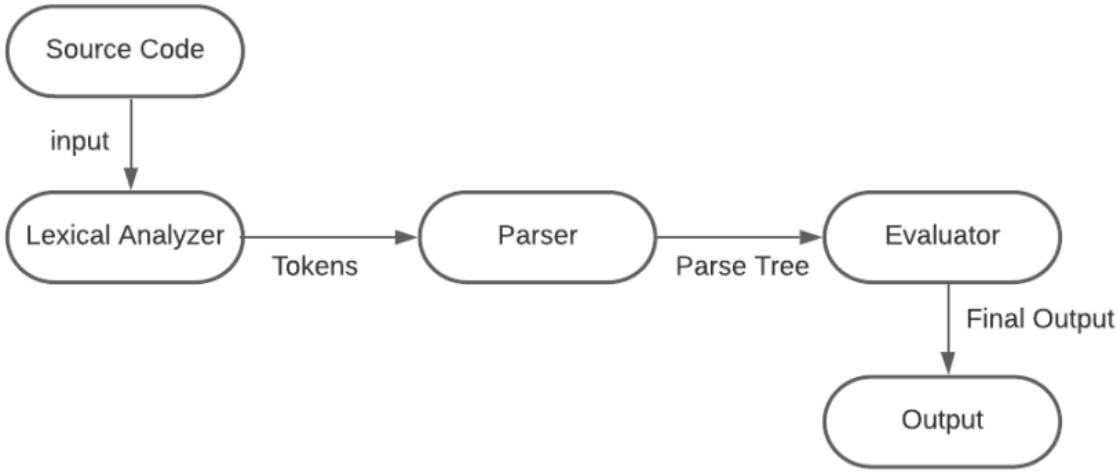
Team 27

SER 502 Project

NAME: ALBITOR

Extension: .albi

Project Flow or Execution Outline:



Source Code: A program that we create is stored in a file known as the source code. We can run this program to obtain its output. The file containing the source code has an "albi" file extension. Lexical

Analyzer: The source code file is fed into the lexical analyzer, which divides the program into smaller units known as tokens through lexical analysis. During this phase, any trailing white spaces and the program is inspected for syntax errors or inconsistencies.

Parser: The parser receives the tokens generated by the lexical analyzer and performs syntax analysis on the program. It creates a parse tree or syntax tree that outlines the program's structure. Additionally, it identifies any grammatical or syntax errors present. lark parser in python is used. The design language employed in this process is Prolog. DCG grammar is used. Top Down parsing technique is used for parse trees. List data structure is used.

Evaluator: The parse tree produced by the parser is utilized to generate the program's output. The Evaluator verifies that the program is semantically accurate and executes the required calculations. List data structure and prolog is used. It assesses the entire program to generate the desired output, which is both accurate and expected.

Output: This is the final output generated after the completion of the execution of our program in Albitor language.

TOKENS:

The token that can be recognized by the Albitor is listed below:

1. **Data Types:** a. String b. Integer c. Boolean
2. **Variable/Identifier:** Variables are named starting with a lowercase letter and can include uppercase and underscores. They can also be assigned to a value with "=" token (assignment) between the expression and identifier
3. **Conditional Statements:** a. Ternary Operator : (Boolean?true:false) b. if-then-else (eg: if Boolean true else false)
4. **Loops:** a. For loop (for assignment ; boolean ; increment) b. While loop (while condition true) c. For in range loop (for identifier in range(expression, expression)
5. **Arithmetic Operators:** a. Addition (+) b. Subtraction (-) c. Multiplication (*) d. Division (/) e. Brackets ('()') f. And (&&) g. Or (||) h. Mod (%)
6. **Comparison:** a. Equals (==) b. Not equals (!=) c. Greater than (>) d. Less than (=) f. Less than or equal to (<=)
7. **True/False:** a. true b. false c. Not (!)
8. **Print:** Print the given identifiers which include boolean, integers ,and strings.

IMPLEMENTATION PLAN:

1. The first step in designing any language would be defining definite grammar rules. So our first step is to define the grammar rules for our programming language Albitor.
2. After defining the grammar, we are supposed to breakdown the given program not tokens which is then fed to the system for evaluation.
3. For a token generation, we will be using Python and Lark which breakdowns our grammar and generates a token tree.
4. Now the final part is establishing semantic relationship between the grammar.
5. For this we will be writing the evaluation for the grammar we have defined in the presence of environment in the prolog.
6. After the grammar evaluation, we should verify whether the language is working by testing it against the programs written in "Albitor" language.
7. Testing the language against some of the test programs would be our final step to test our designed programming language.

GRAMMAR:

```
import lark

def parser_hy(program):

    p = lark.Lark(


    r'''

program : BEG block END
```

```
block : CBO command CBC
command :   statement command
        | statement
statement : print_statement SC
        | assignment_statement SC
        | declaration_statement SC
        | for_loop
        | while_loop
        | if_condition
        | ternary_operation
print_statement :   PRINT PO var PC
assignment_statement :  declaration_statement EQ expression
                | var EQ expression
declaration_statement : data_type var
for_loop :  FOR PO assignment_statement SC compare_operations SC assignment_statement PC block
        | FOR PO var IN RANGE PO arithmetic_operation COM arithmetic_operation PC PC block
while_loop :    WHILE PO compare_operations PC block
if_condition :  IF PO compare_operations PC block
            | IF PO compare_operations PC block ELSE block
expression :    boolean
            | string_literal
            | arithmetic_operation
arithmetic_operation :  arithmetic_operation1 ADDITION arithmetic_operation
                | arithmetic_operation1 SUBTRACTION arithmetic_operation
                | arithmetic_operation1
arithmetic_operation1 : arithmetic_operation2 MULTIPLICATION arithmetic_operation1
                | arithmetic_operation2 DIVISION arithmetic_operation1
                | arithmetic_operation2 PERCENTAGE arithmetic_operation1
                | arithmetic_operation2
```

```
arithmetic_operation2 : PO arithmetic_operation PC
                      | NUMBER
                      | var
compare_operations :    compare_statement LOGICAL_OPERATOR compare_operations
                      | compare_statement
compare_statement :     arithmetic_operation COMPARISON_OPERATOR arithmetic_operation
                      | NOT compare_statement
ternary_operation : compare_operations QUE block COL block
COMPARISON_OPERATOR:    "<"
                      | "<="
                      | ">"
                      | ">="
                      | "=="
                      | "!="
LOGICAL_OPERATOR :      "and"
                      | "or"
boolean :   TRUE
          | FALSE
NUMBER :    /[+-]?\d+/
data_type : INTEGER
          | BOOL
          | STRING
INTEGER :   "int"
BOOL    :   "bool"
STRING  :   "str"
BEG : "begin"
END : "end"
NOT : "not"
ADDITION : "+"
```

```
SUBTRACTION : "-"

DIVISION : "/"

MULTIPLICATION : "*"

PERCENTAGE : "%"

EQ : "="

SC : ";"

CBO : "{"

CBC : "}"

PO : "("

PC : ")"

QUE : "?"

COL : ":"

DQO : "'"

DQC : "'"

IF : "if"

ELSE : "else"

IN : "in"

FOR : "for"

PRINT : "print"

TRUE : "true"

FALSE : "false"

COM : ","

RANGE : "range"

WHILE : "while"

var :   VARIABLE_NAME

string_literal :    /"(\\.|[^\\"])*"/

%import common.WORD -> VARIABLE_NAME

WHITE : /\s+(?=([^\"]*[\"][^\"]*[\"])*[^\"]*$)/

%ignore WHITE
```

```
    ''',
    parser = 'lalr',
    start = 'program'
    )
    return str(p.parse(program))
def progaram_parser(program):
    tree = parser_hy(program)
    return tree
```

EVALUATION:

```
:- discontiguous eval_for_loop/3.
:- discontiguous eval_compare_statement/4.
run_program(P):- eval_program(P, []).

% Predicate for evaluating the program.
eval_program(tree(program,[_,Block,_]), InitialEnv):- eval_block(Block,InitialEnv, _EnvRes).

% Predicate for evaluating the block.
eval_block(tree(block,[_,Cmd,_]), Env, EnvRes):- eval_command(Cmd, Env, EnvRes).

% Predicate for evaluating the command.
eval_command(tree(command,[Stmt,Cmd]),Env, EnvRes):- eval_statement(Stmt,Env, EnvTemp),
eval_command(Cmd,EnvTemp, EnvRes).
eval_command(tree(command,[Stmt]), Env, EnvRes):- eval_statement(Stmt, Env, EnvRes).

% Predicate for evaluating statement.
eval_statement(tree(statement,[Print_stmt,_]), Env, EnvRes):- eval_print_statement(Print_stmt, Env,
EnvRes).
eval_statement(tree(statement,[Assignment_stmt,_]), Env, EnvRes):-
eval_assignment_statement(Assignment_stmt, Env, EnvRes).
```

```
eval_statement(tree(statement,[Declaration_stmt]), Env, EnvRes):-
eval_declaration_statement(Declaration_stmt, Env, EnvRes).

eval_statement(tree(statement,[For_loop]), Env, EnvRes):- eval_for_loop(For_loop, Env, EnvRes).

eval_statement(tree(statement,[While_loop]), Env, EnvRes):- eval_while_loop(While_loop, Env, EnvRes).

eval_statement(tree(statement,[If_condition]), Env, EnvRes):- eval_if_condition(If_condition, Env,
EnvRes).

eval_statement(tree(statement,[Ternary_operation]), Env, EnvRes):-
eval_ternary_operation(Ternary_operation, Env, EnvRes).


% Predicate for evaluating Print statement.

eval_print_statement(tree(print_statement,[_,_,Var,_]),Env,Env) :-

   eval_var(Var, Env, Var1),

   search(Var1, Env, X),

   write(X),nl.


% Predicate for evaluating Assignment statement.

eval_assignment_statement(tree(assignment_statement,[Declaration_stmt,_,Expr]),Env, EnvRes):-

   eval_declaration_statement(Declaration_stmt,Env, Env1),

   eval_expression(Expr,Env, Env2, Value),

   assign(Env1, Value, Env2, EnvRes).


eval_assignment_statement(tree(assignment_statement,[Var,_,Expr]),Env, EnvRes):-

   eval_var(Var,Env, Env1),

   eval_expression(Expr,Env, Env, Value),

   assign(Env1, Value, Env, EnvRes).


% Predicate for evaluating Declaration statement.

eval_declaration_statement(tree(declaration_statement,[_Data_type, Var]),Env, EnvRes):-

   eval_var(Var, Env, EnvRes).
```

```
% Predicate for evaluating for loop.
eval_for_loop(tree(for_loop,[token(_FOR,'for'),_,Assignment_stmt,_,Compare_operations,_,
Assignment_stmt_1,_,Block]), Env, EnvRes) :-
    eval_assignment_statement(Assignment_stmt, Env, Env1),
    eval_for_loop1(Compare_operations, Assignment_stmt_1, Block, Env1, EnvRes).

eval_for_loop1(Compare_operations, Assignment_stmt, Block, Env, EnvRes) :-
    eval_compare_operations(Compare_operations, Env, Env1, true),
    eval_block(Block, Env1, Env2),
    eval_assignment_statement(Assignment_stmt, Env2, Env3),
    eval_for_loop1(Compare_operations, Assignment_stmt, Block, Env3, EnvRes).
eval_for_loop1(Compare_operations, _Assignment_stmt, _Block, Env, Env) :-
    eval_compare_operations(Compare_operations, Env, Env, false).

eval_for_loop(tree(for_loop,[token(_FOR,'for'),_,Var,token(_IN,'in'),token(_RANGE,'range'),_,Number,_,N
umber_1,_,_,Block]),Env,EnvRes):-
    eval_var(Var,Env,Var1),
    eval_arithmetic_operation(Number,Env,Env1,Value),
    assign(Var1,Value,Env1,Env2),
    eval_for_loop2(Var1, Number_1, Block, Env2, EnvRes).

eval_for_loop2(Var, Number, Block, Env, EnvRes) :-
    search(Var, Env, Val1),
    eval_arithmetic_operation(Number, Env, _EnvT, Val2),
    Val1 < Val2,
    eval_block(Block, Env, Env1),
    Val3 is Val1 + 1,
    assign(Var, Val3, Env1, Env2),
```

```
    eval_for_loop2(Var, Number, Block, Env2, EnvRes).


eval_for_loop2(Var, Number, _, Env, EnvRes) :-
    search(Var, Env, Val1),
    eval_arithmetic_operation(Number, Env, EnvRes, Val2),
    Val1 >= Val2.


% Predicate for evaluating while loop.
eval_while_loop(tree(while_loop,[token(WHILE,'while'),_,Compare_operations,_,Block]), Env, EnvRes):-
    eval_compare_operations(Compare_operations, Env, Env1, true),
    eval_block(Block, Env1, Env2),
    eval_while_loop(tree(while_loop,[token(WHILE,'while'),_,Compare_operations,_,Block]),  Env2,
EnvRes).


eval_while_loop(tree(while_loop,[token(_WHILE,'while'),_,Compare_operations,_,_]), Env, Env):-
    eval_compare_operations(Compare_operations, Env, _Env1, false).


% Predicate for evaluating if block.


% only if true
eval_if_condition(tree(if_condition, [token(_IF,'if'),_,Compare_operations,_,Block]), Env, EnvRes) :-
    eval_compare_operations(Compare_operations, Env, _Env1, true),
    eval_block(Block, Env, EnvRes).


% if-else true
eval_if_condition(tree(if_condition, [token(_IF,'if'),_,Compare_operations,_,Block,token(_ELSE,'else'),_]),
Env, EnvRes) :-
    eval_compare_operations(Compare_operations, Env, _Env1, true),
    eval_block(Block, Env, EnvRes).
```

```
% if-else false
eval_if_condition(tree(if_condition, [token(_IF,'if'),_,Compare_operations,_,_,token(_ELSE,'else'),Block]), Env, EnvRes) :-
    eval_compare_operations(Compare_operations, Env, _Env1, false),
    eval_block(Block, Env, EnvRes).


% Predicate for evaluating expression.
eval_expression(tree(expression,[Boolean]), Env, EnvRes, Value):-
    eval_boolean(Boolean, Env, EnvRes, Value).

eval_expression(tree(expression,[String_literal]), Env, EnvRes, Value):-
    eval_string_literal(String_literal, Env, EnvRes, Value).

eval_expression(tree(expression,[Arithmetic_operation]), Env, EnvRes, Value):-
    eval_arithmetic_operation(Arithmetic_operation, Env, EnvRes, Value).


% Predicate used for evaluating the expression for addition and subtraction.
eval_arithmetic_operation(tree(arithmetic_operation,[Arithmetic_operation1,token(_ADDITION,'+'),Arithmetic_operation]), Env,Env, Answer):-
    eval_arithmetic_operation(Arithmetic_operation1, Env, Env1, Value1),
    eval_arithmetic_operation(Arithmetic_operation, Env, Env1, Value2),
    Answer is Value1 + Value2.

eval_arithmetic_operation(tree(arithmetic_operation,[Arithmetic_operation1,token(_SUBTRACTION,'-'),Arithmetic_operation]), Env,EnvRes, Answer):-
    eval_arithmetic_operation(Arithmetic_operation1, Env, EnvTemp, Value1),
    eval_arithmetic_operation(Arithmetic_operation, EnvTemp, EnvRes, Value2),
```

```
    Answer is Value1 - Value2.


eval_arithmetic_operation(tree(arithmetic_operation,[Arithmetic_operation1]), Env,EnvRes, Answer):-
    eval_arithmetic_operation(Arithmetic_operation1, Env, EnvRes, Value1),
    Answer is Value1.



% Predicate used for evaluating the expression for multiplication, division and percentage.
eval_arithmetic_operation(tree(arithmetic_operation1,[Arithmetic_operation2,token(_MULTIPLICATION,
'*'),Arithmetic_operation1]), Env,EnvRes, Answer):-
    eval_arithmetic_operation(Arithmetic_operation2, Env, EnvTemp, Value1),
    eval_arithmetic_operation(Arithmetic_operation1, EnvTemp, EnvRes, Value2),
    Answer is Value1 * Value2.

eval_arithmetic_operation(tree(arithmetic_operation1,[Arithmetic_operation2,token(_DIVISION,'/'),Arit
hmetic_operation1]), Env,EnvRes, Answer):-
    eval_arithmetic_operation(Arithmetic_operation2, Env, EnvTemp, Value1),
    eval_arithmetic_operation(Arithmetic_operation1, EnvTemp, EnvRes, Value2),
    Answer is Value1 / Value2.

eval_arithmetic_operation(tree(arithmetic_operation1,[Arithmetic_operation2,token(_PERCENTAGE,'%')
,Arithmetic_operation1]), Env,EnvRes, Answer):-
    eval_arithmetic_operation(Arithmetic_operation2, Env, EnvTemp, Value1),
    eval_arithmetic_operation(Arithmetic_operation1, EnvTemp, EnvRes, Value2),
    Answer is Value1 mod Value2.


eval_arithmetic_operation(tree(arithmetic_operation1,[Arithmetic_operation2]), Env,EnvRes, Answer):-
    eval_arithmetic_operation(Arithmetic_operation2, Env, EnvRes, Value1),
    Answer is Value1.
```

```
% Predicate used for evaluating arithmetic operation2.
eval_arithmetic_operation(tree(arithmetic_operation2,[_,Arithmetic_operation,_]), Env, EnvRes, Value):-
    eval_arithmetic_operation(Arithmetic_operation, Env, EnvRes, Val1),
    Value is Val1.

% Predicate used for evaluating number
eval_arithmetic_operation(tree(arithmetic_operation2, [token(_NUMBER,Val)]), Env,Env, Value):-
    atom_number(Val,Value).
eval_arithmetic_operation(tree(arithmetic_operation2, [Var]), Env, Env, Value):-
    eval_var(Var, Env, Var1),
    search(Var1, Env, Value).
eval_string_literal(tree(string_literal, [token(_,String)]), Env, Env, String).

% Predicate for compare_operations
eval_compare_operations(tree(compare_operations, [Compare_stmt, token(_LOGICAL_OPERATOR,
'and'), Compare_operations]), Env, EnvRes, true) :-
    eval_compare_statement(Compare_stmt, Env, Env1, true),
    eval_compare_operations(Compare_operations, Env1, EnvRes, true).

eval_compare_operations(tree(compare_operations, [Compare_stmt, token(_LOGICAL_OPERATOR,
'and'), Compare_operations]), Env, EnvRes, false) :-
    not(eval_compare_statement(Compare_stmt, Env, _Env1, true));
    (eval_compare_operations(Compare_operations, _Env1, EnvRes, false)).

% Logical OR operator
eval_compare_operations(tree(compare_operations, [Compare_stmt, token(_LOGICAL_OPERATOR, 'or'),
Compare_operations]), Env, EnvRes, true) :-
    (eval_compare_statement(Compare_stmt, Env, _Env1,true);
     eval_compare_operations(Compare_operations, _Env1, EnvRes,true)).
```

```
eval_compare_operations(tree(compare_operations, [Compare_stmt, token(_LOGICAL_OPERATOR, 'or'),
Compare_operations]), Env, EnvRes, false) :-
    not(eval_compare_statement(Compare_stmt, Env, Env1,true)),
    eval_compare_operations(Compare_operations, Env1, EnvRes,false).


% Predicate for single compare operation
eval_compare_operations(tree(compare_operations, [Compare_stmt]), Env, EnvRes, true) :-
    eval_compare_statement(Compare_stmt, Env, EnvRes,true).

eval_compare_operations(tree(compare_operations, [Compare_stmt]), Env, EnvRes, false) :-
    not(eval_compare_statement(Compare_stmt, Env, EnvRes,true)).


% Predicate used for evaluating compare statement.
eval_compare_statement(tree(compare_statement,
[Arithmetic_operation1,token(_COMPARISON_OPERATOR,'>'),Arithmetic_operation2]),Env, EnvRes,
true):-
    eval_arithmetic_operation(Arithmetic_operation1, Env, Env1, Value1),
    eval_arithmetic_operation(Arithmetic_operation2, Env1, EnvRes, Value2),
    Value1 > Value2.

eval_compare_statement(tree(compare_statement,
[Arithmetic_operation1,token(_COMPARISON_OPERATOR,'<'),Arithmetic_operation2]),Env, EnvRes,
true):-
    eval_arithmetic_operation(Arithmetic_operation1, Env, Env1, Value1),
    eval_arithmetic_operation(Arithmetic_operation2, Env1, EnvRes, Value2),
    Value1 < Value2.
```

```
eval_compare_statement(tree(compare_statement,
[Arithmetic_operation1,token(_COMPARISON_OPERATOR,'>='),Arithmetic_operation2]),Env, EnvRes,
true):-
    eval_arithmetic_operation(Arithmetic_operation1, Env, Env1, Value1),
    eval_arithmetic_operation(Arithmetic_operation2, Env1, EnvRes, Value2),
    Value1 >= Value2.


eval_compare_statement(tree(compare_statement,
[Arithmetic_operation1,token(_COMPARISON_OPERATOR,'<='),Arithmetic_operation2]),Env, EnvRes,
true):-
    eval_arithmetic_operation(Arithmetic_operation1, Env, Env1, Value1),
    eval_arithmetic_operation(Arithmetic_operation2, Env1, EnvRes, Value2),
    Value1 =< Value2.


eval_compare_statement(tree(compare_statement,
[Arithmetic_operation1,token(_COMPARISON_OPERATOR,'=='),Arithmetic_operation2]),Env, EnvRes,
true):-
    eval_arithmetic_operation(Arithmetic_operation1, Env, Env1, Value1),
    eval_arithmetic_operation(Arithmetic_operation2, Env1, EnvRes, Value2),
    Value1 = Value2.


eval_compare_statement(tree(compare_statement,
[Arithmetic_operation1,token(_COMPARISON_OPERATOR,'!='),Arithmetic_operation2]),Env, EnvRes,
true):-
    eval_arithmetic_operation(Arithmetic_operation1, Env, _Env1, Value1),
    eval_arithmetic_operation(Arithmetic_operation2, Env, EnvRes, Value2),
    Value1 \= Value2.



% Predicate used for evaluating compare statement.
eval_compare_statement(tree(compare_statement,[Var]),Env, EnvRes) :-
    eval_var(Var,Env, EnvRes).
```

```
eval_compare_statement(tree(compare_statement,[Number]),Env, EnvRes) :-
    eval_number(Number,Env, EnvRes).
eval_compare_statement(tree(compare_statement,[_,_Arithmetic_operation,_]),Env, EnvRes) :-
    eval_var(_Var,Env, EnvRes).

eval_compare_statement(tree(compare_statement,[token(_NOT,'not'), Compare_stmt]), Env, EnvRes,
true) :-
    not(eval_compare_statement(Compare_stmt, Env, EnvRes, true)).

eval_compare_statement(tree(compare_statement,[token(_NOT,'not'), Compare_stmt]), Env, EnvRes,
false) :-
    eval_compare_statement(Compare_stmt, Env, EnvRes, true).

% Predicate used for evaluating Ternary Operation.
eval_ternary_operation(tree(ternary_operation,[Compare_operations,_,Block,_,_]),Env,EnvRes):-
    eval_compare_operations(Compare_operations, Env, _EnvTemp, true),
    eval_block(Block, Env, EnvRes).

eval_ternary_operation(tree(ternary_operation,[Compare_operations,_,_,_,Block]),Env,EnvRes):-
    eval_compare_operations(Compare_operations, Env, _EnvTemp, false),
    eval_block(Block, Env, EnvRes).

%Predicate to evaluate boolean.
eval_boolean(tree(boolean,[token(_TRUE,'true')]),Env,Env,true).
eval_boolean(tree(boolean,[token(_FALSE,'false')]),Env,Env,false).

%Predicate to evaluate number and var
eval_number(tree(number,[Number]),_, Number).
eval_var(tree(var,[token(_VARIABLE_NAME, Var)]),_, Var).
```

```
% assign identifiers with values
assign(A, B, [], [(A, B)]).
assign(A, B, [(A, _)|C], [(A, B)|C]).
assign(A, B, [H|T], [H|U]) :-
   H \= (A, _),
   assign(A, B, T, U).

% lookup for a value in variable set
search(A, [(A,B)|_], B).
search(A, [_|B], C) :- search(A, B, C).
```

TEST PROGRAMS:

1. Program for even odd numbers

```
begin
{
int a=12;
str n="a is even number";
str m="a is odd number";
int c=a-2*(a/2);
if ( c == 0) {
print(n);
}
else {
print(m);
}
}
end
```

2. Program for Ternary Operator

```
begin
{
int a=5;
int b=6;
str s="greater number:";
int c=0;
a<b?{c=b;}:{c=a;}
print(s);
print(c);
}
End
```

3. Program for arithmetic operations

```
begin
{
int n=15;
int m=3;
int s=n+m;
int p=n-m;
int q=n*m;
int r=n/m;
int aa=n%m;
str a= "Addition:";
print (a);
print (s);
str b="Substraction:";
print (b);
print (p);
str c="Multiplication:";
print (c);
```

```
print (q);
str d="Division:";
print (d);
print (r);
bb="reminder";
print(bb);
print(aa);
}
End
```

4. Program for while loop

```
begin
{
int a=1;
int b=6;
while(a<=b)
{
print(a);
a=a+1;
}
}
end
```

5. Program for sum of numbers

```
begin
{
int n=5;
int sum=0;
for(int i=0;i<=n;i=i+1)
{
sum=sum+i;
```

```
}
str a="sum of first 5 numbers is";
print(a);
print(sum);
}
end
```

6. Program for “FOR IN RANGE” loop

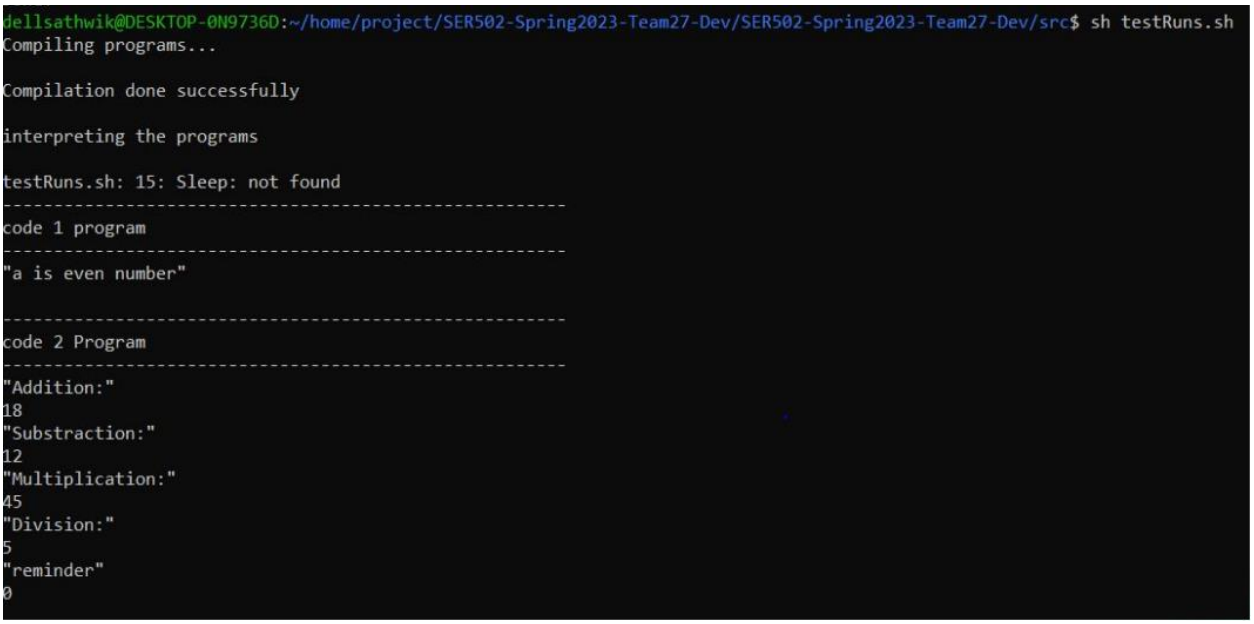
```
begin
{
int x=10;
int y=20;
int n=0;
for(i in range(x,y))
{
n=n+1;
}
str a="number of elements";
print(a);
print(n);
}
end
```

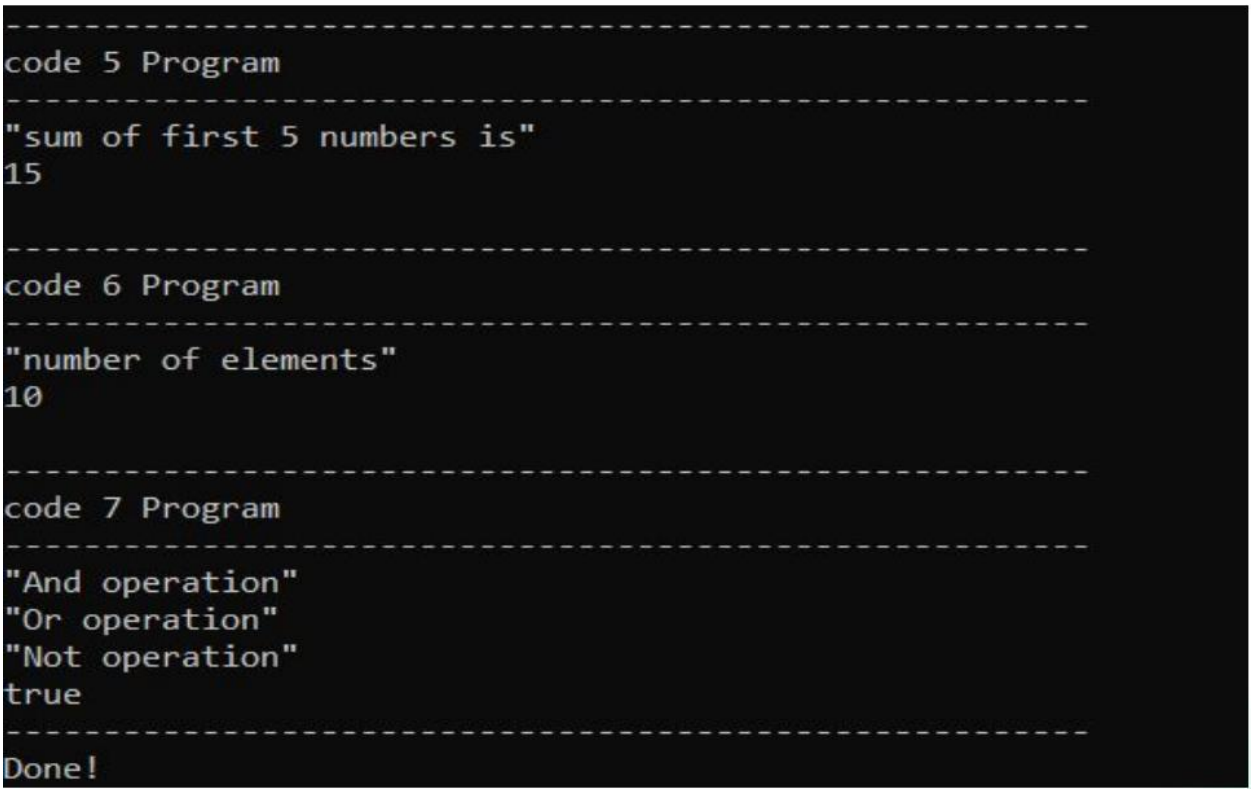
7. Program for “AND”, “OR” and “NOT” operations

```
begin
{
int x=5;
int y=10;
int z=0;
if(x > 0 && y > 0)
{
```

```
str a="And operation";

print(a);

}

if(x > 0 or y < 0)

{

str b="Or operation";

print(b);

}

if(not z != 0)

{

str c="Not operation";

print(c);

}

bool x=true;

print(x);

}

end
```

PROGRAM OUTPUT:





CONTRIBUTION:

Milestone1:

Srilakshmi Sravani Andaluri - Designed Grammer rules

Vamsi Krishna Somepalli - Designed grammer rules and designed overflow

Sathwik Reddy Dontham - Documentation and designed overflow

Saish Vemulapalli - Designed grammer rules and documentation

Rohith Reddy Byreddy - Designed Grammer rules

Milestone2:

Grammar: Sathwik Reddy, Sravani

Parser(Token Generation): Vamsi Somepalli, Rohith Reddy, Saish Vemulapalli

Evaluation: Sathwik Reddy, Sravani, Vamsi Somepalli, Rohith Reddy, Saish Vemulapalli

Testing: Vamsi Somepalli, Rohith Reddy

Shell Script: Sathwik Reddy, Rohith Reddy

Presentation: Saish Vemulapalli, Rohith Reddy

Documentation: Sravani Andaluri, Vamsi Somepalli

Overall Contribution:

Sathwik Reddy Dontham - Grammar Design, Initial Grammar, Added evals, Shell Script.

Vamsi Krishna Somepalli - Grammar Design, Grammar, Added evals, Testing, Documentation.

Rohith Reddy Byreddy - Grammar design, parse tree generation from files, added evals, test cases.

Srilakshmi Sravani Andaluri - Initial Grammar, added evals, parse tree evaluation, test cases, Documentation.

Saish Vemulapalli - Grammar, added evals, testing, presentation.

**********THE END**********