Name: Omar Duadu Student ID: A00315051

Cloud and Computer Architecture – JavaCC Assignment

Motivation

The goal of this assignment was to create a compiler (parser) for a new custom programming language. The requirements of the language are the following:

- Syntax Uniqueness
- Variable storage
- Arithmetic operations
- Conditional operations or Loop

New programming languages and their syntax differences were something I have always found fascinating. Having received this assignment, I made a personal goal of making a concise language syntax that resembles that of the short notes I frequently make.

My chosen language name is ***"Concise"*** or "***ConciseLang***". An example syntax design choice that illustrates this is the full stop "." Symbol as the statement delimiter and the colon ":" for variable assignment.

Language syntax Tokens

The language syntax is designed to be minimalistic and intuitive.

An example illustrating this was the tokens chosen for arithmetic operations. Given that addition and multiplication raise the value, whereas subtraction and division reduce it. The symbols selected for addition and multiplication have a higher elevation (" ' ", " ^ ") then the characters chosen for subtraction and division (" _ ", " ~ ").

Here are the language-defined tokens:

• Statement delimiter:	“ . ”
• Declaration & Assignment:	“ : ”
• Arithmetic Operators:	
- Plus:	“ ‘ ”
- Minus:	“ _ ”
- Multiplication:	“ ^ “
- Division:	“ ~ ”
• Logical Operators:	
- Equal:	“ = “
- Not Equal:	“ # ”
• Print/Display Token:	“ **view** ”

• Ternary Operator:

- “$(Boolean Expression), ArithExpression | ArithExpression.”
- “!$(Boolean Expression), String Literal | String Literal.”

• Comments:	
- Single Line:	“%”
- Muti-Line:	Start “%%. End “%%”

The tokens are further outlined in the user guide.

Challenges

Token Definition:

An issue that occurred early on was when defining the parser tokens.

The variable token definition (*IDENTIFIER*) comprises letters, numbers and symbols with zero or more occurrences.

I mistakenly placed the variable Token as the first defined Token, which resulted in it taking precedence over later defined tokens, such as the reserved keywords like "*repeat*" or "*view*".

Thankfully, after much debugging, this was an easy fix, and all that was required was to change the ordering of the tokens, with *IDENTIFIER* being last.

Variable Declaration & Assignment:

At first, I made the declaration and assignment as separate expressions, and the design flow ensured that only declared variables could hold their assignment. However this behaviour increased complexity, and caused some confusion.

For Example:

$$!a : 5.$$

"!" is the Declaration Symbol for Integer variables. The variable "a" is declared and assigned an integer value 5.

If the following occurred:

$$a : 5.$$

The variable "a" will not be assigned the value 5 as it's yet to be declared.

After careful consideration, I decided to forgo the declaration assignment to keep the language concise and to mimic the dynamic behaviour of languages such as Python, in which the variables will be declared at the time of assignment. Now:

$$a : 5.$$

The variable "a" will be declared and assigned the value 5.

Variable Storage

After defining and implementing the variable parsing, I needed to find a way to store the variables. Thankfully, this was an easy fix by googling and referencing the Javacc guide. I remembered that at the end of the day, this is a Java program, and HashMaps can be used to store the parsed variables.

Using the following variable:

$$Map < String, Integer > varMap = newHashMap < > ();$$

For example:

$$a : 5.$$

The following operation occurs, varMap.put("a", 5)

Ternary condition & return

The next challenge I encountered was implementing the ternary operator. Following the professor's advice, the desired ternary functionality was to work as follows: Given any positive value or true logical expression, it would execute the first expression; otherwise, it would execute the second expression.

Multiple issues needed to be resolved here:

- The need to create a boolean Expression that returns true for any number greater than zero or any true logical expression

Using my earlier Mathematical Expressions as reference I created another function that Extended the arithmetic operations and included the Logical operators.

This was an easy fix as the Javacc guide provides many examples that demonstrate this.

- Conditionally execute an expression

The solution I chose was to execute both expressions and store them in temporary variables val1 and val2, and then, using Java code, return the correct value based on the boolean condition.

- Allowing the ternary to execute any expression

This was a complex problem to resolve; due to the nature of the Javacc parser, I was not able to respond dynamically to the parsed expressions. Since the parser needed to know what the expression would be, I could only create two different ternary operators. The first handles integer expressions (TernaryInt), and the second handles String literals (TernaryStr).

Finally, to mimic the capability of Java in using ternaries for declaration and variable assigning, I updated my parser so that It can handle all combinations with Ternaries.

LOOP

The biggest challenge faced in this assignment was to implement looping.

I know an easy solution for this was to implement it all in Java and only print the variable to the screen. For Example:

```
repeat n 0 to 5
```

Which will print the numbers 0 to 5 to the screen. However I wanted my parser to be as flexible and close to real programming languages, and any feature implemented could not be hard coded.

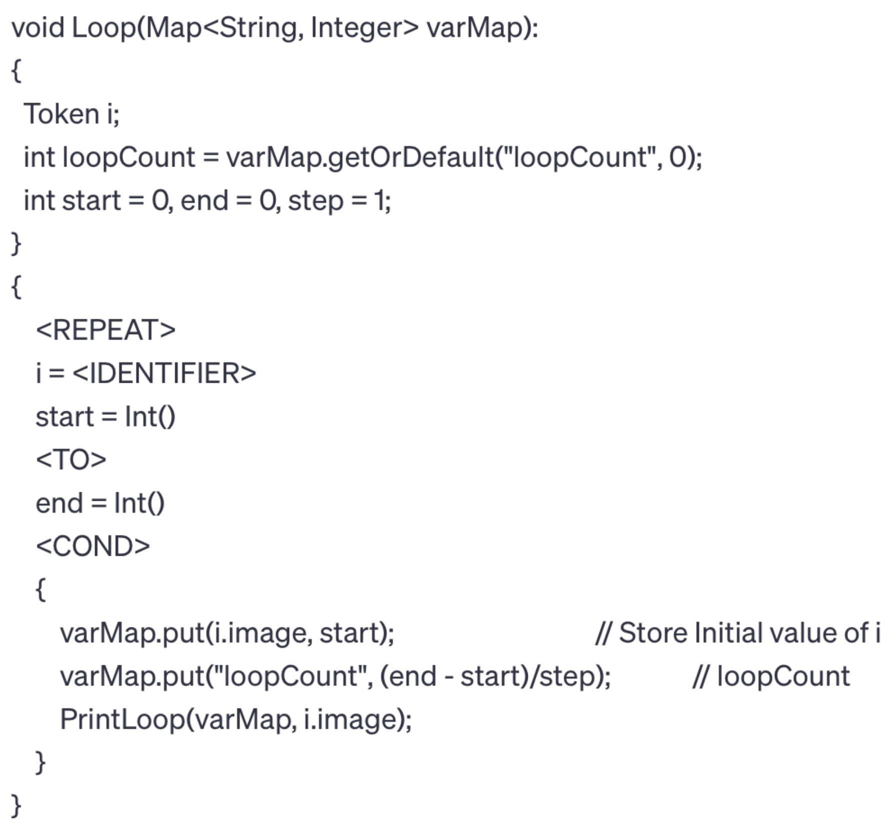
As such what I tried to implement was the following:

```
repeat n [start - end, step],  LoopExpression.
```

Which will result in the LoopExpression being called (end-start)/step times.

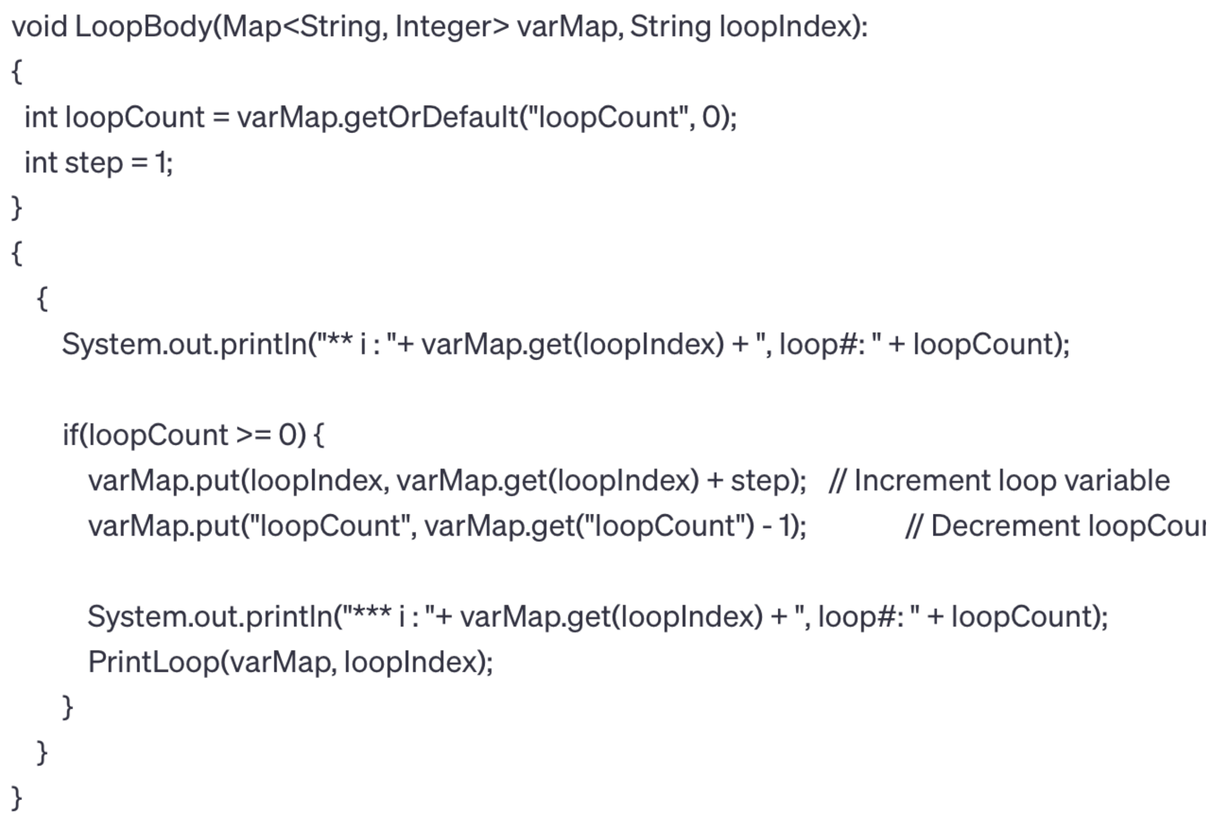
Through online research and consulting my professor I realised the best way to implement this was by recursively calling the LoopExpression.

To facilitate this I created two Parsing expressions, one to handle the Loop initialisation(Loop) and the other responsible for calling the LoopExpression (LoopBody).



```
void Loop(Map<String, Integer> varMap):
{
  Token i;
  int loopCount = varMap.getOrDefault("loopCount", 0);
  int start = 0, end = 0, step = 1;
}
{
   <REPEAT>
   i = <IDENTIFIER>
   start = Int()
   <TO>
   end = Int()
   <COND>
   {
      varMap.put(i.image, start);                    // Store Initial value of i
      varMap.put("loopCount", (end - start)/step);         // loopCount
      PrintLoop(varMap, i.image);
   }
}
```

JavaCC Loop Expression



```
void LoopBody(Map<String, Integer> varMap, String loopIndex):
{
  int loopCount = varMap.getOrDefault("loopCount", 0);
  int step = 1;
}
{
   {
      System.out.println("** i : "+ varMap.get(loopIndex) + ", loop#: " + loopCount);

      if(loopCount >= 0) {
         varMap.put(loopIndex, varMap.get(loopIndex) + step);   // Increment loop variable
         varMap.put("loopCount", varMap.get("loopCount") - 1);           // Decrement loopCou
         
         System.out.println("*** i : "+ varMap.get(loopIndex) + ", loop#: " + loopCount);
         PrintLoop(varMap, loopIndex);
      }
   }
}
```

JavaCC LoopBody Expression

In Loop: Two variables were created, the loop variable (n) and the loop count (loopCount). After assigning them the parsed loop input they are stored in the variable HashMap and the LoopBody is called.

The LoopBody expression fetches the loopCount and executes the LoopExpression if the loopCount is greater than zero.

Even after this, the loop expression was not executed automatically. The loop variables worked as desired, but the program paused each iteration for the user to input the LoopExpression.

The next challenge was to figure out a way in which the JavaCC statement could be parsed and then repeatedly called each loop.

Searching the JavaCC FAQ, I realised this was a common challenge, and the best approach was to output some intermediate representation and then interpret it. I found a possible solution was to use the JJTree; however, due to time constraints, I unfortunately could not complete it.

Language Limitations

Statements:

- All statements must end with a "." Delimiter. Except boolean and arithmetic expressions.

Variable:

- Can only declare & Assign Integers
- Cannot handle doubles/floats, only Integers in the form 05, or 5

  For example Syntax error if given double 33.33

Arithmetic operations:

- Do not take brackets into account

Logical Operations:

- No OR, and AND operators.

Ternary:

- Only one statement can be executed in ternary expressions
- Ternary can only handle and return Strings or Integers

  Both expressions in ternary need to be the same type, either Integer or String.

Character Literals:

- Single Characters are printed as String literal

  Using double quotes as single quotes is reserved for addition operator. For Example. "c".

Loops:

- No Looping constructs

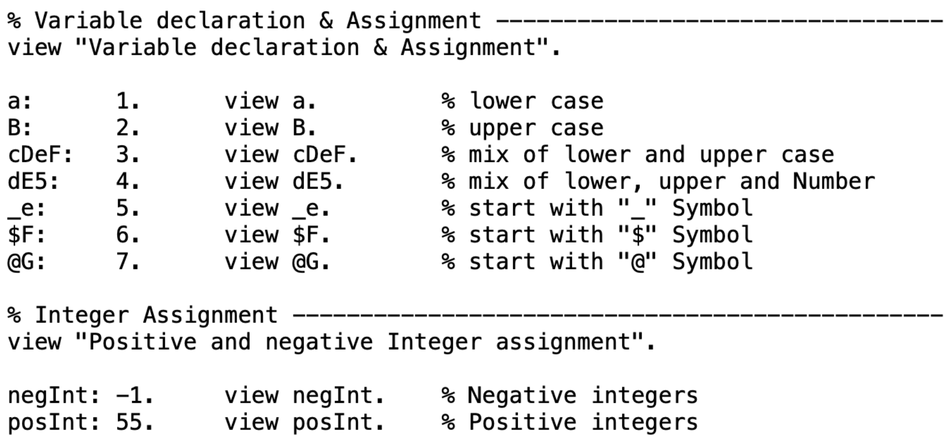
Test Methodology

The testing was performed in conjunction with the feature development. All feature combinations were tested. At completion the full language was tested using 3 test scripts and file outlining all possible use cases.

The test scripts are {ConciseLang_UseCases, Test_Script1.txt, Test_Script2.txt, Test_Script3.txt}

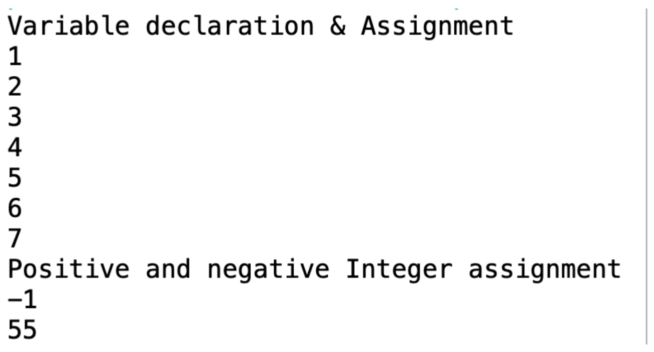
The following are the tests performed in ConciseLang_UseCases.txt.

Test Variable Declaration and Assignment:



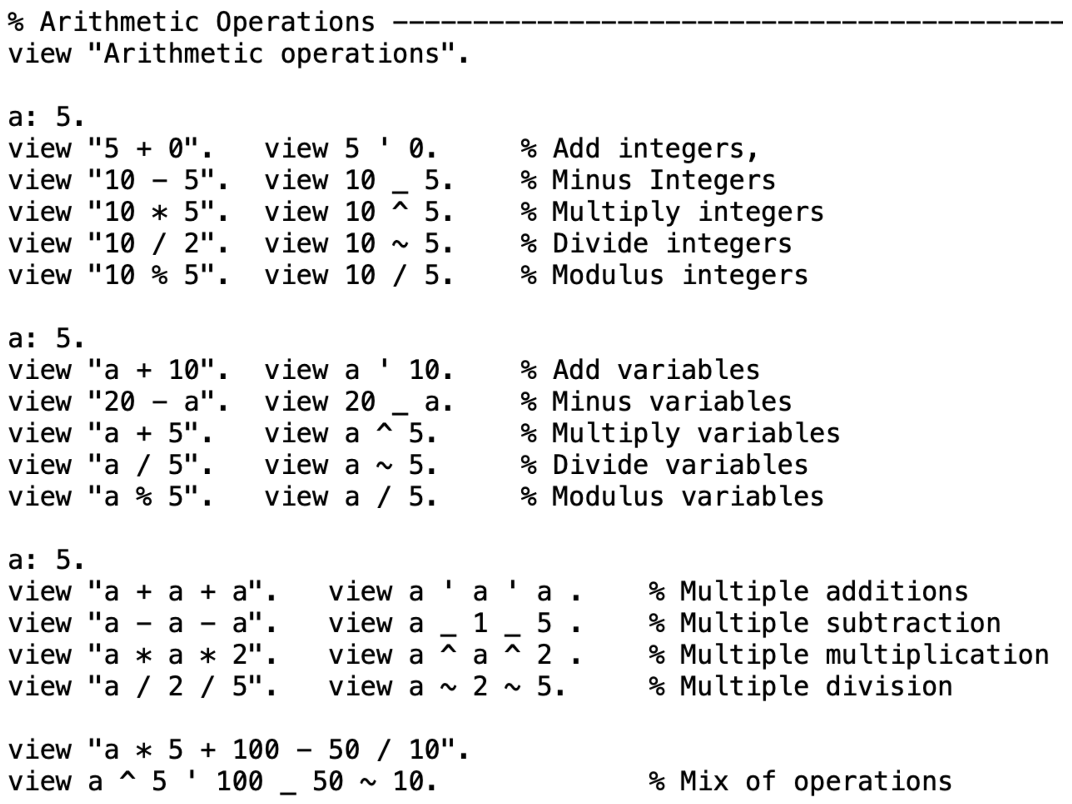
```
% Variable declaration & Assignment ----------------------------------
view "Variable declaration & Assignment".

a:      1.      view a.         % lower case
B:      2.      view B.         % upper case
cDeF:   3.      view cDeF.      % mix of lower and upper case
dE5:    4.      view dE5.       % mix of lower, upper and Number
_e:     5.      view _e.        % start with "_" Symbol
$F:     6.      view $F.        % start with "$" Symbol
@G:     7.      view @G.        % start with "@" Symbol

% Integer Assignment -------------------------------------------------
view "Positive and negative Integer assignment".

negInt: -1.     view negInt.    % Negative integers
posInt: 55.     view posInt.    % Positive integers
```

Variable Declaration and Assignment



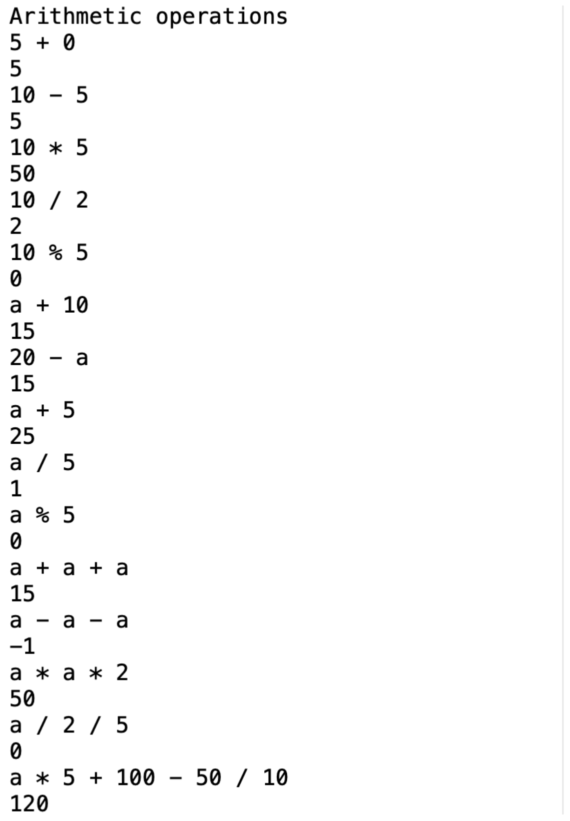
```
Variable declaration & Assignment
1
2
3
4
5
6
7
Positive and negative Integer assignment
-1
55
```

Result



```
% Arithmetic Operations ------------------------------------------
view "Arithmetic operations".

a: 5.
view "5 + 0".   view 5 ' 0.     % Add integers,
view "10 - 5".  view 10 _ 5.    % Minus Integers
view "10 * 5".  view 10 ^ 5.    % Multiply integers
view "10 / 2".  view 10 ~ 5.    % Divide integers
view "10 % 5".  view 10 / 5.    % Modulus integers

a: 5.
view "a + 10".  view a ' 10.    % Add variables
view "20 - a".  view 20 _ a.    % Minus variables
view "a + 5".   view a ^ 5.     % Multiply variables
view "a / 5".   view a ~ 5.     % Divide variables
view "a % 5".   view a / 5.     % Modulus variables

a: 5.
view "a + a + a".   view a ' a ' a .    % Multiple additions
view "a - a - a".   view a _ 1 _ 5 .    % Multiple subtraction
view "a * a * 2".   view a ^ a ^ 2 .    % Multiple multiplication
view "a / 2 / 5".   view a ~ 2 ~ 5.     % Multiple division

view "a * 5 + 100 - 50 / 10".
view a ^ 5 ' 100 _ 50 ~ 10.             % Mix of operations
```

Arithmetic operations

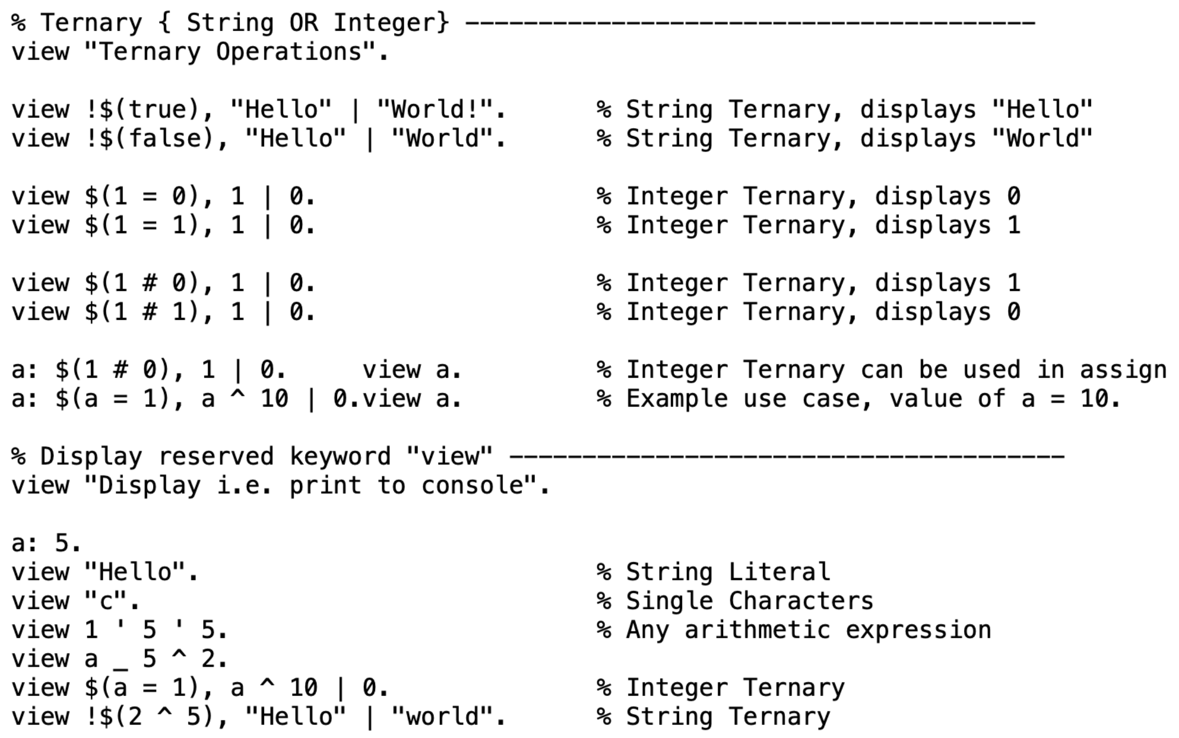


```
Arithmetic operations
5 + 0
5
10 - 5
5
10 * 5
50
10 / 2
2
10 % 5
0
a + 10
15
20 - a
15
a + 5
25
a / 5
1
a % 5
0
a + a + a
15
a - a - a
-1
a * a * 2
50
a / 2 / 5
0
a * 5 + 100 - 50 / 10
120
```

Result

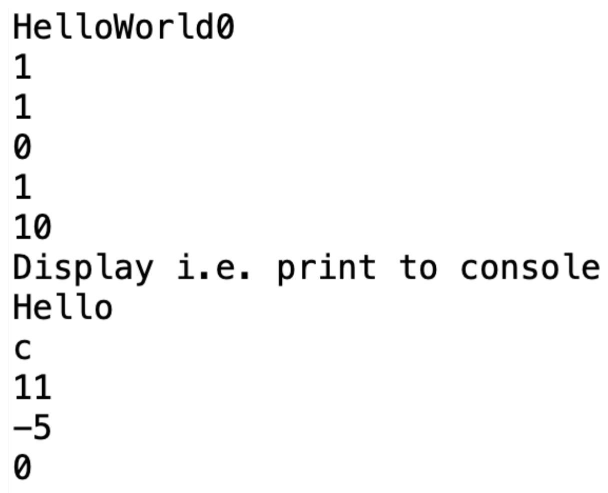
Test Arithmetic Operations:

Test Ternary and View:



```
% Ternary { String OR Integer} ---------------------------------------
view "Ternary Operations".

view !$(true), "Hello" | "World!".      % String Ternary, displays "Hello"
view !$(false), "Hello" | "World".      % String Ternary, displays "World"

view $(1 = 0), 1 | 0.                   % Integer Ternary, displays 0
view $(1 = 1), 1 | 0.                   % Integer Ternary, displays 1

view $(1 # 0), 1 | 0.                   % Integer Ternary, displays 1
view $(1 # 1), 1 | 0.                   % Integer Ternary, displays 0

a: $(1 # 0), 1 | 0.     view a.         % Integer Ternary can be used in assign
a: $(a = 1), a ^ 10 | 0.view a.         % Example use case, value of a = 10.

% Display reserved keyword "view" ---------------------------------------
view "Display i.e. print to console".

a: 5.
view "Hello".                           % String Literal
view "c".                               % Single Characters
view 1 ' 5 ' 5.                         % Any arithmetic expression
view a _ 5 ^ 2.
view $(a = 1), a ^ 10 | 0.              % Integer Ternary
view !$(2 ^ 5), "Hello" | "world".      % String Ternary
```

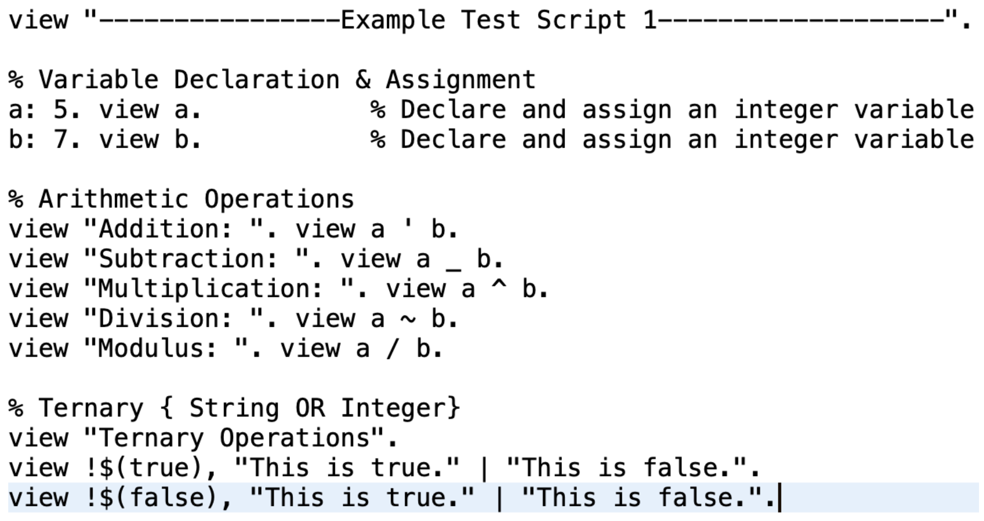
Ternary and Display operations



```
HelloWorld0
1
1
0
1
10
Display i.e. print to console
Hello
c
11
-5
0
```

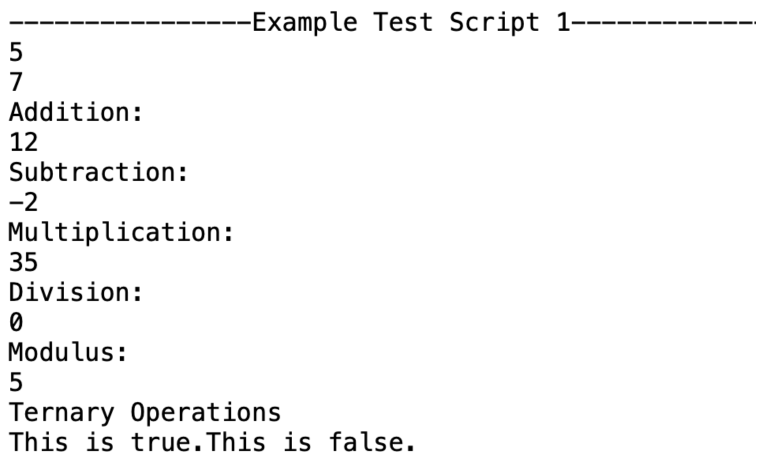
Result

Test Script #1:



```
view "----------------Example Test Script 1--------------------".

% Variable Declaration & Assignment
a: 5. view a.           % Declare and assign an integer variable
b: 7. view b.           % Declare and assign an integer variable

% Arithmetic Operations
view "Addition: ". view a ' b.
view "Subtraction: ". view a _ b.
view "Multiplication: ". view a ^ b.
view "Division: ". view a ~ b.
view "Modulus: ". view a / b.

% Ternary { String OR Integer}
view "Ternary Operations".
view !$(true), "This is true." | "This is false.".
view !$(false), "This is true." | "This is false.".
```

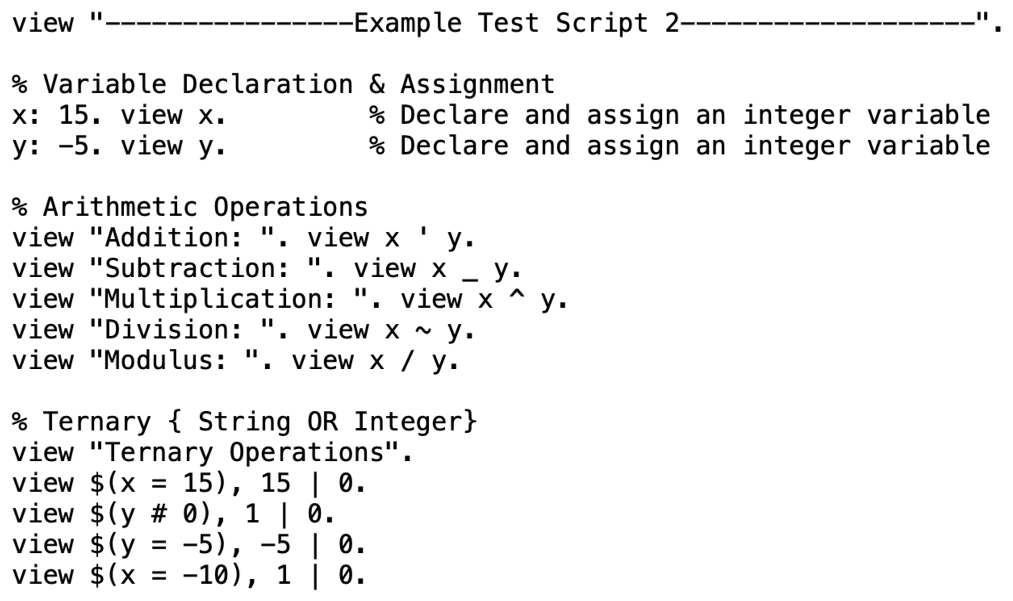
Script 1



```
----------------Example Test Script 1------------
5
7
Addition:
12
Subtraction:
-2
Multiplication:
35
Division:
0
Modulus:
5
Ternary Operations
This is true.This is false.
```

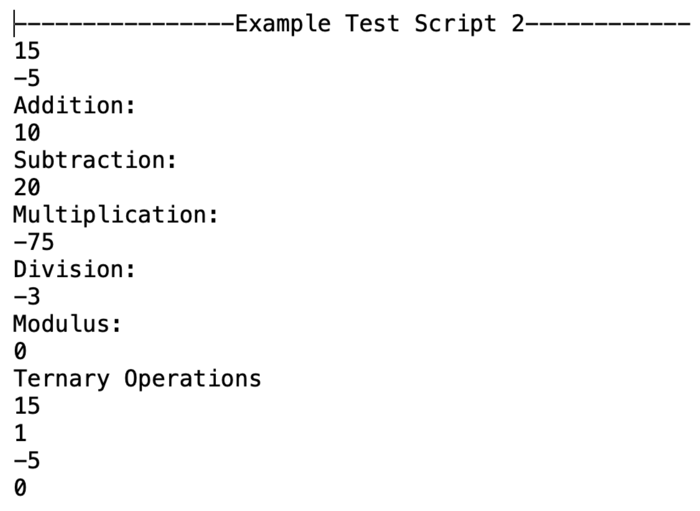
Result

Test Script #2:



```
view "----------------Example Test Script 2-------------------".

% Variable Declaration & Assignment
x: 15. view x.         % Declare and assign an integer variable
y: -5. view y.         % Declare and assign an integer variable

% Arithmetic Operations
view "Addition: ". view x ' y.
view "Subtraction: ". view x _ y.
view "Multiplication: ". view x ^ y.
view "Division: ". view x ~ y.
view "Modulus: ". view x / y.

% Ternary { String OR Integer}
view "Ternary Operations".
view $(x = 15), 15 | 0.
view $(y # 0), 1 | 0.
view $(y = -5), -5 | 0.
view $(x = -10), 1 | 0.
```

Script 2



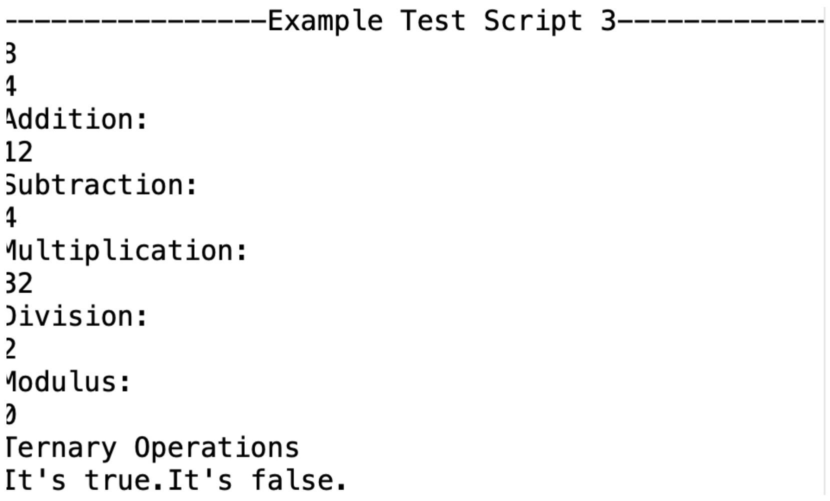
```
|----------------Example Test Script 2------------
15
-5
Addition:
10
Subtraction:
20
Multiplication:
-75
Division:
-3
Modulus:
0
Ternary Operations
15
1
-5
0
```

Result

Test Script #3:

```
view "----------------Example Test Script 3-------------------".

% Variable Declaration & Assignment
num1: 8. view num1.       % Declare and assign an integer variable
num2: 4. view num2.       % Declare and assign an integer variable

% Arithmetic Operations
view "Addition: ". view num1 ' num2.
view "Subtraction: ". view num1 _ num2.
view "Multiplication: ". view num1 ^ num2.
view "Division: ". view num1 ~ num2.
view "Modulus: ". view num1 / num2.

% Ternary { String OR Integer}
view "Ternary Operations".
view !$(true), "It's true." | "It's false.".
view !$(false), "It's true." | "It's false.".
```

Script 3



```
----------------Example Test Script 3------------
8
4
Addition:
12
Subtraction:
4
Multiplication:
32
Division:
2
Modulus:
0
Ternary Operations
It's true.It's false.
```

Result