Lab-4 CONSTRUCTION OF SYMBOL TABLE

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

Q) Make Symbol Table.

symbolTable.c

```
#include "lexAnalyszer.h"
#include "stack.h"

#define TABLE_SIZE 100

typedef struct symbolTableEntry{
    token token;
    struct symbolTableEntry*nextTokenEntry;
    struct symbolTable *local;
}symbolTableEntry;

typedef struct symbolTable {
    symbolTableEntry*entry[TABLE_SIZE];
} symbolTable;

symbolTable globalTable = {0};

symbolTable *AddFunctionEntryInGlobalTableAndMakeItsLocalTable (token currToken,
stack *scopeStack);
symbolTableEntry *MakeTableEntryAndAddInTable(token t, symbolTable *st);
symbolTableEntry*MakeSymbol(token t);
void insertToken(symbolTable*st, symbolTableEntry *entry);
int searchToken(symbolTable*st, symbolTableEntry *entry);
int isSame(symbolTableEntry a, symbolTableEntry b);
int hash(symbolTableEntry*entry);
void printLocalSymbolTable(symbolTable*st, FILE *dst);
void printGlobalSymbolTable(symbolTable *st, FILE *dst);

int main() {
    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 currToken;
    token prevToken = {0};
    symbolTable *currLocalTable = &globalTable;
    stack scopeStack = {.top = -1};

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

            if (strcmp(currToken.tokenName, "symbol") == 0){
                if (strcmp(currToken.tokenValue, "{") == 0) {
                    pushStack(&scopeStack, currToken.tokenValue[0]);
                }
                else if (strcmp(currToken.tokenValue, "}") == 0) {
                    popStack(&scopeStack);
                    if (isEmptyStack(&scopeStack))
                        currLocalTable = &globalTable;
                }
                continue;
            }
            if (strcmp(currToken.tokenName, "id") == 0) {
                long pos = ftell(tmp);
                token nextToken = getNextToken(tmp, &row, &col);
                if (nextToken.tokenValue[0] == '(' && strcmp(prevToken.tokenName,
"keyword") == 0){
                    strcpy(currToken.tokenName, "Func");
                    currLocalTable =
AddFunctionEntryInGlobalTableAndMakeItsLocalTable (currToken, &scopeStack);
                }
                else {
                    fseek(tmp, pos, SEEK_SET);
                    MakeTableEntryAndAddInTable(currToken, currLocalTable);
                }
            }
        }
        prevToken = currToken;
    }

    printGlobalSymbolTable(&globalTable, dst);
```

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

symbolTable *AddFunctionEntryInGlobalTableAndMakeItsLocalTable (token currToken,
stack *scopeStack) {
    symbolTableEntry *currGlobalEntry = MakeTableEntryAndAddInTable (currToken,
&globalTable);
    currGlobalEntry->local = (symbolTable*)calloc(1, sizeof(symbolTable));
    MakeTableEntryAndAddInTable(currToken, currGlobalEntry->local);
    return currGlobalEntry->local;
}

symbolTableEntry *MakeTableEntryAndAddInTable(token t, symbolTable *st) {
    symbolTableEntry*entry = MakeSymbol(t);

    if (!searchToken(st, entry)) {
        insertToken(st, entry);
    } else {
        free(entry);
    }
    return entry;
}

symbolTableEntry*MakeSymbol(token t) {
    symbolTableEntry*entry = malloc(sizeof(symbolTableEntry));
    memset(entry, 0, sizeof(symbolTableEntry));
    entry->token = t;
    entry->nextTokenEntry = NULL;
    return entry;
}

void insertToken(symbolTable*st, symbolTableEntry*entry) {
    int idx = hash(entry);

    symbolTableEntry *currEntry = st->entry[idx];

    if (currEntry == NULL) {
        st->entry[idx] = entry;
        return;
    }

    while (currEntry->nextTokenEntry != NULL) {
        currEntry = currEntry->nextTokenEntry;
    }

    currEntry->nextTokenEntry = entry;
}

int searchToken(symbolTable*st, symbolTableEntry*entry) {
    int idx = hash(entry);
    symbolTableEntry*currToken = st->entry[idx];
```

```
    while (currToken != NULL) {
        if (isSame(*entry, *currToken))
            return 1;
        currToken = currToken->nextTokenEntry;
    }

    return 0;
}

int isSame(symbolTableEntry a, symbolTableEntry b) {
    if (strcmp(a.token.tokenValue, b.token.tokenValue) != 0) return 0;
    if (strcmp(a.token.tokenType, b.token.tokenType) != 0) return 0;
    if (strcmp(a.token.tokenReturnType, b.token.tokenReturnType) != 0) return 0;
    if (a.token.size != b.token.size) return 0;
    return 1;
}

int hash(symbolTableEntry*entry) {
    int h = 0;
    h += strlen(entry->token.tokenName);
    h += strlen(entry->token.tokenType);
    h += entry->token.size;
    return h % TABLE_SIZE;
}

void printGlobalSymbolTable(symbolTable *st, FILE *dst) {
    fprintf(dst, "%-6s %-10s %-10s %-6s %-12s %-18s\n",
            "", "Name", "Type", "Size", "Return Type", "Ptr to Local Table");

    int idx = 1;

    for (int i = 0; i < TABLE_SIZE; i++) {
        symbolTableEntry *entry = st->entry[i];

        while (entry != NULL) {
            char *name = entry->token.tokenValue[0] ? entry->token.tokenValue : "-
";
            char *type = strcmp(entry->token.tokenName, "id") != 0 ? entry-
>token.tokenName[0] ? entry->token.tokenName : "-" : entry->token.tokenType;
            char *ret  = entry->token.tokenReturnType[0] ? entry-
>token.tokenReturnType : "-";

            char sizeStr[20];
            if (entry->token.size != 0)
                snprintf(sizeStr, sizeof(sizeStr), "%d", entry->token.size);
            else
                strcpy(sizeStr, "-");

            char localPtrStr[20];
            if (entry->local != NULL)
                snprintf(localPtrStr, sizeof(localPtrStr), "%p", (void *)entry-
>local);
            else
```

```
                strcpy(localPtrStr, "-");

            fprintf(dst, "%-6d %-10s %-10s %-6s %-12s %-18s\n",
                    idx++,
                    name,
                    type,
                    sizeStr,
                    ret,
                    localPtrStr);

            entry = entry->nextTokenEntry;
        }
    }
    fprintf(dst, "\n");
    idx = 1;
    for (int i = 0; i < TABLE_SIZE; i++) {
        symbolTableEntry *entry = st->entry[i];

        while (entry != NULL) {
            if(entry->local != NULL) {
                fprintf(dst, "%s's Local Table:\n", entry->token.tokenValue);
                printLocalSymbolTable(entry->local, dst);
            }
            entry = entry->nextTokenEntry;
        }
    }
}


void printLocalSymbolTable(symbolTable*st, FILE *dst) {
    fprintf(dst, "%-6s %-10s %-10s %-6s %-12s\n",
            "", "Name", "Type", "Size", "Return Type");
    int idx = 1;
    for (int i = 0; i < TABLE_SIZE; i++) {
        symbolTableEntry*entry = st->entry[i];

        while (entry != NULL) {
            char *name = entry->token.tokenValue[0] ? entry->token.tokenValue : "-
";
            char *type = entry->token.tokenType[0] ? entry->token.tokenType :
entry->token.tokenName;
            char *ret  = entry->token.tokenReturnType[0] ? entry-
>token.tokenReturnType : "-";
            char sizeStr[20];

            if (entry->token.size != 0) {
                snprintf(sizeStr, sizeof(sizeStr), "%d", entry->token.size);
            } else {
                strcpy(sizeStr, "-");
            }

            fprintf(dst, "%-6d %-10s %-10s %-6s %-12s\n",
                    idx++,
                    name,
```

```
                    type,
                    sizeStr,
                    ret);

            entry = entry->nextTokenEntry;
        }
    }

}
```

helper files

lexeranalyzer.h

```
#ifndef LEXANALYSZER_H
#define LEXANALYSZER_H

#include "preprocessing.h"

//structs
typedef struct token{
    char tokenName [50];
    char tokenValue [50];
    char tokenType [50];
    char tokenReturnType [50];
    int row,col,size;
}token;

//global variables
static 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"
};

static const char *types[] = {
    "void", "char", "short", "int", "long",
    "float", "double", "signed", "unsigned",
    "_Bool", "_Complex"
};

static char type[50] = "";

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

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

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

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

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

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

//Helpers
static void PrintToken(token t, FILE *dst);
static void copyFile(FILE *src, FILE *dst);
static void postprocess(FILE *src, FILE *dst);
static int findSizeOf( char *word );
static int isType(char *s);

//Token Server
static 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.tokenName[0]) return curr;

            curr = isIdentifier(ch, src, row, col);
            if (curr.tokenName[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.tokenValue[0] = ch;
            strcpy(curr.tokenName, "symbol");
            (*col)++;
            if (ch == ';' || ch == '{')
                type[0] = '\0';
            return curr;
        }
        ch = fgetc(src);
    }
    return curr;
}

//Token identifying

static 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';
    if ( isType ( word ))
        strcpy( type, word );

    for (int k = 0; k < (int)(sizeof(keywords)/sizeof(keywords[0])); k++) {
        if (strcmp(word, keywords[k]) == 0) {
            strcpy(curr.tokenValue, word);
            strcpy(curr.tokenName, "keyword");
            *col += c;
            return curr;
        }
    }
```

```
    fseek(src, -c+1, SEEK_CUR);
    return curr;
}

static 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.tokenName, "id");

            word[i] = 0;
            strcpy ( curr.tokenValue, word );
            if ( isType ( word ) )
                strcpy( type, word);
            else {
                if ( strcmp ( word, "printf" ) == 0 ||
                        strcmp ( word, "scanf" ) == 0 )
                    return curr;
            ch = fgetc( src );
            if( ch == '(' ) {
                strcpy ( curr.tokenReturnType, type );
                fseek ( src, -1, SEEK_CUR );
            }
            else {
                strcpy ( curr.tokenType, type );
            }
            curr.size = findSizeOf( type );

            }
            return curr;
        }
        word[i++] = ch;
    }

    return curr;
}

static 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.tokenName[0]) return curr;

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

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

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

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

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

    return curr;
}

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

    return curr;
}

static 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.tokenName, "ariOp");
        (*col) += 2;
        return curr;
    }
    else{
        strcpy(curr.tokenName, "ariOp");
        (*col)++;
        fseek(src, -1, SEEK_CUR);
    }

    return curr;
}

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

        fseek(src, -1, SEEK_CUR);

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

    return curr;
}

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

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

    return curr;
}

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

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

    return curr;
}

static 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.tokenName, "assignOp");
            (*col)++;

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

            return curr;
        }

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

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

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

            return curr;
        }

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

        return curr;
    }

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

    static 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.tokenName[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.tokenName[i++] = prev;
            if (isdigit(ch)) state = 5;
            else state = 4;
        }
        else if (state == 3) {
            curr.tokenName[i++] = prev;
            if (isdigit(ch)) state = 6;
            else if (ch == '+' || ch == '-') state = 7;
            else state = 4;
        }
        else if (state == 5) {
            curr.tokenName[i++] = prev;
            if (isdigit(ch)) continue;
            if (ch == 'e' || ch =='E') state = 3;
            else state = 4;
        }
        else if (state == 6) {
            curr.tokenName[i++] = prev;
            if (isdigit(ch)) continue;
            else state = 4;
        }
        else {
            curr.tokenName[i++] = prev;
            if (isdigit(ch)) state = 6;
            else state = 4;
        }
    }

    fseek(src, -1, SEEK_CUR);
```

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

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

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

static 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;
    }
}

static int isType(char *s) {
    int n = sizeof(types) / sizeof(types[0]);

    for (int i = 0; i < n; i++) {
        if (strcmp(s, types[i]) == 0)
            return 1;
    }
    return 0;
}

static int findSizeOf(char *word) {

    if (strcmp(word, "int") == 0) return sizeof(int);
    if (strcmp(word, "char") == 0) return sizeof(char);
    if (strcmp(word, "void") == 0) return 0;
    if (strcmp(word, "short") == 0) return sizeof(short);
    if (strcmp(word, "long") == 0) return sizeof(long);
    if (strcmp(word, "float") == 0) return sizeof(float);
    if (strcmp(word, "double") == 0) return sizeof(double);
    if (strcmp(word, "signed") == 0) return sizeof(int);
    if (strcmp(word, "unsigned") == 0) return sizeof(unsigned int);
    if (strcmp(word, "_Bool") == 0) return sizeof(_Bool);

    return 0;
```

```
}

#endif
```

stack.h

```
#ifndef STACK_H
#define STACK_H

#define STACK_SIZE 100
#include "lexAnalyszer.h"

typedef struct stack{
    char arr[STACK_SIZE];
    int top;
}stack;

// prototypes
static void pushStack(stack *s, char c);
static char popStack(stack *s);
static int isEmptyStack(stack *s);

static void pushStack(stack *s, char c) {
    if (s->top == STACK_SIZE - 1)
        return;

    s->arr[++(s->top)] = c;
}

static char popStack(stack *s) {
    if (s->top == -1)
        return '\0';

    return s->arr[(s->top)--];
}

static int isEmptyStack(stack *s) {
    return s->top == -1;
}

#endif
```

preprocessing.h

```
#ifndef PREPROCESSING_H
#define PREPROCESSING_H

#include <stdio.h>
```

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

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

static Macro macros[100];
static int macroCount = 0;

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

static 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);
    }
}

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

static 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;
}

#endif
```

Input File :

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


int ahbs;
```

```c
char func1(int ch, char* s){
    int c = 0;
    int x = 9;
    return 4;
}

int main(){
                    if(PI) printf("Nothing here\n");
                    int a = 5;
                    char b = 7;
                    int v,k;

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

Output :

```
       Name       Type       Size   Return Type  Ptr to Local Table
1      func1      Func       1      char         0x3b235e50
2      main       Func       4      int          0x3b236720
3      ahbs       int        4      -            -

func1's Local Table:
       Name       Type       Size   Return Type
1      func1      Func       1      char
2      s          char       1      -
3      ch         int        4      -
4      c          int        4      -
5      x          int        4      -
main's Local Table:
       Name       Type       Size   Return Type
1      printf     id         -      -
2      b          char       1      -
3      main       Func       4      int
4      a          int        4      -
5      v          int        4      -
6      k          int        4      -
```