

Netaji Subhas University of Technology

|  |
| --- |
| COMPUTER SCIENCE AND ENGINEERING  THIRD YEAR  COCSC14 |

Principles of Compiler Construction

Submitted by:-

Laksh Sachdeva

2022UCS1572

|  |  |
| --- | --- |
| **Sr No.** | **PRACTICALS** |
| 1 | Implement a program for symbol table using hashing in C programming |
| 2 | Implement a two-pass assembler 8085/8086 |
| 3 | Develop Lexical Analyzer for ‘C’ using LEX tool |
| 4 | Represent C language using Context Free Language. |
| 5 | Develop a simple calculator using LEX and YACC tools. |
| 6 | Develop a Parser for ‘C’ using the LEX and YACC tools. |
| 7 | Add assignment statement , If then else statement and while loop to the calculator and generate the three address code for the same. |

INDEX

1. **Implement a program for symbol table using hashing in C programming**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MAX 11

struct symb

{

int add; // Address

char label[10]; // Label

} sy[MAX];

void search();

void Iprob(int a[], int key, int num);

int create(int num);

void display(int a[]);

int main()

{

int a[MAX], num, key, i, ch;

char ans;

// Initialize the symbol table

for (i = 0; i < MAX; i++)

{

a[i] = 0;

sy[i].add = 0;

strcpy(sy[i].label, "");

}

do

{

printf("\nEnter your choice:");

printf("\n1 - Create entry in the symbol table\n2 - Search in the symbol table\n");

scanf("%d", &ch);

switch (ch)

{

case 1:

do

{

printf("\nEnter the address: ");

scanf("%d", &num);

key = create(num);

printf("Enter the label: ");

scanf("%s", sy[key].label);

Iprob(a, key, num);

printf("\nContinue? 0 for no, 1 for yes: ");

scanf(" %c", &ans);

} while (ans == '1');

display(a);

break;

case 2:

search();

break;

default:

printf("Invalid choice!\n");

}

} while (ch <= 2);

return 0;

}

int create(int num)

{

int key = num % MAX;

return key;

}

void Iprob(int a[], int key, int num)

{

int i, flag = 0, count = 0;

if (a[key] == 0)

{

a[key] = num;

sy[key].add = num;

}

else

{

for (i = 0; i < MAX; i++)

{

if (a[i] != 0)

count++;

}

if (count == MAX)

{

printf("\nHash table is full");

display(a);

exit(1);

}

for (i = key + 1; i < MAX; i++)

{

if (a[i] == 0)

{

a[i] = num;

flag = 1;

sy[i].add = num;

break;

}

}

if (flag == 0)

{

for (i = 0; i < key; i++)

{

if (a[i] == 0)

{

a[i] = num;

sy[i].add = num;

break;

}

}

}

}

}

void display(int a[])

{

FILE \*fp;

int i;

fp = fopen("symbol.txt", "w");

printf("\nSymbol Table:");

printf("\nHash Index | Address | Label\n");

for (i = 0; i < MAX; i++)

{

if (a[i] != 0)

{

printf("%d | %d | %s\n", i, sy[i].add, sy[i].label);

fprintf(fp, "%d %d %s\n", i, sy[i].add, sy[i].label);

}

}

fclose(fp);

}

void search()

{

FILE \*fp;

char la[10];

int found = 0, i;

printf("Enter the label: ");

scanf("%s", la);

fp = fopen("symbol.txt", "r");

for (i = 0; i < MAX; i++)

{

fscanf(fp, "%d %d %s", &i, &sy[i].add, sy[i].label);

if (sy[i].add != 0 && strcmp(sy[i].label, la) == 0)

{

printf("\nThe label '%s' is present in the symbol table at address: %d\n", la, sy[i].add);

found = 1;

break;

}

}

if (!found)

{

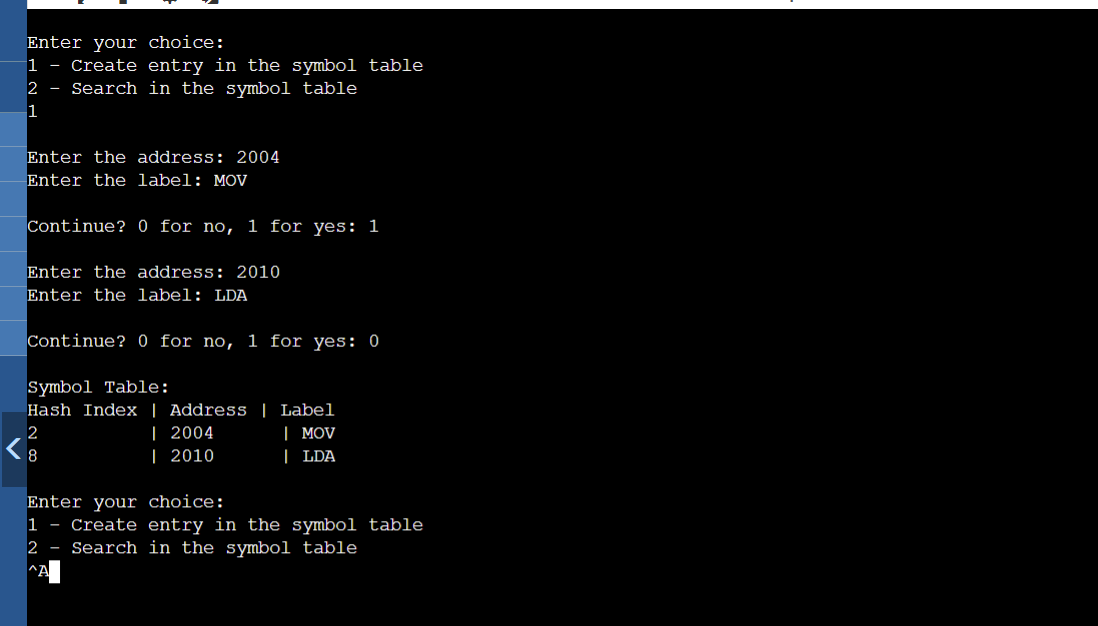
printf("\nThe label '%s' is not present in the symbol table\n", la);

}

fclose(fp);

}

OUTPUT



2) **Implement a two-pass assembler 8085/8086**

Main File – main.c

#include <bits/stdc++.h>

using namespace std;

string convert(int decimal) {

    string binary = "" ;

    for (int i = decimal; i > 0; i /= 2) {

        binary = to\_string(i % 2) + binary;

    }

    if (binary.length() < 8) {

        binary = string(8 - binary.length(), '0') + binary;

    }

    return binary;

}

struct Mnemonics {

    string name;

    string binary;

    int size;

} mot[20];

struct Symbol {

    string name;

    string type;

    int location;

    int size;

    int section\_id;

    string is\_global;

};

struct Section {

    int id;

    string name;

    int size;

};

vector<Symbol> symtab;  // Symbol Table

vector<Section> sect;   // Section Table

int lc = 0;             // Location Counter

int sec\_id = 0;         // Section Id

int var\_lc;             // Store location of variable in Pass2

ifstream infile;        // Input File Stream

ofstream outfile;       // Output File Stream

string word;            // Read word by word from file

string temp;            // Temporary variable

int control, size;      // Control variables for search

void init() {

    // Initializing Machine Opcode Table

    //name binary size

    mot[0] = {"ADD", "00000001", 1};

    mot[1] = {"ADDI", "00000010", 5};

    mot[2] = {"CMP", "00000011", 5};

    mot[3] = {"INC", "00000100", 1};

    mot[4] = {"JE", "00000101", 5};

    mot[5] = {"JMP", "00000110", 5};

    mot[6] = {"LOAD", "00000111", 5};

    mot[7] = {"LOADI", "00001000", 5};

    mot[8] = {"MVTI", "00001001", 5};

    mot[9] = {"MOV", "00001010", 1};

    mot[10] = {"STOP", "00001111", 1};

    mot[11] = {"STORE", "00001100", 5};

    mot[12] = {"STOREI", "00001101", 1};

}

int search\_mot(const string& opcode) {

    for (int i = 0; i < 13; ++i) {

        if (mot[i].name == opcode) {

            return i;

        }

    }

    return -1;

}

int search\_symbol(const string& variable) {

    for (const auto& sym : symtab) {

        if (sym.name == variable) {

            return sym.location;

        }

    }

    return -1;

}

int size\_evaluation(const string& data) {

    int size = 0;

    for (char c : data) {

        if (c == ',') {

            size += 4;

        }

    }

    size += 4;

    return size;

}

string data\_break(const string& data) {

    string final;

    string temp = "";

    for (char c : data) {

        if (c == ',') {

            final += convert(atoi(temp.c\_str())) + ",";

            temp = "";

        } else {

            temp += c;

        }

    }

    if (!temp.empty()) {

        final += convert(atoi(temp.c\_str()));

    }

    return final;

}

void store\_symlab() {

    outfile.open("symbol.csv");

    outfile << "Name,Type,Location,Size,SectionId,IsGlobal\n";

    for (const auto& sym : symtab) {

        outfile << sym.name << ','

                << sym.type << ','

                << sym.location << ','

                << sym.size << ','

                << sym.section\_id << ','

                << sym.is\_global << '\n';

    }

    outfile.close();

}

void store\_sec() {

    outfile.open("section.csv");

    outfile << "ID,Name,Size\n";

    for (const auto& sec : sect) {

        outfile << sec.id << ','

                << sec.name << ','

                << sec.size << '\n';

    }

    outfile.close();

}

void pass1() {

    infile.open("input.txt");

    while (infile >> word) {

        control = search\_mot(word);

        if (control == -1) {

            temp = word;

            if (word.find(":") != string::npos) {

                // Label found

                Symbol s = {temp.erase(word.length() - 1, 1), "label", lc, -1, sec\_id, "false"};

                symtab.push\_back(s);

            } else if (word == "section") {

                infile >> word;

                ++sec\_id;

                Section a = {sec\_id, word, 0};

                sect.push\_back(a);

                if (sec\_id != 1) {

                    sect[sec\_id - 2].size = lc;

                    lc = 0;

                }

            } else if (word == "global") {

                infile >> word;

                Symbol b = {word, "label", -1, -1, -1, "true"};

                symtab.push\_back(b);

            } else if (word == "extern") {

                infile >> word;

                Symbol b = {word, "external", -1, -1, -1, "false"};

                symtab.push\_back(b);

            } else {

                infile >> word;

                infile >> word;

                int len = size\_evaluation(word);

                Symbol b = {temp, "var", lc, len, sec\_id, "false"};

                symtab.push\_back(b);

                lc += len;

            }

        } else {

            if (!(control == 7 || control == 12)) infile >> word;

            // LOADI and STOREI do not have any parameter

            if (control == 2 || control == 8 || control == 9) infile >> word;

            lc += mot[control].size;

        }

    }

    sect[sec\_id - 1].size = lc;

    store\_symlab();

    store\_sec();

    infile.close();

}

void pass2() {

    infile.open("input.txt");

    outfile.open("output.txt");

    while (infile >> word) {

        control = search\_mot(word);

        if (control == -1) {

            temp = word;

            if (word.find(":") != string::npos) {

                outfile << "";

            } else if (word == "global") {

                infile >> word;

                outfile << "global " << word << '\n';

            } else if (word == "extern") {

                infile >> word;

                outfile << "extern " << word << '\n';

            } else if (word == "section") {

                infile >> word;

                outfile << "section ." << word << '\n';

                lc = 0;

            } else {

                infile >> word;

                infile >> word;

                outfile << convert(lc) << " " << data\_break(word) << '\n';

                int len = size\_evaluation(word);

                lc += len;

            }

        } else {

            outfile << convert(lc) << " " << mot[control].binary;

            if (control == 0 || control == 3) {

                infile >> word;

                outfile << " " << word;

            } else if (control == 1 || control == 4 || control == 5 || control == 6 || control == 11) {

                infile >> word;

                var\_lc = search\_symbol(word);

                if (var\_lc == -1) {

                    outfile << " " << convert(atoi(word.c\_str()));

                } else {

                    outfile << " " << convert(var\_lc);

                }

            } else if (control == 9) {

                infile >> word;

                outfile << " " << word;q

                infile >> word;

                outfile << " " << word;

            }

            lc += mot[control].size;

            outfile << '\n';

        }

    }

    outfile.close();

    infile.close();

}

int main() {

    init();

    pass1();

    lc = 0;

    pass2();

    return 0;

}

Input.txt

global main

extern test

section data

X dd 10,20,40,5,7,9,53,8,11,13

sum dd 0,10

section text

MVI I,X

MVI B,0

MVI C,0

L1: LOADI

ADD C

MOV C,A

INC B

CMP B,10

JE L2

ADDI 4

JMP L1

L2: STORE sum

STOP

Section.csv

ID,Name,Size

1,data,48

2,text,48

Symbol.csv

Name,Type,Location,Size,SectionId,IsGlobal

main,label,-1,-1,-1,true

test,external,-1,-1,-1,false

X,var,0,40,1,false

sum,var,40,8,1,false

MVI,var,0,4,2,false

B,0,var,4,8,2,false

L1,label,12,-1,2,false

B,var,19,8,2,false

L2,label,42,-1,2,false

Output.txt

global main

extern test

section .data

00000000 00001010,00010100,00101000,00000101,00000111,00001001,00110101,00001000,00001011,00001101

00101000 00000000,00001010

section .text

00000000 00000000

00000100 00000000,00000000

00001100 00001000

00010001 00000001 C

00010010 00001010 C,A INC

00010011 00000000,00001010

00011011 00000101 00101010

00100000 00000010 00000100

00100101 00000110 00001100

00101010 00001100 00101000

00101111 00001111

**3) Develop Lexical Analyzer for ‘C’ using LEX tool**

* C\_lex\_analyser.l

%{

int COMMENT=0;

%}

identifier [a-zA-Z][a-zA-Z0-9]\*

%%

#.\*\n {printf("%sThis is a PREPROCESSOR DIRECTIVE\n",yytext);}

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("\n%s is a KEYWORD",yytext);}

"/\*" {COMMENT = 1;}

"\*/" {COMMENT = 0;}

{identifier}\( {if(!COMMENT)printf("\nFUNCTION: \n%s",yytext);}

{identifier}(\[[0-9]\*\])? {if(!COMMENT) printf("\n%s is an IDENTIFIER",yytext);}

\".\*\" {if(!COMMENT)printf("\n%s is a STRING",yytext);}

[0-9]+ {if(!COMMENT) printf("\n%s is a NUMBER ",yytext);}

\{ {if(!COMMENT) printf("\nBLOCK BEGINS");}

\} {if(!COMMENT) printf("\nBLOCK ENDS");}

\) {if(!COMMENT);printf("\n)");}

= {if(!COMMENT) printf("\n%s is an ASSIGNMENT OPERATOR",yytext);}

\<= |

\>= |

\< |

\== |

\!= |

\> {if(!COMMENT) printf("\n%s is a RELATIONAL OPERATOR",yytext);}

\, |

\; {if(!COMMENT) printf("\n%s is a SEPERATOR",yytext);}

%%

int main(int argc, char \*\*argv)

{

FILE \*file;

file=fopen("input.c","r");

if(!file)

{

printf("could not open the file");

exit(0);

}

yyin=file;

yylex();

printf("\n");

return(0);

}

int yywrap()

{

return(1);

}

* Input.c

#include <stdio.h>

void main(int a)

{

    /\* HELLO This is a comment\*/

    int a, b, c;

    a = 1;

    b = 2;

    if (a > b)

        c = 0;

    else

        c = -1;

    printf("The value of c: %d", c);

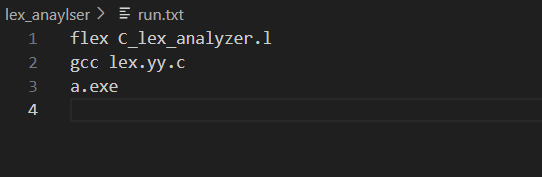
    for (int i = 0; i < 5; i++)

        i++;

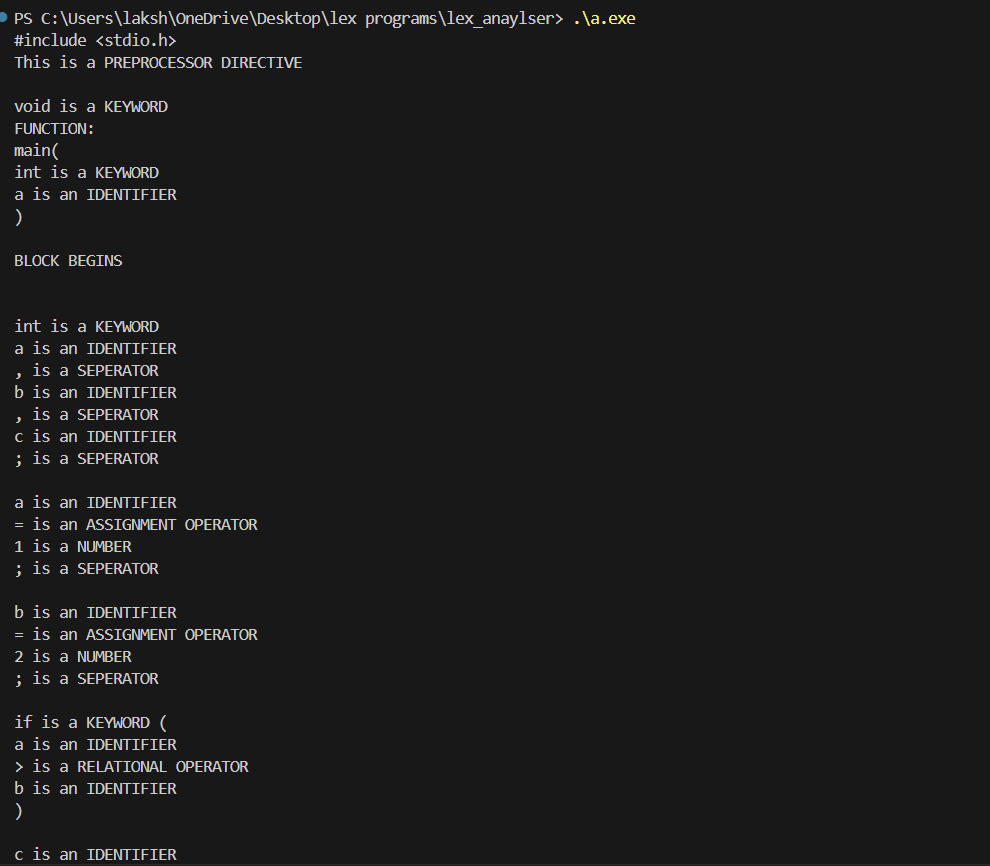
    return 0;

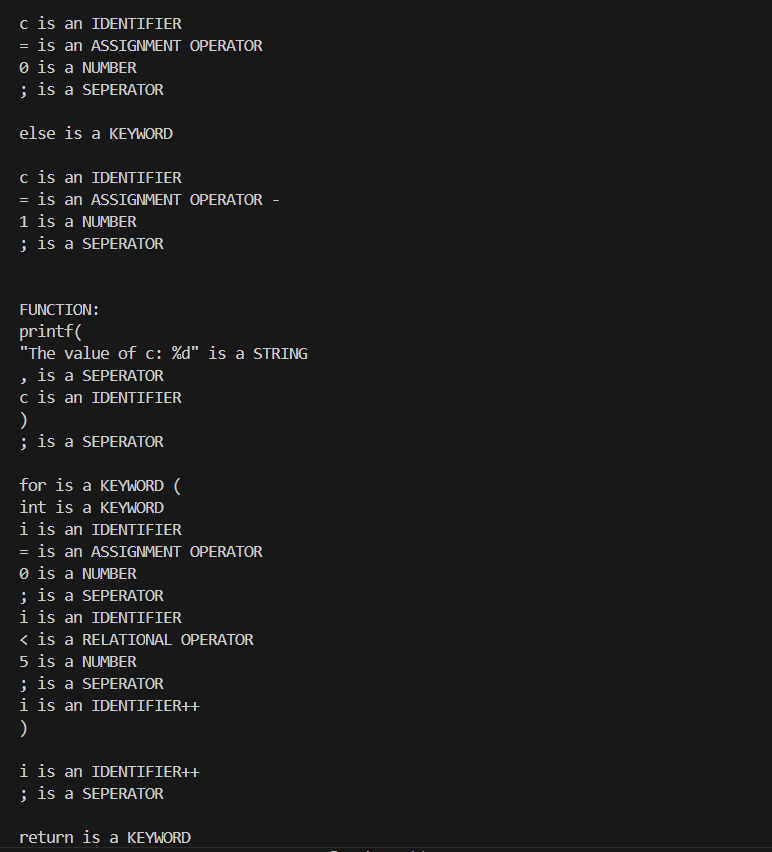
}

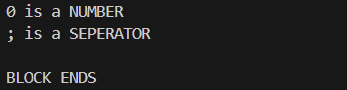
* To RUN



* OUTPUT







**4) Represent C language using Context Free Language**

The Context Free Grammar for C language can be given by G = (V, T, S, P):

where:

V = set of non-terminals

= {program\_unit, translation\_unit, external\_decl, function\_definition, decl, decl\_list, decl\_specs, storage\_class\_spec, type\_spec, type\_qualifier, struct\_or\_union\_spec, struct\_or\_union, struct\_decl\_list, init\_declarator\_list, init\_declarator, struct\_decl, spec\_qualifier\_list, struct\_declarator\_list, struct\_declarator\_list, struct\_declarator, enum\_spec, enumerator\_list, enumerator, declarator, direct\_declarator, pointer, type\_qualifier\_list, param\_list, param\_decl, id\_list, initializer, initializer\_list, type\_name, abstract\_declarator, direct\_abstract\_declarator, stat, labeled\_stat, exp\_stat,compound\_stat, stat\_list, selection\_stat, iteration\_stat, jump\_stat, exp assignment\_exp, assignment\_operator, conditional\_exp, logical\_or\_exp, logical\_and\_exp, inclusive\_or\_exp, exclusive\_or\_exp, and\_exp, equality\_exp, relational\_exp, shift\_expression, additive\_exp, mult\_exp, cast\_exp, unary\_exp, unary\_operator, postfix\_exp, primary\_exp, argument\_exp\_list, consts, int\_const, char\_const, float\_const, id, string, enumeration\_const, storage\_const, type\_const, qual\_const, struct\_const, enum\_const, DEFINE, IF, ELSE, FOR, DO, WHILE, BREAK, SWITCH, CONTINUE, RETURN, CASE, DEFAULT, GOTO, SIZEOF, PUNC, or\_const, and\_const, eq\_const, shift\_const, rel\_const, inc\_const, point\_const, HEADER}

T = set of terminals

= {All ASCII characters}

S = start symbol = program\_unit

P = set of productions

program\_unit -> HEADER program\_unit

| DEFINE primary\_exp program\_unit

| translation\_unit

translation\_unit -> external\_decl

| translation\_unit external\_decl

external\_decl -> function\_definition

| decl

function\_definition -> decl\_specs declarator decl\_list compound\_stat

| declarator decl\_list compound\_stat

| decl\_specs declarator compound\_stat

| declarator compound\_stat

decl -> decl\_specs init\_declarator\_list ';'

| decl\_specs ';'

decl\_list -> decl

| decl\_list decl

decl\_specs -> storage\_class\_spec decl\_specs

| storage\_class\_spec

| type\_spec decl\_specs

| type\_spec

| type\_qualifier decl\_specs

| type\_qualifier

storage\_class\_spec -> storage\_const

type\_spec -> type\_const

| struct\_or\_union\_spec

| enum\_spec

type\_qualifier -> qual\_const

struct\_or\_union\_spec -> struct\_or\_union id '{' struct\_decl\_list '}' ';'

| struct\_or\_union id

struct\_or\_union -> struct\_const

struct\_decl\_list -> struct\_decl

| struct\_decl\_list struct\_decl

init\_declarator\_list -> init\_declarator

| init\_declarator\_list ',' init\_declarator

init\_declarator -> declarator

| declarator '=' initializer

struct\_decl -> spec\_qualifier\_list struct\_declarator\_list ';'

spec\_qualifier\_list -> type\_spec spec\_qualifier\_list

| type\_spec

| type\_qualifier spec\_qualifier\_list

| type\_qualifier

struct\_declarator\_list -> struct\_declarator

| struct\_declarator\_list ',' struct\_declarator

struct\_declarator -> declarator

| declarator ':' conditional\_exp

| ':' conditional\_exp

enum\_spec -> enum\_const id '{' enumerator\_list '}'

| enum\_const '{' enumerator\_list '}'

| enum\_const id

enumerator\_list -> enumerator

| enumerator\_list ',' enumerator

enumerator -> id

| id '=' conditional\_exp

declarator -> pointer direct\_declarator

| direct\_declarator

direct\_declarator -> id

| '(' declarator ')'

| direct\_declarator '[' conditional\_exp ']'

| direct\_declarator '[' ']'

| direct\_declarator '(' param\_list ')'

| direct\_declarator '(' id\_list ')'

| direct\_declarator '(' ')'

pointer -> '\*' type\_qualifier\_list

| '\*'

| '\*' type\_qualifier\_list pointer

| '\*' pointer

type\_qualifier\_list -> type\_qualifier

| type\_qualifier\_list type\_qualifier

param\_list -> param\_decl

| param\_list ',' param\_decl

param\_decl -> decl\_specs declarator

| decl\_specs abstract\_declarator

| decl\_specs

id\_list -> id

| id\_list ',' id

initializer -> assignment\_exp

| '{' initializer\_list '}'

| '{' initializer\_list ',' '}'

initializer\_list -> initializer

| initializer\_list ',' initializer

type\_name -> spec\_qualifier\_list abstract\_declarator

| spec\_qualifier\_list

abstract\_declarator -> pointer

| pointer direct\_abstract\_declarator

| direct\_abstract\_declarator

direct\_abstract\_declarator -> '(' abstract\_declarator ')'

| direct\_abstract\_declarator '[' conditional\_exp ']'

| '[' conditional\_exp ']'

| direct\_abstract\_declarator '[' ']'

| '[' ']'

| direct\_abstract\_declarator '(' param\_list ')'

| '(' param\_list ')'

| direct\_abstract\_declarator '(' ')'

| '(' ')'

stat -> labeled\_stat

| exp\_stat

| compound\_stat

| selection\_stat

| iteration\_stat

| jump\_stat

labeled\_stat -> id ':' stat

| CASE int\_const ':' stat

| DEFAULT ':' stat

exp\_stat -> exp ';'

| ';'

compound\_stat -> '{' decl\_list stat\_list '}'

| '{' stat\_list '}'

| '{' decl\_list '}'

| '{' '}'

stat\_list -> stat

| stat\_list stat

selection\_stat -> IF '(' exp ')' stat %prec "then"

| IF '(' exp ')' stat ELSE stat

| SWITCH '(' exp ')' stat

iteration\_stat -> WHILE '(' exp ')' stat

| DO stat WHILE '(' exp ')' ';'

| FOR '(' exp ';' exp ';' exp ')' stat

| FOR '(' exp ';' exp ';' ')' stat

| FOR '(' exp ';' ';' exp ')' stat

| FOR '(' exp ';' ';' ')' stat

| FOR '(' ';' exp ';' exp ')' stat

| FOR '(' ';' exp ';' ')' stat

| FOR '(' ';' ';' exp ')' stat

| FOR '(' ';' ';' ')' stat

jump\_stat -> GOTO id ';'

| CONTINUE ';'

| BREAK ';'

| RETURN exp ';'

| RETURN ';'

exp -> assignment\_exp

| exp ',' assignment\_exp

assignment\_exp -> conditional\_exp

| unary\_exp assignment\_operator assignment\_exp

assignment\_operator -> PUNC

| '='

conditional\_exp -> logical\_or\_exp

| logical\_or\_exp '?' exp ':' conditional\_exp

logical\_or\_exp -> logical\_and\_exp

| logical\_or\_exp or\_const logical\_and\_exp

logical\_and\_exp -> inclusive\_or\_exp

| logical\_and\_exp and\_const inclusive\_or\_exp

inclusive\_or\_exp -> exclusive\_or\_exp

| inclusive\_or\_exp '|' exclusive\_or\_exp

exclusive\_or\_exp -> and\_exp

| exclusive\_or\_exp '^' and\_exp

and\_exp -> equality\_exp

| and\_exp '&' equality\_exp

equality\_exp -> relational\_exp

| equality\_exp eq\_const relational\_exp

relational\_exp -> shift\_expression

| relational\_exp '<' shift\_expression

| relational\_exp '>' shift\_expression

| relational\_exp rel\_const shift\_expression

shift\_expression -> additive\_exp

| shift\_expression shift\_const additive\_exp

additive\_exp -> mult\_exp

| additive\_exp '+' mult\_exp

| additive\_exp '-' mult\_exp

mult\_exp -> cast\_exp

| mult\_exp '\*' cast\_exp

| mult\_exp '/' cast\_exp

| mult\_exp '%' cast\_exp

cast\_exp -> unary\_exp

| '(' type\_name ')' cast\_exp

unary\_exp -> postfix\_exp

| inc\_const unary\_exp

| unary\_operator cast\_exp

| SIZEOF unary\_exp

| SIZEOF '(' type\_name ')'

unary\_operator -> '&' | '\*' | '+' | '-' | '~' | '!'

postfix\_exp -> primary\_exp

| postfix\_exp '[' exp ']'

| postfix\_exp '(' argument\_exp\_list ')'

| postfix\_exp '(' ')'

| postfix\_exp '.' id

| postfix\_exp point\_const id

| postfix\_exp inc\_const

primary\_exp -> id

| consts

| string

| '(' exp ')'

argument\_exp\_list -> assignment\_exp

| argument\_exp\_list ',' assignment\_exp

consts -> int\_const

| char\_const

| float\_const

| enumeration\_const

int\_const -> [0-9]+

char\_const -> "'"."'"

float\_const -> [0-9]+"."[0-9]+

id -> [a-zA-z\_][a-zA-z\_0-9]\*

string -> \".\*\"

enum\_const -> "enum"

storage\_const -> "auto"

| "register"

| "static"

| "extern"

| "typedef"

type\_const -> "void"

| "char"

| "short"

| "int"

| "long"

| "float"

| "double"

| "signed"

| "unsigned"

qual\_const -> "const"

| "volatile"

struct\_const -> "struct"

| "union"

DEFINE -> "#define"[ ]+[a-zA-z\_][a-zA-z\_0-9]\*

IF -> "if"

ELSE -> "else"

FOR -> "for"

DO -> "do"

WHILE -> "while"

BREAK -> "break"

SWITCH -> "switch"

CONTINUE -> "continue"

RETURN -> "return"

CASE -> "case"

DEFAULT -> "default"

GOTO -> "goto"

SIZEOF -> "sizeof"

PUNC -> "\*="

| "/="

| "+="

| "%="

| ">>="

| "-="

| "<<="

| "&="

| "^="

| "|="

or\_const -> "||"

and\_const -> "&&"

eq\_const -> "=="

| "!="

shift\_const -> ">>"

| "<<"

rel\_const -> "<="

| ">="

inc\_const -> "++"

| "--"

point\_const -> "->"

HEADER -> "#include"[ ]+<[a-zA-z\_][a-zA-z\_0-9.]\*>

5) **Develop a simple calculator using LEX and YACC tools.**

* ##cal.l file

%{

/\* Definition section \*/

#include<stdio.h>

#include "cl.tab.h"

extern int yylval;

%}

/\* Rule Section \*/

%%

[0-9]+ {

yylval=atoi(yytext);

return NUMBER;

}

[\t] ;

[\n] return 0;

. return yytext[0];

%%

int yywrap()

{

return 1;

}

* ##cl.y FILE

%{

/\* Definition section \*/

#include<stdio.h>

int flag=0;

%}

%token NUMBER

%left '+' '-'

%left '\*' '/' '%'

%left '(' ')'

/\* Rule Section \*/

%%

ArithmeticExpression: E{

printf("\nResult=%d\n", $$);

return 0;

};

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

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

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

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

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

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

| NUMBER {$$=$1;}

;

%%

//driver code

void main()

{

printf("\nEnter Any Arithmetic Expression :\n");

yyparse();

if(flag==0)

printf("\nEntered arithmetic expression is Valid\n\n");

}

void yyerror(char \*a)

{

printf("\nEntered arithmetic expression is Invalid\n\n");

flag=1;

}

* To Run this Code File Use this ->

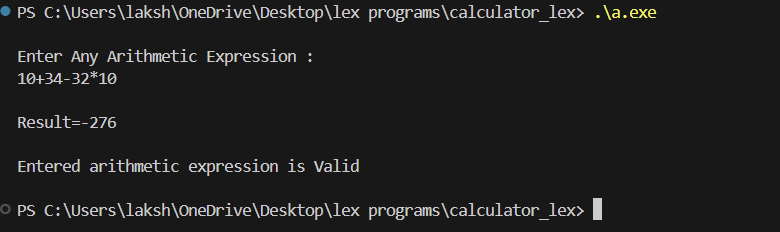
bison -d cl.y

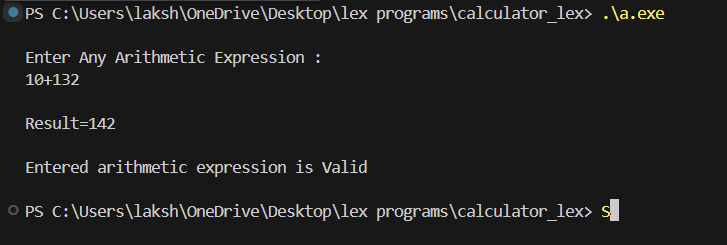
flex cal.l

gcc lex.yy.c cl.tab.c -w

a.exe

* OUTPUT





6) **Develop a Parser for ‘C’ using the LEX and YACC tools.**

* Parser.y

%{

#include<stdio.h>

int yylex(void);

int yyerror(const char \*s);

int success = 1;

%}

%token int\_const char\_const float\_const id string storage\_const type\_const qual\_const struct\_const enum\_const DEFINE

%token IF FOR DO WHILE BREAK SWITCH CONTINUE RETURN CASE DEFAULT GOTO SIZEOF PUNC or\_const and\_const eq\_const shift\_const rel\_const inc\_const

%token point\_const ELSE HEADER

%left '+' '-'

%left '\*' '/'

%right UMINUS

%nonassoc "then"

%nonassoc ELSE

%start program\_unit

%%

program\_unit : HEADER program\_unit

| DEFINE primary\_exp program\_unit

| translation\_unit

;

translation\_unit : external\_decl

| translation\_unit external\_decl

;

external\_decl : function\_definition

| decl

;

function\_definition : decl\_specs declarator decl\_list compound\_stat

| declarator decl\_list compound\_stat

| decl\_specs declarator compound\_stat

| declarator compound\_stat

;

decl : decl\_specs init\_declarator\_list ';'

| decl\_specs ';'

;

decl\_list : decl

| decl\_list decl

;

decl\_specs : storage\_class\_spec decl\_specs

| storage\_class\_spec

| type\_spec decl\_specs

| type\_spec

| type\_qualifier decl\_specs

| type\_qualifier

;

storage\_class\_spec : storage\_const

;

type\_spec : type\_const

| struct\_or\_union\_spec

| enum\_spec

;

type\_qualifier : qual\_const

;

struct\_or\_union\_spec : struct\_or\_union id '{' struct\_decl\_list '}' ';'

| struct\_or\_union id

;

struct\_or\_union : struct\_const

;

struct\_decl\_list : struct\_decl

| struct\_decl\_list struct\_decl

;

init\_declarator\_list : init\_declarator

| init\_declarator\_list ',' init\_declarator

;

init\_declarator : declarator

| declarator '=' initializer

;

struct\_decl : spec\_qualifier\_list struct\_declarator\_list ';'

;

spec\_qualifier\_list : type\_spec spec\_qualifier\_list

| type\_spec

| type\_qualifier spec\_qualifier\_list

| type\_qualifier

;

struct\_declarator\_list : struct\_declarator

| struct\_declarator\_list ',' struct\_declarator

;

struct\_declarator : declarator

| declarator ':' conditional\_exp

| ':' conditional\_exp

;

enum\_spec : enum\_const id '{' enumerator\_list '}'

| enum\_const '{' enumerator\_list '}'

| enum\_const id

;

enumerator\_list : enumerator

| enumerator\_list ',' enumerator

;

enumerator : id

| id '=' conditional\_exp

;

declarator : pointer direct\_declarator

| direct\_declarator

;

direct\_declarator : id

| '(' declarator ')'

| direct\_declarator '[' conditional\_exp ']'

| direct\_declarator '[' ']'

| direct\_declarator '(' param\_list ')'

| direct\_declarator '(' id\_list ')'

| direct\_declarator '(' ')'

;

pointer : '\*' type\_qualifier\_list

| '\*'

| '\*' type\_qualifier\_list pointer

| '\*' pointer

;

type\_qualifier\_list : type\_qualifier

| type\_qualifier\_list type\_qualifier

;

param\_list : param\_decl

| param\_list ',' param\_decl

;

param\_decl : decl\_specs declarator

| decl\_specs abstract\_declarator

| decl\_specs

;

id\_list : id

| id\_list ',' id

;

initializer : assignment\_exp

| '{' initializer\_list '}'

| '{' initializer\_list ',' '}'

;

initializer\_list : initializer

| initializer\_list ',' initializer

;

type\_name : spec\_qualifier\_list abstract\_declarator

| spec\_qualifier\_list

;

abstract\_declarator : pointer

| pointer direct\_abstract\_declarator

| direct\_abstract\_declarator

;

direct\_abstract\_declarator : '(' abstract\_declarator ')'

| direct\_abstract\_declarator '[' conditional\_exp ']'

| '[' conditional\_exp ']'

| direct\_abstract\_declarator '[' ']'

| '[' ']'

| direct\_abstract\_declarator '(' param\_list ')'

| '(' param\_list ')'

| direct\_abstract\_declarator '(' ')'

| '(' ')'

;

stat : labeled\_stat

| exp\_stat

| compound\_stat

| selection\_stat

| iteration\_stat

| jump\_stat

;

labeled\_stat : id ':' stat

| CASE int\_const ':' stat

| DEFAULT ':' stat

;

exp\_stat : exp ';'

| ';'

;

compound\_stat : '{' decl\_list stat\_list '}'

| '{' stat\_list '}'

| '{' decl\_list '}'

| '{' '}'

;

stat\_list : stat

| stat\_list stat

;

selection\_stat : IF '(' exp ')' stat %prec "then"

| IF '(' exp ')' stat ELSE stat

| SWITCH '(' exp ')' stat

;

iteration\_stat : WHILE '(' exp ')' stat

| DO stat WHILE '(' exp ')' ';'

| FOR '(' exp ';' exp ';' exp ')' stat

| FOR '(' exp ';' exp ';' ')' stat

| FOR '(' exp ';' ';' exp ')' stat

| FOR '(' exp ';' ';' ')' stat

| FOR '(' ';' exp ';' exp ')' stat

| FOR '(' ';' exp ';' ')' stat

| FOR '(' ';' ';' exp ')' stat

| FOR '(' ';' ';' ')' stat

;

jump\_stat : GOTO id ';'

| CONTINUE ';'

| BREAK ';'

| RETURN exp ';'

| RETURN ';'

;

exp : assignment\_exp

| exp ',' assignment\_exp

;

assignment\_exp : conditional\_exp

| unary\_exp assignment\_operator assignment\_exp

;

assignment\_operator : PUNC

| '='

;

conditional\_exp : logical\_or\_exp

| logical\_or\_exp '?' exp ':' conditional\_exp

;

logical\_or\_exp : logical\_and\_exp

| logical\_or\_exp or\_const logical\_and\_exp

;

logical\_and\_exp : inclusive\_or\_exp

| logical\_and\_exp and\_const inclusive\_or\_exp

;

inclusive\_or\_exp : exclusive\_or\_exp

| inclusive\_or\_exp '|' exclusive\_or\_exp

;

exclusive\_or\_exp : and\_exp

| exclusive\_or\_exp '^' and\_exp

;

and\_exp : equality\_exp

| and\_exp '&' equality\_exp

;

equality\_exp : relational\_exp

| equality\_exp eq\_const relational\_exp

;

relational\_exp : shift\_expression

| relational\_exp '<' shift\_expression

| relational\_exp '>' shift\_expression

| relational\_exp rel\_const shift\_expression

;

shift\_expression : additive\_exp

| shift\_expression shift\_const additive\_exp

;

additive\_exp : mult\_exp

| additive\_exp '+' mult\_exp

| additive\_exp '-' mult\_exp

;

mult\_exp : cast\_exp

| mult\_exp '\*' cast\_exp

| mult\_exp '/' cast\_exp

| mult\_exp '%' cast\_exp

;

cast\_exp : unary\_exp

| '(' type\_name ')' cast\_exp

;

unary\_exp : postfix\_exp

| inc\_const unary\_exp

| unary\_operator cast\_exp

| SIZEOF unary\_exp

| SIZEOF '(' type\_name ')'

;

unary\_operator : '&' | '\*' | '+' | '-' | '~' | '!'

;

postfix\_exp : primary\_exp

| postfix\_exp '[' exp ']'

| postfix\_exp '(' argument\_exp\_list ')'

| postfix\_exp '(' ')'

| postfix\_exp '.' id

| postfix\_exp point\_const id

| postfix\_exp inc\_const

;

primary\_exp : id

| consts

| string

| '(' exp ')'

;

argument\_exp\_list : assignment\_exp

| argument\_exp\_list ',' assignment\_exp

;

consts : int\_const

| char\_const

| float\_const

| enum\_const

;

%%

int main()

{

yyparse();

if(success)

printf("Parsing Successful\n");

return 0;

}

int yyerror(const char \*msg)

{

extern int yylineno;

printf("Parsing Failed\nLine Number: %d %s\n",yylineno,msg);

success = 0;

return 0;

}

* Parser.l

%option yylineno

%{

#include<stdio.h>

#include"parser.tab.h"

%}

%%

"#include"[ ]+<[a-zA-z\_][a-zA-z\_0-9.]\*> {return HEADER;}

"#define"[ ]+[a-zA-z\_][a-zA-z\_0-9]\* {return DEFINE;}

"auto"|"register"|"static"|"extern"|"typedef" {return storage\_const;}

"void"|"char"|"short"|"int"|"long"|"float"|"double"|"signed"|"unsigned" {return type\_const;}

"const"|"volatile" {return qual\_const;}

"enum" {return enum\_const;}

"struct"|"union" {return struct\_const;}

"case" {return CASE;}

"default" {return DEFAULT;}

"if" {return IF;}

"switch" {return SWITCH;}

"else" {return ELSE;}

"for" {return FOR;}

"do" {return DO;}

"while" {return WHILE;}

"goto" {return GOTO;}

"continue" {return CONTINUE;}

"break" {return BREAK;}

"return" {return RETURN;}

"sizeof" {return SIZEOF;}

"||" {return or\_const;}

"&&" {return and\_const;}

"=="|"!=" {return eq\_const;}

"<="|">=" {return rel\_const;}

">>"|"<<" {return shift\_const;}

"++"|"--" {return inc\_const;}

"->" {return point\_const;}

"\*="|"/="|"+="|"%="|">>="|"-="|"<<="|"&="|"^="|"|=" {return PUNC;}

[0-9]+ {return int\_const;}

[0-9]+"."[0-9]+ {return float\_const;}

"'"."'" {return char\_const;}

[a-zA-z\_][a-zA-z\_0-9]\* {return id;}

\".\*\" {return string;}

"//"(\\.|[^\n])\*[\n] ;

[/][\*]([^\*]|[\*]\*[^\*/])\*[\*]+[/] ;

[ \t\n] ;

";"|"="|","|"{"|"}"|"("|")"|"["|"]"|"\*"|"+"|"-"|"/"|"?"|":"|"&"|"|"|"^"|"!"|"~"|"%"|"<"|">" {return yytext[0];}

%%

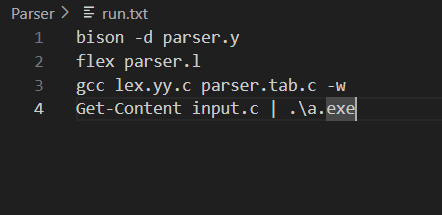
int yywrap(void)

{

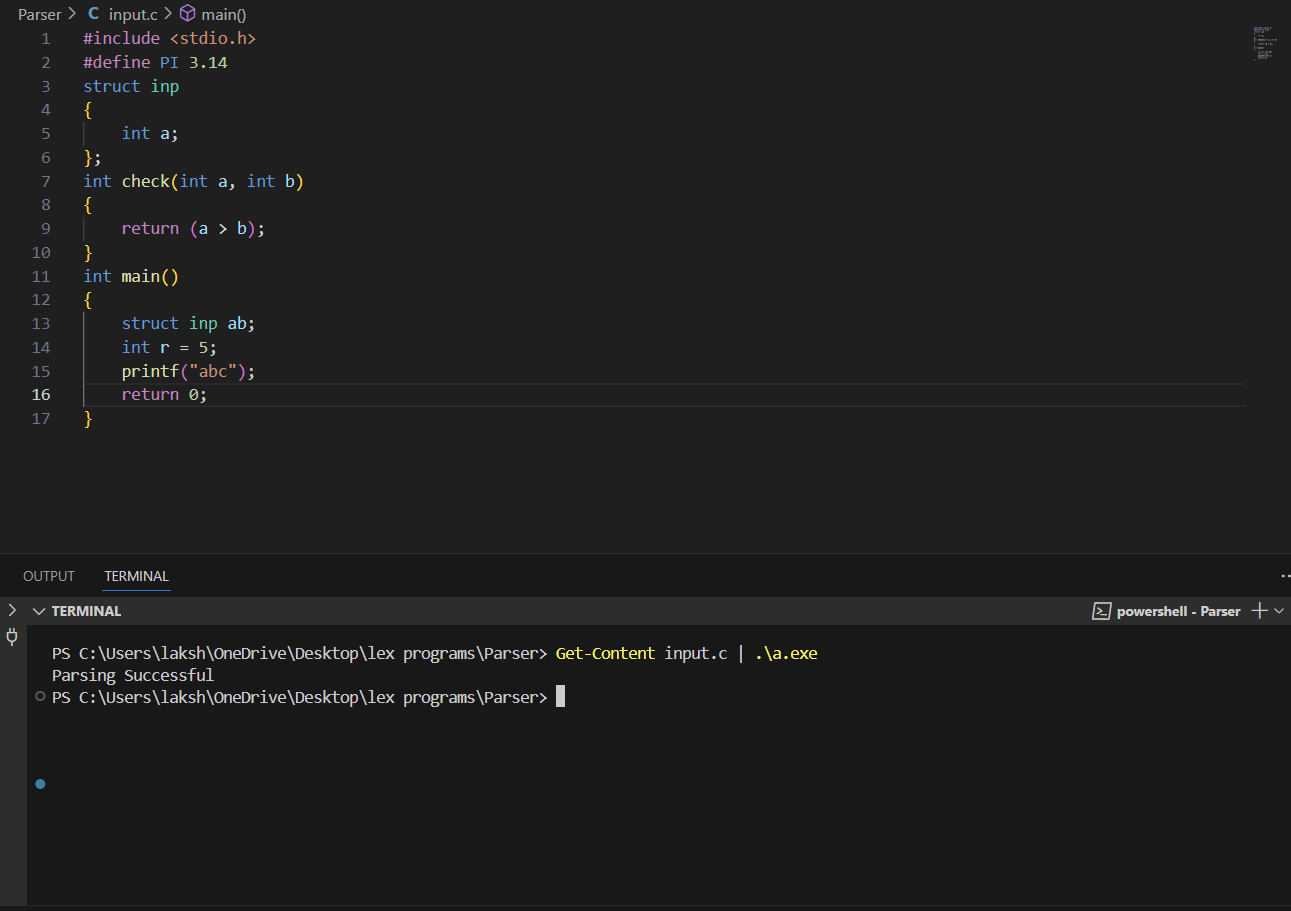
return 1;

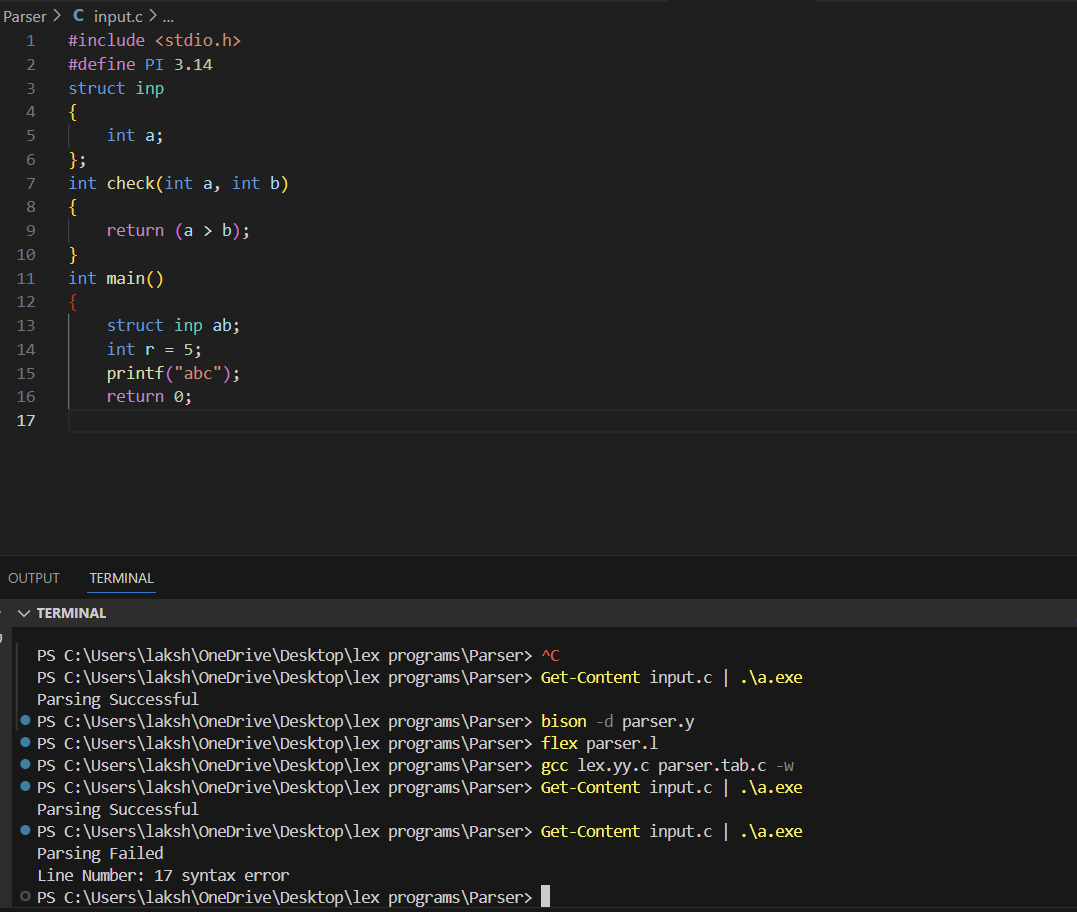
}

🡪 How to run code ?



* OUTPUT





**7) Add assignment statement , If then else statement and while loop to the calculator and generate the three address code for the same.**

#include <stdio.h>

#include <string.h>

// Label counter for TAC generation

int labelCounter = 0;

int tempCounter = 0;

// Function to generate new labels and temporary variables

char\* newLabel() {

static char label[10];

sprintf(label, "L%d", labelCounter++);

return label;

}

char\* newTemp() {

static char temp[10];

sprintf(temp, "t%d", tempCounter++);

return temp;

}

// Function prototypes

void generateAssignment(char\* id, char\* expr);

void generateIfElse(char\* condition, char\* trueBlock, char\* falseBlock);

void generateWhileLoop(char\* condition, char\* body);

int main() {

// Example usage

generateAssignment("x", "a + b");

generateIfElse("a < b", "x = a + b", "x = a - b");

generateWhileLoop("a < b", "a = a + 1");

return 0;

}

// Function to generate TAC for assignment statements

void generateAssignment(char\* id, char\* expr) {

printf("%s = %s\n", id, expr);

}

// Function to generate TAC for if-then-else statements

void generateIfElse(char\* condition, char\* trueBlock, char\* falseBlock) {

char\* L1 = newLabel();

char\* L2 = newLabel();

char\* L3 = newLabel();

printf("if %s goto %s\n", condition, L1);

printf("goto %s\n", L2);

printf("%s: %s\n", L1, trueBlock);

printf("goto %s\n", L3);

printf("%s: %s\n", L2, falseBlock);

printf("%s:\n", L3);

}

// Function to generate TAC for while loops

void generateWhileLoop(char\* condition, char\* body) {

char\* L1 = newLabel();

char\* L2 = newLabel();

printf("%s: if not %s goto %s\n", L1, condition, L2);

printf("%s\n", body);

printf("goto %s\n", L1);

printf("%s:\n", L2);

}

Output

