Pandit Deendayal Energy University, Gandhinagar

School of Technology

Department of Computer Science & Engineering

Compiler Design and System Software (20CP302P)

Name: Ravi Jamanbhai Makwana
Enrolment No: 21BCP418
Semester: 5
Division: 6
Group: G12
Branch: Computer Science

INDEX

S. No.	List of experiments	Date	Sign
1	Write C/C++ program to identify keywords, identifiers, and others from the given input file.	28/07/23	
2	a. Write a LEX program to count the number of tokens and display each token with its length in the given statement. b. Write a LEX program to identify keywords, identifiers, numbers, and other characters and generate tokens for each.	04/08/23	
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 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 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 input.c.	11/08/23, 18/08/23	
4	WAP to implement Recursive Decent Parser (RDP) parser for given grammar.	25/08/23	
5	Write a program to calculate first and follow of a given LL (1) grammar.	01/09/23	
6	WAP to construct operator precedence parsing table for the given grammar and check the validity of the string.	08/09/23	
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-> E+T | T T-> T*F | F F-> P^F | P P->(E) | id And parse the sentence: id + id * id.	27/10/23	

Practical 1

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

Code:

compiler.c

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>

// Function to check if a word is a keyword
int isKeyword(const char *word) {
    char keywords[][20] = {
        "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"
    };

    for (int i = 0; i < sizeof(keywords) / sizeof(keywords[0]); i++) {
        if (strcmp(word, keywords[i]) == 0) {
            return 1; // Found as a keyword
        }
    }
    return 0; // Not a keyword
}

// Function to check if a word is an identifier
int isIdentifier(const char *word) {
    if ((word[0] >= 'a' && word[0] <= 'z') || (word[0] >= 'A' && word[0] <= 'Z') || word[0] == '_') {
        // Check the rest of the characters in the word
        for (int i = 1; i < strlen(word); i++) {
            if (!((word[i] >= 'a' && word[i] <= 'z') || (word[i] >= 'A' && word[i] <= 'Z')
                  || (word[i] >= '0' && word[i] <= '9') || word[i] == '_')) {
                return 0; // Not a valid identifier
            }
        }
        return 1; // Valid identifier
    }
    return 0; // Not an identifier
}
```

```
// Function to check if a word is a string literal
bool isStringLiteral(const char *word) {
    return (word[0] == '"' && word[strlen(word) - 1] == '"');
}

// Function to check if a word is a number
bool isNumber(const char *word) {
    char *endptr;
    strtod(word, &endptr);
    return (*endptr == '\0'); // If endptr points to the null terminator, it's a valid number.
}

int main() {
    FILE *file = fopen("input.c", "r");
    if (file == NULL) {
        printf("Error opening the file.\n");
        return 1;
    }

    char word[100];
    while (fscanf(file, "%s", word) == 1) {
        // Check if the word is a keyword
        if (isKeyword(word)) {
            printf("%s is a keyword.\n", word);
        }
        // Check if the word is an identifier
        else if (isIdentifier(word)) {
            printf("%s is an identifier.\n", word);
        }
        // Check if the word is a string literal
        else if (isStringLiteral(word)) {
            printf("%s is a string literal.\n", word);
        }
        // Check if the word is a number
        else if (isNumber(word)) {
            printf("%s is a number.\n", word);
        }
        // If not a keyword, identifier, string literal, or number, classify as others
        else {
            printf("%s is classified as others.\n", word);
        }
    }

    fclose(file);
    return 0;
}
```

Input.c

```
int main ( ) {
   int var1 = 18 ;
   char str [ 10 ] = "ravi" ;
   if ( var1 == 18 ) {
      printf( "Yes" ) ;
   }
}
```

Output:

```
int is a keyword.
main is an identifier.
( is classified as others.
) is classified as others.
{ is classified as others.
int is a keyword.
var1 is an identifier.
= is classified as others.
18 is a number.
; is classified as others.
char is a keyword.
str is an identifier.
[ is classified as others.
10 is a number.
] is classified as others.
= is classified as others.
"ravi" is a string literal.
; is classified as others.
if is a keyword.
( is classified as others.
var1 is an identifier.
== is classified as others.
18 is a number.
) is classified as others.
{ is classified as others.
printf( is classified as others.
"Yes" is a string literal.
) is classified as others.
; is classified as others.
} is classified as others.
} is classified as others.
```

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:

a.l :

```
%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:

```
PS D:\CompilerDesign\Lab-2> flex final_2a.l
PS D:\CompilerDesign\Lab-2> gcc lex.yy.c
PS D:\CompilerDesign\Lab-2> ./a.exe
float n , x = 0 ;
float is Token having length = 5
n is Token having length = 1
, is Token having length = 1
x is Token having length = 1
= is Token having length = 1
0 is Token having length = 1
; is Token having length = 1
No. of tokens generated are: 7
```

b. Write a LEX program to identify keywords, identifiers, numbers and other characters and generate tokens for each.

Code:

b.l :

```
%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|lo
ng|register|return|short|signed|sizeof|static|struct|switch|typedef|union|unsigned|void|volatile|w
hile                {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:

```
PS D:\CompilerDesign\Lab-2> flex final_2b.l
PS D:\CompilerDesign\Lab-2> gcc lex.yy.c
PS D:\CompilerDesign\Lab-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=5;
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 digit 5 is: 1
The length of Other ; is: 1
```

<u>Practical 3</u>

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:

a.l:

```
%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:

```
#include <stdio.h>

int main()
{
    // Initialize
    int a = 0;
    /*
    int b = 10;
    int c = 20;
    */
}
```

Output:

output.c:

Lab-3 > output.c > main()

```
#include <stdio.h>

int main()
{

    int a = 0;

}
```

comments.txt:

Lab-3 > comments.txt

```
// Initialize
/*
    int b = 10;
    int c = 20;
    */

```

b. Write a LEX program to count the number of characters, words, and lines in the given input.

Code:

b.l:

```
%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);
}
```

input.txt:

Hello there
Ravi Makwana
21BCP418

Output:

```
PS D:\CompilerDesign\Lab-3> flex final_3b.l
PS D:\CompilerDesign\Lab-3> gcc lex.yy.c
PS D:\CompilerDesign\Lab-3> ./a.exe
Character count: 30
Word count: 5
Line count: 2
PS D:\CompilerDesign\Lab-3> 
```

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

Code:

c.l:

```
%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 D:\CompilerDesign\Lab-3> flex final_3c.l
PS D:\CompilerDesign\Lab-3> gcc lex.yy.c
PS D:\CompilerDesign\Lab-3> ./a.exe
56
59
23
23
28
31
7
10
5
5
PS D:\CompilerDesign\Lab-3>
```

Practical 4

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

Code:

```
// null is symbolized by '^'

/*rules:
      S->cAd
      A->aA'
      A'->cA'| null
*/
#include <bits/stdc++.h>

using namespace std;

string st = "caccccc^d";
int stringPointer = 0;

//method: if you encounter a terminal, then increment the stringPointer then and there only
//      but if you encounter a variable, call its function, inside which you can increment the stringPointer
ONLY after if conditions, not in the start!
//      if you call a function, and it returns back the control, then don't increment the stringPointer, as it
is


//function prototypes
bool S(string st);
bool A(string st);
bool A_dash(string st);




bool S(string st){
   if (st[stringPointer] == 'c'){
      stringPointer ++;
      if (A(st) == true){
         if (st[stringPointer] == 'd'){
            stringPointer++;
            return true;
         }
         else return false;
      }
      else return false;
   }
```

```
    else return false;
}

bool A(string st){
    if (st[stringPointer] == 'a'){
        stringPointer++;
        if (A_dash(st) == true){
            return true;
        }
        else return false;
    }
    else return false;
}

bool A_dash(string st){
    if (st[stringPointer] == 'c'){
        stringPointer++;
        if (A_dash(st) == true){
            return true;
        }
        else{
            return false;
        }
    }
    else if (st[stringPointer] == '^'){
        stringPointer++;
        return true;
    }
    else return false;
}

int main(){

    if (S(st) == true) cout<<"Accepted";
    else cout<<"Not accepted";
    return 0;
}
```

Output:

```
Accepted
PS D:\COLLEGE\System Software & Compiler Design Lab\prac4>
```

Practical 5

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

Code:

```
// To find First and Follow set of a given grammar

#include <bits/stdc++.h>
#include <map>
#include <vector>
#include <set>
using namespace std;


//map for first set of all Rules (of all heads)
map<string, set<char>> firstSet;

//map for follow set of all Rules (of all heads)
map<string, set<char>> followSet;

//map for grammar rules
//it stores alphabetic wise by default
map <string, set<string>> m = {
  {"A",{"a","^"}},
  {"B", {"b"}},
  {"S", {"aAb", "B"}}
};




//function that calculates the first set for an element, stores it in fs & returns it.
set<char> calculateFirst(string head, set<string> v){
  set<char> fs;

  // traversing list (rules) for the head variable
  for (auto j : v){
    char firstLetterOfRule = j[0];

    if (islower(firstLetterOfRule) || firstLetterOfRule == '^'){
      fs.insert(firstLetterOfRule);
    }

    else{
      string newHead = "";
```

```
        newHead +=firstLetterOfRule;

        set<string> newSet = m[newHead];

        set<char> tempSet = calculateFirst(newHead, newSet);
        for (auto i : tempSet){
          fs.insert(i);
        }
      }
    }
    return fs;
}

//function that calculates the follow set
void calculateFollow(string LHS, set<string> RHS){

    // traversing list (rules) for the head variable
    for (auto rule : RHS){
      int ruleTraverser = 0;
      int ruleLength = rule.length();

      while (ruleTraverser < ruleLength){
        if (isupper(rule[ruleTraverser])){

          //saving current character to a variable
          char tempHead = rule[ruleTraverser];
          string header = "";
          header += tempHead;

          //checking if it is the last character of the rule, in which case, we will take Follow of LHS
          if (ruleTraverser == ruleLength-1){

            set<char> tempSet = followSet[LHS];
            //now this will be the follow set of the character too
            for (auto ele : tempSet){
              followSet[header].insert(ele);
            }
          }
          else{
            //if the next character is a terminal (lowercase), simply add it to the Follow set
            if (islower(rule[ruleTraverser+1])){

              followSet[header].insert(rule[ruleTraverser+1]);
            }
            //if the next character is also a variable (uppercase), then its First will be the Follow set
            else {
              char nextCharacter0 = rule[ruleTraverser+1];
              string nextCharacter = "";
              nextCharacter += nextCharacter0;
```

```
                set<char> tempSet = firstSet[nextCharacter];
                //now this will be follow set of the character
                for (auto ele :tempSet){
                    followSet[header].insert(ele);
                }
            }
        }
    }
    ruleTraverser++;
  }

 }
}

int main(){

 //initializing iterator for map
 map<string, set<string>>:: iterator i;

 //calculating First set
 for (i=m.begin(); i!= m.end(); i++){
    set<char> fs = calculateFirst(i->first, i->second);
    // insert this first set into map of firstSet
    firstSet[i->first] = fs;
 }

 //calculating Follow set
 followSet["S"].insert('$');    //since start symbol's follow set always has '$'
 for (i=m.begin(); i!= m.end(); i++){
    calculateFollow(i->first, i->second);
 }


 //printing the first set
 cout<<"The Follow Set of each variable is: \n";
 map<string, set<char>>:: iterator it;
 for (it=firstSet.begin(); it!= firstSet.end(); it++){
    string head = it->first;
    cout<<head<<": ";
    for (auto ele : it->second){
        cout<<ele;
        if (ele != *prev(it->second.end())){
            cout<<",";
        }
    }
    cout<<endl;
 }
 cout<<endl;
```

```
    //printing the follow set
    cout<<"The Follow Set of each variable is: \n";
    map<string, set<char>>:: iterator it2;
    for (it2=followSet.begin(); it2!= followSet.end(); it2++){
        string head = it2->first;
        cout<<head<<": ";
        for (auto ele : it2->second){
            cout<<ele;
            if (ele != *prev(it2->second.end())){
                cout<<",";
            }
        }
        cout<<endl;
    }

    return 0;
}
```

Output:

```
The Follow Set of each variable is:
A: ^,a
B: b
S: a,b

The Follow Set of each variable is:
A: b
B: $
S: $
PS D:\COLLEGE\System Software & Compiler Design Lab\prac5>
```

Practical 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
# Take user input for the grammar
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: "))
productions = []
print("Enter the productions:")
for _ in range(no_of_productions):
    productions.append(input())

# Initialize Firstopp and Lastopp dictionaries
Firstopp = {}
Lastopp = {}

# Helper function to add symbols to Firstopp for a non-terminal


def add_to_Firstopp(non_terminal, symbol):
    if non_terminal not in Firstopp:
        Firstopp[non_terminal] = set()
    Firstopp[non_terminal].add(symbol)

# Helper function to add symbols to Lastopp for a non-terminal


def add_to_Lastopp(non_terminal, symbol):
    if non_terminal not in Lastopp:
        Lastopp[non_terminal] = set()
    Lastopp[non_terminal].add(symbol)


# Initialize the productions_dict
productions_dict = {}
```

```python
for nT in non_terminals:
    productions_dict[nT] = []

# Print the initialized productions_dict
# print("Initialized productions_dict:")
# for non_terminal, prods in productions_dict.items():
#     print(f"{non_terminal} -> {prods}")

# Parse the user input productions and build the productions_dict
for production in productions:
    nonterm_to_prod = production.split("->")
    alternatives = nonterm_to_prod[1].split("|")
    for alternative in alternatives:
        productions_dict[nonterm_to_prod[0]].append(alternative)

# Print the populated productions_dict
print("Populated productions_dict:")
for non_terminal, prods in productions_dict.items():
    print(f"{non_terminal} -> {prods}")

# Compute Firstopp for each non-terminal
for non_terminal in non_terminals:
    for production in productions_dict[non_terminal]:
        symbols = production.split()
        print(symbols)
        for symbol in symbols:
            if symbol in non_terminals:
                add_to_Firstopp(non_terminal, symbol)
            elif symbol in terminals:
                add_to_Firstopp(non_terminal, symbol)
                break

# Compute Lastopp for each non-terminal
for non_terminal in non_terminals:
    for production in productions_dict[non_terminal]:
        symbols = production.split()
        for symbol in reversed(symbols):
            if symbol in non_terminals:
                add_to_Lastopp(non_terminal, symbol)
            elif symbol in terminals:
                add_to_Lastopp(non_terminal, symbol)
                break

# Print the Firstopp and Lastopp sets
print("Firstopp:")
for non_terminal, first_set in Firstopp.items():
    print(f'Firstopp({non_terminal}) = {{{", ".join(first_set)}}}')

print("Lastopp:")
for non_terminal, last_set in Lastopp.items():
    print(f'Lastopp({non_terminal}) = {{{", ".join(last_set)}}}')

counter = 0
while counter < no_of_productions:
    for non_terminal, first_set in Firstopp.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:
                Firstopp[non_terminal] |= Firstopp[symbol]
```

```python
    counter += 1


# Remove non-terminals from Lastopp sets
counter = 0
while counter < no_of_productions:
    for non_terminal, last_set in Lastopp.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:
                Lastopp[non_terminal] |= Lastopp[symbol]
    counter += 1
# Remove non-terminals from Firstopp sets
for non_terminal, first_set in Firstopp.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 Lastopp sets
for non_terminal, last_set in Lastopp.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 Firstopp and Lastopp sets
print("Firstop:")
for non_terminal, first_set in Firstopp.items():
    print(f'Firstop({non_terminal}) = {{{", ".join(first_set)}}}')

print("Lastop:")
for non_terminal, last_set in Lastopp.items():
    print(f'Lastop({non_terminal}) = {{{", ".join(last_set)}}}')
# Create an empty matrix with rows and columns for terminals

# Add a dollar symbol ('$') to the terminals list
terminals.append('$')

# Create an empty matrix with rows and columns for terminals
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 Firstopp+ list of B
for non_terminal in non_terminals:
    for productions in productions_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 Firstopp[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 Lastopp+ list of C
for non_terminal in non_terminals:
    for productions in productions_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 Lastopp[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 productions_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] = '='

# Rule 4: Add relations $<· a and a ·> $ for all terminals in the Firstopp+ and Lastopp+
lists, respectively of S
for alpha in Firstopp[starting_symbol]:
    col_index = terminals.index(alpha)
    terminal_matrix[-1][col_index] = '<'
for beta in Lastopp[starting_symbol]:
    row_index = terminals.index(beta)
    terminal_matrix[row_index][-1] = '>'

dollar_index = terminals.index('$')
terminal_matrix[-1][dollar_index] = 'acc'
# Map symbols to printable representations

# Add a space for empty cells

# Create a list of lists for the table
table_data = []
for i in range(len(terminals)):
    row = [terminals[i]]
    row.extend([terminal_matrix[i][j] for j in range(len(terminals))])
    table_data.append(row)

# Add headers for columns
headers = [''] + terminals

# Print the table using tabulate
table = tabulate(table_data, headers, tablefmt="grid")

print("Operator Precedence Table:")
print(table)

# Define a function to parse an input expression using the operator precedence table


def parse_expression(expressions):
    stack = ['$']  # Initialize the stack with '$'
    expression = expressions.split()
    # Append '$' to the input expression
    input_buffer = list(expression) + ['$']
    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]

        # Find the indices of the top of the stack and the current input in the terminal
list
        top_stack_index = terminals.index(top_stack)
        current_input_index = terminals.index(current_input)

        # Get the relation from the operator precedence table
        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)]
            # stack.append(popped)
        elif relation == 'acc':
            print("Input expression is accepted.")
            return
        else:
            print("Input expression is not accepted.")
            return


# Input an expression to parse
while True:
    choice = int(input(
        "Enter 1 if you want to check if a string is accepted by a parser and 2 if you want
to exit"))
    expression_to_parse = input("Enter an expression to parse: ")
    parse_expression(expression_to_parse)
    if choice == 2:
        break
```

Output:

```
Enter no. of terminals: 8
Enter the terminals:
+
-
*
/
^
(
)
id
Enter no. of non-terminals: 4
Enter the non-terminals:
E
T
F
P
Enter the starting symbol: E
Enter no of productions: 4
Enter the productions:
E->E + T|E - T|T
T->T * F|T / F|F
F->P ^ F|P
P->( E )| id
Populated productions_dict:
E -> ['E + T', 'E - T', 'T']
T -> ['T * F', 'T / F', 'F']
F -> ['P ^ F', 'P']
P -> ['( E )', ' id']
['E', '+', 'T']
['E', '-', 'T']
['T']
['T', '*', 'F']
['T', '/', 'F']
['F']
['P', '^', 'F']
['P']
['(', 'E', ')']
['id']
Firstopp:
Firstopp(E) = {-, E, +, T}
Firstopp(T) = {F, *, /, T}
Firstopp(F) = {^, P}
Firstopp(P) = {id, (}
Lastopp:
```

```
Lastopp(E) = {-, +, T}
Lastopp(T) = {F, *, /}
Lastopp(F) = {F, ^, P}
Lastopp(P) = {), id}
Firstop:
Firstop(E) = {-, *, ^, id, +, /, (}
Firstop(T) = {^, id, *, /, (}
Firstop(F) = {^, id, (}
Firstop(P) = {id, (}
Lastop:
Lastop(E) = {^, id, -, ), *, +, /}
Lastop(T) = {^, id, ), *, /}
Lastop(F) = {), ^, id}
Lastop(P) = {), id}
Operator Precedence Table:
+----+-----+-----+-----+-----+-----+-----+------+-----+
|    | +   | -   | *   | /   | ^   | (   | )   | id   | $   |
+====+=====+=====+=====+=====+=====+=====+=====+======+=====+
| +  | >   | >   | <   | <   | <   | <   | >   | <    | >   |
+----+-----+-----+-----+-----+-----+-----+-----+------+-----+
| -  | >   | >   | <   | <   | <   | <   | >   | <    | >   |
+----+-----+-----+-----+-----+-----+-----+-----+------+-----+
| *  | >   | >   | >   | >   | <   | <   | >   | <    | >   |
+----+-----+-----+-----+-----+-----+-----+-----+------+-----+
| /  | >   | >   | >   | >   | <   | <   | >   | <    | >   |
+----+-----+-----+-----+-----+-----+-----+-----+------+-----+
| ^  | >   | >   | >   | >   | <   | <   | >   | <    | >   |
+----+-----+-----+-----+-----+-----+-----+-----+------+-----+
| (  | <   | <   | <   | <   | <   | <   | =   | <    |     |
+----+-----+-----+-----+-----+-----+-----+-----+------+-----+
| )  | >   | >   | >   | >   | >   |     | >   |      | >   |
+----+-----+-----+-----+-----+-----+-----+-----+------+-----+
| id | >   | >   | >   | >   | >   |     | >   |      | >   |
+----+-----+-----+-----+-----+-----+-----+-----+------+-----+
| $  | <   | <   | <   | <   | <   | <   |     | <    | acc |
+----+-----+-----+-----+-----+-----+-----+-----+------+-----+
Enter 1 if you want to check if a string is accepted by a parser and 2 if you want to exit1
Enter an expression to parse: id ^ id ^ ( id + id ) * id
['id', '^', 'id', '^', '(', 'id', '+', 'id', ')', '*', 'id', '$']
$
id
$
^
id
```

```
^
^
(
id
(
+
id
+
(
)
^
^
$
*
id
*
$
Input expression is accepted.
```

Practical 7

Aim:

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

Code:

prog1.l-

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

%%

[0-9]+ { yylval = atoi(yytext); return NUMBER; }
["\t"] ;
["\n"] return 0;
. return yytext[0];

%%

void yyerror(char *s)
{
   printf("%d: %s at %s\n", yylineno, s, yytext);
}
```

prog1.y-

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

%token NAME NUMBER
%%

statement: expression      { printf("Result: %d\n", $1); }
         | statement expression { printf("Result: %d\n", $2); }
         ;

expression: expression '+' expression { $$ = $1 + $3; }
```

```
        | expression '-' expression { $$ = $1 - $3; }
        | expression '*' expression { $$ = $1 * $3; }
        | expression '/' expression { $$ = $1 / $3; }
        | expression '^' expression { $$ = pow($1, $3); }
        | '(' expression ')'      { $$ = $2; }
        | NUMBER                  { $$ = $1; }
        ;

%%

int main(){
   yyparse;
}
```

Output:

```
PS D:\CompilerDesign\Lab-7\Yacc> flex prog1.l
PS D:\CompilerDesign\Lab-7\Yacc> bison -d prog1.y
prog1.y: conflicts: 25 shift/reduce
PS D:\CompilerDesign\Lab-7\Yacc> bison -d -v prog1.y
prog1.y: conflicts: 25 shift/reduce
PS D:\CompilerDesign\Lab-7\Yacc>
```

<u>prog1.output-</u>

```
Lab-7 > Yacc > prog1.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


Grammar

    0 $accept: statement $end

    1 statement: expression
    2          | statement expression

    3 expression: expression '+' expression
    4           | expression '-' expression
    5           | expression '*' expression
    6           | expression '/' expression
    7           | expression '^' expression
    8           | '(' expression ')'
    9           | NUMBER


Terminals, with rules where they appear

$end (0) 0
'(' (40) 8
')' (41) 8
'*' (42) 5
'+' (43) 3
'-' (45) 4
'/' (47) 6
'^' (94) 7
error (256)
NAME (258)
NUMBER (259) 9
```

b. Write a YACC program for desktop calculator with ambiguous grammar and additional information (Operator Precedence).

Code:

prog2.l-

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

%%

[0-9]+ { yylval = atoi(yytext); return NUMBER; }
["\t"] ;
["\n"] return 0;
. return yytext[0];

%%

void yyerror(char *s)
{
   printf("%d: %s at %s\n", yylineno, s, yytext);
}
```

prog2.y-

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

%token NAME NUMBER
%left '-' '+'
%left '*' '/'
%right '^'
%nonassoc UMINUS
%%

statement: expression      { printf("Result: %d\n", $1); }
        | statement expression { printf("Result: %d\n", $2); }
        ;

expression: expression '+' expression { $$ = $1 + $3; }
        | expression '-' expression { $$ = $1 - $3; }
```

```
        | expression '*' expression { $$ = $1 * $3; }
        | expression '/' expression { $$ = $1 / $3; }
        | expression '^' expression { $$ = pow($1, $3); }
        | '–' expression %prec UMINUS { $$ = –$2; }
        | '(' expression ')'      { $$ = $2; }
        | NUMBER                  { $$ = $1; }
        ;

%%

int main(){
    yyparse;
}
```

Output:

```
PS D:\CompilerDesign\Lab-7\Yacc2> flex prog2.l
PS D:\CompilerDesign\Lab-7\Yacc2> bison -d prog2.y
PS D:\CompilerDesign\Lab-7\Yacc2> bison -d -v prog2.y
PS D:\CompilerDesign\Lab-7\Yacc2>
```

prog2.output-

```
Lab-7 > Yacc2 > prog2.output
Terminals unused in grammar

   NAME


Grammar

    0 $accept: statement $end

    1 statement: expression
    2          | statement expression

    3 expression: expression '+' expression
    4           | expression '-' expression
    5           | expression '*' expression
    6           | expression '/' expression
    7           | expression '^' expression
    8           | '◆' expression
    9           | '(' expression ')'
   10           | NUMBER


Terminals, with rules where they appear

$end (0) 0
'(' (40) 9
')' (41) 9
'*' (42) 5
'+' (43) 3
'-' (45) 4
'/' (47) 6
'^' (94) 7
'◆' (226) 8
error (256)
NAME (258)
NUMBER (259) 10
UMINUS (260)


Nonterminals, with rules where they appear

```

c. Design, develop and implement a YACC program to demonstrate Shift Reduce Parsing technique for the grammar rules:

E-> E+T | T
T-> T*F | F
F-> P^F | P
P->(E) | id

And parse the sentence: id + id * id.

Code:

prog3.l-

```
%option noyywrap
%{
   #include "prog3.tab.h"
   #include <stdio.h>
   extern int yylval;
%}

%%

[0-9]+ { yylval = atoi(yytext); return NUMBER; }
["\t"] ;
["\n"] return 0;
. return yytext[0];

%%

void yyerror(char *s)
{
   printf("Error: %s at %s\n", s, yytext);
}
```

prog3.y-

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

%token NUMBER
```

```
%%

statement: E      { printf("Result: %d\n", $1); }
         | statement E { printf("Result: %d\n", $2); }
         ;

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

T : T '*' F { $$ = $1 * $3; }
  | F { $$ = $1; }
  ;

F : P '^' F { $$ = $1 ^ $3; }
  | P { $$ = $1; }
  ;

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

%%

int main(){
    yyparse;
}
```

Output:

```
PS D:\CompilerDesign\Lab-7\Yacc3> flex prog3.l
PS D:\CompilerDesign\Lab-7\Yacc3> bison -d prog3.y
PS D:\CompilerDesign\Lab-7\Yacc3> bison -d -v prog3.y
PS D:\CompilerDesign\Lab-7\Yacc3>
```

<u>prog3.output-</u>

Lab-7 > Yacc3 > prog3.output

```
Grammar

    0 $accept: statement $end

    1 statement: E
    2          | statement E

    3 E: E '+' T
    4  | T

    5 T: T '*' F
    6  | F

    7 F: P '^' F
    8  | P

    9 P: '(' E ')'
   10  | NUMBER


Terminals, with rules where they appear

$end (0) 0
'(' (40) 9
')' (41) 9
'*' (42) 5
'+' (43) 3
'^' (94) 7
error (256)
NUMBER (258) 10


Nonterminals, with rules where they appear

$accept (9)
    on left: 0
statement (10)
    on left: 1 2, on right: 0 2
E (11)
    on left: 3 4, on right: 1 2 3 9
T (12)
```