CD LAB-4 CONSTRUCTION OF SYMBOL TABLE

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

Q1. 1. Using getNextToken( ) implemented in Lab No 3, design a Lexical Analyser to implement the following symbol tables.
a. local symbol table
b. global symbol table

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#define TableLength 30
typedef struct {
char tokenname[100];
char type[100];
int size;
int isArray;
int row;
int col;
} Token;
struct HashEntry {
Token tok;
int isOccupied;
};
struct HashEntry TABLE[TableLength];
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",
NULL
};
int getTypeSize(const char *type) {
if (strcmp(type, "int") == 0) return 4;
if (strcmp(type, "char") == 0) return 1;
if (strcmp(type, "float") == 0) return 4;
if (strcmp(type, "double") == 0) return 8;
return 0;
}
void Initialize() {
for (int i = 0; i < TableLength; i++) {
TABLE[i].isOccupied = 0;
}
}
int HASH(char *str) {
unsigned long hash = 5381;
```

```
int c;
while ((c = *str++)) {
hash = ((hash << 5) + hash) + c;
}
return hash % TableLength;
}
int SEARCH(char *str) {
int index = HASH(str);
for (int i = 0; i < TableLength; i++) {
int probeIndex = (index + i) % TableLength;
if (TABLE[probeIndex].isOccupied == 0) {
return 0;
}
if (TABLE[probeIndex].isOccupied == 1 &&
strcmp(TABLE[probeIndex].tok.tokenname, str) == 0) {
return 1;
}
}
return 0;
}
void INSERT(Token tk) {
if (SEARCH(tk.tokenname) == 1) {
return;
}
int index = HASH(tk.tokenname);
for (int i = 0; i < TableLength; i++) {
int probeIndex = (index + i) % TableLength;
if (TABLE[probeIndex].isOccupied == 0) {
TABLE[probeIndex].tok = tk;
TABLE[probeIndex].isOccupied = 1;
return;
}
}
printf("Hash table is full, cannot insert %s\n", tk.tokenname);
}
void Display() {
printf("Local Symbol Table:\n");
printf("Index\tToken Name\tType\tSize\n");
for (int i = 0; i < TableLength; i++) {
if (TABLE[i].isOccupied == 1) {
printf("%d\t%s\t\t%s\t%d\n", i, TABLE[i].tok.tokenname, TABLE[i].tok.type,
TABLE[i].tok.size);
}
}
}
void identifyOperators(char c, Token *token) {
token->tokenname[0] = c;
token->tokenname[1] = '\0';
if (strchr("+-*/", c)) {
strcpy(token->type, "Arithmetic operator");
token->size = 0;
} else if (c == '=') {
```

```
strcpy(token->type, "Assignment operator");
token->size = 0;
} else if (strchr("<>!", c)) {
strcpy(token->type, "Relational operator");
token->size = 0;
} else if (c == '&' || c == '|') {
strcpy(token->type, "Logical operator");
token->size = 0;
} else {
strcpy(token->type, "Special symbol");
token->size = 0;
}
}
void identifyKeywords(char *buf, Token *token) {
for (int i = 0; keywords[i] != NULL; i++) {
if (strcmp(buf, keywords[i]) == 0) {
strcpy(token->type, "Keyword");
strcpy(token->tokenname, buf);
token->size = 0;
token->isArray = 0;
return;
}
}
strcpy(token->type, "Identifier");
strcpy(token->tokenname, buf);
token->size = getTypeSize("int");
token->isArray = 0;
}
void identifyNumericalConstants(char *buf, Token *token) {
strcpy(token->type, "Numerical constant");
strcpy(token->tokenname, buf);
token->size = sizeof(int);
token->isArray = 0;
}
void identifyStringLiterals(char *buf, Token *token) {
strcpy(token->type, "String literal");
strcpy(token->tokenname, buf);
token->size = strlen(buf) - 2;
token->isArray = 0;
}
Token getNextToken(FILE *file, int *row, int *col) {
Token token = { .tokenname = "", .type = "", .size = 0, .isArray = 0 };
char c, buf[100];
int bufIndex = 0;
while ((c = fgetc(file)) != EOF) {
(*col)++;
if (c == '\n') {
(*row)++;
*col = 0;
continue;
}
if (isspace(c)) continue;
```

```
if (c == '/') {
char next = fgetc(file);
if (next == '/') {
while ((c = fgetc(file)) != '\n' && c != EOF);
(*row)++;
*col = 0;
continue;
} else if (next == '*') {
while (1) {
c = fgetc(file);
if (c == EOF) break;
if (c == '*') {
if ((c = fgetc(file)) == '/') break;
}
}
continue;
} else {
ungetc(next, file);
}
}
if (c == '#') {
while ((c = fgetc(file)) != '\n' && c != EOF);
(*row)++;
*col = 0;
continue;
}
if (c == '"') {
bufIndex = 0;
buf[bufIndex++] = c;
while ((c = fgetc(file)) != '"' && c != EOF) {
if (bufIndex < 100 - 1) {
buf[bufIndex++] = c;
}
}
if (bufIndex < 100) {
buf[bufIndex++] = c;
}
buf[bufIndex] = '\0';
identifyStringLiterals(buf, &token);
token.row = *row;
token.col = *col;
return token;
}
if (isalpha(c) || c == '_') {
bufIndex = 0;
buf[bufIndex++] = c;
while (isalnum(c = fgetc(file)) || c == '_') {
if (bufIndex < 100 - 1) {
buf[bufIndex++] = c;
}
}
ungetc(c, file);
```

```
buf[bufIndex] = '\0';
identifyKeywords(buf, &token);
token.row = *row;
token.col = *col;
return token;
}
if (isdigit(c)) {
bufIndex = 0;
buf[bufIndex++] = c;
while (isdigit(c = fgetc(file))) {
if (bufIndex < 100 - 1) {
buf[bufIndex++] = c;
}
}
ungetc(c, file);
buf[bufIndex] = '\0';
identifyNumericalConstants(buf, &token);
token.row = *row;
token.col = *col;
return token;
}
identifyOperators(c, &token);
token.row = *row;
token.col = *col;
return token;
}
return token;
}
int main() {
FILE *file = fopen("sampleinput.c", "r");
if (!file) {
perror("Unable to open file");
return EXIT_FAILURE;
}
Initialize();
Token token;
int row = 1, col = 0;
while (1) {
token = getNextToken(file, &row, &col);
if (strlen(token.tokenname) == 0) break;
if (token.isArray) {
char *sizeStart = strchr(token.tokenname, '[');
if (sizeStart) {
int arraySize = atoi(sizeStart + 1);
token.size = getTypeSize(token.type) * arraySize;
} else {
token.size = getTypeSize(token.type);
}
}
INSERT(token);
}
fclose(file);
```

```
Display();
return EXIT_SUCCESS;
}
```

```
Local Symbol Table:
Index   Token Name      Type    Size
0       false           Identifier      4
1       main            Identifier      4
2       ;               Special symbol  0
3       <               Relational operator     0
4       =               Assignment operator     0
5       key             Identifier      4
6       {               Special symbol  0
7       arr             Identifier      4
8       int             Keyword 0
9       }               Special symbol  0
10      a               Identifier      4
11      b               Identifier      4
12      search          Identifier      4
13      (               Special symbol  0
14      )               Special symbol  0
15      *               Arithmetic operator     0
16      +               Arithmetic operator     0
17      ,               Special symbol  0
18      for             Keyword 0
19      i               Identifier      4
20      if              Keyword 0
21      0               Numerical constant      4
22      sum             Identifier      4
23      [               Special symbol  0
24      10              Numerical constant      4
25      ]               Special symbol  0
26      true            Identifier      4
27      bool            Identifier      4
28      s               Identifier      4
29      return          Keyword 0
```