Q) Define symbol, alphabet, string & language

⇒ Symbol

Symbol are the basic building blocks, which can be any character / token.

eg: A, B, C .... Z
0, 1, 2, 3

⇒ Alphabet

An alphabet is a finite non empty set of symbols. (every language has its own alphabet)

Here in TOC, we use symbol Σ for depicting alphabet.

eg: $\Sigma = \{0, 1\}$
$\Sigma = \{a, b, c, \ldots, z\}$
$\Sigma = \{a, b\}$

⇒ String

It is a finite sequence of symbols.

eg: $\Sigma = \{a, b\}$
String: aabb, aa, b, bab, and so on....

⇒ Language

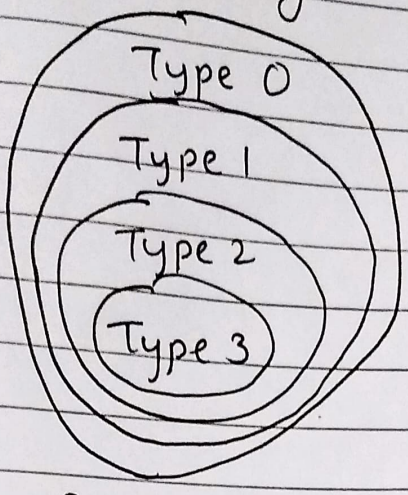
A language is defined as a set of strings

Symbol → Alphabet → String → Language

Page No.:
Date:

Q) Chomsky Hierarchy

=>



(1) Type 3 or Regular grammar

(i) A grammar is called regular grammar if the productions are of the form

$A \to \in$ $A \to a$

$A \to aB$ $A \to bA$, where $a \in T$ & $A, B \in V$

(ii) It is accepted by finite automata.

(2) Type 2 or context free grammar

(i) A grammar is context free if the production are of the form $A \to \alpha$ where $A \in V$ & $\alpha \in (V, T)^*$

(ii) It is accepted by Push down automata

(3) Type 1 or context sensitive grammar

(i) A grammar is said to be context sensitive if all production are of the form $\alpha \to \beta$ where 'β' is atleast as large as 'α'

(ii) It is accepted by linear bounded automata.

(4) Type 0 or unrestricted grammar

(i) There is no restriction on production rule.

(ii) It generates recursively enumerable language.

(iii) It is accepted by Turing machine

Q1. Difference between DFA and NFA

Characteristic	DFA (Deterministic Finite Automata)	NFA (Non-deterministic Finite Automata)
State Transitions	For each input symbol, exactly one next state	For each input symbol, zero, one, or multiple possible next states
ε (Epsilon) Transitions	Not allowed	Allowed (can transition without reading input)
Implementation	Easier to implement	More complex to implement
Processing Time	Predictable, linear time	Can be less predictable due to multiple paths
Memory Usage	Generally uses more memory	Generally uses less memory
State Complexity	Usually has more states	Usually has fewer states
Current State	Always in one state at a time	Can be in multiple states simultaneously
Processing	Processes input from left to right only	Can process input with branching paths
Acceptance	Accepts if final state is reached	Accepts if any path leads to final state
Conversion	Can be converted to NFA	Can be converted to DFA (subset construction)

Q2. Difference between PDA, TM , FA

Characteristic	Finite Automata (FA)	Pushdown Automata (PDA)	Turing Machine (TM)
Memory Type	No memory storage	Stack memory	Infinite tape (read/write)
Memory Access	None	Push/Pop only from top	Read/Write anywhere on tape
Head Movement	Left to right only	Left to right only	Bi-directional (left/right)
Languages Recognized	Regular Languages	Context-Free Languages	Recursively Enumerable Languages
Power/Capability	Least powerful	More powerful than FA	Most powerful
Language Hierarchy	Lowest in hierarchy	Middle in hierarchy	Highest in hierarchy
Input Processing	Read-only	Read-only with stack operations	Read/Write/Modify
State Control	Fixed number of states	Fixed states with stack	Fixed states with infinite tape
Examples of Languages	$a^n b^n$ (n fixed)	$a^n b^n$ (any n)	Any computable function
Time Complexity	Linear O(n)	Can be polynomial	Can be non-halting
Applications	Lexical Analysis, Pattern Matching	Parsing, Expression Evaluation	General Computing
Determinism	Can be deterministic or non-deterministic	Can be deterministic or non-deterministic	Can be deterministic or non-deterministic

Q3. Difference between PDA and NPDA

Characteristic	DPDA (Deterministic PDA)	NPDA (Non-deterministic PDA)
State Transitions	Only one possible transition for each input and stack symbol	Multiple possible transitions for each input and stack symbol
ε (Epsilon) Transitions	Generally not allowed	Allowed
Language Recognition	Subset of Context-Free Languages (Deterministic CFLs)	All Context-Free Languages
Examples of Languages	Well-balanced parentheses, palindromes over single symbol	Palindromes over multiple symbols
Power	Less powerful than NPDA	More powerful than DPDA
Implementation	Easier to implement	More complex to implement
Processing Time	More predictable, generally faster	Less predictable due to backtracking
Stack Operations	One deterministic operation at a time	Multiple possible stack operations
Decision Making	Based on current state, input, and top of stack	Can choose among multiple possible moves
Language Equivalence	Not equivalent to NPDA	Cannot always be converted to DPDA
Practical Applications	Parsing programming languages, Expression evaluation	Natural language processing, Pattern matching
Ambiguity Handling	Cannot handle ambiguous constructs	Can handle ambiguous constructs

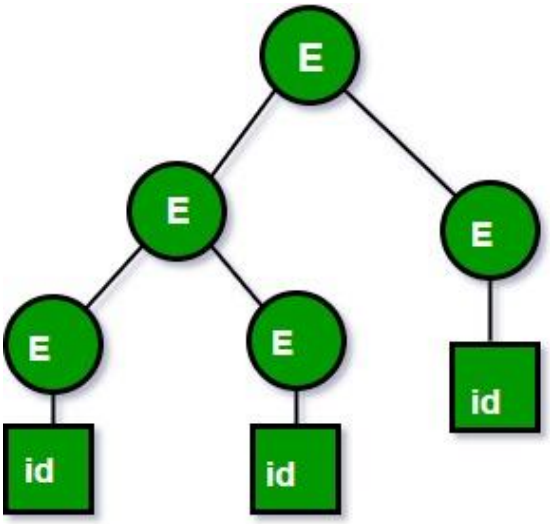
Q4. Ambiguous grammar

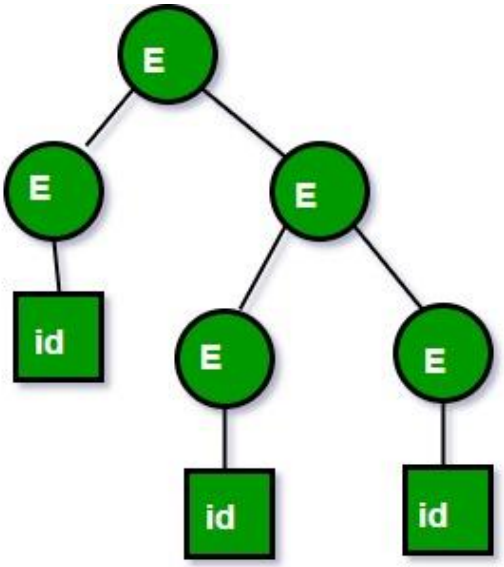
=>

1. An **ambiguous grammar** is a type of grammar in which a sentence (or string) can have more than one structure, or **parse tree**.
2. A parse tree visually represents the structure of the sentence according to the grammar rules, showing how the sentence is built step by step.
3. An **ambiguous grammar** is a CFG where:
    a. Some sentences can be generated with more than one **parse tree**.
    b. This means there are multiple ways to structure a sentence according to the grammar rules.

4. Example of Ambiguous Grammar

a) Consider this rule:
   `E -> E + E | id`
b) For a sentence like `id + id + id`, you can create two different parse trees:
c) One where the first `id + id` is grouped together first.
d) Another where the last `id + id` is grouped first.
e) Each parse tree represents a different structure of the same sentence, making this grammar **ambiguous** because there isn't a unique way to form the sentence.





Q5. Closure properties of regular language

=>

1. **Kleene Closure**: Repeating the language infinitely many times still gives us a regular language. For example, if L is the language {a, b}, L* is { "", a, b, aa, ab, ba, bb, aaa, ... }.
2. **Positive Closure**: Similar to Kleene closure, but with at least one repetition. So, L+ includes all strings in L* except the empty string.
3. **Complement**: The complement of a language L, with respect to a complete set of strings (alphabet) E, includes all strings in E* that are not in L. Since regular languages are closed under complementation, the result is also regular.
4. **Reverse**: Reversing each string in L gives another regular language. For example, if L = {0, 01, 100}, then L^R (the reverse of L) is {0, 10, 001}.
5. **Union**: Combining all strings from two regular languages L and M forms a new regular language. So, L ∪ M (L union M) is regular.
6. **Intersection**: The set of strings that are common to both L and M is also a regular language, called L ∩ M.
7. **Set Difference**: The difference L - M is the set of strings in L but not in M, and it is also regular.
8. **Homomorphism**: This maps each symbol in the alphabet of L to a string in another alphabet. For example, if h(0) = ab and h(1) = c, applying this to L keeps it regular.
9. **Inverse Homomorphism**: Given a homomorphism h, the inverse homomorphism finds all strings in L that, when h is applied, result in strings in another regular language.

Q6. Formal definition of PDA

=>

A **Pushdown Automaton (PDA)** is a type of machine that can process certain types of languages by using a stack to keep track of symbols. Here's a breakdown of its parts in simple terms:

- **Q**: The set of all possible states the PDA can be in.
- **Σ**: The set of input symbols (what the machine reads).
- **Γ**: The set of stack symbols (what can be added to or removed from the stack).
- $\mathbf{q_0}$: The starting state where the PDA begins.
- **Z**: The initial stack symbol, which is in the stack at the beginning.
- **F**: The set of final or accepting states.
- **δ**: The transition function, which defines the rules for moving between states, reading input symbols, and changing the stack.

In a given state, the PDA reads an input symbol and the top symbol of the stack. Based on these, it:

1. Moves to a new state.
2. Adds or removes symbols from the stack.

Example: PDA for Language $\{a^n b^n \mid n > 0\}$

This PDA accepts strings where there are an equal number of a's followed by b's, such as "aaabbb".

For this PDA:

- **States (Q)**: {q0, q1}
- **Input symbols (Σ)**: {a, b}
- **Stack symbols (Γ)**: {A, Z}
- **Transition function (δ)** defines how the PDA reacts:
  - It pushes A onto the stack for each a read in state q0.
  - For each b read in state q0, it pops A from the stack and moves to state q1.
  - When it reaches an empty stack after all symbols are read, it accepts the string.

How the PDA Works with "aaabbb"

1. **Starting** in q0, the stack has only Z.
2. Each a pushes A onto the stack. After reading three a's, the stack has AAA on top of Z.
3. When b's start, each b pops one A from the stack and moves to q1.
4. After reading all b's, only Z remains in the stack.
5. With an empty input and Z on the stack, it pops Z and accepts the input.