$x1\#x2\# \ldots \#xn\#x\#$$

   for some $n \geq 0$, where the strings $x1, x2, \ldots, xn$ are also elements of L and $x1, x2, \ldots, xn, x$ are all distinct. If L is finite, then nothing is printed after the # following the last element of L.

12. Define: Context Sensitive Grammars

A context sensitive grammar is an unrestricted grammar in which every production has the form

$$\alpha \rightarrow \beta \text{ with } |\beta| >= \alpha$$

A context sensitive language (CSL) is a language that can be generated by such a grammar.

1. Define the following terms:

Initial Function

The initial function are the following:

1. Constant functions: For each k ≥ 0 and each a ≥ 0, the constant function C_a^k : $N^k \rightarrow N$ is defined by the formula

   $$C_a^k(X) = a \qquad \text{for every } X \in N^k$$

   In the case k = 0 we may identify the function C_a^k with the number a.
2. The successor function s : N → N is defined by the formula

   $$s(x) = x + 1$$
3. Projection functions: For each k ≥ 1 and each i with 1 ≤ i ≤ k, the projection function p_i^k:$N^k \rightarrow N$ is defined by the formula

   $$p_i^k(x_1, x_2, \dots, x_i, \dots, x_k) = x_i$$

Composition

Suppose f is a partial function from N^k to N, and for each i with 1 ≤ i ≤ k, g_i is a partial function from N^m to N. The partial function obtained from f and $g_1, g_2, \dots, g_k$ by composition is the partial function h from N^m to N defined by the formula

$$h(X) = f(g_1(X), g_2(X), \dots, g_k(X)) \qquad (X \in N^m)$$

Primitive Recursion Operation

Suppose n ≥ 0, and g and h are functions of n and n + 2 variables, respectively. The function obtained from g and h by the operation of primitive recursion is the function f : $N^{n+1} \rightarrow N$ defined by the formulas

$$f(X,0) = g(X)$$

$$f(X, k + 1) = h(X, k, f(x, k)) \text{ for every } X \in N^n \text{ and every } k \geq 0.$$

Primitive Recursive Functions

The set PR of primitive recursive functions is defined as follows.

1. All initial functions are elements of PR.
2. For any k ≥ 0 and m ≥ 0, if f : $N^k \rightarrow N$ and $g_1, g_2, \dots, g_k$: $N^m \rightarrow N$ are elements of PR, then the function $f(g_1, g_2, \dots, g_k)$ obtained from f and $g_1, g_2, \dots, g_k$ by composition is an element of PR.
3. For any n ≥ 0, any function g : $N^{n+1} \rightarrow N$ in PR, and any function h : $N^{n+2} \rightarrow N$ in PR, the function f : $N^{n+1} \rightarrow N$ obtained from g and h by primitive recursion is in PR.
4. No other functions are in the set PR.

Bounded Quantifications

Let P be an (n + 1)-place predicate. The bounded existential quantification of P is the (n + 1)-place predicate E_p defined by

$$E_p(X, k) = \text{(there exists y with } 0 \leq y \leq k \text{ such that P(X, y) is true)}$$

The bounded universal quantification of P is the (n + 1)-place predicate A_P defined by

$$A_P(X, k) = \text{(for every y satisfying } 0 \leq y \leq k, \text{ P(X, y) is true)}$$

Unbounded Minimalization

If P is an (n + 1)-place predicate, the unbounded minimalization of P is the partial function M_P : $N^n \rightarrow N$ defined by

$$M_p(X) = \min \{y \mid P(X, y) \text{ is true}\}$$

$M_p(X)$ is undefined at any $X \varepsilon N^n$ for which there is no y satisfying P(X, y).
The notation μy[P(X, y)] is also used for $M_p(X)$. In the special case in which P(X, y) = (f(X, y) = 0), we write $M_p = M_f$ and refer to this function as the unbounded minimalization of f.

Bounded Minimalization

For an (n + 1)-place predicate P, the bounded minimalization of P is the function $m_p : N^{n+1} \rightarrow N$ defined by

$$m_p(X, k) = \begin{cases} \min \{y \mid 0 \le y \le k \text{ and } P(X, y)\} & \text{if this set is not empty} \\ k + 1 & \text{otherwise} \end{cases}$$

The symbol μ is often used for the minimalization operator, and we sometimes write

$$m_P(X, k) = \mu^k y[P(X, y)]$$

An important special case is that in which P(X, y) is (f(X, y) = 0), for some $f : N^{n+1} \rightarrow N$. In this case m_p is written m_f and referred to as the bounded minimalization of f.

μ – Recursive Functions

The set M of μ - recursive, or simply recursive, partial functions is defined as follows.

1. Every initial function is an element of M.
2. Every function obtained from elements of M by composition or primitive recursion is an element of M.
3. For every n ≥ 0 and every total function $f : N^{n+1} \rightarrow N$ in M, the function $M_f : N^n \rightarrow N$ defined by
$$M_f(X) = \mu y[f(X, y) = 0] \text{ is an element of M.}$$
4. No other functions are in the set M.

Gödel Number of a sequence of Natural Numbers

For any finite sequence $x_0, x_1, \dots, x_n$ of natural numbers, the Gödel Number of the sequence is the number

$$gn(x_0, x_1, \dots, x_n) = 2^{x0}3^{x1}5^{x2} \dots (PrNo(n))^{xn}$$

Where PrNo(n) is the n^{th} prime.

2. Theorem: Let f: $\Sigma_1^* \rightarrow \Sigma_2^*$. Then f is computable if and only if f is μ-recursive.

Proof:

- As above, we denote by g_1 and g_2 the Godel-numbering function for the two alphabets, and by g_1' and g_2' their respective left inverse.
- Suppose on the hand that f is computable, and let T_f be a TM (whose tape alphabet contains the symbols of Σ_1 and Σ_2) computing f.
- We want to show that $p_f : N \rightarrow N$ is μ-recursive, it is sufficient to show that p_f that a composite Turing machine can be constructed form three simpler ones: one that takes the initial tape $\Delta 1^n$ and halts with tape $\Delta\, g_1'$; the TM T_f, which then halts with tape $\Delta f(g_1'(n))$; and a third that calculates the Godel number of this string to yield the result, $g_2(f(g_1'(n))) = \rho_f(n)$.
- On the other hand, suppose that f is μ-recursive. Then by definition, p_f is μ-recursive. Let T be a TM computing it. From the formula for p_f we have
$$g_2'(\rho_f(n)) = g_2'(g_2(f(g_1'(n))))$$
$$= f(g_1'(n))$$

- for any n. applying this formula when n= $g_1(x)$, we obtain
  $f(x)=f(g_1'(g_1(x)))= g_2'(\rho_f(g_1(x)))$
- Just as before, we can now construct a composite TM T_f to perform this computation, and it follows that f is computable.