deepC Programming Language

Team 17:
Chandan Yadav
Nachiappan Lakshmanan
Nayan Jain
GuruPreetam Kadiri

Contents

- Introduction
- Implementation
- Intermediate Code Generation
- Runtime Environment

Introduction

deepC - is an imperative programming language in which the program describes a sequence of steps that change the state of the computer. Unlike many declarative programming languages deepC tells the computer "HOW" to accomplish things instead of "WHAT" to accomplish. deepC has all the basic characteristics, like many other high level imperative languages.

Design Goal

Simple - The language must avoid all unnecessary complexity and improve developers efficiency of solving the problem.

Clean - The language must provide clean, reliable, consistent applications which can be easily read and understood. It is very important that when another person reads the code he must be able to understand the syntactic structure of the code and visualise the semantics of the code.

Good performance - The language must provide good performance in terms of time and space.

Modularity - The language must help break the programs to distinct components, which can later combined to solve other problems.

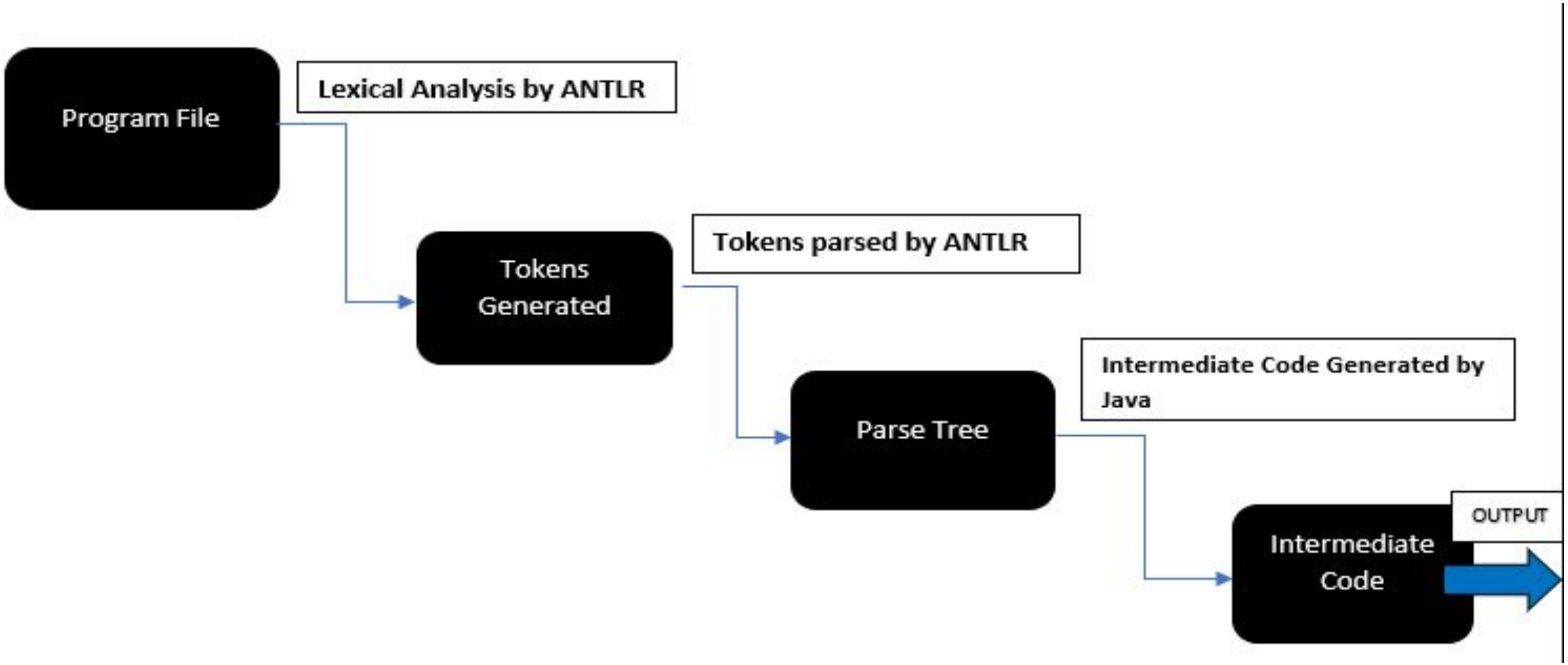
Tech Stack

ANTLR - **ANother Tool for Language Recognition** is used for lexical analysis and parse tree generation

Java 8 - Business Logic for intermediate code generation and runtime execution is written in java

Maven- jar file generation and ANTLR dependency in pom file

Implementation



Program File

Lexical Analysis by ANTLR

Tokens Generated

Tokens parsed by ANTLR

Parse Tree

Intermediate Code Generated by Java

Intermediate Code

OUTPUT

Data Structures

Stack - A recursive data structure used in runtime environment for expression evaluation.

Hash Map - A hash map is used to store the current state of each variable in runtime environment

Array List - An array list is used as a storage during intermediate code generation

Steps to execute deepC

1. Download the repository
2. Navigate to the target folder
3. Enter the command java -jar deepcdemo-0.0.1-SNAPSHOT-jar-with-dependencies path/filename.dpc
4. The above command will generate the intermediate code with the .idpc extension as the same name as the sample program
5. You can find the intermediate code generated in the target/tests folder

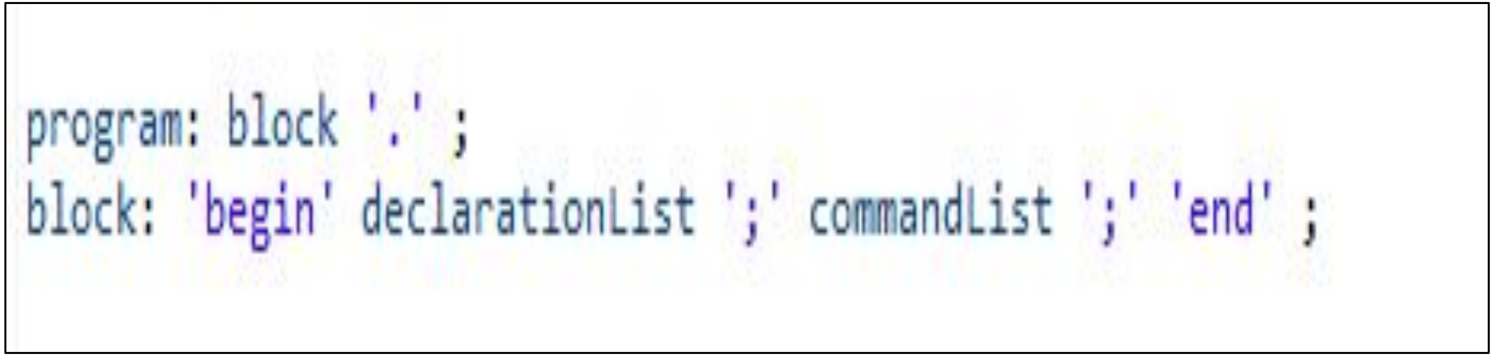
Language design

Every program should start with a key word "**begin**" and end with a keyword “**end**” followed by a “.“ (dot), signifying the start and end of the program respectively.

Block can have a list of declarations followed by a list of commands.

Every declaration ends with a “;” semicolon. Consecutive commands are separated by “;” .

Grammar: Initial point of program execution



```
program: block '.' ;
block: 'begin' declarationList ';' commandList ';' 'end' ;
```

Declaration

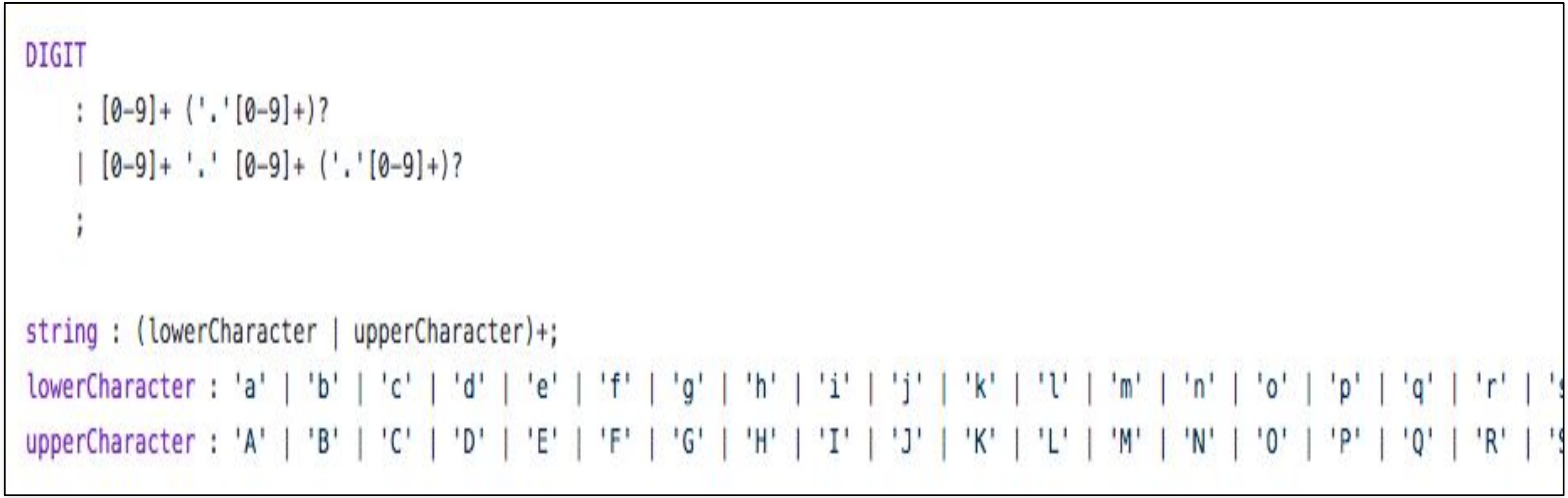
Declaration section of the language supports declaration for three data types:

integer - a data type that stores an integer (numeric type)

booleanValue - a data type that stores boolean values (true, false)

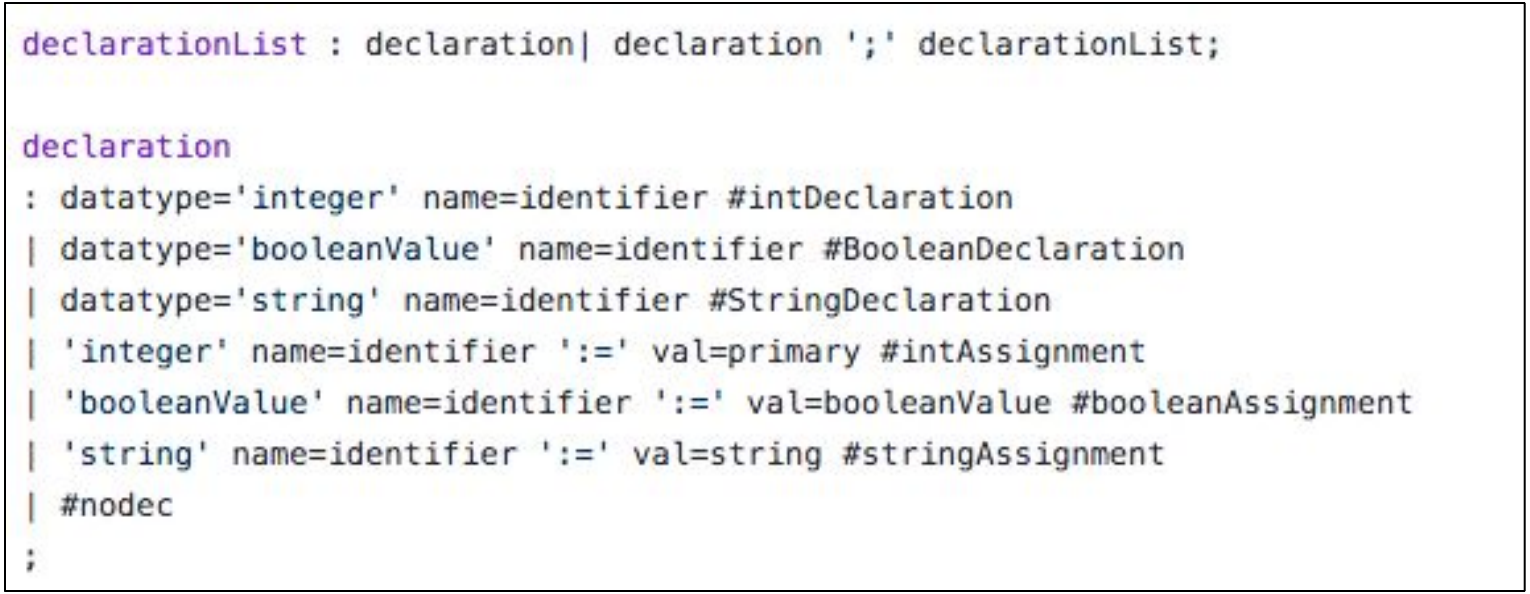
string - a data type that stores sequence of characters (includes lower and upper case). (string value assignments to variables can also be done)

Grammar - Integer, boolean & String



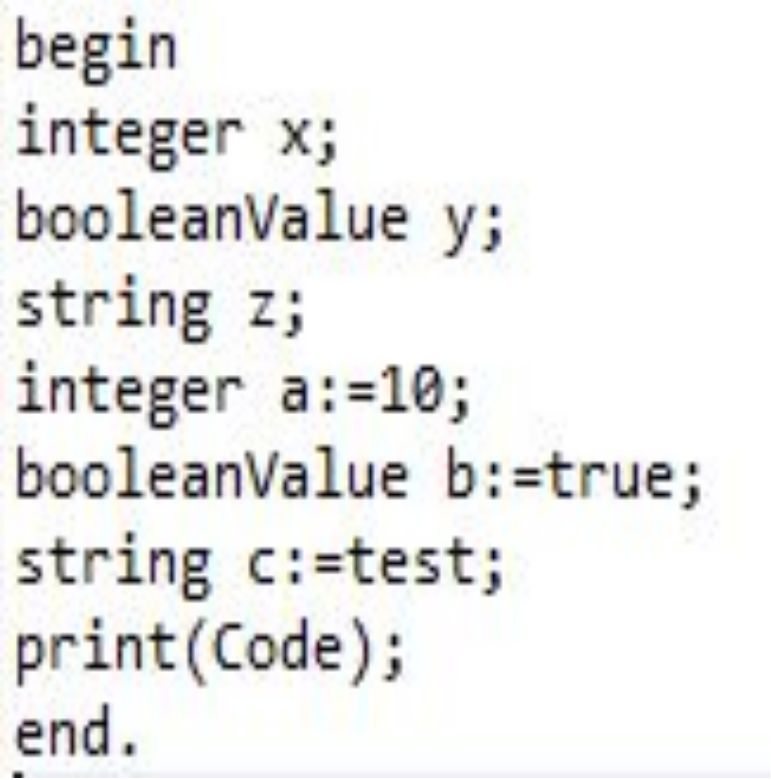
```
DIGIT
    : [0-9]+ ('.'[0-9]+)?
    | [0-9]+ '.' [0-9]+ ('.'[0-9]+)?
    ;


string : (lowerCharacter | upperCharacter)+;
lowerCharacter : 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | '
upperCharacter : 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J' | 'K' | 'L' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R' | '
```

contd



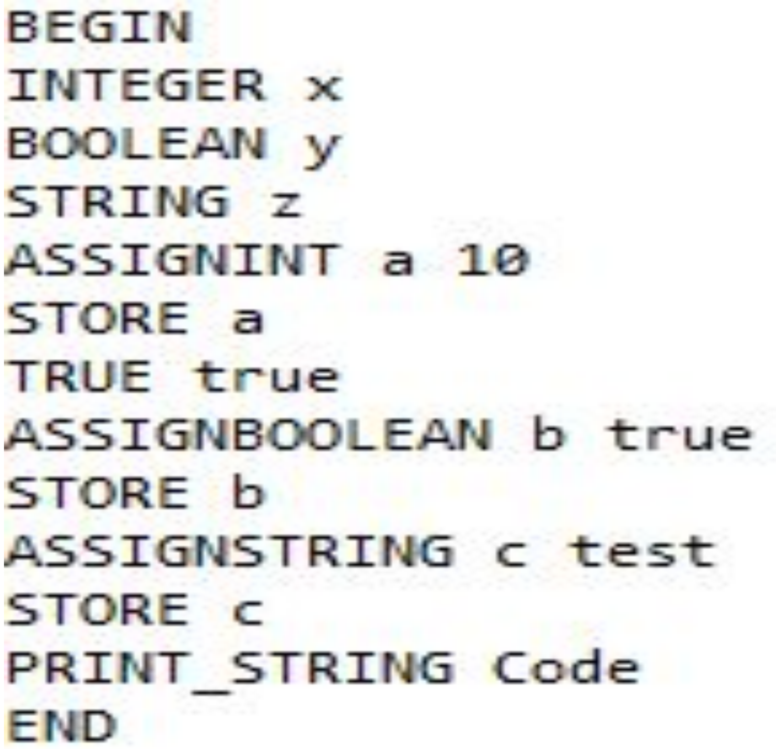
```
declarationList : declaration| declaration ';' declarationList;

declaration
: datatype='integer' name=identifier #intDeclaration
| datatype='booleanValue' name=identifier #BooleanDeclaration
| datatype='string' name=identifier #StringDeclaration
| 'integer' name=identifier ':=' val=primary #intAssignment
| 'booleanValue' name=identifier ':=' val=booleanValue #booleanAssignment
| 'string' name=identifier ':=' val=string #stringAssignment
| #nodec
;
```

Sample Code



```
begin
integer x;
booleanValue y;
string z;
integer a:=10;
booleanValue b:=true;
string c:=test;
print(Code);
end.
```

Intermediate Code



```
BEGIN
INTEGER x
BOOLEAN y
STRING z
ASSIGNINT a 10
STORE a
TRUE true
ASSIGNBOOLEAN b true
STORE b
ASSIGNSTRING c test
STORE c
PRINT_STRING Code
END
```

Command List

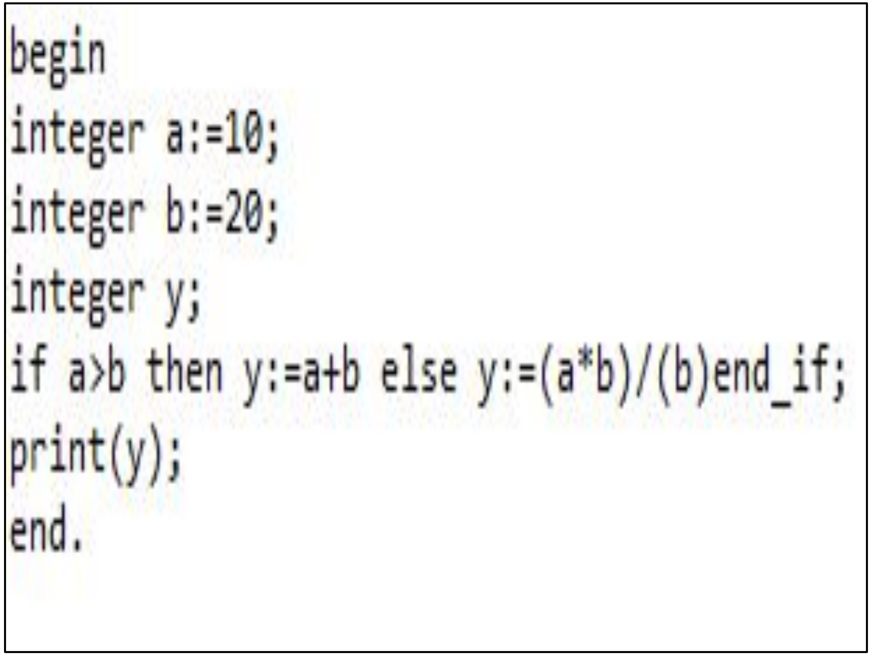
Command list in deepC is a combination of the following types

- If then else construct
- Ternary evaluator
- while condition
- Traditional for loop construct
- for loop in the given range
- Initializer
- Print statement
- End condition

if then else construct

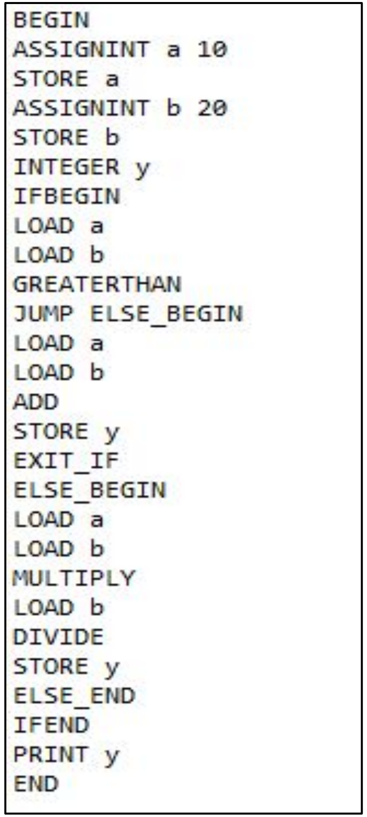
- Begins with a an “if" followed by a booleanValueList, “then", commandList, "else", commandList, ”end_if”.
- booleanValueList will also accept a sequence of “and”, “or”, “not".
  for example(x<7 and y> 6 or z>8)
- The core functionality of the booleanValueList is to return true / false for the condition followed by “if" keyword. Validates the condition to true/false.
- The intermediate code bookmarks the code with labels for the runtime to execute conditionally.

Sample Code



```
begin
integer a:=10;
integer b:=20;
integer y;
if a>b then y:=a+b else y:=(a*b)/(b)end_if;
print(y);
end.
```

Intermediate Code



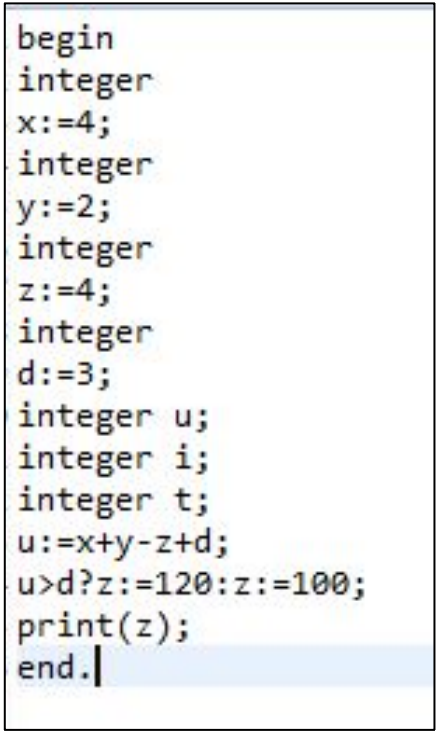
```
BEGIN
ASSIGNINT a 10
STORE a
ASSIGNINT b 20
STORE b
INTEGER y
IFBEGIN
LOAD a
LOAD b
GREATERTHAN
JUMP ELSE_BEGIN
LOAD a
LOAD b
ADD
STORE y
EXIT_IF
ELSE_BEGIN
LOAD a
LOAD b
MULTIPLY
LOAD b
DIVIDE
STORE y
ELSE_END
IFEND
PRINT y
END
```

Ternary construct

begins with an “if”, booleanValueList, ”?”, (booleanValue “:" booleanValue or initialise “:" initialise)

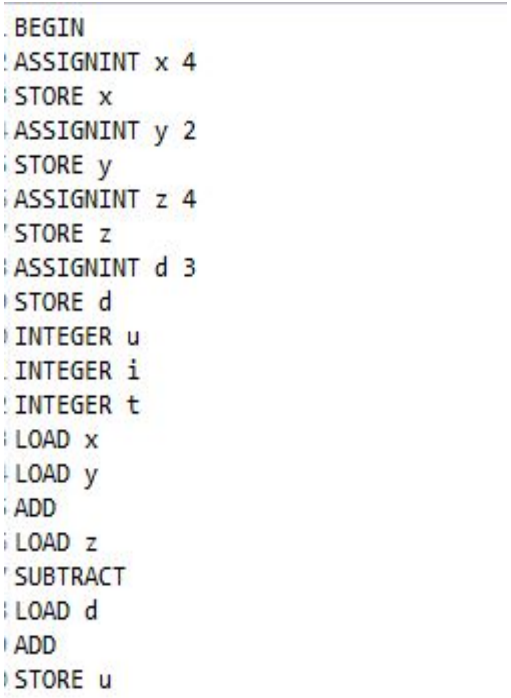
based upon the validation condition, if it returns true then command initialise or booleanValue to the left of the “:” will execute or to the right will execute.

Sample Code

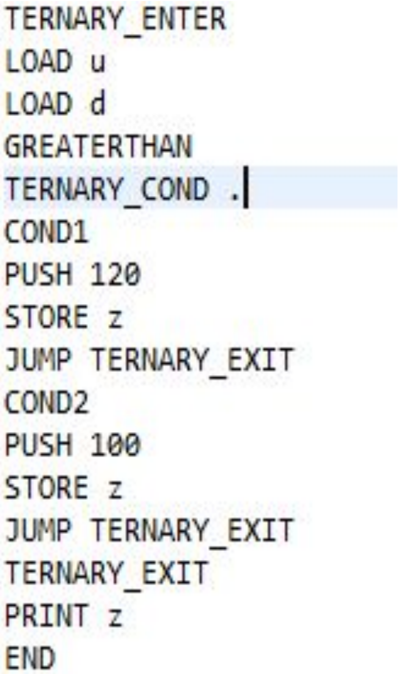


```
begin
integer
x:=4;
integer
y:=2;
integer
z:=4;
integer
d:=3;
integer u;
integer i;
integer t;
u:=x+y-z+d;
u>d?z:=120:z:=100;
print(z);
end.
```

Intermediate Code



```
BEGIN
ASSIGNINT x 4
STORE x
ASSIGNINT y 2
STORE y
ASSIGNINT z 4
STORE z
ASSIGNINT d 3
STORE d
INTEGER u
INTEGER i
INTEGER t
LOAD x
LOAD y
ADD
LOAD z
SUBTRACT
LOAD d
ADD
STORE u
```



```
TERNARY_ENTER
LOAD u
LOAD d
GREATERTHAN
TERNARY_COND .
COND1
PUSH 120
STORE z
JUMP TERNARY_EXIT
COND2
PUSH 100
STORE z
JUMP TERNARY_EXIT
TERNARY_EXIT
PRINT z
END
```

Looping constructs - traditional FOR loop

A for-loop (or simply for loop) is a control flow statement for specifying iteration, which allows code to be executed repeatedly.

Traditional for loop:- begins with a “for”, ”(“, initilaisation, booleanValueList, endCondition , ”)”, “do”, commandList, “end_for”.

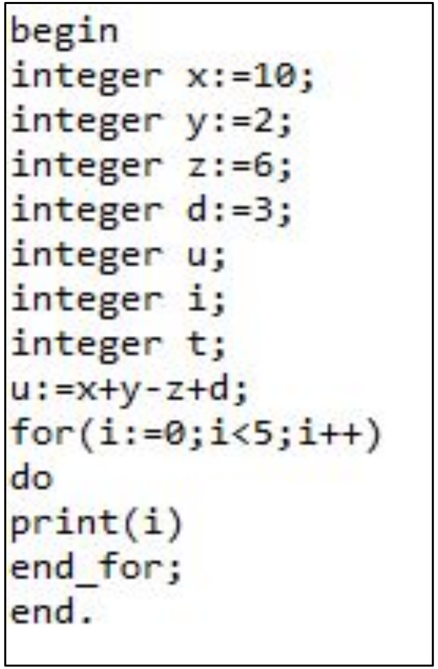
Initialisation is a part of command List operation. Initialise a variable with numeric type.(This is a way to associate a value to the identifier)

contd

booleanValueList to validate the for condition each time, and end condition which tells the order in which the initialised variable should increment.

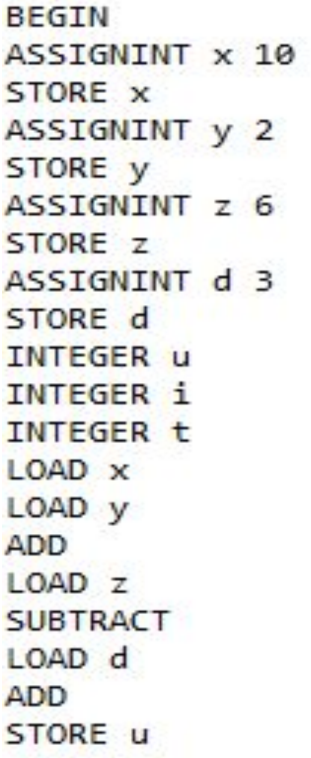
End condition can be either a “++” | "--" operator or incrementing the variable by a particular value “x:=x+1” or decrementing it "x=x-1"

Sample Code

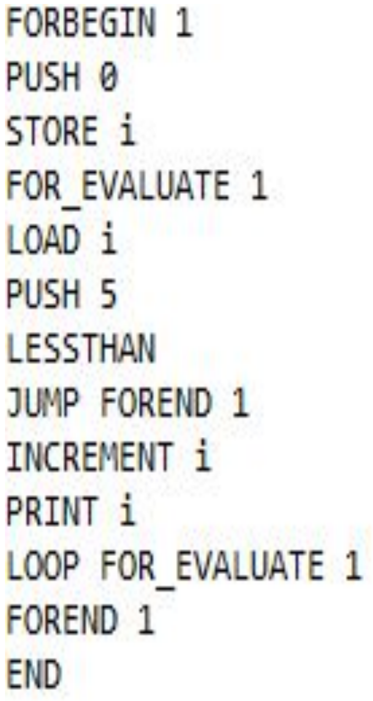


```
begin
integer x:=10;
integer y:=2;
integer z:=6;
integer d:=3;
integer u;
integer i;
integer t;
u:=x+y-z+d;
for(i:=0;i<5;i++)
do
print(i)
end_for;
end.
```

Intermediate Code



```
BEGIN
ASSIGNINT x 10
STORE x
ASSIGNINT y 2
STORE y
ASSIGNINT z 6
STORE z
ASSIGNINT d 3
STORE d
INTEGER u
INTEGER i
INTEGER t
LOAD x
LOAD y
ADD
LOAD z
SUBTRACT
LOAD d
ADD
STORE u
```



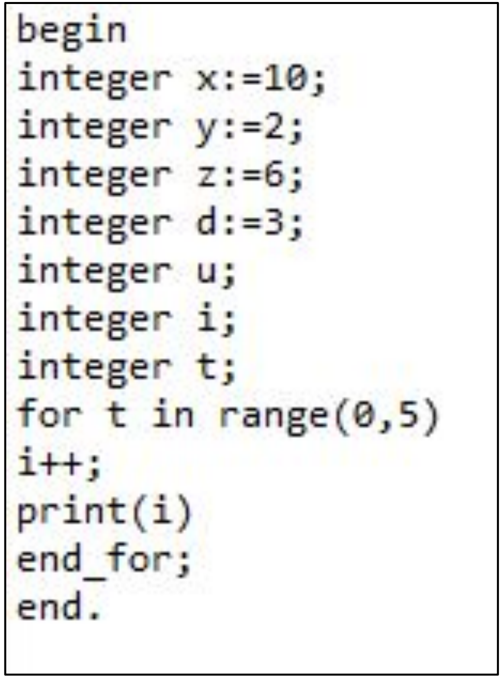
```
FORBEGIN 1
PUSH 0
STORE i
FOR_EVALUATE 1
LOAD i
PUSH 5
LESSTHAN
JUMP FOREND 1
INCREMENT i
PRINT i
LOOP FOR_EVALUATE 1
FOREND 1
END
```

Python type FOR loop

Python type For Loop:- begins with a “for" identifier” ""in" "range" between two numbers. It will do a list of commands.

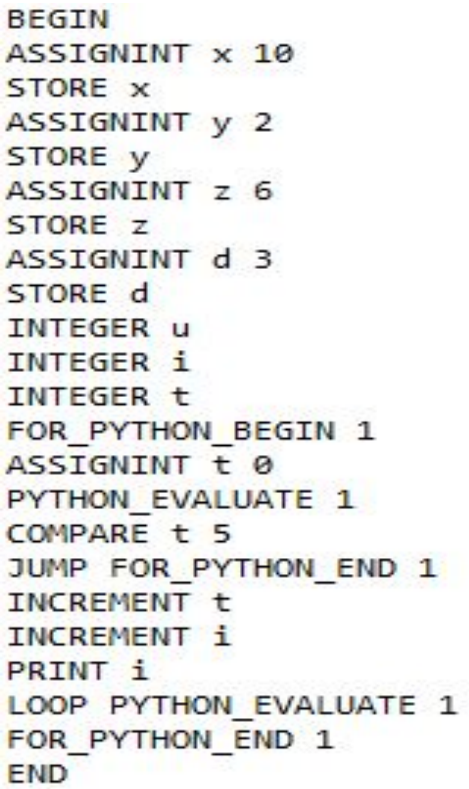
Very much similar to the tradition for loop except for the fact that it has a different syntactic structure.

Sample Code



```
begin
integer x:=10;
integer y:=2;
integer z:=6;
integer d:=3;
integer u;
integer i;
integer t;
for t in range(0,5)
i++;
print(i)
end_for;
end.
```

Intermediate Code



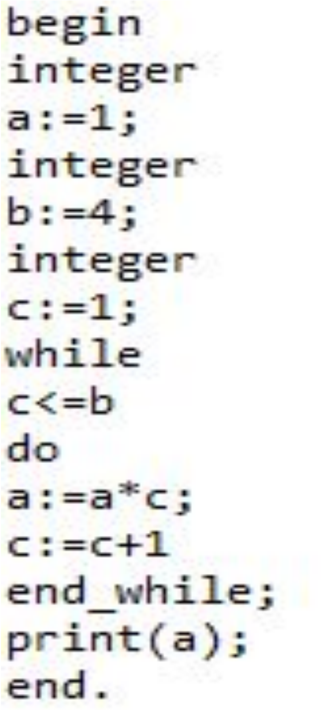
```
BEGIN
ASSIGNINT x 10
STORE x
ASSIGNINT y 2
STORE y
ASSIGNINT z 6
STORE z
ASSIGNINT d 3
STORE d
INTEGER u
INTEGER i
INTEGER t
FOR_PYTHON_BEGIN 1
ASSIGNINT t 0
PYTHON_EVALUATE 1
COMPARE t 5
JUMP FOR_PYTHON_END 1
INCREMENT t
INCREMENT i
PRINT i
LOOP PYTHON_EVALUATE 1
FOR_PYTHON_END 1
END
```

While Loop

A while loop is a control flow statement that allows code to be executed repeatedly based on a given Boolean condition

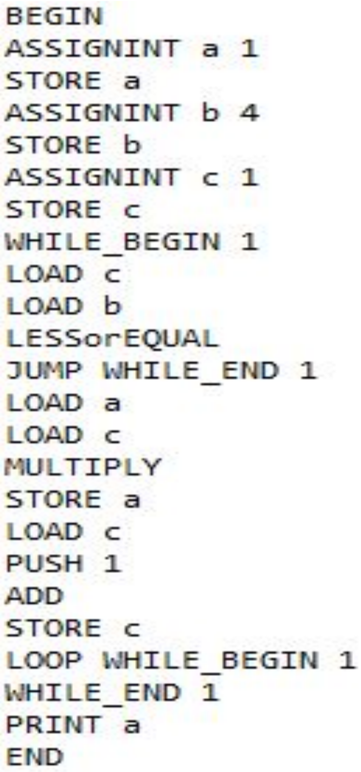
Begins with a 'while' while_condition 'do' commandList ‘end_while'
while_condition is the booleanValueList which validates the condition to true / false.

Sample Code



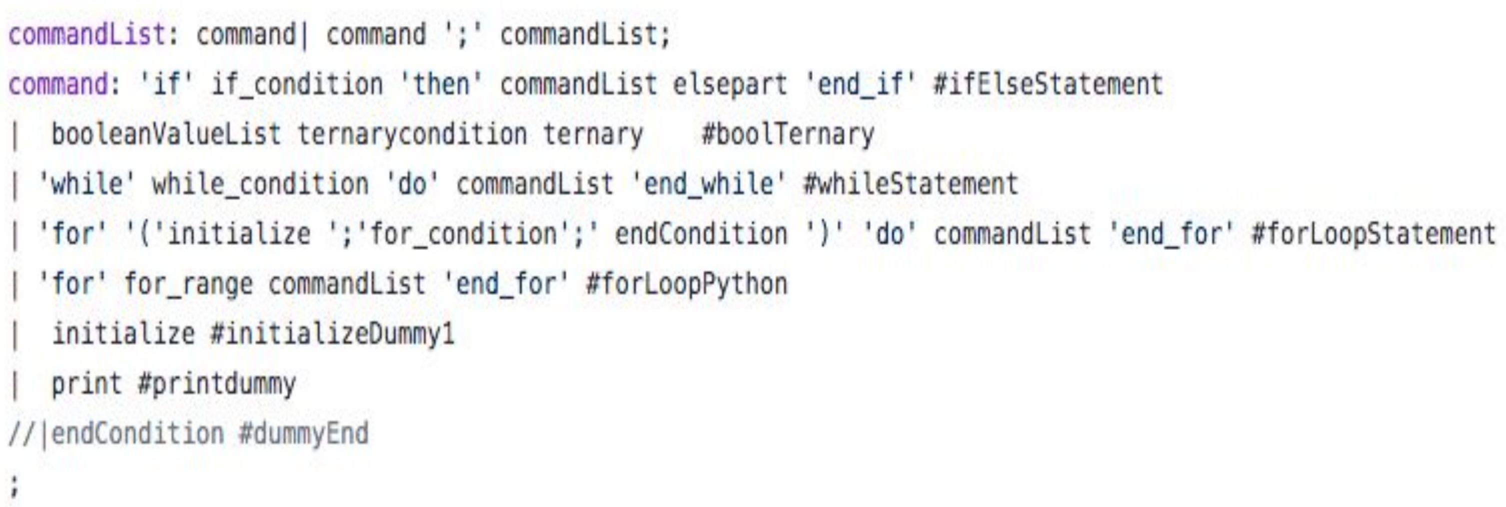
```
begin
integer
a:=1;
integer
b:=4;
integer
c:=1;
while
c<=b
do
a:=a*c;
c:=c+1
end_while;
print(a);
end.
```

Intermediate Code



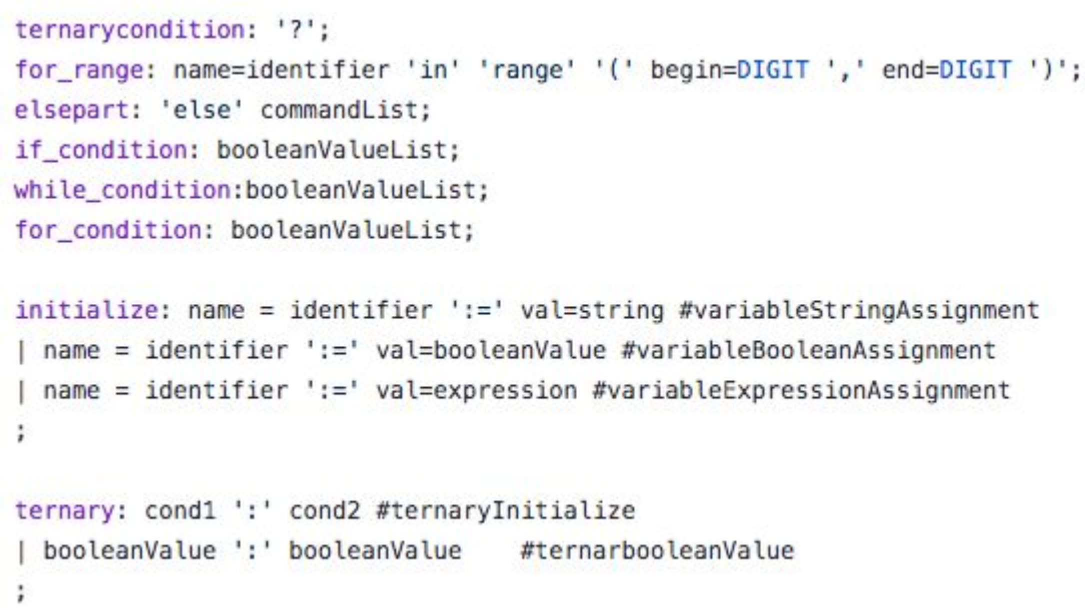
```
BEGIN
ASSIGNINT a 1
STORE a
ASSIGNINT b 4
STORE b
ASSIGNINT c 1
STORE c
WHILE_BEGIN 1
LOAD c
LOAD b
LESSorEQUAL
JUMP WHILE_END 1
LOAD a
LOAD c
MULTIPLY
STORE a
LOAD c
PUSH 1
ADD
STORE c
LOOP WHILE_BEGIN 1
WHILE_END 1
PRINT a
END
```

Grammar-

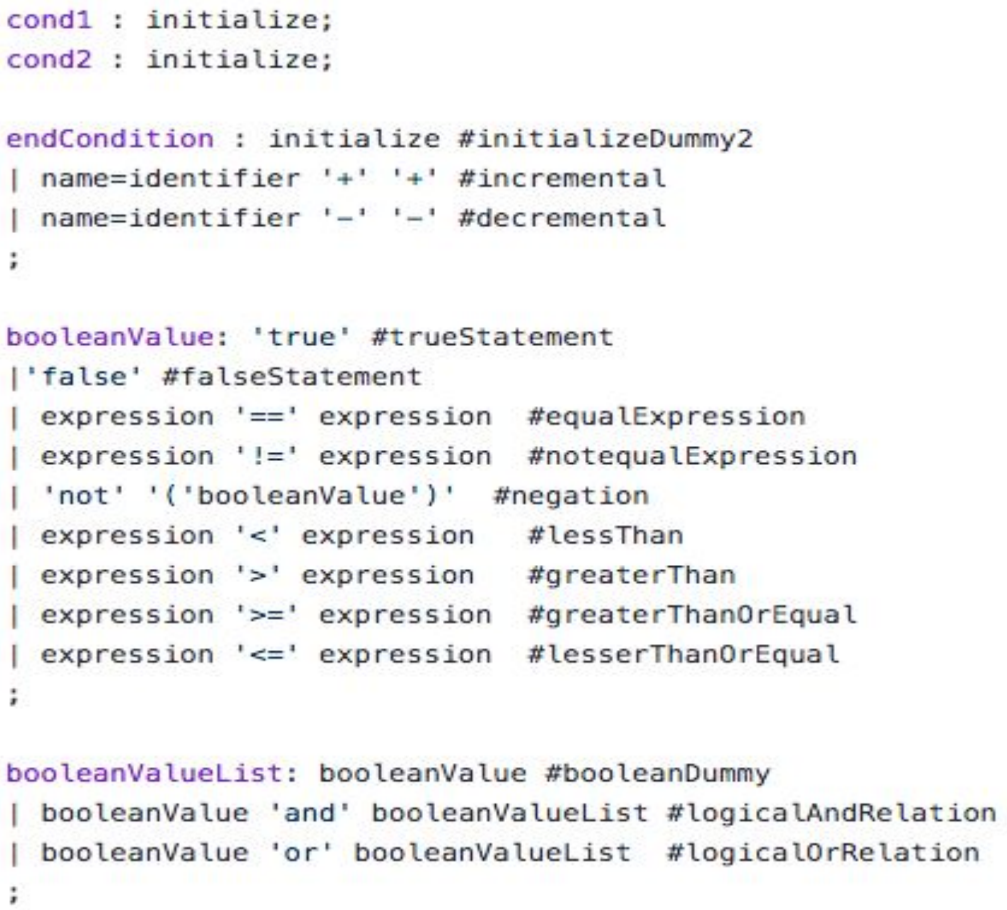


```
commandList: command| command ';' commandList;
command: 'if' if_condition 'then' commandList elsepart 'end_if' #ifElseStatement
|  booleanValueList ternarycondition ternary    #boolTernary
| 'while' while_condition 'do' commandList 'end_while' #whileStatement
| 'for' '('initialize ';'for_condition';' endCondition ')' 'do' commandList 'end_for' #forLoopStatement
| 'for' for_range commandList 'end_for' #forLoopPython
|  initialize #initializeDummy1
|  print #printdummy
//|endCondition #dummyEnd
;
```

contd



```
ternarycondition: '?';
for_range: name=identifier 'in' 'range' '(' begin=DIGIT ',' end=DIGIT ')';
elsepart: 'else' commandList;
if_condition: booleanValueList;
while_condition:booleanValueList;
for_condition: booleanValueList;

initialize: name = identifier ':=' val=string #variableStringAssignment
| name = identifier ':=' val=booleanValue #variableBooleanAssignment
| name = identifier ':=' val=expression #variableExpressionAssignment
;

ternary: cond1 ':' cond2 #ternaryInitialize
| booleanValue ':' booleanValue    #ternarbooleanValue
;
```

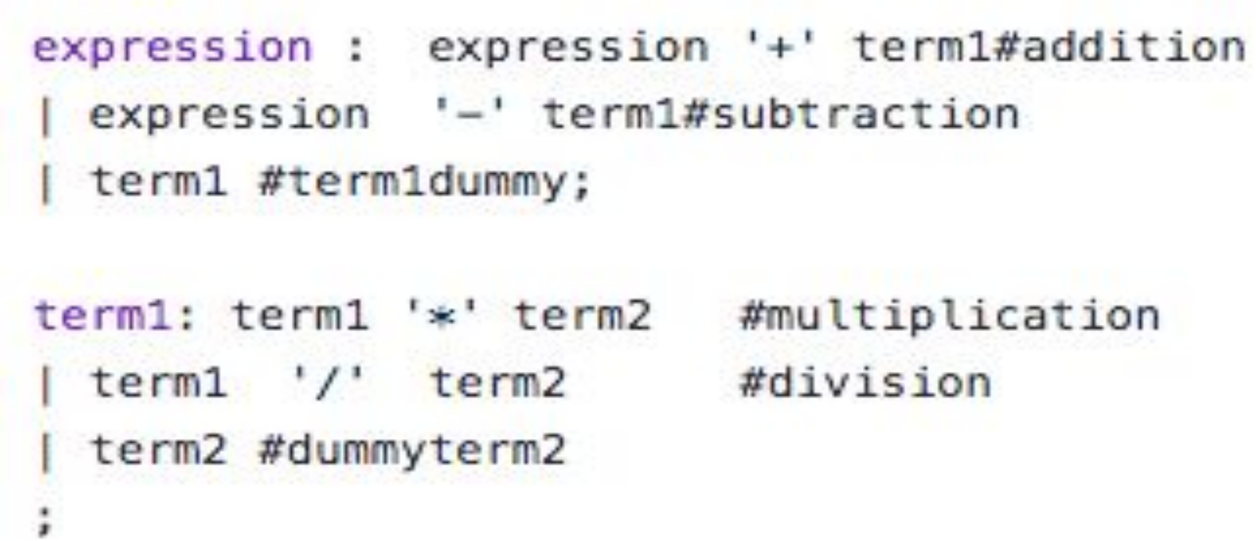


```
cond1 : initialize;
cond2 : initialize;

endCondition : initialize #initializeDummy2
| name=identifier '+' '+' #incremental
| name=identifier '-' '-' #decremental
;

booleanValue: 'true' #trueStatement
|'false' #falseStatement
| expression '==' expression  #equalExpression
| expression '!=' expression  #notequalExpression
| 'not' '('booleanValue')'  #negation
| expression '<' expression   #lessThan
| expression '>' expression   #greaterThan
| expression '>=' expression  #greaterThanOrEqual
| expression '<=' expression  #lesserThanOrEqual
;

booleanValueList: booleanValue #booleanDummy
| booleanValue 'and' booleanValueList #logicalAndRelation
| booleanValue 'or' booleanValueList  #logicalOrRelation
;
```

Expression evaluation

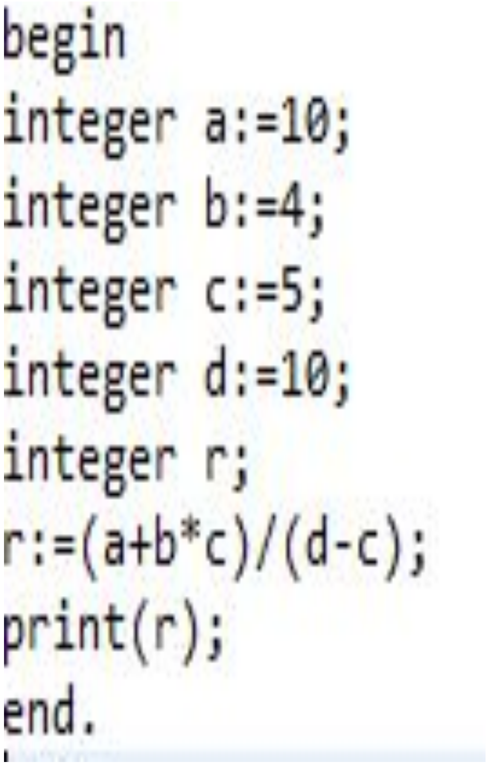
To evaluate an algebraic **expression** means to find the **value** of the **expression** when the variable is replaced by a given number. To evaluate an **expression**, we substitute the given number for the variable in the **expression** and then simplify the **expression** using the order of operations.

Expression Evaluation: Grammar



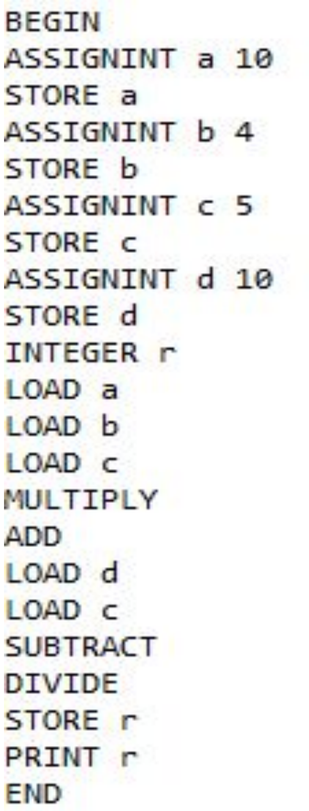
```
expression :  expression '+' term1#addition
| expression  '-' term1#subtraction
| term1 #term1dummy;

term1: term1 '*' term2   #multiplication
| term1  '/'  term2      #division
| term2 #dummyterm2
;
```

Sample Code



```
begin
integer a:=10;
integer b:=4;
integer c:=5;
integer d:=10;
integer r;
r:=(a+b*c)/(d-c);
print(r);
end.
```

Intermediate Code



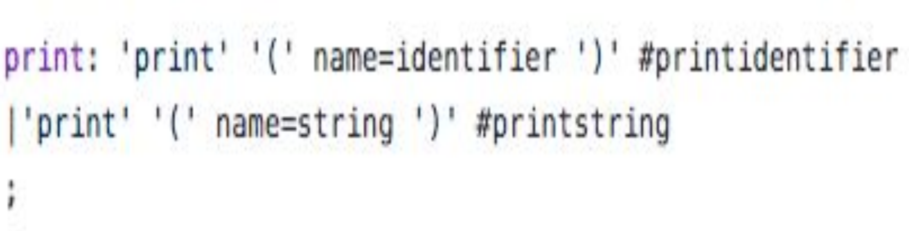
```
BEGIN
ASSIGNINT a 10
STORE a
ASSIGNINT b 4
STORE b
ASSIGNINT c 5
STORE c
ASSIGNINT d 10
STORE d
INTEGER r
LOAD a
LOAD b
LOAD c
MULTIPLY
ADD
LOAD d
LOAD c
SUBTRACT
DIVIDE
STORE r
PRINT r
END
```

Result

Result: 6

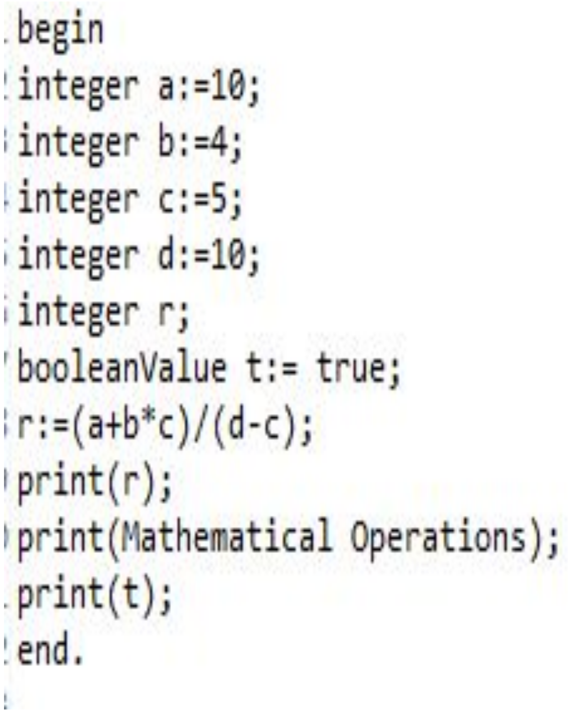
Print

It is a construct for printing all the identifier values. It supports all the datatypes supported by our language.



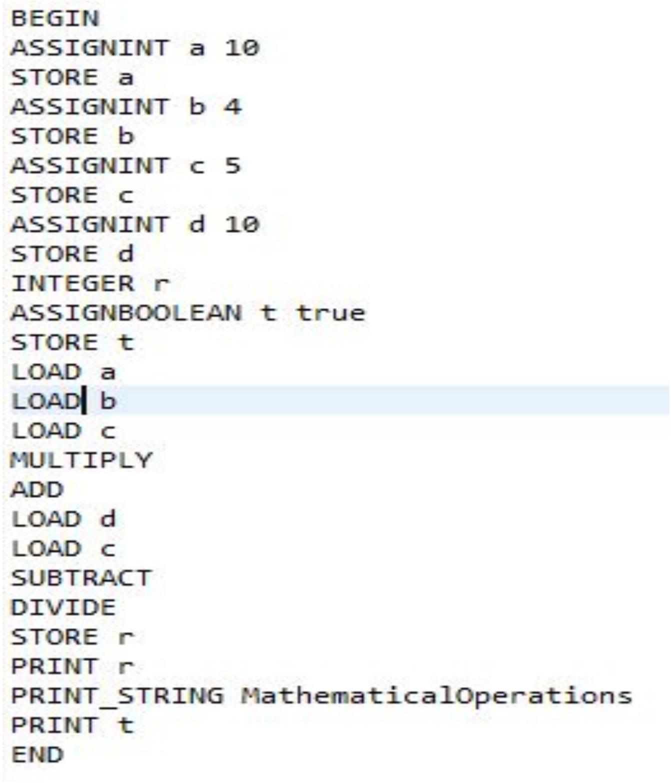
```
print: 'print' '(' name=identifier ')' #printidentifier
|'print' '(' name=string ')' #printstring
;
```

Sample Code



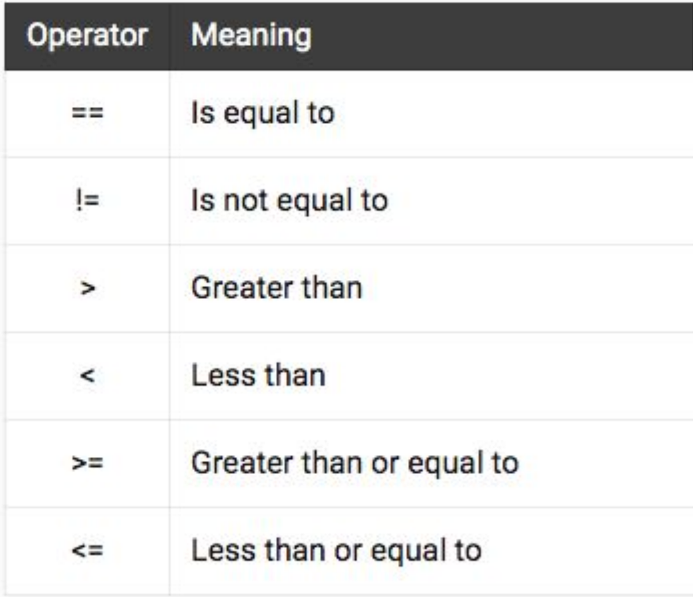
```
begin
integer a:=10;
integer b:=4;
integer c:=5;
integer d:=10;
integer r;
booleanValue t:= true;
r:=(a+b*c)/(d-c);
print(r);
print(Mathematical Operations);
print(t);
end.
```

Intermediate Code



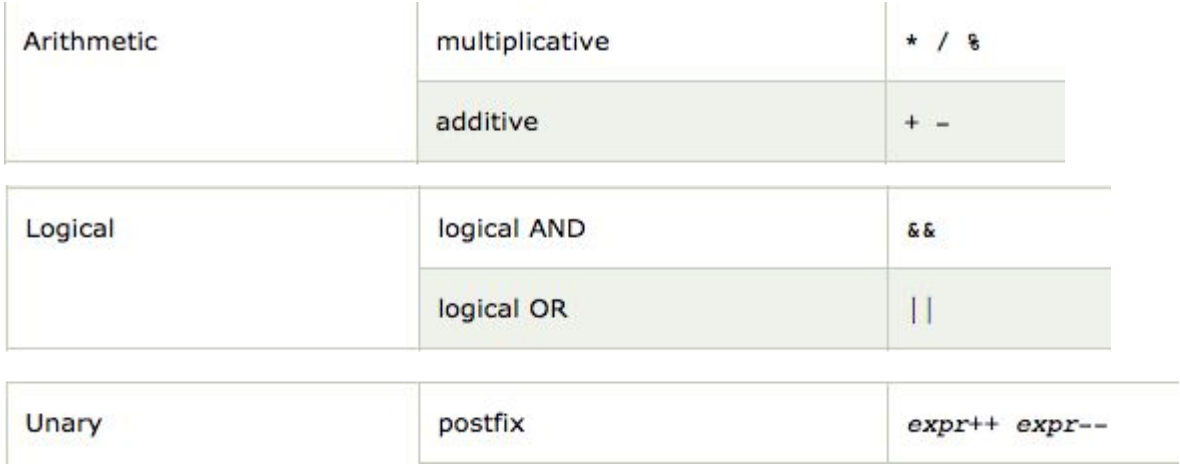
```
BEGIN
ASSIGNINT a 10
STORE a
ASSIGNINT b 4
STORE b
ASSIGNINT c 5
STORE c
ASSIGNINT d 10
STORE d
INTEGER r
ASSIGNBOOLEAN t true
STORE t
LOAD a
LOAD b
LOAD c
MULTIPLY
ADD
LOAD d
LOAD c
SUBTRACT
DIVIDE
STORE r
PRINT r
PRINT_STRING MathematicalOperations
PRINT t
END
```

Operators used - Relational operators



Operator	Meaning
==	Is equal to
!=	Is not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

Arithmetic, Logical, Unary operators



Arithmetic	multiplicative	`* / %`
	additive	`+ -`
Logical	logical AND	`&&`
	logical OR	`||`
Unary	postfix	*expr*`++` *expr*`--`

Implementation - First step

LEXICAL ANALYSIS is the very first phase our compiler design. A lexer takes the modified source code which is written in the form of sentences . In other words, it helps you to convert a sequence of characters into a sequence of tokens. The lexical analyzer break the syntax into a series of tokens.

```
ANTLRFileStream inputfile = new ANTLRFileStream(program);
```

contd

program is the source code written in deepC which will will be available in the file test.smp.

Code:-

```
simpleLexer lexer = new simpleLexer(inputfile);
CommonTokenStream tokens = new CommonTokenStream(lexer);
```

Second step - Parsing

In the syntax analysis phase, a compiler verifies whether or not the tokens generated by the lexical analyzer are grouped according to the syntactic rules of the language. This is done by a parser. 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 will generate a parse tree to verify the source code written in deepC.

```
simpleParser parser = new simpleParser(tokens);

ProgramContext tree = parser.program();
```

Intermediate Code generation

We will generate the intermediate code by giving semantics to the parse tree in java (ANTLR gives override functions for the grammar written).

Runtime environment will interpret the intermediate code and produce the output. This is also written in java.

Intermediate code generation definition

ASSIGNINT : Assigns an integer value to a variable

STORE: Stores the value to a variable from the top of the stack

INTEGER: Declares a variable as integer

LOAD : Loads the value of the variable on top of the stack

INCREMENT: Increments the value of a variable

DECREMENT : decrements the value of a variable

MULTIPLY: pops two values from the stack and multiplies them and pushes the result on top of the stack.

ADD: pops two values from the stack and adds them and pushes the result on top of the stack.

SUBTRACT: pops two values from the stack and subtracts them and pushes the result on top of the stack.

Intermediate code generation definition

DIVIDE: pops two values from the stack and divides them and pushes the result on top of the stack.

PRINT: prints the value of a variable.

PRINT_STRING: prints the value of a string.

EQUAL : pops the last two values in a stack and checks if they are equal, if equal it pushes true to the booleanStack, else pushes false to the booleanStack

NOTEQUAL:pops the last two values in a stack and checks if they are not equal, if not equal it pushes true to the booleanStack, else pushes false to the booleanStack

NEGATE: pops the last element from the stack and negates the value of the variable.

AND: pops the last two values in a stack and performs AND operation on them

OR: pops the last two values in a stack and performs OR operation on them

JUMP: jumps the instruction reader to the next label

GREATERTHAN: pops the last two values in a stack and checks if first is greater than second

Intermediate code generation definition

GREATERorEQUAL: pops the last two values in a stack and checks if first is greater than equal to second, if greater than or equal pushes true to the booleanStack else false.

LESSorEQUAL:pops the last two values in a stack and checks if first is less than or equal to second, if less than or equal pushes true to the booleanStack else false.

LESSTHAN: pops the last two values in a stack and checks if first is less than second, if less than pushes true to the booleanStack else false.

PUSH: pushes the variable or data to the stack

FOREND: label determining the end of for loop.

FOR_PYTHON_END: marks the beginning of for in range(DIGIT,DIGIT).

WHILE_BEGIN: marks the beginning of while loop.

WHILE_END: marks the end of while loop.

ELSE_END: marks the end of else..

Intermediate code generation definition

ELSE_BEGIN: marks the beginning of the else.

IFBEGIN: marks the beginning of if condition.

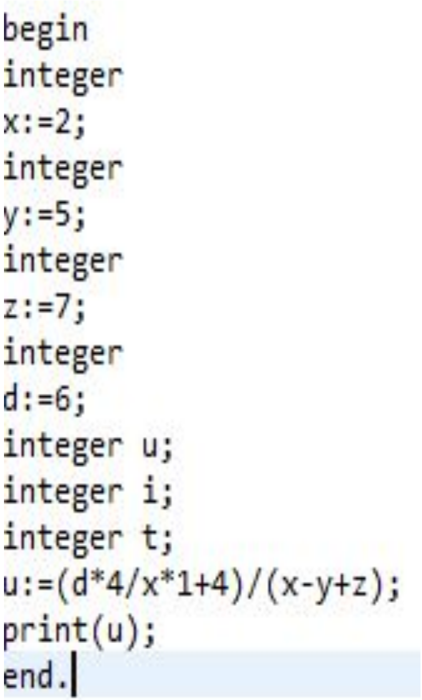
IFEND: marks the end of if condition

LOOP: indicates that the program has to redirect to the location of the label to start looping.

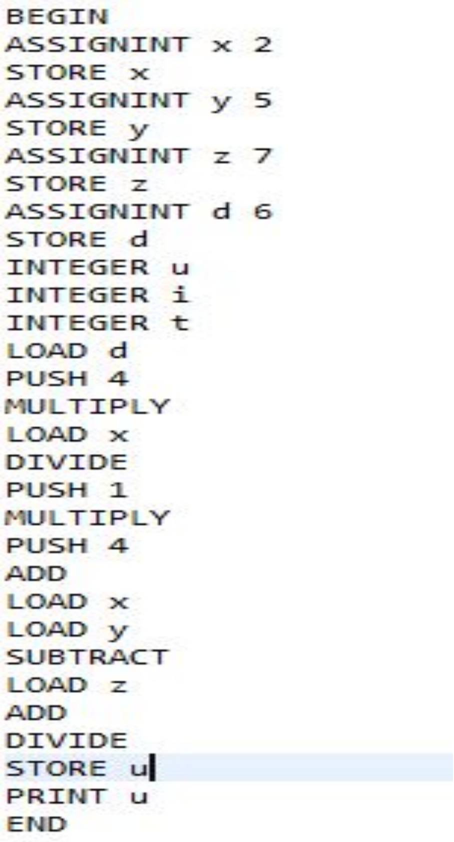
COMPARE: pops the last two values in a stack and compares them, if equal it pushed true to the top of the booleanStack else false.

JUMP: transfers the program execution to the location of label

Program 1: Associativity & Precedence



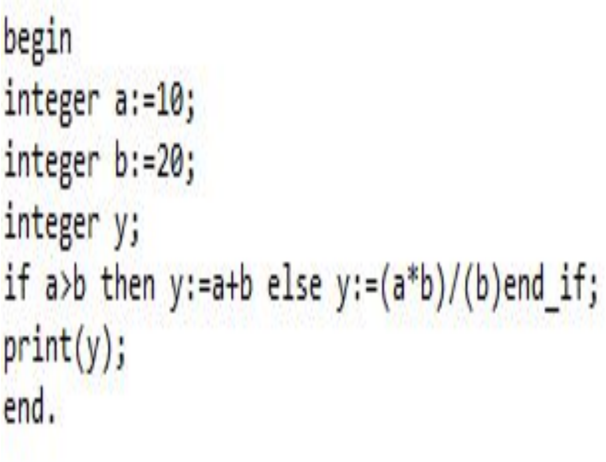
```
begin
integer
x:=2;
integer
y:=5;
integer
z:=7;
integer
d:=6;
integer u;
integer i;
integer t;
u:=(d*4/x*1+4)/(x-y+z);
print(u);
end.
```



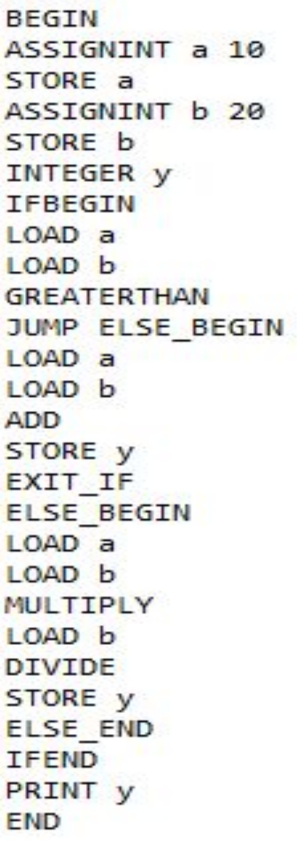
```
BEGIN
ASSIGNINT x 2
STORE x
ASSIGNINT y 5
STORE y
ASSIGNINT z 7
STORE z
ASSIGNINT d 6
STORE d
INTEGER u
INTEGER i
INTEGER t
LOAD d
PUSH 4
MULTIPLY
LOAD x
DIVIDE
PUSH 1
MULTIPLY
PUSH 4
ADD
LOAD x
LOAD y
SUBTRACT
LOAD z
ADD
DIVIDE
STORE u
PRINT u
END
```

Result : 4

Program 2: if then else



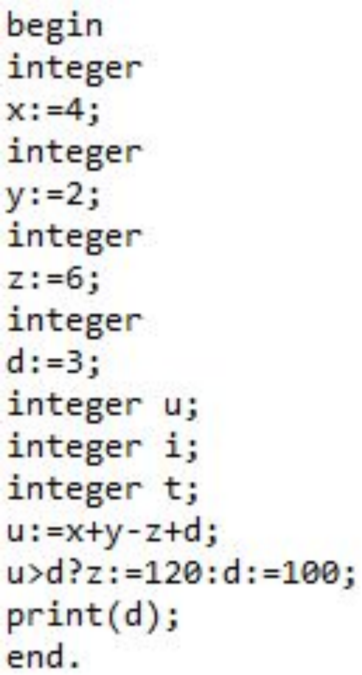
```
begin
integer a:=10;
integer b:=20;
integer y;
if a>b then y:=a+b else y:=(a*b)/(b)end_if;
print(y);
end.
```



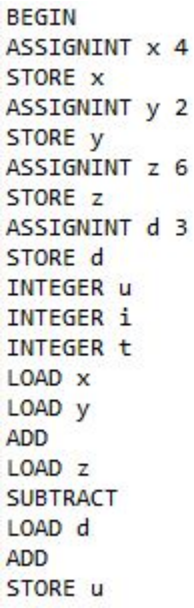
```
BEGIN
ASSIGNINT a 10
STORE a
ASSIGNINT b 20
STORE b
INTEGER y
IFBEGIN
LOAD a
LOAD b
GREATERTHAN
JUMP ELSE_BEGIN
LOAD a
LOAD b
ADD
STORE y
EXIT_IF
ELSE_BEGIN
LOAD a
LOAD b
MULTIPLY
LOAD b
DIVIDE
STORE y
ELSE_END
IFEND
PRINT y
END
```

Result : 10

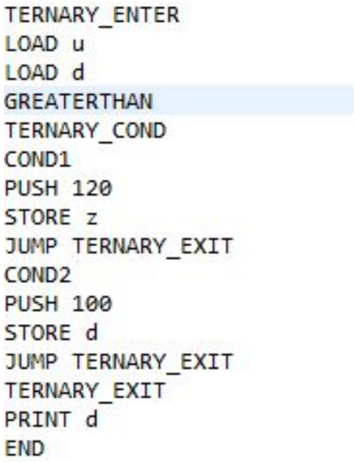
Program 3: Ternary Operation



```
begin
integer
x:=4;
integer
y:=2;
integer
z:=6;
integer
d:=3;
integer u;
integer i;
integer t;
u:=x+y-z+d;
u>d?z:=120:d:=100;
print(d);
end.
```



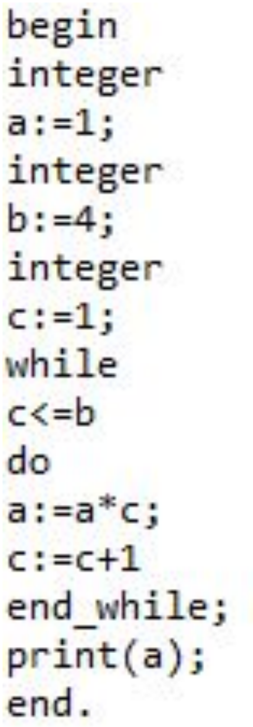
```
BEGIN
ASSIGNINT x 4
STORE x
ASSIGNINT y 2
STORE y
ASSIGNINT z 6
STORE z
ASSIGNINT d 3
STORE d
INTEGER u
INTEGER i
INTEGER t
LOAD x
LOAD y
ADD
LOAD z
SUBTRACT
LOAD d
ADD
STORE u
```



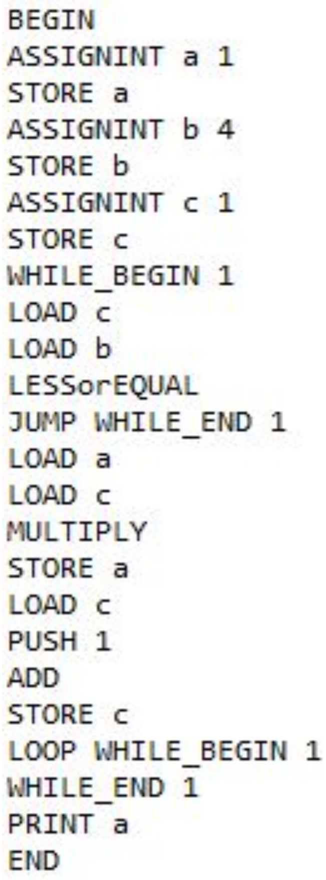
```
TERNARY_ENTER
LOAD u
LOAD d
GREATERTHAN
TERNARY_COND
COND1
PUSH 120
STORE z
JUMP TERNARY_EXIT
COND2
PUSH 100
STORE d
JUMP TERNARY_EXIT
TERNARY_EXIT
PRINT d
END
```

Result : d = 100

Program 4: while loop Operation



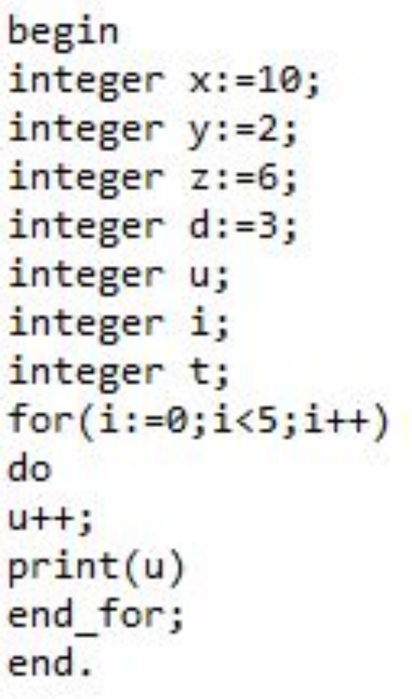
```
begin
integer
a:=1;
integer
b:=4;
integer
c:=1;
while
c<=b
do
a:=a*c;
c:=c+1
end_while;
print(a);
end.
```



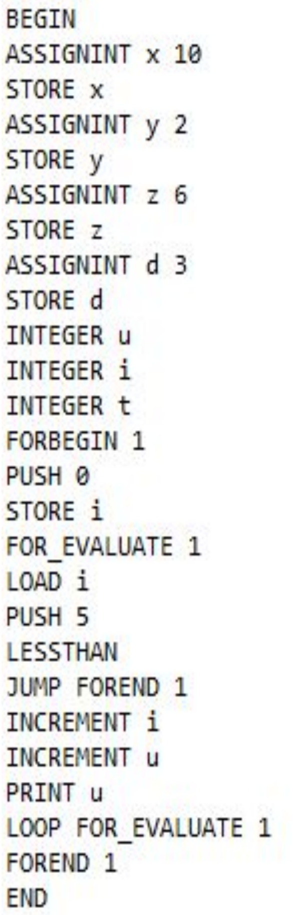
```
BEGIN
ASSIGNINT a 1
STORE a
ASSIGNINT b 4
STORE b
ASSIGNINT c 1
STORE c
WHILE_BEGIN 1
LOAD c
LOAD b
LESSorEQUAL
JUMP WHILE_END 1
LOAD a
LOAD c
MULTIPLY
STORE a
LOAD c
PUSH 1
ADD
STORE c
LOOP WHILE_BEGIN 1
WHILE_END 1
PRINT a
END
```

Result : a = 24

Program 5: Traditional for loop Operation



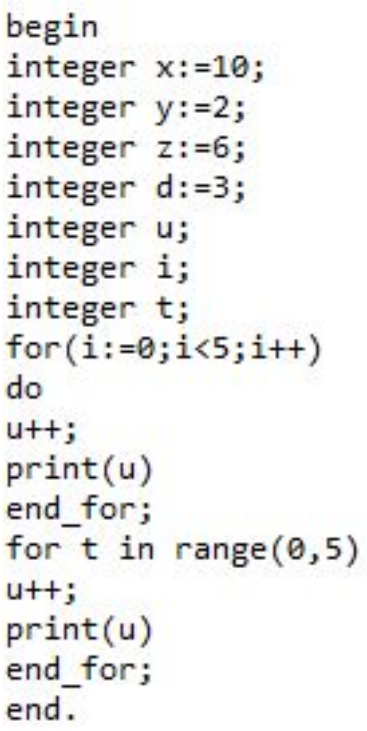
```
begin
integer x:=10;
integer y:=2;
integer z:=6;
integer d:=3;
integer u;
integer i;
integer t;
for(i:=0;i<5;i++)
do
u++;
print(u)
end_for;
end.
```



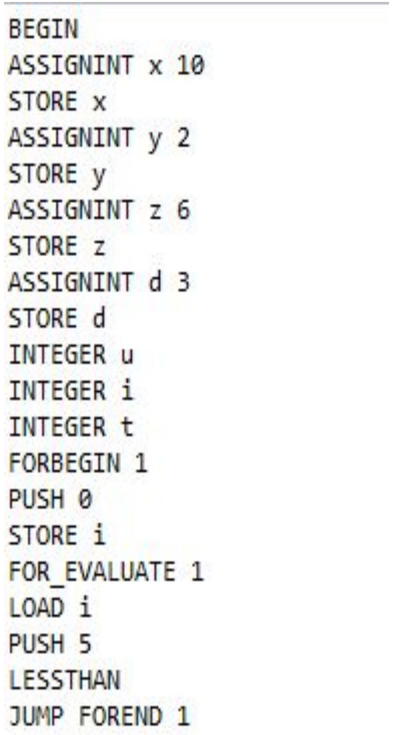
```
BEGIN
ASSIGNINT x 10
STORE x
ASSIGNINT y 2
STORE y
ASSIGNINT z 6
STORE z
ASSIGNINT d 3
STORE d
INTEGER u
INTEGER i
INTEGER t
FORBEGIN 1
PUSH 0
STORE i
FOR_EVALUATE 1
LOAD i
PUSH 5
LESSTHAN
JUMP FOREND 1
INCREMENT i
INCREMENT u
PRINT u
LOOP FOR_EVALUATE 1
FOREND 1
END
```

Result : 1 2 3 4 5

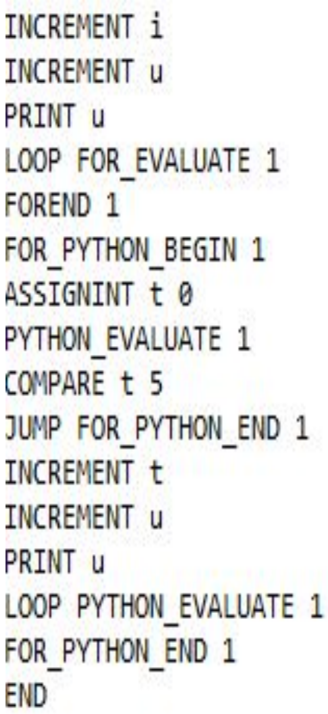
Program 6: Python type for loop Operation



```
begin
integer x:=10;
integer y:=2;
integer z:=6;
integer d:=3;
integer u;
integer i;
integer t;
for(i:=0;i<5;i++)
do
u++;
print(u)
end_for;
for t in range(0,5)
u++;
print(u)
end_for;
end.
```



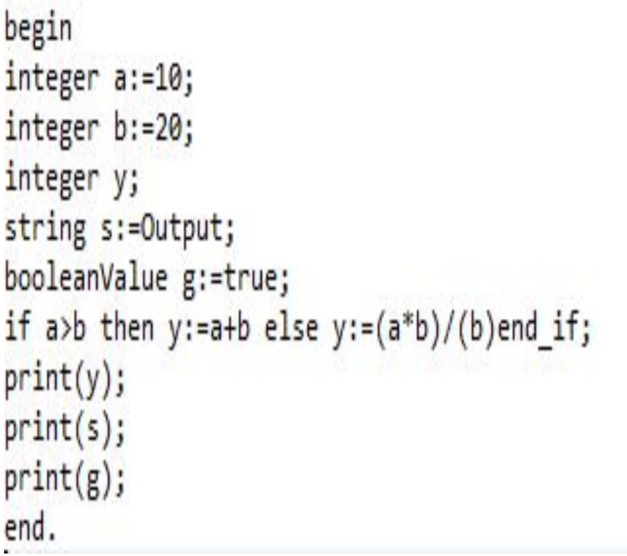
```
BEGIN
ASSIGNINT x 10
STORE x
ASSIGNINT y 2
STORE y
ASSIGNINT z 6
STORE z
ASSIGNINT d 3
STORE d
INTEGER u
INTEGER i
INTEGER t
FORBEGIN 1
PUSH 0
STORE i
FOR_EVALUATE 1
LOAD i
PUSH 5
LESSTHAN
JUMP FOREND 1
```



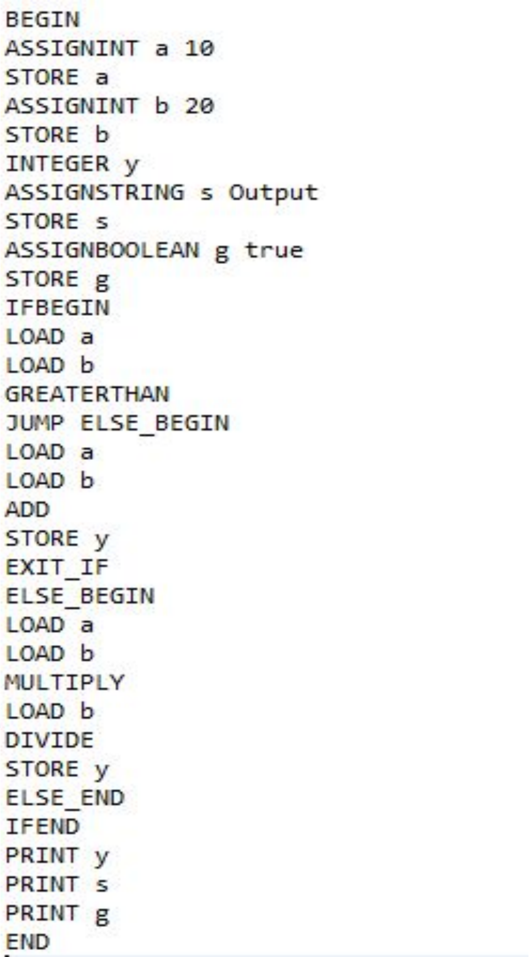
```
INCREMENT i
INCREMENT u
PRINT u
LOOP FOR_EVALUATE 1
FOREND 1
FOR_PYTHON_BEGIN 1
ASSIGNINT t 0
PYTHON_EVALUATE 1
COMPARE t 5
JUMP FOR_PYTHON_END 1
INCREMENT t
INCREMENT u
PRINT u
LOOP PYTHON_EVALUATE 1
FOR_PYTHON_END 1
END
```

Result : 1 2 3 4 5 6 7 8 9 10

Program 7: Print Statement



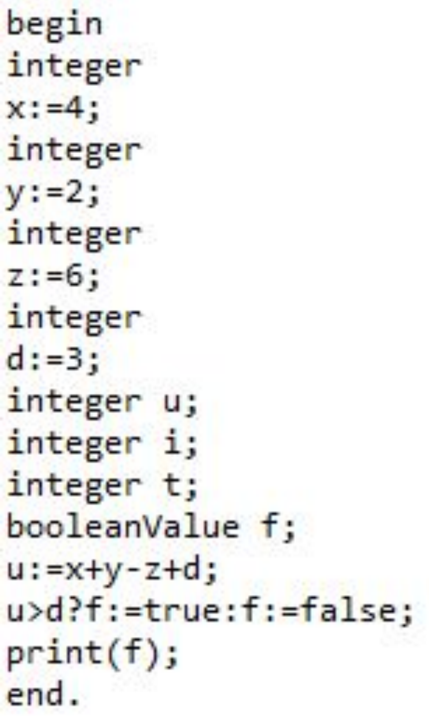
```
begin
integer a:=10;
integer b:=20;
integer y;
string s:=Output;
booleanValue g:=true;
if a>b then y:=a+b else y:=(a*b)/(b)end_if;
print(y);
print(s);
print(g);
end.
```



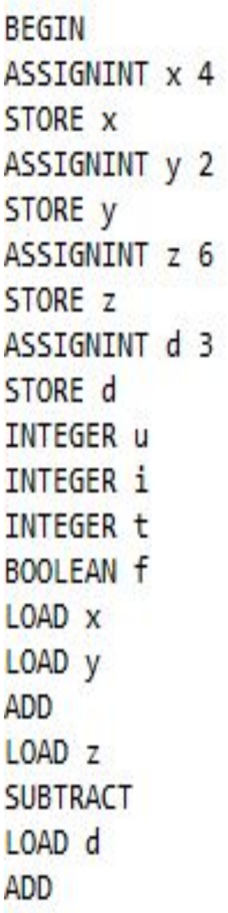
```
BEGIN
ASSIGNINT a 10
STORE a
ASSIGNINT b 20
STORE b
INTEGER y
ASSIGNSTRING s Output
STORE s
ASSIGNBOOLEAN g true
STORE g
IFBEGIN
LOAD a
LOAD b
GREATERTHAN
JUMP ELSE_BEGIN
LOAD a
LOAD b
ADD
STORE y
EXIT_IF
ELSE_BEGIN
LOAD a
LOAD b
MULTIPLY
LOAD b
DIVIDE
STORE y
ELSE_END
IFEND
PRINT y
PRINT s
PRINT g
END
```

Result : 10
Output
true

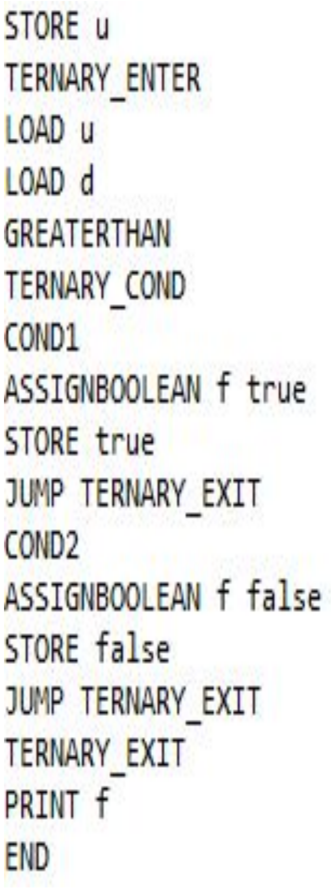
Program 8: Ternary with boolean Statement



```
begin
integer
x:=4;
integer
y:=2;
integer
z:=6;
integer
d:=3;
integer u;
integer i;
integer t;
booleanValue f;
u:=x+y-z+d;
u>d?f:=true:f:=false;
print(f);
end.
```



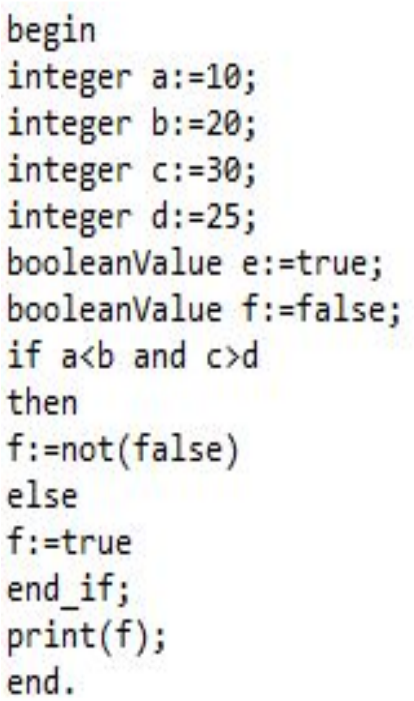
```
BEGIN
ASSIGNINT x 4
STORE x
ASSIGNINT y 2
STORE y
ASSIGNINT z 6
STORE z
ASSIGNINT d 3
STORE d
INTEGER u
INTEGER i
INTEGER t
BOOLEAN f
LOAD x
LOAD y
ADD
LOAD z
SUBTRACT
LOAD d
ADD
```



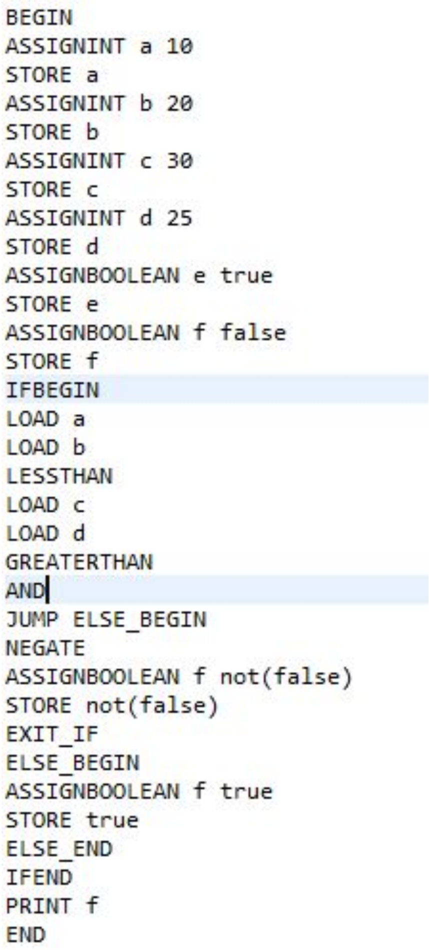
```
STORE u
TERNARY_ENTER
LOAD u
LOAD d
GREATERTHAN
TERNARY_COND
COND1
ASSIGNBOOLEAN f true
STORE true
JUMP TERNARY_EXIT
COND2
ASSIGNBOOLEAN f false
STORE false
JUMP TERNARY_EXIT
TERNARY_EXIT
PRINT f
END
```

Result : false

Program 9: Relational Operators(and,or,not) with boolean



```
begin
integer a:=10;
integer b:=20;
integer c:=30;
integer d:=25;
booleanValue e:=true;
booleanValue f:=false;
if a<b and c>d
then
f:=not(false)
else
f:=true
end_if;
print(f);
end.
```



```
BEGIN
ASSIGNINT a 10
STORE a
ASSIGNINT b 20
STORE b
ASSIGNINT c 30
STORE c
ASSIGNINT d 25
STORE d
ASSIGNBOOLEAN e true
STORE e
ASSIGNBOOLEAN f false
STORE f
IFBEGIN
LOAD a
LOAD b
LESSTHAN
LOAD c
LOAD d
GREATERTHAN
AND
JUMP ELSE_BEGIN
NEGATE
ASSIGNBOOLEAN f not(false)
STORE not(false)
EXIT_IF
ELSE_BEGIN
ASSIGNBOOLEAN f true
STORE true
ELSE_END
IFEND
PRINT f
END
```

Result : true

Runtime environment

1. The runtime environment for this language is written in Java.
2. The intermediate file is parsed and the tokens are analysed and corresponding states are handled efficiently.
3. The data structures used are Stack, Hashmap and Lists.
4. Stack is used to handle the expression evaluation, store and load the values as the program executes
5. Map is used to hold the value of each variable, when the value is assigned or changed.
6. We have used case statements to increase the efficiency of the execution.

Conclusion

- Unlike any imperative languages such as Java, Python and C, deepC is also implemented in a similar way but with limited data structures due to certain constraints. We would like to enhance the language design by incorporating other functionalities and data structures such as function calls, Stack, Maps, Arrays etc.
- Also improve the grammar by additional functionalities..

Thank You