## 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** | 1. Write a LEX program to count the number of tokens and display each token with its length in the given statement. 2. Write a LEX program to identify keywords, identifiers, numbers, and other characters and generate tokens for each. | 04/08/23 |  |
| **3** | 1. 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 2. . Write a LEX program to count the number of characters, words and lines in the given input. 3. 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** | 1. Write a YACC program for desktop calculator with ambiguous grammar (evaluate arithmetic expression involving operators: +, -, \*, / and ^) 2. Write a YACC Program for desktop calculator with ambiguous grammar and additional information. 3. 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" ;

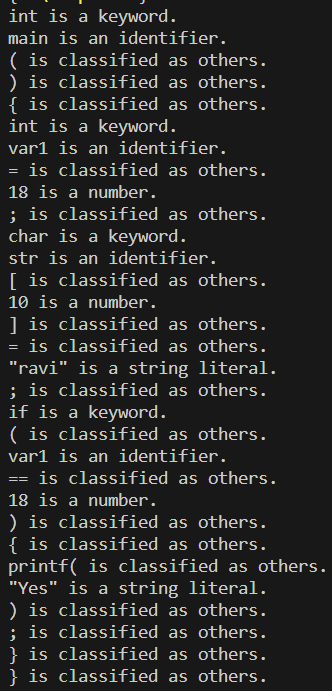
if ( var1 == 18 ) {

printf( "Yes" ) ;

}

}

**Output:**

****

**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);}

. ;

%%

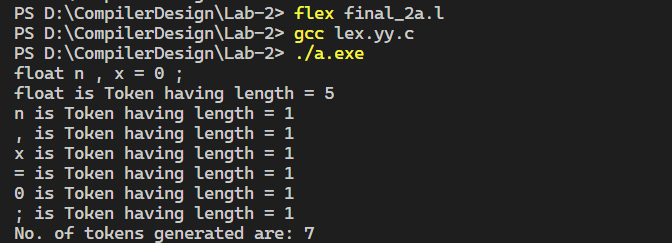
int main()

{

yylex();

}

**Output:**

****

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|long|register|return|short|signed|sizeof|static|struct|switch|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() {

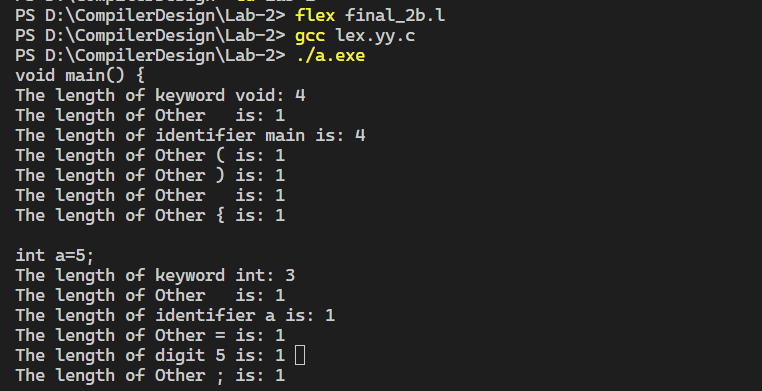
yylex();

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:**

****

**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:**

**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;

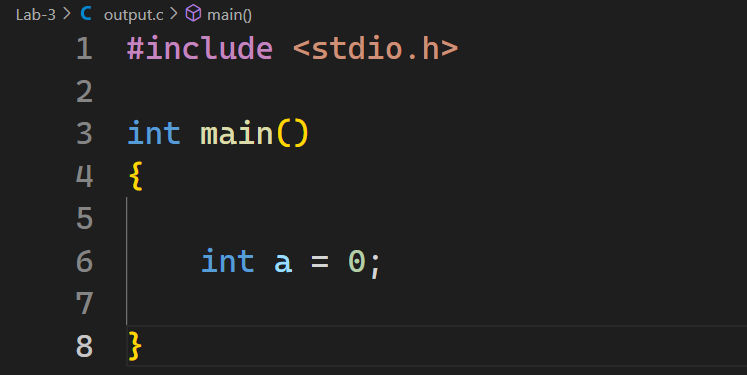
    int c = 20;

    \*/

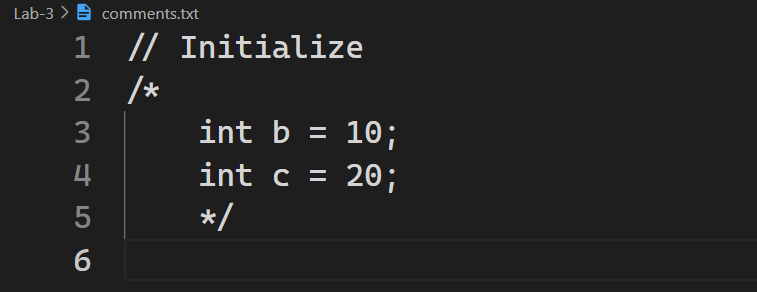
}

**Output:**

**output.c:**

****

**comments.txt:**

****

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);

}

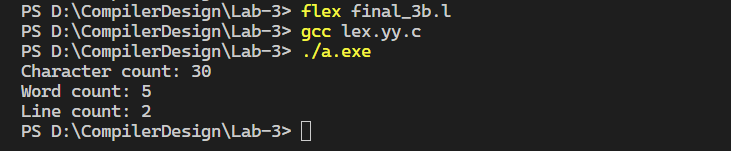
**input.txt:**

Hello there

Ravi Makwana

21BCP418

**Output:**



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

}

%%

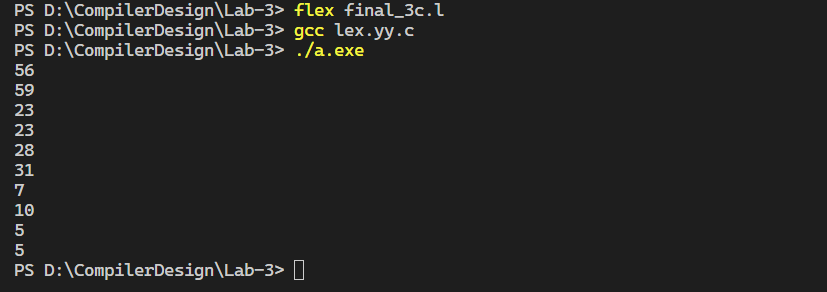
int main() {

yylex();

return 0;

}

**Output:**



**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 = "cacccc^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:**

****

**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<<",";

}

}

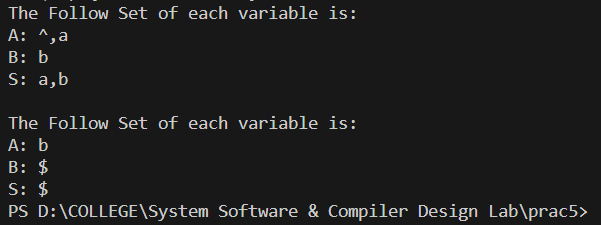
cout<<endl;

}

return 0;

}

**Output:**



**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 = {}

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]

    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"))

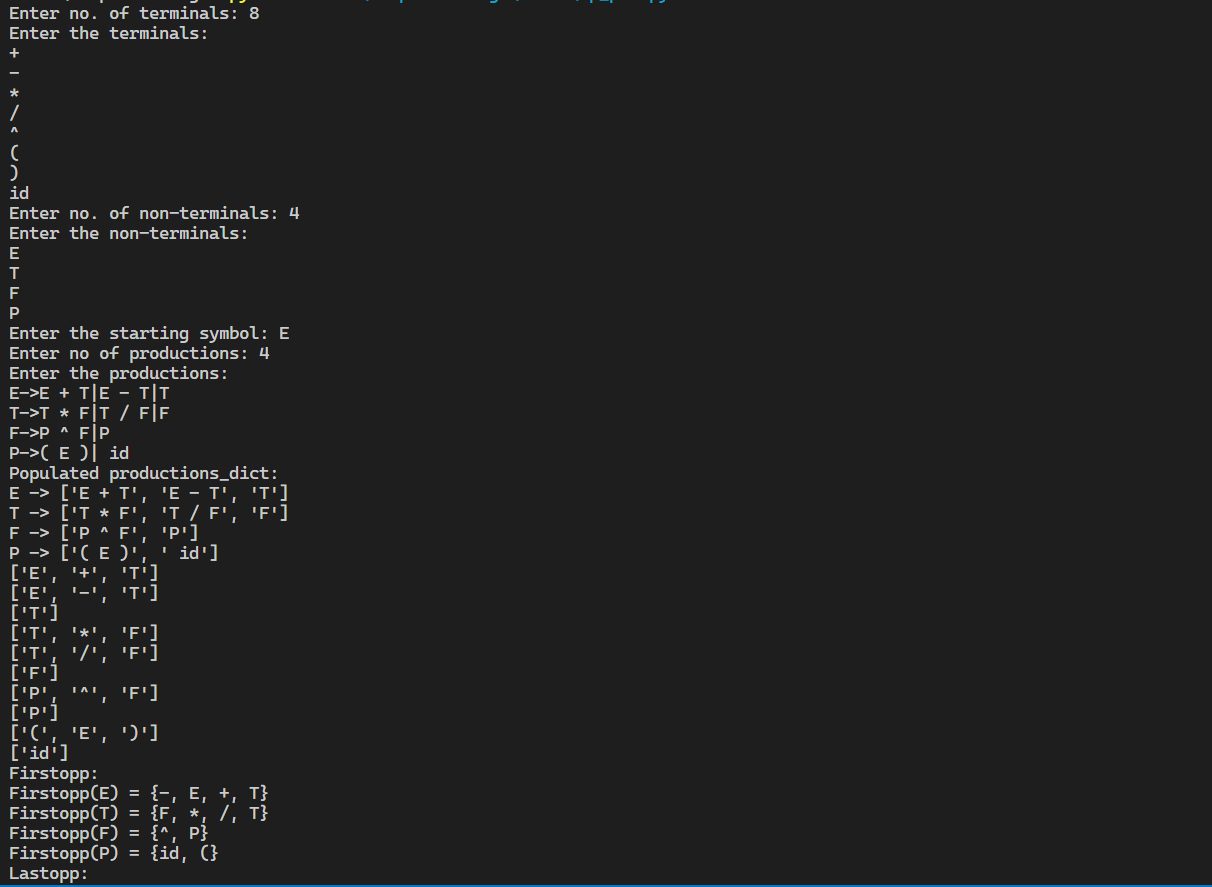
    expression\_to\_parse = input("Enter an expression to parse: ")

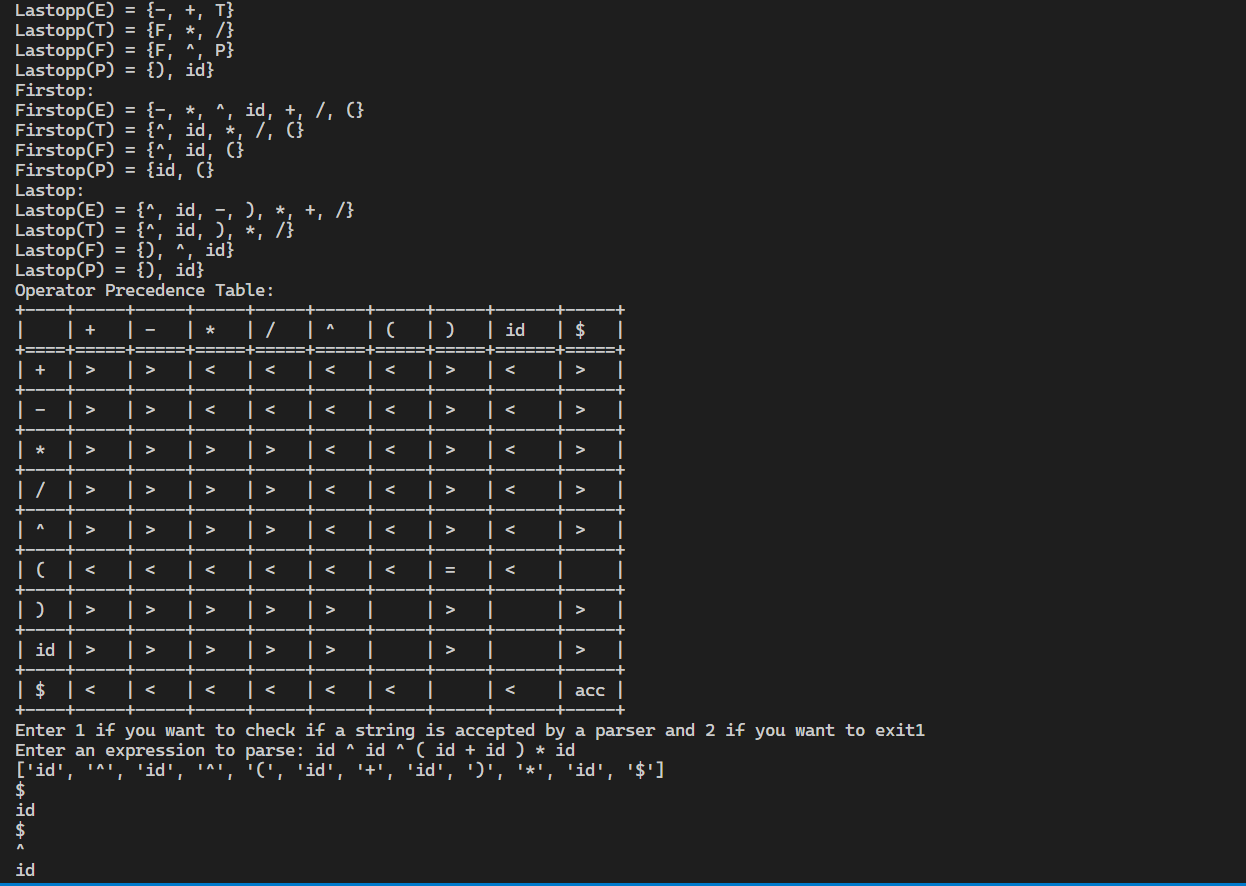
    parse\_expression(expression\_to\_parse)

    if choice == 2:

        break

**Output:**







**Practical 7**

**Aim:**

1. 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; }

;

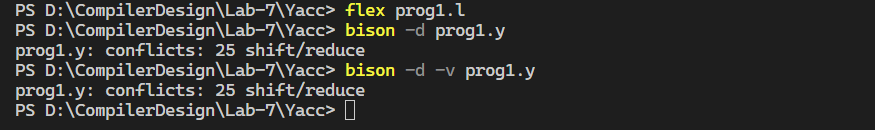
%%

int main(){

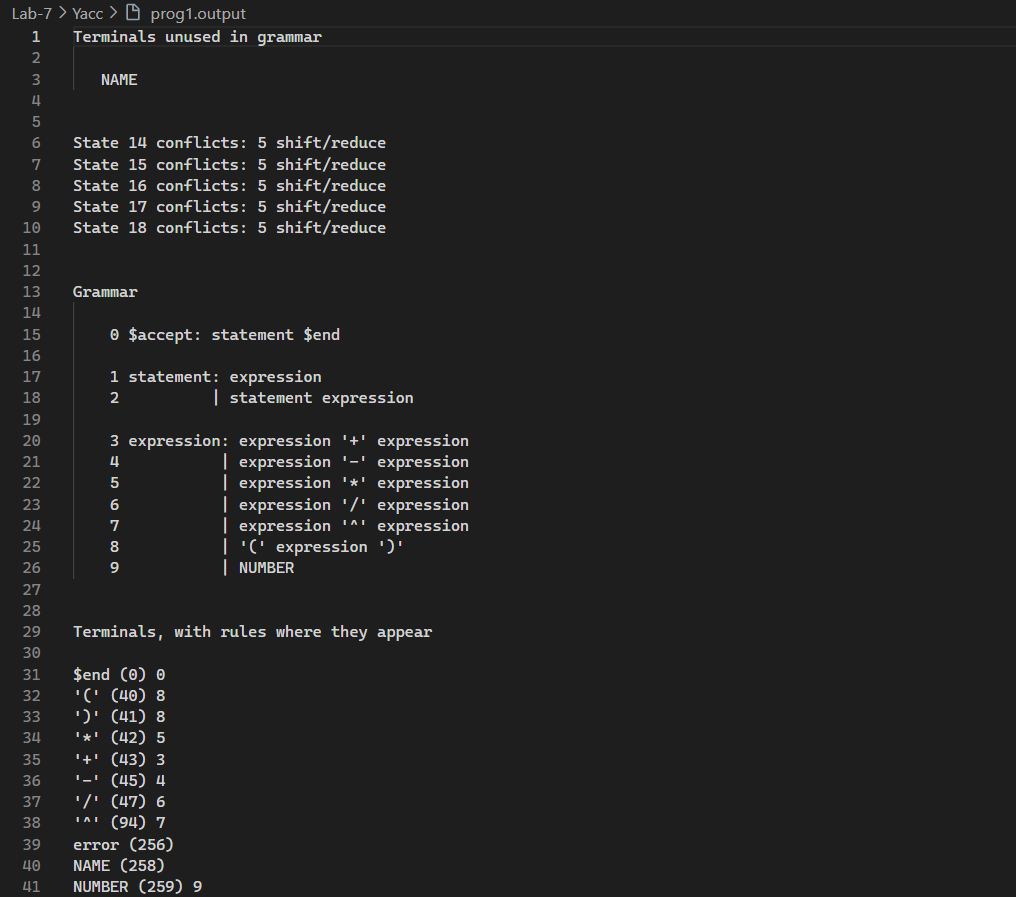
yyparse;

}

**Output:**



**prog1.output-**



1. 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; }

;

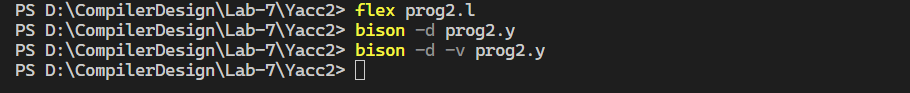
%%

int main(){

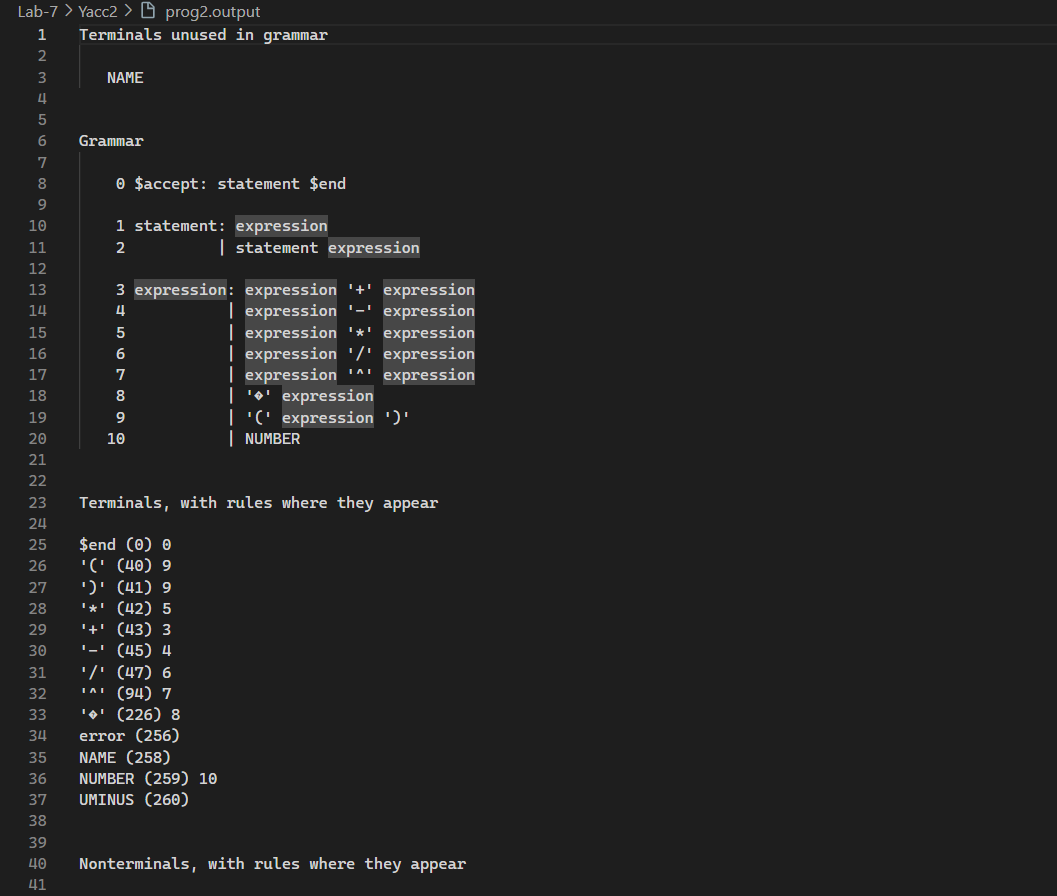
yyparse;

}

**Output:**



**prog2.output-**



1. 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; }

;

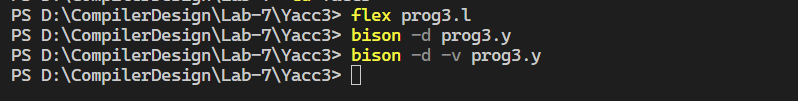
%%

int main(){

yyparse;

}

**Output:**



**prog3.output-**

