TOC
Autumn-2022

1) **a) Construct a Context-Free Grammar (CFG) for the regular expression (0 + 1)0 1, that is, any combination of 0 and 1 followed by a single 0 and ending with any number of 1.**

Sol:

Here is a possible context-free grammar that generates the regular expression **(0 + 1)0 1**

S -> A0B

A -> 0 | 1

B -> 1B | ε

Here, **S** is the start symbol, **A** generates either **0** or **1**, and **B** generates any number of **1**s. The symbol **ε** represents the empty string.

The production **S -> A0B** generates a string that starts with either **0** or **1**, followed by a **0**, and ends with any number of **1**s. The production **B -> 1B | ε** allows for any number of **1**s to be generated, including the empty string.

To see how this grammar generates strings that match the regular expression, consider the following derivation:

S => A0B (use S -> A0B)

=> 00B (use A -> 0)

=> 001 (use B -> ε)

The resulting string **001** matches the regular expression **(0 + 1)0 1**. Similarly, the grammar can generate other strings that match the regular expression, such as **1011**, **01111**, **111111**, etc.

1) **a) Or) Construct a CFG for the regular expression 0*1(0+1)*, that is, any number of 0 followed by a single 1 and ending with any combination of 0 and 1.**

Sol:

Here is a context-free grammar (CFG) that generates the language described by the regular expression 0*1(0+1)*:

S -> 0S1 | A

A -> B | C

B -> 0B | 1B | ε

C -> 0C | 1C | ε

In this grammar, S is the starting symbol, which generates strings that begin with any number of 0s followed by a single 1, and end with any combination of 0s and 1s. The non-terminal symbols A, B, and C are used to generate the middle portion of the strings, which can contain any number of 0s and 1s.

The productions of the grammar can be explained as follows:

- The first production, S -> 0S1, generates any number of 0s followed by a single 1, and then recursively generates the remaining portion of the string using the non-terminal symbol S.
- The second production, A -> B | C, generates the middle portion of the string by either using the non-terminal symbol B or the non-terminal symbol C.
- The third production, B -> 0B | 1B | ε, generates any combination of 0s and 1s, including the empty string ε.
- The fourth production, C -> 0C | 1C | ε, generates any combination of 0s and 1s, including the empty string ε.

Together, these productions generate all strings that match the regular expression 0*1(0+1)*.

1) **b) Consider G whose productions are**
   S → aAS/a,
   A→ SbA/SS/ba.
   Show that Saabbaa by constructing a derivation tree, by rightmost derivation, whose yield is aabbaa.

Sol:

The step-by-step derivation tree for the string Saabbaa using rightmost derivation:

Step 1: S (start with S)

Step 2: aAS (apply S → aAS)

Step 3: aaSbA (apply A → SS)

Step 4: aabAbA (apply A → ba)

Step 5: aabbAba (apply A → ba)

Step 6: aabbaaba (apply A → SbA)

Step 7: aabbaa (apply A → ba)

So, the rightmost derivation tree for Saabbaa is as follows:

S

**/ **

a AS

/ \

S bA

/ \ \

a AS ba

/ \

S bA

/ \

a ba

The yield of this tree is **aabbaa**, which is the required string.

1) C)

Consider the grammar

S → ICts/iCtSeS/a

C → b

This grammar is ambiguous. Show in particular that the string ibtibtaea has two:

(i) parse trees, (ii) leftmost derivations, (iii) rightmost derivations.

Sol:

To demonstrate the ambiguity of the given grammar, let's construct two different parse trees for the string "ibtibtaea":

S

/ | \

I C b

| |

b t

|

I

/ \

C tS

| | \

t e S

| |

a C

|

E

S

**/| **

**/ | **

I C -b

| |

b t

|

I

**/ **

**/ **

C tSeS

| | |

t e S C

| |

a e

Both of these trees are valid parse trees for the string "ibtibtaea", but they represent different ways of grouping the input symbols into phrases.

To derive the leftmost derivation, we always expand the leftmost non-terminal symbol at each step. Here are the leftmost derivations for the two parse trees above:

(i) I C t S e S

I C t e S

I C t e a C

i C t e a e

(ii) I C t S e S

I C t Se S

I C t Sea C

i C t Sea e

To derive the rightmost derivation, we always expand the rightmost non-terminal symbol at each step. Here are the rightmost derivations for the two parse trees above:

(i) I C t S e S

I C t S e a C

I C t S C e a C-b

I C t e S C e a C-b

I C t e a C C e a C-b

i C t e a C e a C-b

(ii) I C t S e S

I C t Se S

I C t S C e a C-b

I C t e S C e a C-b

I C t e a C C e a C-b

i C t e a C e a C-b

As shown above, the string "ibtibtaea" has two parse trees, two leftmost derivations, and two rightmost derivations, which demonstrates the ambiguity of the given grammar.

2) a) How do you use the pumping lemma to determine if a language is context free?

Sol:

The pumping lemma for context-free languages is a lemma that can be used to show that a language is not context-free. The lemma states that if a language L is context-free, then there

exists a pumping length p such that for any string w in L of length at least p, there exists a way to break w into five substrings u, v, x, y, and z such that:

- $|uv| \leq p$
- $v \neq \varepsilon$
- For all $i \geq 0$, $uviwz \in L$

If a language L satisfies the pumping lemma, then L is not context-free. This is because the pumping lemma allows us to construct a string that can be pumped an infinite number of times, but a context-free language can only generate a finite number of strings.

To use the pumping lemma to determine if a language is context-free, we can follow these steps:

1. Choose a string w in the language L of length at least p.
2. Break w into five substrings u, v, x, y, and z as described in the pumping lemma.
3. For all $i \geq 0$, construct the string uviwz.
4. If any of the strings uviwz is not in L, then L is not context-free.

If all of the strings uviwz are in L, then we cannot conclude that L is not context-free. However, we cannot conclude that L is context-free either. The pumping lemma only proves that a language is not context-free, it cannot prove that a language is context-free.

2) a) Or) Describe a regular language using a context-free grammar using example.

Sol:

An example of a regular language described using a context-free grammar:

```
S -> aS | ε
```

This grammar generates the language of all strings that are made up of zero or more occurrences of the letter `a`.

To see this, let's consider the different ways that a string can be generated by this grammar. If the starting symbol `S` is immediately followed by the empty string `ε`, then the string will be empty. Otherwise, the starting symbol `S` will be followed by a string of the form `aS`. In this case, the string will be the concatenation of the letter `a` and the string generated by the grammar starting from `S`. This process can be repeated any number of times, so the grammar can generate strings of any length that are made up of zero or more occurrences of the letter `a`.

Here are some examples of strings that are generated by this grammar:

- ""
- "a"
- "aa"
- "aaa"
- ...

Here are some examples of strings that are not generated by this grammar:

- "b"
- "ab"
- "abc"

As you can see, this grammar only generates strings that are made up of zero or more occurrences of the letter `a`. Any other string will not be generated by this grammar.

2) b) Can you give a CFG for the following languages over the alphabet Σ ={a,b}

all strings in the languages, L = {a^n b^2n c^4n | n ≥ 0}

if you can not, justify the reason.

Sol:

No, it is not possible to define the language L = {a^n b^2n c^4n | n ≥ 0} using a context-free grammar (CFG).

Context-free grammars are not powerful enough to generate languages that have a direct relationship between the counts of different symbols. In this case, the number of 'a's is related to the number of 'b's and 'c's in a specific pattern (n, 2n, 4n). A CFG can generate languages where the counts of two symbols are related (e.g., a^n b^n), but it cannot handle relationships between three or more symbols.

To generate the language L = {a^n b^2n c^4n | n ≥ 0}, you would need a more powerful grammar formalism like a context-sensitive grammar or a Turing machine.

2) c)

Context-free grammar (CFG) for the language L = {a^n b^2n | n ≥ 0}:

S -> aSbb | ε

In this grammar, the start symbol S generates strings consisting of 'a's followed by twice the number of 'b's. The production rule S -> aSbb allows the generation of 'a's and 'b's in the desired pattern, and the rule S -> ε allows the generation of the empty string.

Context-free grammar (CFG) for the language of all nonempty strings that read the same from the left or right:

S -> aSa | bSb | a | b

In this grammar, the start symbol S generates strings that are palindromes. The production rule S -> aSa allows the generation of palindromes starting and ending with 'a', the rule S -> bSb allows the generation of palindromes starting and ending with 'b', and the rules S -> a and S -> b generate single characters 'a' and 'b' respectively.

Note: These CFGs generate languages, but they do not provide a mechanism for verifying or enforcing conditions on the languages. They simply define rules for generating strings that belong to the specified languages.

2(c) OR)

To simplify a context-free grammar (CFG), one of the steps is to eliminate unit productions. Unit productions are productions of the form A -> B, where A and B are nonterminal symbols. The procedure to remove unit productions from a CFG is as follows:

1. Identify all unit productions in the grammar.

2. For each unit production A -> B, find all productions B -> α, where α can be a combination of terminals and nonterminals.

3. Replace the unit production A -> B with the productions A -> α, for each B -> α found in the previous step.

4.Repeat steps 2 and 3 until no more unit productions exist.

Let's apply this procedure to remove unit productions from the given grammar:

Original Grammar:

S -> AB

A -> a

B -> C/b

C -> D

D -> E

E -> a

Step 1: Identify unit productions.

A -> a

B -> C/b

C -> D

D -> E

E -> a

Step 2 and 3: Replace unit productions.

A -> a

B -> D/b

C -> E

D -> a

E -> a

After removing unit productions, the simplified grammar becomes:

S -> AB

A -> a

B -> D/b

C -> E

D -> a

E -> a

Note: Removing unit productions helps simplify the CFG by eliminating indirect productions and reducing the size of the grammar

3) a) What is ε-rules and unit rules?

Sol:

In context-free grammar, an ε-rule is a production rule that produces the empty string.

It is represented by the symbol ε. For example, the following is an ε-rule:

S -> ε

This rule states that the non-terminal symbol S can be replaced by the empty string.

A unit rule is a production rule that produces a single non-terminal symbol.

It is represented by the symbol A, where A is a non-terminal symbol. For example, the following is a unit rule:

S -> A

This rule states that the non-terminal symbol S can be replaced by the non-terminal symbol A.

ε-rules and unit rules are often considered to be undesirable in context-free grammars, as they can make the grammar more difficult to understand and analyze.

3) b) Construct a pushdown automaton that recognizes the following language L : { a^n b^2n | n ≥ 0 }

Solution –

The pushdown automaton for the language L = {a^n b^2n | n ≥ 0} can be constructed as follows: -

The pushdown automaton has a stack with the initial symbol 'Z'.

– The set of states is Q = {q0, q1, q2, q3}.

- The set of input symbols is Σ = {a, b}.

- The set of stack symbols is Γ = {Z, A, B}.

- The initial state is q0.

- The accepting state is q3.

The transition function δ is defined as follows:

- δ(q0, a, Z) = {(q0, AZ)}

- δ(q0, b, AZ) = {(q1, ε)}

- $\delta(q1, b, A) = \{(q1, BB)\}$

- $\delta(q1, \varepsilon, Z) = \{(q2, \varepsilon)\}$

- $\delta(q2, \varepsilon, Z) = \{(q3, \varepsilon)\}$

In words, the automaton works as follows:

- In state q0, if the input symbol is a and the stack symbol is Z, push A onto the stack and remain in state q0.

- In state q0, if the input symbol is b and the stack symbol is AZ, pop both symbols and move to state q1.

- In state q1, if the input symbol is b and the top symbol of the stack is A, replace it with BB and remain in state q1.

- In state q1, if the input symbol is ε and the top symbol of the stack is Z, move to state q2.

- In state q2, if the input symbol is ε and the top symbol of the stack is Z, move to state q3.

The automaton accepts the language L because:

- For each a input symbol, the automaton pushes one A symbol onto the stack.

- For each b input symbol, the automaton replaces one A symbol on the stack with two B symbols.

- At the end of the input, the automaton must have exactly two B symbols for each A symbol, and the stack must be empty, which is verified by states q2 and q3.

- If any input is not of the form a^n b^2n, the automaton will reject it because it will get stuck in some state or have symbols on the stack that cannot be matched by the remaining input.

3) b) Or) Suppose the Pushdown Automaton P = ({p, q, £, <0,1) 10, 1, Zol, 8, g, Zo, (f)) has the following transition functions: 8(9, 0, Zo) = ((g, XZo)), §(g, 1, X) = {(p, E)), §(g, 0, X) = {(q, XX)}, §(p, 1, X) = {(p, 8)}, §(p, e, Zo) =' {(I, Zo)}. Starting from the initial ID (g, w, Zo), show all the reachable ID's when the input w is: (g) 00001111 (ii) 00011

Sol:

Given the Pushdown Automaton P with the transition functions as:

$\delta(q0, 0, Z0) = \{(q1, XZ0)\}$

$\delta(q1, 0, X) = \{(q1, XX)\}$

$\delta(q1, 1, X) = \{(q2, \varepsilon)\}$

$\delta(q2, \varepsilon, Z0) = \{(q3, Z0)\}$

We start with the initial ID (g, w, Z0) and try to simulate the PDA for the input strings:

(i) w = 00001111

Initial ID: (g, 00001111, Z0)

Using the first transition function, we get:

(g, 00001111, Z0) → (g, 0001111, XZ0)

Using the second transition function, we get:

(g, 0001111, XZ0) → (q1, 001111, XXZ0)

(g, 001111, XXZ0) → (q1, 01111, XXXZ0)

(g, 01111, XXXZ0) → (q1, 1111, XXXXZ0)

(g, 1111, XXXXZ0) → (q1, 111, XXXXXZ0)

(g, 111, XXXXXZ0) → (q1, 11, XXXXXXZ0)

(g, 11, XXXXXXZ0) → (q2, 1, XXXXXZ0)

(g, 1, XXXXXZ0) → (q3, ε, Z0)

Final ID: (q3, ε, Z0)

Thus, the PDA accepts the input string "00001111".

(ii) w = 00011

Initial ID: (g, 00011, Z0)

Using the first transition function, we get:

(g, 00011, Z0) → (g, 0011, XZ0)

Using the second transition function, we get:

(g, 0011, XZ0) → (q1, 011, XXZ0)

(g, 011, XXZ0) → (q1, 11, XXXZ0)

(g, 11, XXXZ0) → (q2, 1, XXZ0)

(q2, 1, XXZ0) → (q3, ε, Z0)

Final ID: (q3, ε, Z0)

Thus, the PDA accepts the input string "00011".

Therefore, the reachable IDs from the initial ID for the given input strings are:

(i) (g, 00001111, Z0) → (q3, ε, Z0)
(ii) (g, 00011, Z0) → (q3, ε, Z0)

3) c) Convert the following context free grammar(CFG) to an equivalent pushdown automation(PDA):

S-> aSb | bY | Ya

Y-> bY | aY | c | ε

Sol:

The pushdown automata (PDA) that are equivalent to the given context-free grammars (CFGs):

S-> aSb | bY | Ya

Y-> bY | aY | c | ε

The PDA for this CFG is as follows:

- States: q0, q1, q2, q3
- Symbols: a, b, c, $
- Stack symbols: S, Y, ε
- Initial state: q0
- Final states: q3
- Transitions:
    - q0, a, S -> q1
    - q1, b, S -> q2
    - q2, a, Y -> q3
    - q1, b, Y -> q3
    - q0, c, ε -> q3
    - q1, a, ε -> q1

- q2, b, ε -> q2
- q3, $, ε -> q3

3) c) Or) Convert the following context free grammar(CFG) to an equivalent pushdown automation(PDA):

S -> aXbX

X -> aY | bY | ε

Y -> X | c

Sol:

The pushdown automata (PDA) that are equivalent to the given context-free grammars (CFGs):

S -> aXbX

X -> aY | bY | ε

Y -> X | c

The PDA for this CFG is as follows:

- States: q0, q1, q2, q3
- Symbols: a, b, c, $
- Stack symbols: S, X, Y, ε
- Initial state: q0
- Final states: q3
- Transitions:
  - q0, a, S -> q1
  - q1, a, X -> q2
  - q2, b, X -> q3
  - q1, b, Y -> q3
  - q0, c, ε -> q3
  - q1, a, ε -> q1
  - q2, b, ε -> q2
  - q3, $, ε -> q3

4) a)

4) b) Give the implementation level description of Turing machine that accepts the following languages:

i) L = {w | w is the set of strings with an equal number of 0's and 1's}

ii) L = {ww^R | w is any string of 0's and 1's}

iii) L = n-1, where n>0.

Sol:

The implementation level descriptions of the Turing machines that accept the following languages:

i) L = {w | w is the set of strings with an equal number of 0's and 1's}

This Turing machine can be implemented as follows:

- Start in the initial state.
- Count the number of 0's and 1's on the tape.
- If the number of 0's and 1's is equal, accept the input.
- Otherwise, reject the input.

ii) L = {ww^R | w is any string of 0's and 1's}

This Turing machine can be implemented as follows:

- Start in the initial state.
- Copy the input string to the right of the original string.
- Reverse the right string.
- Compare the two strings.
- If the two strings are equal, accept the input.
- Otherwise, reject the input.

iii) L=n-1, where n>0

This Turing machine can be implemented as follows:

- Start in the initial state.
- Write n-1 on the tape.
- Move the head to the right of the n-1.
- Read the number on the tape.
- If the number is equal to n, accept the input.
- Otherwise, reject the input.

5) a)

Convert the following CFG into an equivalent CFG in CNF:

S -> aSb | bY | Ya

Y -> bY | aY | c | ε

Sol:

The steps to convert the given CFG into an equivalent CFG in CNF:

1. Remove null productions.
2. Remove unit productions.
3. Replace all productions of the form A → aB with A → XB and X → a.
4. Replace all productions of the form A → BC with A → XC and X → B.

The resulting CFG in CNF is as follows:

S -> Xb | Yc | Ya

Y -> Xc | aY | c | ε

- Step 1: Null productions are productions that produce the empty string. They can be removed without affecting the language generated by the grammar.
- Step 2: Unit productions are productions that produce a single terminal. They can be removed by replacing them with the terminal.
- Step 3: Productions of the form A → aB can be replaced with A → XB and X → a, where X is a new non-terminal. This transformation ensures that all productions generate a non-terminal on the left-hand side.
- Step 4: Productions of the form A → BC can be replaced with A → XC and X → B, where X is a new non-terminal. This transformation ensures that all productions generate at most two non-terminals on the left-hand side.

The resulting CFG in CNF is equivalent to the original CFG in the sense that they generate the same language.

5) a) Or)

Convert the following CFG into an equivalent CFG in CNF:

S -> aXbX

X -> aY | bY | ε

Y -> X | c

Sol:

The steps to convert the given CFG into an equivalent CFG in CNF:

1. Remove null productions.
2. Remove unit productions.
3. Replace all productions of the form A → aB with A → XB and X → a.
4. Replace all productions of the form A → BC with A → XC and X → B.

The resulting CFG in CNF is as follows:

S -> aXbX | ac | bc

X -> aY | bY | ε

Y -> X | c

- Step 1: Null productions are productions that produce the empty string. They can be removed without affecting the language generated by the grammar.
- Step 2: Unit productions are productions that produce a single terminal. They can be removed by replacing them with the terminal.
- Step 3: Productions of the form A → aB can be replaced with A → XB and X → a, where X is a new non-terminal. This transformation ensures that all productions generate a non-terminal on the left-hand side.
- Step 4: Productions of the form A → BC can be replaced with A → XC and X → B, where X is a new non-terminal. This transformation ensures that all productions generate at most two non-terminals on the left-hand side.

The resulting CFG in CNF is equivalent to the original CFG in the sense that they generate the same language.

5) b) The problem "is a number 'm' prime" is decidable or not?

Sol:

The problem "is a number 'm' prime" is decidable. There is a simple algorithm that can be used to determine whether a number is prime. The algorithm works by dividing the number by all of

the numbers from 2 to the square root of the number. If the number is divisible by any of these numbers, then it is not prime. Otherwise, the number is prime.

For example, to determine whether 17 is prime, we divide it by all of the numbers from 2 to the square root of 17, which is 4.1. 17 is not divisible by 2, 3, or 4, so it is prime.

5) b) Or) The problem "A graph is connected" is decidable or not?

Sol:

The problem of determining if a given graph G is connected is decidable. There are several algorithms that can be used to solve this problem. One simple algorithm is to start at any vertex in the graph and perform a depth-first search (DFS). If every vertex in the graph is reachable from the starting vertex, then the graph is connected. Otherwise, the graph is not connected.

An example of how the DFS algorithm works:

```
def is_connected(G):
  """Returns True if G is connected, False otherwise."""
  visited = set()
  def dfs(v):
    visited.add(v)
    for u in G[v]:
      if u not in visited:
        dfs(u)
  dfs(v)
  return len(visited) == len(G)
```

The DFS algorithm is guaranteed to terminate for all input graphs. Therefore, the problem of determining if a given graph G is connected is decidable.

Another algorithm that can be used to determine if a given graph G is connected is to use breadth-first search (BFS). The BFS algorithm works by starting at any vertex in the graph and expanding the search to all of the vertices that are one edge away from the starting vertex. This process is repeated until all of the vertices in the graph have been visited. If every vertex in the graph is visited, then the graph is connected. Otherwise, the graph is not connected.

The BFS algorithm is guaranteed to terminate for all input graphs. Therefore, the problem of determining if a given graph G is connected is decidable.

5) c) What are np complete and np hard problem? How to show that a problem is NP complete?

NP-hard	NP-Complete
A problem is NP-Hard problem(say X) that can be solved if and only if there is a NP-Complete problem(say Y) that can be reducible into X in polynomial time. Or The problem is NP-Hard if every problem from NP polynomially reduces to it.	A problem is NP-Complete problem that can be solved by a non-deterministic Algorithm/Turing Machine in polynomial time. Or The problem is NP-Complete if it belongs to NP, and every problem from NP polynomially reduces to it.

To show that a problem is NP-complete, we need to prove two things:

1. The problem is in NP. This means that there is a non-deterministic algorithm that can solve the problem in polynomial time.
2. The problem is NP-hard. This means that any problem in NP can be reduced to the problem in polynomial time.

To prove that a problem is in NP, we can use a simple algorithm that works by guessing a solution and then verifying that the solution is correct. For example, to solve the problem of determining if a given number is prime, we can guess a number that is a factor of the given number and then verify that the number is divisible by the guessed number.

To prove that a problem is NP-hard, we can use a reduction. A reduction is an algorithm that takes an instance of one problem and converts it into an instance of another problem. If the reduction can be done in polynomial time, then the second problem is NP-hard.

For example, we can reduce the problem of determining if a given number is prime to the problem of determining if a given Boolean formula is satisfiable. The reduction works by converting the Boolean formula into a number and then checking if the number is prime. If the number is prime, then the Boolean formula is satisfiable. Otherwise, the Boolean formula is not satisfiable.

The Cook-Levin theorem states that SAT is NP-complete. This means that any problem in NP can be reduced to SAT in polynomial time. Therefore, if we can prove that a problem is NP-hard, then we can also prove that the problem is NP-complete by reducing SAT to the problem.

Some examples of NP-complete problems:

- Boolean satisfiability problem (SAT)
- Clique problem
- Subset sum problem
- Traveling salesman problem (TSP)
- Graph coloring problem
- Knapsack problem

These problems are all NP-hard and are believed to be computationally intractable. However, there are many approximation algorithms that can be used to find solutions to these problems that are close to optimal.