Lecture notes on Automata Theory and Computability(subject code: 15CS54) – Module -1: By Prof B I Khodanpur, DSCE

Module – 1: Syllabus:-

Why study the theory of computation(ch-1)

Languages and strings(ch-2)

A Language Hierarchy(ch-3)

Computation(ch-4)

Finite State Machines(ch-5 from 5.1 to 5.10)

Why study the theory of computation(ch-1)

Defn: Automata is an abstract machine for modelling computations.

Why Abstract machines?

Abstract machine allows us to model the essential parameters, and ignore the non-essential parameters.

What is computability?

It is very difficult to define, but Our notion of computation: Examples are
Add 2 numbers
Find the roots of a quadratic equation
Multiply 2 matrices
And so on…..
Important to note that: **all the above have algorithms**
What is not computable: Example-

- **Halting problem of a program**:

simply write a program that examines other programs to determine if they halt or loop forever. Obviously whether or not a program halts depends on the data it is fed so in this case we mean program to be code plus the data it operates on.

- **Why it not computable**:

simple answer – **No algorithm exists**

Some computations take lot of time to be meaning full: Example

Travelling salesman problem

- **When computations are not finished within a reasonable time, such computations are useless, also known as NP-problem(non-deterministic polynomial problems)**

Tractable/Intractable Problems:

Tractable Problem: a problem that is solvable by a polynomial-time algorithm. The upper bound is polynomial. Examples: Quick sort(O(nlogn)
Intractable Problem: a problem that cannot be solved by a polynomial-time algorithm. The lower bound is exponential. Examples: Travelling Salesman problem

Some important applications of automata theory

- **In compilers**:
  Lexical analysis
  Parser generators
- **Modelling the circuits**: Example On-Off switch.

Some important applications of automata theory in general:

Word search and Translation of Natural Languages
Parity checkers, Vending machines, communication protocols
Video games
DNA
Security
Artificial Intelligence
To model organic structures of molecules
Fluid Flow

Snowflake and crystal formation
Chaos theory
Cosmology
Financial analysis

Why not use English to Program?

- Firstly all Natural Languages like English, Kannada etc are **Context Sensitive Languages**
- **That is to say – meaning depends on the context.**
- Example: Take a English word " Charge "
- There are many meanings for this word
- Like - Cost, -Flight, - Charge the Battery
- - Positive Charge, etc

Characteristics of Natural Languages:

- In most of the situations – meaning depends on the context.
- They are developed for communication among the Human beings.
- Human beings are capable or trained to interpret a sentence depending on the situations.
- **Where as, Machine are not in Context.**
- **Machine will not be able to interpret depending on the situation.**

Characteristics of Formal Languages:

- Meaning of a word or sentence does not depend on the context.
- Words and sentences have only one meaning irrespective of the context.
- They are simple.
- **Easy to write Compilers and Interpreters**
- **They are precise in their meaning.**
- **With this Machine do what they are instructed to do**

What is the gist of this subject?

A systematic way of depicting the problem so that it solution can be understood and analysed.

What are the properties of various types of languages.

Regular Languages(RL)
Context Free languages(CFL)
Context Sensitive Languages(CSL)
Recursively Enumerable Languages(REL)

Various types of Automata will be studied:

- **There are different types of automata for recognizing different languages**
- Deterministic Finite Automata - RL
- Pushdown Automata – CFL

- Linear Bounded Automata – CSL
- Turing Machine – REL

How to study:

- Subject is mathematical and lot of logical thinking is required.
- There are number of Theorems and proofs.
- Understand the definition – mathematically i.e. Examples are not substitute for definitions.
- Examples are only to make the definition clear.
- Work out number of problems from various other books.
- Key to understanding this subject – attempt to work harder problems even if you are not able get answers.
- **If you plan to take up - Gate examination for PG studies – you must understand it thoroughly.**

Languages and Strings(chapter-2)

Alphabet - Σ definition:
Defn: An alphabet is a non-empty, finite set of characters/symbols
Use Σ to denote an alphabet set
Examples
Σ= { a, b }
Σ= { 0, 1, 2 }
Σ = { a, b, c,…z, A, B, … Z }
Σ = { #, $, *, @, & }
String definition: A *string* is a finite sequence, possibly empty, of characters drawn from some alphabet Σ.
ε is the empty string
Σ* is the set of all possible strings over an alphabet Σ.

Examples of strings:
Σ = {a, b}
Strings derived from Σ are…..
….. ε , a, b, aa, ab, ba, bb, aaa, aab, aba, ..
Σ = {0, 1}
Strings derived from Σ are…..
….. ε , 0, 1, 00, 01, 10, 11, 000, 001, 010, ..
Σ = {a}
Strings derived from Σ are…..
…… ε, a, aa, aaa, aaaa, aaaaa, aaaaa,….

Functions on Strings

Length – to find the length of a string Operator used | |

Concatenation – to join two or more strings. Operator - s||t, or nothing i.e. st

Replication – strings raised to some power. Operator - a^3

Reversal – reverse a string

Operator - $(w)^R$

Examples of Length of a string

- **| ε | = 0**
- **|101| = 3**
- **|VTU_Edusat| = 10**

Examples of Concatenation of a string

- **x = good, y = student**
- **Concatenation operation x||y or xy**
- **xy = goodstudent**

Examples of Replication of a string

- **a^3 = aaa**
- **$(good)^3$ = goodgoodgood**
- **$a^0 b^3$ = ε bbb = bbb**

Examples of Reversal of a string

- **$(abc)^R$ = cba**
- **x= ab, y=cd, $(xy)^R$ = dcba**
- **x^R y^R =badc**

Relation on Strings

- **Substring:**
- **aaa is substring of aaa and also aaabbccc**
- **Proper substring:**

Defn: A string *s* is a proper substring of a string *t* iff *s* is a substring of *t* and *s* ≠*t*

Examples:

S = good then proper substrings are ..

..... ε, g, go, goo only

<u>Prefix and Suffix functions</u>

- **A string s is a prefix of t iff ∃x ϵ Σ*(t = sx)**
- **ε, a, ab,abb are prefixes of string abb**
- **Proper prefix:**
- **ε, a, ab, are proper prefixes of string abb**
- **A string s is a suffix of t iff ∃x ϵ Σ*(t = xs)**
- **ε, b, bb, abb are suffixes of string abb**
- **Proper suffix:**
- **ε, b,bb,are proper suffixes of string abb**

Languages:

Defn: A language is (finite or infinite) set of strings over a finite alphabet Σ

Example if Σ = { a } following languages can be derived

- **Language L1= {a, aaa, aaaaa, aaaaaaa,.....}**
- **Language L2= {ε, aa, aaaa, aaaaaa,.....}**
- **Language L3= {a, aaaaa, aaaaaaaaa,.....}**
- **Language L4= {a, aaa, a^7, a^9 , a^{13} , ...}**

Note: number of languages that can be derived even from singe alphabet set is INFINITE

Techniques for defining Languages by enumeration/defining property

Examples: (by enumeration)

- **Let L = {w ϵ {a, b}* : all string begin with a}**
- **L={a, ab, aab, abbbb, ...}**
- **Strings not in L are:**
- **{b, ba, ε, bbbbb, baaaaaa, ...}**
- **Let L = {w ϵ {a}* : |w| is even}**
- **L={ε, aa, aaaa, aaaaaa, aaaaaaaa, ..}**
- **Strings not in L are:**
- **{a, aaa, aaaaa, aaaaaaa, ...} //odd no of a's**

Examples: (defining property)

- **Let L = {w ϵ {a, b}* : all string ending in a}**
- **L={a, aba, aaba, bbbba, ...}**
- **Strings not in L are:**
- **{b, bb, ε, bbbbb, aaaaaab, ...}**
- **Let L = {w ϵ {a}* : |w| mod 3 =1}**
- **L={a, a^4, a^7, a^{10}, ...}**
- **Strings not in L are:**
- **{ε, a^2,a^3, a^5, a^6, a^8, a^9, ...}, ...}**

Functions on Languages.

Languages are sets. Therefore, all set operations like Union, Intersection, Difference, and Complement can be applied.

- **Example if Σ = { a }**
- **L1 = {ε, a^2,a^4, a^6, a^8, a^{10}, a^{12}, ...} //even no of a's**

L2 = {a^1,a^3, a^5, a^7, a^9, a^{11}, ...} //add no of a's

Set Operations on Languages

- **L1 = {ε, a^2,a^4, a^6, a^8, a^{10}, a^{12}, ...} //even no of a's**
- **L2 = {a^1,a^3, a^5, a^7, a^9, a^{11}, ...} //add no of a's**
- **L1 U L2 = Σ* or { a }* // union operation**
- **L1 ∩ L2 = Φ or {} // intersection operation**
- **L1 - L2 = L1 // difference operation**

- **L2 – L1 = L2 // difference operation**
- **~(L1 - L2) = L2 // complement operation**
- **~(L2 – L1) = L1 // complement operation**

Concatenation of Languages

- **L1 = {aa, ab}**
- **L2 = {xx, yy}**
- **L1L2={aaxx, aayy, abxx, abyy}**

Some important results

- **L1 = { } =Φ**
- **L2 = {xx, yy}**
- **L1L2={}**
- **In general for L**

L Φ = ΦL = Φ

Some important results

- **L1 = {ε}**
- **L2 = {xx, yy}**
- **L1L2=L2**
- **In general for all L**
- **L {ε} = L {ε} = L**
- **(L1L2)L3= L1(L2L3) // associative**
- **L1 = { a^n | n >= 0}**
- **L2 = { b^n | n >= 0}**
- **L1L2= {a^n b^m | n,m >=0} = a*b* // note n & m**
- **Kleene star operation**
- **L*={ set of all strings that can be formed by concatenating zero or more strings from L}**
- **a* = {ε, a, aa, aaa, aaaa, aaaaa, ...infinite}**

What is L^+ ?

- **L^+ = LL* //assuming L does not have Ɛ**
- **L^+ = L* - {ε}**

Example

a* = {ε, a, aa, aaa, aaaa, aaaaa, ...infinite}

a^+ = a* - {ε}

Assigning Meaning to the strings of a Language

Following codes of C/Java have the same meaning.

-- int x=4; x++;

-- int x=4; ++x;

-- int x=4; x=x+1;

-- int x=4; x=x-(-1)

chapter-5

Finite State Machines(FSM)

Defn: A FSM(DFSM) , M is a quintuple:

(K, ∑, δ, s, A)

- **K is a finite set of states,**
- **∑ is the input alphabet,**
- **s ϵ K is the start state**
- **A subset of K is the set of accepting states and**
- **δ is the transition function it maps from:**

  k x ∑ to k

Finite State Machines(FSM)

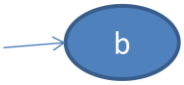
- **How to draw the Transition diagram of FSM**

Notations used:

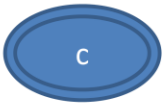
- Notation for state

- Notation of transition:

- Notation for start state:



- Notation for accepting state:
  (note 2 concentric circles)



Finite State Machines(FSM)

On any input if FSM reaches any of the states of A, i.e. accepting states, then the input strings is accepted by FSM M.

Examples:

- **Problem_1: Write a FSM to accept L, where**
- **L = {w ϵ {a,b}* | w contains a}**
- **L = {a, aa, aaa, baa, baaabbb,……}**
- **~L = {ε, b, bb, bbb, bbbb,…….}**
- **All strings in L should reach any - A state**

All strings in ~L should not reach any –A state
How to write a Transition Diagram: steps are…
Find the minimum string accepted, this decides the no of states in the FSM, in most of the cases
Then, take longer strings and make them accepted, while modifying the transitions,
Check for minimum strings that are not to be accepted, are really not accepted as per the transition diagram.
See that each state has transitions equal to the no of alphabets present.
Two transition on the same alphabet do not go to different states.

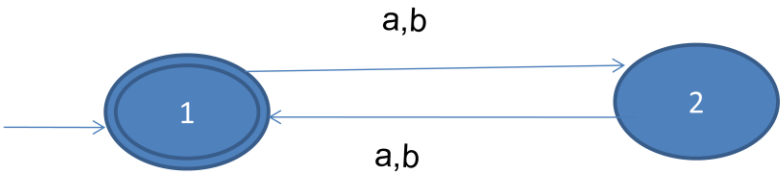
<u>Solution to the problem-1</u>

- **L = {a, aa, aaa, baa, baaabbb,……}**
- **~L = {ε, b, bb, bbb, bbbb,…….}**
- **Whenever a string from L is input, it should land in final state.**
- **Whenever a string from ~L is input, it should not land in final state, it can be in any other state.**

<u>Problem – 2:</u>
Write a DFSM to accept the language
L = { w ϵ {a, b}* | |w| is even length}
Step 1: Write strings accepted by L i.e.
L = { ε, aa, bb, ab, ba, aaaa, bbbb, bbaa, baba,..}
(note : ε is even, because its length is 0, which is even)
~L= { a, b, aaa, bbb, aba, bab, bba, aab, aabbb,..}
Step 2: since min string are {ε, aa}, 2 states are required.
Step 3: Write Transition Diagram.

Problem - 2

- **Transition Diagram:**



- **L = { ε, aa, bb, ab, ba, abab, aabb, bbaa, baba,..}**
- **~L= { a, b, aaa, bbb, aba, bab, bba, aab, aabbb,..}**

Problem - 2

- **Transition table: δ is transition function**
- **maps δ:** k x ∑ to k

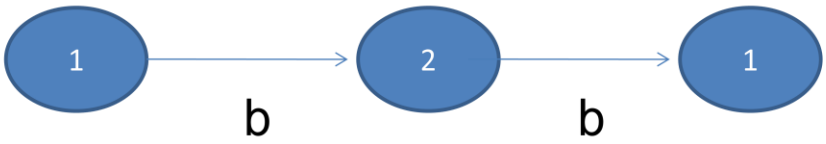
δ	a	b
-> * 1	2	2
2	1	1

Problem - 2

- **To show some strings are accepted: How**

Example : show string bb is accepted.

Draw the states reached by inputting one character at a time, as shown



Since reading all the characters state 1 , is reached , which is final state, Therefore string bb is accepted

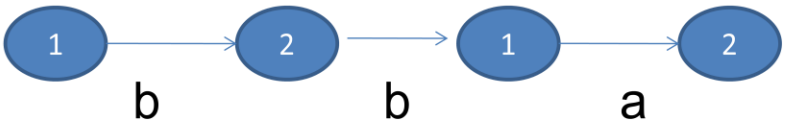
Problem – 2

Problem - 2

- **To show some strings are not accepted: How**

Example : show string bba is accepted.

Draw the states reached by inputting one character at a time, as shown



Since reading all the characters state 2 , is reached , **which is not a final state**, Therefore string bba is not accepted

Problem – 3

Write a DFSM to accept the language

L = { w ϵ {a, b}* | ab is a substring of w}

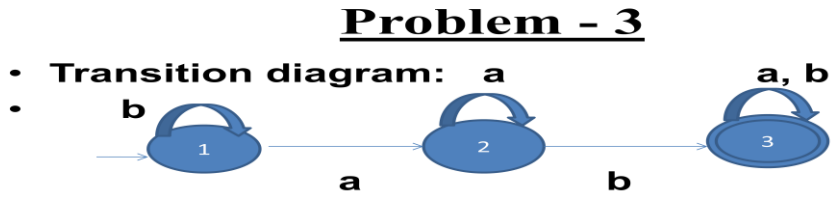
Step 1: Write strings accepted by L i.e.

L = { ab, abab, aaab, abaaa, abbbb, bbababab, babb, bbab, baba,..}

~L= { a, b, aa, bb, bbb, bba, bba, aaa, bbbbb,..}

Step 2: since min string is { ab}, 3 states are required.

Step 3:Write Transition Diagram.

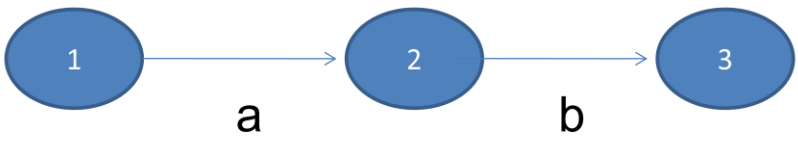


Problem - 3

- **To show some strings are accepted: How**

Example : show string ab is accepted.

Draw the states reached by inputting one character at a time, as shown



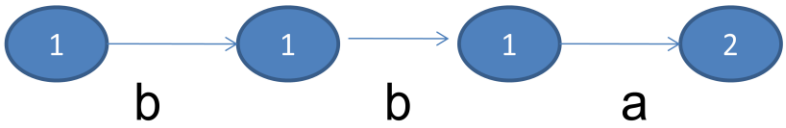
Since reading all the characters state 3 , is reached , which is final state, Therefore string ab is accepted

Problem -3

- **To show some strings are not accepted: How**

Example : show string bba is not accepted.

Draw the states reached by inputting one character at a time, as shown



Since reading all the characters state 2 , is reached , which is not a final state, Therefore string bba is not accepted

Problem – 4

Write a DFSM to accept the language

L = { w ϵ {a, b}* | every w ends in b }

L = { b, ab, abab, aaab, abaab, abbbb, bbababab, babb, bbab, babb,..}

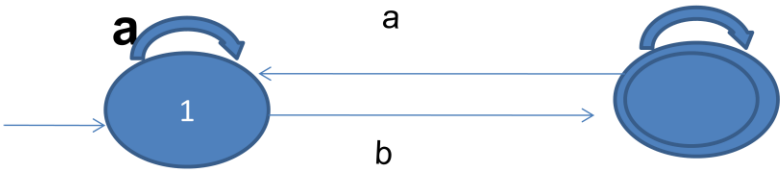
~L= { a, aa, ba, bba, baa, baba, aaa, bbbba,..}

Step 2: since min string are { b}, 2 states are required.

Step 3: Write Transition Diagram.

Problem - 4

- **Transition Diagram:** **b**
- **a**



- **L = { b, bbb, aaab, ababab, bbbbbbab,..}**
- **~L= { a, ba, aaa, bbba, aba, baba, bba, aaba, aabbba,..}**

Problem - 4

- **Transition table: δ is transition function**
- **maps δ:** k x ∑ to k

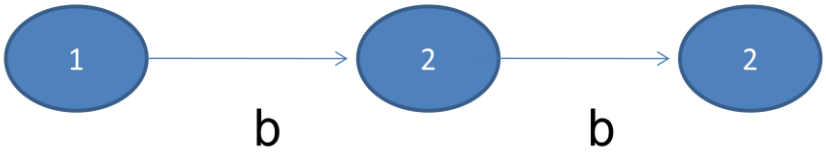
δ	a	b
1	1	2
-> * 2	1	2

Problem - 4

- **To show some strings are accepted: How**

Example : show string bb is accepted.

Draw the states reached by inputting one character at a time, as shown



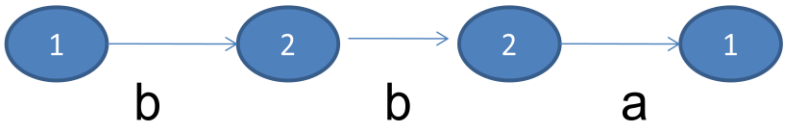
Since reading all the characters state 2 , is reached , which is final state, Therefore string bb is accepted

Problem - 4

- **To show some strings are not accepted: How**

Example : show string bba is accepted.

Draw the states reached by inputting one character at a time, as shown



Since reading all the characters state1, is reached , which is not a final state, Therefore string bba is not accepted

Difficulties with FSMs

Write a DFSM to accept the language

L = { w ϵ {a, b}* | every w ends in ab or ba }

L = { ab, ba, abab, aaba, abaab, abbba, bbababab, baba, bbab, baba,..}

~L= { a, aa, bb, abb, baa, babb, aaa, bbbbabb,..}

Step 2: since min string are {ab, ba}, we are not able to guess no of states.

Note : this is a difficult problem, we end up in spending lot of time to find the solution

Write a DFSM to accept the language –another difficult problem

L = { w ϵ {a, b}* | 3rd character from right is a}

L = { **a**bb,bbbb**a**bb, ab**a**bb, a**a**ba, aa**a**aa, ab**a**ba, bbabab**a**bb, b**a**ba, bb**a**bb, b**a**ba,..}

~L= { a, aa, bb, abbb, baabbb, babba, bbb, bbbbbb,..}

Step 2: since min string are not there, we are not able to guess no of states.

Note : this is a difficult problem, we end up in spending lot of time to find the solution

How to solve difficult problems – study Nondeteministic finite state machines(NFSM)

Nondeterministic Finite State Machines(NFSM) –definition:

Defn: A NFSM , M is a quintuple:

(K, ∑, Δ, s, A)

- **K is a finite set of states,**
- **∑ is the input alphabet,**
- **s ϵ K is the start state**

- **A subset of K is the set of accepting states and**
- **Δ is the transition function it maps from:**

(K x (∑ U {Ɛ})) to K

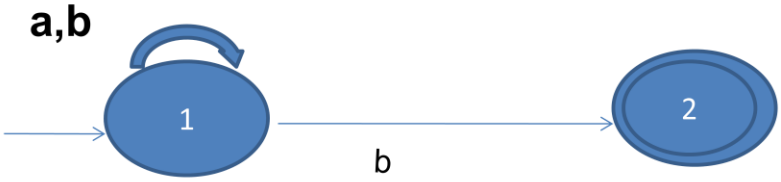
Example of NFSMs

Write a NFSM to accept the language

L = { w ϵ {a, b}* | |w| ends in b} //problem 3 NFSM see below

Problem - 3

- **Transition Diagram:**
- **a,b**



- **L = { b, bbb, aaab, ababab, bbbbbbab,..}**
- **~L= { a, ba, aaa, bbba, aba, baba, bba, aaba, aabbba,..}**
- **Note: every state can have zero or one or more(equal to the no of alphabets) transitions**

Write a DFSM to accept the language

L = { w ϵ {a, b}* | every w ends in ab or ba } //problem 3

L = { ab, ba, abab, aaba, abaab, abbba, bbababab, baba, bbab, baba,..}

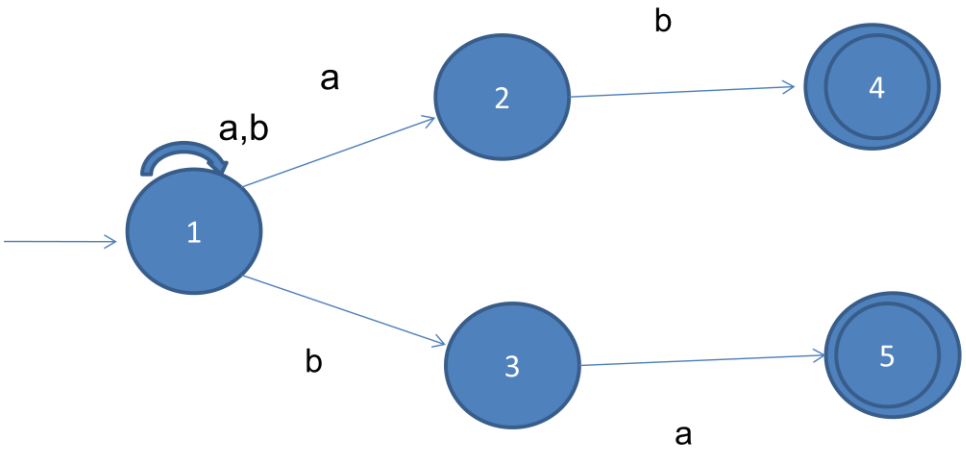
~L= { a, aa, bb, abb, baa, babb, aaa, bbbbabb,..}

Step 2: since min string are {ab, ba}, we are not able to guess no of states.

Note : this is a difficult problem, we end up in spending lot of time to find the solution

Problem – 3 – NFSM

- **Solution: regular expression= (a+b)*(ab+ba)**



b

How to go about, with difficult problems

Write a DFSM to accept the language

L = { w ϵ {a, b}* | 3rd character from right is a}

L = { abb,bbbbabb, ababb, aaba, aaaaa, ababa, bbabababb, baba, bbabb, baba,..}

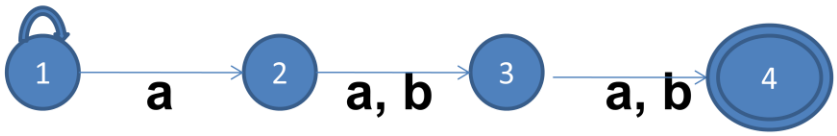
~L= { a, aa, bb, abbb, baabbb, babba, bbb, bbbbbb,..}

Step 2: since min string are not there, we are not able to guess no of states.

Note : this is a difficult problem, we end up in spending lot of time to find the solution

Problem – 4 – NFSM

- **Solution: Regular Expression**
- **(a+b)*a(a+b)(a+b)**

- **a,b**
- 

Write a NFSM to accept the language

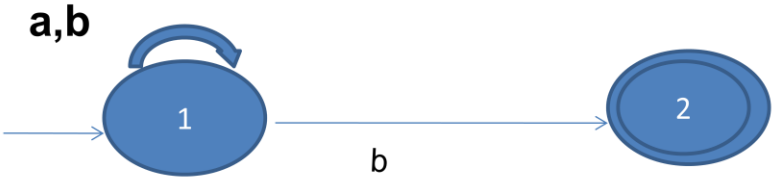
L = { w ϵ {a, b}* | every w ends in b } //solution problem – 1 NFSM

L = { b, ab, abab, aaab, abaab, abbbb, bbababab, babb, bbab, babb,..}

~L= { a, aa, ba, bba, baa, baba, aaa, bbbba,..}

Problem – 1 - NFSM

- Transition Diagram: regular exp = (a+b) *b
- a,b

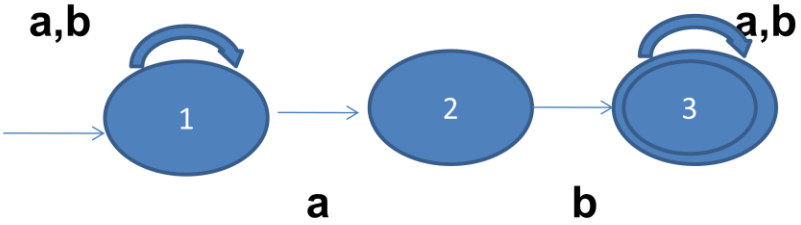


- L = { b, bbb, aaab, ababab, bbbbbbab,..}
- ~L= { a, ba, aaa, bbba, aba, baba, bba, aaba, aabbba,..}
- Note: every state can have zero or one or more(equal to the no of alphabets) transitions

Write NFSM to recognize L = { w ϵ {a, b}* | |w| contains ab}

- Solution: problem – 2 - NFSM
- regular expression (a+b)*ab(a+b)*

Problem – 2 - NFSM

- **Transition Diagram:**
- 

-
- **L = { ab, bab, aab, bbab, aaab,..}**
- **~L= { a, ba, aaa, bbba, bbbb, baaa, aaaaa,**
-

Write a NFSM to recognize the language

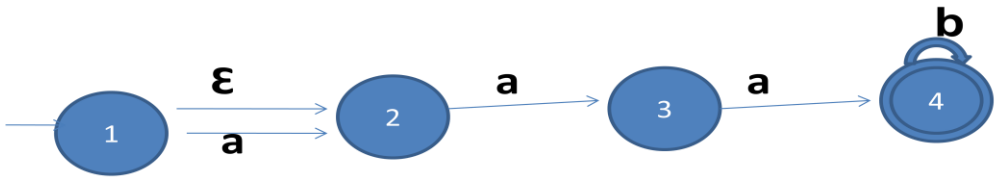
L = {w ϵ {a, b}* | w is made up of an optional *a* followed by *aa,* zero or more *b*'s}

- **L = { aa, aaa, aab, aaab, aabbb, aaabbb,....}**
- **~L = { a, ba, baa, bbbbb, bbbbbba, ...}**
- **Regular expression = re = (a+ Ɛ)aa(b}***

Solution:

Problem – 5 – NFSM

- **Solution- regular expression:**
- **re = (a+ ε)aa(b}***



Procedure to convert NFSM to DFSM

- **Example of NFSMs**
- **Write a NFSM to accept the language**

L = { w ϵ {a, b}* | |w| ends in b}

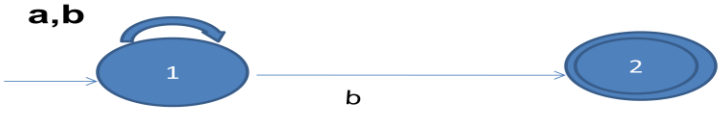
We know all the parameter related to NFSM

- **K is a finite set of states**
- **∑ is the input alphabet**
- **s ϵ K is the start state**
- **A subset of K is the set of accepting states and**

δ is the transition function. Consider the following problem

Problem – 1 - NFSM

- **Transition Diagram: regular exp = (a+b) *b**
- **a,b**



- K is a finite set of states= {1,2}
- ∑ is the input alphabet = {a,b}
- s ϵ K is the start state = 1
- A subset of K is the set of accepting states ={2}
- δ is the transition function as indicated in above fig

Procedure of conversion

We need to calculate the following parameter of DFSM, note only three parameters, i.e. K', A', δ' need to be calculated

- **K' is a finite set of states = ?**
- **∑ is the input alphabet = no change**
- **s ϵ K is the start state = no change**
- **A' subset of K is the set of accepting states = ?**
- **δ' is the transition function = ?**
- **s'=s={1} // note the set notation**
- **Compute δ'**
- **Active states ={{1}}, consider {1}**

δ'({1},a)= {1} .. This exists

δ'({1},b)= {1,2} .. This does not exists, add

New Active states ={{1},{1,2}}, consider {1,2}

δ'({1,2},a)= δ({1},a)Uδ({2},a)={1} U Φ ={1} exists

δ'({1,2},b)= δ({1},b)Uδ({2},b)={1,2} U Φ ={1,2} exists,

No new states are added, therefore Procedure terminates

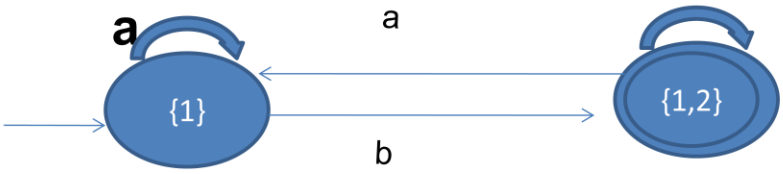
- **Let us write the results in tabular form of** δ', i.e. transition function.
-

δ'	a	b
-> {1}	{1}	{1,2}
* {1,2}	{1}	{1,2}

Solution is as follows:

Problem – 1 - DFSM

- **Transition Diagram:** **b**
- **a**



- **This is the same solution we got when we constructed the DFSM, except the state are in set notation, which we can rename them as {1} = 3, and {1,2} =4. Now it is identical to the original solution.**

Consider the following problem for which we know the NFSM. And apply the procedure to convert it to FSM

Procedure:

s'=s={1} // note the set notation

Compute δ'

Active states ={{1}}, consider {1}

δ'({1},a)= {1,2} .. add

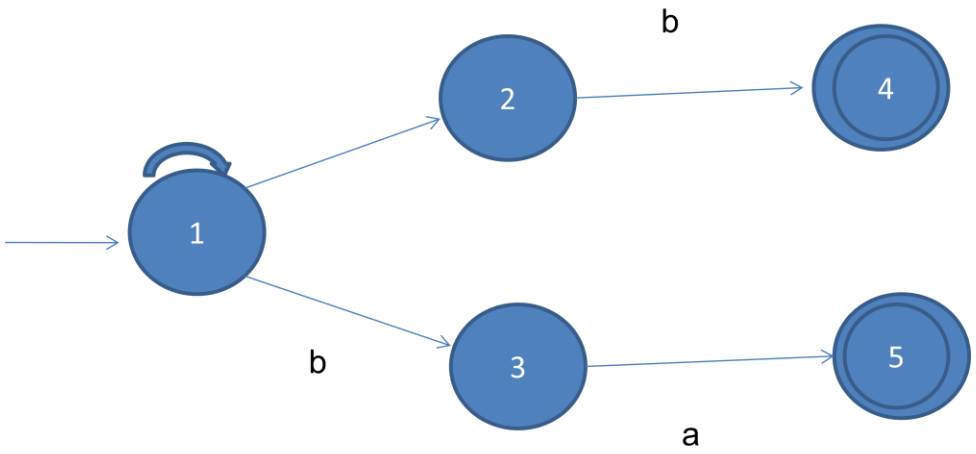
δ'({1},b)= {1,3} .. add

Active states ={{1},{1,2},{1,3}}, consider {1,2}

δ'({1,2},a)= {1,2},a)U Φ ={1,2} ..exists

δ'({1,2},b)= {1,3}U{4})={1,3,4} add

Problem – 3 – NFSM



b

Active states ={{1},{1,2},{1,3},{1,3,4}}, consider {1,3}

δ'({1,3},a)= {1,2} U {5} ={1,2,5} ..add

δ'({1,3},b)= {1,3}U {} ={1,3} exists

Active states ={{1},{1,2},{1,3},{1,3,4}, {1,2,5}}, consider {1,3,4}

δ'({1,3,4},a)= {1,2}U {5} U {}={1,2,5} ..exists

δ'({1,3,4},b)= {1,3}U{}U{} ={1,3} exists

Active states ={{1},{1,2},{1,3},{1,3,4},{1,2,5}}, consider {1,2,5}

δ'({1,2,5},a)= {1,2}U {} U{}={1,2,5} ..exists

δ'({1,2,5},b)= {1,3}U{4}U{}={1,3,4} exists

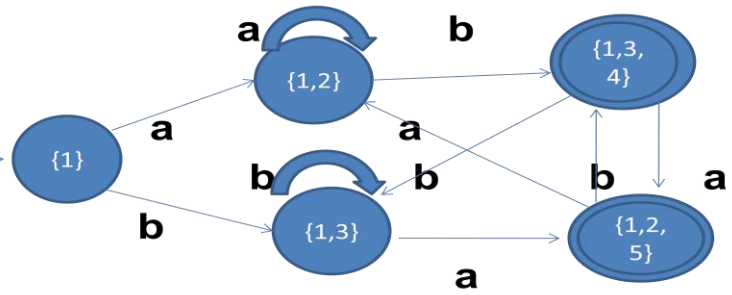
no new states are added, therefore Procedure terminates

write transition table:

	a	b
{1}	{1,2}	{1,3}
{1,2}	{1,2}	{1,3,4}
{1,3}	{1,2,5}	{1,3}
{1,3,4}	{1,2,5}	{1,3}
{1,2,5}	{1,2}	{1,3,4}

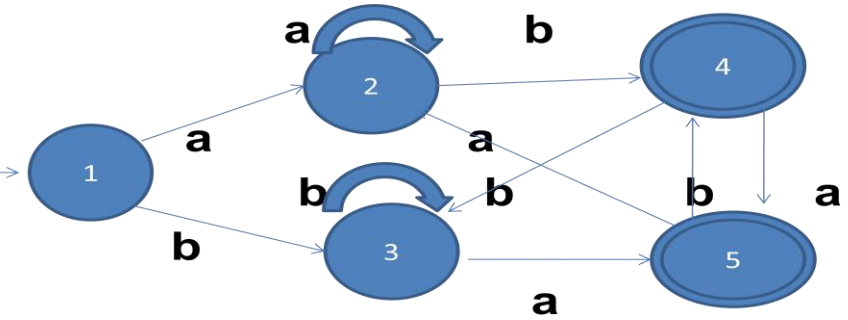
Write transition diagram.

- **Transition diagram:**



Rename the states and check for some representative strings in the fig given below.

- **Transition diagram: DFSM**



- **Check for some strings for the correctness**

Lecture – 5 : chapter 5

Procedure to convert NFSM to DFSM:

The problem which we are attempting to convert has Ɛ transition.

We need to calculate eps for each state using the algorithm as follows:

Eps(q: state) // algorithm

1. **result = {q} and some**
2. **while there exists p ϵ result r not ϵ of result and some transition(p, Ɛ, r) ϵ transition function do:**

   insert r into result.

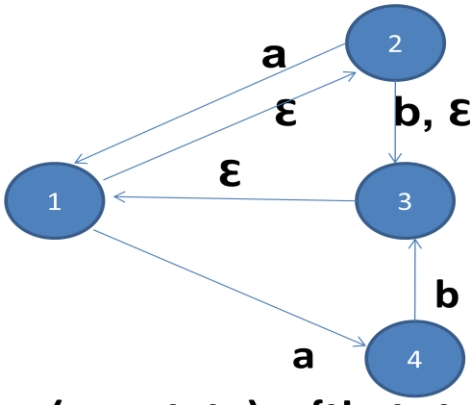
3. **return result.**

Note: It means connect all states that can be reached on Ɛ

<u>Example-1 for calculation of eps:</u>

Example-1 for calculation of eps

- Consider a diagram of NFSM



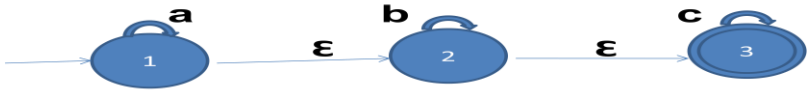
eps(1) = {1,2,3}
eps(2) = {1,2,3}
eps(3) = {1,2,3}
eps(4) = {4 }

Note: eps(any state) = {that state is always included}

a b

Example-2 for calculation of eps:

Example-2 for calculation of eps



eps(1) = {1, 2, 3}
eps(2) = { 2, 3}
eps(3) = { 3}

Problem – 3 – NFSM to DFSM:

Write a NFSM to recognize the language

L = {w ϵ {a, b}* | w is made up of an optional *a* followed by *aa* zero or more *b*'s}

L = { aa, aaa, aab, aaab, aabbb, aaabbb,....}

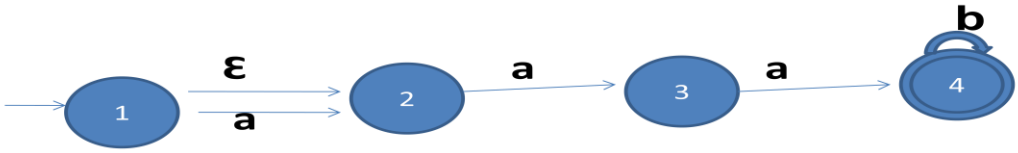
~L = { a, ba, baa, bbbbb, bbbbbba, ...}

Regular expression = re = (a+ Ɛ)aa(b}*

Write the transition diagram of NFSM

Problem – 3 – NFSM

- Solution- regular expression:
- re = (a+ Ɛ)aa(b}*



- eps(1) = {1, 2}
- eps(2) = {2}
- eps(3) = {3} eps(4) = {4}

Procedure to convert NFSM to DFSM

We know all the parameter related to NFSM

- K is a finite set of states
- ∑ is the input alphabet
- s ϵ K is the start state
- A subset of K is the set of accepting states and
- δ is the transition function

Procedure of conversion:

We need to calculate the following parameter of DFSM, note only three parameters, i.e. K', A', δ' need to be calculated

- **K’ is a finite set of states = ?**
- **∑ is the input alphabet = no change**
- **s ϵ K is the start state = no change**
- **A’ subset of K is the set of accepting states = ?**

δ' is the transition function = ?

We need to calculate the following parameter of DFSM, note only three parameters, i.e. K’, A’, δ’ need to be calculated

- **K’ is a finite set of states = ?**
- **∑ is the input alphabet = no change**
- **s ϵ K is the start state = no change**
- **A’ subset of K is the set of accepting states = ?**
- **δ' is the transition function = ?**

<u>Procedure:</u>

s’=s=eps{1} = {1,2}// this is start state DFSM

Compute δ'

Active states ={{1,2}}, consider {1,2}

δ‘({1,2},a)= eps{δ(1,a) U δ(2,a)} = eps(2) U eps(2)

= {2,3} This state does not exist, therefore add

δ‘({1,2},b)= eps{δ(1,b) U δ(2,b)} = eps(Φ) U eps(Φ)

= Φ, This state does not exist, therefore add

Now Active states ={{1,2},{2,3},Φ}, consider {2,3}

δ'({2,3},a)= eps{δ(2,a) U δ(3,a)} = eps(3) U eps(4) = {3,4} this state does not exists, add

δ'({2,3},b)= eps{δ(2,b) U δ(3,b)} = eps(Φ) U eps(Φ) = Φ already exists, do not add

Now Active states ={{1,2},{2,3},Φ,{3,4}}, consider {3,4}

δ'({3,4},a)= eps{δ(3,a) U δ(4,a)} = eps(4) U eps(Φ) = {4} this state does not exists, add

δ'({3,4},b)= eps{δ(3,b) U δ(4,b)} = eps(Φ) U eps(4) = {4} does not exists, add

Now Active states ={{1,2},{2,3},Φ,{3,4}{4}}, consider {4}

δ'({4},a)= eps{δ(4,a)} = eps(Φ) = Φ this state exists, no need to add

δ'({4},b)= eps{δ(4,b)} = eps(4) = {4} this state exists, no need to add

Note: No new states are added, therefore algorithm terminates. Now we have all the states of DFSM and its transition functions

- **Let us write the results in tabular form of δ',** i.e. transition function

δ'+	a	b
->{1,2}	{2,3}	Φ
{2,3}	{3,4}	Φ
Φ	Φ	Φ
*{3,4}	{4}	{4}
*{4}	Φ	{4}

DFSM is constructed as follows:

First find out the accepting states of NFSM In this case it is {4}

Look at all final active states of DFSM. In this case it is:

Active states ={{1,2},{2,3},Φ,{3,4}{4}}

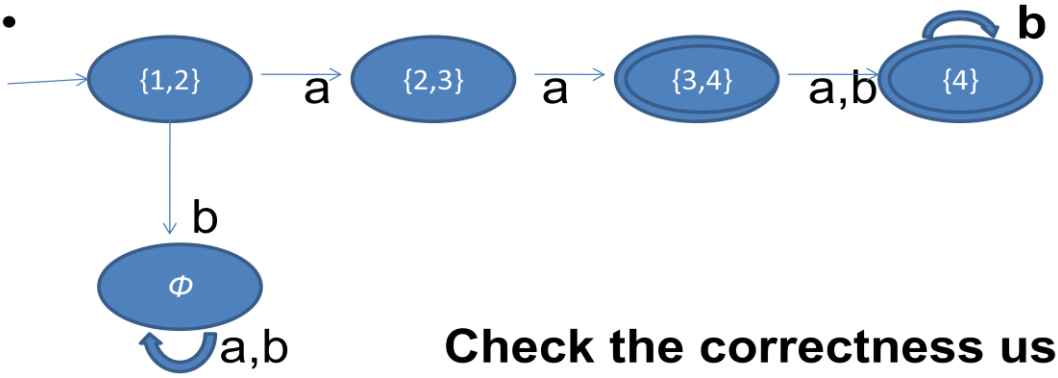
Find all the states containing state {4}

There are two state namely {3,4} and {4}

They will accepting states of DFSM

Transitin diagram- DFSM:

Transitin diagram- DFSM

- **Write transition diagram using K', A' and δ'**
- 

Check the correctness using

- **L = { aa, aaa, aab, aaab, aabbb, aaabbb,....}**
- **~L = { a, ba, baa, bbbbb, bbbbbba, ...}**

FSM to operational systems:

Now that we know how to design a DFSM if it is simple and if it is complex, we write NFSM and convert the same to DFSM, using the procedure discussed.

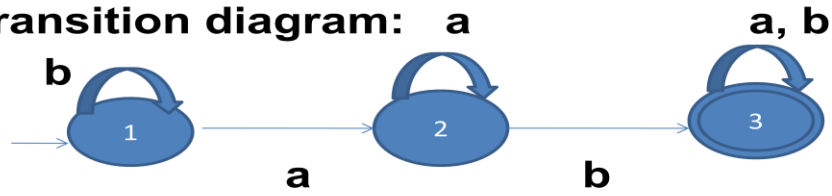
FSM can be simulated using Software or Hardware depending on the requirement.

In the next section we will discuss simulating using a pseudo code.

Simulation the deterministic FSM:

Simulation the deterministic FSM

- **Transition diagram:**



- Above transition diagram for DFSM to accept the language :

  L = { w ϵ {a, b}* | ab is a substring of w}

Hardcoding a Deterministic FSM:

Until accept or reject do:

1: s=get-next-symbol.

If s=b go to 1.

else if s=a then go to 2

2: s=get-next-symbol

If s=a go to 2.

else if s=b then go to 3

3: If s=a or b go to 3.

else if s= end-of-file then accept.

else reject.

Simple interpreter for deterministicFSM:

dfsmsimulate(M:DFSM, w: string)

1 st = s

2 Repeat

2.1 c=get-next-symbol(w)

2.2 if c ≠ end-of-file then:

2.2.1 st = δ(st,c)

until c = end-of-file

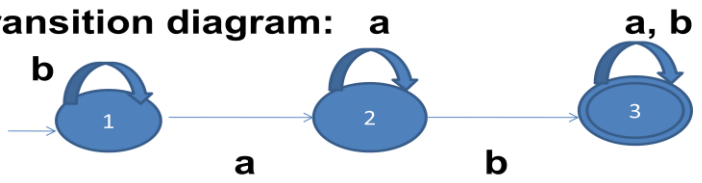
3 If st ϵ A then accept

else reject.

Trace dfsmsimulate:

Trace dfsmsimulate

- Transition diagram:



+

Note: Each state has 2 transitions.

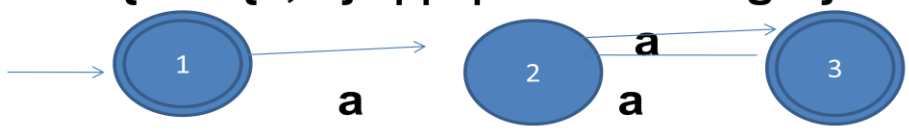
Final st = 3, therefore string bbaabb is accepted (see trace in next slide)

Trace table for string bbaabb

st	c	δ(st,c)
1	b	**1**
1	b	**1**
1	a	**2**
2	a	**2**
2	b	**3**
3	b	**3**

What is minimization of FSMs:

What is minimization of FSMs

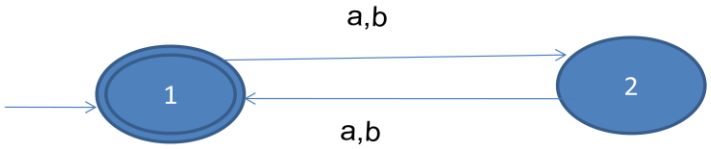
- Write a DFSM to accept the language
  L = { w ϵ {a, b}* | |w| is even length} Example
- 

L = { ε, aa, bb, ab, ba, abab, aabb, bbaa, baba,..}

- **~L= { a, b, aaa, bbb, aba, bab, bba, aab, aabbb,..}**

What is minimization of FSMs

- **Transition Diagram:**



- **L = { ε, aa, bb, ab, ba, abab, aabb, bbaa, baba,..}**
- **~L= { a, b, aaa, bbb, aba, bab, bba, aab, aabbb,..}**

Note : The behaviour of state 1 and 3 are identical as shown below:

- **δ (1,a) = 2**
- **δ (3,a) = 2**

Therefore there is no need to have two separate states 1 and 3 and they can be combined as shown in the above diagram. Also note that two states are a must and it can not be further minimized.

Lecture – 6:

Equivalence class: Definition

An equivalence relation has following properties.

- **It is reflexive**
- **It is symmetric**

- **It is transitive.**

Example:

-- relation- has the same birth date

-- relation- defined by =

Not an example: relation <=

Equivalence relations can also be represented by a digraph since they are a binary relation on a set. For example the digraph of the equivalence relation congruent mod *3* on {*0, 1, 2, 3, 4, 5 , 6*} is as shown below. It consists of three connected components

<u>Equivalence class –Example:</u>

- **0 mod 3 = 0** **The results are {0,1,2}**
- **1 mod 3 = 1**
- **2 mod 3 = 2**
- **3 mod 3 = 0**
- **4 mod 3 = 1**
- **5 mod 3 = 2**
- **6 mod 3 = 0**

{0, 3, 6} --- have 0 as their remainder

{1, 4} --- have 1 as their remainder

{2, 5} --- have 2 as their remainder

Defn:Indistinguishable:

We say that x and y are indistinguishable with respect to L, which we will write as

$x \approx_L y$ iff:

all $z \in \Sigma^*$ (either xz and yz $\in$ L or neither is)

consider x=aa and y = bb and z=ba

since aaba and bbba are in L, therefore

$x \approx_L y$

Defn: Distinguishable:

We say that x and y are distinguishable with respect to L, iff they are not indistinguishable. If x and y are distinguishable then there exists at least one string z, such that one but not both of xz and yz is in L

consider x=aa and y = a and z=ba

since aaba and aba both are in not L,

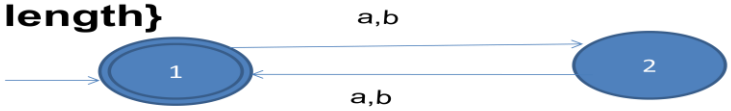
aaba is in L and aba is not in L

- **Note : indistinguishable is a equivalence class**

Equivalence class-two partitions:

Equivalence class-two partitions.

- Transition Diagram for L = { w ϵ {a, b}* | |w| is even length}



- L = { ε, aa, bb, ab, ba, abab, aabb, bbaa, baba,..}
- ~L= { a, b, aaa, bbb, aba, bab, bba, aab, aabbb,..}
- Any string from L is distinguishable from any string from ~L

L = { ε, aa, bb, ab, ba, aaaa, bbbb, bbaa, baba,..}

All the elements are indistinguishable

~L= { a, b, aaa, bbb, aba, bab, bba, aab, aabbb,..}

All the elements are indistinguishable.

For example aa and a are distinguishable, because

Take an element bb from the language L, aabb is in L, abb

is not in L , therefore they are not in the same eq.class

For example aa and bb are indistinguishable, because

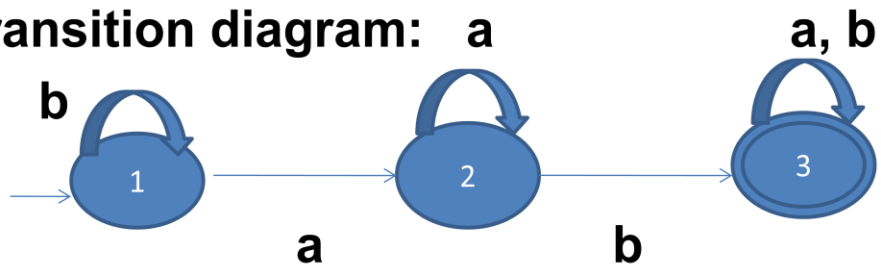
Take an element bb from the language L, aabb is in L, bbbb

is also in L , therefore they are in the same eq.class

Equivalence class-three partitions.

Equivalence class-three partitions.

- Transition diagram:



L = set of all string containing string – ab

where Σ = {a, b}

L = { ab, abab, aaab, abaaa, abbbb, bbababab, babb, bbab, baba,..}

~L= { a, b, aa, bb, bbb, bba, bba, aaa, bbbbb,..}

All the strings of L belong to state 3.

How to separate ~L into two separate states.

What is the basis?

- [1] = {Ɛ, b, bb, bbb, bbbb, ...}
- [2] = {a ,aa, aaa, aaaa, ...}

For example consider b from block-1 and a from block-2, take b from Σ*

bb is not in L whereas ab is in L, therefore they

are not indistinguishable, i.e. Distinguishable

- Equivalence class partitions:

In general equivalence partitions of L and ~L,

Can further partitions.

In the last example we saw partition of ~L

in to 2 eq classes.

We will see now how L can be partitioned into more than one block or eq. Classes.

Equivalence class-more partitions

Find the equivalence partition of L , where

L = {w ϵ {a, b} | w has no adjacent characters are the same} // problem 5.26(p-89)

L= {ε, a, aba, ababa, b, ab, bab, abab,….}

~L = { aa, abaa, ababb, bbbbbb, aaaa, ...}

Check any of L or ~L can be further separated,

Consider L= {ε, a, aba, ababa, b, ab, bab, abab,….} and select a and b

Also select a from Σ*

Test: aa is in ~l and ba is L . Therefore L is not an eq class, it should be further refined.

- **[1] = {a, aba, ababa,...etc will get separated}**
- **[2] = {b, ab, bab, abab, ...}**

- **Consider L= {ε, a, aba, ababa, b, ab, bab, abab,….} and select ε and a**
- **Also select a from Σ***
- **Ɛa = a is in L, but**
- **aa is not in L, therefore ε and a are not in the same eq. class, finally,**
- **[1] = {ε}**
- **[2] = {a, aba, ababa, …}**
- **[3] = {b, ab, bab, abab, …}**
- **[4] = {aa, abaa, ababb,…..}**

The transition diagram for above partitions is ……

Equivalence class-solution.

Solution to the problem : (workout on the board.)

Finally how to know that DFSM is not possible. i.e. When the partitions are infinite, no DFSM is possible.

- **Example: L = {a^n b^n | n > 0}**

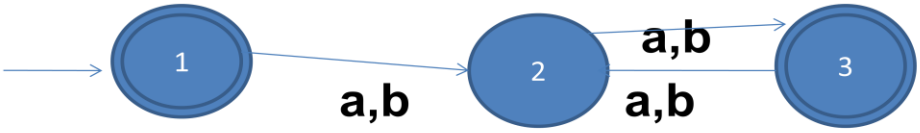
This will have infinite partitions, no DFSM is possible, and L is not regular language.

Theorem 5.6(Myhill-Nerode):

Theorem: A language is regular iff the number of eq. Classes of L is finite.

Minimization of FSMs-problem -1:

Minimization of FSMs-problem -1

- Write a DFSM to accept the language
  L = { w ϵ {a, b}* | |w| is even length} Example
- 

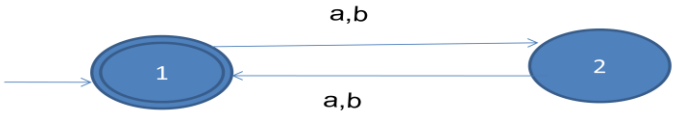
L = { ε, aa, bb, ab, ba, abab, aabb, bbaa, baba,..}

- **~L= { a, b, aaa, bbb, aba, bab, bba, aab, aabbb,..}**

Procedure for Minimization of DFSM:
Partition the states in to non-accepting and accepting states.
{2} and {1,3}
Check whether they are distinguishable?
Workout(on the board)
Minimizied FSM-problem-1

Minimizied FSM-problem-1

- **Transition Diagram:**

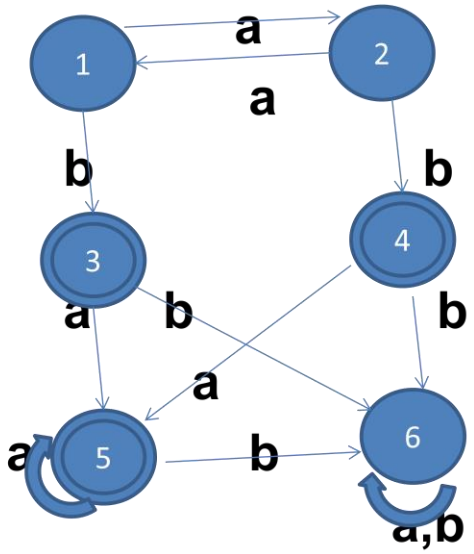


- **L = { ε, aa, bb, ab, ba, abab, aabb, bbaa, baba,..}**
- **~L= { a, b, aaa, bbb, aba, bab, bba, aab, aabbb,..}**

Minimization of FSMs -Problem-2:

Minimization of FSMs -Problem-2

- Minimize the DFSM



Start with what is clearly distinguishable i.e.

Non-Accepting states and accepting states

- **{1, 2, 6} and {3, 4, 5} check further if they are**
- **Distinguishable input**

Continue subdividing state, until all distinguishable states are separated.

Lecture:7

Moore Machine(transducer)

Transducer: A device that converts variations in a physical quantity, such as pressure or brightness, into an electrical signal, or vice versa.

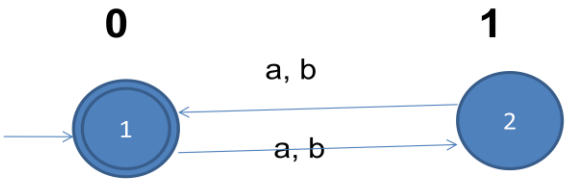
Defn: A Moore machine, M is a seven tuple:

(K, ∑, O, δ, D, s, A), where

- **K is a finite set of states,**
- **∑ is the input alphabet,**
- **O is the output alphabet,**
- **s ϵ K is the start state**
- **A subset of K is the set of accepting states(not imp)**
- **δ is the transition function it maps from K x ∑ to K**
- **D is the display or output function from K to (O)***

Example of Moore Machine:

Example of Moore Machine

- **In Moore machine each state is associated with a output. Suppose we want Moore machine to output '0' when the length of input string is even, otherwise output '1'**
- **Σ = {a, b}**
- 

Transition table for the Moore Machine(see fig a bove):

Transition table

Transition function	Input = a	Input = b	output
1	2	2	0
2	1	1	1

Mealy machine(transducer):

Defn: A Mealy machine, M is a six tuple:

(K, ∑, O, δ, s, A), where

- **K is a finite set of states,**
- **∑ is the input alphabet,**
- **O is the output alphabet,**
- **s ϵ K is the start state**
- **A subset of K is the set of accepting states and**
- **δ is the transition function it maps from**

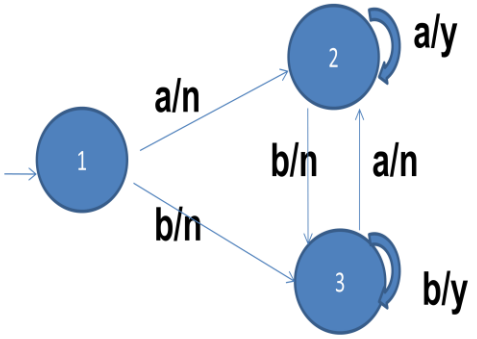
(K x ∑) to (K x O*)

Note: output is associated with each input.

Example of Mealy machine:

Example of Mealy machine

- L = { w ϵ {a, b}*| w ends in aa or bb}
- Requirements are **when input ends aa it should have output 'y' and when input ends bb it should have output 'n'**



Transition table:

Transition table

Trans fun	a	output	b	output
1	2	n	3	n
2	2	y	3	n
3	2	n	3	y

Computation(chapter 4):

In this chapter, effort is made to:

Define problems as languages to be decided

Define programs as state machines whose input is a string and whose output is *accept or reject*

- **Key ideas :**

1. **Decision procedures**
2. **Non determinism**
3. **Function on languages.**

<u>Decision Procedures</u>

Defn: A decision problem is one for which we must make a *yes/no* decision.

A decision procedure is an algorithm to solve a decision problem.

It a program whose result is a Boolean value.

In order to return a Boolean value, a decision procedure must be guarantee to halt on all inputs

<u>Decision procedures are to answer question such as:</u>

-- Is string s in Language L?

-- Given a machine, does it accept any string?

-- Given two machines, do they accept the same strings?

-- Given a machine, is it the smallest m/c that does its job?

<u>Three imp things about Procedures:</u>

1.Does there exist a decision procedure(algorithm)

2.If any decision procedures exist, find one

3. If exists, find the most efficient one, and how efficient it is?

<u>Decision procedures are programs, and they must have two correctness properties:</u>

1. **Program must be guaranteed to halt.**
2. **The answer must be correct.**

<u>Example – 1:</u>

Checking for even numbers:

```
even(x: integer)=
        If (x/2)*2=x then return True
        else return False.
```

If x=3 then x/2=1 and 1*2 != 3 therefore “false”

If x=8 then x/2=4 and 4*2 = 8 therefore “true”

<u>Example – 2:</u>

Checking for Prime numbers:

```
prime(x: positive integer) =
        For i = 2 to ceiling(sqrt(x)) do:
            If (x/i)*i = x then return False
        return True
```

Assume x = 7 then ceiling(sqrt(x))= ceiling(2.65) = 3

i = 2; 7/2*2 != 7 next iteration

i = 3; 7/3*2 != 7 next iteration(no more iterations)

Returns True

<u>Example-3:</u>

Checking for Programs that halt on a particular input.

haltOnw(p: program, w: string)=

1. **Simulate the execution of p on w**
2. **If the simulation halts return True**

   else return False.

note: 1. this is not a procedure, because it never returns False.

2. No decision procedure exists for this.

<u>Determinism and non determinism:</u>

Consider a program:

Choose(action 1;;

action 2;;

....

action n;;)

<u>Observation on choose:</u>

Returns some successful value, if there is one

If there is no successful value, the choose will:

- Halt and return False if all the actions halt and return False

- Fail to halt if any of the actions fails to halt.

(note that this has a potential to return successful value, it may be taking more time)

Deterministic and non Deterministic:

If a program does not use choose then it is deterministic.

If a program includes choose then it is non deterministic.

Functions on Languages and Programs:

The function chop:

chop(L): is all the odd length strings in L with their middle character chopped out.

The function firstchars:

firstchars(L): determines the first characters by looking at all strings in L

Examples of chop(L):

Examples of chop(L)

n	in $A^n B^n C^n$	in chop $A^n B^n C^n$
0	ε	
1	abc	ac
2	aabbcc	
3	aaabbbccc	aaabbccc
4	aaaabbbbcccc	
5	aaaaabbbbbccccc	aaaaabbbbccccc

Examples of firstchars(L)

Examples of firstchars(L)

L	Firstchars(L)
Φ	Φ
{ ε }	Φ
{a}*	{a}*
$A^n B^n$	{a}*
{a, b}*	{a}* U {b}*

Closure of Languages:

Closure of Languages

	finite	infinite
union	yes(1)	yes(5)
intersection	yes(2)	no(6)
chop()	yes(3)	no(7)
firstchars()	no(4)	yes(8)

Langauage Hierarchy:

Hierarchy of Languages and corresponding automata

Regular languages: FSMs

Context-free languages: PDAs

D(decidable) Languages: Turing machine

SD(semi decidable) languages: Turing machine

Importance of classification:

The factors are:

1.Computational efficiency: As function of input length

FSMs - Linear with respect to input string

PDAs - cube of the length of input string

TM - exponentially with respect to input String

2.Decidability: Answer to the questions

FSM - accepts some string?

FSM - is it minimal?

FSMs- are two FSMs identical?

PDAs- only some of the above can be answered

TM - none of the above can be answered

3.Clarity: tools that enable analysis- exist?

FSM - yes

PDA - yes

TM - none

..... End of Module – 1 .....

Chapter-6

Regular Expressions

Regular Expression (RE)

A RE is a string that can be formed according to the following rules:

1. ø is a RE.
2. ε is a RE.
3. Every element in ∑ is a RE.
4. Given two REs α and β,αβ is a RE.
5. Given two REs α and β, α U β is a RE.
6. Given a RE α, α* is a RE.
7. Given a RE α, α+ is a RE.
8. Given a RE α, (α) is a RE.

if ∑ = {a,b}, the following strings are regular expressions:

ø, ε, a,b, (a U b)*, abba U ε.

Semantic interpretation function L for the language of regular expressions:

1. L (ø) = ø, the language that contains no strings.
2. L (ε) = {ε}, the language that contains empty string.
3. For any cϵ∑, L(c) = {c}, the language that contains single character string c.
4. For any regular expressions α and β, L (αβ) = L (α) L (β).
5. For any regular expressions α and β, L (α U β) = L (α) U L (β).
6. For any regular expression α, L (α*) = (L (α))*.
7. For any regular expression α, L (α+) = L (αα*) = L (α) (L (α))*
8. For any regular expression α, L ((α)) = L (α).

Analysing Simple Regular Expressions

1.L((a U b)*b) = L((a U b)*)L(b)

= (L((a U b)))*L(b)

= (L(a) U L(b))*L(b)
=({a} U {b})*{b}
= {a,b}*{b}

(a U b)*b is the set of all strings over the alphabet {a, b} that end in b.

2. L(((a U b) (a U b))a(a U b)*)

= L(((a U b)(a U b)))L(a) L((a U b)*)
= L((a U b)(a U b)) {a} (L((a U b)})*
= L((a U b))L((a U b)) {a} {a,b}*
= {a, b} { a, b} {a} {a, b}*

- ((a U b)(a U b))a(a U b)* is

{xay : x and y are strings of a's and b's and lxl = 2}.

Finding RE for a given Language

1.Let L = {w ϵ {a, b }*: |w| is even}.

L = {aa,ab,abba,aabb,ba,baabaa,-------}
RE = ((a U b)(a U b))* or (aa U ab U ba U bb)*

2. Let L = {w ϵ {a, b }*: w starting with string abb}.

L = {abb,abba,abbb,abbab-------}
RE = abb(a U b)*

3. Let L = {w ϵ {a, b }*: w ending with string abb}.

L = {abb,aabb,babb,ababb-------}
RE = (a U b)*abb

4. L = {w ϵ {0, 1}* : w have 001 as a substring}.

L = {001,1001,000101,-------}
RE = (0 U 1)*001(0 U 1)*

5. L = {w ϵ {0, 1}* : w does not have 001 as a substring}.

L = {0,1,010,110,101,----}
RE = (1 U 01)*0*

6. L = {w ϵ {a, b}* : w contains an odd number of a's}.

L = {a,aaa,ababa,bbaaaaba------}
RE = b*(ab*ab*)* a b* or b*ab*(ab*ab*)*

7. L = {w ϵ {a, b}* :#a(w) mod 3 = 0}.
L = {aaa,abbaba,baaaaaa,---}
RE = (b*ab*ab*a)*b*

8. Let L = {w ϵ {a, b }*:#a(w) <= 3}.
L = {a,aa,ba,aaab,bbbabb,-------}
RE = b*(a U ε)b*(a U ε)b*(a U ε)b*

9. L = {w ϵ {0, 1}* : w contains no consecutive 0's}
L={0, ε,1,01,10,1010,110,101,-----}
RE = (0 U ε)(1 U 10)

10. L = {w ϵ {0, 1}* : w contains at least two 0's}
L={00,1010,1100,0001,1010,100,000,-----}
RE = (0 U 1)*0(0 U 1)*0(0 U 1)*

11.L = { $a^n b^m$ / n>=4 and m<= 3}
RE= (aaaa)a*(ε U b U bb U bbb)

12.L = { $a^n b^m$ / n<=4 and m>= 2}
RE= (ε U a U aa U aaa U aaaa)bb(b)*

13. L = { $a^{2n} b^{2m}$ / n>=0 and m>= 0}
RE= (aa)*(bb)*

14. L = { $a^n b^m$:(m+n) is even}
(m+n) is even when both a's and b's are even or both odd.
RE = (aa)*(bb)* U a(aa)*b(bb)*

Three operators of RE in precedence order(highest to lowest)

1. Kleene star
2. Concatenation
3. Union

Eg: (a U bb*a) is evaluated as (a U (b(b*)a))

Kleene's Theorem

Theorem 1:
Any language that can be defined by a regular expression can be accepted by some finite state machine.
Theorem 2:
Any language that can be accepted by a finite state machine can be defined by some regular expressions.
Note: These two theorems are proved further.

Buiding an FSM from a RE

Theorem 1:For Every RE, there is an Equivalent FSM.
Proof: The proof is by construction.
We can show that given a RE α,
we can construct an FSM M such that L (α) = L (M).
Steps:

1. If α is any cϵ∑ ,we construct simple FSM shown in Figure(1)

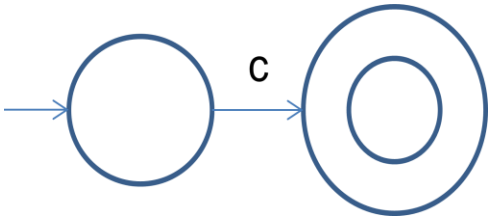


Figure (1)

2. If α is any ø, we construct simple FSM shown in Figure(2).

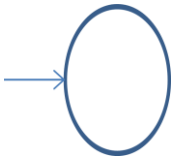


Figure (2)

3. If α is ε,we construct simple FSM shown in Figure(3).

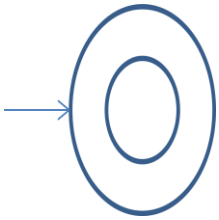
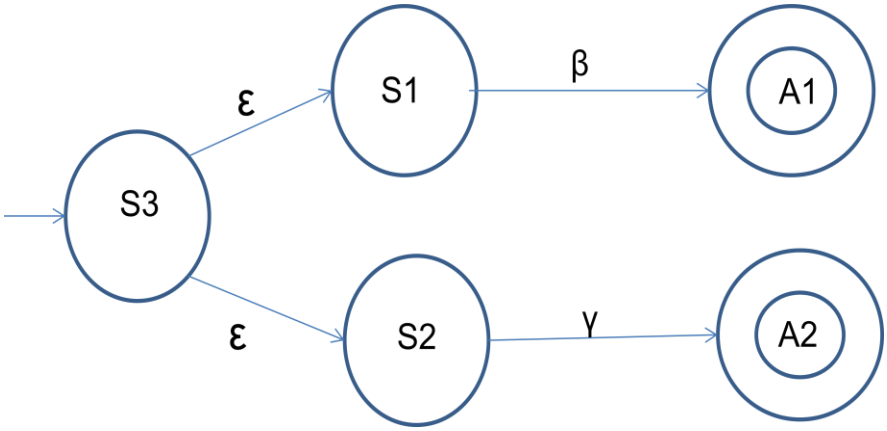


Figure (3)

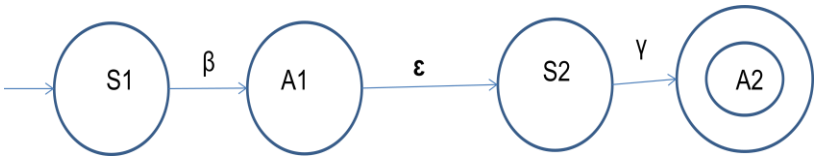
4. Let β and γ be regular expressions.
   If L(β) is regular,then FSM M1 = (K1, ∑ , δ1, s1, A1).
   If L(γ) is regular,then FSM M2 = (K2, ∑ , δ2, s2, A2).
   If α is the RE β U γ, FSM M3=(K3, ∑ , δ3, s3, A3) and
   L(M3)=L(α)=L(β) U L(γ)
   M3 = ({S3} U K1 U K2, ∑ , δ3, s3, A1 U A2), where
   δ3 = δ1 U δ2 U { ((S3, ε), S1),((S3, ε),S2)}.



α = β U γ

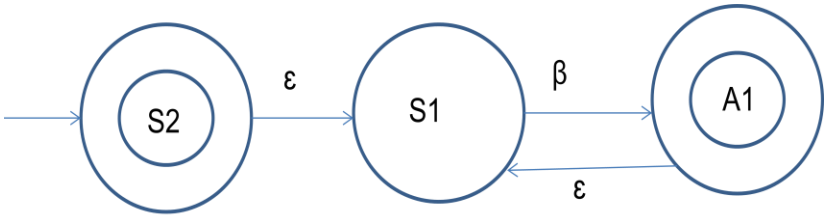
5. If α is the RE βγ, FSM M3=(K3, ∑ , δ3, s3, A3) and
   L(M3)=L(α)=L(β)L(γ)
   M3 = (K1 U K2, ∑ , δ3, s1, A2), where

δ3 = δ1 U δ2 U { ((q, ε), S2):qϵA1}.



α = βγ

6. If α is the regular expression β*, FSM M2 = (K2, ∑, δ2 s2, A2) such that L (M2) = L (α)) = L (β)*.
   M2 = ({S2} U K1, ∑, δ2,S2,{S2} U A1), where
   δ2 = δ1 U {((S2, ε),S1)} U {((q, ε),S1):q ϵ A1}.



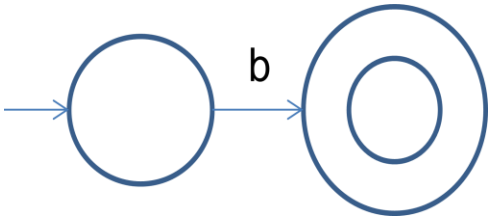
α = β*

Algorithm to construct FSM, given a regular expression α

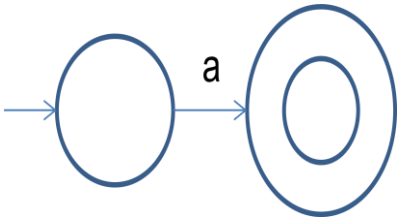
regextofsm(α : regular expression) =
Beginning with the primitive subexpressions of α and working outwards until an FSM for an of α has been built do:
Construct an FSM as described in previous theorem.

Building an FSM from a Regular Expression

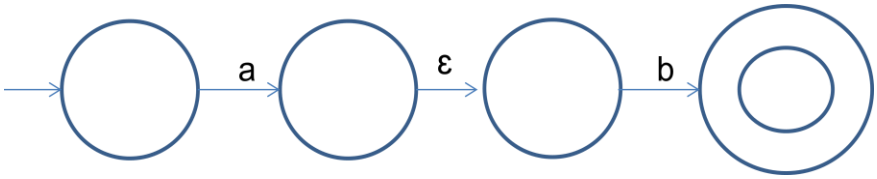
1. **Consider the regular expression (b U ab)*.**



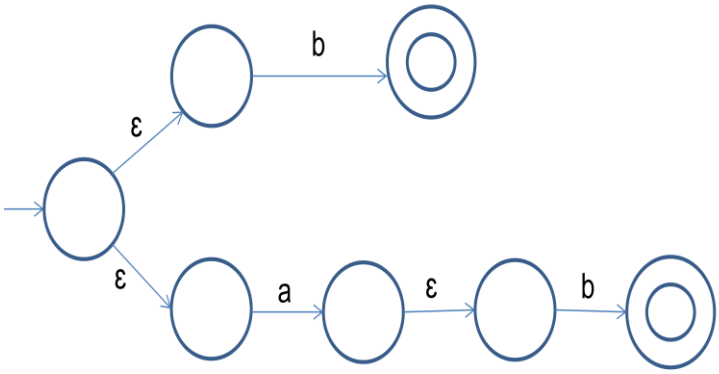
An FSM for b



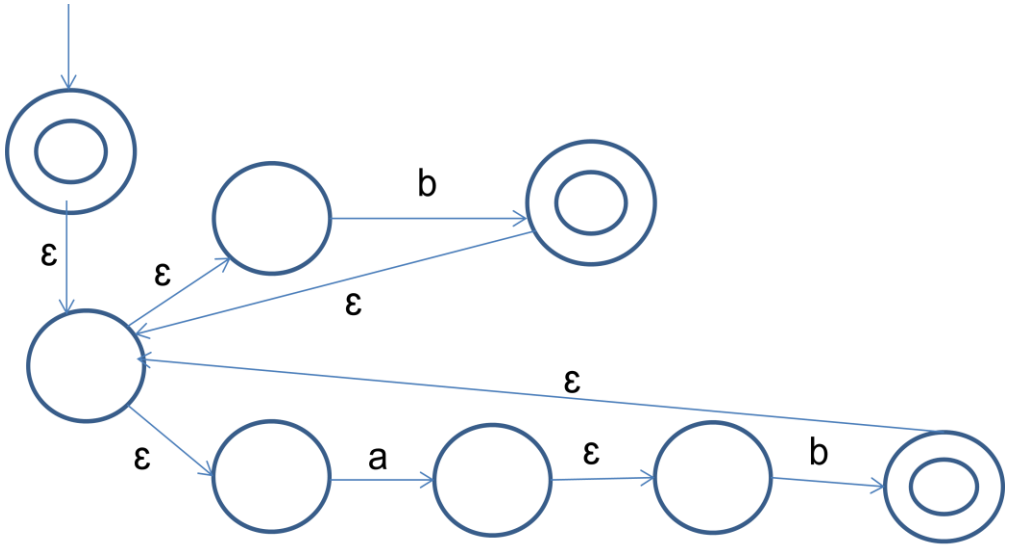
An FSM for a



An FSM for ab

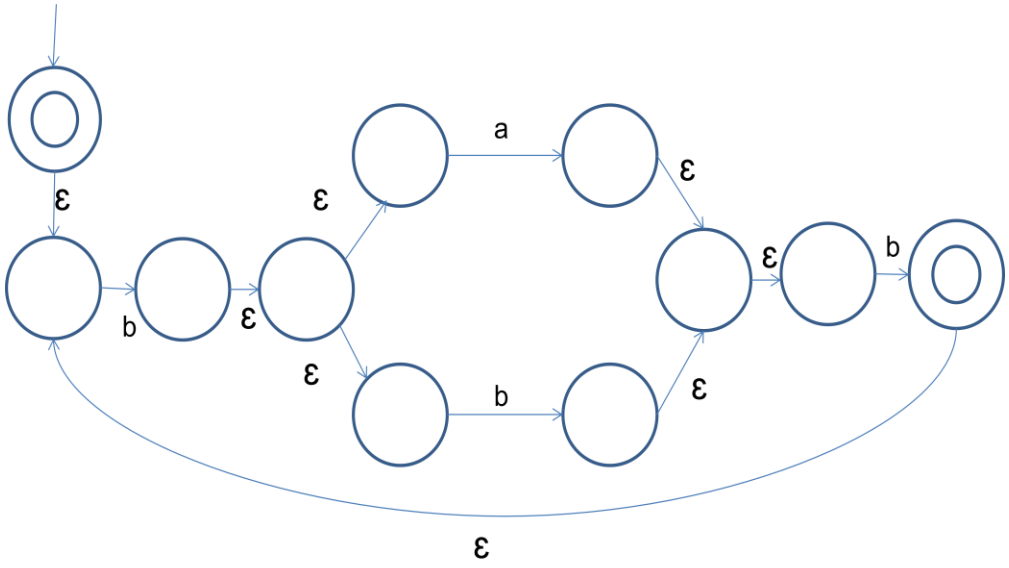


An FSM for (b U ab)

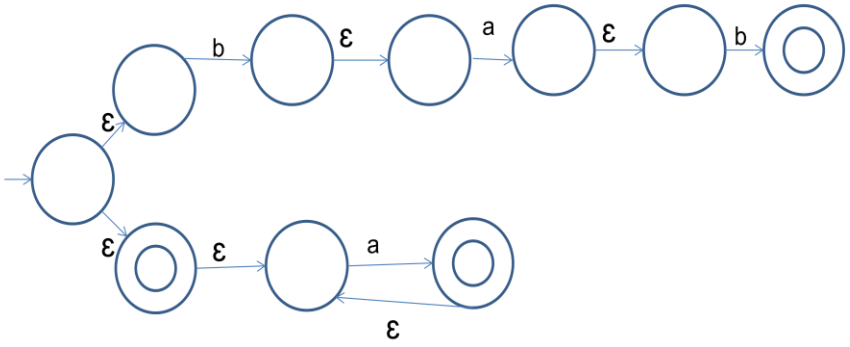


An FSM for (b U ab)*

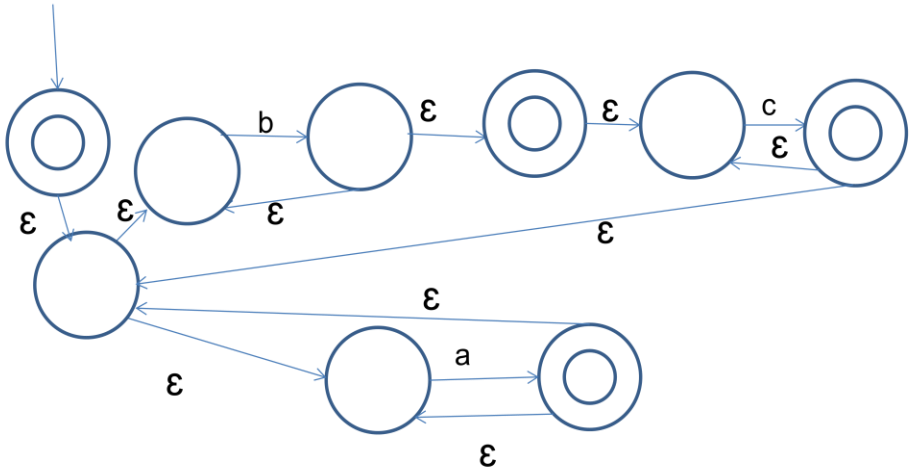
2. Construct FSM for the RE (b(a U b)b)*



3. Construct FSM for the RE **bab U a***

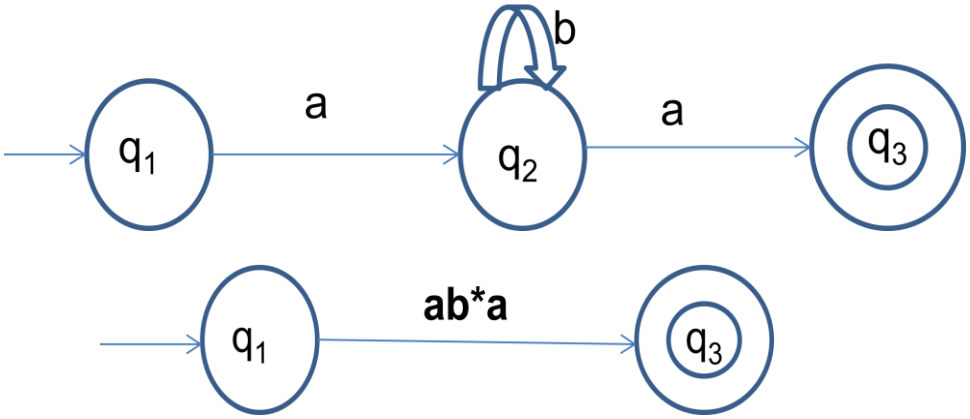


FSM for RE = (a* U b*c*)*



Building a Regular Expression from an FSM

Building an Equivalent Machine M



Algorithm for FSM to RE(heuristic)

fsmtoregexheuristic(M: FSM) =

1. Remove from M-any unreachable states.
2. No accepting states then return the RE ø.
3. If the start state of M is has incoming transitions into it, create a new start state s.
4. If there is more than one accepting state of M or one accepting state with outgoing transitions from it, create a new accepting state.
5. M has only one state, So L (M} = { ε } and return RE ε.

6. Until only the start state and the accepting state remain do:

   6.1. Select some state rip of M.

   6.2. Remove rip from M.

   6.3. Modify the transitions. The labels on the rewritten

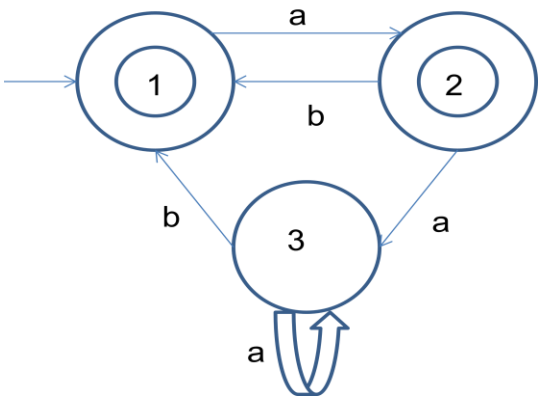
   transitions may be any regular expression.

7. Return the regular expression that labels from the

start state to the accepting state.

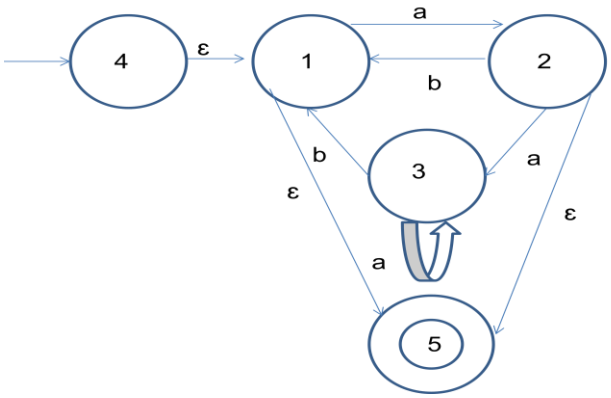
Example 1 for building a RE from FSM

Let M be:

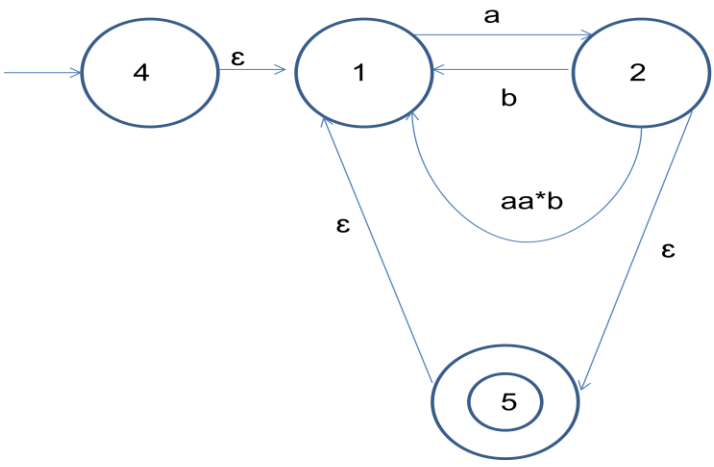


Step 1:Create a new start state and a new accepting state and link them to M

After adding new start state 4 and accepting state 5



Step 2: let rip be state 3



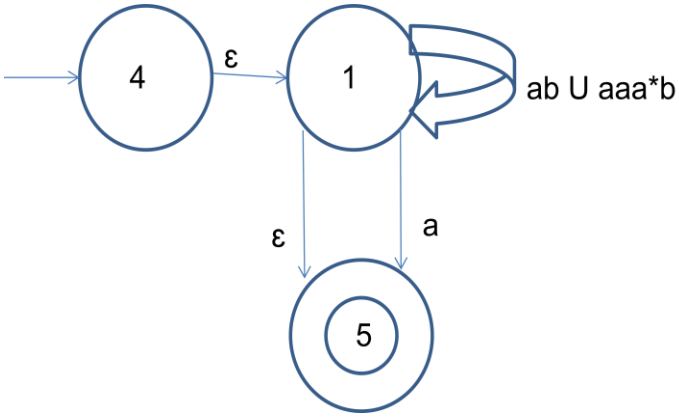
After removing rip state 3

1-2-1:ab U aaa*b

1-2-5:a

Step 3: Let rip be state 2

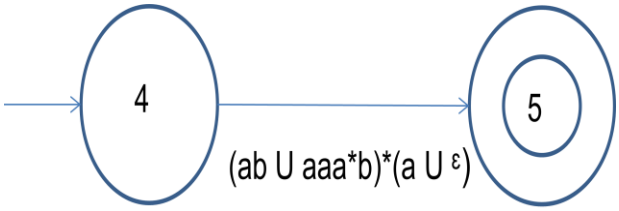
After removing rip state 2



4-1-5: (ab U aaa*b)*(a U ε)

Step 4: Let rip be state 1

After removing rip state 1



RE = (ab U aaa*b)*(a U ε)

Theorem 2 :For Every FSM ,there is an equivalent regular expression

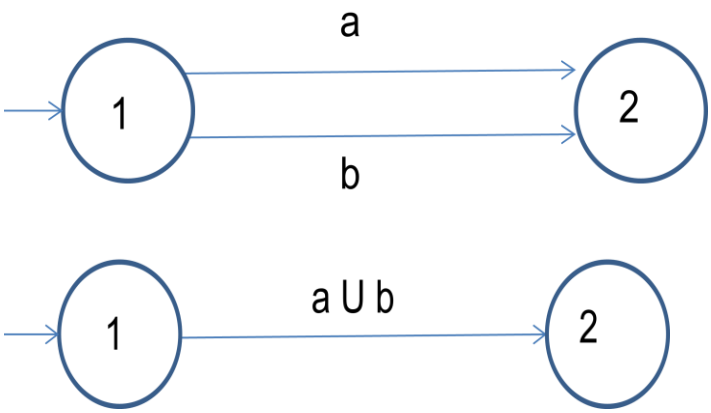
Statement : Every regular language can be defined with a regular expression.

Proof : By Construction

Let FSM M = (K,∑,δ,S,A),construct a regular expression α such that

L(M) = L(α)

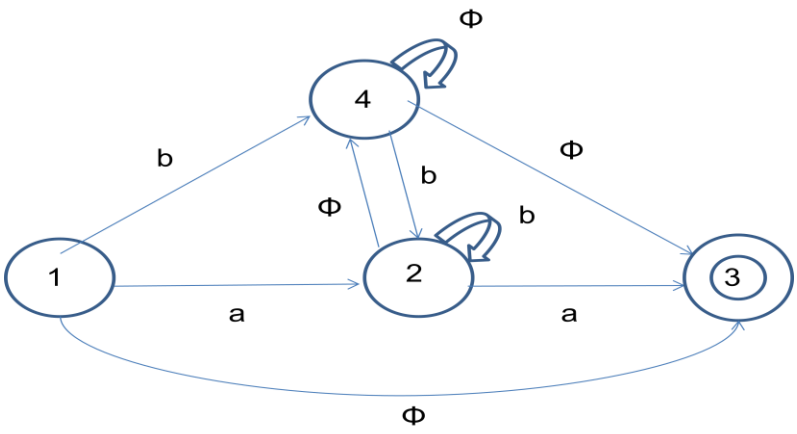
Collapsing Multiple Transitions



{C1,C2,C3.....Cn} - Multiple Transition

Delete and replace by {C1 U C2 U C3.....U Cn}

If any of the transitions are missing, add them without changing L(M) by labeling all of the new transitions with the RE ø.



Select a state rip and remove it and modify the transitions as shown below.

Consider any states p and q.once we remove rip,how can M get from p to q?

Let R(p,q) be RE that labels the transition in M from P to Q.Then the new machine M' will be removing rip,so R'(p,q)

R'(p,q) = R(p,q) U R(p,rip)R(rip,rip)*R(rip,q)

Ripping States out one at a time

R'(1,3) = R(1,3) U R(1,rip)R(rip,rip)*R(rip,3)

= R(1,3) U R(1,2)R(2,2)*R(2,3)

= ø U ab*a

= ab*a

Algorithm to build RE that describes L(M) from any FSM M = (K,∑,δ,S,A)

Two Sub Routines:

1. **standardize** : To convert M to the required form
2. **buildregex** : Construct the required RE from

   modified machine M

1.Standardize (M:FSM)

i. Remove unreachable states from M

ii. Modify start state

iii. Modify accepting states

iv. If there is more than one transition between states p and q ,collapse them to single transition

v. If there is no transition between p and q and p ∉A, q ∉S,then create a transiton between p and q labled Φ

2.buildregex(M:FSM)

i. If M has no accepting states then return RE Φ

ii. If M has only one accepting state ,return RE ε

iii. until only the start state and the accepting state remain do:

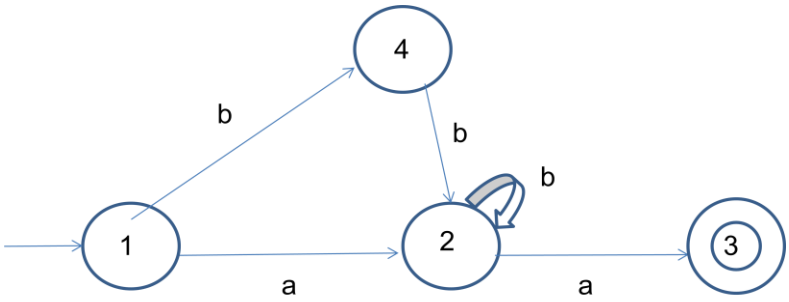
   a. Select some state rip of M

   b. Find R'(p,q) = R(p,q) U R(p,rip).R(rip,rip)*.R (rip,q)

   c. Remove rip on d all transitions into ad out of it

iv. Return the RE that labels from start state to the accepting state

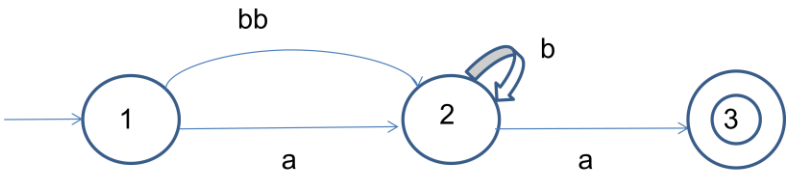
Example 2: Build RE from FSM



Step 1: let RIP be state 4

1-4-2 : bb

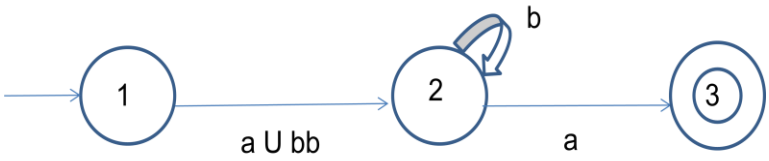
After removing rip state 4



Step 2: Collapse multiple transitions from state 1 to state 2

1-2: a U bb

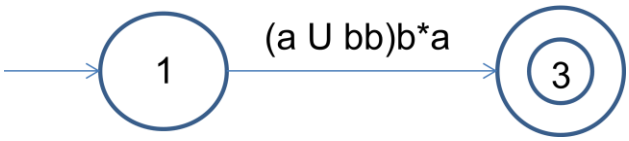
After collapsing multiple transitions from state 1 to state 2



Step 3: let rip be state 2

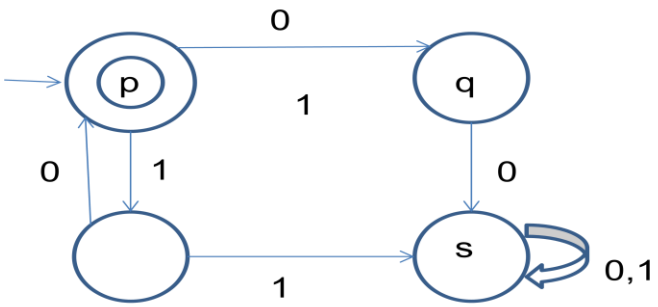
1-3: (a U bb)b*a

After removing rip state 2



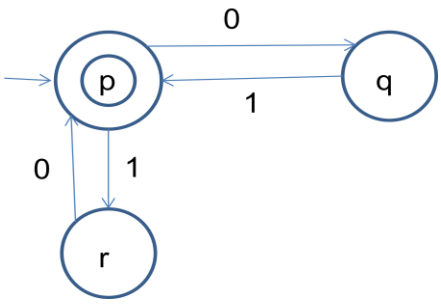
RE = (a U bb)b*a

Example 3: Build RE From FSM



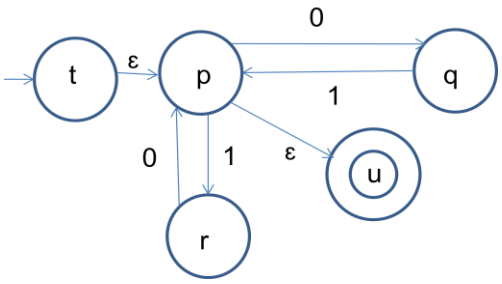
Step 1: Remove state s as it is dead state

After removing state s



Step 2: Add new start state t and new accepting state u

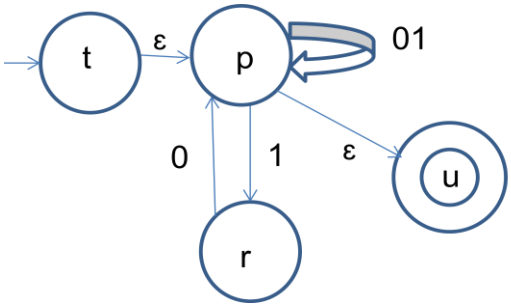
After adding t and u



Step 3: Let rip be state q

p-q-p: 01

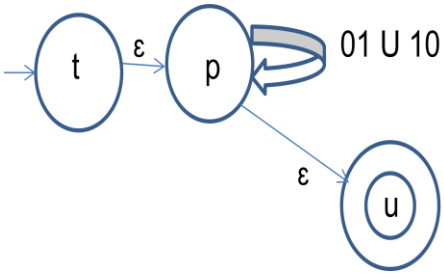
After removing rip state q



Step 4: Let rip be state r

p-r-p: 10

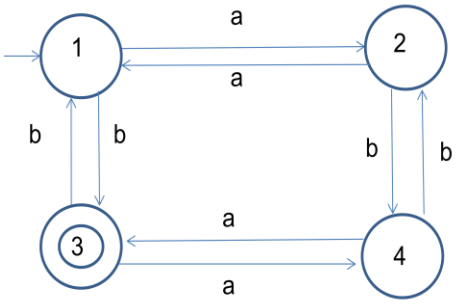
After removing rip state r



RE = (01 U 10)*

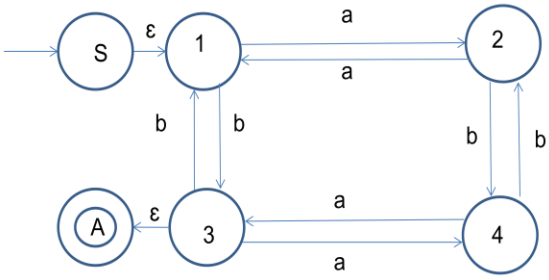
Example 4:A simple FSM with no simple RE

L = {w ε {a,b}* : w contains an even no of a's and an odd number of b's}



[3] even a's odd b's

Step 1: Add new start state S and new accepting state A.

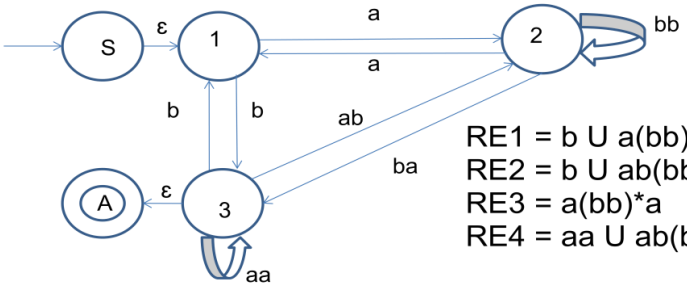


2-4-2: bb
3-4-3: aa
2-4-3: ba
3-4-2: ab

Step 2: let rip be state 4

Result after removing rip state 4

1-2-1: a(bb)*a
1-2-3: a(bb)*ba
3-2-1: ab(bb)*a
3-2-3: ab(bb)*ba

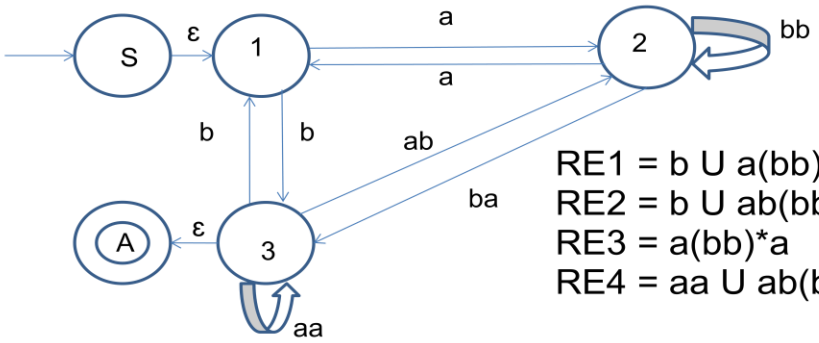


RE1 = b U a(bb)*ba
RE2 = b U ab(bb)*a
RE3 = a(bb)*a
RE4 = aa U ab(bb)*ba

Step 3: let rip be state 2

Result after removing rip state 4

1-2-1: a(bb)*a
1-2-3: a(bb)*ba
3-2-1: ab(bb)*a
3-2-3: ab(bb)*ba

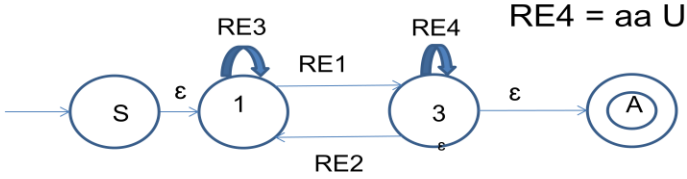


RE1 = b U a(bb)*ba
RE2 = b U ab(bb)*a
RE3 = a(bb)*a
RE4 = aa U ab(bb)*ba

Step 3: let rip be state 2

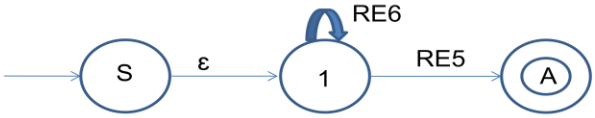
Redrawn

RE1 = b U a(bb)*ba
RE2 = b U ab(bb)*a
RE3 = a(bb)*a
RE4 = aa U ab(bb)*ba



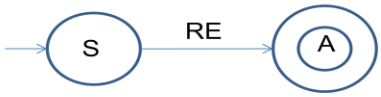
Step 4: let rip be state 3

RE5 = (RE1)(RE4)*
RE6 = (RE3) U (RE1)(RE4)*(RE2)



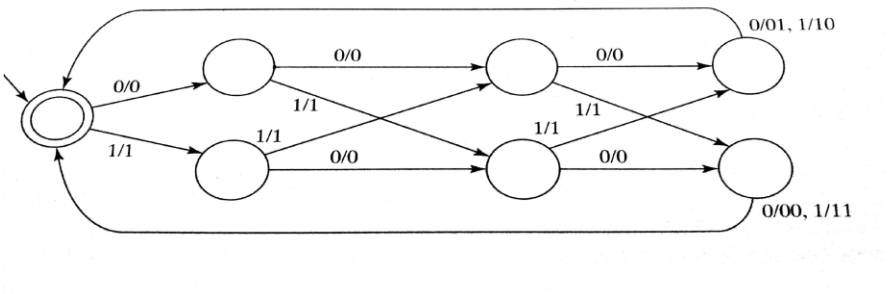
RE = (RE6)* (RE5)

Last Step: let rip be state 1



RE = (RE6)*(RE5)
= ((RE3) U (RE1)(RE4)*(RE2))*((RE1)(RE4)*)
= ((a(bb)*a) U (b U a(bb)*ba)(aa U ab(bb)*ba)*(b U ab(bb)*a))*((b U a(bb)*ba)((aa U ab(bb)*ba)*)

Example 5:Using fsmtoregexheuristic construct a RE for the following FSM(Example 5.3 from textbook)



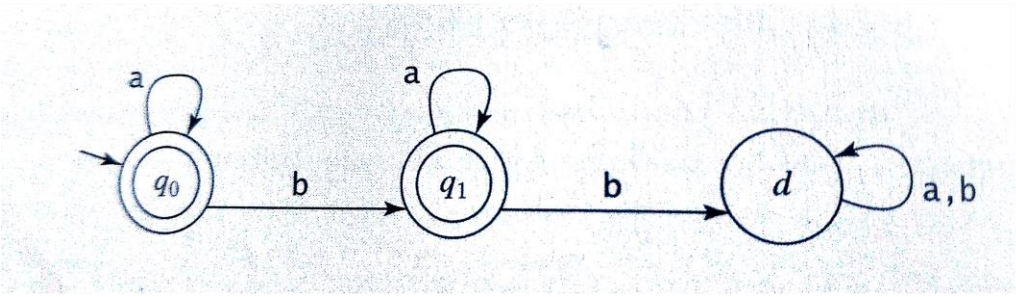
RE = (0000 U 0001 U 1100 U 1101 U 0010 U 1110 U
1100 U 0100 U 0011 U 1111 U 1101 U 0101)

Writing Regular Expressions

- Let L = {w ϵ {a,b}*: there is no more than one b}

  L = {**ε,**b,a,aa,ab,ba,aba,baa,abaa,aabaa,-----}

  RE = a*(b U ε)a*

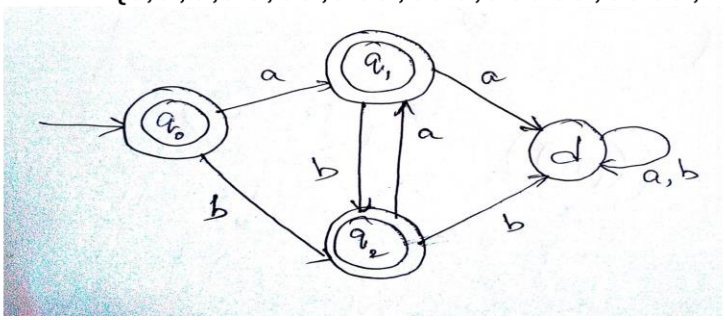


Writing Regular Expressions

- Let L = {w ϵ {a,b}*: No two consecutive letters are same}

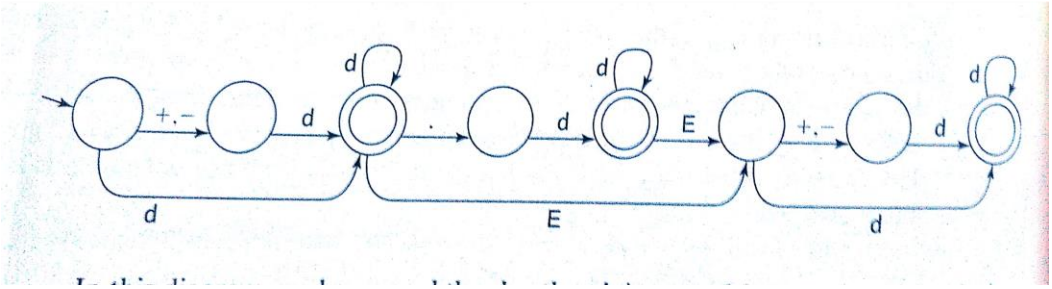
  RE = (b U ε)(ab)*(a U ε) or (a U ε)(ba)*(b U ε)

  L = {ε,a,b,ab,ba,aba,bab,ababa,baba,-----}



Writing Regular Expressions

- Floating point Numbers
  D stands for (0 U 1 U 2 U 3 U 4 U 5 U 6 U 7 U 8 U 9)
  RE = (ε U + U -)D^+(ε U .D^+)(ε U (E(ε U + U -)D^+)
  L = { 24.06, +24.97E-05,------}



Building DFSM

- It is possible to construct a DFSM directly from a set of patterns
- Suppose we are given a set K of n keywords and a text string s.
- Find the occurences of s in keywords K
- K can be defined by RE

  $(\Sigma^*(K_1 \text{ U } K_2 \text{ U}.....\text{U } Kn)\Sigma^*)^+$
- Accept any string in which at least one keyword occurs

Algorithm- buildkeywordFSM

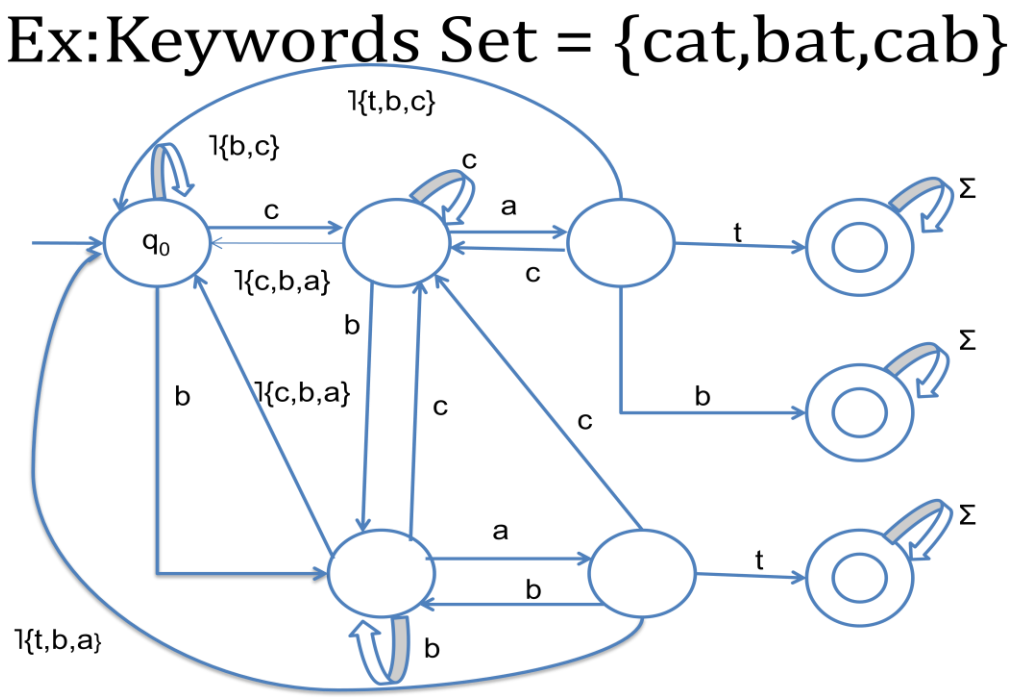
- To build dfsm that accepts any string with atleast one of the specified keywords

Buildkeyword(K:Set of keywords)

- Create a start state q_0
- For each element k of K do

  Create a branch corresponding to k

- Create a set of transitions that describe what to do when a branch dies
- Make the states at the end of each branch accepting



Applications Of Regular Expressions

- Many Programming languages and scripting systems provide support for regular expression matching
- Re's are used in emails to find spam messages
- Meaningful words in protein sequences are called motifs
- Used in lexical analysis
- To Find Patterns in Web
- To Create Legal passwords
- Regular expressions are useful in a wide variety of text processing tasks,

- More generally string processing, where the data need not be textual.
- Common applications include data validation, data scraping (especially web scraping), data wrangling, simple parsing, the production of syntax highlighting systems, and many other tasks.

RE for Decimal Numbers

RE = -? ([0-9]$^+$(\.[0-9]*)? | \.[0-9]$^+$)

- (α)? means the RE α can occur 0 or 1 time.
- (α)* means the RE α can repeat 0 or more times.
- (α)$^+$ means the RE α can repeat 1 or more times.

24.23,-24.23, .12, 12. ----- are some examples

Requirements for legal password

- A password must begin with a letter
- A password may contain only letters numbers and a underscore character
- A password must contain atleast 4 characters and no more than 8 characters

((a-z) U (A-Z))

((a-z) U (A-Z) U (0-9) U _)

((a-z) U (A-Z) U (0-9) U _)

((a-z) U (A-Z) U (0-9) U _)

((a-z) U (A-Z) U (0-9) U _ U ε)

((a-z) U (A-Z) U (0-9) U _ U ε)

((a-z) U (A-Z) U (0-9) U _ U ε)

((a-z) U (A-Z) U (0-9) U _ U ε)

Very lengthy regular expression

Different notation for writing RE

- α means that the pattern α must occur exactly once.
- α* means that the pattern may occur any number of times(including zero).
- α^+ means that the pattern α must occur atleast once.
- α{n,m} means that the pattern must occur **atleast n times** but not more than **m times**
- α{n} means that the pattern must occur **n times exactly**
- So RE of a legal password is :

RE = ((a-z) U (A-Z))((a-z) U (A-Z) U (0-9) U _){3,7}

Examples: RNSIT_17,Bangalor, VTU_2017 etc

- RE for an ip address is :

RE = ((0-9){1,3}(\.(0-9){1,3}){3})

Examples: 121.123.123.123

118.102.248.226

10.1.23.45

Manipulating and Simplifying Regular Expressions

Let α, β, γ represent regular expressions and we have the following identities.

1. Identities involving union
2. Identities involving concatenation
3. Identities involving Kleene Star

Identities involving Union

- Union is Commutative

  α U β = β U α

- Union is Associative

  $(\alpha \cup \beta) \cup \gamma = \alpha \cup (\beta \cup \gamma)$

- Φ is the identity for union

  $\alpha \cup \Phi = \Phi \cup \alpha = \alpha$

- union is idempotent

  $\alpha \cup \alpha = \alpha$

- For any 2 sets A and B, if $B \subseteq A$, then $A \cup B = A$

  $a^* \cup aa = a^*$, since $L(aa) \subseteq L(a^*)$.

Identities involving concatenation

- Concatenation is associative

  $(\alpha\beta)\gamma = \alpha(\beta\gamma)$

- ε is the identity for concatenation

  $\alpha\varepsilon = \varepsilon\alpha = \alpha$

- Φ is a zero for concatenation.

  $\alpha\Phi = \Phi\alpha = \Phi$

- Concatenation distributes over union

  $(\alpha \cup \beta)\gamma = (\alpha\gamma) \cup (\beta\gamma)$

  $\gamma(\alpha \cup \beta) = (\gamma\alpha) \cup (\gamma\beta)$

Identities involving Kleene Star

- $\Phi^* = \varepsilon$
- $\varepsilon^* = \varepsilon$
- $(\alpha^*)^* = \alpha^*$
- $\alpha^*\alpha^* = \alpha^*$

- If $\alpha^* \subseteq \beta^*$ then $\alpha^*\beta^* = \beta^*$
- Similarly If $\beta^* \subseteq \alpha^*$ then $\alpha^*\beta^* = \alpha^*$

  $a^*(a \cup b)^* = (a \cup b)^*$, since $L(a^*) \subseteq L((a \cup b)^*)$.
- $(\alpha \cup \beta)^* = (\alpha^*\beta^*)^*$
- If $L(\beta) \subseteq L(\alpha)$ then $(\alpha \cup \beta)^* = \alpha^*$

$(a \cup \varepsilon)^* = a^*$,since $\{\varepsilon\} \subseteq L(a^*)$.

Simplification of Regular Expressions

1. $((a^* \cup \Phi)^* \cup aa) = (a^*)^* \cup aa$ //$L(\Phi) \subseteq L(a^*)$

   $= a^* \cup aa$ //$(\alpha^*)^* = \alpha^*$

   $= a^*$ // $L(aa) \subseteq L(a^*)$
2. $(b \cup bb)^*b^* = b^*b^*$ //$L(bb) \subseteq L(b^*)$

   $= b^*$ // $\alpha^*\alpha^* = \alpha^*$
3. $((a \cup b)^* b^* \cup ab)^*$

   $= ((a \cup b)^* \cup ab)^*$ //$L(b^*) \subseteq L(a \cup b)^*$

   $= (a \cup b)^*$ //$L(a^*) \subseteq L(a \cup b)^*$)
4. $((a \cup b)^* (a \cup \varepsilon)b^* = (a \cup b)^*$ //$L((a \cup \varepsilon)b^*) \subseteq L(a \cup b)^*$
5. $(\Phi^* \cup b)b^* = (\varepsilon \cup b)b^*$ //$\Phi^* = \varepsilon$

   $= b^*$ //$L(\varepsilon \cup b) \subseteq L(b^*)$
6. $(a \cup b)^*a^* \cup b = (a \cup b)^* \cup b$ // $L(a^*) \subseteq L((a \cup b)^*)$

   $= (a \cup b)^*$ // $L(b) \subseteq L((a \cup b)^*)$
7. $((a \cup b)^+)^* = (a \cup b)^*$

Chapter-7

Regular Grammars

Regular grammars sometimes called as right linear grammars.

A regular grammar G is a quadruple (V, ∑ , R, S)

- V is the rule alphabet which contains nonterminals and terminals.
- ∑ (the set of terminals) is a subset of V
- R (the set of rules) is a finite set of rules of the form

  X → Y
- S (the start symbol) is a nonterminal.

All rules in R must:

- Left-hand side should be a single nonterminal.
- Right-hand side is **ε** or a single terminal or a single terminal followed by a single nonterminal.

Legal Rules

S→a

S→ε

T→aS

Not legal rules

S→aSa

S→TT

aSa→T

S→T

- The language generated by a grammar G = (V, ∑ , R, S) denoted by L(G) is the set of all strings w in ∑* such that it is possible to start with S.
- Apply some finite set of rules in R, and derive w.
- Start symbol of any grammar G will be the symbol on the left-hand side of the first rule in R_G

Example of Regular Grammar

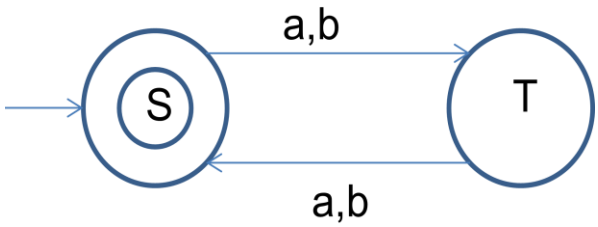
Example 1:Even Length strings

Let L = {wϵ {a, b }*: lwl is even}.

The following regular expression defines L:

((aa) U (ab) U (ba) U (bb))* or ((a U b)(a U b))*

DFSM accepting L



Regular Grammar G defining L

S→**ε**

S→aT

S→bT

T→aS

T→bS

Derivation of string using Rules

Derivation of string "abab"

S => aT

=> abT

=> abaS

=> ababS

=> abab

Regular Grammars and Regular Languages

THEOREM

Regular Grammars Define Exactly the Regular Languages

Statement:

The class of languages that can be defined with regular grammars is exactly the regular languages.

Proof: Regular grammar → FSM

FSM → Regular grammar

The following algorithm constructs an FSM M from a regular grammar G = (V, ∑ , R, S) and assures that

L (M) = L (G):

Algorithm-Grammar to FSM

grammartofsm (**G: regular grammar**) =

1. Create in M a separate state for each nonterminal in V.
2. Make the state corresponding to S the start state.
3. If there are any rules in R of the form X→w, for some

   w ϵ ∑, then create an additional state labeled #.

4. For each rule of the form X→ wY,

add a transition from X to Y labeled w.

5. For each rule of the form X→w, add a transition from X to # labeled w.
6. For each rule of the form X→ε, mark state X as accepting.
7. Mark state # as accepting.
8. If M is incomplete then M requires a dead state.

Add a new state D. For every (q, i) pair for which no transition has already been defined, create a transition from q to D labeled i. For every i in Σ, create a transition from D to D labeled i.

Example 2:Grammar→FSM

Strings that end with aaaa

Let L = {wϵ {a, b }*: w end with the pattern aaaa}.

RE = (a U b)*aaaa

Regular Grammar G

S→aS

S→bS

S→aB

B→aC

C→aD

D→a

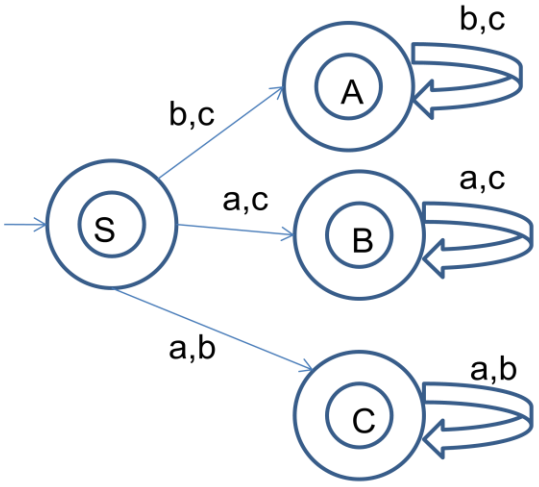
Example 3:The Missing Letter Language

Let ∑ = {a, b, c}.

$L_{Missing}$ = { w : there is a symbol a € ∑ not appearing in w}.

Grammar G generating $L_{Missing}$

FSM for Missing Letter Language



S→ε
S→aB
S→aC
S→bA
S→bC
S→cA
S→CB

A→ ε
A→bA
A→cA
B→ ε
B→aB
B→cB

C→ ε
C→aC
C→bC

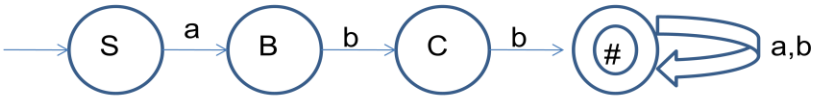
Example 4 :Strings that start with abb.

Let L = {w ϵ {a, b }*: w starting with string abb}.

RE = abb(a U b)*

Regular Grammar G

S→aB
B→bC
C→bT
T→aT
T→bT
T→ε



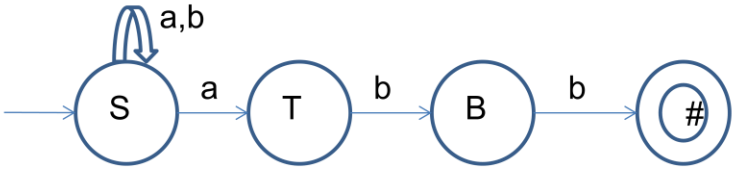
Example 5 :Strings that end with abb.

Let L = {w ϵ {a, b }*: w ending with string abb}.

RE = (a U b)*abb

Regular Grammar G

S→aS
S→bS
S→aT
T→bB
B→b



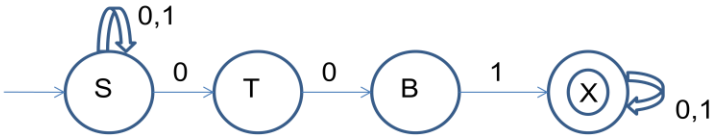
Example 6:Strings that contain substring 001.

Let L = {w ϵ {0, 1 }*: w containing the substring 001}.

RE = (0 U 1)*001(0 U 1)*

Regular Grammar G

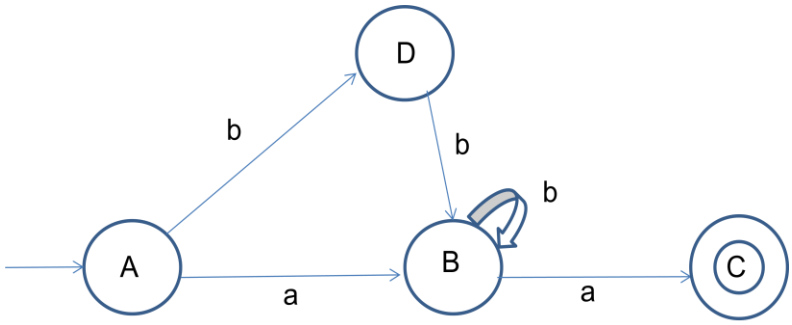
S→0S
S→1S
S→0T
T→0P
P→1X
X→0X
X→1X
X →ε



Algorithm FSM to Grammar

1. Make M deterministic (to get rid of ε-transitions).
2. Create a nonterminal for each state in the new M.
3. The start state becomes the starting nonterminal.
4. For each transition δ(T, a) = U, make a rule of the form T → aU.
5. For each accepting state T, make a rule of the form T → ε.

Example 7:Build grammar from FSM



RE = (a U bb)b*a

Grammar

A→aB

A→bD

B→bB

B→aC

D→bB

C→ε

Derivation of string "aba"

A => aB

=> abB

=> abaC

=> aba

Derivation of string "bba"

A => bB

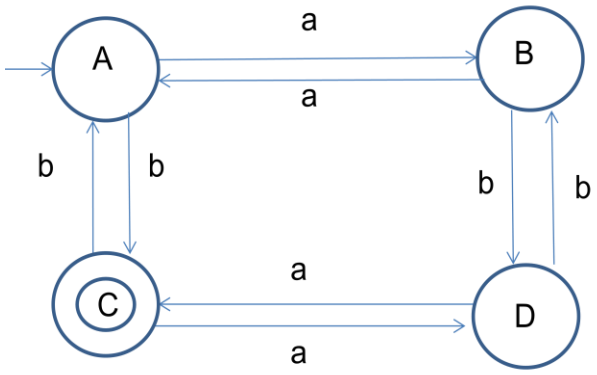
=> bbB

=> bbaC

=> bba

Example 8:A simple FSM with no simple RE

L = {w ε {a,b}* : w contains an even no of a's and an odd number of b's}



Grammar

A→aB

A→bC

B→aA

B→bD

C→bA

C→aD

D→bB

D→aC

C→ε

Derivation of string "ababb"

A => aB

=> abD

=> abaC

=> ababA

=> ababbC

=> ababb

RE,RG and FSM for given Language

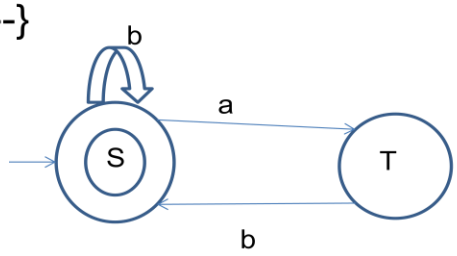
Let L = { wϵ {a, b }*: every a in w is immediately followed by atleast one b.}

L = { b,ab,abab,abb,-------}

RE = (ab U b)*

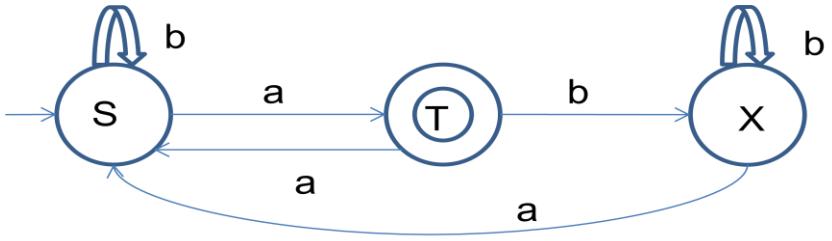
Regular Grammar

S→aT
S→bS
S→ε
T→bS



Satisfying Multiple Criteria

Let L = { wϵ {a, b }*: w contain an **odd number of a's** and **w ends in a**}.



S→bS

S→aT

T→ ε

T→aS

T→bX

X→aS

X→bX

Conclusion on Regular Grammars

- Regular grammars define exactly the regular languages.
- But regular grammars are often used in practice as FSMs and REs are easier to work.
- But as we move further there will no longer exist a technique like regular expressions.
- So we discuss about context-free languages and context-free-grammars are very important to define the languages of push-down automata.

Chapter-8

Regular and Nonregular Languages

- The language a*b* is regular.
- The language $A^nB^n = \{a^nb^n : n>=0\}$ is not regular.
- The language {w Є {a,b}*:every a is immediately followed by b} is regular.
- The language {w Є {a, b}*:every a has a matching b somewhere and no b matches more than one a} is not regular.

- Given a new language L, how can we know whether or not it is regular?

Theorem 1: The Regular languages are countably infinite

Statement:

There are countably infinite number of regular languages.

Proof:

- We can enumerate all the legal DFSMs with input alphabet ∑.
- Every regular language is accepted by at least one of them.

- So there cannot be more regular languages than there are DFSMs.

- But the number of regular languages is infinite

because it includes the following infinite set of

languages:

{a}, { aa} , { aaa}, { aaaa}. { aaaaa}, { aaaaaa } ----

- Thus there are at most a countably infinite number of

regular languages.

Theorem 2 : The finite Languages

Statement: Every finite language is regular.

Proof:

- If L is the empty set, then it is defined by the R.E Ø and so

is regular.

- If it is any finite language composed of the strings

$s_1, s_2, \ldots s_n$ for some positive integer n, then it is defined by

the R.E: $s_1 \cup s_2 \cup \ldots \cup s_n$

- So it too is regular

❖ Regular expressions are most useful when the elements of L match one or more patterns.

❖ FSMs are most useful when the elements of L share some simple structural properties.

Examples:

- **L_1 = {w Є {0-9}*: w is the social security number of the current US president}.**

  L_1 is clearly finite and thus regular. There exists a simple FSM to accept it.

- **L_2 = {1 if Santa Claus exists and 0 otherwise}.**
- **L_3 = {1 if God exists and 0 otherwise}.**

  L_2 and L_3 are perhaps a little less clear.

  So either the simple FSM that accepts { 0} or the simple FSM that accepts { 1} and nothing else accepts L_2 and L_3.

- **L_4 = {1 if there were people in north America more than 10000 years age and 0 otherwise}.**
- **L_5 = {1 if there were people in north America more than 15000 years age and 0 otherwise}.**

L_4 is clear. It is the set { 1}.

L_5 is also finite and thus regular.

- **L_6 = {w Є {0-9}*: w is the decimal representation, without leading 0's, of a prime Fermat number}**

- Fermat numbers are defined by

  $F_n = 2^{2^n} + 1$, *n >= 0.*

- The first five elements of F are {3, 5, 17, 257,65537}.
- All of them are prime. It appears likely that no other Fermat numbers are prime. If that is true,then L_6 is finite and thus regular.

- lf it turns out that the set of Fermat numbers is infinite,then it is almost surely not regular.

Four techniques for showing that a language L(finite or infinite) is regular:

1. Exhibit a R.E for L.
2. Exhibit an FSM for L.
3. Show that the number of equivalence of $\approx_L$ is finite.
4. Exhibit a regular grammar for L.

Closure Properties of Regular Languages

The Regular languages are closed under

- Union
- Concatenation
- Kleene star
- Complement
- Intersection
- Difference
- Reverse
- Letter substitution

Closure under Union, Concatenation and Kleene star

Theorem: The regular languages are closed under union, concatenation and Kleene star.

Proof: By the same constructions that were used in the proof of Kleene's theorem.

Closure under Complement

Theorem:

The regular languages are closed under complement.

Proof:

- If L_1 is regular, then there exists a DFSM $M_1=(K,\sum,\delta,s,A)$ that accepts it.
- The DFSM $M_2=(K, \sum,\delta,s,K-A)$, namely M_1 with accepting and nonaccepting states swapped, accepts $\neg(L(M_1)$ because it rejects all strings that M_1 accepts and rejects all strings that M_1 accepts.

Steps:

1. Given an arbitrary NDFSM M_1,construct an equivalent DFSM M' using the algorithm ndfsmtodfsm.
2. If M_1 is already deterministic, M' = M_1.
3. M' must be stated completely, so if needed add dead state and all transitions to it.
4. Begin building M_2 by setting it equal to M'.
5. Swap accepting and nonaccepting states. So $M_2=(K, \sum,\delta,s,K-A)$

Example:

- Let L = {w Є {0,1}* : w is the string ending with 01}

  RE = (0 U 1)*01
- The complement of L(M) is the DFSM that will accept strings that do not end with 01.

Closure under Intersection

Theorem:

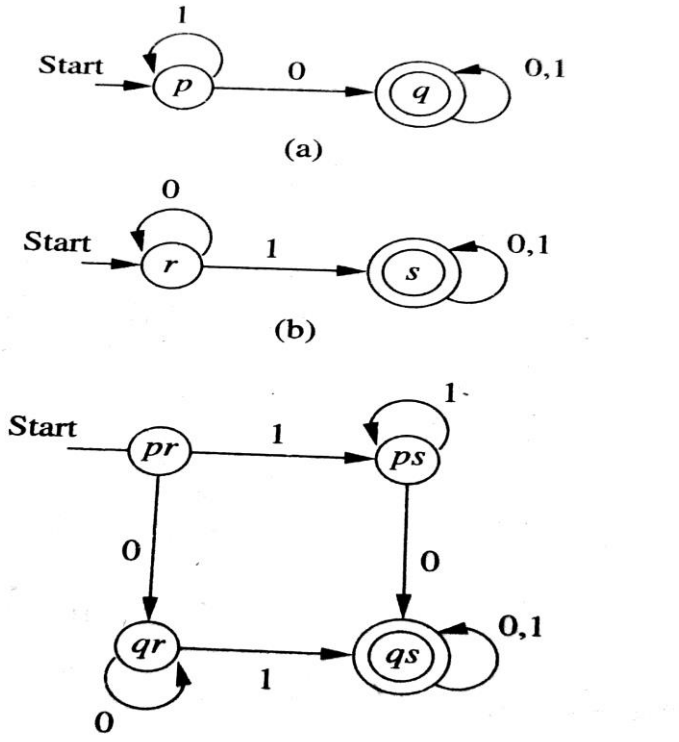
The regular languages are closed under intersection.

Proof:

- Note that

  $L(M_1) \cap L(M_2) = \neg (\neg L(M_1) \cup \neg L(M_2))$.

- We have already shown that the regular languages are closed under both complement and union.
- Thus they are closed under intersection.
- Example:



- Fig (a) is DFSM L1 which accepts strings that have 0.
- Fig(b) is DFSM L2 which accepts strings that have 1.

- Fig(c) is Intersection or product construction which accepts that have both 0 and 1.

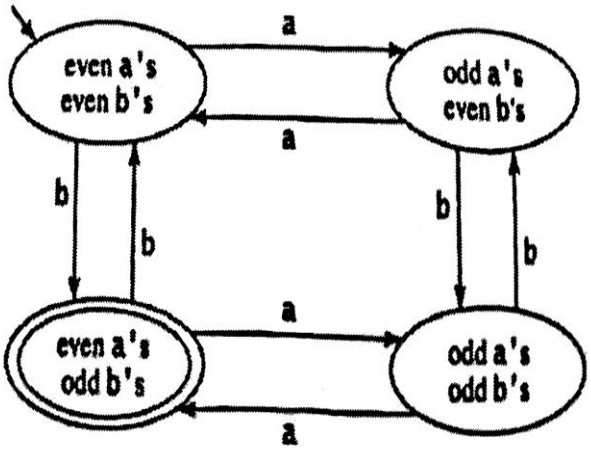
The Divide and Conquer Approach

- Let L = {w Є {a,b}* : w contains an even number of a's and an odd number of b's and all a's come in runs of three }.
- L is regular because it is the intersection of two regular languages,

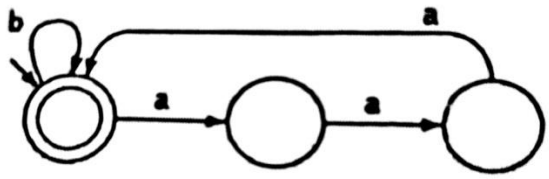
  $L = L_1 \cap L_2$, where
- L1 = {w Є {a,b}* : w contains an even number of a's

  and an odd number of b's},and

  L2 = {w Є {a,b}*: all a's come in runs of three}.
- L1 is regular as we have an FSM accepting L1



- L2 = {w Є {a,b}*: all a's come in runs of three}.
- L2 is regular as we have an FSM accepting L2



L = {w Є {a,b}* : w contains an even number of a's and an odd number of b's and all a's come in runs of three }.

L is regular because it is the intersection of two regular languages,$L = L_1 \cap L_2$

Closure under Set difference

Theorem:

The regular languages are closed under set difference.

Proof:

$L(M_1) - L(M_2) = L(M_1) \cap \neg L(M_2)$

- Regular languages are closed under both complement

and intersection is shown.

- Thus regular languages are closed under set difference.

Closure under Reverse

Theorem:

The regular languages are closed under reverse.

Proof:

- $L^R = \{ w \in \sum^* : w = x^R \text{ for some } x \in L\}$.

Example:

1. Let L = {001,10,111} then L^R = {100,01,111}
2. Let L be defined by RE (0 U 1)0* then L^R is 0*(0 U 1)

reverse(L) = $\{x \in \Sigma^* : x = w^R \text{ for some } w \in L\}$.

By construction.

- Let M = (K, Σ, δ, s, A) be any FSM that accepts L.
- Initially, let M′ be M.

- Reverse the direction of every transition in M′.
- Construct a new state q. Make it the start state of M′.
- Create an ε-transition from q to every state that was an accepting state in M.
- M′ has a single accepting state, the start state of M.

Closure under letter substitution or Homomorphism

- The regular languages are closed under letter substitution.
- Consider any two alphabets, $\sum_1$ and $\sum 2$.
- Let **sub** be any function from $\sum_1$ to $\sum_2$*.
- Then **letsub** is a letter substitution function from L_1 to L_2 iff **letsub**(L_1) = { w Є $\sum_2$*:∃y Є L_1(w = y except that every character c of y has been replaced by **sub**(c))}.
- Example 1

Consider $\sum_1$ = {a,b} and $\sum_2$ = {0,1}

Let **sub** be any function from $\sum_1$ to $\sum_2$*.

sub(a) = 0, sub(b) = 11

letsub**(**$a^n b^n$: n >= 0}) = { $0^n 1^{2n}$: n >= 0}

- Example 2

Consider $\sum_1$ = {0,1,2} and $\sum_2$ = {a,b}

Let **h** be any function from $\sum_1$ to $\sum_2$*.

h(0) = a, h(1) = ab, h(2) = ba

h(0120) = h(0)h(1)h(2)h(0)

= aabbaa

h(01*2) = h(0)(h(1))*h(2)

= a(ab)*ba

Long Strings Force Repeated States

Theorem: Let M=(K,∑,δ,s,A) be any DFSM. If M accepts any string of length |K | or greater, then that string will force M to visit some state more than once.

Proof:

- M must start in one of its states.
- Each time it reads an input character, it visits some state. So ,in processing a string of length n, M creates a total of n+1 state visits.
- If n+1 > | K |, then, by the pigeonhole principle, some state must get more than one visit.
- So, if n>= | K |,then M must visit at least one state more than once.

The Pumping Theorem for Regular Languages

Theorem: If L is regular language, then:

∃k >= 1 (∀strings w ϵ L, where |w| >= k (∃x, y, z (w = xyz,

|xy| <= k,

y ≠ **ε**,and

∀q >= 0(xy^qz ϵ L)))).

Proof:

- If L is regular then it is accepted by some DFSM M=(K,∑,δ,s,A).

  Let k be |K|
- Let w be any string in L of length k or greater.
- By previous theorem to accept w, M must traverse some loop at least once.

- We can carve w up and assign the name y to the first substring to drive M through a loop.
- Then x is the part of w that precedes y and z is the part of w that follows y.
- We show that each of the last three conditions must then hold:
- **|xy| <= k**

  M must not traverse thru a loop.

  It can read k - 1 characters without revisiting any states.

  But kth character will take M to a state visited before.

- **$y \neq \varepsilon$**

  Since M is deterministic, there are no loops traversed by **ε.**

- **$\forall q >= 0 (xy^q z \in L)$**

  y can be pumped out once and the resulting string must

  be in L.

Steps to prove Language is not regular by contradiction method.

1. Assume L is regular.
2. Apply pumping theorem for the given language.
3. Choose a string w, where w ϵ L and IwI >= k.
4. Split w into xyz such that |xy| <= k and $y \neq \varepsilon$.
5. Choose a value for q such that $xy^q z$ is not in L.
6. Our assumption is wrong and hence the given language is not regular.

Problems on Pumping theorem (Showing that the language is not regular)

1. **Show that A^nB^n is not Regular**

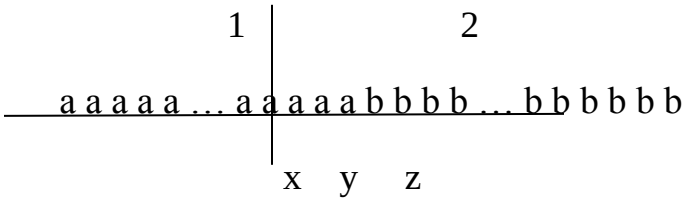
Let *L* be $A^nB^n = \{ a^nb^n : n >= 0\}$.

Proof by contradiction.

Assume the given language is regular.

Apply pumping theorem and split the string w into xyz

Choose w to be a^kb^k (We get to choose any w).



We show that there is no x, y, z with the required properties:

$k \leq |xy|$,

$y \neq \varepsilon$

$\forall$ q > = 0 (xy^qz is in L y must be in region 1.

So $y = a^p \leq$ Since |xy| 1.≥for some p Let q = 2, producing: ak+pbk L, since it has more a∉which 's than b' s.

2. **$\{a^ib^j : i, j \geq 0$ and $i - j = 5\}$.**

- Not regular.
- L consists of all strings of the form a*b* where the number of a's is five more than the number of b's.
- We can show that L is not regular by pumping.
- Let $w = a^{k+5}b^k$.
- Since $|xy| \leq k$, y must equal a^p for some $p > 0$.

- We can pump y out once, which will generate the string $a^{k+5-p}b^k$, which is not in L because the number of a's is is less than 5 more than the number of b's.

Subject: Automata Theory and Computability
Sub Code: 15CS54
Module -III
Context-Free Grammars and Pushdown Automata (PDA)

Course Outcomes-(CO)

At the end of the course student will be able to:

i. Explain core concepts in Automata and Theory of Computation.
ii. Identify different Formal language Classes and their Relationships.
iii. Design Grammars and Recognizers for different formal languages.
iv. Prove or disprove theorems in automata theory using their properties.
v. Determine the decidability and intractability of Computational problems.

Syllabus of Module 3

i. Context-Free Grammars(CFG): Introduction to Rewrite Systems and Grammars
ii. CFGs and languages, designing CFGs,
iii. Simplifying CFGs,
iv. Proving that a Grammar is correct,
v. Derivation and Parse trees, Ambiguity,
vi. Normal Forms.
vii. Pushdown Automata (PDA): Definition of non-deterministic PDA,
viii. Deterministic and Non-deterministic PDAs,
ix. Non-determinism and Halting, Alternative equivalent definitions of a PDA,
x. Alternatives those are not equivalent to PDA.

Text Books:

1. Elaine Rich, Automata, Computability and Complexity, 1st Edition, Pearson Education, 2012/2013. Text Book 1: Ch 11, 12: 11.1 to 11.8, 12.1 to 12.6 excluding 12.3.
2. K L P Mishra, N Chandrasekaran , 3rd Edition, Theory of Computer Science, PHI, 2012

Reference Books:

1. John E Hopcroft, Rajeev Motwani, Jeffery D Ullman, Introduction to Automata Theory, Languages, and Computation, 3rd Edition, Pearson Education, 2013
2. Michael Sipser : Introduction to the Theory of Computation, 3rd edition, Cengage learning,2013
3. John C Martin, Introduction to Languages and The Theory of Computation, 3rd Edition,Tata McGraw –Hill Publishing Company Limited, 2013
4. Peter Linz, “An Introduction to Formal Languages and Automata”, 3rd Edition, Narosa Publishers, 1998
5. Basavaraj S. Anami, Karibasappa K G, Formal Languages and Automata theory, WileyIndia, 2012

Learning Outcomes:

At the end of the module student should be able to:

Sl.No	TLO's
1.	Define context free grammars and languages
2.	Design the grammar for the given context free languages.
3.	Apply the simplification algorithm to simplify the given grammar
4.	Prove the correctness of the grammar
5.	Define leftmost derivation and rightmost derivation
6.	Draw the parse tree to a string for the given grammar.
7.	Define ambiguous and inherently ambiguous grammars.
8.	Prove whether the given grammar is ambiguous grammar or not.
9.	Define Chomsky normal form. Apply the normalization algorithm to convert the grammar to Chomsky normal form.
10.	Define Push down automata (NPDA). Design a NPDA for the given CFG.
11.	Design a DPDA for the given language.
12.	Define alternative equivalent definitions of a PDA.

1. Introduction to Rewrite Systems and Grammars

What is Rewrite System?

A rewrite system (or production system or rule based system) is a list of rules, and an algorithm for applying them. Each rule has a left-hand side and a right hand side.

$$X \rightarrow Y$$

(LHS) (RHS)

Examples of rewrite-system rules: $S \rightarrow aSb$, $aS \rightarrow \varepsilon$, $aSb \rightarrow bSabSa$

When a rewrite system R is invoked on some initial string w, it operates as follows:

simple-rewrite(R: rewrite system, w: initial string) =

1. Set working-string to w.
2. Until told by R to halt do:
   - **1.1** Match the LHS of some rule against some part of working-string.
   - **1.2** Replace the matched part of working-string with the RHS of the rule that was matched.
3. Return working-string.

If it returns some string s then R can derive s from w or there exists a derivation in R of s from w.

Examples:

1. A rule is simply a pair of strings where, if the string on the LHS matches, it is replaced by the string on the RHS.
2. The rule $axa \rightarrow aa$ squeeze out whatever comes between a pair of a's.
3. The rule $ab^*ab^*a \rightarrow aaa$ squeeze out b's between a's.

Rewrite systems can be used to define functions. We write rules that operate on an input string to produce the required output string. Rewrite systems can be used to define languages. The rewrite system that is used to define a language is called a grammar.

Grammars Define Languages

A grammar is a set of rules that are stated in terms of two alphabets:

- a terminal alphabet, Σ, that contains the symbols that make up the strings in L(G),
- a nonterminal alphabet, the elements of which will function as working symbols that will be used while the grammar is operating. These symbols will disappear by the time the grammar finishes its job and generates a string.
- A grammar has a unique start symbol, often called S.

A rewrite system formalism specifies:

- The form of the rules that are allowed.
- The algorithm by which they will be applied.
- How its rules will be applied?

Using a Grammar to Derive a String

Simple-rewrite (G, S) will generate the strings in L(G). The symbol ⇒ to indicate steps in a derivation.

Given: $S \rightarrow aS$ ---- rule 1

$S \rightarrow \varepsilon$ ---- rule 2

A derivation could begin with: $S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb$

Generating Many Strings

LHS of Multiple rules may match with the working string.

Given: $S \rightarrow aSb$ ----- rule 1

$S \rightarrow bSa$ ----- rule 2

$S \rightarrow \varepsilon$ ----- rule 3

Derivation so far: $S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow$

Three are three choices at the next step:

$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb$ (using rule 1),

$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabSabb$ (using rule 2),

$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb$ (using rule 3).

One rule may match in more than one way.

Given: $S \rightarrow aTTb$ ----- rule 1

$T \rightarrow bTa$ ----- rule 2

$T \rightarrow \varepsilon$ ----- rule 3

Derivation so far: $S \Rightarrow aTTb \Rightarrow$

Two choices at the next step:

$S \Rightarrow aTTb \Rightarrow abTaTb \Rightarrow$

$S \Rightarrow aTTb \Rightarrow aTbTab \Rightarrow$

When to Stop

Case 1: The working string no longer contains any nonterminal symbols (including, when it is ε). In this case, we say that the working string is generated by the grammar.

Example: S ⇒ aSb ⇒ aaSbb ⇒ aabb

Case 2: There are nonterminal symbols in the working string but none of them appears on the left-hand side of any rule in the grammar. In this case, we have a blocked or non-terminated derivation but no generated string.

Given: S → aSb ----- rule 1
S→ bTa ----- rule 2
S → ε ----- rule 3

Derivation so far: S ⇒ aSb ⇒ abTab ⇒

Case 3: It is possible that neither case 1 nor case 2 is achieved.

Given: S → Ba -----rule 1
B → bB -----rule 2

Then all derivations proceed as: S ⇒ Ba ⇒ bBa ⇒ bbBa ⇒ bbbBa ⇒ bbbbBa ⇒ ...

The grammar generates the language Ø.

2. Context –Free Grammar and Languages

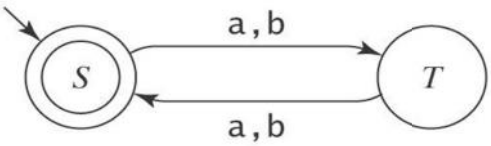
Recall Regular Grammar which has a left-hand side that is a single nonterminal and have a right-hand side that is ε or a single terminal or a single terminal followed by a single nonterminal.

X → Y
(NT) (ε or T or T NT)

Example: L = {w Î {a, b}* : |w| is even} RE = ((aa) (ab) (ba) (bb))*

G:
$S \to \varepsilon$
$S \to aT$
$S \to bT$
$T \to aS$
$T \to bS$

M:



Context Free Grammars

X → Y
(NT) (No restriction)

No restrictions on the form of the right hand sides. But require single non-terminal on left hand side.

Example: S → ε, S → a, S → T, S → aSb, S→ aSbbT are allowed.

ST→ aSb, a→ aSb, ε → a are not allowed.

The name for these grammars “Context Free” makes sense because using these rules the decision to replace a nonterminal by some other sequence is made without looking at the context in which the nonterminals occurs.

Definition Context-Free Grammar

A context-free grammar G is a quadruple, (V, Σ, R, S), where:

- V is the rule alphabet, which contains nonterminals and terminals.
- Σ (the set of terminals) is a subset of V,
- R (the set of rules) is a finite subset of $(V - \Sigma) \times V^*$,
- S (the start symbol) is an element of V - Σ.

Given a grammar G, define $x \Rightarrow_G y$ to be the binary relation derives-in-one-step, defined so that $\forall x,y \in V^*$ ($x \Rightarrow_G y$ iff $x = \alpha A \beta$, $y = \alpha \gamma \beta$ and there exists a rule $A \rightarrow \gamma$ is in R_G)

Any sequence of the form $w_0 \Rightarrow_G w_1 \Rightarrow_G w_2 \Rightarrow_G \ldots \Rightarrow_G w_n$ is called a derivation in G. Let $\Rightarrow_G^*$ be the reflexive, transitive closure of $\Rightarrow_G$. We'll call $\Rightarrow_G^*$ the derive relation.

A derivation will halt whenever no rules on the left hand side matches against working-string. At every step, any rule that matches may be chosen.

Language generated by G, denoted L(G), is: $L(G) = \{w \in \Sigma^* : S \Rightarrow_G^* w\}$. A language L is context-free iff it is generated by some context-free grammar G. The context-free languages (or CFLs) are a proper superset of the regular languages.

Example: $L = A^nB^n = \{a^nb^n : n \geq 0\} = \{\varepsilon, ab, aabb, aaabbb, \ldots\}$

G = {{S, a, b}, {a, b}, R, S}, where: R = { $S \rightarrow aSb$, $S \rightarrow \varepsilon$}

Example derivation in G: $S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \Rightarrow aaabbb$ or $S \Rightarrow^* aaabbb$

Recursive Grammar Rules

A grammar is recursive iff it contains at least one recursive rule. A rule is recursive iff it is $X \rightarrow w_1 Y w_2$, where: $Y \Rightarrow^* w_3 X w_4$ for some w_1, w_2, w_3, and w_4 in V^*. Expanding a non-terminal according to these rules can eventually lead to a string that includes the same non-terminal again.

Example1: $L = A^nB^n = \{a^nb^n : n \geq 0\}$ Let G = ({S, a, b}, {a, b}, {$S \rightarrow a\ S\ b$, $S \rightarrow \varepsilon$}, S)

Example 2: Regular grammar whose rules are {$S \rightarrow a\ T$, $T \rightarrow a\ W$, $W \rightarrow a\ S$, $W \rightarrow a$ }

Example 3: The Balanced Parenthesis language

Bal = {w ∈ {),(}*: the parenthesis are balanced} = {ε , (), (()), ()(), (()()) …….....}

G={{S,),(}, {),(},R,S} where R={ $S \rightarrow \varepsilon$ $S \rightarrow SS$ $S \rightarrow (S)$ }

Some example derivations in G:

$S \Rightarrow (S) \Rightarrow ()$

$S \Rightarrow (S) \Rightarrow (SS) \Rightarrow ((S)S) \Rightarrow (() S)) \Rightarrow (() (S)) \Rightarrow (()())$

So, $S \Rightarrow^* ()$ and $S \Rightarrow^* (()())$

Recursive rules make it possible for a finite grammar to generate an infinite set of strings.

Self-Embedding Grammar Rules

A grammar is self-embedding iff it contains at least one self-embedding rule. A rule in a grammar G is self-embedding iff it is of the form $X \rightarrow w_1 Y w_2$, where $Y \Rightarrow^* w_3 X w_4$ and both $w_1 w_3$ and $w_4 w_2$ are in Σ^+. No regular grammar can impose such a requirement on its strings.

Example: $S \rightarrow aSa$ is self-embedding

$S \rightarrow aS$ is recursive but not self- embedding

$S \rightarrow aT$

$T \rightarrow Sa$ is self-embedding

Example : PalEven = $\{ww^R : w \in \{a, b\}^*\}$= The L of even length palindrome of a's and b's.

L = {ε, aa, bb, aaaa, abba, baab, bbbb, ..……}

G = {{S, a, b}, {a, b}, R, S}, where:

R = { $S \rightarrow aSa$ ----- rule 1

$S \rightarrow bSb$ ----- rule 2

$S \rightarrow \varepsilon$ ----- rule 3 }.

Example derivation in G:

$S \Rightarrow aSa \Rightarrow abSba \Rightarrow abba$

Where Context-Free Grammars Get Their Power

If a grammar G is not self-embedding then L(G) is regular. If a language L has the property that every grammar that defines it is self-embedding, then L is not regular.

More flexible grammar-writing notations

a. Notation for writing practical context-free grammars. The symbol | should be read as "or". It allows two or more rules to be collapsed into one.

Example:

$S \rightarrow a S b$

$S \rightarrow b S a$ can be written as $S \rightarrow a S b \mid b S a \mid \varepsilon$

$S \rightarrow \varepsilon$

b. Allow a nonterminal symbol to be any sequence of characters surrounded by angle brackets.

Example1: BNF for a Java Fragment

<block> ::= {<stmt-list>} | {}

<stmt-list> ::= <stmt> | <stmt-list> <stmt>

<stmt> ::= <block> | while (<cond>) <stmt> |

if (<cond>) <stmt> |

do <stmt> while (<cond>); |

<assignment-stmt>; |

return | return <expression> |

<method-invocation>;

Example2: A CFG for C++ compound statements:

<compound stmt> → { <stmt list> }

<stmt list> → <stmt> <stmt list> | epsilon

<stmt> → <compound stmt>

<stmt> → if (<expr>) <stmt>

<stmt> → if (<expr>) <stmt> else <stmt>

<stmt> → while (<expr>) <stmt>

<stmt> → do <stmt> while (<expr>) ;

<stmt> → for (<stmt> <expr> ; <expr>) <stmt>

<stmt> → case <expr> : <stmt>

<stmt> → switch (<expr>) <stmt>

<stmt> → break ; | continue ;

<stmt> → return <expr> ; | goto <id> ;

Example3: A Fragment of English Grammar

Notational conventions used are

- Nonterminal = whose first symbol is an uppercase letter
- NP = derive noun phrase
- VP = derive verb phrase

S → NP VP

NP → the Nominal | a Nominal | Nominal | ProperNoun | NP PP

Nominal → N | Adjs N

N → cat | dogs | bear | girl | chocolate | rifle

ProperNoun → Chris | Fluffy

Adjs → Adj Adjs | Adj

Adj → young | older | smart

VP → V | V NP | VP PP

V → like | likes | thinks | shots | smells

PP → Prep NP

Prep → with

3. Designing Context-Free Grammars

If L has a property that every string in it has two regions & those regions must bear some relationship to each other, then the two regions must be generated in tandem. Otherwise, there is no way to enforce the necessary constraint.

Example 1: L = $\{a^n b^n c^m : n, m \geq 0\}$ = L = {ε, ab, c, abc, abcc, aabbc, …….}
The c^m portion of any string in L is completely independent of the $a^n b^n$ portion, so we should generate the two portions separately and concatenate them together.
G = ({S, A, C, a, b, c}, {a, b, c}, R, S} where:

R = { S → AC /* generate the two independent portions
A → aAb | ε /* generate the $a^n b^n$ portion, from the outside in
C → cC | ε } /* generate the c^m portion

Example derivation in G for string abcc:
S ⇒ AC ⇒ aAbC ⇒ abC ⇒ abcC ⇒ abccC ⇒ abcc

Example 2: L={ $a^i b^j c^k$: j=i+k, i ,k ≥ 0} on substituting j=i+k ⇒ L = $\{a^i b^i b^k c^k : i, k \geq 0\}$
L = {ε, abbc, aabbbbcc, abbbcc …….}
The $a^i b^i$ portion of any string in L is completely independent of the $b^k c^k$ portion, so we should generate the two portions separately and concatenate them together.
G = ({S, A, B, a, b, c}, {a, b, c}, R, S} where:

R = { S → AB /* generate the two independent portions
A → aAb | ε /* generate the $a^i b^i$ portion, from the outside in
B → bBc | ε } /* generate the $b^k c^k$ portion

Example derivation in G for string abbc:
S ⇒ AB ⇒ aAbB ⇒ abB ⇒ abbBc ⇒ abbc

Example 3: L={ $a^i b^j c^k$: i=j+k, j ,k ≥0} on substituting i=j+k ⇒ L = $\{a^k a^j b^j c^k : j, k \geq 0\}$
L = {ε, ac, ab, aabc, aaabcc, …….}
The $a^i b^i$ is the inner portion and $a^k c^k$ is the outer portion of any string in L.
G = ({S, A, a, b, c}, {a, b, c}, R, S} where:

R = { S → aSc | A /* generate the $a^k c^k$ outer portion
A → aAb | ε /* generate the $a^j b^j$ inner portion}

Example derivation in G for string aabc:
S ⇒ aSc ⇒ aAc ⇒ aaAbc ⇒ aabc

Example 4: L = $\{a^n ww^R b^n : w \in \{a, b\}^*\}$ = {ε, ab, aaab, abbb, aabbab, aabbbbab, ..……}
The $a^n b^n$ is the inner portion and ww^R is the outer portion of any string in L.
G = {{S, A, a, b}, {a, b}, R, S}, where:

R = {S → aSb ----- rule 1
S → A ----- rule 2
A→ aAa ----- rule 3
A→ bAb ----- rule 4
A → ε ----- rule 5 }.

Example derivation in G for string aabbab:
S ⇒ aSb ⇒ aAb ⇒ aaAab ⇒ aabAbab ⇒ aabbab

Example 5: Equal Numbers of a's and b's. = $\{w \in \{a, b\}^*: \#_a(w) = \#_b(w)\}$.

L = {ε, ab, ba, abba, aabb, baba, bbaa, …….}

G = {{S, a, b}, {a, b}, R, S}, where:

R = { $S \rightarrow aSb$ ----- rule 1

$S \rightarrow bSa$ ----- rule 2

$S \rightarrow SS$ ----- rule 3

$S \rightarrow \varepsilon$ ----- rule 4 }.

Example derivation in G for string abba:

$S \Rightarrow aSa \Rightarrow abSba \Rightarrow abba$

Example 6

$L = \{a^ib^j : 2i = 3j + 1\} = \{a^2b^1 , a^5b^3 , a^8b^5 \ldots\ldots\}$

G = {{S, a, b}, {a, b}, R, S}, where:

a^ib^j	$2i = 3j + 1$
a^2b^1	$2*2= 3*1 + 1 = 4$
a^5b^3	$2*5= 3*3 + 1 = 10$
a^8b^5	$2*8= 3*5 + 1 = 16$

R={ $S \rightarrow aaaSbb \mid aab$ }

Example derivation in G for string aaaaabbb:

$S \Rightarrow aaaSbb \Rightarrow aaaaabbb$

4. Simplifying Context-Free Grammars

Two algorithms used to simplify CFG

a. To find and remove unproductive variables using removeunproductive(G:CFG)
b. To find and remove unreachable variables using removeunreachable(G:CFG)

a. Removing Unproductive Nonterminals:

Removeunproductive (G: CFG) =

1. G′ = G.
2. Mark every nonterminal symbol in G′ as unproductive.
3. Mark every terminal symbol in G′ as productive.
4. Until one entire pass has been made without any new symbol being marked do:
   For each rule $X \rightarrow \alpha$ in R do:
   If every symbol in α has been marked as productive and X has not yet been marked as productive then:
   Mark X as productive.
5. Remove from G′ every unproductive symbol.
6. Remove from G′ every rule that contains an unproductive symbol.
7. Return G′.

Example: G = ({S, A, B, C, D, a, b}, {a, b}, R, S), where

R = { $S \rightarrow AB \mid AC$

$A \rightarrow aAb \mid \varepsilon$

$B \rightarrow bA$

$C \rightarrow bCa$

$D \rightarrow AB$ }

1) a and b terminal symbols are productive
2) A is productive(because $A \rightarrow aAb$)
3) B is productive(because $B \rightarrow bA$)
4) S & D are productive(because $S \rightarrow AB$ & $D \rightarrow AB$)
5) C is unproductive

On eliminating C from both LHS and RHS the rule set R′ obtained is

R′ = { $S \rightarrow AB$ $A \rightarrow aAb \mid \varepsilon$ $B \rightarrow bA$ $D \rightarrow AB$ }

b. Removing Unreachable Nonterminals

Removeunreachable (G: CFG) =

1. G′ = G.
2. Mark S as reachable.
3. Mark every other nonterminal symbol as unreachable.
4. Until one entire pass has been made without any new symbol being marked do:
   For each rule $X \rightarrow \alpha A \beta$ (where $A \in V - \Sigma$) in R do:
   If X has been marked as reachable and A has not then:
   Mark A as reachable.
5. Remove from G′ every unreachable symbol.
6. Remove from G′ every rule with an unreachable symbol on the left-hand side.
7. Return G′.

Example

G = ({S, A, B, C, D, a, b}, {a, b}, R, S), where

R′ = {$S \rightarrow AB$

$A \rightarrow aAb \mid \varepsilon$

$B \rightarrow bA$

$D \rightarrow AB$ }

S, A, B are reachable but D is not reachable, on eliminating D from both LHS and RHS the rule set R′′ is

R′′ = { $S \rightarrow AB$

$A \rightarrow aAb \mid \varepsilon$

$B \rightarrow bA$ }

5. Proving the Correctness of a Grammar

Given some language L and a grammar G, can we prove that G is correct (ie it generates exactly the strings in L)
To do so, we need to prove two things:

1. Prove that G generates only strings in L.
2. Prove that G generates all the strings in L.

6. Derivations and Parse Trees

Algorithms used for generation and recognition must be systematic. The expansion order is important for algorithms that work with CFG. To make it easier to describe such algorithms, we define two useful families of derivations.

a. A leftmost derivation is one in which, at each step, the leftmost nonterminal in the working string is chosen for expansion.
b. A rightmost derivation is one in which, at each step, the rightmost nonterminal in the working string is chosen for expansion.

Example 1 : S → AB | aaB A → a | Aa B → b
Left-most derivation for string aab is S ⇒ AB ⇒ AaB ⇒ aaB ⇒ aab
Right-most derivation for string aab is S ⇒ AB ⇒ Ab ⇒ Aab ⇒ aab

Example 2: S→iCtS | iCtSeS | x C→y
Left-most Derivation for string iytiytxex is S ⇒ iCtS ⇒ iytS ⇒ iytiCtSeS ⇒ iytiytSeS ⇒ iytiytxe ⇒ iytiytxex
Right-most Derivation for string iytiytxex is S ⇒ iCtSeS ⇒ iCtSex ⇒ iCtiCtSex ⇒ iCtiCtxex ⇒ iCtiytxex ⇒ iytiytxex

Example 3: A Fragment of English Grammar are
S → NP VP
NP → the Nominal | a Nominal | Nominal | ProperNoun | NP PP
Nominal → N | Adjs N
N → cat | dogs | bear | girl | chocolate | rifle
ProperNoun → Chris | Fluffy
Adjs → Adj Adjs | Adj
Adj → young | older | smart
VP → V | V NP | VP PP
V → like | likes | thinks | shots | smells
PP → Prep NP
Prep → with

Left-most Derivation for the string “the smart cat smells chocolate”

S ⇒ NP VP
⇒ the Nominal VP
⇒ the Adjs N VP
⇒ the Adj N VP
⇒ the smart N VP
⇒ the smart cat VP
⇒ the smart cat V NP
⇒ the smart cat smells NP
⇒ the smart cat smells Nominal
⇒ the smart cat smells N
⇒ the smart cat smells chocolate

Right-most Derivation for the string “the smart cat smells chocolate”

S ⇒ NP VP
⇒ NP V NP
⇒ NP V Nominal
⇒ NP V N
⇒ NP V chocolate
⇒ NP smells chocolate
⇒ the Nominal smells chocolate
⇒ the Adjs N smells chocolate
⇒ the Adjs cat smells chocolate
⇒ the Adj cat smells chocolate
⇒ the smart cat smells chocolate

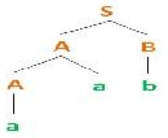
Parse Trees

Regular grammar: in most applications, we just want to describe the set of strings in a language. Context-free grammar: we also want to assign meanings to the strings in a language, for which we care about internal structure of the strings. Parse trees capture the essential grammatical structure of a string. A program that produces such trees is called a parser. A parse tree is an (ordered, rooted) tree that represents the syntactic structure of a string according to some formal grammar. In a parse tree, the interior nodes are labeled by non terminals of the grammar, while the leaf nodes are labeled by terminals of the grammar or ε.

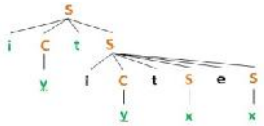
A parse tree, derived by a grammar G = (V, S, R, S), is a rooted, ordered tree in which:

1. Every leaf node is labeled with an element of $\sum \cup \{ \varepsilon \}$,
2. The root node is labeled S,
3. Every other node is labeled with some element of: $V - \sum$, and
4. If m is a nonleaf node labeled X and the children of m are labeled $x_1, x_2, \ldots, x_n$, then R contains the rule $X \rightarrow x_1, x_2, \ldots, x_n$.

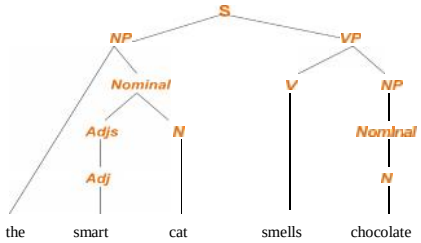
Example 1: S → AB | aaB A → a | Aa B → b
Left-most derivation for the string aab is S ⇒ AB ⇒ AaB ⇒ aaB ⇒ aab
Parse tree obtained is



Example 2: S→iCtS | iCtSeS | x C→y
Left-most Derivation for string iytiytxex isS ⇒ iCtS ⇒ iytS ⇒ iytiCtSeS ⇒ iytiytSeS ⇒ iytiytxeS ⇒ iytiytxex



Example 3: Parse Tree -Structure in English for the string "the smart cat smells chocolate". It is clear from the tree that the sentence is not about cat smells or smart cat smells.



A parse tree may correspond to multiple derivations. From the parse tree, we cannot tell which of the following is used in derivation:

S ⇒ NP VP ⇒ the Nominal VP ⇒
S ⇒ NP VP ⇒ NP V NP ⇒

Parse trees capture the important structural facts about a derivation but throw away the details of the nonterminal expansion order. The order has no bearing on the structure we wish to assign to a string.

Generative Capacity

Because parse trees matter, it makes sense, given a grammar G, to distinguish between:

1. G's weak generative capacity, defined to be the set of strings, L(G), that G generates, and
2. G's strong generative capacity, defined to be the set of parse trees that G generates.

When we design grammar, it will be important that we consider both their weak and their strong generative capacities.

7. Ambiguity

Sometimes a grammar may produce more than one parse tree for some (or all) of the strings it generates. When this happens we say that the grammar is ambiguous. A grammar is ambiguous iff there is at least one string in L(G) for which G produces more than one parse tree.

Example 1: Bal={w ∈ {),(}*: the parenthesis are balanced}.

G={{S,),(}, {),(},R,S} where R={ S → ε S → SS S → (S) }

Left-most Derivation1 for the string (())() is S ⇒ S⇒(S)S ⇒ ((S))S ⇒ (())S ⇒ (())(S) ⇒ (())()

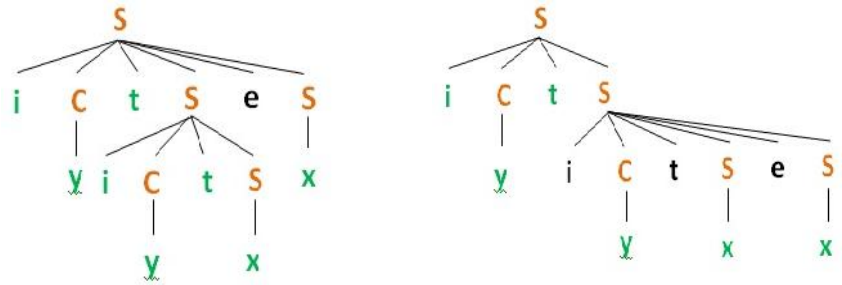
Left-most Derivation2 for the string (())() is S ⇒ SS ⇒SSS ⇒SS ⇒ (S)S ⇒ ((S))S ⇒ (())S ⇒ (())(S) ⇒ (())()

Since both the parse trees obtained for the same string (())() are different, the grammar is ambiguous.

Example 2: S→ iCtS | iCtSeS | x C→ y

Left-most Derivation for the string iytiytxex is S ⇒ iCtS ⇒ iytS ⇒ iytiCtSeS ⇒ iytiytSeS ⇒ iytiytxeS ⇒ iytiytxex

Right-most Derivation for the string iytiytxex is S ⇒ iCtSeS ⇒ iCtSex ⇒ iCtiCtSex ⇒iCtiCtxex ⇒ iCtiytxex ⇒ iytiytxex

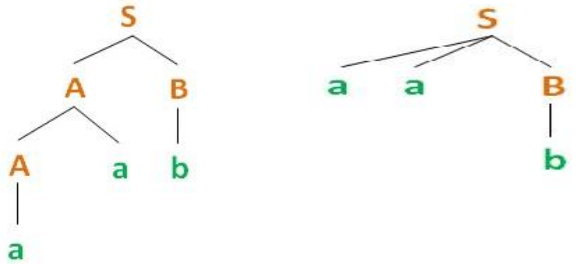


Since both the parse trees obtained for the same string iytiytxex are different, the grammar is ambiguous.

Example 3: S → AB | aaB A → a | Aa B → b

Left-most derivation for string aab is S ⇒ AB ⇒ AaB ⇒ aaB ⇒ aab

Right-most derivation for string aab is S ⇒ aaB ⇒ aab



Since both the parse trees obtained for the same string aab are different, the grammar is ambiguous.

Why Is Ambiguity a Problem?

With regular languages, for most applications, we do not care about assigning internal structure to strings.

With context-free languages, we usually do care about internal structure because, given a string w, we want to assign meaning to w. It is generally difficult, if not impossible, to assign a unique meaning without a unique parse tree. So an ambiguous G, which fails to produce a unique parse tree is a problem.

Example : Arithmetic Expressions

G = (V, Σ, R, E), where

V = {+, *, (,), id, E},

Σ = {+, *, (,), id},

R = {E → E + E, E → E ∗ E, E → (E), E → id }

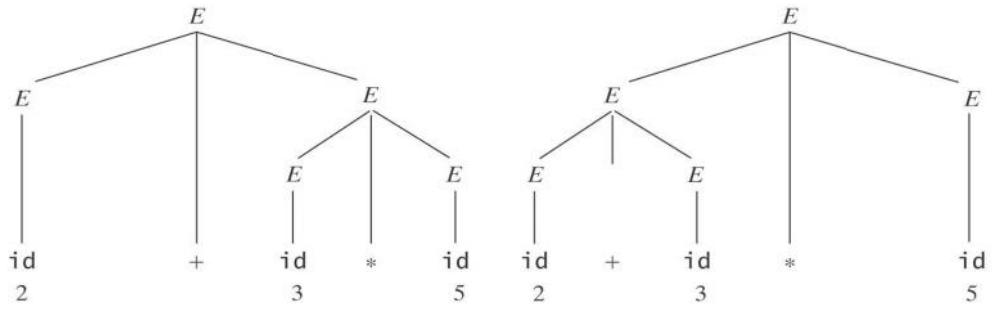
Consider string 2+3*5 written as id +id*id, left-most derivation for string id +id*id is

E ⇒ E*E ⇒ E+E*E ⇒ id+E*E ⇒ id+id*E ⇒ id+id*id.

Similarly the right-most derivation for string id +id*id is

E ⇒ E+E ⇒ E+E*E ⇒ E+E*id ⇒ E+id*id ⇒ id+id*id.

The parse trees obtained for both the derivations are:-



Should the evaluation of this expression return 17 or 25? Designers of practical languages must be careful that they create languages for which they can write unambiguous grammars.

Techniques for Reducing Ambiguity

No general purpose algorithm exists to test for ambiguity in a grammar or to remove it when it is found. But we can reduce ambiguity by eliminating

a. ε rules like S → ε
b. Rules with symmetric right-hand sides
    - A grammar is ambiguous if it is both left and right recursive.
    - Fix: remove right recursion
    - S → SS or E → E + E
c. Rule sets that lead to ambiguous attachment of optional postfixes.

a. Eliminating ε-Rules

Let G =(V, Σ, R, S) be a CFG. The following algorithm constructs a G′ such that L(G′) = L(G)-{ε} and G′ contains no ε rules:

removeEps(G: CFG) =

1. Let G′ = G.
2. Find the set N of nullable variables in G′.
3. Repeat until G′ contains no modifiable rules that haven't been processed:
   Given the rule $P \rightarrow \alpha Q \beta$, where $Q \in N$, add the rule $P \rightarrow \alpha\beta$ if it is not already present and if $\alpha\beta \neq \varepsilon$ and if $P \neq \alpha\beta$.
4. Delete from G′ all rules of the form $X \rightarrow \varepsilon$.
5. Return G′.

Nullable Variables & Modifiable Rules

A variable X is nullable iff either:

(1) there is a rule $X \rightarrow \varepsilon$, or

(2) there is a rule $X \rightarrow PQR\ldots$ and P, Q, R, … are all nullable.

So compute N, the set of nullable variables, as follows:

2.1. Set N to the set of variables that satisfy (1).

2.2. Until an entire pass is made without adding anything to N do

Evaluate all other variables with respect to (2).

If any variable satisfies (2) and is not in N, insert it.

A rule is modifiable iff it is of the form: $P \rightarrow \alpha Q \beta$, for some nullable Q.

Example: G = {{S, T, A, B, C, a, b, c}, {a, b, c}, R, S),

R = {$S \rightarrow aTa$ $T \rightarrow ABC$ $A \rightarrow aA \mid C$ $B \rightarrow Bb \mid C$ $C \rightarrow c \mid \varepsilon$ }

Applying removeEps

Step2: N = { C }

Step2.2 pass1: N = { A, B, C }

Step2.2 pass2: N = { A, B, C, T }

Step2.2 pass3: no new element found.

Step2: halts.

Step3: adds the following new rules to G′.

{ $S \rightarrow aa$

$T \rightarrow AB \mid BC \mid AC \mid A \mid B \mid C$

$A \rightarrow a$

$B \rightarrow b$ }

The rules obtained after eliminating ε-rules :

{ $S \rightarrow aTa \mid aa$

$T \rightarrow ABC \mid AB \mid BC \mid AC \mid A \mid B \mid C$

$A \rightarrow aA \mid C \mid a$

$B \rightarrow Bb \mid C \mid b$

$C \rightarrow c$ }

What If £ Č L?

Sometimes L(G) contains ε and it is important to retain it. To handle this case the algorithm used is

atmostoneEps(G: CFG) =

1. G′′ = removeEps(G).
2. If S_G is nullable then /* i. e., ε ∈ L(G)
   2.1 Create in G′′ a new start symbol S*.
   2.2 Add to $R_{G''}$ the two rules:S* → ε and S* → S_G.
3. Return G′′.

Example: Bal={w ∈ {),(}*: the parenthesis are balanced}.

R={ S → SS
S → (S)
S → ε }

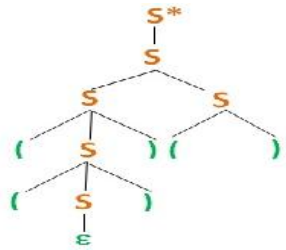
⟹

R={ S → SS
S → (S)
S → ( ) }

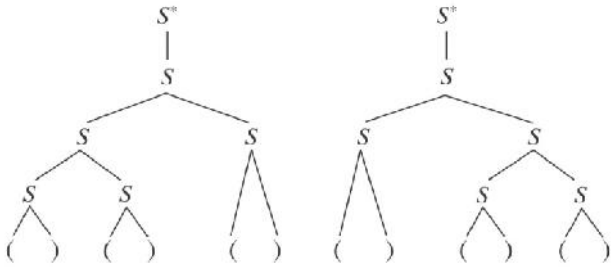
⟹

R={ S* → ε
S* → S
S → SS
S → (S)
S → ( )}

The new grammar built is better than the original one. The string (())() has only one parse tree.



But it is still ambiguous as the string ()()() has two parse trees?



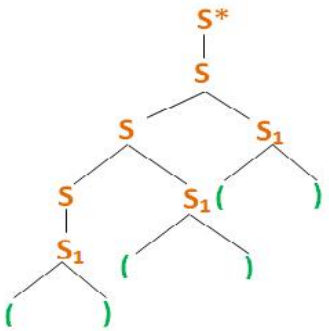
Replace S → SS with one of:
S → S S_1 /* force branching to the left
S → S_1S /* force branching to the right

So we get: S* → ε | S
S → SS_1 /* force branching only to the left
S → S_1 /* add rule
S_1 → (S) | ()

Unambiguous Grammar for Bal={w ∈ {),(}*: the parenthesis are balanced}.
G={{S,),(}, {),(},R,S} where

$$S^* \rightarrow \varepsilon_n \mid S$$
$$S \rightarrow SS_1 \mid S_1$$
$$S_1 \rightarrow (S) \mid ()$$

The parse tree obtained for the string ()()() is



Unambiguous Arithmetic Expressions
Grammar is ambiguous in two ways:

a. It fails to specify associatively.

Ex: there are two parses for the string id + id + id, corresponding to the bracketing (id + id) + id and id + (id + id)

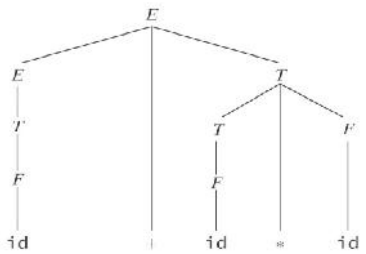
b. It fails to define a precedence hierarchy for the operations + and *.

Ex: there are two parses for the string id + id * id, corresponding to the bracketing (id + id)* id and id + (id * id)

The unambiguous grammar for the arithmetic expression is:

$$E \rightarrow E + T$$
$$E \rightarrow T$$
$$T \rightarrow T * F$$
$$T \rightarrow F$$
$$F \rightarrow (E)$$
$$F \rightarrow id$$

For identical operators: forced branching to go in a single direction (to the left). For precedence Hierarchy: added the levels T (for term) and F (for factor)
The single parse tree obtained from the unambiguous grammar for the arithmetic expression is:



Proving that the grammar is Unambiguous

A grammar is unambiguous iff for all strings w, at every point in a leftmost derivation or rightmost derivation of w, only one rule in G can be applied.

$S^* \rightarrow \varepsilon$ ---(1)

$S^* \rightarrow S$ ---(2)

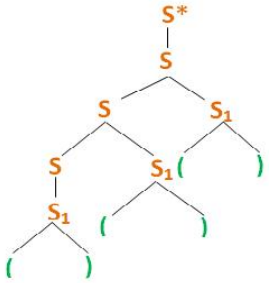
$S \rightarrow SS_1$ ---(3)

$S \rightarrow S_1$ ---(4)

$S_1 \rightarrow (S)$ ---(5)

$S_1 \rightarrow ()$ ---(6)

$S^* \Rightarrow S \Rightarrow SS_1 \Rightarrow SS_1S_1 \Rightarrow S_1S_1S_1 \Rightarrow () S_1S_1 \Rightarrow () () S_1 \Rightarrow () () ()$



Inherent Ambiguity

In many cases, for an ambiguous grammar G, it is possible to construct a new grammar G′ that generate L(G) with less or no ambiguity. However, not always. Some languages have the property that every grammar for them is ambiguous. We call such languages inherently ambiguous.

Example: $L = \{a^i b^j c^k : i, j, k \geq 0, i=j \text{ or } j=k\}$.

Every string in L has either (or both) the same number of a's and b's or the same number of b's and c's. $L = \{a^n b^n c^m : n, m \geq 0\} \cup \{a^n b^m c^m : n, m \geq 0\}$.

One grammar for L has the rules:

$S \rightarrow S_1 \mid S_2$

$S_1 \rightarrow S_1c \mid A$ /* Generate all strings in $\{a^n b^n c^m\}$.

$A \rightarrow aAb \mid \varepsilon$

$S_2 \rightarrow aS_2 \mid B$ /* Generate all strings in $\{a^n b^m c^m\}$.

$B \rightarrow bBc \mid \varepsilon$

Consider the string $a^2b^2c^2$.

It has two distinct derivations, one through S_1 and the other through S_2

$S \Rightarrow S_1 \Rightarrow S_1c \Rightarrow S_1cc \Rightarrow Acc \Rightarrow aAbcc \Rightarrow aaAbbcc \Rightarrow aabbcc$

$S \Rightarrow S_2 \Rightarrow aS_2 \Rightarrow aaS_2 \Rightarrow aaB \Rightarrow aabBc \Rightarrow aabbBcc \Rightarrow aabbcc$

Given any grammar G that generates L, there is at least one string with two derivations in G.

Both of the following problems are undecidable:

- Given a context-free grammar G, is G ambiguous?
- Given a context-free language L, is L inherently ambiguous

8. Normal Forms

We have seen in some grammar where RHS is ε , it makes grammar harder to use. Lets see what happens if we carry the idea of getting rid of ε -productions a few steps farther. To make our tasks easier we define normal forms.

Normal Forms - When the grammar rules in G satisfy certain restrictions, then G is said to be in Normal Form.

- Normal Forms for queries & data can simplify database processing.
- Normal Forms for logical formulas can simplify automated reasoning in AI systems and in program verification system.
- It might be easier to build a parser if we could make some assumptions about the form of the grammar rules that the parser will use.

Normal Forms for Grammar

Among several normal forms, two of them are:-

- Chomsky Normal Form(CNF)
- Greibach Normal Form(GNF)

Chomsky Normal Form (CNF)

In CNF we have restrictions on the length of RHS and the nature of symbols on the RHS of the grammar rules.

A context-free grammar G = (V, Σ, R, S) is said to be in Chomsky Normal Form (CNF), iff every rule in R is of one of the following forms:

$X \rightarrow a$ where $a \in \Sigma$, or

$X \rightarrow BC$ where B and $C \in V\text{-}\Sigma$

Example: $S \rightarrow AB, \quad A \rightarrow a, B \rightarrow b$

Every parse tree that is generated by a grammar in CNF has a branching factor of exactly 2 except at the branches that leads to the terminal nodes, where the branching factor is 1.

Using this property parser can exploit efficient data structure for storing and manipulating binary trees. Define straight forward decision procedure to determine whether w can be generated by a CNF grammar G. Easier to define other algorithms that manipulates grammars.

Greibach Normal Form (GNF)

GNF is a context free grammar G = (V, Σ, R, S), where all rules have one of the following forms: $X \rightarrow a\beta$ where $a \in \Sigma$ and $\beta \in (V\text{-}\Sigma)^*$

Example: $S \rightarrow aA \mid aAB, A \rightarrow a, B \rightarrow b$

In every derivation precisely one terminal is generated for each rule application. This property is useful to define a straight forward decision procedure to determine whether w can be generated by GNF grammar G. GNF grammars can be easily converted to PDA with no ε transitions.

Converting to Chomsky Normal Form

Apply some transformation to G such that the language generated by G is unchanged.

1. Rule Substitution.

   Example: X→ aYc Y→ b Y → ZZ equivalent grammar constructed is X→ abc | aZZc

There exists 4-steps algorithm to convert a CFG G into a new grammar G_c such that: L(G) = $L(G_c) - \{\varepsilon\}$

convertChomsky(G:CFG) =

1. G' = removeEps(G:CFG) $S \to \varepsilon$
2. G" = removeUnits(G':CFG) $A \to B$
3. G"' = removeMixed(G":CFG) $A \to aB$
4. G'^{v} = removeLong(G"' :CFG) $S \to ABCD$

return G_c

Remove Epsilon using removeEps(G:CFG)

Find the set N of nullable variables in G.

X is nullable iff either $X \to \varepsilon$ or $(X \to A, A \to \varepsilon) : X \to \varepsilon$

Example1: G: $S \to aACa$

$A \to B \mid a$

$B \to C \mid c$

$C \to cC \mid \varepsilon$

Now, since $C \to \varepsilon$, C is nullable

since $B \to C$, B is nullable

since $A \to B$, A is nullable

Therefore N = { A,B,C}

removeEps returns G':

$S \to aACa \mid aAa \mid aCa \mid aa$

$A \to B \mid a$

$B \to C \mid c$

$C \to cC \mid c$

Remove Unit Productions using removeUnits(G:CFG)

Unit production is a rule whose right hand side consists of a single nonterminal symbol.

Ex: $A \to B$. Remove all unit production from G'.

Consider the remaining rules of G'.

G′: $S \to aACa \mid aAa \mid aCa \mid aa$

$A \to B \mid a$

$B \to C \mid c$

$C \to cC \mid c$

Remove $A \to B$ But $B \to C \mid c$, so Add $A \to C \mid c$

Remove $B \to C$ Add $B \to cC$ ($B \to c$, already there)

Remove $A \to C$ Add $A \to cC$ ($A \to c$, already there)

removeUnits returns G" :

$S \rightarrow aACa \mid aAa \mid aCa \mid aa$

$A \rightarrow cC \mid a \mid c$

$B \rightarrow cC \mid c$

$C \rightarrow cC \mid c$

Remove Mixed using removeMixed(G":CFG)

Mixed is a rule whose right hand side consists of combination of terminals or terminals with nonterminal symbol. Create a new nonterminal T_a for each terminal $a \in \Sigma$. For each T_a, add the rule $T_a \rightarrow a$

Consider the remaining rules of G" :

$S \rightarrow aACa \mid aAa \mid aCa \mid aa$

$A \rightarrow cC \mid a \mid c$

$B \rightarrow cC \mid c$

$C \rightarrow cC \mid c$

removeMixed returns G"' :

$S \rightarrow T_aACT_a \mid T_aAT_a \mid T_aCT_a \mid T_aT_a$

$A \rightarrow T_cC \mid a \mid c$

$B \rightarrow T_cC \mid c$

$C \rightarrow T_cC \mid c$

$T_a \rightarrow a$

$T_c \rightarrow c$

Remove Long using removeLong(G"' :CFG)

Long is a rule whose right hand side consists of more than two nonterminal symbol.

R: $A \rightarrow BCDE$ is replaced as: $A \rightarrow BM_2$

$M_2 \rightarrow CM_3$

$M_3 \rightarrow DE$

Consider the remaining rules of G"':

$S \rightarrow T_aACT_a \mid T_aAT_a \mid T_aCT_a$

Now, by applying removeLong we get :

$S \rightarrow T_aS_1$

$S_1 \rightarrow AS_2$

$S_2 \rightarrow CT_a$

$S \rightarrow T_aS_3$

$S_3 \rightarrow AT_a$

$S \rightarrow T_aS_2$

Now, by apply removeLong returns G^{iv} :

$S \rightarrow T_aS_1 \mid T_aS_3 \mid T_aS_2 \mid T_aT_a$

$S_1 \rightarrow AS_2$

$S_2 \rightarrow CT_a$

$S_3 \rightarrow AT_a$

$A \rightarrow T_cC \mid a \mid c$

$B \rightarrow T_cC \mid c$

$C \rightarrow T_cC \mid c$

$T_a \rightarrow a$

$T_c \rightarrow c$

Example 2: Apply the normalization algorithm to convert the grammar to CNF

G: $S \rightarrow aSa \mid B$

$B \rightarrow bbC \mid bb$

$C \rightarrow cC \mid \varepsilon$

removeEps(G:CFG) returns

G′: $S \rightarrow aSa \mid B$

$B \rightarrow bbC \mid bb$

$C \rightarrow cC \mid c$

removeUnits(G':CFG) returns

G′′ : $S \rightarrow aSa \mid bbC \mid bb$

$B \rightarrow bbC \mid bb$

$C \rightarrow cC \mid c$

removeMixed(G":CFG) returns

G''': $S \rightarrow T_aST_a \mid T_bT_bC \mid T_bT_b$

$B \rightarrow T_bT_bC \mid T_bT_b$

$C \rightarrow T_cC \mid c$

$T_a \rightarrow a$

$T_b \rightarrow b$

$T_c \rightarrow c$

removeLong(G''' :CFG) returns

G^{iv}: $S \rightarrow T_a S_1 \mid T_b S_2 \mid T_bT_b$

$S_1 \rightarrow S\ T_a$

$S_2 \rightarrow T_b\ C$

$B \rightarrow T_b S_2 \mid\ T_bT_b$

$C \rightarrow\ T_cC \mid c$

$T_a \rightarrow a$

$T_b \rightarrow b$

$T_c \rightarrow c$

Example 3: Apply the normalization algorithm to convert the grammar to CNF

G: $S \rightarrow ABC$
$A \rightarrow aC \mid D$
$B \rightarrow bB \mid \varepsilon \mid A$
$C \rightarrow Ac \mid \varepsilon \mid Cc$
$D \rightarrow aa$

removeEps(G:CFG) returns

G′: $S \rightarrow ABC \mid AC \mid AB \mid A$
$A \rightarrow aC \mid D \mid a$
$B \rightarrow bB \mid A \mid b$
$C \rightarrow Ac \mid Cc \mid c$
$D \rightarrow aa$

removeUnits(G':CFG) returns

G′′ : $S \rightarrow ABC \mid AC \mid AB \mid aC \mid aa \mid a$
$A \rightarrow aC \mid aa \mid a$
$B \rightarrow bB \mid aC \mid aa \mid a \mid b$
$C \rightarrow Ac \mid Cc \mid c$
$D \rightarrow aa$

removeMixed(G":CFG) returns

G′′′ : $S \rightarrow ABC \mid AC \mid AB \mid T_a C \mid T_a T_a \mid a$
$A \rightarrow T_a C \mid T_a T_a \mid a$
$B \rightarrow T_b B \mid T_a C \mid T_a T_a \mid a \mid b$
$C \rightarrow A T_c \mid C T_c \mid c$
$D \rightarrow T_a T_a$
$T_a \rightarrow a$
$T_b \rightarrow b$
$T_c \rightarrow c$

removeLong(G''' :CFG) returns

G′^{v}: $S \rightarrow AS_1 \mid AC \mid AB \mid T_a C \mid T_a T_a \mid a$
$S_1 \rightarrow BC$
$A \rightarrow T_a C \mid T_a T_a \mid a$
$B \rightarrow T_b B \mid T_a C \mid T_a T_a \mid a \mid b$
$C \rightarrow A T_c \mid C T_c \mid c$
$D \rightarrow T_a T_a$
$T_a \rightarrow a$
$T_b \rightarrow b$
$T_c \rightarrow c$

9. Pushdown Automata

An acceptor for every context-free language. A pushdown automata , or PDA, is a finite state machine that has been augmented by a single stack.

Definition of a (NPDA) Pushdown Automaton

M = (K, S, G, Δ , s, A), where:

K is a finite set of states,
S is the input alphabet,
G is the stack alphabet,
s ∈ K is the initial state,
A ⊆ K is the set of accepting states, and
Δ is the transition relation.

Δ is the transition relation. It is a finite subset of

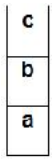
$$(K \times (\Sigma \cup \{\varepsilon\}) \times \Gamma^*) \times (K \times \Gamma^*)$$

state | input or ε | string of symbols to pop from top of stack | state | string of symbols to push on top of stack

Configuration

A configuration of PDA M is an element of K X S* X G*. Current state, Input that is still left to read and, Contents of its stack.

The initial configuration of a PDA M, on input w, is (s, w, ε).



will be written as cba

If $s_1s_2...s_n$ is pushed onto the stack: the value after the push is $s_1s_2...s_ncba$

Yields-in-one-step

Yields-in-one-step written $|-_M$ relates configuration$_1$ to configuration$_2$ iff M can move from configuration$_1$ to configuration$_2$ in one step. Let c be any element of ∑ U {ε }, let $\gamma 1$, $\gamma 2$ and γ be any elements of G*, and let w be any element of S*. Then:

$$(q_1, cw, \gamma 1\gamma) |-_M (q_2, w, \gamma 2\gamma) \text{ iff } ((q_1, c, \gamma 1), (q_2, \gamma 2)) \in \Delta .$$

The relation yields, written $|-_M^*$ is the reflexive, transitive closure of $|-_M$ C_1 yields configuration C_2 iff $C_1 |-_M^* C_2$

Computation

A computation by M is a finite sequence of configurations $C_0, C_1, C_2, \ldots, C_n$ for some $n \geq 0$ such that:

- C_0 is an initial configuration,
- C_n is of the form (q, ε, γ), for some $q \in K$ and some string γ in G^*, and
- $C_0 \vdash_M C_1 \vdash_M C_2 \vdash_M \ldots \vdash_M C_n$.

Nondeterminism

If M is in some configuration (q_1, s, γ) it is possible that:

- Δ contains exactly one transition that matches. In that case, M makes the specified move.
- Δ contains more than one transition that matches. In that case, M chooses one of them.
- Δ contains no transition that matches. In that case, the computation that led to that configuration halts.

Accepting

Let C be a computation of M on input w then C is an accepting configuration

iif $C = (s, w, \varepsilon) \vdash_M^* (q, \varepsilon, \varepsilon)$, for some $q \in A$.

A computation accepts only if it runs out of input when it is in an accepting state and the stack is empty.

C is a rejecting configuration iif $C = (s, w, \varepsilon) \vdash_M^* (q, w', \alpha)$,

where C is not an accepting computation and where M has no moves that it can makes from (q, w', α). A computation can reject only if the criteria for accepting have not been met and there are no further moves that can be taken.

Let w be a string that is an element of S^* . Then:

- M accepts w iif atleast one of its computations accepts.
- M rejects w iif all of its computations reject.

The language accepted by M, denoted L(M), is the set of all strings accepted by M. M rejects a string w iff all paths reject it.

It is possible that, on input w, M neither accepts nor rejects. In that case, no path accepts and some path does not reject.

Transition

Transition $((q_1, c, \gamma 1), (q_2, \gamma 2))$ says that "If c matches the input and g_1 matches the current top of the stack, the transition from q_1 to q_2 can be taken. Then, c will be removed from the input, $\gamma 1$ will be popped from the stack, and $\gamma 2$ will be pushed onto it. M cannot peek at the top of the stack without popping

- If $c = \varepsilon$, the transition can be taken without consuming any input.
- If $\gamma 1 = \varepsilon$, the transition can be taken without checking the stack or popping anything. Note: it's not saying "the stack is empty".
- If $\gamma 2 = \varepsilon$, nothing is pushed onto the stack when the transition is taken.

Example1: A PDA for Balanced Parentheses. Bal={w ∈ {),(}*: the parenthesis are balanced}

M = (K, S, G, Δ, s, A),

where:

K = {s} the states

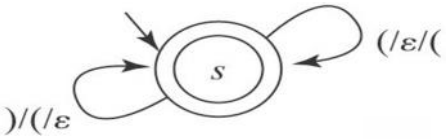
S = {(,)} the input alphabet

Γ = {(} the stack alphabet

A = {s} the accepting state

Δ = { ((s, (,ε), (s, ( )) ----- (1)

((s,), ( ), (s, ε)) ----- (2) }



An Example of Accepting -- Input string = (())()

(S, (())(),ε) |- (S, ())(), ( ) |- (S,))()),(() |- (S,)(), () |- (S, (), ε) |- (S,), () |- (S, ε, ε)

The computation accepts the string (((())() as it runs out of input being in the accepting state S and stack empty.

Transition	State	Unread input	Stack
	S	(())()	
1	S	())()	(
1	S	))()	((
2	S	)()	(
2	S	()	
1	S	)	(
2	S		

Example1 of Rejecting -- Input string = (()))

(S, ())),ε) |- (S, ()))),() |- (S,))),(() |- (S,)),() |- (S,), ε)

Transition	State	Unread input	Stack
	S	(()))	
1	S	()))	(
1	S	)))	((
2	S	))	(
2	S	)	

The computation has reached the final state S and stack is empty, but still the string is rejected because the input is not empty.

Example2 of Rejecting -- Input string = (())

Transition	State	Unread input	Stack
	S	(())	ε
1	S	())	(
1	S	))	((
1	S	)	(((
2	S	)	((
2	S	ε	(

(S, (()),ε) |- (S, ()),(|- (S,( )),(() |- (S,)),((() |- (S,),(() |- (S, ε ,()

The computation has reached the final state S and runs out of input, but still the string is rejected because the stack is not empty.

Example 2: A PDA for $A^nB^n = \{a^nb^n: n \geq 0\}$

M = (K, S, G, Δ, s, A),

where:

K = {s, f} the states

S = {a, b} the input alphabet

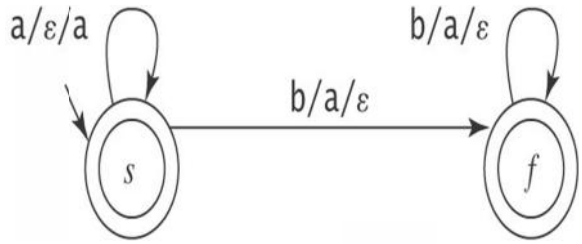
Γ = {a} the stack alphabet

A = {s, f} the accepting state

Δ = { ((s, a, ε), (s, a)) -----(1)

((s, b, a), (f, ε)) -----(2)

((f, b, a), (f, ε)) } -----(3)



An Example of Accepting -- Input string = aabb

(f, aabb,ε) |- (f, abb, a) |- (f, bb, aa) |- (f, b, a) |- (f, ε, ε)

The computation has reached the final state f, the input string is consumed and the stack is empty. Hence the string aabb is accepted.

Example3: A PDA for $\{wcw^R: w \in \{a, b\}^*\}$

M = (K, S, G, Δ, s, A),

where:

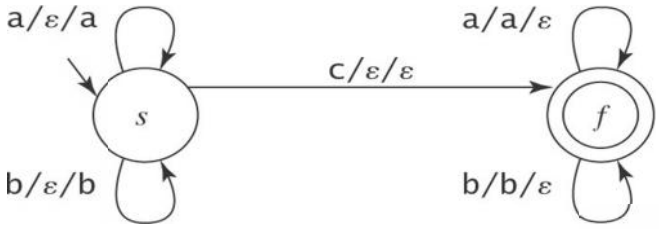
K = {s, f} the states

S = {a, b, c} the input alphabet

Γ = {a, b} the stack alphabet

A = {f} the accepting state

Δ = {((s, a, ε), (s, a) -----(1)
((s, b, ε), (s, b)) -----(2)
((s, c, ε), (f, ε)) -----(3)
((f, a, a), (f, ε)) -----(4)
((f, b, b), (f, ε))} -----(5)

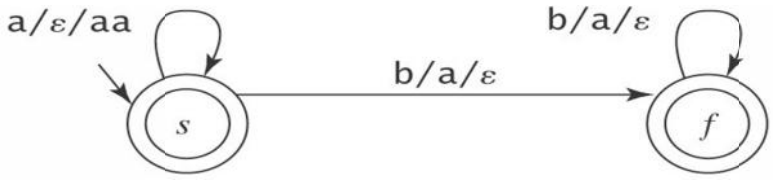


An Example of Accepting -- Input string = abcba
(s, abcba,ε) |- (s, bcba, a) |- (s, cba,ba) |- (f, ba, ba) |- (f, a, a) |- (f, ε, ε)
The computation has reached the final state f, the input string is consumed and the stack is empty. Hence the string abcba is accepted.

Example 4: A PDA for A^nB^{2n} = {a^nb^{2n}: n ≥ 0}
M = (K, S, G, Δ, s, A),
where:

K = {s, f} the states
S = {a, b} the input alphabet
Γ = {a} the stack alphabet
A = {s, f} the accepting state

Δ = { ((s, a, ε), (s, aa)) -----(1)
((s, b, a), (f, ε)) -----(2)
((f, b, a), (f, ε)) } -----(3)



An Example of Accepting -- Input string = aabbbb
(s, aabbbb,ε) |- (s, abbbb, aa) |- (s, bbbb,aaaa) |- (f, bbb, aaa) |- (f, bb, aa) |-(f, b, a) |- (f, ε, ε)

10. Deterministic and Nondeterministic PDAs

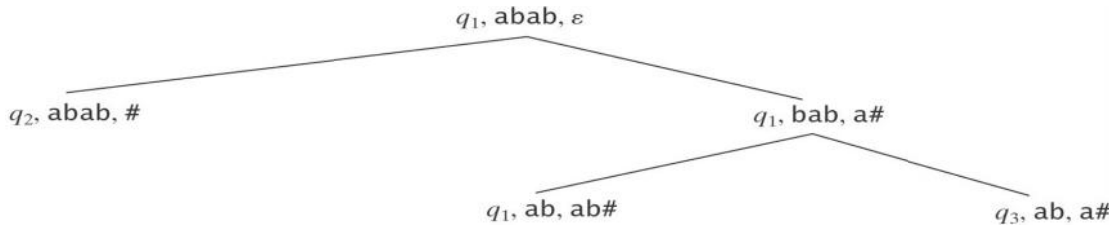
A PDA M is deterministic iff:

- Δ_M contains no pairs of transitions that compete with each other, and
- whenever M is in an accepting configuration it has no available moves.
- If q is an accepting state of M, then there is no transition ((q, e , e) ,(p, a)) for any p or a.

Unfortunately, unlike FSMs, there exist NDPDA s for which no equivalent DPDA exists.

Exploiting Nondeterministic

Previous examples are DPDA, where each machine followed only a single computational path. But many useful PDAs are not deterministic, where from a single configuration there exist multiple competing moves. As in FSMs, easiest way to envision the operation of a NDPDA M is as a tree.



Each node in the tree corresponds to a configuration of M and each path from the root to a leaf node may correspond to one computation that M might perform. The state, the stack and the remaining input can be different along different paths. As a result, it will not be possible to simulate all paths in parallel, the way we did for NDFSMs.

Example 1: PDA for PalEven ={ww^R: w ∈ {a, b}*}.

The L of even length palindrome of a's and b's. = {ε, aa, bb, aaaa, abba, baab, bbbb, ..…….}

M = (K, S, G, Δ, s, A),

where:

K = {s, f} the states

S = {a, b} the input alphabet

Γ = {a, b} the stack alphabet

A = {f} the accepting state

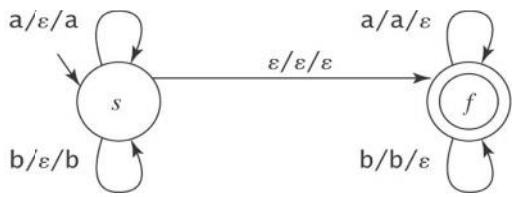
Δ = {((s, a, ε), (s, a)) -----(1)

((s, b, ε), (s, b)) -----(2)

((s, ε, ε), (f, ε)) -----(3)

((f, a, a), (f, ε)) -----(4)

((f, b, b), (f, ε))} -----(5)



Example 2: PDA for $\{w \in \{a, b\}^* : \#_a(w) = \#_b(w)\}$= Equal Numbers of a's and b's.

L = {ε, ab, ba, abba, aabb, baba, bbaa, …….}

M = (K, S, G, Δ, s, A),

where:

K = {s} the states

S = {a, b} the input alphabet

Γ = {a, b} the stack alphabet

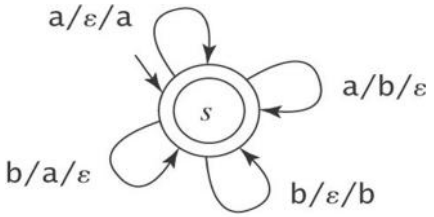
A = {s} the accepting state

Δ ={((s, a, ε), (s, a)) -----(1)

((s, b, ε), (s, b)) -----(2)

((s, a, b), (s, ε)) -----(3)

((s, b, a), (s, ε))} -----(4)

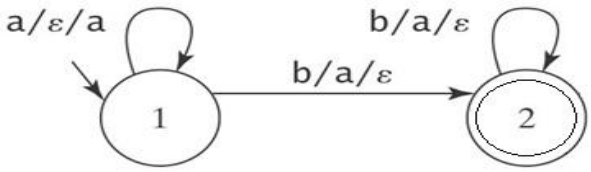


Example 3: The a Region and the b Region are Different. L = $\{a^m b^n : m \neq n; m, n > 0\}$

It is hard to build a machine that looks for something negative, like ≠. But we can break L into two sublanguages: $\{a^m b^n : 0 < n < m\}$ and $\{a^m b^n : 0 < m < n\}$

- If stack and input are empty, halt and reject
- If input is empty but stack is not (m > n) (accept)
- If stack is empty but input is not (m < n) (accept)

Start with the case where n = m

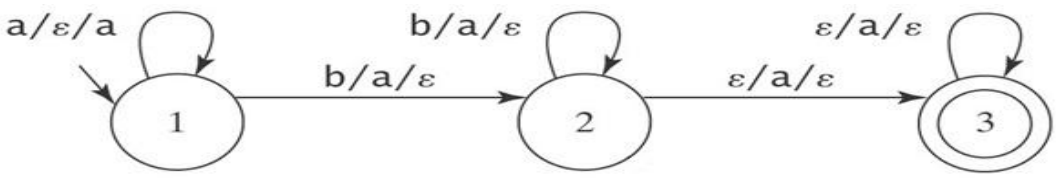


Δ = { ((1, a, ε), (1, a)) -----(1)

((1, b, a), (2, ε)) -----(2)

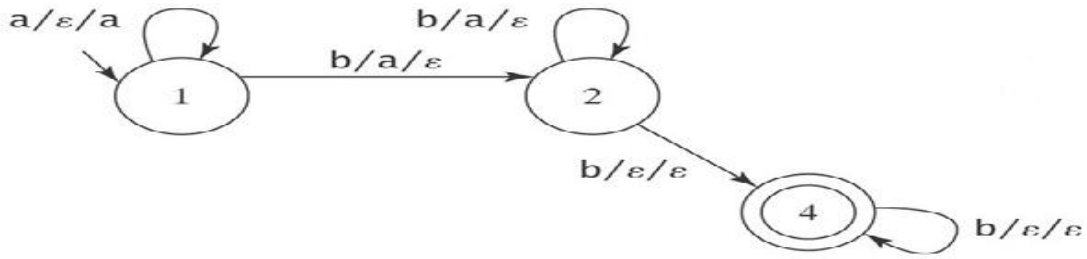
((2, b, a), (2, ε)) } -----(3)

If input is empty but stack is not (m > n) (accept):



Δ = { ((1, a, ε), (1, a)) -----(1)
((1, b, a), (2, ε)) -----(2)
((2, b, a), (2, ε)) -----(3)
((2, ε, a), (3, ε)) -----(4)
((3, ε, a), (3, ε)) } -----(5)

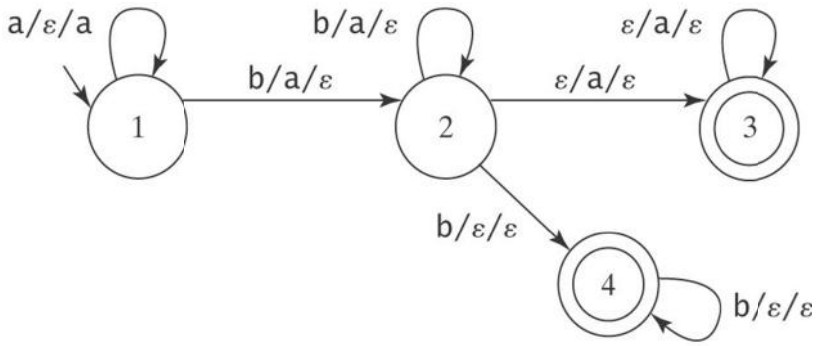
If stack is empty but input is not (m < n) (accept):



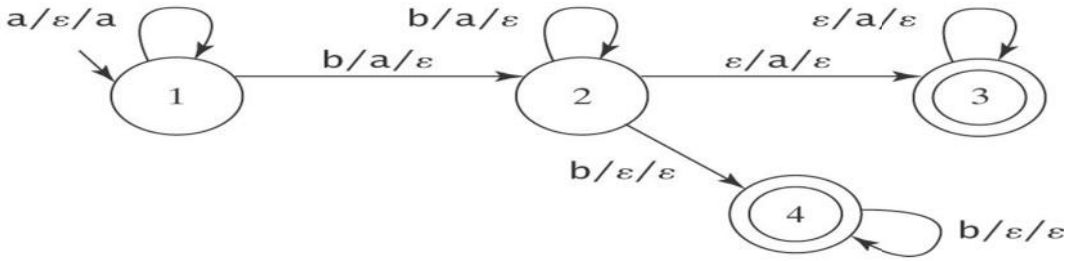
Δ = { ((1, a, ε), (1, a)) -----(1)
((1, b, a), (2, ε)) -----(2)
((2, b, a), (2, ε)) -----(3)
((2, b, ε), (4, ε)) -----(4)
((4, b, ε), (4, ε)) } -----(5)

Putting all together the PDA obtained is
M = (K, S, G, Δ, s, A),
where:

K = {1,2,3,4} the states
S = {a, b} the input alphabet
Γ = {a} the stack alphabet
A = {3,4} the accepting state



Δ = { ((1, a, ε), (1, a)) -----(1)
((1, b, a), (2, ε)) -----(2)
((2, b, a), (2, ε)) -----(3)
((2, ε, a), (3, ε)) -----(4)
((3, ε, a), (3, ε)) } -----(5)
((2, b, ε), (4, ε)) -----(6)
((4, b, ε), (4, ε)) } -----(7)



Two problems with this M:

1. We have no way to specify that a move can be taken only if the stack is empty.
2. We have no way to specify that the input stream is empty.
3. As a result, in most of its moves in state 2, M will have a choice of three paths to take.

Techniques for Reducing Nondeterminism

We saw nondeterminism arising from two very specific circumstances:

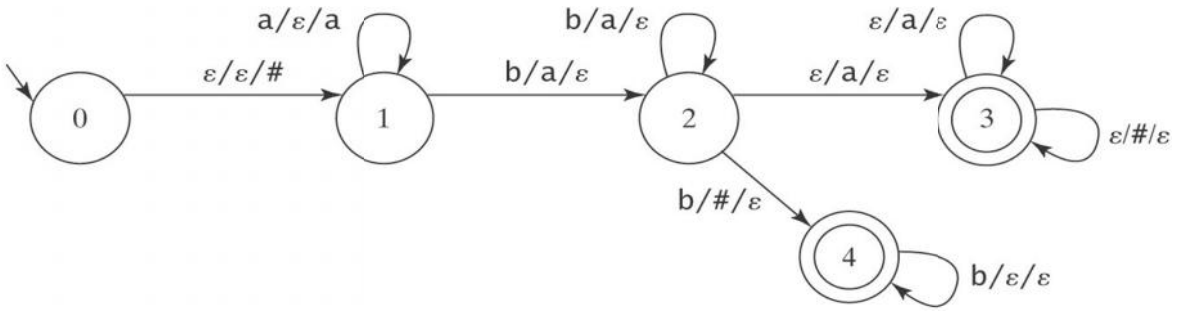
- A transition that should be taken only if the stack is empty competes against one or more moves that require a match of some string on the stack.
- A transition that should be taken only if the input stream is empty competes against one or more moves that require a match against a specific input character.

Case1: A transition that should be taken only if the stack is empty competes against one or more moves that require a match of some string on the stack.

Problem: Nondeterminism could be eliminated if it were possible to check for an empty stack.

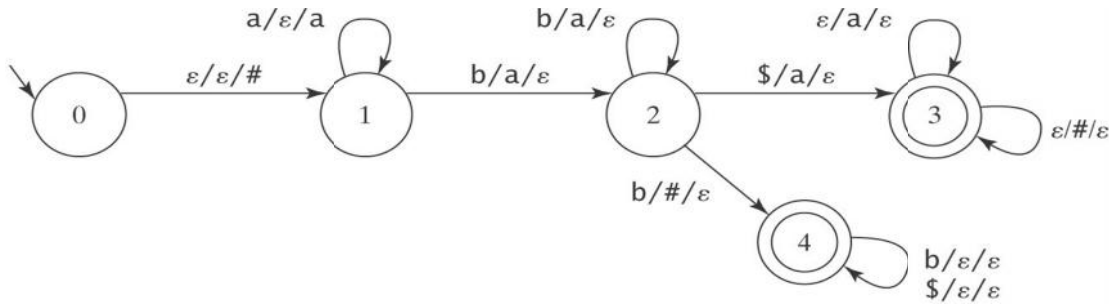
Solution: Using a special bottom-of-stack marker (#)

Before doing anything, push a special character onto the stack. The stack is then logically empty iff that special character (#) is at the top of the stack. Before M accepts a string, its stack must be completely empty, so the special character must be popped whenever M reaches an accepting state.



Now the transition back to state 2 no longer competes with the transition to state 4, which can only be taken when the # is the only symbol on the stack. The machine is still nondeterministic because the transition back to state 2 competes with the transition to state 3.

Case2: A transition that should be taken only if the input stream is empty competes against one or more moves that require a match against a specific input character.
Problem: Nondeterminism could be eliminated if it were possible to check for an empty input stream.
Solution: using a special end-of-string marker ($)
Adding an end-of-string marker to the language to be accepted is a powerful tool for reducing nondeterminism. Instead of building a machine to accept a language L, build one to accept L$, where $ is a special end-of-string marker.



Now the transition back to state 2 no longer competes with the transition to state 3, since the can be taken when the $ is read. The $ must be read on all the paths, not just the one where we need it.

11. Nondeterminism and Halting

Recall Computation C of a PDA M = (K, S, G, Δ, s, A) on a string w is an accepting computation iif C= (s, w, ε) $|-_M^*$ (q, ε, ε), for some q ∈ A.
A computation C of M halts iff at least one of the following condition holds:

- C is an accepting computation, or
- C ends in a configuration from which there is no transition in Δ that can be taken.

M halts on w iff every computation of M on w halts. If M halts on w and does not accept, then we say that M rejects w.
For every CFL L, we've proven that there exists a PDA M such that L(M) = L.
Suppose that we would like to be able to:

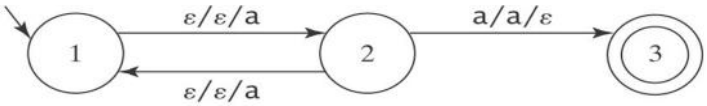
1. Examine a string and decide whether or not it is L.
2. Examine a string that is in L and create a parse tree for it.
3. Examine a string that is in L and create a parse tree for it in time that is linear in the length of the string.
4. Examine a string and decide whether or not it is in the complement of L.

For every regular language L, there exists a minimal deterministic FSM that accepts it. That minimal DFSM halts on all inputs, accepts all strings that are in L, and rejects all strings that are not in L.

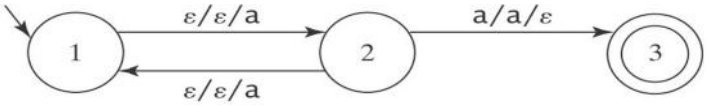
But the facts about CFGs and PDAs are different from the facts about RLs and FSMs.

1. There are context-free languages for which no deterministic PDA exists.
2. It is possible that a PDA may
    - not halt,
    - not ever finish reading its input.

However, for an arbitrary PDA M, there exists M′ that halts and L(M′) = L(M).
There exists no algorithm to minimize a PDA. It is undecidable whether a PDA is minimal.
Problem 2 : Let M be a PDA that accepts some language L. Then, on input w, if w ∈ L then M will halt and accept. But if w L then, while M will not accept w, it is possible that it will not reject it either.
Example1: Let S = {a} and consider M =



For L(M) = {a}. The computation (1, a, e) |- (2, a, a) |- (3, e, e) causes M to accept a.
Example2: Consider M =



For L(M) = {aa} or on any other input except a:
(1, aa, e) |- (2, aa, a) |-(1, aa, aa) |- (2, aa, aaa) |- (1, aa, aaaa) |- (2, aa, aaaaa) |- ……..
M will never halt because of one path never ends and none of the paths accepts.
The same problem with NDFSMs had a choice of two solutions.

- Converting NDFSM to and equivalent DFSM using ndfsmtodfsm algorithm.
- Simulating NDFSM using ndfsmsimulate.

Neither of these approaches work on PDAs. There may not even be an equivalent deterministic PDA.
Solution fall into two classes:

- Formal ones that do not restrict the class of the language that are being considered- converting grammar into normal forms like Chomsky or Greibach normal form.
- Practical ones that work only on a subclass of the CFLs- use grammars in natural forms.

12. Alternative Equivalent Definitions of a PDA

PDA M = (K, S, G, Δ , s, A):

1. Allow M to pop and to push any string in G*.
2. M may pop only a single symbol but it may push any number of them.
3. M may pop and push only a single symbol.

M accepts its input w only if , when it finishes reading w, it is in an accepting state and its stack is empty.

There are two alternatives to this:

1. PDA by Final state: Accept if, when the input has been consumed, M lands in an accepting state, regardless of the contents of the stack.
2. PDA by Empty stack: Accept if, when the input has been consumed, the stack is empty, regardless of the state M is in.

All of these definitions are equivalent in the sense that, if some language L is accepted by a PDA using one definition, it can be accepted by some PDA using each of the other definition.
For example:- If some language L is accepted by a PDA by Final state then it can be accepted by PDA by Final state and empty stack. If some language L is accepted by a PDA by Final state and empty stack then can be accepted by PDA by Final state.
We can prove by showing algorithms that transform a PDA of one sort into and equivalent PDA of the other sort.
Equivalence

1. Given a PDA M that accepts by accepting state and empty stack, construct a new PDA M′ that accepts by accepting state alone, where L(M′) = L(M).
2. Given a PDA M that accepts by accepting state alone, construct a new PDA M′ that accepts by accepting state and empty stack, where L(M′) = L(M).

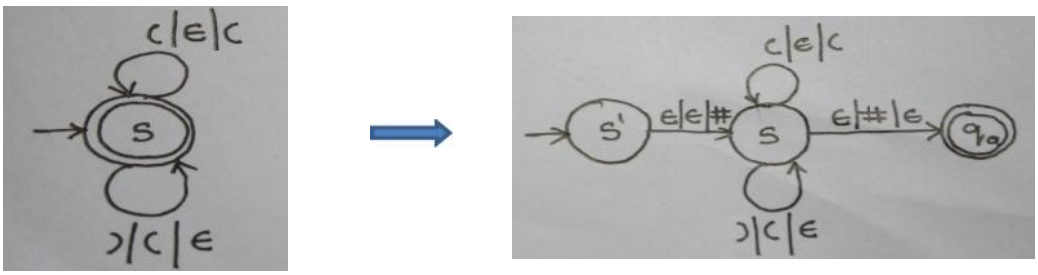
Hence we can prove that M′ and M accept the same strings.

1. Accepting by Final state Alone

Define a PDA M = (K, S, G, Δ , s, A). Accepts when the input has been consumed, M lands in an accepting state, regardless of the contents of the stack. M accepts if C= (s, w, ε) $|-_M^*$ (q, ε, g), for some q ∈ A.
M′ will have a single accepting state q_a. The only way for M′ to get to q_a will be to land in an accepting state of M when the stack is logically empty. Since there is no way to check that the stack is empty, M′ will begin by pushing a bottom-of-stack marker #, onto the stack. Whenever # is the top symbol of the stack, then stack is logically empty.
The construction proceeds as follows:

1. Initially, let M′ = M.
2. Create a new start state s′.
   Add the transition ((s′, ε , ε),(s, #)),
3. For each accepting state a in M do:
   Add the transition ((a, ε ,#),(q_a, ε)),
4. Make q_a the only accepting state in M′

Example:



It is easy to see that M′ lands in its accepting state(q_a) iff M lands in some accepting state with an empty stack. Thus M′ and M accept the same strings.

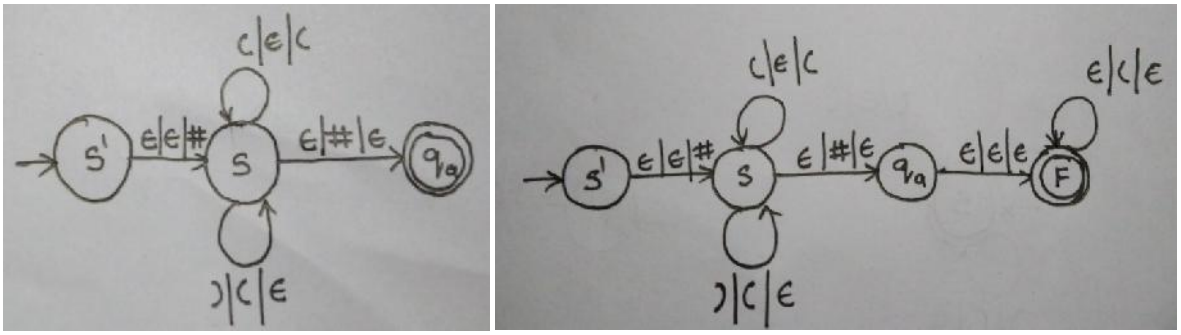
2. Accepting by Final state and Empty stack

The construction proceeds as follows:

1. Initially, let M′ = M.
2. Create a new accepting state F
3. From each accepting state a in M do:
   Add the transition ((a, ε , ε),(F, ε)),
4. Make F the only accepting state in M′
5. For every element g of Γ,
   Add the transition to M′ ((F, ε, g), (F, ε)).

In other words, iff M accepts, go to the only accepting state of M′ and clear the stack. Thus M′ will accept by accepting state and empty stack iff M accepts by accepting state.

Example:-



Thus M′ and M accept the same strings.

13. Alternatives that are not equivalent to the PDA

We defined a PDA to be a finite state machine to which we add a single stack.

Two variants of that definition, each of which turns out to define a more powerful class of a machine.

1. First variant: add a first-in, first-out (FIFO) queue in place of a stack. Such machines are called tag systems or Post machines.
2. Second variant: add two stacks instead of one. The resulting machines are equivalent in computational power to Turing Machines.

Sl.No	Sample Questions
1.	Define context free grammars and languages.
2.	Show a context-free grammar for each of the following languages L: a) BalDelim = {w : where w is a string of delimeters: (,), [,], {, }, that are properly balanced}. b) $\{a^ib^j : 2i = 3j + 1\}$. c) $\{a^ib^j : 2i \neq 3j + 1\}$. d) $\{a^ib^jc^k : i, j, k \geq 0 \text{ and } (i \neq j \text{ or } j \neq k)\}$.
3.	Define CFG. Design CFG for the language $L=\{ a^nb^m : n \neq m\}$
4.	Apply the simplification algorithm to simplify the given grammar S → AB|AC A → aAb | ε B → bA C →bCa D → AB
5.	Prove the correctness of the grammar for the language: $L=\{w \in \{a, b\}^* : \#_a(w) = \#_b(w)\}$.
6.	Define leftmost derivation and rightmost derivation. Given the following CFG. E→ E + T|T T→ T*F|F F→(E)|a|b|c Draw parse tree for the following sentences and also derive the leftmost and rightmost derivations i) (a+b)*c ii) (a) + b*c
7.	Consider the following grammar G: S → 0S1 | SS | 10 Show a parse tree produced by G for each of the following strings: a) 010110 b) 00101101
8.	Define ambiguous and explain inherently ambiguous grammars.
9.	Prove whether the given grammar is ambiguous grammar or not. E→ E + E E→ E*E|a|b|c
10.	Prove that the following CFG is ambiguous S→iCtS|iCtSeS|x C→y for the sting iytiytxex
11.	Define Chomsky normal form. Apply the normalization algorithm to convert the grammar to Chomsky normal form. a. S → aSa S → B B → bbC B → bb C → ε C → cC b. S → ABC A → aC | D B → bB | ε | A C → Ac | ε | Cc D → aa
12.	Define Push down automata (NPDA). Design a NPDA for the CFG given in Question (2).
13.	Design a PDA for the given language.L$, where $L = \{w \in \{a, b\}^* : \#_a(w) = \#_b(w)\}$.
14.	Design a PDA for the language: $L=\{ a^ib^jc^k : i+j=k ,i>=0,j>=0\}$
15.	Design a PDA for the language $L=\{ a^nb^{2n} : n>=1\}$
16.	Design a PDA for the language: $L=\{ a^ib^jc^k : i+k=j ,i>=0,k>=0\}$
17.	Design a PDA for the language: $L=\{ a^ib^jc^k : k+j=i ,k>=0,j>=0\}$

Module-4

- **Context-Free and Non-Context-Free Languages**
- **Where Do the Context-Free Languages Fit in the Big Picture?**
- **Showing that a Language is Context-Free**
- **Pumping theorem for CFL**
- **Important closure properties of CFLs**
- **Deterministic CFLs**
- **Algorithms and Decision Procedures for CFLs: Decidable questions**
- **Undecidable questions**
- **Turing Machine: Turing machine model**
- **Representation**
- **Language acceptability by TM**
- **Design of TM**
- **Techniques for TM construction.**

Context-Free and Non-Context-Free Languages

- The language $A^nB^n = \{a^nb^n \mid n \geq 0\}$ is context-free.
- The language $A^nB^nC^n = \{ a^nb^nc^n \mid n \geq 0\}$ is not context free because a PDA's stack cannot count all three of the letter regions and compare them.

Where Do the Context-Free Languages Fit in the Big Picture?

THEOREM: The Context-Free Languages Properly Contain the Regular Languages.

Theorem: The regular languages are a proper subset of the context-free languages.

Proof: We first show that every regular language is context-free. We then show that there exists at least one context-free language that is not regular.

Every regular language is context-free : We show this by construction.

- If L is regular then it is accepted by some DFSM M = (K, **Σ**, δ, s, A).
- From M we construct a PDA
  M' = (K',**Σ', Γ',Δ',s',A'**) to accept *L*. where Δ' is constructed as follows:
  For every transition (q_i,c,q_j)in δ, add to Δ' the transition $((q_i,c,\varepsilon),(q_i,\varepsilon))$,So *L*(M)=*L(M')*.

So, the set of regular languages is a subset of the CFL.

There exists at least one context-free language that is not regular : The regular languages are a *proper* subset the context-free languages because there exists at least one language $\mathbf{a^n b^n}$ that is context –free but not regular.

Theorem: There is a countably infinite number of context-free languages.

Proof:

Every context-free language is generated by some context-free grammar G = (V,Σ,R,S).

There cannot be more CFLs than CFGs. So there are at most a countably infinite number of context-free languages. There is not a one-to-one relationship between CFLs and CFGs, since there are an infinite number of grammars that generate any given language. But we know that, every regular language is context free and there is a countably infinite number of regular languages.

So there is at least and at most a countably infinite number of CFLs.

Showing That a Language is Context-Free

Two techniques that can be used to show that language L is context-free:

- Exhibit a context-free grammar for it.
- Exhibit a (possibly nondeterministic) PDA for it.

Theorem: The length of the yield of any tree T with height h and branching factor b is <= b^h.

Proof:

If h is 1, then a single rule applies. So the longest yield is of length less than or equal to b.

Assume the claim is true for h=n. We show that it is true for h=n+1.

Consider any tree with h=n+1. It consists of a root, and some number of subtrees, each of height <=n. By the induction hypothesis, the length of the yield of each of those subtrees is <= b^n. So the length of the yield must be <=b.(b^n)=b^{n+1}=b^h.

The Pumping Theorem for Context-Free languages

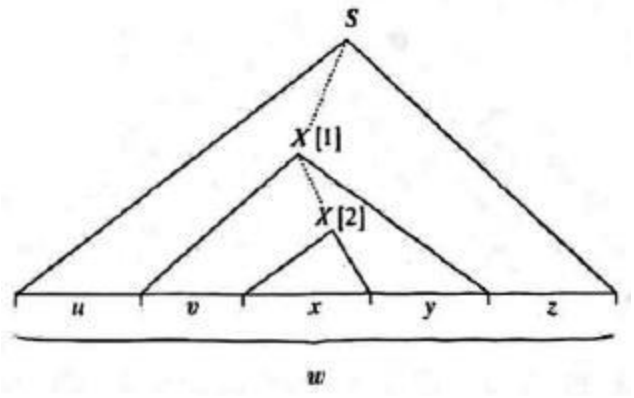
Statement: If L is CFL, then: ∃k≥1 (∀strings *w*∈L, where |w|≥k (∃u,v,x,y,z (*w*=uvxyz, vy≠Ɛ, |vxy| ≤ k and ∀q ≥ 0 (uv^qxy^qz is in L))))

Proof: If L is context-free, then there exists a CFG G=(V,Σ,R,S) with n nonterminal symbols and branching factor b.

Let *k* be b^{n+1}.

Any string that can be generated by G and whose parse tree contains no paths with repeated nonterminals must have length less than or equal to b^n. Assuming that b≥2, it must be the case that $b^{n+1} > b^n$. So let w be any string in L(G) where |w|≥k.

Let T be any smallest parse tree for w. T must have height at least n+1. Choose some path in T of length at least n + 1. Let X be the bottom-most repeated non terminal along that path. Then w can be rewritten as uvxyz as shown in below tree,



$S \Rightarrow^* uXz \Rightarrow^* uvXyz \Rightarrow^* uvxyz.$

The tree rooted at [1] has height at most n+1.Thus its yield, *vxy*, has length less than or equal to b^{n+1},which is k. Further, vy≠Ɛ .Since if *vy* were Ɛ then there would be a smaller parse tree for w and we choose T so that h at wasn't so.

Finally, v and y can be pumped: *uxz* must be in L because rule2 could have been used immediately at[1]. And, for any q≥1, uv^qxy^qz must be in L because rule1 could have been used q times before finally using rule2.

Application of pumping lemma (Proving Language is Not Context Free)

Ex1: Prove that the Language L = {$a^nb^nc^n$| n>=0} is Not Context-Free.

Solution: If L is CFL then there would exist some k such that any string *w*, where |w|>=k must satisfy the conditions of the theorem.

Let w = $a^kb^kc^k$, where 'k' is the constant from the Pumping lemma theorem. For *w* to satisfy the conditions of the Pumping Theorem there must be some u,v,x,y and z, such that w=uvxyz, vy≠Ɛ, |vxy|≤k and ∀q ≥ 0 , uv^qxy^qz is in L.

w=aaa…aaabbb…bbbccc…ccc, select v and y as follows:

w=aaa…a**aabbb**|**…bbbcc**|**c**…ccc
v y

Let q=2, then

w=aaa…a|aabbaabb|b..bbc|cccc|…ccc
V² y²

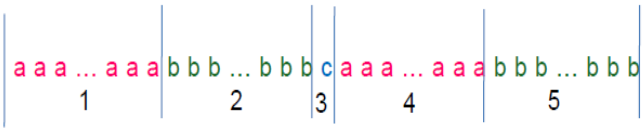
The resulting string will have letters out of order and thus not in L.

So L is not context-free.

Ex 2: Prove that the Language L= {WcW: w∈{a,b}*}is Not Context-Free.

For w to satisfy the conditions of the Pumping Theorem there must be some u,v,x,y,and z, such that w = uvxyz , vy≠Ɛ, |vxy| ≤ k and ∀q≥0, uv^qxy^qz is in L. We show that no such u,v,x,y and z exist.

Imagine w divided into five regions as follows:



Call the **part before the c the leftside and the part after the c the right side.** We consider all the cases for where v and y could fall and show that in none of them are all the condition so f the theorem met:

• If either v or y overlaps region 3, set q to 0. The resulting string will no longer contain a c and so is not in WcW.

• If both v and y occur before region 3 or they both occur after region 3, then set q to 2. One side will be longer than the other and so the resulting string is not in WcW.

• If either v or y overlaps region1 ,then set q to 2. In order to make the right side match. Something would have to be pumped into region 4. But any v,y pair that did that would violate the requirement that lvxyl ≤ k.

• If either v or y overlaps region2, then set q to 2. In order to make the right side match, something would have to be pumped into region 5. But any v,y pair that did that would violate the requirement that lvxyl ≤ k.

• There is no way to divide w into uvxyz such that all the conditions of the Pumping Theorem are met . So WcW is not context-free.

Some Important Closure Properties of Context-Free Languages

Theorem: The context- free languages are closed under Union, Concatenation, Kleene star, Reverse, and Letter substitution.

(1) The context-free languages are closed under union:

• If L1 and L2 are context free languages then there exists a context-free grammar $G_1 = (V_1, \Sigma_1, R_1, S_1)$ and $G_2 = (V_2, \Sigma_2, R_2, S_2)$ such that $L_1 = L(G_1)$ and $L_2 = L(G_2)$.

• We will build a new grammar G such that $L(G)=L(G_1)UL(G_2)$. G will contain all the rules of both G_1 and G_2.

• We add to G a new start symbol S and two new rules. $S\rightarrow S_1$ and $S\rightarrow S_2$. The two new rules allow G to generate a string iff at least one of G_1or G_2 generates it.

So, G = (V_1 U V_2 U {S}, Σ_1 U Σ_2, R_1 U R_2 U {S→ S_1,S→S_2}, S)

(2)The context-free languages are closed under concatenation

• If L_1 and L_2 are context free languages then there exist context-free grammar $G_1= (V_1,\Sigma_1,R_1,S_1)$ and $G_2=(V_2,\Sigma_2,R_2,S_2)$ such that $L_1= L(G_1)$ and $L_2= L(G_2)$.

• We will build a new grammar G such that $L(G) = L(G_1)L(G_2)$.

• G will contain all the rules of both $G_1$and G_2.

• We add to G a new start symbol S and one new rule. $S\rightarrow S_1S_2$

So $G= (V_1UV_2U\{S\}, \Sigma_1 U \Sigma_2, R_1UR_2U\{S\rightarrow S_1S_2\},S)$

(3) The context-free Languages are closed under Kleene star:

• If L_1 is a context free language then there exists a context-free grammar $G_1=(V_1,\Sigma_1,R_1,S_1)$ such that $L_1= L(G_1)$.

• We will build a new grammar G such that $L(G)=L(G_1)^*$ G will contain all the rules of G_1.

• We add to G a new start symbol S and two new rules. $S\rightarrow \varepsilon$ and $S\rightarrow SS_1$

So $G = (V_1 U \{S\}, \Sigma_1, R_1U \{S\rightarrow \varepsilon, S\rightarrow SS_1), S)$

(4) The context-free languages are closed under reverse

• If L is a context free language then it is generated by some Chomsky Normal Form from grammar G= (V,Σ,R, S).

• Every rule in G is of the form X→BC or X→a, where X, B, and C are elements of (V-Σ) and a $\in$ Σ

• So construct, from G, a new grammar G^1 , Such that $L(G^1)= L^R$.

• $G^1= (V_G, \Sigma_G, R', S_G)$, Where R' is constructed as follows:

- For every rule in G of the form X→BC, add to R' the rule X→CB
- For every rule in G of the form X→ a then add to R' the rule X→ a

(5)The context-free languages are closed under letter Substitution

• Consider two alphabets Σ_1 and Σ_2 .

•Let ***sub*** be any function from Σ_1 to Σ_2^*.

•Then ***letsub*** is a letter substitution function from L_1 to L_2 iff **letsub(L_1)** ={ w∈ Σ_2^*: ∃y ∈ L1 (w=y except that every character c of y has replaced by sub(c))}.

Example : Let y= VTU ∈ L_1 And sub(c) is given as : sub(V) = Visvesvaraya

sub(T) = Technological

sub(U)= University

Then , sub(VTU) = Visvesvaraya Technological University

Closure Under Intersection, Complement, and Difference

Theorem:The Context-free language are not closed under intersection, complement or difference.

1) The context-free languages are not closed under intersection

The proof is by counter example. Let: $L_1=\{a^nb^nc^m|n,m\geq 0\}$ $L_2=\{a^mb^nc^n|n,m\geq 0\}$ Both $L_1$and L_2 are context-free since there exist straight forward CFGs for them.

But now consider: $L = L_1 \cap L_2 = \{ a^nb^nc^n \mid n,m\geq 0\}$. If the context-free languages were closure under intersection. L would have to be context-free. But we have proved that L is not CFG by using pumping lemma for CFLs.

(2) The context-free languages are not closure under

Given any sets $L_1$and L_2 , $L_1 \cap L_2 = \neg(\neg L_1 \cup \neg L_2)$

• The context-free languages are closed under union.

• But we just showed that they are not, thus they are not closed under complement either.

• So, if they were also closed under complement, they would necessarily be closed under intersection.

(3) The context-free languages are not closed under difference (subtraction) :

Given any language L and $\neg L=\Sigma^*$- L.

Σ^* is context-free So, if the context-free languages were closed under difference, the complement of any CFL would necessarily be context-free But we just showed that is not so.

Closure Under Intersection With the Regular Languages

Theorem: The context-free languages are closed under intersection with the regular languages.

Proof: The proof is by construction.

• If L_1 is context-free, then there exists some PDA $M_1 = (K_1, \Sigma, \Gamma_1, \Delta_1, S_1, A_1)$ that accepts it.

• If L_2 is regular then there exists a DFSM $M_2 = (K_2, \Sigma, \delta, S_2, A_2)$ that accepts it.

• We construct a new PDA, M_3 that accepts $\mathbf{L_1 \cap L_2}$. M_3 will work by simulating the parallel execution of M_1 and M_2.

• $M_3 = (K_1 X K_2, \Sigma, \Gamma_1, \Delta_3, (S_1, S_2), A_1 X A_2)$, Where Δ_3 is built as follows:

•For each transition $((q_1, a, \beta), (p_1, \gamma))$ in Δ_1 and each transition $((q_2, a), p_2)$ in δ, add Δ_3 the transition: $(((q_1,q_2), a, \beta), ((p_1,p_2), \gamma))$.

•For each transition $((q_1, \varepsilon, \beta), (p_1, \gamma))$ in Δ_1 and each state q_2 in k_2, add to Δ_3 the transition: $(((q_1,q_2), \varepsilon, \beta), ((p_1,p_2), \gamma))$.

Closure Under Difference with the Regular Language.

Theorem: The difference $(L_1 - L_2)$ between a context-free language L_1 and a regular language L_2 is context-free.

Proof: $L_1 - L_2 = L_1 \cap \neg L_2$

• If L_2 is regular, then, since the regular languages are closed under complement, $\neg L_2$ is also regular.

• Since L_1 is context-free, by Theorem we already proved that $L_1 \cap \neg L_2$ is context-free.

Using the Pumping Theorem in Conjunction with the Closure Properties

Languages that impose no specific order constraints on the symbols contained in their strings are not always context-free. But it may be hard to prove that one isn't just by using the Pumping Theorem. In such a case it is proved by considering the fact that the context-free languages are closed under intersection with the regular languages.

Deterministic Context-Free Languages

The technique used to show that the regular languages are closed under complement starts with a given (possibly nondeterministic) FSM M_1, we used the following procedure to construct a new FSM M_2 such that $L(M_2) = \neg L(M_1)$:

The regular languages are closed under complement, intersection and difference. Why are the context-free languages different? Because the machines that accept them may necessarily be nondeterministic.

1. From M_1, construct an equivalent DFSM M', using the algorithm ***ndfsmtodfsm***, presented in the proof of Theorem5.3. (If $M_1$is already deterministic. M'=M_1.)

2. M' must be stated completely. so if it is described with an implied dead state, add the dead state and all required transitions to it.

3. Begin building M_2 by setting it equal to M'. Then swap the accepting and the non-accepting states. So M_2 M'= ($K_{M'}$,.Σ,$\delta_{M'}$,$s_{M'}$,$K_{M'}$-$A_{M'}$).

We have no PDA equivalent of **ndfstodfsm** because there provably isn't one. We defined a PDA M to be deterministic iff:

- Δ_M contains opairs of transitions that compete with each other, and
- if q is an accepting state of M, then there is no transition ((q,ε,ε),(p,a)) for any p or a.

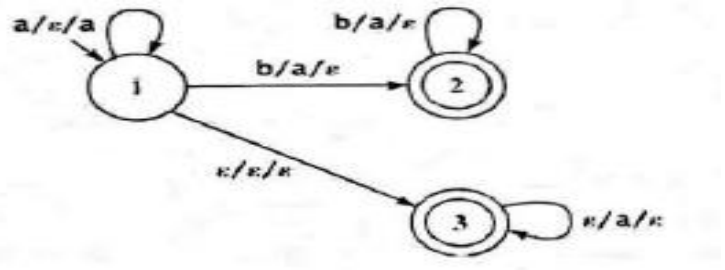
What is a Deterministic Context-Free language?

- Let $ be an end-of-string marker. A language L is deterministic context-free iff L$ can be accepted by some deterministic PDA.

EXAMPLE: Why an End-of-String Marker is Useful

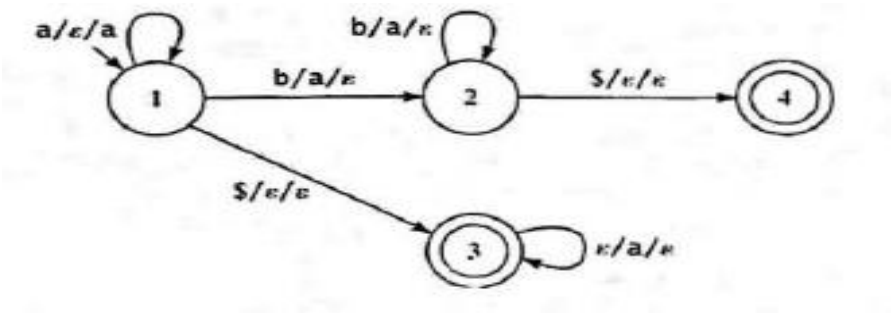
Let L= a* ∪ { $a^n b^n$| n>0}

Consider any PDA M that accepts L. When it begins reading a's, M must push them onto the stack in case there are going to be b's. But if it runs out of input without seeing b's, it needs a way to pop those a's from the stack before it can accept. Without an end-of-string marker, there is no way to allow that popping to happen only when all the input has been read.



For example, the PDA accepts L, but it is nondeterministic because the transition to state3 (where the a's will be popped) can compete with both of the other transitions from state1.

With an end-of-string marker, we can build the deterministic PDA, which can only take the transition to state3, the a-popping state. When it sees the $:



NOTE: Adding the end-of-string marker cannot convert a language that was not context-free into one that is.

CFLs and Deterministic CFLs

Theorem: Every deterministic context-free language is context-free.

Proof:

If L is deterministic context-free, then L$ is accepted by some deterministic PDA M=(K,Σ,Γ,Δ,s,A) . From M, we construct M' such that L (M') = L. We can define the following procedure to construct M':

without$(M:PDA)=

1.Initially. set M' to M.

/*Make the copy that does not read any input.

2.For every state q in M, add to M' a new state q'.

3.For every transition $((q, \varepsilon, \gamma_1),(p,\gamma_2))$ in Δ M do:

3.1. Add to Δ_M the transition $((q',\varepsilon,\gamma_1),(p',\gamma_2))$.

/*Link up the two copies.

4.For every transition $((q,\$,\gamma_1),(p,\gamma_2))$ in Δ M do:

4.1. Add to $\Delta_{M'}$ the transition $((q,\varepsilon,\gamma_1),(p',\gamma_2))$.

4.2. Remove $((q,\$,\gamma_1),(p,\gamma_2))$ from $\Delta_{M'}$

/*Set the accepting state s of M'.

5.$A_{M'} = \{q':q \in A\}$.

Closure Properties of the Deterministic Context-Free Languages

1) Closure Under Complement

Theorem: The deterministic context-free languages are closed under complement.

Proof: The proof is by construction. If L is a deterministic context-free language over the alphabet Σ, then L$ is accepted by some deterministic PDA M = (K, Σ U{$}, Γ, Δ, s, A).

We need to describe an algorithm that constructs a new deterministic PDA that accepts (¬L)$.

We defined a construction that proceeded in two steps:

- Given an arbitrary FSM, convert it to an equivalent DFSM, and then
- Swap accepting and non accepting states.

A deterministic PDA may fail to accept an input string *w* for any one of several reasons:

1. Its computation ends before it finishes reading *w*.
2. Its computation ends in an accepting state but the stack is not empty.
3. Its computation loops forever, following ε-transitions, without ever halting in either an accepting or a non accepting state.
4. Its computation ends in a non accepting state.

If we simply swap accepting and non accepting states we will correctly fail to accept every string that M would have accepted (i.e., every string in L$). But we will not necessarily accept every string in (¬L)$. To do that, we must also address issues 1through 3 above.

An additional problem is that we don't want to accept ¬L(M). That includes strings that do not end in $. We must accept only strings that do end in $ and that are in (¬L)$.

2) Non closure Under Union

Theorem: The deterministic context-free languages are not closed under union.

Proof: We show a counter example:

Let, L_1={ $a^i b^j c^k$| i,j,k ≥0 and i≠ j } and L_2={ $a^i b^j c^k$ | i,j,k≥0 and j ≠ k}

Let, L' = L_1U L_2= { $a^i b^j c^k$ | i,j,k≥0 and ((i≠ j) and (j ≠ k)) }.

Let, L'' = ¬L'.

$=\{ a^i b^j c^k \mid i,j,k \geq 0 \text{ and } (i=j=k)\} \cup \{w \in \{a,b,c\}^* : \text{the letters are out of order}\}$.

Let, $L''' = L'' \cap a^*b^*c^* = \{a^n b^n c^n \mid n \geq 0\}$

But L''' is $A^n B^n C^n = \{ a^n b^n c^n \mid n \geq 0\}$, which we have shown is not context-free.

3) Non Closure Under Intersection

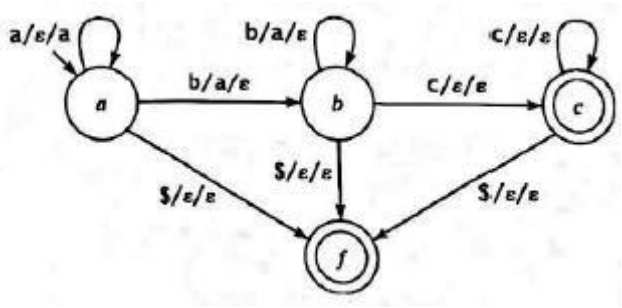
Theorem: The deterministic context-free languages are not closed under inter section.

Proof: We show a counter example:

Let, $L_1 = \{ a^i b^j c^k \mid i,j,k \geq 0 \text{ and } i = j \}$ and $L_2 = \{ a^i b^j c^k \mid i,j,k \geq 0 \text{ and } j = k\}$

Let, $L' = L_1 \cap L_2 = \{ a^n b^n c^n \mid n \geq 0\}$

L_1 and L_2 are deterministic context-free. The deterministic PDA shown accepts $L_1\$$, A similar one accepts L_2. But we have shown that their intersection L' is not context-free much less deterministic context-free.



A hierarchy within the class of context-free languages

Some CFLs are not Deterministic

Theorem: The class of deterministic context-free languages is a proper subset of the class of context-free languages. Thus there exist nondeterministic PDAs for which no equivalent deterministic PDA exists.

Proof: We show that there exists at least one context-free language that is not deterministic context-free.

Consider $L = \{ a^i b^j c^k \mid i,j,k \geq 0 \text{ and } ((i \neq j) \text{ or } (j \neq k)) \}$. L is context-free.

If L were deterministic context-free, then, its complement

$L' = \{ a^i b^j c^k \mid i,j,k \geq 0 \text{ and } (i=j=k) \} \cup \{w \in \{a,b,c\}^* : \text{the letters are out of order}\}$

Would also be deterministic context-free and thus context-free. If L' were context-free, then $L'' = L' \cap a^*b^*c^*$ would also be context-free (since the context-free languages are closed under inter section with the regular languages).

But $L'' = A^n B^n C^n = \{a^n b^n c^n \mid n \geq 0\}$, which is not context free.

So L is context-free but not deterministic context-free.

Since L is context-free, it is accepted by some (non deterministic) PDA M. M is an example of an on deterministic PDA for which no equivalent deterministic PDA L exists. If such a deterministic PDA did exist and accept L, it could be converted into a deterministic PDA that accepted L$. But, if that machine existed. L would be deterministic context-free and we just showed that it is not.

Inherent Ambiguity versus Non determinism

There are context-free languages for which unambiguous grammars exist and there are others that are inherently ambiguous, by which we mean that every corresponding grammar is ambiguous.

Example:

The language $L_1 = \{a^i b^j c^k \mid i, j, k \geq 0 \text{ and } ((i = j) \text{ or } (j = k))\}$ can also be described as $\{a^n b^n c^m | n,m \geq 0\} \cup \{a^n b^m c^m | n,m \geq 0\}$. L_1 is inherently ambiguous because every string that is also in $A^n B^n C^n = \{a^n b^n c^n | n \geq 0\}$ is an element of both sub languages and so has at least two derivations in any grammar for L_1.

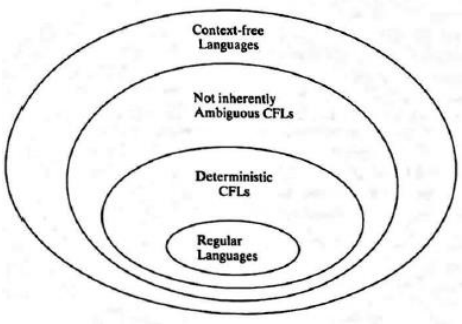
• Now consider the language $L_2 = \{a^n b^n c^m d | n,m \geq 0\} \cup \{a^n b^m c^m e | n,m \geq 0\}$ is not inherently ambiguous.

• Any string in is an element of only one of them (since each such string must end in d or e but not both).

There exists no PDA that can decide which of the two sublanguages a particular string is in until it has consumed the entire string.

What is the relationship between the deterministic context-free languages and the languages that are not inherently ambiguous?

The answer is shown in below Figure.



There exist deterministic context-free languages that are not regular. One example is $A^n B^n = \{a^n b^n | n,m \geq 0\}$.

•There exist context-free languages and not inherently ambiguous. Examples:

(a) $PalEven = \{ww^R : w \in \{a,b\}^*\}$.

(b) $\{a^n b^n c^m d | n,m \geq 0\} \cup \{a^n b^m c^m e | n,m \geq 0\}$.

•There exist languages that are in the outer donut because they are inherently ambiguous. Two examples are:

- $\{a^i b^j c^k | i,j,k \geq 0 \text{ and } ((i \neq j) \text{or} (j \neq k))\}$
- $\{a^i b^j c^k | i,j,k \geq 0 \text{ and } ((i=j) \text{ or } (j=k))\}$

Regular Language is Deterministic Context-Free

Theorem: Every regular language is deterministic context-free.

Proof: The proof is by construction. {$} is regular. So, if L is regular then so is L$ (since the regular languages are closed under concatenation).So there is a DFSM M that accepts it. Using the construction to show that every regular language is context-free Construct, from M a PDA P that accepts L$. P will be deterministic.

Every Deterministic CFL has an Unambiguous Grammar

Theorem: For every deterministic context-free language there exists an unambiguous grammar.

Proof: If a language L is deterministic context-free, then there exists a deterministic PDA M that accepts L$. We prove the theorem by construction of an unambiguous grammar G such that L(M)= L(G). We construct G as follows:

The algorithm PDAtoCFG proceeded in two steps:

1.Invoke ***convenPDAtorestricted(M)*** to build M', an equivalent PDA in restricted normal form.

2. Invoke ***buildgrammar(M'),*** to build an equivalent grammar G

So the construction that proves the theorem is:

buildunambiggrammar(M:deterministicPDA)=

1. Let G=*buildgrammar*(***convertPDAtodetnormalform*(M)**).
2. Let G' be the result of substituting ε for $ in each rule in which $ occurs.
3. Return G'.

NOTE: The algorithm ***convertPDAtodetnormalform***, is described in the theorem that proves the deterministic context-free languages are closed under complement.

The Decidable Questions

Membership

"Given a language L and a string *w*, is *w* in L?'

This question can be answered for every context-free language and for every context-free language L there exists a PDA M such that M accepts L. But existence of a PDA that accepts L does not guarantee the existence of a procedure that decides it.

It turns out that there are two alternative approaches to solving this problem, both of which work:

- **Use a grammar:** Using facts about every derivation that is produced by a grammar in Chomsky normal form, we can construct an algorithm that explores a finite number of derivation paths and finds one that derives a particular string w iff such a path exists.
- **Use a PDA** : While not all PDAs halt, it is possible, for any context-free language L, to craft a PDA M that is guaranteed to halt on all inputs and that accepts all strings in L and rejects all strings that are not in L.

Using a Grammar to Decide

Algorithm for deciding whether a string *w* is in a language L:

***decideCFLusingGrammar*(L: CFL,*w*: string) =**

1. If L is specified as a PDA, use ***PDA to CFG***, to construct a grammar G such that L(G) =L (M).

2. If L is specified as a grammar G, simply use G.

3. If w =ε then if S_G is nullable then accept, otherwise reject.

4. If w ≠ ε then:

   4.1. From G, construct G' such that L (G') = L(G)-{ε} and G' is in Chomsky normal form.

   4.2. If G derives to, it does so in (2 • |*w*| - 1) steps. Try all derivations in G of that number of steps. If one of them derives w, accept. Otherwise reject.

Using a PDA to Decide

A two-step approach.

- We first show that, for every context-free language L, it is possible to build a PDA that accepts L-{ε} and that has no ε-transitions.
- Then we show that every PDA with no ε-transitions is guaranteed to halt

Elimination of ε-Transitions

Theorem: Given any context-free grammar G=(V,Σ,R,S), there exists a PDA M such that L(M)=L(G)-{ε} and M contains no transitions of the form

((q1,ε,α),(q2,β)). In other words, every transition reads exactly one input character.

Proof: The proof is by a construction that begins by converting G to Greibach normal form. Now consider again the algorithm ***cfgtoPDAtopdown***, which builds, from any context-free grammar G, a PDA M that, on input w, simulates G deriving w, starting from S.

M= ({p,q},Σ,V,Δ, p,{q}), where Δ contains:

1. The start-up transition ((p,ε,ε),(q,S)), which pushes the start symbol on to the stack and goes to state q.
2. For each rule $X \rightarrow s_1 s_2 ... s_n$, in R, the transition $((q,\varepsilon,X),(q,s_1 s_2 ... s_n))$, which replaces X by $s_1 s_2 ... s_n$. If n=0 (i.e., the right-hand side of the rule is ε), then the transition ((q, ε, X), (q, ε)).
3. For each character c ∈ Σ. the transition ((q, c, c), (q,ε)), which compares an expected character from the stack against the next input character.

If G contains the rule $X \rightarrow c s_2 ... s_n$, (where *c* ∈Σ and s_2 through s_n, are elements of V-Σ), it is not necessary to push *c* onto the stack, only to pop it with a rule from step 3.

Instead, we collapse the push and the pop into a single transition. So we create a transition that can be taken only if the next input character is *c*. In that case, the string s_2 through s_n is pushed onto the stack.

Since terminal symbols are no longer pushed onto the stack. We no longer need the transitions created in step3 of the original algorithm.

So, M=({p,q},Σ,V,Δ,p,{q}), where Δ contains:

1. The start-up transitions: For each rule $S \rightarrow c s_2 ... s_n$ the transition $((p,c,\varepsilon),(q,s_2 ... s_n))$.
2. For each rule $X \rightarrow c s_2 ... s_n$ (where c∈Σ and s_2 through s_n, are elements of V-Σ), the transition $((q,c,X),(q,s_2 ... s_n))$.

cfgtoPDAnoeps(G:context-freegrammar)=

1. Convert G to Greibach normal form, producing G'.
2. From G' build the PDA M described above.

Halting Behavior of PDAs Without ε-Transitions

Theorem: Let M be a PDA that contains no transitions of the form $((q_1,\varepsilon,s1),(q_2,s_2))$. i.e., no ε-transitions. Consider the operation of M on input w∈Σ*. M must halt and either accept or reject w. Let n=|w|.

We make three additional claims:

a) Each individual computation of M must halt within *n* steps.

b) The total number of computations pursued by M must be less than or equal to b^n, where *b* is the maximum number of competing transitions from any state in M.

c) The total number of steps that will be executed by all computations of M is bounded by nb^n

Proof:

a) Since each computation of M must consume one character of w at each step and M will halt when it runs out of input, each computation must halt within n steps.

b) M may split into at most b branches at each step in a computation. The number of steps in a computation is less than or equal to n. So the total number of computations must be less than or equal to b^n.

c) Since the maximum number of computations is b^n and the maximum length of each is n, the maximum number of steps that can be executed before all computations of M halt is nb^n.

So a second way to answer the question, "Given a context-free language L and a string w, is w in L?" is to execute the following algorithm:

decideCFLusingPDA(L:CFL,w:string)=

1. If L is specified as a PDA, use ***PDAtoCFG,*** to construct a grammar G such that L(G)=L(M).
2. If L is specified as a grammar G, simply use G.
3. If w=ε then if S_G is nullable then accept, otherwise reject.
4. If w≠ε then:
    4.1. From G, construct G' such that L(G')=L(G)-{ε} and G' is in Greibach normal form.
    4.2. From G' construct, using ***cfgtoPDAnoeps***, a PDA M' such that L(M')=L(G') and M' has no ε-transitions.
    4.3. We have proved previously that, all paths of M' are guaranteed to halt within a finite number of steps. So run M' on w, Accept if M' accepts and reject otherwise.

Emptiness and Finiteness

Decidability of Emptiness and Finiteness

Theorem: Given a context-free language L. There exists a decision procedure that answers each of the following questions:

1. Given a context-free language L, is L=⦰?
2. Given a context-free language L, is L infinite?

Since we have proven that there exists a grammar that generates L iff there exists a PDA that accepts it. These questions will have the same answers whether we ask them about grammars or about PDAs.

Proof :

***decideCFLempty*(G: context-free grammar) =**

1. Let G' =***removeunproductive(G).***
2. If S is not present in G' then return True else return False.

***decideCFLinfinite*(G:context-freegrammar)=**

1. Lexicographically enumerate all strings in Σ* of length greater than b^n and less than or equal to $b^{n+1}+b^n$.
2. If, for any such string w, ***decideCFL*(L,w)** returns *True* then return *True*. L is infinite.
3. If, for all such strings w, ***decideCFL*(L,w)** returns *False* then return *False*. L is not infinite.

The Undecidable Questions

• Given a context-free language L, is L=Σ*?

• Given a CFL L, is the complement of L context-free?

• Given a context-free language L, is L regular?

• Given two context-free languages L_1 and L_2 is $L_1=L_2$?

• Given two context-free languages L_1 and L_2, is $L_1 \subseteq L_2$?

• Given two context-free languages $L_1$and L_2, is $L_1 \cap L_2 = \emptyset$?

• Given a context-free language L, is L inherently ambiguous?

• Given a context-free grammar G, is G ambiguous?

TURING MACHINE

The Turing machine provides an ideal theoretical model of a computer. Turing machines are useful in several ways:

• Turing machines are also used for determining the undecidability of certain languages and

• As an automaton, the Turing machine is the most general model, It accepts type-0 languages.

• It can also be used for computing functions. It turns out to be a mathematical model of partial recursive functions.

• Measuring the space and time complexity of problems.

Turing assumed that while computing, a person writes symbols on a one-dimensional paper (instead of a two dimensional paper as is usually done) which can be viewed as a tape divided into cells. In

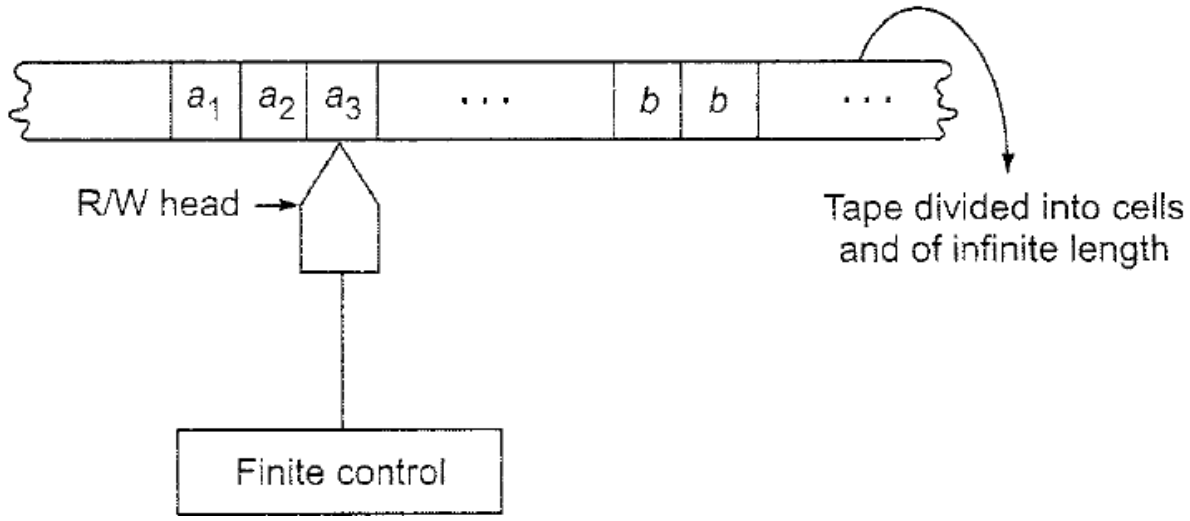
Turing machine one scans the cells one at a time and usually performs one of the three simple operations, namely:

(i) Writing a new symbol in the cell being currently scanned,

(ii) Moving to the cell left of the present cell, and

(iii) Moving to the cell right of the present cell.

Turing machine model



•Each cell can store only one symbol.

•The input to and the output from the finite state automaton are affected by the R/W head which can examine one cell at a time.

In one move, the machine examines the present symbol under the R/W head on the tape and the present state of an automaton to determine:

(i) A new symbol to be written on the tape in the cell under the R/W head,

(ii) A motion of the R/W head along the tape: either the head moves one cell left (L),or one cell right (R).

(iii) The next state of the automaton, and

(iv) Whether to halt or not.

Definition:

Turing machine M is a 7-tuple, namely $(Q, \Sigma, \mathbf{\Gamma}, \boldsymbol{\delta}, q_0, b, F)$, where

1. Q is a finite nonempty set of states.

2. $\mathbf{\Gamma}$ is a finite nonempty set of tape symbols,

3. $b \in \mathbf{\Gamma}$ is the blank.

4. Σ is a nonempty set of input symbols and is a subset o f $\mathbf{\Gamma}$ and $b \notin \Sigma$.

5. δ is the transition function mapping (q,x) onto (q',y,D) where D denotes the direction of movement of R/W head; D=L orR according as the movement is to the left or right.
6. $q_0 \in Q$ is the initial state, and
7. $F \subseteq Q$ is the set of final states.

Notes:

(1)The acceptability of a string is decided by the reachability from the initial state to some final state.

(2) δ may not be defined for some elements of QX Γ.

REPRESENTATION OF TURINGMACHINES

We can describe a Turing machine employing

(i) Instantaneous descriptions using move-relations.

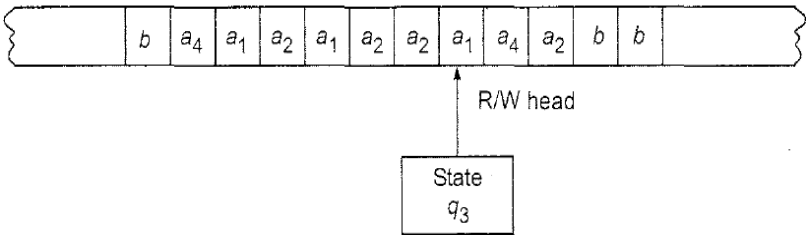
(ii) Transition table, and

(iii) Transition diagram (Transition graph).

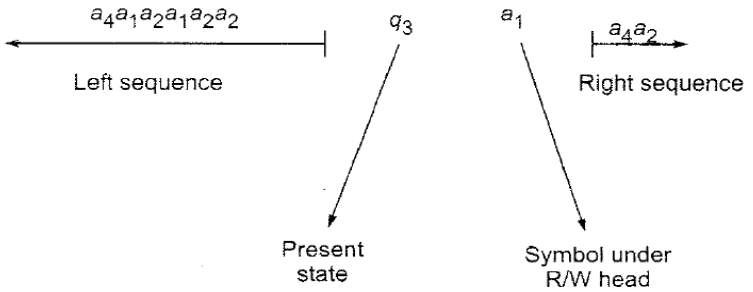
REPRESENTATION BY INSTANTANEOUS DESCRIPTIONS

Definition: An ID of a Turing machine M is a string $\alpha\beta\gamma$, where β is the present state of M, the entire input string is split as $\alpha\gamma$, the first symbol of γ is the current symbol $\boldsymbol{a}$ under the R/W head and γ has all the subsequent symbols of the input string, and the string α is the substring of the input string formed by all the symbols to the left of $\boldsymbol{a}$.

EXAMPLE: A snapshot of Turing machine is shown in below Fig. Obtain the instantaneous description.



The present symbol under the R/W head is a_1. The present state is q_3. So a_1 is written to the right of q_3 The nonblank symbols to the left of al form the string $a_4a_1a_2a_1a_2a_2$, which is written to the left of q_3. The sequence of nonblank symbols to the right of a_1 is a_4a_2. Thus the ID is as given in below Fig.



Notes: (1) For constructing the ID, we simply insert the current state in the input string to the left of the symbol under the R/W head.

(2) We observe that the blank symbol may occur as part of the left or right substring.

REPRESENTATION BY TRANSITION TABLE

We give the definition of δ in the form of a table called the transition table If (q, a)=(γ,α,β). We write $\alpha\beta\gamma$ under the α-column and in the q-row. So if we get $\alpha\beta\gamma$ in the table, it means that α is written in the current cell, β gives the movement of the head (L or R) and γ denotes the new state into which the Turing machine enters.

EXAMPLE:

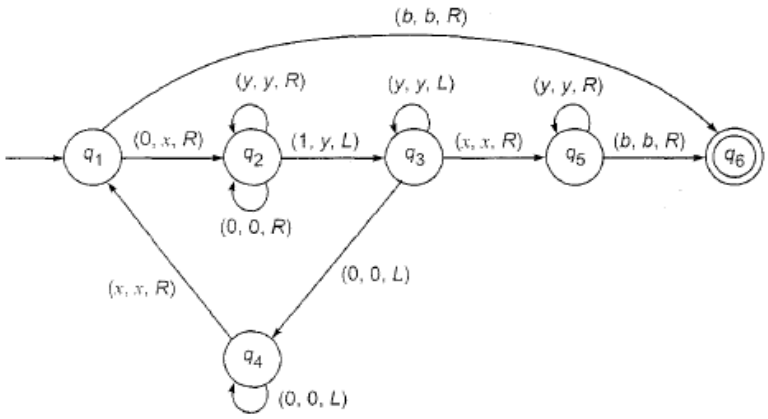
Consider, for example, a Turing machine with five states $q_1,...,q_5$ where q_1 is the initial state and q_5 is the (only) final state. The tape symbols are 0,1and b. The transition table given below describes δ:

Present state	Tape symbol		
	b	0	1
$\rightarrow q_1$	$1Lq_2$	$0Rq_1$	
q_2	bRq_3	$0Lq_2$	$1Lq_2$
q_3		bRq_4	bRq_5
q_4	$0Rq_5$	$0Rq_4$	$1Rq_4$
(q_5)	$0Lq_2$		

REPRESENTATION BY TRANSITION DIAGRAM (TD)

The states are represented by vertices. Directed edges are used to represent transition of states. The labels are triples of the form (α,β,γ)where $\alpha,\beta \in \Gamma$and$\gamma \in$ {L,R}.When there is a directed edge from q_i to q_j with label (α,β,γ),it means that $\delta(q_i,\alpha)=(q_j,\beta,\gamma)$.

EXAMPLE:



LANGUAGE ACCEPTABILITY BY TURING MACHINES

Let us consider the Turing machine M=(Q,Σ,**Γ**,**δ**,q0,b,F). A string *w* in Σ* is said to be accepted by M, if $q_0 w \overset{*}{\vdash} \alpha_1 p \alpha_2$ for some P∈F and $\alpha_1, \alpha_2 \in \Gamma^*$.

EXAMPLE: Consider the Turing machine M described by the table below

Present state	Tape symbol				
	0	1	x	y	b
→ q_1	xRq_2				bRq_5
q_2	$0Rq_2$	yLq_3		yRq_2	
q_3	$0Lq_4$		xRq_5	yLq_3	
q_4	$0Lq_4$		xRq_1		
q_5				$yxRq_5$	bRq_5
(q6)					

IDs for the strings (a) 011 (b)0011 (c)001

(a) $q_1011 \vdash xq_211 \vdash q_3xy1 \vdash xq_5y1 \vdash xyq_51$

As (q_5,1) is not defined, M halts; so the input string 011 is not accepted

(b) $q_10011 \vdash xq_2011 \vdash x0q_211 \vdash xq_30y1 \vdash q_4x0y1 \vdash xq_10y1$
$\vdash xxq_2y1 \vdash xxyq_21 \vdash xxq_3yy \vdash xq_3xyy \vdash xxq_5yy$
$\vdash xxyq_5y \vdash xxyyq_5b \vdash xxyybq_6$

M halts. As q_6 is an accepting state, the input string 0011is accepted byM.

(c) $q_1001 \vdash xq_201 \vdash x0q_21 \vdash xq_30y \vdash q_4x0y \vdash xq_10y$
$\vdash xxq_2y \vdash xxyq_2$

M halts. As q_2 is not an accepting state,001 is not accepted by M.

DESIGN OF TURING MACHINES

Basic guidelines for designing a Turing machine:

1. The fundamental objective in scanning a symbol by the R/W head is to know what to do in the future. The machine must remember the past symbols scanned. The Turing machine can remember this by going to the next unique state.

2. The number of states must be minimized. This can be achieved by changing the states only when there is a change in the written symbol or when there is a change in the movement of the R/W head.

EXAMPLE 1

Design aTuring machine to recognize all strings consisting of an even number of 1's.

Solution:

The construction is made by defining moves in the following manner:

(a) q_1 is the initial state. M enters the state q_2 on scanning 1 and writes b.

(b) If M is in state q_2 and scans 1, it enters q_1 and writes b.

(c) q_1 is the only accepting state.

Symbolically M= ($\{q_1,q_2\},\{1,b\},\{1,b\},\delta,q,b,\{q_1\}$), Where δ is defined by ,

Present state	1
$\rightarrow q_1$	bq_2R
q_2	bq_1R

Let us obtain the computation sequence of 11:

$q_111 \vdash bq_21 \vdash bbq_1$

As q_1 is an accepting state 11 is accepted.

Let us obtain the computation sequence of 111:

$q_1111 \vdash bq_211 \vdash bbq_11 \vdash bbbq_2$

As q_2 is an not accepting state 111 is not accepted.

EXAMPLE 2: Design a TM that accepts $\{0^n1^n \mid n \geq 0\}$

Solution: We require the following moves:

(a) If the leftmost symbol in the given input string *w* is 0, replace it by *x* and move right till we encounter a leftmost 1in *w*. Change it to *y and* move backwards.

(b) Repeat (a) with the leftmost 0. If we move back and forth and no 0 or 1 remains. Move to a final state.

(c) For strings not in the form 0^n1^n, the resulting state has to be non-final.

we construct a **TM *M as follows:*** $M = (Q, \Sigma, \Gamma, \delta, q_0, b, F)$

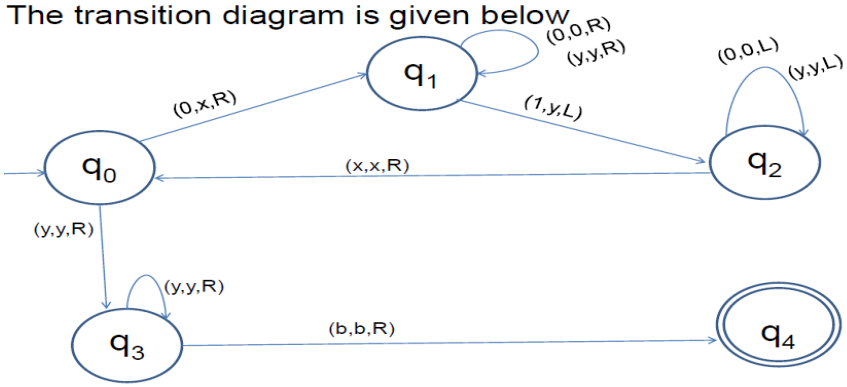
$Q = \{q_0, q_1, q_2, q_3, q_f\}$

$F = \{q_f\}$

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

$\Gamma = \{0, 1, x, y, b\}$

The transition diagram is given below



Computation sequence of 0011:

$$q_0 0011 \vdash xq_1 011 \vdash x0q_1 11 \vdash xq_2 0y1 \vdash q_2 x0y1 \vdash xq_0 0y1 \vdash xxq_1 y1 \vdash xxyq_1 1 \vdash xxq_2 yy \vdash xq_2 xyy \vdash xxq_0 yy \vdash xxyq_3 y \vdash xxyyq_3 = xxyyq_3 b \vdash xxyybq_4 b$$

q_4 is final state, hence 0011 is accepted by M.

TECHNIQUES FOR TM CONSTRUCTION

1. TURING MACHINE WITH STATIONARY HEAD

Suppose, we want to include the option that the head can continue to be in the same cell for some input symbol. Then we define (q,a) as (q',y,S).This means that the TM, on reading the input symbol a, changes the state to q' and writes y in the current cell in place of a and continues to remain in the same cell. In this model (q, a) =(q', y, D) where D = L, R or S.

2. STORAGE IN THE STATE

We can use a state to store a symbol as well. So the state becomes a pair *(q,a)* where q is the state and a is the tape symbol stored in (q, a). So the new set of states becomes Q$\times\Gamma$.

EXAMPLE: Construct a TM that accepts the language 0 1* + 1 0*.

We have to construct a TM that remembers the first symbol and checks that it does not appear afterwards in the input string.

So we require two states, q_0, q_1. The tape symbols are 0,1 and b. So the TM, having the **'storage facility in state**', is $M=(\{q_0,q_1\}X\{0,1,b\},\{0,1\},\{0,1,b\},\delta,[q_0,b],[q_1,b]\})$

We describe δ by its implementation description.

1. In the initial state, M is in q_0 and has b in its data portion. On seeing the first symbol of the input sting w, M moves right, enters the state q_1 and the first symbol, say a, it has seen.

2. M is now in $[q_1,a]$.

   (i) If its next symbol is b, M enters $[q_1,b]$, an accepting state.

   (ii) If the next symbol is a, M halts without reaching the final state (i.e.δ is not defined).

   (iii) If the next symbol is ā, (ā=0 if a=1 and ā=1 if a=0), M moves right without changing state.

3. Step2 is repeated until M reaches $[q_1,b]$ or halts (δ is not defined for an input symbol in w).

3. MULTIPLE TRACK TURING MACHINE

In a multiple track TM, a single tape is assumed to be divided into several tracks. Now the tape alphabet is required to consist of k-tuples of tape symbols, k being the number of tracks. In the case of the standard Turing machine, tape symbols are elements of r; in the case of TM with multiple tracks, it is Γ^k.

4. SUBROUTINES

First a TM program for the subroutine is written. This will have an initial state and a 'return' state. After reaching the return state, there is a temporary halt for using a subroutine, new states are introduced. When there is a need for calling the subroutine, moves are effected to enter the initial state for the subroutine. When the return state of the subroutine is reached, return to the main program of TM.

EXAMPLE: Design a TM which can multiply two positive integers.

Solution: The input (m,n), m,n being given ,the positive integers represented by 0^m10^n. M starts with 0^m10^n in its tape. At the end of the computation, 0^{mn} (mn in unary representation) surrounded by b's is obtained as the output.

The major steps in the construction are as follows:

1. 0^m10^n1 is placed on the tape (the output will be written after the rightmost 1).
2. The leftmost 0 is erased.
3. A block of n 0's is copied onto the right end.
4. Steps 2 and 3 are repeated m times and 10^m10^{mn} is obtained on the tape.
5. The prefix $10^m1$of 10^m10^{mn} is erased, leaving the product 0^{mn} as the output.

For every 0 in 0^m, 0^n is added onto the right end. This requires repetition of step3. We define a subroutine called COPY for step3. For the subroutine COPY the initial state is q_1 *and the final state is* q_5 is given by the transition table as below:

The transition table for the **SUBROUTINE COPY**

State	Tape symbol			
	0	1	2	b
q_1	$q_2 2R$	$q_4 1L$	—	—
q_2	$q_2 0R$	$q_2 1R$	—	$q_3 0L$
q_3	$q_3 0L$	$q_3 1L$	$q_1 2R$	—
q_4	—	$q_5 1R$	$q_4 0L$	—
q_5	—	—	—	—

The Turing machine M has the initial state q_0. The initial ID for M is $q_00^m10^n$. O^n seeing 0,the following moves take place

$$q_00^m10^n1 \vdash bq_60^{m-1}10^n1 \vdash^* b0^{m-1}q_610^n1 \vdash b0^{m-1}1q_10^n1$$

q_1 is the initial state of COPY. The following moves take place for M_1:

$$q_10^n1 \vdash 2q_20^{n-1}1 \vdash^* 20^{n-1}1q_3b \vdash 20^{n-1}q_310 \vdash^* 2q_10^{n-1}10$$

After exhausting 0s, q_1 encounters 1. M_1 moves to state q_4. All 2's are converted back to 0's and M_1 halts in q_5. The TM M picks up the computation by starting from q_5 The q_0 and q_6 are the states of M. Additional states are created to check whether reach 0 in 0^m gives rise to 0^m at the end of the rightmost 1 in the input string. Once this is over, M erases 10^n1 and finds 0^{mn} in the input tape.

M can be defined by M=($\{q_0,q_1,....q_{12}\}\{0,1\},\{0,,2,b\},\boldsymbol{\delta},q_0,b,\{q_{12}\}$) where $\boldsymbol{\delta}$ is defined by table given below:

	0	1	2	b
q_0	q_6bR	—	—	—
q_6	q_60R	q_11R	—	—
q_5	q_70L	—	—	—
q_7	—	q_81L	—	—
q_8	q_90L	—	—	$q_{10}bR$
q_9	q_90L	—	—	q_0bR
q_{10}	—	$q_{11}bR$	—	—
q_{11}	$q_{11}bR$	$q_{12}bR$	—	—

ADDITIONAL PROBLEMS

1. Design a Turing machine to obtain complement of a binary number.

IDEA OF CONSTRUCTION:

1) If symbol is 0 change it to 1, move read write head to RIGHT
2) If symbol is 1 change it to 0, move read write head to RIGHT
3) Symbol is b (blank) don't change, move read write head to RIGHT, and HALT.

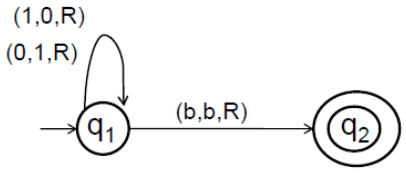
The construction is made by defining moves in the following manner:

(a) q_1 is the initial state. On scanning 1, no change in state and write 0 and move head to RIGHT.

(c) If M is in state $q_1$and scans blank, it enters q_2 and writes b move to right.

(d) q_2 is the only accepting state.

Symbolically, $M=(\{q_1,q_2\},\{1,0,b\},\{1,0,b\},\boldsymbol{\delta},q_1,b,\{q_2\})$ Where $\boldsymbol{\delta}$ is defined by:



The computation sequence of 1010:

$q_11010 \vdash 0q_1010 \vdash 01q_110 \vdash 010q_10 \vdash 0101q_1b$

$\vdash 0101bq_2b$

2. Design a TM that converts binary number into its 2's complement representation.

IDEA OF CONSTRUCTION:

- Read input from left to right until right end blank is scanned.
- Begin scan from right to left keep symbols as it is until 1 found on input file.

- If 1 found on input file, move head to left one cell without changing input.
- Now until left end blank is scanned, change all 1's to 0 and 0's to 1.

We require the following moves:

(a) Let q_1 be initial state, until blank is scanned, move head to RIGHT without changing anything. On scanning blank, move head to RIGHT change state to q_2 without changing the content of input.

(b) If q_2 is the state, until 1 is scanned, move head to LEFT without changing anything. On reading 1, change state to q_3, move head to LEFT without changing input.

(c) If q_3 is the state, until blank is scanned, move head to LEFT, if symbol is 0 change to 1, otherwise if symbol is 1 change to 0.On finding blank change state to q_4, move head to LEFT without Changing input.

(d) q_4 is the only accepting state.

We construct a **TM *M as follows:***

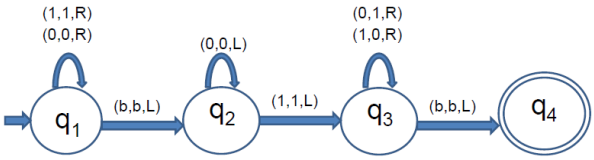
M = (Q, Σ,, **δ**, q_0,b, F)

Q = {q_1,q_2,q_3,q_4}

F = {q_4 }

Σ = {0,1}

Γ= {0,1,b}



3.Design a TM that add two integers

IDEA OF CONSTRUCTION:

- Read input from LEFT to RIGHT until blank (separator of two numbers) is found.
- Continue LEFT to RIGHT until blank (end of second number) is found.
- Change separator b to 1 move head to RIGHT.
- move header to Left (to point rightmost 1)
- Change 1 to b and move right, Halt.

We require the following moves:

(a) In q_1 TM skips1's until it reads b (separator),changes to1and goes to q_1

(b) In q_2 TM skips1's until it reads b (end of input), turns left and goes to q_3

(c) In q_3, TM reads 1 and changes to b go to q_4.

(d) q_4 is the final state, TM halts.

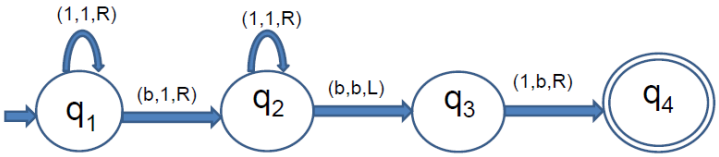
we construct a **TM M as follows:** M = (Q, Σ, **Γ**, **δ**, q_0,b, F)

$Q = \{q_1,q_2,q_3,q_4\}$

$F = \{q_4\}$

$\Sigma = \{ b,1\}$

Γ= {1,b}



4. Design a TM that accepts the set of all palindromes over {0,1}*

IDEA OF CONSTRUCTION:

- If it is 0 and changes to X, similarly if it is 1, it is changed to Y, and moves right until it finds blank.
- Starting at the left end it checks the first symbol of the input,
- Nowmovesonestepleftandcheckwhetherthesymbolreadmatchesthemostrecentlychanged.Ifsoiti salsochangedcorrespondingly.
- Now machine moves back left until it finds 0 or 1.
- This process is continued by moving left and right alternately until all 0's and 1's have been matched.

We require the following moves:

1.If state is q0and it scans 0.

- Then go to state q1 and change the 0 to an X,
- move RIGHT over all 0's and 1's, until it finds either X or Y or B
- Now move one step left and change state to q_3
- It verifies that the symbol read is 0, and changes the 0 to X and goes to state q_5.

2. If state is q0 and it scans 1

- Then go to state q_2 and change the 1 to an Y,
- Move RIGHT over all 0's and 1's, until it finds either X or Y or B
    - Now move one step left and change state to q_4
    - It verifies that the symbol read is 1, and changes the 1 to Y and goes to state q_5.

3. If state is q_5

- Move LEFT over all 0's and 1's, until it finds either X or Y.
- Now move one step RIGHT and change state to q_0.

- Now at q_0 there are two cases:

  1. If 0's and 1's are found on input , it repeats the matching cycle just described.
  2. If X's and Y's are found on input, then it changes all the 0's to X and all the 1's to Y's. **The input was a palindrome of even length,** Thus, state changed to q_6.

4.If state is q_3 or q_4

If X's and Y's are found on input, it concludes that: **The input was a palindrome of odd length, t**hus, state changed to q_6.

We construct a **TM *M as follows:***

$M = (Q, \Sigma, \Gamma, \delta, q_0, b, F)$

$Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}$

$F = \{q_6\}$

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

$\Gamma = \{X, Y, b\}$

state	0	1	X	Y	B
q_0	(q_1, X, R)	(q_2, Y, R)	(q_6, X, R)	(q_6, Y, R)	(q_6, B, R)
q_1	$(q_1, 0, R)$	$(q_1, 1, R)$	(q_3, X, L)	(q_3, Y, L)	(q_3, B, L)
q_2	$(q_2, 0, R)$	$(q_2, 1, R)$	(q_4, X, L)	(q_4, Y, L)	(q_4, B, L)
q_3	(q_5, X, L)	-	(q_6, X, R)	(q_6, Y, R)	-
q_4	-	(q_5, Y, L)	(q_6, X, R)	(q_6, Y, R)	-
q_5	$(q_5, 0, L)$	$(q_5, 1, L)$	(q_0, X, R)	(q_0, Y, R)	-

PRACTICE PROBLEMS

1. Design a Turing machine to replace all a's with X and all b's with Y.
2. Design a Turing machine to accept $a^n b^m$ n>m.
3. Design a Turing machine to accept $a^n b^n$ n<m.
4. Design a Turing machine to accept (0+1)*00(0+1)* .
5. Design a Turing machine to increment a given input.
6. Design a Turing machine to decrement a given input.
7. Design a Turing machine to subtract two unary numbers.
8. Design a Turing machine to multiply two unary numbers.
9. Design a Turing machine to accept a string 0's followed by a 1.

10. Design a Turing machine to verify if the given binary number is an even number or not.
11. Design a Turing machine to shift the given input by one cell to left.
12. Design a Turing machine to shift the given input to the right by one cell .
13. Design a Turing machine to rotate a given input by one cell.
14. Design a Turing machine to erase the tape.
15. Design a Turing machine to accept $a^n b^n c^n$.
16. Design a Turing machine to accept any string of a's & b's with equal number of a's & b's.
17. Design a Turing machine to accept $a^n b^{2n}$.
18. Design a Turing machine to accept $a^n b^k c^m$: where n=m+k.
19. Design a Turing machine to accept $a^n b^k c^m$: where m=n+k.
20. Design a Turing machine to accept $a^n b^k c^m$: where k=m+n.

From: Dr shreedhara K S, Professor in CSE, UBDT college of engineering, Davanagere

Automata Theory and Computability

Module 5

The model of Linear Bounded automata: Decidability: Definition of an algorithm, decidability, decidable languages, Undecidable languages, halting problem of TM, Post correspondence problem. Complexity: Growth rate of functions, the classes of P and NP, Quantum Computation: quantum computers, Church-Turing thesis.

- A word automata is a plural of word "automation", which means to automate or mechanize. Mechanization of a process means performing it on a machine without human intervention.
- The basic aim of Computer Science is to design Computing Machine (CM).
- To design Computing Machine for a problem it is necessary to ensure that the problem is solvable and computable.
- If it is not solvable in a reasonable amount of time, it is solvable in principle only
- As a student of Computer Science , we should know what is computable, and if it is computable, how it can be implemented on a machine.
- Aim of automata theory is to draw a boundary between what is computable and what is not, if computation is performed on a machine,
  Machine may be of two types
  1. problem specific dedicated machine
  2. Generic machine.

Church-Turing thesis-1936

- Any algorithmic procedure that can be carried out by a human or a computer, can also be carried out by a Turing machine.
- Now it is universally accepted by computer scientists that TM is a Mathematical model of an algorithm.
- TM has an algorithm and an algorithm has a TM. If there is an algorithm problem is decidable, TM solves that problem
- The statement of the thesis –

" Every function which would naturally be regarded as computable can be computed by a Turing machine"

Implies

- Any mechanical computation can be performed by a TM
- For every computable problem there is a TM
- If there is no TM that decides P there is no algorithm that can solve problem P.
- In our general life, we have several problems and some of these have solutions, but some have not, we simply say a problem is decidable if there is a solution otherwise undecidable.

example:

- Does Sun rises in the East? YES
- Will tomorrow be a rainy day ? (YES/NO ?)

Decidable and Undecidable Languages

- A problem is said to be decidable if its language is recursive OR it has solution.

Example:

Decidable :

-Does FSM accept regular language?

- is the power of NFA and DFA same

Undecidable:

- For a given CFG is L(G) ambiguous?

L is *Turing decidable* (or just decidable) if there exists a Turing machine M that accepts all strings in L and rejects all strings not in L. Note that by rejection means that the machine halts after a finite number of steps and announces that the input string is not acceptable.

- There are two types of TMs (based on halting):

1. **(*Recursive*)**
   TMs that *always* halt, no matter accepting or non-accepting ≡ **DECIDABLE** PROBLEMS
2. **(*Recursively enumerable*)**
   TMs that *are guaranteed to halt* only on acceptance.
   If non-accepting, it may or may not halt (i.e., could loop forever).
   - Undecidable problems are those that are not recursive

Recursive languages

A Language L over the alphabet∑ is called recursive if there is a TM M that accepts every word in L and rejects every word in L'

Accept (M)=L

Reject(M)=L'

loop(M)=ø

Example: b(a+b)*

M is a *Turing Machine* and L is a *recursive language* that M accepts,
if a string w ∈ L then M *halts in a final state* and
if w ∉ L then M *halts in a non-final state*

Recursively Enumerable Language:

A Language L over the alphabet∑ is called recursively enumerable if there is a TM M that accepts every word in L and either rejects or loops every word in L' the complement of L

Accept (M)=L

Reject(M) +Loop(M)= L'

Example: (a+b)*bb(a+b)*

M is a *Turing Machine* and L is a *recursively enumerable language* that M accepts,
if a string w ∈ L then M *halts in a final state* and
if w ∉ L then M *halts in a non-final state or loops forever*

Theorem: If L is recursive language, so is $\overline{L}$

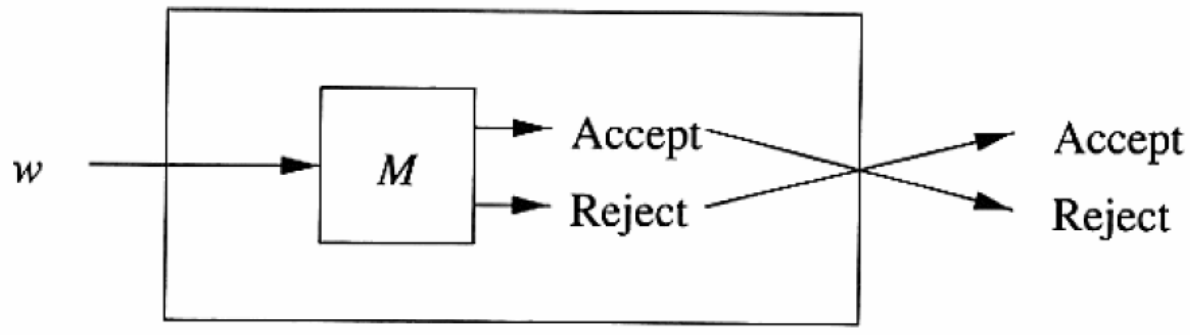
PROOF: Let $L = L(M)$ for some TM M that always halts. We construct a TM $\overline{M}$ such that $\overline{L} = L(\overline{M})$ by the construction suggested in Fig. 10. That is, $\overline{M}$ behaves just like M. However, M is modified as follows to create $\overline{M}$:

1. The accepting states of M are made nonaccepting states of $\overline{M}$ with no transitions; i.e., in these states $\overline{M}$ will halt without accepting.

2. $\overline{M}$ has a new accepting state r; there are no transitions from r.

3. For each combination of a nonaccepting state of M and a tape symbol of M such that M has no transition (i.e., M halts without accepting), add a transition to the accepting state r.

Since M is guaranteed to halt, we know that $\overline{M}$ is also guaranteed to halt. Moreover, $\overline{M}$ accepts exactly those strings that M does not accept. Thus $\overline{M}$ accepts $\overline{L}$. □



Recursively Enumerable Languages closed under complementation? (NO)

1. **Prove that Recursive Languagess are closed under Union**
   - Let M_u = TM for $L_1 \cup L_2$
   - M_u construction:
     1. Make 2-tapes and copy input w on both tapes
     2. Simulate M_1 on tape 1
     3. Simulate M_2 on tape 2
     4. If either M_1 or M_2 accepts, then M_u accepts
     5. Otherwise, M_u rejects.

2. **Prove that Recursive Languages are closed under Intersection**
   - Let M_n = TM for $L_1 \cap L_2$
   - M_n construction:
     1. Make 2-tapes and copy input w on both tapes
     2. Simulate M_1 on tape 1
     3. Simulate M_2 on tape 2
     4. If M_1 AND M_2 accepts, then M_n accepts
     5. Otherwise, M_n rejects.

3. Recursive languages are also closed under:
   a. Concatenation
   b. Kleene closure (star operator)
   c. Homomorphism, and inverse homomorphism
4. RE languages are closed under:
   a. Union, intersection, concatenation, Kleene closure
5. RE languages are *not* closed under:
   a. Complementation

1. **Decidable Languages about DFA : Prove that**

A_{DFA} is a decidable language.

$$A_{\text{DFA}} = \{\langle B, w\rangle \mid B \text{ is a DFA that accepts input string } w\}.$$

M = "On input $\langle B, w\rangle$, where B is a DFA and w is a string:

1. Simulate B on input w.
2. If the simulation ends in an accept state, *accept*. If it ends in a nonaccepting state, *reject*."

Proof: To prove we construct a TM that halts and also accept A_{DFA}
Define TM as

1. let B be a DFA and w input string (B,w) as input for TM M
2. Simulate B and input w in TM M
3. if the simulation ends in an accepting state of B then M accepts w. if it ends in non accepting state of B then M rejects w.

2. Prove that A_{NFA} is a decidable language.

$$A_{NFA} = \{\langle B, w\rangle | \ B \text{ is an NFA that accepts input string } w\}.$$

N = "On input $\langle B, w\rangle$ where B is an NFA, and w is a string:

1. Convert NFA B to an equivalent DFA C, using the **last** procedure .
2. Run TM M on input $\langle C, w\rangle$.
3. If M accepts, *accept*; otherwise, *reject*."

3. Prove that A_{REX} is a decidable language.

$$A_{REX} = \{\langle R, w\rangle | \ R \text{ is a regular expression that generates string } w\}.$$

P = "On input $\langle R, w\rangle$ where R is a regular expression and w is a string:

1. Convert regular expression R to an equivalent DFA A
2. Run TM N on input $\langle A, w\rangle$.
3. If N accepts, *accept*; if N rejects, *reject*."

Halting and Acceptance Problems:

$$A_{TM} = \{\langle M, w\rangle | \ M \text{ is a TM and } M \text{ accepts } w\}.$$

Acceptance Problem:
Does a Turing machine accept an input string?

1. A_{TM} is recursively enumerable.

Simulate M on w. if M enters an accepting state, We prove by contradiction. We assume **A_{TM} is decidable by a TM H that eventually halts on all input, then**

$$H(\langle M, w\rangle) = \begin{cases} accept & \text{if } M \text{ accepts } w \\ reject & \text{if } M \text{ does not accept } w. \end{cases}$$

$D =$ "On input $\langle M\rangle$, where M is a TM:

1. Run H on input $\langle M, \langle M\rangle\rangle$.
2. Output the opposite of what H outputs; that is, if H accepts, *reject* and if H rejects, *accept*."

$$D(\langle M\rangle) = \begin{cases} accept & \text{if } M \text{ does not accept } \langle M\rangle \\ reject & \text{if } M \text{ accepts } \langle M\rangle. \end{cases}$$

$$D(\langle D\rangle) = \begin{cases} accept & \text{if } D \text{ does not accept } \langle D\rangle \\ reject & \text{if } D \text{ accepts } \langle D\rangle. \end{cases}$$

We construct new TM D with H as a subroutine. D calls H to determine what M does when it receive the input <M>. Based on the received information on (M,<M>), D rejects M if M accepts <M> and accepts M if M rejects <M>.

D described as follows:
1. <M> is an input to D
2. D calls H to run on (M,<M>)
3. D rejects <M> if H accepts (M, <M>) and accepts <M>if H rejects (M, <M>)

This means D accepts <D> if D does not accept <D>, which is a contradiction. Hence **A_{TM}** is Undecidable.

The Post Correspondence Problem

PCP is a combinatorial problem formulated by Emil Post in 1946. This problem has many applications in the field theory of formal languages. A correspondence system P is a finite set of ordered pairs of non empty strings over some alphabet. Let $A = w_1, w_2, \ldots, w_n$ $B = v_1, v_2, \ldots, v_n$

There is a **P**ost **C**orrespondence Solution if there is a sequence $i, j, \ldots, k$ such that:

$$w_i w_j \cdots w_k = v_i v_j \cdots v_k$$

Index	w_j	V_i
1	100	001
2	11	111
3	111	11

Let W = $w_2w_1w_3=v_2v_1v_3$ = 11100111 we have got a solution. But we may not get solution always for various other combinations and strings of different length. Hence PCP is undecidable.

The **M**odified **P**ost **C**orrespondence Problem

$A = w_1, w_2, \ldots, w_n$ $B = v_1, v_2, \ldots, v_n$

$$1, i, j, \ldots, k$$

$$w_1 w_i w_j \cdots w_k = v_1 v_i v_j \cdots v_k$$

If the index start with 1 and then any other sequence then it is called MPCP

Algorithm: An algorithm is "a finite set of precise instructions for performing a computation or for solving a problem"

- A program is one type of algorithm
  - All programs are algorithms
  - Not all algorithms are programs!
- The steps to compute roots of quadratic equation is an algorithm
- The steps to compute the cosine of 90° is an algorithm

Algorithms generally share a set of properties:

- Input: what the algorithm takes in as input
- Output: what the algorithm produces as output
- Definiteness: the steps are defined precisely
- Correctness: should produce the correct output
- Finiteness: the steps required should be finite
- Effectiveness: each step must be able to be performed in a finite amount of time

- Generality: the algorithm *should* be applicable to all problems of a similar form

Comparing Algorithms (While comparing two algorithm we use time and space complexities)

- Time complexity
  - The amount of time that an algorithm needs to run to completion
- Space complexity
  - The amount of memory an algorithm needs to run
- To analyze running time of the algorithm we use following cases
  - Best case
  - Worst case

    Average case

Asymptotic analysis

- The big-Oh notation is used widely to characterize running times and space bounds
- The big-Oh notation allows us to ignore constant factors and lower order terms and focus on the main components of a function which affect its growth
- Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there are positive constants c and n_0 such that

$$f(n) \leq cg(n) \text{ for } n \geq n_0$$

- Example: $2n + 10$ is $O(n)$
  - $2n + 10 \leq cn$
  - $(c - 2)\, n \geq 10$
  - $n \geq 10/(c - 2)$

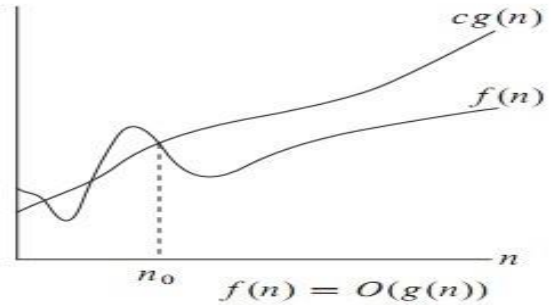
  It is true for $c = 3$ and $n_0 = 10$

- $7n-2$ is $O(n)$

  need $c > 0$ and $n_0 \geq 1$ such that $7n-2 \leq c \bullet n$ for $n \geq n_0$

  this is true for $c = 7$ and $n_0 = 1$

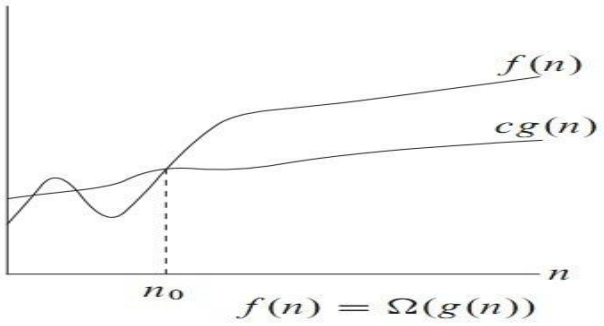
f(n)=O(g(n)) *iff* there exist positive constants c and n0 such that f(n) ≤ cg(n) for all n ≥ n0
O-notation to give an upper bound on a function



Big oh provides an asymptotic upper bound on a function.

Omega provides an asymptotic lower bound on a function.

$$\Omega(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le cg(n) \le f(n) \text{ for all } n \ge n_0\}.$$



- The big-Oh notation gives an upper bound on the growth rate of a function
- The statement "***f***(***n***) is ***O***(***g***(***n***))" means that the growth rate of ***f***(***n***) is no more than the growth rate of ***g***(***n***)
- We can use the big-Oh notation to rank functions according to their growth rate

$$f(n) = a_0 + a_1 n + a_2 n^2 + \ldots + a_d n^d$$

- If is ***f***(***n***) a polynomial of degree ***d***, then ***f***(***n***) is ***O***(n^d), i.e.,
    1. Drop lower-order terms
    2. Drop constant factors
- Use the smallest possible class of functions
    1. Say "2***n*** is ***O***(***n***)" instead of "2***n*** is ***O***(n^2)"
- Use the simplest expression of the class
    Say "3***n*** + 5 is ***O***(***n***)" instead of "3***n*** + 5 is ***O***(3***n***)"
- Following are the terms usually used in algorithm analysis:
    1. Constant ≈ ***1***
    2. Logarithmic ≈ log ***n***
    3. Linear ≈ ***n***
    4. N-Log-N ≈ ***n*** log ***n***
    5. Quadratic ≈ n^2
    6. Cubic ≈ n^3
    7. Exponential ≈ 2^n

Class P Problems:

P stands for deterministic polynomial time. A deterministic machine at each time executes an instruction. Depending on instruction, it then goes to next state which is unique. Hence time complexity of DTM is the maximum number of moves made by M is processing any input string of length n, taken over all input of length n.

- The class P consists of those problems that are solvable in polynomial time.
- More specifically, they are problems that can be solved in time $O(n^k)$ for some constant k, where n is the size of the input to the problem
- The key is that n is the **size of input**

Def: A language L is said to be in class P if there exists a DTM M such that M is of time complexity P(n) for some polynomial P and M accepts L.

Class NP Problems

Def: A language L is in class NP if there is a nondeterministic TM such that M is of time complexity P(n) for some polynomial P and M accepts L.

- **NP is not the same as non-polynomial complexity/running time. NP does not stand for not polynomial.**
- **NP = Non-Deterministic polynomial time**
- NP means verifiable in polynomial time
- Verifiable?
    - If we are somehow given a 'certificate' of a solution we can verify the legitimacy in polynomial time

- Problem is in NP iff it is decidable by some non deterministic Turing machine in polynomial time.
- It is provable that a Non Deterministic Turing Machine is equivalent to a Deterministic Turing Machine
- Remember NFA to DFA conversion?
    - Given an NFA with n states how many states does the equivalent DFA have?
    - Worst case …. 2^n
    - The deterministic version of a polynomial time
- non deterministic Turing machine will run in exponential time (worst case)
- Since it takes polynomial time to run the program, just run the program and get a solution
- But is NP a subset of P? It is not yet clear whether P = NP or not

Quantum Computers

- **Computers are physical objects, and computations are physical processes. What computers can or cannot compute is determined by the law of physics alone, and not by pure mathematics. Computation with coherent atomic-scale dynamics.**
  The behavior of a quantum computer is governed by the laws of quantum mechanics.

- In 1982 Richard Feynmann, a Nobel laurite in physics suggested to build computer based on quantum mechanics.
- Quantum mechanics arose in the early 1920s, when classical physics could not explain everything.
- QM will provide tools to fill up the gulf between the small and the relatively complex systems in physics.
- Bit (0 or 1) is the fundamental concept of classical computation and information. Classical computer built from electronic circuits containing wires and gates.
- Quantum bit and quantum circuits which are analogous to bits and circuits. Two possible states of a qubit (Dirac)are $|0\rangle$ $|1\rangle$
- Quantum bit is qubit described mathematically (where (alpha) is complex number)

$$\alpha_0|0\rangle + \alpha_1|1\rangle$$

- Qubit can be in infinite number of state other than dirac |0> or |1>

- The operations are induced by the apparatus *linearly*, that is, if

$$|0\rangle \to \frac{i}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle \qquad |1\rangle \to \frac{1}{\sqrt{2}}|0\rangle + \frac{i}{\sqrt{2}}|1\rangle$$

Then

$$\alpha_0|0\rangle+\alpha_1|1\rangle \to \alpha_0\left(\frac{i}{\sqrt{2}}|0\rangle+\frac{1}{\sqrt{2}}|1\rangle\right)+\alpha_1\left(\frac{1}{\sqrt{2}}|0\rangle+\frac{i}{\sqrt{2}}|1\rangle\right) \quad =\left(\alpha_0\frac{i}{\sqrt{2}}+\alpha_1\frac{1}{\sqrt{2}}\right)|0\rangle+\left(\alpha_0\frac{1}{\sqrt{2}}+\alpha_1\frac{i}{\sqrt{2}}\right)|1\rangle$$

Any linear operation that takes states $\alpha_0|0\rangle+\alpha_1|1\rangle$ satisfying and maps them to be UNITARY

i.e. $|\alpha_0|^2+|\alpha_1|^2=1$

Linear Algebra: $|0\rangle$ Corresponds to $\begin{pmatrix}1\\0\end{pmatrix}$

$|1\rangle$ Corresponds to $\begin{pmatrix}0\\1\end{pmatrix}$

$\alpha_0|0\rangle+\alpha_1|1\rangle$ Corresponds to $\alpha_0\begin{pmatrix}1\\0\end{pmatrix}+\alpha_1\begin{pmatrix}0\\1\end{pmatrix}=\begin{pmatrix}\alpha_0\\\alpha_1\end{pmatrix}$

If we concatenate two qubits

$(\alpha_0|0\rangle+\alpha_1|1\rangle)$ $(\beta_0|0\rangle+\beta_1|1\rangle)$

we have a 2-qubit system with 4 basis states

$|0\rangle|0\rangle=|00\rangle$ $|0\rangle|1\rangle=|01\rangle$ $|1\rangle|0\rangle=|10\rangle$ $|1\rangle|1\rangle=|11\rangle$

And it is describes the state as $\alpha_0\beta_0|00\rangle+\alpha_0\beta_1|01\rangle+\alpha_1\beta_0|10\rangle+\alpha_1\beta_1|11\rangle$

- Quantum computer is a system built from quantum circuits, containing wires and elementary quantum gates, to carry out manipulation of quantum information.

Variants of Turing Machines

Various types of TM are

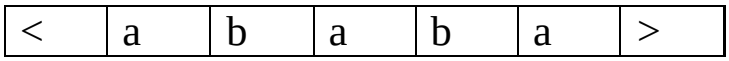
1. **With Multiple tapes**
2. **With one tape but multiple heads**
3. **With two dimensional tapes**
4. **Non deterministic TM**

1. **Multiple tapes: It consists of finite control with k tape heads and k tapes each tape is infinite in both directions. On a single move depending on the state of the finite control and symbol scanned by each of the tape head the machine can change state Or print new symbol on each of cell scanned etc..**

2. **With One tape but Multiple heads: a K head TM has fixed k number of heads and move of TM depends on the state and the symbol scanned by each head. (head can move left, right or stationary).**
3. **Multidimensional TM: It has finite control but the tape consists of a K-dimensional array of cells infinite in all 2 k directions. Depending on the state and symbol scanned , the device changes the state, prints a new symbol, and moves its tape head in one of the 2 k directions, either positively of negatively along one of the k axes.**
4. **Non deterministic TM: In the TM for a given state and tape symbol scanned by the tape head, the machine has a finite number of choices for the next move. Each choice consists of new state, a tape symbol to print and direction of head motion.**

Linear Bounded Automata

LBA is a restricted form of a Non deterministic Turing machine. It is a multitrack turing machine which has only one tape and this tape is exactly same length as that of input. It accepts the string in the similar manner as that of TM. For LBA halting means accepting. In LBA computation is restricted to an area bounded by length of the input. This is very much similar to programming environment where size of the variable is bounded by its data type. Lba is 7-tuple on Deterministic TM with



Left EM **Right EM**

$$M= (Q,\sum,\Gamma, Delta, q_{accept}, q_{reject}, q0)$$

- Two extra symbols < and > are used left end marker and right end marker.
- Input lies between these markers