Pandit Deendayal Energy University, Gandhinagar

School of Technology

Department of Computer Science & Engineering

System Software & Compiler Design Lab (20CP302P)

Name: **Padshala Smit Jagdishbhai**

Enrolment No: **21BCP187**

Semester: **V**

Division: **3 (G6)**

Branch: **Computer Science Engineering**

Practical: 1

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

Code:

lexical_analysis.java

```
// Smit Padshala
// 21BCP187
import java.io.*;
import java.util.*;

public class lexical_analysis {

    public static void main(String[] args) {
        String fileName = "input.java";

        try {
            BufferedReader reader = new BufferedReader(new FileReader(fileName));
            String line;
            ArrayList<String> keywordList = new ArrayList<>();
            ArrayList<String> identifierList = new ArrayList<>();
            ArrayList<String> stringLiteralList = new ArrayList<>();
            ArrayList<String> numberList = new ArrayList<>();
            ArrayList<String> otherList = new ArrayList<>();

            while ((line = reader.readLine()) != null) {
                analyzeLine(line, keywordList, identifierList, stringLiteralList,
numberList, otherList);
            }

            reader.close();

            System.out.println("Keywords: " + keywordList);
            System.out.println("Identifiers: " + identifierList);
            System.out.println("String Literals: " + stringLiteralList);
            System.out.println("Numbers: " + numberList);
            System.out.println("Others: " + otherList);
```

```
        } catch (Exception e) {
            System.err.println("Error reading the file: " + e.getMessage());
        }
    }

    public static void analyzeLine(String line, ArrayList<String> keywordList,
                                   ArrayList<String> identifierList,
ArrayList<String> stringLiteralList,
                                   ArrayList<String> numberList,
ArrayList<String> otherList) {


        StringBuilder currentToken = new StringBuilder();
        boolean insideStringLiteral = false;

        for (int i = 0; i < line.length(); i++) {
            char currentChar = line.charAt(i);

            if (currentChar == '\"') {
                if (insideStringLiteral) {
                    currentToken.append(currentChar);
                    stringLiteralList.add(currentToken.toString());
                    System.out.println("String Literal: " +
currentToken.toString());
                    currentToken.setLength(0);
                }
                insideStringLiteral = !insideStringLiteral;
            } else if (insideStringLiteral) {
                currentToken.append(currentChar);
            } else if (Character.isWhitespace(currentChar)) {
                processToken(currentToken.toString(), keywordList,
identifierList, stringLiteralList, numberList, otherList);
                currentToken.setLength(0);
            } else {
                currentToken.append(currentChar);
            }
        }

        processToken(currentToken.toString(), keywordList, identifierList,
stringLiteralList, numberList, otherList);
    }

    public static void processToken(String token, ArrayList<String> keywordList,
                                    ArrayList<String> identifierList,
ArrayList<String> stringLiteralList,
```

```
                                    ArrayList<String> numberList,
ArrayList<String> otherList) {
        token = token.trim();


        String[] keywords = {"abstract", "assert", "boolean", "break", "byte",
"case", "catch", "char", "class",
            "const", "continue", "default", "do", "double", "else", "enum",
"exports", "extends", "final",
            "finally", "float", "for", "if", "implements", "import",
"instanceof", "int", "interface", "long",
            "module", "native", "new", "open", "opens", "package", "private",
"protected", "provides", "public",
            "requires", "return", "short", "static", "strictfp", "super",
"switch", "synchronized", "this", "throw",
            "throws", "transient", "transitive", "try", "var", "void",
"volatile", "while", "with"};

        if (token.isEmpty()) {
            return;
        }

        if (Arrays.asList(keywords).contains(token)) {
            keywordList.add(token);
            System.out.println(token + " : Keyword");
        } else if (isValidIdentifier(token)) {
            identifierList.add(token);
            System.out.println(token + " : Identifiers");
        } else if (isStringLiteral(token)) {
            stringLiteralList.add(token);
            System.out.println(token + " : String Literals");
        } else if (isNumber(token)) {
            numberList.add(token);
            System.out.println(token + " : Numbers");
        } else {
            otherList.add(token);
            System.out.println(token + " : Others");
        }
    }


    public static boolean isValidIdentifier(String word) {
        char firstChar = word.charAt(0);
        if (!Character.isLetter(firstChar) && firstChar != '_') {
            return false;
```

```
        } else{
            for (int i = 1; i < word.length(); i++) {
                char currentChar = word.charAt(i);
                if (!Character.isLetterOrDigit(currentChar) && currentChar !=
'_') {
                    return false;
                }
            }
            return true;
        }
    }

    public static boolean isStringLiteral(String token) {
        return token.startsWith("\"") && token.endsWith("\"");
    }

    public static boolean isNumber(String word) {
        try {
            Double.parseDouble(word);
            return true;
        } catch (NumberFormatException e) {
            return false;
        }
    }
}
```

input.java

```
// Smit Padshala
// 21BCP187
public class input {
    public static void main ( String [ ] args ) {
        int d = 7 , s = 10 ;
        float r ;
        r = d + s ;
        String str = "Hello World";
        System . out . println ( "Value of r is: " + r ) ;
    }
}
```

Output:

```
// : Others
Smit : Identifiers
Padshala : Identifiers
// : Others
21BCP187 : Others
public : Keyword
class : Keyword
input : Identifiers
{ : Others
public : Keyword
static : Keyword
void : Keyword
main : Identifiers
```

```
( : Others
String : Identifiers
[ : Others
] : Others
args : Identifiers
) : Others
{ : Others
int : Keyword
d : Identifiers
= : Others
7 : Numbers
, : Others
s : Identifiers
= : Others
10 : Numbers
```

```
; : Others
float : Keyword
r : Identifiers
; : Others
r : Identifiers
= : Others
d : Identifiers
+ : Others
s : Identifiers
; : Others
String : Identifiers
str : Identifiers
= : Others
```

```
String Literal: Hello World"
; : Others
System : Identifiers
. : Others
out : Identifiers
. : Others
println : Identifiers
( : Others
String Literal: Value of r is: "
+ : Others
r : Identifiers
) : Others
; : Others
} : Others
} : Others
```

```
Keywords: [public, class, public, static, void, int, float]
Identifiers: [Smit, Padshala, input, main, String, args, d, s, r, r, d, s, String, str, System, out, println, r]
String Literals: [Hello World", Value of r is: "]
Numbers: [7, 10]
Others: [//, //, 21BCP187, {, (, [, ], ), {, =, ,, =, ;, ;, =, +, ;, =, ;, ., ., (, +, ), ;, }, }]
PS D:\sem5\Compiler Lab\Lab 1>
```

Practical: 2

Aim:

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();
}
```

Output:

```
D:\sem5\Compiler Lab\Lab 2>flex 21BCP187_lex2a.l

D:\sem5\Compiler Lab\Lab 2>gcc lex.yy.c

D:\sem5\Compiler Lab\Lab 2>a.exe
int a, b = 5;
int is Token having length = 3
a, is Token having length = 2
b is Token having length = 1
= is Token having length = 1
5; is Token having length = 2
No. of tokens generated are: 5
```

Aim:

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|goto
|if|int|long|register|return|short|signed|sizeof|static|struct|switch|typedef|union|unsig
ned|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:

```
D:\sem5\Compiler Lab\Lab 2>flex 21BCP187_lex2b.l
 2>gcc lex.yy.c
                         2>gcc lex.yy.c
D:\sem5\Compiler Lab\Lab
 2>a.exe                 2>a.exe
void main() {
The length of keyword void: 4
The length of Other   is: 1
The length of identifier main is: 4
The length of Other ( is: 1
The length of Other ) is: 1
The length of Other   is: 1
The length of Other { is: 1

int a = 187;
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 Other   is: 1
The length of digit 187 is: 3
The length of Other ; is: 1
```

```
string name = "Smit Padshala";
The length of identifier string is: 6
The length of Other   is: 1
The length of identifier name is: 4
The length of Other   is: 1
The length of Other = is: 1
The length of Other   is: 1
The length of Other " is: 1
The length of identifier Smit is: 4
The length of Other   is: 1
The length of identifier Padshala is: 8
The length of Other " is: 1
The length of Other ; is: 1

return name;
The length of keyword return: 6
The length of Other   is: 1
The length of identifier name is: 4
The length of Other ; is: 1

}
The length of Other } is: 1

^C
D:\sem5\Compiler Lab\Lab 2>
```

Practical: 3

Aim:

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;
}
```

Output:

```
D:\sem5\Compiler Lab\Lab 2>cd D:\sem5\Compiler Lab\Lab
 3

>flex 21BCP187_prog3a.l

D:\sem5\Compiler Lab\Lab 3>gcc lex.yy.c

D:\sem5\Compiler Lab\Lab 3>a.exe
void main() {
printf("Hello");
// printing Hello
/* this is multiline
comment*/
return 0;
}

D:\sem5\Compiler Lab\Lab 3>
```

Lab 3 > C output.c > ...

```
void main() {
printf("Hello");


return 0;
}

```

Lab 3 > comments.txt

```
// printing Hello
/* this is multiline
comment*/

```

Aim:

b. 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("input.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);
}
```

Lab 3 > input.txt

```
Hello World!
I'm Smit Padshala
21BCP187
```

Output:

```
D:\sem5\Compiler Lab\Lab 3>flex 21BCP187_prog3b.l

D:\sem5\Compiler Lab\Lab 3>gcc lex.yy.c

D:\sem5\Compiler Lab\Lab 3>a.exe
Character count: 36
Word count: 6
Line count: 2
```

Aim:

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:

```
D:\sem5\Compiler Lab\Lab 3>flex 21BCP187_prog3c.l

D:\sem5\Compiler Lab\Lab 3>gcc lex.yy.c

D:\sem5\Compiler Lab\Lab 3>a.exe
49
52
5
5
7
10
14
17
6
6
```

Practical: 4

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

Code:

```
// 21BCP187
// Smit Padshala
import java.util.*;
class RecursiveDescentParser {
    static int ptr;
    static char[] input;
    public static void main(String args[]) {
        System.out.println("Enter the input string:");
        Scanner sc = new Scanner(System.in);
        String s = sc.nextLine();
        input = s.toCharArray();
        if (input.length < 1) {
            System.out.println("The input string is invalid.");
            System.exit(0);
        }
        ptr = 0;
        boolean isValid = E();
        if ((isValid) & (ptr == input.length)) {
            System.out.println("The input string is valid.");
        } else {
            System.out.println("The input string is invalid.");
        }
    }
    static boolean E() {
        int fallback = ptr;
        if (T()) {
            if (EPrime()) {
                return true;
            }
        }
        ptr = fallback;
        return false;
    }
    static boolean EPrime() {
        int fallback = ptr;
        if (ptr < input.length && (input[ptr] == '+' || input[ptr] == '-')) {
            ptr++;
            if (T()) {
```

```
                if (EPrime()) {
                    return true;
                }
            }
            ptr = fallback;
            return false;
        }
        return true;
    }
    static boolean T() {
        int fallback = ptr;
        if (F()) {
            if (TPrime()) {
                return true;
            }
        }
        ptr = fallback;
        return false;
    }
    static boolean TPrime() {
        int fallback = ptr;
        if (ptr < input.length && (input[ptr] == '*' || input[ptr] == '/')) {
            ptr++;
            if (F()) {
                if (TPrime()) {
                    return true;
                }
            }
            ptr = fallback;
            return false;
        }
        return true;
    }
    static boolean F() {
        int fallback = ptr;
        if (P()) {
            if (FPrime()) {
                return true;
            }
        }
        ptr = fallback;
        return false;
    }

    static boolean FPrime() {
```

```java
        int fallback = ptr;
        if (ptr < input.length && input[ptr] == '^') {
            ptr++;
            if (F()) {
                return true;
            }
            ptr = fallback;
            return false;
        }
        return true;
    }

    static boolean P() {
        int fallback = ptr;
        if (ptr < input.length && input[ptr] == '(') {
            ptr++;
            if (E()) {
                if (ptr < input.length && input[ptr] == ')') {
                    ptr++;
                    return true;
                }
            }
            ptr = fallback;
            return false;
        } else if (ptr < input.length && input[ptr] == 'i') {
            ptr++;
            return true;
        }
        return false;
    }
}
```

Output:

```
Enter the input string:
i+i*i/i*(i+i*i)^i
The input string is valid.
PS D:\sem5\Compiler Lab\Lab 4>
 RecursiveDescentParser }
Enter the input string:
i+i*i*i^i+i)
The input string is invalid.
```

Practical: 5

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

Code:

```
# 21BCP187

# Smit Padshala

F = {}
Fo = {}
non_term = set()
term = set()
# Function to compute FIRST set for a non-terminal
def first_set(nt):
    if F.get(nt):
        return F[nt]

    F[nt] = set()
    for prod in grammar[nt]:
        for sym in prod:
            if sym in term:
                F[nt].add(sym)
                break
            elif sym == '@':
                F[nt].add('@')
                break
            else:
                F[nt].update(first_set(sym))
                if '@' not in F[sym]:
                    break
    return F[nt]
# Function to compute FOLLOW set for a non-terminal
def follow_set(nt):
    if Fo.get(nt):
        return Fo[nt]

    Fo[nt] = set()
    if nt == start_symbol:
        Fo[nt].add('$')
    for n, prods in grammar.items():
        for prod in prods:
            for i, sym in enumerate(prod):
                if sym == nt:
```

```
                    if i < len(prod) - 1:
                        next_sym = prod[i + 1]
                        if next_sym in term:
                            Fo[nt].add(next_sym)
                        else:
                            F_next = first_set(next_sym)
                            Fo[nt].update(F_next.difference({'@'}))
                            if '@' in F_next:
                                Fo[nt].update(follow_set(n))
                    else:
                        Fo[nt].update(follow_set(n))
    return Fo[nt]
try:
    print("Enter Details of LL1 Grammar.\nEntered Grammar should be LL1")
    t_count = int(input("Enter the number of terminals: "))
    print("Enter the terminals:")
    term = set(input() for _ in range(t_count))
    nt_count = int(input("Enter the number of non-terminals: "))
    print("Enter the non-terminals:")
    non_term = set(input() for _ in range(nt_count))
    start_symbol = input("Enter the starting symbol: ")
    p_count = int(input("Enter the number of productions: "))
    print("Enter the productions in the format NonTerminal -> Production1 |
Production2 | ...")
    grammar = {}
    for nt in non_term:
        grammar[nt] = set()
    for _ in range(p_count):
        p_input = input()
        if '->' in p_input:
            nt, prods = p_input.split('->')
            nt = nt.strip()
            prods = prods.split('|')
            grammar[nt] = grammar[nt].union([p.strip() for p in prods])
        else:
            print(f"Invalid production: {p_input}. Use 'NonTerminal ->
Production1 | Production2' format.")
            continue

    # Compute FIRST and FOLLOW sets
    for nt in non_term:
        first_set(nt)
        follow_set(nt)
    print("\nFIRST sets:")
    for nt in non_term:
```

```
        print(f'FIRST({nt}) = {sorted(list(F[nt]))}')
    print("\nFOLLOW sets:")
    for nt in non_term:
        print(f'FOLLOW({nt}) = {sorted(list(Fo[nt]))}')
except Exception as e:
    print(f"An error occurred: {e}")
```

Output:

```
Enter Details of LL1 Grammar.
Entered Grammar should be LL1
Enter the number of terminals: 2
Enter the terminals:
a
b
Enter the number of non-terminals: 3
Enter the non-terminals:
S
A
B
Enter the starting symbol: S
Enter the number of productions: 3
Enter the productions in the format NonTerminal -> Production1 |
 Production2 | ...
S -> AaAb | BbBa
A -> @
B -> @

FIRST sets:
FIRST(B) = ['@']
FIRST(S) = ['@', 'a', 'b']
FIRST(A) = ['@']

FOLLOW sets:
FOLLOW(B) = ['a', 'b']
FOLLOW(S) = ['$']
FOLLOW(A) = ['a', 'b']
```

Practical: 6

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

Code:

```
# Smit Padshala
# 21BCP187

from tabulate import tabulate

firstop = {}
lastop = {}
productions = []
prod_dict = {}
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:")
```

```python
for _ in range(no_of_productions):
    productions.append(input())


for nT in non_terminals:
    prod_dict[nT] = []

for production in productions:
    nonterm_to_prod = production.split("->")
    alternatives = nonterm_to_prod[1].split("|")
    for alternative in alternatives:
        prod_dict[nonterm_to_prod[0]].append(alternative)

print("Populated prod_dict:")
for non_terminal, prods in prod_dict.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 prod_dict[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 prod_dict[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:")
```

```python
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 prod_dict[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 prod_dict[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 prod_dict[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] = '='
```

```python
# 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)
```

Output:

```
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
Populated productions_dict:
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']
Firstopp:
Firstopp(S) = {x}
Firstopp(A) = {a, q}
Firstopp(B) = {q}
Lastopp:
Lastopp(S) = {z, y}
Lastopp(A) = {a, q, S}
Lastopp(B) = {q}
Firstop:
Firstop(S) = {x}
Firstop(A) = {a, q}
Firstop(B) = {q}
Lastop:
Lastop(S) = {z, y}
Lastop(A) = {z, q, y, a}
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 expression is accepted.
```

Practical: 7

Aim:

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

Code:

Lex7a.l

```
%option noyywrap
%{
    #include "y.tab.h"
    extern int yylval;
%}

%%
[0-9]+      { yylval = atoi(yytext);  return NUMBER; }
[ \t] ;           /* ignore whitespace */
\n    return 0;   /* logical EOF */
.     return yytext[0];
%%
```

Yacc7a.y

```
%{
```

```
#include <stdio.h>

#include <math.h>

%}


%token NAME num

%%


S: E      { printf("Result: %d\n", $1); }

        | S E { printf("Result: %d\n", $2); }

        ;


E: E '+' E { $$ = $1 + $3; }

         | E '-' E { $$ = $1 - $3; }

         | E '*' E { $$ = $1 * $3; }

         | E '/' E { $$ = $1 / $3; }

         | E '^' E { $$ = pow($1, $3); }

         | '(' E ')'      { $$ = $2; }

         | num                 { $$ = $1; }

         ;
```

```
%%


int main(){

    yyparse;

}
```

Output:

```
D:\sem5\Compiler Lab\Lab 7>bison yacc7a.y
yacc7a.y: conflicts: 25 shift/reduce
```

Yacc7a.output

Lab 7 > yacc7a.output

```
Terminals unused in grammar

   NAME


State 14 conflicts: 5 shift/reduce
State 15 conflicts: 5 shift/reduce
State 16 conflicts: 5 shift/reduce
State 17 conflicts: 5 shift/reduce
State 18 conflicts: 5 shift/reduce

```

Aim:

b. Write a YACC program for desktop calculator with ambiguous grammar and additional information.

Code:

Yacc7b.y

```
%{
#include <stdio.h>
#include <math.h>
%}


%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 {$$ = $1 * $3;}
| Ex '/' Ex {if($3 == 0)
                    yyerror("divide by zero");
                  else
                    $$ = $1 / $3;
                  }
| Ex '^' Ex {$$ = pow($1,$3);}
| '-' Ex %prec UMINUS {$$ = -$2;}
| '(' Ex ')' {$$ = $2;}
| num {$$ = $1;}
;
%%
int main() {
   yyparse();
   return 0;
}
```

Yacc7b.l

```
%{
#include "yacc7b.tab.h"
%}
```

```
%%
[0-9]+ { yylval = atoi(yytext); return num; }
[ \t] ; /* Ignore whitespace */
\n return 0; /* Logical EOF */
. return yytext[0];
%%
int yywrap() {
    return 1;
}
void yyerror(char *s) {
    printf("error");
}
```

Output:

Lab 7 > yacc7b.output

```
Grammar

    0 $accept: s $end

    1 s: NAME '=' Ex
    2  | Ex

    3 Ex: Ex '+' Ex
    4   | Ex '-' Ex
    5   | Ex '*' Ex
    6   | Ex '/' Ex
    7   | Ex '^' Ex
    8   | '-' Ex
    9   | '(' Ex ')'
   10   | num
```

```
D:\sem5\Compiler Lab\Lab 7>bison -d -v yacc7b.y

D:\sem5\Compiler Lab\Lab 7>flex yacc7b.l

D:\sem5\Compiler Lab\Lab 7>gcc -o parserb lex.yy.c yacc7b.tab.c -lm
yacc7b.tab.c: In function 'yyparse':
yacc7b.tab.c:599:16: warning: implicit declaration of function 'yyle
 # define YYLEX yylex ()
                ^
yacc7b.tab.c:1244:16: note: in expansion of macro 'YYLEX'
       yychar = YYLEX;
                ^~~~~
yacc7b.y:19:33: warning: implicit declaration of function 'yyerror'
                                 yyerror("divide by zero");
                                 ^~~~~~~

D:\sem5\Compiler Lab\Lab 7>parser.exe
5+10+7+8
= 30

D:\sem5\Compiler Lab\Lab 7>parser.exe
3+9*5/56*+0+11
error
```

Aim:

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.

Code:

Yacc7c.y

```
%{
#include <stdio.h>
#include <math.h>
%}


%token NUMBER
%right '^'
%%
statement: E { printf("Result: %d\n", $1); }
        | statement E { printf("Result: %d\n", $2); }
```

```
        ;

E : E '+' T { $$ = $1 + $3; }
  | E '-' T { $$ = $1 - $3; }
  | T { $$ = $1; }
  ;

T : T '*' F { $$ = $1 * $3; }
  | T '/' F { if ($3 == 0) yyerror("division by zero"); else $$ = $1 / $3; }
  | F { $$ = $1; }
  ;

F : P '^' F { $$ = pow($1, $3); }
  | P { $$ = $1; }
  ;

P : '(' E ')' { $$ = $2; }
  | NUMBER { $$ = $1; }
  ;
%%
```

```
int main() {
    yyparse();
    return 0;
}
```

Yacc7.l

```
%{
#include "yacc7c.tab.h"
%}
%%
[0-9]+ { yylval = atoi(yytext); return NUMBER; }
[ \t] ; /* Ignore whitespace */
\n return 0; /* Logical EOF */
. return yytext[0];
%%
int yywrap() {
    return 1;
}
void yyerror(char *s) {
```

```
    printf("Syntax error\n");
}
```

Output:

```
D:\sem5\Compiler Lab\Lab 7>bison -d -v yacc7c.y

D:\sem5\Compiler Lab\Lab 7>flex yacc7c.l

D:\sem5\Compiler Lab\Lab 7>gcc -o parserc lex.yy.c yacc7c.tab.c -lm
yacc7c.tab.c: In function 'yyparse':
yacc7c.tab.c:594:16: warning: implicit declaration of function 'yylex
claration]
 # define YYLEX yylex ()
                ^
yacc7c.tab.c:1239:16: note: in expansion of macro 'YYLEX'
       yychar = YYLEX;
                ^~~~~
yacc7c.y:19:36: warning: implicit declaration of function 'yyerror' [
X;
                ^~~~3 == 0) yyerror("division by zero"); else $$ = $1
~                                    ^~~~~~~
yacc7c.y:19:36: warn
ing: implicit declaration of function 'yyerror' [-Wimplicit-function-
   | T '/' F { if ($3 == 0) yyerror("division by zero"); else $$ = $1
                                    ^~~~~~~

D:\sem5\Compiler Lab\Lab 7>parserc.exe
5+4*8-7/1^0+1
Result: 31
```