Ayush Goyal
190905522 CSE D Roll 62

Compiler Design Week 6: Lab 7: RD Parser for Declaration Statements

Lab Excercise:

1. For given subset of grammar 7.1, design RD parser with appropriate error messages with expected character and row and column number.

Program → main () { declarations assign_stat }
declarations→ data-type identifier-list; declarations | ∈
data-type → int | char
identifier-list → id | id, identifier-list
assign_stat → id=id; | id = num;

Solution:

The main code for RD Parser is as follows: "rdParser.c"

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

FILE *f1;

struct token tkn;

void program();
void declarations();
int data_type();
void identifier_list(struct token);
void assign_stat(struct token);

void printValid(){
   printf("\n *** Sucessful! ***\n");
   exit(0);
}

void printInvalid(struct token *tkn){
   printf("Error at row: %d, Column: %d for lexeme \" %s \" \n", tkn->row, tkn->col, tkn->lexeme);
   printf("\n *** Unsucessful! ***\n");
   exit(0);
}
```

```
void program(){
    tkn = getNextToken(f1);
    if (strcmp(tkn.lexeme, "main") == 0){
        tkn = getNextToken(f1);
        if (strcmp(tkn.lexeme, "(") == 0){
            tkn = getNextToken(f1);
            if (strcmp(tkn.lexeme, ")") == 0){
                tkn = getNextToken(f1);
                if (strcmp(tkn.lexeme, "{") == 0){
                    declarations();
                    tkn = getNextToken(f1);
                    if (strcmp(tkn.lexeme, "}") == 0)
                        return;
                    else
                        printInvalid(&tkn);
                }
                else
                    printInvalid(&tkn);
            }
            else
                printInvalid(&tkn);
        }
        else
            printInvalid(&tkn);
    }
    else
        printInvalid(&tkn);
}

void declarations(){
    tkn = getNextToken(f1);
    if (data_type(tkn.lexeme)){
        tkn = getNextToken(f1);
        identifier_list(tkn);
        tkn = getNextToken(f1);
        if (strcmp(tkn.lexeme, ";") == 0)
            declarations();
        else
            printInvalid(&tkn);
    }
    else
        assign_stat(tkn);
}

int data_type(char *lx){
    if (strcmp(lx, "int") == 0 || strcmp(lx, "char") == 0)
        return 1;
    else
        return 0;
}
```

```
void identifier_list(struct token tkn){
    struct token tkn2;
    tkn2 = tkn;
    if (strcmp(tkn2.type, "Identifier") == 0){
        tkn2 = getNextToken(f1);
        if (strcmp(tkn2.lexeme, ",") == 0){
            tkn2 = getNextToken(f1);
            identifier_list(tkn2);
        }
        else if (strcmp(tkn2.lexeme, ";") == 0){
            fseek(f1, -1, SEEK_CUR);
            return;
        }
        else
            printInvalid(&tkn2);
    }
}


void assign_stat(struct token tkn){
    struct token tkn2;
    if (strcmp(tkn.type, "Identifier") == 0){
        tkn2 = getNextToken(f1);
        if (strcmp(tkn2.lexeme, "=") == 0){
            tkn2 = getNextToken(f1);
            if (strcmp(tkn2.type, "Number") == 0 || strcmp(tkn2.type, "Identifier")){
                tkn2 = getNextToken(f1);
                if (strcmp(tkn2.lexeme, ";") == 0)
                    return;
                else
                    printInvalid(&tkn);
            }
            else
                printInvalid(&tkn);
        }
        else
            printInvalid(&tkn);
    }
}


int main(){
    f1 = fopen("sampleParser.c", "r");
    if (f1 == NULL){
        printf("Error! File cannot be opened!\n");
        return 0;
    }
    struct sttable st[10][100];
    int flag = 0, i = 0, j = 0;
    int tabsz[10];
    char w[25];
    w[0] = '\0';
```

```
    program();
    printValid();
}
```

I have used previously implemented lexical analyzer and initializer code as follows:

“lexicalAnalyzer.h”

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

const char *keywords[] = {"int", "return", "for", "while", "do", "else", "case", "break",
"unsigned","const"};
const char *datypes[] = {"int", "char", "void", "float", "bool"};

int isdtype(char *w){
	for (int i = 0; i < sizeof(datypes) / sizeof(char *); i++){
		if (strcmp(w, datypes[i]) == 0)
			return 1;
	}
	return 0;
}

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

struct token{
	char lexeme[128];
	unsigned int row, col;
	char type[64];
};

struct sttable{
	int sno;
	char lexeme[128];
	char dtype[64];
	char type[64];
	int size;
};

int findTable(struct sttable *tab, char *nam, int n){
	for (int i = 0; i < n; i++){
		if (strcmp(tab[i].lexeme, nam) == 0)
			return 1;
```

```
	}
	return 0;
}

struct sttable fillTable(int sno, char *lexn, char *dt, char *t, int s){
	struct sttable tab;
	tab.sno = sno;
	strcpy(tab.lexeme, lexn);
	strcpy(tab.dtype, dt);
	strcpy(tab.type, t);
	tab.size = s;
	return tab;
}

void printTable(struct sttable *tab, int n){
	for (int i = 0; i < n; i++)
		printf("%d %s %s %s %d\n", tab[i].sno, tab[i].lexeme, tab[i].dtype, tab[i].type,
tab[i].size);
}

static int row = 1, col = 1;
char buf[2048];
char dbuf[128];
int ind = 0;
const char specialsymbols[] = {'?', ';', ':', ','};
const char arithmeticsymbols[] = {'*'};

int charIs(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(){
	row++;
	col = 1;
}
```

```
int sz(char *w){
	if (strcmp(w, "int") == 0)
		return 4;
	if (strcmp(w, "char") == 0)
		return 1;
	if (strcmp(w, "void") == 0)
		return 0;
	if (strcmp(w, "float") == 0)
		return 8;
	if (strcmp(w, "bool") == 0)
		return 1;
}

struct token getNextToken(FILE *fa){
	int c;
	struct token tkn =
		{
			.row = -1
		};
	int gotToken = 0;
	while (!gotToken && (c = fgetc(fa)) != EOF){
		if (charIs(c, specialsymbols)){
			fillToken(&tkn, c, row, col, "Special Symbol");
			gotToken = 1;
			col++;
		}
		else if (charIs(c, arithmeticsymbols)){
			fseek(fa, -1, SEEK_CUR);
			c = getc(fa);
			if (isalnum(c)){
				fillToken(&tkn, c, row, col, "Arithmetic Operator");
				gotToken = 1;
				col++;
			}
			fseek(fa, 1, SEEK_CUR);
		}
		else if (c == '('){
			fillToken(&tkn, c, row, col, "Left Bracket");
			gotToken = 1;
			col++;
		}
		else if (c == ')'){
			fillToken(&tkn, c, row, col, "Right Bracket");
			gotToken = 1;
			col++;
		}
		else if (c == '{'){
			fillToken(&tkn, c, row, col, "LC");
			gotToken = 1;
			col++;
		}
```

```
else if (c == '}'){
	fillToken(&tkn, c, row, col, "RC");
	gotToken = 1;
	col++;
}
else if (c == '['){
	fillToken(&tkn, c, row, col, "LS");
	gotToken = 1;
	col++;
}
else if (c == ']'){
	fillToken(&tkn, c, row, col, "RS");
	gotToken = 1;
	col++;
}
else if (c == '+'){
	int x = fgetc(fa);
	if (x != '+'){
		fillToken(&tkn, c, row, col, "Arithmetic Operator");
		gotToken = 1;
		col++;
		fseek(fa, -1, SEEK_CUR);
	}
	else{
		fillToken(&tkn, c, row, col, "Unary Operator");
		strcpy(tkn.lexeme, "++");
		gotToken = 1;
		col += 2;
	}
}
else if (c == '-'){
	int x = fgetc(fa);
	if (x != '-'){
		fillToken(&tkn, c, row, col, "Arithmetic Operator");
		gotToken = 1;
		col++;
		fseek(fa, -1, SEEK_CUR);
	}
	else{
		fillToken(&tkn, c, row, col, "Unary Operator");
		strcpy(tkn.lexeme, "++");
		gotToken = 1;
		col += 2;
	}
}
else if (c == '='){
	int x = fgetc(fa);
	if (x != '='){
		fillToken(&tkn, c, row, col, "Assignment Operator");
		gotToken = 1;
		col++;
		fseek(fa, -1, SEEK_CUR);
```

```
		}
		else{
			fillToken(&tkn, c, row, col, "Relational Operator");
			strcpy(tkn.lexeme, "++");
			gotToken = 1;
			col += 2;
		}
	}
	else if (isdigit(c)){
		fillToken(&tkn, c, row, col++, "Number");
		int j = 1;
		while ((c = fgetc(fa)) != EOF && isdigit(c)){
			tkn.lexeme[j++] = c;
			col++;
		}
		tkn.lexeme[j] = '\0';
		gotToken = 1;
		fseek(fa, -1, SEEK_CUR);
	}
	else if (c == '#'){
		while ((c = fgetc(fa)) != EOF && c != '\n')
			;
		newLine();
	}
	else if (c == '\n'){
		newLine();
		c = fgetc(fa);
		if (c == '#'){
			while ((c = fgetc(fa)) != EOF && c != '\n')
				;
			newLine();
		}
		else if (c != EOF)
			fseek(fa, -1, SEEK_CUR);
	}
	else if (isspace(c))
		col++;
	else if (isalpha(c) || c == '_'){
		tkn.row = row;
		tkn.col = col++;
		tkn.lexeme[0] = c;
		int j = 1;
		while ((c = fgetc(fa)) != EOF && isalnum(c)){
			tkn.lexeme[j++] = c;
			col++;
		}
		tkn.lexeme[j] = '\0';
		if (isKeyword(tkn.lexeme))
			strcpy(tkn.type, "Keyword");
		else
			strcpy(tkn.type, "Identifier");
		gotToken = 1;
```

```
		fseek(fa, -1, SEEK_CUR);
	}
	else if (c == '/'){
		int d = fgetc(fa);
		col++;
		if (d == '/'){
			while ((c = fgetc(fa)) != EOF && c != '\n')
				col++;
			if (c == '\n')
				newLine();
		}
		else if (d == '*'){
			do{
				if (d == '\n')
					newLine();
				while ((c == fgetc(fa)) != EOF && c != '*'){
					col++;
					if (c == '\n')
						newLine();
				}
				col++;
			}while((d == fgetc(fa)) != EOF && d != '/' && (col++));
			col++;
		}
		else{
			fillToken(&tkn, c, row, --col, "Arithmetic Operator");
			gotToken = 1;
			fseek(fa, -1, SEEK_CUR);
		}
	}
	else if (c == '"'){
		tkn.row = row;
		tkn.col = col;
		strcpy(tkn.type, "String Literal");
		int k = 1;
		tkn.lexeme[0] = '"';
		while ((c = fgetc(fa)) != EOF && c != '"'){
			tkn.lexeme[k++] = c;
			col++;
		}
		tkn.lexeme[k] = '"';
		gotToken = 1;
	}
	else if (c == '<' || c == '>' || c == '!'){
		fillToken(&tkn, c, row, col, "Relational Operator");
		col++;
		int d = fgetc(fa);
		if (d == '='){
			col++;
			strcat(tkn.lexeme, "=");
		}
		else{
```

```
                        if (c == '!')
                                strcpy(tkn.type, "Logical Operator");
                        fseek(fa, -1, SEEK_CUR);
                }
                gotToken = 1;
        }
        else if (c == '&' || c == '|'){
                int d = fgetc(fa);
                if (c == d){
                        tkn.lexeme[0] = tkn.lexeme[1] = c;
                        tkn.lexeme[2] = '\0';
                        tkn.row = row;
                        tkn.col = col;
                        col++;
                        gotToken = 1;
                        strcpy(tkn.type, "Logical Operator");
                }
                else
                        fseek(fa, -1, SEEK_CUR);
                col++;
        }
        else
                col++;
    }
    return tkn;
}
```

“initializer.h”

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

void initialize(){
    FILE *fa, *fb;
    char filename[100];
    char c, c2;
    fa = fopen("sampleinput.c", "r");
    fb = fopen("sample.c", "w+");
    while ((c = fgetc(fa)) != EOF){
        if (c == ' '){
            putc(c, fb);
            while (c == ' ')
                c = getc(fa);
        }
        if (c == '/'){
            c2 = getc(fa);
            if (c2 == '/'){
                while (c != '\n')
                    c = getc(fa);
```

```
            }
            else if (c2 == '*'){
                do{
                    while (c != '*')
                        c = getc(fa);
                    c = getc(fa);
                }while (c != '/');
            }
            else{
                putc(c, fb);
                putc(c2, fb);
            }
        }
        else
            putc(c, fb);
    }
    fclose(fa);
    fclose(fb);
    fa = fopen("sample.c", "r");
    fb = fopen("temporary.c", "w+");
    while ((c = fgetc(fa)) != EOF){
        if (c == '"'){
            putc(c, fb);
            c = getc(fa);
            while (c != '"'){
                putc(c, fb);
                c = getc(fa);
            }
        }
        else if (c == '#'){
            while (c != '\n')
                c = getc(fa);
            c = getc(fa);
        }
        putc(c, fb);
    }
    fclose(fa);
    fclose(fb);
    fa = fopen("temporary.c", "r");
    fb = fopen("sample.c", "w");
    c = getc(fa);
    while (c != EOF){
        putc(c, fb);
        c = getc(fa);
    }
    fclose(fa);
    fclose(fb);
    remove("temporary.c");
}
```

Now that we have all the files ready, we use a sample C program as input called "sampleParser.c" and run the code to get the output as shown below in correspondance to the required grammar given in the question:

```
pgcse@pglab-cp: ~/Downloads/AyushGoyal_CDLab/Lab_6
pgcse@pglab-cp:~/Downloads/AyushGoyal_CDLab/Lab_6$ gcc rdParser.c -o rdp
pgcse@pglab-cp:~/Downloads/AyushGoyal_CDLab/Lab_6$ cat sampleParser.c
main()
{
    int a, b;
    b = 20;
}pgcse@pglab-cp:~/Downloads/AyushGoyal_CDLab/Lab_6$ ./rdp

 *** Sucessful! ***
pgcse@pglab-cp:~/Downloads/AyushGoyal_CDLab/Lab_6$ cat sampleParser.c
main()
{
    int a, b;
    b = 20;
    a = 10;
}pgcse@pglab-cp:~/Downloads/AyushGoyal_CDLab/Lab_6$ ./rdp
Error at row: 5, Column: 5 for lexeme " a "

 *** Unsucessful! ***
pgcse@pglab-cp:~/Downloads/AyushGoyal_CDLab/Lab_6$
```

THE END