

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;  
}
```

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <ctype.h>
5
6  #define MAX_SYMBOLS 100
7  #define MAX_NAME_LENGTH 50
8  #define MAX_INPUT_LENGTH 1000
9
10 typedef struct {
11     char name[MAX_NAME_LENGTH];
12     char type[20];
13     void* address;
14     size_t size;
15 } Symbol;
16
17 Symbol symbolTable[MAX_SYMBOLS];
18 int symbolCount = 0;
19
20 void insertSymbol(const char* name, const char* type, size_t size) {
21     if (symbolCount >= MAX_SYMBOLS) {
22         printf("Symbol table is full.\n");
23         return;
24     }
25
26     // Check if symbol already exists
27     for (int i = 0; i < symbolCount; i++) {
28         if (strcmp(symbolTable[i].name, name) == 0) {
29             return; // Symbol already exists, don't insert again
30         }
31     }
32
33     strcpy(symbolTable[symbolCount].name, name);
34     strcpy(symbolTable[symbolCount].type, type);
35     symbolTable[symbolCount].size = size;
36     symbolTable[symbolCount].address = malloc(size);
37
38     if (symbolTable[symbolCount].address == NULL) {
39         printf("Memory allocation failed for symbol %s.\n", name);
40         return;
41     }
42
43     symbolCount++;
44 }
45
46 void displaySymbolTable() {
47     printf("\nSymbol Table:\n");
48     printf("%-20s %-10s %-15s %s\n", "Name", "Type", "Address", "Size");
49     printf("-----\n");
50     for (int i = 0; i < symbolCount; i++) {
51         printf("%-20s %-10s %-15p %zu\n",
52             symbolTable[i].name,
53             symbolTable[i].type,
54             symbolTable[i].address,

```

```

46 void displaySymbolTable() {
47     for (int i = 0; i < symbolCount; i++) {
48         printf("%-20s %-10s %-15p %zu\n",
49             symbolTable[i].type,
50             symbolTable[i].address,
51             symbolTable[i].size);
52     }
53 }
54
55 int isOperator(const char* str) {
56     return (strlen(str) == 1 && strchr("+-*/", str[0])) ||
57            (strcmp(str, "AND") == 0) ||
58            (strcmp(str, "OR") == 0) ||
59            (strcmp(str, "NOT") == 0);
60 }
61
62 void tokenizeAndInsert(const char* input) {
63     char token[MAX_NAME_LENGTH] = "";
64     int tokenIndex = 0;
65
66     for (int i = 0; input[i] != '\0'; i++) {
67         if (islower(input[i])) {
68             // Symbol (single lowercase letter)
69             token[0] = input[i];
70             token[1] = '\0';
71             insertSymbol(token, "int", sizeof(int));
72         } else if (strchr("+-*/", input[i])) {
73             // Operator (single character)
74             token[0] = input[i];
75             token[1] = '\0';
76             insertSymbol(token, "operator", sizeof(char));
77         } else if (isupper(input[i])) {
78             // Start of a potential operator (AND, NOT, OR)
79             tokenIndex = 0;
80             while (isupper(input[i]) && tokenIndex < MAX_NAME_LENGTH - 1) {
81                 token[tokenIndex++] = input[i++];
82             }
83             token[tokenIndex] = '\0';
84             i--; // Move back one character as the loop will increment
85
86             if (isOperator(token)) {
87                 insertSymbol(token, "operator", sizeof(char));
88             }
89         }
90         // Ignore other characters (like spaces)
91     }
92 }
93
94 int main() {
95     char input[MAX_INPUT_LENGTH];
96
97     printf("Enter expression (terminate with '$'):\n");
98     fgets(input, MAX_INPUT_LENGTH, stdin);
99     input[strlen(input) - 1] = '\0'; // Remove newline

```

```
97
98 int main() {
99     char input[MAX_INPUT_LENGTH];
100
101     printf("Enter expression (terminate with '$'):\n");
102     fgets(input, MAX_INPUT_LENGTH, stdin);
103     input[strcspn(input, "\n")] = 0; // Remove newline
104
105     if (strcmp(input, "$") == 0) {
106         printf("No input provided.\n");
107         return 0;
108     }
109
110     tokenizeAndInsert(input);
111
112     displaySymbolTable();
113
114     // Clean up allocated memory
115     for (int i = 0; i < symbolCount; i++) {
116         free(symbolTable[i].address);
117     }
118
119     return 0;
120 }
```


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 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;
}

```

```

1  #include <stdio.h>
2  #include <ctype.h>
3  #include <string.h>
4
5  typedef struct
6  {
7      char category[50];
8      char lexeme[100];
9  } Token;
10 Token getNextToken(char **input)
11 {
12     Token token;
13     int ch = **input;
14     while (isspace(ch))
15     {
16         (*input)++;
17         ch = **input;
18     }
19     if (ch == '\0')
20     {
21         strcpy(token.category, "END_OF_INPUT");
22         strcpy(token.lexeme, "");
23         return token;
24     }
25     if (isalpha(ch))
26     {
27         int i = 0;
28         while (isalnum(ch) || ch == '_')
29         {
30             token.lexeme[i++] = ch;
31             (*input)++;
32             ch = **input;
33         }
34         token.lexeme[i] = '\0';
35         strcpy(token.category, "identifier");
36     }
37     else if (isdigit(ch))
38     {
39         int i = 0;
40         while (isdigit(ch))
41         {
42             token.lexeme[i++] = ch;
43             (*input)++;
44             ch = **input;
45         }
46         token.lexeme[i] = '\0';
47         strcpy(token.category, "integer literal");
48     }
49     else if (ch == '+')
50     {
51         token.lexeme[0] = ch;
52         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;
}

```

```

}
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\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

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <ctype.h>
5
6  #define MAX_EXPR_LENGTH 100
7  #define MAX_OPERANDS 50
8  #define MAX_OPERATORS 50
9
10 typedef struct {
11     char op;
12     char arg1[10];
13     char arg2[10];
14     char result[10];
15 } Quadruple;
16
17 int getPrecedence(char op);
18 void generateIntermediateCode(char* expression);
19
20 int main() {
21     char expression[MAX_EXPR_LENGTH];
22
23     printf("Intermediate Code Generator\n");
24
25     printf("Enter the expression: ");
26     fgets(expression, sizeof(expression), stdin);
27     expression[strcspn(expression, "\n")] = 0;
28     generateIntermediateCode(expression);
29
30     return 0;
31 }
32 void generateIntermediateCode(char* expression) {
33     int len = strlen(expression);
34     char* exp = expression;
35     char operands[MAX_OPERANDS][10];
36     char operators[MAX_OPERATORS];
37     int operandCount = 0, operatorCount = 0;
38     Quadruple* quads = NULL;
39     int quadCount = 0;
40     int tempVarCount = 1;
41
42     if (len > 3) {
43         for (int i = 0; i < len; i++) {
44             if (isdigit(exp[i])) {
45
46                 char operand[10] = {0};
47                 int j = 0;
48                 while (isdigit(exp[i])) {
49                     operand[j++] = exp[i++];
50                 }
51                 i--;
52                 strcpy(operands[operandCount++], operand);
```

home > karan > C k.c

```
32 void generateIntermediateCode(char* expression) {
42     if (len > 3) {
43         for (int i = 0; i < len; i++) {
44             if (isalnum(exp[i])) {
48                 while (isalnum(exp[i])) {
49                     operand[i++] = exp[i++];
50                 }
51                 i--;
52                 strcpy(operands[operandCount++], operand);
53             } else if (exp[i] != ' ') {
54                 operators[operatorCount++] = exp[i];
55             }
56         }
57     }
58 }
59
60
61 quads = (Quadruple*)malloc(sizeof(Quadruple) * operatorCount);
62
63 for (int i = 0; i < operatorCount; i++) {
64
65     int precedence = getPrecedence(operators[i]);
66
67     quads[quadCount].op = operators[i];
68     strcpy(quads[quadCount].arg1, operands[i]);
69     strcpy(quads[quadCount].arg2, operands[i+1]);
70     sprintf(quads[quadCount].result, "%d", tempVarCount++);
71
72     if (i < operatorCount - 1) {
73         strcpy(operands[i+1], quads[quadCount].result);
74     }
75
76     quadCount++;
77 }
78
79 printf("\nIntermediate Code:\n");
80 for (int i = 0; i < quadCount; i++) {
81     printf("%s = %s %c %s\n", quads[i].result, quads[i].arg1, quads[i].op, quads[i].arg2);
82 }
83
84 free(quads);
85 }
86
```

home > karan > `C` k.c

```
32 void generateIntermediateCode(char* expression) {
79     printf("\nIntermediate Code:\n");
80     for (int i = 0; i < quadCount; i++) {
81         printf("%s = %s %c %s\n", quads[i].result, quads[i].arg1, quads[i].op, quads[i].arg2);
82     }
83
84     free(quads);
85 }
86
87 int getPrecedence(char op) {
88     switch (op) {
89         case '+':
90         case '-':
91             return 1;
92         case '*':
93         case '/':
94             return 2;
95         case '^':
96             return 3;
97         default:
98             return 0;
99     }
100 }
101
```

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.
█
```