190905104
Parth Shukla
Lab 6

RD PARSER FOR DECLARATION STATEMENTS

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;

Slight modification to the grammar,
Program -> returntype main() {declaration assign_stat}
returntype -> int

First:
Program: int
returntype: int
declaration: datatype, E
datatype: int, char
identifier-list: id
assign_stat: id
Follow:
Program: $
returntype: main
declaration: id
datatype: id
identifier-list: ;
assign_stat: }

q1.c

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

FILE *f;
struct token t;
void program();
void returntype();
void declarations();
int datatype();
```

```
void identifierlist(struct token);
void assign_stat(struct token);

void invalid(struct token *tkn){
        printf("error at row:%d, col:%d for lexeme \" %s \" \n",tkn->row,tkn-
>col,tkn->lexeme);
        printf("-----ERROR!-----\n");
        exit(1);
}
void valid(){
        printf("------SUCCESS!------\n");
        exit(0);
}

void program(){
   t = getNextToken(f);
   returntype();
   if(strcmp(t.lexeme,"main") == 0){
               t = getNextToken(f);
               if(strcmp(t.lexeme,"(") == 0){
                       t = getNextToken(f);
                       if(strcmp(t.lexeme,")") == 0){
                               t = getNextToken(f);
                               if(strcmp(t.lexeme,"{") == 0){
                                       declarations();
                                       t = getNextToken(f);
                                       if(strcmp(t.lexeme,"}") == 0) return;
                                       else invalid(&t);
                               }
                               else invalid(&t);
                       }
                       else invalid(&t);
               }
               else invalid(&t);
       }
       else invalid(&t);
}

void returntype(){
   t = getNextToken(f);
   if(strcmp(t.lexeme, "int")){
      return;
   }
   else{
      invalid(&t);
   }
}

void declarations(){
        t = getNextToken(f);
        if(datatype(t.lexeme)){
                t = getNextToken(f);
```

```
                identifierlist(t);
                t = getNextToken(f);
                if(strcmp(t.lexeme,";") == 0) declarations();
                else invalid(&t);
        }
        else assign_stat(t);   // incase for the production declarations->E, it will be
handled in assign_stat
}

int datatype(char *lx){
        if(strcmp(lx,"int")==0 || strcmp(lx,"char") == 0) return 1;
        else return 0;
}
void identifierlist(struct token t){
        struct token tkn;
        tkn = t;
        if(strcmp(tkn.type,"identifier") == 0){
                tkn = getNextToken(f);
                if(strcmp(tkn.lexeme,",") == 0){
                        tkn = getNextToken(f);
                        identifierlist(tkn);
                }
                else if(strcmp(tkn.lexeme,";") == 0){
                        fseek(f,-1,SEEK_CUR);
                        return;
                }
                else invalid(&tkn);
        }
}

void assign_stat(struct token t){
        struct token tkn;
        if(strcmp(t.type,"identifier") == 0){
                tkn = getNextToken(f);
                if(strcmp(tkn.lexeme,"=") == 0){
                        tkn = getNextToken(f);
                        if(strcmp(tkn.type,"number") == 0||
strcmp(tkn.type,"identifier")){
                                tkn = getNextToken(f);
                                if(strcmp(tkn.lexeme,";") == 0) return;
                                else invalid(&t);
                        }
                        else invalid(&t);
                }
                else invalid(&t);
        }
}

int main(){
   f = fopen("sample.c", "r");
   if(!f){
      printf("Error in opening file\n");
```

```
        exit(1);
    }

        program();
        valid();
}
```

la.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 *dtypes[] = {"int","char","void","float","bool"};
int isdtype(char *w){
        for(int i = 0;i<sizeof(dtypes)/sizeof(char*);i++){
                if(strcmp(w,dtypes[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 symTable{
        int sno;
        char lexeme[128];
        char dtype[64];
        char type[64];
        int size;
};
int findTable(struct symTable *tab,char *nam,int n){
        for(int i = 0;i<n;i++){
                if(strcmp(tab[i].lexeme,nam)==0) return 1;
        }
        return 0;
}
```

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

void printTable(struct symTable *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 op");
			strcpy(tkn.lexeme, "++");
			gotToken = 1;
			col += 2;
		}
	}
	else if (c == '=')
	{
		int x = fgetc(fa);
		if (x != '=')
		{
			fillToken(&tkn, c, row, col, "assignment op");
			gotToken = 1;
			col++;
			fseek(fa, -1, SEEK_CUR);
		}
		else
		{
			fillToken(&tkn, c, row, col, "relational op");
			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 op");
                        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 op");
                col++;
                int d = fgetc(fa);
                if (d == '=')
                {
                        col++;
                        strcat(tkn.lexeme, "=");
                }
                else
                {
                        if (c == '!')
                                strcpy(tkn.type, "logical op");
                        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;
}
```

sample.c

```
int main()
{
int a;
char c;
a = 1;
}
```