Finite Automata and Regular Languages

FUNDAMENTALS

Alphabet: alphabet is a finite non-empty set of symbols
Σ denotes an alphabet

String:

- a string over an alphabet ‘A’ is a finite ordered sequence of symbols from ‘A’. The length of the string is the number of symbols in string with repetitions counted
- an empty string denoted by ‘ε’, is the (unique) string of length zero.
- If S and T are sets of strings then ST = {xy | x є S and y є T }
- Given an alphabet A,
    - $A^0 = \{ \varepsilon \}$
    - $A^{n+1} = A.A^n$
    - . . .
    - $A^* = U(n = 0 \text{ to infinity}) A^n$

LANGUAGES

- A language ‘L’ over Σ is any finite or infinite set of strings over Σ.
- The elements in L are strings – finite sequences of symbols
- A language which does not contain any elements is called an ‘empty language’
- Empty language resembles empty set => { } = Φ ≠ ε
- A language L over an alphabet A is subset of A^* i.e. L (A^*
- An empty language is a language that does not accept any strings including ε. (->O)
- A language which only accepts ε (->OO (concentric))

OPERATIONS

Operations on strings

1. **Concatenation**: Combines two strings by putting one after the other (a.b)
    - Concatenation of empty string with any other string gives the string itself

2. **Substring:** If ‘w’ is a string, then ‘v’ is a substring of ‘w’ if there exists string x and y such that w = xvy. ‘x’ is called the prefix and ‘y’ is called the suffix of w.

3. **Reversal:** if ‘w’ is a string, then w^R is reversal of string spelled backwards.
    - $x=(x^R)^R$
    - $(xz)^R = z^R.x^R$

4. **Kleen star operation:** Let ‘w’ be a string w^* is set of strings obtained by applying any number of concatenations of w with itself, including empty string.
    - Example: $a^* = \{\varepsilon, a, aa, aaa, ...\}$

Operations on Langauges

1. **Union:** Given some alphabet Σ, for any two languages, L_1, L_2 oven Σ, the union L_1 union L_2 of L_1 and L_2 is the language L_1 union L_2 = {w є Σ^* | w є L_1 or w є L_2 }

2. **Intersection:** Given some alphabet Σ, for any two languages, L_1, L_2 oven Σ, the intersection $L_1 \cap L_2$ of L_1 and L_2 is the language $L_1 \cap L_2 = \{w \in \Sigma^* \mid w \in L_1 \text{ and } w \in L_2\}$
3. **Difference / Relative Complement:** Given some alphabet Σ, for any two languages, L_1, L_2 oven Σ, the differnece $L_1 - L_2$ of L_1 and L_2 is the language $L_1 - L_2 = \{w \in \Sigma^* \mid w \in L_1$ and w does not belong to $L_2\}$
4. **Concatenation:** Given some alphabet Σ, for any two languages, L_1, L_2 oven Σ, the concatenation $L_1\ L_2$ of L_1 and L_2 is the language $L_1 L_2 = \{w \in \Sigma^* \mid \exists\ u \in L_1, \exists\ v \in L_2 \text{ and } w = uv\}$
    - **Properties**
    - $L\emptyset = \emptyset = \emptyset L$
    - $L\{\varepsilon\} = L = \{\varepsilon\} L$
    - $(L_1 \cup \{\varepsilon\}) L_2 = L_1L_2 \cup L_2$
    - $L_1(L_2 \cup \{\varepsilon\}) = L_1L_2 \cup L_1$
    - $L^nL = LL^n = L^{n+1}$
        - $L_1L_2 \neq L_2L_1$
5. **Kleen * Closure (L^*)**: Given an alphabet Σ, for any language L over Σ, the * closure L^* of L is language, $L^* = \bigcup_{n>=0} L^n$
6. **Kleen + Closure (L^+)**: Given an alphabet Σ, for any language L over Σ, the kleen $^+$ closure L^+ of L is language, $L+ = \bigcup_{n>=1} L^n$

    Properties
    - $\emptyset^* = \{\varepsilon\}$
    - $L^+ = L^*L$
    - $(L^*)^* = L^*$
    - $L^*L^* = L^*$

FINITE STATE MACHINES

FSM is the simplest computational model of limited memory computers. It is designed to solve decision problems, i.e, to decide whether the given input satisfies certain conditions. The next state and output of FSM is a function of input and the current state.

Types of FSM

- Mealy machine
- Moore machine

Finite Automata

- FA is a state machine that comprehensively captures all possible states and transitions that a machine can take while responding to a stream (sequence) of input symbols.
- FA is recognizer of ‘regular languages’

State Machine

- Finite state machines
    - Mealy machine
    - Moore machine
- Finite automata
    - DFA
    - NFA

- epsilon – NFA

Types of Finite Automata

1. **Deterministic Finite Automata**
   - DFA can exist in only one state at a time
   - DFA is defined by 5-tuple : {Q, Σ, q_0, F, δ}
     - Q → Finite number of states (elements)
     - Σ → Finite set of symbols (alphabets)
     - q_o →Start/Initial state
     - F → Set of final states
     - δ → Transition function, which is a mapping between
       - $\delta: Q \times \Sigma \rightarrow Q$
   - How to use DFA

Transition Diagram
State machines are represented by directed graphs called transition (state) diagrams.
Vertices : states
Arcs : transition
Double concentric circles: Final states

Transition Table
Transition function can be represented by tables.
Note: minimum number of states for k-divisibility is k-states

2. Non-deterministic Finite Automata

- The machine can exist in multiple states at the same time
- Each transition function maps to a set of states
- NFA is defined by 5-tuple: {Q, Σ, q_0, F, δ}
  - Q → Finite number of states (elements)
  - Σ → Finite set of symbols (alphabets)
  - q_o →Start/Initial state
  - F → Set of final states
  - δ → Transition function, which is a mapping between
    - $\delta: Q \times \Sigma \rightarrow 2^Q$
- How to use NFA

Difference between NFA and DFA

DFA	NFA
1. All transitions are deterministic i.e., each transition leads to exactly one state	1. Transitions could be non-deterministic i.e., a transition could lead to a subset of states
2. For each state, the transition on all possible	2. For each state, not all symbols necessarily

symbols should be defined	have to be defined
3. Accepts input if the last state is in F	3. Accepts input if one of the last states is in F.
4. Practical implementation is feasible	4. Practical implementation has to be deterministic. It needs conversion from NFA to DFA

Relation between DFA and NFA

1. A language L is accepted by a DFA if and only of it is accepted by NFA
2. DFA is a special case of NFA
3. Every language accepted by NFA is also accepted by a DFA. $D_f = N_f$

NFA WITH Є – MOVES

- Є-transitions in finite automata allows a state to jump to another state without consuming any input symbol

Conversion and equivalence:
Є – NFA → NFA → DFA

NFA without Є – moves:

- Two FA, $N_Є$ and N are said to be equivalent, if $L(N_Є) = L(N)$ i.e., any language described by some $N_Є$ there is a N that accepts the same language.
- For $N_Є = \{Q, Σ, q_0, F, δ\}$ and N = {Q, Σ`, $q_0$, F`, δ`}, find
- δ`(q, a) = Є – closure (δ (Є-closure(q), a))
- F` = {F U (q_0)}, if Є – closure (q_0) contains a member of F=F, otherwise
- *Note: when transforming Ne to N, only transitions are required to be changes and states remains same.*

CONVERSION OF NFA TO DFA

Let NFA be defined as $N = \{Q_N, Σ, q_0, F_N, δ_N\}$
The equivalent DFA, $D = \{Q_D, Σ, q_0, F_D, δ_D\}$

Step 1: $Q_D = 2^{QN}$, i.e., Q_D is set of all subsets of Q, i.e., it is power set of Q_N
Step 2: F_D is set of subsets S of Q_N such that $S ∩ F_N \neq Ø$, i.e., F_D is all sets of N's states that include once accepting state of N.
Step 3: For each set, S <= Q_N and for eacg input symbol a in
Σ : $δ_D(S,a) = U_{P ∈ S}\ δ_N(P, a)$
That is to compute $δ_D(S, a)$, look at all states P in S, see what states N goes to starting from P on input a, and take union of all those states.

Note: For any NFA, N with 'n' states, the corresponding DFA can have 2^n states.

MINIMIZATION OF DFA

Minimization of DFA means that we are minimizing the representation of a FA to it's least form. Here basically we replace multiple states with a single state without disturbing the representation. In a DFA states p and q are equivalent when:

δ(p, w) Є F => δ(q, w) Є F and δ(p, w) not Є F => δ(q, w) not Є F
if len(w) = 0 and p and q follow the above property: q and p are 0-equivalent
...
if len(w) = n and p and q follow the above property: p and q are n-equivalent
Therefore, instead of having two states, we can combine them into one state and decrese the number of states in the final answer.

We are using Partitioning Method here.

Step1: Identify the start and final state
Step2: If there is any state which is unreachable from the initial state, delete it.
Step3: draw the state transition table
Step4: Find the 0-equivalent sets. Separate non-final states from final states
Step5: Find the 1-equivalent sets. Separate the non equivalent from equivalent
Step5: Keep finding the n-equivalent sets until the next equivalent calculation is identical to the previous one.
Remove the dead states

EQUIVALENCE BETWEEN NFA AND DFA

There is a DFA for any NDA, i.e.,
L(D) = L(N).

Construction:

- In DFA or NFA, whenever an arrow is followed, there is a set of possible states. This set of states is a subset of Q.
- Track the information about subsets of states that can be reached from the initial state after following arrows.
- Consider each subset of states of NFA as a state of DFA nad every subset of states containing a final state as a final state of DFA.

Equivalence of Finite Automata

- Two automata A and B are said to be equivalent if both accept exactly the same set of input strings
- If two automata M_1 and M_2 are equivalent then
    - If there is a path from the start state of M_1 to a final state of M_1 labeled as a_1, a_2 ... a_k then there is a path from the start state of M_2 to the final state of M_2 labeled as a_1, a_2, ... a_k
    - If there is a path from the start state of M_2 to a final state of M_1 labeled as b_1, b_2 ... b_k then there is a path from the start state of M_1 to the final state of M_2 labeled as b_1, b_2, ... b_k

Union: The union of languages L and M is the set of strings that are in both L and M
Concatenation: The concatenation of languages L and M is the set of strings that can be formed by taking any string in L nad concatenating it with any string in M.

Closure, star or Kleen star of Language L:
Kleen star is denoted as L^*. It represents the set of strings that can be formed by taking any number of strings from L with repetition and concatenating them. It is a Unary operator.
L_0 is the set; we can make selecting zero strings from L.
L_0 = {epsilon}

L_1 is the language consisting of selecting one string from L.
L_2 is the language consisting of concatenations selecting two strings from L.
...
L^* is the union of L_0, L_1, ... Linf
Ex: L = {0, 10}
L^* = {0, 00, 000, 10, 010, ...}

Intersection:
Let two DFAs M_1 and M_2 accept the languages L_1 and L_2.
$M_1 = \{Q_1, \Sigma, q_0^1, F_1, \delta_1\}$
$M_2 = \{Q_2, \Sigma, q_0^2, F_2, \delta_2\}$
The intersection of M_1 and M_2 can be given as
$M = \{Q, \Sigma, q_0, F, \delta\}$
Q = Pairs of states, one from M_1 and one from M_2 i.e.,
$Q = \{(q_1, q_2) \mid q_1 \in Q_1$ and $q_2 \in Q_2\}$
$Q = Q_1 \text{ x } Q_2$
$q_0 = (q_0^1, q_0^2)$
$\delta((q_i^1, q_j^2), x) = (\delta_1(q_i^1, x), \delta_2(q_j^2, x))$
$F = \{(q_1, q_2) \mid q_1 \in F$ and $q_2 \in F\}$

Union:
Let two DFAs M_1 and M_2 accept the languages L_1 and L_2.
$M_1 = \{Q_1, \Sigma, q_0^1, F_1, \delta_1\}$
$M_2 = \{Q_2, \Sigma, q_0^2, F_2, \delta_2\}$
The union of M_1 and M_2 can be given as
$M = \{Q, \Sigma, q_0, F, \delta\}$
Q = Pairs of states, one from M_1 and one from M_2 i.e.,
$Q = \{(q_1, q_2) \mid q_1 \in Q_1$ and $q_2 \in Q_2\}$
$Q = Q_1 \text{ x } Q_2$
$q_0 = (q_0^1, q_0^2)$
$\delta((q_i^1, q_j^2), x) = (\delta_1(q_i^1, x), \delta_2(q_j^2, x))$
$F = \{(q_1, q_2) \mid q_1 \in F$ or $q_2 \in F\}$

Difference:
The difference of L1 and L2 can be given as:
L1 – L2 with $M = \{Q, \Sigma, q_0, F, \delta\}$
$Q = Q_1 \text{ x } Q_2$
$\delta((q_i^1, q_j^2), x) = (\delta_1(q_i^1, x), \delta_2(q_j^2, x))$
$F = \{(q_1, q_2) \mid q_1 \in F$ and q_2 does not $\in F\}$

Reversing a DFA:

- M is a DFA which recognizes the language L
- M^R will accept the language L^R
- To construct M^R
  - Reverse all transitions
  - Turn the start state to the final state
  - Turn the final states to the start state
  - Merge states and modify the FA such that the resultant contains a single start state.

MEALY AND MOORE MACHINES

Moore Machine

A Moore machine is a finite state machine, where outputs are determined by current state alone. A moore machine associates an output symbol with each state and each time a state is entered, an output is obtained simultaneously. So, first output always occurs as soon as the machine starts.

Moore machine is defined by 6-tuple:
$\{Q_1, \Sigma, \delta_1, q_0, \Delta, \lambda\}$

- Q → Finite number of states
- Σ → Finite set of input symbol
- Δ →It is an output alphabet
- δ → Transition function, Q x Σ →Q (state function)
- λ → Output function, Q→Δ (machine function)
- q_o →Initial state of machine

Note: The output symbol at a given time depends only on present state of moore machine.

Mealy Machine

- A Mealy machine is a FSM, where outputs are determined by current state and input.
- It associates an output symbol with each transition and the output depends on current input
- Mealy machine is defined as 6-tuple $\{Q_1, \Sigma, \delta_1, q_0, \Delta, \lambda\}$ where,
    - Q → Finite number of states
    - Σ → Finite set of input symbol
    - Δ →It is an output alphabet (finite set of output symbols)
    - δ → Transition function, Q x Σ →Q (state function)
    - λ → Output function, Q→Δ (machine function)
    - q_o →Initial state of machine ($q_0 \in Q$)

Note: In Moore machine, for input string of length n, the output sequence consists of (n+1) symbols
In Mealy machine, for input sequence of length n, the output sequence also consists of 'n' symbols

Equivalence of Moore and Mealy machine

(a) Mealy machine equivalent to Moore Machine

If M1 = $\{Q, \Sigma, \delta, q_0, \Delta, \lambda\}$ is a Moore machine, then there is a Mealy machine equivalent to M1.

Proof: Let M2 = $\{Q_1, \Sigma, \delta, q_0, \Delta, \lambda^1\}$ and define $\lambda^1(q, a)$ to be $\lambda(\delta(q, a))$ for all states q and input symbol 'a'. Then, M1 and M2 enter the same sequence of state on the same input, and with each transition M2 emits the o/p that M1 associates with the state entered.

Steps:
1. Always start with the initial state
Note: during the conversion of mealy to moore, the number of states may increase. If mealy machine has N states and m outputs, the number of states in the equivalent moore machine may become Nxm

(b) Moore machine equivalent to Mealy machine
The number of states in the conversion of moore machine to mealy machine will remain same.

REGULAR LANGUAGES

Introduction

The examples that we have seen; set of all binary numbers divisible by 2, 3 etc are nothing bur regular languages.

FA are ***acceptors*** of regular languages. Theyare some mathematical represntation to which if you give a string, they will accept it or not. FA (with and without output) are acceptors of regular languages. Acceptors accept “strings” from a language and tell us if the string is accepted or not.

The ***generators*** of regular languages are regular grammars. They generate the language. They are of two types (Right Linear Grammar and Left Linear Grammar).

A ***representator*** is a mathematical expression that represents a Regular Langauge. The RL are mathematically representated by expressions called Regular Expressions.

Let’s talk about RE. They are the representations of the languages that are exactly accepted by the FA. Or if we take any language which is accepted by FA, then we can represent it using Regular expressions.

Operations on Regular Expressions
1. Union (+)
2. Concatenation (●)
3. Kleene Closure (*)

Examples:

Regular Expression	Regular Language
Ø	{ }
ε	{ε}
a	{a}
a*	{ε, a, aa, aaa, ...}
a^+	a.a*/a*.a/{a, aa, aaa, ...}
(a+b)*	{ε, a, b, aa, ab, ba, bb, aaa, aab, aba, ...} (set of all strings possible over (a,b))

Points to remember:
1. If the language is finite, there is a finite automata and regular expression that accept and represent that lanaguage.

Identities of RE

1. Ø + R = R + Ø = R
2. Ø.R = R.Ø = Ø
3. ε.R = R.ε = R
4. ε* = ε
5. Ø * = ε
6. ε + RR* = R*R + ε = R*
7. (a + b)* = (a* + b*)*
   = (a*b*)*
   = (a* +b)* = (a+b*)* = a*(ba*)* = b*(ab*)*

The set of regular languages over an alphabet Σ is defined recursively as below. Any language belonging to this set is a regular language over Σ.

Definition of set of regular languages

- **Basis clause**: Ø, {ε}, {a} for any symbol a Є Σ, are regular languages.
- **Inductive clause**: If L_r and L_s are regular language, then L_r U L_s, L_r ● L_s, L_r^* are regular languages.
- **External clause**: Nothing is a regular language, unless it is obtained from above two clauses.

Regular Language: Any language represented by regular expression(s) is called a regular language.
Ex: the regular expression a^* denotes a language which has {ε, a, aa, aaa, ...}

Regular Expression

- Regular expressions are used to denote regular languages
- The set of regular expressions over an alphabet Σ is defined recursively as below. Any element of that set is a regular expression.
- **Basis clause**: Ø, ε, a are regular expressions corresponding to Ø, {ε}, {a} respectively where a is an element of Σ.
- **Inductive clause:** If r and s are regular expressins corresponding to languages L_r and L_s then (r+s), (rs) and (r^*) are regular expressions corresponding to the languages L_r U L_s, L_r ● L_s, L_r^* respectively.
- **External clause:** Nothing is a regular expression unless it is obtained from the above two clauses.

Closure property of RE: The iteration or closure of a regular expression R, written as R^* is also a regular expression.

CONVERSION OF RE TO FA: Just look at the RE given to you and bitch just make that FA.

CONVERSION OF FA TO RE: The methods used is state elimination method.
Step 1: The initial state should not have any incoming edge. If it is there, create a new initial state.
Step 2: Final state should not have outgoing edges. If it is there, create a new final state. There should be only one final state
Step 3: Eliminate the states other than initial and final states.

IS THE LANGUAGE REGULAR OR NOT?
To check if the language is regular or not.

1. If langauge is finite, it is a regular language
2. If the language is infinite, the FA should have a loop and inside the loop there should be a pattern. If there is no such pattern then no FA will be possible tpo recognize that language. This is what pumping lemma says.
If an infinite language has to be accepted by a finite automata then there should be a loop inside the finite automata. Therefore Pumping Lemma is a negative test. This means that pumping lemma says that the languge is not regular if you don't find a pattern but it doesn't say that the language IS regular if you find a pattern. Pumping lemma doesnt gaurantee that. Therefore, if you fail the pumping lemma, the language is NOT REGULAR but if you pass it you can't say IT IS REGULAR.

To find if a given language is regular or not, make a RE or a FA out of the language, if it is possible to get, language is regular. In some cases you may need to find that the language does not take infinite memory.

CLOSURE PROPERTIES OF REGULAR SETS

1. **Union:** If L and M are regular languages, LUM is regular language closed under union.
2. **Concatenation and Kleene Closure**: If L and M are regular languages, L.M is regular language and L* is also regular.
3. **Intersection:** L intersection M is regular, if L and M are regular languages
4. **Difference**: L – M contains strings in L but not in M, wgere L and M are regular languages
5. **Complementation**: The complement of language L is Σ* - L.
Since Σ* is surely regular, the complement of a regular langugae is always regular. Where Σ* is a universal language.
6. **Homomorphism:** If L is a regular language, h is homomorphism on its alphabet then h(L) = {h(w) | w is in L} is also a regular langugae.

GRAMMAR

A grammar is generally represented by {V, T, P, S} where
V = set of all vertices
T = set of all terminals
P = set of all productions
S = start symbol

Ex. A grammar is given as:
S → aSB
S → aB
B → b

for this V = {S, B}, T = {a, b}, P = {production formulas given above}, S = start symbol. So if this is the grammar, let's find out what is the language generated by this grammar.
Getting a string from this grammar is called *derivation.*

S => aSB (sentential/sequential form)(in every step, only one variable will be substituted by the production formula.)
If you start replacing the leftmost symbol first, that is called leftmost derivation. If you start replacing the rightmost symbol first, it is called rightmost derivation.
Let's do the leftmost derivation first.

S => aaBB (sentential/sequential form)
S => aabB(sentential/sequential form)
S => aabb (derivation)

Derivation Tree: The yield of the parse/derivation tree is gonna be the string.

TYPES OF GRAMMARS
Chomsky gave four different grammars;
Type 3: Regular Grammars
A Regular grammar is the generator of regular languages. These regular languages are accepted by Fininte Automata. The Regular languages generated can be represented by Regular Expressions. The Type 3 or Regular Grammar is a Grammar of the form:

$A \rightarrow \alpha B / \beta$ (RIGHT LINEAR GRAMMAR)
$A, B \in V$
$\alpha, \beta \in T$

or

$A \rightarrow B\alpha / \beta$ (LEFT LINEAR GRAMMAR)
$A, B \in V$
$\alpha, \beta \in T$

To check whether the grammar is of Type 3: Give a look at the production formulas and check if all of the productions are either Right Linear or Left Linear.

Constructing Regular Grammar from FA
1. FA (L) →Right Linear Grammar (L) → Left Linear Grammar (L^R)
2. FA(L) → FA(L^R) → Right Linear Grammar (L^R) → Left Linear Grammar (L)

Constructing an FA from the Regular Grammar
1. Constructing FA from Right linear grammar is straight forward
2. To construct FA from a left linear grammar
LLG(L) → RLG (L^R) → FA (L^R) → FA $(L^R)^R$

Type 2: Context Free Grammar
The Type 2 or Context Free Grammar is of the form:

$A \rightarrow \alpha$ where $A \in V$ and $\alpha \in (V \cup T)^*$
If a grammar is regular, it is always context free. It is called a context free grammar because (eg.)

A → aAb / ab
If we derive: A →aAb
→aabb
I can replace the A in the first production with anything without worrying about the context in which the vertex is present! In context sensitive grammar, I'll have to look at the context of A (aA, Ab, etc.) eg. aA → aa

The class of languages generated by context free grammar is called context free languages. The Automata that is the acceptor of context free grammar is the Push Down Automata

Properties of Type 2 grammar: Context Free Grammar.

The classification of the Context Free Grammar:

1. Context Free Grammar: Ambiguous and non-ambiguous
2. Context Free Grammar: Deterministic and Non-Deterministic
3. Context Free Grammar: Left Recursive and Right Recursive

The entire CFG can be divided into the above mentioned categories. These categories are not mutually exclusive. A CFG can be deterministic and ambiguous or any other combination or combinations from the categories.

The entire discussion on CFG is done in Compiler design so Ravindra has skipped them here. Let's move on to the Properties of CFG.

Interesting problem: Given a string w, and CFG G. Does w belong to langauge generated by G?

Derive the grammar in 1 steps, 2 steps, 3 steps ... if the string can be generated in n-steps then this algorithm will stop after n-steps. If the string does not belong to the language generated by G, you may get into an infinite loop. We can solve this by modifying the Grammar.

If we eliminate the epsilon-productions and unit-productions then we'll always increase the string length with every step in the derivation. Therefore after reaching the number of sentential forms which are greater than the |w| then we can discard all the sentential forms produced until now and conclude that the Grammar does not produce a language that accepts the string 'w'.

In worst case we may have a grammar like this:

A → BC (production to a vertex increasing the length of the string)
A → a (terminal production)

In worst case the number of steps that we'll have to take to see if the string 'w' belongs to the language generated by the grammar G is 2|w|. First |w| for the number of productions to reach |w| length and then the putting of terminal will take |w| steps. (For a grammar which does not have unit or epsilon production). Let's see the algorithms.

In first step: P productions, or P sentential forms to examine
In second step: P^2 sentential forms, square because I have to replace the variables with P possible productions....
.
.
.
P^{2w}

So this algorithm which does not have any epsilon or unit productions is going to take $O(P^{2w+1})$ in order to find out the membership of a string w in a language generated by a given grammar.
$P + P^2 + P^3 + ... + P^{2w}$
So we're not going to be using some exponential algorithm.

Note: whatever algorithm you're going to use, make sure that your context free grammar does not have epsilon productions and unit productions and should not contain useless symbols.

One such algorithm is CYK algorithm. Using this you can find out the memebership of a string in a language generated by Grammar G in the order O(|w|^3). Before we go to the CYK algorithm, let's see hwo to eliminate the epsilon and unit productions from the grammar.

Elimination of ε – productions

Here we are going to eliminate the epsilon productions from the grammar. A question arises: Can we eliminate all the epsilon productions?

Ans: Is L(G) has epsilon in it, then NO. It needs to have atleast one production which generated epsilon. This means that if the language does not produce have ε in it, then you can eliminate all ε-productions from the grammar. Otherwise S -> ε should be a production.

Let's take an example.:

S -> aSb/aAb
A -> ε

Step1: Find out all the null productions (Here it is A -> ε)
Step 2: Find out all the nullable variables.

Nullable variables are:

if a variable can directly generate ε (A => ε) or
if a variable can after some derivations can generate ε like (A => ( ) => ( ) => ε)

That variable is called nullable.

In this example: A is nullable

Step 3: Go to the right hand side of every production and wherever that nullable variable is present, write that production with and without the nullable variable.

S -> aSb/aAb/ab
S -> ε (now you can eliminate this)

S -> aSb/aAb/ab . But there is no A variable involved, so final grammar becomes.

S -> aSb/ab

Example 2:

S -> AB
A -> aAA/ ε
B -> bBB / ε

Step 1: Nullables are {A, B, S}.

This means that the start symbol is generating ε. Therefore, ε belongs to L(G). In this language ε is present so all the productions may not be eliminated which have epsilon.

Step2: Let's rewrite the grammar with and without all the nullables

S -> AB/B/A/ ε
A -> aAA/aA/a
B -> bBB/bB/b

This is the language which we have derived without the epsilon productions. S will have an epsilon production because the language has an epsilon in it.

Example 3:

S -> AbaC
A -> BC
B -> b/ ε
C -> D/ ε
D -> d

Step 1: Nullables are {C, B, A}

Answer:
S ->AbaC/baC/Aba/ba
A ->BC/B/C
(here we are not going to include ε in place of BC because A is not a start symbol. If LHS would have been start symbol that means the ε is in the language).
B -> b
C -> D
D -> d

Now when you look at the examples above, eliminating ε productions are going to add unit productions. Now for the purpose of finding if string ‘w’ belongs to the language generated by the Grammar G, we need to count the meaningful production steps. For eg.
A -> B
B -> C
C -> d
(3 steps)

is equivalent to saying A -> d (1 step)
The length of the string or the number of terminals are also not increasing.

Elimination of unit – productions

Let’s see how to remove unit productions.
Example 1:
S -> Aa/B
B -> A/bb
A -> a/bc/B

Note:The language generated by the transformations on the Grammar should not affect the langauge generated by the grammar. Even if you delete the unit productions, the final language generated hould not be affected.

Step 1: Write the grammar without the unit productions

S -> Aa
B -> bb
A -> a/bc

Step 2: What are the unit productions, and what the elimination of them have effect, add the effect to the grammar without unit productions

a. S -> B in turn B ->bb (this be missed if I delete B in the language)
S -> B -> A ->a,bc
This implies I need to add bb, a, bc to S => Aa
Therefore S => Aa/bb/a/bc

b. B -> A
New B: B -> bb/a/bc

c. A -> B
New A: A -> a/bc/bb

The final grammar becomes

S -> Aa/bb/a/bc
B -> bb/a/bc
A -> a/bc/bb

Example 2:
S -> AB
A -> a
B -> C/b
C -> D
D -> E
E -> a

B -> C -> D -> E -> a => B could not reach 'a' if I delete it, B -> b/a
C -> D, could miss 'a' => C -> a is the final productions
D -> E could miss 'a' => D -> a is the final production

Now S -> AB => C, D and R are not reachable, USELESS SYMBOLS, so let's see how to remove them.

Elimination of useless symbols
Useful symbols:
1. A symbol 'A' in a grammar is useful if it is able to derive some terminal or string of terminals.
2. This symbol should be reachable from start state.

Test Derivability and Reachability to check for useless symbols.

Example 1:
S -> AB/a
A -> BC / b
B -> aB / C
C -> aC / B

Step 1: Useful symbols: {a, b, S, A}

Step 2: If any production is made of useful symbols, that becomes a useful symbol (left side). Here B and C are useless.

Step 3: Delete all the productions which have useless symbols as the LHS and also delete all symbols containing useless symbols on RHS.

Remaining:

S -> a
A -> b

Step 4: check reachability from S to symbols

S -> a

Example 2:
S -> AB/AC
A -> aAb/bAa/a
B -> bbA/aaB/AB
C -> abcA/aDb
D -> bD/aC

Step 1: Useful symbols : {a, b, A, B, S},

Step 2: included in step 1 here

Step 3: eliminate C and D productions, change S production

Final productions before testing reachability:

S-> AB
A -> aAb/bAa/a
B -> bbA/aaB/AB

A and B both are reachable from S so the final grammar becomes

S-> AB
A -> aAb/bAa/a
B -> bbA/aaB/AB

Example 3:

S -> ABC/BaB
A -> aA/BaC/aaa
B -> bBb/a
C -> CA/AC

Step 1: Useful symbols: {a, b, B, A, S}
Step 2: in step one included
Step 3: eliminate C as the production, change S and A

Final useful grammar becomes;

S -> BaB
A -> aA/aaa
B -> bBb/a

A is not reachable from S alone or through B, so removing A.

Final useful grammar becomes:

S** -> **BaB
B** -> **bBb/a

PUSH DOWN AUTOMATA

In order to accept Context Free Languages, we need a machine called Push Down Automata. PDA is nothing but a FA to which a memory element is added and that memory element is stack. A PDA is defined as a sept-tuple {Q, Σ, δ, q_0, Z_0, F, Γ } where.

Q: finite set of states
Σ : input alphabet
δ: Transition function

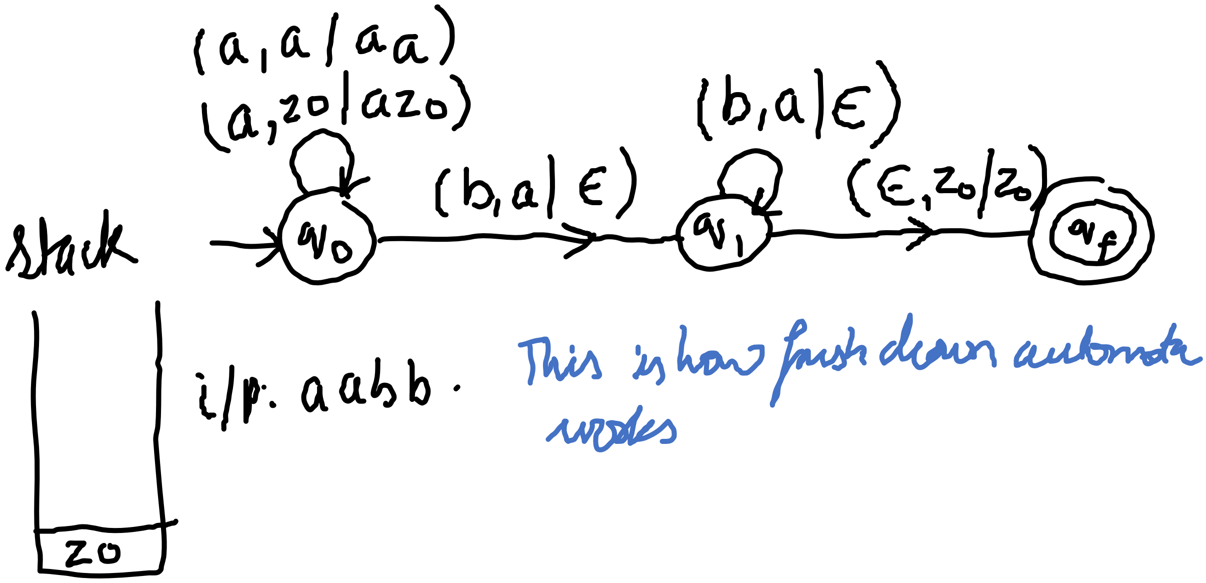
q_0 : Initial State
Z_0: Bottom of the stack
F : set of final states
Γ : stack alphabet

If the PDA is deterministic:
δ: Q x (Σ U ε) x Γ -> Q x Γ* (Automata is in state and sees a symbol or empty symbol and the top of the stack, then it goes to a state in Q and pushes a symbol in the stack)

If PDA is non-deterministic
δ: Q x (Σ U ε) x Γ -> $2^{Q\ x\ \Gamma^*}$ (This means if you are in some state and see an input or empty symbol and see the top of the stack, you decide to go to more than one state and push more than one symbol to the top of the stack, then it is non-determinism)

Let's take examples to understand.

Example 1: Let is say we have language $a^n b^n$ | n>=1
Here you have to see all a's first and then see all b's and count a against b.
If I have aabb, then initially z0 is in the stack. Push 'a' as you see in the input 'a' of the string and pop 'a' as you see input 'b' of the string. At the end when ε is reached and the top of stack is z0, then the string is accepted. Let's see the PDA for this.



This is how push down automata works

Another way to represent this PDA is using transition function.

Transition function based:

$\delta(q_{0}, a, z_0) = (q_{0}, az_0)$

$\delta(q_{0}, a, a) = (q_{0}, aa)$

$\delta(q_{0}, b, a) = (q_{1}, \varepsilon)$

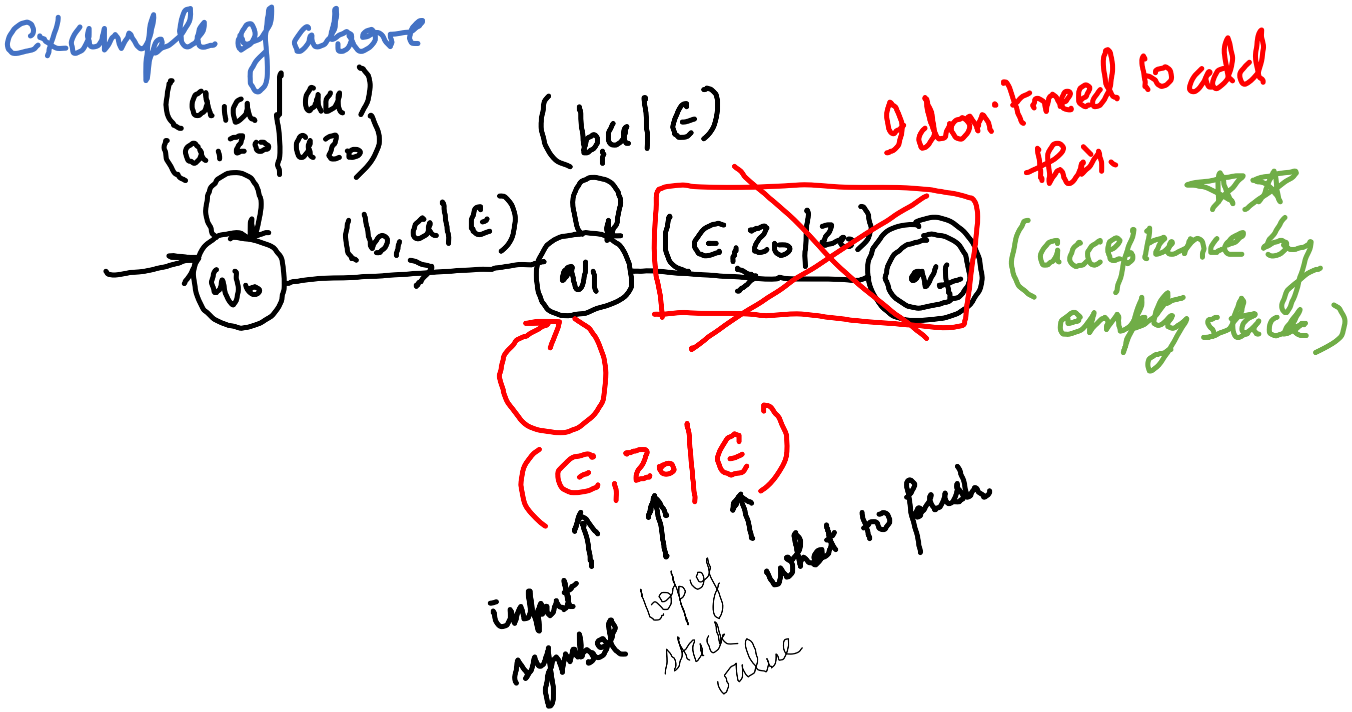
$\delta(q_{1}, b, a) = (q_{1}, \varepsilon)$

$\delta(\varepsilon, b, z_0) = (q_{f}, z_0)$

Push Down Automata acceptance can be of two types:

1. Acceptance by final state
2. Acceptance by empty stack

Both these PDAs are equivalent in power. Either deterministic or non-deterministic, both type of PDAs is equivalent in power. Let's see an example



example of above

I don't need to add this.

☆☆ (acceptance by empty stack)

input symbol

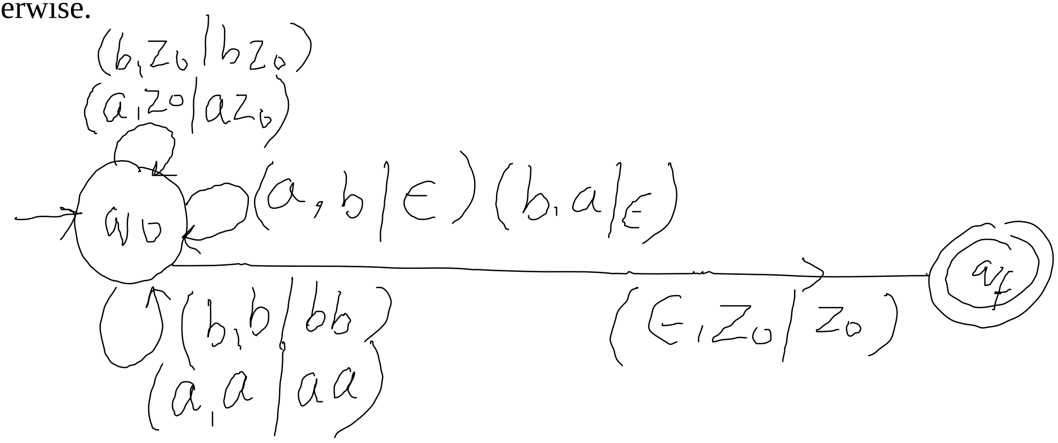
top of stack value

what to push

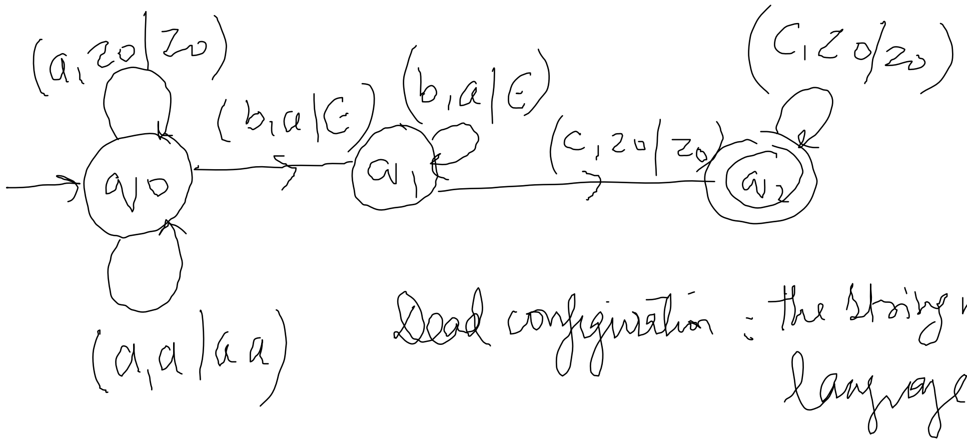
Example 2: L = {w | $n_a(w) = n_b(w)$}

This is like counting all a's and then match them against all b's.

Solution: push if top of stack is empty or the same symbol, pop the stack if the other symbol is seen otherwise.



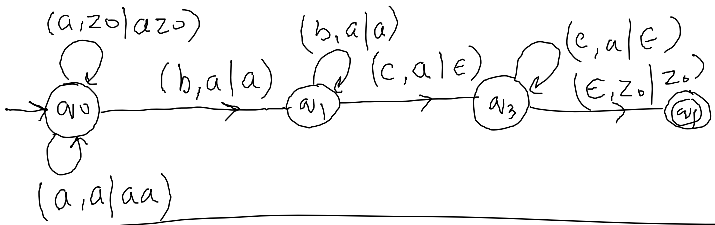
Example 3: L = {a^{n}b^{n}c^m | n,m >=1}
Here a and b should be of same number and c could be any other number. Compare a and b and then simply see c's



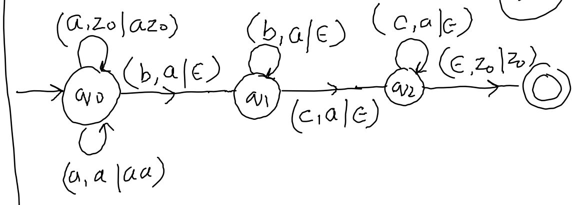
Dead configuration : the string not in language is seen

When the input is in the language, it will be accepted by the PDA. If the input is not seen in the language, then it will be considered as dead configuration by the PDA and the PDA will halt in that state. This is halting. This will make PDA stop in a non-finite state and it will not be accepted by PDA. If the input is not in the language, the PDA will definitely halt in one of the intermediate state. It will not go to the final state or go in an infinite loop. HALTING will be must.

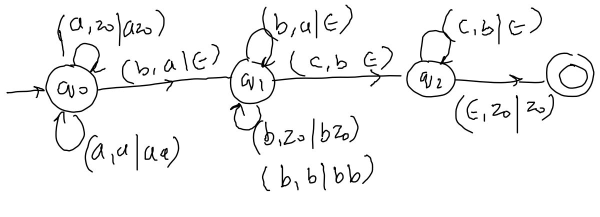
example: $\mathcal{L} = \{a^n b^m c^n \mid n, m \geq 1\}$ ✓



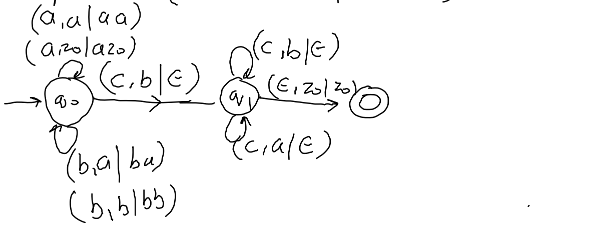
Example: $\mathcal{L} = \{a^{m+n} b^m c^n \mid m,n \geq 1\}$ ✓



Example: $\mathcal{L} = \{a^n b^{m+n} c^m \mid n,m \geq 1\}$



example: $\mathcal{L} = \{a^n b^m c^{n+m} \mid n,m \geq 1\}$



Some other examples are:

L = {a^{n}b^{n}c^{m}d^m | n, m >=1}
L = {a^{n}b^{m}c^{m}d^n | n, m >=1}
L = {a^{n}b^{m}c^{n}d^m | n, m >=1} (this is not a context free language because PDA will halt.

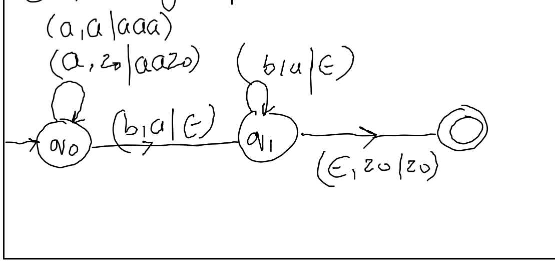
Another Example: L = {a^{n}b^{2n} | n >=1}

Here we can make a PDA in two ways.

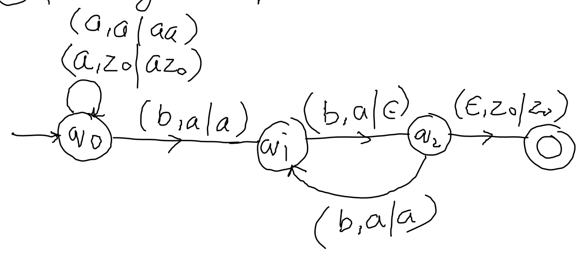
1. For every a push 2 a's and then match the number of b's
2. For every 2 b's pop one a

Let’s see both of them:

① for every a push 2 a's then match b



② for every 2 b's pop one a

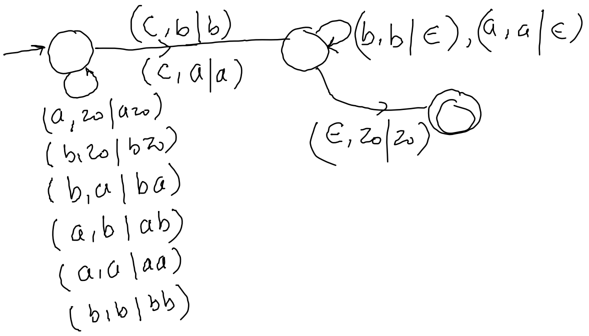


Another example of non- Context free language: L = {$a^nb^nc^n$ | n >=1}

Example: L = {wcw^R , w belongs to $\{a, b\}^+$}

we need to see w then centre and then compare w^R to w.

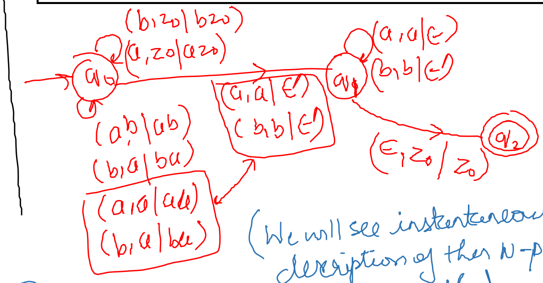
words can be: abcba, bacab, aacaa, bbcbb, etc



Example. $L = \{ww^R , w \in \{a, b\}^+\}$

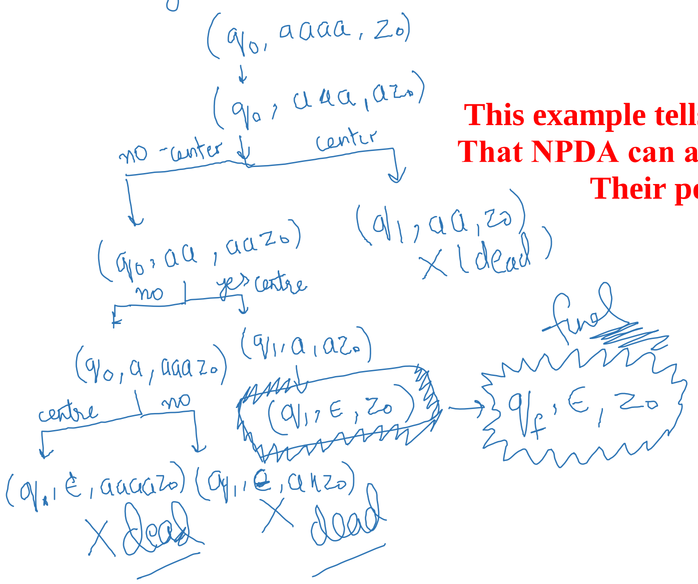
This is an example of ND-PDA

There is no central element. So one part of the PDA will assume that a given element is centre, other will not



(We will see instantaneous description of the N-PDA w/ an example)

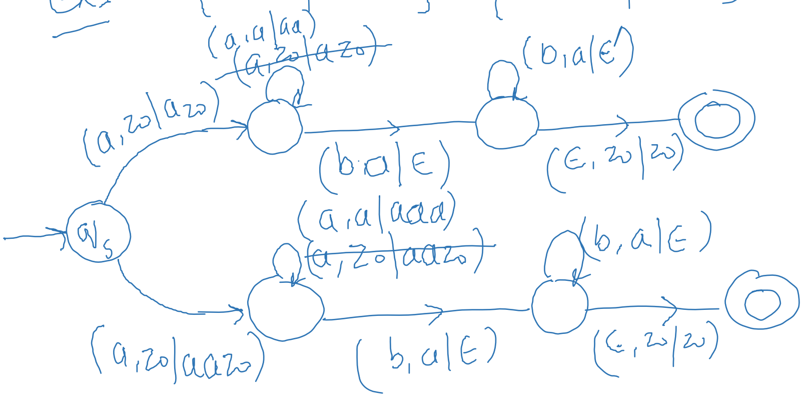
eg. aaaa (instantaneous description of the ND-PDA



This example tells us that there are languages That NPDA can accept which DPDA can’t. So, Their powers are different.

Let’s take one more example.

Ex: $L = \{a^n b^n \mid n \geq 1\} \cup \{a^n b^{2n} \mid n \geq 1\}$

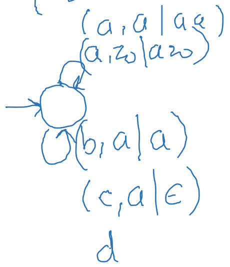


ex: $L = \{a^i b^j c^k d^l \mid i = k \text{ or } j = l\}$

let's segregate it:

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

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



Let’s see some examples of some languages and determine if they are regular or not or context free or not.

$a^{m+n} b^n c^m \mid m,n \geq 1$	not regular because of infinite storage & comparison. can give DPDA for it so D-CFL
$a^m b^{m+n} c^n \mid m,n \geq 1$	"
$a^m b^n c^{m+n} \mid m,n \geq 1$	"
$a^m b^m c^n d^n \mid m,n \geq 1$	"
$a^m b^n c^m d^n \mid m,n \geq 1$	no PDA for it, so not CFL.
$a^m b^n c^n d^m \mid m,n \geq 1$	CFL not RL because PDA can be made
$a^m b^i c^m d^k \mid m,n \geq 1$	yes PDA so CFL.
$a^m b^n \mid m > n$	Deterministic PDA is there, so D-CFL
$a^n b^{2n} \mid n \geq 1$	yes CFL
$a^n b^{n^2} \mid n \geq 1$	yes CFL (take eg. of $a^n b^{2n}$)
$a^n b^{2^n} \mid n \geq 1$	2^n not in AP so not possible PDA ∴ XCFL
$w w^R \mid w \in (a,b)^*$	yes PDA so CFL (it has NPDA so CFL)
$w w \mid w \in (a,b)^*$	not CFL because comparison w/ stack bottom not possible
$a^n b^n c^m \mid n > m$	m will not have anything to compare against. PDA not possible so not CFL
$a^n b^n c^n d^n \mid n < 10^{10}$	finite language, ∴ RL, DCFL & so CFL too
$a^n b^{2n} c^{3n} \mid n \geq 1$	not CFL.
$x c y \mid x,y \in (0,1)^*$	it is RL → DCFL ∴ CFL
$x x^R \mid x \in (a,b)^*, \lvert x \rvert = l$	2^l strings possible, finite, so RL & thus CFL

$www^R \mid w \in (a.b)^*$	not a context free lang.
$a^n b^{3^n} \mid n \geq 1$	not a context free language
$a^m b^n \mid m \neq n$	CFL ,PDA possible.
$a^m b^n \mid m = 2n+1$	CFL, m is AP: 1, 3, 5, 7 ...
$a^i b^i \mid i \neq 2j+1$	CF, check remaining symbol presence
$a^{n^2} \mid n \geq 1$	n^2 not in AP, so no CFL looks not possible
$a^{2^n} / n \geq 1$	X CFL (no AP)
$a^{n!} / n \geq 1$	X CFL (noAP)
a^m / m is a prime	X CFL (noAP)
a^k / k is even	RL, CFL
$a^i b^j c^k / i > j > k$	no CFL
$a^i b^j c^k / j = i+k$	CF, DCFL
$a^i b^j c^k d^l$ / $i=k$ or $j=l$	NDCFL
$a^i b^j c^k d^l$ / $i=k$ & $j=l$	XCFL
$a^m b^l c^k d^n / m, l, k, n \geq 1$	RL ∴ CFL
$a^n b^{4m} / n, m \geq 1$	RL ∴ CFL
$\{a^3, a^8, a^{13}, \dots\}$	AP; RL&CFL
$\{a^{2n+1} / n \geq 1\}$	RL, CF \
$\{a^{n^n} / n \geq 1\}$	X RL, X CFL
$\{w / w \in \{a,b\}^*, \|W\| \geq 100\}$	RL, CFL
$\{w \mid w \in \{a,b,c\}^*, n_a(w) = n_b(w) = n_c(w)\}$	no CFL
$\{w \mid w \in \{a,b\}, n_a(w) \geq n_b(w)+1$	CFL