PROJECT REPORT

COMPILER CONSTRUCTION (CSC-323)

BS (CS) - 5 (A)

Basic C++ Compiler

GROUP MEMBERS

Name	Enrollment	Work Distribution
M.Ahsan	**02-134202-087**	**Research, Coding**
Aliyan Rehman	**02-134202-052**	**Report Writing and Coding**
Usama Ashraf	**02-134202-044**	**Report Writing and Research**

Submitted to Lab Engineer:

MRS. SABA IMTIAZ

BAHRIA UNIVERSITY KARACHI CAMPUS

Department of Computer Science

- *Abstract:*

In this day and age, we as humans interact with and benefit from various software applications all the time in our daily life. These software applications *(like Word, PDF, Adobe, Facebook, Instagram, Twitter, YouTube, Google Maps, Video games, FinTech Apps, Food Groceries App, Weather Apps, etc. etc.)* are made using programming languages. And, in turn, these programming languages are built with the help of compilers and interpreters. Low and behold, compilers and interpreters are something essential to the yields of the modern-day computing phenomena! Hence, in order to become better, productive and competitive computer scientists or engineers, we are expected to be well-versed in programming languages. This is only possible if we understand compilers and the art of compiler design at a greater, much deeper level. In other words, good grip on compilers will make us a better programmer and give us a nice balanced flavor of theory and practice as we are applying knowledge about mathematical models: regular expressions, automata, grammars and graph algorithms, etc. And what better way to achieve this then working on making a compiler for your own programming language.

- *Problem statement:*

Programming language with a suitable compiler that is simple, customized, easy-to-use and is based on C++ Language. The compiler should at least include compiler phases of lexical analyzer, syntax analyzer and semantic analyzer. The learning outcomes for the project should be increased and clearer understanding of both, the technical and practical aspects of compiler design.

- *Overview of Compiler Design:*

A compiler is a computer program that ***transforms a source language (high-level language (HLL)) into a machine language (low-level language (LLL))*** without changing the program meaning. It is an intermediary between the machine-readable language and the human-readable language. The principles of compiler design give an overview of the translation and optimization processes.

A compiler can perform various operations such as parsing, preprocessing, lexical analysis, and semantic analysis. It can also perform code generation and code optimization. These operations are implemented in phases that consist of inputs and outputs.

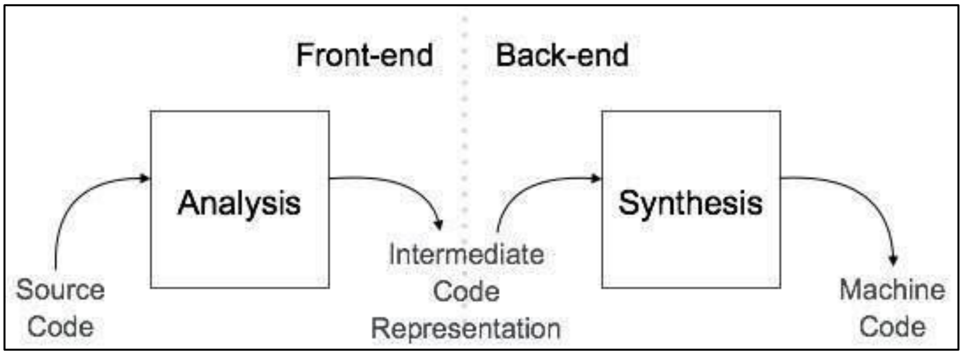
There are various types of compilers. Some of the common types include: -

- **Source-to-source compiler**
- **Cross compiler**

- **JIT (Just in time) compiler**
- **Hardware compiler**

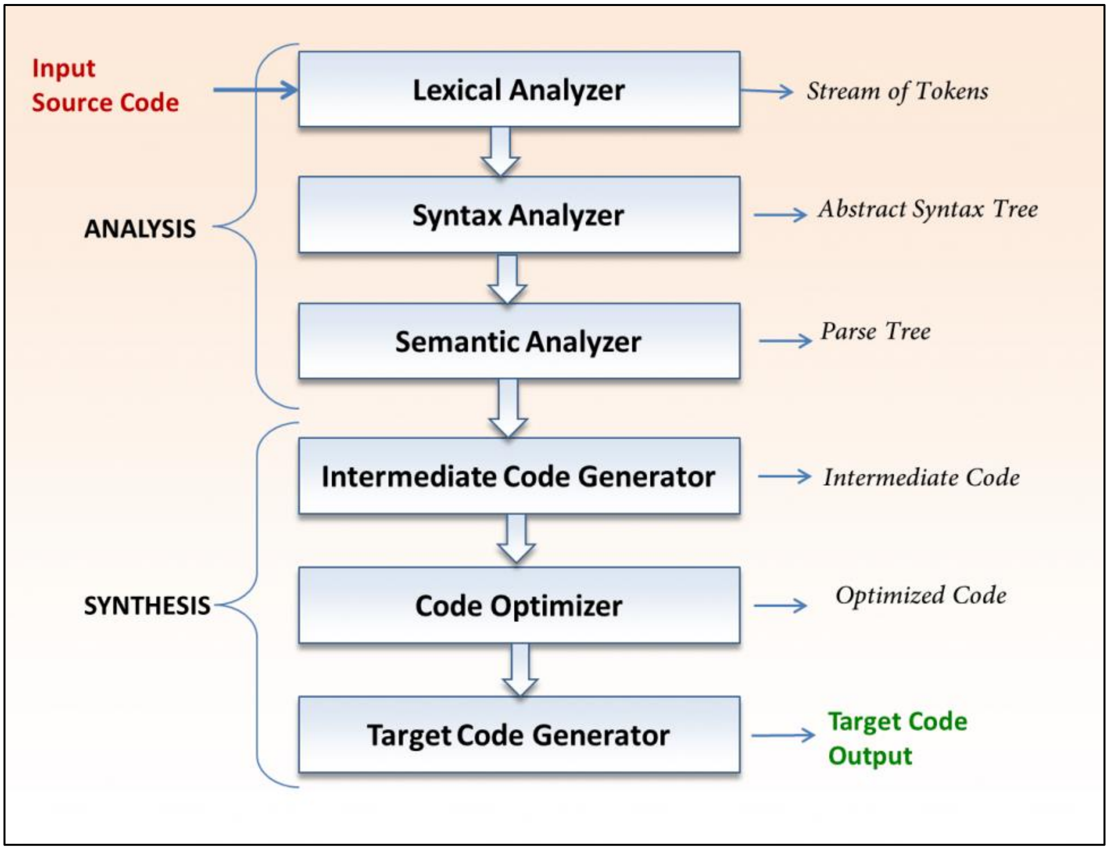
It typically also is a crucial part of the *language processing system* that many modern programming languages like C, C++ are based upon. *It comes along with components like pre-processor, assembler, linker, loader, debugger etc. etc. in it*.

The compiler design architecture can be divided into two main parts: ***analysis and synthesis*.**



- ***Phases of a compiler:***

The following diagram shows the main phases of a compiler. These phases are in the two aforementioned parts of the compiler design architecture.



- **Lexical analysis:** This is the first phase of the compiler that receives the source code, scans, and transforms it into *lexemes*. These *lexemes* are represented by the lexical analyzer in a *token* form. *Tokens* consist of various categories such as separators, identifiers, operators, comments, and keywords.

- **Syntax analysis:** This phase is also referred to as *parsing*. It uses the tokens generated in the previous phase to produce a syntax tree (parse tree). It checks whether the token expressions are syntactically correct.

- **Semantic analysis:** This phase checks whether the language rules in the parse tree have been followed. Semantic information is added to the parse trees produced in the syntax analysis. The semantic analysis phase performs various operations such as checking for errors, associating variables with corresponding definitions, and issuing warnings. The output of this phase is in the form of annotated syntax tree.

- **Intermediate code generation:** This phase involves generating an intermediate code that can be translated into the final machine code. Intermediate representation can be in various forms such as three-address code (language independent), byte code, or stack code.

- **Code optimization:** This is an optional phase that improves or optimizes the intermediate code to enable the output to be run faster. This phase eliminates unnecessary code lines and ensures that the output occupies less space.
- **Code generation:** This is the final stage that transforms the optimized code into the desired machine code.

- *Introducing Basic C++ Compiler and its compiler:*

Basic C++ is a simple and customized programming language based on Cpp. It is heavily inspired by the C++ and python programming language and whose reflection can be seen in many of its features. The language is very rigid as it is strictly bounded to the tokens and grammar defined, which for now are simple and don't offer much flexibility in writing our code for input. .

The Lexer Func is used to break input text into a collection of tokens specified by a collection of regular expression rules. The Parser class is used to recognize language syntax that has been specified in the form of a context free grammar.

Basic C++ Language Rules/CFG's

<assign_st> --> ID assign_op <ID_const> ;

<ID_const> --> ID | <const>

<const> --> int_const | float_const | char_const | str_const

<cond> --> <ID_const> RO <ID_const> | <ID_const>

<body> --> { <m_st> } | <s_st>

<m_st> --> <s_st x m_st> | null

<s_st> --> <decl> | <while_st> | <for_st> | <do_while> | <assign_st> | <if_else> | <switch_case>

<decl> --> DT ID <init> ;

<init> --> = <ID_const>

For Loop:

<for_st> --> for(<X> <Y>; <Z>) { <body> }

<X> --> <decl> | <Assign_st> | ;

<Y> --> <const> | null

<X> --> ID INCDEC | <assign_st> |null

DO While:

<do_while> --> do { <m_st> } while (<cond>) ;

While:

<while> --> while (<cond>) <body>

If-Else:

<if_else> --> if (<cond>) <body> <else>

<else> --> else <body>

Switch Case:

<switch_case> --> switch <ID_const> { <case> <default> }

<case> --> case <const> : <body> <break_cont>

<break_cont> --> break | continue | null

<default> --> default : <body> | null

Structure:

<struct> --> DT ID struct { <struct_body> }

<struct_body> --> ID DT | null

Language Specification:

Motivation:

To create a Language which have basic concepts of C++ Language and other languages. Which will be used to give and idea of programming and confidence to new developers in their future endeavors

Targeted Audience:

New Developers

Language Paradigm:

Structured Paradigm

Case sensitivity:

Yes, it will be case sensitive

Keywords:

Keyword	Keyword	Keyword
If	Str(string)	For
Else	Bin (bool)	True
Integer (int)	Nil (void)	False
Fract (float)	Null	Write(cout)
Lfract(double)	While	Read(cin)
Alpha(char)	do	struct
Break	continue	Switch
Case	default	Goto
Return		

Data Types:

1. Integer (int)
2. Fract (float)
3. Lfract (Double)
4. Alpha (char)
5. Bin (Boolean)
6. str (string)
7. nil (void)

Iterative statements:

For

While

Do while

Conditional Statements:

If

If-Else

Comments (Multi line + Single line):

# :**Single line**

“““Statement; ””” :**Multi line Comment**

Line terminator:

;

Operators (Mention Each Operator with their class part):

Addsub Class

+ ,-

Divmul Class

*,/,%

Assignment Operator Class

=,+=,-=,/=,*=,%=

And Class

&&

OR Class

||

INCDEC Class

++,--

NOT Class

!

Relational Operator Class

>,>=,<=,<,==,!=

Punctuators (Mention each punctuator with their class part):

(rd bracket, ( )

(rd bracket,))

(cur bracket, {)

(cur bracket, })

(sq bracket, [ )

(sq bracket,])

(dot, .)

(comma, ,)

(semi colon, ;)

Identifiers:

RE= (_ + A-Z + a-z) (0-9 + _ + A-Z + a-z)*

Syntax Specification: (start, end, declaration, functions, array 1D/2D, classes/structures, loops, conditional statements):

Prototype

return_type func_name (parameters);

Definition

for(condition){statement;}

while(condition){statement;}

do{statement;}

while(condition);

Structure

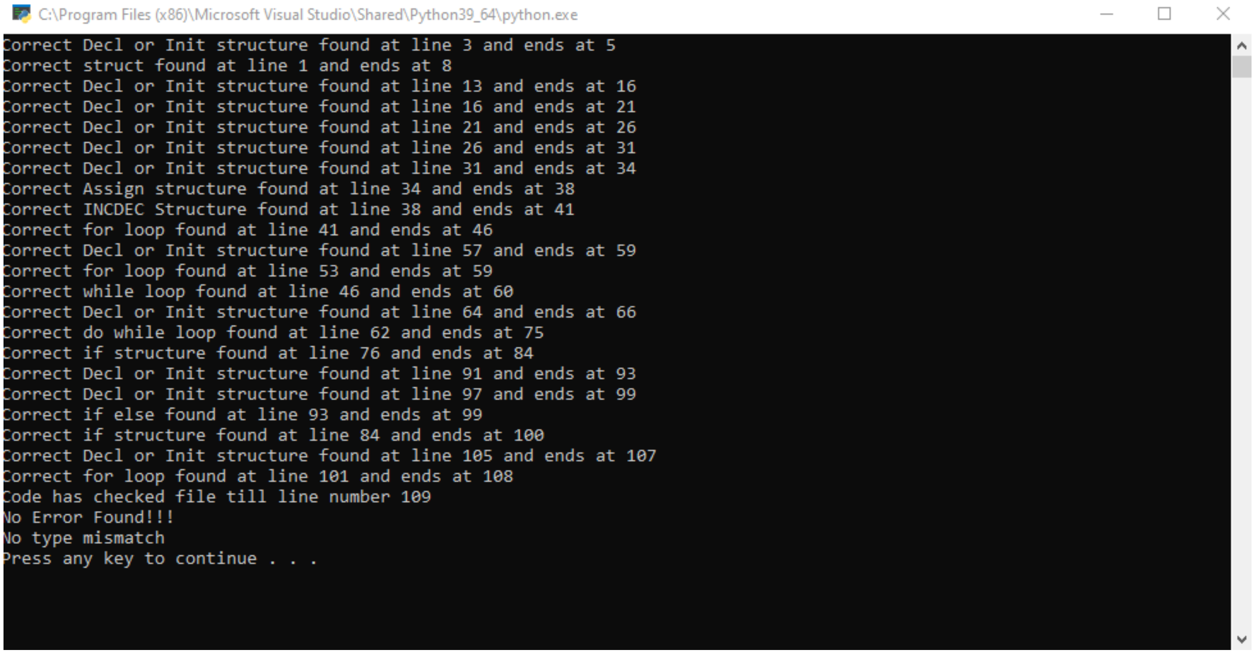
Struct{statements};

Variable Declaration

Datatype identifier = value

Output

Screenshots


C:\Program Files (x86)\Microsoft Visual Studio\Shared\Python39_64\python.exe

```
Correct Decl or Init structure found at line 3 and ends at 5
Correct struct found at line 1 and ends at 8
Correct Decl or Init structure found at line 13 and ends at 16
Correct Decl or Init structure found at line 16 and ends at 21
Correct Decl or Init structure found at line 21 and ends at 26
Correct Decl or Init structure found at line 26 and ends at 31
Correct Decl or Init structure found at line 31 and ends at 34
Correct Assign structure found at line 34 and ends at 38
Correct INCDEC Structure found at line 38 and ends at 41
Correct for loop found at line 41 and ends at 46
Correct Decl or Init structure found at line 57 and ends at 59
Correct for loop found at line 53 and ends at 59
Correct while loop found at line 46 and ends at 60
Correct Decl or Init structure found at line 64 and ends at 66
Correct do while loop found at line 62 and ends at 75
Correct if structure found at line 76 and ends at 84
Correct Decl or Init structure found at line 91 and ends at 93
Correct Decl or Init structure found at line 97 and ends at 99
Correct if else found at line 93 and ends at 99
Correct if structure found at line 84 and ends at 100
Correct Decl or Init structure found at line 105 and ends at 107
Correct for loop found at line 101 and ends at 108
Code has checked file till line number 109
No Error Found!!!
No type mismatch
Press any key to continue . . .
```

Sample Code:

```
#Hello World program

struct{

integer f;

};
```

```
integer main(){
integer a;
integer z=5;
fract d=5.5;
alpha sb='a';
bin y = true;
y=false;
z++;
for()
{
}
while(a<5)
{
for()
{
integer a;
}
}
do
{
integer a;

}
while(int<7);
}
if(a<8)
{
}
if(a<8)
{
```

fract d;

}

else

{

fract d;

}

for()

{

integer a;

}

}

"""Ahsan is good boy"""

Code

Lexer Function:

```
def lexer(code):
    # Split the code into lines
    lines = code.split('\n')

    # Keep track of the current line number
    line_number = 1
    # Iterate through the lines of code
    for line in lines:
        # Use the regular expression to find all tokens in the line
        match = re.match(token_pattern, line)
        while match:
            token = match.group(0)
            if token in keywords:
                yield ('KW', token, line_number)
            elif token=='while':
                yield ('while', token, line_number)
            elif token=='for':
                yield ('for', token, line_number)
            elif token=='do':
                yield ('do', token, line_number)
            elif token=='if':
                yield ('if', token, line_number)
            elif token=='else':
                yield ('else', token, line_number)
            elif token=='write':
                yield ('write', token, line_number)
            elif token=='read':
```

```
    yield ('read', token, line_number)
elif token=='struct':
    yield ('struct', token, line_number)
elif token=='break':
    yield ('break', token, line_number)
elif token=='switch':
    yield ('switch', token, line_number)
elif token=='case':
    yield ('case', token, line_number)
elif token=='return':
    yield ('return', token, line_number)
elif token=='main':
    yield ('main', token, line_number)
elif token.isdigit():
    yield ('Int_Const', token, line_number)
elif '.' in token:
    yield ('Flt_Const', token, line_number)
elif token in DT:
    yield ('DT', token, line_number)
elif token.startswith('\"'):
    yield ('Str_Const', token, line_number)
elif token.startswith('\''):
    yield ('Ch_Const', token, line_number)
elif token in ['&&']:
    yield ('And Operator', token, line_number)
elif token in ['||']:
    yield ('Or Operator', token, line_number)
elif token in ['!']:
    yield ('Not Operator', token, line_number)
elif token in ['=', '+=', '-=', '*=', '/=','%=']:
    yield ('Aop', token, line_number)
elif token in ['{']:
    yield ('RParen', token, line_number)
elif token in ['}']:
    yield ('LParen', token, line_number)
elif token in ['[']:
    yield ('RSqBrac', token, line_number)
elif token in [']']:
    yield ('LSqBrac', token, line_number)
elif token in ['(']:
    yield ('RRdBrac', token, line_number)
elif token in [')']:
    yield ('LRdBrac', token, line_number)
elif token in [';']:
    yield (';', token, line_number)
elif token in ['.']:
    yield ('Dot', token, line_number)
elif token in [',']:
    yield ('comma', token, line_number)
elif token in ['!=', '==', '<', '>', '<=', '>=']:
    yield ('RO', token, line_number)
elif token in ['+', '-']:
    yield ('AddSub', token, line_number)
elif token in ["++", '--']:
    yield ('IncDec', token, line_number)
elif token in ['*', '/', '%']:
    yield ('DivMul Operator', token, line_number)
elif token.isspace():
```

```
                print("",end="")
            elif token == "true" or token=="false":
                yield ('Bool_Const', token, line_number)
            else:
                yield ('ID', token, line_number)
            line = line[match.end():]
            match = re.match(token_pattern, line)
        if line.strip():
            yield 'error'
            yield ('ERROR', f"Invalid token found on line 
{line_number}",line.strip())
            break


        line_number += 1
```

Syntax Class:

```
class syntax:

id_const={"Int_Const":True , "Flt_Const" : True ,"Str_Const":True,"Ch_Const" : True, "ID": 
True , "Bool_Const":True}


def class_part (self,token,path):

  file=open(path,"a")

  cp = token[0]

  file.write(cp)

  file.write("\n")

  file.close()



def check_decl(self,start_line_num, lines):

  if "DT"in lines[start_line_num] :

    if "ID" in lines[start_line_num+1]:

        if "Aop" in lines[start_line_num+2]:

          if  syntax.id_const[lines[start_line_num + 3].strip()]:

            if ";" in lines[start_line_num + 4]:

              end_line_num = start_line_num + 4

              return end_line_num

            else:
              print("expected ;")
              return -1
          else:
           print("expected Const")
           return -1
        elif ";" in lines[start_line_num+2]:
            end_line_num= start_line_num+2
            return end_line_num
        else:
         print("expected Aop or ;")
         return -1
def check_assign(self,start_line_num, lines):
  if "ID"in lines[start_line_num] :
        if "Aop" in lines[start_line_num+1]:
          if syntax.id_const[lines[start_line_num + 2].strip()]:
            if ";" in lines[start_line_num + 3]:
              end_line_num = start_line_num + 3
              return end_line_num
            else:
              print("expected ;")
              return -1
          else:
           print("expected Const")
           return -1
        elif "IncDec" in lines[start_line_num+1]:
            if ";" in lines[start_line_num+2]:
                end_line_num = start_line_num + 2
                #print("Increment or Decrement Structure found")
                return end_line_num
        else:
```

```
      print("expected Aop or INCDEC")
      return -1
  else:
      print("expected ID")
      return -1

def check_else(self,start_line_num,lines):
  if "else" in lines[start_line_num]:
    if "RParen" in lines[start_line_num + 1] and "LParen" not in lines[start_line_num + 2]:
        end_line_num_body=syntax.check_body(self,start_line_num + 2,lines)
        if end_line_num_body != -1:
            end_line_num = end_line_num_body
            return end_line_num

        elif "LParen" in lines[start_line_num + 2]:
            end_line_num = start_line_num + 2
            return end_line_num
        else:
            print("expected }")
            return -1
    else:
            print("expected {")
            return -1

  else:
    print("Expected else")
    return -1

def check_if(self,start_line_num, lines):
  if "if" in lines[start_line_num]:
    if "RRdBrac" in lines[start_line_num + 1]:
     if syntax.id_const[lines[start_line_num + 2].strip()]:
      if "RO" in lines[start_line_num+3]:
       if syntax.id_const[lines[start_line_num + 4].strip()]:
        if "LRdBrac" in lines[start_line_num + 5]:
         if "RParen" in lines[start_line_num + 6] and "LParen" not in lines[start_line_num +
7]:
          end_line_num_body=syntax.check_body(self,start_line_num + 7,lines)
          if end_line_num_body != -1:
            start_line_num = end_line_num_body
            if "LParen" in lines[start_line_num + 1] and "else"  not in lines[start_line_num +
2] :
             end_line_num = start_line_num + 1
             return end_line_num
            if "LParen" in lines[start_line_num + 1] and "else"  in lines[start_line_num + 2] :
             end_line_num=syntax.check_else(self,start_line_num+2,lines)
             if end_line_num != -1:
              print(f"Correct if else found at line {start_line_num+1} and ends at
{end_line_num + 1}")
              return end_line_num
            else:
              print("No correct if else structure found")
              return -1



         elif "LParen" in lines[start_line_num + 7] and "else"  not in lines[start_line_num + 8]
:
            end_line_num = start_line_num + 7
            return end_line_num
         elif "LParen" in lines[start_line_num + 7] and "else"  in lines[start_line_num + 8] :
            end_line_num=syntax.check_else(self,start_line_num+8,lines)
            if end_line_num != -1:
             print(f"Correct if else found at line {start_line_num+1} and ends at
{end_line_num + 1}")
             return end_line_num
            else:
             print("No correct while loop found")
             return -1
         else:
            print("expected }")
            return -1
        else:
            print("expected {")
            return -1
      else:
            print("expected )")
            return -1
    else:
            print("expected (")
            return -1
  else:
    print("Expected if")
    return -1



def check_for_loop(self,start_line_num, lines):
  if "for" in lines[start_line_num]:
    if "RRdBrac" in lines[start_line_num + 1]:
      if "LRdBrac" in lines[start_line_num + 2]:
        if "RParen" in lines[start_line_num + 3] and "LParen" not in lines[start_line_num +
4]:
          end_line_num_body=syntax.check_body(self,start_line_num + 4,lines)
          if end_line_num_body != -1:
```

```
                     end_line_num = end_line_num_body
                     return end_line_num

              elif "LParen" in lines[start_line_num + 4]:
                     end_line_num = start_line_num + 4
                     return end_line_num


              else:
                     print("expected { or }")
                     return -1
          else:
                     print("expected )")
                     return -1
      else:
                     print("expected (")
                     return -1
   else:
      print("Expected for")
      return -1



def check_do_while_loop(self,start_line_num, lines):
 if "do" in lines[start_line_num]:
  if "RParen" in lines[start_line_num + 1] and "LParen" not in lines[start_line_num + 2]:
   end_line_num_body=syntax.check_body(self,start_line_num + 2,lines)
   if end_line_num_body != -1:
    start_line_num = end_line_num_body

    if "LParen" in lines[start_line_num + 1]:
     if "while" in lines[start_line_num + 2]:
```

```
   if "RRdBrac" in lines[start_line_num + 3]:
    if syntax.id_const[lines[start_line_num + 4].strip()]:
     if "RO" in lines[start_line_num+5]:
      if syntax.id_const[lines[start_line_num + 6].strip()]:
       if "LRdBrac" in lines[start_line_num + 7]:
        if ";" in lines[start_line_num + 8]:
         end_line_num = start_line_num + 8
         return end_line_num
        else:
                     print("expected ;")
                     return -1
       else:
                     print("expected )")
                     return -1
      else:
                     print("expected Const or Variable")
                     return -1
     else:
                     print("expected RO")
                     return -1
    else:
                     print("expected Const or Variable")
                     return -1
   else:
                     print("expected (")
                     return -1
  else:
        print("expected while")
        return -1
else:
       print("Expected { or }")
       return -1

 else:
       print("Expected do")
       return -1

def check_while_loop(self,start_line_num, lines):
   if "while" in lines[start_line_num]:
       if "RRdBrac" in lines[start_line_num + 1]:
         if syntax.id_const[lines[start_line_num + 2].strip()]:
           if "RO" in lines[start_line_num+3]:
            if syntax.id_const[lines[start_line_num + 4].strip()]:
              if "LRdBrac" in lines[start_line_num + 5]:
                if "RParen" in lines[start_line_num + 6] and "LParen" not in lines[start_line_num +
7]:
                   end_line_num_body=syntax.check_body(self,start_line_num + 6,lines)
                   if end_line_num_body != -1:
                       end_line_num = end_line_num_body
                       return end_line_num

                elif "LParen" in lines[start_line_num + 7]:
                       end_line_num = start_line_num + 7
                       return end_line_num

                else:
                       print("expected { or }")
                       return -1
              else:
                       print("expected )")
                       return -1
            else:
                       print("expected Const or Variable")
                       return -1
           else:
                       print("expected RO")
                       return -1
         else:
                       print("expected Const or variable")
                       return -1
       else:
           print("expected (")
           return -1
   else:
       print("Expected while")
       return -1
def check_body(self,start_line_num, lines):
      syn=syntax()
      i=start_line_num
      line_num=0
      while i < len(lines):
       line_num=i
       line=lines[i]
       if "while" in line:
           end_line_num = syn.check_while_loop(line_num, lines)
           if end_line_num != -1:
               print(f"Correct while loop found at line {i+1} and ends at {end_line_num + 1}")
               i=end_line_num
```

```
        line_num=i
        return end_line_num
    else:
        print("No correct while loop found")
        return -1

if "for" in line:
    end_line_num = syn.check_for_loop(line_num, lines)
    if end_line_num != -1:
        print(f"Correct for loop found at line {i+1} and ends at {end_line_num + 1}")
        i=end_line_num
        return end_line_num

    else:
        print("No correct for loop found")
        return -1
if "do" in line:
    end_line_num = syn.check_do_while_loop(line_num, lines)
    if end_line_num != -1:
        print(f"Correct do while loop found at line {i+1} and ends at {end_line_num + 1}")
        i=end_line_num
        return end_line_num

    else:
        print("No correct Do while loop found")
        return -1
if "if" in line:
    end_line_num = syn.check_if(line_num, lines)
    if end_line_num != -1:
        print(f"Correct if structure found at line {i+1} and ends at {end_line_num + 1}")
```

```
                i=end_line_num
                return end_line_num

            else:
                print("No correct if structure found")
                return -1
        if "DT" in line:
         if "main" in lines[i+1]:
             i+=1
             continue
         elif "ID" in lines[i+1]:
             end_line_num = syn.check_decl(line_num, lines)
             if end_line_num != -1:
                print(f"Correct Decl or Init structure found at line {i+1} and ends at {end_line_num +
1}")
                i=end_line_num
                return end_line_num

             else:
                print("No correct Decl found")
                return -1
        if "ID" in line:
            end_line_num = syn.check_assign(line_num, lines)
            if end_line_num != -1 and end_line_num!=line_num+2:
                print(f"Correct Assign structure found at line {i+1} and ends at {end_line_num + 1}")
                i=end_line_num
                return end_line_num
            elif end_line_num==line_num+2:
                print(f"Correct INCDEC Structure found at line {i+1} and ends at {end_line_num +
1}")
                i=end_line_num
```

```
               return end_line_num

           else:
               print("No correct Init structure found")
               return -1

       if "}" in line:
           i+=1
           end_line_num=i
           return end_line_num

       else:
           i+=1


def check_struct(self,start_line_num, lines):
    if "struct" in lines[start_line_num]:
          if "RParen" in lines[start_line_num+1] and "LParen" not in lines[start_line_num + 2]:
             end_line_num_body=syntax.check_body(self,start_line_num + 2,lines)
             if end_line_num_body != -1:
                       start_line_num = end_line_num_body
                       if "LParen" in lines[start_line_num+1]:
                        if ";" in lines[start_line_num+2]:
                           end_line_num = start_line_num + 2
                           return end_line_num
                        else:
                           print("Expected ;")
                           return -1
                       else:
                           print("Expected }")
```

```
                          return -1

          elif "LParen" in lines[start_line_num+2]:
              if ";" in lines[start_line_num+3]:
                   end_line_num = start_line_num + 3
                   return end_line_num
              else:
                   print("Expected ;")
                   return -1
          else:
                   print("Expected }")
                   return -1
    else:
                   print("Expected struct")
                   return -1
def check_syntax(self):
 path_cp_file = "D:\Compiler for Basic C++\Compiler For Basic C++\Cp.txt"
 syn=syntax()
 line_num =0
 i=0
 with open(path_cp_file, 'r') as f:
   lines = f.readlines()
   while i < len(lines):
       line_num=i
       line=lines[i]
       if "while" in line:
           end_line_num = syn.check_while_loop(line_num, lines)
           if end_line_num != -1:
               print(f"Correct while loop found at line {i+1} and ends at {end_line_num + 2}")
               i=end_line_num
```

```
        line_num=i
    else:
        print("No correct while loop found")
        break
if "for" in line:
    end_line_num = syn.check_for_loop(line_num, lines)
    if end_line_num != -1:
        print(f"Correct for loop found at line {i+1} and ends at {end_line_num + 2}")
        i=end_line_num

    else:
        print("No correct for loop found")
        break
if "do" in line:
    end_line_num = syn.check_do_while_loop(line_num, lines)
    if end_line_num != -1:
        print(f"Correct do while loop found at line {i+1} and ends at {end_line_num + 2}")
        i=end_line_num

    else:
        print("No correct Do while loop found")
        break
if "if" in line:
    end_line_num = syn.check_if(line_num, lines)
    if end_line_num != -1:
        print(f"Correct if structure found at line {i+1} and ends at {end_line_num + 2}")
        i=end_line_num

    else:
        print("No correct if structure found")
```

```
          break
    if "DT" in line:
     if "main" in lines[i+1]:
        i+=1
        continue
     elif "ID" in lines[i+1]:
        end_line_num = syn.check_decl(line_num, lines)
        if end_line_num != -1:
          print(f"Correct Decl or Init structure found at line {i+1} and ends at {end_line_num + 2}")
          i=end_line_num

        else:
          print("No correct Decl found")
          break
    if "ID" in line:
       end_line_num = syn.check_assign(line_num, lines)
       if end_line_num != -1 and end_line_num!=line_num+2:
          print(f"Correct Assign structure found at line {i+1} and ends at {end_line_num + 2}")
          i=end_line_num
       elif end_line_num==line_num+2:
          print(f"Correct INCDEC Structure found at line {i+1} and ends at {end_line_num + 2}")
          i=end_line_num

       else:
          print("No correct Init structure found")
          break
    if "struct" in line:
       end_line_num = syn.check_struct(line_num, lines)
       if end_line_num != -1:
        print(f"Correct struct found at line {i+1} and ends at {end_line_num + 2}")
        i=end_line_num


      else:
        print("No correct struct structure found")
        break
    if "$" in line:
      i+=1
    else:
      i+=1
  print(f'Code has checked file till line number {line_num+1}')
```

Semantic File:

```
import re
file=open("D:\Compiler for Basic C++\Compiler For Basic C++\code_comless.txt","r")
inp=file.read()
dt=['nil',
'bin',
'str',
'alpha',
'lfract',
'fract',
'integer']
tab=[]
tok=[]
tok=inp.split(' ')


def check_type_mismatch(code):
  pattern =
re.compile(r'(integer|fract|lfract|alpha|str|bin)\s+\w+\s*(=\s*\S+)?(,|;)')

  lines = code
  found_mismatch = False

  for line in lines:

    match = pattern.search(line)
    if match:

      variable_type = match.group(1)
      variable_value = match.group(2)

      if variable_value:
        variable_value = variable_value.strip('= ')
        if (variable_type == 'integer' and not variable_value.isdigit()) or \
           (variable_type == 'fract' and not variable_value.replace('.', '',
1).isdigit()) or \
           (variable_type == 'lfract' and not variable_value.replace('.', '',
1).isdigit()):

          found_mismatch = True
          print(f'Type mismatch in declaration: {line}')
        elif variable_type == 'alpha' and not (variable_value.startswith("'") and
variable_value.endswith("'")):
          found_mismatch = True
          print(f'Type mismatch in declaration: {line}')
        elif variable_type == 'str' and not (variable_value.startswith('"') and
variable_value.endswith('"')):
          found_mismatch = True
          print(f'Type mismatch in declaration: {line}')

  return found_mismatch




def lookup(name,scope):
    i=0
    if len(tab)!=0:
        while(i<len(tab)):
            if name in tab[i] and scope in tab[i]:
                return False
            else:
                i+=1
        return True
    else:
        return True

def insert(name,typee,scope):
    tab.append([name,typee,scope])

def main():
    error=False
    flag=True
    scopee=1
    i=0
    while(i<len(tok)):
        if tok[i]=='{':
            flag=True
        scopee =scopee+1
        i+=1
        while(i<len(tok) and flag==True):
            namee=""
            tipe=""
            if tok[i] in dt:
                tipe=tok[i]
                i+=1
                if re.match('^[a-zA-Z]+$',tok[i]):
                    namee=tok[i]
                    if lookup(namee,scopee):
                        insert(namee,tipe,scopee)
                    else:
                        print("ERROR:",namee," is already defined at scope",scopee)
                        error=True
                i+=1
                if tok[i]=='=':
                    None
                    i+=1
                    if tok[i-3]=='integer' and re.match('^[0-9]+$',tok[i]):
                        i+=1
                    elif tok[i-3]=='alpha' and re.match("^'[A-Za-z0-9]'$",tok[i]):
                        i+=1
                    else:
                        print("ERROR:",namee,"'s Datatype Mismatch at scope",scopee)
                        error=True
                        i+=1
                if tok[i]==';':
                    i+=1
                else:
                    print("ERROR:",namee,"'s Terminator missing at scope",scopee)
                    error=True
                if tok[i]=='}':
                    i+=1
                    scopee = scopee-1
                    flag=False
            elif re.match('^[a-zAZ]+$',tok[i]):
                print("ERROR:",tok[i]," is not Declared as scope",scopee)
                error=True
                i+=1
            else:
                i+=1
        else:
            i+=1
    if(not error):
        print("No Error Found!!!")
```

Comment_Remove File:

```
  def comment_rem(path_base,path_comless):
 file=open(path_base,"r")
 file2=open(path_comless,"w")
 str=file.read() #to read file
 i=0
 k=0
# print("Before # Comment Removal\n")
# print(str)
# print("\nAfter # Comment Removal\n")
 while(i!=len(str)):
     if(str[i]=='\"' and str.startswith("\"\"",i+1)): #to remove block comments
         if(str.find("\"\"\"",i+3)!=-1):
          k=str.find("\"\"\"",i+3)
          i=k+3
         else:
             print("Error!!! Comment not closed")
     elif(str.startswith("#",i) and str[i+1]!="\"" and str[i+1]!="\'"): #to remove # comments
      k=str.find("\n",i)
      i=k+1
     else: #to print remaining code
         #print(str[i],end='')
         file2.write(str[i])
         i=i+1
 file.close()
 file2.close()
```

- ***CONCLUSION AND FUTURE WORK:***

In this report, we have thoroughly described the nature and functionality of our programming language Basic C++, along with discussing the key concepts and knowledge of compiler construction phases like lexical and syntax analysis etc. while also applying them to the subject matter, which was the making of our very own custom compiler.

Moreover, exploring the topic, experience of working on the project and the end-goal achieved not only expanded our understanding but also made us realize the future enhancements we can make to our project. First of all, we can build an even better parser that is highly flexible and caters for the many variations and approaches a user can take in writing the code. Since we encountered issues in building a grammar which could deal with the amendments we made to our Lexer, we hope to dig deep and find a solution to this problem in the future.

Also, we wish to add the other phases of compiler to our project in order to create a high-quality compiler which can be deployed in the real world. In other words. work on an applications-specific compiler.