LAB 8

1. Initialize an empty stack to hold operators and operands.

2. Initialize an empty list to store the three-address instructions.

3. Split the infix expression into tokens (variables, operators, and parentheses).

4. Initialize a variable for generating temporary variable names (e.g., t1, t2, etc.).

5. Create a precedence table for operators, including their associativity.

6. Iterate through each token from left to right:

   a. If the token is an operand, push it onto the stack.

   b. If the token is an operator:

      1. While the stack is not empty and the operator at the top of the stack has higher or equal precedence (and is left-associative):

         i. Pop the operator from the stack.

         ii. Pop the necessary number of operands from the stack (based on the operator's arity).

         iii. Create a new temporary variable to store the result.

         iv. Generate a three-address instruction with the operator and operands.

         v. Push the temporary variable back onto the stack.

      2. Push the current operator onto the stack.

   c. If the token is an open parenthesis '(', push it onto the stack.

   d. If the token is a closing parenthesis ')':

      1. Pop operators from the stack and generate three-address instructions until an open parenthesis '(' is encountered.

      2. Pop and discard the open parenthesis.

7. After processing all tokens, pop any remaining operators from the stack and generate three-address instructions.

8. The list of three-address instructions now contains the code in three-address format.

Example (using the provided infix expression): a * (b + c) / (d - e)

The complete algorithm involves generating three-address code directly from the infix expression, respecting operator precedence and associativity.

Lab 6

Algorithm to Compute First and Follow Sets for Terminals and Non-terminals:

1. Read the input CFG from the file, where each line corresponds to one production rule. Use the '#' symbol to represent epsilon.
2. Initialize the First and Follow sets for terminals (T) and non-terminals (NT).
3. For each production rule in the CFG: a. Parse the rule into its left-hand side (non-terminal) and right-hand side (sequence of symbols, including non-terminals and terminals).
4. For each non-terminal (NT): a. Initialize its First set as an empty set. b. Initialize its Follow set as an empty set.
5. Iterate through the CFG production rules repeatedly until there are no changes to any First or Follow sets: a. For each production rule in the CFG: i. Calculate the First set for the right-hand side of the rule. - For each symbol in the right-hand side: 1. If it's a terminal, add it to the First set of the non-terminal on the left-hand side. 2. If it's a non-terminal: - If it has an epsilon ('#') in its First set, add the First set of the non-terminal to the left-hand side, excluding epsilon. - If it doesn't have an epsilon, add the First set of the non-terminal to the left-hand side. ii. Calculate the Follow set for the non-terminals in the right-hand side of the rule. - For each non-terminal in the right-hand side: 1. Add the First set of the following terminal (if exists) in the right-hand side of the non-terminal to its Follow set. 2. If the non-terminal is the last symbol in the right-hand side or it can derive epsilon, add the Follow set of the left-hand side non-terminal (excluding epsilon) to its Follow set.
6. Repeat step 5 until there are no changes to any First or Follow sets.
7. Print the computed First and Follow sets for terminals and non-terminals.

Pseudo code:

```
1. Read the CFG from the input file.
2. Initialize First and Follow sets for terminals (T) and non-terminals (NT).
3. Parse the production rules from the CFG.
4. Initialize First and Follow sets for NT.
5. Repeat until there are no changes:
  a. For each production rule in the CFG:
    i. Calculate First set for RHS symbols.
    ii. Calculate Follow set for non-terminals in RHS.
6. Print the computed First and Follow sets for T and NT.
```

LAB 2

1. Open the file with the given file name in read mode.

2. Initialize variables to count lines, words, characters, vowels, consonants, digits, and others.

3. Set flags for tracking word boundaries and checking for vowels.

4. Loop through the characters in the file until the end of the file is reached:

   a. Read a character from the file.

   b. Check if the character is a newline character ('\n'):

      - Increment the line count.

      - Set the word boundary flag to true (assuming a new word can start after a newline).

   c. Check if the character is a whitespace character (e.g., space or tab):

      - If the word boundary flag is false (i.e., we are inside a word), increment the word count.

      - Set the word boundary flag to true.

   d. Check if the character is a letter (check if it's a vowel or consonant):

      - Increment the character count.

      - Check if it's a vowel and increment the vowel count if true, or increment the consonant count if false.

   e. Check if the character is a digit:

      - Increment the character count.

      - Increment the digit count.

   f. If none of the above conditions apply, increment the character count and increment the count for other characters.

5. Close the file.

6. Print the counts of lines, words, characters, vowels, consonants, digits, and other characters.

PSEUDO CODE

```
1. Open the file with the given file name in read mode.

2. Initialize lineCount, wordCount, charCount, vowelCount, consonantCount, digitCount, and otherCount to 0.

3. Set wordBoundaryFlag to true.
```

```
4. Loop until the end of the file:
  a. Read a character from the file.
  b. If character is '\n':
    i. Increment lineCount.
    ii. Set wordBoundaryFlag to true.
  c. If character is whitespace:
    i. If wordBoundaryFlag is false:
      1. Increment wordCount.
      2. Set wordBoundaryFlag to true.
  d. If character is a letter:
    i. Increment charCount.
    ii. If character is a vowel:
      1. Increment vowelCount.
    iii. Else, increment consonantCount.
  e. If character is a digit:
    i. Increment charCount.
    ii. Increment digitCount.
  f. Else:
    i. Increment charCount.
    ii. Increment otherCount.
5. Close the file.
6. Print lineCount, wordCount, charCount, vowelCount, consonantCount, digitCount, and otherCount.
```

LAB3

```
1. Open the file with the given file name in read mode.

2. Initialize variables for counting tokens and for token type detection.
```

3. Create a state variable to keep track of the current state while parsing the file.

4. Loop through the characters in the file until the end of the file is reached:

   a. Read a character from the file.

   b. Implement a finite state machine (FSM) with transition logic to identify tokens based on the input character and the current state. The FSM should handle the following transitions:

      - Identifiers (including keywords)
      - Constants (integers and floating-point numbers)
      - String literals
      - Operators and punctuation
      - Comments (including single-line and multi-line comments)
      - Whitespace, tab spaces, and newlines

5. When a complete token is identified, update the token count and detect its type based on the FSM's final state.

6. Continue parsing the file until the end.

7. Close the file.

8. Print the number of tokens and their types.

Here's a simplified **pseudocode** representation of the algorithm:

```
1. Open the file with the given file name in read mode.
2. Initialize variables: tokenCount = 0, tokenType = UNKNOWN_TOKEN, currentState = INITIAL_STATE.

3. Loop until the end of the file:
  a. Read a character from the file.
  b. Update currentState using the finite state machine transitions.
  // Implement FSM transitions
  c. If the currentState is in a final state:
    i. Increment tokenCount.
    ii. Detect token type based on the currentState.
    iii. Update the currentState to INITIAL_STATE.
    iv. Continue to the next character.
4. Close the file.
5. Print the number of tokens and their types.
```

LAB 4

To implement a recursive descent parser for the given context-free grammar (CFG) and determine whether an input string is accepted or rejected, you can follow these steps:

Algorithm for Recursive Descent Parsing and Recognition:

1. Define functions for each non-terminal symbol in the CFG. These functions should correspond to the production rules for each non-terminal and attempt to recognize the input string accordingly.

2. Start with the highest-level non-terminal function (in this case, `S`) and call it to initiate parsing.

3. Within each non-terminal function, implement logic to check whether the current input matches one of the possible production rules. This includes checking for terminals, non-terminals, and epsilon transitions.

4. If a non-terminal function successfully matches a production rule, it should consume the corresponding part of the input string and call other non-terminal functions as needed to continue the parsing process. If a non-terminal function fails to match any production rule, it should return, indicating a rejection of the input.

5. Continue this process recursively until you reach the end of the input string or identify a failure to match the CFG. If you reach the end of the input string and have successfully matched the CFG, the input string is accepted; otherwise, it is rejected.

6. At each step, maintain a pointer to the current position in the input string and update it as you successfully match productions or backtrack in case of a failure.

7. Print the result (accepted or rejected) based on the outcome of the parsing process.

Here's a simplified pseudocode representation of the algorithm:

```plaintext

1. Define functions for each non-terminal symbol in the CFG:
```

S() - Start symbol

E() - Expression

T() - Term

F() - Factor

2. Start parsing with S():

   a. Call E() to initiate parsing.

3. Implement each non-terminal function to match the production rules for the CFG:

   a. Check if the current input matches the production rules for the non-terminal.

   b. If the input matches the production rules, consume the input, call other non-terminal functions as needed, and continue parsing.

   c. If no production rule matches, return to indicate a rejection.

4. Maintain a pointer to the current position in the input string and update it as you match productions.

5. Continue the recursive descent parsing process until you reach the end of the input string or identify a failure.

6. If you reach the end of the input string and have successfully matched the CFG, the input string is accepted. Otherwise, it is rejected.

7. Print the result (accepted or rejected) based on the outcome of the parsing process.

LAB 5

To generate an LL(1) parsing table from a CFG provided in an input file, you can follow these steps:

Algorithm to Compute and Print LL(1) Parsing Table:

1. Read the CFG from the input file, where each line corresponds to one production rule. Parse each production rule and store it.

2. Initialize the LL(1) parsing table as a two-dimensional array or data structure.

3. For each production rule, calculate the FIRST sets for the right-hand side of the rule. Use these FIRST sets to populate the parsing table.

4. For each production rule, calculate the FOLLOW sets for the left-hand side of the rule. Use these FOLLOW sets to populate the parsing table.

5. For each production rule, for each terminal in the FIRST set of the right-hand side, add the production rule to the corresponding cell in the parsing table.

6. For each production rule, if ε (epsilon) is in the FIRST set of the right-hand side, for each terminal in the FOLLOW set of the left-hand side, add the production rule to the corresponding cell in the parsing table.

7. Print the populated LL(1) parsing table.

Here's a pseudocode representation of the algorithm:

```plaintext
1. Read the CFG from the input file and store the production rules in a data structure.

2. Initialize an LL(1) parsing table as a 2D array.

3. For each production rule:
   a. Calculate the FIRST sets for the right-hand side of the rule.
   b. Calculate the FOLLOW sets for the left-hand side of the rule.

4. For each production rule:
   a. For each terminal in the FIRST set of the right-hand side:
```

      i. Add the production rule to the corresponding cell in the parsing table.

5. For each production rule:

   a. If ε (epsilon) is in the FIRST set of the right-hand side:

      i. For each terminal in the FOLLOW set of the left-hand side:

         - Add the production rule to the corresponding cell in the parsing table.

6. Print the populated LL(1) parsing table.

LAB 7:

To implement an LL(1) parser in C that constructs the LL(1) parsing table and uses it to parse an input string while checking for acceptance, you can follow these steps:

Algorithm for LL(1) Parsing:

1. Read the CFG from the input file, where each line corresponds to one production rule. Parse and store each production rule.

2. Construct the FIRST sets for non-terminals based on previous computations or your LL(1) analysis.

3. Construct the FOLLOW sets for non-terminals based on the FIRST sets and production rules.

4. Construct the LL(1) parsing table based on the FIRST sets, FOLLOW sets, and the production rules.

5. Check whether the CFG is LL(1) by examining the parsing table. If any cell in the table has more than one entry, it is not LL(1). Print that it's not LL(1) and exit.

6. If the CFG is LL(1), initialize a stack for LL(1) parsing and set up the input string to be parsed.

7. Push the start symbol (e.g., S) onto the stack.

8. Loop until the stack is empty:

a. Pop the top element from the stack.

b. If it's a terminal, compare it with the current character in the input string.

- If they match, advance to the next character in the input string.

- If they don't match, print that the string is not accepted and exit.

c. If it's a non-terminal, look up the entry in the LL(1) parsing table corresponding to the current non-terminal and the current input character.

- If the table entry contains a production rule, push the right-hand side of the rule onto the stack in reverse order.

- If the table entry is empty, print that the string is not accepted and exit.

9. After parsing, if the stack is empty, and you've consumed the entire input string, print that the string is accepted. If the stack is not empty, print that the string is not accepted.

10. If the CFG is not LL(1) based on the parsing table, print that it's not LL(1).

Here's a simplified pseudocode representation of the algorithm:

```plaintext
1. Read the CFG from the input file and store the production rules.

2. Construct FIRST and FOLLOW sets for non-terminals based on previous computations or analysis.

3. Construct the LL(1) parsing table based on the FIRST sets, FOLLOW sets, and production rules.

4. Check if the parsing table is LL(1). If not, print that it's not LL(1) and exit.

5. Initialize a stack and the input string.

6. Push the start symbol onto the stack.
```

```
7. Loop until the stack is empty:
   a. Pop the top element from the stack.
   b. If it's a terminal, compare it with the current character in the input string.
      - If they match, advance to the next character in the input string.
      - If they don't match, print that the string is not accepted and exit.
   c. If it's a non-terminal, look up the parsing table entry and update the stack based on the rule.

8. After parsing, check if the stack is empty and the input string is fully consumed. Print acceptance or rejection based on the outcome.

9. If the parsing table indicated that the CFG is not LL(1), print that it's not LL(1).

// Implement the specific logic for parsing using the LL(1) parsing table and stack.
```

In your C program, you'll need to implement the details of the parsing process based on the pseudocode, handle the parsing table, stack operations, and comparison with the input string, and print the results accordingly.