**Question 1:**  **[10 marks]**

State whether each of the following statements is True or False:

1. An advantage of compilers is that compiled programs are more portable as compared to interpreted programs. **FALSE**
2. The back-end of a compiler is independent of the source language grammar. **TRUE**
3. An expression with mismatching parentheses has a lexical error. **FALSE**
4. We perform left most derivation in reverse to perform top down parsing. **FALSE**
5. The output of lexical analysis phase is a parse tree. **FALSE**
6. Parsers do not parse the whole code if some errors exist in the program. **FALSE**
7. Syntax analyzer checks whether the parse tree constructed follows the rules of language. **FALSE**
8. Regular Grammar is also context-free. **TRUE**
9. In a scanner, a longest match strategy ensures that the scanner's token recognition will always be unambiguous and correct. **TRUE**
10. Ambiguity in a grammar can always be resolved by converting it to a Left-Recursion Free grammar. **FALSE**

**Question 2: [10 marks]**

In many programming languages, the switch statement is used for conditional execution based on the value of a single variable. Your task is to write a Context-Free Grammar (CFG) that describes the syntax of a C-style switch statement.

The switch statement allows you to check the value of a variable (or expression) and execute different blocks of code depending on the value. It can contain multiple case labels, each associated with a specific value, and optionally, a default case to handle situations where none of the case values match.

For this exercise, assume the condition for the switch statement (the variable being checked) and statements that are executed under each case, are already defined, and focus on the structure of the switch statement itself.

**Sample code:**

switch(condition) {

case value1:

// statements;

break;

case value2:

// statements;

break;

default:

// statements;

}

Solution:

SwitchStatement 🡪 **switch** **(** Condition **)** **{** CaseStatements DefaultCase **}**

CaseStatements 🡪 CaseStatement CaseStatements | **ε**

CaseStatement 🡪 **case** Value **:** Statements **break** **;**

DefaultCase 🡪 **default** **:** Statements **break** **;** | **ε**

Statements 🡪 Statement Statements | **ε**

Statement 🡪 **others**

Condition 🡪 **id** | **num** | Expression

Value 🡪 **num** | **id** // Can be any type of literal or identifier for the case values

**Question 3:**  **[5 Marks]**

Given the following grammar ***G*** for a programming language, for which you are given the task to write a **Recursive Descent Parser**.

1. **Start → Stmt**  eof
2. **Stmt → Assign | CondStmt | Loop**
3. **Assign →**  id =  **Expr**  ;
4. **CondStmt →**  if  **Expr**  then  **Stmt |** if  **Expr**  then  **Stmt** else  **Stmt**
5. **Loop →**  while  **Expr**  do  **Stmt**
6. **Expr → Expr** +  **Term | Term**
7. **Term → Term**  \*  **Factor | Factor**
8. **Factor →**  id  **|**  num  **|**  ( **Expr**  )

To write an effective Recursive Descent Parser, the grammar rules should be unambiguous, free of left recursion and left factored. Clearly identify and indicate each rule that needs to be modified to make the grammar suitable for parsing. (No need to resolve the issue just pinpoint the rules that cause the issue).

**Solution:**

**Left Recursion:**

**Expr → Expr + Term | Term**

**Term → Term \* Factor | Factor**

**Ambiguity:**

**CondStmt → if Expr then Stmt | if Expr then Stmt else Stmt**

**Need of left factoring:**

**CondStmt → if Expr then Stmt | if Expr then Stmt else Stmt**

**Question 4: [15 Marks]**

1. Consider the following left recursive grammar: **[7]**

***X → X Y b* | *Y a* | *b***

***Y → Y b* | *X a* | *a***

Identify and remove Left Recursion from the grammar and rewrite the complete grammar at the end.

**Solution:**

**Grammar: X → XYb | Ya | b**

**Y → Yb | Xa | a**

**Step-1: X → XYb | Ya | b**

**Y → Yb | Z**

**Z → Xa | a**

**Step-2: X → XYb | Ya | b**

**Y → ZY'**

**Y' → ɛ | bY'**

**Z → Xa | a**

**Step-3: X → XYb | Ya | b**

**Y → XaY' | aY'**

**Y' → ɛ | bY'**

**Step-4: X → XYb | XaY'a | aY'a | b**

**Y → XaY' | aY'**

**Y' → ɛ | bY'**

**Step-5: X → XZ | W**

**Z → Yb | aY'a**

**W → aY'a | b**

**Y → XaY' | aY'**

**Y' → ɛ | bY'**

**Step-6: X → WX'**

**X' → ɛ | ZX'**

**Z → Yb | aY'a**

**W → aY'a | b**

**Y → XaY' | aY'**

**Y' → ɛ | bY'**

1. Identify the issue of back tracking in the given grammar using an example and resolve the issue by applying left factoring**. [5]**

***S → a* | *ab* | *abc* | *abcd***

**Solution:**

**Step-1: S → aX**

**X → ɛ | b | bc | bcd**

**Step-2: S → aX**

**X → ɛ | bY**

**Y → ɛ | c | cd**

**Step-3: S → aX**

**X → ɛ | bY**

**Y → ɛ | cZ**

**Z → ɛ | d**

1. List down the lexemes, tokens and the attributes of the tokens at the end of the lexical analysis from the following program segment. **[3]**

**for (int i = 0; i <= size; i++)**

**cout <<“Number of occurrences” << i<<endl;**

**Solution:**

**Token lexeme**

**Keyword for**

**left\_parenthesis (**

**keyword int**

**identifier i**

**equal\_operator =**

**integer\_literal 0**

**delimeter ;**

**identifier i**

**less\_equal\_operator <=**

**identifier size**

**delimeter ;**

**identifier i**

**plus\_plus\_operator ++**

**right\_parenthesis )**

**identifier cout**

**less\_less\_operator <<**

**string\_literal “Number of occurrences”**

**less\_less\_operator <<**

**identifier i**

**less\_less\_operator <<**

**identifier endl**

**delimeter ;**

**Question 5: [2+5+3=10 marks]**

Consider the following grammar *G1:*

***S 🡪 a* | *S # S* | *S @ S***

Here, *S* is the start symbol and the only non-terminal. The symbols *a, #,* and@ are terminals.

1. Give a concrete argument why the grammar is ambiguous using a sample string.

**Solution:**

**A grammar is ambiguous if there is a string that can be derived using two different parse trees or derivations. The string a#a#a can be derived in two different ways with different structures:**

* **First Derivation: (a#a)#a**
* **Second Derivation: a#(a#a)**

**Since the string a#a#a has more than one derivation, this demonstrates that the grammar G1 is ambiguous.**

1. Assume that
   * the operator ***#*** has low precedence and is right-associative
   * the operator ***@*** has high precedence and is left-associative

Give a new grammar ***G2*** which describes the same language as ***G1*** and follows the rules just given. You are allowed to introduce new non-terminals, and it's not necessary to demonstrate the unambiguity of ***G2***, other than by highlighting similarities with corresponding unambiguous grammars from the course materials.

Solution:

***S 🡪 T # S* | T**

**T 🡪 *T @* a| a**

1. We look at the grammars ***G1*** and ***G2***, as well as the following grammar ***G3*** (where the latter contains **+** as new terminal symbol)

***S → a* | *S # S* | *S @ S* | *+ S +***

Which of the languages ***L(G1), L(G2),*** and ***L(G3),*** are regular and which not. Explain and give a regular expression for those which languages happen to be regular.

**Solution:**

**G1 and G2 are the same language (provided the first subproblem was solved correctly . . . ). They are regular and a corresponding regular expression is:**

**a((# ∣ @)a) ∗**

**For G3, the + signs are treated in the form that the number of + “generated” left of the S equals the number of + right of S. That’s prototypical for non-regular languages. Of course the language here does not just contains +’s but also #’s or @’s but the basic fact that a finite-state automaton cannot count symbols unboundedly and remember the number applies also here (for the +’s), thus a FSA cannot recognize L(G3) and the language is therefore not regular.**