FLAT ASSIGNMENT

NAME – B.SAI CHARAN
ROLL NO – 21071A6702
SECTION –CS DS-A
VNR VJIET.

Part-A

a) Define string & differentiate between prefix and suffix of a string with example?

String Definition:

In the realm of computer programming, a string refers to a sequence of characters that is commonly utilized to symbolize text or a gathering of characters.

Prefix Explanation:

When dealing with strings, a prefix denotes a substring that is located at the start or the beginning of the given string.

Suffix Explanation:

Regarding strings, a suffix signifies a substring that is positioned at the end or the conclusion of the provided string.

Example:

Consider the string "Hello new World!". Prefix : "Hello"

Suffix: " World!"

b) Design regular expression for the language of all the strings containing any number

of a’s and b’s

The regular expression will be:

r.e. = (a + b)*

This will give the set as L = {ε, a, aa, b, bb, ab, ba, aba, bab,
.....}, any combination of a and b.

The (a + b)* shows any combination with a and b even a null string.

c) Construct cfg for the language having any number of a's over the set ∑ = {a}

If we want to represent the set that includes the empty string and all possible combinations of 'a' repeated any number of times (including zero), the correct regular expression would be R = a*.

This regular expression denotes that 'a' can occur zero or more times in the string. Therefore, the language set generated by this regular expression is L = {ε, a, aa, aaa, ...}, representing all possible combinations of 'a' with varying lengths, including the empty string.

Part-B

1)(a)Construct minimal DFA over{a,b} na(w)=0mod3 and nb=0mod2

To construct a minimal DFA (Deterministic Finite Automaton) that recognizes the language over {a, b} where the number of 'a's is divisible by 3 and the number of 'b's is divisible by 2, we can follow these steps:

Step 1: Identify the possible remainders when dividing the count of 'a's by 3 and the count of 'b's by 2. In this case, we have 3 possible remainders for 'a': 0, 1, and 2, and 2 possible remainders for 'b': 0 and 1.

Step 2: Create states for each combination of remainders. In this case, we will have a total of 3 x 2 = 6 states.

Step 3: Determine the transitions between states based on the input symbols 'a' and 'b'. For each state, determine the next state based on the remainder obtained after appending the input symbol.

Step 4: Designate one of the states as the start state and mark the states that represent the accepting or final states.

Step 5: Combine any equivalent states to create a minimal DFA, ensuring that all the states are reachable from the start state and that no redundant states or transitions are present.

Based on these steps, the minimal DFA over {a, b} for the given language is as follows:

The initial state is (0, 0), representing no 'a's and no 'b's.

The final state(s) are the states where both X and Y are 0, denoting a multiple of 3 'a's and a multiple of 2 'b's.

Based on these steps, the minimal DFA for the given language is as follows:

```
┌─a─┐
↓       |
→ (0,0) → (1,0)
┌─b─┐  |
|    ↓ ↓ (0,1) (1,1)
```

In this DFA, the arrow indicates the initial state, and the double circles indicate the final state(s).

The transitions are as follows:

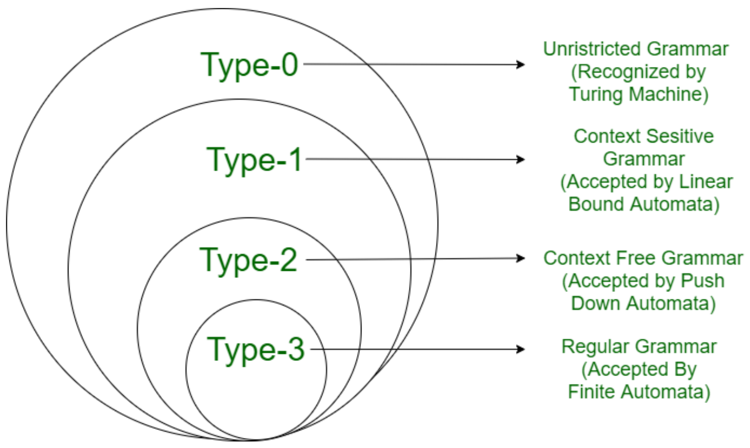
From state (0, 0), on input 'a', it goes to state (1, 0).

From state (1, 0), on input 'a', it loops back to state (0, 0). From state (0, 0), on input 'b', it goes to state (0, 1).

From state (0, 1), on input 'b', it loops back to state (0, 1). From state (1, 0), on input 'b', it goes to state (1, 1).

From state (1, 1), on input 'b', it loops back to state (1, 1)

(b)Illustrate with a diagram Chomsky hierarchy of languages.


Type-0
Type-1
Type-2
Type-3
Unristricted Grammar (Recognized by Turing Machine)
Context Sesitive Grammar (Accepted by Linear Bound Automata)
Context Free Grammar (Accepted by Push Down Automata)
Regular Grammar (Accepted By Finite Automata)

The Chomsky hierarchy classifies formal grammars and languages into four levels based on their generative power and the complexity of their rules. Each level is represented by a different type of grammar, ranging from the most powerful to the least powerful.

Type-0 Grammar (Unrestricted Grammar):

Allows for the most general form of rules and languages.

No restrictions on the production rules.

Represents languages that can be recognized by Turing machines.

Example: Turing machines and their associated languages.

Type-1 Grammar (Context-Sensitive Grammar):

Rules must be of the form $\alpha \rightarrow \beta$, where α and β are strings of terminals and/or non-terminals, and $|\alpha| \leq |\beta|$.

The left-hand side non-terminal can be replaced by the right-hand side string, but the length must remain the same or increase.

Represents languages that can be recognized by linear-bounded automata.

Example: Natural languages and some programming languages.

Type-2 Grammar (Context-Free Grammar):

Rules must be of the form $A \rightarrow \gamma$, where A is a single non-terminal and γ is a string of terminals and/or non-terminals.

The left-hand side non-terminal can be replaced by the right-hand side string, regardless of the context.

Represents languages that can be recognized by pushdown automata.

Example: Programming languages with context-free syntax.

Type-3 Grammar (Regular Grammar):

Rules must be of the form A → aB or A → a, where A and B are non-terminals, and a is a terminal.

The left-hand side non-terminal is always followed by a terminal or an empty string.

Represents languages that can be recognized by finite-state automata.

Example: Regular expressions and regular languages.

The Chomsky hierarchy provides a framework for classifying languages based on their complexity and the types of grammars required to generate or recognize them. Each level represents a subset of languages within the hierarchy, with higher levels having more expressive power but also more complex rules.

2) Apply pumping lemma to show that the language L={a^p |p is a prime } is not a regular?

To apply the pumping lemma to show that the language L = {a^p | p is a prime} is not regular, we need to demonstrate that for any regular language L, there exists a pumping length p such that any string in L with length p or longer can be "pumped" in a way that violates the conditions of the pumping lemma.

Here's the proof by contradiction:

Assume that L = {a^p | p is a prime} is a regular language. Then, according to the pumping lemma for regular languages, there exists a pumping length p for L.

Let's consider a prime number q larger than p, specifically q > p.

Now, let's choose a string w = a^q in L. Since q is a prime number, w satisfies the condition of the language L.

According to the pumping lemma, we can decompose the string w = a^q into three parts: w = xyz, such that:

|xy| ≤ p (the length of xy is at most p),

|y| > 0 (the length of y is greater than 0), and
For any non-negative integer k, the string xy^kz is still in
L. Let's analyze the possible decompositions of w = a^q:

Case 1: y contains only a's (y = a^j for j > 0)

In this case, when we pump y by setting k = 0 (xy^0z), we obtain the string xz = a^(q-j), which is not in L since q-j is not prime. This contradicts the fact that xy^kz should still be in L for any k.

Case 2: y contains both a's and other characters (y = a^jz for j > 0 and z contains characters other than a)
In this case, when we pump y by setting k = 0 (xy^0z), we obtain the string xz, which has a length less than q. Since q is a prime number and xz has a length less than q, xz cannot be in L. This contradicts the fact that xy^kz should still be in L for any k.

In both cases, we have reached a contradiction, showing that the assumption that L is a regular language is false. Therefore, the language L = {a^p | p is a prime} is not a regular language.

3)Explain ambiguous grammar with an example .Eliminate left recursion for the given grammar? E->E+T|T , T->T*F|F , F->id|(E)

An ambiguous grammar is a type of grammar in which a single input string can have multiple possible parse trees or interpretations. This means that there is more than one way to derive the same string using the production rules of the grammar. Ambiguity can arise due to the presence of overlapping or conflicting production rules.

Example of an ambiguous grammar:

Consider the following grammar:

E -> E + T | T T -> T * F | F

F -> id | (E)

Let's take the input string "id + id * id" to demonstrate ambiguity.

Possible parse trees:

Parse Tree 1:

```
         E
       /   \
      E      T
     /      / \
    T      F   id
    |      |
    F      id
    |
    id
```

Parse Tree 2:

```
         E
       / \
      T    E
     / \     \
    F   +     T
    |        / \
    id      T    F
            |    |
            F    id
            |
            id
```

Both parse trees derive the input string "id + id * id" using the given grammar, but they have different interpretations of operator precedence. In Parse Tree 1,
the multiplication () is evaluated before the addition (+), while in Parse Tree 2, the addition (+) is evaluated before the multiplication ().

To eliminate left recursion in the given grammar, we need to modify the production rules to avoid the recursive loop. Here's the modified grammar:

E -> TE'

E' -> +TE' | ε T -> FT'

T' -> *FT' | ε F -> id | (E)