Lab 03 – Construction of Token Generator

Name: Pranamya G Kulal
Class: CSE A1
Reg no: 220905018
Roll no: 8

Q1) 1. 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.

Code: la.h

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

struct token {
    char lexeme[64];
    int row, col;
    char type[30];
};

static int row = 1, col = 1;
char specialsymbols[] = {'?', ';', ':', ',', '(', ')', '{', '}', '.'};
char *Keywords[] = {"for", "if", "else", "while", "do", "break", "continue", "return", "int", "double",
"float", "char", "long", "short", "sizeof", "typedef", "switch", "case", "struct", "const", "void",
"exit"};
char arithmeticsymbols[] = {'*','+','-','/', '%'};

int isKeyword(char *str) {
    for (int i = 0; i < sizeof(Keywords) / sizeof(char *); i++) {
        if (strcmp(str, Keywords[i]) == 0)
            return 1;
    }
    return 0;
}

int charBelongsTo(int c, char *arr, int len) {
    for (int i = 0; i < len; i++) {
        if (c == arr[i])
            return 1;
    }
    return 0;
}

void fillToken(struct token *tkn, char c, int row, int col, char *type) {
    tkn->row = row;
    tkn->col = col;
    strcpy(tkn->type, type);
    tkn->lexeme[0] = c;
    tkn->lexeme[1] = '\0';
}

void newLine() {
    ++row;
    col = 1;
}

struct token getNextToken(FILE *fin) {
    int c, d;
    struct token tkn = {.row = -1};
    int gotToken = 0;

    while (!gotToken && (c = getc(fin)) != EOF) {
        // SKIP COMMENTS
        if (c == '/') {
            d = getc(fin);
            if (d == '/') {  // Skip single-line comments
                while ((c = getc(fin)) != EOF && c != '\n') ++col;
                if (c == '\n') newLine();
                continue;
            } else if (d == '*') {  // Skip multi-line comments
                do {
                    if (c == '\n') newLine();
                    while ((c = getc(fin)) != EOF && c != '*') {
                        if (c == '\n') newLine();
                    }
                    if (c == '*') d = getc(fin);
                } while (c != EOF && d != '/');
                continue;
            } else {
                fseek(fin, -1, SEEK_CUR);  // Not a comment
            }
        }

        // Handle preprocessor directives (lines starting with '#')
        if (c == '#') {
            tkn.row = row;
            tkn.col = col++;
            tkn.lexeme[0] = '#';
            int k = 1;
            while ((c = getc(fin)) != '\n' && !isspace(c)) {
                tkn.lexeme[k++] = c;
                col++;
            }
            tkn.lexeme[k] = '\0';  // Null-terminate the lexeme

            // Process #include directive
            if (strcmp(tkn.lexeme, "#include") == 0) {
                c = getc(fin); ++col;
                if (c == '<' || c == '"') {
                    while ((c = getc(fin)) != EOF && c != (c == '<' ? '>' : '"')) ++col;
                    if (c != EOF) {
                        while ((c = getc(fin)) != '\n') ++col;  // Skip the rest of the line
                        newLine();
                    }
                    continue;
                } else {
                    // Invalid #include
                    strcpy(tkn.type, "InvalidDirective");
                    strcat(tkn.lexeme, "<invalid>");
                    while ((c = getc(fin)) != '\n') {
                        strncat(tkn.lexeme, (char*)&c, 1);
                        ++col;
                    }
                    newLine();
                    gotToken = 1;
                }
            }
        }

        // Process special symbols
        if (charBelongsTo(c, specialsymbols, sizeof(specialsymbols) / sizeof(char))) {
            fillToken(&tkn, c, row, col, (char[]){c, '\0'});
            gotToken = 1;
            ++col;
        }

        // Process arithmetic operators
        else if (charBelongsTo(c, arithmeticsymbols, sizeof(arithmeticsymbols) / sizeof(char))) {
            d = getc(fin);
            if (d == '=' || (c == '+' || c == '-') && d == c) {
                fillToken(&tkn, c, row, col, (char[]){c, c == '=' ? '=' : '\0', '\0'});
                col += 2;
            } else {
                fillToken(&tkn, c, row, col, (char[]){c, '\0'});
                ++col;
                fseek(fin, -1, SEEK_CUR);
            }
            gotToken = 1;
        }

        // Process relational operators
        else if (c == '=' || c == '<' || c == '>' || c == '!') {
            d = getc(fin);
            if (d == '=') {
                fillToken(&tkn, c, row, col, (char[]){c, '=', '\0'});
                col += 2;
            } else {
                fillToken(&tkn, c, row, col, (char[]){c, '\0'});
                ++col;
                fseek(fin, -1, SEEK_CUR);
            }
            gotToken = 1;
        }

        // Process numbers
        else if (isdigit(c)) {
            tkn.row = row;
            tkn.col = col++;
            tkn.lexeme[0] = c;
            int k = 1;
            while ((c = getc(fin)) != EOF && isdigit(c)) {
                tkn.lexeme[k++] = c;
                ++col;
            }
            tkn.lexeme[k] = '\0';
            strcpy(tkn.type, "Number");
            gotToken = 1;
            fseek(fin, -1, SEEK_CUR);
        }

        // Discard whitespaces
        else if (isspace(c)) {
            ++col;
        }

        // Process keywords and identifiers
        else if (isalpha(c) || c == '_') {
            tkn.row = row;
            tkn.col = col++;
            tkn.lexeme[0] = c;
            int k = 1;
            while ((c = getc(fin)) != EOF && isalnum(c)) {
                tkn.lexeme[k++] = c;
                ++col;
            }
            tkn.lexeme[k] = '\0';
            strcpy(tkn.type, isKeyword(tkn.lexeme) ? "Keyword" : "Identifier");
            gotToken = 1;
            fseek(fin, -1, SEEK_CUR);
        }

        // Process String Literals
        else if (c == '"') {
            tkn.row = row;
            tkn.col = col;
```

```
            strcpy(tkn.type, "StringLiteral");
            int k = 1;
            tkn.lexeme[0] = '"';
            while ((c = getc(fin)) != EOF && c != '"') {
                tkn.lexeme[k++] = c;
                ++col;
            }
            tkn.lexeme[k] = '"';
            gotToken = 1;
        }

        else {
            ++col;
        }
    }
    return tkn;
}
```

Code: parser.c

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

int main() {
    FILE *fin = fopen("sampleread.c", "r");
    if (!fin) {
        printf("Error! File cannot be opened!\n");
        return 0;
    }
    struct token tkn;
    int count = 1;
    while ((tkn = getNextToken(fin)).row != -1)
        printf( "%d. < %s , %d , %d, %s>\n", count++, tkn.type, tkn.row, tkn.col,tkn.lexeme);
    fclose(fin);
}
```

Input file: sampleread.c

```
#include<stdio.h>

int main(){
    int num = 18;
    if(num == 18) printf("age equals 18");
    else if(num <= 18) printf("age less than 18");
    return 0;
}
```

Terminal output

```
CD_LAB_A1@debianpc-02:~/Desktop/220905018/Lab3$ gcc -o parser parser.c
CD_LAB_A1@debianpc-02:~/Desktop/220905018/Lab3$ ./parser
1. < Keyword , 1 , 58, int>
```

```
2. < Identifier , 1 , 62, main>
3. < ( , 1 , 66, (>
4. < ) , 1 , 67, )>
5. < { , 1 , 68, {>
6. < Keyword , 1 , 74, char>
7. < Identifier , 1 , 79, c>
8. < , , 1 , 80, ,>
9. < Identifier , 1 , 82, buf>
10. < Number , 1 , 86, 100>
11. < ; , 1 , 90, ;>
12. < Identifier , 1 , 96, FILE>
13. < * , 1 , 101, *>
14. < Identifier , 1 , 102, fptr>
15. < = , 1 , 107, =>
16. < Identifier , 1 , 109, fopen>
17. < ( , 1 , 114, (>
18. < StringLiteral , 1 , 115, "sampleread.c">
19. < , , 1 , 127, ,>
20. < StringLiteral , 1 , 129, "r">
21. < ) , 1 , 130, )>
22. < ; , 1 , 131, ;>
23. < Keyword , 1 , 137, if>
24. < ( , 1 , 139, (>
25. < Identifier , 1 , 140, fptr>
26. < == , 1 , 145, =>
27. < Identifier , 1 , 148, NULL>
28. < ) , 1 , 152, )>
29. < { , 1 , 153, {>
30. < Identifier , 1 , 163, printf>
31. < ( , 1 , 169, (>
32. < StringLiteral , 1 , 170, "Cannot open file\n">
33. < ) , 1 , 188, )>
34. < ; , 1 , 189, ;>
35. < Keyword , 1 , 199, exit>
36. < ( , 1 , 203, (>
37. < Number , 1 , 204, 0>
38. < ) , 1 , 205, )>
39. < ; , 1 , 206, ;>
40. < } , 1 , 212, }>
41. < Identifier , 1 , 218, c>
42. < = , 1 , 220, =>
43. < Identifier , 1 , 222, fgetc>
44. < ( , 1 , 227, (>
45. < Identifier , 1 , 228, fptr>
46. < ) , 1 , 232, )>
47. < ; , 1 , 233, ;>
48. < Keyword , 1 , 239, while>
49. < ( , 1 , 244, (>
50. < Identifier , 1 , 245, c>
51. < != , 1 , 247, !>
52. < Identifier , 1 , 250, EOF>
53. < ) , 1 , 253, )>
```

```
54. < { , 1 , 254, {>
55. < Keyword , 1 , 264, int>
56. < Identifier , 1 , 268, i>
57. < = , 1 , 270, =>
58. < Number , 1 , 272, 0>
59. < ; , 1 , 273, ;>
60. < Keyword , 2 , 9, if>
61. < ( , 2 , 11, (>
62. < Identifier , 2 , 12, c>
63. < == , 2 , 14, =>
64. < = , 2 , 18, =>
65. < ) , 2 , 20, )>
66. < { , 2 , 21, {>
67. < Identifier , 2 , 35, buf>
68. < Identifier , 2 , 39, i>
69. < + , 2 , 40, +>
70. < = , 2 , 44, =>
71. < Identifier , 2 , 46, c>
72. < ; , 2 , 47, ;>
73. < Identifier , 2 , 61, c>
74. < = , 2 , 63, =>
75. < Identifier , 2 , 65, fgetc>
76. < ( , 2 , 70, (>
77. < Identifier , 2 , 71, fptr>
78. < ) , 2 , 75, )>
79. < ; , 2 , 76, ;>
80. < Keyword , 2 , 90, if>
81. < ( , 2 , 92, (>
82. < Identifier , 2 , 93, c>
83. < == , 2 , 95, =>
84. < = , 2 , 99, =>
85. < ) , 2 , 101, )>
86. < { , 2 , 102, {>
87. < Identifier , 2 , 120, buf>
88. < Identifier , 2 , 124, i>
89. < + , 2 , 125, +>
90. < = , 2 , 129, =>
91. < Identifier , 2 , 131, c>
92. < ; , 2 , 132, ;>
93. < Identifier , 2 , 150, buf>
94. < Identifier , 2 , 154, i>
95. < = , 2 , 156, =>
96. < Number , 2 , 159, 0>
97. < ; , 2 , 161, ;>
98. < Identifier , 2 , 179, printf>
99. < ( , 2 , 185, (>
100. < StringLiteral , 2 , 186, "Relational operator: %s\n">
101. < , , 2 , 211, ,>
102. < Identifier , 2 , 213, buf>
103. < ) , 2 , 216, )>
104. < ; , 2 , 217, ;>
105. < } , 2 , 231, }>
```

```
106. < Keyword , 2 , 232, else>
107. < { , 2 , 236, {>
108. < Identifier , 2 , 254, buf>
109. < Identifier , 2 , 258, i>
110. < = , 2 , 261, =>
111. < Number , 2 , 265, 0>
112. < ; , 2 , 267, ;>
113. < Identifier , 2 , 285, printf>
114. < ( , 2 , 291, (>
115. < StringLiteral , 2 , 292, "Assignment operator: %s\n">
116. < , , 2 , 317, ,>
117. < Identifier , 2 , 319, buf>
118. < ) , 2 , 322, )>
119. < ; , 2 , 323, ;>
120. < } , 2 , 337, }>
121. < } , 2 , 347, }>
122. < Keyword , 2 , 348, else>
123. < { , 2 , 352, {>
124. < Keyword , 2 , 366, if>
125. < ( , 2 , 368, (>
126. < Identifier , 2 , 369, c>
127. < == , 2 , 371, =>
128. < < , 2 , 375, <>
129. < Identifier , 2 , 381, c>
130. < == , 2 , 383, =>
131. < > , 2 , 387, >>
132. < Identifier , 2 , 393, c>
133. < == , 2 , 395, =>
134. < ! , 2 , 399, !>
135. < ) , 2 , 401, )>
136. < { , 2 , 402, {>
137. < Identifier , 2 , 420, buf>
138. < Identifier , 2 , 424, i>
139. < + , 2 , 425, +>
140. < = , 2 , 429, =>
141. < Identifier , 2 , 431, c>
142. < ; , 2 , 432, ;>
143. < Identifier , 2 , 450, c>
144. < = , 2 , 452, =>
145. < Identifier , 2 , 454, fgetc>
146. < ( , 2 , 459, (>
147. < Identifier , 2 , 460, fptr>
148. < ) , 2 , 464, )>
149. < ; , 2 , 465, ;>
150. < Keyword , 2 , 484, if>
151. < ( , 2 , 486, (>
152. < Identifier , 2 , 487, c>
153. < == , 2 , 489, =>
154. < = , 2 , 493, =>
155. < ) , 2 , 495, )>
156. < Identifier , 2 , 497, buf>
157. < Identifier , 2 , 501, i>
158. < + , 2 , 502, +>
159. < = , 2 , 506, =>
160. < Identifier , 2 , 508, c>
161. < ; , 2 , 509, ;>
162. < Identifier , 2 , 527, buf>
163. < Identifier , 2 , 531, i>
164. < = , 2 , 534, =>
165. < Number , 2 , 538, 0>
166. < ; , 2 , 540, ;>
167. < Identifier , 2 , 558, printf>
168. < ( , 2 , 564, (>
169. < StringLiteral , 2 , 565, "Relational operator: %s\n">
170. < , , 2 , 590, ,>
171. < Identifier , 2 , 592, buf>
172. < ) , 2 , 595, )>
173. < ; , 2 , 596, ;>
174. < } , 2 , 610, }>
175. < Keyword , 2 , 611, else>
176. < Identifier , 2 , 616, buf>
177. < Identifier , 2 , 620, i>
178. < = , 2 , 623, =>
179. < Number , 2 , 627, 0>
180. < ; , 2 , 629, ;>
181. < } , 2 , 639, }>
182. < Identifier , 2 , 649, c>
183. < = , 2 , 651, =>
184. < Identifier , 2 , 653, fgetc>
185. < ( , 2 , 658, (>
186. < Identifier , 2 , 659, fptr>
187. < ) , 2 , 663, )>
188. < ; , 2 , 664, ;>
189. < } , 2 , 670, }>
190. < Identifier , 2 , 676, fclose>
191. < ( , 2 , 682, (>
192. < Identifier , 2 , 683, fptr>
193. < ) , 2 , 687, )>
194. < ; , 2 , 688, ;>
195. < Keyword , 2 , 694, return>
196. < Number , 2 , 701, 0>
197. < ; , 2 , 702, ;>
198. < } , 2 , 704, }>
```