**COMPILER DESIGN LABORATORY ASSIGNMENT**

**(CSE606)**

**Bachelor of Technology (CSE)**

By

Heer Patel (22000384)

Third Year, Semester 6

# Course In-charge: Prof. Vaibhavi Patel

A close-up of a logo

AI-generated content may be incorrect.

Department of Computer Science and Engineering

School Engineering and Technology

Navrachana University, Vadodara

Spring Semester

(Jan-May 2025)

**TABLE OF CONTENT**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Sr. No** | **Experiment Title** | | | |
|  | | |
| 1 |  | 1. Write a program to recognize strings starts with ‘a’ over {a, b}. 2. Write a program to recognize strings end with ‘a’. 3. Write a program to recognize strings end with ‘ab’. Take the input from text file. 4. Write a program to recognize strings contains ‘ab’. Take the input from text file. |
| 2 |  | 1. Write a program to recognize the valid identifiers and keywords. 2. Write a program to recognize the valid operators. 3. Write a program to recognize the valid number. 4. Write a program to recognize the valid comments. 5. Program to implement Lexical Analyzer. |
| 3 |  | To Study about Lexical Analyzer Generator (LEX) and Flex(Fast Lexical Analyzer) |
| 4 |  | Implement following programs using Lex.  a. Write a Lex program to take input from text file and count no of characters, no. of lines & no. of words.  b. Write a Lex program to take input from text file and count number of vowels and consonants.  c. Write a Lex program to print out all numbers from the given file.  d. Write a Lex program which adds line numbers to the given file and display the same into different file.  e. Write a Lex program to printout all markup tags and HTML comments in file. |
| 5 |  | a. Write a Lex program to count the number of C comment lines from a given C program. Also eliminate them and copy that program into separate file.  b. Write a Lex program to recognize keywords, identifiers, operators, numbers, special symbols, literals from a given C program. |
| 6 |  | Program to implement Recursive Descent Parsing in C. |
| 7 |  | a. To Study about Yet Another Compiler-Compiler(YACC).  b. Create Yacc and Lex specification files to recognizes arithmetic expressions involving +, -, \* and / .  c. Create Yacc and Lex specification files are used to generate a calculator which accepts integer type arguments.  d. Create Yacc and Lex specification files are used to convert infix expression to postfix expression. |

1. **Write a program to recognize strings starts with ‘a’ over {a, b}.**

**CODE:**

#include <stdio.h>

int main()

{

char input[10];

int i=0,state=0;

printf("Enter a String: ");

scanf("%s",&input);

while(input[i]!='\0')

{

switch(state)

{

case 0:

if(input[i]=='a')

{

state=1;

}

else if(input[i]=='b')

{

state=2;

}

else

state=3;

break;

case 1:

if(input[i]=='a'||input[i]=='b')

{

state=1;

}

else

state=3;

break;

case 2:

if(input[i]=='a'||input[i]=='b')

{

state=2;

}

else

state=3;

break;

case 3:

state=3;

break;

}

i++;

}

if(state==1)printf("String is Valid");

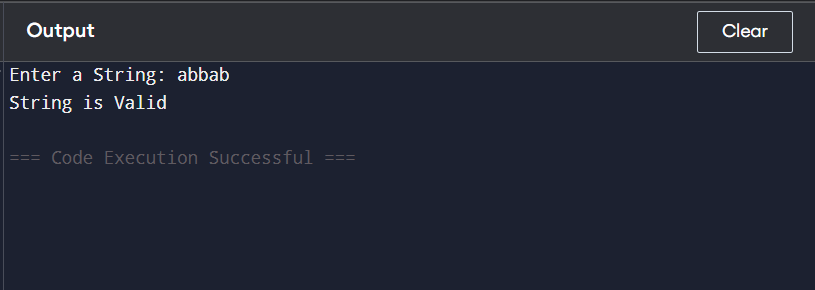
else if(state==0 || state==2)printf("String is Invalid");

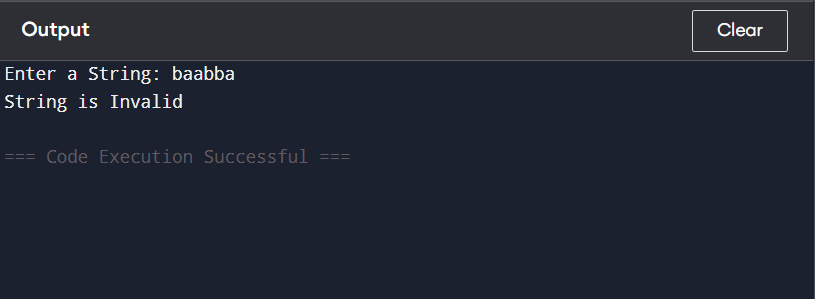
else printf("string is not recongized");

return 0;

}

**OUTPUT:**

****

****

1. **Write a program to recognize strings end with ‘a’.**

**CODE:**

// Online C compiler to run C program online

#include <stdio.h>

int main()

{

char input[10];

int i=0, state=0;

printf("Enter a string:" );

scanf("%s",&input);

while(input[i]!='\0')

{

switch(state)

{

case 0:

if (input[i]=='a')

{

state=1;

}

else if (input[i]=='b')

{

state=0;

}

else

state=2;

break;

case 1:

if (input[i]=='a')

{

state=1;

}

else if (input[i]=='b')

{

state=0;

}

else

state=2;

break;

case 2:

state=2;

break;

}

i++;

}

if(state==1){ printf("String is Valid");}

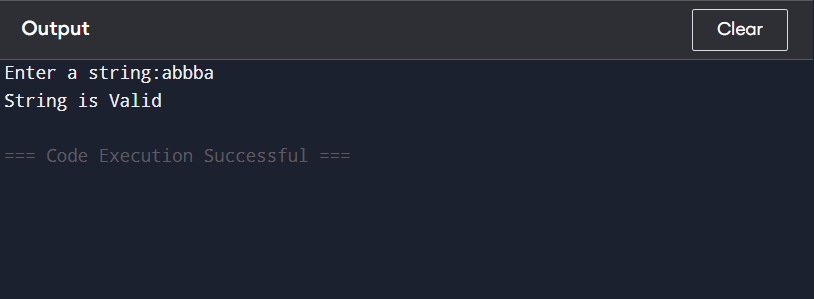
else if(state==0) {printf("String is Invalid");}

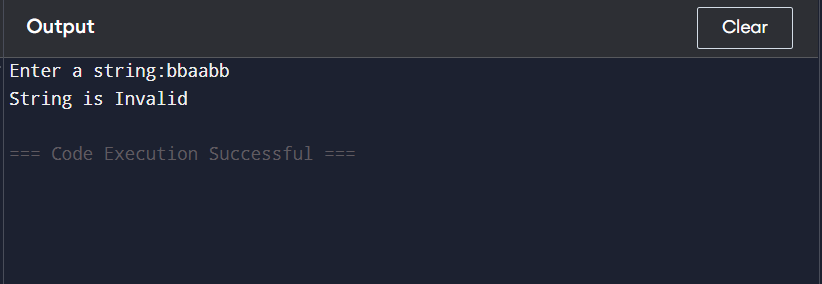
else {printf("String is not recongized");}

return 0;

}

**OUTPUT:**

****

****

1. **Write a program to recognize strings end with ‘ab’. Take the input from text file.**

**CODE:**

#include<stdio.h>

#include<stdlib.h>

int main(){

FILE \*file;

char input[100], c;

int state = 0, i = 0;

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

if (file == NULL) {

printf("Error opening file!\n");

return 1;

}

fscanf(file, "%s", input);

fclose(file);

while (input[i] != '\0') {

c = input[i];

switch (state) {

case 0:

if (c == 'a')

state = 1;

else

state = 3;

break;

case 1:

if (c == 'a')

state = 1;

else if (c == 'b')

state = 2;

else

state = 3;

break;

case 2:

if (c == 'a')

state = 1;

else if (c == 'b')

state = 2;

else

state = 3;

break;

case 3:

printf("\n %s is not recognized.\n", input);

return 0;

}

i++;

}

if (state == 2) // Final state should be 2 for valid strings ending in 'ab'

printf("\n %s is valid\n", input);

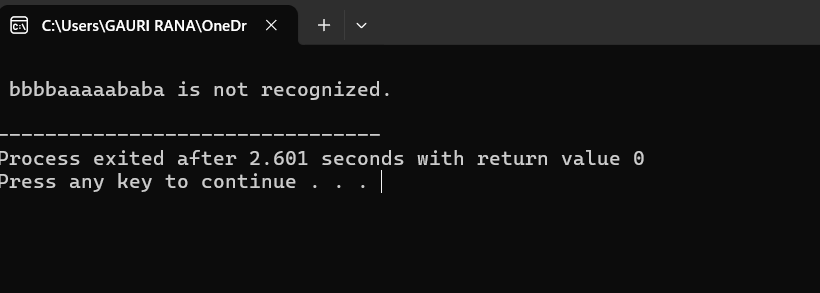
else

printf("\n %s is not valid\n", input);

return 0;

}

**OUTPUT:**



1. **Write a program to recognize strings contains ‘ab’. Take the input from text file.**

**CODE:**

#include<stdio.h>

#include<stdlib.h>

int main(){

FILE \*file;

char input[100], c;

int state = 0, i = 0;

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

if (file == NULL) {

printf("Error opening file!\n");

return 1;

}

fscanf(file, "%s", input);

fclose(file);

while (input[i] != '\0') {

c = input[i];

switch (state)

{

case 0:

if (c == 'a')

state = 1;

break;

case 1:

if (c == 'b')

{

state = 2;

}

else if (c != 'a')

{

state = 0;

}

break;

case 2:

printf("\n %s is valid\n", input);

return 0;

}

i++;

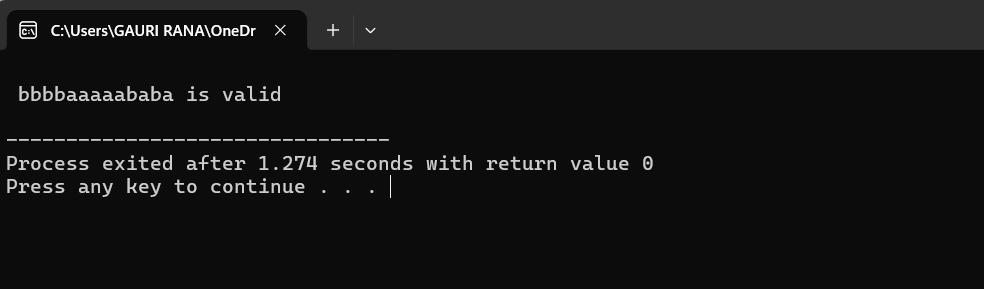
}

printf("\n %s is not valid\n", input);

return 0;

}

**OUTPUT:**

****

1. **Write a program to recognize the valid identifiers and keywords.**

**CODE:**

#include <stdio.h>

#include <string.h>

#include <ctype.h>

#define MAX\_LEN 100

#define START 0

#define LETTER\_STATE 1

#define INVALID\_STATE -1

const char keywords[][10] = {

"auto", "break", "case", "char", "const", "continue", "default", "do",

"double", "else", "enum", "extern", "float", "for", "goto", "if",

"inline", "int", "long", "register", "restrict", "return", "short",

"signed", "sizeof", "static", "struct", "switch", "typedef", "union",

"unsigned", "void", "volatile", "while"

};

int isKeyword(char str[MAX\_LEN])

{

int i;

int keywordCount = sizeof(keywords) / sizeof(keywords[0]);

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

{

if (strcmp(str, keywords[i]) == 0)

{

return 1; // It is a keyword

}

}

return 0; // Not a keyword

}

int isValidIdentifier(char str[MAX\_LEN])

{

int state = START;

int i, length = strlen(str);

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

{

switch (state)

{

case START:

if (isalpha(str[i]) || str[i] == '\_')

{

state = LETTER\_STATE; // Transition to LETTER\_STATE

}

else

{

state = INVALID\_STATE;

}

break;

case LETTER\_STATE:

if (isalnum(str[i]) || str[i] == '\_')

{

state = LETTER\_STATE; // Stay in LETTER\_STATE

}

else

{

state = INVALID\_STATE; // Invalid character found

}

break;

case INVALID\_STATE:

return 0; // If invalid state is reached, return false

}

}

return (state == LETTER\_STATE); // Valid identifier if it ends in LETTER\_STATE

}

int main() {

char input[MAX\_LEN];

printf("Enter a string: ");

scanf("%s", input);

if (isKeyword(input))

{

printf("\"%s\" is a keyword.\n", input);

}

else if (isValidIdentifier(input))

{

printf("\"%s\" is a valid identifier.\n", input);

}

else

{

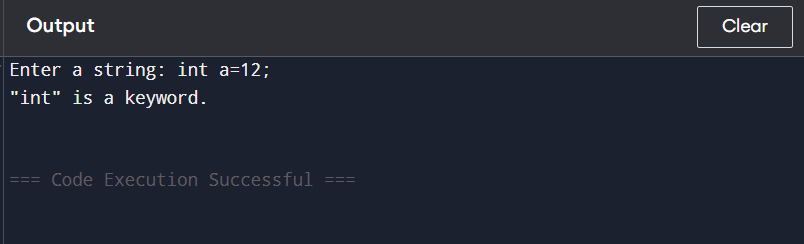
printf("\"%s\" is NOT a valid identifier or keyword.\n", input);

}

return 0;

}

**OUTPUT:**



1. **Write a program to recognize the valid operators.**

**CODE:**

#include <stdio.h>

#include <string.h>

// Function to check if input is a valid operator

int isValidOperator(const char \*op) {

const char \*validOperators[] = {

"+", "-", "\*", "/", "%", // Arithmetic

"==", "!=", ">", "<", ">=", "<=", // Relational

"&&", "||", "!", // Logical

"=", "+=", "-=", "\*=", "/=", "%=", // Assignment

"&", "|", "^", "~", "<<", ">>", // Bitwise

"++", "--" // Increment/Decrement

};

int n = sizeof(validOperators) / sizeof(validOperators[0]);

for (int i = 0; i < n; i++) {

if (strcmp(op, validOperators[i]) == 0) {

return 1;

}

}

return 0;

}

int main() {

char input[10];

printf("Enter an operator: ");

scanf("%s", input);

if (isValidOperator(input)) {

printf("\"%s\" is a valid C operator.\n", input);

} else {

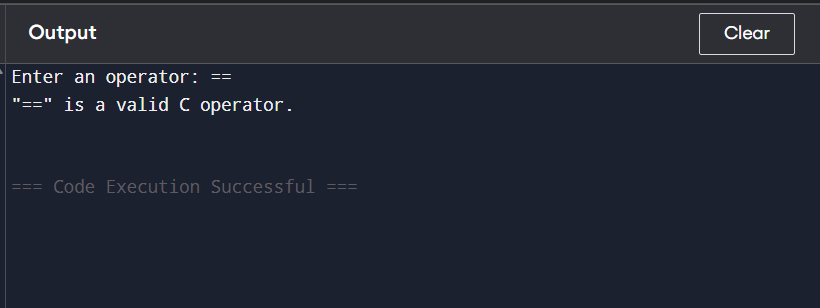
printf("\"%s\" is NOT a valid C operator.\n", input);

}

return 0;

}

**OUTPUT:**



1. Write a program to recognize the valid number.

**CODE:**

#include <stdio.h>

#include <ctype.h>

#include <string.h>

// Function to check if a string is a valid number

int isValidNumber(const char \*str) {

int i = 0, hasDecimal = 0, hasExponent = 0;

// Optional sign

if (str[i] == '+' || str[i] == '-') {

i++;

}

int digitBeforeE = 0;

while (str[i] != '\0') {

if (isdigit(str[i])) {

digitBeforeE = 1;

i++;

} else if (str[i] == '.' && !hasDecimal && !hasExponent) {

hasDecimal = 1;

i++;

} else if ((str[i] == 'e' || str[i] == 'E') && !hasExponent && digitBeforeE) {

hasExponent = 1;

i++;

// Exponent sign

if (str[i] == '+' || str[i] == '-') {

i++;

}

// Must be followed by digits

if (!isdigit(str[i])) return 0;

} else {

return 0;

}

}

return digitBeforeE;

}

int main() {

char input[100];

printf("Enter a number: ");

scanf("%s", input);

if (isValidNumber(input)) {

printf("\"%s\" is a valid number.\n", input);

} else {

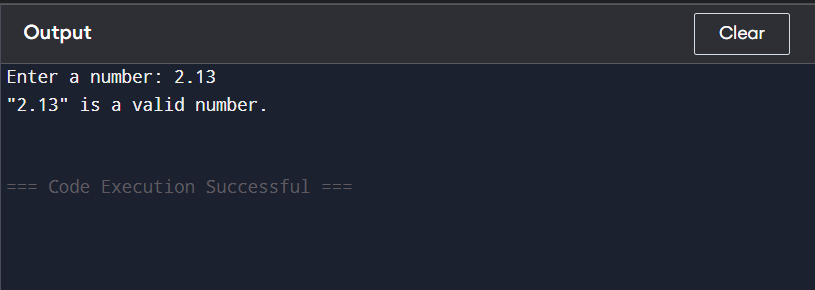
printf("\"%s\" is NOT a valid number.\n", input);

}

return 0;

}

**OUTPUT:**



1. Write a program to recognize the valid comments.

CODE:

#include <stdio.h>

#include <string.h>

// Function to check if the input is a valid comment

int isValidComment(const char \*str) {

int len = strlen(str);

// Check for single-line comment

if (len >= 2 && str[0] == '/' && str[1] == '/') {

return 1; // Valid single-line comment

}

// Check for multi-line comment

if (len >= 4 && str[0] == '/' && str[1] == '\*'

&& str[len - 2] == '\*' && str[len - 1] == '/') {

return 1; // Valid multi-line comment

}

return 0; // Not a valid comment

}

int main() {

char input[1000];

printf("Enter a comment: ");

fgets(input, sizeof(input), stdin); // Use fgets to allow spaces and multi-line symbols

// Remove trailing newline from fgets

size\_t ln = strlen(input);

if (ln > 0 && input[ln - 1] == '\n') {

input[ln - 1] = '\0';

}

if (isValidComment(input)) {

printf("The input is a valid comment.\n");

} else {

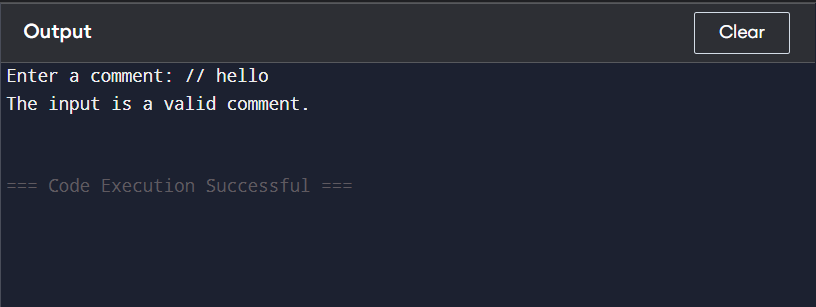
printf("The input is NOT a valid comment.\n");

}

return 0;

}

OUTPUT:

****

1. Program to implement Lexical Analyzer.

CODE:

#include <stdio.h>

#include <ctype.h>

#include <string.h>

#define MAX 100

// List of keywords in C

const char \*keywords[] = {

"int", "float", "char", "double", "if", "else", "while",

"for", "do", "return", "break", "continue", "void", "switch",

"case", "default", "const", "static", "struct", "typedef"

};

int keywordCount = sizeof(keywords) / sizeof(keywords[0]);

// Function to check if a string is a keyword

int isKeyword(const char \*str) {

for (int i = 0; i < keywordCount; i++) {

if (strcmp(str, keywords[i]) == 0)

return 1;

}

return 0;

}

// Function to check if a character is a special symbol

int isSpecialSymbol(char ch) {

return strchr(";(){}[],", ch) != NULL;

}

// Function to check if a character is an operator

int isOperator(char ch) {

return strchr("+-\*/=<>!&|%", ch) != NULL;

}

int main() {

char input[MAX];

char token[MAX];

int i = 0, j = 0;

printf("Enter the input code (single line):\n");

fgets(input, MAX, stdin);

while (input[i] != '\0')

{

if (isspace(input[i]))

{

i++;

continue;

}

// Comments (Check this BEFORE operators!)

if (input[i] == '/' && input[i + 1] == '/')

{

printf("Single-line Comment:");

i += 2;

while (input[i] != '\0' && input[i] != '\n')

{

putchar(input[i++]);

}

printf("\n");

break; // ignore rest of line after comment

}

// Multi-line comment

else if (input[i] == '/' && input[i + 1] == '\*')

{

printf("Multi-line Comment:");

i += 2;

while (!(input[i] == '\*' && input[i + 1] == '/') && input[i] != '\0')

{

putchar(input[i++]);

}

i += 2; // Skip '\*/'

printf("\n");

}

// Identifiers or Keywords

else if (isalpha(input[i]) || input[i] == '\_')

{

j = 0;

while (isalnum(input[i]) || input[i] == '\_')

{

token[j++] = input[i++];

}

token[j] = '\0';

if (isKeyword(token))

{

printf("Keyword: %s\n", token);

}

else

{

printf("Identifier: %s\n", token);

}

}

// Numbers

else if (isdigit(input[i]))

{

j = 0;

while (isdigit(input[i]))

{

token[j++] = input[i++];

}

token[j] = '\0';

printf("Number: %s\n", token);

}

// Operators

else if (isOperator(input[i])) {

if ((input[i] == '+' || input[i] == '-' || input[i] == '=' || input[i] == '!' || input[i] == '<' || input[i] == '>') && input[i + 1] == '=')

{

printf("Operator: %c%c\n", input[i], input[i + 1]);

i += 2;

}

else if ((input[i] == '&' || input[i] == '|') && input[i + 1] == input[i])

{

printf("Operator: %c%c\n", input[i], input[i + 1]);

i += 2;

}

else

{

printf("Operator: %c\n", input[i++]);

}

}

// Special Symbols

else if (isSpecialSymbol(input[i]))

{

printf("Special Symbol: %c\n", input[i++]);

}

// Unknown Characters

else

{

printf("Unknown Character: %c\n", input[i++]);

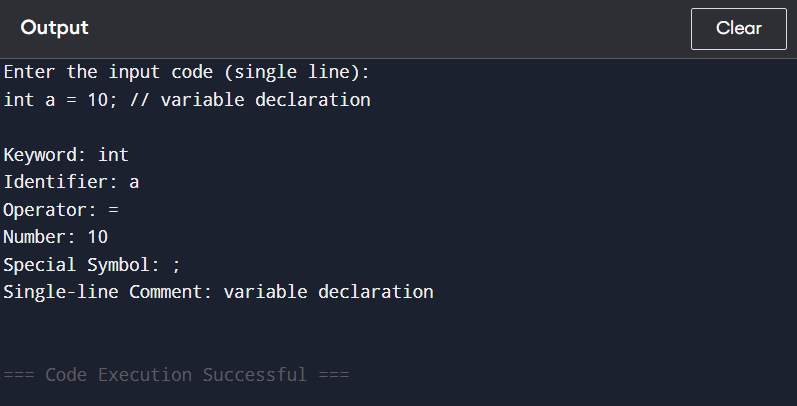
}

}

return 0;

}

OUTPUT:

****

1. **To Study about Lexical Analyzer Generator (LEX) and Flex (Fast Lexical Analyzer)**

What is a Lexical Analyzer?

A Lexical Analyzer (Lexer or Scanner) is the first phase of a compiler. It:

* Reads the source code character by character.
* Groups characters into tokens (identifiers, keywords, numbers, operators, etc.).
* Passes these tokens to the parser (syntax analyzer).

What is LEX?

LEX is a Lexical Analyzer Generator:

* Developed in the 1970s by AT&T Bell Labs.
* You write rules in **a .l file** (LEX file) using regular expressions.
* LEX generates a C program that performs lexical analysis.

Structure of a LEX/Flex Program

a.l (LEX/Flex) file is divided into three sections:

%{

// C declarations and headers

%}

%%

// Rules section

REGEX { C code to execute when REGEX matches }

%%

// User-defined functions (main, helpers, etc.)

1. **Implement following programs using Lex.**
2. **Write a Lex program to take input from text file and count no of characters, no. of lines & no. of words.**

**CODE:**

%{

#include <stdio.h>

int characters = 0;

int words = 0;

int lines = 0;

%}

%%

[a-zA-Z0-9]+ { words++; characters += yyleng; }

\n { lines++; characters++; }

. { characters++; }

%%

void main() {

yyin = fopen("cd1.txt", "r");

if (!yyin) {

printf("Cannot open file.\n");

return;

}

yylex();

printf("Number of lines: %d\n", lines);

printf("Number of words: %d\n", words);

printf("Number of characters: %d\n", characters);

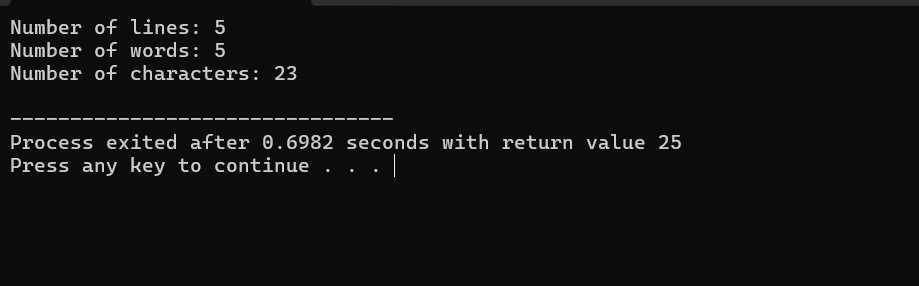
}

int yywrap() {

return 1;

}

**OUTPUT:**

****

1. **Write a Lex program to take input from text file and count number of vowels and consonants.**

**CODE:**

%{

#include<stdio.h>

int consonants=0,vowels=0;

%}

%%

[aeiouAEIOU] {vowels++;}

[a-zA-Z] {consonants++;}

\n ;

. ;

%%

int main() {

yyin=fopen("cd1.txt","r");

yylex();

printf("This file conatins...");

printf("\n\t%d vowels ",vowels);

printf("\n\t%d consonants ",consonants);

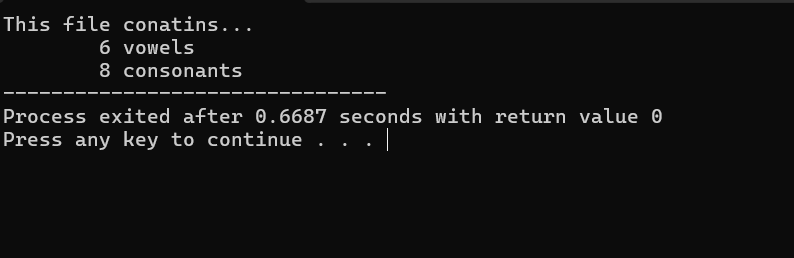
return 0;

}

int yywrap(){ return(1);

}

**OUTPUT:**

****

1. **Write a Lex program to print out all numbers from the given file.**

**CODE:**

%{

#include<stdio.h>

%}

%%

[0-9]+(\.[0-9]+)?([eE][+-]?[0-9]+)? printf("%s is valid number \n",yytext);

\n ;

. ;

%%

int main()

{

yyin=fopen("number.txt","r");

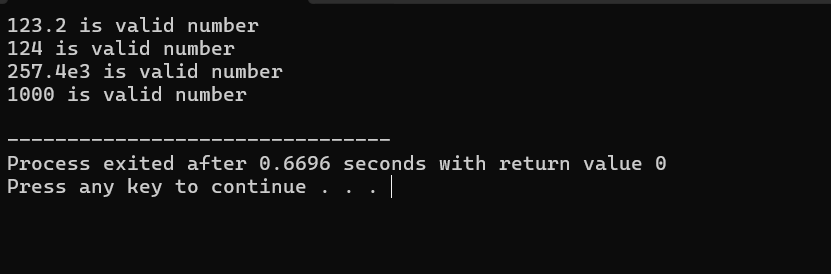
yylex();

return 0;

}

int yywrap(){return(1);}

**OUTPUT:**

****

1. **Write a Lex program which adds line numbers to the given file and display the same into different file.**

**CODE:**

%{

int line\_number = 1;

%}

%%

.+ {fprintf(yyout,"%d: %s",line\_number,yytext);line\_number++;}

%%

int main()

{

yyin=fopen("cd1.txt","r");

yyout=fopen("26\_cd3.txt","w");

yylex();

printf("done");

return 0;

}

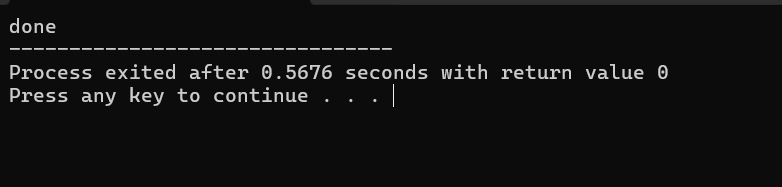
int yywrap()

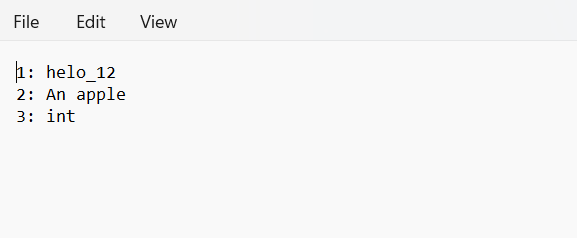
{

return(1);

}

**OUTPUT:**

****

****

1. **Write a Lex program to printout all markup tags and HTML comments in file.**

**CODE:**

%{

#include<stdio.h>

int num=0;

%}

%%

"<"[/A-Za-z0-9]+">" printf("%s is valid markup tag\n",yytext);

"<!--"[^--]\*"-->" num++;

\n ;

. ;

%%

int main()

{

yyin=fopen("myfile.txt","r");

yylex();

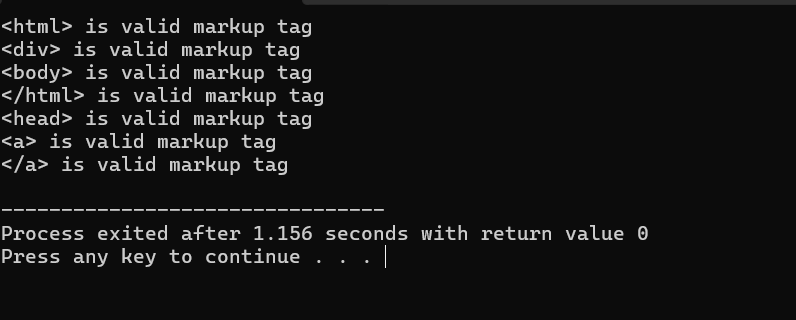
return 0;

}

int yywrap(){return(1);

}

**OUTPUT:**

****

1. **Write a Lex program to count the number of C comment lines from a given C program. Also eliminate them and copy that program into separate file.**

**CODE:**

%{

#include <stdio.h>

int comment\_lines = 0;

FILE \*out;

%}

%x COMMENT

%%

"//".\* { comment\_lines++; } // Single-line comment

"/\*" { comment\_lines++; BEGIN(COMMENT); } // Start of multi-line

<COMMENT>"\*/" { BEGIN(INITIAL); } // End of multi-line

<COMMENT>\n { comment\_lines++; } // Count line inside comment

<COMMENT>. { } // Ignore other characters

\n { fputc('\n', out); } // Write newlines

. { fputc(yytext[0], out); } // Write everything else

%%

int main() {

FILE \*in = fopen("input1.txt", "r"); // Read from text file

if (!in) {

printf("Cannot open input.txt\n");

return 1;

}

out = fopen("output1.txt", "w"); // Write to text file

if (!out) {

printf("Cannot create output.txt\n");

return 1;

}

yyin = in;

yylex();

printf("Total comment lines: %d\n", comment\_lines);

fclose(in);

fclose(out);

return 0;

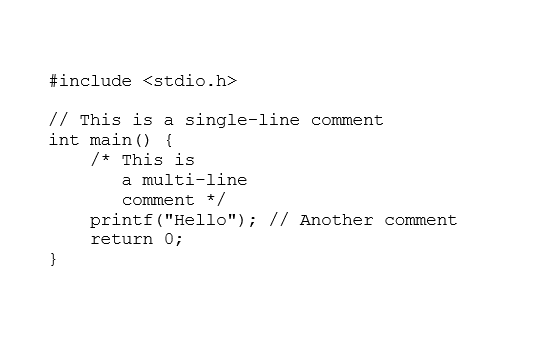
}

int yywrap() {

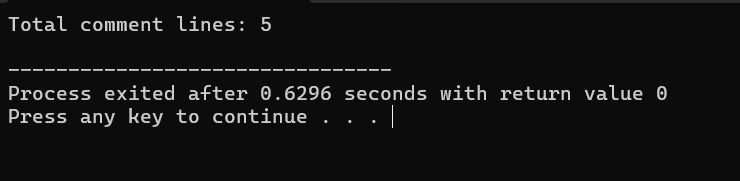
return 1;

}

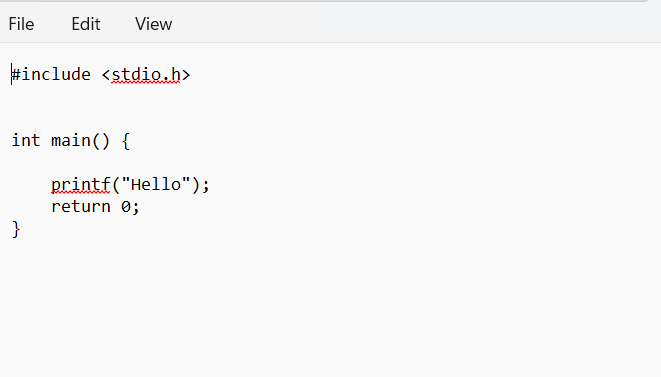
Input1.txt



**OUTPUT:**

****

Output1.txt



1. **Write a Lex program to recognize keywords, identifiers, operators, numbers, special symbols, literals from a given C program.**

**CODE:**

%{

#include <stdio.h>

%}

%%

"int"|"float"|"if"|"else"|"while"|"return"|"char"|"for"|"do"|"switch" { printf("%s is a keyword\n", yytext); }

[aeiouAEIOU] {printf("%s is a vowels\n",yytext);}

[a-zA-Z\_][a-zA-Z0-9\_]\* { printf("%s is an identifier\n", yytext); }

%%

int main() {

yyin=fopen("cd1.txt","r");

yylex();

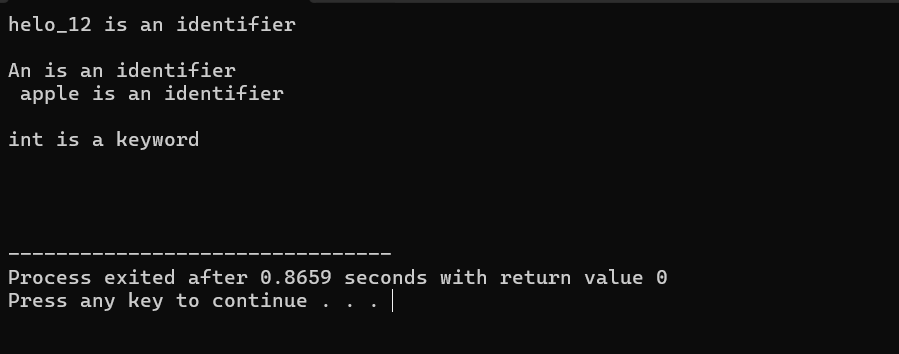
return 0;

}

int yywrap() { return 1;

}

**OUTPUT:**

****

1. **Program to implement Recursive Descent Parsing in C.**

**CODE:**

// Recursive Descent Parser

#include <stdio.h>

#include <stdlib.h>

char s[20]; // input string

int i = 1; // start from index 1

char l; // current character

void match(char t)

{

if (l == t)

{

l = s[i];

i++;

}

else

{

printf("Syntax error\n");

exit(1);

}

}

void E\_() // E' -> +iE' | -iE' | e

{

if (l == '+')

{

match('+');

match('i');

E\_();

}

else if (l == '-')

{

match('-');

match('i');

E\_();

}

else

{

return;

}

}

void E() // E -> iE'

{

if (l == 'i')

{

match('i');

E\_();

}

else

{

printf("Syntax error\n");

exit(1);

}

}

int main()

{

printf("Enter the string (end with $): ");

scanf("%s", s);

l = s[0];

E();

if (l == '$')

{

printf("String is valid\n");

}

else

{

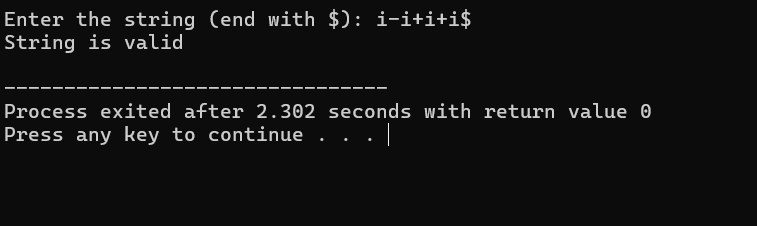
printf("Syntax error\n");

}

return 0;

}

**OUTPUT:**

****

1. **To Study about Yet Another Compiler-Compiler (YACC).**

YACC (Yet Another Compiler-Compiler) is a parser generator for context-free grammars, used to build the syntax analysis phase of a compiler.

* It generates C code for a parser.
* Works closely with a Lex (lexer) program.
* Parses tokens (provided by Lex) based on grammar rules.

**Workflow: Lex + YACC**

Source Code --> Lex (Tokenizer) --> Tokens --> YACC (Parser) --> Syntax Tree / Actions

Structure of a YACC File (.y file)

%{

// C declarations (headers, variables, etc.)

%}

%token ID NUM // token declarations

%%

// Grammar rules

expr : expr '+' term { printf("Add\n"); }

| term ;

term : NUM { printf("Number\n"); }

;

%%

int main()

{

return yyparse(); // Start parsing

}

int yyerror(char \*msg) {

printf("Syntax Error: %s\n", msg);

return 0;

}

1. **Create Yacc and Lex specification files to recognizes arithmetic expressions involving +, -, \* and / .**

**CODE:**

**n.l file:**

%{

#include<stdlib.h>

void yyerror(char \*);

#include"n.tab.h"

%}

%%

[0-9]+ {yylval = atoi(yytext); return NUM;}

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

[-+\*\n] {return \*yytext;}

[ \t] { }

. yyerror("invlaid character");

%%

int yywrap(){

return 0;

}

**n.y file:**

%{

#include <stdio.h>

void yyerror (char \*);

int yylex(void);

%}

%token NUM

%token id

%%

S:E '\n' {printf("valid syntax"); return (0);}

E:E'+'T { }

|E'-'T { }

|T { }

T:T'\*'F { }

|F { }

F: NUM { }

| id { }

%%

void yyerror(char \*s){

fprintf(stderr,"%s\n",s);

}

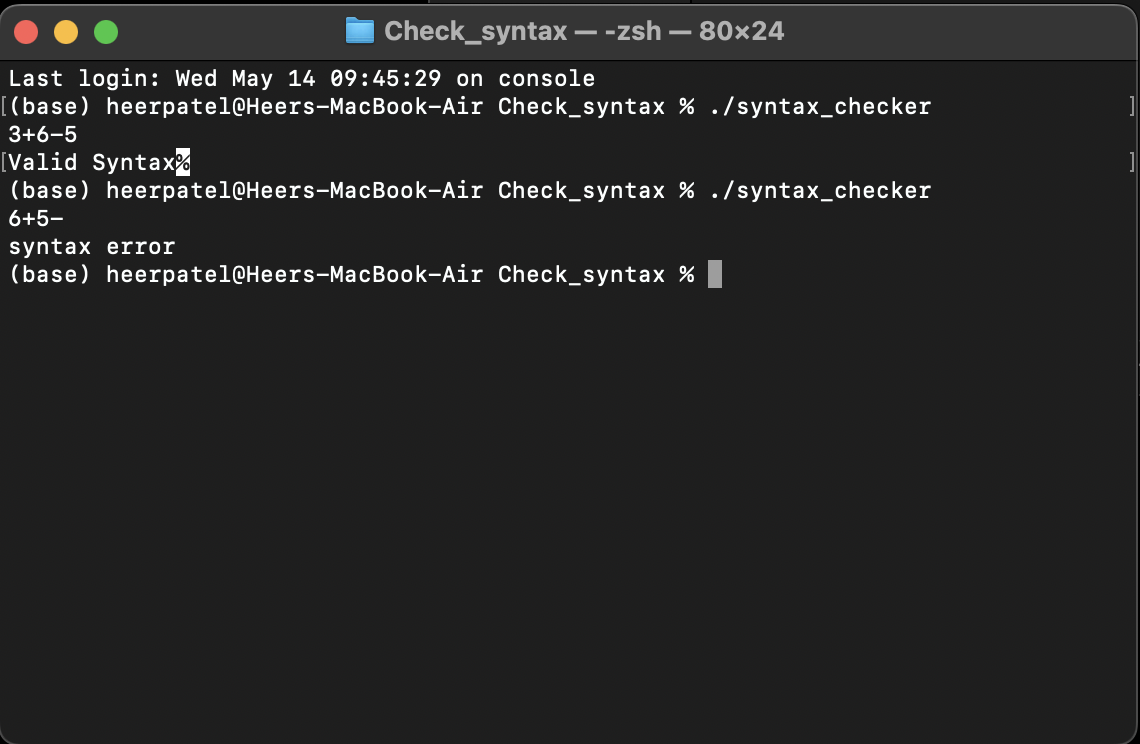
int main(){

yyparse();

return 0;

}

**OUTPUT:**

****

1. **Create Yacc and Lex specification files are used to generate a calculator which accepts integer type arguments.**

**CODE:**

**n.l file:**

%{

#include <stdlib.h>

void yyerror(char \*);

#include "n.tab.h"

%}

%%

[0-9]+ {yylval = atoi(yytext); return NUM;}

[-+\*\n] {return \*yytext;}

[ \t] { }

. yyerror ("invalid charcter");

%%

int yywrap(){

return 0;

}

**n.y file:**

%{

#include <stdio.h>

int yylex(void);

void yyerror(char \*);

%}

%token NUM

%%

S:E '\n' {printf("%d\n",$1); return (0);}

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

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

| T {$$ = $1;}

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

| F {$$ = $1;}

F:NUM {$$= $1;}

%%

void yyerror(char \*s){

fprintf(stderr,"%s\n",s);

}

int main(){

yyparse();

return 0;

}

**OUTPUT:**

**A computer screen shot of a program

AI-generated content may be incorrect.**

1. **Create Yacc and Lex specification files are used to convert infix expression to postfix expression.**

**CODE:**

**n.l file:**

%{

#include <stdlib.h>

void yyerror(char \*);

#include "n.tab.h"

%}

%%

[0-9]+ {yylval.num = atoi(yytext); return INTEGER;}

[a-zA-Z\_][a-zA-Z\_0-9]\* {yylval.str = yytext; return ID;}

[-+\*\n] {return \*yytext;}

[ \t] ;

. yyerror("invalid character");

%%

int yywrap(){

return 1;

}

**n.y file:**

%{

#include <stdio.h>

int yylex(void);

void yyerror(char \*);

%}

%union{

char \*str;

int num;

}

%token <num> INTEGER

%token <str> ID

%%

S:E '\n' {printf("\n");}

E:E'+'T {printf("+");}

|E'-'T {printf("-");}

|T { }

T:T'\*'F {printf("\*");}

|F { }

F: INTEGER {printf("%d",$1);}

| ID {printf("%s",$1);}

%%

void yyerror(char \*s){

fprintf(stderr,"%s\n",s);

}

int main(){

yyparse();

return 0;

}

**OUTPUT:**

A computer screen with white text

AI-generated content may be incorrect.