Compiler design Lab assessment 2
Name: Karan Sehgal
Registration no.: 22BCE3939

Q1:Write a C program to implement a symbol table.

Implementing a symbol table data structure in C to store and manage identifiers (variables, functions, etc.) used in a program, including their associated attributes (type, scope, memory location, etc.).

CODE:-

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

#define MAX_SYMBOLS 100
#define MAX_NAME_LENGTH 50
#define MAX_INPUT_LENGTH 1000

typedef struct {
char name[MAX_NAME_LENGTH];
char type[20];
void* address;
size_t size;
} Symbol;

Symbol symbolTable[MAX_SYMBOLS];
int symbolCount = 0;

void insertSymbol(const char* name, const char* type, size_t size) {
if (symbolCount >= MAX_SYMBOLS) {
printf("Symbol table is full.\n");
return;
}

// Check if symbol already exists
for (int i = 0; i < symbolCount; i++) {
if (strcmp(symbolTable[i].name, name) == 0) {
return; // Symbol already exists, don't insert again
}
}

strcpy(symbolTable[symbolCount].name, name);
strcpy(symbolTable[symbolCount].type, type);
symbolTable[symbolCount].size = size;
symbolTable[symbolCount].address = malloc(size);
```

```
if (symbolTable[symbolCount].address == NULL) {
printf("Memory allocation failed for symbol %s.\n", name);
return;
}

symbolCount++;
}

void displaySymbolTable() {
printf("\nSymbol Table:\n");
printf("%-20s %-10s %-15s %s\n", "Name", "Type", "Address", "Size");
printf("-------------------------------------------------\n");
for (int i = 0; i < symbolCount; i++) {
printf("%-20s %-10s %-15p %zu\n",
symbolTable[i].name,
symbolTable[i].type,
symbolTable[i].address,
symbolTable[i].size);
}
}

int isOperator(const char* str) {
return (strlen(str) == 1 && strchr("+-*/", str[0])) ||
(strcmp(str, "AND") == 0) ||
(strcmp(str, "OR") == 0) ||
(strcmp(str, "NOT") == 0);
}

void tokenizeAndInsert(const char* input) {
char token[MAX_NAME_LENGTH] = "";
int tokenIndex = 0;

for (int i = 0; input[i] != '\0'; i++) {
if (islower(input[i])) {
// Symbol (single lowercase letter)
token[0] = input[i];
token[1] = '\0';
insertSymbol(token, "int", sizeof(int));
} else if (strchr("+-*/", input[i])) {
// Operator (single character)
token[0] = input[i];
token[1] = '\0';
insertSymbol(token, "operator", sizeof(char));
} else if (isupper(input[i])) {
// Start of a potential operator (AND, NOT, OR)
tokenIndex = 0;
while (isupper(input[i]) && tokenIndex < MAX_NAME_LENGTH - 1) {
token[tokenIndex++] = input[i++];
}
```

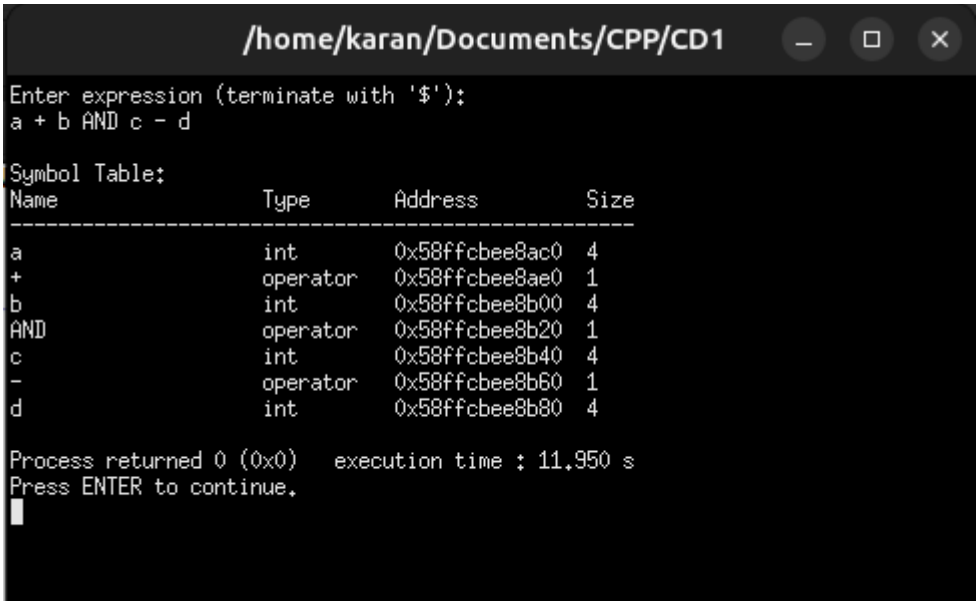
```
token[tokenIndex] = '\0';
i--; // Move back one character as the loop will increment

if (isOperator(token)) {
insertSymbol(token, "operator", sizeof(char));
}
}
// Ignore other characters (like spaces)
}
}

int main() {
char input[MAX_INPUT_LENGTH];

printf("Enter expression (terminate with '$'):\n");
fgets(input, MAX_INPUT_LENGTH, stdin);
input[strcspn(input, "\n")] = 0; // Remove newline

if (strcmp(input, "$") == 0) {
printf("No input provided.\n");
return 0;
}

tokenizeAndInsert(input);

displaySymbolTable();

// Clean up allocated memory
for (int i = 0; i < symbolCount; i++) {
free(symbolTable[i].address);
}

return 0;
}
```

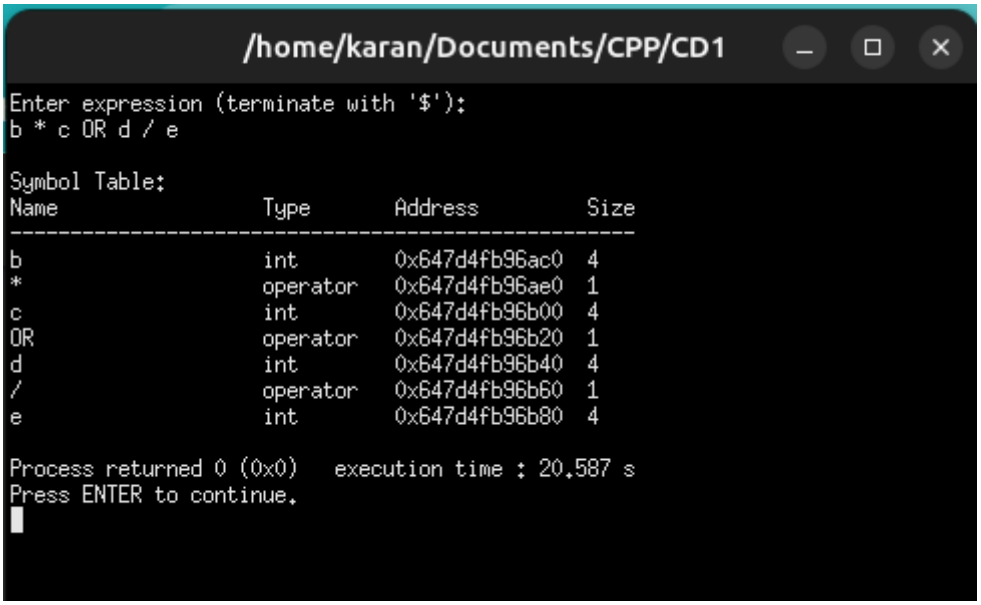
```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

#define MAX_SYMBOLS 100
#define MAX_NAME_LENGTH 50
#define MAX_INPUT_LENGTH 1000

typedef struct {
    char name[MAX_NAME_LENGTH];
    char type[20];
    void* address;
    size_t size;
} Symbol;

Symbol symbolTable[MAX_SYMBOLS];
int symbolCount = 0;

void insertSymbol(const char* name, const char* type, size_t size) {
    if (symbolCount >= MAX_SYMBOLS) {
        printf("Symbol table is full.\n");
        return;
    }

    // Check if symbol already exists
    for (int i = 0; i < symbolCount; i++) {
        if (strcmp(symbolTable[i].name, name) == 0) {
            return; // Symbol already exists, don't insert again
        }
    }

    strcpy(symbolTable[symbolCount].name, name);
    strcpy(symbolTable[symbolCount].type, type);
    symbolTable[symbolCount].size = size;
    symbolTable[symbolCount].address = malloc(size);

    if (symbolTable[symbolCount].address == NULL) {
        printf("Memory allocation failed for symbol %s.\n", name);
        return;
    }

    symbolCount++;
}

void displaySymbolTable() {
    printf("\nSymbol Table:\n");
    printf("%-20s %-10s %-15s %s\n", "Name", "Type", "Address", "Size");
    printf("-------------------------------------------------\n");
    for (int i = 0; i < symbolCount; i++) {
        printf("%-20s %-10s %-15p %zu\n",
               symbolTable[i].name,
               symbolTable[i].type,
               symbolTable[i].address,
```

```
46  void displaySymbolTable() {
50      for (int i = 0; i < symbolCount; i++) {
51          printf("%-20s %-10s %-15p %zu\n",
53                 symbolTable[i].type,
54                 symbolTable[i].address,
55                 symbolTable[i].size);
56      }
57  }
58
59  int isOperator(const char* str) {
60      return (strlen(str) == 1 && strchr("+-*/", str[0])) ||
61             (strcmp(str, "AND") == 0) ||
62             (strcmp(str, "OR") == 0) ||
63             (strcmp(str, "NOT") == 0);
64  }
65
66  void tokenizeAndInsert(const char* input) {
67      char token[MAX_NAME_LENGTH] = "";
68      int tokenIndex = 0;
69
70      for (int i = 0; input[i] != '\0'; i++) {
71          if (islower(input[i])) {
72              // Symbol (single lowercase letter)
73              token[0] = input[i];
74              token[1] = '\0';
75              insertSymbol(token, "int", sizeof(int));
76          } else if (strchr("+-*/", input[i])) {
77              // Operator (single character)
78              token[0] = input[i];
79              token[1] = '\0';
80              insertSymbol(token, "operator", sizeof(char));
81          } else if (isupper(input[i])) {
82              // Start of a potential operator (AND, NOT, OR)
83              tokenIndex = 0;
84              while (isupper(input[i]) && tokenIndex < MAX_NAME_LENGTH - 1) {
85                  token[tokenIndex++] = input[i++];
86              }
87              token[tokenIndex] = '\0';
88              i--; // Move back one character as the loop will increment
89
90              if (isOperator(token)) {
91                  insertSymbol(token, "operator", sizeof(char));
92              }
93          }
94          // Ignore other characters (like spaces)
95      }
96  }
97
98  int main() {
99      char input[MAX_INPUT_LENGTH];
100
101     printf("Enter expression (terminate with '$'):\n");
102     fgets(input, MAX_INPUT_LENGTH, stdin);
103     input[strcspn(input, "\n")] = 0; // Remove newline
```

```c
int main() {
    char input[MAX_INPUT_LENGTH];

    printf("Enter expression (terminate with '$'):\n");
    fgets(input, MAX_INPUT_LENGTH, stdin);
    input[strcspn(input, "\n")] = 0; // Remove newline

    if (strcmp(input, "$") == 0) {
        printf("No input provided.\n");
        return 0;
    }

    tokenizeAndInsert(input);

    displaySymbolTable();

    // Clean up allocated memory
    for (int i = 0; i < symbolCount; i++) {
        free(symbolTable[i].address);
    }

    return 0;
}
```

OUTPUT:


/home/karan/Documents/CPP/CD1
Enter expression (terminate with '$'):
a + b AND c - d
Symbol Table:
Name Type Address Size
a int 0x58ffcbee8ac0 4
+ operator 0x58ffcbee8ae0 1
b int 0x58ffcbee8b00 4
AND operator 0x58ffcbee8b20 1
c int 0x58ffcbee8b40 4
- operator 0x58ffcbee8b60 1
d int 0x58ffcbee8b80 4
Process returned 0 (0x0) execution time : 11.950 s
Press ENTER to continue.


/home/karan/Documents/CPP/CD1
Enter expression (terminate with '$'):
b * c OR d / e
Symbol Table:
Name Type Address Size
b int 0x647d4fb96ac0 4
* operator 0x647d4fb96ae0 1
c int 0x647d4fb96b00 4
OR operator 0x647d4fb96b20 1
d int 0x647d4fb96b40 4
/ operator 0x647d4fb96b60 1
e int 0x647d4fb96b80 4
Process returned 0 (0x0) execution time : 20.587 s
Press ENTER to continue.

Q2: Write a C program to develop a lexical analyser to recognize a few patterns in C.

Develop a C program to implement a lexical analyzer that recognizes various token types in a C-like language, including keywords, identifiers, numbers, operators, and punctuation marks.

Key functionalities:

- Read input code character by character.
- Identify and classify tokens based on defined patterns.
- Print recognized tokens and their types.
- Handle potential errors like unrecognized characters.

CODE:

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>

typedef struct
{
char category[50];
char lexeme[100];
} Token;
Token getNextToken(char **input)
{
Token token;
int ch = **input;
while (isspace(ch))
{
(*input)++;
ch = **input;
}
if (ch == '\0')
{
strcpy(token.category, "END_OF_INPUT");
strcpy(token.lexeme, "");
return token;
}
if (isalpha(ch))
{
int i = 0;
while (isalnum(ch) || ch == '_')
{
token.lexeme[i++] = ch;
(*input)++;
ch = **input;
}
```

```
token.lexeme[i] = '\0';
strcpy(token.category, "identifier");
}
else if (isdigit(ch))
{
int i = 0;
while (isdigit(ch))
{
token.lexeme[i++] = ch;
(*input)++;
ch = **input;
}
token.lexeme[i] = '\0';
strcpy(token.category, "integer literal");
}
else if (ch == '+')
{
token.lexeme[0] = ch;
token.lexeme[1] = '\0';
strcpy(token.category, "addition operator");
(*input)++;
}
else if (ch == '-')
{
token.lexeme[0] = ch;
token.lexeme[1] = '\0';
strcpy(token.category, "subtraction operator");
(*input)++;
}
else if (ch == '*')
{
token.lexeme[0] = ch;
token.lexeme[1] = '\0';
strcpy(token.category, "multiplication operator");
(*input)++;
}
else if (ch == '/')
{
token.lexeme[0] = ch;
token.lexeme[1] = '\0';
strcpy(token.category, "division operator");
(*input)++;
}
else if (ch == '=')
{
token.lexeme[0] = ch;
token.lexeme[1] = '\0';
strcpy(token.category, "assignment operator");
(*input)++;
```

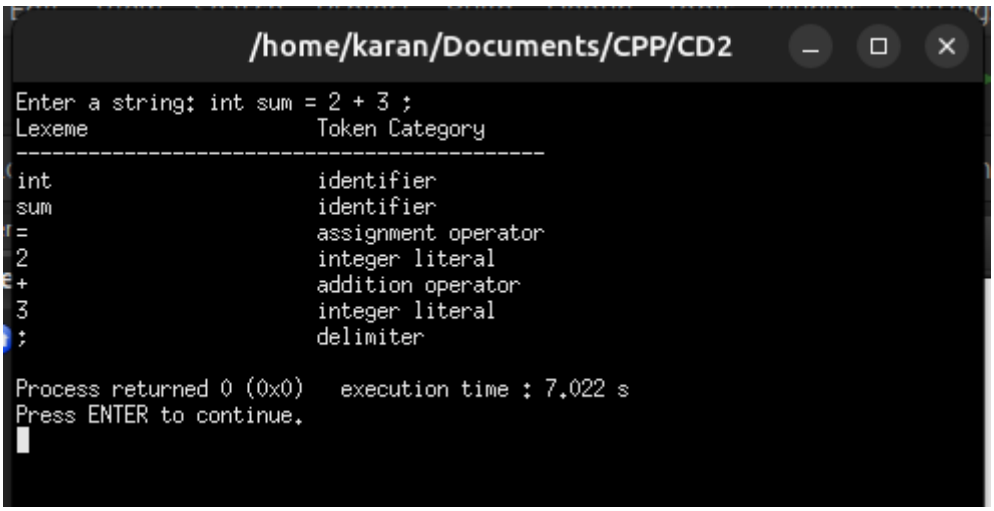
```
}
else if (ch == '(' || ch == ')' || ch == '{' || ch == '}' || ch == ';' || ch == ',')
{
token.lexeme[0] = ch;
token.lexeme[1] = '\0';
strcpy(token.category, "delimiter");
(*input)++;
}
else
{
token.lexeme[0] = ch;
token.lexeme[1] = '\0';
strcpy(token.category, "unknown");
(*input)++;
}
return token;
}
int main()
{
char input[100];
printf("Enter a string: ");
if (fgets(input, sizeof(input), stdin) != NULL)
{
size_t len = strlen(input);
if (len > 0 && input[len - 1] == '\n')
{
input[len - 1] = '\0';
}
}
char *ptr = input;
Token token;
printf("Lexeme \t\t\t Token Category \n");
printf("----------------------------------------\n");
do
{
token = getNextToken(&ptr);
if (strcmp(token.category, "END_OF_INPUT") != 0)
{
printf("%-10s \t\t %s\n", token.lexeme, token.category);
}
} while (strcmp(token.category, "END_OF_INPUT") != 0);

return 0;
}
```

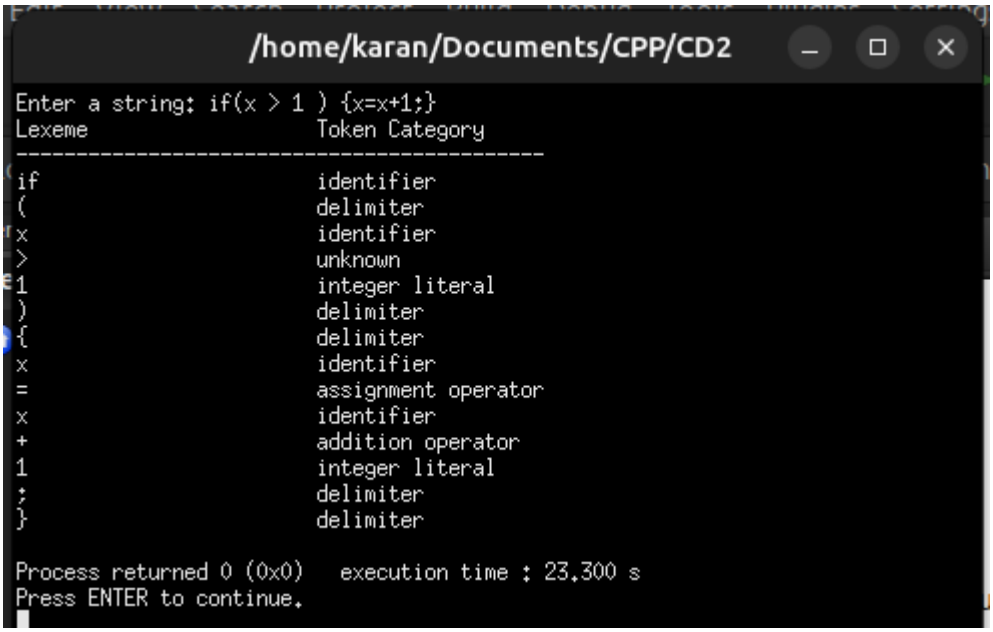
```c
#include <stdio.h>
#include <ctype.h>
#include <string.h>

typedef struct
{
    char category[50];
    char lexeme[100];
} Token;
Token getNextToken(char **input)
{
    Token token;
    int ch = **input;
    while (isspace(ch))
    {
        (*input)++;
        ch = **input;
    }
    if (ch == '\0')
    {
        strcpy(token.category, "END_OF_INPUT");
        strcpy(token.lexeme, "");
        return token;
    }
    if (isalpha(ch))
    {
        int i = 0;
        while (isalnum(ch) || ch == '_')
        {
            token.lexeme[i++] = ch;
            (*input)++;
            ch = **input;
        }
        token.lexeme[i] = '\0';
        strcpy(token.category, "identifier");
    }
    else if (isdigit(ch))
    {
        int i = 0;
        while (isdigit(ch))
        {
            token.lexeme[i++] = ch;
            (*input)++;
            ch = **input;
        }
        token.lexeme[i] = '\0';
        strcpy(token.category, "integer literal");
    }
    else if (ch == '+')
    {
        token.lexeme[0] = ch;
        token.lexeme[1] = '\0';
```

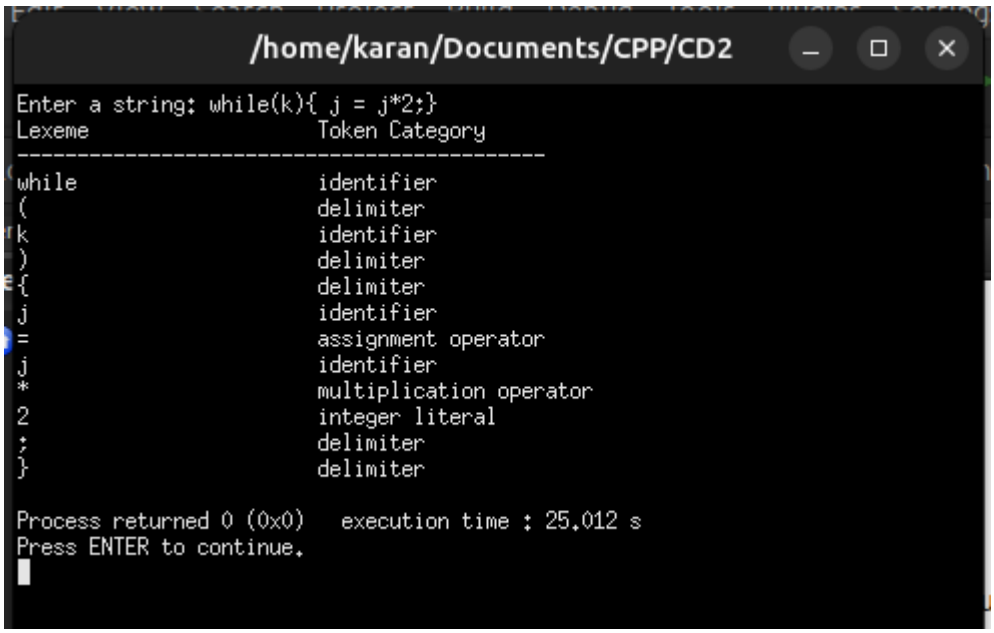
```
{
    else if (ch == '+')
    {
        token.lexeme[0] = ch;
        token.lexeme[1] = '\0';
        strcpy(token.category, "addition operator");
        (*input)++;
    }
    else if (ch == '-')
    {
        token.lexeme[0] = ch;
        token.lexeme[1] = '\0';
        strcpy(token.category, "subtraction operator");
        (*input)++;
    }
    else if (ch == '*')
    {
        token.lexeme[0] = ch;
        token.lexeme[1] = '\0';
        strcpy(token.category, "multiplication operator");
        (*input)++;
    }
    else if (ch == '/')
    {
        token.lexeme[0] = ch;
        token.lexeme[1] = '\0';
        strcpy(token.category, "division operator");
        (*input)++;
    }
    else if (ch == '=')
    {
        token.lexeme[0] = ch;
        token.lexeme[1] = '\0';
        strcpy(token.category, "assignment operator");
        (*input)++;
    }
    else if (ch == '(' || ch == ')' || ch == '{' || ch == '}' || ch == ';' || ch == ',')
    {
        token.lexeme[0] = ch;
        token.lexeme[1] = '\0';
        strcpy(token.category, "delimiter");
        (*input)++;
    }
    else
    {
        token.lexeme[0] = ch;
        token.lexeme[1] = '\0';
        strcpy(token.category, "unknown");
        (*input)++;
    }
    return token;
}
```

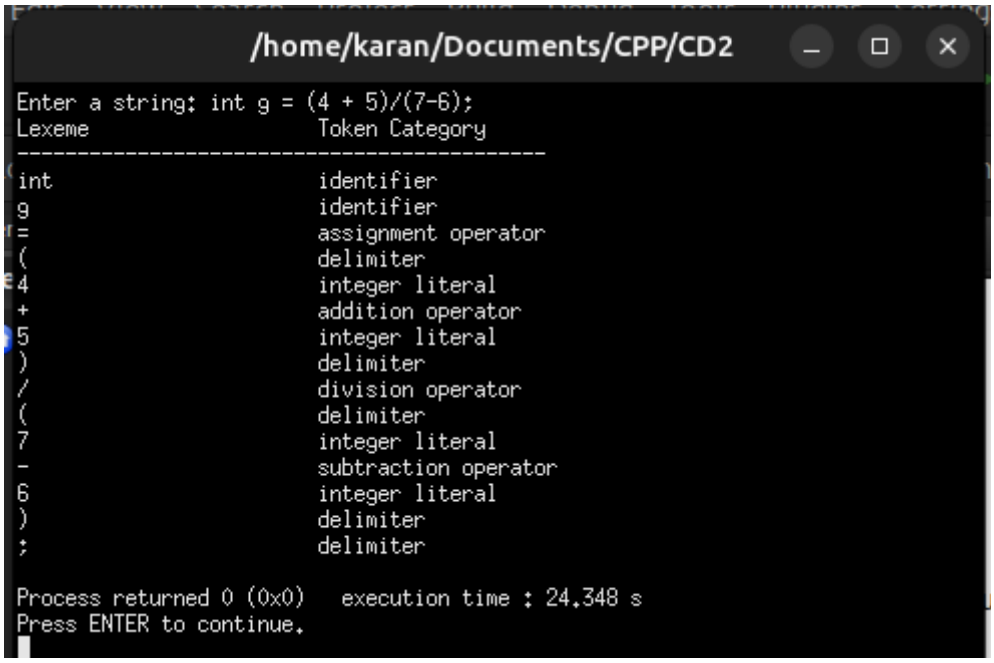
```c
int main()
{
    char input[100];
    printf("Enter a string: ");
    if (fgets(input, sizeof(input), stdin) != NULL)
    {
        size_t len = strlen(input);
        if (len > 0 && input[len - 1] == '\n')
        {
            input[len - 1] = '\0';
        }
    }
    char *ptr = input;
    Token token;
    printf("Lexeme \t\t\t Token Category \n");
    printf("-----------------------------------------\n");
    do
    {
        token = getNextToken(&ptr);
        if (strcmp(token.category, "END_OF_INPUT") != 0)
        {
            printf("%-10s \t\t %s\n", token.lexeme, token.category);
        }
    } while (strcmp(token.category, "END_OF_INPUT") != 0);

    return 0;
}
```

OUTPUT:


/home/karan/Documents/CPP/CD2
Enter a string: int sum = 2 + 3 ;
Lexeme Token Category
int identifier
sum identifier
= assignment operator
2 integer literal
+ addition operator
3 integer literal
; delimiter
Process returned 0 (0x0) execution time : 7.022 s
Press ENTER to continue.


/home/karan/Documents/CPP/CD2
Enter a string: if(x > 1) {x=x+1;}
Lexeme Token Category
if identifier
(delimiter
x identifier
> unknown
1 integer literal
) delimiter
{ delimiter
x identifier
= assignment operator
x identifier
+ addition operator
1 integer literal
; delimiter
} delimiter
Process returned 0 (0x0) execution time : 23.300 s
Press ENTER to continue.


/home/karan/Documents/CPP/CD2
Enter a string: while(k){ j = j*2;}
Lexeme Token Category
while identifier
(delimiter
k identifier
) delimiter
{ delimiter
j identifier
= assignment operator
j identifier
* multiplication operator
2 integer literal
; delimiter
} delimiter
Process returned 0 (0x0) execution time : 25.012 s
Press ENTER to continue.


/home/karan/Documents/CPP/CD2
Enter a string: int g = (4 + 5)/(7-6);
Lexeme Token Category
int identifier
g identifier
= assignment operator
(delimiter
4 integer literal
+ addition operator
5 integer literal
) delimiter
/ division operator
(delimiter
7 integer literal
- subtraction operator
6 integer literal
) delimiter
; delimiter
Process returned 0 (0x0) execution time : 24.348 s
Press ENTER to continue.

Q3:Write a program to implement lexical analyzer using lex tool.

CODE:

```
%{
#include <stdio.h>
#include <stdlib.h>
%}

/* Definitions section */
DIGIT   [0-9]
LETTER  [a-zA-Z]

%%

if          { printf("Keyword: if\n"); }
else        { printf("Keyword: else\n"); }
while       { printf("Keyword: while\n"); }
{DIGIT}+    { printf("Number: %s\n", yytext); }
{LETTER}+   { printf("Identifier: %s\n", yytext); }

"+"         { printf("Operator: +\n"); }
"-"         { printf("Operator: -\n"); }
"*"         { printf("Operator: *\n"); }
"/"         { printf("Operator: /\n"); }
">"         { printf("Operator: >\n"); }
"<"         { printf("Operator: <\n"); }
"="         { printf("Operator: =\n"); }


","         { printf("Delimiter: ,\n"); }
";"         { printf("Delimiter: ;\n"); }
"("         { printf("Delimiter: (\n"); }
")"         { printf("Delimiter: )\n"); }
"{"         { printf("Delimiter: {\n"); }
"}"         { printf("Delimiter: }\n"); }
"["         { printf("Delimiter: [\n"); }
"]"         { printf("Delimiter: ]\n"); }

.           { /* Ignore any other characters */ }

%%

int main(int argc, char **argv) {
    yylex();
```

```
    return 0;
}

int yywrap() {
    return 1;
}
```

```
%{
#include <stdio.h>
#include <stdlib.h>
%}

/* Definitions section */
DIGIT    [0-9]
LETTER   [a-zA-Z]

%%

if          { printf("Keyword: if\n"); }
else        { printf("Keyword: else\n"); }
while       { printf("Keyword: while\n"); }
{DIGIT}+    { printf("Number: %s\n", yytext); }
{LETTER}+   { printf("Identifier: %s\n", yytext); }

"+"         { printf("Operator: +\n"); }
"-"         { printf("Operator: -\n"); }
"*"         { printf("Operator: *\n"); }
"/"         { printf("Operator: /\n"); }
">"         { printf("Operator: >\n"); }
"<"         { printf("Operator: <\n"); }
"="         { printf("Operator: =\n"); }


","         { printf("Delimiter: ,\n"); }
";"         { printf("Delimiter: ;\n"); }
"("         { printf("Delimiter: (\n"); }
")"         { printf("Delimiter: )\n"); }
"{"         { printf("Delimiter: {\n"); }
"}"         { printf("Delimiter: }\n"); }
"["         { printf("Delimiter: [\n"); }
"]"         { printf("Delimiter: ]\n"); }

.           { /* Ignore any other characters */ }

%%

int main(int argc, char **argv) {
    yylex();
    return 0;
}

int yywrap() {
    return 1;
}
~
```

OUTPUT:

```
karan@karansehgal-vivobook:~$ vim file4.l
karan@karansehgal-vivobook:~$ flex file4.l
karan@karansehgal-vivobook:~$ gcc lex.yy.c -o file4 -ll
karan@karansehgal-vivobook:~$ ./file4
int x = 5;
Identifier: int
Identifier: x
Operator: =
Number: 5
Delimiter: ;
```

```
if(a < b ){a = a - b;}
Keyword: if
Delimiter: (
Identifier: a
Operator: <
Identifier: b
Delimiter: )
Delimiter: {
Identifier: a
Operator: =
Identifier: a
Operator: -
Identifier: b
Delimiter: ;
Delimiter: }
```

```
int x = 5 + (3 * 8 );
Identifier: int
Identifier: x
Operator: =
Number: 5
Operator: +
Delimiter: (
Number: 3
Operator: *
Number: 8
Delimiter: )
Delimiter: ;
```

Q7:Write a C program for stack to use dynamic storage allocation.
Implement a stack data structure in C using dynamic memory allocation. The implementation should include functions for creating a stack, pushing elements onto the stack, popping elements from the stack, peeking at the top element, checking if the stack is empty, and displaying the contents of the stack.

CODE:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

#define MAX_EXPR_LENGTH 100
#define MAX_OPERANDS 50
#define MAX_OPERATORS 50

typedef struct {
char op;
char arg1[10];
char arg2[10];
char result[10];
} Quadruple;

int getPrecedence(char op);
void generateIntermediateCode(char* expression);

int main() {
char expression[MAX_EXPR_LENGTH];

printf("Intermediate Code Generator\n");

printf("Enter the expression: ");
fgets(expression, sizeof(expression), stdin);
expression[strcspn(expression, "\n")] = 0;
generateIntermediateCode(expression);

return 0;
}
void generateIntermediateCode(char* expression) {
int len = strlen(expression);
char* exp = expression;
char operands[MAX_OPERANDS][10];
char operators[MAX_OPERATORS];
int operandCount = 0, operatorCount = 0;
Quadruple* quads = NULL;
int quadCount = 0;
int tempVarCount = 1;
```

```
if (len > 3) {
for (int i = 0; i < len; i++) {
if (isalnum(exp[i])) {

char operand[10] = {0};
int j = 0;
while (isalnum(exp[i])) {
operand[j++] = exp[i++];
}
i--;
strcpy(operands[operandCount++], operand);
} else if (exp[i] != ' ') {

operators[operatorCount++] = exp[i];
}
}
}


quads = (Quadruple*)malloc(sizeof(Quadruple) * operatorCount);

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

int precedence = getPrecedence(operators[i]);

quads[quadCount].op = operators[i];
strcpy(quads[quadCount].arg1, operands[i]);
strcpy(quads[quadCount].arg2, operands[i+1]);
sprintf(quads[quadCount].result, "t%d", tempVarCount++);

if (i < operatorCount - 1) {
strcpy(operands[i+1], quads[quadCount].result);
}

quadCount++;
}

printf("\nIntermediate Code:\n");
for (int i = 0; i < quadCount; i++) {
printf("%s = %s %c %s\n", quads[i].result, quads[i].arg1, quads[i].op, quads[i].arg2);
}

free(quads);
}

int getPrecedence(char op) {
switch (op) {
case '+':
case '-':
```

```
return 1;
case '*':
case '/':
return 2;
case '^':
return 3;
default:
return 0;
}
}
```

home > karan > C k.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

#define MAX_EXPR_LENGTH 100
#define MAX_OPERANDS 50
#define MAX_OPERATORS 50

typedef struct {
    char op;
    char arg1[10];
    char arg2[10];
    char result[10];
} Quadruple;

int getPrecedence(char op);
void generateIntermediateCode(char* expression);

int main() {
    char expression[MAX_EXPR_LENGTH];

    printf("Intermediate Code Generator\n");

    printf("Enter the expression: ");
    fgets(expression, sizeof(expression), stdin);
    expression[strcspn(expression, "\n")] = 0;
    generateIntermediateCode(expression);

    return 0;
}
void generateIntermediateCode(char* expression) {
    int len = strlen(expression);
    char* exp = expression;
    char operands[MAX_OPERANDS][10];
    char operators[MAX_OPERATORS];
    int operandCount = 0, operatorCount = 0;
    Quadruple* quads = NULL;
    int quadCount = 0;
    int tempVarCount = 1;

    if (len > 3) {
        for (int i = 0; i < len; i++) {
            if (isalnum(exp[i])) {

                char operand[10] = {0};
                int j = 0;
                while (isalnum(exp[i])) {
                    operand[j++] = exp[i++];
                }
                i--;
                strcpy(operands[operandCount++], operand);
```

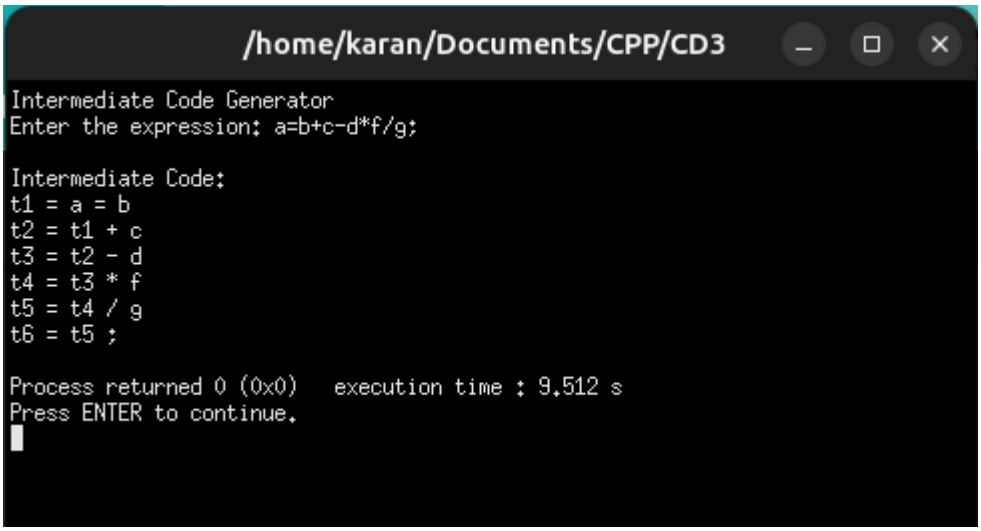
home > karan > C k.c

```c
void generateIntermediateCode(char* expression) {
    if (len > 3) {
        for (int i = 0; i < len; i++) {
            if (isalnum(exp[i])) {
                while (isalnum(exp[i])) {
                }
                i--;
                strcpy(operands[operandCount++], operand);
            } else if (exp[i] != ' ') {

                operators[operatorCount++] = exp[i];
            }
        }
    }


    quads = (Quadruple*)malloc(sizeof(Quadruple) * operatorCount);

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

        int precedence = getPrecedence(operators[i]);

        quads[quadCount].op = operators[i];
        strcpy(quads[quadCount].arg1, operands[i]);
        strcpy(quads[quadCount].arg2, operands[i+1]);
        sprintf(quads[quadCount].result, "t%d", tempVarCount++);

        if (i < operatorCount - 1) {
            strcpy(operands[i+1], quads[quadCount].result);
        }

        quadCount++;
    }

    printf("\nIntermediate Code:\n");
    for (int i = 0; i < quadCount; i++) {
        printf("%s = %s %c %s\n", quads[i].result, quads[i].arg1, quads[i].op, quads[i].arg2);
    }

    free(quads);
}
```

home > karan > C k.c

```c
void generateIntermediateCode(char* expression) {
    printf("\nIntermediate Code:\n");
    for (int i = 0; i < quadCount; i++) {
        printf("%s = %s %c %s\n", quads[i].result, quads[i].arg1, quads[i].op, quads[i].arg2);
    }

    free(quads);
}

int getPrecedence(char op) {
    switch (op) {
        case '+':
        case '-':
            return 1;
        case '*':
        case '/':
            return 2;
        case '^':
            return 3;
        default:
            return 0;
    }
}

```

OUTPUT:


/home/karan/Documents/CPP/CD3
Intermediate Code Generator
Enter the expression: a=b+c-d*f/g;
Intermediate Code:
t1 = a = b
t2 = t1 + c
t3 = t2 - d
t4 = t3 * f
t5 = t4 / g
t6 = t5 ;
Process returned 0 (0x0) execution time : 9.512 s
Press ENTER to continue.