EXP NO: 1. a **DATE:**

DEVELOP A SIMPLE C PROGRAM TO DEMONSTRATE A BASIC STRING OPERATIONS

AIM:

To write a C program that takes a string input from the user and converts all its characters to uppercase using the toupper() function from the <ctype.h> library.

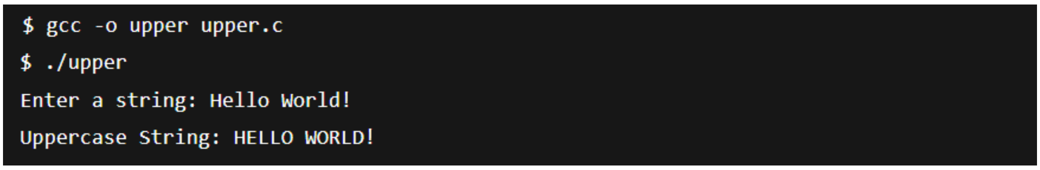
ALGORITHM:

1. **Start**
2. Declare a character array str to store the input string.
3. Prompt the user to enter a string.
4. Use fgets() to read the string input from the user.
5. Check if the last character is a newline (\n) and replace it with \0 (null terminator).
6. Loop through each character of the string:
7. Use toupper() to convert each character to uppercase.
8. Store the converted character back in the string.
9. Print the modified uppercase string.
10. **End**

PROGRAM:

```
#include <stdio.h>
#include <ctype.h> #include
<string.h>  int main() {     char
str[100];     printf("Enter a string:
");     fgets(str, sizeof(str), stdin);
size_t len = strlen(str);     if (len > 0
&& str[len - 1] == '\n') {
        str[len - 1] = '\0';
    }
    for (int i = 0; str[i] != '\0'; i++) {
str[i] = toupper((unsigned char)str[i]);
    }
    printf("Uppercase String: %s\n", str);
return 0;
}
```

OUTPUT:



```
$ gcc -o upper upper.c
$ ./upper
Enter a string: Hello World!
Uppercase String: HELLO WORLD!
```

EXP NO: 1.b **DATE:**

DEVELOP A SIMPLE C PROGRAM TO DEMONSTRATE A BASIC STRING OPERATIONS

AIM:

To write a C program that checks whether a given substring exists within a string without using the strstr() function. If found, print its starting index; otherwise, print "Substring not found."

ALGORITHM:

1. Start
2. Declare two character arrays: one for the main string and one for the substring.
3. Take input for both strings from the user.
4. Compute the lengths of both strings.
5. Loop through the main string and check for a match with the substring:
    - Compare characters one by one.
    - If a match is found, print the starting index and exit.
6. If no match is found, print "Substring not found."
7. End

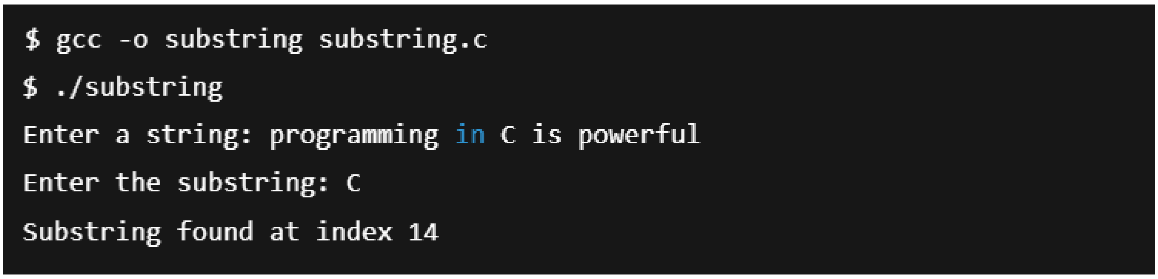
PROGRAM:

```
#include <stdio.h>
#include <string.h>
int findSubstring(char str[], char sub[]) {
int strLen = strlen(str), subLen = strlen(sub);
   for (int i = 0; i <= strLen - subLen; i++) {
int j;
       for (j = 0; j < subLen; j++) {
if (str[i + j] != sub[j]) {
               break;
           }
       }
       if (j == subLen) {
           return i;  // Found at index i
       }
   }     return -1;  // Not
found
} int main() {     char str[100],
sub[50];     printf("Enter a string:
");     fgets(str, sizeof(str), stdin);
printf("Enter the substring: ");
fgets(sub, sizeof(sub), stdin);
str[strcspn(str, "\n")] = '\0';
sub[strcspn(sub, "\n")] = '\0';     int
index = findSubstring(str, sub);
```

```
    if (index != -1)
        printf("Substring found at index %d\n", index);
else
        printf("Substring not found\n");

    return 0;
}
```

OUTPUT



```
$ gcc -o substring substring.c
$ ./substring
Enter a string: programming in C is powerful
Enter the substring: C
Substring found at index 14
```

EXP NO: 1.c **DATE:**

AIM:

To write a C program that compares two strings entered by the user and determines whether they are the same.

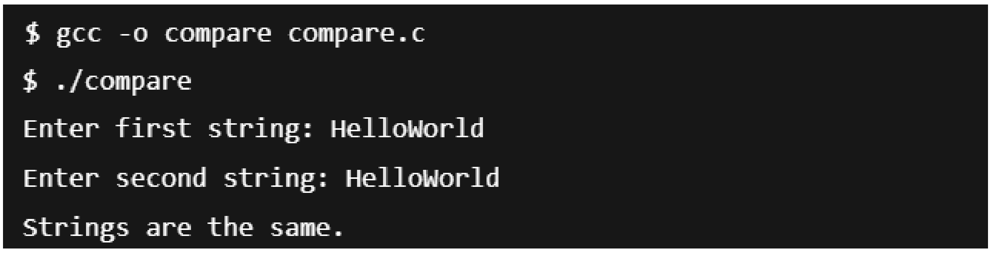
ALGORITHM:

1. Start
2. Declare two character arrays to store the strings.
3. Take input for both strings from the user.
4. Use strcmp() to compare the two strings.
5. If the result is 0, print "Strings are the same."
6. Otherwise, print "Strings are different."
7. End

PROGRAM:

```
#include <stdio.h>
#include <string.h>
int main() {
char str1[100], str2[100];
printf("Enter first string: ");
fgets(str1, sizeof(str1), stdin);
printf("Enter second string: ");
fgets(str2, sizeof(str2), stdin);
str1[strcspn(str1, "\n")] = '\0';
str2[strcspn(str2, "\n")] = '\0';
if (strcmp(str1, str2) == 0)
printf("Strings are the same.\n");
else
printf("Strings are different.\n");
return 0;
}
```

OUTPUT:



```
$ gcc -o compare compare.c
$ ./compare
Enter first string: HelloWorld
Enter second string: HelloWorld
Strings are the same.
```

EXP NO: 1.d **DATE:**

DEVELOP A SIMPLE C PROGRAM TO DEMONSTRATE A BASIC STRING OPERATIONS

AIM:

To write a C program that removes all spaces from a string entered by the user.

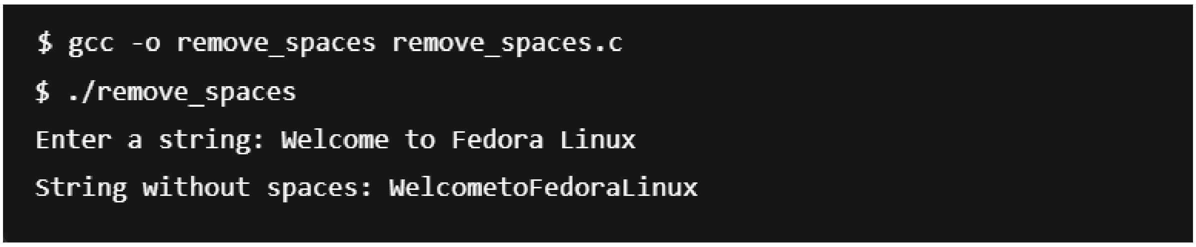
ALGORITHM:

1. Start
2. Declare a character array for input.
3. Take string input from the user.
4. Traverse the string:
    - Copy only non-space characters to a new position in the array.
5. Print the modified string.
6. End

PROGRAM:

```
#include <stdio.h>
void removeSpaces(char str[]) {
int i, j = 0;
for (i = 0; str[i] != '\0'; i++) {
if (str[i] != ' ') {
str[j++] = str[i];
}
}
str[j] = '\0';
}
int main() {
char str[100];
printf("Enter a string: ");
fgets(str, sizeof(str), stdin);
removeSpaces(str);
printf("String without spaces: %s\n", str);
return 0;
}
```

OUTPUT:



```
$ gcc -o remove_spaces remove_spaces.c
$ ./remove_spaces
Enter a string: Welcome to Fedora Linux
String without spaces: WelcometoFedoraLinux
```

EXP NO: 1.e **DATE:**

DEVELOP A SIMPLE C PROGRAM TO DEMONSTRATE A BASIC STRING OPERATIONS

AIM:

To write a C program that calculates the frequency of each character in a given string.

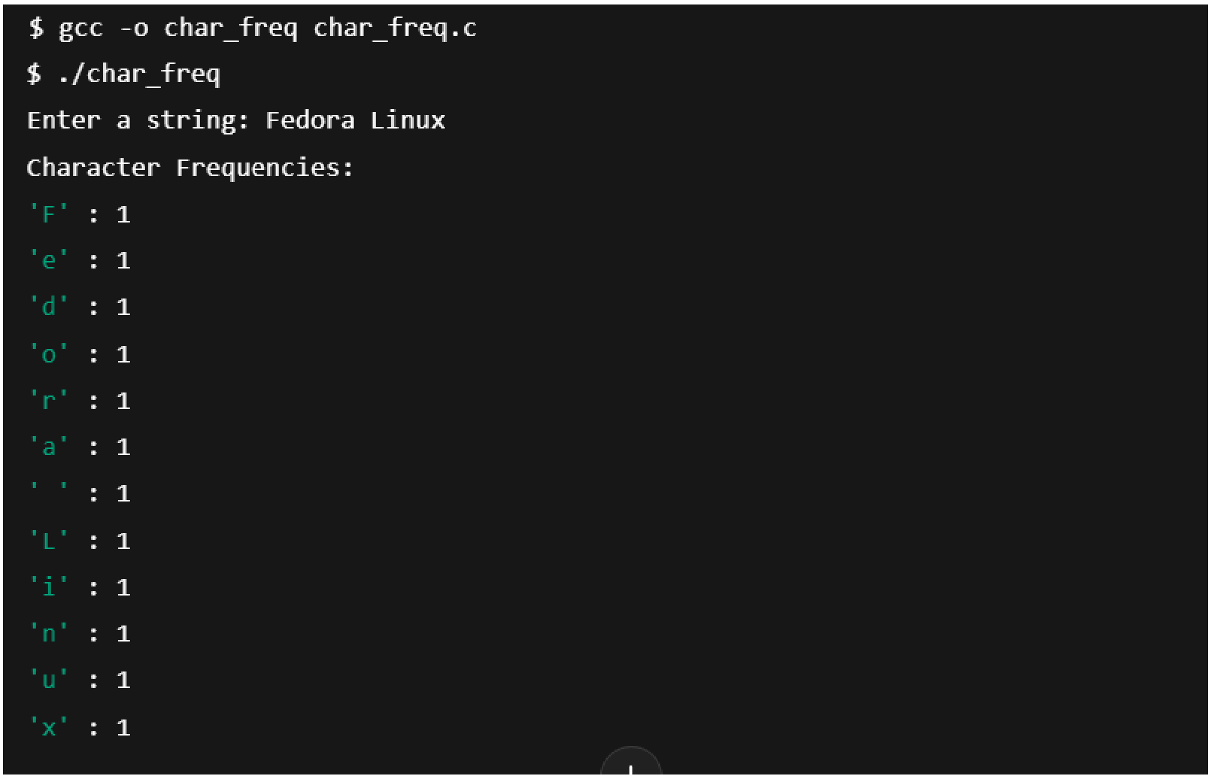
ALGORITHM:

1. Start
2. Declare a character array for input.
3. Declare an integer array freq[256] initialized to 0 (for ASCII character frequencies).
4. Take string input from the user.
5. Traverse the string:
   o Increment the frequency count for each character.
6. Print characters with their respective frequencies.
7. End

PROGRAM:

```
#include <stdio.h>
#include <string.h>
void countFrequency(char str[]) {
int freq[256] = {0};
for (int i = 0; str[i] != '\0'; i++) {
freq[(unsigned char)str[i]]++;
}
printf("Character Frequencies:\n");
for (int i = 0; i < 256; i++) {
if (freq[i] > 0) {
printf("'%c' : %d\n", i, freq[i]);
}
}
}
int main() {
char str[100];
printf("Enter a string: ");
fgets(str, sizeof(str), stdin);
countFrequency(str);
return 0;
}
```

OUTPUT:



```
$ gcc -o char_freq char_freq.c
$ ./char_freq
Enter a string: Fedora Linux
Character Frequencies:
'F' : 1
'e' : 1
'd' : 1
'o' : 1
'r' : 1
'a' : 1
' ' : 1
'L' : 1
'i' : 1
'n' : 1
'u' : 1
'x' : 1
```

EXP NO: 1. f **DATE:**

DEVELOP A SIMPLE C PROGRAM TO DEMONSTRATE A BASIC STRING OPERATIONS

AIM:

To write a C program that concatenates two strings entered by the user.

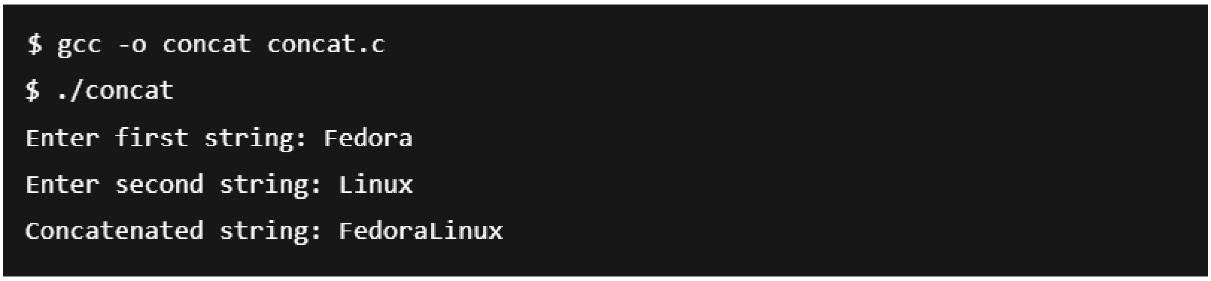
ALGORITHM:

1. Start
2. Declare two character arrays for input.
3. Take input for both strings.
4. Use strcat() to concatenate the second string to the first.
5. Print the concatenated result.
6. End

PROGRAM:

```
#include <stdio.h>
#include <string.h>
int main() {
char str1[100], str2[50];
printf("Enter first string: ");
fgets(str1, sizeof(str1), stdin);
printf("Enter second string: ");
fgets(str2, sizeof(str2), stdin);
str1[strcspn(str1, "\n")] = '\0';
str2[strcspn(str2, "\n")] = '\0';
strcat(str1, str2);
printf("Concatenated string: %s\n", str1);
return 0;
}
```

OUTPUT:



```
$ gcc -o concat concat.c
$ ./concat
Enter first string: Fedora
Enter second string: Linux
Concatenated string: FedoraLinux
```

EXP NO: 1.g **DATE:**

DEVELOP A SIMPLE C PROGRAM TO DEMONSTRATE A BASIC STRING OPERATIONS

AIM:

To write a C program that replaces all occurrences of a specific character in a string with another character.

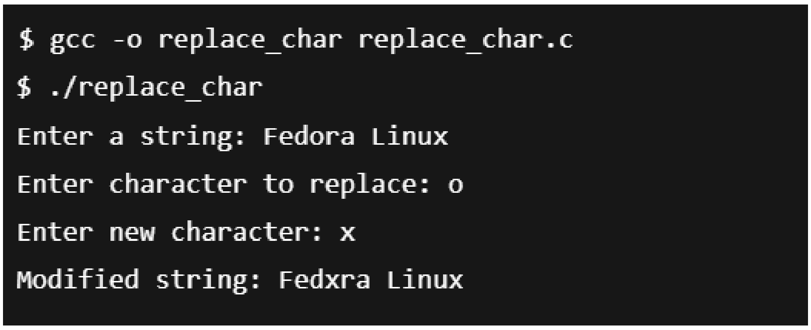
ALGORITHM:

```
1. Start
2. Declare a character array for input.
3. Take string input from the user.
4. Take input for the character to replace and its replacement.
5. Traverse the string:
o Replace occurrences of the old character with the new one.
6. Print the modified string.
7. End
```

PROGRAM:

```
#include <stdio.h>
void replaceChar(char str[], char oldChar, char newChar) {
for (int i = 0; str[i] != '\0'; i++) {
if (str[i] == oldChar) {
str[i] = newChar;
}
}
}
int main() {
char str[100], oldChar, newChar;
printf("Enter a string: ");
fgets(str, sizeof(str), stdin);
printf("Enter character to replace: ");
scanf("%c", &oldChar);
getchar(); // Consume leftover newline character
printf("Enter new character: ");
scanf("%c", &newChar);
replaceChar(str, oldChar, newChar);
printf("Modified string: %s\n", str);
return 0;
}
```

OUTPUT:



```
$ gcc -o replace_char replace_char.c
$ ./replace_char
Enter a string: Fedora Linux
Enter character to replace: o
Enter new character: x
Modified string: Fedxra Linux
```

RESULT:

Thus the above program takes a string input, calculates and displays its length, copies and prints the string, concatenates it with a second input string, and finally compares both strings to check if they are the same or different.

EXP NO: 2 **DATE:**

DEVELOP A C PROGRAM TO ANALYZE A GIVEN C CODE SNIPPET AND RECOGNIZE DIFFERENT TOKENS, INCLUDING KEYWORD, IDENTIFIERS, OPERAT OR AND SPECIAL SYMBOLS

AIM:

To develop a C program that analyzes a given C code snippet and recognizes different tokens, including keywords, identifiers, operators, and special symbols.

ALGORITHM:

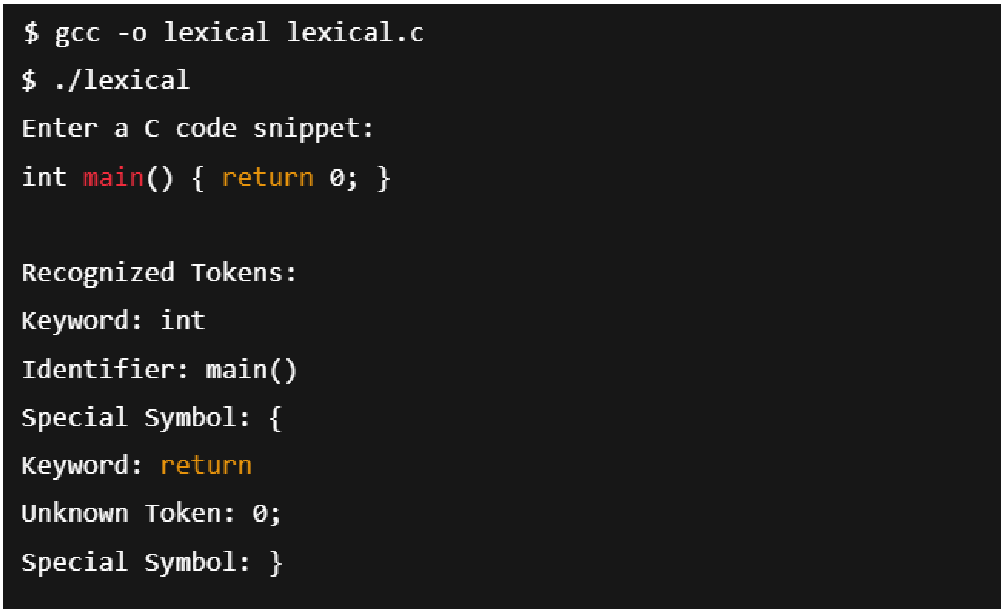
1. Start
2. Take a C code snippet as input from the user or a file.
3. Initialize necessary arrays and variables for keywords, identifiers, operators, and special
4. symbols.
5. Tokenize the input string using spaces, newlines, and other delimiters.
6. For each token:
    - Check if it is a keyword (compare with a predefined list of C keywords).
    - Check if it is an identifier (valid variable/function name that doesn't match a
    - keyword).
    - Check if it is an operator (e.g., +, -, *, /, ==, &&).
    - Check if it is a special symbol (e.g., {, }, (,), ;, ,).
7. Print the categorized tokens.
8. End

PROGRAM:

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
// List of C keywords
const char *keywords[] = {
"int", "float", "char", "double", "if", "else", "for", "while",
"do", "return", "void", "switch", "case", "break", "continue",
"default", "struct", "typedef", "enum", "union", "static",
"extern", "const", "sizeof", "goto", "volatile", "register"
};
const int num_keywords = sizeof(keywords) / sizeof(keywords[0]);
18
// List of C operators
const char *operators[] = {"+", "-", "*", "/", "=", "==", "!=", "<", ">", "<=", ">=", "&&", "||",
"++", "--"};
const int num_operators = sizeof(operators) / sizeof(operators[0]);
// List of special symbols
const char special_symbols[] = {';', '(', ')', '{', '}', '[', ']', ',', '#', '&', '|', ':', '"', '\''};
```

```
// Function to check if a word is a keyword
int isKeyword(char *word) {
for (int i = 0; i < num_keywords; i++) {
if (strcmp(word, keywords[i]) == 0)
return 1;
}
return 0;
}
// Function to check if a character is an operator
int isOperator(char *word) {
for (int i = 0; i < num_operators; i++) {
if (strcmp(word, operators[i]) == 0)
return 1;
}
return 0;
}
// Function to check if a character is a special symbol
int isSpecialSymbol(char ch) {
for (int i = 0; i < sizeof(special_symbols); i++) {
if (ch == special_symbols[i])
return 1;
}
return 0;
}
// Function to classify tokens
19
void analyzeTokens(char *code) {
char *token = strtok(code, " \t\n"); // Tokenizing by spaces, tabs, and newlines
printf("\nRecognized Tokens:\n");
while (token != NULL) {
if (isKeyword(token))
printf("Keyword: %s\n", token);
else if (isOperator(token))
printf("Operator: %s\n", token);
else if (isalpha(token[0]) || token[0] == '_') // Identifiers start with a letter or underscore
printf("Identifier: %s\n", token);
else if (isSpecialSymbol(token[0]))
printf("Special Symbol: %s\n", token);
else
printf("Unknown Token: %s\n", token);
token = strtok(NULL, " \t\n");
}
}
// Main function
int main() {
char code[500];
printf("Enter a C code snippet:\n");
fgets(code, sizeof(code), stdin);
analyzeTokens(code);
return 0;
}
```

OUTPUT:



```
$ gcc -o lexical lexical.c
$ ./lexical
Enter a C code snippet:
int main() { return 0; }

Recognized Tokens:
Keyword: int
Identifier: main()
Special Symbol: {
Keyword: return
Unknown Token: 0;
Special Symbol: }
```

RESULT:

Thus, the above program reads a C code snippet, tokenizes it using space, tab, and newline as delimiters, classifies each token as a keyword, identifier, operator, or special symbol based on predefined lists, and prints the recognized tokens along with their types.

EXP NO: 3 **DATE:**

DEVELOP A LEXICAL ANALYZER TO RECOGNIZE A FEW PATTERNS IN C. (EX. IDENTIFIERS, CONSTANTS, COMMENTS, AND OPERATORS, ETC.) USING LEX TOOL.

AIM:

To develop a Lexical Analyzer using the LEX tool that recognizes different tokens in a given C program snippet, including Identifier, Constants, Comments, Operators, Keywords, Special Symbols.

ALGORITHM:

1. Start
2. Define token patterns in LEX for:
   - Keywords (e.g., int, float, if, else)
   - Identifiers (variable/function names)
   - Constants (integer and floating-point numbers)
   - Operators (+, -, =, ==, !=, *, /)
   - Comments (// single-line, /* multi-line */)
   - Special Symbols ({, }, (,), ;, ,)
3. Read input source code.
4. Match the code tokens using LEX rules.
5. Print each recognized token with its type.
6. End

PROGRAM:

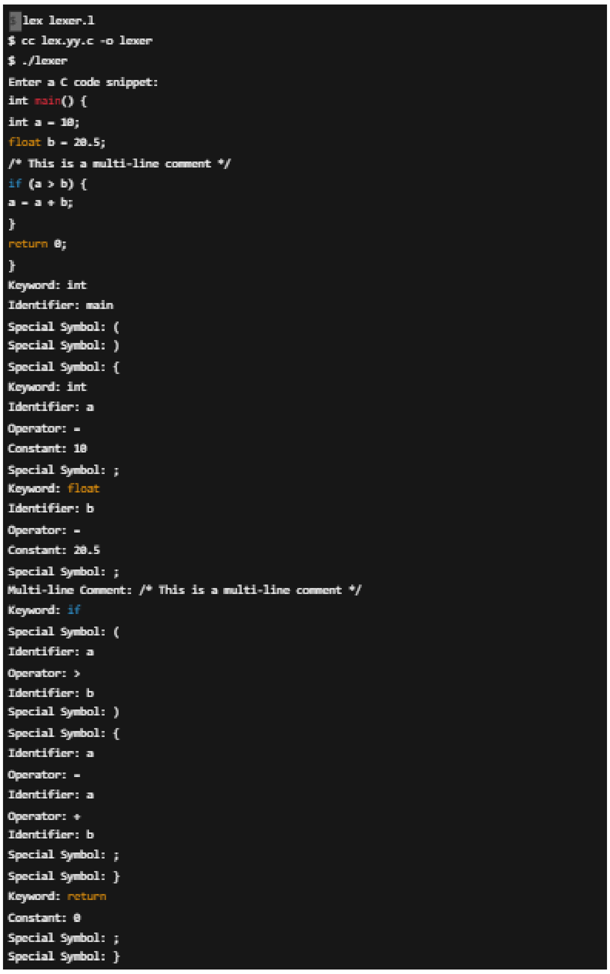
```
%{
#include <stdio.h>
%}
%option noyywrap
%%
// Keywords
"int"|"float"|"char"|"double"|"if"|"else"|"return"|"for"|"while"|"do" {
printf("Keyword: %s\n", yytext);
}
// Identifiers (starting with a letter or underscore, followed by letters, digits, or underscores)
[a-zA-Z_][a-zA-Z0-9_]* {
printf("Identifier: %s\n", yytext);
}
// Constants (integer and floating-point numbers)
[0-9]+(\.[0-9]+)? {
printf("Constant: %s\n", yytext);
}
// Operators
"+"|"-"|"*"|"/"|"="|"=="|"!="|"<"|">"|"&&"|"||"|"++"|"--" {
printf("Operator: %s\n", yytext);
}
// Single-line comments
"//".* {
printf("Comment: %s\n", yytext);
}
// Multi-line comments
"/*"([^*]|\*+[^*/])*\*+"/" {
printf("Multi-line Comment: %s\n", yytext);
}
// Special symbols
";"|","|"("|")"|"{"|"}"|"["|"]" {
printf("Special Symbol: %s\n", yytext);
}
// Ignore whitespaces and newlines
[ \t\n] ;
%%
int main() {
printf("Enter a C code snippet:\n");
yylex();
return 0;
}
```

OUTPUT:

```
lex lexer.l
cc lex.yy.c -o lexer
./a.out
```

Sample Input

```
int main() {
int a = 10;
float b = 20.5;
/* This is a multi-line comment */
if (a > b) {
a = a + b;
}
return 0;
}
```



```
$ lex lexer.l
$ cc lex.yy.c -o lexer
$ ./lexer
Enter a C code snippet:
int main() {
int a = 10;
float b = 20.5;
/* This is a multi-line comment */
if (a > b) {
a = a + b;
}
return 0;
}
Keyword: int
Identifier: main
Special Symbol: (
Special Symbol: )
Special Symbol: {
Keyword: int
Identifier: a
Operator: =
Constant: 10
Special Symbol: ;
Keyword: float
Identifier: b
Operator: =
Constant: 20.5
Special Symbol: ;
Multi-line Comment: /* This is a multi-line comment */
Keyword: if
Special Symbol: (
Identifier: a
Operator: >
Identifier: b
Special Symbol: )
Special Symbol: {
Identifier: a
Operator: =
Identifier: a
Operator: +
Identifier: b
Special Symbol: ;
Special Symbol: }
Keyword: return
Constant: 0
Special Symbol: ;
Special Symbol: }
```

RESULT:

Thus the above program reads a C code snippet, tokenizes it using LEX rules, recognizes and categorizes keywords, identifiers, constants, operators, comments, and special symbols, and then displays each token along with its type.

EXP NO:4 **DATE:**

DESIGN AND IMPLEMENT A DESK CALCULATOR USING THE LEX TOOL

Problem Statement

Recognizes whether a given arithmetic expression is valid, using the operators +, -, *, and /. The program should ensure that the expression follows basic arithmetic syntax rules (e.g., proper placement of operators, operands, and parentheses).

AIM:

To design and implement a Desk Calculator using the LEX tool**,** which validates arithmetic expressions containing **+**, **-**, *****, /, numbers, and parentheses. The program ensures that the expression follows correct arithmetic syntax rules.

ALGORITHM:

1. **Start**
2. Define token patterns in **LEX** for:
   - **Numbers** (integer and floating-point)
   - **Operators** (+, -, *, /)
   - **Parentheses** ((,))
   - **Whitespace (to ignore spaces and tabs)**
3. Read an arithmetic expression as input.
4. Use **LEX rules** to identify and validate tokens.
5. If an **invalid token** is encountered, print an error message.
6. If the expression is valid, print "Valid arithmetic expression."
7. **End**

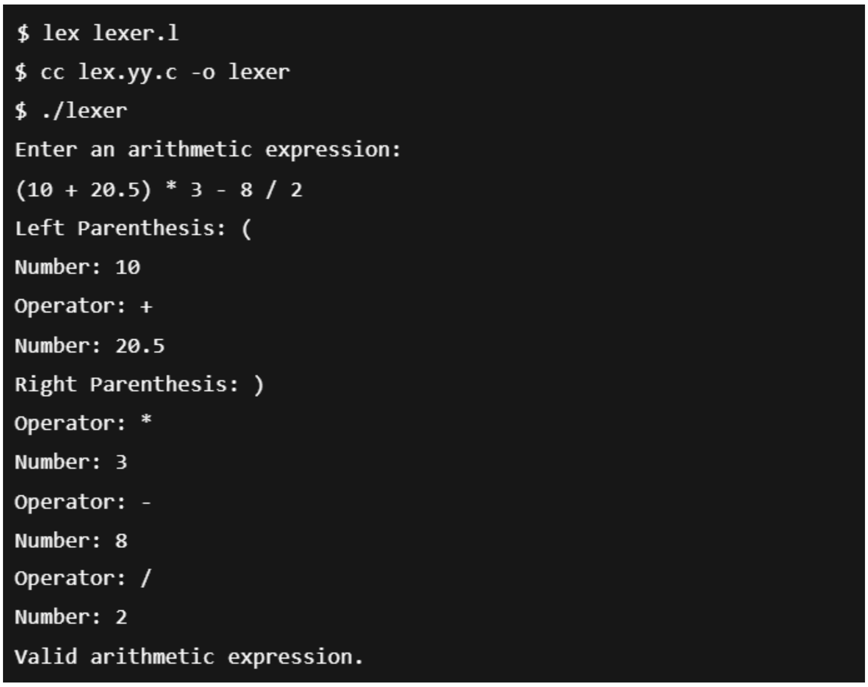
PROGRAM:

```
%{
#include <stdio.h>
int isValid = 1; // Flag to track if the expression is valid
%}
%option noyywrap
%%
// Numbers (integer and floating-point)
[0-9]+(\.[0-9]+)? {
printf("Number: %s\n", yytext);
}
// Operators
"+"|"-"|"*"|"/" {
printf("Operator: %s\n", yytext);
}
// Parentheses
"(" { printf("Left Parenthesis: %s\n", yytext); }
")" { printf("Right Parenthesis: %s\n", yytext); }
29
// Ignore spaces and tabs
```

```
[ \t]+ ;
// Invalid tokens
. {
printf("Error: Invalid token '%s'\n", yytext);
isValid = 0;
}
%%
int main() {
printf("Enter an arithmetic expression:\n");
yylex();
if (isValid)
printf("Valid arithmetic expression.\n");
else
printf("Invalid arithmetic expression.\n");
return 0;
}
```

OUTPUT :

```
lex calculator.l
cc lex.yy.c -o calculator
./a.out
```



```
$ lex lexer.l
$ cc lex.yy.c -o lexer
$ ./lexer
Enter an arithmetic expression:
(10 + 20.5) * 3 - 8 / 2
Left Parenthesis: (
Number: 10
Operator: +
Number: 20.5
Right Parenthesis: )
Operator: *
Number: 3
Operator: -
Number: 8
Operator: /
Number: 2
Valid arithmetic expression.
```

RESULT:

Thus the above program reads an arithmetic expression, tokenizes it using **LEX rules**, and validates the syntax by recognizing **numbers, operators (**+, -, *, /**), and parentheses**. If the expression is **valid**, it prints "Valid arithmetic expression." Otherwise, it detects and reports **invalid tokens.**

EXP NO: 5 **DATE:**

RECOGNIZE A VALID VARIABLE WHICH STARTS WITH A LETTER FOLLOWED BY ANY NUMBER OF LETTERS OR DIGITS USING LEX AND YACC

Problem Statement:

Recognizes a valid variable name. The variable name must start with a letter (either uppercase or lowercase) and can be followed by any number of letters or digits. The program should validate whether a given string adheres to this naming convention.

AIM:

To develop a **LEX and YACC program** that recognizes a **valid variable name** in C programming, which:

 Starts with a **letter** (a-z or A-Z)

 Followed by **any number of letters or digits** (a-z, A-Z, 0-9, _)

 Does not allow invalid characters (e.g., 123abc, @var, x!y)

ALGORITHM:

1. A Yacc source program has three parts as follows: Declarations %% translation rules %% supporting C routines
2. Declarations Section: This section contains entries that:
    - Include standard I/O header file.
    - Define global variables.
    - Define the list rule as the place to start processing.
    - Define the tokens used by the parser.
3. Rules Section: The rules section defines the rules that parse the input stream. Each rule of a grammar production and the associated semantic action.
4. Programs Section: The programs section contains the following subroutines. Because these subroutines are included in this file, it is not necessary to use the yacc library when processing this file.
    - Main- The required main program that calls the yyparse subroutine to start the program.
    - yyerror(s) -This error-handling subroutine only prints a syntax error message.
    - yywrap -The wrap-up subroutine that returns a value of 1 when the end of input occurs. The
    - calc.lex file contains include statements for standard input and output, as programmer file
    - information if we use the -d flag with the yacc command. The y.tab.h file contains definitions
    - for
    - the tokens that the parser program uses.
5. calc.lex contains the rules to generate these tokens from the input stream.

PROGRAM:

LEX PROGRAM

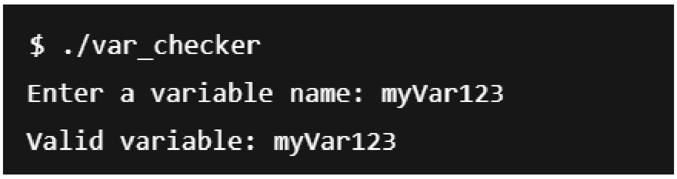
```
%{
#include "y.tab.h"
%}
%option noyywrap
%%
// Pattern for valid variable names
[a-zA-Z][a-zA-Z0-9]* { return IDENTIFIER; }
// Ignore whitespace
[ \t\n] { /* Skip */ }
. { return yytext[0]; }
%%
```

YACC PROGRAM

```
%{
#include <stdio.h>
void yyerror(const char *msg);
%}
%token IDENTIFIER
%%
stmt: IDENTIFIER { printf("Valid variable: %s\n", yytext); }
;
%%
void yyerror(const char *msg) {
printf("Invalid variable\n");
}
35
int main() {
printf("Enter a variable name: ");
yyparse();
return 0;
}
```

OUTPUT :

```
yacc -d parser.y
lex lexer.l
cc lex.yy.c y.tab.c -o var_checker
./a.out
```



```
$ ./var_checker
Enter a variable name: myVar123
Valid variable: myVar123
```

RESULT: Thus the above program reads an input string, checks whether it follows the rules for a valid variable name, and produces the following output.

EXP NO: 6 **DATE:**

EVALUATE THE EXPRESSION THAT TAKES DIGITS, *, + USING LEX AND YACC

AIM:

To design and implement a **LEX and YACC program** that evaluates arithmetic expressions containing **digits, +, and *** while following operator precedence rules.

ALGORITHM:

1. Using the flex tool, create lex and yacc files.
2. In the definition section of the lex file, declare the required header files along with an
3. external integer variable yylval.
4. In the rule section, if the regex pertains to digit convert it into integer and store yylval.
5. Return the number.
6. In the user definition section, define the function yywrap()
7. In the definition section of the yacc file, declare the required header files along with
8. the flag variables set to zero. Then define a token as number along with left as '+' , '-'
9. , 'or' , '*' , '/' , '%' or '(' ')'
10. In the rules section, create an arithmetic expression as E. Print the result and return
11. zero.
12. Define the following:
    - E: E '+' E (add)
    - E: E '-' E (sub)
    - E: E '*' E (mul)
    - E: E '/' E (div)
13. If it is a single number return the number.
14. In driver code, get the input through yyparse(); which is also called as main function.
15. Declare yyerror() to handle invalid expressions and exceptions.
16. Build lex and yacc files and compile.

PROGRAM:

LEX CODE :

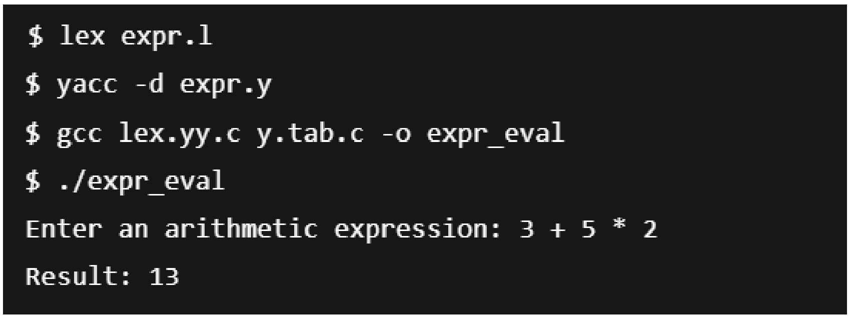
```
expr.l
%{
#include "y.tab.h"
%}
%%
[0-9]+ {
yylval = atoi(yytext);
return NUMBER;
}
[+\n] return yytext[0];
[*] return yytext[0];
[ \t] ; /* Ignore whitespace */
. yyerror("Invalid character");
```

```
%%
```

YACC Program :

```
expr.y
%{
#include <stdio.h>
#include <stdlib.h>
int yylex();
void yyerror(const char *s);
%}
%token NUMBER
%left '+' /* Lower precedence */
%left '*' /* Higher precedence */
%%
expression:
expression '+' expression { $$ = $1 + $3; }
| expression '*' expression { $$ = $1 * $3; }
| NUMBER { $$ = $1; }
;
%%
int main() {
printf("Enter an arithmetic expression:\n");
yyparse();
return 0;
}
void yyerror(const char *s) {
fprintf(stderr, "Error: %s\n", s);
}
```

OUTPUT :



```
$ lex expr.l
$ yacc -d expr.y
$ gcc lex.yy.c y.tab.c -o expr_eval
$ ./expr_eval
Enter an arithmetic expression: 3 + 5 * 2
Result: 13
```

RESULT:

Thus the above program to evaluate the expression that takes digits, *, + using lex and yacc is been implemented and executed successfully based on the precedence.