Ayush Goyal
190905522 CSE D 62

Compiler Design Week 6,7,8 Lab 7,8,9:

The Inputs and Outputs of all the labs have been displayed below, followed by the finally appeneded code which was produced at the end of all three labs.

Week 6 Lab 7: RD Parser for Declaration Statements

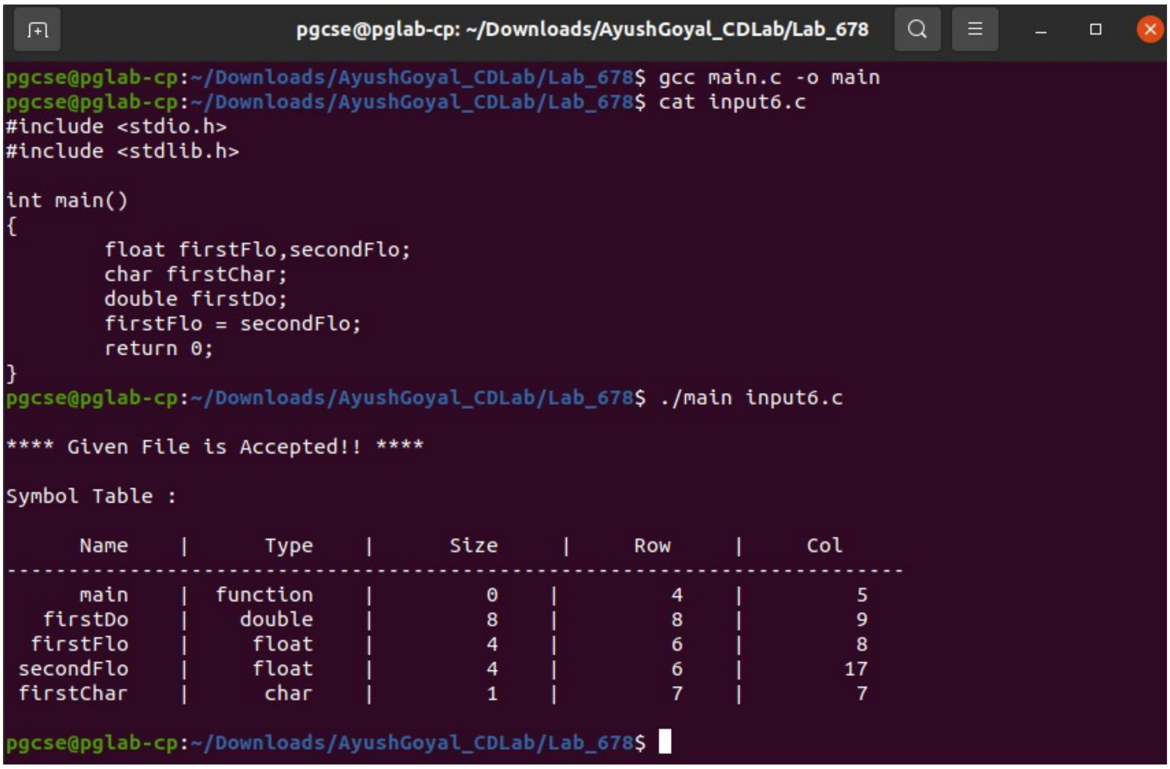
Lab Excercise:

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 | ϵ
data-type → int | char
identifier-list → id | id, identifier-list
assign_stat → id=id; | id = num;

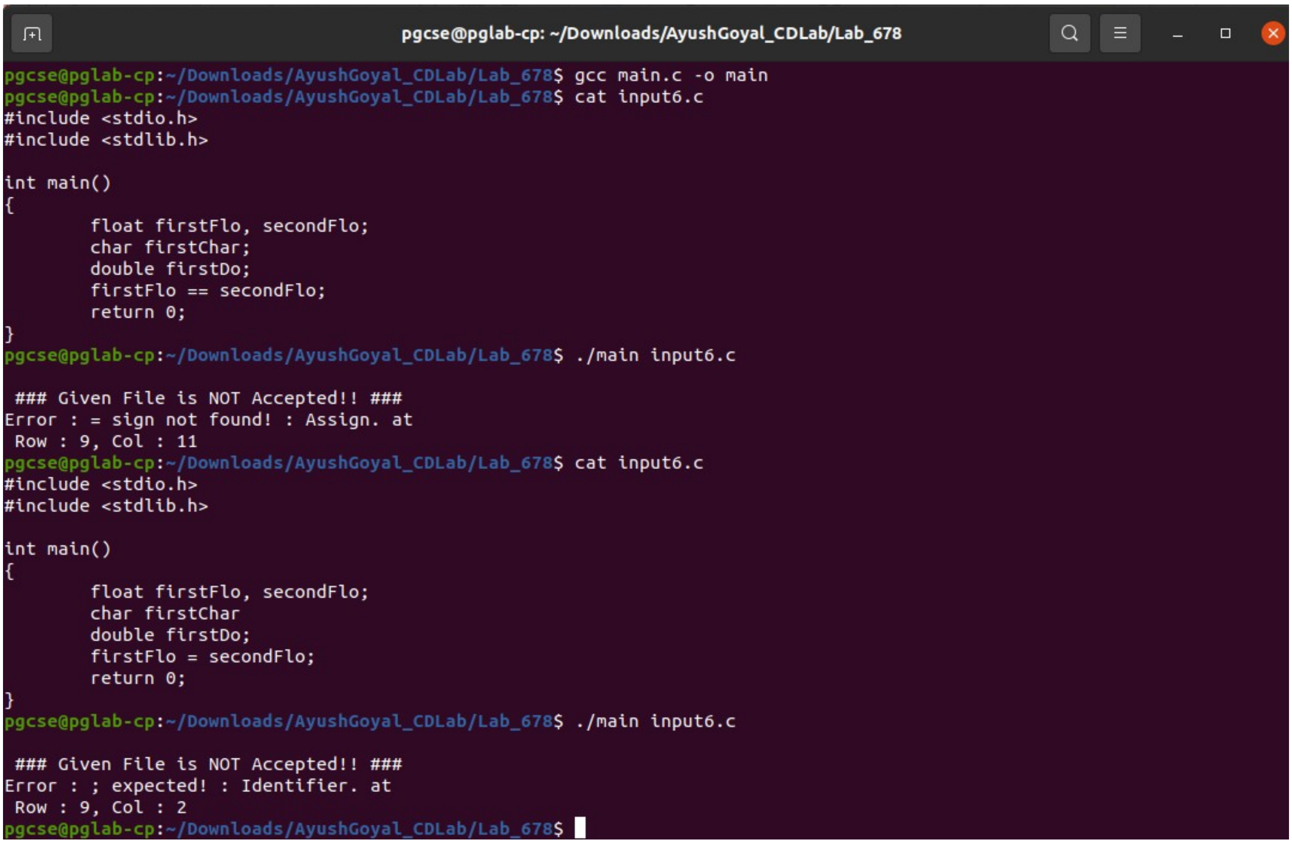
Output:

The file input is “input6.c” and the contents are displayed below:



```
pgcse@pglab-cp:~/Downloads/AyushGoyal_CDLab/Lab_678$ gcc main.c -o main
pgcse@pglab-cp:~/Downloads/AyushGoyal_CDLab/Lab_678$ cat input6.c
#include <stdio.h>
#include <stdlib.h>

int main()
{
        float firstFlo,secondFlo;
        char firstChar;
        double firstDo;
        firstFlo = secondFlo;
        return 0;
}
pgcse@pglab-cp:~/Downloads/AyushGoyal_CDLab/Lab_678$ ./main input6.c

**** Given File is Accepted!! ****

Symbol Table :

      Name    |      Type     |      Size      |      Row      |      Col
--------------------------------------------------------------------------
      main    |  function     |         0     |         4     |         5
   firstDo    |    double     |         8     |         8     |         9
  firstFlo    |     float     |         4     |         6     |         8
 secondFlo    |     float     |         4     |         6     |        17
 firstChar    |      char     |         1     |         7     |         7

pgcse@pglab-cp:~/Downloads/AyushGoyal_CDLab/Lab_678$
```

When we introduce different kinds of errors in our input file, “input6.c” and then feed the input file to our parser, we can see that the error is handled:



```
pgcse@pglab-cp:~/Downloads/AyushGoyal_CDLab/Lab_678$ gcc main.c -o main
pgcse@pglab-cp:~/Downloads/AyushGoyal_CDLab/Lab_678$ cat input6.c
#include <stdio.h>
#include <stdlib.h>

int main()
{
        float firstFlo, secondFlo;
        char firstChar;
        double firstDo;
        firstFlo == secondFlo;
        return 0;
}
pgcse@pglab-cp:~/Downloads/AyushGoyal_CDLab/Lab_678$ ./main input6.c

 ### Given File is NOT Accepted!! ###
Error : = sign not found! : Assign. at
 Row : 9, Col : 11
pgcse@pglab-cp:~/Downloads/AyushGoyal_CDLab/Lab_678$ cat input6.c
#include <stdio.h>
#include <stdlib.h>

int main()
{
        float firstFlo, secondFlo;
        char firstChar
        double firstDo;
        firstFlo = secondFlo;
        return 0;
}
pgcse@pglab-cp:~/Downloads/AyushGoyal_CDLab/Lab_678$ ./main input6.c

 ### Given File is NOT Accepted!! ###
Error : ; expected! : Identifier. at
 Row : 9, Col : 2
pgcse@pglab-cp:~/Downloads/AyushGoyal_CDLab/Lab_678$
```

Thus, we can confirm the acceptance of the required grammar in the question and the symbol table is also displayed for the same. Whenever it encounters any error, we have displayed the problem along with its location the row column format as shown in the output screenshots.

Week 7 Lab 8: RD Parser for Array Declarations and Expression Statements

Lab Excercise:

Design the recursive descent parser to parse array declarations and expression statements with error reporting. Subset of grammar 7.1 is as follows:

Program → main () { declarations statement-list }

identifier-list → id | id, identifier-list | id[number] , identifier-list | id[number]

statement_list → statement statement_list | ∈

statement → assign-stat;

assign_stat → id = expn

expn→ simple-expn eprime

eprime→relop simple-expn| ∈

simple-exp→ term seprime

seprime→addop term seprime | ∈

term → factor tprime

tprime → mulop factor tprime | ∈

factor → id | num

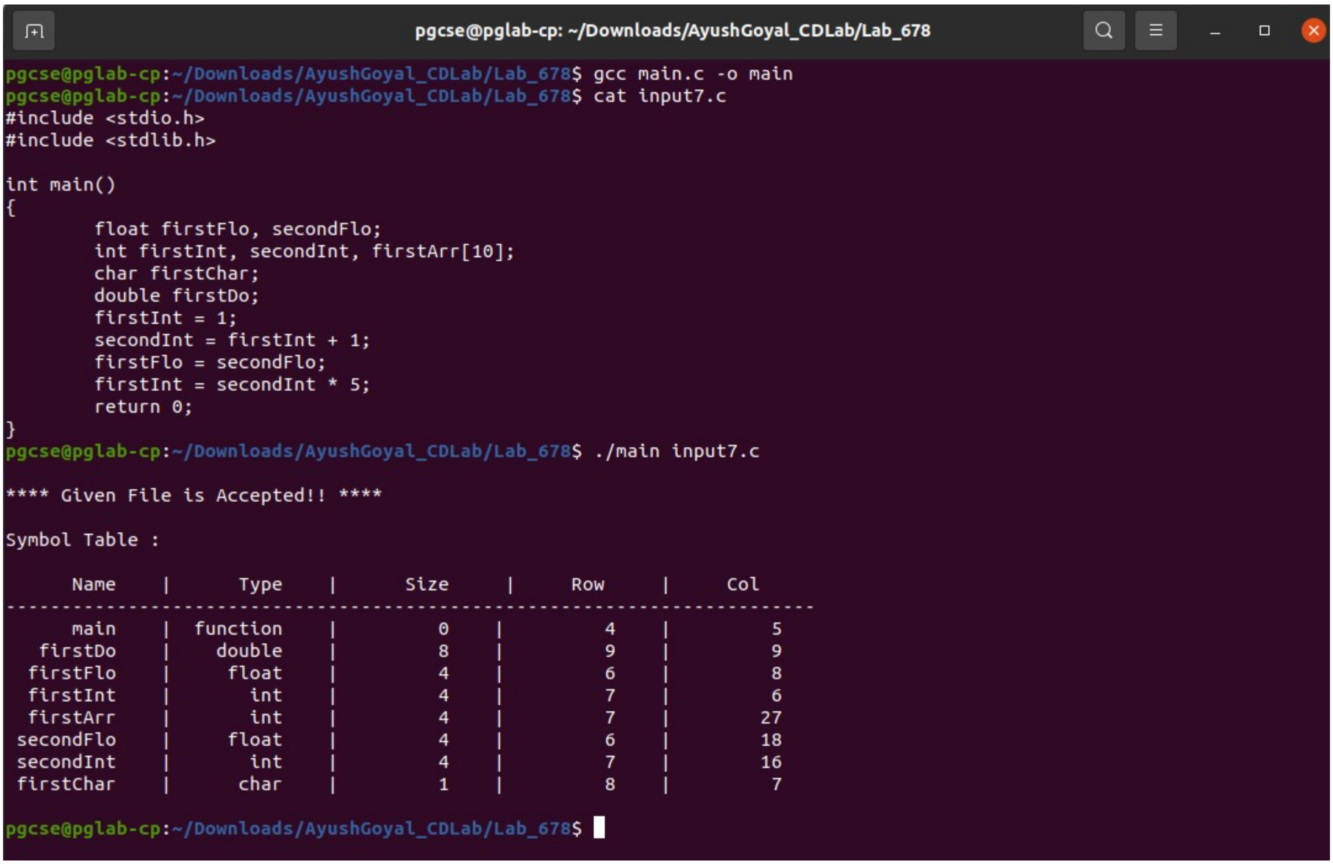
relop → = = | != | <= | >= | > | <

addop → + | -

mulop → * | / | %

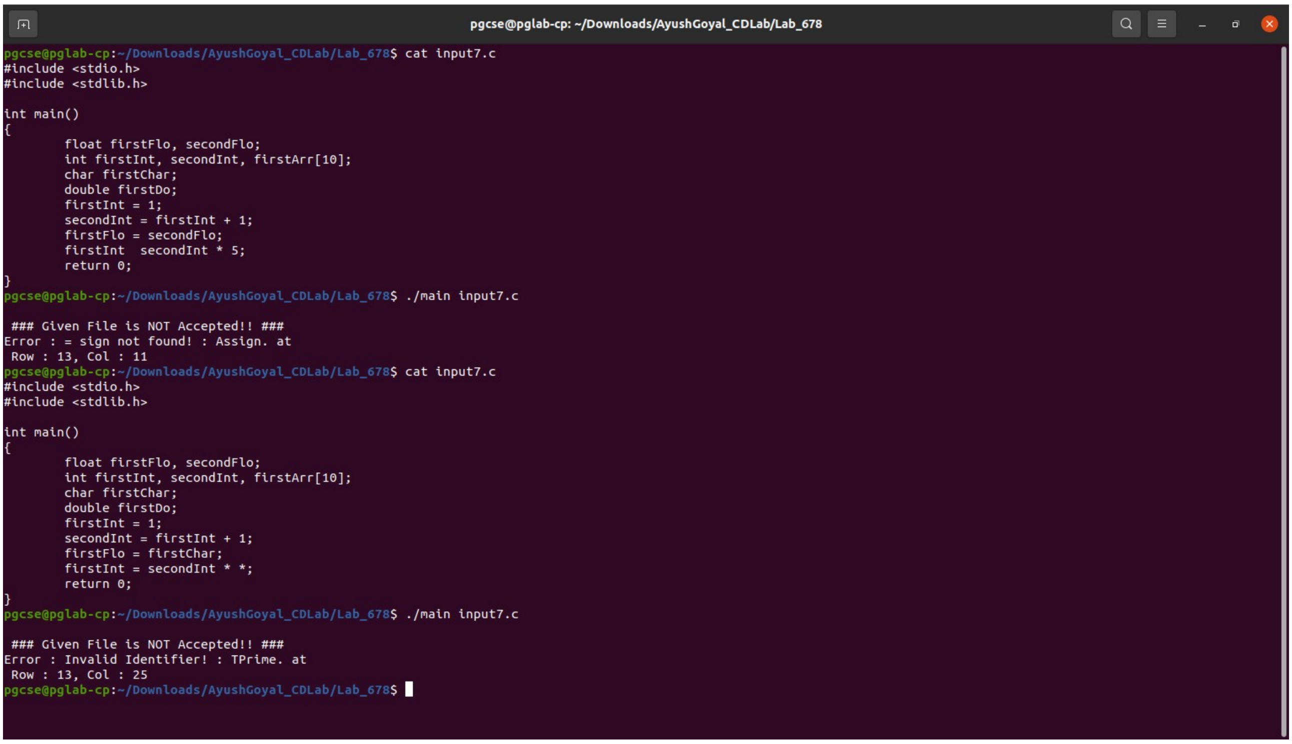
Output:

The file input is “input7.c” and the contents are displayed below:



```
pgcse@pglab-cp:~/Downloads/AyushGoyal_CDLab/Lab_678$ gcc main.c -o main
pgcse@pglab-cp:~/Downloads/AyushGoyal_CDLab/Lab_678$ cat input7.c
#include <stdio.h>
#include <stdlib.h>

int main()
{
        float firstFlo, secondFlo;
        int firstInt, secondInt, firstArr[10];
        char firstChar;
        double firstDo;
        firstInt = 1;
        secondInt = firstInt + 1;
        firstFlo = secondFlo;
        firstInt = secondInt * 5;
        return 0;
}
pgcse@pglab-cp:~/Downloads/AyushGoyal_CDLab/Lab_678$ ./main input7.c

**** Given File is Accepted!! ****

Symbol Table :

      Name    |       Type    |       Size     |       Row     |      Col
-------------------------------------------------------------------------
      main    |   function    |          0    |          4    |          5
   firstDo    |     double    |          8    |          9    |          9
  firstFlo    |      float    |          4    |          6    |          8
  firstInt    |        int    |          4    |          7    |          6
  firstArr    |        int    |          4    |          7    |         27
 secondFlo    |      float    |          4    |          6    |         18
 secondInt    |        int    |          4    |          7    |         16
 firstChar    |       char    |          1    |          8    |          7

pgcse@pglab-cp:~/Downloads/AyushGoyal_CDLab/Lab_678$ 
```

When we introduce different kinds of errors in our input file, “input7.c” and then feed the input file to our parser, we can see that the error is handled:



```
pgcse@pglab-cp:~/Downloads/AyushGoyal_CDLab/Lab_678$ cat input7.c
#include <stdio.h>
#include <stdlib.h>

int main()
{
        float firstFlo, secondFlo;
        int firstInt, secondInt, firstArr[10];
        char firstChar;
        double firstDo;
        firstInt = 1;
        secondInt = firstInt + 1;
        firstFlo = secondFlo;
        firstInt  secondInt * 5;
        return 0;
}
pgcse@pglab-cp:~/Downloads/AyushGoyal_CDLab/Lab_678$ ./main input7.c

 ### Given File is NOT Accepted!! ###
Error : = sign not found! : Assign. at
 Row : 13, Col : 11
pgcse@pglab-cp:~/Downloads/AyushGoyal_CDLab/Lab_678$ cat input7.c
#include <stdio.h>
#include <stdlib.h>

int main()
{
        float firstFlo, secondFlo;
        int firstInt, secondInt, firstArr[10];
        char firstChar;
        double firstDo;
        firstInt = 1;
        secondInt = firstInt + 1;
        firstFlo = firstChar;
        firstInt = secondInt * *;
        return 0;
}
pgcse@pglab-cp:~/Downloads/AyushGoyal_CDLab/Lab_678$ ./main input7.c

 ### Given File is NOT Accepted!! ###
Error : Invalid Identifier! : TPrime. at
 Row : 13, Col : 25
pgcse@pglab-cp:~/Downloads/AyushGoyal_CDLab/Lab_678$ 
```

Thus, we can confirm the acceptance of the required grammar in the question and the symbol table is also displayed for the same. Whenever it encounters any error, we have displayed the problem along with its location the row column format as shown in the output screenshots.

Week 8 Lab 9: RD Parser for Decision Making and Looping Statements

Lab Excercise:

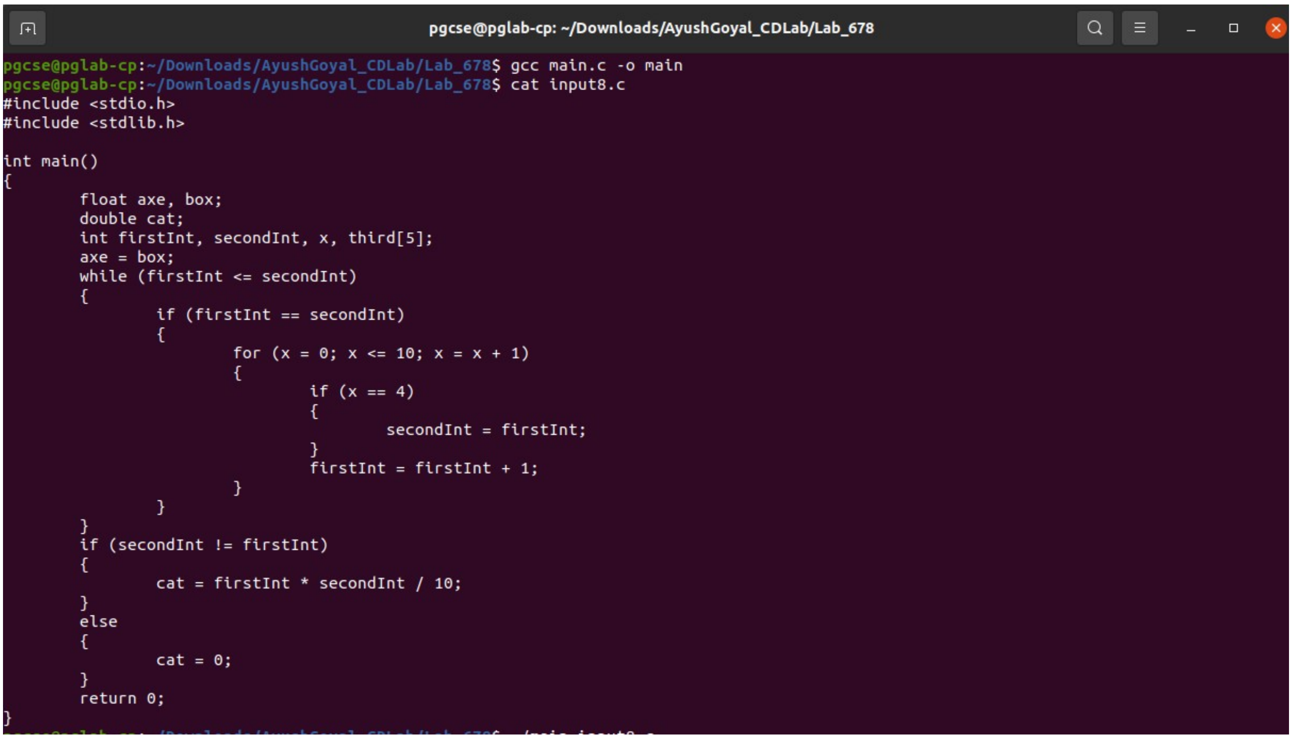
Modify the Recursive Descent parser implemented in the previous lab to parse decision making and looping statements with error reporting. Subset of grammar 7.1 is as follows:

statement → assign-stat; | decision_stat | looping-stat

decision-stat → if (expn) {statement_list} dprime

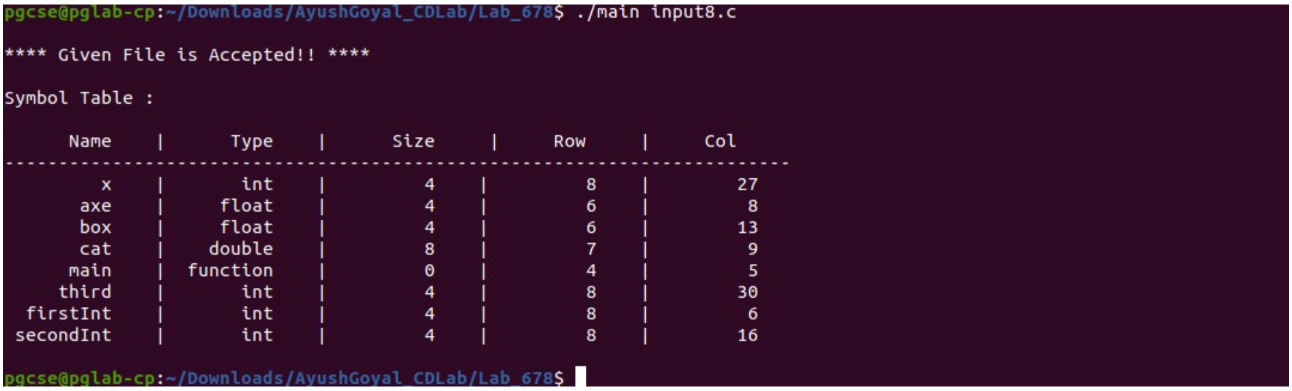
Output:

The file input is “input8.c” and the contents are displayed below:



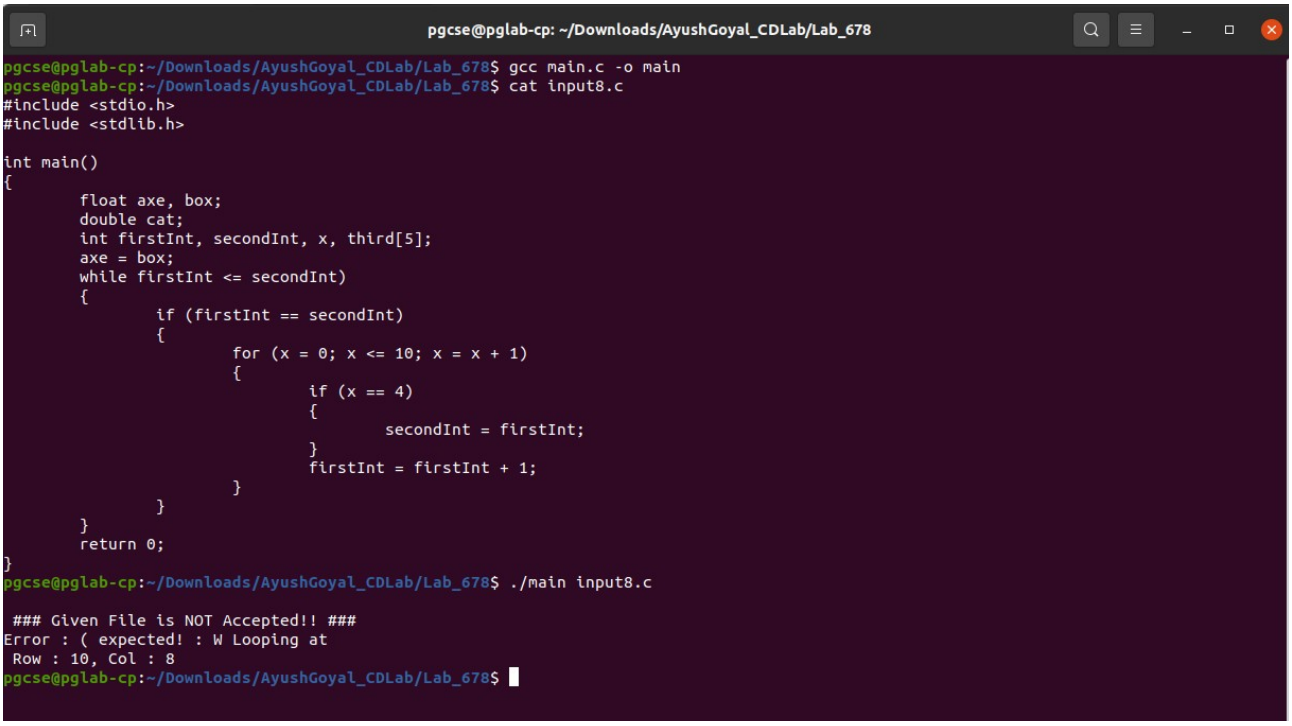
```
pgcse@pglab-cp:~/Downloads/AyushGoyal_CDLab/Lab_678$ gcc main.c -o main
pgcse@pglab-cp:~/Downloads/AyushGoyal_CDLab/Lab_678$ cat input8.c
#include <stdio.h>
#include <stdlib.h>

int main()
{
        float axe, box;
        double cat;
        int firstInt, secondInt, x, third[5];
        axe = box;
        while (firstInt <= secondInt)
        {
                if (firstInt == secondInt)
                {
                        for (x = 0; x <= 10; x = x + 1)
                        {
                                if (x == 4)
                                {
                                        secondInt = firstInt;
                                }
                                firstInt = firstInt + 1;
                        }
                }
        }
        if (secondInt != firstInt)
        {
                cat = firstInt * secondInt / 10;
        }
        else
        {
                cat = 0;
        }
        return 0;
}
pgcse@pglab-cp:~/Downloads/AyushGoyal_CDLab/Lab_678$ ./main input8.c
```

On execution, we can see that this file is getting accepted and the symbol table has been generated:



```
pgcse@pglab-cp:~/Downloads/AyushGoyal_CDLab/Lab_678$ ./main input8.c

**** Given File is Accepted!! ****

Symbol Table :

      Name    |       Type    |       Size     |       Row     |      Col
--------------------------------------------------------------------------------
         x    |        int    |          4    |          8    |         27
       axe    |      float    |          4    |          6    |          8
       box    |      float    |          4    |          6    |         13
       cat    |     double    |          8    |          7    |          9
      main    |   function    |          0    |          4    |          5
     third    |        int    |          4    |          8    |         30
  firstInt    |        int    |          4    |          8    |          6
 secondInt    |        int    |          4    |          8    |         16

pgcse@pglab-cp:~/Downloads/AyushGoyal_CDLab/Lab_678$
```

We can introduce any kind of errors in our input file, “input8.c” and then feed the input file to our parser, we can see that the error is handled:



```
pgcse@pglab-cp:~/Downloads/AyushGoyal_CDLab/Lab_678$ gcc main.c -o main
pgcse@pglab-cp:~/Downloads/AyushGoyal_CDLab/Lab_678$ cat input8.c
#include <stdio.h>
#include <stdlib.h>

int main()
{
        float axe, box;
        double cat;
        int firstInt, secondInt, x, third[5];
        axe = box;
        while firstInt <= secondInt)
        {
                if (firstInt == secondInt)
                {
                        for (x = 0; x <= 10; x = x + 1)
                        {
                                if (x == 4)
                                {
                                        secondInt = firstInt;
                                }
                                firstInt = firstInt + 1;
                        }
                }
        }
        return 0;
}
pgcse@pglab-cp:~/Downloads/AyushGoyal_CDLab/Lab_678$ ./main input8.c

 ### Given File is NOT Accepted!! ###
Error : ( expected! : W Looping at
 Row : 10, Col : 8
pgcse@pglab-cp:~/Downloads/AyushGoyal_CDLab/Lab_678$ 
```

****
****

FINAL CODE AFTER THE THREE LABS:

(All files have been named and the contents has been displayed as given)

“main.c”:

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
#define false 0
#define true 1
#include "Lexical/lexeme.h"
#include "Parser/rdParser.h"

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 ?: printf("token : %s\n", tkn->lexeme);
    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;
}
```

“constants.h”:

```
#ifndef __RDCONSTANTS_H__
#define __RDCONSTANTS_H__
enum NON_TERMINALS
{ // types for non terminals.
	PROGRAM,
	DECLARATIONS,
	DATA_TYPE,
	IDENTIFIERLIST,
	IDENTIFIERLISTPRIME,
	IDENTIFIERLISTPRIMEPRIME,
	STATEMENT_LIST,
	STATEMENT,
	ASSIGN_STAT,
	EXPN,
	EPRIME,
	SIMPLE_EXP,
	SEPRIME,
	TERM,
	TPRIME,
	FACTOR,
	DECISION,
	DPRIME,
	LOOPING,
	RELOP,
	ADDOP,
	MULOP
};

// ToDo: calculate the first and follow of the grammar

char first[][6][20] = {
	{"int"},
	{"int", "char", "double", "float"},
	{"int", "char", "double", "float"},
	{"id"},
	{",", "["},
	{","},
	{"id", "if", "while", "for"},
	{"id", "if", "while", "for"},
	{"id"},
	{"id", "num"},
	{"==", "!=", ">=", "<=", ">", "<"},
	{"id", "num"},
	{"+", "-"},
	{"id", "num"},
	{"*", "/", "%"},
	{"id", "num"},
	{"if"},
	{"else"},
	{"while", "for"},
        {"==", "!=", ">=", "<=", ">", "<"},
        {"+", "-"},
        {"*", "/", "%"},
};
char follow[][15][20] = {
        {"$"},
        {"return", "}", "id", "if", "while", "for"},
        {"id"},
        {";"},
        {";"},
        {";"},
        {"}"},
        {"id", "if", "while", "for", "}"},
        {";", ")"},
        {";", ")", "==", "!=", ">=", "<=", ">", "<"},
        {";", ")", "==", "!=", ">=", "<=", ">", "<"},
        {";", ")", "==", "!=", ">=", "<=", ">", "<", "id", "num"},
        {";", ")", "==", "!=", ">=", "<=", ">", "<", "id", "num"},
        {"+", "-", ";", ")", "==", "!=", ">=", "<=", ">", "<", "id", "num"},
        {"+", "-", ";", ")", "==", "!=", ">=", "<=", ">", "<", "id", "num"},
        {"*", "/", "%", "+", "-", ";", ")", "==", "!=", ">=", "<=", ">", "<", "id", "num"},
        {"id", "if", "while", "for", "}"},
        {"id", "if", "while", "for", "}"},
        {"id", "if", "while", "for", "}"},
        {"id", "num"},
        {"id", "num"},
        {"id", "num"}};

int firstSz[] = {1, 4, 4, 1, 2, 1, 4, 4, 1, 2, 6, 2, 2, 2, 3, 2, 1, 1, 2, 6, 2, 3};
int followSz[] = {1, 6, 1, 1, 1, 1, 1, 5, 2, 8, 8, 10, 10, 12, 12, 15, 5, 5, 5, 2, 2, 2};
#endif
```

“rdParser.h”:

```
#ifndef __RDPARSER_H__
#define __RDPARSER_H__
void get();
#include "constants.h"
#include "utils.h"
#include "Procedures/procedures.h"
#endif
```

“utils.h”:

```
#ifndef __RDUTILS_H__
#define __RDUTILS_H__
```

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

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 failure(char *msg)
{
	printf("\n ### Given File is NOT Accepted!! ###\nError : %s at\n Row : %d, Col : %d\n", 
msg, tkn->row, tkn->col);
	exit(0);
}
```

```
void success()
{
	printf("\n**** Given File is Accepted!! ****\n");
}

#endif
```

“Assign.h”:

```
#ifndef __ASSIGN_H__
#define __ASSIGN_H__
void AssignStat()
{
	get();
	if (search_symbol(tkn->lexeme) != -1)
	{
		get();
		if (strcmp(tkn->lexeme, "=") == 0)
		{
			get();
			prev_flag = true;
			if (search_first(EXPN, tkn->lexeme, tkn->type) == 1)
				Expn();
			else
				failure("Invalid Identifier or number : Assign.");
		}
		else
		{
			failure("= sign not found! : Assign.");
		}
	}
	else
	{
		failure("Invalid Identifier! : Assign.");
	}
}

#endif
```

“Data_Type.h”:

```
#ifndef __DATA_TYPE_H__
#define __DATA_TYPE_H__
void DataType()
{
	get();
	if (isdatatype(tkn->lexeme) == 0)
```

```
    {
        failure("Data Type Expected! : Data_Type.");
    }
}
#endif
```

“Decision.h”:

```
#ifndef __DECISION_H__
#define __DECISION_H__

void Decision()
{
    get();
    if (strcmp(tkn->lexeme, "if") == 0)
    {
        get();
        if (strcmp(tkn->lexeme, "(") == 0)
        {
            get();
            prev_flag = true;
            if (search_first(EXPN, tkn->lexeme, tkn->type) == 1)
            {
                Expn();
                get();
                if (strcmp(tkn->lexeme, ")") == 0)
                {
                    get();
                    if (strcmp(tkn->lexeme, "{") == 0)
                    {
                        get();
                        prev_flag = true;
                        if (search_first(STATEMENT_LIST, tkn->lexeme, tkn->type) == 1)
                        {
                            Statement_List();
                            get();
                            if (strcmp(tkn->lexeme, "}") == 0)
                            {
                                get();
                                prev_flag = true;
                                if (search_first(DPRIME, tkn->lexeme, tkn->type) == 1)
                                {
                                    DPrime();
                                }
                            }
                            else
                                failure("} expected! : Decision.");
                        }
                    }
                    else
                        failure("{ expected! : Decision.");
                }
                else
                    failure(") expected! Decision.");
            }
            else
            {
                failure("Invalid Expression. : Decision.");
            }
        }
        else
            failure("( expected! : Decision.");
    }
    else
        failure("No decision statement found! : Decision.");
}

void DPrime()
{
    get();
    if (strcmp(tkn->lexeme, "else") == 0)
    {
        get();
        if (strcmp(tkn->lexeme, "{") == 0)
        {
            get();
            prev_flag = true;
            if (search_first(STATEMENT_LIST, tkn->lexeme, tkn->type) == 1)
            {
                Statement_List();
                get();
                if (strcmp(tkn->lexeme, "}") != 0)
                {
                    failure("} expected! : DPrime.");
                }
            }
            else
            {
                failure("Invalid Statement! : DPrime.");
            }
        }
        else
            failure("{ expected! : DPrime.");
    }
    else
    {
        prev_flag = true;
    }
}
#endif
```

“Declarations.h”:

```
#ifndef __DECLARATIONS_H__
#define __DECLARATIONS_H__
void Declarations()
{
        get();
        prev_flag = true;
        if (search_first(DATA_TYPE, tkn->lexeme, tkn->type) == 1)
        {
                DataType();
                get();
                if (search_first(IDENTIFIERLIST, tkn->lexeme, tkn->type) == 1)
                {
                        prev_flag = true;
                        Identifier();
                        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 : Declaration.");
                                }
                        }
                        else
                        {
                                failure("Semi Colon Expected! : Declaration.");
                        }
                }
                else
                {
                        failure("Identifier expected! : Declaration.");
                }
        }
}
#endif
```

“Expression.h”:

```
#ifndef __EXPRESSION_H__
#define __EXPRESSION_H__
void Expn()
```

```
{
    if (search_first(SIMPLE_EXP, tkn->lexeme, tkn->type) == 1)
    {
        Simple_Exp();
        if (search_first(EPRIME, tkn->lexeme, tkn->type) == 1)
            EPrime();
    }
    else
    {
        failure("Invalid Identifier or Number! : Expn.");
    }
}

void EPrime()
{
    if (search_first(RELOP, tkn->lexeme, tkn->type) == 1)
    {
        Relop();
        get();
        prev_flag = true;
        if (search_first(SIMPLE_EXP, tkn->lexeme, tkn->type) == 1)
            Simple_Exp();
        else
            failure("Invalid Identifier or Number! : EPrime.");
    }
    else if (search_follow(EPRIME, tkn->lexeme, tkn->type) != 1)
    {
        failure("Invalid Operator or ; expected! : EPrime.");
    }
}

void Simple_Exp()
{
    if (search_first(TERM, tkn->lexeme, tkn->type) == 1)
    {
        Term();
        if (search_first(SEPRIME, tkn->lexeme, tkn->type) == 1)
            SePrime();
    }
    else
    {
        failure("Invalid Identifier or number! : Simple Exp.");
    }
}

void SePrime()
{
    if (search_first(ADDOP, tkn->lexeme, tkn->type) == 1)
    {

        Addop();
        get();
```

```
		prev_flag = true;
		if (search_first(TERM, tkn->lexeme, tkn->type) == 1)
		{
			Term();
			if (search_first(SEPRIME, tkn->lexeme, tkn->type) == 1)
				SePrime();
		}
		else
			failure("Invalid Identifier or number! : SePrime.");
	}
	else if (search_follow(SEPRIME, tkn->lexeme, tkn->type) != 1)
	{
		failure("Invalid Operator! : SePrime.");
	}
}

void Term()
{
	if (search_first(FACTOR, tkn->lexeme, tkn->type) == 1)
	{
		Factor();
		get();
		prev_flag = true;
		if (search_first(TPRIME, tkn->lexeme, tkn->type) == 1)
		{
			TPrime();
		}
	}
	else
	{
		failure("Invalid Identifier or number! : Term.");
	}
}

void TPrime()
{
	if (search_first(MULOP, tkn->lexeme, tkn->type) == 1)
	{
		Mulop();
		get();
		prev_flag = true;
		if (search_first(FACTOR, tkn->lexeme, tkn->type) == 1)
		{
			Factor();
			get();
			prev_flag = true;
			if (search_first(TPRIME, tkn->lexeme, tkn->type) == 1)
			{
				TPrime();
			}
		}
		else
```

```
        {
            failure("Invalid Identifier! : TPrime.");
        }
    }
    else if (search_follow(TPRIME, tkn->lexeme, tkn->type) != 1)
    {
        failure("Invalid Operator! : TPrime.");
    }
}

void Factor()
{
    get();
    if (search_symbol(tkn->lexeme) == -1 && tkn->type != NUMERIC_CONSTANT)
    {
        failure("Invalid Identifier or Number : Factor.");
    }
}

#endif
```

“Identifier.h”:

```
#ifndef __IDENTIFIER_H__
#define __IDENTIFIER_H__
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(IDENTIFIERLIST, tkn->lexeme, tkn->type) != 1)
        {
            failure("; expected! : Identifier.");
        }
    }
    else
    {
        failure("Invalid Identifier! : Identifier.");
    }
}

void IdentifierPrime()
```

```
{
	get();
	if (strcmp(tkn->lexeme, ",") == 0)
	{
		get();
		if (search_first(IDENTIFIERLIST, tkn->lexeme, tkn->type) == 1)
		{
			prev_flag = true;
			Identifier();
		}
		else if (strcmp(tkn->lexeme, "[") == 0)
		{
		}
		else
		{
			failure("Invalid Identifier! : Identifier`.");
		}
	}
	else if (strcmp(tkn->lexeme, "[") == 0)
	{
		get();
		if (tkn->type == NUMERIC_CONSTANT)
		{
			get();
			if (strcmp(tkn->lexeme, "]") == 0)
			{
				if (search_first(IDENTIFIERLISTPRIMEPRIME, tkn->lexeme, tkn-
>type) == 1)
					IdentifierPrimePrime();
			}
			else
			{
				failure("] expected!  : Identifier`.");
			}
		}
		else
		{
			failure("Number expected!  : Identifier`.");
		}
	}
	else
	{
		if (search_follow(IDENTIFIERLISTPRIME, tkn->lexeme, tkn->type) == 1)
			prev_flag = true;
		else
		{
			failure("; expected!  : Identifier`.");
		}
	}
}

void IdentifierPrimePrime()
```

```
{
	get();
	if (strcmp(tkn->lexeme, ",") == 0)
	{
		if (search_first(IDENTIFIERLIST, tkn->lexeme, tkn->type) == 1)
			Identifier();
		else
		{
			failure("Invalid Identifier!  : Identifier``.");
		}
	}
	else
	{
		if (search_follow(IDENTIFIERLISTPRIMEPRIME, tkn->lexeme, tkn->type) == 1)
			prev_flag = true;
		else
		{
			failure("; expected!  : Identifier``.");
		}
	}
}
#endif
```

“Looping.h”:

```
#ifndef __LOOPING_H__
#define __LOOPING_H__

void Looping()
{
	get();
	if (strcmp(tkn->lexeme, "while") == 0)
	{
		get();
		if (strcmp(tkn->lexeme, "(") == 0)
		{
			get();
			prev_flag = true;
			if (search_first(EXPN, tkn->lexeme, tkn->type) == 1)
			{
				Expn();
				get();
				if (strcmp(tkn->lexeme, ")") != 0)
				{
					failure(") expected! : W Looping.");
				}
				get();
				if (strcmp(tkn->lexeme, "{") == 0)
```

```
                {
                    get();
                    prev_flag = true;
                    // if (search_first(STATEMENT_LIST, tkn->lexeme, tkn-
>type) == 1)
                    {
                        Statement_List();
                        get();
                        if (strcmp(tkn->lexeme, "}") != 0)
                        {
                            failure("} expected! : W Looping.");
                        }
                    }
                    // else printf("%s\n", tkn->lexeme), failure("Invalid
statement! : Looping.");
                }
                else
                    failure("{ expected! : W Looping");
            }
            else
                failure("Invalid Expression. : W Looping.");
        }
        else
            failure("( expected! : W Looping");
    }
    else if (strcmp(tkn->lexeme, "for") == 0)
    {
        get();
        if (strcmp(tkn->lexeme, "(") == 0)
        {
            get();
            prev_flag = true;
            if (search_first(ASSIGN_STAT, tkn->lexeme, tkn->type) == 1)
            {
                AssignStat();
                get();
                if (strcmp(tkn->lexeme, ";") == 0)
                {
                    get();
                    prev_flag = true;
                    if (search_first(EXPN, tkn->lexeme, tkn->type) == 1)
                    {
                        Expn();
                        get();
                        if (strcmp(tkn->lexeme, ";") == 0)
                        {
                            get();
                            prev_flag = true;
                            if (search_first(ASSIGN_STAT, tkn->lexeme,
tkn->type) == 1)
                            {
                                AssignStat();
                                get();
                                if (strcmp(tkn->lexeme, ")") == 0)
                                {
                                    get();
                                    if (strcmp(tkn->lexeme, "{") == 0)
                                    {
                                        get();
                                        prev_flag = true;
                                        if (search_first(STATEMENT_LIST, tkn->lexeme, tkn->type) == 1)
                                        {
                                            Statement_List();
                                            get();
                                            if (strcmp(tkn->lexeme, "}") != 0)
                                            {
                                                failure("} expected! : F Looping");
                                            }
                                        }
                                        else
                                            failure("Invalid Identifier! : F Looping");
                                    }
                                    else
                                    {
                                        failure("{ expected! : F Looping");
                                    }
                                }
                                else
                                    failure(") expected : F Looping");
                            }
                            else
                                failure("Invalid Identifier! : F Looping");
                        }
                        else
                            failure("; expected! : F Looping");
                    }
                    else
                        failure("Invalid Expression! : F Looping");
                }
                else
                    failure("; expected! : F Looping");
            }
            else
                failure("Invalid Identifier! : F Looping");
        }
        else
            failure("( expected! : F Looping");
    }
    else
        failure("Invalid Loop! : F Looping");
}

#endif
```

“Operators.h”:

```
#ifndef __OPERATORS_H__
#define __OPERATORS_H__
void Relop()
{
    get();
    if (strcmp(tkn->lexeme, "==") == 0 || strcmp(tkn->lexeme, "!=") == 0 || strcmp(tkn->lexeme, ">=") == 0 || strcmp(tkn->lexeme, "<=") == 0 || strcmp(tkn->lexeme, ">") == 0 || strcmp(tkn->lexeme, "<") == 0)
    {
        return;
    }
    else
    {
        failure("Invalid Operator! : Operators.");
    }
}

void Addop()
{
    get();
    if (strcmp(tkn->lexeme, "+") == 0 || strcmp(tkn->lexeme, "-") == 0)
    {
        return;
    }
    else
    {
        failure("Invalid Operator! : Operators.");
    }
}

void Mulop()
{
    get();
    if (strcmp(tkn->lexeme, "*") == 0 || strcmp(tkn->lexeme, "/") == 0 || strcmp(tkn->lexeme, "%%") == 0)
    {
        return;
    }
    else
    {
        failure("Invalid Operator! : Operators.");
```

```
		}
}
#endif
```

“procedures.h”:

```
#ifndef __PROCEDURES_H__
#define __PROCEDURES_H__

void Program();
void Declarations();
void Statement_List();
void Statement();
void Expn();
void EPrime();
void Simple_Exp();
void SePrime();
void Term();
void TPrime();
void Factor();
void Relop();
void Addop();
void Mulop();
void DataType();
void Identifier();
void IdentifierPrime();
void IdentifierPrimePrime();
void AssignStat();
void Decision();
void DPrime();
void Looping();

#include "Program.h"
#include "Declarations.h"
#include "Data_Type.h"
#include "Identifier.h"
#include "Statement.h"
#include "Assign.h"
#include "Expression.h"
#include "Decision.h"
#include "Looping.h"
#include "Operators.h"

#endif
```

“Program.h”:

```
#ifndef __PROGRAM_H__
#define __PROGRAM_H__

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();
                            get();
                            if (search_first(STATEMENT_LIST, tkn-
>lexeme, tkn->type) == 1)

                            {
                                prev_flag = true;
                                Statement_List();
                                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! : Program.");
                                            }
                                        }
                                        else
                                        {
                                            failure("No Semi-
Colon found! : Program.");
                                        }
                                    }
                                    else
                                    {
                                        failure("Numeric Value
Expected! : Program.");
                                    }
                                }
                                else
                                {
                                    failure("No return statement
found! : Program.");
                                }
                            }
                            else
                            {
                                failure("Invalid Identifier! : Program.");
                            }
                        }
                        else
                        {
                            failure("Data Type expected! : Program.");
                        }
                    }
                    else
                    {
                        failure("No starting curly bracket found! : Program.");
                    }
                }
                else
                {
                    failure("No function closing parentheses found! : Program.");
                }
            }
            else
            {
                failure("No function starting parentheses found! : Program.");
            }
        }
        else
        {
            failure("No main found! : Program.");
```

```
		}
	}
	else
	{
		failure("No return type found! : Program.");
	}
}

#endif
```

“Statement.h”:

```
#ifndef __STATEMENT_H__
#define __STATEMENT_H__
void Statement_List()
{
	get();
	prev_flag = true;
	if (search_first(STATEMENT, tkn->lexeme, tkn->type) == 1)
	{
		Statement();
		get();
		prev_flag = true;
		if (search_first(STATEMENT_LIST, tkn->lexeme, tkn->type) == 1)
		{
			Statement_List();
		}
	}
	else if (search_follow(STATEMENT_LIST, tkn->lexeme, tkn->type) != 1)
	{
		failure("Invalid Statement! : Statement List.");
	}
}

void Statement()
{
	get();
	prev_flag = true;
	if (search_first(ASSIGN_STAT, tkn->lexeme, tkn->type) == 1)
	{
		AssignStat();
		get();
		if (strcmp(tkn->lexeme, ";") != 0)
		{
			failure("; expected! : Statement.");
		}
	}
	else if (search_first(DECISION, tkn->lexeme, tkn->type) == 1)
	{
		Decision();
	}
	else if (search_first(LOOPING, tkn->lexeme, tkn->type) == 1)
	{
		Looping();
	}
	else if (search_follow(STATEMENT, tkn->lexeme, tkn->type) != 1)
	{
		failure("Invalid Statement! : Statement.");
	}
}
#endif
```

“Lexical/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
```

“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
```

“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
```

“lexeme.h”:

```
#ifndef __LEXEME_H__
#define __LEXEME_H__
#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
```

“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
```

“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
```

“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
```

“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
```

****
****

THE END

****
****