classmate
Date
Page

Automata Theory :-

Problems are a set of problem instances. A problem instance is a specific instance of a problem. We predominantly deal with yes/no problems which have 2 sets, the accept/yes and the reject/no sets.

A problem is computable only if all the instances are computable/solvable by the model.

A computational model solves a problem P if (and only if ?)
i) ∀ inputs belonging to the YES instance of P, then the device outputs YES / ACCEPT
ii) ∀ inputs belonging to the NO instance of P, then the device outputs NO / REJECT

If (i) & (ii) hold, P is computable.

Alphabet:- Any finite, non-empty set of symbols

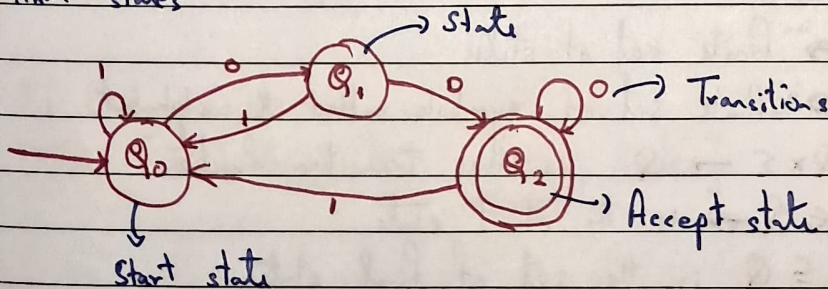
Strings / Words:- Finite sequence of symbols from an alphabet

Language:- Set of words/strings from the current alphabet

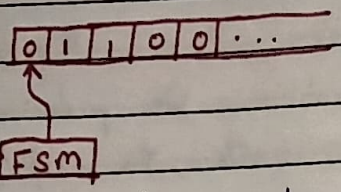
Generally, ε denotes the empty string

Deterministic Finite State Automata Model [DFA]

Has a single start state, unique transitions and zero or more final states



Input => $\Sigma = \{0, 1\}$



One way infinite Tape.

After the input is read entirely, and the machine is in the final/accept state, then it returns YES/ACCEPT; else it outputs NO/REJECT

Run:- The total computation done during the reading and execution of the entire input string. The string is accepted or rejected after the complete run.

Definition:- Let the DFA be M. Then the language that M accept is :-

$L(M) = \{w \mid w \text{ results in an accepting run}\}$

$\{w \mid \text{set of all strings } w \text{ such that } M(w) \text{ accepts}\}$

This is also known as the language of the automaton.

Definition:- For any language L, we say M solves/decides L if

i) $\forall\, w \in L$, $M(w)$ accepts

ii) $\forall\, w \notin L$, $M(w)$ rejects

Definition:- A DFA M is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where

Q is a finite set of state

Σ is a finite set of symbols called the alphabet

$\delta : Q \times \Sigma \to Q$ is the transition function

$q_0 \in Q$ is the start state

$F \subseteq Q$ is the set of final states

11
110
1001
1111

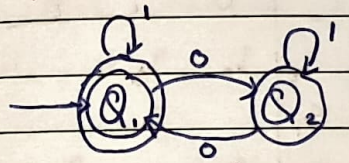
classmate
Date
Page

Characteristics of a DFA :-

(i) Single Start State
(ii) Unique Transitions
(iii) Zero or more final states

Language of a DFA is the set of all binary strings ~~ending in 00~~ that result in an Accept state at the end of the computation.

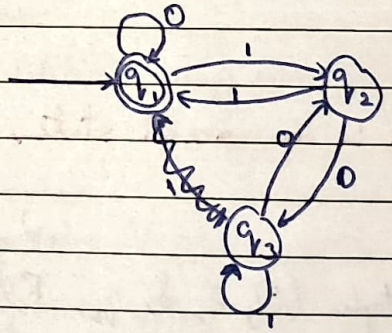
Examples:- (i) $\Sigma = \{0, 1\}$, $L(m) = \{\omega \mid \omega$ has an even no of 0s$\}$



(ii) $\Sigma = \{0, 1\}$, $L(m) = \{\omega \mid \omega$ is divisible by $3\}$

Any input string would leave remainders 0, 1, 2
DFA will have 3 states

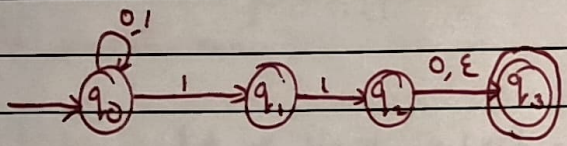
If ω is divisible by 3, the after reading a 0 it should stay in the final state, and so on.



If L is solvable by a DFA, then $\neg L$ is also solvable by a DFA. [Just toggle the accepting states]

Non-deterministic finite state Automata (NFA)

Characteristics:- (i) Single start state
(ii) Zero or more final states
(iii) Multiple transitions are possible on the same input state
(iv) Some transitions might be missing.
(v) ε-transitions or empty transitions exist which allow the NFA to transition without reading any input.



	0	1	ε
q_0	q_0	q_0, q_1	
q_1		q_2	
q_2	q_3		q_3
q_3			

A given input can have multiple runs.
Take 10110 as the input string. It could accept, reject or crash in this case.
Crashes occur when the machine has no transition for an input
For the 10110 example, a possible crash run is.
$q_0 \xrightarrow{1} q_0 \xrightarrow{0} q_0 \xrightarrow{1} q_1 \xrightarrow{1} q_2 \xrightarrow{\varepsilon} q_3 \xrightarrow{0}$ crash

Crashes are rejecting runs.
If $\exists$ at least one run results in an accept state, the input is accepted, else rejects

Definition :- An NFA is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where
$Q \Rightarrow$ finite set of states
$\Sigma \Rightarrow$ finite set of alphabets (contains ε)
$\delta \Rightarrow$: $Q \times \Sigma \longrightarrow P(Q)$, is the transition fn
$q_0 \in Q \Rightarrow$ start state
$F \subseteq Q \Rightarrow$ set of accepting states

$P(Q)$ → powerset of Q

Intuitively NFAs are more powerful than DFAs because of non-determinism.

The languages accepted by DFAs are a subset of the languages accepted by NFAs → L_2

$L_2 \subseteq L_1$

$L_1 \subseteq L_2$ (Proof upcoming) → L_1

Hence $L_1 = L_2$ → Both are equally powerful.

We prove this by showing that we can construct a DFA to accept the same languages as an NFA. These DFAs are called Remembering DFAs.

Steps :-

(i) Let R be a remembering DFA

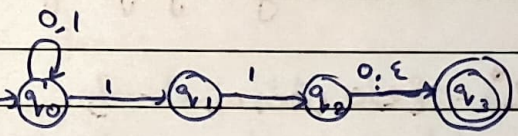
(ii) R on an input enters a state labelled by all possible states that the NFA can enter on that input

(iii) This trims away the non-determinism of the NFA without losing its language

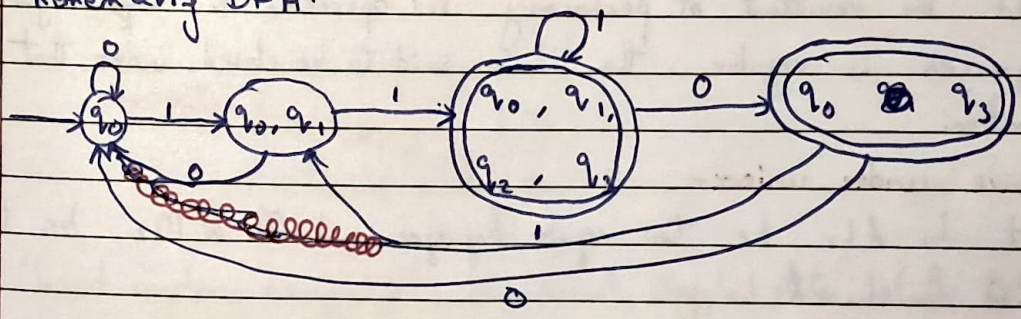
(iv) Also, if the NFA has k states, R has at most 2^k states [Power sets]

Example :-

NFA =>



Remembering DFA:



Unit-2 :-

Regular Languages:- A language is called regular if ∃ a finite state automaton that decides it.

Let M be a finite state automaton and,

$L(M) = \{w \mid w \text{ is accepted by } M\}$

then L(M) is regular.

Any language has a set of operations that can be performed on it associated with it.

Regular Operations :- Let L_1, L_2 be languages

Binary → i) Union :- $L_1 \cup L_2 = \{x \mid x \in L_1 \text{ or } x \in L_2\}$

↳ ii) Concatenation :- $L_1 \cdot L_2 = \{xy \mid x \in L_1, y \in L_2\}$

Unary → iii) Star :- $L_1^* = \{x_1 x_2 \ldots x_k \mid k \geq 0 \text{ and each } x_i \in L_1\}$

Countably ∞ ↙ ↳ concat any no. of strings from L_1

Ex :- $\Sigma = \{a\}$; $\Sigma^* = \{\varepsilon, a, aa, \ldots\}$

$\Sigma = \{\emptyset\}$; $\Sigma^* = \{\varepsilon\}$

ε belongs to the * of any language by convention

$L = \{0,1\} \rightarrow L^* = \{\varepsilon, 0, 1, 00, 01, \ldots\}$

Closure of Regular Languages

If the resultant of performing set operation on regular languages is regular, then it is said to be closed under that operation.

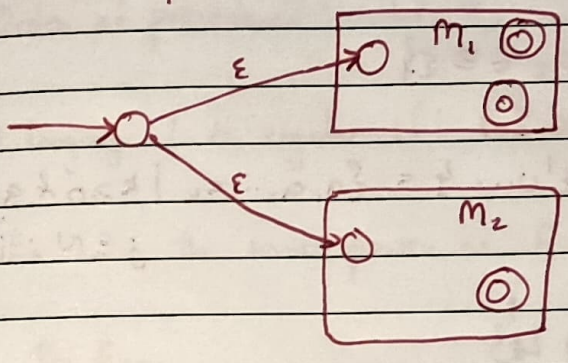
(i) **Closure under union:-**

Let L_1 & L_2 be two regular languages and M_1 & M_2 be the FSA for L_1 & L_2.

Construct an NFA by having a common start state with 2 ε-

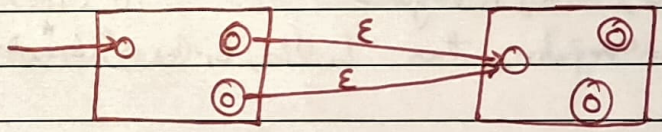
transitions to the start states of M_1 & M_2. This decides $L_1 \cup L_2$. Hence Union is closed

Math proof in slides [Lec-3]



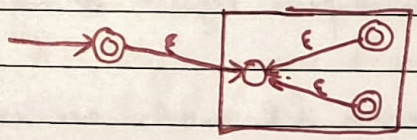
(ii) Closure under concatenation :-

Construct an NFA with start state as that of M_1 and ε transitions b/w final states of M_1 to the start state of M_2.



(iii) Closure under star :-

Construct a new start state which is an accepting state, have an ε transition to the old start state, and have ε-loop backs from the final states to the old start state



(iv) Closed under complement :- [Just toggle states of the DFA]
Not for NFA ↵

v) Closure under intersection:- Yes, proof using De Morgan's Law with Union and Complement.

Notation:- Σ is an alphabet

i) $\Sigma^0 = \{\varepsilon\}$

ii) $\Sigma^2 = \{a_1 a_2 \mid a_1, a_2 \in \Sigma\}$

iii) $\Sigma^k = \{a_1 \cdots a_k \mid a_i \in \Sigma\}$

iv) $\Sigma^* = \bigcup_{i \geq 0} \Sigma^i = \{\Sigma^0 \cup \Sigma^1 \cup \cdots\} = \{a_1 a_2 \cdots a_k \mid k \geq 0 \text{ \& } a_j \in \Sigma \; \forall j \in N, j \leq k\}$

$L \subseteq \Sigma^*$ and $L^* = \{\bigcup_{i \geq 0} L^i\}$

First Principles:- [Ab initio]

Re Lang alt def:- Let Σ be an alphabet. Then the following are Re langs over Σ

i) ϕ is regular

ii) for each $a \in \Sigma$, $\{a\}$ is regular

iii) If L_1 & L_2 are regular, then $L_1 \cup L_2$, $L_1 \cdot L_2$, L_1^*, L_2^* are regular.

Syntax for Regular Expressions:- R is said to be a regular expression if it has one of the following forms:-

i) ϕ is a regular expression. $L(\phi) = \phi$

ii) ε is a regular expression. $L(\varepsilon) = \{\varepsilon\}$

iii) Any $a \in \Sigma$ is a regular expression, $L(a) = \{a\}$

Regular Expressions are ways to express regular languages

iv) $R_1 + R_2$ is a regular expression if R_1 and R_2 are regular expressions, $L(R_1 + R_2) = L(R_1) \cup L(R_2)$

v) R^* is a regular expression if R is a regular expression, $L(R^*) = (L(R))^*$

vi) $R_1 R_2$ is a regular expression if R_1, R_2 are regular expressions. $L(R_1 \cdot R_2) = L(R_1) \cdot L(R_2)$

Regular Expressions are like the alphabet of the Regular Language

vii) (R) is a regular expression if R is a regular expression. L((R)) = R

Order of precedence :- (), *, ., +

A language L is regular iff for some regular expression R, L(R) = L

REs have the same power as FSA:-

Reg Expr	L(R)
01	{01}
01+1	{01, 1}
(0+1)*	{ε, 0, 1, 00, 01, ...}
(01+ε)1	{011, 1}
(0+1)*01	{01, 001, 101, 0001, ...}
(0+10)*(ε+1)	{ε, 0, 10, 00, 001, 010, 0101, ...}

Check slides for examples - Lec3. I'm a copy ;) high

DFA to RE:-

If a lang is regular, it accepts an RE.

Given a DFA, we recursively construct a 2 state generalized NFA with

i) a start and end state [Single; no incoming arrows to start, outgoing from end]

ii) a single arrow from start to end

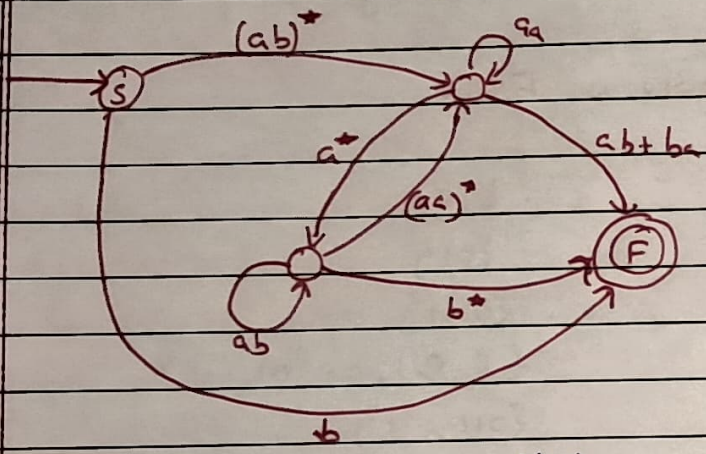
iii) The label of the arrow is the RE corresponding to the language accepted by M.

→ (S) —(R1+R2)R3*+R4→ (E) } → Generalized NFA.

hello cold world hello h & 11 &

Generalized NFAs

→ Transitions may have REs
→ Unique start state with only outgoing arrows
→ Unique final state with only incoming arrows
→ During a run on a GNFA, blocks of symbols might be read instead of single symbols. [These correspond to REs]



This GNFA has multiple accepting strings like b, abababab, aaabbbaa, etc.

↳ Uses ε transitions [think!]

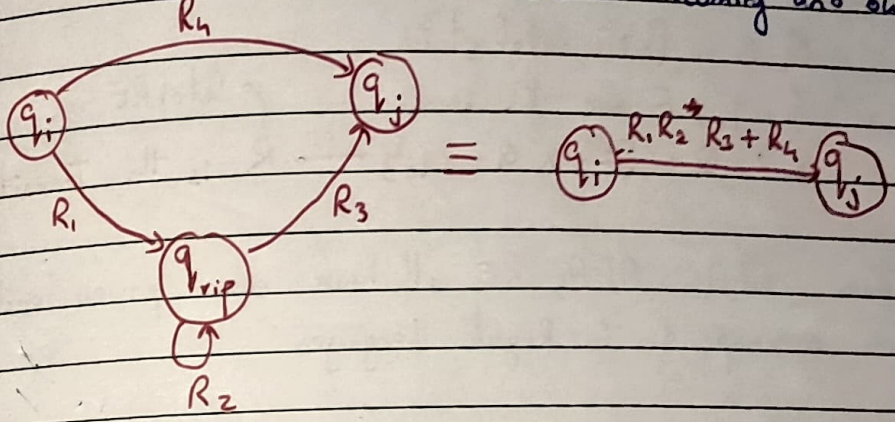
"All transitions can be ε ↵
due to it being RE!

Starting from a DFA, we will begin by constructing a GNFA on k states. We then follow a recursive procedure to keep reducing the no. of states by 1 until we reach a GNFA with 2 states.

(i) Add a new start state with an ε transition to the old start states.
Add a new final state with ε transitions from all the old final states

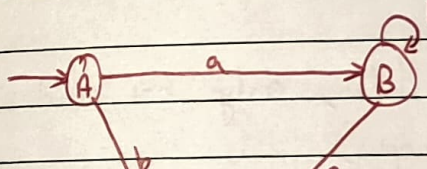
(ii) & Pick any state [! = start / finish] of the GNFA
Call it q_{rip}

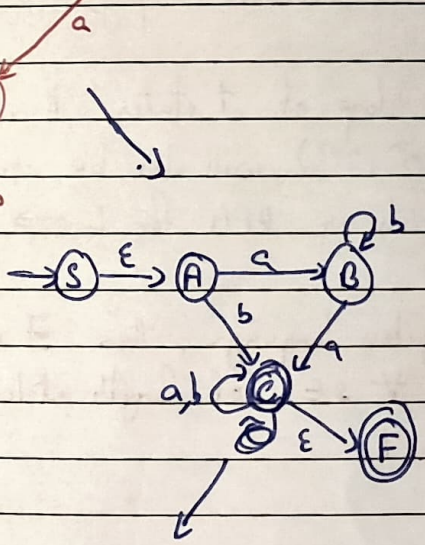
(iii) Rip out q_{rip} and adjust the states of the GNFA.
Account all incoming and outgoing edges

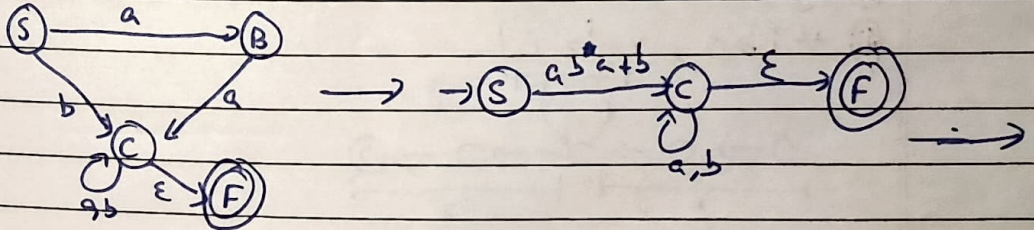


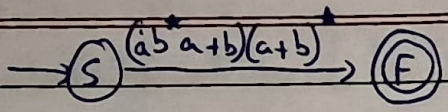
Ex:-

DFA →









Formally, a GNFA is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where

Q is a finite set of states

Σ, q_0, F are the same

$\delta : Q - \{F\} \times Q - \{q_0\} \longrightarrow R$ is the transition fn. (R = set of RE)

Thus DFAs, NFAs, RE all have equal power and all of them correspond to Regular Languages

Pumping Lemma:-

Let $\Sigma = \{0, 1\}$. Consider $L = \{0^n 1^n \mid n \geq 0\}$.

For a DFA of n-distinct states, only n-1 sized strings can be accepted (without looping)

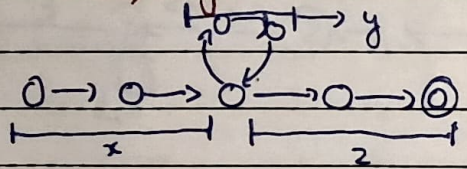
If there is a loop of t states, then even $0^{n+t}1^n$ (or perhaps $0^n 1^{n+t}$) will also be read, which means that there exists no DFA for L $\longrightarrow$ L is not regular.

If L is a regular language, then $\exists$ a number p (pumping length) where $\forall\ s \in L$ of length at least p, $\exists\ x, y, z$ s.t $s = xyz$, s.t

(i) $|xy| \leq p$

(ii) $|y| \geq 1$

(iii) $\forall\ i \geq 0,\ xy^iz \in L$



We usually prove a language is not regular by contraposition

Contraposition = $\forall p$ s.t $s \in L$ with $|s| \geq p$, $\forall (x,y,z)$, with $s = xyz$ s.t $\neg(|xy| \leq p) \vee \neg(|y| \geq 1 \vee \neg(\forall i \geq 0, xy^i z \in L)$

Proof:- Say the DFA M has p states.

By the pigeonhole principle, any string $\geq p$ would encounter a loop.

Let $s = s_1 s_2 \ldots s_n$ with $n \geq p$ and suppose $r_1 r_2 \ldots r_{n+1}$ be the sequence of states encountered while ~~implied~~ implementing a run of s in M.

There would be two states $r_i = r_j$ where $i \neq j$. We can hence divide s into three parts → $\underbrace{s_1 s_2 \ldots s_{i-1}}_{x}$ $\underbrace{s_i \ldots s_j}_{y}$ and $\underbrace{s_{j+1} \ldots s_n}_{z}$

(i) We can traverse the loop bit any number of times and so $\forall i \geq 0$, $xy^i z \in L$

(ii) Also, as $j \neq 1$, $|y| \geq 1$

(iii) The DFA reads $|xy|$ before the end of the first loop, hence $|xy| \leq p$

Context Free Grammars

Grammars:- provide a way to generate strings belonging to a language
↳ set of rules → Systematic procedures

Contain variables and terminals
↳ consist of strings over the alphabet corresponding to the language that the grammar generates

Formally, Grammars are 4-tuples (V, Σ, P, S) such that

(i) V is the set of variables
(ii) Σ is the set of terminals (disjoint from V)
(iii) P is the set of production rules
(iv) S is the start variable

$P: (V \cup \Sigma)^* V (V \cup \Sigma)^* \longrightarrow (V \cup \Sigma^*)$

↳ Has at least one variable on the left hand side

S is generally the variable on the LHS of the first rule.

Ex:- G: $X \longrightarrow 1X$
$X \longrightarrow 0Y$ $\quad S = X$
$Y \longrightarrow 0X$ $\quad V = [X, Y]$
$Y \longrightarrow 1Y$ $\quad \Sigma = [0, 1]$
$Y \longrightarrow \varepsilon$

The seq of substitutions using the rules of G required to obtain a certain string is called a derivation.
. The generated string is said to be derived by the grammar

$X \rightarrow 1X \rightarrow 11X \rightarrow 110Y \rightarrow 1101Y \rightarrow 1101$
is a valid derivation.

A string belongs to the Language of a Grammar if $\exists$ a derivation for it.

$L(G)$ is $\{ \omega \in \Sigma^+ \mid S \Rightarrow \omega \}$
↳ derivation

Regular Grammar :- If the rules of the Grammar G are of the form

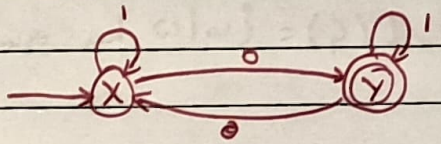
Var → Ter Var

Var → Ter

Var → ε .

then the language of the grammar is regular. These are also called Right-linear grammar (a single var to the right of the terminals in the RHS)

DFA:-



Regular Grammar

A run in the DFA are analogous to the derivations in ~~CFG~~s

Left linear grammar: Var → Var Ter

Var → Ter

Var → ε

Mixing Left and Right Linear Grammars does not result in Regular Langs

Context Free Grammars :- If the rules of the Grammar is of the form V → (~~V∪Σ~~) (V ∪ T)*, then such a Grammar is Context Free.

Regular Grammars ⊂ CFG

Ex:- S → 0S1 | ε → L(G) = {ω | ω = 0^n 1^n, n ≥ 0}

CFGs are more powerful than Regular Grammars / DFA / NFA

S → 0S1 | SS | ε → If 0 & 1 are {}, then it generates balanced paranthesis

Tips:-

1) Check if the CFL is a union of simpler languages

$S \rightarrow S_1 | S_2$

$S_1 \rightarrow \ldots$

$S_2 \rightarrow \ldots$

2) Grammars with rules like $S \rightarrow aSb$ help construct machines that might need unbounded memory.

Q) Construct a CFG s.t $L(G) = \{w | w$ has equal 0s & 1s$\}$

$S \rightarrow SS$

$S \rightarrow 0S1$

$S \rightarrow 1S0$

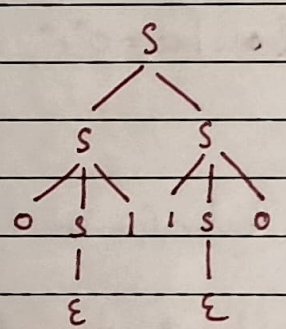
$S \rightarrow \varepsilon$

Parse Trees for CFGs

Ordered trees that provide alt representations of the derivations of a grammar.

Parsing is used in compilers.

Root of the tree is the start node, leaves are the terminals and the leaves are read from left to right. We move from a node to its children by applying the rules of the CFG.



Any string can have multiple derivations, but the parse trees are the same.

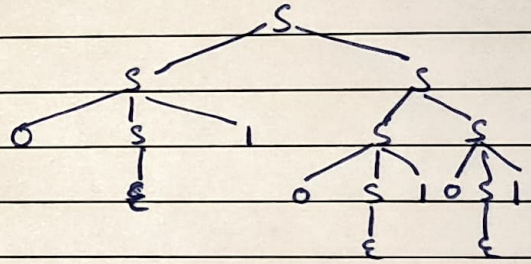
Definition:- If a string is derived by always replacing the leftmost var, it is called the leftmost derivation.

Definition:- Similar for rightmost derivation.

Derivations need not be leftmost or rightmost.

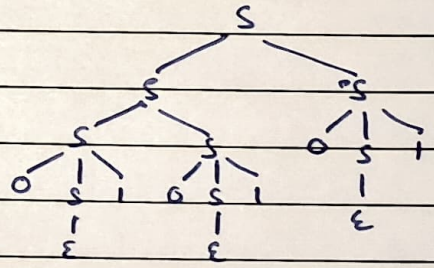
Ambiguous Grammars:- A CFG is called ambiguous if $\exists\ w \in L(G)$ such that there are two or more parse trees / left derivations / right derivations for w.

Consider $w = 01\,01\,01$ with $S \to 0S1 \mid SS \mid \varepsilon$



Leftmost :- $S \to SS \to 0S1S \to 01S \to 01SS \to 010S1S \to 01010S1$

$0101010 \leftarrow 0101051$ ↵



Ambiguity is not desirable, esp in compilers, since there would be different interpretations and hence different outcomes.

We can remove paranthesis or assign precedence while breaking and change of rules. add vars ↵

Chomsky Normal Form:-

A CFG G is said to be in CNF if all of its rules are of this form

Var → Var Var
Var → ter
Start var → ε

(not this)

where Var is any variable incl. the Start Var.

Suppose we have an algorithm, which given a CFG G and an input str w s.t it outputs Yes if it accepts or No if it rejects.
One idea is to try all derivations. But this means that the algorithm would never halt if G does not generate w.

But CNF can help as it guarantees that {if $w \in L(G)$, then a CFG in CNF has derivations of len $2n-1$ for input strings w of length n} {Any CFL can be generated by a CFG in CNF} ②

Algo:-
i) Convert G to CNF
ii) List all derivations of $2n-1$ steps where $|w| = n$
iii) Pattern match to find out.

Proof:- ① Base Case :- Let $|w| = 1$. Then one application of the 2nd rule is sufficient. So the derivation of w would need $2|w|-1 = 1$ step

Induction Hypothesis:- Assume that this is true for any string of length at most k where $k \geq 1$.

Induction Step:- Take $|w| = k+1$. By the first rule, w can be sub divided into two parts A, B where $|A|, |B| \leq k$. So both these parts are derived in $2^{|A|-1}$ & $2^{|B|-1}$ respectively, by induction.

So, to derive w, we would take

$2^{|A|-1} + 2^{|B|-1} + 1 \rightarrow 2^{(|A|+|B|)-1}$

$2^{k+1-1} \leftarrow 2^{(k+1)-1} \leftarrow 2^{|w|-1}$

Hence proved.

(2) To convert a CFG G into a CNF

i) Add a new Start variable $S' \rightarrow S$

ii) Remove ε rules (nullable symbols and rules)

iii) Remove unit (short) rules of the form $A \rightarrow B$

iv) Remove long rules of the form $A \rightarrow u_1 u_2 \cdots u_k$

(ii) For each occurence of 'A' on the right hand side, add a new rule with one occurence of A deleted.

$B \rightarrow uAvAw \Rightarrow B \rightarrow uAvAw \mid uAvw \mid uvAw \mid uvw$

If we had a rule like $B \rightarrow A$, we would have needed to add $B \rightarrow \varepsilon$ (unless the rule has been removed), since B is nullable. Loop until all ε-rules are removed

~~$S \rightarrow 0A1 \mid 2$~~

(iii) We remove rules like $A \rightarrow B$ by using transitivity (i.e, if $B \rightarrow u$, add a new rule $A \rightarrow u$) unless the rule was removed. Loop until finished.

$S \rightarrow A \mid 11$ $\quad$ $S \rightarrow B \mid 1 \mid 11$ $\quad$ $S \rightarrow B \mid 1 \mid 11$ $\quad$ $S \rightarrow B \mid 1 \mid 11$

$A \rightarrow B \mid 1$ $\rightarrow$ $A \rightarrow B \mid 1$ $\rightarrow$ $A \rightarrow S \mid 0 \mid 1$ $\rightarrow$ $A \rightarrow S \mid 0 \mid 1$

$B \rightarrow S \mid 0$ $\quad$ $B \rightarrow S \mid 0$ $\quad$ $B \rightarrow S \mid 0$ $\quad$ $B \rightarrow 1 \mid 11 \mid 0$

$S \rightarrow 0 \mid 1 \mid 11$ $\quad$ $S \rightarrow 0 \mid 1 \mid 11$

$A \rightarrow 0 \mid 1 \mid 11$ $\leftarrow$ $A \rightarrow S \mid 0 \mid 1$

$B \rightarrow 0 \mid 1 \mid 11$ $\quad$ $B \rightarrow 0 \mid 1 \mid 11$

(iv) Add new variables !

$A \to u_1 u_2 \cdots u_k \Rightarrow A \to u_1 A_1$

$A_1 \to u_2 \cdots u_k \Rightarrow A_1 \to u_2 A_2$

Loop till done

Now add newer variables like

$A \to u_1 A_1 \Rightarrow$ ~~$A_{-1} A_1$~~ $A \to A_{-1} A_1$

where $A_{-1} \to u_1$ & $A_1 \to A_{-2} A_2 \cdots$

Ex:- $S \to ASA \mid aB$

$A \to B \mid S$

$B \to b \mid \varepsilon$

(i) $S' \to S$

$S \to ASA \mid aB$

$A \to B \mid S$

$B \to b \mid \varepsilon$

(ii) (a) $S' \to S$

$S \to ASA \mid aB \mid a$

$A \to B \mid S \mid \varepsilon$

$B \to b$

(b) $S' \to S$

$S \to ASA \mid aB \mid a \mid AS \mid SA \mid S$

$A \to B \mid S$

$B \to b$

(iii) (a) Remove $S \to S$ (b) Remove $S' \to S$ (c) $S' \to ASA \mid aB \mid a \mid AS \mid SA \mid S$

$S \to$ "

$A \to S \mid b$

$B \to b$

(d) $S' \to ASA \mid aB \mid a \mid As \mid SA \mid S$

$S \to$ "

$A \to b \mid$ "

$B \to b$

(iv) Replace non unitary SA with U, a with V

$S' \to AU \mid VB \mid a \mid AS \mid SA$

$S \to$ "

$A \to b \mid$ "

$B \to b$

$U \to SA$

$V \to a$

Push down Automata :-

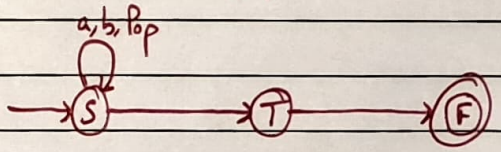
Since there is a need for unbounded memory, the automata for a CFL = FSM + Memory device. By ignoring the memory device, it can emulate a DFA / NFA.

The memory device used is a stack

Transition can happen either based on what it reads, or the element popped from the top.

It can push new elements into the stack

It can also pops from the stack.



Use slides for ~~eter~~ example runs [Lec - 5 or 6 idk]