NATIONAL INSTITUTE OF TECHNOLOGY, DELHI

COMPILER DESIGN PROJECT
MINI-C COMPILER
(ZERCOIN)

WORKED ON, BY : VADLAMUDDI NEELVITTAL BHARATH (191210053)
VINAY CHOUDHARY (191210059)
VINAY JAISWAL (191210060)

UNDER GUIDANCE OF : DR. SHELLY SACHDEVA
NEHA BANSAL
KANIKA SONI
PRABHAT PUSHP

DEPARTMENT OF COMPUTER SCIENCE ENGINEERING

INDEX

ABSTRACT

OBJECTIVE

The main goal of this project is to design a mini compiler for a subset of the C language as part of the Compiler Design Lab(CO351) course. The compiler is to be built in four phases finishing at the Intermediate Code Generation Phase. The subset of the C language chosen is to include certain data types, constructs and functions as mentioned in the specifications below. The implementation will be carried out with LEX and YACC.

PHASES OF THE PROJECT

- Implementation of Scanner/Lexical Analyser
- Implementation of Parser
- Implementation of Semantic Checker for C language
- Intermediate Code generation for C language

MINI-C-COMPILER SPECIFICATIONS:

The compiler is going to support the following cases :

1. **Keywords** - int, break, continue, else, for, void, goto, char, do, if, return, while.
2. **Identifiers** - identified by the regular expression (_ |{letter})({letter}|{digit}| _){0,31}.
3. **Comments** - single line comments (specified with // or /* ... */), multi-line comments (specified with /* ... */).
4. **Strings** - can identify strings mentioned in double quotes.
5. **Preprocessor directives** - can identify filenames (stdio.h) after #include.
6. **Data types** - int,char (supports comma declaration).
7. **Arrays** - int A[n]
8. **Punctuators** - [ ] , <> , {} , , , : , = , ; , # , ” ” , ’ ’
9. **Operators** - arithmetic (+, - , * , /) ,increment(++) and decrement(--), assignment (

   =)
10. **Condition** - if else
    - Syntax:
      - if (condition == true){

        //code

        }

        else{

        //code

        }
11. **Loops -**

- Syntax

- while(condition){

  //code
  }

- for(initialization;condition;increment/decrement){
  //code
  }

- while(condition){
  //code
  }

- do{
  //code

  }while(condition);

The loop control structures that are supported are break,continue and goto.

12. Functions-

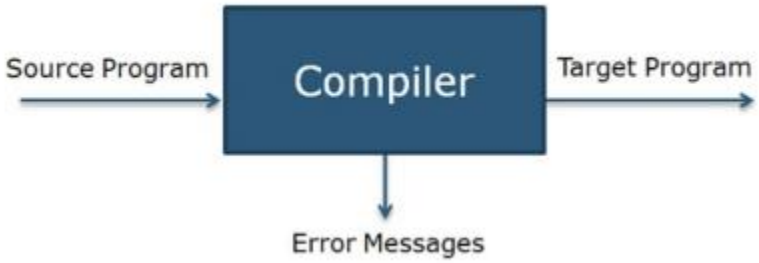
Void functions with no return type and a single parameter will be implemented
Syntax - void sample_function(int a){ //code }
The mini compiler will be implemented in a straightforward fashion , using Lex and YACC tools starting off with the Scanner as the first module.

TOOLS USED :
YACC, LEX

INTRODUCTION

Compiler

A compiler is a program that can read a program in one language - the source language and translate it into an equivalent program in another language - the target language.

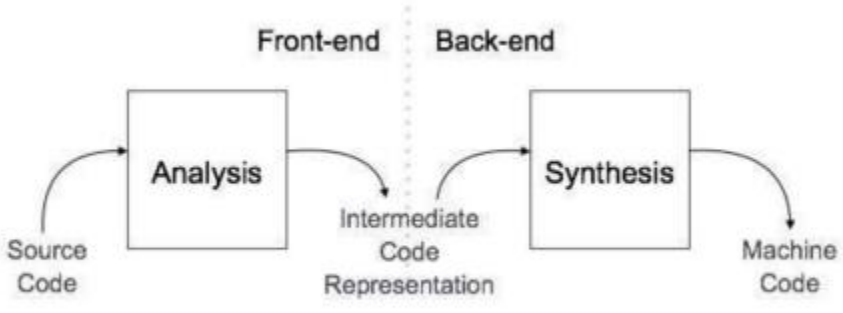


Structure of a compiler

A compiler can broadly be divided into two phases based on the way they compile i.e analysis phase (front end) and synthesis phase (back end).

The analysis phase breaks up the source program into constituent pieces and imposes a grammatical structure on them. It then uses this structure to create an intermediate representation of the source program. If the analysis part detects that the source program is either syntactically ill formed or semantically unsound, then it must provide informative messages, so the user can take corrective action. The analysis part also collects information about the source program and stores it in a data structure called a symbol table, which is passed along with the intermediate representation to the synthesis part.

The synthesis phase constructs the desired target program from the intermediate representation and the information in the symbol table. The analysis part is often called the front end of the compiler; the synthesis part is the back end.



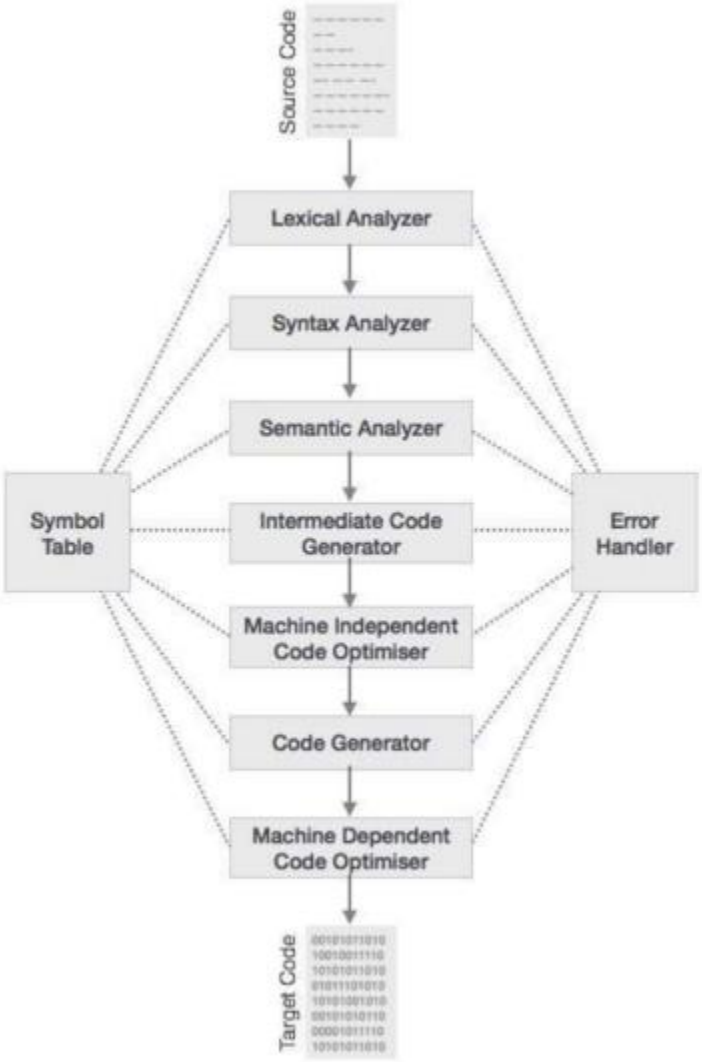
Analysis Phase – The analysis phase can be divided into three phases as follows:

1. Lexical Analyser
2. Syntax Analyser
3. Semantic Analyser

Synthesis Phase – The synthesis phase can be divided into three phases as follows:

1. Intermediate Code Generator
2. Code Optimiser
3. Code Generator

The entire process can be visualised as follows -



LEXICAL ANALYSER

Lexical analysis is the first phase of a compiler. It takes the modified source code from language preprocessors that are written in the form of sentences. The lexical analyzer breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code.

Lexemes are said to be a sequence of characters (alphanumeric) in a token. There are some predefined rules for every lexeme to be identified as a valid token. These rules are defined by grammar rules, by means of a pattern. A pattern explains what can be a token, and these patterns are defined by means of regular expressions.

If the lexical analyzer finds a token invalid, it generates an error. The lexical analyzer works closely with the syntax analyzer. It reads character streams from the source code, checks for legal tokens, and passes the data to the syntax analyzer when it demands.

SAMPLE INPUT AND OUTPUT :

- **Input 1**

```
/* The lexical analyser should remove all the valid comments and throw an
exception for all the invalid comments */

#include<stdio.h>

int main(){

    // Single line comment

    /* Single line comment with different beginning */

    /* Multi-line comment

    should be removed by lex */

    /* Nested comments are /* not */ allowed */

  return 0;

}

```

⇨ **Output 1**

```
D:\Mini C\Phase 1>output test-case-comments.c
        <stdio.h>                                       : Preprocessor directive
        int                                             : Keyword
        main                                            : Keyword
        (                                               : Open Round Bracket
        )                                               : Closed Round Bracket
        {                                               : Open Curly Bracket
Error 14: Nested comments are invalid
Lexical analysis finished
```

⇨ **Input 2**

```c
#include<stdio.h>
//Illegal header format
#include

int main(){
  //Illegal identifier
  int 2invalid;
  //Identifier too long
  int abcdefghijklmnopqrstuvwxyzabcdefgh;
  int x , y;
  //Illegeal character
  y = x $ x;
  /* Nested comments /*are not*/ allowed */
  return 0;
}
```

⇨ **Output 2**

```
D:\Mini C\Phase 1>output test-case-errors.c
        <stdio.h>                                               : Preprocessor directive
Error 2: Header format not allowed
        int                                                     : Keyword
        main                                                    : Keyword
        (                                                       : Open Round Bracket
        )                                                       : Closed Round Bracket
        {                                                       : Open Curly Bracket
        int                                                     : Keyword
Error 6: Illegal identifier format
        ;                                                       : Delimiter
        int                                                     : Keyword
Error 8: Identifier too long,must be between 1 to 32 characters
        ;                                                       : Delimiter
        int                                                     : Keyword
        x                                                       : Identifier
        ,                                                       : Comma
        y                                                       : Identifier
        ;                                                       : Delimiter
        y                                                       : Identifier
        =                                                       : Assignment Operator
        x                                                       : Identifier
Error 11: Illegal character
        x                                                       : Identifier
        ;                                                       : Delimiter
Error 12: Nested comments are invalid
Lexical analysis finished
```

⇨ **Input 3**

```
1   #include<stdio.h>
2   #include "string.h"
3
4   int main(){
5
6       printf("This is a valid string.");
7       printf("This string does not terminate);
8       return 0;
9   }
10
11
```

⇨ **Output 3**

```
D:\Mini C\Phase 1>output test-case-strings.c
        <stdio.h>                                       : Preprocessor directive
        "string.h"                                      : Preprocessor directive
        int                                             : Keyword
        main                                            : Keyword
        (                                               : Open Round Bracket
        )                                               : Closed Round Bracket
        {                                               : Open Curly Bracket
        printf                                          : Identifier
        (                                               : Open Round Bracket
        "This is a valid string."                       : String
        )                                               : Closed Round Bracket
        ;                                               : Delimiter
        printf                                          : Identifier
        (                                               : Open Round Bracket
Error 6: Illegally terminated string
Lexical analysis finished
```

⇨ **Input 4**

```c
/* This file tests the detection of allowed keywords , identifiers and other tokens such as punctuators , operators */

//Identifies prepocessor directives
#include<stdio.h>
#include "stdlib.h"
//Identifies macro preprocessor directives
#define MAX 100

int main(){

    //Identifies keywords int , long , char , if , else and operators
    int a;
    long int b;
    char c;
    int e,f,g;

    //Identifies constants
    int f = 0xAB;
    c = 'A';
    f = 1;
    g = -5;
```

```
23      //Identifies arithmetic operators
24      e = f+g;
25      e = f-g;
26
27      // Identifies conditions
28      if(c=='A')
29          a = 10;
30      else
31          a = 30;
32
33      // Identifies loops
34
35      for ( a = 0; a<3; a++){
36      }
37
38      return 0;
39  }
40
```

⇨ **Output 4**

```
D:\Mini C\Phase 1>output test-case-tokens.c
        <stdio.h>                                       : Preprocessor directive
        "stdlib.h"                                      : Preprocessor directive
        MAX                                             : Macropreprocessor directive
        100                                             : Integer Constant
        int                                             : Keyword
        main                                            : Keyword
        (                                               : Open Round Bracket
        )                                               : Closed Round Bracket
        {                                               : Open Curly Bracket
        int                                             : Keyword
        a                                               : Identifier
        ;                                               : Delimiter
        long                                            : Keyword
        int                                             : Keyword
        b                                               : Identifier
        ;                                               : Delimiter
        char                                            : Keyword
        c                                               : Identifier
        ;                                               : Delimiter
        int                                             : Keyword
        e                                               : Identifier
        ,                                               : Comma
        f                                               : Identifier
        ,                                               : Comma
        g                                               : Identifier
        ;                                               : Delimiter
        int                                             : Keyword
        f                                               : Identifier
        =                                               : Assignment Operator
        0xAB                                            : Hexadecimal Constant
        ;                                               : Delimiter
        c                                               : Identifier
        =                                               : Assignment Operator
        'A'                                             : Character
        ;                                               : Delimiter
        f                                               : Identifier
        =                                               : Assignment Operator
        1                                               : Integer Constant
        ;                                               : Delimiter
        g                                               : Identifier
        =                                               : Assignment Operator
```

```
-5                                                  : Integer Constant
;                                                   : Delimiter
e                                                   : Identifier
=                                                   : Assignment Operator
f                                                   : Identifier
+                                                   : Arithmetic Operator
g                                                   : Identifier
;                                                   : Delimiter
e                                                   : Identifier
=                                                   : Assignment Operator
f                                                   : Identifier
-                                                   : Arithmetic Operator
g                                                   : Identifier
;                                                   : Delimiter
if                                                  : Keyword
(                                                   : Open Round Bracket
c                                                   : Identifier
==                                                  : Comparison Operator
'A'                                                 : Character
)                                                   : Closed Round Bracket
a                                                   : Identifier
=                                                   : Assignment Operator
10                                                  : Integer Constant
;                                                   : Delimiter
else                                                : Keyword
a                                                   : Identifier
=                                                   : Assignment Operator
30                                                  : Integer Constant
;                                                   : Delimiter
for                                                 : Keyword
(                                                   : Open Round Bracket
a                                                   : Identifier
=                                                   : Assignment Operator
0                                                   : Integer Constant
;                                                   : Delimiter
a                                                   : Identifier
<                                                   : Comparison Operator
3                                                   : Integer Constant
;                                                   : Delimiter
a                                                   : Identifier
++                                                  : Increment Operator
)                                                   : Closed Round Bracket
{                                                   : Open Curly Bracket
}                                                   : Closed Curly Bracket
```

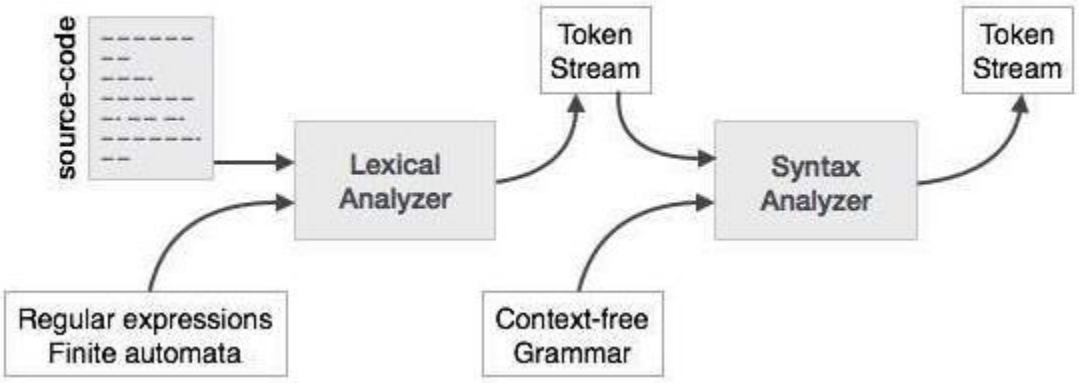
```
        0                                                       : Integer Constant
        ;                                                       : Delimiter
        }                                                       : Closed Curly Bracket

Printing Symbol Table


---------------------------------------

        (lexeme, token)
        (    e, IDENTIFIER)
        (    f, IDENTIFIER)
        (    g, IDENTIFIER)
        (   if, KEYWORD)
        ( char, KEYWORD)
        ( main, KEYWORD)
        ( else, KEYWORD)
        (  for, KEYWORD)
        (  int, KEYWORD)
        ( long, KEYWORD)
        (return, KEYWORD)
        (    a, IDENTIFIER)
        (    b, IDENTIFIER)
        (    c, IDENTIFIER)
---------------------------------------
```

```
Printing Constant Table


-----------------------------------------

        (lexeme, token)
        (  100, INT_CONSTANT)
        (    0, INT_CONSTANT)
        (    1, INT_CONSTANT)
        (    3, INT_CONSTANT)
        (   10, INT_CONSTANT)
        (   -5, INT_CONSTANT)
        (   30, INT_CONSTANT)
        ( 0xAB, HEX_CONSTANT)
-----------------------------------------
Lexical analysis finished
```

PARSER

A parser is a compiler or interpreter component that breaks data into smaller elements for easy translation into another language. A parser takes input in the form of a sequence of tokens or program instructions and usually builds a data structure in the form of a parse tree or an abstract syntax tree.

In the syntax analysis phase the parser verifies whether or not the tokens generated by the lexical analyser are grouped according to the syntactic rules of the language.The parser obtains a string of tokens from the lexical analyzer and verifies that the string can be the grammar for the source language. It detects and reports any syntax errors and produces a parse tree from which intermediate code can be generated.



SAMPLE INPUT AND OUTPUT :

- **Input 1**

```c
int main()
{

  int a = 10;
  int x = 1, y = 0;
  for (i = 0; i < 10; i = i + 9)
  {
    for (g = 3; g < 10; g = g - 5)
    {
      printf("%d %d", i, g);
    }
  }
  diff = x - y;
  int rem = x % y;
  printf("Total = %d \n", total);

  int result = IncreaseBy10(x);
}

int IncreaseBy10(int x)
{
  return x + 10;
}
```

- **Output 1**

```
D:\Mini C\Phase-2>output test-case-nested.c
Line no: 16 Error message: syntax error Token: (

Parsing failed.
```

⇨ Input 2

```c
int main(){
    int a, b;
    int *c;
    int a, b = 2 * 2 - 8 && 9;
    a = a - b;
    while (a > 0)
    {
        a = a - 1;
    }
    if (a == 2){
        if (b == 2 + a){
            a = a - 2;
            }
        else{
            a = a + 2;
        }       You, 2 days ago • first c
    }
}
int function(){
    // Here's a comment
    char *a = "abcd";

    return 0;
}
```

⇨ Output 2

```
D:\Mini C\Phase-2>output test-case-noerrors.c

Parsing complete.

        Symbol table

----------------------------------------

        (lexeme, value, Data type, Line Number)
        ( main, 2147483647.000000, INT, 0)
        (function, 2147483647.000000, INT, 27)
        (    a, 2147483647.000000, CHAR, 2)
        (    b, 2147483647.000000, INT, 2)
        (    c, 2147483647.000000, INT, 3)
----------------------------------------

        Constant table

----------------------------------------

        (lexeme, value, Data type, Line Number)
        (    0, 0.000000, INT, 9)
        (    1, 1.000000, INT, 11)
        (    2, 2.000000, INT, 5)
        (    8, 8.000000, INT, 5)
        (    9, 9.000000, INT, 5)
        ("abcd", 0.000000, CHAR, 30)
----------------------------------------

D:\Mini C\Phase-2>
```

⇨ **Input 3**

```c
int main()
{
    int b;
    int a = b + 5;

    // printf("THIS IS A STRING");

   for(i =0 ; i < 5; i = i + 1       You,
   {
       a = a * 2;
   }
}

// Missing Paranthesis for Function
int func()
{
    int a = 2 * 2;
```

⇨ **Output 3**

```
D:\Mini C\Phase-2>output test-case-paranthesis.c
Line no: 8 Error message: syntax error Token: {

Parsing failed.

D:\Mini C\Phase-2>
```

SEMANTIC CHECKER

Semantics of a language provide meaning to its constructs, like tokens and syntax structure. Semantics help interpret symbols, their types, and their relations with each other. Semantic analysis judges whether the syntax structure constructed in the source program derives any meaning or not.

The semantic analyzer uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition. It also gathers type information and saves it in either the syntax tree or the symbol table, for subsequent use during intermediate-code generation.

An important part of semantic analysis is type checking, where the compiler checks that each operator has matching operands. For example, many programming language definitions require an array index to be an integer; the compiler must report an error if a floating-point number is used to index an array.

The language specification may permit some type conversions called coercions. For example, a binary arithmetic operator may be applied to either a pair of integers or to a pair of floating-point numbers. If the operator is applied to a floating-point number and an integer, the compiler may convert or coerce the integer into a floating-point number or report an error.

SAMPLE INPUT AND OUTPUT :

- **Input 1**

```c
#include<stdio.h>

int a(int b, int c){
    return b;
}

void main(){
    int c = a(3.5,2);
}
```

- **Output 1**

```
D:\Mini C\Phase-3>output test-arg-mismatch.c
Line no: 7 Error message: Parameter and argument types do not match Token: )

D:\Mini C\Phase-3>
```

⇨ **Input 2**

```c
#include <stdio.h>

int b[10];

int main()
{
    int a[10];
    // Correct array expression
    a[1] = b[1] + b[2];

    // Incorrect array index,exceeds dimensions
    a[9] = b[1];

    // Invalid array index
    b[2] = a[1];

    return 1;
}       You, 2 days ago • first commit …
```

⇨ **Output 2**

```
D:\Mini C\Phase-3>output test-arrays.c

Parsing complete.

       Symbol table

---------------------------------------

       (lexeme,            value, Data type, Line Number, isArray, ArrayDimensions, isFunction, Nesting Level, num_params
       (  main, 2147483647.000000,       INT,           4,       0,              0,          1,             1,          0)
       (     a, 2147483647.000000,       INT,           6,       1,             10,          0,             2,          0)
       (     b, 2147483647.000000,       INT,           2,       1,             10,          0,             1,          0)
---------------------------------------

       Constant table

---------------------------------------

       (lexeme,   value, Data type, Line Number
       (    1, 1.000000,       INT,          16)
       (    1, 1.000000,       INT,          14)
       (    1, 1.000000,       INT,          11)
       (    1, 1.000000,       INT,           8)
       (    1, 1.000000,       INT,           8)
       (    2, 2.000000,       INT,          14)
       (    2, 2.000000,       INT,           8)
       (    9, 9.000000,       INT,          11)
       (   10, 10.000000,       INT,           6)
       (   10, 10.000000,       INT,           2)
---------------------------------------
```

⇨ **Input 3**

```c
#include<stdio.h>

int a(int b, int c){
    return b;
}

int a(int d){
    return d;
}
```

⇨ **Output 3**

```
D:\Mini C\Phase-3>output test-dup-func.c
Line no: 6 Error message: Duplicate Functions Not allowed
 Token: )

Parsing failed.

D:\Mini C\Phase-3>
```

INTERMEDIATE CODE GENERATOR

In the analysis-synthesis model of a compiler, the front end analyzes a source program and creates an intermediate representation , from which the back end generates target code. Ideally details of the source language are confined to the front end , and the details of the target machine to the back end. With a suitably defined intermediate representation , a compiler for language i and machine j can then be built by combining the front end of language and i and back end of machine j. This tackles the need the create a native compiler for each language and machine.

During the translation of a source program into the object code for a target machine, a compiler may generate a middle-level language code, which is known as intermediate code or intermediate text. The complexity of this code lies between the source language code and the object code. The intermediate code can be represented in the form of postfix notation, syntax tree, directed acyclic graph (DAG), three-address code, quadruples, and triples.

SAMPLE INPUT AND OUTPUT :

- **Input 1**

```c
#include <stdio.h>
int main()
{
    int temp=5;
    int arr[100];
    arr[3]=2;
    int k=arr[temp-3];
    return 0;
}
```

⇨ Output 1

```
D:\Mini C\Phase-4>output test-array-case.c
'clear' is not recognized as an internal or external command,
operable program or batch file.

Parsing complete.

        Symbol table

---------------------------------------

        (lexeme,        Data type,      Line Number,    isArray,        ArrayDimensions,        isFunction,     Nesting Level,  num_params
        (k,             INT,            7,              0,              0,                      0,              2,
0       )
        (main,          INT,            2,              0,              0,                      1,              1,
0       )
        (arr,           INT,            5,              1,              100,                    0,              2,
0       )
        (temp,          INT,            4,              0,              0,                      0,              2,
0       )
---------------------------------------


_____________________________________________

         Intermediate Code Generation

_____________________________________________

func begin main
temp = 5
t0 = 4 * 3
t1 = arr[t0]
t1 = 2
t2 = temp - 3
t3 = 4 * t2
t4 = arr[t3]
k = t4
return 0
func end
exit

_____________________________________________
```

⇨ Input 2

```c
#include <stdio.h>

int main () {

   int a = 10;
   while( a < 20 ) {
      a++;

      if( a > 15) {
         break;
      }
   }

   return 0;
}
```

⇨ Output 2

```
D:\Mini C\Phase-4>output test-break-case.c
'clear' is not recognized as an internal or external command,
operable program or batch file.

Parsing complete.

        Symbol table

----------------------------------------

        (lexeme,        Data type,      Line Number,    isArray,        ArrayDimensions,        isFunction,     Nesting Level,  num_params
        (main,          INT,            2,              0,              0,                      1,              1,              0       )
        (a,             INT,            4,              0,              0,                      0,              2,              0       )
----------------------------------------


_____________________________________________

         Intermediate Code Generation

_____________________________________________

func begin main
a = 10
t0 = a < 20
L1 :
if t0 goto L2
goto L3
L2 :
a = a + 1
t1 = a > 15
if t1 goto L4
goto L5
L4 :
goto L1
goto L3
L5 :
return 0
func end
exit

_____________________________________________
```

⇨ Input 3

```c
#include <stdio.h>

int main () {

   int a = 10;
   do {
      a = a + 1;
   }while( a < 20 );

   return 0;
}
```

⇨ Output 3

```
D:\Mini C\Phase-4>output test-dowhile-case.c
'clear' is not recognized as an internal or external command,
operable program or batch file.

Parsing complete.

        Symbol table

----------------------------------------

        (lexeme,        Data type,      Line Number,    isArray,        ArrayDimensions,        isFunction,     Nesting Level,  num_params
        (main,          INT,            2,              0,              0,                      1,              1,              0       )
        (a,             INT,            4,              0,              0,                      0,              2,              0       )
----------------------------------------


_____________________________________________

         Intermediate Code Generation

_____________________________________________

func begin main
a = 10
L1 :
t0 = a + 1
a = t0
t1 = a < 20
if t1 goto L1
goto L2
L2 :
return 0
func end
exit

_____________________________________________
```

REFERENCES:

1. Aho A.V, Sethi R, and Ullman J.D. Compilers: Principles, Techniques, and Tools. Addison-Wesley, 1986

2. http://www.di.univr.it/documenti/OccorrenzaIns/matdid/matdid065185.pdf

3. http://cse.iitkgp.ac.in/~bivasm/notes/LexAndYaccTutorial.pdf

4. GeeksforGeeks.

NOTE OF THANKS

We are extremely grateful to our faculty-in-charge Dr.Shelly Ma’am for providing us such a great opportunity to work on this project and learn the basics of Compiler Design.

We extend our gratitude to TA’s in-charge, Neha Basal Ma’am, Kanika Soni Ma’am and Prabhat Pushp Sir for their constant support and guidance throughout the making of the project and making it a success.