Compiler Design Lab 6

Dipesh Singh – 190905520

Productions :

Program -> int main() {declarations assign_stat return num;}

decalrations -> data_type indentifier_list; declarations | ϵ

data_type -> int | char | float | double

identifier_list -> id identifier_prime

identifier_prime -> , identifier_list | ϵ

assign_stat -> id = assign_prime

assign_prime -> id ; | num ;

Source Code :

makefile :

```
run: main.o
	gcc -o parse main.o

main.o: main.c rdParser.h hash.h tables.h structs.h utils.h
constants.h removePreprocess.h removeExcess.h getNextToken.h
	gcc -c main.c
```

constants.h :

```
#ifndef __CONSTANTS_H__
#define __CONSTANTS_H__

char keywords[34][10] = {
	"true",
	"false",
	"auto",
	"double",
	"int",
	"struct",
	"break",
	"else",
	"long",
	"switch",
	"case",
	"enum",
	"register",
	"typedef",
	"char",
	"extern",
     "return",
     "union",
     "const",
     "float",
     "short",
     "unsigned",
     "continue",
     "for",
     "signed",
     "void",
     "default",
     "goto",
     "sizeof",
     "voltile",
     "do",
     "if",
     "static",
     "while"};            // list of keywords
char data_types[][10] = { // list of data types
     "double",
     "int",
     "char",
     "float"};
char operators[5] = { // list of operators
     '+',
     '-',
     '/',
     '%',
     '*'};
char brackets[6] = { // list of brackets
     '(',
     ')',
     '[',
     ']',
     '{',
     '}'};
char special_symbols[12] = { // list of special symbols
     '*',
     ';',
     ':',
     '.',
     ',',
     '^',
     '&',
     '!',
     '>',
     '<',
     '~',
     '`'};

enum TYPE // lexeme type enumerator
{
     IDENTIFIER,
     KEYWORD,
     STRING_LITERAL,
     NUMERIC_CONSTANT,
     OPERATOR,
     BRACKET,
     SPECIAL_SYMBOL,
     RELATIONAL_OPERATOR,
     CHARACTER_CONSTANT
};

char types[][30] = { // map for type to string
     "IDENTIFIER",
     "KEYWORD",
     "STRING_LITERAL",
     "NUMERIC_CONSTANT",
     "OPERATOR",
     "BRACKET",
     "SPECIAL_SYMBOL",
     "RELATIONAL_OPERATOR",
     "CHARACTER_CONSTANT"};

#endif
```

structs.h :

```
#ifndef __STRUCTS_H__
#define __STRUCTS_H__

struct node
{
     char *cur;
     int row, col;
     struct node *next;
     enum TYPE type;
}; // element for hash table

struct symbol
{
     char *name;
     char *data_type;
     struct symbol *next;
     unsigned int size, row, col;
}; // element for symbol table

struct token
{
     char *lexeme;
     enum TYPE type;
     int row, col;
}; // token returned by getNextToken()

#endif
```

utils.h :

```
#ifndef __UTILS_H__
#define __UTILS_H__
int iskeyword(char buffer[]) // function to check for keyword
{
     for (int i = 0; i < 34; i++)
     {
          if (strcmp(buffer, keywords[i]) == 0)
          {
               return 1;
          }
     }
     return 0;
}

int isdatatype(char buffer[])
{ // function to check for data_Type
     for (int i = 0; i < 4; i++)
     {
          if (strcmp(buffer, data_types[i]) == 0)
               return 1;
     }
     return 0;
}

int isoperator(char c)
{ // function to check for operator
     for (int i = 0; i < 5; i++)
     {
          if (operators[i] == c)
               return 1;
     }
     return 0;
}

int isspecial(char c)
{ // function to check for special symbol
     for (int i = 0; i < 12; i++)
     {
          if (special_symbols[i] == c)
               return 1;
     }
     return 0;
}

int isbracket(char c)
{ // function to check for bracket
     for (int i = 0; i < 6; i++)
     {
          if (brackets[i] == c)
               return 1;
     }
     return 0;
}
#endif
```

hash.h :

```
#ifndef __HASH_H__
#define __HASH_H__
int hash(int size) // hashing function
{
     return (size) % MAX_SIZE;
}

void display_st() // display the symbol table
{
     printf("      Name    |      Type    |      Size     |     Row
|     Col     \n");

printf("------------------------------------------------------------------
---------------\n");
     for (int i = 0; i < MAX_SIZE; i++)
     {
          if (st[i] == NULL)
               continue;
          else
          {
               Symbol cur = st[i];
               while (cur)
               {
                    printf("%10s    |%10s    |%10d    |%10d    |
%10d    \n", cur->name, cur->data_type, cur->size, cur->row, cur-
>col);
                    cur = cur->next;
               }
          }
     }
}

int search_symbol(char identifier[]) // to search in symbol_table
{
     int index = hash(strlen(identifier));
     if (st[index] == NULL)
          return -1;
     Symbol cur = st[index];
     int i = 0;
     while (cur != NULL)
     {
          if (strcmp(identifier, cur->name) == 0)
               return i;
          cur = cur->next;
          i++;
     }
     return -1;
```

```
}

int search(char buffer[], enum TYPE type) // to search in hash
table
{
     int index = hash(strlen(buffer));
     if (hashTable[index] == NULL)
          return 0;
     Node cur = hashTable[index];
     while (cur != NULL)
     {
          if (strcmp(cur->cur, buffer) == 0)
               return 1;
          cur = cur->next;
     }
     return 0;
}

void insert_symbol(char identifier[], char data_type[], int row,
int col)
{ // insert in symbol table
     if (search_symbol(identifier) == -1)
     {
          Symbol n = (Symbol)malloc(sizeof(struct symbol));
          char *str = (char *)calloc(strlen(identifier) + 1,
sizeof(char));
          strcpy(str, identifier);
          n->name = str;
          n->next = NULL;
          n->row = row;
          n->col = col;
          char *typee = (char *)calloc(strlen(data_type) + 1,
sizeof(char));
          strcpy(typee, data_type);
          n->data_type = typee;
          if (strcmp(data_type, "int") == 0)
               n->size = 4;
          else if (strcmp(data_type, "double") == 0)
               n->size = 8;
          else if (strcmp(data_type, "char") == 0)
               n->size = 1;
          else if (strcmp(data_type, "function") == 0)
               n->size = 0;
          else
               n->size = 4;
          int index = hash(strlen(identifier));
          //
          if (st[index] == NULL)
          {
               st[index] = n;
               return;
          }
          Symbol cur = st[index];
          while (cur->next != NULL)
               cur = cur->next;
          cur->next = n;
     }
}

Token insert(char buffer[], int row, int col, enum TYPE type)
{ // insert in hash table
     Token tkn = (Token)malloc(sizeof(struct token));
     char *lexeme = (char *)calloc(strlen(buffer) + 1,
sizeof(char));
     strcpy(lexeme, buffer);
     tkn->lexeme = lexeme;
     tkn->type = type;
     tkn->col = col;
     tkn->row = row;
     if (type == IDENTIFIER || search(buffer, type) == 0)
     {
          // printf("< %s | %d | %d | %s >\n", buffer, row, col,
types[type]);
          int index = hash(strlen(buffer));
          Node n = (Node)malloc(sizeof(struct node));
          char *str = (char *)calloc(strlen(buffer) + 1,
sizeof(char));
          strcpy(str, buffer);
          n->cur = str;
          n->next = NULL;
          n->row = row;
          n->col = col;
          n->type = type;
          if (hashTable[index] == NULL)
          {
               hashTable[index] = n;
               return tkn;
          }
          Node cur = hashTable[index];
          while (cur->next != NULL)
          {
               cur = cur->next;
          }
          cur->next = n;
     }
     return tkn;
}
#endif
```

tables.h :

```
#ifndef __TABLES_H__
#define __TABLES_H__
#define MAX_SIZE 20
typedef struct node *Node;
typedef struct symbol *Symbol;
typedef struct token *Token;
Node hashTable[MAX_SIZE]; // hash table
Symbol st[MAX_SIZE];       // symbol table
#endif
```

removeExcess.h :

```
#ifndef __REMOVEEXCESS_H__
#define __REMOVEEXCESS_H__
int removeExcess(char *fileName)
{ // to remove spaces, tabs and comments
     FILE *fa, *fb;
     int ca, cb;
     fa = fopen(fileName, "r");
     if (fa == NULL)
     {
          printf("Cannot open file \n");
          exit(0);
     }
     fb = fopen("space_output.c", "w");
     ca = getc(fa);
     while (ca != EOF)
     {
          if (ca == ' ' || ca == '\t')
          {
               putc(' ', fb);
               while (ca == ' ' || ca == '\t')
                    ca = getc(fa);
          }
          if (ca == '/')
          {
               cb = getc(fa);
               if (cb == '/')
               {
                    while (ca != '\n')
                         ca = getc(fa);
               }
               else if (cb == '*')
               {
                    do
                    {
                         while (ca != '*')
                              ca = getc(fa);
                         ca = getc(fa);
                    } while (ca != '/');
               }
               else
               {
                    putc(ca, fb);
                    putc(cb, fb);
               }
          }
          else
```

```
               putc(ca, fb);
          ca = getc(fa);
     }
     putc('$', fb);
     fclose(fa);
     fclose(fb);
     return 0;
}
#endif
```

removePreprocess.h :

```
#ifndef __REMOVEPREPROCESS_H__
#define __REMOVEPREPROCESS_H__
int removePreprocess()
{ // to ignore preprocessor directives
     FILE *finp = fopen("space_output.c", "r");
     char c = 0;
     char buffer[100];
     buffer[0] = '\0';
     int i = 0;
     char *includeStr = "include", *defineStr = "define", *mainStr
= "main";
     int mainFlag = 0, row = 1;
     while (c != EOF)
     {
          c = fgetc(finp);
          if (c == '#' && mainFlag == 0)
          {
               c = 'a';
               while (isalpha(c) != 0)
               {
                    c = fgetc(finp);
                    buffer[i++] = c;
               }
               buffer[i] = '\0';
               if (strstr(buffer, includeStr) != NULL ||
strstr(buffer, defineStr) != NULL)
               {
                    row++;
                    while (c != '\n')
                    {
                         c = fgetc(finp);
                    }
               }
               else
               {
                    for (int j = 0; j < i; j++)
                         ;
                    while (c != '\n')
                    {
                         c = fgetc(finp);
                    }
```

```
                }
                i = 0;
                buffer[0] = '\0';
            }
            else
            {
                if (mainFlag == 0)
                {
                    buffer[i++] = c;
                    buffer[i] = '\0';
                    if (strstr(buffer, mainStr) != NULL)
                    {
                        mainFlag = 1;
                    }
                }
                if (c == ' ' || c == '\n')
                {
                    buffer[0] = '\0';
                    i = 0;
                }
            }
        }
        fclose(finp);
        return row;
}
#endif
```

getNextToken.h :

```
#ifndef __GETNEXTTOKEN_H__
#define __GETNEXTTOKEN_H__
Token getNextToken(FILE *finp, int *row_pointer, int *col_pointer,
char data_type_buffer[], char *c)
{
        char buffer[100];
        int i = 0, col;
        Token tkn = NULL;
        if (isalpha(*c) != 0 || *c == '_')
        {
            buffer[i++] = *c;
            col = (*col_pointer);
            while (isalpha(*c) != 0 || *c == '_' || isdigit(*c) != 0)
            {
                *c = fgetc(finp);
                (*col_pointer)++;
                if (isalpha(*c) != 0 || *c == '_' || isdigit(*c) !=
0)
                    buffer[i++] = *c;
            }
            buffer[i] = '\0';
            if (isdatatype(buffer) == 1)
            {
                strcpy(data_type_buffer, buffer);
```

```
               tkn = insert(buffer, (*row_pointer), col - 1,
KEYWORD); // data type
          }
          else if (iskeyword(buffer) == 1)
          {
               tkn = insert(buffer, (*row_pointer), col - 1,
KEYWORD); // keyword
          }
          else
          {
               tkn = insert(buffer, (*row_pointer), col - 1,
IDENTIFIER); // identifier
               if (*c == '(')
                    insert_symbol(buffer, "function", *row_pointer,
col - 1);
               else
                    insert_symbol(buffer, data_type_buffer,
*row_pointer, col - 1);
               // data_type_buffer[0] = '\0';
          }
          i = 0;
          if (*c == '\n')
               (*row_pointer)++, (*col_pointer) = 1;
          buffer[0] = '\0';
     }
     else if (isdigit(*c) != 0)
     {
          buffer[i++] = *c;
          col = (*col_pointer);
          while (isdigit(*c) != 0 || *c == '.')
          {
               *c = fgetc(finp);
               (*col_pointer)++;
               if (isdigit(*c) != 0 || *c == '.')
                    buffer[i++] = *c;
          }
          buffer[i] = '\0';
          tkn = insert(buffer, (*row_pointer), col - 1,
NUMERIC_CONSTANT); // numerical constant
          i = 0;
          if (*c == '\n')
               (*row_pointer)++, (*col_pointer) = 1;
          buffer[0] = '\0';
     }
     else if (*c == '\"')
     {
          col = (*col_pointer);
          buffer[i++] = *c;
          *c = 0;
          while (*c != '\"')
          {
               *c = fgetc(finp);
               (*col_pointer)++;
               buffer[i++] = *c;
          }
          buffer[i] = '\0';
          tkn = insert(buffer, (*row_pointer), col - 1,
STRING_LITERAL); // string literals
          buffer[0] = '\0';
          i = 0;
          *c = fgetc(finp);
          (*col_pointer)++;
     }
     else if (*c == '\'')
     {
          col = (*col_pointer);
          buffer[i++] = *c;
          *c = 0;
          *c = fgetc(finp);
          (*col_pointer)++;
          buffer[i++] = *c;
          if (*c == '\\')
          {
               *c = fgetc(finp);
               (*col_pointer)++;
               buffer[i++] = *c;
          }
          *c = fgetc(finp);
          (*col_pointer)++;
          buffer[i++] = *c;
          buffer[i] = '\0';
          tkn = insert(buffer, (*row_pointer), col - 1,
CHARACTER_CONSTANT); // character constants
          buffer[0] = '\0';
          i = 0;
          *c = fgetc(finp);
          (*col_pointer)++;
     }
     else
     {
          col = (*col_pointer);
          if (*c == '=')
          { // relational and logical operators
               *c = fgetc(finp);
               (*col_pointer)++;
               if (*c == '=')
               {
                    tkn = insert("==", (*row_pointer), col - 1,
RELATIONAL_OPERATOR);
               }
               else
               {
                    tkn = insert("=", (*row_pointer), col - 1,
RELATIONAL_OPERATOR);
                    fseek(finp, -1, SEEK_CUR);
                    (*col_pointer)--;
               }
          }
          else if (*c == '>' || *c == '<' || *c == '!')
          {
               char temp = *c;
               *c = fgetc(finp);
               (*col_pointer)++;
               if (*c == '=')
               {
                    char temp_str[3] = {
                         temp,
                         '=',
                         '\0'};
                    tkn = insert(temp_str, (*row_pointer), col - 1,
RELATIONAL_OPERATOR);
               }
               else
               {
                    char temp_str[2] = {
                         temp,
                         '\0'};
                    tkn = insert(temp_str, (*row_pointer), col - 1,
RELATIONAL_OPERATOR);
                    fseek(finp, -1, SEEK_CUR);
                    (*col_pointer)--;
               }
          }
          else if (isbracket(*c) == 1)
          { // parentheses and special symbols
               char temp_string[2] = {
                    *c,
                    '\0'};
               tkn = insert(temp_string, (*row_pointer), col - 1,
BRACKET);
          }
          else if (isspecial(*c) == 1)
          { // parentheses and special symbols
               char temp_string[2] = {
                    *c,
                    '\0'};
               tkn = insert(temp_string, (*row_pointer), col - 1,
SPECIAL_SYMBOL);
          }
          else if (isoperator(*c) == 1)
          { // operators
               char temp = *c;
               *c = fgetc(finp);
               (*col_pointer)++;
               if (*c == '=' || (temp == '+' && *c == '+') || (temp
== '-' && *c == '-'))
               {
                    char temp_string[3] = {
                         temp,
                         *c,
                         '\0'};
                    tkn = insert(temp_string, (*row_pointer), col -
1, OPERATOR);
               }
               else
               {
                    char temp_String[2] = {
                         temp,
                         '\0'};
                    tkn = insert(temp_String, (*row_pointer), col -
1, OPERATOR);
                    fseek(finp, -1, SEEK_CUR);
                    (*col_pointer)--;
               }
          }
          else if (*c == '\n') // new line
               (*row_pointer)++, (*col_pointer) = 1;
          else if (*c == '$')
          {
               Token eof = (Token)malloc(sizeof(struct token));
               eof->lexeme = "EOF";
               return eof;
          }
          *c = fgetc(finp);
          (*col_pointer)++;
     }
     return tkn;
}
#endif
```

rdParser.h :

```
#ifndef __RDPARSER_H__
#define __RDPARSER_H__
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
#define false 0
#define true 1
#include "removePreprocess.h"
#include "removeExcess.h"
#include "constants.h"
#include "structs.h"
#include "utils.h"
#include "tables.h"
#include "hash.h"
#include "getNextToken.h"
#endif
```

main.c :

```
#include "rdParser.h"

int row, col_global;
char data_type_buffer[100], c = 0;
FILE *finp;
Token tkn = NULL;
int prev_flag = false;

enum NON_TERMINALS { // types for non terminals.
     PROGRAM,
     DECLARATIONS,
     ASSIGNSTAT,
     ASSIGNSTATPRIME,
     IDENTIFIERLIST,
     IDENTIFIERLISTPRIME
};

char first[][4][20] = {{"int"}, {"int", "char", "double",
"float"}, {"id"}, {"id", "num"}, {"id"}, {","}};
char follow[][2][20] = {{"$"}, {"id"}, {"id", "num"}, {"return"},
{";"}, {";"}};

int firstSz[] = {1, 4, 1, 2, 1, 1};
int followSz[] = {1, 1, 2, 1, 1, 1};

int search_first(enum NON_TERMINALS val, char* buffer, enum TYPE
type){
     if(type == IDENTIFIER){
          return search_symbol(buffer) != -1 && search_first(val,
"id", KEYWORD);
     }
     if(type == NUMERIC_CONSTANT){
          return search_first(val, "num", KEYWORD);
     }
     for(int i = 0; i<firstSz[val]; i++){
          if(strcmp(buffer, first[val][i]) == 0){
               return 1;
          }
     }
     return 0;
}

int search_follow(enum NON_TERMINALS val, char* buffer, enum TYPE
type){
     if(type == IDENTIFIER){
          return search_follow(val, "id", KEYWORD) &&
search_symbol(buffer) != -1;
     }
     if(type == NUMERIC_CONSTANT){
          return search_follow(val, "num", KEYWORD);
     }
```

```
	for(int i = 0; i<followSz[val]; i++){
		if(strcmp(buffer, follow[val][i]) == 0){
			return 1;
		}
	}
	return 0;
}

void Program();
void Declarations();
void DataType();
void Identifier();
void IdentifierPrime();
void AssignStat();
void AssignStatPrime();
void success();
void failure(char *msg);

void get()
{
	tkn = prev_flag == true ? tkn : NULL;
	while (tkn == NULL)
	{
		tkn = getNextToken(finp, &row, &col_global,
data_type_buffer, &c);
	}
	if (strcmp(tkn->lexeme, "EOF") == 0)
	{
		failure("End of file encountered!");
	}
	prev_flag = false;
}

int main(int argn, char *args[])
{
	if (argn < 2)
	{
		printf("No file specified, exiting ...\n");
		return 0;
	}
	removeExcess(args[1]);
	row = removePreprocess();
	enum TYPE type;
	for (int i = 0; i < MAX_SIZE; i++)
		hashTable[i] = NULL;
	finp = fopen("space_output.c", "r");
	if (finp == NULL)
	{
		printf("Cannot Find file, exiting ... ");
		return 0;
	}
	int temp_row = --row;
	while (temp_row > 0)
```

```
     {
          c = fgetc(finp);
          if (c == '\n')
               temp_row--;
     }
     row;
     col_global = 1;
     get();
     prev_flag = true;
     if(search_first(PROGRAM, tkn->lexeme, tkn->type) == 1){
          Program();
     }
     else{
          failure("No Return type found!");
     }
     printf("\nSymbol Table : \n\n");
     display_st();
     printf("\n");
     return 0;
}

void Program()
{
     get();
     if (strcmp(tkn->lexeme, "int") == 0)
     {
          get();
          if (strcmp(tkn->lexeme, "main") == 0)
          {
               get();
               if (strcmp(tkn->lexeme, "(") == 0)
               {
                    get();
                    if (strcmp(tkn->lexeme, ")") == 0)
                    {
                         get();
                         if (strcmp(tkn->lexeme, "{") == 0)
                         {
                              get();
                              if(search_first(DECLARATIONS, tkn-
>lexeme, tkn->type) == 1){
                                   prev_flag = true;
                                   Declarations();

                              }
                              else{
                                   failure("Data Type expected!");
                              }
                              get();
                              if(search_first(ASSIGNSTAT, tkn-
>lexeme, tkn->type) == 1){
                                   prev_flag = true;
                                   AssignStat();
                              }
                              else{
                                   failure("Invalid Identifier!");
                              }
                              get();
                              if (strcmp(tkn->lexeme, "return") ==
0)
                              {
                                   get();
                                   if(tkn->type ==
NUMERIC_CONSTANT){
                                        get();
                                        if(strcmp(tkn->lexeme, ";")
== 0){
                                             get();
                                             if(strcmp(tkn->lexeme,
"}") == 0){
                                             success();
                                             }
                                             else{
                                                  failure("No
closing curly braces found!");
                                             }
                                        }
                                        else{
                                             failure("No Semi-Colon
found!");
                                        }
                                   }
                                   else{
                                        failure("Numeric Value
Expected!");
                                   }
                              }
                              else
                              {
                                   failure("No return statement
found!");
                              }
                         }
                         else
                         {
                              failure("No starting curly bracket
found!");
                         }
                    }
                    else
                    {
                         failure("No function closing parentheses
found!");
                    }
               }
               else
               {
                    failure("No function starting parentheses 
found!");
               }
          }
          else
          {
               failure("No main found!");
          }
     }
     else
     {
          failure("No return type found!");
     }
}

void Declarations()
{
     get();
     if (isdatatype(tkn->lexeme))
     {
          get();
          if(search_first(IDENTIFIERLIST, tkn->lexeme, tkn->type) 
== 1){
               prev_flag = true;
               Identifier();
          }
          else{
               failure("Identifier expected!");
          }
          get();
          if (strcmp(tkn->lexeme, ";") == 0)
          {
               get();
               prev_flag = true;
               if(search_first(DECLARATIONS, tkn->lexeme, tkn-
>type) == 1){
                    Declarations();
               }
               else if(search_follow(DECLARATIONS, tkn->lexeme, 
tkn->type) == 0){
                    failure("Invalid Identifier");
               }
          }
          else
          {
               failure("Semi Colon Expected!");
          }
     }
     else
     {
          prev_flag = true;
     }
```

```
}

void DataType()
{
     get();
     if (isdatatype(tkn->lexeme) == 0)
     {
          failure("Data Type Expected!");
     }
}

void Identifier()
{
     get();
     if (search_symbol(tkn->lexeme) != -1)
     {
          get();
          prev_flag = true;
          if(search_first(IDENTIFIERLISTPRIME, tkn->lexeme, tkn-
>type) == 1){
               IdentifierPrime();
          }
          else if(search_follow(IDENTIFIERLISTPRIME, tkn->lexeme,
tkn->type) == 0){
               failure(", or ; expected");
          }
     }
     else
     {
          failure("Invalid Identifier!");
     }
}

void IdentifierPrime()
{
     get();
     if (strcmp(tkn->lexeme, ",") == 0)
     {
          get();
          if(search_first(IDENTIFIERLIST, tkn->lexeme, tkn->type)
== 1){
               prev_flag = true;
               Identifier();
          }
          else{
               failure("Invalid Identifier!");
          }
     }
     else
     {
          prev_flag = true;
     }
}
```

```
void AssignStat()
{
     get();
     if (search_symbol(tkn->lexeme) != -1)
     {
          get();
          if (strcmp(tkn->lexeme, "=") == 0)
          {
               get();
               if(search_first(ASSIGNSTATPRIME, tkn->lexeme, tkn-
>type) == 1){
                    prev_flag = true;
                    AssignStatPrime();
               }
               else{
                    failure("Invalid Identifier or numeric value
expected!");
               }
          }
          else
          {
               failure("= sign not found!");
          }
     }
     else
     {
          failure("Invalid Identifier!");
     }
}

void AssignStatPrime()
{
     get();
     if (tkn->type == IDENTIFIER && search_symbol(tkn->lexeme) != -
1)
     {
          get();
          if (strcmp(tkn->lexeme, ";") != 0)
          {
               failure("Semi-Colon Expected!");
          }
     }
     else if (tkn->type == NUMERIC_CONSTANT)
     {
          get();
          if (strcmp(tkn->lexeme, ";") != 0)
          {
               failure("Semi-Colon Expected!");
          }
     }
     else
     {
            failure("Invalid Assignment!");
      }
}

void failure(char *msg)
{
      printf("\n***NOT ACCEPTED***\n%s Row : %d, Col : %d\n", msg,
tkn->row, tkn->col);
      exit(0);
}

void success()
{
      printf("\n###ACCEPTED###\n");
}
```

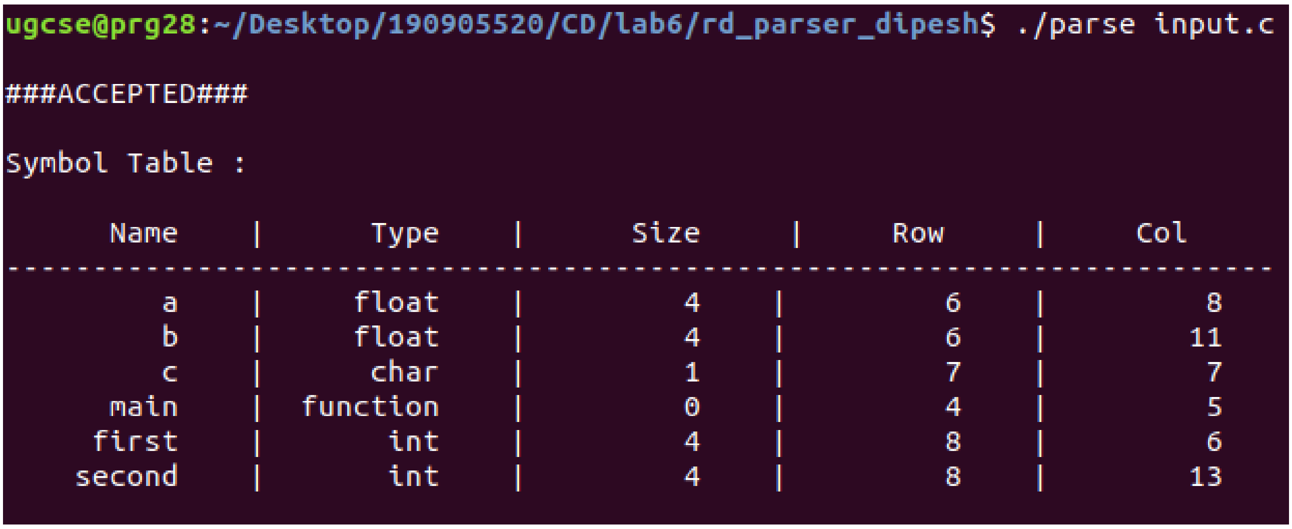
Case 1 : Accepted.

input.c :

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

int main()
{
	float a, b;
	char c;
	int first, second;
	a = b;
	return 0;
}
```

Output :



```
ugcse@prg28:~/Desktop/190905520/CD/lab6/rd_parser_dipesh$ ./parse input.c

###ACCEPTED###

Symbol Table :

      Name     |       Type     |       Size      |      Row      |      Col
-----------------------------------------------------------------------------------
         a     |      float     |          4     |          6     |          8
         b     |      float     |          4     |          6     |         11
         c     |       char     |          1     |          7     |          7
      main     |   function     |          0     |          4     |          5
     first     |        int     |          4     |          8     |          6
    second     |        int     |          4     |          8     |         13
```

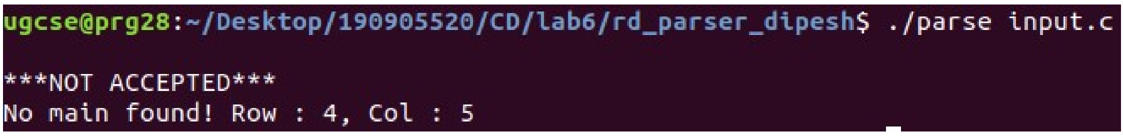
Case 2 : Missing Main function.

input.c :

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

int {
	float a, b;
	char c;
	int first, second;
	a = b;
	return 0;
}
```

Output :



```
ugcse@prg28:~/Desktop/190905520/CD/lab6/rd_parser_dipesh$ ./parse input.c

***NOT ACCEPTED***
No main found! Row : 4, Col : 5
```

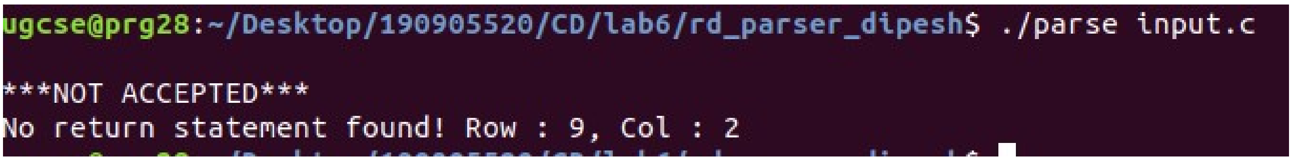
Case 3 : Missing return Statement.

input.c :

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

int main(){
     float a, b;
     char c;
     int first, second;
     a = b;
      0;
}
```

Output :



```
ugcse@prg28:~/Desktop/190905520/CD/lab6/rd_parser_dipesh$ ./parse input.c

***NOT ACCEPTED***
No return statement found! Row : 9, Col : 2
```

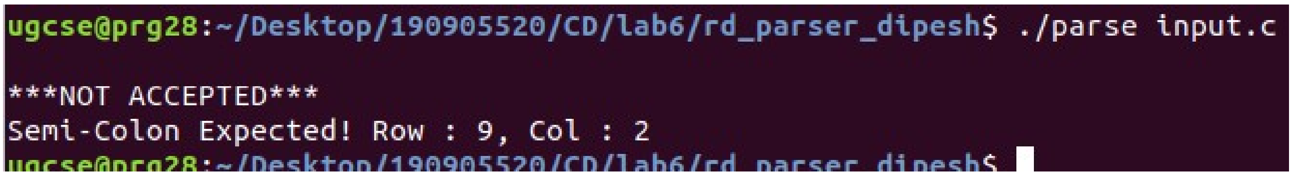
Case 4 : Missing semi colon.

input.c :

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

int main(){
     float a, b;
     char c;
     int first, second;
     a = b
     return 0;
}
```

Output :



```
ugcse@prg28:~/Desktop/190905520/CD/lab6/rd_parser_dipesh$ ./parse input.c

***NOT ACCEPTED***
Semi-Colon Expected! Row : 9, Col : 2
ugcse@prg28:~/Desktop/190905520/CD/lab6/rd_parser_dipesh$
```

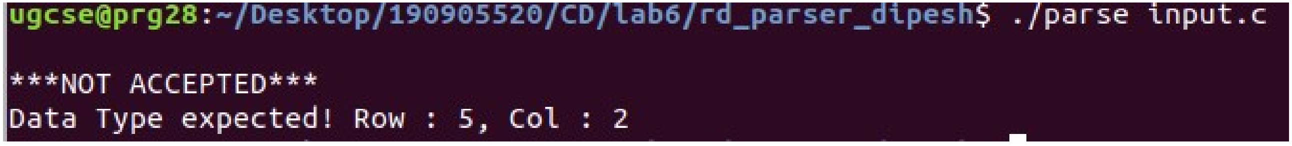
Case 5 : Missing data type in declaration

input.c :

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

int main(){
      a, b;
     char c;
     int first, second;
     a = b
     return 0;
}
```

Output :



```
ugcse@prg28:~/Desktop/190905520/CD/lab6/rd_parser_dipesh$ ./parse input.c

***NOT ACCEPTED***
Data Type expected! Row : 5, Col : 2
```

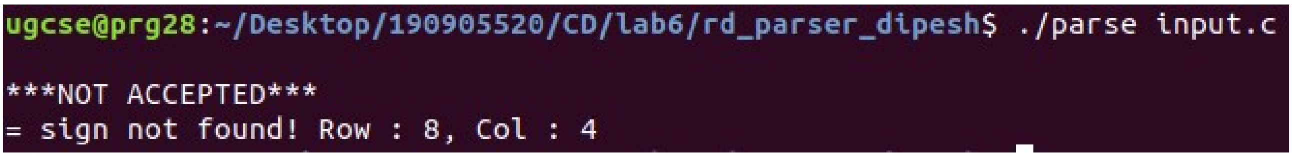
Case 6 : Missing = sign.

Input.c :

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

int main(){
     float a, b;
     char c;
     int first, second;
     a  b;
     return 0;
}
```

Output :



```
ugcse@prg28:~/Desktop/190905520/CD/lab6/rd_parser_dipesh$ ./parse input.c

***NOT ACCEPTED***
= sign not found! Row : 8, Col : 4
```