Compiler Design Lab Final Project Report

By : Sahil Garg (160905058)

Amrit Goyal (160905210)

Jenit Jain (160905070)

1. **Objective:** The main objective of this project is to create a compiler that can help us transform a program written in a high-level programming language from source code into object code. In this project we made a compiler for the language ERPLAG which is a strongly typed language with primitive data types as integer and floating point. It also supports two other data types: boolean and arrays. Having made use of concepts taught to us in compiler design like making a lexical analyzer and a symbol table we have made use of C programming language to create this compiler.
2. **Grammar:**

<program> -> <moduleDeclarations> <otherModules><driverModule><otherModules>

<moduleDeclarations> ->à <moduleDeclaration><moduleDeclarations> | ε

<moduleDeclaration> ->à DECLARE MODULE ID SEMICOL

<otherModules> -> <module><otherModules>| ε

<driverModule> -> DEF DRIVER PROGRAM ENDDEF <moduleDef>

<module> -> DEF MODULE ID ENDDEF TAKES INPUT SQBO <input\_plist> SQBC SEMICOL <ret> <moduleDef>

<ret> -> RETURNS SQBO <output\_plist> SQBC SEMICOL | ε

<input\_plist> -> <input\_plist> COMMA ID COLON <dataType> | ID COLON <dataType>

<output\_plist> -> <output\_plist> COMMA ID COLON <type> | ID COLON <type>

<dataType> -> INTEGER | REAL | BOOLEAN | ARRAY SQBO <range> SQBC OF <type>

<type> -> INTEGER | REAL | BOOLEAN

<moduleDef> -> START <statements> END

<statements> -><statement> <statements> | ε

<statement> -> <ioStmt>|<simpleStmt>|<declareStmt>|<condionalStmt>|<iterativeStmt>

<ioStmt> -> GET\_VALUE BO ID BC SEMICOL | PRINT BO <var> BC SEMICOL

<var> -> ID <whichId> | NUM | RNUM

<whichId> -> SQBO ID SQBC | ε

<simpleStmt> -> <assignmentStmt> | <moduleReuseStmt>

<assignmentStmt> -> ID <whichStmt>

<whichStmt> -> <lvalueIDStmt> | <lvalueARRStmt>

<lvalueIDStmt> -> ASSIGNOP <expression> SEMICOL

<lvalueARRStmt> -> SQBO <index> SQBC ASSIGNOP <expression> SEMICOL

<index> -> NUM | ID

<moduleReuseStmt> -> <optional> USE MODULE ID WITH PARAMETERS <idList>SEMICOL

<optional> -> SQBO <idList> SQBC ASSIGNOP | ε

<idList>-> <idList> COMMA ID | ID

<expression> -> <arithmeticExpr> | <booleanExpr>

<arithmeticExpr> -> <arithmeticExpr> <op> <term> | <term>

<term> -> <term> <op> <factor> | <factor>

<factor> -BO <arithmeticExpr> BC | <var>

<op> -> PLUS | MINUS | MUL | DIV

<booleanExpr> -> <booleanExpr> <logicalOp> <booleanExpr>

<logicalOp> -> AND | OR

<booleanExpr> -> <arithmeticExpr> <relationalOp> <arithmeticExpr> | BO <booleanExpr> BC

<relationalOp> -> LT | LE | GT | GE | EQ | NE

<declareStmt> -> DECLARE <idList> COLON <dataType> SEMICOL

<condionalStmt> -> SWITCH BO ID BC START <caseStmt><default> END

<caseStmt> -> CASE <value> COLON <statements> BREAK SEMICOL <caseStmt>

<value> -> NUM | TRUE | FALSE

<default> -> DEFAULT COLON <statements> BREAK SEMICOL | ε

<iterativeStmt> -> FOR BO ID IN <range> BC START <statements> END | WHILE BO <booleanExpr> BC START <statements> END

<range> -> NUM RANGEOP NUM

1. **Language used for Implementation:** C only
2. **Type of Parser: Top down parser**. Top-down parsing is a parsing strategy where one first looks at the highest level of the parse tree and works down the parse tree by using the rewriting rules of a formal grammar. More on this in the methodology
3. **Brief Methodology:**

The compiler designed includes a lexical analyzer, symbol table generator and parser, with error detection. The top-down parser is an LL parser. The methodologies for individual components are described as below -

- Lexical analyzer : The lexer.c file reads the input file or the ERPLAG code character by character. The different keywords and patterns for tokens are identified by comparing the patterns with the string of characters read. Keywords are stored in keywords.txt. The list of keywords is read from this file. The function removeComments() removes comments and creates clean code that can then be used for lexeme construction. PopulateLexemeTable() generates the list of tokens and prints them on the screen. All patterns are defined using constant arrays elements or through if-else code.

-Symbol Table generation : The createIDTable() function is used to add the constructed lexemes to the symbol table. Seperate structures in C are used to store data about each identified added to the symbol table, such as datatype, function its part of and scope. When adding an identifier to the symbol table, it checks if already exists or not. If it does not, then a new entity is added to the symbol table. Hashing using open hash set (linked list) is used for searching in the symbol table for more efficiency.

- Parser : The parser developed is an LL top-down parser, with the code in parser.c. It first reads the grammar from grammar.txt, generates a table for the grammar and stores it (function createGrammar()). Then, first and follow sets are found for each variable in the grammar using construction of a tree and performing depth-first search. Then, the parse table is creating using the first and follow sets, and productions (represented as int) are added to the table. The parsing for the code is then doing using an arbitrary stack for input string, and a queue to match it with the productions. ParseInputSourceCode() performs the parsing after the grammar is created and checks for errors. In case an error occurs and no production is matching, error is printed with the appropriate line number and parsing stops. The closest production’s follow variables are displayed to show the user what may be the expected correct input.

1. **User documentation/readme text:**

This project was created by a group of 3 Students- Sahil Garg, Jenit Jain and Amrit Goyal as a part of the course lab “Compiler Design”. The language is "ERPLAG" and its specifications are written down in the PDF attached.

Instructor: Roopashri Shetty

How to run

In the Final project directory, just type the following commands

make

compiler name\_of\_file

where "name\_of\_file" is the input test file (like "testcase1.txt”).

You should have gcc, version 5.0 or above installed on your machine.

**7. Code:**

**driver.c**

/\*

\* COMPILER PROJECT- ERPLAG COMPILER

\* CSE - D 2019

\* Sahil Garg, Jenit Jain, Amrit Goyal

\*/

#include <stdio.h>

#include <math.h>

#include <string.h>

#include <stdlib.h>

#include "lexerDef.h"

#include "lexer.h"

#include "parserDef.h"

#include "parser.h"

#include "symbolDef.h"

#include "symbolTable.h"

#define LIMIT 25

void printSep()

{

printf("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

}

int main(int argc, char \* argv[])

{

//Initialise the hash table for keywords

hashTable table = initializeHashTable(31);

FILE \*fp = fopen("keywords.txt", "r");

int type, option;

char input[21];

while(~fscanf(fp, "%d %s", &type, input))

{

addKeyword(table, type, input);

}

fclose(fp);

// Initialise the Grammer

Grammar G = (Grammar)malloc(RULECNT\*sizeof(gHead));

G = createGrammar();

findFirst(G);

findFollow(G);

createParseTable(G);

while(1)

{

//Display the menu

printf("\nERPLAG COMPILER MENU OPTIONS:\n");

printf("1: For producing clean code by removal of comments.\n");

printf("2: For printing the token list generated by the lexer.\n");

printf("3: For printing the Symbol table.\n");

printf("4: For parsing to verify the syntactic correctness of the input source code\n");

printf("5: For creating the parsetree and printing it(inorder traversal)\n");

printf("6: To exit.\n");

int option;

scanf("%d",&option);

if(option == 1)

{

removeComments(argv[1],"cleancode.txt");

printSep();

printf("cleancode.txt created\n");

printSep();

}

else if(option == 2)

{

printSep();

int error\_exist = populateLexemeTable(argv[1], table, 1);

if (error\_exist == 1)

{

printf("Recheck your code for Lexical errors\n");

}

printSep();

}

else if (option == 3)

{

removeComments(argv[1], "cleancode.txt");

parseTree Tree = parseInputSourceCode("cleancode.txt", table, G, 0);

tokenInfo \*T = getFirstToken();

hashTable2 tableId = initializeHashTable2(hash\_capacity\_2);

hashTable2 tableFunc = initializeHashTable2(hash\_capacity\_2);

totalScopeList\* scopeTable = (totalScopeList \*)malloc(LIMIT \* sizeof(totalScopeList));

int i;

for(i = 0; i < LIMIT; ++i)

{

scopeTable[i].scope\_start = 0;

scopeTable[i].scope\_end = 0;

}

createIDTable(T, scopeTable, tableId, tableFunc, 0);

T = getFirstToken();

secondRun(T, tableId, tableFunc, 0);

printf("Symbol Table sucessfully created.\n");

printf("%-4s %-10s\t%-20s %-10s %-15s %-6s %-6s %-6s\n", "SNO", "Var-Name", "Datatype", "Func-Name", "Start - End", "Level", "Size", "Offset");

printVariables(tableId, scopeTable);

}

else if(option == 4)

{

removeComments(argv[1], "cleancode.txt");

printSep();

parseTree Tree = parseInputSourceCode("cleancode.txt", table, G, 0);

printSep();

}

else if (option == 5)

{

removeComments(argv[1], "cleancode.txt");

printSep();

// parseTree Tree = parseInputSourceCode("clean.txt", table, G, 0);

printParseTree("cleancode.txt", table, G);

printf("\n");

printSep();

}

else if (option == 6)

{

break;

}

else

{

printf("\nInvalid Option\n");

}

}

return(0);

}

**hash.c:**

/\*

\* COMPILER PROJECT- ERPLAG COMPILER

\* CSE - D 2019

\* Sahil Garg, Jenit Jain, Amrit Goyal

\*/

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include "lexerDef.h"

#define hash\_capacity 31

hashTable initializeHashTable(int l)

{

hashTable table;

table = (hashTable)malloc(l \* sizeof(entry));

int i;

for(i = 0; i < l; ++i)

{

table[i].code = -1;

table[i].flag = available;

table[i].start = (linklist \*)malloc(sizeof(linklist));

}

return table;

}

int hashValue(char \*key)

{

int len = strlen(key);

int i, value = 0, p = 1, prime = 5;

//Rolling Hash on strings

for(i = len - 1; i >= 0; --i)

{

value = (value + p \* (key[i]-'0')) % hash\_capacity;

if (value < 0)

{

value += hash\_capacity;

}

p = (p \* prime) % hash\_capacity;

}

return value;

}

store findKeyword(hashTable T, char \*searchKey)

{

store res;

int pos = hashValue(searchKey);

if(T[pos].flag == available)

{

res.code = -1;

strcpy(res.word, "\0");

}

else if(strcmp(T[pos].word, searchKey) == 0)

{

res.code = T[pos].code;

strcpy(res.word, searchKey);

}

else

{

linklist\* root = (linklist \*)malloc(sizeof(linklist));

root = T[pos].start;

while (root != NULL)

{

if(strcmp(root->word, searchKey) == 0)

{

res.code = root->code;

strcpy(res.word, searchKey);

return res;

}

root = root->next;

}

res.code = -1;

}

return res;

}

void addKeyword(hashTable T, int c, char \*key)

{

int pos = hashValue(key);

if(T[pos].flag == available || T[pos].flag == deleted)

{

T[pos].code = c;

strcpy(T[pos].word,key);

T[pos].flag = occupied;

T[pos].start = NULL;

}

else

{

linklist\* root = (linklist \*)malloc(sizeof(linklist));

strcpy(root->word, key);

root->code = c;

root->next = NULL;

linklist\* temp = T[pos].start;

if(temp == NULL)

{

T[pos].start = root;

}

else

{

while(temp->next != NULL)

{

temp = temp->next;

}

temp->next = root;

}

}

}

**hash2.c:**

/\*

\* COMPILER PROJECT- ERPLAG COMPILER

\* CSE - D 2019

\* Sahil Garg, Jenit Jain, Amrit Goyal

\*/

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include "symbolDef.h"

hashTable2 initializeHashTable2(int l)

{

hashTable2 table;

table = (hashTable2)malloc(l \* sizeof(entry2));

int i;

for(i = 0; i < l; ++i)

{

table[i].flag = available;

table[i].start = NULL;

table[i].ID = NULL;

}

return(table);

}

int hashValue2(char \*key)

{

int len = strlen(key);

int i, value = 0, p = 1, prime = 5;

//Rolling Hash on strings

for(i = len - 1; i >= 0; --i)

{

value = (value + p \* (key[i]-'0')) % hash\_capacity\_2;

if (value < 0)

{

value += hash\_capacity\_2;

}

p = (p \* prime) % hash\_capacity\_2;

}

return value;

}

store2 findScope(hashTable2 T, char \*searchKey, int scope, int option)

{

store2 res;

int pos = hashValue2(searchKey);

if(T[pos].flag == available)

{

res.code = -1;

}

else if(option == 0)

{

if(strcmp(T[pos].ID->entity.ivar.word,searchKey) == 0)

{

if(scope == T[pos].ID->entity.ivar.scope)

{

res.code = 1;

res.node = T[pos].ID;

}

else

{

linklist2\* root = (linklist2 \*)malloc(sizeof(linklist2));

root = T[pos].start;

while (root != NULL)

{

if(strcmp(root->ID->entity.ivar.word, searchKey) == 0)

{

if(scope == root->ID->entity.ivar.scope)

{

res.code = 1;

res.node = (root->ID);

return res;

}

}

root = root->next;

}

res.code = -1;

}

}

else

{

linklist2\* root = (linklist2 \*)malloc(sizeof(linklist2));

root = T[pos].start;

while (root != NULL)

{

if(strcmp(root->ID->entity.ivar.word, searchKey) == 0)

{

if(scope == root->ID->entity.ivar.scope)

{

res.code = 1;

res.node = (root->ID);

return res;

}

}

root = root->next;

}

res.code = -1;

}

}

else

{

if(strcmp(T[pos].ID->entity.ifunc.word,searchKey) == 0)

{

res.node = (T[pos].ID);

res.code = 1;

}

else

{

linklist2\* root = (linklist2 \*)malloc(sizeof(linklist2));

root = T[pos].start;

while (root != NULL)

{

if(strcmp(root->ID->entity.ifunc.word, searchKey) == 0)

{

res.code = 1;

res.node = (root->ID);

return res;

}

root = root->next;

}

res.code = -1;

}

}

return res;

}

void addScope(hashTable2 T, IDEntry\* M, int option)

{

int pos;

if(option == 0)

{

pos = hashValue2(M->entity.ivar.word);

}

else

{

pos = hashValue2(M->entity.ifunc.word);

}

if(T[pos].flag == available || T[pos].flag == deleted)

{

T[pos].ID = M;

T[pos].flag = occupied;

T[pos].start = NULL;

}

else

{

linklist2\* root = (linklist2 \*)malloc(sizeof(linklist2));

root->ID = M;

root->next = NULL;

linklist2 \*temp = T[pos].start;

if(temp == NULL)

{

T[pos].start = root;

}

else

{

while (temp->next != NULL)

{

temp = temp->next;

}

temp->next = root;

}

}

}

**lexer.c**

/\*

\* COMPILER PROJECT- ERPLAG COMPILER

\* CSE - D 2019

\* Sahil Garg, Jenit Jain, Amrit Goyal

\*/

#include "lexerDef.h"

#include "lexer.h"

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#include <string.h>

#include <stdlib.h>

#include <ctype.h>

int id\_num = 1;

int error\_exist = 0;

int line\_num = 1;

int comment\_flag = 0;

tokenInfo\* end;

tokenInfo\* first;

tokenInfo\* scan;

FILE \*f\_write;

static FILE\* f\_read;

int BUFFSIZE = 256;

int isAlpha(char c)

{

if ((c >= 'A' && c <= 'Z') || (c >= 'a' && c <= 'z'))

{

return 1;

}

return 0;

}

int isDigit(char c)

{

if (c >= '0' && c <= '9')

{

return 1;

}

return 0;

}

int expo(int e, int n)

{

int res = 1, val = e;

while(n)

{

if (n & 1)

{

res = res \* val;

}

val = val \* val;

n /= 2;

}

return res;

}

char \*getStream()

{

//Get line from input file into bufferfer

char \*temp = (char \*)malloc(BUFFSIZE \* sizeof(char));

fgets(temp, BUFFSIZE, f\_read);

if (feof(f\_read))

{

//reached end of file

return "";

}

if (strlen(temp) == BUFFSIZE)

{

int pos = (BUFFSIZE - 1);

//remove spaces and tabs

while((pos >= 0) && (!(temp[pos]==' ' || temp[pos]=='\t')))

{

pos -= 1;

}

temp[pos + 1] = '\0';

fseek(f\_read, -((BUFFSIZE - 1) - pos), SEEK\_CUR);

if (pos < 0)

{

//when no spaces are there in buffer (in order to avoid infinite loop)

fgets(temp, BUFFSIZE, f\_read);

}

}

return temp;

}

tokenInfo\* getFirstToken()

{

return first->n;

}

tokenInfo\* getNextToken()

{

if(scan == NULL)

{

scan = first;

}

else

{

scan = scan->n;

}

return scan;

}

tokenInfo\* initToken()

{

//Initialise the basic common values of Token

tokenInfo\* temp = (tokenInfo \*)malloc(sizeof(tokenInfo));

temp->lno = line\_num;

temp->n = NULL;

temp->prev = end;

end->n = temp;

end = end->n;

return temp;

}

void addDollar()

{

//Add Dollar to end of file

tokenInfo \*temp = initToken();

temp->s = $;

}

void addSym(int n)

{

//Add a symbol to the tokenstream

tokenInfo\* temp = initToken();

int i, done = 0;

for(i = INTEGER; i != ID; ++i)

{

if (i == n)

{

done = 1;

temp->s = (term)i;

break;

}

}

if (done == 0)

{

FILE \*f\_write;

f\_write = fopen("error.txt","a");

printf("Wrong symbol number %d passed to addSym\n",n);

fclose(f\_write);

}

}

void addID(char \*id)

{

//Add an ID to the tokenstream

tokenInfo\* temp = initToken();

temp->s = ID;

strcpy(temp->val.s, id);

}

void addRNum(float num, int rnum\_no)

{

//Add a floating point number to tokenstream

tokenInfo\* temp = initToken();

temp->s = RNUM;

temp->val.f = num;

temp->df = rnum\_no;

}

void addNum(int num, int num\_no)

{

//Add an integer to tokenstream

tokenInfo\* temp = initToken();

temp->s = NUM;

temp->val.i = num;

temp->df = num\_no;

}

void DFA(hashTable table, char \*buffer, int pr)

{

//DFA for pattern matching and assigning correct token, if possible

int pos = 0;

int l = strlen(buffer);

char lookahead = buffer[0];

static int num\_no = 1;

static int rnum\_no = 1;

while(pos < l)

{

//ignore content between the comments

if (comment\_flag == 1)

{

while((pos < l) && (!(buffer[pos] == '\*' && buffer[pos+1] == '\*')))

{

if (buffer[pos] == '\n')

{

line\_num += 1;

}

pos += 1;

}

if (pos < l && buffer[pos] == '\n')

{

line\_num += 1;

}

if(pos == l)

{

pos -= 1;

}

lookahead = buffer[pos];

}

if (isAlpha(lookahead))

{

//ID or keyword is checked through hashing

char id[21] = "";

int i = 0;

id[i] = lookahead;

pos += 1;

while ((pos < l) && ((isAlpha(buffer[pos]) || buffer[pos] == '\_' || isDigit(buffer[pos])) && buffer[pos] != '\n'))

{

id[++i] = buffer[pos++];

}

store res = findKeyword(table, id);

if (res.code == -1)

{

//ID as not found in hash table

if (strlen(id) <= 8)

{

if (pr == 1)

{

printf("Line Number : %d, Lexeme : ID - %s\n", line\_num, id);

}

addID(id);

}

else

{

if(pr == 1)

{

printf("ERROR : Identifier %s at line %d is longer than 8 characters.\n", id, line\_num);

}

error\_exist = 1;

}

}

else

{

//Keyword as found in hash table

if (pr == 1)

{

int j, len = strlen(res.word);

printf("Line Number : %d, Lexeme : KEYWORD - ", line\_num);

for(j = 0; j < len; ++j)

{

printf("%c", toupper(res.word[j]));

}

printf("\n");

}

addSym(res.code);

}

pos -= 1;

}

else if (isDigit(lookahead))

{

//NUM or RNUM

int num1 = 0, num2 = 0;

int dec = 0;

int power = 0;

num1 = buffer[pos] - '0';

while((pos+1 < l) && isDigit(buffer[pos+1]))

{

num1 = (num1 \* 10) + (buffer[pos+1]-'0');

pos += 1;

}

if((pos+1 < l) && buffer[pos+1] == '.')

{

if((pos+2 < l) && buffer[pos+2] != '.')

{

pos += 1;

if((pos+1 < l) && (!isDigit(buffer[pos+1])))

{

if(pr == 1)

{

printf("ERROR : Invalid real number at line %d, column %d : No digit after floating point.\n", line\_num, (pos+1));

}

error\_exist = 1;

pos += 1;

}

while((pos+1 < l) && isDigit(buffer[pos+1]))

{

pos += 1;

num2 = (num2 \* 10) + (buffer[pos] - '0');

dec++;

}

if((pos+1 < l) && (buffer[pos+1] == 'e' || buffer[pos+1] == 'E'))

{

pos += 2;

if(buffer[pos] == '+')

{

while((pos+1 < l) && isDigit(buffer[pos+1]))

{

pos += 1;

power = (power \* 10) + (buffer[pos] - '0');

}

}

else if(buffer[pos] == '-')

{

while((pos+1 < l) && isDigit(buffer[pos+1]))

{

pos += 1;

power = (power \* 10) + (buffer[pos] - '0');

}

power = -(power);

}

else if(isDigit(buffer[pos]))

{

while(isDigit(buffer[pos]))

{

power = (power \* 10) + (buffer[pos] - '0');

pos += 1;

}

pos -= 1;

}

else

{

if(pr == 1)

{

printf("ERROR : Invalid real number at line %d, column %d : invalid character after exponent symbol.\n",line\_num, pos);

}

error\_exist = 1;

}

}

else

{

power = 0;

}

float num = (float)num1 + ((float)num2/(expo(10,dec)));

if (power >= 0)

{

num \*= expo(10, power);

}

else

{

power = -(power);

num /= expo(10, power);

}

if (pr == 1)

{

printf("Line Number : %d, Lexeme : RNUM - %lf\n", line\_num, num);

}

rnum\_no += 1;

addRNum(num, rnum\_no);

}

else

{

if (pr == 1)

{

printf("Line Number : %d, Lexeme : NUM - %d\n", line\_num, num1);

}

num\_no += 1;

addNum(num1, num\_no);

}

}

else

{

if ((pos+1 < l) && isAlpha(buffer[pos+1]))

{

if (pr == 1)

{

printf("ERROR : Invalid Identifier at line %d, column %d : invalid start character (can't be number).\n", line\_num, pos);

}

error\_exist = 1;

while ((pos < l) && ((isAlpha(buffer[pos]) || buffer[pos] == '\_' || isDigit(buffer[pos])) && buffer[pos] != '\n'))

{

pos += 1;

}

}

else

{

if (pr == 1)

{

printf("Line Number : %d, Lexeme : NUM - %d\n", line\_num, num1);

}

num\_no += 1;

addNum(num1, num\_no);

}

}

}

else if (lookahead == '+')

{

if (pr == 1)

{

printf("Line Number : %d, Lexeme : PLUS - +\n", line\_num);

}

addSym(PLUS);

}

else if (lookahead == '-')

{

if (pr == 1)

{

printf("Line Number : %d, Lexeme : MINUS - -\n", line\_num);

}

addSym(MINUS);

}

else if (lookahead == '\*')

{

if((pos+1 < l) && buffer[pos+1] == '\*')

{

if (pr == 1)

{

printf("Line Number : %d, Lexeme : COMMENTMARK - \*\*\n", line\_num);

}

addSym(COMMENTMARK);

pos += 2;

if(comment\_flag == 1)

{

comment\_flag ^= 1;

lookahead = buffer[pos];

}

else

{

comment\_flag ^= 1;

}

while((pos < l) && (!(buffer[pos] == '\*' && buffer[pos+1] == '\*')))

{

if (buffer[pos] == '\n')

{

line\_num += 1;

}

pos += 1;

}

if (pos < l && buffer[pos] == '\n')

{

line\_num += 1;

}

pos -= 1;

}

else

{

if (pr == 1)

{

printf("Line Number : %d, Lexeme : MUL - \*\n", line\_num);

}

addSym(MUL);

}

}

else if (lookahead == '/')

{

if (pr == 1)

{

printf("Line Number : %d, Lexeme : DIV - /\n", line\_num);

}

addSym(DIV);

}

else if (lookahead == '<')

{

if((pos+1 < l) && buffer[pos+1] == '=')

{

if (pr == 1)

{

printf("Line Number : %d, Lexeme : LE - <=\n", line\_num);

}

addSym(LE);

pos += 1;

}

else if((pos+1 < l) && buffer[pos+1] == '<')

{

if((pos+2 < l) && buffer[pos+2] == '<')

{

if (pr == 1)

{

printf("Line Number : %d, Lexeme : DRIVERDEF - <<<\n", line\_num);

}

addSym(DRIVERDEF);

pos += 2;

}

else

{

if (pr == 1)

{

printf("Line Number : %d, Lexeme : DEF - <<\n", line\_num);

}

addSym(DEF);

pos += 1;

}

}

else

{

if (pr == 1)

{

printf("Line Number : %d, Lexeme : LT - <\n", line\_num);

}

addSym(LT);

}

}

else if (lookahead == '>')

{

if((pos+1 < l) && buffer[pos+1] == '=')

{

if (pr == 1)

{

printf("Line Number : %d, Lexeme : GE - >=\n", line\_num);

}

addSym(GE);

pos += 1;

}

else if((pos+1 < l) && buffer[pos+1] == '>')

{

if((pos+2 < l) && buffer[pos+2] == '>')

{

if (pr == 1)

{

printf("Line Number : %d, Lexeme : DRIVERENDDEF - >>>\n", line\_num);

}

addSym(DRIVERENDDEF);

pos += 2;

}

else

{

if (pr == 1)

{

printf("Line Number : %d, Lexeme : ENDDEF - >>\n", line\_num);

}

addSym(ENDDEF);

pos += 1;

}

}

else

{

if (pr == 1)

{

printf("Line Number : %d, Lexeme : GT - >\n", line\_num);

}

addSym(GT);

}

}

else if (lookahead == '=')

{

if((pos+1 < l) && buffer[pos+1] == '=')

{

if (pr == 1)

{

printf("Line Number : %d, Lexeme : EQ - ==\n", line\_num);

}

addSym(EQ);

pos += 1;

}

else

{

if(pr == 1)

{

printf("ERROR : Invalid symbol '=' at line %d, column %d.\n",line\_num, pos);

}

error\_exist = 1;

pos += 1;

}

}

else if (lookahead == '!')

{

if((pos+1 < l) && buffer[pos+1] == '=')

{

if (pr == 1)

{

printf("Line Number : %d, Lexeme : NE - !=\n", line\_num);

}

addSym(NE);

pos += 1;

}

else

{

if(pr == 1)

{

printf("ERROR : Invalid symbol '!' at line %d, column %d.\n",line\_num, pos);

}

error\_exist = 1;

}

}

else if (lookahead == ':')

{

if((pos+1 < l) && buffer[pos+1] == '=')

{

if (pr == 1)

{

printf("Line Number : %d, Lexeme : ASSIGNOP - :=\n", line\_num);

}

addSym(ASSIGNOP);

pos += 1;

}

else

{

if (pr == 1)

{

printf("Line Number : %d, Lexeme : COLON - :\n", line\_num);

}

addSym(COLON);

}

}

else if (lookahead == '.')

{

if((pos+1 < l) && buffer[pos+1]=='.')

{

if ((pos+2 < l) && isDigit(buffer[pos+2]))

{

if (pr == 1)

{

printf("Line Number : %d, Lexeme : RANGEOP - ..\n", line\_num);

}

addSym(RANGEOP);

pos += 1;

}

else

{

if (pr == 1)

{

printf("ERROR : Invalid Symbol at line %d, column %d. (Had expected range operation)\n", line\_num, pos);

}

error\_exist = 1;

while(pos < l && buffer[pos] == '.')

{

pos += 1;

}

pos -= 1;

}

}

else

{

if(pr == 1)

{

printf("ERROR : Invalid symbol '.' at line %d, column %d.\n",line\_num, pos);

}

error\_exist = 1;

}

}

else if (lookahead == ',')

{

if (pr == 1)

{

printf("Line Number : %d, Lexeme : COMMA - ,\n", line\_num);

}

addSym(COMMA);

}

else if (lookahead == ';')

{

if (pr == 1)

{

printf("Line Number : %d, Lexeme : SEMICOL - ;\n", line\_num);

}

addSym(SEMICOL);

}

else if (lookahead == '(')

{

if (pr == 1)

{

printf("Line Number : %d, Lexeme : BO - (\n", line\_num);

}

addSym(BO);

}

else if (lookahead == ')')

{

if (pr == 1)

{

printf("Line Number : %d, Lexeme : BC - )\n", line\_num);

}

addSym(BC);

}

else if (lookahead == '[')

{

if (pr == 1)

{

printf("Line Number : %d, Lexeme : SQBO - [\n", line\_num);

}

addSym(SQBO);

}

else if (lookahead == ']')

{

if (pr == 1)

{

printf("Line Number : %d, Lexeme : SQBC - ]\n", line\_num);

}

addSym(SQBC);

}

else if(buffer[pos] == '\n')

{

line\_num += 1;

}

pos += 1;

//ignore whitespaces

while((pos < l) && (buffer[pos] == ' ' || buffer[pos] == '\t'))

{

pos += 1;

}

lookahead = buffer[pos];

}

}

int populateLexemeTable(char \*opfile, hashTable table, int pr)

{

f\_read = fopen(opfile,"r");

first = (tokenInfo \*)malloc(sizeof(tokenInfo));

end = (tokenInfo \*)malloc(sizeof(tokenInfo));

first->df = -1;

end = first;

char buffer[BUFFSIZE];

error\_exist = 0;

while(1)

{

strcpy(buffer, getStream());

if(strcmp(buffer, "") == 0)

{

if (pr == 1)

{

printf("Line Number : %d, Lexeme : $\n", line\_num);

}

addDollar();

break;

}

DFA(table, buffer, pr);

}

scan = first;

comment\_flag = 0;

line\_num = 1;

id\_num = 1;

return error\_exist;

}

void removeComments(char \*testcaseFile, char \*cleanFile)

{

FILE \*f\_write = fopen(cleanFile, "w");

f\_read = fopen(testcaseFile, "r");

char buffer[BUFFSIZE];

strcpy(buffer, "\0");

comment\_flag = 0;

while(1)

{

strcpy(buffer, getStream());

if(strcmp(buffer, "") == 0)

{

break;

}

int pos = 0;

int l = strlen(buffer);

while(pos < l)

{

//ignore content between the comments

if (comment\_flag == 1)

{

while((pos < l) && (!(buffer[pos] == '\*' && buffer[pos+1] == '\*')))

{

pos += 1;

}

}

while((pos < l) && (!(buffer[pos] == '\*' && buffer[pos+1] == '\*')))

{

fprintf(f\_write, "%c", buffer[pos]);

pos += 1;

}

if(pos != l)

{

//found start/end of comment

pos += 2;

if(comment\_flag == 1)

{

comment\_flag ^= 1;

fprintf(f\_write, "\n");

}

else

{

comment\_flag ^= 1;

}

}

}

}

fclose(f\_write);

}

**lexer.h**

/\*

\* COMPILER PROJECT- ERPLAG COMPILER

\* CSE - D 2019

\* Sahil Garg, Jenit Jain, Amrit Goyal

\*/

#ifndef lexer

#define lexer

#include "lexerDef.h"

#endif

//List of functions for Hashing for Keywords

hashTable initializeHashTable(int l);

int hashValue(char \*key);

store findKeyword(hashTable T, char \*searchKey);

void addKeyword(hashTable T, int c, char \*key);

//List of functions for Lexemes/Tokens

tokenInfo\* initToken();

void addDollar();

void addSym(int n);

void addID(char \*id);

void addRNum(float num, int rnum\_no);

void addNum(int num, int num\_no);

void DFA(hashTable table, char \*buf, int pr);

char\* getStream();

tokenInfo\* getFirstToken();

tokenInfo\* getNextToken();

void removeComments(char \*testcaseFile, char \*cleanFile);

int populateLexemeTable(char \*opfile, hashTable h, int f\_or\_c);

**lexerDef.h**

/\*

\* COMPILER PROJECT- ERPLAG COMPILER

\* CSE - D 2019

\* Sahil Garg, Jenit Jain, Amrit Goyal

\*/

#ifndef lexerDef

#define lexerDef

//List of Definitions for Hashing Keywords

typedef enum{available, occupied, deleted}status;

struct linklist {

int code;

char word[21];

struct linklist \*next;

};

struct store {

int code;

char word[21];

};

struct entry {

int code;

char word[21];

status flag;

struct linklist \*start;

};

typedef struct entry entry;

typedef struct store store;

typedef struct linklist linklist;

typedef entry \*hashTable;

//List of Definitions for Lexemes/Toxens

typedef enum

{

INTEGER, REAL, BOOLEAN, OF, ARRAY, START, END, DECLARE, MODULE, DRIVER,

PROGRAM, GET\_VALUE, PRINT, USE, WITH, PARAMETERS, TRUE\_, FALSE\_, TAKES,

INPUT, RETURNS, AND, OR, FOR, IN, SWITCH, CASE, BREAK, DEFAULT, WHILE,

PLUS, MINUS, MUL, DIV, LT, LE, GE, GT, EQ, NE, DRIVERDEF, DRIVERENDDEF,

DEF, ENDDEF, COLON, RANGEOP, SEMICOL, COMMA, ASSIGNOP, SQBO, SQBC, BO,

BC, COMMENTMARK, ID, NUM, RNUM, $, e, null\_point = -1

}term;

struct tokenInfo

{

term s;

//common place to store the identifiers

union

{

int i;

float f;

char s[10];

}val;

int lno, df;

struct tokenInfo \*n;

struct tokenInfo \*prev;

};//term structure

typedef struct tokenInfo tokenInfo;

#endif

**parser.c**

/\*

\* COMPILER PROJECT- ERPLAG COMPILER

\* CSE - D 2019

\* Sahil Garg, Jenit Jain, Amrit Goyal

\*/

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <assert.h>

#include "parser.h"

#include "parserDef.h"

char \*token[TSIZE] =

{

"INTEGER", "REAL", "BOOLEAN", "OF", "ARRAY", "START", "END", "DECLARE", "MODULE", "DRIVER",

"PROGRAM", "GET\_VALUE", "PRINT", "USE", "WITH", "PARAMETERS", "TRUE\_", "FALSE\_", "TAKES",

"INPUT", "RETURNS", "AND", "OR", "FOR", "IN", "SWITCH", "CASE", "BREAK", "DEFAULT", "WHILE",

"PLUS", "MINUS", "MUL", "DIV", "LT", "LE", "GE", "GT", "EQ", "NE", "DRIVERDEF", "DRIVERENDDEF",

"DEF", "ENDDEF", "COLON", "RANGEOP", "SEMICOL", "COMMA", "ASSIGNOP", "SQBO", "SQBC", "BO",

"BC", "COMMENTMARK", "ID", "NUM", "RNUM", "$", "e", "null\_point"

};

char \*lexeme[TSIZE] =

{

"integer", "real", "boolean", "of", "array", "start", "end", "declare", "module", "driver",

"program", "get\_value", "print", "use", "with", "parameters", "true", "false", "takes",

"input", "returns", "AND", "OR", "for", "in", "switch", "case", "break", "default", "while",

"+", "-", "\*", "/", "<", "<=", ">", ">=", "==", "!=", "<<<", ">>>", "<<", ">>", ":", "..",

";", ",", ":=", "[", "]", "(", ")", "\*\*", "", "", "", "$", "e", "null\_point"

};

char \*nonterminal\_map[NTSIZE] =

{

"program", "moduleDeclarations", "moduleDeclaration", "otherModules", "driverModule", "module",

"ret", "input\_plist", "N1", "output\_plist", "N2", "dataType", "type", "moduleDef", "statements",

"statement", "ioStmt", "var", "whichID", "simpleStmt", "assignmentStmt", "whichStmt",

"lvalueIDStmt", "lvalueARRStmt", "index\_nt", "moduleReuseStmt", "optional", "idList", "N3",

"expression", "arithmeticOrBooleanExpr", "N7", "AnyTerm", "N8", "arithmeticExpr", "N4", "term\_1",

"N5", "factor", "op1", "op2", "logicalOp", "relationalOp", "declareStmt", "conditionalStmt",

"caseStmts", "N9", "value", "default\_1", "iterativeStmt", "range", "error"

};

int sz[TSIZE];

int sz2[TSIZE];

int vis[TSIZE];

static int table[NTSIZE][TSIZE];

char \*getLexeme(term t)

{

char \*s = (char \*)malloc(TSIZE\*sizeof(char));

if ((int)t < TSIZE)

{

strcpy(s, lexeme[(int)t]);

}

else

{

strcpy(s, "error");

}

return s;

}

char \*getNonTerm(non\_term t)

{

char \*s = (char \*)malloc(TSIZE\*sizeof(char));

strcpy(s, nonterminal\_map[(int)t]);

return s;

}

char \*getToken(term t)

{

char \*s = (char \*)malloc(TSIZE\*sizeof(char));

strcpy(s, token[(int)t]);

return s;

}

void initParseTable()

{

int i, j;

for(i = 0; i < NTSIZE; ++i)

{

for(j = 0; j < TSIZE; ++j)

{

table[i][j] = -1;

}

}

}

void fillFirstRules(gNode n, int idx, Grammar G)

{

int i;

for(i = 0; i < 20; ++i)

{

if (first[(int)n->ge.nt][i] == -1)

{

break;

}

if (first[(int)n->ge.nt][i] == (term)e)

{

if (n->next == NULL)

{

fillFollowRules(G[idx].t, idx, G);

}

else

{

if (n->next->t == terminal)

{

table[(int)G[idx].t][(int)n->next->ge.t] = idx;

}

else

{

fillFirstRules(n->next, idx, G);

}

}

}

else

{

table[(int)G[idx].t][(int)first[(int)n->ge.nt][i]] = idx;

}

}

}

void fillFollowRules(non\_term nt, int idx, Grammar G)

{

int i;

for(i = 0; i < 20; ++i)

{

if (follow[(int)nt][i] == -1)

{

break;

}

table[(int)G[idx].t][(int)follow[(int)nt][i]] = idx;

}

}

void createParseTable(Grammar G)

{

initParseTable();

int i;

for(i = 0; i < RULECNT; ++i)

{

if (G[i].top->t == terminal)

{

if (G[i].top->ge.t != (term)e)

{

table[(int)G[i].t][(int)G[i].top->ge.t] = i;

}

else

{

fillFollowRules(G[i].t, i, G);

}

}

else

{

fillFirstRules(G[i].top, i, G);

}

}

}

gNode initGrammarNode(gElement ge, tag t)

{

gNode temp = (gNode)malloc(sizeof(struct gNode));

if (temp != NULL)

{

temp->ge = ge;

temp->t = t;

temp->next = NULL;

}

return temp;

}

Grammar createGrammar()

{

int i, j, k;

FILE \*fp = fopen("Grammar.txt", "r");

char data[150], temp[150];

Grammar G = (Grammar)malloc(RULECNT \* sizeof(gHead));

for(i = 0; i < RULECNT; ++i)

{

fgets(data, 150, fp);

int len = strlen(data), eq = 0;

for(j = 0; j < len; ++j)

{

if (data[j] == '<')

{

int pos = 0;

j += 1;

while(j < len && data[j] != '>')

{

temp[pos] = data[j];

j += 1;

pos += 1;

}

temp[pos] = '\0';

if (eq == 0)

{

for(k = 0; k < NTSIZE; ++k)

{

if (strcmp(nonterminal\_map[k], temp) == 0)

{

G[i].t = (non\_term)k;

G[i].top = NULL;

break;

}

}

}

else

{

for(k = 0; k < TSIZE; ++k)

{

if (strcmp(nonterminal\_map[k], temp) == 0)

{

gNode s = initGrammarNode((gElement)(non\_term)k, non\_terminal);

gNode last = G[i].top;

if (last == NULL)

{

G[i].top = s;

}

else

{

while(last->next != NULL)

{

last = last->next;

}

last->next = s;

}

break;

}

}

}

}

else if (data[j] == '=')

{

eq = 1;

}

else if (data[j] == ' ')

{

continue;

}

else

{

int pos = 0;

while(j < len && data[j] != ' ' && data[j] != '\n' && data[j] != '\r')

{

temp[pos] = data[j];

j += 1;

pos += 1;

}

temp[pos] = '\0';

for(k = 0; k < TSIZE; ++k)

{

if (strcmp(token[k], temp) == 0)

{

gNode s = initGrammarNode((gElement)(term)k, terminal);

gNode last = G[i].top;

if (last == NULL)

{

G[i].top = s;

}

else

{

while(last->next != NULL)

{

last = last->next;

}

last->next = s;

}

break;

}

}

}

}

}

fclose(fp);

return G;

}

void printGrammar(Grammar G)

{

int i;

for(i = 0; i < RULECNT; ++i)

{

printf("<%s> =", nonterminal\_map[(int)G[i].t]);

gNode temp = G[i].top;

while(temp != NULL)

{

if (temp->t == terminal)

{

printf(" %s", token[(int)temp->ge.t]);

}

else

{

printf(" <%s>", nonterminal\_map[(int)temp->ge.nt]);

}

temp = temp->next;

}

printf("\n");

}

}

void addFirst(term q, int ind)

{

int i;

for(i=0; i < sz[ind]; i++)

{

if(first[ind][i] == q)

{

break;

}

}

if(i == sz[ind])

{

first[ind][sz[ind]] = q;

sz[ind]++;

}

}

void addFollow(term q, int ind)

{

int i;

for(i=0;i < sz2[ind]; i++)

{

if(follow[ind][i] == q)

{

break;

}

}

if(i == sz2[ind])

{

follow[ind][sz2[ind]] = q;

sz2[ind]++;

}

}

void dfs(non\_term node, Grammar G)

{

if(vis[(int)node])

{

return;

}

int i, j;

for(i = 0; i < RULECNT; i++)

{

if(G[i].t == node)

{

gNode cur = G[i].top;

while(1)

{

if(cur->t == terminal)

{

addFirst(cur->ge.t, (int)node);

break;

}

dfs(cur->ge.nt,G);

int nul = 0;

int q = (int)cur->ge.nt;

for(j = 0; ; ++j)

{

if (first[q][j] == -1)

{

break;

}

if(strcmp(token[(int)first[q][j]],"e")==0)

{

nul = 1;

}

else

{

addFirst(first[q][j], (int)node);

}

}

if(nul == 1)

{

cur = cur->next;

if(cur == NULL)

{

addFirst((term)e,(int)node);

break;

}

}

else

{

break;

}

}

}

}

vis[(int)node] = 1;

}

void findFirst(Grammar G)

{

int i, j;

for(i = 0; i < TSIZE; ++i)

{

sz[i] = 0;

vis[i] = 0;

for(j = 0; j < 20; ++j)

{

first[i][j] = -1;

}

}

for(i = 0; i < RULECNT; ++i)

{

non\_term tmp=G[i].t;

if(!vis[(int)tmp])

{

dfs(tmp,G);

vis[(int)tmp] = 1;

}

}

// printf("FIRST : \n");

// for(i = 0; i < TSIZE; ++i)

// {

// printf("%s : ", nonterminal\_map[i]);

// for(j = 0; j < 20; ++j)

// {

// if (first[i][j] == -1)

// {

// break;

// }

// printf("%s ", token[first[i][j]]);

// }

// printf("\n");

// }

}

void findFollow(Grammar G)

{

int i, j, k;

for(i = 0; i < TSIZE; ++i)

{

sz2[i] = 0;

for(j = 0; j < 20; ++j)

{

follow[i][j] = -1;

}

}

addFollow((term)$, 0);

for(k = 0; k < 2; ++k)

{

for(i = 0; i < RULECNT; ++i)

{

gNode cur = G[i].top, nxt;

non\_term par = G[i].t;

while(cur != NULL)

{

if(cur->t == terminal)

{

cur = cur->next;

}

else

{

nxt = cur->next;

while(nxt != NULL)

{

if(nxt->t == terminal)

{

addFollow(nxt->ge.t, (int)(cur->ge.nt));

break;

}

else

{

int flg = 0;

int q = (int)(nxt->ge.nt);

for(j = 0; j < sz[q]; ++j)

{

if(strcmp(token[(int)first[q][j]], "e") == 0)

{

flg=1;

}

else

{

addFollow(first[q][j],(int)(cur->ge.nt));

}

}

if(!flg)

{

break;

}

else

{

nxt=nxt->next;

}

}

}

if(nxt == NULL)

{

int q = (int)par;

for(j = 0; j < sz2[q]; ++j)

{

addFollow(follow[q][j],(int)(cur->ge.nt));

}

}

cur = cur->next;

}

}

}

}

// printf("FOLLOW : \n");

// for(i = 0; i < TSIZE; ++i)

// {

// printf("%s : ", nonterminal\_map[i]);

// for(j = 0; j < 20; ++j)

// {

// if (follow[i][j] == -1)

// {

// break;

// }

// printf("%s ", token[follow[i][j]]);

// }

// printf("\n");

// }

}

pNode returnCurrent(pNode p)

{

while (p != NULL && (p->child.size == p->no\_of\_child))

{

p = p->parent;

}

return p;

}

Queue initQueue()

{

Queue q;

q.front = NULL;

q.back = NULL;

q.size = 0;

return q;

}

Queue enqueue(pNode q, pNode p)

{

p->parent = q;

if (q->child.size == 0)

{

q->child.front = p;

q->child.back = q->child.front;

}

else

{

q->child.back->next = p;

q->child.back = p;

if (q->child.front->next == NULL)

{

q->child.front->next = q->child.back;

}

}

q->child.size += 1;

return q->child;

}

PTStack pop(PTStack s)

{

PTStack temp;

gNode delme = s.top;

temp.top = s.top->next;

temp.size = s.size-1;

free(delme);

return temp;

}

PTStack mergeStack(gHead T, PTStack s)

{

//For given rule in gHead push the reverse of RHS in stack after poping top element

s = pop(s);

PTStack ms;

ms.top = NULL;

ms.size = 0;

gNode store;

while (T.top != NULL)

{

store = T.top;

T.top = T.top->next;

if (ms.size == 0)

{

store->next = NULL;

}

else

{

store->next = ms.top;

}

ms.top = store;

ms.size++;

}

while (ms.size > 0)

{

store = ms.top;

ms.top = ms.top->next;

ms.size--;

store->next = s.top;

s.top = store;

s.size++;

}

return s;

}

pNode createParseTreeNode(pNode Parent, tag t, int level)

{

no\_of\_parse\_tree\_nodes += 1;

pNode temp = (pNode)malloc(sizeof(struct pNode));

temp->parent = Parent;

temp->t = t;

temp->level = level;

temp->child = initQueue();

temp->next = NULL;

temp->token\_link = NULL;

return temp;

}

pNode createNonTerminal(pNode Parent, non\_term non\_term\_1, tag t, int no\_of\_child, int level)

{

pNode temp = createParseTreeNode(Parent, t, level);

temp->e.nt = non\_term\_1;

temp->no\_of\_child = no\_of\_child;

return temp;

}

pNode createTerminal(pNode Parent, term term\_1, tag t, int level, tokenInfo\* token)

{

pNode temp = createParseTreeNode(Parent, t, level);

temp->e.t = term\_1;

temp->no\_of\_child = 0;

temp->token\_link = token;

return temp;

}

parseTree initParseTree()

{

parseTree pt;

pt.root = NULL;

pt.height = 0;

pt.c\_node = pt.root;

return pt;

}

parseTree parseInputSourceCode(char \* testCaseFile, hashTable h, Grammar G, int print)

{

FILE \*fp;

fp = fopen(testCaseFile, "r");

int i, flag = 0;

parse\_status = 1;

parseTree P = initParseTree();

// populating Symbol Table and creating the Token file

int error\_lexer = populateLexemeTable(testCaseFile, h, 0);

if (error\_lexer == 1)

{

parse\_status = 0;

printf("Found errors in lexer. So parsing can't be done.\n");

return P;

}

no\_of\_parse\_tree\_nodes = 0;

// Initializing stack to contain the $, start Symbol= program;

PTStack stack;

stack.top = initGrammarNode((gElement)(term)$,terminal);

stack.size = 1;

gNode duplicate;

duplicate = initGrammarNode((gElement)(non\_term)program,non\_terminal);

duplicate->next = stack.top;

stack.top = duplicate;

stack.size += 1;

char \*stackToken = "\0";

char \*local\_token = "\0";

char \*local\_lexeme = "\0";

pNode temp;

tokenInfo\* currentToken;

currentToken = getNextToken();

int rule, stacksize\_i, stacksize\_f;

while((currentToken!=NULL) && (((stack.top->t == terminal) && (stack.top->ge.t != $)) || (stack.top->t == non\_terminal)))

{

//some changes occured in Grammer while handling $ and e

G = createGrammar();

if (stack.top->t == terminal)

{

stackToken = getToken(stack.top->ge.t);

if (stack.top->ge.t==e)

{

stack = pop(stack);

temp = createTerminal(P.c\_node, (term)e, (tag)terminal, P.c\_node->level+1, NULL);

P.c\_node->child = enqueue(P.c\_node,temp);

if (P.c\_node->level == P.height)

{

P.height = P.c\_node->level + 1;

}

P.c\_node = returnCurrent(P.c\_node);

}

else

{

if (stack.top->ge.t == currentToken->s)

{

stack = pop(stack);

temp = createTerminal(P.c\_node, (term)currentToken->s, (tag)terminal, P.c\_node->level+1, currentToken);

P.c\_node->child = enqueue(P.c\_node,temp);

if (P.c\_node->level == P.height)

{

P.height = P.c\_node->level+1;

}

P.c\_node = returnCurrent(P.c\_node);

currentToken = getNextToken();

}

else

{

local\_token = getToken(currentToken->s);

printf("\n%d\n", stack.top->ge.t);

stackToken = getToken(stack.top->ge.t);

if (currentToken->s == NUM)

{

printf("\nERROR: The token (%s) for lexeme (%d) does not match at line no. %d. The expected token here is %s", local\_token,currentToken->val.i,currentToken->lno,stackToken);

}

else if (currentToken->s == RNUM)

{

printf("\nERROR: The token (%s) for lexeme (%f) does not match at line no. %d. The expected token here is %s", local\_token,currentToken->val.f,currentToken->lno,stackToken);

}

else if (currentToken->s == ID)

{

printf("\nERROR: The token (%s) for lexeme (%s) does not match at line no. %d. The expected token here is %s", local\_token,currentToken->val.s,currentToken->lno,stackToken);

}

else

{

local\_lexeme = getLexeme(currentToken->s);

printf("\nERROR: The token (%s) for lexeme (%s) does not match at line no. %d. The expected token here is %s", local\_token,local\_lexeme,currentToken->lno,stackToken);

}

printf("\n%d\n",stack.top->ge.t);

flag = 1;

printf("\nParsing not successful.\n");

break;

}

}

}

else

{

stackToken = getNonTerm(stack.top->ge.nt);

rule = table[(int)stack.top->ge.nt][(int)currentToken->s];

if (rule != -1)

{

stacksize\_i = stack.size-1;

stack = (PTStack)mergeStack(G[rule],stack);

stacksize\_f = stack.size;

if (P.c\_node == NULL)

{

temp = createNonTerminal(P.c\_node,(non\_term)G[rule].t,(tag)non\_terminal,stacksize\_f-stacksize\_i,0);

P.root = temp;

P.c\_node = temp;

}

else

{

temp = createNonTerminal(P.c\_node,(non\_term)G[rule].t,(tag)non\_terminal,stacksize\_f-stacksize\_i,P.c\_node->level+1);

P.c\_node->child = enqueue(P.c\_node,temp);

if (P.c\_node->level == P.height)

{

P.height = P.c\_node->level+1;

}

P.c\_node = P.c\_node->child.back;

}

}

else

{

local\_token = getToken(currentToken->s);

if (currentToken->s == NUM)

{

printf("\nERROR: The token (%s) for lexeme (%d) does not match at line no. %d. The expected token here are %s", local\_token,currentToken->val.i,currentToken->lno,stackToken);

}

else if (currentToken->s == RNUM)

{

printf("\nERROR: The token (%s) for lexeme (%f) does not match at line no. %d. The expected token here are %s", local\_token,currentToken->val.f,currentToken->lno,stackToken);

}

else if (currentToken->s == ID)

{

printf("\nERROR: The token (%s) for lexeme (%s) does not match at line no. %d. The expected token here are %s", local\_token,currentToken->val.s,currentToken->lno,stackToken);

}

else

{

local\_lexeme = getLexeme(currentToken->s);

printf("\nERROR: The token (%s) for lexeme (%s) does not match at line no. %d. The expected token here is %s", local\_token,local\_lexeme,currentToken->lno,stackToken);

}

for (i = 0; i < TSIZE; ++i)

{

if (table[(int)stack.top->ge.nt][i] != -1)

{

stackToken = getToken(i);

printf("\n %s",stackToken);

}

}

printf("\nParsing not successful.\n");

flag = 1;

break;

}

}

}

if(flag == 0)

{

parse\_status = 1;

printf("\nParsing successful.\n");

}

else

{

parse\_status = 0;

}

return P;

}

void printNode(pNode P)

{

char \*p, \*s, \*eps, \*dash, \*yes, \*no;

eps = (char \*)"e";

dash = (char \*)"---";

yes = (char \*)"y";

no = (char \*)"n";

if (P->no\_of\_child == 0)

{

if (P->e.t != e)

{

s = getToken(P->token\_link->s);

p = getNonTerm(P->parent->e.nt);

if (P->e.t == NUM)

{

printf("\n%-15d:%-5d:%-12s:%-10d:%-25s:%-12s:%-20s",P->token\_link->val.i, P->token\_link->lno, s, P->token\_link->val.i, p, yes, s);

}

else

{

if (P->e.t == RNUM)

{

printf("\n%-15f:%-5d:%-12s:%-10f:%-25s:%-12s:%-20s",P->token\_link->val.f, P->token\_link->lno, s, P->token\_link->val.f, p, yes, s);

}

else

{

if (P->e.t == ID)

{

printf("\n%-15s:%-5d:%-12s:%-10s:%-25s:%-12s:%-20s",P->token\_link->val.s, P->token\_link->lno, s, no, p, yes, s);

}

else

{

char \*lex = getLexeme(P->token\_link->s);

printf("\n%-15s:%-5d:%-12s:%-10s:%-25s:%-12s:%-20s",lex, P->token\_link->lno, s, no, p, yes, s);

}

}

}

}

else

{

p = getNonTerm(P->parent->e.nt);

printf("\n%-15s:%-5s:%-12s:%-10s:%-25s:%-12s%-20s",eps, no, no, no, p, yes, no);

}

}

else

{

if (P->parent != NULL)

{

p = getNonTerm(P->parent->e.nt);

}

else

{

p = "ROOT";

}

s = getNonTerm(P->e.nt);

printf("\n%-15s:%-5s:%-12s:%-10s:%-25s:%-12s:%-20s",no, no, no, no, p, no, s);

}

}

void printParseTreeQueue(pNode p)

{

if (p->no\_of\_child == 0)

{

printNode(p);

}

else

{

int br = 0;

pNode trav = p->child.front;

while(trav != NULL)

{

pNode doit = trav;

pNode store = trav->next;

printParseTreeQueue(doit);

trav = store;

br = 1;

break;

}

//to ensure we found the leftmost child and it is not NULL

assert(br == 1);

printNode(p);

while(trav != NULL)

{

pNode doit = trav;

pNode store = trav->next;

printParseTreeQueue(doit);

trav = store;

}

}

}

void printParseTree(char \* testCaseFile, hashTable h, Grammar G)

{

parseTree Tree = parseInputSourceCode(testCaseFile, h, G, 0);

if (Tree.height == 0)

{

return;

}

if (parse\_status == 0)

{

printf("There are errors in parsing. So no parsetree is created\n");

return ;

}

printf("%-15s:%-5s:%-12s:%-10s:%-25s:%-12s:%-20s","lexCurrentNode", "lNo", "token", "valIfNo", "pNode", "IsLeaf(y/n)", "nodeSymbol");

printParseTreeQueue(Tree.root);

}

**parser.h**

/\*

\* COMPILER PROJECT- ERPLAG COMPILER

\* CSE - D 2019

\* Sahil Garg, Jenit Jain, Amrit Goyal

\*/

#ifndef parser

#define parser

#include "parserDef.h"

#include <stdio.h>

#include <stdlib.h>

#endif

//List of functions for Initialising Grammer

char \*getLexeme(term t);

char \*getNonTerm(non\_term t);

char \*getToken(term t);

void initParseTable();

void fillFirstRules(gNode n, int idx, Grammar G);

void fillFollowRules(non\_term nt, int idx, Grammar G);

void createParseTable(Grammar G);

gNode initGrammarNode(gElement ge, tag t);

Grammar createGrammar();

void printGrammar(Grammar G);

void addFirst(term q, int ind);

void addFollow(term q, int ind);

void dfs(non\_term node, Grammar G);

void findFirst(Grammar G);

void findFollow(Grammar G);

//List of functions for parser & ParseTree

pNode returnCurrent(pNode p);

Queue initQueue();

Queue enqueue(pNode q, pNode p);

PTStack pop(PTStack s);

PTStack mergeStack(gHead T, PTStack s);

pNode createParseTreeNode(pNode Parent, tag t, int level);

pNode createNonTerminal(pNode Parent, non\_term nt, tag t, int no\_of\_child, int level);

pNode createTerminal(pNode Parent, term term\_1, tag t, int level, tokenInfo\* token);

parseTree initParseTree();

parseTree parseInputSourceCode(char \* testCaseFile, hashTable h, Grammar G, int print);

void printParseTreeQueue(pNode p);

void printParseTree(char \* testCaseFile, hashTable h, Grammar G);

**parserDef.h**

/\*

\* COMPILER PROJECT- ERPLAG COMPILER

\* CSE - D 2019

\* Sahil Garg, Jenit Jain, Amrit Goyal

\*/

#ifndef parserDef

#define parserDef

#include "lexer.h"

#include "lexerDef.h"

#define RULECNT 96

#define NTSIZE 52

#define TSIZE 60

//List of Definitions for Parser

int parse\_status;

int no\_of\_parse\_tree\_nodes;

typedef enum

{

program, moduleDeclarations, moduleDeclaration, otherModules, driverModule, module, ret,

input\_plist, N1, output\_plist, N2, dataType, type, moduleDef, statements, statement,

ioStmt, var, whichID, simpleStmt, assignmentStmt, whichStmt, lvalueIDStmt, lvalueARRStmt,

index\_nt, moduleReuseStmt, optional, idList, N3, expression, arithmeticOrBooleanExpr, N7,

AnyTerm, N8, arithmeticExpr, N4, term\_1, N5, factor, op1, op2, logicalOp, relationalOp,

declareStmt, conditionalStmt, caseStmts, N9, value, default\_1, iterativeStmt, range

}non\_term;

typedef union

{

term t;

non\_term nt;

}gElement;

typedef enum {terminal, non\_terminal} tag;

term first[TSIZE][20];

term follow[TSIZE][20];

struct gNode

{

tag t;

gElement ge;

struct gNode\* next;

};

struct gHead

{

non\_term t;

struct gNode\* top;

};

struct PTStack

{

struct gNode\* top;

int size;

};

typedef struct pNode \*pNode;

typedef struct Queue Queue;

struct Queue

{

pNode front;

pNode back;

int size;

};

struct pNode

{

tag t;

gElement e;

int level, no\_of\_child;

tokenInfo\* token\_link;

pNode next;

pNode parent;

Queue child;

};

struct parseTree

{

pNode root;

pNode c\_node;

int height;

};

typedef struct gHead gHead;

typedef struct gNode \*gNode;

typedef struct gHead \*Grammar;

typedef struct parseTree parseTree;

typedef struct PTStack PTStack;

#endif

**symbolDef.h**

/\*

\* COMPILER PROJECT- ERPLAG COMPILER

\* CSE - D 2019

\* Sahil Garg, Jenit Jain, Amrit Goyal

\*/

#ifndef symbolDef

#define symbolDef

#include "lexerDef.h"

//List of Definitions for Hashing Scoped variables/functions

typedef struct IDEntry IDEntry;

typedef enum{defined, declared}funcStatus;

#define hash\_capacity\_2 31

struct linklist2

{

IDEntry \*ID;

struct linklist2 \*next;

};

struct store2

{

int code;

IDEntry \*node;

};

struct entry2

{

IDEntry \*ID;

status flag;

struct linklist2 \*start;

};

typedef struct store2 store2;

typedef struct entry2 entry2;

typedef struct linklist2 linklist2;

typedef entry2 \*hashTable2;

//List of Definitions for Symbol Table entries for variables/functions

int semantic\_status;

struct variable

{

term type;

};

struct array

{

term type;

int s\_idx, e\_idx;

};

struct IDVariable

{

char word[21];

int lno;

int scope;

int code;

int v\_a;

int bytes;

int depth;

int offset;

char func\_name[21];

union

{

struct variable v;

struct array a;

}var;

};

struct parameter

{

union

{

struct variable v;

struct array a;

}var;

int v\_a;

struct parameter \*next;

};

struct IDFunction

{

char word[21];

int lno;

int used;

int scope;

funcStatus status;

struct parameter \*inputList;

struct parameter \*outputList;

};

struct IDEntry

{

union

{

struct IDVariable ivar;

struct IDFunction ifunc;

}entity;

};

struct scopeChain

{

int scope;

int in\_cond;

int in\_loop;

struct scopeChain \*next;

struct scopeChain \*prev;

};

struct totalScopeList

{

int scope\_start;

int scope\_end;

};

typedef struct variable variable;

typedef struct array array;

typedef struct IDVariable IDVariable;

typedef struct scopeChain scopeChain;

typedef struct totalScopeList totalScopeList;

typedef struct parameter parameter;

typedef struct IDFunction IDFunction;

#endif

**symbolTable.c**

/\*

\* COMPILER PROJECT- ERPLAG COMPILER

\* CSE - D 2019

\* Sahil Garg, Jenit Jain, Amrit Goyal

\*/

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include <assert.h>

#include "parser.h"

#include "symbolDef.h"

#include "symbolTable.h"

static int id\_counter = 0;

int getBytes(term s)

{

if(s == INTEGER)

{

return 2;

}

else if(s == REAL)

{

return 4;

}

else

{

return 1;

}

}

int checkModuleReturn(IDEntry\* ctrl\_var, tokenInfo\* token, hashTable2 tableID)

{

int flag = 1;

while(token->s != START)

{

token = token->n;

}

token = token->n;

//check till function definition ends

//do not use "END" as marker symbol as for/while/switch can exist inside function

while(token != NULL && token->s != DEF && token->s != DRIVERDEF)

{

if(token->s != ID)

{

token = token->n;

continue;

}

if(strcmp(ctrl\_var->entity.ivar.word, token->val.s) != 0)

{

token = token->n;

continue;

}

//idntifier matched with one in output parameter list

//if user take an input value

if(token->prev != NULL && token->prev->prev != NULL && token->prev->prev->s == GET\_VALUE)

{

flag = 0;

break;

}

token = token->n;

//or user assigns an input value

while(token != NULL && token->s != SEMICOL)

{

if(token->s == ASSIGNOP)

{

flag = 0;

break;

}

token = token->n;

}

if(flag == 0)

{

break;

}

}

return flag;

}

void checkFor(IDEntry\* ctrl\_var, tokenInfo\* token, hashTable2 tableID, int print)

{

int flag;

while(token != NULL && token->s != START)

{

token = token->n;

}

token = token->n;

//check till for loop ends

while(token != NULL && token->s != END)

{

flag = 0;

if(token->s != ID)

{

token = token->n;

continue;

}

if(strcmp(ctrl\_var->entity.ivar.word, token->val.s) != 0)

{

token = token->n;

continue;

}

//identifier matched with one in for loop declaration

assert(token->prev->prev != NULL);

if(token->prev->prev->s == GET\_VALUE)

{

if(print == 1)

{

printf("\nERROR : The for statement must not redefine the variable \"%s\" at line number %d", ctrl\_var->entity.ifunc.word, token->lno);

}

semantic\_status = 0;

token = token->n;

continue;

}

//check if variable is not of array type

tokenInfo\* go = token;

while(go->prev->s != SEMICOL && go->prev->s != START)

{

go = go->prev;

}

if(go->s == ID && go->n->s == SQBO)

{

flag = 1;

}

token = token->n;

while(token != NULL && token->s != SEMICOL)

{

if(token->s == ASSIGNOP && flag == 0)

{

if(print == 1)

{

printf("\nERROR : The for statement must not redefine the variable \"%s\" at line number %d", ctrl\_var->entity.ifunc.word, token->lno);

}

semantic\_status = 0;

while(token->s != SEMICOL)

{

token = token->n;

}

break;

}

token = token->n;

}

}

}

void createIDTable(tokenInfo\* token, totalScopeList\* scope\_tot, hashTable2 tableID, hashTable2 tableFunc, int print)

{

store2 res;

scopeChain\* TREE = (scopeChain\*)malloc(sizeof(scopeChain));

TREE->scope = 0;

TREE->in\_cond = 0;

TREE->in\_loop = 0;

TREE->next = NULL;

TREE->prev = NULL;

int universal\_scope = 0;

char current\_function[20];

strcpy(current\_function, " ");

char name;

int i = -2;

int default\_flag = 0;

int switch\_type;

IDEntry\* for\_start = NULL;

int switch\_end;

int switch\_start;

int switch\_error[3];

int j;

for(j = 0; j < 3; ++j)

{

switch\_error[j] = -1;

}

int scope\_level = 0, function\_offset = 0;

while(token != NULL)

{

if(token->s == MODULE)

{

assert(token->n != NULL);

res = findScope(tableFunc, token->n->val.s, TREE->scope, 1);

if(token->prev != NULL && token->prev->s != USE)

{

// function declaration/definition starts, so current function name changes

if(token->prev->s == DEF)

{

scope\_level = 0;

function\_offset = 0;

strcpy(current\_function, token->n->val.s);

}

//function name not found

if(res.code == -1)

{

token = token->n;

IDEntry\* par\_data = (IDEntry\*)malloc(sizeof(IDEntry));

strcpy(par\_data->entity.ifunc.word, token->val.s);

if(token->prev->prev->s == DECLARE)

{

par\_data->entity.ifunc.status = declared;

par\_data->entity.ifunc.inputList = NULL;

par\_data->entity.ifunc.outputList = NULL;

}

else

{

par\_data->entity.ifunc.scope = TREE->scope;

par\_data->entity.ifunc.lno = token->lno;

par\_data->entity.ifunc.status = defined;

par\_data->entity.ifunc.inputList = NULL;

par\_data->entity.ifunc.outputList = NULL;

}

addScope(tableFunc, par\_data, 1);

}

else

{

assert(token->prev != NULL);

if(token->prev->s == DEF)

{

token = token->n;

if(res.node->entity.ifunc.status == declared)

{

res.node->entity.ifunc.scope = TREE->scope;

res.node->entity.ifunc.lno = token->lno;

res.node->entity.ifunc.status = defined;

}

else

{

if (print == 1)

{

printf("\nERROR : Overloading of module \"%s\" at line number %d not allowed",current\_function,token->lno);

}

semantic\_status = 0;

}

}

else if(token->prev->s == DECLARE)

{

token = token->n;

if (print == 1)

{

printf("\nERROR : Redundant declaration of module \"%s\" at line number %d",current\_function,token->lno);

}

semantic\_status = 0;

}

}

}

else

{

//Function calling statements

token = token->n;

if(strcmp(current\_function, token->val.s) == 0)

{

if (print == 1)

{

printf("\nERROR : Function \"%s\" at line number %d cannot be invoked recursively",current\_function,token->lno);

}

semantic\_status = 0;

}

else if(res.code == -1)

{

if (print == 1)

{

printf("\nERROR : Undeclared module \"%s\" at line number %d",token->val.s, token->lno);

}

semantic\_status = 0;

}

else

{

res.node->entity.ifunc.used = 1;

}

}

}

else if(token->s == INPUT)

{

token = token->n->n;

while (token != NULL && token->s != SQBC)

{

//ignore comma in input list of function

if(token->s == COMMA)

{

token = token->n;

}

IDEntry\* par\_data = (IDEntry\*)malloc(sizeof(IDEntry));

strcpy(par\_data->entity.ivar.word, token->val.s);

par\_data->entity.ivar.code = id\_counter++;

par\_data->entity.ivar.lno = token->lno;

//ignore the colon

assert(token->n != NULL);

token = token->n->n;

if(token->s == ARRAY)

{

par\_data->entity.ivar.v\_a = 1;

//ignore opening square bracket

token = token->n->n;

par\_data->entity.ivar.var.a.s\_idx = token->val.i;

//ignore range operator

token = token->n->n;

par\_data->entity.ivar.var.a.e\_idx = token->val.i;

//ignore closing bracket and "of"

token = token->n->n->n;

assert(token != NULL);

par\_data->entity.ivar.var.a.type = token->s;

par\_data->entity.ivar.bytes = getBytes(token->s);

}

else

{

par\_data->entity.ivar.v\_a = 0;

par\_data->entity.ivar.var.v.type = token->s;

par\_data->entity.ivar.bytes = getBytes(token->s);

}

par\_data->entity.ivar.scope = TREE->scope;

res = findScope(tableID, par\_data->entity.ivar.word, TREE->scope, 0);

if(res.code >= 0)

{

if (print == 1)

{

printf("\nERROR : Repeat declaration of identifier \"%s\" in function input list at line number %d",par\_data->entity.ivar.word, par\_data->entity.ivar.lno);

}

semantic\_status = 0;

}

else

{

strcpy(par\_data->entity.ivar.func\_name, current\_function);

par\_data->entity.ivar.depth = scope\_level + 1;

par\_data->entity.ivar.offset = function\_offset;

addScope(tableID, par\_data, 0);

int bytes = par\_data->entity.ivar.bytes;

if (par\_data->entity.ivar.v\_a == 1)

{

bytes \*= (par\_data->entity.ivar.var.a.e\_idx - par\_data->entity.ivar.var.a.s\_idx + 1);

}

function\_offset += bytes;

res = findScope(tableFunc,current\_function, TREE->scope, 1);

//create link list of parameter values

if(res.node->entity.ifunc.inputList == NULL)

{

parameter\* root = (parameter\*)malloc(sizeof(parameter));

if(par\_data->entity.ivar.v\_a == 0)

{

root->v\_a = 0;

root->var.v.type = par\_data->entity.ivar.var.v.type;

root->next = NULL;

}

else

{

root->v\_a = 1;

root->var.a.type = par\_data->entity.ivar.var.a.type;

root->var.a.s\_idx = par\_data->entity.ivar.var.a.s\_idx;

root->var.a.e\_idx = par\_data->entity.ivar.var.a.e\_idx;

root->next = NULL;

}

res.node->entity.ifunc.inputList = root;

}

else

{

parameter\* root = res.node->entity.ifunc.inputList;

while(root->next!=NULL)

{

root = root->next;

}

root->next = (parameter\*)malloc(sizeof(parameter));

if(par\_data->entity.ivar.v\_a==0)

{

root->next->v\_a = 0;

root->next->var.v.type = par\_data->entity.ivar.var.v.type;

root->next->next = NULL;

}

else

{

root->next->v\_a=1;

root->next->var.a.type = par\_data->entity.ivar.var.a.type;

root->next->var.a.s\_idx = par\_data->entity.ivar.var.a.s\_idx;

root->next->var.a.e\_idx = par\_data->entity.ivar.var.a.e\_idx;

root->next->next = NULL;

}

}

}

assert(token != NULL);

token = token->n;

}

}

else if(token->s == RETURNS)

{

token = token->n->n;

while(token != NULL && token->s != SQBC)

{

//ignore comma in outputlist of function

if(token->s == COMMA)

{

token = token->n;

}

IDEntry\* par\_data = (IDEntry\*)malloc(sizeof(IDEntry));

strcpy(par\_data->entity.ivar.word, token->val.s);

par\_data->entity.ivar.code = id\_counter++;

par\_data->entity.ivar.lno = token->lno;

//ignore the colon

assert(token->n != NULL);

token = token->n->n;

if (token->s == ARRAY)

{

//ignore opening square bracket

token = token->n->n;

//ignore range operator

token = token->n->n;

//ignore closing bracket and "of"

token = token->n->n->n;

assert(token != NULL);

if (print == 1)

{

printf("\nERROR : Output list of function can't contain array as parameter\n");

}

semantic\_status = 0;

}

else

{

par\_data->entity.ivar.v\_a = 0;

par\_data->entity.ivar.var.v.type = token->s;

par\_data->entity.ivar.bytes = getBytes(token->s);

par\_data->entity.ivar.scope = TREE->scope;

res = findScope(tableID, par\_data->entity.ivar.word, 0, TREE->scope);

if(res.code >= 0)

{

if (print == 1)

{

printf("\nERROR : Repeat declaration of identifier \"%s\" at line number %d",par\_data->entity.ivar.word,par\_data->entity.ivar.lno);

}

semantic\_status = 0;

}

else

{

strcpy(par\_data->entity.ivar.func\_name, current\_function);

par\_data->entity.ivar.depth = scope\_level + 1;

par\_data->entity.ivar.offset = function\_offset;

addScope(tableID, par\_data, 0);

int bytes = par\_data->entity.ivar.bytes;

if (par\_data->entity.ivar.v\_a == 1)

{

bytes \*= (par\_data->entity.ivar.var.a.e\_idx - par\_data->entity.ivar.var.a.s\_idx + 1);

}

function\_offset += bytes;

int invalid = checkModuleReturn(par\_data, token, tableID);

if(invalid == 1)

{

if (print == 1)

{

printf("\nERROR : The variable \"%s\" returned by the function \"%s\" is not assigned any value within the function definition", par\_data->entity.ivar.word, current\_function);

}

semantic\_status = 0;

}

res = findScope(tableFunc,current\_function, TREE->scope, 1);

//create link list of parameter values

if(res.node->entity.ifunc.outputList == NULL)

{

parameter\* root = (parameter\*)malloc(sizeof(parameter));

assert(par\_data->entity.ivar.v\_a == 0);

root->v\_a = 0;

root->var.v.type = par\_data->entity.ivar.var.v.type;

root->next = NULL;

res.node->entity.ifunc.outputList = root;

}

else

{

parameter\* root= res.node->entity.ifunc.outputList;

while(root->next != NULL)

{

root = root->next;

}

root->next=(parameter\*)malloc(sizeof(parameter));

assert(par\_data->entity.ivar.v\_a == 0);

root->next->v\_a=0;

root->next->var.v.type=par\_data->entity.ivar.var.v.type;

root->next->next=NULL;

}

}

}

token = token->n;

}

}

else if(token->s == DECLARE)

{

term type;

int V\_A;

int s\_idx;

int e\_idx;

assert(token->n != NULL);

//declaration of variables as module declaration checked before

if(token->n->s != MODULE)

{

token = token->n;

IDEntry\* par\_data = (IDEntry\*)malloc(sizeof(IDEntry));

strcpy(par\_data->entity.ivar.word, token->val.s);

par\_data->entity.ivar.code = id\_counter++;

par\_data->entity.ivar.lno = token->lno;

//ignore all the variable names for now to get datatype first

//information updated later

int ctr = -2; //as ctr reaches 0 only when atleast 2 variables are declared

while(token != NULL && token->s != COLON)

{

token = token->n;

ctr += 1;

}

//ignore colon

token = token->n;

if(token->s == ARRAY)

{

V\_A = 1;

par\_data->entity.ivar.v\_a = 1;

tokenInfo \*traverse = token;

//ignore opening square brackets

traverse = traverse->n->n;

s\_idx = traverse->val.i;

par\_data->entity.ivar.var.a.s\_idx = s\_idx;

//ignore range operator

traverse = traverse->n->n;

e\_idx = traverse->val.i;

par\_data->entity.ivar.var.a.e\_idx = e\_idx;

//ignore closing bracket and "of"

traverse = traverse->n->n->n;

assert(traverse != NULL);

type = traverse->s;

par\_data->entity.ivar.var.a.type = type;

par\_data->entity.ivar.bytes = getBytes(type);

}

else

{

V\_A = 0;

par\_data->entity.ivar.v\_a = 0;

type = token->s;

par\_data->entity.ivar.var.v.type = token->s;

par\_data->entity.ivar.bytes = getBytes(type);

}

par\_data->entity.ivar.scope = TREE->scope;

//process 1st variable

res = findScope(tableID, par\_data->entity.ivar.word, TREE->scope, 0);

if(res.code >= 0)

{

if (print == 1)

{

printf("\nERROR : Repeat declaration of identifier \"%s\" at line number %d",par\_data->entity.ivar.word, par\_data->entity.ivar.lno);

}

semantic\_status = 0;

}

else

{

strcpy(par\_data->entity.ivar.func\_name, current\_function);

par\_data->entity.ivar.depth = scope\_level;

par\_data->entity.ivar.offset = function\_offset;

addScope(tableID, par\_data, 0);

int bytes = par\_data->entity.ivar.bytes;

if (par\_data->entity.ivar.v\_a == 1)

{

bytes \*= (par\_data->entity.ivar.var.a.e\_idx - par\_data->entity.ivar.var.a.s\_idx + 1);

}

function\_offset += bytes;

}

token = token->prev;

assert(token->s == COLON);

while(ctr > 0)

{

token = token->prev;

ctr -= 1;

}

//ctr = 0 check to ensure this executes only in case of multiple variable declarations

while(ctr == 0 && token->s != COLON)

{

if (token->s == COMMA)

{

token = token->n;

continue;

}

IDEntry\* par\_data = (IDEntry\*)malloc(sizeof(IDEntry));

strcpy(par\_data->entity.ivar.word, token->val.s);

if(V\_A == 1)

{

par\_data->entity.ivar.v\_a = 1;

par\_data->entity.ivar.var.a.s\_idx = s\_idx;

par\_data->entity.ivar.var.a.e\_idx = e\_idx;

par\_data->entity.ivar.var.a.type = type;

par\_data->entity.ivar.bytes = getBytes(type);

}

else

{

par\_data->entity.ivar.v\_a = 0;

par\_data->entity.ivar.var.v.type = type;

par\_data->entity.ivar.bytes = getBytes(type);

}

par\_data->entity.ivar.code = id\_counter++;

par\_data->entity.ivar.scope = TREE->scope;

par\_data->entity.ivar.lno = token->lno;

res = findScope(tableID,par\_data->entity.ivar.word,TREE->scope,0);

if(res.code >= 0)

{

if (print == 1)

{

printf("\nERROR : Repeat declaration of identifier \"%s\" at line number %d",par\_data->entity.ivar.word,par\_data->entity.ivar.lno);

}

semantic\_status = 0;

}

else

{

strcpy(par\_data->entity.ivar.func\_name, current\_function);

par\_data->entity.ivar.depth = scope\_level;

par\_data->entity.ivar.offset = function\_offset;

addScope(tableID, par\_data, 0);

int bytes = par\_data->entity.ivar.bytes;

if (par\_data->entity.ivar.v\_a == 1)

{

bytes \*= (par\_data->entity.ivar.var.a.e\_idx - par\_data->entity.ivar.var.a.s\_idx + 1);

}

function\_offset += bytes;

}

token = token->n;

}

}

}

else if(token->s == ID)

{

scopeChain\* CURR = TREE;

//traverse the symbol table tree for checking declaration of variable

while(1)

{

res = findScope(tableID,token->val.s, CURR->scope, 0);

if(res.code != -1)

{

break;

}

else

{

if(CURR->prev != NULL)

{

CURR = CURR->prev;

}

else

{

break;

}

}

}

if(res.code == -1)

{

if (print == 1)

{

printf("\nERROR : Undeclared variable \"%s\" at line number %d",token->val.s, token->lno);

}

semantic\_status = 0;

}

}

else if(token->s == DEF)

{

TREE->scope = ++universal\_scope;

}

else if(token->s == DRIVERDEF)

{

TREE->scope = ++universal\_scope;

IDEntry\* par\_data = (IDEntry\*)malloc(sizeof(IDEntry));

strcpy(par\_data->entity.ifunc.word, "program");

par\_data->entity.ifunc.status = defined;

par\_data->entity.ifunc.scope = TREE->scope;

par\_data->entity.ifunc.lno = token->lno;

addScope(tableFunc,par\_data, 1);

strcpy(current\_function,"program");

}

else if(token->s == SWITCH)

{

switch\_start = (token->lno) + 1;

scopeChain\* go = (scopeChain\*)malloc(sizeof(scopeChain));

go->scope = ++universal\_scope;

go->next = NULL;

go->prev = TREE;

go->in\_cond = 0;

go->in\_loop = 0;

TREE->next = go;

TREE = TREE->next;

assert(token->n != NULL);

//ignore the opening bracket

token = token->n->n;

assert(token != NULL);

go = TREE->prev;

//check if switch case variable is declared or not

while(1)

{

res = findScope(tableID, token->val.s, go->scope, 0);

if(res.code != -1)

{

break;

}

else

{

if(go->prev != NULL)

{

go = go->prev;

}

else

{

break;

}

}

}

if(res.code == -1)

{

if (print == 1)

{

printf("\nERROR : Undeclared variable \"%s\" at line number %d",token->val.s, token->lno);

}

semantic\_status = 0;

}

else

{

switch\_type = res.node->entity.ivar.var.v.type;

if(switch\_type == REAL)

{

if (print == 1)

{

printf("\nERROR : Line %d has a switch statement with an identifier of type real.", token->lno);

}

semantic\_status = 0;

}

TREE->in\_cond = 1;

}

}

else if(token->s == CASE && TREE->in\_cond == 1)

{

token = token->n;

int case\_type = token->s;

if(switch\_type == INTEGER && case\_type != NUM)

{

switch\_error[0] = 1;

}

else if(switch\_type == BOOLEAN && case\_type != TRUE\_ && case\_type != FALSE\_)

{

switch\_error[1]=1;

}

}

else if(token->s == DEFAULT && TREE->in\_cond == 1)

{

if(switch\_type == BOOLEAN)

{

switch\_error[2]=1;

}

default\_flag = 1;

}

else if(token->s == FOR || token->s == WHILE)

{

scopeChain\* go = (scopeChain\*)malloc(sizeof(scopeChain));

go->scope = ++universal\_scope;

go->next = NULL;

go->prev = TREE;

go->in\_cond = 0;

go->in\_loop = 1;

TREE->next = go;

TREE = TREE->next;

if (token->s == FOR)

{

//ignore the brackets

token = token->n->n;

go = TREE->prev;

assert(token != NULL);

store2 ctrl\_var;

while(1)

{

ctrl\_var = findScope(tableID, token->val.s, go->scope, 0);

if(ctrl\_var.code != -1)

{

break;

}

else

{

if(go->prev != NULL)

{

go = go->prev;

}

else

{

break;

}

}

}

if (ctrl\_var.code == -1)

{

if (print == 1)

{

printf("\nERROR : Line %d has \"FOR\" loop variable undefined\n", token->lno);

}

semantic\_status = 0;

}

else

{

checkFor(ctrl\_var.node, token->prev->prev, tableID, print);

}

}

}

else if(token->s == START)

{

// printf("Entered Function : %s\n", current\_function);

scope\_tot[TREE->scope].scope\_start = token->lno;

scope\_level += 1;

}

else if(token->s == END)

{

// printf("Exiting function : %s\n", current\_function);

scope\_tot[TREE->scope].scope\_end = token->lno;

scope\_level -= 1;

if((TREE->in\_loop) == 1)

{

TREE = TREE->prev;

//move out of scope so remove the element from stack

free(TREE->next);

}

else if((TREE->in\_cond) == 1)

{

switch\_end = token->lno;

if(default\_flag == 0 && switch\_type != BOOLEAN)

{

if (print == 1)

{

printf("\nERROR : The case statements lines <%d> to <%d> must have a default statement.", switch\_start, switch\_end);

}

semantic\_status = 0;

}

default\_flag = 0;

if(switch\_error[0] == 1)

{

if (print == 1)

{

printf("\nERROR : The switch statement lines <%d> to <%d> cannot have case statement with case keyword followed by any other value other than an integer ",switch\_start, switch\_end);

}

semantic\_status = 0;

}

if(switch\_error[1] == 1)

{

if (print == 1)

{

printf("\nERROR : Lines <%d>to <%d> of the switch statement cannot have the case statements with integer labels.", switch\_start, switch\_end);

}

semantic\_status = 0;

}

if(switch\_error[2] == 1)

{

if (print == 1)

{

printf("\nERROR : The switch statement lines <%d> to <%d> should not have a default statement. ",switch\_start, switch\_end);

}

semantic\_status = 0;

}

int i;

for(i = 0; i < 3; ++i)

{

switch\_error[i] = -1;

}

TREE = TREE->prev;

//move out of scope so remove the element from stack

free(TREE->next);

}

}

token = token->n;

}

}

void secondRun(tokenInfo\* token, hashTable2 tableID, hashTable2 tableFunc, int print)

{

store2 res;

scopeChain\* TREE = (scopeChain\*)malloc(sizeof(scopeChain));;

TREE->scope = 0;

TREE->next = NULL;

TREE->prev = NULL;

TREE->in\_cond = 0;

TREE->in\_loop = 0;

int universal\_scope = 0;

char current\_function[10];

strcpy(current\_function," ");

while(token != NULL)

{

if(token->s == MODULE)

{

if(token->prev->s == DEF)

{

strcpy(current\_function,token->n->val.s);

}

}

else if(token->s == DEF)

{

TREE->scope = ++universal\_scope;

}

else if(token->s == DRIVERDEF)

{

TREE->scope = ++universal\_scope;

strcpy(current\_function, "program");

}

else if(token->s == SWITCH)

{

scopeChain\* go = (scopeChain\*)malloc(sizeof(scopeChain));

go->scope = ++universal\_scope;

go->next = NULL;

go->prev = TREE;

go->in\_cond = 0;

go->in\_loop = 0;

TREE->next = go;

TREE = TREE->next;

assert(token->n != NULL);

//ignore the opening bracket

token = token->n->n;

assert(token != NULL);

go = TREE->prev;

//check if switch case variable is declared or not

while(1)

{

res = findScope(tableID, token->val.s, go->scope, 0);

if(res.code != -1)

{

break;

}

else

{

if(go->prev != NULL)

{

go = go->prev;

}

else

{

break;

}

}

}

if (res.code != -1)

{

TREE->in\_cond = 1;

}

}

else if(token->s == END && (TREE->in\_cond == 1 || TREE->in\_loop == 1))

{

TREE = TREE->prev;

free(TREE->next);

}

else if(token->s == FOR || token->s == WHILE)

{

scopeChain\* go = (scopeChain\*)malloc(sizeof(scopeChain));

go->scope = ++universal\_scope;

go->next = NULL;

go->prev = TREE;

go->in\_cond = 0;

go->in\_loop = 1;

TREE->next = go;

TREE = TREE->next;

}

else if(token->s == USE)

{

int flag;

//ignore "module" lexeme

token = token->n->n;

res = findScope(tableFunc, token->val.s, 0, 1);

if(strcmp(current\_function, token->val.s) == 0)

{

//case of recursion, handled before

while(token != NULL && token->s != SEMICOL)

{

token = token->n;

}

}

else if(res.code == -1)

{

//case of undeclared function, handled before

while(token != NULL && token->s != SEMICOL)

{

token = token->n;

}

}

else

{

while(token->prev != NULL && token->prev->s != SEMICOL)

{

token = token->prev;

}

//check return type of function

store2 actual\_par;

parameter\* formal\_par = res.node->entity.ifunc.outputList;

if (formal\_par == NULL)

{

if (token->s == SQBO)

{

if (print == 1)

{

printf("\nERROR : The function \"%s\" should not return any value at line number %d",res.node->entity.ifunc.word,token->lno);

}

semantic\_status = 0;

}

}

else

{

if(token->s != SQBO)

{

if (print == 1)

{

printf("\nERROR : The values returned by function \"%s\" are not stored at line number %d",res.node->entity.ifunc.word,token->lno);

}

semantic\_status = 0;

}

else if(token->s == SQBO)

{

//ignore SQBO

//checking of variables if declared has been done in previously

token = token->n;

scopeChain\* go = TREE;

while(1)

{

actual\_par = findScope(tableID, token->val.s, go->scope,0);

if(actual\_par.code != -1)

{

break;

}

else

{

if (go->prev != NULL)

{

go = go->prev;

}

else

{

break;

}

}

}

flag = 0;

while(formal\_par != NULL)

{

if(actual\_par.code != -1)

{

if(actual\_par.node->entity.ivar.var.v.type != formal\_par->var.v.type)

{

if (print == 1)

{

printf("\nERROR : Type mismatch of output parameter \"%s\" at line number %d ",actual\_par.node->entity.ivar.word, token->lno);

}

semantic\_status = 0;

}

}

formal\_par = formal\_par->next;

//ignore comma

token = token->n->n;

if(token->prev->s != SQBC)

{

go = TREE;

while(1)

{

actual\_par = findScope(tableID, token->val.s, go->scope,0);

if(actual\_par.code!=-1)

{

break;

}

else

{

if (go->prev != NULL)

{

go = go->prev;

}

else

{

break;

}

}

}

}

else

{

token = token->prev;

flag = 1;

break;

}

}

if(flag == 1 && formal\_par != NULL)

{

if (print == 1)

{

printf("\nERROR : The number of values returned from function \"%s\" are more than the number of accepting at line number %d",res.node->entity.ifunc.word,token->lno);

}

semantic\_status = 0;

}

else if(flag == 0)

{

if (print == 1)

{

printf("\nERROR : The number of values returned from function \"%s\" are less than the number of accepting at line number %d",res.node->entity.ifunc.word,token->lno);

}

semantic\_status = 0;

}

}

}

//check input type of function

flag = 0;

while(token != NULL && token->s != PARAMETERS)

{

token = token->n;

}

//ignore PARAMETERS

token = token->n;

formal\_par = res.node->entity.ifunc.inputList;

scopeChain\* go = TREE;

while(1)

{

actual\_par = findScope(tableID, token->val.s, go->scope,0);

if(actual\_par.code != -1)

{

break;

}

else

{

if (go->prev != NULL)

{

go = go->prev;

}

else

{

break;

}

}

}

int flag = 0;

while(formal\_par != NULL)

{

if(actual\_par.code != -1)

{

if(actual\_par.node->entity.ivar.v\_a == 0)

{

if(formal\_par->v\_a != 0)

{

if (print == 1)

{

printf("\nERROR : Passed parameter \"%s\" at line number %d is a variable, whereas an array type was expected",actual\_par.node->entity.ivar.word, token->lno);

}

semantic\_status = 0;

}

else

{

if(actual\_par.node->entity.ivar.var.v.type != formal\_par->var.v.type)

{

if (print == 1)

{

printf("\nERROR : Type mismatch of input parameter \"%s\" at line number %d.", actual\_par.node->entity.ivar.word, token->lno);

}

semantic\_status = 0;

}

}

}

else

{

if(formal\_par->v\_a != 1)

{

if (print == 1)

{

printf("\nPassed parameter \"%s\" at line number %d is an array, whereas a variable type was expected.",actual\_par.node->entity.ivar.word, token->lno);

}

semantic\_status = 0;

}

else

{

if(actual\_par.node->entity.ivar.var.a.type != formal\_par->var.a.type)

{

if (print == 1)

{

printf("\nERROR : Type mismatch of input parameter \"%s\" at line number %d.",actual\_par.node->entity.ivar.word, token->lno);

}

semantic\_status = 0;

}

if((actual\_par.node->entity.ivar.var.a.s\_idx != formal\_par->var.a.s\_idx) || (actual\_par.node->entity.ivar.var.a.e\_idx != formal\_par->var.a.e\_idx))

{

if (print == 1)

{

printf("\nERROR : Range mismatch of input parameter \"%s\" at line number %d.",actual\_par.node->entity.ivar.word, token->lno);

}

semantic\_status = 0;

}

}

}

}

formal\_par = formal\_par->next;

//ignore comma

token = token->n->n;

if(token->prev != NULL && token->prev->s != SEMICOL)

{

go = TREE;

while(1)

{

actual\_par = findScope(tableID, token->val.s, go->scope, 0);

if(actual\_par.code != -1)

{

break;

}

else

{

if(go->prev != NULL)

{

go=(go->prev);

}

else

{

break;

}

}

}

}

else

{

token = token->prev;

flag = 1;

break;

}

}

if(flag == 1 && formal\_par != NULL)

{

if (print == 1)

{

printf("\nERROR : The number of formal parameters of function \"%s\" are more than that of the actual parameters at line number %d",res.node->entity.ifunc.word,token->lno);

}

semantic\_status = 0;

}

else if(flag == 0)

{

if (print == 1)

{

printf("\nERROR : The number of formal parameters of function \"%s\" are less than that of the actual parameters at line number %d",res.node->entity.ifunc.word,token->lno);

}

semantic\_status = 0;

}

}

}

token = token->n;

}

}

void printVariables(hashTable2 table, totalScopeList \*par)

{

int pos = 0;

int SNO = 1;

while(pos < hash\_capacity\_2)

{

linklist2 \*root = table[pos].start;

IDVariable go;

if (table[pos].flag == occupied)

{

go = table[pos].ID->entity.ivar;

if (go.v\_a == 0)

{

printf("%4d %-10s\t%-20s %-10s %4d to %4d %6d %6d %6d\n", SNO, go.word, getLexeme(go.var.v.type), go.func\_name, par[go.scope].scope\_start, par[go.scope].scope\_end, go.depth, go.bytes, go.offset);

SNO += 1;

}

else

{

int range = (go.var.a.e\_idx - go.var.a.s\_idx + 1);

int bytes = range \* go.bytes;

printf("%4d %-10s\tArray:%3d,%-10s %-10s %4d to %4d %6d %6d %6d\n", SNO, go.word, range, getLexeme(go.var.a.type), go.func\_name, par[go.scope].scope\_start, par[go.scope].scope\_end, go.depth, bytes, go.offset);

SNO += 1;

}

while(root != NULL)

{

go = root->ID->entity.ivar;

if (go.v\_a == 0)

{

printf("%4d %-10s\t%-20s %-10s %4d to %4d %6d %6d %6d\n", SNO, go.word, getLexeme(go.var.v.type), go.func\_name, par[go.scope].scope\_start, par[go.scope].scope\_end, go.depth, go.bytes, go.offset);

SNO += 1;

}

else

{

int range = (go.var.a.e\_idx - go.var.a.s\_idx + 1);

int bytes = range \* go.bytes;

printf("%4d %-10s\tArray:%3d,%-10s %-10s %4d to %4d %6d %6d %6d\n", SNO, go.word, range, getLexeme(go.var.a.type), go.func\_name, par[go.scope].scope\_start, par[go.scope].scope\_end, go.depth, bytes, go.offset);

SNO += 1;

}

root = root->next;

}

}

pos += 1;

}

}

void printFunction(hashTable2 table, totalScopeList\* par)

{

int pos = 0;

while(pos < hash\_capacity\_2)

{

linklist2\* root = table[pos].start;

IDFunction go;

if (table[pos].flag == occupied)

{

go = table[pos].ID->entity.ifunc;

printf("%-15s %10d %10d\n", go.word, par[go.scope].scope\_start, par[go.scope].scope\_end);

while(root != NULL)

{

go = root->ID->entity.ifunc;

printf("%-15s %10d %10d\n", go.word, par[go.scope].scope\_start, par[go.scope].scope\_end);

root = root->next;

}

}

pos += 1;

}

}

**symbolTable.h**

/\*

\* COMPILER PROJECT- ERPLAG COMPILER

\* CSE - D 2019

\* Sahil Garg, Jenit Jain, Amrit Goyal

\*/

#ifndef symbolTable

#define symbolTable

#include "symbolDef.h"

//List of functions for Hashing Scoped variables/functions

hashTable2 initializeHashTable2(int l);

int hashValue2(char \*key);

store2 findScope(hashTable2 T, char \*searchKey, int scope, int option);

void addScope(hashTable2 T, IDEntry\* M, int option);

//List of functions for Symbol table

void createIDTable(tokenInfo\* token, totalScopeList\* scope\_tot, hashTable2 tableID, hashTable2 tableFunc, int print);

void secondRun(tokenInfo\* token, hashTable2 tableID, hashTable2 tableFunc, int print);

int checkModuleReturn(IDEntry\* ctrl\_var, tokenInfo\* token, hashTable2 tableID);

void checkFor(IDEntry\* ctrl\_var, tokenInfo\* token, hashTable2 tableID, int print);

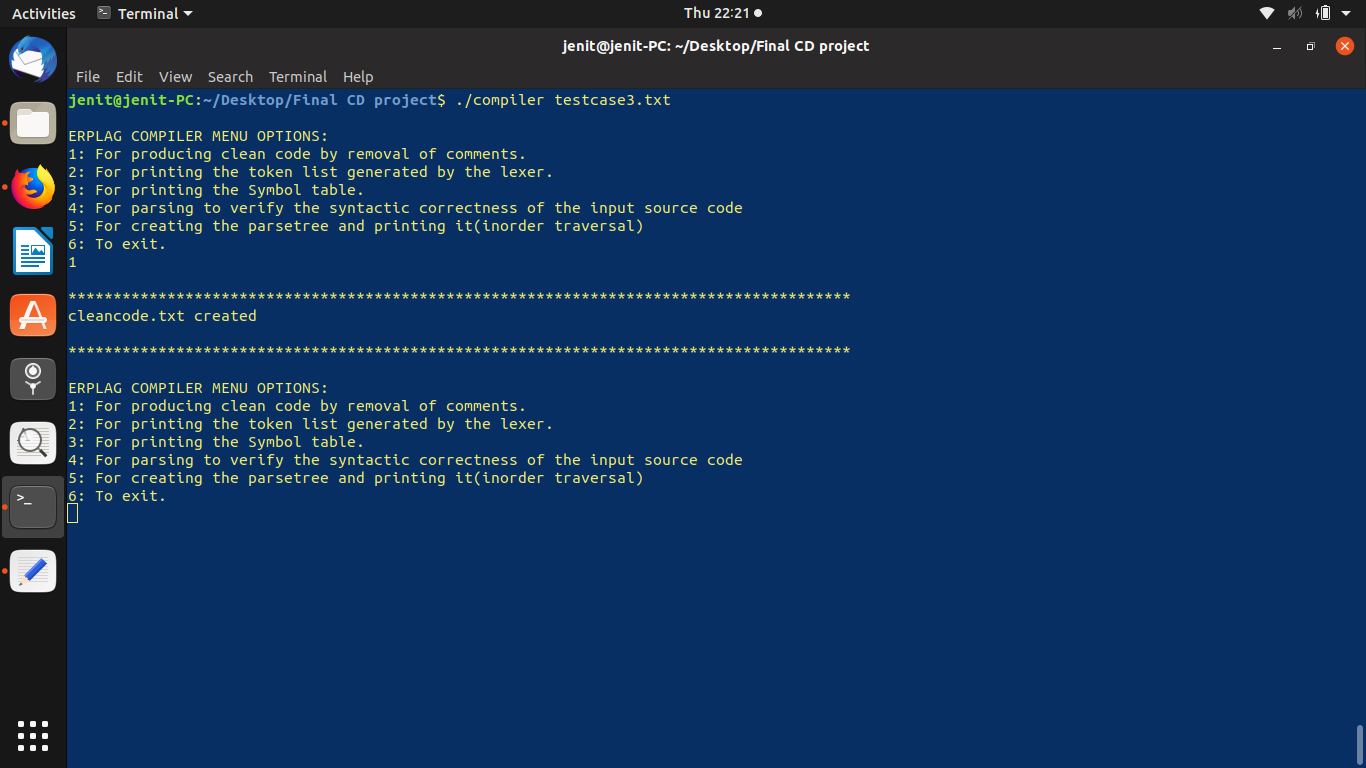
void printVariables(hashTable2 table, totalScopeList \*par);

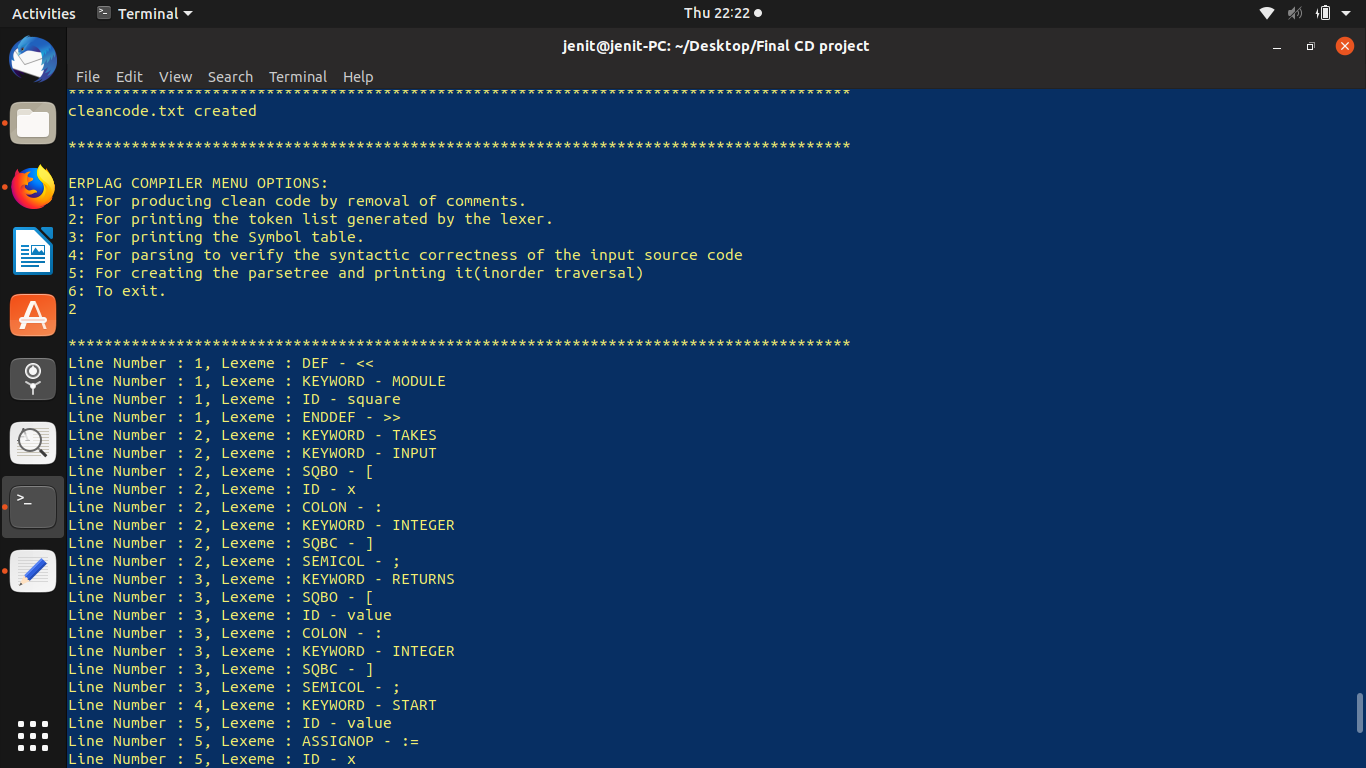
void printFunction(hashTable2 table, totalScopeList\* par);

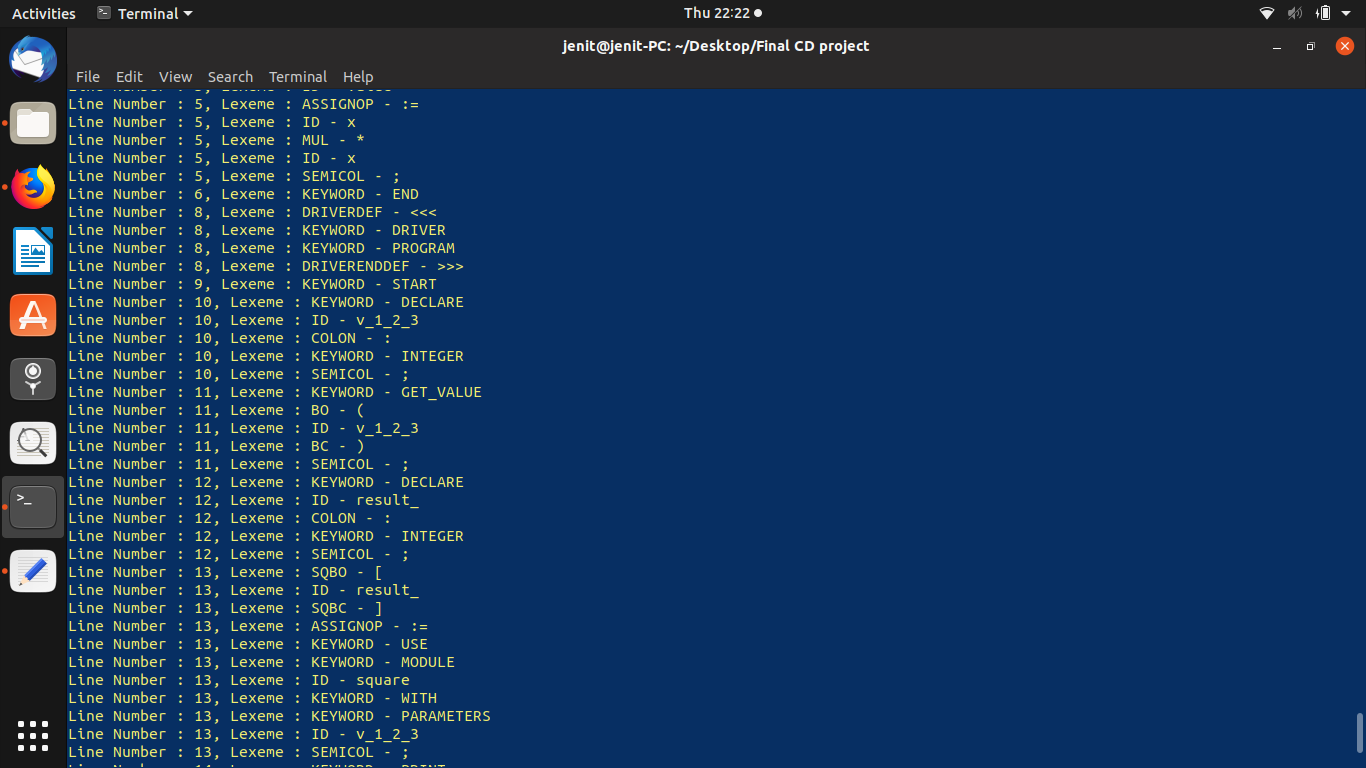
int getBytes(term s);

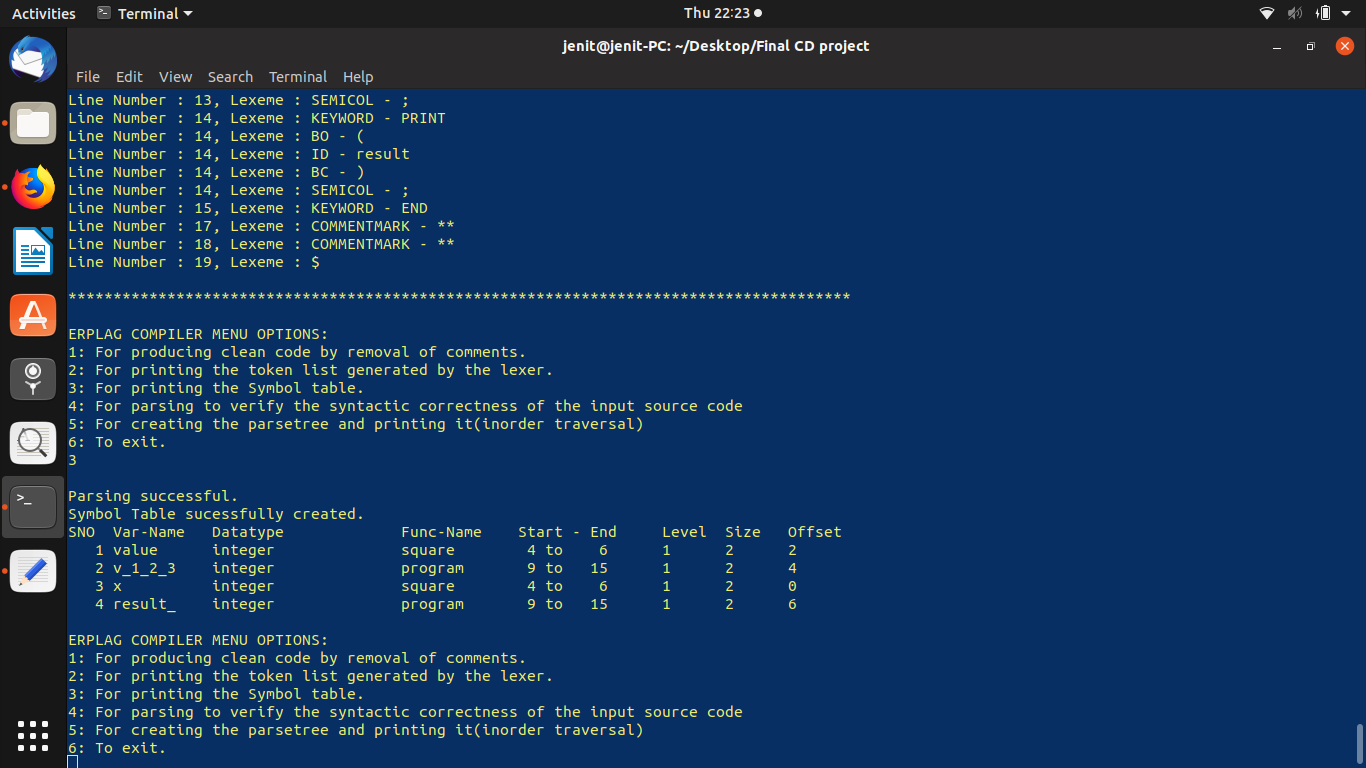
#endif

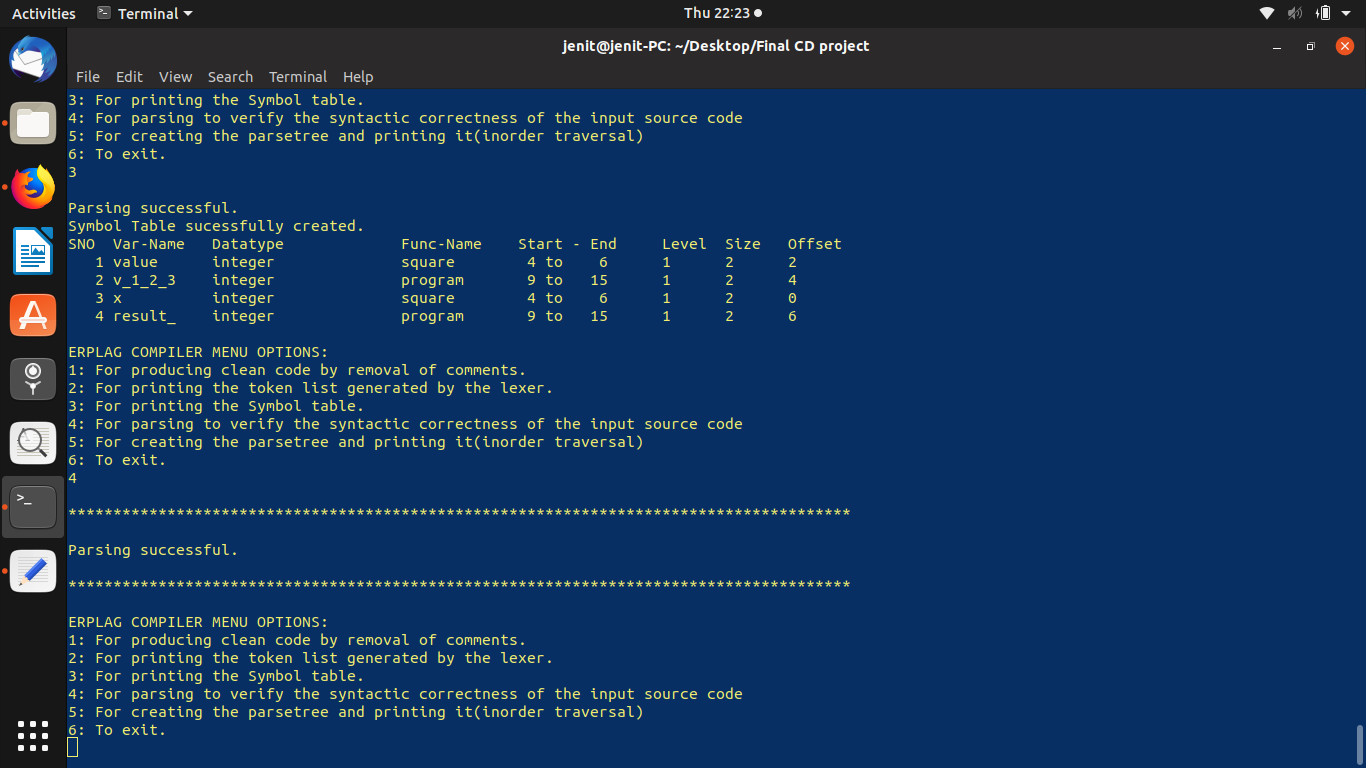
**Snapshot of sample input and output:**

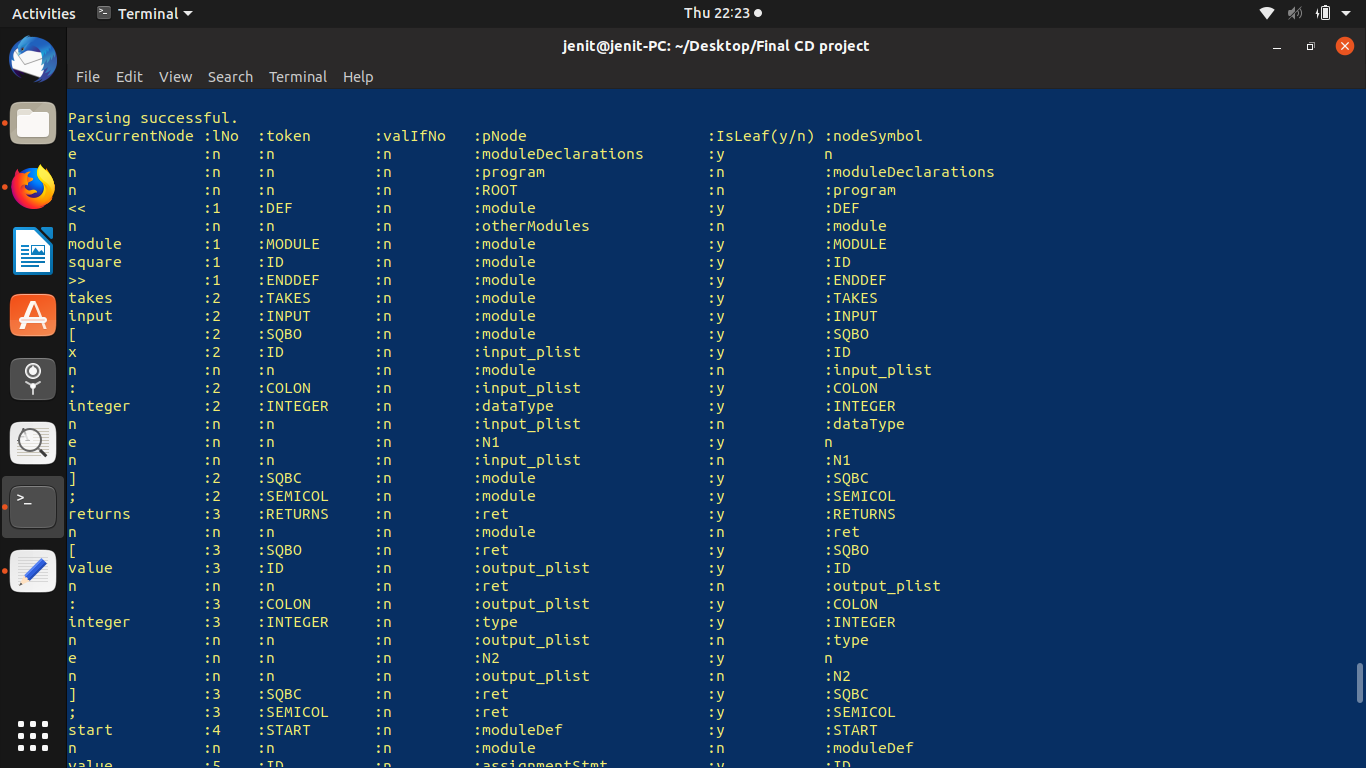


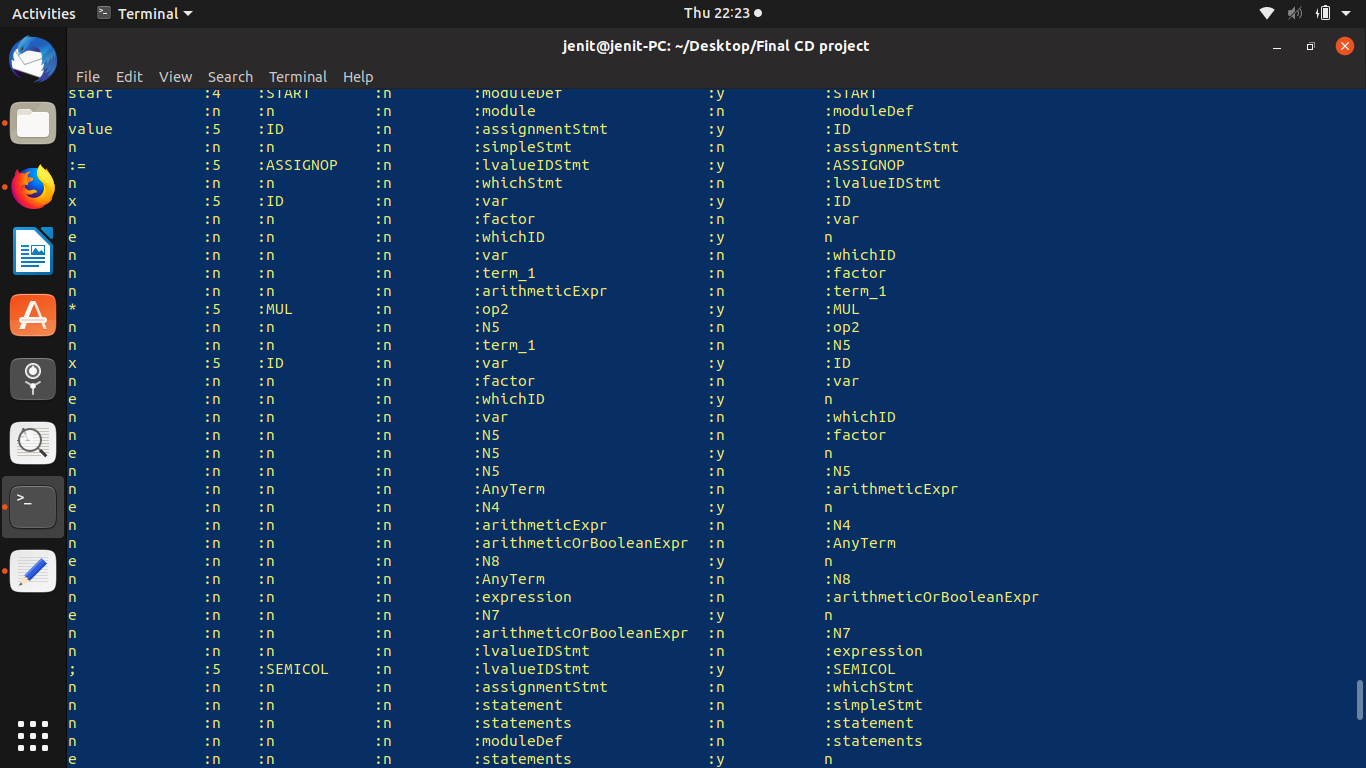


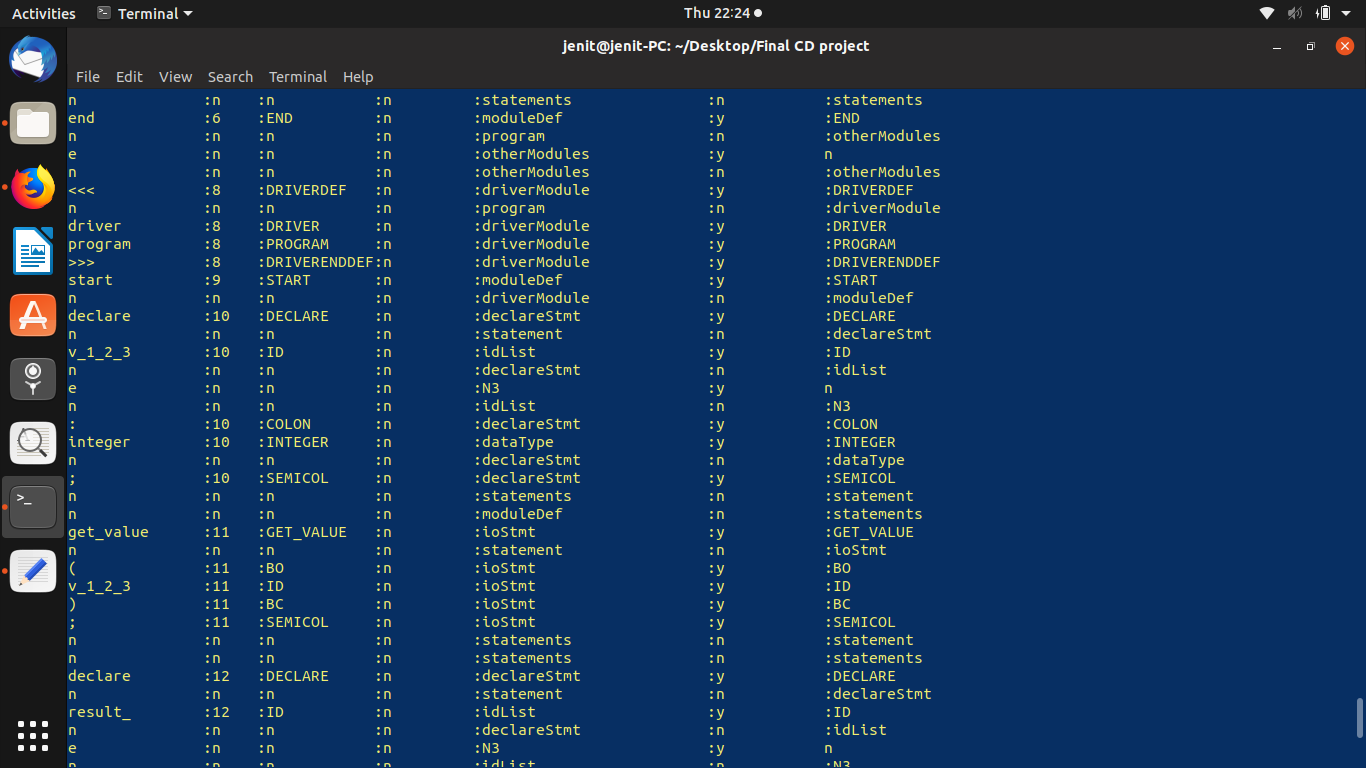


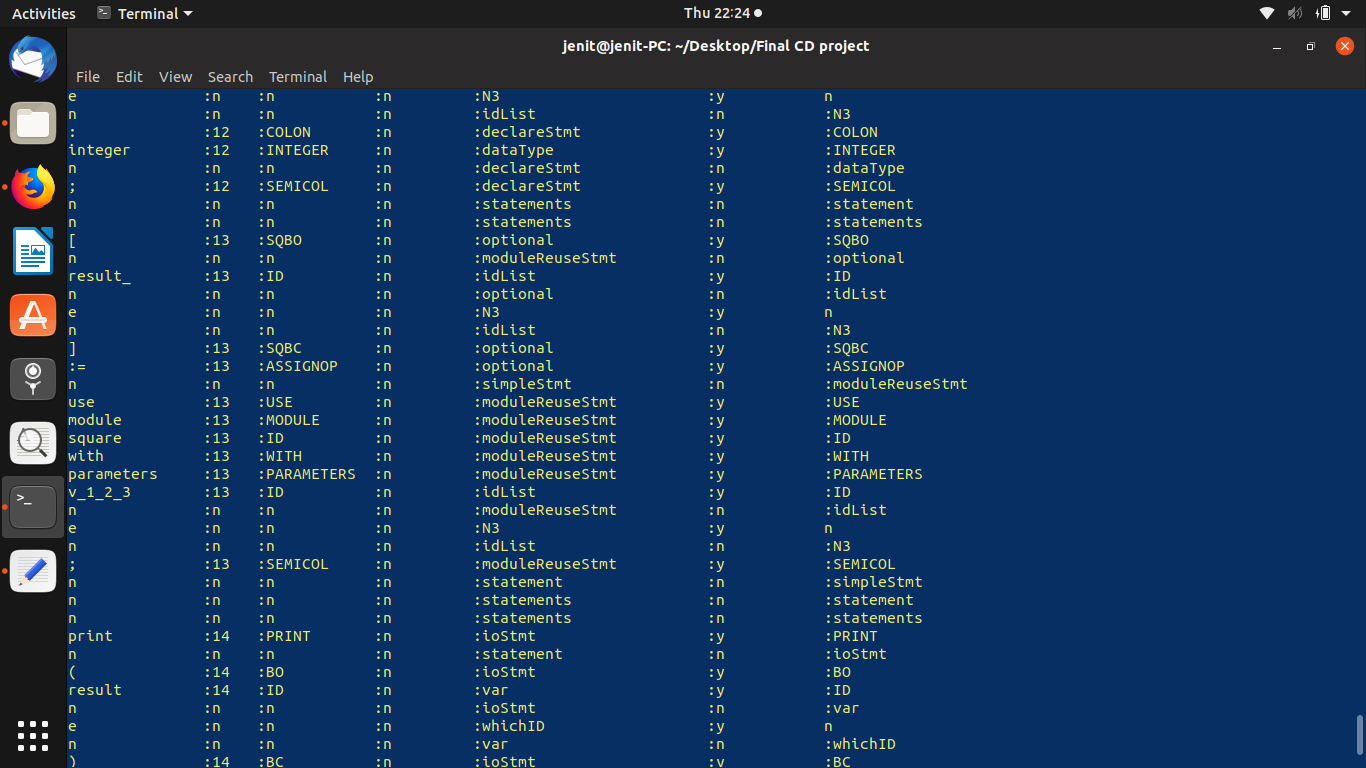


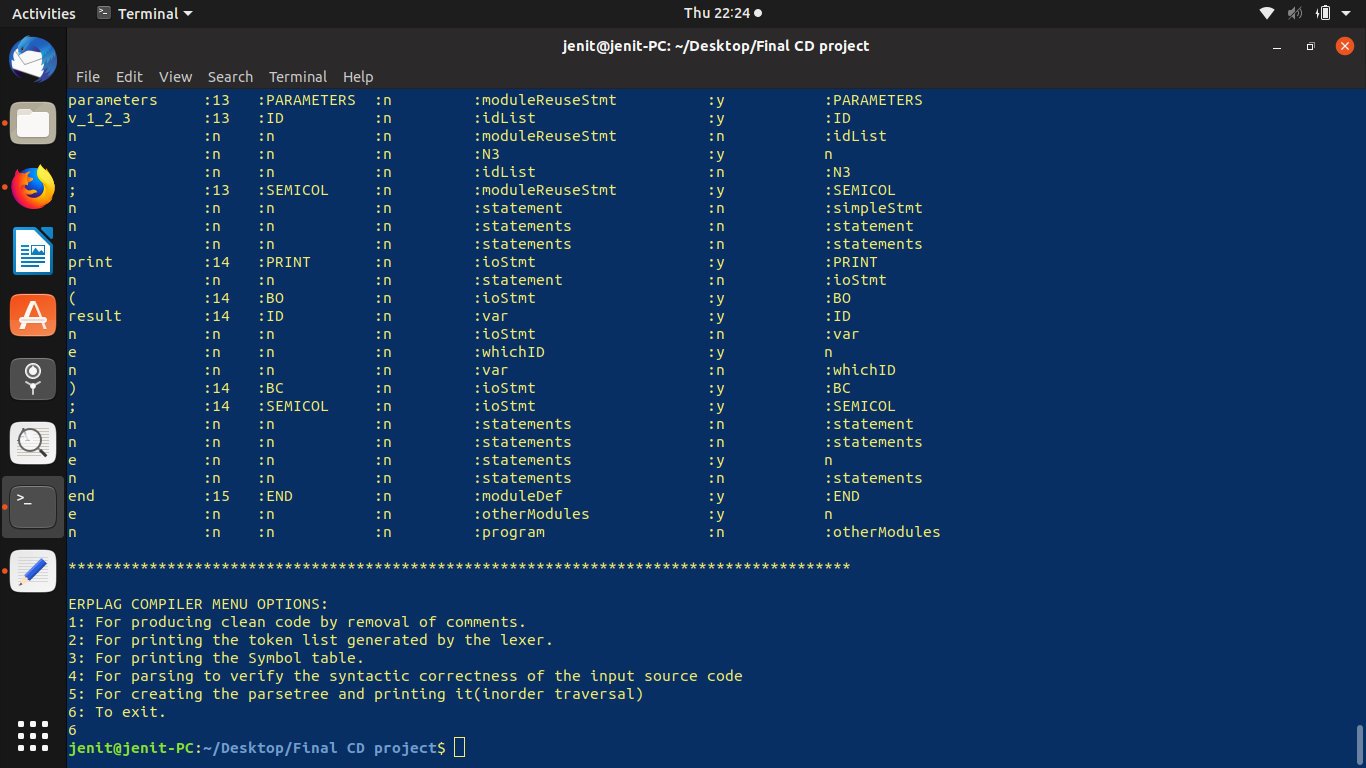












***Input file : Testcase3.txt***

<<module square>>

takes input [x:integer];

returns [value:integer];

start

value:= x\*x;

end

<<<driver program>>>

start

declare v\_1\_2\_3:integer;

get\_value(v\_1\_2\_3);

declare result\_:integer;

[result\_]:=use module square with parameters v\_1\_2\_3;

print(result);

end

\*\*notice that this test case does not declare the module before the driver module, still it is syntactically correct

as moduleDeclarations is optional. Testing whether a module is declared before usage is a matter of semantic analysis\*\*

- Done By : Sahil Garg, Amrit Goyal, Jenit Jain