****

**School of Engineering and Technology**

**Computer Science and Engineering Department**

**Laboratory Manual**

TY B Tech CSE - VI semester

Academic Year: 2024-25

Complier Design Laboratory(CS606)

Kirtan Visnagara(22000392)

**Course In-charge**

Prof. Vaibhavi Patel

Table of contents:

|  |  |  |
| --- | --- | --- |
| **Sr. No.** | **Task/ Practical/ Activity** | **Pg. No.** |
| 1. | a) Write a program to recognize strings starts with ‘a’ over {a, b}.  b) Write a program to recognize strings end with ‘a’.  c) Write a program to recognize strings end with ‘ab’. Take the input from text file.  d) Write a program to recognize strings contains ‘ab’. Take the input from text file. | 3 |
| 2. | a) Write a program to recognize the valid identifiers and keywords.  b) Write a program to recognize the valid operators.  c) Write a program to recognize the valid number.  d) Write a program to recognize the valid comments.  e) Program to implement Lexical Analyzer. | 8 |
| 3. | To Study about Lexical Analyzer Generator (LEX) and Flex(Fast Lexical Analyzer. | 17 |
| 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. | 19 |
| 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 | 26 |
| 6. | Program to implement Recursive Descent Parsing in C. | 30 |
| 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. | 32 |

**PRACTICAL 1**

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

b) Write a program to recognize strings end with ‘a’.

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

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

**Input:** a) #include <stdio.h>

int main() {

char str[100];

int i = 0;

printf("Enter a string over {a, b}: ");

scanf("%s", str);

switch (str[0]) {

case 'a':

printf("The string starts with 'a'.\n");

break;

case 'b':

printf("The string does NOT start with 'a'.\n");

break;

default:

printf("Invalid input. Only 'a' and 'b' are allowed.\n");

break;

}

return 0;

}

b) #include <stdio.h>

#include <string.h>

int main() {

char str[100];

int len;

printf("Enter a string over {a, b}: ");

scanf("%s", str);

len = strlen(str);

// Check if all characters are from {a, b}

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

if (str[i] != 'a' && str[i] != 'b') {

printf("Invalid input. Only 'a' and 'b' are allowed.\n");

return 0;

}

}

switch (str[len - 1]) {

case 'a':

printf("The string ends with 'a'.\n");

break;

case 'b':

printf("The string does NOT end with 'a'.\n");

break;

}

return 0;

}

c) #include <stdio.h>

#include <string.h>

#include <stdlib.h>

int main() {

char str[100];

FILE \*fp;

// Open the file in read mode

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

if (fp == NULL) {

printf("Error: Could not open file.\n");

return 1;

}

// Read each line (string) from the file

while (fscanf(fp, "%s", str) != EOF) {

int len = strlen(str);

int valid = 1;

// Validate that the string only contains 'a' or 'b'

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

if (str[i] != 'a' && str[i] != 'b') {

valid = 0;

break;

}

}

if (!valid) {

printf("Invalid string (not over {a,b}): %s\n", str);

continue;

}

// Check if the string ends with "ab"

if (len >= 2 && str[len - 2] == 'a' && str[len - 1] == 'b') {

printf("Accepted: %s\n", str);

} else {

printf("Rejected: %s\n", str);

}

}

fclose(fp);

return 0;

}

d) #include <stdio.h>

#include <string.h>

int main() {

char str[100];

FILE \*fp;

// Open the file for reading

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

if (fp == NULL) {

printf("Error: Could not open input.txt\n");

return 1;

}

// Read strings from the file one by one

while (fscanf(fp, "%s", str) != EOF) {

int len = strlen(str);

int valid = 1;

// Check if the string only contains 'a' and 'b'

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

if (str[i] != 'a' && str[i] != 'b') {

valid = 0;

break;

}

}

if (!valid) {

printf("Invalid string (not over {a,b}): %s\n", str);

continue;

}

// Check if the string contains the substring "ab"

if (strstr(str, "ab") != NULL) {

printf("Accepted (contains 'ab'): %s\n", str);

} else {

printf("Rejected (no 'ab'): %s\n", str);

}

}

fclose(fp);

return 0;

}

**input.txt:**

ab

aab

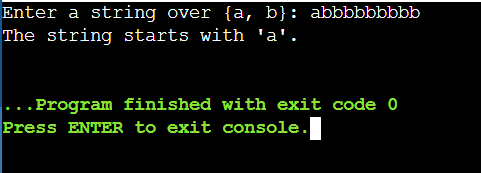
baba

baa

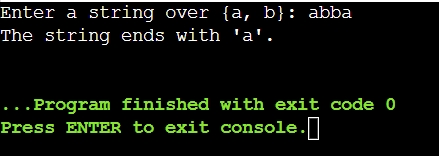
abb

bb

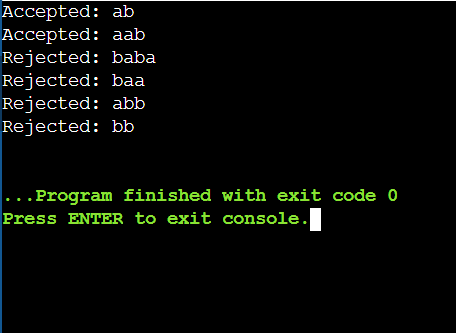
**Output:** a)



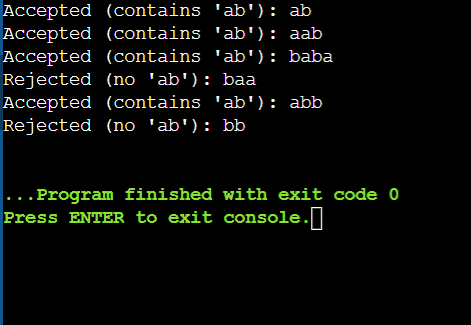
b)



c)



d)



**PRACTICAL 2**

**Task:** a) Write a program to recognize the valid identifiers and keywords.

b) Write a program to recognize the valid operators.

c) Write a program to recognize the valid number.

d) Write a program to recognize the valid comments.

e) Program to implement Lexical Analyzer.

**Input:** a) #include <stdio.h>

#include <string.h>

#include <ctype.h>

// Function to check if the string is a keyword

int isKeyword(char str[]) {

char \*keywords[] = {

"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"

};

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

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

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

return 1;

}

return 0;

}

// Main function

int main() {

char str[100];

int valid = 1;

printf("Enter a string: ");

scanf("%s", str);

// First, check if it's a keyword

if (isKeyword(str)) {

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

return 0;

}

// Check the first character using switch-case

switch (str[0]) {

case 'a' ... 'z':

case 'A' ... 'Z':

case '\_':

// Valid start, check the rest

for (int i = 1; str[i] != '\0'; i++) {

if (!isalnum(str[i]) && str[i] != '\_') {

valid = 0;

break;

}

}

if (valid)

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

else

printf("'%s' is NOT a valid identifier.\n", str);

break;

default:

printf("'%s' is NOT a valid identifier.\n", str);

break;

}

return 0;

}

b) #include <stdio.h>

#include <string.h>

int isOperator(char op[]) {

// List of valid C operators

char \*operators[] = {

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

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

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

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

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

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

"?", ":", // Conditional

"->", ".", // Structure and member access

"sizeof", // Sizeof

};

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

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

if (strcmp(op, operators[i]) == 0)

return 1;

}

return 0;

}

int main() {

char input[10];

printf("Enter an operator: ");

scanf("%s", input);

if (isOperator(input)) {

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

} else {

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

}

return 0;

}

c) #include <stdio.h>

#include <ctype.h>

#include <string.h>

int isValidNumber(const char \*str) {

int i = 0, dotCount = 0, eCount = 0;

// Optional sign at the beginning

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

i++;

}

int digitBeforeE = 0;

for (; str[i] != '\0'; i++) {

if (isdigit(str[i])) {

digitBeforeE = 1;

continue;

} else if (str[i] == '.') {

if (dotCount || eCount) return 0; // Only one dot and it must be before 'e'

dotCount++;

} else if (str[i] == 'e' || str[i] == 'E') {

if (eCount || !digitBeforeE) return 0; // Only one 'e' and must follow digits

eCount++;

i++;

if (str[i] == '-' || str[i] == '+') i++; // Allow sign after 'e'

if (!isdigit(str[i])) return 0; // Must be digits after 'e'

while (isdigit(str[i])) i++;

if (str[i] != '\0') return 0; // No extra characters after exponent digits

return 1;

} else {

return 0; // Invalid character

}

}

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;

}

d) #include <stdio.h>

#include <string.h>

int main() {

char comment[200];

int i = 0;

printf("Enter a comment: ");

fgets(comment, sizeof(comment), stdin);

// Remove trailing newline if exists

size\_t len = strlen(comment);

if (comment[len - 1] == '\n') {

comment[len - 1] = '\0';

len--;

}

switch (comment[0]) {

case '/':

switch (comment[1]) {

case '/':

printf("Valid single-line comment.\n");

break;

case '\*':

// Check for closing '\*/'

i = 2;

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

switch (comment[i]) {

case '\*':

if (comment[i + 1] == '/') {

printf("Valid multi-line comment.\n");

goto end;

}

break;

}

i++;

}

printf("Invalid comment: multi-line not properly closed.\n");

break;

default:

printf("Invalid comment: doesn't start properly.\n");

break;

}

break;

default:

printf("Invalid comment: doesn't start with '/'.\n");

break;

}

end:

return 0;

}

e) #include <stdio.h>

#include <string.h>

#include <ctype.h>

char \*keywords[] = {

"int", "float", "char", "if", "else", "while", "for", "return", "void", "main"

};

int isKeyword(char \*str) {

for (int i = 0; i < sizeof(keywords) / sizeof(keywords[0]); i++) {

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

return 1;

}

return 0;

}

int isOperator(char ch) {

return ch == '+' || ch == '-' || ch == '\*' || ch == '/' || ch == '=' ||

ch == '<' || ch == '>' || ch == '!' || ch == '%';

}

int main() {

char line[256], token[100];

int i = 0, j = 0;

printf("Enter a line of code:\n");

fgets(line, sizeof(line), stdin);

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

char ch = line[i];

// Skip whitespace

if (isspace(ch)) {

i++;

continue;

}

// Identifiers or Keywords

if (isalpha(ch) || ch == '\_') {

j = 0;

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

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

}

token[j] = '\0';

if (isKeyword(token))

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

else

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

}

// Numbers

else if (isdigit(ch)) {

j = 0;

while (isdigit(line[i])) {

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

}

if (line[i] == '.') { // Handle float

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

while (isdigit(line[i])) {

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

}

}

token[j] = '\0';

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

}

// Operators

else if (isOperator(ch)) {

if ((ch == '=' || ch == '!' || ch == '<' || ch == '>') && line[i + 1] == '=') {

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

i += 2;

} else {

printf("Operator\t: %c\n", line[i]);

i++;

}

}

// Separators & Symbols

else {

switch (ch) {

case ';':

case ',':

case '(':

case ')':

case '{':

case '}':

case '[':

case ']':

printf("Symbol\t\t: %c\n", ch);

break;

default:

printf("Unknown char\t: %c\n", ch);

break;

}

i++;

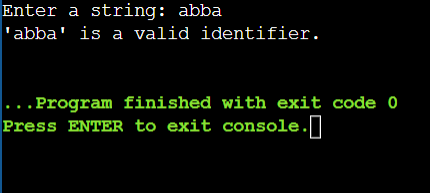
}

}

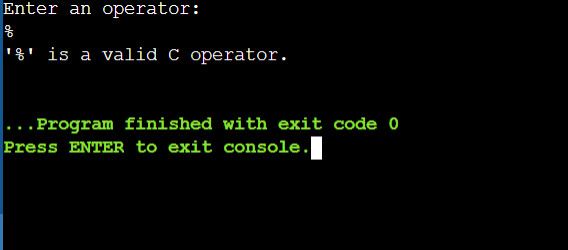
return 0;

}

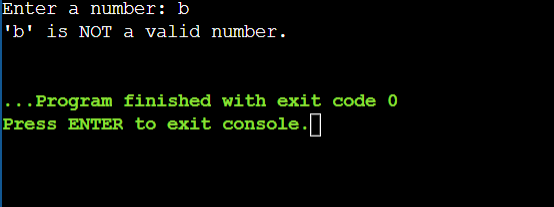
**Output:** a)



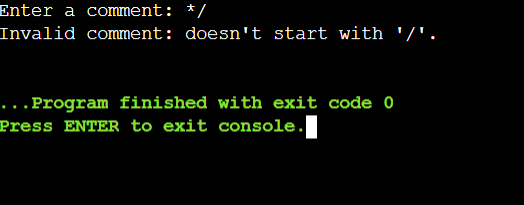
b)



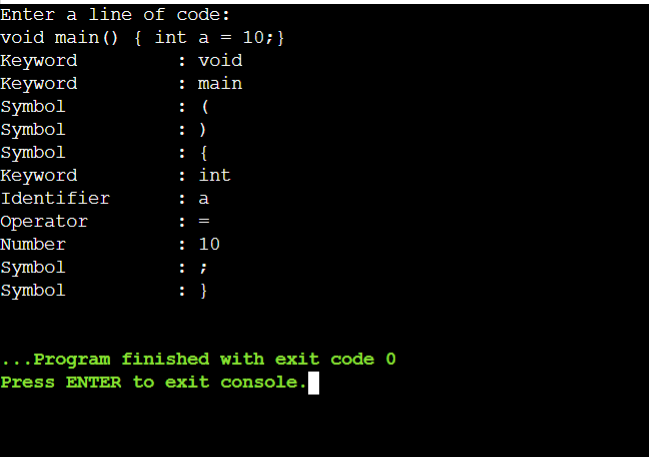
c)



d)



e)



**PRACTICAL 3**

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

**Theory:**

**What is a Lexical Analyzer?**

A **lexical analyzer** (or scanner) is the first phase of a compiler. It takes the source code as input and divides it into **tokens** such as keywords, identifiers, constants, operators, and symbols.

**What is LEX?**

* **LEX** is a **Lexical Analyzer Generator** tool developed in the 1970s.
* It takes a set of **regular expressions** (token patterns) and generates a C program (lex.yy.c) that performs lexical analysis.
* Works closely with **YACC**, a parser generator.

**What is Flex?**

* **Flex** stands for **Fast Lex**, and is an enhanced version of LEX.
* It's widely used due to its speed and compatibility with modern systems.
* Generates a yylex() function that scans the input and recognizes tokens based on rules.

**Structure of a LEX/Flex Program:**

A typical LEX or Flex program is divided into **three sections**, separated by %%:

%{

/\* C Declarations (headers, variables) \*/

%}

%%

/\* Regular expression rules and actions \*/

[0-9]+ { printf("Number: %s\n", yytext); }

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

"+" { printf("Operator: +\n"); }

. { /\* ignore unrecognized characters \*/ }

%%

int yywrap() { return 1; }

**Steps to Compile and Run a Flex Program:**

1. **Write the Flex file**, e.g., sample.l
2. **Generate the C code:** flex sample.l
3. **Compile the generated file:** gcc lex.yy.c -lfl
4. **Run the executable:** ./a.out

**Key Functions and Terms:**

| **Term** | **Meaning** |
| --- | --- |
| yytext | Holds the matched string (token) |
| yylex() | The main function that Flex generates to scan tokens |
| yywrap() | Called at end-of-file; returns 1 to stop scanning |
| lex.yy.c | The generated C source file from Flex |

**Applications:**

* Used in building compilers and interpreters
* Lexical analysis in programming language processors
* Parsing configuration files, interpreters, etc.

**PRACTICAL 4**

**Task:** 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 that adds line numbers to the given file and displays the same into a different file.

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

**Input:** a)

**count.l:**

%{

#include <stdio.h>

int char\_count = 0;

int word\_count = 0;

int line\_count = 0;

%}

%%

\n { char\_count++; line\_count++; }

[ \t]+ { char\_count += yyleng; } // Whitespace between words

[A-Za-z0-9\_]+ { char\_count += yyleng; word\_count++; }

. { char\_count++; } // Any other character

%%

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

if (argc > 1) {

FILE \*file = fopen(argv[1], "r");

if (!file) {

perror("File opening failed");

return 1;

}

yyin = file;

}

yylex();

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

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

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

return 0;

}

int yywrap() {

return 1;

}

**input.txt:**

void main() {

char a;

printf("My name is Kirtan.");}

b)

**vowel\_consonant.l**

%{

#include <stdio.h>

int vowel\_count = 0;

int consonant\_count = 0;

%}

%%

[aAeEiIoOuU] { vowel\_count++; }

[b-df-hj-np-tv-zB-DF-HJ-NP-TV-Z] { consonant\_count++; }

.|\n { /\* Ignore other characters \*/ }

%%

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

if (argc > 1) {

FILE \*file = fopen(argv[1], "r");

if (!file) {

perror("File opening failed");

return 1;

}

yyin = file;

}

yylex();

printf("Number of vowels: %d\n", vowel\_count);

printf("Number of consonants: %d\n", consonant\_count);

return 0;

}

int yywrap() {

return 1;

}

**input.txt**

Hello World!

Lex is a tool used for lexical analysis.

It counts things like words, vowels, and lines.

12345 and symbols too! @#%&

c)

**print\_numbers.l**

%{

#include <stdio.h>

%}

%%

[0-9]+(\.[0-9]+)? { printf("Number found: %s\n", yytext); }

[ \t\n]+ { /\* Skip whitespace \*/ }

. { /\* Skip all other characters \*/ }

%%

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

if (argc > 1) {

FILE \*file = fopen(argv[1], "r");

if (!file) {

perror("File opening failed");

return 1;

}

yyin = file;

}

yylex();

return 0;

}

int yywrap() {

return 1;

}

**input.txt**

The total cost is 49.99 dollars.

You saved 20!

Next payment is in 15 days.

1.5 + 2 = 3.5

d)

**line\_numbering.l**

%{

#include <stdio.h>

int line\_num = 1;

FILE \*outfile;

%}

%%

^.\*\n {

fprintf(outfile, "%d: %s", line\_num++, yytext);

}

.|\n { /\* Skip stray characters \*/ }

%%

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

if (argc < 3) {

fprintf(stderr, "Usage: %s <input file> <output file>\n", argv[0]);

return 1;

}

FILE \*infile = fopen(argv[1], "r");

if (!infile) {

perror("Error opening input file");

return 1;

}

outfile = fopen(argv[2], "w");

if (!outfile) {

perror("Error opening output file");

fclose(infile);

return 1;

}

yyin = infile;

yylex();

fclose(infile);

fclose(outfile);

return 0;

}

int yywrap() {

return 1;

}

**input.txt**

This is the first line.

This is the second.

And this is the third.

e)

**html\_tags\_comments.l**

%{

#include <stdio.h>

%}

%%

"<!--"([^<]|<)\*?"-->" { printf("HTML Comment: %s\n", yytext); }

"<"[^>]\*">" { printf("HTML Tag: %s\n", yytext); }

.|\n { /\* Ignore other characters \*/ }

%%

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

if (argc < 2) {

fprintf(stderr, "Usage: %s <input HTML file>\n", argv[0]);

return 1;

}

FILE \*infile = fopen(argv[1], "r");

if (!infile) {

perror("Error opening input file");

return 1;

}

yyin = infile;

yylex();

fclose(infile);

return 0;

}

int yywrap() {

return 1;

}

**input.txt**

<!DOCTYPE html>

<html>

<head>

<!-- This is a comment -->

<title>Sample Page</title>

</head>

<body>

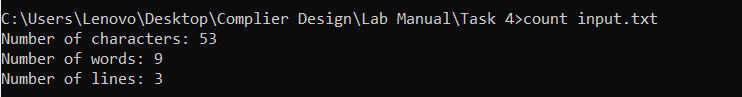
<h1>Hello World</h1>

<!-- Another comment -->

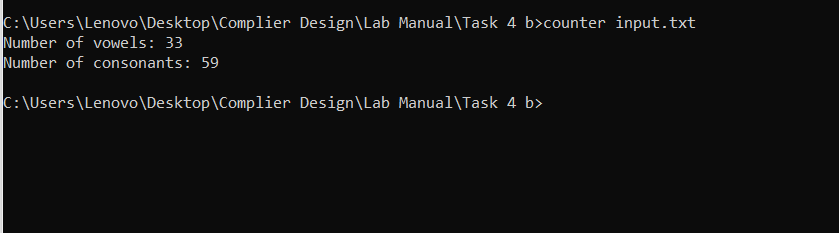
</body>

</html>

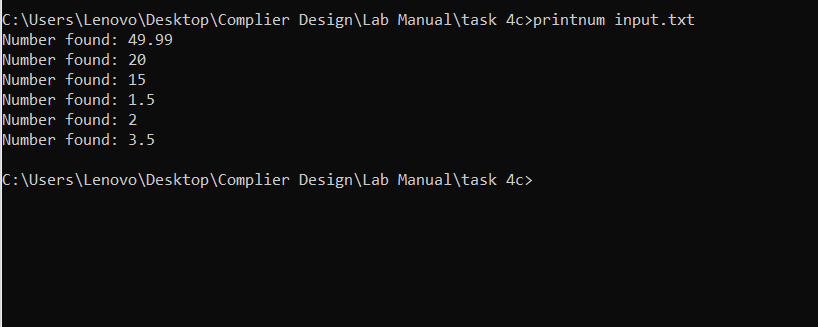
**Output:** a)



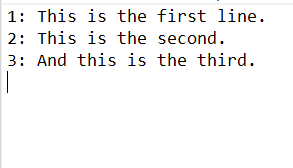
b)



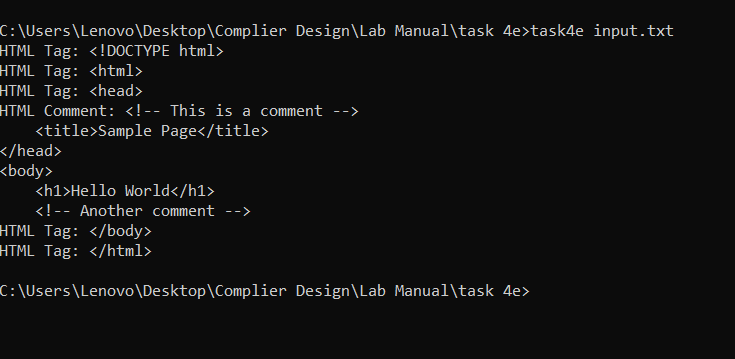
c)



d)



e)



**PRACTICAL 5**

**Task:** 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

**Input:** a)

**remove\_comments.l**

%{

#include <stdio.h>

int comment\_count = 0;

FILE \*outfile;

%}

%%

"//".\* { comment\_count++; /\* Skip single-line comment \*/ }

"/\*"([^\*]|\\*+[^\*/])\*"\*"+"/" { comment\_count++; /\* Skip multi-line comment \*/ }

.|\n { fputc(yytext[0], outfile); }

%%

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

if (argc < 3) {

fprintf(stderr, "Usage: %s <input\_file.c> <output\_file.c>\n", argv[0]);

return 1;

}

FILE \*infile = fopen(argv[1], "r");

if (!infile) {

perror("Error opening input file");

return 1;

}

outfile = fopen(argv[2], "w");

if (!outfile) {

perror("Error opening output file");

fclose(infile);

return 1;

}

yyin = infile;

yylex();

printf("Total number of comments removed: %d\n", comment\_count);

fclose(infile);

fclose(outfile);

return 0;

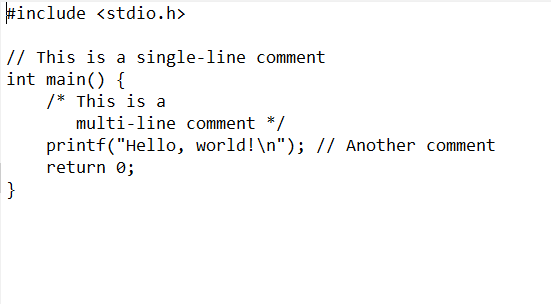
}

int yywrap() {

return 1;

}

**input.c**



b)

**tokenizer\_c.l**

%{

#include <stdio.h>

#include <string.h>

int is\_keyword(const char \*str);

%}

%option noyywrap

%%

"auto"|"break"|"case"|"char"|"const"|"continue"|"default"|"do"|"double"|\

"else"|"enum"|"extern"|"float"|"for"|"goto"|"if"|"inline"|"int"|"long"|\

"register"|"return"|"short"|"signed"|"sizeof"|"static"|"struct"|"switch"|\

"typedef"|"union"|"unsigned"|"void"|"volatile"|"while" { printf("Keyword: %s\n", yytext); }

[ \t\n]+ ;

"=="|"!="|"<="|">="|"="|"+"|"-"|"\*"|"/"|"%"|"&&"|"||"|"!"|"<"|">" { printf("Operator: %s\n", yytext); }

[0-9]+(\.[0-9]+)? { printf("Number: %s\n", yytext); }

\"([^\\\"]|\\.)\*\" { printf("String Literal: %s\n", yytext); }

\'.\' { printf("Character Literal: %s\n", yytext); }

[{}()\[\],;.] { printf("Special Symbol: %s\n", yytext); }

[a-zA-Z\_][a-zA-Z0-9\_]\* {

if (is\_keyword(yytext))

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

else

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

}

. { printf("Unrecognized Character: %s\n", yytext); }

%%

int is\_keyword(const char \*str) {

const char \*keywords[] = {

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

"else", "enum", "extern", "float", "for", "goto", "if", "inline", "int", "long",

"register", "return", "short", "signed", "sizeof", "static", "struct", "switch",

"typedef", "union", "unsigned", "void", "volatile", "while", NULL

};

for (int i = 0; keywords[i] != NULL; i++) {

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

return 1;

}

return 0;

}

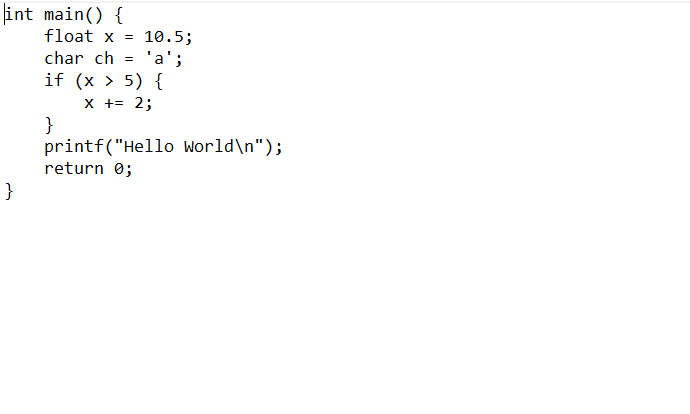
int main() {

yylex();

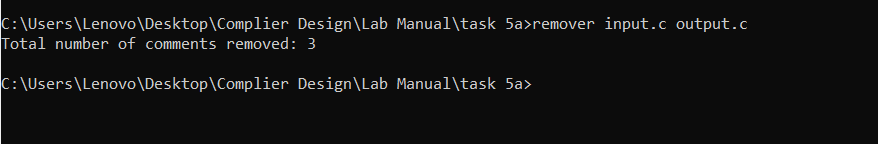
return 0;

}

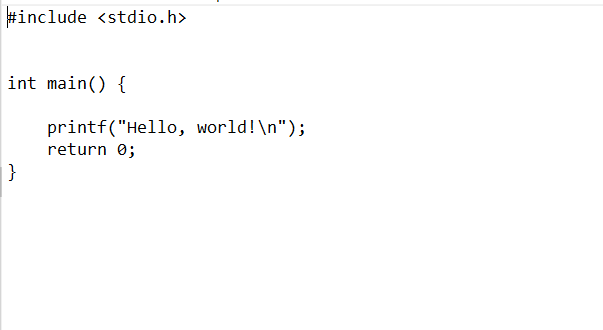
**input.txt**



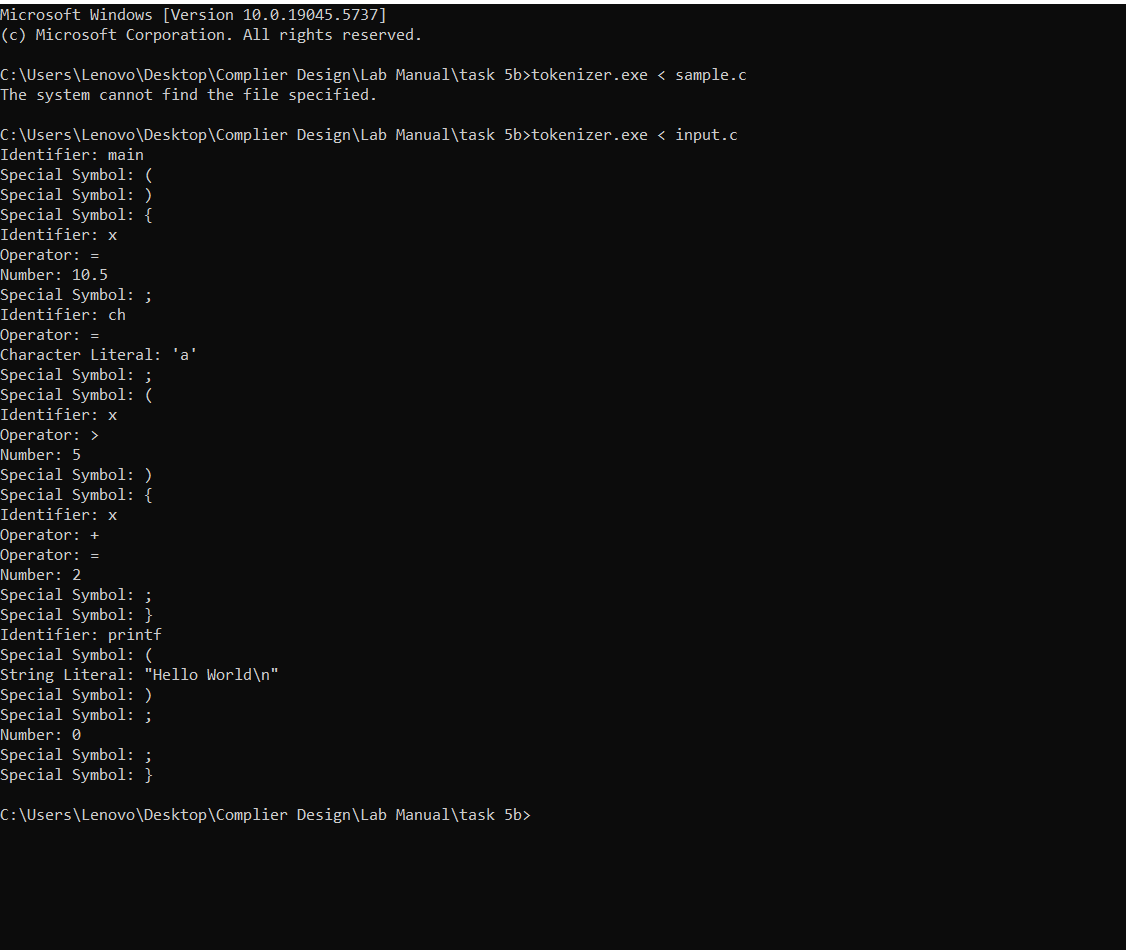
**Output:** a)



**output.c**

****

b)



**PRACTICAL 6**

**Task:** Program to implement Recursive Descent Parsing in C.

**Input:** #include <stdio.h>

#include <stdlib.h>

**// Grammar:**

**// E -> i E'**

**// E' -> + i E' | - i E' | ε**

char s[20];

int i = 1;

char l;

void match(char t) {

if (l == t) {

l = s[i++];

} else {

printf("Syntax error\n");

exit(1);

}

}

void E\_() {

if (l == '+') {

match('+');

match('i');

E\_();

} else if (l == '-') {

match('-');

match('i');

E\_();

} else {

// ε (do nothing)

}

}

void E() {

if (l == 'i') {

match('i');

E\_();

} else {

printf("Syntax error\n");

exit(1);

}

}

int main() {

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

scanf("%s", s);

l = s[0];

E();

if (l == '$') {

printf("Success\n");

} else {

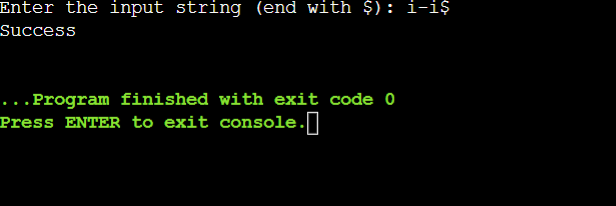
printf("Syntax error\n");

}

return 0;

}

**Output:**



**PRACTICAL 7**

**Task:** 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.

**Input:** a) YACC (Yet Another Compiler Compiler) is a parser generator developed to work with context-free grammars. It is commonly used to design and implement the syntax analysis phase of a compiler. YACC takes a formal description of a language's grammar and produces source code for a parser that recognizes valid strings in that language.

**How YACC Works:** YACC generates a parser in C from a grammar specification written in a .y file. This parser processes tokens, which are typically provided by a lexical analyzer like Lex. The typical workflow includes:

1. Writing a .l file with Lex to tokenize the input.
2. Writing a .y file with YACC to define grammar rules.
3. Compiling both files together to produce an executable parser.

**Structure of a YACC Program:** A YACC program is divided into three main sections:

1. **Definition Section**: Includes token declarations and C header inclusions.
2. **Rules Section**: Contains grammar rules and corresponding C code actions.
3. **User Subroutines Section**: Contains helper functions such as main() and yyerror().

**Key Concepts:**

* **Tokens**: Symbols returned by Lex and recognized by YACC.
* **Grammar Rules**: Production rules that define how tokens are combined.
* **Semantic Actions**: C code attached to grammar rules that executes during parsing.

**Advantages of YACC:**

* Automates parser generation, reducing manual coding.
* Ensures syntax correctness via context-free grammar.
* Integrates smoothly with Lex.

**Applications:** YACC is widely used in building compilers, interpreters, and domain-specific languages. It is also used in expression evaluators, configuration parsers, and query processors.

b)

**expr.l**

%{

#include <stdlib.h>

void yyerror(char \*);

#include "b.tab.h"

%}

%%

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

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

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

[ \t] { }

. yyerror("invalid character");

%%

int yywrap() {

return 0;

}

**b.y**

%{

#include <stdio.h>

int yylex(void);

void yyerror(char \*);

%}

%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;

}

c)

**lex.l**

%{

#include <stdlib.h>

void yyerror(char \*);

#include "yacc.tab.h"

%}

%%

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

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

[ \t] { }

. yyerror("invalid character");

%%

int yywrap() {

return 0;

}

**yacc.y**

%{

#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;

}

d)

**first.l**

%{

#include<stdio.h>

#include "first.tab.h"

void yyerror(char \*);

%}

%%

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

[A-Za-z\_][A-Za-z0-9\_]\* { yylval.str = yytext; return ID; }

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

[ \t] ;

. yyerror("invalid character");

%%

int yywrap(){

return 1;

}

**first.y**

%{

#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);}

%%

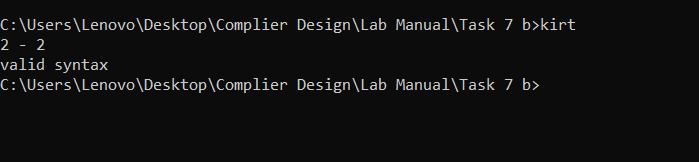
void yyerror(char \*s){

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

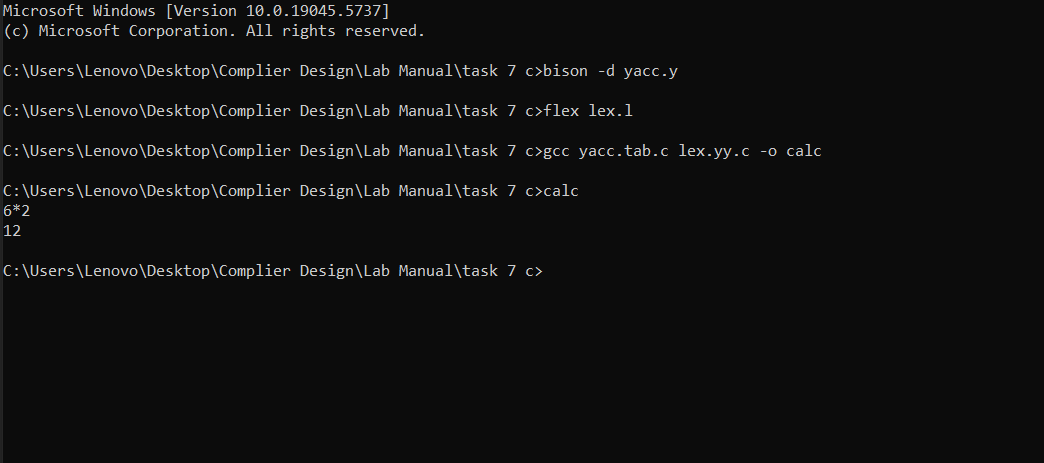
}

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

**Output:** b)



c)



d)

