Compiler Manual:

A manual for the compiler I have designed is given below:

Tokens:

We know that a lexical analyzer reads and converts the input into a stream of tokens to be analyzed by the parser. The tokens that I have used in the making of my compiler has been described below:

1. SLCOM: SLCOM is a token that is returned to the parser when the lexical analyzer matches the pattern for a single line comment. The regex for a single line comment is : "#".*
2. NUMERIC: The token NUMERIC is returned to the parser when a pattern of "Num" is matched.
3. WHOLE: The token WHOLE is returned to the parser when a pattern of "Whole" is matched.
4. TEXT: The token TEXT is returned to the parser when a pattern of "Text" is matched.
5. STRING: This token is returned to the parser when a regex pattern: "\"" (.)* "\"" is matched. Anything between double quotations is a String.
6. NUMf: This token is returned to the parser when a floating point number is scanned. The regex patterns for floating point are:

   [+]?[0-9]+[.][0-9]+

   [-][0-9]+[.][0-9]+

7. 7. NUMi: This token is returned to the parser when an integer number is scanned. The regex pattern for integer is: [0-9]+
8. SHOW: This token is returned to the parser when an string “Show” is matched by the scanner.
9. LPT: This token is returned to the parser when a pattern “(“ is matched.
10. RPT: This token is returned to the parser when a pattern “)“ is matched.
11. LCB: This token is returned to the parser when a pattern “{“ is matched.
12. RCB: This token is returned to the parser when a pattern “}“ is matched.
13. LTB: This token is returned to the parser when a pattern “[“ is matched.
14. RTB: This token is returned to the parser when a pattern “]“ is matched.
15. CM: This token is returned to the parser when a pattern “,“ is matched.
16. SM: This token is returned to the parser when a pattern “;“ is matched.
17. COL: This token is returned to the parser when a pattern “:“ is matched.
18. ADD: This token is returned to the parser when a pattern “+“ is matched.
19. SUB: This token is returned to the parser when a pattern “-“ is matched.

20. MUL: This token is returned to the parser when a pattern “*“ is matched.
21. DIV: This token is returned to the parser when a pattern “/“ is matched.
22. GT: This token is returned to the parser when a pattern “>“ is matched.
23. LT: This token is returned to the parser when a pattern “<“ is matched.
24. EQL: This token is returned to the parser when a pattern “=“ is matched.
25. VAR: This token is returned to the parser for a variable. A variable is something that matches the regex pattern: [a-z]+ .
26. IF: This token is returned to the parser when a pattern “Whether“ is matched.
27. ELSEIF: This token is returned to the parser when a pattern “OrElse“ is matched.
28. ELSE: This token is returned to the parser when a pattern “Or“ is matched.
29. MATCH: This token is returned to the parser when a pattern “Match“ is matched.
30. CHECK: This token is returned to the parser when a pattern “Check“ is matched.
31. NOMATCH: This token is returned to the parser when a pattern “NoMatch“ is matched.
32. REPEAT: This token is returned to the parser when a pattern “Repeat“ is matched.

33. UNTIL: This token is returned to the parser when a pattern “Until“ is matched.
34. ITER: This token is returned to the parser when a pattern “iter“ is matched.
35. DO: This token is returned to the parser when a pattern “do“ is matched.
36. MULFN: This token is returned to the parser when a pattern “MUL“ is matched.
37. SUMFN: This token is returned to the parser when a pattern “SUM“ is matched.
38. PALINDROMEFN: This token is returned to the parser when a pattern “PLD“ is matched.
39. VOIDMAIN: This token is returned to the parser when a pattern “Null START“ is matched.

Main Function:

In every language the program execution starts from the main function. In this language, the program has to be written inside the START function. The structure is defined below:

```
Null START(){
        # write program here
}
```

Data Types:

There are mainly 3 data types in my language:

Num	Stands for Numeric. Supports any real number. e.g: 47.53, 15.6, -0.05 etc.
Whole	Stands for Whole number / integer. Supports any integer number. e.g: 10,53,47 etc
Text	Supports any string. A string is anything that is included inside double quotation. e.g: “Hello World!”

Operators, Operations & Precedence :

The operators and their operations are described below:

1. Assignment: The colon (:) operator is used for assignment. It has right associativity. E.g;
   Num y: 2.5;
   A value of 2.5 is assigned to variable ‘y’ which is of type: Numeric.
2. Addition: The plus (+) operator adds two expressions around it & is a binary operator. E.g;
   x: 2+4;
   Plus operator adds 2&4 and then colon assigns the value of 6 to x.
3. Subtraction: The minus (-) operator can be unary or binary. When unary, it multiplies the value of -1 to the

expression it is with. When binary, it subtracts the 2nd expression from the 1st. E.g:

x: 6-4;

A value of 2 is assigned to x

x: -2;

A value of -2 is assigned to x.

4. Multiplication: The multiplicative (*) operator is binary, t returns the product of the two expressions around it. E.g:

   x: 6*4;

   The product of 6 & 4 which is equal to 24 is assigned to x.

5. Division: The division (/) operator is binary. It divides the 1st expression by the 2nd expression & returns the value. A division by ‘0’ error is displayed if the 2nd expression has value equal to 0. E.g:

   x: 8/2;

   A value of 4 is assigned to x.

6. Greater: The greater (>) operator is binary. It checks if its left operand is greater than its right operand.

   e.g:

   x: 4>2;

   A value of 1 is assigned to x.

7. Less: The less (<) operator is binary. It checks if its left operand is smaller than its right operand.

   e.g:

x:4<2;

A value of 0 is assigned to x.

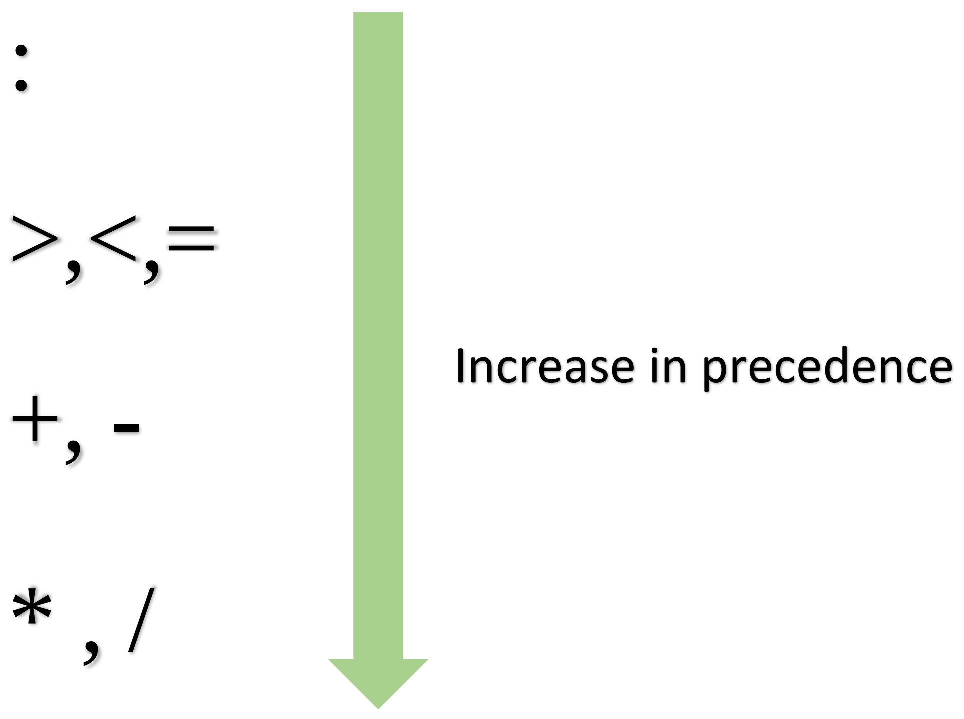
8. Equal: The equal (=) operator is binary. It checks if its left operand is equal to its right operand.

   e.g:

   x:4=2;

   A value of 0 is assigned to x.

The precedence of operators is as follows:



Variable Declarations & Assignment:

```
Num a, b: 1.5, c: 0.0;

Whole x;

x: 10;

Text s : “abcd”;
```

Variables can be declared & assigned on the same line or in a separate line. The colon (:) symbol is used for assignment.

Displaying Values:

The Show() function is used to display anything that’s passed inside it as a parameter. It can be an expression, variable or a string.

e.g:

```
Show ( “Hello World!”);
```

Output: Hello World!

Comment:

Anything starting with “#” is considered a single line comment. The parser has no action for the comment while parsing.

e.g:

```
#This is a comment
```

Built-in Functions:

1. SUM(x,y): The SUM() function takes two arguments and returns the summation value of those arguments.
2. MUL(x,y): The MUL() function takes two arguments & displays the result of multiplication between them.
3. PLD(s): The PLD() function takes a string argument & checks if it is a palindromic string or not. Returns 1 if it is otherwise 0.

Whether, OrElse , Or Statements:

This is the if, else if, else statement equivalent of the C language. The statement has three structures which are as follows:

1. Only one Whether statement:

```
Whether(exp){
# if expression is true, execute this block

}
```

2. One Whether statement and Or statement:

```
Whether(exp){
# if expression is true, execute this block
}
Or{
 #Otherwise execute this block
}
```

3. One Whether statement, one/multiple Or Else statements and one Or statement:

```
Whether(exp1){
#if expression 1 is true then execute this block
}
OrElse(exp2){
#if expression 2 is true then execute this block
}
.
.
.
OrElse(expn){
#if expression n is true then execute this block
}
Or{
#Otherwise execute this block
}
```

Check statements:

The check statement has one argument in which it takes an expression. There are one or more Match cases which matches the expression with certain values. If no value is matched then contents of NoMatch block is executed. The structure is as follows:

```
Check( exp1 ) {
        Match ( exp2 ) {
        #if exp1 matches with exp2 then execute this block
        }
        Match( exp3 ) {
        #if exp1 matches with exp3 then execute this block
        }
        .
        .
        .
        Match( expn ) {
        #if exp1 matches with expn then execute this block
        }
        NoMatch {
    #if no expressions match with exp1 then execute this block
        }
    }
```

Repeat loop statements:

The Repeat loop has 3 parts separated by two colons in between its parenthesis. The structure is as follows:

```
Repeat ( start_val, end_val, inc ){
        #do something
}
```

start_val -> start loop at “start_val”.

end_val - > end loop at “end_val”.

inc -> increment start_val by “inc” units after every loop.

If increment isn’t supplied then the statement follows this structure:

```
Repeat ( start_val, end_val){
        #do something
}
```

Here the value of “inc” is by default set to 1.

Until do loop statements:

The structure of Until do loops is as follows:

```
Until(start_val) , iter[iter_val] do : {
        #do something here
}
```

The loop starts at “start_val” and ends at 0. After every turn the value of start_val is either incremented or decremented by “iter_val” units. If iter_val is positive the value increments otherwise decrements.