ASSIGNMENT

Course Code	19CSC305A
Course Name	Compilers
Programme	B. Tech
Department	Computer Science and Engineering
Faculty	FET

Name of the Student	Subhendu Maji
Reg. No	18ETCS002121
Semester/Year	05th /2018
Course Leader/s	Ms. Suvidha

<table>
<tr><th colspan="4">Declaration Sheet</th></tr>
<tr><td>Student Name</td><td colspan="3">Subhendu Maji</td></tr>
<tr><td>Reg. No</td><td colspan="3">18ETCS002121</td></tr>
<tr><td>Programme</td><td>B. Tech</td><td>Semester/Year</td><td>05th /2018</td></tr>
<tr><td>Course Code</td><td colspan="3">19CSC305A</td></tr>
<tr><td>Course Title</td><td colspan="3">Compilers</td></tr>
<tr><td>Course Date</td><td></td><td>to</td><td></td></tr>
<tr><td>Course Leader</td><td colspan="3">Ms. Suvidha</td></tr>
<tr><td colspan="4">Declaration
The assignment submitted herewith is a result of my own investigations and that I have conformed to the guidelines against plagiarism as laid out in the Student Handbook. All sections of the text and results, which have been obtained from other sources, are fully referenced. I understand that cheating and plagiarism constitute a breach of University regulations and will be dealt with accordingly.</td></tr>
<tr><td>Signature of the Student</td><td></td><td>Date</td><td></td></tr>
<tr><td>Submission date stamp
(by Examination & Assessment Section)</td><td colspan="3"></td></tr>
<tr><td colspan="2">Signature of the Course Leader and date</td><td colspan="2">Signature of the Reviewer and date</td></tr>
<tr><td colspan="2"></td><td colspan="2"></td></tr>
</table>

Contents

<table>
<tr><th colspan="6">Assignment</th></tr>
<tr><td colspan="2">Register No.</td><td>18ETCS002121</td><td>Name of the Student</td><td colspan="2">Subhendu Maji</td></tr>
<tr><td rowspan="2">Sections</td><td rowspan="2"></td><td rowspan="2">Marking Scheme</td><td colspan="3">Marks</td></tr>
<tr><td>Max Marks</td><td>First Examiner Marks</td><td>Moderator</td></tr>
<tr><td rowspan="7">Part A 1</td><td colspan="5"></td></tr>
<tr><td>A 1.1</td><td>Identification and grouping of Tokens</td><td>05</td><td></td><td></td></tr>
<tr><td>A 1.2</td><td>Implementation in Lex</td><td>03</td><td></td><td></td></tr>
<tr><td>A 1.3</td><td>Design of Context Free Grammar</td><td>05</td><td></td><td></td></tr>
<tr><td>A 1.4</td><td>Implementation in Yacc</td><td>07</td><td></td><td></td></tr>
<tr><td>A 1.5</td><td>Results and Comments</td><td>05</td><td></td><td></td></tr>
<tr><td></td><td>Part-A 1 Max Marks</td><td>25</td><td></td><td></td></tr>
<tr><td></td><td colspan="2">Total Assignment Marks</td><td>25</td><td></td><td></td></tr>
</table>

<table>
<tr><th colspan="5">Course Marks Tabulation</th></tr>
<tr><td>Component- CET B Assignment</td><td>First Examiner</td><td>Remarks</td><td>Second Examiner</td><td>Remarks</td></tr>
<tr><td>A.1</td><td></td><td></td><td></td><td></td></tr>
<tr><td>Marks (out of 25)</td><td></td><td></td><td></td><td></td></tr>
<tr><td colspan="5">Signature of First Examiner   Signature of Moderator</td></tr>
</table>

Part A. Question No. 1

Solution to Part A Question No. 1:

A1.1 Identification and grouping of Tokens

The indivisible unit of a program is called a token. A compiler breaks the program into the smallest possible units and proceeds to the various stages of the compilation. Such units are tokens. As seen here, tokens are the units in which compilers act on. Typically, in a language like C, there are the following tokens: Identifiers, Keywords, Constants, Strings, Operators and Special Symbols.

In the compiler designed for this assignment, the number of tokens is larger, as it makes designing the context free grammar easier. The tokens that are used in the implemented compiler are:

Token	**Description**
Keyword Tokens	
`INT`	Token returned when the keyword defining the *integer* datatype is matched.
`LONG`	Token returned when the keyword defining the *long* datatype is matched.
`LONG_LONG`	Token returned when the keyword defining the *long long* datatype is matched.
`SHORT`	Token returned when the keyword defining the *short* datatype is matched.
`SIGNED`	Token returned when the keyword defining the *signed* datatype is matched.
`UNSIGNED`	Token returned when the keyword defining the *unsigned* datatype is matched.
`BREAK`	Token returned when the keyword defining the *break* keyword is matched.
`CONTINUE`	Token returned when the keyword defining the *continue* keyword is matched.
`RETURN`	Token returned when the keyword defining the *return* keyword is matched.
`WHILE`	Token returned when the keyword defining the *while* loop is matched.
`FOR`	Token returned when the keyword defining the *for* loop is matched.
`IF`	Token returned when the keyword defining the *if* statement is matched
`ELSE`	Token returned when the keyword defining the *else* statement is matched.
Identifier Tokens	
`IDENTIFIER`	Token returned when an identifier (variable name) is matched.
Constant and String Tokens	
`digit`	Token returned when an integer number is matched.
`hex`	Token returned when a hex number is matched.
`letter`	Token returned when a string literal is matched.

DECREMENT	Token returned when -- is matched
INCREMENT	Token returned when ++ is matched
ADD_ASSIGN	Token returned when += is matched
SUB_ASSIGN	Token returned when -= is matched
MUL_ASSIGN	Token returned when *- is matched
DIV_ASSIGN	Token returned when /- is matched
MOD_ASSIGN	Token returned when %- is matched
LOGICAL_AND	Token returned when && is matched
LOGICAL_OR	Token returned when || is matched
LS_EQ	Token returned when <= is matched
GR_EQ	Token returned when >= is matched
EQ	Token returned when == is matched
NOT_EQ	Token returned when != is matched

A1.2 Implementation in Lex

Structure of Lex Program

The structure of the lex program consists of three sections:

```
{definition section}
%%
{rules section}
%%
{C code section}
```

The definition section defines macros and imports header files written in C. It is also possible to write any C code here, which will be copied verbatim into the generated source file.

The rules section associates regular expression patterns with C statements. When the lexer sees text in the input matching a given pattern, it will execute the associated C code.

The C code section contains C statements and functions that are copied verbatim to generated source file. These statements presumably contain code called by the rules in the rules section.

For Source Code refer https://github.com/subhendu17620/C-Compiler/blob/main/Syntax%20Analysis/lexl.l

A1.3 Design of Context Free Grammar

In formal language theory, a context-free grammar (CFG) is a formal grammar in which every production rule is of the form:

$$A \rightarrow \alpha$$

Where,

A is a single nonterminal symbol,

α is a string of terminals and/or non-terminals (α can be empty)

A formal grammar is considered "context free" when its production rules can be applied regardless of the context of a nonterminal. No matter which symbols surround it, the single nonterminal on the left-hand side can always be replaced by the right-hand side. This is what distinguishes it from a context-sensitive grammar.

The context free grammar designed for this compiler can be thought of having three major segments: Definition of a function, definition of a statement and definition of a expression.

Definition of a function

Every function definition has the following parts: a return type, function name, parameter list, and curly brackets that signify the start and end of the function. Thus, the context free grammar production rules that express this are:

$$\boldsymbol{program} \rightarrow \ function\_declaration$$

$$\boldsymbol{function\_declaration}$$

$$\rightarrow \ type\ IDENTIFIER\ OPEN\_BRAC\ parameter\_list\ CLOSE\_BRAC\ compound\_statement$$

$$\boldsymbol{parameter\_list} \ \rightarrow \ type\ IDENTIFIER$$

$$|\ type\ IDENTIFIER\ COMMA\ parameter\_list$$

$$|\ \lambda$$

Definition of a statement

A statement in a program can consist of many things, such as a declaration of a variable, start of a loop, start of a conditional switch, and they can even be nested within one another. Hence the production rules that express this are:

```
%union
{
    double dval;
    entry_t* entry;
    int ival;
}

%token <entry> IDENTIFIER

 /* Constants */
%token <dval> DEC_CONSTANT HEX_CONSTANT
%token STRING

 /* Logical and Relational operators */
%token LOGICAL_AND LOGICAL_OR LS_EQ GR_EQ EQ NOT_EQ

 /* Short hand assignment operators */
%token MUL_ASSIGN DIV_ASSIGN MOD_ASSIGN ADD_ASSIGN SUB_ASSIGN
%token LEFT_ASSIGN RIGHT_ASSIGN AND_ASSIGN XOR_ASSIGN OR_ASSIGN
```

```
%token INCREMENT DECREMENT

 /* Data types */
%token SHORT INT LONG LONG_LONG SIGNED UNSIGNED CONST

 /* Keywords */
%token IF FOR WHILE CONTINUE BREAK RETURN

%type <dval> expression
%type <dval> sub_expr
%type <dval> constant
%type <dval> unary_expr
%type <dval> arithmetic_expr
%type <dval> assignment_expr
%type <entry> lhs
%type <ival> assign_op

%start starter

%left ','
%right '='
%left LOGICAL_OR
%left LOGICAL_AND
%left EQ NOT_EQ
%left '<' '>' LS_EQ GR_EQ
%left '+' '-'
%left '*' '/' '%'
%right '!'


%nonassoc UMINUS
%nonassoc LOWER_THAN_ELSE
%nonassoc ELSE


%%

 /* Program is made up of multiple builder blocks. */
starter: starter builder
             |builder;

 /* Each builder block is either a function or a declaration */
builder: function|
       declaration;

 /* This is how a function looks like */
function: type IDENTIFIER '(' argument_list ')' compound_stmt;

 /* Now we will define a grammar for how types can be specified */

type :data_type pointer
    |data_type;

pointer: '*' pointer
    |'*'
    ;

data_type :sign_specifier type_specifier
    |type_specifier
    ;

sign_specifier :SIGNED
    |UNSIGNED
    ;

type_specifier :INT                    {current_dtype = INT;}
    |SHORT INT                         {current_dtype = SHORT;}
    |SHORT                             {current_dtype = SHORT;}
```

```
     |LONG                              {current_dtype = LONG;}
     |LONG INT                          {current_dtype = LONG;}
     |LONG_LONG                         {current_dtype = LONG_LONG;}
     |LONG_LONG INT                     {current_dtype = LONG_LONG;}
     ;

 /* grammar rules for argument list */
 /* argument list can be empty */
argument_list :arguments
     |
     ;
 /* arguments are comma separated TYPE ID pairs */
arguments :arguments ',' arg
     |arg
     ;

 /* Each arg is a TYPE ID pair */
arg :type IDENTIFIER
    ;

 /* Generic statement. Can be compound or a single statement */
stmt:compound_stmt
     |single_stmt
     ;

 /* The function body is covered in braces and has multiple statements. */
compound_stmt :'{' statements '}'
     ;

statements:statements stmt
     |
     ;

 /* Grammar for what constitutes every individual statement */
single_stmt :if_block
     |for_block
     |while_block
     |declaration
     |function_call ';'
     |RETURN ';'
     |CONTINUE ';'
     |BREAK ';'
     |RETURN sub_expr ';'
     ;

for_block:FOR '(' expression_stmt  expression_stmt ')' stmt
     |FOR '(' expression_stmt expression_stmt expression ')' stmt
     ;

if_block:IF '(' expression ')' stmt %prec LOWER_THAN_ELSE
                |IF '(' expression ')' stmt ELSE stmt
     ;

while_block: WHILE '(' expression   ')' stmt
         ;

declaration:type declaration_list ';'
              |declaration_list ';'
              | unary_expr ';'

declaration_list: declaration_list ',' sub_decl
        |sub_decl;

sub_decl: assignment_expr
     |IDENTIFIER                      {$1 -> data_type = current_dtype;}
     |array_index
    /*|struct_block ';'*/
     ;
```

```
/* This is because we can have empty expession statements inside for loops */
expression_stmt:expression ';'
    |';'
    ;

expression:
    expression ',' sub_expr                             {$$ = $1,$3;}
    |sub_expr                                           {$$ = $1;}
        ;

sub_expr:
    sub_expr '>' sub_expr                       {$$ = ($1 > $3);}
    |sub_expr '<' sub_expr                      {$$ = ($1 < $3);}
    |sub_expr EQ sub_expr                       {$$ = ($1 == $3);}
    |sub_expr NOT_EQ sub_expr                   {$$ = ($1 != $3);}
    |sub_expr LS_EQ sub_expr                    {$$ = ($1 <= $3);}
    |sub_expr GR_EQ sub_expr                    {$$ = ($1 >= $3);}
    |sub_expr LOGICAL_AND sub_expr              {$$ = ($1 && $3);}
    |sub_expr LOGICAL_OR sub_expr               {$$ = ($1 || $3);}
    |'!' sub_expr                               {$$ = (!$2);}
    |arithmetic_expr                            {$$ = $1;}
    |assignment_expr                            {$$ = $1;}
    |unary_expr                                 {$$ = $1;}
    /* |IDENTIFIER                                     {$$ = $1->value;}
    |constant                                   {$$ = $1;} */
        //|array_index
    ;


assignment_expr :lhs assign_op arithmetic_expr     {$$ = $1->value = Evaluate($1->value,$2,$3);}
    |lhs assign_op array_index                     {$$ = 0;}
    |lhs assign_op function_call                   {$$ = 0;}
    |lhs assign_op unary_expr                      {$$ = $1->value = Evaluate($1->value,$2,$3);}
    |unary_expr assign_op unary_expr               {$$ = 0;}
    ;

unary_expr: lhs INCREMENT                          {$$ = $1->value = ($1->value)++;}
    |lhs DECREMENT                                 {$$ = $1->value = ($1->value)--;}
    |DECREMENT lhs                                 {$$ = $2->value = --($2->value);}
    |INCREMENT lhs                                 {$$ = $2->value = ++($2->value);}

lhs:IDENTIFIER                                     {$$ = $1; if(! $1->data_type) $1-
>data_type = current_dtype;}
    //|array_index
    ;

assign_op:'='                                      {$$ = '=';}
    |ADD_ASSIGN                                    {$$ = ADD_ASSIGN;}
    |SUB_ASSIGN                                    {$$ = SUB_ASSIGN;}
    |MUL_ASSIGN                                    {$$ = MUL_ASSIGN;}
    |DIV_ASSIGN                                    {$$ = DIV_ASSIGN;}
    |MOD_ASSIGN                                    {$$ = MOD_ASSIGN;}
    ;

arithmetic_expr: arithmetic_expr '+' arithmetic_expr    {$$ = $1 + $3;}
    |arithmetic_expr '-' arithmetic_expr                {$$ = $1 - $3;}
    |arithmetic_expr '*' arithmetic_expr                {$$ = $1 * $3;}
    |arithmetic_expr '/' arithmetic_expr                {$$ = ($3 == 0) ? yyerror("Divide by 0!")
 : ($1 / $3);}
    |arithmetic_expr '%' arithmetic_expr                {$$ = (int)$1 % (int)$3;}
    |'(' arithmetic_expr ')'                            {$$ = $2;}
    |'-' arithmetic_expr %prec UMINUS                   {$$ = -$2;}
    |IDENTIFIER                                         {$$ = $1 -> value;}
    |constant                                           {$$ = $1;}
    ;

constant: DEC_CONSTANT                                  {$$ = $1;}
```

```
    |HEX_CONSTANT                                          {$$ = $1;}
    ;

array_index: IDENTIFIER '[' sub_expr ']'

function_call: IDENTIFIER '(' parameter_list ')'
             |IDENTIFIER '(' ')'
             ;

parameter_list:
              parameter_list ','  parameter
              |parameter
              ;

parameter: sub_expr
                |STRING

        ;
%%
```

A1.4 Implementation in Yacc

Structure of Yacc Program

The parser written is known as the Yacc program. The structure of the Yacc file is similar to that of the lexer, consisting of three sections:

```
{declarations}
%%
{rules}
%%
{routines}
```

The declarations section of a yacc file may consist of the following:

- %token - identifies the token names that the yacc file accepts
- %start - identifies a nonterminal name for the start symbol
- %right - identifies tokens that are right-associative with other tokens
- %left - identifies tokens that are left-associative with other tokens
- %nonassoc - identifies tokens that are not associative with other tokens

The rules section consists of the context free grammar used to generate the parse tree. A general rule has the following structure:

```
nonterminal
      : sentential form
      | sentential form
      ................
      | sentential form
      ;
```

Actions may be associated with rules and are executed when the associated sentential form is matched.

The routines section may include the C program that specifies the input file, action routines and other user defined functions.

The entire code for syntax analysis and semantics analysis is broken down into 3 files: *lexer.l, Parser.y* and *symboltable.h*

File	File Contents
lexer.l	A lex file containing the lex specification of regular expressions
Parser.y	A parser file containing the grammars.
symboltable.h	Contains the definition of the symbol table and the constants table and also defines functions for inserting into the hash table and displaying its contents.

Syntax Analysis

syntax analysis is the process of checking that the code is syntactically correct. The purpose of syntax analysis or parsing is to check that we have a valid sequence of tokens. Tokens are valid sequence of symbols, keywords, identifiers etc. The parser needs to be able to handle the infinite number of possible valid programs that may be presented to it. The usual way to define the language is to specify a grammar.

A grammar is a set of rules (or productions) that specifies the syntax of the language (i.e. what is a valid sentence in the language). There can be more than one grammar for a given language. The parser analyzes the source code (token stream) against the production rules to detect any errors in the code. The output of this phase is a parse tree.

The syntax analyser for the C language by writing two scripts, one that acts as a lexical analyzer (lexer) and outputs a stream of tokens, and the other one that acts as a parser.

The lexer is known as the lex program. Lex reads an input stream specifying the lexical analyzer and outputs source code implementing the lexer in the C programming language.

Lex specification and regular expression are defined in *lexl.l*

Definition of the symbol table and the constants table and also defines functions for inserting into the hash table and displaying its contents is done in *symboltable.h*

YACC implementation is done in *parser.y*

For source code with comment refer
https://github.com/subhendu17620/C-Compiler/tree/main/Syntax%20Analysis

Semantics Analysis

Semantic analysis is the task of ensuring that the declarations and statements of a program are Semantically correct, i.e. that their meaning is clear and consistent with the way in which control structures and data types are supposed to be used.

Semantic analysis can compare information in one part of a parse tree to that in another part (e.g compare reference to variable agrees with its declaration, or that parameters to a function call match the function definition).

Implementing the semantic actions is conceptually simpler in recursive descent parsing because they are simply added to the recursive procedures. Some of the functions of Semantic analysis are that it maintains and updates the symbol table, check source programs for semantic errors and warnings like type mismatch, global and local scope of a variable, re-definition of variables, usage of undeclared variables.

Lex specification and regular expression are defined in *lexer.l*

Definition of the symbol table and the constants table and also defines functions for inserting into the hash table and displaying its contents is done in *symboltable.h*

YACC implementation is done in *parser.y*

For source code with comments refer
https://github.com/subhendu17620/C-Compiler/tree/main/Semantics%20Analysis

A1.5 Results and Comments

Execution of Syntax analyzer

Syntax Analysis > compile

```bash
#!/bin/bash

lex lexl.l
yacc -d parser.y -v
gcc -w -g y.tab.c -ll -o parser


```

Figure 1 Bash file to run semantics analyzer

Test Case 1:

```
Syntax Analysis > test1.c > ...
1
2   int main()
3   {
4       int x, y;
5       long long int total, diff;
6       int *ptr;
7       int a = 86;
8
9       if (a > 90)
10      {
11          printf("Grade is AA");
12      }
13      else if (a > 80)
14      {
15          printf("Grade is AB")
16      }
17      else
18      {
19          printf("Grade is BB");
20      }
21
22      x = -10, y = 20;
23      x = x * 3 / 2;
24      total = x + y;
25  }
26
```

Figure 2 Test file 1 which contains a semicolon error at line 15

```
~/projects/compiler/Compiler Assignment/Syntax Analysis
❯ chmod +x compile

~/projects/compiler/Compiler Assignment/Syntax Analysis
❯ ./compile

~/projects/compiler/Compiler Assignment/Syntax Analysis
❯./parser test1.c
Line no: 16 Error message: syntax error Token: }

Parsing failed

        Symbol table
==================================================
 lexeme              value               data-type
==================================================
 total               2147483647          283
 printf              2147483647          0
 x                   2147483647          281
 a                   86                  281
 main                2147483647          0
 y                   2147483647          281
 ptr                 2147483647          281
 diff                2147483647          283
==================================================

~/projects/compiler/Compiler Assignment/Syntax Analysis
❯ 
```

Figure 3 execution with test test file 1

Test Case 2:

```
Syntax Analysis > test2.c > main()
int main()
{
    int c = 0, d, e, f;

    c = c + 1; // c = 1
    d = c * 5; // d = 5
    e = d / 4; // e = 1
    f = d % 3; // f = 2
    f = f / 0;
}

```

Figure 4 Test file 2 with divide by zero error

As we can see in Fig. 5 the arithmetic results of lexeme's in the table. The modulus (%) operator is giving wrong answer which can be seen as limitation of this program.

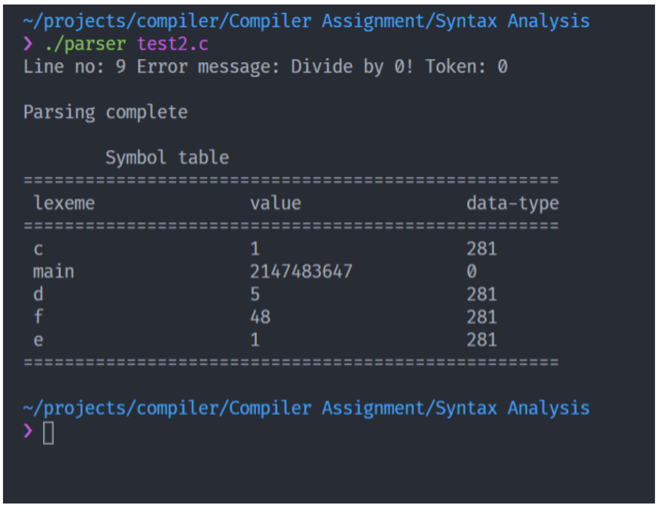

Figure 5 Execution of test file 2

Test Case 3:

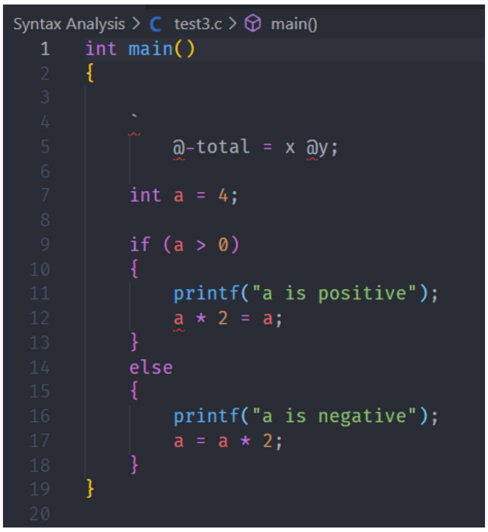

Figure 6 Test Case file 3

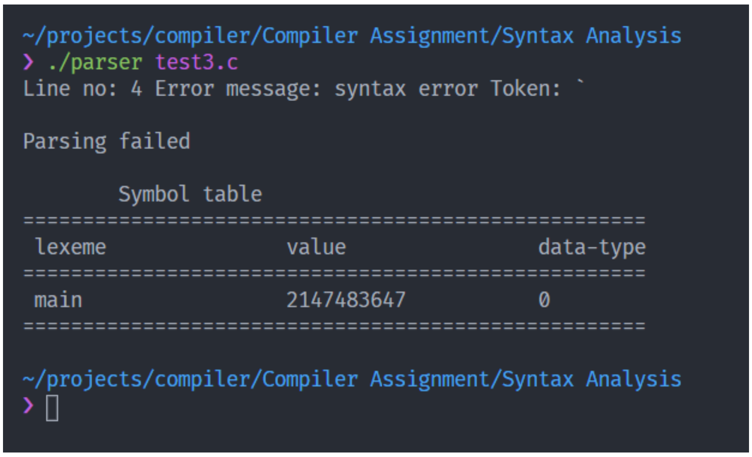

Figure 7 Execution of test file 3

Execution of Semantics Analyzer

```
Semantics Analysis > compile
#!/bin/bash
lex lexer.l
yacc -d parser.y -v
gcc -w -g y.tab.c -ll -o semantic_analyser
```

Figure 8 bash file for execution of semantics analyser

Test Case 1:

```
Semantics Analysis > test1.c > ...
1   int fun(int a, int b, int c)
2   {
3       return 2;
4   }
5
6   int main()
7   {
8
9       int x = 1;
10
11      fun();
12      return 3;
13  }
```

Figure 9 Test File 1

```
~/projects/compiler/Compiler Assignment/Semantics Analysis main
❯ chmod +x compile

~/projects/compiler/Compiler Assignment/Semantics Analysis main
❯ ./compile

~/projects/compiler/Compiler Assignment/Semantics Analysis main*
❯ ./semantic_analyser test1.c
Line no: 11 Error message: Number of parameters and arguments do not match Token: )

~/projects/compiler/Compiler Assignment/Semantics Analysis main*
❯
```

Figure 10 Execution of test file 1

Test Case 2:

```
Semantics Analysis > test2.c > main()
int main()
{

    int arr2[10];

    int x = arr2[5], y = arr2[9]; //Valid
    int w = arr2[4.5];
    int z = arr2[11];
    return 0;
}

```

Figure 11 Test file 2

```
~/projects/compiler/Compiler Assignment/Semantics Analysis
❯ ./semantic_analyser test2.c
Line no: 8 Error message: Array index out of bound Token: ]

~/projects/compiler/Compiler Assignment/Semantics Analysis
❯ 
```

Figure 12 Execution of test file 2

Test Case 3:

```
Semantics Analysis > test3.c > main()

int main()
{
    int x;
    int y;

    y = 1;
    if (y > 0)
    {
        int x; //This is fine
    }

    if (y == 1)
    {
        int x; // This is also fine
        y = 2; // This too
        int z;
    }
}

```

Figure 13 Test file 3

```
~/projects/compiler/Compiler Assignment/Semantics Analysis
❯ ./semantic_analyser test3.c

PARSING COMPLETE


SYMBOL TABLES

Scope: 0

====================================================================================================
 lexeme               data-type            array_dimension      num_params           param_list

====================================================================================================
 main                 278                  -1                    0

====================================================================================================


Scope: 1

====================================================================================================
 lexeme               data-type            array_dimension      num_params           param_list

====================================================================================================
 x                    278                  -1                    0
 y                    278                  -1                    0

====================================================================================================


Scope: 2

====================================================================================================
 lexeme               data-type            array_dimension      num_params           param_list

====================================================================================================
 x                    278                  -1                    0

====================================================================================================
Scope: 3

====================================================================================================
 lexeme               data-type            array_dimension      num_params           param_list

====================================================================================================
 x                    278                  -1                    0
 z                    278                  -1                    0

====================================================================================================


CONSTANT TABLE
=========================
 lexeme     data-type

=========================
 1          278
 2          278
 0          278

=========================

~/projects/compiler/Compiler Assignment/Semantics Analysis
❯ 
```

Figure 14 Execution of test file 3

Conclusion

The syntax analyzer and the semantic analyzer for a subset of C language, which include selection statements, compound statements, iteration statements (for, while and do-while) and user defined functions is generated. It is important to define unambiguous grammar in the syntax analysis phase. The semantic analyzer performs type checking, reports various errors such as undeclared variable, type mismatch, errors in function call (number and datatypes of parameters mismatch) and errors in array indexing.

Bibliography

1. http://epaperpress.com/lexandyacc/download/LexAndYaccTutorial.pdf - Lex and YACC
2. https://www.programiz.com/c-programming/precedence-associativity-operators - C Precedence and Associativity
3. https://www.geeksforgeeks.org/yacc-program-to-implement-a-calculator-and-recognize-a-valid-arithmetic-expression/

- For source Code of Syntax and semantics analyzer refer https://github.com/subhendu17620/C-Compiler