Name : Vivek Singh
RegNumber : 230905408
Roll Number: A50

Lab 3: CONSTRUCTION OF TOKEN GENERATOR

Q1. Write functions to identify the following tokens.
a. Arithmetic, relational and logical operators.
b. Special symbols, keywords, numerical constants, string literals and identifiers.

Q2. 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.

Helper functions:(lib.c)

```c
/* lib.c */
#include <stdio.h>
#include <ctype.h>
#include <string.h>

/* ================= TOKEN STRUCT ================= */

typedef struct {
char lexeme[50];
char type[30];
int row;
int col;
} Token;

/* ================= KEYWORD CHECK ================= */

int isKeyword(char *w) {
char *k[] = {
"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"
};

int l = 0, r = 31, m, c;

while (l <= r) {
m = l + (r - l) / 2;
c = strcmp(w, k[m]);
```

```
if (c == 0) return 1;
if (c < 0) r = m - 1;
else l = m + 1;
}
return 0;
}

/* ================ COMMENT REMOVAL ================ */

void removeComments(FILE *src, FILE *dst) {
char c, next;

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

/* Possible comment start */
if (c == '/') {
next = fgetc(src);

/* Single-line comment */
if (next == '/') {
while ((c = fgetc(src)) != '\n' && c != EOF);
}

/* Multi-line comment */
else if (next == '*') {
char prev = 0;
while ((c = fgetc(src)) != EOF) {
if (prev == '*' && c == '/')
break;
prev = c;
}
}

/* Not a comment */
else {
fputc(c, dst);
fseek(src, -1, SEEK_CUR);
}
}

/* Normal character */
else {
fputc(c, dst);
}
}
}



/* ================= PREPROCESSOR REMOVAL ================= */
```

```
void removePreprocessor(FILE *src, FILE *dst) {
char c;
while ((c = fgetc(src)) != EOF) {
if (c == '#') {
while (c != '\n' && c != EOF)
c = fgetc(src);
} else {
fputc(c, dst);
}
}
}

/* ================= TOKEN HELPERS ================= */

int isArithmetic(char c) {
return (c=='+'||c=='-'||c=='*'||c=='/'||c=='%');
}

int isSpecialSymbol(char c) {
char s[] = ";,(){}[]";
for (int i = 0; s[i]; i++)
if (c == s[i]) return 1;
return 0;
}

int isIdentifier(char *w) {
if (!isalpha(w[0]) && w[0] != '_') return 0;
for (int i = 1; w[i]; i++)
if (!isalnum(w[i]) && w[i] != '_') return 0;
return 1;
}

int isNumber(char *w) {
int dot = 0;
for (int i = 0; w[i]; i++) {
if (w[i] == '.') dot++;
else if (!isdigit(w[i])) return 0;
}
return dot <= 1;
}

/* ================= GET NEXT TOKEN ================= */

Token getNextToken(FILE *fp) {
Token t;
char c, buffer[50];
int i = 0;
static int row = 1, col = 0;
```

```
t.lexeme[0] = '\0';

while ((c = fgetc(fp)) != EOF) {
col++;

if (c == '\n') {
row++;
col = 0;
continue;
}

if (isspace(c)) continue;

t.row = row;
t.col = col;

/* String literal (ignored content) */
if (c == '"') {
while ((c = fgetc(fp)) != '"' && c != EOF);
strcpy(t.lexeme, "\"...\"");
strcpy(t.type, "STRING_LITERAL");
return t;
}

/* Identifier / Keyword */
if (isalpha(c) || c == '_') {
buffer[i++] = c;
while (isalnum(c = fgetc(fp)) || c == '_') {
buffer[i++] = c;
col++;
}
buffer[i] = '\0';
fseek(fp, -1, SEEK_CUR);

strcpy(t.lexeme, buffer);
strcpy(t.type, isKeyword(buffer) ? "KEYWORD" : "IDENTIFIER");
return t;
}

/* Number */
if (isdigit(c)) {
buffer[i++] = c;
while (isdigit(c = fgetc(fp)) || c == '.') {
buffer[i++] = c;
col++;
}
buffer[i] = '\0';
fseek(fp, -1, SEEK_CUR);

strcpy(t.lexeme, buffer);
```

```
strcpy(t.type, "NUMBER");
return t;
}

/* Arithmetic operator */
if (isArithmetic(c)) {
t.lexeme[0] = c;
t.lexeme[1] = '\0';
strcpy(t.type, "ARITHMETIC_OP");
return t;
}

/* Relational operator */
if (c=='<'||c=='>'||c=='='||c=='!') {
char n = fgetc(fp);
buffer[0] = c;
buffer[1] = n;
buffer[2] = '\0';

if (!strcmp(buffer,"<=")||!strcmp(buffer,">=")||
!strcmp(buffer,"==")||!strcmp(buffer,"!=")) {
strcpy(t.lexeme, buffer);
} else {
fseek(fp, -1, SEEK_CUR);
t.lexeme[0] = c;
t.lexeme[1] = '\0';
}
strcpy(t.type, "RELATIONAL_OP");
return t;
}

/* Special symbol */
if (isSpecialSymbol(c)) {
t.lexeme[0] = c;
t.lexeme[1] = '\0';
strcpy(t.type, "SPECIAL_SYMBOL");
return t;
}
}

strcpy(t.type, "EOF");
return t;
}
```

Main Analyzer(analyser.c):

```
/* analyser.c */
#include <stdio.h>
#include <string.h>

/* ===== Must match Token definition in lib.c ===== */
typedef struct {
char lexeme[50];
char type[30];
int row;
int col;
} Token;

/* ===== Function declarations from lib.c ===== */
void removeComments(FILE *, FILE *);
void removePreprocessor(FILE *, FILE *);
Token getNextToken(FILE *);

/* ===== Token storage ===== */
Token tokens[1000];
int tokenCount = 0;

/* ===== Seen tables for unique tokens ===== */
char seenKeywords[50][30];
int seenKeywordCount = 0;

char seenOperators[50][30];
int seenOperatorCount = 0;

char seenSymbols[50][30];
int seenSymbolCount = 0;

/* ===== Helper: check if already seen ===== */
int alreadySeen(char table[][30], int count, char *lexeme) {
for (int i = 0; i < count; i++) {
if (strcmp(table[i], lexeme) == 0)
return 1;
}
return 0;
}

/* ===== Helper: mark as seen ===== */
void markSeen(char table[][30], int *count, char *lexeme) {
strcpy(table[*count], lexeme);
(*count)++;
}

int main() {
```

```
FILE *src, *noComments, *clean;
Token t;

/* ===== Open files ===== */
src = fopen("input.c", "r");
noComments = fopen("nocomments.c", "w");
clean = fopen("clean.c", "w");

if (!src || !noComments || !clean) {
printf("Error opening files.\n");
return 1;
}

/* ===== Step 1: Remove comments ===== */
removeComments(src, noComments);
fclose(src);
fclose(noComments);

/* ===== Step 2: Remove preprocessor directives ===== */
noComments = fopen("nocomments.c", "r");
removePreprocessor(noComments, clean);
fclose(noComments);
fclose(clean);

/* ===== Step 3–5: Lexical analysis ===== */
clean = fopen("clean.c", "r");
if (!clean) {
printf("Error opening cleaned file.\n");
return 1;
}

while (1) {
t = getNextToken(clean);
if (strcmp(t.type, "EOF") == 0)
break;

/* ===== Store unique KEYWORDS ===== */
if (strcmp(t.type, "KEYWORD") == 0) {
if (!alreadySeen(seenKeywords, seenKeywordCount, t.lexeme)) {
tokens[tokenCount++] = t;
markSeen(seenKeywords, &seenKeywordCount, t.lexeme);
}
}

/* ===== Store unique OPERATORS ===== */
else if (strstr(t.type, "OP") != NULL) {
if (!alreadySeen(seenOperators, seenOperatorCount, t.lexeme)) {
tokens[tokenCount++] = t;
markSeen(seenOperators, &seenOperatorCount, t.lexeme);
}
```

```
}

/* ===== Store unique SPECIAL SYMBOLS ===== */
else if (strcmp(t.type, "SPECIAL_SYMBOL") == 0) {
if (!alreadySeen(seenSymbols, seenSymbolCount, t.lexeme)) {
tokens[tokenCount++] = t;
markSeen(seenSymbols, &seenSymbolCount, t.lexeme);
}
}

/* ===== Store all other tokens ===== */
else {
tokens[tokenCount++] = t;
}
}

fclose(clean);

/* ===== Output ===== */
printf("\nLEXEME\t\tTYPE\t\t\tROW\tCOL\n");
printf("------------------------------------------------------\n");

for (int i = 0; i < tokenCount; i++) {
printf("%-12s %-24s \t%d\t%d\n",
tokens[i].lexeme,
tokens[i].type,
tokens[i].row,
tokens[i].col);
}

return 0;
}
```

INPUT:(input.c)

```c
#include <stdio.h>
#include <ctype.h>
//single line comment
/*multiple line comment
*cuifhbv
*kjvbkiutrjgbn
*/
int main() {
        int a, b, sum=0;
        sum= a+b;
        printf("Sum= %d", sum);
        return 0;
}
```

Output

```
CD_A2@CL3-15:~/Documents/230905408/lab3$ ./lexer

LEXEME          TYPE                        ROW     COL
-------------------------------------------------------------
int             KEYWORD                     5       1
main            IDENTIFIER                  5       5
(               SPECIAL_SYMBOL              5       9
)               SPECIAL_SYMBOL              5       10
{               SPECIAL_SYMBOL              5       12
a               IDENTIFIER                  6       9
,               SPECIAL_SYMBOL              6       10
b               IDENTIFIER                  6       12
sum             IDENTIFIER                  6       15
=               RELATIONAL_OP               6       18
0               NUMBER                      6       19
;               SPECIAL_SYMBOL              6       20
sum             IDENTIFIER                  7       5
a               IDENTIFIER                  7       10
+               ARITHMETIC_OP               7       11
b               IDENTIFIER                  7       12
printf          IDENTIFIER                  8       5
"..."           STRING_LITERAL              8       12
sum             IDENTIFIER                  8       15
return          KEYWORD                     9       5
0               NUMBER                      9       12
}               SPECIAL_SYMBOL              10      1
CD_A2@CL3-15:~/Documents/230905408/lab3$
```