Lab-3 CONSTRUCTION OF TOKEN GENERATOR

- Name - Aditya Sinha
- Reg.No - 230905218
- CSE-A-27

Q) Make a lexical analyzer.

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

typedef struct token{
    char token_name [50];
    unsigned int row,col;
}token;

typedef struct {
    char name[100];
    char value[100];
} Macro;

Macro macros[100];
int macroCount = 0;

const char *keywords[] = {
    "auto", "break", "case", "char", "const",
    "continue", "default", "do", "double",
    "else", "enum", "extern", "float", "for",
    "goto", "if", "int", "long", "register",
    "return", "short", "signed", "sizeof",
    "static", "struct", "switch", "typedef",
    "union", "unsigned", "void", "volatile", "while"
};

//Preprocessing part
void addMacro(const char *name, const char *value);
const char* getMacroValue(const char *name);
void preprocess(FILE *src, FILE * dst);

//Token identifying
token isKeyword(int ch, FILE *src, int *row, int *col);
token isIdentifier(int ch, FILE *src, int *row, int *col);

token isOperator(int ch, FILE *src, int *row, int *col);

token isRelationalOperator(int ch, FILE *src, int *row, int *col);
token isArithmeticOperator(int ch, FILE *src, int *row, int *col);
token isLogicalOperator(int ch, FILE *src, int *row, int *col);
token isBitwiseOperator(int ch, FILE *src, int *row, int *col);
token isConditionalOperator(int ch, int *row, int *col);
token isAssignmentOperator(int ch, FILE *src, int *row, int *col);

token isStringLiteral(int ch, FILE *src, int *row, int *col);

token isNumber(int ch, FILE *src, int *row, int *col);

//Token Server
token getNextToken(FILE *src, int *row, int *col);

//Helpers
void PrintToken(token t, FILE *dst);
void copyFile(FILE *src, FILE *dst);
void postprocess(FILE *src, FILE *dst);

int main(){
    printf ("Enter program to lexical analyse: ");
    char file[100];
    scanf ("%s", file);

    FILE *src = fopen(file, "r");
    if (src == NULL) {
        perror("error ");
        return 1;
    }

    FILE *tmp = fopen("tmp.txt", "w+");
    FILE *dst = fopen("ans.txt", "w+");

    preprocess(src, tmp);
    fseek(tmp, 0, SEEK_SET);

    int ch;
    int row = 1;
    int col = 1;
    token curr;

    while ((ch = fgetc(tmp)) != EOF) {
        ;
        if (ch == ' ' || ch == '\t') {
            col++;
            continue;
        }
        else if (ch == '\n') {
            row++;
            col = 1;
            putc(ch, dst);
            continue;
        }
        else {
            fseek(tmp, -1, SEEK_CUR);
            curr = getNextToken(tmp, &row, &col);
        }
        PrintToken(curr, dst);
    }

    fclose(tmp);

    fseek(dst, 0, SEEK_SET);
    tmp = fopen("tmp.txt", "w+");

    copyFile(dst, tmp);

    fseek(tmp, 0, SEEK_SET);
    fclose(dst);

    dst = fopen("ans.txt", "w");
    postprocess(tmp, dst);

    fclose(src);
    fclose(tmp);
    fclose(dst);
    return 0;
}

//Preprocessing part

void preprocess(FILE *src, FILE *dst) {
    int ch;
    char token[256];
    int tlen;

    while ((ch = fgetc(src)) != EOF) {

        if (ch == '/') {
            int next = fgetc(src);

            if (next == '/') {
                putc(' ', dst);
                putc(' ', dst);
                while ((ch = fgetc(src)) != EOF && ch != '\n')
                    putc(' ', dst);
                if (ch == '\n')
                    putc('\n', dst);
                continue;
            }

            if (next == '*') {
                putc(' ', dst);
                putc(' ', dst);
                int prev = 0;
                while ((ch = fgetc(src)) != EOF) {
                    if (ch == '\n')
                        putc('\n', dst);
                    else
                        putc(' ', dst);
                    if (prev == '*' && ch == '/')
                        break;
                    prev = ch;
                }
                continue;
            }

            putc('/', dst);
            ungetc(next, src);
            continue;
        }

        if (ch == '#') {
            char directive[20];
            int dlen = 0;

            directive[dlen++] = ch;

            while ((ch = fgetc(src)) != EOF && !isspace(ch)) {
                directive[dlen++] = ch;
            }
            directive[dlen] = '\0';

            if (strcmp(directive, "#include") == 0) {
                while(ch != '\n')
                    ch = fgetc(src);
                fputc(ch, dst);
                continue;
            }
            if (strcmp(directive, "#define") == 0) {
                char name[100];
                char value[100];
                int i = 0;

                while (ch != EOF && isspace(ch))
                    ch = fgetc(src);

                while (ch != EOF && (isalnum(ch) || ch == '_')) {
                    name[i++] = ch;
                    ch = fgetc(src);
                }
                name[i] = '\0';

                while (ch != EOF && isspace(ch))
                    ch = fgetc(src);

                i = 0;
                while (ch != EOF && ch != '\n') {
                    value[i++] = ch;
                    ch = fgetc(src);
                }
                value[i] = '\0';

                addMacro(name, value);
                fputc(ch, dst);
                continue;
            }

            fputs(directive, dst);
            if (ch != EOF)
                putc(ch, dst);
            continue;
        }

        if (isalpha(ch) || ch == '_') {
            tlen = 0;
            token[tlen++] = ch;

            while ((ch = fgetc(src)) != EOF && (isalnum(ch) || ch == '_'))
{
                token[tlen++] = ch;
            }
            token[tlen] = '\0';

            const char *val = getMacroValue(token);
            if (val)
                fputs(val, dst);
            else
                fputs(token, dst);

            if (ch != EOF)
                ungetc(ch, src);

            continue;
        }

        putc(ch, dst);
    }
}

void addMacro(const char *name, const char *value) {
    if (macroCount >= 100) return;
    strcpy(macros[macroCount].name, name);
    strcpy(macros[macroCount].value, value);
    macroCount++;
}

const char* getMacroValue(const char *name) {
    for (int i = 0; i < macroCount; i++) {
        if (strcmp(macros[i].name, name) == 0)
            return macros[i].value;
    }
    return NULL;
}

//Token Server

token getNextToken(FILE *src, int *row, int *col){
    token curr;
    memset(&curr, 0, sizeof(curr));

    int ch;
    ch = fgetc(src);

    while (ch != EOF) {

        if (isalpha(ch)) {
            curr = isKeyword(ch, src, row, col);
            if (curr.token_name[0]) return curr;

            curr = isIdentifier(ch, src, row, col);
            if (curr.token_name[0]) return curr;
        }

        else if (isdigit(ch)) {
            curr = isNumber(ch, src, row, col);
            return curr;
        }

        else if (ch == '"') {
            curr = isStringLiteral(ch, src, row, col);
            return curr;
        }

        else if (strchr("+-&|*/%<>!^?", ch)) {
            curr = isOperator(ch, src, row, col);
            return curr;
        }

        else {//Symbol
            curr.col = *col;
            curr.row = *row;
            curr.token_name[0] = ch;
            (*col)++;
            return curr;
        }
        ch = fgetc(src);
    }
    return curr;
}

//Token identifying

token isKeyword(int ch, FILE *src, int *row, int *col) {
    token curr;
    memset(&curr, 0, sizeof(curr));
    int c = 1;
    curr.col = *col;
    curr.row = *row;
    char word[50];
    int i = 0;
    word[i++] = (char)ch;
    while ((ch = fgetc(src)) != EOF) {
        c++;
        if (!isalpha(ch)) {
            fseek(src, -1, SEEK_CUR);
            c--;
            break;
        }
        if (i < (int)sizeof(word) - 1)
            word[i++] = ch;
    }

    word[i] = '\0';

    for (int k = 0; k < (int)(sizeof(keywords)/sizeof(keywords[0])); k++) {
        if (strcmp(word, keywords[k]) == 0) {
            strcpy(curr.token_name, word);
            *col += c;
            return curr;
        }
    }
    fseek(src, -c+1, SEEK_CUR);
    return curr;
}

token isIdentifier(int ch, FILE *src, int *row, int *col) {
    token curr;
    memset(&curr, 0, sizeof(curr));

    curr.col = *col;
    curr.row = *row;
    char word[50];
    word[0] = ch;
    (*col)++;
    int i = 1;
    while (ch != EOF) {
        ch = fgetc(src);
        (*col)++;
        if(ch != '_' && !isalnum(ch)){
            fseek(src, -1, SEEK_CUR);
            (*col)--;
            strcpy(curr.token_name, "id");
            return curr;
        }
        word[i++] = ch;
    }
    return curr;
}

token isOperator(int ch, FILE *src, int *row, int *col) {
    token curr;
    memset(&curr, 0, sizeof(curr));

    curr = isRelationalOperator(ch, src, row, col);
    if (curr.token_name[0]) return curr;

    curr = isArithmeticOperator(ch, src, row, col);
    if (curr.token_name[0]) return curr;

    curr = isLogicalOperator(ch, src, row, col);
    if (curr.token_name[0]) return curr;

    curr = isBitwiseOperator(ch, src, row, col);
    if (curr.token_name[0]) return curr;

    curr = isConditionalOperator(ch, row, col);
    if (curr.token_name[0]) return curr;

    curr = isAssignmentOperator(ch, src, row, col);
    if (curr.token_name[0]) return curr;

    return curr;
}

token isRelationalOperator(int ch, FILE *src, int *row, int *col) {
    token curr;
    memset(&curr, 0, sizeof(curr));
    if(ch != '=' && ch != '<' && ch != '>' && ch != '!')
        return curr;
    curr.col = *col;
    curr.row = *row;
    int prev = ch;
    ch = fgetc(src);

    if (ch == '='){
        strcpy(curr.token_name, "relOp");
        (*col) += 2;
        return curr;
    }
    else if (prev == '<' || prev == '>'){
        strcpy(curr.token_name, "relOp");
        (*col)++;
        return curr;
    }
    else
        fseek(src, -1, SEEK_CUR);

    return curr;
}

token isArithmeticOperator(int ch, FILE *src, int *row, int *col) {
    token curr;
    memset(&curr, 0, sizeof(curr));
    if(ch != '+' && ch != '-' && ch != '*' && ch != '/' && ch != '%')
        return curr;
    curr.col = *col;
    curr.row = *row;
    int prev = ch;
    ch = fgetc(src);

    if (prev == '+' && ch == '+' ||
        prev == '-' && ch == '-'){
        strcpy(curr.token_name, "ariOp");
        (*col) += 2;
        return curr;
    }
    else{
        strcpy(curr.token_name, "ariOp");
        (*col)++;
        fseek(src, -1, SEEK_CUR);
    }

    return curr;
}

token isLogicalOperator(int ch, FILE *src, int *row, int *col) {
    token curr;
    memset(&curr, 0, sizeof(curr));
    if(ch != '&' && ch != '|' && ch != '!')
        return curr;
    curr.col = *col;
    curr.row = *row;
    int prev = ch;
    ch = fgetc(src);

    if (prev != '!' && ch == prev){
        strcpy(curr.token_name, "logOp");
        (*col) += 2;
        return curr;
    }
    else if (ch == '!') {
        strcpy(curr.token_name, "logOp");
        (*col)++;

        fseek(src, -1, SEEK_CUR);

        return curr;
    }
    else
        fseek(src, -1, SEEK_CUR);

    return curr;
}

token isBitwiseOperator(int ch, FILE *src, int *row, int *col) {
    token curr;
    memset(&curr, 0, sizeof(curr));
    if(ch != '<' && ch != '>' && ch != '|' && ch != '&' && ch != '^' && ch
!= '~')
        return curr;
    curr.col = *col;
    curr.row = *row;
    int prev = ch;
    ch = fgetc(src);

    if (prev == '^' || prev == '~') {
        strcpy(curr.token_name, "bitOp");
        (*col)++;
        return curr;
    }
    else if ((ch == '<' || ch == '>') && prev == ch){
        strcpy(curr.token_name, "bitOp");
        (*col) += 2;
        return curr;
    }
    if ((prev == '&' || prev == '|') && prev != ch) {
        strcpy(curr.token_name, "bitOp");
        (*col)++;

        fseek(src, -1, SEEK_CUR);
        return curr;
    }
    else
        fseek(src, -1, SEEK_CUR);

    return curr;
}

token isConditionalOperator(int ch, int *row, int *col) {
    token curr;
    memset(&curr, 0, sizeof(curr));

    if (ch == '?' || ch == ':') {
        strcpy(curr.token_name, "condOp");
        curr.row = *row;
        curr.col = *col;
        (*col)++;
    }

    return curr;
}

token isAssignmentOperator(int ch, FILE *src, int *row, int *col) {
    token curr;
    memset(&curr, 0, sizeof(curr));

    if (ch != '=' && ch != '+' && ch != '-' && ch != '*' &&
        ch != '/' && ch != '%' && ch != '<' && ch != '>' &&
        ch != '&' && ch != '|' && ch != '^')
        return curr;

    curr.col = *col;
    curr.row = *row;

    int next = fgetc(src);
    if (ch == '=' && next != '=') {
        strcpy(curr.token_name, "assignOp");
        (*col)++;

        if (next != EOF)
            fseek(src, -1, SEEK_CUR);

        return curr;
    }

    if (next == '=') {
        strcpy(curr.token_name, "assignOp");
        (*col) += 2;
        return curr;
    }

    if ((ch == '<' || ch == '>') && next == ch) {
        int next2 = fgetc(src);

        if (next2 == '=') {
            strcpy(curr.token_name, "assignOp");
            (*col) += 3;
            return curr;
        }
        fseek(src, -2, SEEK_CUR);

        return curr;
    }

    if (next != EOF)
        fseek(src, -1, SEEK_CUR);

    return curr;
}

token isStringLiteral(int ch, FILE *src, int *row, int *col) {
    token curr;
    curr.col = *col;
    curr.row = *row;
    strcpy(curr.token_name, "stringLit");
    ch = fgetc(src);
    (*col)++;
    while (ch != '"') {
        ch = fgetc(src);
        if (ch == '\n') {
            *col = 1;
            (*row)++;
        }
        else (*col)++;
    }
    (*col)++;
    return curr;
}
```

```
token isNumber(int ch, FILE *src, int *row, int *col) {
    token curr;
    memset(&curr, 0, sizeof(curr));
    curr.col = *col;
    curr.row = *row;
    int i = 0;

    int state = 1;
    int prev;

    while (state != 4) {
        prev = ch;

        if (ch != EOF) {
            ch = fgetc(src);
            (*col)++;
        }

        if (state == 1) {
            curr.token_name[i++] = prev;
            if (isdigit(ch)) continue;
            else if (ch == 'e' || ch =='E') state = 3;
            else if (ch == '.') state = 2;
            else state = 4;
        }
        else if (state == 2) {
            curr.token_name[i++] = prev;
            if (isdigit(ch)) state = 5;
            else state = 4;
        }
        else if (state == 3) {
            curr.token_name[i++] = prev;
            if (isdigit(ch)) state = 6;
            else if (ch == '+' || ch == '-') state = 7;
            else state = 4;
        }
        else if (state == 5) {
            curr.token_name[i++] = prev;
            if (isdigit(ch)) continue;
            if (ch == 'e' || ch =='E') state = 3;
            else state = 4;
        }
        else if (state == 6) {
            curr.token_name[i++] = prev;
            if (isdigit(ch)) continue;
            else state = 4;
        }
        else {
            curr.token_name[i++] = prev;
            if (isdigit(ch)) state = 6;
            else state = 4;
        }
    }
    fseek(src, -1, SEEK_CUR);

    curr.token_name[i] = '\0';
    return curr;
}

void PrintToken(token t, FILE *dst) {
    fprintf(dst, "<%s,%d,%d>", t.token_name, t.row, t.col);
}

void copyFile(FILE *src, FILE *dst) {
    int ch;
    while((ch = fgetc(src)) != EOF) {
        putc(ch, dst);
    }
}

void postprocess(FILE *src, FILE *dst) {
    int ch;
    int newLine = 1;
    while((ch = fgetc(src)) != EOF){
        if(newLine && ch == '\n') continue;
        fputc(ch, dst);

        if (ch == '\n')
            newLine = 1;
        else
            newLine = 0;
    }
}
```

Original File :

```
#include <stdio.h>
#include <stdlib.h>
#define PI 44e-5

int main(){
                     if(PI) printf("Nothing here\n");

                     //bcsjhkdvbdsjkh
                     /*vhjskdvk
                     bdsfhgbdf
                     bdfgbnfgdn*/
    printf          ("Demo fi                 le\n");
        return 0;
}
```

Output :

```
<int,5,1><id,5,5><(,5,9><),5,10><{,5,11>
<if,6,22><(,6,24><44e-5,6,25><),6,30><id,6,32><(,6,38><stringLit,6,39>
<),6,55><;,6,56>
<id,12,5><(,12,21><stringLit,12,22><),12,52><;,12,53>
<return,13,9><0,13,16><;,13,17>
<},14,1>
```