TOC FINAL PYQ

Q1. **List the applications of Context Free Grammar.**

Ans=> Programming Language Syntax:

- Defining syntax rules for programming languages (e.g., C, Java, Python).
- Used in the construction of **parsers** in compilers and interpreters.

1.**Natural Language Processing (NLP)**:

- Modeling grammatical structure in natural languages.
- Parsing sentences to understand their syntactic structure.

2. **Automata Theory**:

- Recognizing context-free languages using **Pushdown Automata**.
- Studying language properties and classifications.

3. **Document Structure**:

- Defining the structure of markup languages such as **HTML**, **XML**, and **JSON**.
- Creating document type definitions (DTDs) and schema validation.

4. **Query Language Parsing**:

- Parsing queries in database query languages like **SQL**.

5. **Compiler Design**:

- Constructing Abstract Syntax Trees (ASTs) to represent hierarchical structure of source code.
- Performing syntax error detection and recovery.

6. **Game Development**:

- Designing grammars for scripting languages or rules within games.

7. **Machine Translation**:

- Parsing source language grammar to map it to target language grammar.

Q2**. b) Explain Left -most derivation and right- most derivation tree along with suitable**

diagram.

Ans=>**1. Left-Most Derivation**

- **Definition**: In a left-most derivation, at every step, the leftmost non-terminal in the current string is replaced using a production rule.
- **Example**:

S → AB

A → a

B → b

Deriving the string ab using left-most derivation:

1. Start with S.
2. Replace S → AB.
3. Replace the leftmost non-terminal A → a: aB.
4. Replace the leftmost non-terminal B → b: ab.

- **Left-Most Derivation Tree**:

```
    S
   / \
  A   B
 /     \
a       b
```

---

2. Right-Most Derivation

- **Definition**: In a right-most derivation, at every step, the rightmost non-terminal in the current string is replaced using a production rule.
- **Example**:

S → AB

A → a

B → b

Deriving the string ab using right-most derivation:

1. Start with S.
2. Replace S → AB.
3. Replace the rightmost non-terminal B → b: Ab.
4. Replace the rightmost non-terminal A → a: ab.

Right-Most Derivation Tree:

```
   S
  / \
 A   B
/     \
a      b
```

Q3.Alticulate grammar for set of all strings starting and ending with different symbol

over alphabet set {a, b}.

ans=>**Grammar**

Case 1: Strings start with a and end with b:

S→aXb
X→aX|bX|ϵ
L={abbaabaab}

Case 2: Strings start with b and end with a:

S→bXa

X→aX|bX|ϵ

L={baabaababba}

New paper

Q1. Describe the Recursively Enumerable Language with example.

Ans=>A **Recursively Enumerable (RE) Language** is a class of languages that can be recognized by a **Turing Machine**. These languages are also known as **Type-0** languages in the Chomsky hierarchy. A language is considered recursively enumerable if there exists a Turing machine that can list all the strings in the language, i.e., the machine will halt and accept strings that belong to the language, but for strings that do not belong to the language, the machine might run forever (i.e., it might not halt).

Example of Recursively Enumerable Language:

Consider the language **L = { w | w contains an equal number of 0's and 1's }**. This is the set of all strings where the number of '0's equals the number of '1's.

To design a Turing machine for this language, the machine would:

1. Scan the string for '0's and '1's.
2. For every '0' it finds, it marks it, and for every '1' it finds, it marks it as well.
3. If it can successfully match every '0' with a '1', the machine will halt and accept the string (indicating it belongs to the language).
4. If the string is invalid (e.g., too many '0's or '1's), the machine may run indefinitely, not halting.

Q2. Prove that for any transition function and for any two input strings x and y, δ(q,xy)=δ(δ(q,x),y).

Ans=>**Proof:**

Let q be an initial state, and x and y be two input strings.

Step 1: Process x First

- The machine starts in state q and reads the input string x. After processing x, the machine reaches a new state, say q1=δ(q,x).
- Now, the machine needs to process the input string y starting from state q1.

Step 2: Process y from the new state

- After processing x, the machine transitions to state q1=δ(q,x) and it will now start reading the string y.
- The transition function tells us the next state after reading each symbol of y, so the overall state after processing y from state q1 is:
  δ(δ(q,x),y)
  This means the machine continues processing y from state q1,

Step 3: Process xy in one go

- Now, if we process the entire string xy starting from the initial state q, the machine will first process x and then continue processing y, just like in Step 2.
- The machine starts in state q, processes x, and then processes y, leading to the same final state:
  δ(q,xy)
  This represents the final state reached after processing the entire string xy in one go.

Step 4: Compare the two approaches

- From the above steps, we see that processing x first and then y

  δ(δ(q,x),y)) is equivalent to processing the entire string xy at once

  δ(q,xy)

Q3. Let L be the set of all palindromes over {a,b}. Construct a grammar G generating L.

Ans=>
S -> aSa | bSb | ε

Explanation:

- **S:** The start symbol.
- **aSa, bSb:** Recursive rules that allow for palindromes of increasing length. For example, if 'aba' is a palindrome, then 'aabaa' and 'bbabb' are also palindromes.
- **ε:** The empty string is also considered a palindrome.

Q4. Prove that following regular expressions are equivalent.

aa(b*+ a) +a(ab* + aa) =aa(b* + a)

ans=> To prove the equivalence of the two regular expressions, we can use the following algebraic laws:

1. **Distributive Law:** A(B+C) = AB + AC
2. **Associative Law:** (AB)C = A(BC)
3. **Identity Law:** A + ε = A

Let's break down the left-hand side (LHS) of the equation:

LHS = aa(b* + a) + a(ab* + aa)

Applying the distributive law to both terms:

LHS = aab* + aaa + aab* + aaa

Combining like terms:

LHS = 2(aab* + aaa)

Now, let's factor out 'aa' from both terms:

LHS = 2aa(b* + a)

Since multiplying by 2 doesn't change the language recognized by a regular expression, we can remove the 2:

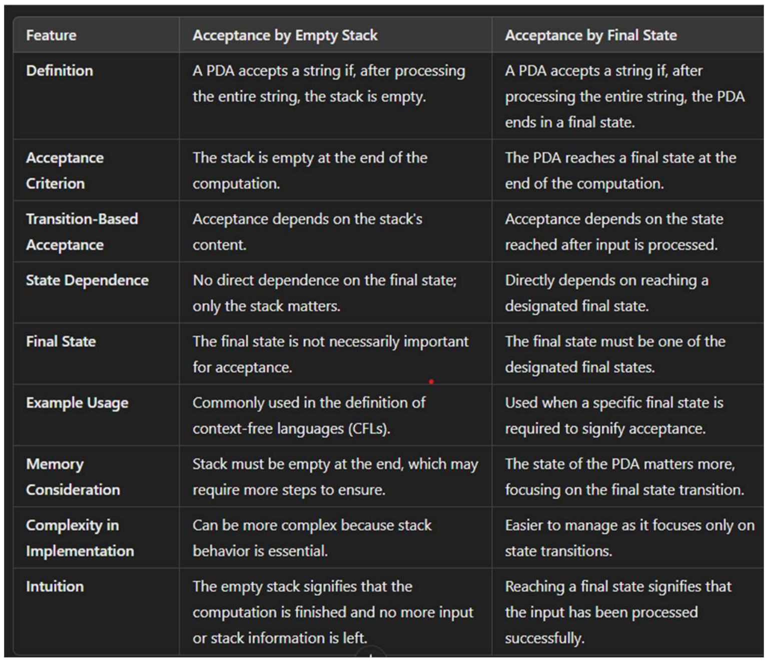
LHS = aa(b* + a)

This is the same as the right-hand side (RHS) of the equation.

Therefore, we have shown that:

LHS = RHS

Q5. What is the difference between PDA acceptance by empty stack and final state?



Feature	Acceptance by Empty Stack	Acceptance by Final State
Definition	A PDA accepts a string if, after processing the entire string, the stack is empty.	A PDA accepts a string if, after processing the entire string, the PDA ends in a final state.
Acceptance Criterion	The stack is empty at the end of the computation.	The PDA reaches a final state at the end of the computation.
Transition-Based Acceptance	Acceptance depends on the stack's content.	Acceptance depends on the state reached after input is processed.
State Dependence	No direct dependence on the final state; only the stack matters.	Directly depends on reaching a designated final state.
Final State	The final state is not necessarily important for acceptance.	The final state must be one of the designated final states.
Example Usage	Commonly used in the definition of context-free languages (CFLs).	Used when a specific final state is required to signify acceptance.
Memory Consideration	Stack must be empty at the end, which may require more steps to ensure.	The state of the PDA matters more, focusing on the final state transition.
Complexity in Implementation	Can be more complex because stack behavior is essential.	Easier to manage as it focuses only on state transitions.
Intuition	The empty stack signifies that the computation is finished and no more input or stack information is left.	Reaching a final state signifies that the input has been processed successfully.

Newpaper

Q1. Comment on TOC in area of computer science.

Ans=> The Theory of Computation (TOC) is a fundamental area in computer science that deals with understanding the nature of computation and what can be computed, as well as how efficiently it can be done.
Key concepts in TOC include:

Automata Theory: This part of TOC focuses on abstract machines (automata) and their capabilities to recognize languages. Automata are used to model computation in its simplest forms.

Formal Languages: TOC studies formal languages, which are sets of strings composed of symbols. Understanding formal languages is crucial for the development of programming languages, compilers, and parsers.

Computability Theory: This explores the boundaries of what can and cannot be computed. It investigates problems like the halting problem.

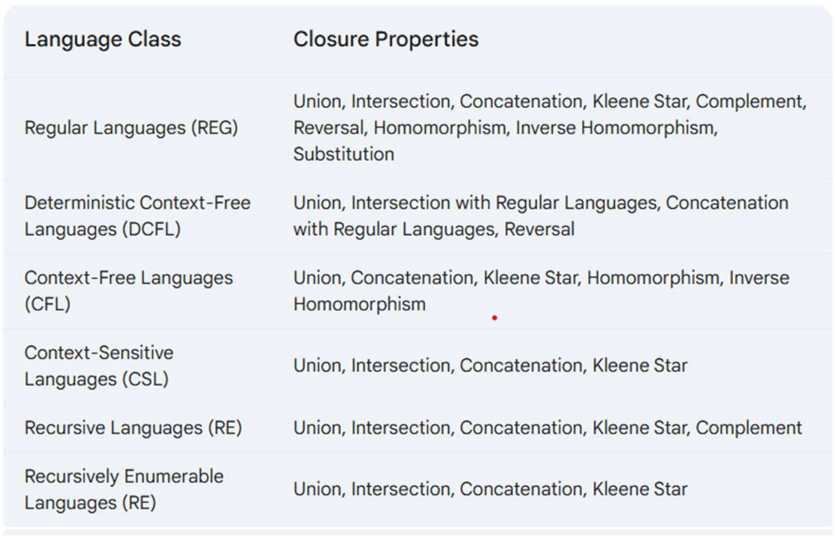
Complexity Theory: Complexity theory focuses on classifying computational problems based on how difficult they are to solve. It distinguishes between problems that can be solved efficiently and those that cannot**.**

Decidability: A key question in TOC is whether a given problem can be solved algorithmically .A problem is said to be **decidable** if there exists an algorithm that can provide a solution in finite time for all possible inputs. If no such algorithm exists, the problem is **undecidable**.

Importance of TOC in Computer Science:
Programming Languages: TOC is fundamental in the design of programming languages. It helps in understanding language syntax, parsing techniques, and the implementation of language features.

Artificial Intelligence and Machine Learning: TOC contributes to AI and ML by providing models for problem-solving and decision-making. Understanding the computational complexity of AI algorithms is crucial for building practical and scalable solutions.

Q2. List the Closure properties of language classes.



Language Class	Closure Properties
Regular Languages (REG)	Union, Intersection, Concatenation, Kleene Star, Complement, Reversal, Homomorphism, Inverse Homomorphism, Substitution
Deterministic Context-Free Languages (DCFL)	Union, Intersection with Regular Languages, Concatenation with Regular Languages, Reversal
Context-Free Languages (CFL)	Union, Concatenation, Kleene Star, Homomorphism, Inverse Homomorphism
Context-Sensitive Languages (CSL)	Union, Intersection, Concatenation, Kleene Star
Recursive Languages (RE)	Union, Intersection, Concatenation, Kleene Star, Complement
Recursively Enumerable Languages (RE)	Union, Intersection, Concatenation, Kleene Star

Q3. If L is a language accepted by a nondeterministic finite automaton (NFA), does a deterministic finite automaton (DFA) exist that accepts L?

Ans=> **Yes**, a deterministic finite automaton (DFA) always exists for any language accepted by a nondeterministic finite automaton (NFA). This is a consequence of the **Nondeterministic-to-Deterministic conversion** theorem. Given any NFA, we can always convert it into a DFA using a process called the **powerset construction** or **subset construction** method.

Newpaper

Q4. Elaborate Sentential form.

Ans=> A **sentential form** is a string of symbols in a formal language that can be derived from the starting symbol (also known as the **start symbol** or **axiom**) of a grammar. In the context of formal grammar, it is an intermediate stage in the process of deriving a string in a language, which consists of a sequence of symbols, some of which may still need to be replaced by other symbols according to production rules.

Example:

Consider the grammar:

S → A B

A → a

B → b

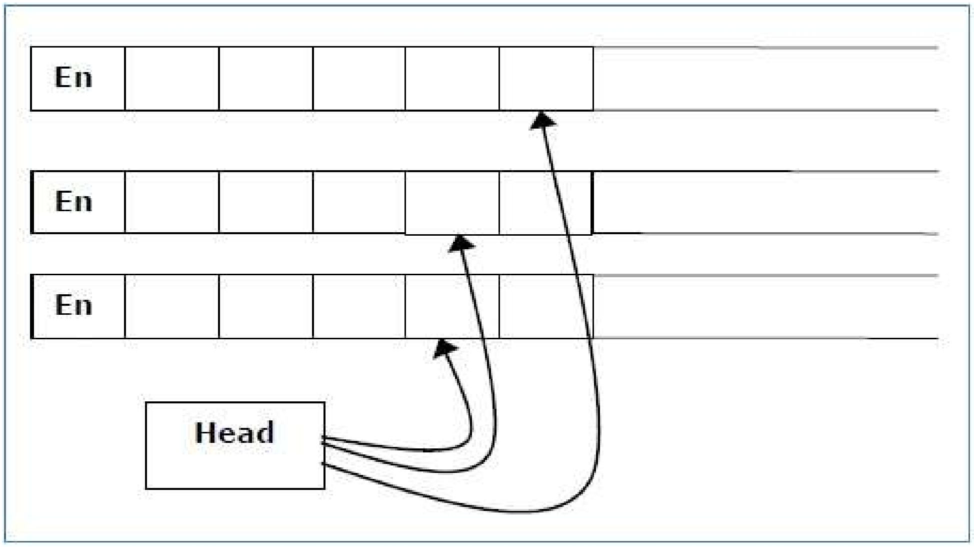
- Start with the start symbol S.
- Using the production S → A B, the sentential form becomes A B.
- Apply A → a to get a B.
- Apply B → b to get a b, which is a terminal string.

Here:

- S is the start symbol.
- A B, a B, and a b are sentential forms, with a b being the final terminal string.

Q5. **Discuss Multi-tape Turing Machine.**

Ans=>Multi-tape Turing Machines have multiple tapes where each tape is accessed with a separate head. Each head can move independently of the other heads. Initially the input is on tape 1 and others are blank. At first, the first tape is occupied by the input and the other tapes are kept blank. Next, the machine reads consecutive symbols under its heads and the TM prints a symbol on each tape and moves its heads.



A Multi-tape Turing machine can be formally described as a 6-tuple (Q, X, B, δ, q_0, F) where –

- **Q** is a finite set of states
- **X** is the tape alphabet
- **B** is the blank symbol
- **δ** is a relation on states and symbols where

$\delta: Q \times X^k \rightarrow Q \times (X \times \{L,R\})^k$

where there is **k** number of tapes

- **q_0** is the initial state
- **F** is the set of final states

Note – Every Multi-tape Turing machine has an equivalent single-tape Turing machine.