NAME – K.BHARATH

ROLL NO – 21071A6725

SECTION : CSE – DS – A

VNRVJIET

FLAT ASSIGNMENT - I

Part-A

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

String:

A string is a sequence of characters, typically used to represent text or a collection of characters in computer programming.

Prefix:

A prefix of a string is a substring that appears at the beginning of the string.

Suffix:

A suffix of a string is a substring that appears at the end of the 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 context free grammer for the language having any number of a’s over the set ∑ = {a}

All combinations of a's means a may be zero, single, double and so on. If a is appearing zero times,

that means a null string. That is we expect the set of {ε, a, aa, aaa, ...}. So we give a regular expression for this as:

R = a*

<u>Part-B</u>

1. **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)

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

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

Step 1: Identify the states based on the requirements of the language:

We need to keep track of two things: the count of 'a's (mod 3) and the count of 'b's (mod 2).

Let's denote the states as (X, Y), where X represents the count of 'a's (mod 3) and Y represents the count of 'b's (mod 2).

Step 2: Determine the transitions between states:

For each state (X, Y), determine the possible transitions based on the input 'a' or 'b'.

For 'a', update the count of 'a's (X) by (X+1) mod 3, and the count of 'b's (Y) remains the same.

For 'b', update the count of 'b's (Y) by (Y+1) mod 2, and the count of 'a's (X) remains the same.

Step 3: Identify the initial state and final state(s):

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:

scss

Copy code

```
      ┌─ 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).

3)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 assume that L is regular and then derive a contradiction.

The pumping lemma states that for any regular language L, there exists a constant n (the "pumping length") such that any string s in L with a length of n or more can be divided into three parts: s = xyz, where the following conditions hold:

The length of xy is at most n.

The length of y is greater than 0.

For all non-negative integers i, the string xy^iz is also in L.

Now let's assume that L = {a^p | p is a prime} is a regular language. We will proceed to derive a contradiction using the pumping lemma.

Assume L is regular, so it satisfies the pumping lemma.

Let n be the pumping length given by the pumping lemma.

Consider the string s = a^n, which is in L since n is a prime number.

According to the pumping lemma, we can divide s into three parts: s = xyz, satisfying the conditions of the lemma.

Since the length of xy is at most n, both x and y must consist only of 'a's.

Since the length of y is greater than 0, y contains at least one 'a'.

The remaining part z consists of 'a's only.

Now, let's pump y by repeating it: s' = xy^2z.

Since y contains at least one 'a', pumping it will increase the number of 'a's in s'.

Therefore, s' will have more than n 'a's.

According to the language L, s' should also be in L, which means s' should have a prime number of 'a's.

However, since s' has more than n 'a's, it cannot have a prime number of 'a's because n was chosen as the pumping length.

This contradicts the definition of L, which states that L contains only strings with a prime number of 'a's.

Thus, our assumption that L is a regular language is false.

Therefore, we can conclude that L = {a^p | p is a prime} is not a regular language.

By deriving a contradiction using the pumping lemma, we have shown that the language L is not regular.