INDEX

S. No.	List of experiments	Date	Sign
1	**Write C/C++ program to identify keywords, identifiers and others from the given input file.**		
2	**a. Write a LEX program to count the number of tokens and display each token with its length in the given statements.** **b. Write a LEX program to identify keywords, identifiers, numbers and other characters and generate tokens for each.**		
3	**a. Write a LEX program to eliminate comment lines (single line and multiline) in a high-level program and copy the comments in comments.txt file and copy the resulting program into a separate file input.c.** **b. Write a LEX program to count the number of characters, words and lines in the given input.** **c. Write a LEX program that read the numbers and add 3 to the numbers if the number is divisible by 7.**		
4	**WAP to implement Recursive Decent Parser (RDP) parser for given grammar.**		
5	**Write a program to calculate first and follow of a given LL (1) grammar.**		
6	**WAP to construct operator precedence parsing table for the given grammar and check the validity of the string.**		
7	**a. Write a YACC program for desktop calculator with ambiguous grammar (evaluate arithmetic expression involving operators: +, -, *, / and ↑).** **b. Write a YACC program for desktop calculator with ambiguous grammar and additional information.** **c. Design, develop and implement a YACC program to demonstrate Shift Reduce Parsing technique for the grammar rules:** $E \rightarrow E + T \mid T$ $T \rightarrow T * F \mid F$ $F \rightarrow P \uparrow F \mid P$ $P \rightarrow (E) \mid id$		

	And parse the sentence: id + id * id.		
8	**Write a program to implement pass-I and pass-II of an assembler.**		
9	**Implement menu driven program to execute any 2 code optimization techniques on given code.**		
10	**Select one block or expression from C language and generate symbol table and target code for the same.**		

EXPERIMENT 1

AIM: Write C/C++/Java/Python program to identify keywords, identifiers, and others from the given input file.

CODE:

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
// Function to check if a token is a C keyword
int isKeyword(char *token)
{
// List of C keywords
char keywords[][10] = {"auto", "break", "case", "char", "const",
"continue", "default",
"do", "double", "else", "enum", "extern", "float", "for", "goto",
"if", "int", "long", "register",
"return", "short", "signed", "sizeof", "static", "struct", "switch",
"typedef", "union",
"unsigned", "void", "volatile", "while"};
// Loop through keywords to check if the token is a keyword
for (int i = 0; i < sizeof(keywords) / sizeof(keywords[0]); i++)
{
if (strcmp(token, keywords[i]) == 0)
{
return 1; // It's a keyword
}
}
return 0;
}
int main()
{
// Declare variables
FILE *file;
char filename[50];
char token[50];
```

```
printf("Enter the name of the input file: ");
scanf("%s", filename);
file = fopen(filename, "r");
if (file == NULL)
{
printf("File not found or could not be opened.\n");
return 1;
}
printf("Identifying tokens:\n");
// Loop through the file, reading tokens
while (fscanf(file, "%s", token) != EOF)
{
if (isKeyword(token))
{
printf("Keyword: %s\n", token);
}
else
{
int i = 0;
int isIdentifier = 1;
int isNumber = 1;
// Check if the token is an identifier or a number
while (token[i])
{
if (!isalpha(token[i]))
{
isIdentifier = 0;
}
if (!isdigit(token[i]) && token[i] != '.')
{
isNumber = 0;
}
i++;
}
// Print the appropriate type of token
```

```
if (isIdentifier)
{
printf("Identifier: %s\n", token);
}
else if (isNumber)
{
printf("Number: %s\n", token);
}
else
{
printf("Operator or Other: %s\n", token);
}
}
}
fclose(file);
return 0;
}
```

Input.txt:

```
1.c    input.txt
Lab 1 > input.txt
1  int a = 25;
2  float b=10.0;
3  sum=a+b;
4
```

OUTPUT:

```
PS C:\Compile Design\Compile Design> cd "c:\Compile Design\Compile Design\Lab 1\" ; if ($?) { gcc 1.c -o 1 }
Enter the name of the input file: input.txt
Identifying tokens:
Keyword: int
Identifier: a
Operator or Other: =
Operator or Other: 25;
Keyword: float
Operator or Other: b=10.0;
Operator or Other: sum=a+b;
PS C:\Compile Design\Compile Design\Lab 1>
```

EXPERIMENT 2

AIM: 2 A) Write a LEX program to count the number of tokens and display each token with its length in the given statements.

CODE:

```
%option noyywrap
%{
int count = 0;
%}
%%
[^\n \t]+ {printf("%s is Token having length = %d\n",yytext,yyleng);count++;}
\n {printf("No. of tokens generated are: %d\n",count);}
. ;
%%
int main()
{
yylex();
}
```

Input.c

```
int a=5,b=10;
```

OUTPUT:

```
PS C:\Compile Design\Compile Design\Lab 2> flex Program_a.l
PS C:\Compile Design\Compile Design\Lab 2> gcc lex.yy.c
PS C:\Compile Design\Compile Design\Lab 2> .\a.exe
int a=10, b=5;
int is Token having length = 3
a=10, is Token having length = 5
b=5; is Token having length = 4
No. of tokens generated are: 3
```

AIM: 2 B) Write a LEX program to identify keywords, identifiers, numbers, and other characters and generate tokens for each.

CODE:

```
%option noyywrap
%{
 int c1 = 0, c2 = 0, c3 = 0, c4 = 0;
%}
%%
auto|break|case|char|const|continue|default|do|double|else|enum|
extern|float|for|g
oto|if|int|long|register|return|short|signed|sizeof|static|struct|switc
h|typedef|union|
unsigned|void|volatile|while {printf("The length of keyword %s: %d \n",
yytext,
yyleng); c1++;}
[a-zA-Z]([a-zA-Z_]|[0-9])* {printf("The length of identifier %s is: %d \n",
yytext, yyleng); c2++;}
[0-9]+ {printf("The length of digit %s is: %d\n", yytext, yyleng); c3++;}
. {printf("The length of Other %s is: %d\n", yytext, yyleng); c4++;}
%%
int main() {
 yylex();
 printf("Total number of tokens: %d \nkeywords: %d, identifiers: %d,
digits:
%d ,others: %d\n", c1+c2+c3+c4, c1, c2, c3, c4);
 return 0;
}
```

OUTPUT:

```
No. of tokens generated are: 5
PS C:\Compile Design\Compile Design\Lab 2> flex Program_b.l
PS C:\Compile Design\Compile Design\Lab 2> gcc lex.yy.c
PS C:\Compile Design\Compile Design\Lab 2> .\a.exe
float k=100;
The length of keyword float: 5
The length of Other   is: 1
The length of identifier k is: 1
The length of Other = is: 1
The length of digit 100 is: 3
The length of Other ; is: 1
```

```
int main(int a ,int b){if a==b printf("%s","same")}
The length of keyword int: 3
The length of Other   is: 1
The length of identifier main is: 4
The length of Other ( is: 1
The length of keyword int: 3
The length of Other   is: 1
The length of identifier a is: 1
The length of Other   is: 1
The length of Other , is: 1
The length of keyword int: 3
The length of Other   is: 1
The length of identifier b is: 1
The length of Other ) is: 1
The length of Other { is: 1
The length of keyword if: 2
The length of Other   is: 1
The length of identifier a is: 1
The length of Other = is: 1
The length of Other = is: 1
The length of identifier b is: 1
The length of Other   is: 1
The length of identifier printf is: 6
The length of Other ( is: 1
The length of Other " is: 1
The length of Other % is: 1
The length of identifier s is: 1
The length of Other " is: 1
```

EXPERIMENT 3

AIM: 3 A) Write a LEX program to eliminate comment lines (single line and multiline) in a high-level program and copy the comments in comments.txt file and copy the resulting program into a separate file input.c.

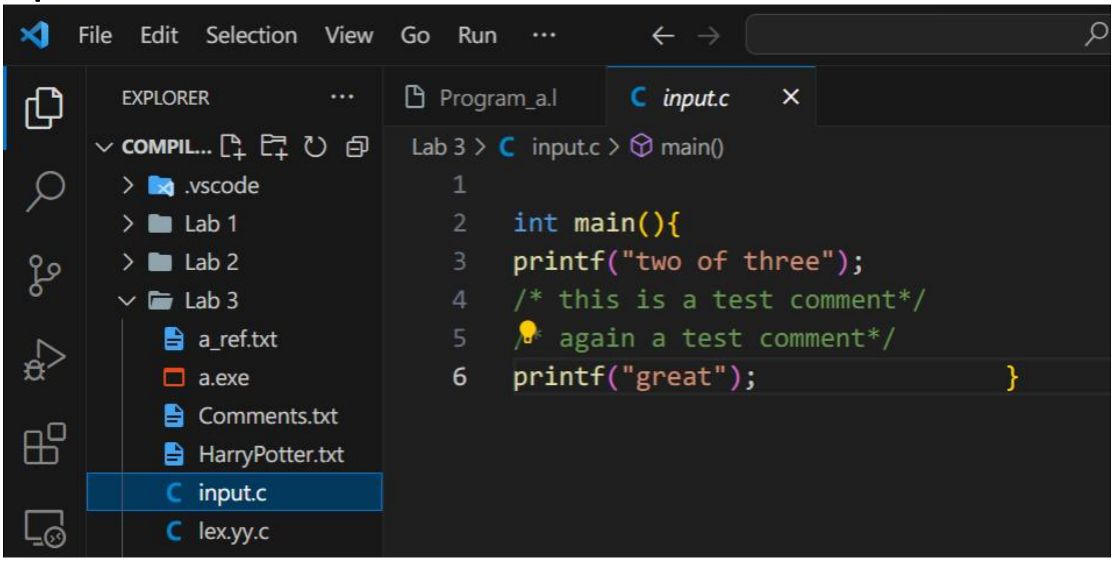
CODE:

```
%option noyywrap
%{
#include <stdio.h>
FILE* output_file;
FILE* comment_file;
%}
%%
\/\/(.*)|\/\*([^]|[^*]|\*[^/])*\*\/ {
 comment_file = fopen("comments.txt", "a");
 if (comment_file) {
 fprintf(comment_file, "%s\n", yytext);
 fclose(comment_file);
 } else {
 fprintf(stderr, "Error opening the file for writing.\n");
 }
}
.|\n {
 output_file = fopen("output.c", "a");
 if (output_file) {
 fprintf(output_file, "%s", yytext);
 fclose(output_file);
 } else {
 fprintf(stderr, "Error opening the file for writing.\n");
 }
```

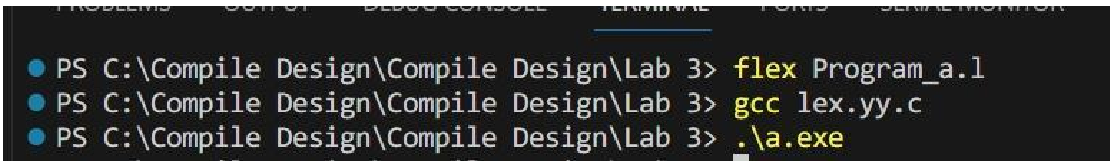


```
}
%%
int main() {
 yyin = fopen("input.c", "r");
 yylex();
 fclose(output_file);
 return 0;
}
```

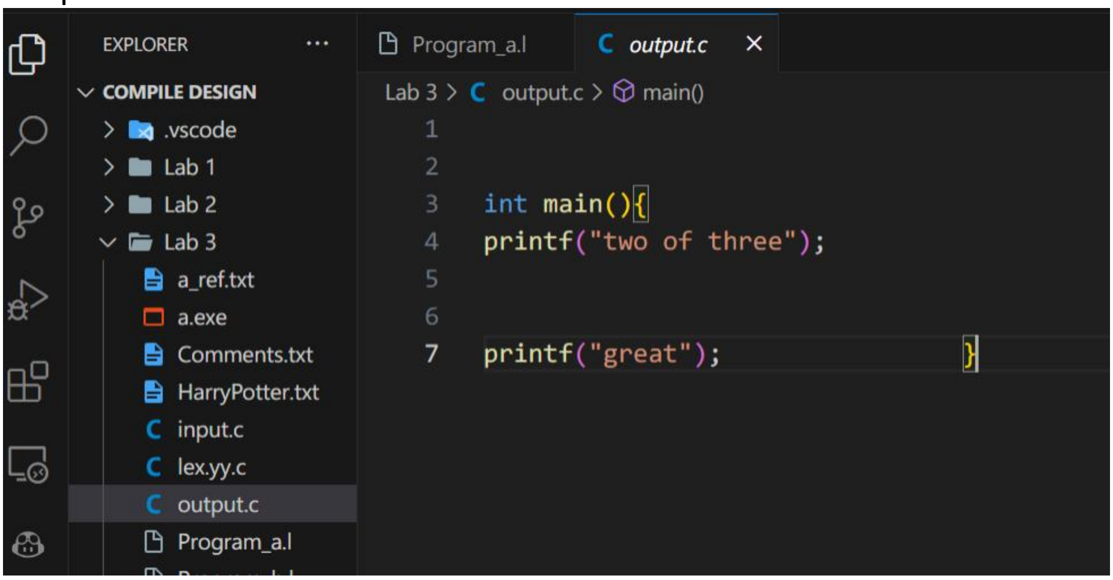
Input.c



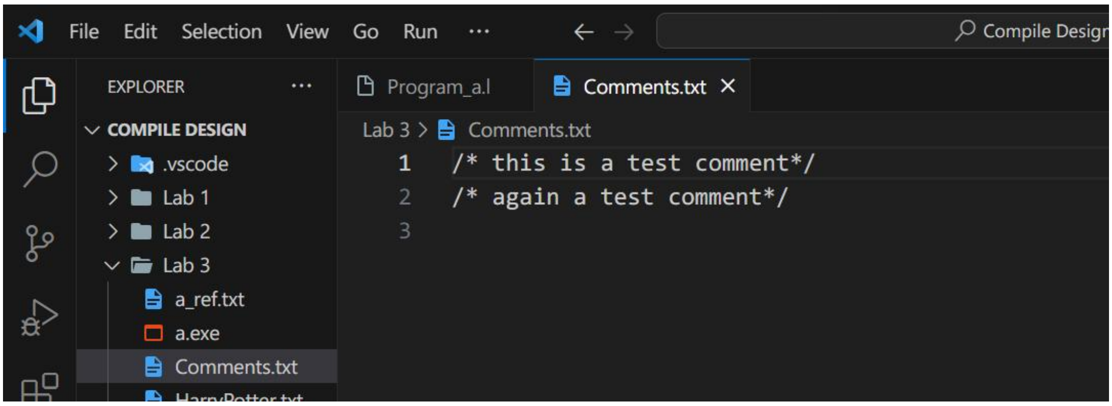
RUNNING THE .l file



Output.c



Comments.txt


File
Edit
Selection
View
Go
Run
Compile Design
EXPLORER
COMPILE DESIGN
.vscode
Lab 1
Lab 2
Lab 3
a_ref.txt
a.exe
Comments.txt
Program_a.l
Comments.txt
Lab 3 > Comments.txt
1 /* this is a test comment*/
2 /* again a test comment*/
3

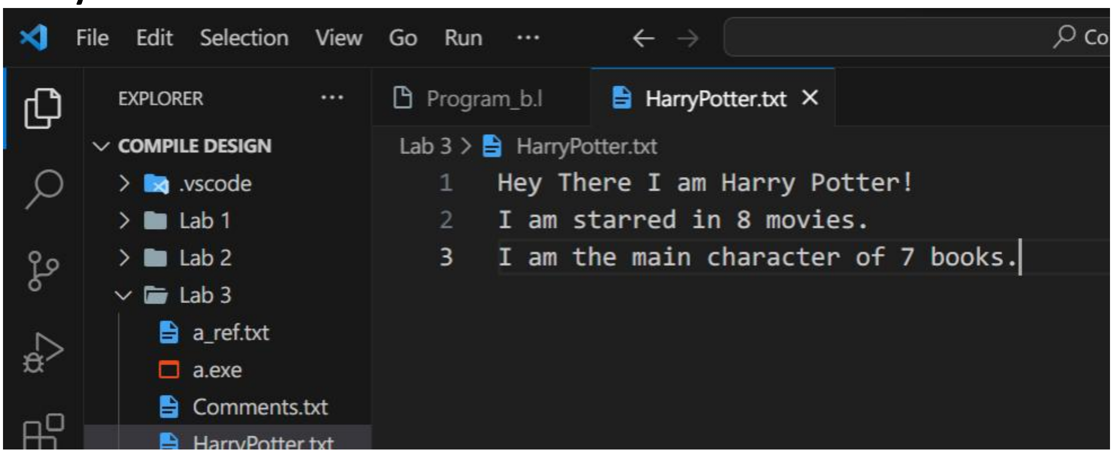
AIM 3B) Write a LEX program to count the number of characters, words and lines in the given input.

CODE:

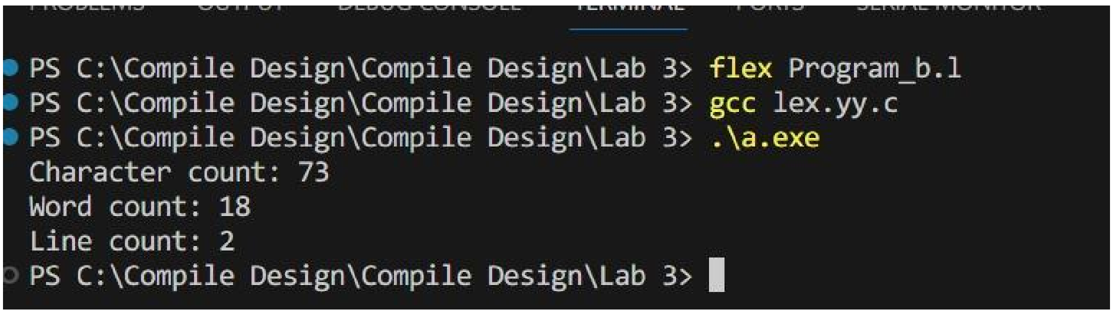
```
%option noyywrap
%{
#include<stdio.h>
int charCount = 0;
int wordCount = 0;
int lineCount = 0;
int inWord = 0;
%}
%%
\n {
 charCount++;
 if (inWord) {
 wordCount++;
 inWord = 0;
 }
 lineCount++;
}
[ \t]+ {
 if (inWord)
 {
 wordCount++;
 inWord = 0;
 }
}
[a-zA-Z]+ {
 charCount += yyleng;
 inWord = 1;
}
. {
 charCount++;
}
%%
int main()
{
 FILE* input = fopen("HarryPotter.txt","r");
 if (!input) {
 fprintf(stderr, "Error opening input file.\n");
```

```
}
yyin = input;
yylex();
if(inWord)
{
wordCount++;
}
fclose(input);
printf("Character count: %d\n", charCount);
printf("Word count: %d\n", wordCount);
printf("Line count: %d\n", lineCount);
}
```

HarryPotter.txt



OUTPUT



AIM 3 C) Write a LEX program that read the numbers and add 3 to the numbers if the number is divisible by 7.

CODE:

```
%option noyywrap
%{
#include <stdio.h>
%}
%%
[0-9]+ {
 int num = atoi(yytext); // Convert matched text to an integer
 if (num % 7 == 0) {
 num += 3;
 }
 printf("%d ", num);
}
.|\n {
 printf("%s", yytext); // Print non-matching characters as they are
}
%%
int main() {
 yylex();
 return 0;
}
```

OUTPUT:

```
PS C:\Compile Design\Compile Design\Lab 3> flex Program_c.l
PS C:\Compile Design\Compile Design\Lab 3> gcc lex.yy.c
PS C:\Compile Design\Compile Design\Lab 3> .\a.exe
23
23
21
24
49
52
```

PRACTICAL 04

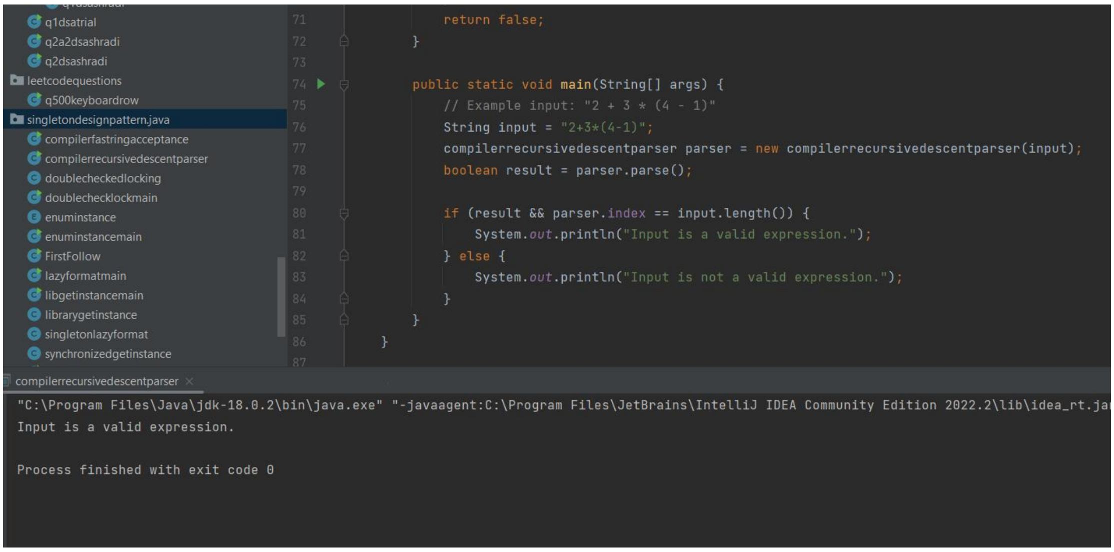
AIM: WAP to implement Recursive Decent Parser (RDP) parser for given grammar.

PROGRAM CODE:

```
import java.util.*;
public class compilerrecursivedescentparser {

    private String input;
    private int index;

    public compilerrecursivedescentparser(String input) {
      this.input = input;
      this.index = 0;
    }

    public boolean parse() {
      return expression();
    }

    private boolean expression() {
      if (term() && expressionPrime()) {
        return true;
      }
      return false;
    }

    private boolean expressionPrime() {
      if (match("+") && term() && expressionPrime()) {
        return true;
      } else if (match("-") && term() && expressionPrime()) {
        return true;
      }
      return true; // Epsilon production (empty string)
    }

    private boolean term() {
      if (factor() && termPrime()) {
        return true;
      }
      return false;
    }

    private boolean termPrime() {
      if (match("*") && factor() && termPrime()) {
        return true;
      } else if (match("/") && factor() && termPrime()) {
        return true;
      }
```

```
        return true; // Epsilon production (empty string)
    }

    private boolean factor() {
        if (match("(") && expression() && match(")")) {
            return true;
        } else if (number()) {
            return true;
        }
        return false;
    }

    private boolean number() {
        int start = index;
        while (index < input.length() && Character.isDigit(input.charAt(index))) {
            index++;
        }
        return index > start;
    }

    private boolean match(String token) {
        if (index < input.length() && input.startsWith(token, index)) {
            index += token.length();
            return true;
        }
        return false;
    }

    public static void main(String[] args) {
        // Example input: "2 + 3 * (4 - 1)"
        String input = "2+3*(4-1)";
        compilerrecursivedescentparser parser = new compilerrecursivedescentparser(input);
        boolean result = parser.parse();

        if (result && parser.index == input.length()) {
            System.out.println("Input is a valid expression.");
        } else {
            System.out.println("Input is not a valid expression.");
        }
    }
}
```

OUTPUT:


q1dsatrial
q2a2dsashradi
q2dsashradi
leetcodequestions
q500keyboardrow
singletondesignpattern.java
compilerfastringacceptance
compilerrecursivedescentparser
doublecheckedlocking
doublechecklockmain
enuminstance
enuminstancemain
FirstFollow
lazyformatmain
libgetinstancemain
librarygetinstance
singletonlazyformat
synchronizedgetinstance
return false;
}
public static void main(String[] args) {
// Example input: "2 + 3 * (4 - 1)"
String input = "2+3*(4-1)";
compilerrecursivedescentparser parser = new compilerrecursivedescentparser(input);
boolean result = parser.parse();
if (result && parser.index == input.length()) {
System.out.println("Input is a valid expression.");
} else {
System.out.println("Input is not a valid expression.");
}
}
}
compilerrecursivedescentparser
"C:\Program Files\Java\jdk-18.0.2\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2022.2\lib\idea_rt.ja
Input is a valid expression.
Process finished with exit code 0

PRACTICAL 05

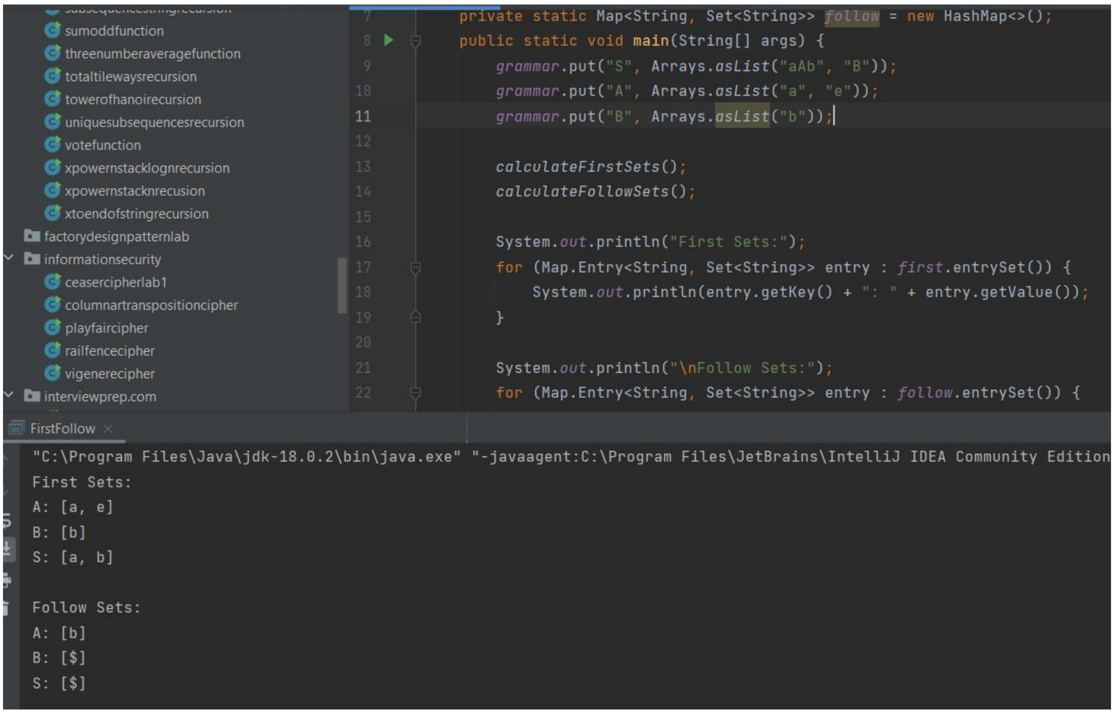
AIM: Write a program to calculate first and follow of a given LL (1) grammar.

PROGRAM CODE:

```
import java.util.*;

public class FirstFollow {
   private static Map<String, List<String>> grammar = new HashMap<>();
   private static Map<String, Set<String>> first = new HashMap<>();
   private static Map<String, Set<String>> follow = new HashMap<>();
   public static void main(String[] args) {
      grammar.put("S", Arrays.asList("aAb", "B"));
      grammar.put("A", Arrays.asList("a", "e"));
      grammar.put("B", Arrays.asList("b"));

      calculateFirstSets();
      calculateFollowSets();

      System.out.println("First Sets:");
      for (Map.Entry<String, Set<String>> entry : first.entrySet()) {
         System.out.println(entry.getKey() + ": " + entry.getValue());
      }

      System.out.println("\nFollow Sets:");
      for (Map.Entry<String, Set<String>> entry : follow.entrySet()) {
         System.out.println(entry.getKey() + ": " + entry.getValue());
      }
   }
   private static void calculateFirstSets() {
      for (String nonTerminal : grammar.keySet()) {
         calculateFirstSet(nonTerminal);
      }
   }
   private static void calculateFirstSet(String nonTerminal) {
      if (first.containsKey(nonTerminal)) {
         return;
      }

      Set<String> firstSet = new HashSet<>();

      for (String production : grammar.get(nonTerminal)) {
         char firstSymbol = production.charAt(0);

         if (Character.isUpperCase(firstSymbol)) {
            calculateFirstSet(String.valueOf(firstSymbol));
            firstSet.addAll(first.get(String.valueOf(firstSymbol)));
         } else {
            firstSet.add(String.valueOf(firstSymbol));
         }
      }

      first.put(nonTerminal, firstSet);
   }
```

```
    private static void calculateFollowSets() {
        follow.put("S", new HashSet<>(Collections.singletonList("$")));

        for (String nonTerminal : grammar.keySet()) {
            calculateFollowSet(nonTerminal);
        }
    }

    private static void calculateFollowSet(String nonTerminal) {
        if (follow.containsKey(nonTerminal)) {
            return;
        }

        Set<String> followSet = new HashSet<>();

        for (Map.Entry<String, List<String>> entry : grammar.entrySet()) {
            String key = entry.getKey();
            List<String> productions = entry.getValue();

            for (String production : productions) {
                int index = production.indexOf(nonTerminal);

                if (index != -1) {
                    if (index < production.length() - 1) {
                        char nextSymbol = production.charAt(index + 1);

                        if (Character.isUpperCase(nextSymbol)) {
                            followSet.addAll(first.get(String.valueOf(nextSymbol)));

                            if (first.get(String.valueOf(nextSymbol)).contains("")) {
                                followSet.remove("");
                                followSet.addAll(follow.get(key));
                            }
                        } else {
                            followSet.add(String.valueOf(nextSymbol));
                        }
                    } else {
                        followSet.addAll(follow.get(key));
                    }
                }
            }
        }

        follow.put(nonTerminal, followSet);
    }
}
```

OUTPUT:


sumoddfunction
threenumberaveragefunction
totaltilewaysrecursion
towerofhanoirecursion
uniquesubsequencesrecursion
votefunction
xpowernstacklognrecursion
xpowernstacknrecusion
xtoendofstringrecursion
factorydesignpatternlab
informationsecurity
ceasercipherlab1
columnartranspositioncipher
playfaircipher
railfencecipher
vigenerecipher
interviewprep.com
private static Map<String, Set<String>> follow = new HashMap<>();
public static void main(String[] args) {
grammar.put("S", Arrays.asList("aAb", "B"));
grammar.put("A", Arrays.asList("a", "e"));
grammar.put("B", Arrays.asList("b"));
calculateFirstSets();
calculateFollowSets();
System.out.println("First Sets:");
for (Map.Entry<String, Set<String>> entry : first.entrySet()) {
System.out.println(entry.getKey() + ": " + entry.getValue());
}
System.out.println("\nFollow Sets:");
for (Map.Entry<String, Set<String>> entry : follow.entrySet()) {
FirstFollow
"C:\Program Files\Java\jdk-18.0.2\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition
First Sets:
A: [a, e]
B: [b]
S: [a, b]
Follow Sets:
A: [b]
B: [$]
S: [$]

EXPERIMENT 6

AIM: WAP to construct operator precedence parsing table for the given grammar and check the validity of the string.

CODE:

```
from tabulate import tabulate
firstOP = {}
lastOP = {}
productions = []
production_dictionary = {}
table_list = []
def add_to_firstOP(nterm, symbol):
 if nterm not in firstOP:
 firstOP[nterm] = set()
 firstOP[nterm].add(symbol)
def add_to_lastOP(nterm, symbol):
 if nterm not in lastOP:
 lastOP[nterm] = set()
 lastOP[nterm].add(symbol)
def replace_err(table):
 for i in range(len(table)):
 for j in range(len(table[i])):
 if table[i][j] == ' ':
 table[i][j] = 'err'
 return table
def parse_expression(str):
 stack = ['$'] # Initialize the stack with '$'
 string = str.split()
 input_buffer = list(string) + ['$'] # Append '$' to the input string
 print(input_buffer)
 index = 0 # Index to traverse the input buffer
 while len(stack) > 0:
 top_stack = stack[-1]
 print(top_stack)
 current_input = input_buffer[index]
 top_stack_index = terminals.index(top_stack)
 current_input_index = terminals.index(current_input)
 relation = terminal_matrix[top_stack_index][current_input_index]
 if relation == '<' or relation == '=':
 stack.append(current_input)
 index += 1
```

```
 elif relation == '>':
 popped = ''
 while relation != '<':
 popped = stack.pop() # Pop elements from the stack until '<' relation is
found
 top_stack = stack[-1] if stack else None
 top_stack_index = terminals.index(top_stack) if top_stack else None
 relation = terminal_matrix[top_stack_index][terminals.index(popped)]
 elif relation == 'acc':
 print("Input string is accepted.")
 return
 else:
 print("Input string is not accepted.")
 return
no_of_terminals = int(input("Enter no. of terminals: "))
terminals = []
print("Enter the terminals:")
for _ in range(no_of_terminals):
 terminals.append(input())
no_of_non_terminals = int(input("Enter no. of non-terminals: "))
non_terminals = []
print("Enter the non-terminals:")
for _ in range(no_of_non_terminals):
 non_terminals.append(input())
starting_symbol = input("Enter the starting symbol: ")
no_of_productions = int(input("Enter no of productions: "))
print("Enter the productions:")
for _ in range(no_of_productions):
 productions.append(input())
for nT in non_terminals:
 production_dictionary[nT] = []
for production in productions:
 nonterm_to_prod = production.split("->")
 alternatives = nonterm_to_prod[1].split("|")
 for alternative in alternatives:
 production_dictionary[nonterm_to_prod[0]].append(alternative)
print("Populated production_dictionary:")
for non_terminal, prods in production_dictionary.items():
 print(f"{non_terminal} -> {prods}")
parsing_string = input("Enter an expression to parse: ")
```

```
# Compute firstOP for each non-terminal
for non_terminal in non_terminals:
 for production in production_dictionary[non_terminal]:
 symbols = production.split()
 print(symbols)
 for symbol in symbols:
 if symbol in non_terminals:
 add_to_firstOP(non_terminal, symbol)
 elif symbol in terminals:
 add_to_firstOP(non_terminal, symbol)
 break
# Compute lastOP for each non-terminal
for non_terminal in non_terminals:
 for production in production_dictionary[non_terminal]:
 symbols = production.split()
 for symbol in reversed(symbols):
 if symbol in non_terminals:
 add_to_lastOP(non_terminal, symbol)
 elif symbol in terminals:
 add_to_lastOP(non_terminal, symbol)
 break
# Print the firstOP and lastOP sets
print("firstOP:")
for non_terminal, first_set in firstOP.items():
 print(f'firstOP({non_terminal}) = {{{", ".join(first_set)}}}')
print("lastOP:")
for non_terminal, last_set in lastOP.items():
 print(f'lastOP({non_terminal}) = {{{", ".join(last_set)}}}')
counter = 0
while counter < no_of_productions:
 for non_terminal, first_set in firstOP.items():
 first_set_copy = first_set.copy() # Create a copy of the set to iterate
over
 for symbol in first_set_copy:
 if symbol in non_terminals:
 firstOP[non_terminal] |= firstOP[symbol]
 counter += 1
# Remove non-terminals from lastOP sets
counter = 0
while counter < no_of_productions:
```

```
 for non_terminal, last_set in lastOP.items():
 last_set_copy = last_set.copy() # Create a copy of the set to iterate over
 for symbol in last_set_copy:
 if symbol in non_terminals:
 lastOP[non_terminal] |= lastOP[symbol]
 counter += 1
# Remove non-terminals from firstOP sets
for non_terminal, first_set in firstOP.items():
 first_set_copy = first_set.copy() # Create a copy of the set to iterate
over
 for symbol in first_set_copy:
 if symbol in non_terminals:
 first_set.remove(symbol)
# Remove non-terminals from lastOP sets
for non_terminal, last_set in lastOP.items():
 last_set_copy = last_set.copy() # Create a copy of the set to iterate over
 for symbol in last_set_copy:
 if symbol in non_terminals:
 last_set.remove(symbol)
# Print the modified firstOP and lastOP sets
print("FirstOP:")
for non_terminal, first_set in firstOP.items():
 print(f'FirstOP({non_terminal}) = {{{", ".join(first_set)}}}')
print("LastOP:")
for non_terminal, last_set in lastOP.items():
 print(f'LastOP({non_terminal}) = {{{", ".join(last_set)}}}')
terminals.append('$')
terminal_matrix = [[' ' for _ in range(len(terminals))] for _ in
range(len(terminals))]
# Rule 1: Whenever terminal a immediately precedes non-terminal B in
any production, put a
<·α where α is any terminal in the firstOP+ list of B
for non_terminal in non_terminals:
 for productions in production_dictionary[non_terminal]:
 production = productions.split()
 for i in range(len(production) - 1):
 if production[i] in terminals and production[i + 1] in non_terminals:
 for alpha in firstOP[production[i + 1]]:
 row_index = terminals.index(production[i])
 col_index = terminals.index(alpha)
```

```
 terminal_matrix[row_index][col_index] = '<'
# Rule 2: Whenever terminal b immediately follows non-terminal C in any production, put β
·>b where β is any terminal in the lastOP+ list of C
for non_terminal in non_terminals:
 for productions in production_dictionary[non_terminal]:
 production = productions.split()
 for i in range(1, len(production)):
 if production[i - 1] in non_terminals and production[i] in terminals:
 for beta in lastOP[production[i - 1]]:
 row_index = terminals.index(beta)
 col_index = terminals.index(production[i])
 terminal_matrix[row_index][col_index] = '>'
# Rule 3: Whenever a sequence aBc or ac occurs in any production, put a
≐ c
for non_terminal in non_terminals:
 for productions in production_dictionary[non_terminal]:
 production = productions.split()
 for i in range(1, len(production) - 1):
 if production[i - 1] in terminals and production[i + 1] in terminals:
 row_index = terminals.index(production[i - 1])
 col_index = terminals.index(production[i + 1])
 terminal_matrix[row_index][col_index] = '='
# Rule 4: Add relations $<· a and a ·> $ for all terminals in the firstOP+ and lastOP+ lists,
respectively of S
for alpha in firstOP[starting_symbol]:
 col_index = terminals.index(alpha)
 terminal_matrix[-1][col_index] = '<'
for beta in lastOP[starting_symbol]:
 row_index = terminals.index(beta)
 terminal_matrix[row_index][-1] = '>'
dollar_index = terminals.index('$')
terminal_matrix[-1][dollar_index] = 'acc'
terminal_matrix = replace_err(terminal_matrix)
for i in range(len(terminals)):
 row = [terminals[i]]
 row.extend([terminal_matrix[i][j] for j in range(len(terminals))])
 table_list.append(row)
headers = [''] + terminals
```

Operator_Precedence_table = tabulate(table_list, headers, tablefmt="grid")
print("Operator Precedence Table:")
print(Operator_Precedence_table)
parse_expression(parsing_string)\

input at command line

```
Design\Compile Design\Lab 6\Operator _Precedence_Parser.py
Enter no. of terminals: 5
Enter the terminals:
x
y
z
a
q
Enter no. of non-terminals: 3
Enter the non-terminals:
S
A
B
Enter the starting symbol: S
Enter no of productions: 3
Enter the productions:
S->x A y | x B y | x A z
A->a S |q
B->q
```

OUTPUT:

```
Populated production_dictionary:
S -> ['x A y ', ' x B y ', ' x A z']
A -> ['a S ', 'q']
B -> ['q']
Enter an expression to parse: x q y
['x', 'A', 'y']
['x', 'B', 'y']
['x', 'A', 'z']
['a', 'S']
['q']
['q']
firstOP:
```

```
firstOP:
firstOP(S) = {x}
firstOP(A) = {a, q}
firstOP(B) = {q}
lastOP:
lastOP(S) = {y, z}
lastOP(A) = {a, S, q}
lastOP(B) = {q}
FirstOP:
FirstOP(S) = {x}
FirstOP(A) = {a, q}
FirstOP(B) = {q}
LastOP:
LastOP(S) = {y, z}
LastOP(A) = {a, q, z, y}
LastOP(B) = {q}
```

```
Operator Precedence Table:
+----+-----+-----+-----+-----+-----+-----+
|    | x   | y   | z   | a   | q   | $   |
+====+=====+=====+=====+=====+=====+=====+
| x  | err | =   | =   | <   | <   | err |
+----+-----+-----+-----+-----+-----+-----+
| y  | err | >   | >   | err | err | >   |
+----+-----+-----+-----+-----+-----+-----+
| z  | err | >   | >   | err | err | >   |
+----+-----+-----+-----+-----+-----+-----+
| a  | <   | >   | >   | err | err | err |
+----+-----+-----+-----+-----+-----+-----+
| q  | err | >   | >   | err | err | err |
+----+-----+-----+-----+-----+-----+-----+
| $  | <   | err | err | err | err | acc |
+----+-----+-----+-----+-----+-----+-----+
['x', 'q', 'y', '$']
$
x
q
x
y
$
Input string is accepted.
```

EXPERIMENT 7

AIM: 7 A) Write a YACC program for desktop calculator with ambiguous grammar (evaluate arithmetic expression involving operators: +, -, *, / and ^).

CODE: (lex file)

```
%{
#include <stdio.h>
#include <stdlib.h>
#include "prog7b.tab.h"
extern int yylval;
%}
%option noyywrap
%%
[0-9]+ { yylval = atoi(yytext); return NUM; }
[ \t] ; /* ignore whitespace */
\n return 0; /* logical EOF */
. return yytext[0];
%%
int main() {
 yylex();
 return 0;
}
int yywrap()
{
 return 1;
}
```

(yacc file)

```
%{
#include <stdio.h>
#include <math.h>
%}
%token NUM
%%
calc: expr { printf("Result: %d\n", $1); };
expr: expr '+' expr { $$ = $1 + $3; }
 | expr '-' expr { $$ = $1 - $3; }
 | expr '*' expr { $$ = $1 * $3; }
 | expr '/' expr { $$ = $1 / $3; }
```

```
 | expr '^' expr { $$ = pow($1, $3); }
 | '-' expr { $$ = - $2; }
 | '(' expr ')' { $$ = $2; }
 | NUM { $$ = $1; }
 ;
%%
int main() {
 yyparse();
 return 0;
}
```

OUTPUT:

```
D:\Compilerlab>flex P7.l

D:\Compilerlab>bison -d pr7.y
pr7.y: conflicts: 30 shift/reduce
```

AIM 7 B) Write a YACC program for desktop calculator with ambiguous grammar and additional information.

CODE: (lex file)

```
%option noyywrap
%{
#include "prog7b.tab.h"
extern int yyval;
extern void yyerror(char *s);
%}
%%
[0-9]+ { yylval = atoi(yytext); return num; }
[ \t] ; /* Ignore whitespace */
\n return 0; /* Logical EOF */
. return yytext[0];
%%
```

(yacc file)

```
%{
#include <stdio.h>
#include <math.h>
void yyerror(char *s);
int yylex();
%}
%token NAME num
%left '+' '-'
%left '*' '/'
%right '^'
%nonassoc UMINUS
%%
s: NAME '=' Ex
| Ex { printf("= %d\n", $1); }
;
Ex: Ex '+' Ex {$$ = $1 + $3;}
| Ex '-' Ex {$$ = $1 - $3;}
| Ex '*' Ex {$$ = $1 * $3;}
| Ex '/' Ex {if($3 == 0){
yyerror("error");
return 1;
}
else
$$ = $1 / $3;
}
| Ex '^' Ex {$$ = pow($1,$3);}
| '-' Ex %prec UMINUS {$$ = -$2;}
| '(' Ex ')' {$$ = $2;}
| num {$$ = $1;}
;
%%
int main() {
yyparse();
return 0;
}
```

```
void yyerror(char *s) {
printf("error");
}
```

Output:

```
D:\Compilerlab>flex pl7b.l

D:\Compilerlab>bison -d prog7b.y

D:\Compilerlab>gcc lex.yy.c prog7b.tab.c
```

```
D:\Compilerlab>.\a.exe
6+6
= 12
```

AIM 7 C) Design, develop and implement a YACC program to demonstrate Shift Reduce Parsing technique for the grammar rules:

CODE: (lex file)

```
%{
#include "p7c.tab.h"
%}
DIGIT [0-9]
WS [ \t]
%%
{DIGIT}+ { yylval = atoi(yytext); return NUMBER; }
{WS}+ /* Skip whitespace */
\n return 0 ;
. return yytext[0];
%%
int yywrap() {
return 1;
}
```

(yacc.file)

```
%{
#include <stdio.h>
#include <math.h>
int yylex(void); // Declare the lexer function
void yyerror(const char* s);
%}
%token NUMBER
%left '+' '-'
%left '*' '/'
%right '^'
%nonassoc UMINUS
%%
statement : exp { printf("%d\n", $1); }
;
exp : term
| exp '+' term { $$ = $1 + $3; }
| exp '-' term { $$ = $1 - $3; }
;
term : factor
| term '*' factor { $$ = $1 * $3; }
| term '/' factor { $$ = $1 / $3; }
;
factor : primary
| factor '^' primary { $$ = pow($1, $3); }
;
primary : NUMBER
```

```
| '-' primary %prec UMINUS { $$ = -$2; }
| '(' exp ')' { $$ = $2; }
;
%%
void yyerror(const char* s) {
fprintf(stderr, "Parse error: %s\n", s);
}
int main() {
yyparse();
return 0;
}
```

Output:

```
D:\Compilerlab>flex plc1.l

D:\Compilerlab>bison -d p7c.y

D:\Compilerlab>gcc lex.yy.c p7c.tab.c
```

pr7.tab	02-11-2023 22:47	C Source File	43 KB
pr7.tab	02-11-2023 22:47	C Header Source F...	3 KB