COMPILER DESIGN LAB

Compiler Design Lab Project

REPORT

Overview

The main objective of this project is to build a own programming language and design a compiler for the language.

A compiler is a program that translates a source code into machine code, byte code, or another programming language. The source code is typically written in high-level languages such as C, C++, or Java. A compiler that supports the source programming language reads the files, analyzes the code, and translates it into a format suitable for the target platform. Some compilers can translate source code into another high-level programming language, rather than machine code or bytecode. This type of compiler might be referred to as a transpiler, transcompiler, or source-to-source translator or it might go by another name. Regardless of the source language or the type of output, a compiler must ensure that the logic of the output code always matches that of the input code and that nothing is lost when converting the code. A compiler is, in the strictest sense, a translator and must ensure that the output is correct and preserves all the original logic.

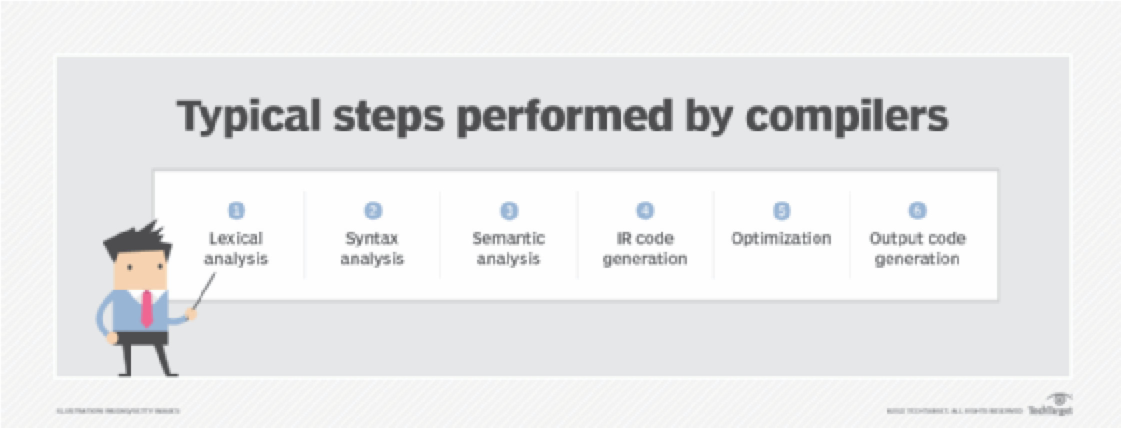
How does a compiler work?

Compilers vary in the methods they use for analyzing and converting source code to output code. Despite their differences, they typically carry out the following steps:

- Lexical analysis. The compiler splits the source code into lexemes, which are individual code fragments that represent specific patterns in the code. The lexemes are then tokenized in preparation for syntax and semantic analyses.
- Syntax analysis. The compiler verifies that the code's syntax is correct, based on the rules for the source language. This process is also referred to as parsing. During this step, the compiler typically creates abstract syntax trees that represent the logical structures of specific code elements.
- Semantic analysis. The compiler verifies the validity of the code's logic. This step goes beyond syntax analysis by validating the code's accuracy. For example, the

semantic analysis might check whether variables have been assigned the right types or have been properly declared.

- IR code generation. After the code passes through all three analysis phases, the compiler generates an intermediate representation (IR) of the source code. The IR code makes it easier to translate the source code into a different format. However, it must accurately represent the source code in every respect, without omitting any functionality.
- Optimization. The compiler optimizes the IR code in preparation for the final code generation. The type and extent of optimization depend on the compiler. Some compilers let users configure the degree of optimization.
- Output code generation. The compiler generates the final output code, using the optimized IR code.



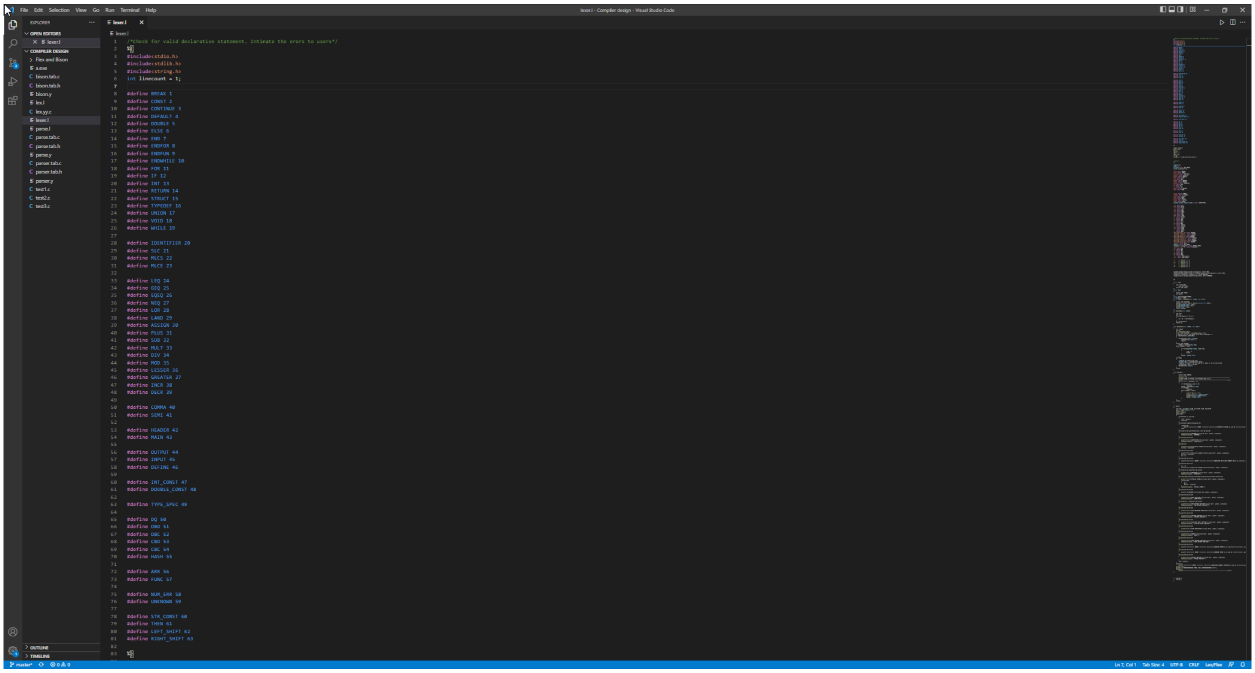
Tools Used

The programs are written in the lex and yacc program.

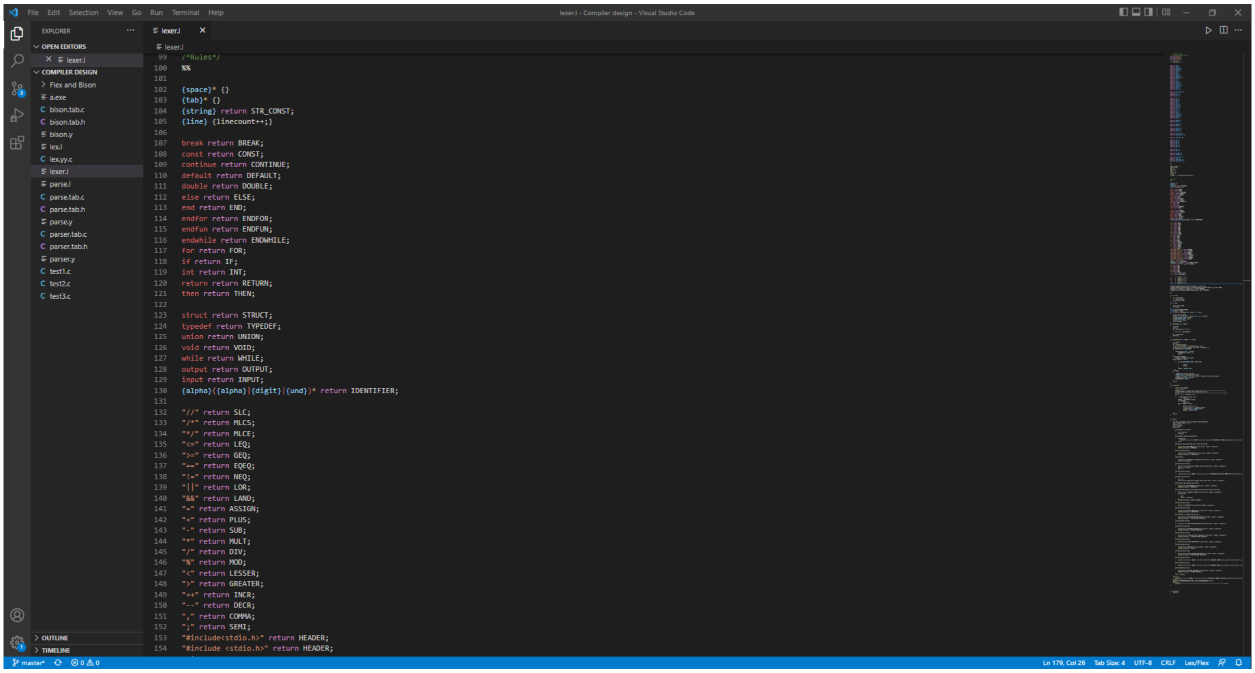
Lex helps you by taking a set of description of possible tokens and producing a C routine, which we call as lexical analyzer or a lexer, that identify the tokens. As input is divided into tokens, a program often needs to establish the relationship among the tokens. A compiler needs to find the expression, statement, declaration, blocks, and procedure in the program. This task is known as parsing and the list of rules that define the relationship that the program is a grammar. Yacc takes a concise description of grammar and produces a C routine that can parse the grammar, a parser. The yacc parser automatically detects whenever a sequence of input tokens matches one of the rules of the grammar and also detects a syntax error whenever its input does not match any of the rules. When a task involves dividing the input into units and establishing some relationship among those units then we should use lex and yacc.

Lex Program

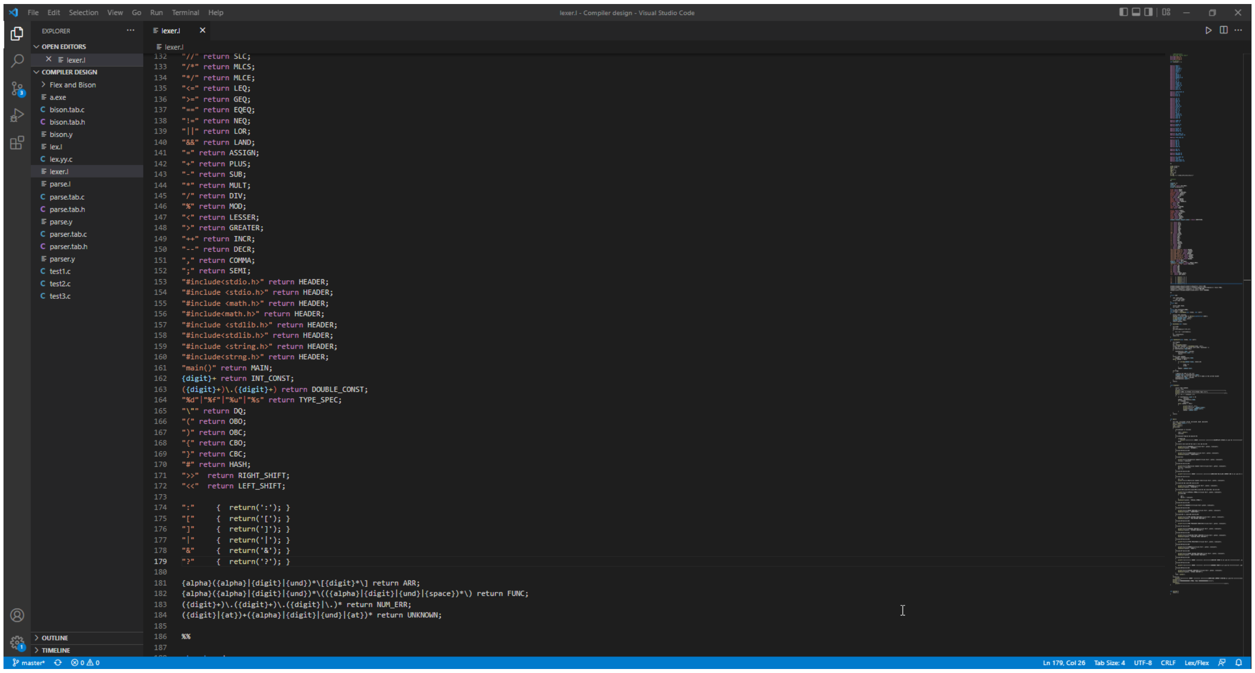
Lex is a tool for building a lexical analyzer or lexer. A lexer takes an arbitrary input stream and tokenizes it. In this program, when we create a lex specification, we create a set of patterns that lex matches against the input. Lex program has three sections one is the definition section another is the rule section and the last one is the user subroutine section which in this case is not written here but in the yacc program. In the definition section we have defined the header files and also y.tab.h which helps it to link up with the yacc program. The rules section contains the patterns that have to be matched and followed by an action part, which returns the tokens to the yacc program when the input is parsed and the pattern is matched.


File Edit Selection View Go Run Terminal Help
lexer.l - Compiler design - Visual Studio Code
EXPLORER
OPEN EDITORS
COMPILER DESIGN
lexer.l
OUTLINE
TIMELINE

```
/*Check for valid declarative statement. Intimate the erors to users*/
%{
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
int linecount = 1;

#define BREAK 1
#define CONST 2
#define CONTINUE 3
#define DEFAULT 4
#define DOUBLE 5
#define ELSE 6
#define END 7
#define ENDFOR 8
#define ENDFUN 9
#define ENDWHILE 10
#define FOR 11
#define IF 12
#define INT 13
#define RETURN 14
#define STRUCT 15
#define TYPEDEF 16
#define UNION 17
#define VOID 18
#define WHILE 19

#define IDENTIFIER 20
#define SLC 21
#define MLCS 22
#define MLCE 23

#define LEQ 24
#define GEQ 25
#define EQEQ 26
#define NEQ 27
#define LOR 28
#define LAND 29
#define ASSIGN 30
#define PLUS 31
#define SUB 32
#define MULT 33
#define DIV 34
#define MOD 35
#define LESSER 36
#define GREATER 37
#define INCR 38
#define DECR 39

#define COMMA 40
#define SEMI 41

#define HEADER 42
#define MAIN 43

#define OUTPUT 44
#define INPUT 45
#define DEFINE 46

#define INT_CONST 47
#define DOUBLE_CONST 48

#define TYPE_SPEC 49

#define DQ 50
#define OBO 51
#define OBC 52
#define CBO 53
#define CBC 54
#define HASH 55

#define ARR 56
#define FUNC 57

#define NUM_ERR 58
#define UNKNOWN 59

#define STR_CONST 60
#define THEN 61
#define LEFT_SHIFT 62
#define RIGHT_SHIFT 63

%}
```

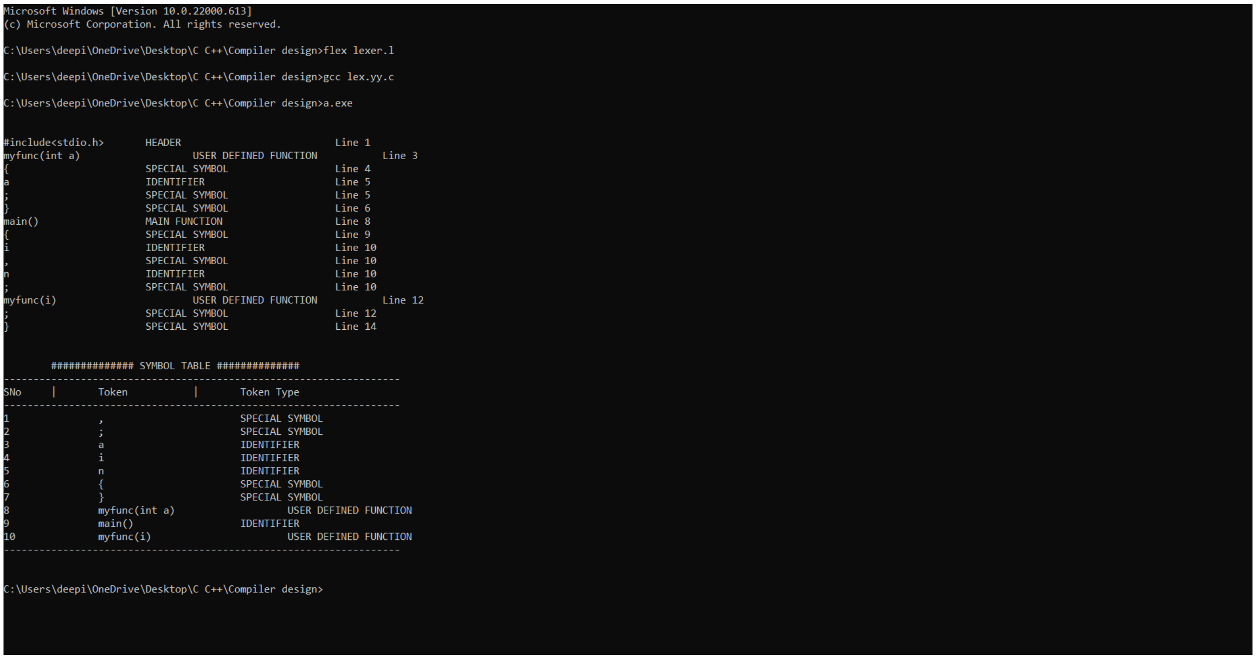


```
/*Rules*/
%%

{space}* {}
{tab}* {}
{string} return STR_CONST;
{line} {linecount++;}

break return BREAK;
const return CONST;
continue return CONTINUE;
default return DEFAULT;
double return DOUBLE;
else return ELSE;
end return END;
endfor return ENDFOR;
endfun return ENDFUN;
endwhile return ENDWHILE;
for return FOR;
if return IF;
int return INT;
return return RETURN;
then return THEN;

struct return STRUCT;
typedef return TYPEDEF;
union return UNION;
void return VOID;
while return WHILE;
output return OUTPUT;
input return INPUT;
{alpha}({alpha}|{digit}|{und})* return IDENTIFIER;

"//" return SLC;
"/*" return MLCS;
"*/" return MLCE;
"<=" return LEQ;
">=" return GEQ;
"==" return EQEQ;
"!=" return NEQ;
"||" return LOR;
"&&" return LAND;
"=" return ASSIGN;
"+" return PLUS;
"-" return SUB;
"*" return MULT;
"/" return DIV;
"%" return MOD;
"<" return LESSER;
">" return GREATER;
"++" return INCR;
"--" return DECR;
"," return COMMA;
";" return SEMI;
"#include<stdio.h>" return HEADER;
"#include <stdio.h>" return HEADER;
```



```
"//" return SLC;
"/*" return MLCS;
"*/" return MLCE;
"<=" return LEQ;
">=" return GEQ;
"==" return EQEQ;
"!=" return NEQ;
"||" return LOR;
"&&" return LAND;
"=" return ASSIGN;
"+" return PLUS;
"-" return SUB;
"*" return MULT;
"/" return DIV;
"%" return MOD;
"<" return LESSER;
">" return GREATER;
"++" return INCR;
"--" return DECR;
"," return COMMA;
";" return SEMI;
"#include<stdio.h>" return HEADER;
"#include <stdio.h>" return HEADER;
```

```
"#include <math.h>" return HEADER;
"#include<math.h>" return HEADER;
"#include <stdlib.h>" return HEADER;
"#include<stdlib.h>" return HEADER;
"#include <string.h>" return HEADER;
"#include<strng.h>" return HEADER;
"main()" return MAIN;
{digit}+ return INT_CONST;
({digit}+)\.({digit}+) return DOUBLE_CONST;
"%d"|"%f"|"%u"|"%s" return TYPE_SPEC;
"\"" return DQ;
"(" return OBO;
")" return OBC;
"{" return CBO;
"}" return CBC;
"#" return HASH;
">>"  return RIGHT_SHIFT;
"<<"  return LEFT_SHIFT;

":"     {  return(':'); }
"["     {  return('['); }
"]"     {  return(']'); }
"|"     {  return('|'); }
"&"     {  return('&'); }
"?"     {  return('?'); }

{alpha}({alpha}|{digit}|{und})*\[{digit}*\] return ARR;
{alpha}({alpha}|{digit}|{und})*\(({alpha}|{digit}|{und}|{space})*\) return FUNC;
({digit}+)\.({digit}+)\.({digit}|\.)* return NUM_ERR;
({digit}|{at})+({alpha}|{digit}|{und}|{at})* return UNKNOWN;

%%
```

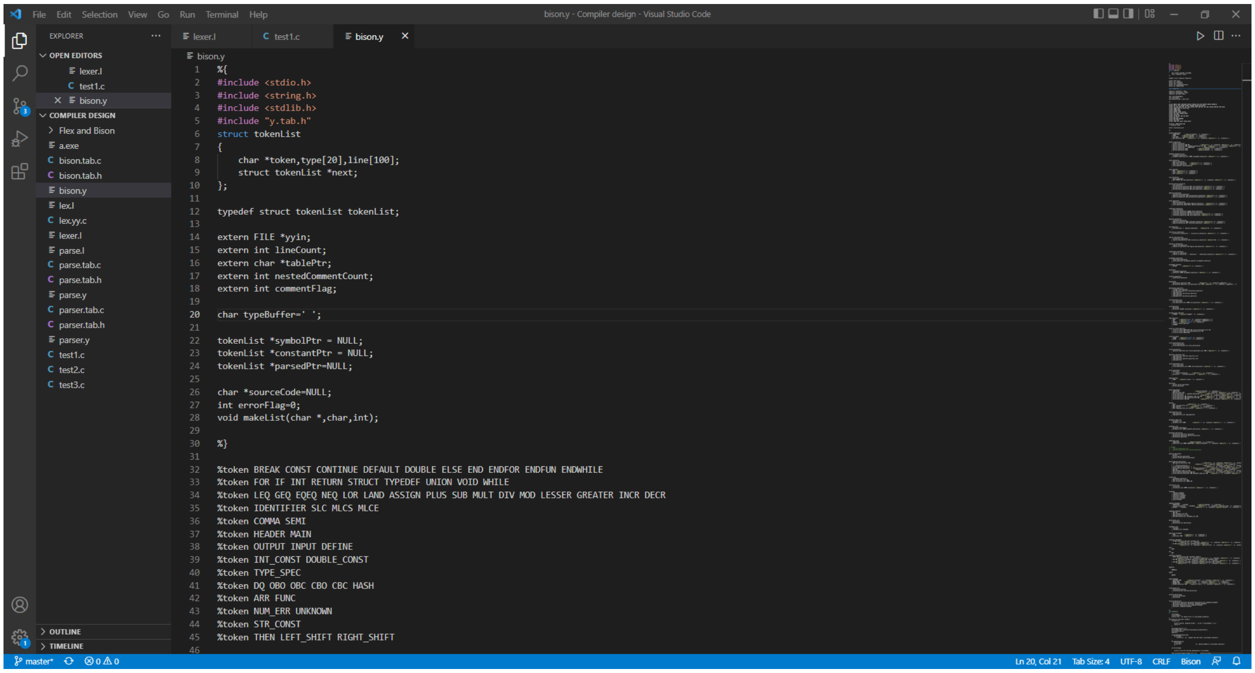


```
Microsoft Windows [Version 10.0.22000.613]
(c) Microsoft Corporation. All rights reserved.

C:\Users\deepi\OneDrive\Desktop\C C++\Compiler design>flex lexer.l

C:\Users\deepi\OneDrive\Desktop\C C++\Compiler design>gcc lex.yy.c

C:\Users\deepi\OneDrive\Desktop\C C++\Compiler design>a.exe


#include<stdio.h>       HEADER                          Line 1
myfunc(int a)                   USER DEFINED FUNCTION           Line 3
{                       SPECIAL SYMBOL                  Line 4
a                       IDENTIFIER                      Line 5
;                       SPECIAL SYMBOL                  Line 5
}                       SPECIAL SYMBOL                  Line 6
main()                  MAIN FUNCTION                   Line 8
{                       SPECIAL SYMBOL                  Line 9
i                       IDENTIFIER                      Line 10
,                       SPECIAL SYMBOL                  Line 10
n                       IDENTIFIER                      Line 10
;                       SPECIAL SYMBOL                  Line 10
myfunc(i)                       USER DEFINED FUNCTION           Line 12
;                       SPECIAL SYMBOL                  Line 12
}                       SPECIAL SYMBOL                  Line 14


        ############## SYMBOL TABLE ##############
-------------------------------------------------------------------
SNo     |       Token           |       Token Type
-------------------------------------------------------------------
1               ,                       SPECIAL SYMBOL
2               ;                       SPECIAL SYMBOL
3               a                       IDENTIFIER
4               i                       IDENTIFIER
5               n                       IDENTIFIER
6               {                       SPECIAL SYMBOL
7               }                       SPECIAL SYMBOL
8               myfunc(int a)                   USER DEFINED FUNCTION
9               main()                  IDENTIFIER
10              myfunc(i)                       USER DEFINED FUNCTION
-------------------------------------------------------------------


C:\Users\deepi\OneDrive\Desktop\C C++\Compiler design>
```

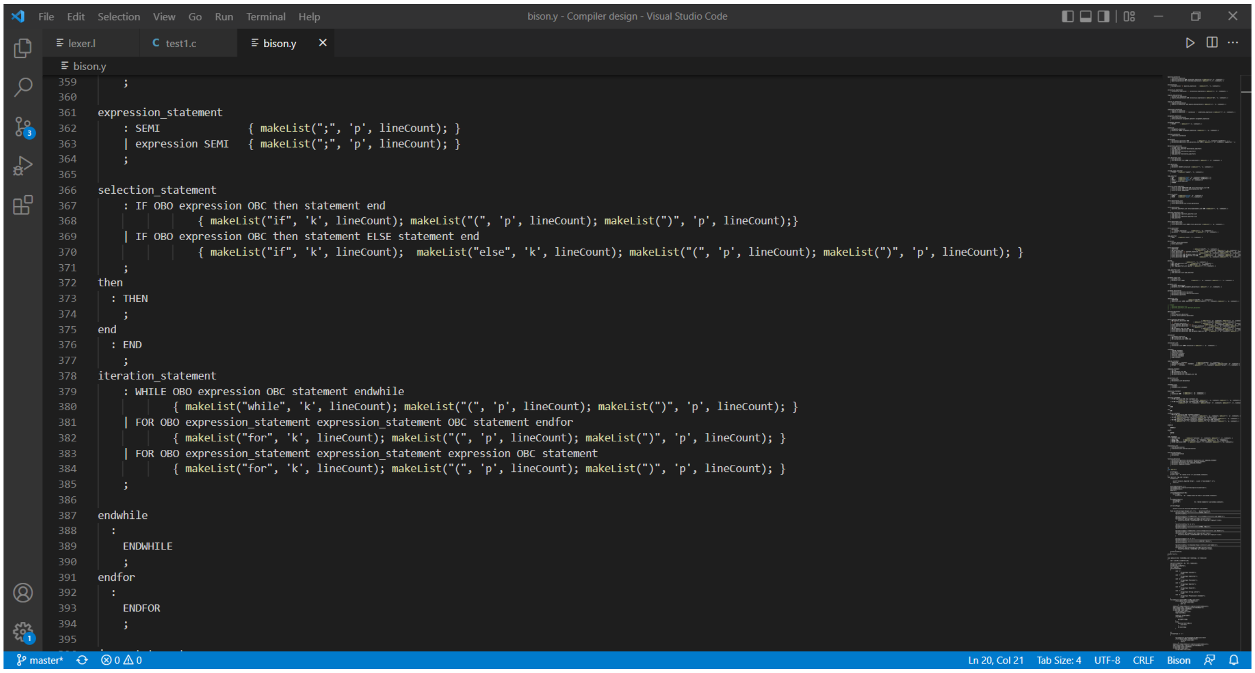
Yacc program

Yacc takes a grammar that we specify and writes a parser that recognizes valid syntax in that grammar. Grammar is a series of rules that the parser uses to recognize syntactically valid input. Again in the same way as lex specification, a yacc grammar has a three-part structure. The first section is the definition section, which handles control information for the yacc generated by the parser. The second section contains the rule of parser and the third section contains the C code. In the following program, the definition section has the header files that will generate the necessary files needed to run the C program next comes the declaration of the tokens which are passed from the lex program. Then we define the grammar for the valid input typed by the user. Then it contains the necessary SDTs (Syntax Directed Translations) which move to a specific function when that particular input is encountered by the parser.

Syntax phase, semantics phase, and intermediate code generation(3 address code) is done in parser file.



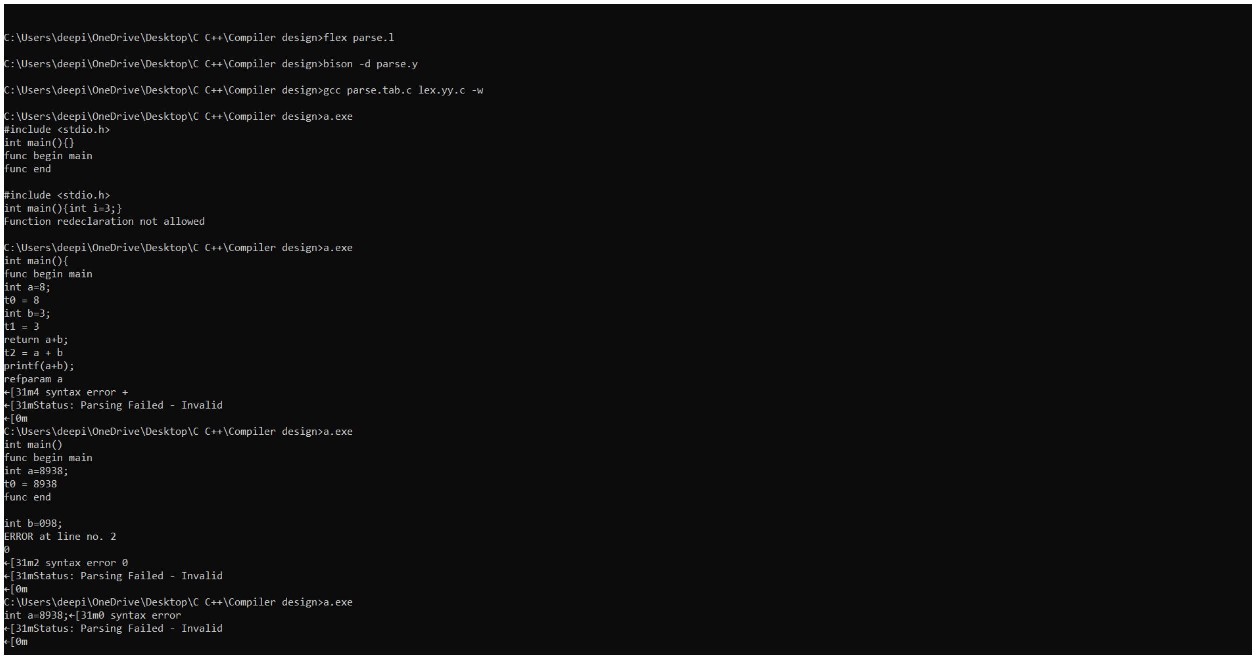
```
%{
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "y.tab.h"
struct tokenList
{
    char *token,type[20],line[100];
    struct tokenList *next;
};

typedef struct tokenList tokenList;

extern FILE *yyin;
extern int lineCount;
extern char *tablePtr;
extern int nestedCommentCount;
extern int commentFlag;

char typeBuffer=' ';

tokenList *symbolPtr = NULL;
tokenList *constantPtr = NULL;
tokenList *parsedPtr=NULL;

char *sourceCode=NULL;
int errorFlag=0;
void makeList(char *,char,int);

%}

%token BREAK CONST CONTINUE DEFAULT DOUBLE ELSE END ENDFOR ENDFUN ENDWHILE
%token FOR IF INT RETURN STRUCT TYPEDEF UNION VOID WHILE
%token LEQ GEQ EQEQ NEQ LOR LAND ASSIGN PLUS SUB MULT DIV MOD LESSER GREATER INCR DECR
%token IDENTIFIER SLC MLCS MLCE
%token COMMA SEMI
%token HEADER MAIN
%token OUTPUT INPUT DEFINE
%token INT_CONST DOUBLE_CONST
%token TYPE_SPEC
%token DQ OBO OBC CBO CBC HASH
%token ARR FUNC
%token NUM_ERR UNKNOWN
%token STR_CONST
%token THEN LEFT_SHIFT RIGHT_SHIFT

```



bison.y - Compiler design - Visual Studio Code
File Edit Selection View Go Run Terminal Help
lexer.l
test1.c
bison.y
master*
Ln 20, Col 21
Tab Size: 4
UTF-8
CRLF
Bison

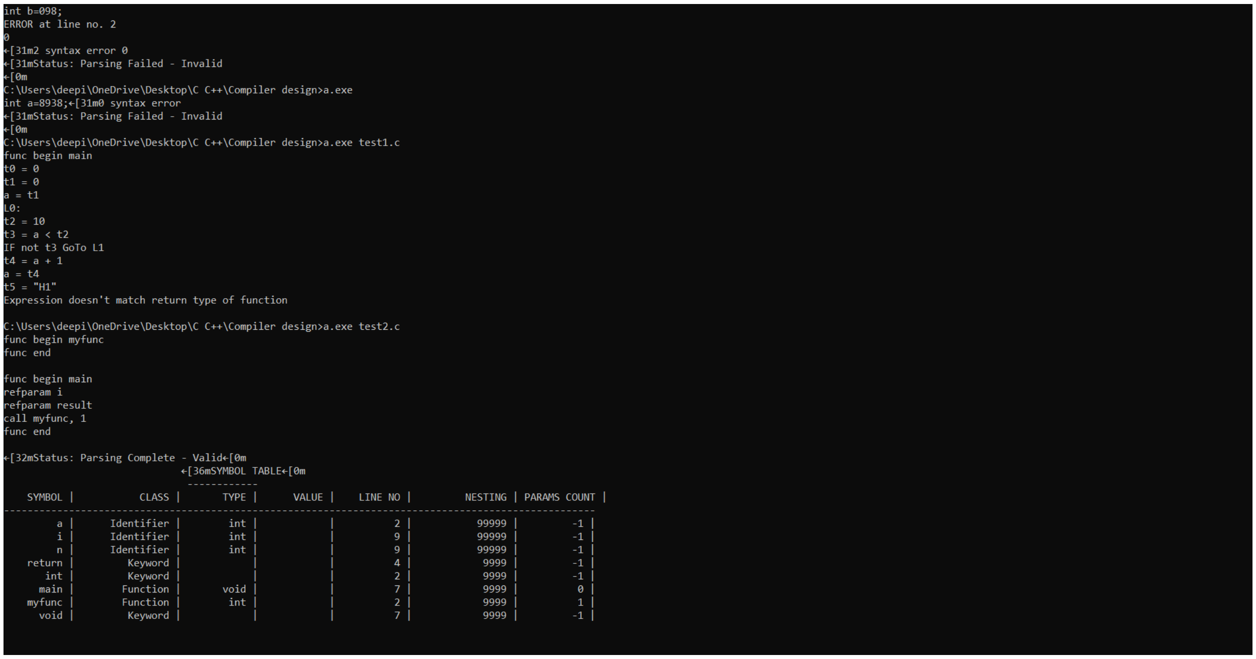
```
    ;

expression_statement
    : SEMI              { makeList(";", 'p', lineCount); }
    | expression SEMI   { makeList(";", 'p', lineCount); }
    ;

selection_statement
    : IF OBO expression OBC then statement end
                { makeList("if", 'k', lineCount); makeList("(", 'p', lineCount); makeList(")", 'p', lineCount);}
    | IF OBO expression OBC then statement ELSE statement end
                { makeList("if", 'k', lineCount);  makeList("else", 'k', lineCount); makeList("(", 'p', lineCount); makeList(")", 'p', lineCount); }
    ;
then
  : THEN
    ;
end
  : END
    ;
iteration_statement
    : WHILE OBO expression OBC statement endwhile
            { makeList("while", 'k', lineCount); makeList("(", 'p', lineCount); makeList(")", 'p', lineCount); }
    | FOR OBO expression_statement expression_statement OBC statement endfor
            { makeList("for", 'k', lineCount); makeList("(", 'p', lineCount); makeList(")", 'p', lineCount); }
    | FOR OBO expression_statement expression_statement expression OBC statement
            { makeList("for", 'k', lineCount); makeList("(", 'p', lineCount); makeList(")", 'p', lineCount); }
    ;

endwhile
  :
    ENDWHILE
    ;
endfor
  :
    ENDFOR
    ;

```

Testing the parser file by given inputs in command line:

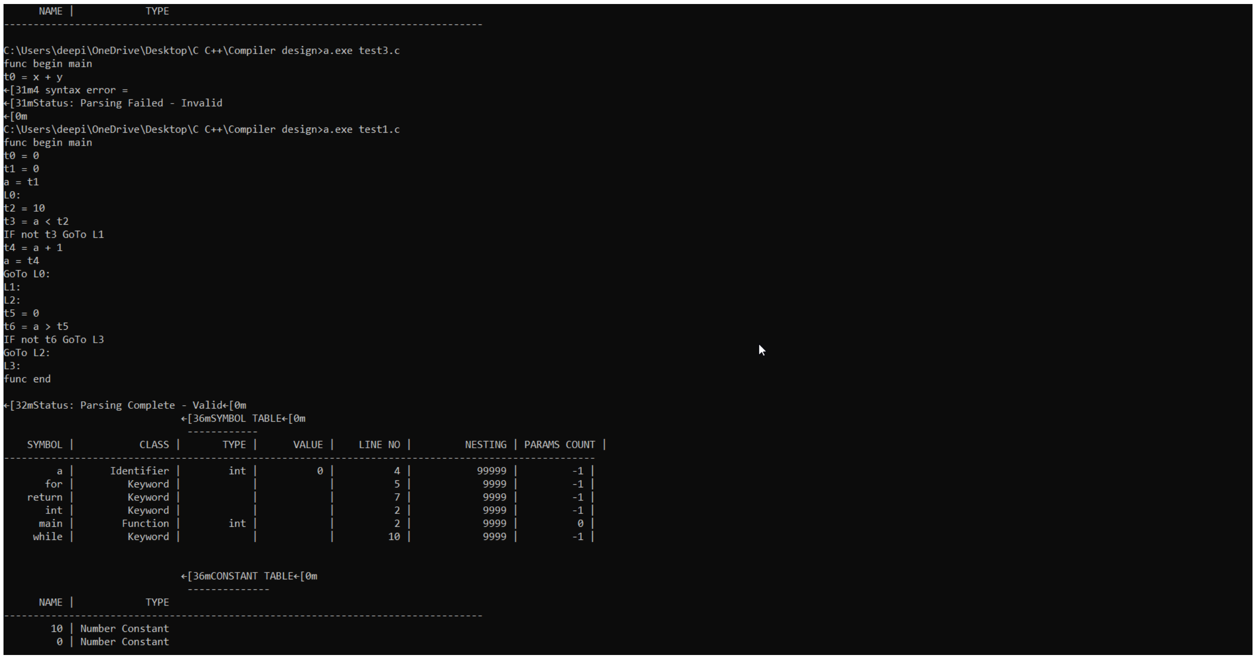


```
C:\Users\deepi\OneDrive\Desktop\C C++\Compiler design>flex parse.l

C:\Users\deepi\OneDrive\Desktop\C C++\Compiler design>bison -d parse.y

C:\Users\deepi\OneDrive\Desktop\C C++\Compiler design>gcc parse.tab.c lex.yy.c -w

C:\Users\deepi\OneDrive\Desktop\C C++\Compiler design>a.exe
#include <stdio.h>
int main(){}
func begin main
func end

#include <stdio.h>
int main(){int i=3;}
Function redeclaration not allowed

C:\Users\deepi\OneDrive\Desktop\C C++\Compiler design>a.exe
int main(){
func begin main
int a=8;
t0 = 8
int b=3;
t1 = 3
return a+b;
t2 = a + b
printf(a+b);
refparam a
←[31m4 syntax error +
←[31mStatus: Parsing Failed - Invalid
←[0m
C:\Users\deepi\OneDrive\Desktop\C C++\Compiler design>a.exe
int main()
func begin main
int a=8938;
t0 = 8938
func end

int b=098;
ERROR at line no. 2
0
←[31m2 syntax error 0
←[31mStatus: Parsing Failed - Invalid
←[0m
C:\Users\deepi\OneDrive\Desktop\C C++\Compiler design>a.exe
int a=8938;←[31m0 syntax error
←[31mStatus: Parsing Failed - Invalid
←[0m
```

Testing the parser file by sending test files:



```
int b=098;
ERROR at line no. 2
0
←[31m2 syntax error 0
←[31mStatus: Parsing Failed - Invalid
←[0m
C:\Users\deepi\OneDrive\Desktop\C C++\Compiler design>a.exe
int a=8938;←[31m0 syntax error
←[31mStatus: Parsing Failed - Invalid
←[0m
C:\Users\deepi\OneDrive\Desktop\C C++\Compiler design>a.exe test1.c
func begin main
t0 = 0
t1 = 0
a = t1
L0:
t2 = 10
t3 = a < t2
IF not t3 GoTo L1
t4 = a + 1
a = t4
t5 = "H1"
Expression doesn't match return type of function

C:\Users\deepi\OneDrive\Desktop\C C++\Compiler design>a.exe test2.c
func begin myfunc
func end

func begin main
refparam i
refparam result
call myfunc, 1
func end

←[32mStatus: Parsing Complete - Valid←[0m
                              ←[36mSYMBOL TABLE←[0m
                               ------------
   SYMBOL |           CLASS |       TYPE |      VALUE |    LINE NO |         NESTING | PARAMS COUNT |
----------------------------------------------------------------------------------------------------
        a |      Identifier |        int |            |          2 |           99999 |          -1 |
        i |      Identifier |        int |            |          9 |           99999 |          -1 |
        n |      Identifier |        int |            |          9 |           99999 |          -1 |
   return |         Keyword |            |            |          4 |            9999 |          -1 |
      int |         Keyword |            |            |          2 |            9999 |          -1 |
     main |        Function |       void |            |          7 |            9999 |           0 |
   myfunc |        Function |        int |            |          2 |            9999 |           1 |
     void |         Keyword |            |            |          7 |            9999 |          -1 |
```



```
      NAME |           TYPE
---------------------------------------------------------------------------------

C:\Users\deepi\OneDrive\Desktop\C C++\Compiler design>a.exe test3.c
func begin main
t0 = x + y
←[31m4 syntax error =
←[31mStatus: Parsing Failed - Invalid
←[0m
C:\Users\deepi\OneDrive\Desktop\C C++\Compiler design>a.exe test1.c
func begin main
t0 = 0
t1 = 0
a = t1
L0:
t2 = 10
t3 = a < t2
IF not t3 GoTo L1
t4 = a + 1
a = t4
GoTo L0:
L1:
L2:
t5 = 0
t6 = a > t5
IF not t6 GoTo L3
GoTo L2:
L3:
func end

←[32mStatus: Parsing Complete - Valid←[0m
                              ←[36mSYMBOL TABLE←[0m
                               ------------
    SYMBOL |           CLASS |       TYPE |      VALUE |    LINE NO |         NESTING | PARAMS COUNT |
----------------------------------------------------------------------------------------------------
         a |      Identifier |        int |          0 |          4 |           99999 |           -1 |
       for |         Keyword |            |            |          5 |            9999 |           -1 |
    return |         Keyword |            |            |          7 |            9999 |           -1 |
       int |         Keyword |            |            |          2 |            9999 |           -1 |
      main |        Function |        int |            |          2 |            9999 |            0 |
     while |         Keyword |            |            |         10 |            9999 |           -1 |


                              ←[36mCONSTANT TABLE←[0m
                               --------------
      NAME |           TYPE
---------------------------------------------------------------------------------
        10 | Number Constant
         0 | Number Constant
```

The parser file detects error and informs user where the error has occurred i.e.. which line the error has occurred.

Make file

For intermediate code generation:

Flex filename.l

Bison -d filename.y

Gcc filename.tab.c lex.yy.c -w

a.exe test_filename.c

For only flex file:

Flex filename.l

Gcc lex.yy.c

```
A.exe
```

For bison file:

```
Bison -d filename.y

Flex filename.l

Lex.yy.c filename.tab.c
```

Limitations

1. Mips code generations is not done.
2. Multi dimensional array is not done.

Features

1. Error detection.
2. Can be able to detect syntax and semantic error with linenumber.
3. Symbol table.
4. Can add comments both single line comments and multi line comments.
5. Functions are implemented.
6. Can be able to add header files.
7. Conditional statements are implemented.
8. Loops can be added - while and for.
9. Function can take parameters.