CD Lab-3 CONSTRUCTION OF TOKEN GENERATOR

Name: Ketan Goud
Reg No: 220905260
Roll No: 39
Section: CSE D- D2

1. Write functions to identify the following tokens.
a. Arithmetic, relational and logical operators.

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

struct token
{
   char token_name[50];
   int row;
   int col;
   char type[20];
   int index;
};
const char* keywords[]={"int", "float", "double", "char", "if", "else", "for","return",
                        "while", "return", "void", "switch", "case", "break", "continue", "default",
                        "struct", "union", "enum", "long", "short", "const", "sizeof"};
#define NUM_KEYWORDS 22

int iskeyword(const char *str)
{
   for (int i = 0; i < NUM_KEYWORDS; i++)
   {
      if (strcmp(str, keywords[i]) == 0)
      {
         return 1;
      }
   }
   return 0;
}
void remove_white_spaces(FILE *fa, FILE *fb)
{
   int ca;
   ca = getc(fa);
   while (ca != EOF)
   {
      if (ca == ' ')
      {
         while (ca == ' ')
         {
            ca = getc(fa);
         }
         putc(' ',fb);
```

```
        }
        else if (ca == '\t')
        {
            while (ca == '\t')
            {
                ca = getc(fa);
            }
            putc(' ',fb);
        }
        else
        {
            putc(ca, fb);
            ca = getc(fa);
        }
    }
    fclose(fa);
    fclose(fb);
}
void remove_headers(FILE *fa, FILE *fb)
{
    char line[1024];
    while (fgets(line, sizeof(line), fa))
    {
        if (line[0] != '#')
            fputs(line, fb);
    }
    fclose(fa);
    fclose(fb);
}
void identify_operators(FILE *fa, FILE *fb)
{
    int ca, next;
    int row = 1, col = 1;
    struct token current;
    int index = 0;
    ca = fgetc(fa);

    while (ca != EOF)
    {
        if (isspace(ca))
        {
            if (ca == '\n')
            {
                row++;
                col = 1;
            }
            else
            {
                col++;
            }
        }
        else if (ca == '+' || ca == '-' || ca == '*' || ca == '/' || ca == '%')
```

```
        {
            current.row = row;
            current.col = col;
            current.index = index;
            current.token_name[0] = ca;
            current.token_name[1] = '\0';
            strcpy(current.type, "Arithmetic");
            fprintf(fb, "<'%s', %d, %d, '%s'>\n", current.token_name, current.row, current.col,
current.type);
            col++;
            index++;
        }
        else if (ca == '=' || ca == '!' || ca == '<' || ca == '>')
        {
            next = fgetc(fa);
            col++;
            if (next == '=')
            {
                current.row = row;
                current.col = col;
                current.index = index;
                current.token_name[0] = ca;
                current.token_name[1] = next;
                current.token_name[2] = '\0';
                strcpy(current.type, "Relational");
                fprintf(fb, "<'%s', %d, %d, '%s'>\n", current.token_name, current.row, current.col,
current.type);
                col++;
                index++;
            }
            else
            {
                ungetc(next, fa);
                col--;
                if(ca=='=')
                {
                    current.row = row;
                    current.col = col;
                    current.index = index;
                    current.token_name[0] = ca;
                    current.token_name[1] = '\0';
                    strcpy(current.type, "Assignment");
                    fprintf(fb, "<'%s', %d, %d, '%s'>\n", current.token_name, current.row, current.col,
current.type);
                    col++;
                    index++;
                }
                else
                {
                    current.row = row;
                current.col = col;
                current.index = index;
```

```
            current.token_name[0] = ca;
            current.token_name[1] = '\0';
            strcpy(current.type, "Relational");
            fprintf(fb, "<'%s', %d, %d, '%s'>\n", current.token_name, current.row, current.col,
current.type);
            col++;
            index++;
            }
        }
    }
    else if (ca == '&' || ca == '|' || ca == '!')
    {
        next = fgetc(fa);
        col++;
        if (ca == '&' && next == '&')
        {
            current.row = row;
            current.col = col;
            current.index = index;
            current.token_name[0] = '&';
            current.token_name[1] = '&';
            current.token_name[2] = '\0';
            strcpy(current.type, "Logical");
            fprintf(fb, "<'%s', %d, %d, '%s'>\n", current.token_name, current.row, current.col,
current.type);
            col++;
            index++;
        }
        else if (ca == '|' && next == '|')
        {
            current.row = row;
            current.col = col;
            current.index = index;
            current.token_name[0] = '|';
            current.token_name[1] = '|';
            current.token_name[2] = '\0';
            strcpy(current.type, "Logical");
            fprintf(fb, "<'%s', %d, %d, '%s'>\n", current.token_name, current.row, current.col,
current.type);
            col++;
            index++;
        }
        else
        {
            ungetc(next, fa);
            col--;
        }
    }
    ca=fgetc(fa);
  }
}
```

```
int main()
{
    FILE *fa, *fb, *fc, *fd, *fe, *ff;
    fa = fopen("digit.c", "r");
    fb = fopen("digit1.c", "w");
    remove_headers(fa, fb);
    fc = fopen("digit1.c", "r");
    fd = fopen("digit2.c", "w");
    remove_white_spaces(fc,fd);
    fe = fopen("digit2.c", "r");
    ff = fopen("digitout.txt","w");
    identify_operators(fe,ff);
    fclose(fe);
    fclose(ff);
}
```

digitout.txt | digitout2.txt | digitout3.txt

```
<'=', 3, 3, 'Assignment'>
<'=', 4, 3, 'Assignment'>
<'=', 5, 3, 'Assignment'>
<'=', 5, 4, 'Assignment'>
<'>=', 7, 3, 'Relational'>
<'=', 9, 2, 'Assignment'>
<'*', 9, 3, 'Arithmetic'>
<'=', 10, 2, 'Assignment'>
<'/', 10, 3, 'Arithmetic'>
<'<=', 12, 4, 'Relational'>
<'=', 14, 2, 'Assignment'>
<'+', 14, 3, 'Arithmetic'>
<'=', 15, 2, 'Assignment'>
<'-', 15, 3, 'Arithmetic'>
<'==', 17, 4, 'Relational'>
<'=', 19, 2, 'Assignment'>
<'+', 19, 3, 'Arithmetic'>
<'-', 19, 4, 'Arithmetic'>
<'=', 20, 2, 'Assignment'>
<'-', 20, 3, 'Arithmetic'>
<'+', 20, 4, 'Arithmetic'>
<'!=', 22, 4, 'Relational'>
<'>', 24, 2, 'Relational'>
<'=', 26, 2, 'Assignment'>
<'+', 26, 3, 'Arithmetic'>
<'=', 30, 2, 'Assignment'>
<'+', 30, 3, 'Arithmetic'>
```

b. Special symbols, keywords, numerical constants, string literals and identifiers.

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

struct token
{
    char token_name[50];
    int row;
    int col;
    char type[20];
    int index;
};
const char *keywords[] = {"int", "float", "double", "char", "if", "else", "for", "return",
                          "while", "return", "void", "switch", "case", "break", "continue", "default",
                          "struct", "union", "enum", "long", "short", "const", "sizeof"};
#define NUM_KEYWORDS 23

int iskeyword(const char *str)
{
    for (int i = 0; i < NUM_KEYWORDS; i++)
    {
        if (strcmp(str, keywords[i]) == 0)
        {
            return 1;
        }
    }
    return 0;
}
void remove_white_spaces(FILE *fa, FILE *fb)
{
    int ca;
    ca = getc(fa);
    while (ca != EOF)
    {
        if (ca == ' ')
        {
            while (ca == ' ')
            {
                ca = getc(fa);
            }
            putc(' ', fb);
        }
        else if (ca == '\t')
        {
            while (ca == '\t')
            {
                ca = getc(fa);
            }
            putc(' ', fb);
```

```
        }
        else
        {
            putc(ca, fb);
            ca = getc(fa);
        }
    }
    fclose(fa);
    fclose(fb);
}
void remove_headers(FILE *fa, FILE *fb)
{
    char line[1024];
    while (fgets(line, sizeof(line), fa))
    {
        if (line[0] != '#')
            fputs(line, fb);
    }
    fclose(fa);
    fclose(fb);
}
void identify_operators(FILE *fa, FILE *fb)
{
    int ca, next;
    int row = 1, col = 1;
    struct token current;
    int index = 0;
    ca = fgetc(fa);

    while (ca != EOF)
    {
        if (isspace(ca))
        {
            if (ca == '\n')
            {
                row++;
                col = 1;
            }
            else
            {
                col++;
            }
        }
        else if (ca == '&' || ca == '|' || ca == '!')
        {
            next = fgetc(fa);
            col++;
            if (ca == '&' && next == '&')
            {
                current.row = row;
                current.col = col;
                current.index = index;
```

```
            current.token_name[0] = '&';
            current.token_name[1] = '&';
            current.token_name[2] = '\0';
            strcpy(current.type, "Logical");
            fprintf(fb, "<'%s', %d, %d, '%s'>\n", current.token_name, current.row, current.col,
current.type);
            col++;
            index++;
        }
        else if (ca == '|' && next == '|')
        {
            current.row = row;
            current.col = col;
            current.index = index;
            current.token_name[0] = '|';
            current.token_name[1] = '|';
            current.token_name[2] = '\0';
            strcpy(current.type, "Logical");
            fprintf(fb, "<'%s', %d, %d, '%s'>\n", current.token_name, current.row, current.col,
current.type);
            col++;
            index++;
        }
        else
        {
            ungetc(next, fa);
            col--;
        }
    }
    else if (isdigit(ca))
    {
        int i = 0;
        while (isdigit(ca) || ca == '.')
        {
            current.token_name[i++] = ca;
            ca = fgetc(fa);
            col++;
        }
        current.token_name[i] = '\0';
        strcpy(current.type, "Numeric");
        fprintf(fb, "<'%s', %d, %d, '%s'>\n", current.token_name, current.row, current.col,
current.type);
        index++;
    }
    else if (isalpha(ca) || ca == '_')
    {
        int i = 0;
        current.row = row;
        current.col = col;
        while (isalnum(ca) || ca == '_')
        {
            current.token_name[i++] = ca;
```

```
            ca = fgetc(fa);
            col++;
        }
        current.token_name[i] = '\0';
        if (iskeyword(current.token_name))
        {
            strcpy(current.type, "Keyword");
        }
        else
        {
            strcpy(current.type, "Identifier");
        }
        fprintf(fb, "<'%s', %d, %d, '%s'>\n", current.token_name, current.row, current.col,
current.type);
        index++;
        ungetc(ca, fa);
    }

    else if (ca == '"')
    {
        char quote = ca;
        int i = 0;
        current.token_name[i++] = quote;
        ca = fgetc(fa);
        col++;
        while (ca != quote && ca != EOF)
        {
            current.token_name[i++] = ca;
            ca = fgetc(fa);
            col++;
        }
        if (ca == quote)
        {
            current.token_name[i++] = quote;
            current.token_name[i] = '\0';
            strcpy(current.type, "String Literal");
            fprintf(fb, "<'%s', %d, %d, '%s'>\n", current.token_name, current.row, current.col,
current.type);
            index++;
            col++;
        }
    }
    else
    {
        current.row = row;
        current.col = col;
        current.index = index;
        current.token_name[0] = ca;
        current.token_name[1] = '\0';
        strcpy(current.type, "Special Symbol");
        fprintf(fb, "<'%s', %d, %d, '%s'>\n", current.token_name, current.row, current.col,
current.type);
```

```
            index++;
            col++;
        }
        ca = fgetc(fa);
    }
}

int main()
{
    FILE *fa, *fb, *fc, *fd, *fe, *ff;
    fa = fopen("digit.c", "r");
    fb = fopen("digit1.c", "w");
    remove_headers(fa, fb);
    fc = fopen("digit1.c", "r");
    fd = fopen("digit2.c", "w");
    remove_white_spaces(fc, fd);
    fe = fopen("digit2.c", "r");
    ff = fopen("digitout2.txt", "w");
    identify_operators(fe, ff);
    fclose(fe);
    fclose(ff);
}
```

digitout.txt | digitout2.txt | digitout3.txt | q2.c

```
<'int', 1, 1, 'Keyword'>
<'main', 1, 5, 'Identifier'>
<'(', 1, 9, 'Special Symbol'>
<')', 1, 10, 'Special Symbol'>
<'{', 2, 1, 'Special Symbol'>
<'int', 3, 2, 'Keyword'>
<'a', 3, 6, 'Identifier'>
<'=', 3, 7, 'Special Symbol'>
<'20', 3, 7, 'Numeric'>
<'int', 4, 2, 'Keyword'>
<'b', 4, 6, 'Identifier'>
<'=', 4, 7, 'Special Symbol'>
<'10', 4, 7, 'Numeric'>
<'int', 5, 2, 'Keyword'>
<'c', 5, 6, 'Identifier'>
<'=', 5, 7, 'Special Symbol'>
<'0', 5, 7, 'Numeric'>
<'d', 5, 9, 'Identifier'>
<'=', 5, 10, 'Special Symbol'>
<'0', 5, 10, 'Numeric'>
<'printf', 6, 2, 'Identifier'>
<'(', 6, 8, 'Special Symbol'>
<'"Sample Problem\n"', 6, 8, 'String Literal'>
<')', 6, 27, 'Special Symbol'>
<';', 6, 28, 'Special Symbol'>
<'if', 7, 2, 'Keyword'>
<'(', 7, 4, 'Special Symbol'>
<'a', 7, 5, 'Identifier'>
<'>', 7, 6, 'Special Symbol'>
<'=', 7, 7, 'Special Symbol'>
<'b', 7, 8, 'Identifier'>
<')', 7, 9, 'Special Symbol'>
<'{', 8, 2, 'Special Symbol'>
<'c', 9, 2, 'Identifier'>
```

2. Design a lexical analyzer that includes a getNextToken() function for processing a simple C program. The analyzer should construct a token structure containing the row number, column number, and token type for each identified token. The getNextToken() function must ignore tokens located within single-line or multi-line comments, as well as those found inside string literals. Additionally, it should strip out preprocessor directives.

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

struct token {
    char token_name[50];
    int row;
    int col;
    char type[20];
    int index;
};

const char *keywords[] = {"int", "float", "double", "char", "if", "else", "for", "return",
                          "while", "void", "switch", "case", "break", "continue", "default",
                          "struct", "union", "enum", "long", "short", "const", "sizeof"};
#define NUM_KEYWORDS 22

int iskeyword(const char *str)
{
    for (int i = 0; i < NUM_KEYWORDS; i++)
    {
        if (strcmp(str, keywords[i]) == 0)
        {
            return 1;
        }
    }
    return 0;
}

void remove_headers(FILE *fa, FILE *fb)
{
    char line[1024];
    while (fgets(line, sizeof(line), fa))
    {
        if (line[0] != '#')
        {
            fputs(line, fb);
        }
    }
    fclose(fa);
    fclose(fb);
}

int is_in_string_or_comment(int ca, FILE *fa, int *col, int *row)
{
```

```
    static int inside_string = 0;
    static int inside_comment = 0;
    int next;

    if (inside_comment)
    {
        if (ca == '*' && (next = fgetc(fa)) == '/')
        {
            inside_comment = 0;
            *col += 2;
        }
        else
        {
            (*col)++;
        }
        return 1;
    }
    if (inside_string)
    {
        if (ca == '"')
        {
            inside_string = 0;
        }
        (*col)++;
        return 1;
    }
    if (ca == '"')
    {
        inside_string = 1;
        return 1;
    }
    if (ca == '/' && (next = fgetc(fa)) == '/')
    {
        inside_comment = 1;
        *col += 2;
        return 1;
    }

    if (ca == '/' && (next = fgetc(fa)) == '*')
    {
        inside_comment = 1;
        *col += 2;
        return 1;
    }
    return 0;
}

struct token getNextToken(FILE *fa, int *row, int *col, int *index)
{
    int ca;
    struct token current;
    ca = fgetc(fa);
```

```
while (ca != EOF && is_in_string_or_comment(ca, fa, col, row))
{
    ca = fgetc(fa);
}
if (ca == EOF)
{
    current.token_name[0] = '\0';
    current.row = *row;
    current.col = *col;
    current.index = -1;
    strcpy(current.type, "EOF");
    return current;
}
while (isspace(ca))
{
    if (ca == '\n')
    {
        (*row)++;
        *col = 1;
    }
    else
    {
        (*col)++;
    }
    ca = fgetc(fa);
}
if (isalpha(ca) || ca == '_')
{
    int i = 0;
    current.row = *row;
    current.col = *col;
    while (isalnum(ca) || ca == '_')
    {
        current.token_name[i++] = ca;
        ca = fgetc(fa);
        (*col)++;
    }
    current.token_name[i] = '\0';
    if (iskeyword(current.token_name))
    {
        strcpy(current.type, "Keyword");
    }
    else
    {
        strcpy(current.type, "Identifier");
    }
    current.index = (*index)++;
    return current;
}
if (isdigit(ca))
{
    int i = 0;
```

```
        while (isdigit(ca) || ca == '.')
        {
            current.token_name[i++] = ca;
            ca = fgetc(fa);
            (*col)++;
        }
        current.token_name[i] = '\0';
        strcpy(current.type, "Numeric");
        current.index = (*index)++;
        return current;
    }
    current.row = *row;
    current.col = *col;
    current.token_name[0] = ca;
    current.token_name[1] = '\0';
    strcpy(current.type, "Special Symbol");
    current.index = (*index)++;
    (*col)++;

    return current;
}

int main()
{
    FILE *fa, *fb, *fc, *ff;
    fa = fopen("digit.c", "r");
    if (fa == NULL)
    {
        printf("Error opening input file digit.c\n");
        return 1;
    }
    fb = fopen("digit1.c", "w");
    remove_headers(fa, fb);
    fc = fopen("digit1.c", "r");
    if (fc == NULL)
    {
        printf("Error opening intermediate file digit1.c\n");
        return 1;
    }
    ff = fopen("q2output.txt", "w");
    int row = 1, col = 1, index = 0;
    struct token t;
    while (1)
    {
        t = getNextToken(fc, &row, &col, &index);
        if (t.index == -1)
        {
            break;
        }
        fprintf(ff, "<'%s', %d, %d, '%s'>\n", t.token_name, t.row, t.col, t.type);
    }
    fclose(fc);
```

```
    fclose(ff);
    return 0;
}
```

digitout.txt | digitout2.txt | digitout3.txt | q2.c

```
<'int', 1, 1, 'Keyword'>
<'main', 1, 5, 'Identifier'>
<'(', 1, 9, 'Special Symbol'>
<')', 1, 10, 'Special Symbol'>
<'{', 2, 1, 'Special Symbol'>
<'int', 3, 2, 'Keyword'>
<'a', 3, 6, 'Identifier'>
<'=', 3, 7, 'Assignment'>
<'20', 3, 7, 'Numeric'>
<'int', 4, 2, 'Keyword'>
<'b', 4, 6, 'Identifier'>
<'=', 4, 7, 'Assignment'>
<'10', 4, 7, 'Numeric'>
<'int', 5, 2, 'Keyword'>
<'c', 5, 6, 'Identifier'>
<'=', 5, 7, 'Assignment'>
<'0', 5, 7, 'Numeric'>
<'d', 5, 9, 'Identifier'>
<'=', 5, 10, 'Assignment'>
<'0', 5, 10, 'Numeric'>
<'printf', 6, 2, 'Identifier'>
<'(', 6, 8, 'Special Symbol'>
<'"', 6, 9, 'Special Symbol'>
<'Sample', 6, 10, 'Identifier'>
<'Problem', 6, 17, 'Identifier'>
<'\', 6, 24, 'Special Symbol'>
<'n', 6, 25, 'Identifier'>
<'"', 6, 26, 'Special Symbol'>
<')', 6, 27, 'Special Symbol'>
<';', 6, 28, 'Special Symbol'>
<'if', 7, 2, 'Keyword'>
<'(', 7, 4, 'Special Symbol'>
<'a', 7, 5, 'Identifier'>
<'>=', 7, 7, 'Relational'>
```