LAB 7 – RD PARSER FOR DECLARATION STATEMENTS

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

Q1) For given subset of grammar 7.1, design RD parser with appropriate error messages with expected character and row and column number.
Program -> main() { declaration assign_stat }
declaration -> data-type-identifier; declarations | E
data-type -> int | char
identifier-list -> id | id, identifier-list
assign_stat -> id=id; | id = num

i) la.c

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

struct token{
    char lexeme[64];
    int row, col;
    int index;
    char returnType[20];
    char dataType[20];
    char tokenType[20];
    int argc;
    char type[20];
};

struct ListElement{
    struct token tkn;
    struct ListElement *next;
};

struct ListElement *TABLE[30];

void Initialize(){
    for (int i = 0; i < 30; i++){
        TABLE[i] = NULL;
    }
}

int hash(char *str){
    int sum = 0;
    for (int i = 0; i < strlen(str); i++){
        sum += str[i];
    }
    return sum % 30;
}
```

```
int search(char *str){
    int val = hash(str);
    if (TABLE[val] == NULL) return 0;
    else{
        struct ListElement *cur = TABLE[val];
        while (cur){
            if (strcmp(cur->tkn.lexeme, str) == 0) return 1;
            cur = cur->next;
        }
    }
    return 0;
}

void display(){
    printf("SL.NO\tLEX_NAME\tRET_TYPE\tDAT_TYPE\tTOK_TYPE\tARGC\n");
    for (int i = 0; i < 30; i++){
      if (TABLE[i] == NULL) continue;
        else{
            struct ListElement *ele = TABLE[i];
            while (ele){
                printf("%d\t%s\t\t%s\t\t%s\t\t%s\t\t%d\n", ele->tkn.index, ele->tkn.lexeme, ele-
>tkn.returnType, ele->tkn.dataType, ele->tkn.tokenType, ele->tkn.argc);
                ele = ele->next;
            }
        }
    }
}

void insert(struct token tk){
    if (search(tk.lexeme) == 1) return;
    int val = hash(tk.lexeme);
    struct ListElement *cur = (struct ListElement *)malloc(sizeof(struct ListElement));
    cur->tkn = tk;
    cur->next = NULL;
    if (TABLE[val] == NULL){
        TABLE[val] = cur;
    }
    else{
        struct ListElement *ele = TABLE[val];
        while (ele->next != NULL){
            ele = ele->next;
        }
        ele->next = cur;
    }
}

static int row = 1, col = 1;
char buf[1024];
const char specialsymbols[] = {'?', ';', ':', ','};
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"};
const char *ReturnTypes[] = {"int", "float", "char", "void","double"};
const char *DataTypes[] = {"int", "float", "char", "void","double","long"};
const char arithmeticsymbols[] = {'*', '/', '-', '+', '^'};
const char *predefFuncs[]={"printf","scanf"};

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

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

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

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

int charBelongsTo(int c, const char *arr){
    int len;
    if (arr == specialsymbols) len = sizeof(specialsymbols) / sizeof(char);
    else if (arr == arithmeticsymbols) len = sizeof(arithmeticsymbols) / sizeof(char);
    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(){
    printf("\n");
    ++row;
    col = 1;
}

struct token getNextToken(FILE *fin){
    int c;
    struct token tkn = {.row = -1};
    int gotToken = 0;
    while (!gotToken && (c = fgetc(fin)) != EOF){
        if (c == '/'){
            int d = fgetc(fin);
            ++col;
            if (d == '/'){
                while ((c = fgetc(fin)) != EOF && c != '\n') ++col;
                if (c == '\n') newLine();
            }
            else if (d == '*'){
                do{
                    if (d == '\n') newLine();
                    while ((c = fgetc(fin)) != EOF && c != '*'){
                        ++col;
                        if (c == '\n') newLine();
                    }
                    ++col;
                } while ((d = fgetc(fin)) != EOF && d != '/' && (++col));
                ++col;
            }
            else{
                fillToken(&tkn, c, row, --col, "/");
                gotToken = 1;
                fseek(fin, -1, SEEK_CUR);
            }
        }
        else if (charBelongsTo(c, specialsymbols)){
            char symbol[2] = {c, '\0'};
            fillToken(&tkn, c, row, col, symbol);
            gotToken = 1;
            ++col;
        }
        else if (charBelongsTo(c, arithmeticsymbols)){
            char symbol[2] = {c, '\0'};
            fillToken(&tkn, c, row, col, symbol);
            gotToken = 1;
            ++col;
        }
        else if (c == '(' || c == ')' || c == '{' || c == '}' || c == '[' || c == ']'){
            char symbol[2] = {c, '\0'};
            fillToken(&tkn, c, row, col, symbol);
            gotToken = 1;
```

```
    ++col;
}
else if (c == '+' || c == '-'){
    int d = fgetc(fin);
    if (d != c){
        char symbol[2] = {c, '\0'};
        fillToken(&tkn, c, row, col, symbol);
        gotToken = 1;
        ++col;
        fseek(fin, -1, SEEK_CUR);
    }
    else{
        char symbol[3] = {c, c, '\0'};
        fillToken(&tkn, c, row, col, symbol);
        gotToken = 1;
        col += 2;
    }
}
else if (c == '=' || c == '<' || c == '>'){
    char symbol[2] = {c, '\0'};
    fillToken(&tkn, c, row, col, symbol);
    gotToken = 1;
    ++col;
    int d = fgetc(fin);
    if (d == '='){
        ++col;
        strcat(tkn.lexeme, "=");
    }
    else fseek(fin, -1, SEEK_CUR);
}
else if (isdigit(c)){
    tkn.row = row;
    tkn.col = col++;
    tkn.lexeme[0] = c;
    int k = 1;
    while ((c = fgetc(fin)) != EOF && isdigit(c)){
        tkn.lexeme[k++] = c;
        col++;
    }
    tkn.lexeme[k] = '\0';
    strcpy(tkn.type, "Number");
    gotToken = 1;
    fseek(fin, -1, SEEK_CUR);
}
else if (c == '#'){
    while ((c = fgetc(fin)) != EOF && c != '\n');
    newLine();
}
else if (c == '\n'){
    newLine();
    c = fgetc(fin);
    if (c == '#'){
```

```
            while ((c = fgetc(fin)) != EOF && c != '\n');
            newLine();
        }
        else if (c != EOF) fseek(fin, -1, SEEK_CUR);
    }
    else if (isspace(c)) ++col;
    else if (isalpha(c) || c == '_'){
        tkn.row = row;
        tkn.col = col++;
        tkn.lexeme[0] = c;
        int k = 1;
        while ((c = fgetc(fin)) != EOF && isalnum(c)) {
            tkn.lexeme[k++] = c;
            ++col;
        }
        tkn.lexeme[k] = '\0';
        if (isKeyword(tkn.lexeme)) strcpy(tkn.type, "Keyword");
        else strcpy(tkn.type, "Identifier");
        gotToken = 1;
        fseek(fin, -1, SEEK_CUR);
    }
    else if (c == ""){
        tkn.row = row;
        tkn.col = col;
        strcpy(tkn.type, "StringLiteral");
        int k = 1;
        tkn.lexeme[0] = "";
        while ((c = fgetc(fin)) != EOF && c != ""){
            tkn.lexeme[k++] = c;
            ++col;
        }
        tkn.lexeme[k] = "";
        gotToken = 1;
    }
    else if (c == '&' || c == '|'){
        int d = fgetc(fin);
        if (d == c){
            tkn.lexeme[0] = tkn.lexeme[1] = c;
            tkn.lexeme[2] = '\0';
            tkn.row = row;
            tkn.col = col;
            ++col;
            gotToken = 1;
            char symbol[3] = {c, c, '\0'};
            fillToken(&tkn, c, row, col, symbol);
        }
        else fseek(fin, -1, SEEK_CUR);
        ++col;
    }
    else ++col;
}
return tkn;
```

```
}

void printToken(struct token *tkn, int *index){
   if (strcmp(tkn->type, "Identifier") == 0){
      char id[10];
      if (search(tkn->lexeme) == 0){
         tkn->index = *index;
         (*index)++;
      }
      else{
         struct ListElement *cur = TABLE[hash(tkn->lexeme)];
         while (cur) {
            if (strcmp(cur->tkn.lexeme, tkn->lexeme) == 0){
               tkn->index = cur->tkn.index;
               break;
            }
            cur = cur->next;
         }
      }
      sprintf(id, "id%d", tkn->index);
      printf("<%s,%d,%d>", id, tkn->row, tkn->col);
      return;
   }
   printf("<%s,%d,%d>", tkn->lexeme, tkn->row, tkn->col);
}

void insertToST(struct token tkn, char *type, char *recTypetkn, int argc){
   if (strcmp(tkn.type, "Identifier") == 0){
      if (strcmp(type, "VAR") == 0){
         tkn.argc = 0;
         strcpy(tkn.returnType, "-");
         strcpy(tkn.tokenType, type);
         strcpy(tkn.dataType, recTypetkn);
         insert(tkn);
      }
      else{
         tkn.argc = argc;
         strcpy(tkn.returnType, recTypetkn);
         strcpy(tkn.tokenType, type);
         strcpy(tkn.dataType, "-");
         insert(tkn);
      }
   }
}
```

ii) parser.c

```
#include "la.c"

struct token cur;
FILE *f;
```

```
void declarations();
void dataTypes();
void identifierList();
void assignStat();

void valid(){
   printf("----------------------SUCCESS----------------------");
   exit(EXIT_SUCCESS);
}

void invalid(){
   printf("----------------------ERROR----------------------");
   exit(EXIT_FAILURE);
}

void declarations(){
   dataTypes();
   identifierList();
   if(strcmp(cur.lexeme,";")==0){
      cur=getNextToken(f);
      if(isDataType(cur.lexeme)==0) return;
      declarations();
   }
   else{
      printf("Missing \";\" at Row : %d and Column : %d\n",cur.row,cur.col);
      exit(EXIT_FAILURE);
   }
}

void dataTypes(){
   if(isDataType(cur.lexeme)) {
      cur=getNextToken(f);
      return;
   }
   else{
      printf("Missing Data Type at Row : %d and Column : %d\n",cur.row,cur.col);
      exit(EXIT_FAILURE);
   }
}

void identifierList(){
   if(strcmp(cur.type,"Identifier")==0){
      cur=getNextToken(f);
      if(strcmp(cur.lexeme,",")==0){
         cur=getNextToken(f);
         identifierList();
      }
      else if(strcmp(cur.type,"Identifier")==0){
         printf("Missing \",\" at Row : %d and Column : %d\n",cur.row,cur.col);
         exit(EXIT_FAILURE);
      }
      else{
```

```
            return;
        }
    }
    else{
        printf("Missing Identifier at Row : %d and Column : %d\n",cur.row,cur.col);
        exit(EXIT_FAILURE);
    }
}

void assignStat(){
    if(strcmp(cur.type,"Identifier")==0){
        cur=getNextToken(f);
        if(strcmp(cur.lexeme,"=")==0){
            cur=getNextToken(f);
            if(strcmp(cur.type,"Identifier")==0){
                cur=getNextToken(f);
                if(strcmp(cur.lexeme,";")==0){
                    cur=getNextToken(f);
                    return;
                }
                else{
                    printf("Missing \";\" at Row : %d and Column : %d\n",cur.row,cur.col);
                    exit(EXIT_FAILURE);
                }
            }
            else if(strcmp(cur.type,"Number")==0){
                cur=getNextToken(f);
                if(strcmp(cur.lexeme,";")==0){
                    cur=getNextToken(f);
                    return;
                }
                else{
                    printf("Missing \";\" at Row : %d and Column : %d\n",cur.row,cur.col);
                    exit(EXIT_FAILURE);
                }
            }
            else{
                printf("Missing Identifier at Row : %d and Column : %d\n",cur.row,cur.col);
                exit(EXIT_FAILURE);
            }
        }
        else{
            printf("Missing \"=\" at Row : %d and Column : %d\n",cur.row,cur.col);
            exit(EXIT_FAILURE);
        }
    }
    else{
        printf("Missing Identifier at Row : %d and Column : %d\n",cur.row,cur.col);
        exit(EXIT_FAILURE);
    }
}
```

```
void program(){
   cur=getNextToken(f);
   if(strcmp(cur.lexeme,"main")==0){
      cur=getNextToken(f);
      if(strcmp(cur.lexeme,"(")==0){
         cur=getNextToken(f);
         if(strcmp(cur.lexeme,")")==0){
            cur=getNextToken(f);
            if(strcmp(cur.lexeme,"{")==0){
               cur=getNextToken(f);
               declarations();
               assignStat();
               if(strcmp(cur.lexeme,"}")==0) return;
               else {
               printf("Missing \"}\" at Row : %d and Column : %d\n",cur.row,cur.col);
               exit(EXIT_FAILURE);
               }
            }
            else {
               printf("Missing \"{\" at Row : %d and Column : %d\n",cur.row,cur.col);
               exit(EXIT_FAILURE);
            }
         }
         else {
            printf("Missing \"(\" at Row : %d and Column : %d\n",cur.row,cur.col);
            exit(EXIT_FAILURE);
         }
      }
      else {
         printf("Missing \"(\" at Row : %d and Column : %d\n",cur.row,cur.col);
         exit(EXIT_FAILURE);
      }
   }
   else {
      printf("\nMissing main function\n\n");
      exit(EXIT_FAILURE);
   }
}

void main(int argc,char** argv){
   if(argc<2){
      perror("Insufficient Arguments\n");
      exit(EXIT_FAILURE);
   }
   if(argc>2){
      perror("Extra Arguments\n");
      exit(EXIT_FAILURE);
   }
   f = fopen(argv[1], "r");
   if (f == NULL) {
      printf("ERROR\n");
      exit(EXIT_FAILURE);
```

```
    }
    program();
    printf("----------------------COMPILED----------------------\n");
    exit(EXIT_SUCCESS);
}
```

iii) input.c CASE1

```
main()
{
    int a,b;
    a=10;
}
```

Terminal output

```
CD_LAB_A1@debianpc-02:~/Desktop/220905018/Lab7-RDPForDeclarationStatements$ gcc -o parser parser.c
CD_LAB_A1@debianpc-02:~/Desktop/220905018/Lab7-RDPForDeclarationStatements$ ./parser input.c

----------------------COMPILED----------------------
```

iv) input.c CASE2

```
int main()
{
    int a,b;
    a=10;
}
```

Terminal output

```
CD_LAB_A1@debianpc-02:~/Desktop/220905018/Lab7-RDPForDeclarationStatements$ gcc -o parser parser.c
CD_LAB_A1@debianpc-02:~/Desktop/220905018/Lab7-RDPForDeclarationStatements$ ./parser input.c

Missing main function
```