Hammad K. Musakhel, 21801175
CS315-3
HW2

1) DART:

-Pretest, posttest, or both

```
/* HW1 | Dart VM v1.10.1) */
void main(){
  print("HW2 Dart Example");

 //pretest

var check = 5;
 while( check != 0){
  print("while loop must function 5 times");
  check--;
 }
 print( “”);
```

The while loop, a pretested loop, is very simple and similar in DART. It checks for the argument provided inside the parentheses to be evaluated as true out false and commences then. The result is “**while loop must function 5 times**” printed 5 times.

```
 for( var x = 5; x > 0; x--){
  print( "x = $x"); //display in descending order
 }
```

The for loop is also a protested loop where the variable inside the argument is tested against its range and then commences and carries out its instructions given in the body. The program produces a result of **5 to 0** in desc4ending order.

```
 //posttest
 var p = 6;
 do{
  print("value is = $p"); //will terminate when on
  p--;
 }while( p != 6 && p != 5);
 print( “");
```

The do…while loop is a posttest loop which then evaluates the condition after executing it for at least one times (if the scenario were false in while). For example in the above case, it does

print 6 once although it has stated in the while to terminate it when variable is equal to 6, but it will execute once and then is post-tested for termination.

User Located Control Mechanism

```
//break
for ( var i = 1; i < 10; i++){
  if ( i % 3 == 0)
    break;
 print( "value of i = $i");
}
print(“");
```

Break is a very important instruction as allows the user to control the flow or termination of the loop based on his/her scenarios defined. In the above example, if ‘I’ is divisible by 3, then it will literally break the corresponding loop. The result is:

```
value of i = 1
value of i = 2
```

```
for ( var i = 0; i < 5; i++){
  for(var k = 1; k < 3; k++){
    if ( k > 0)
      break;
    print( "value of k = $k");
  }
 print( "value of i = $i");
}
print( “");
```

In this case of break, we clearly depict the functioning of break and that is: break only breaks the corresponding loop it is called in for. It won’t affect the outer loop if there is one. The above program shows that k is not printed at all but the value for ‘i' is printed; the result is:

```
value of i = 0
value of i = 1
value of i = 2
value of i = 3
value of i = 4
```

If we change the if to if (k % 3 == 0) then we shall see traces of k as 1 and 2 being printed before every printing of the above loop’s command.

```
//continue
  for ( var j = 1; j < 10; j++){
    if ( j % 3 == 0)
      continue;
   print( "value of j = $j");
  }
```

Here, we depict the character of continue, which is also a very crucial element for user control over loops. Hence by using continue**,** we skip the set of instruction(s) in that loop and jump to the next iteration. The result is for the above program is:

```
value of j = 1
value of j = 2
value of j = 4
value of j = 5
value of j = 7
value of j = 8
```

```
var y = [7, 8, 9];
  var h = [8, 9];
  outer: for (var c in y) {
    for (var e in h) {
      if (e == c){
        continue outer;
      }
    }
      print(c); //only 7 printed
  }
}
```

This program depicts another component important for user controlled mechanism in regarding loops; this method is defining a procedure so that it can be reinstantiated from the point (**label**) where it was previously skipped from; the above program depicts this behavior quite briefly. Two lists are initialized with integers and the **outer:** defines the procedure to go with, however if the elements in either list are same then we call continue order which straightly jump to the starting loop. We already know that continue skips instructions for the loop it is called in for, but here it skips the entire component and starts from **outer.** The outer here acts as a Label that can be identified. Similar approach can also be used for break which results in the halting of the entire component, regardless how many nested loops. The result for the above program is **7,** as it is the only element in **y** not equal to the one in **h**.

2) JS

-Pretest, posttest, or both

```
console.log('HW2')

//pretest
var check = 5
while( check != 1){
   console.log(check)
   check--
}
```

The while loop is JavaScript has the same crux as in others: pretest and proceed. If the condition provided is true, it will proceed otherwise it wouldn't. The above program produced a result of:

5
4
3
2

```
for( var i = 5; i > 0; i--){
   console.log("i = " + i)
}
```

It is also a pretest loop where the "I" is tested against the condition of I>0, otherwise it won't proceed. However, it is still basic and generic.

```
//posttest
var x = 5
do{
   console.log("will execute once no matter what")
}while( x < 3)
```

The do..while loop is a posttest scenario where the instruction does execute at least for once. The instruction(s) is executed before testing, and as we can see the result for this above program is:

will execute once no matter what

It only prints once as the condition provided in while is x < 3 2which false, thus the loop terminates.

User Located Control Mechanism

```
//will only execute with 5 and then terminate
for( var j = 5; j > 0; j--){
   if(j % 2 == 0)
   break
   console.log(j)
}
```

This program allows us to interpret the functioning of break; break literally breaks the corresponding loop making it stop in JavaScript. The above program produces a result of:

5

3

1

Used in for loop, it moves to the next iteration and thus it acts like continue in for loop, however if we use it in a while loop to exit a situation where infinite loop status can be feared to be achieved:

```
while (true) {
var str = prompt("entry code needed: ");
 while(true){
    var str2 = prompt("enter pass= ");
     if (str2.length > 5) {
     console.log("success");
     break;

     console.log("re-enter")

 }
 }
 if (str.length > 6) {
    console.log("success");
    break;

  }
  console.log("xxxx");
}
```

In this case, we use while(true) which means that it is true and will keep on repeating and repeating. We need to find a way to exit the loop when needed. We also have nested loop and we depict that using break in one loop doesn't break the entire looping rather just the one where it is called. The result is fascinating based on what is supplied. If str2 is in fact greater than 5 in length then it will print success and break to the outer loop, and from there it commences and the first loop is assessed again there. The result is **success** for both if used properly. The point is to prove and show the working of break.

```
//skipping the one meeting conditions
for( var j = 4; j > 0; j--){ //3 and 1
   if(j % 2 == 0)
   continue
   console.log(j)
}
```

The above program skips if j is meeting the conditions of **j % 2 == 0** and thus it jumps tp the next iteration just like the generic continue does. The result is:

3

1

```
var a = [7, 8, 9]
var b = [ 8, 9]

OUTER: for (var i in a) {
  for (var j in b) {
    if (a[i] == b[j]){
    continue OUTER;
    }
  }
  console.log(a[i] + " not in the list");
}
```

Here we make use of OUTER with continue so that we can depict the usage of jumping to the start with the help of labels just as done in DART. Labels are an assistance to come out of all the loops and go though the point where the label is defined. Here, since we go to the start, we can see how it is needed; only using continue would only exit the second loop in the above case, thus we need labels here. The result is:

7 not in the list

We can also use break OUTER to exit all the nested loops or to go to points where we intend to go as being in nested loops limits one's options. Labels are very consistent and thus can work in ay case of loops.

3) LUA

-Pretest, posttest, or both

```
print("Hw2")

x = 5
while( x > 0 )
do
  print("value of x:", x)
```

```
  x = x - 1
end
```

This while loop in LUA is categorically the same as others. It is also a pretest-type of loop where it tests first and then commences. In the loop above, **5** is testes against the condition **x > 0** and the only the instruction inside the while body is executed. The result is:

```
value of x: 5
value of x: 4
value of x: 3
value of x: 2
value of x: 1
```

```
for i = 4,1,-1
do
  print(i)
end
```

The for loop is generic although with a different syntax. Here, **i=4** is the variable to test the for loop, **1** is the limit and **-1** is the increment/decrement for the loop. The result is:

```
4

3
2
1
```

```
a = 6
repeat
  print("value of a:", a)
  a = a - 1
until( a < 7 )
```

The repeat..until loop is a posttest loop where the condition is tested later, thus we will get a result, at least one, no matter what. Here, in the above program, we pass the value of a as 6 as we define the condition as until a < 7. Although, a is already less than 7 but still the result is:

```
value of a: 6
```

This testifies the claim that repeat…until is a posttest mechanism provided by LUA.

User Located Control Mechanism

Lua doesn't have an inbuilt mechanism for continue but can be emanated through other means. It has **break** and **go-to** instructions which allow the user to have some control over the functioning of loops.

```
for i = 5, 0, -1
```

```
do
   if( i % 2 == 0)
   then
      break;
   end
   print(i)
end
```

The above program executes a for loop and then has a conditional statement that if the integer i is divisible by two, **break** the corresponding loop. Thus, it only results in printing: **5,** as expected. The standard is maintained, the corresponding loop is only broken.

```
t = 0
while( t < 4)
do
   print("check goto")
   r = 6
   while( r < 10)
   do
      if (r == 6)
      then
         goto quitOuter
      end
      print("successful")
      r = r + 1
   end
   print("2nd success")
   t = t + 1
end
::quitOuter::
```

The above program only prints “check goto” as all the other loops and instructions are by passed due to the usage of goto statement. The goto instruction jumps to the label defined by it and by passes all the loops (internal or external), and thus the result obtained from the above program is only:

```
check goto
```

The goto statement can also be used to make-way for the continue statement as there are numerous code examples provided on the internet. One simple program is:

```
for i=1,10,1
do
  if i % 2 == 0
```

```
  then
     goto continue
  end
  print(i)
  ::continue::
end
```

The above program is a very simple way of executing continue as goto is used to jump to the end of the loop, thus by passing the other instructions: continue is emanated. The result for this is:

```
1
3
5
7
9
```

4) PHP

-Pretest, posttest, or both

```
echo "HW2! <br>";

$check = 4;
while($check > 0){
echo ("$check<br>");
 $check--;
}
```

The while loop in PHP is very similar to other languages' while loop in terms of syntax and functionality. It s a protesting mechanism where the condition is tested, and if it is true it will proceed to carry out the instructions in the while loop's body. The above program proceeds to the while loop's body as the condition is true (check is greater than 0) and hence it prints 4 and downwards to 1, pertaining to the condition provided. The result obtained is:

```
4 3 2 1
```

```
for ($i = 0; $i <= 4; $i++) {
  echo "number: $i <br>";
}
```

This for loop, as usual, has the **pretest** criterion to be fulfilled: the initial condition needs to be satisfied that is it needs to be true in order to proceed. Since the condition is satisfied, that is its true, it does proceed and produces the result:

```
number: 0
number: 1
```

```
number: 2
number: 3
number: 4
```

```
$p = 2;

do {
  echo "$p <br>";
  $p--;
} while ($p >= 5);
```

The above program is for **do-while** loop which follows the **posttest** mechanism. The variable $p is assigned with integer 2 and then it precedes with do and so forth. However, since it is posttest, it will execute once before being testes against the while's condition which will ultimately decide the fate of the loop. Since the condition in the while's argument isn't satisfied/ true, the loop is terminated. The result is:

```
2
```

User Located Control Mechanism

PHP has inbuilt instructions for break, continue, and goto.

```
for ($q = 5; $q > 0; $q--) {
  if ($q == 3) {
    break;
  }
  echo "$q <br>";
}
```

The above program depicts the usage of break in PHP. Break only breaks the corresponding loop and starts with the next loop or the new instructions. The above program produces the result of:

```
5
4
```

It is because as soon as $q equals 3, the loop is broken and terminated.

```
for ($k = 0; $k < 5; $k++) {
  if ($k % 2 == 0) {
    continue;
  }
  echo "number= $k <br>";
}
```

Contrary to break, continue just simply jumps the iterations where t is asked to jump as in the above case it is asked to jump the even numbers thus we only gain the result of:

```
number= 1
```

```
number= 3
```

```
for($j=5; $i<10; $i++) {
  $j--;
  while($j > 0) {
    if($j==4)
        goto end;
  }
  echo "j = $i";
}
end:
echo ‘goto executed’;
```

The goto statement by passes all the loops and jumps straight to the label and prints:

goto executed

As soon as the condition is met and goto is called, it jumps to the defined label to further execute the program; goto is very crucial and allows us to quit from loops no matter how much nested.

5) Python

-Pretest, posttest, or both

```
print('Hw2')
x = 5
while x > 0:
    print(x)
    x -= 1
```

The while loop is logically controlled loop in Python having the basic pretest criterion. The syntax is fairly different for python. The condition is checked, again and again until it’s false; initially, the condition is met/true thus it proceeds. The result is integers from 5 in descending order:

5
4
3
2
1

```
for x in range(2, 10, 2):
  print(x)
```

The for loop is basic with its basic functionality with testing the integers against the limitation. In python we have the option of providing a range and an increment factor, and then carry out the instructions. The result for the above program is:

```
2

4
6
8
```

```
i = 1
while True:
    print(i)
    i = i + 1
    if(i > 3):
        break
```

The language of python doesn't have inbuilt do-while mechanism, but can be emanated. The above program does that for us, the while condition is given as if(condition) and we can see it retains its posttest feature as it will print once no matter what. It is certainly not entirely like the do-while loop though it can be used like it if proper conditions for breaking the loop are given. The result for the above program is:

```
1

2

3
```

The white true bring an infinite loop is broken as the condition is met, when variable i = 4, i > 3, thus it breaks.

User Located Control Mechanism

Python has break and continue as user controlled mechanism regarding loops. It misses out goto.

```
for char in "hammad":
    if char == "m":
        break
    print(char)

print("bitti")
```

Break s called when we need to terminate a loop in between with conditions we make/desire. As the **char == m**, the loop is broken (the corresponding loop), and thus we can see that in the result:

```
h
```

a
bitti

The string is “hammad” and is then iterated through for retrieval of cha through **for x char in string;** as soon as the char matched the condition for break, the loop is broken. We can use this feature to also bring an end to infinity loops.

Continue works like Break, but it doesn’t terminate the loop, rather skips the iteration without performing the instructions for that loop under continue. An example is:

```
for char in "hammad":
   if char == "m":
      continue
   print(char)

print(" continue bitti”)
```

We have taken the same program as that of break, but changed only break to continue. The change is:

```
h
a
a
d
 continue bitti
```

As mentioned before, it only skips the iteration. Python doesn’t use Goto statements or Labels to jump out of loops rather it can use different variations of break, continue, or while true: to facilitate that.

6) Ruby

-Pretest, posttest, or both

The logically controlled loops in Ruby are while and do-while loops.

```
puts 'Hw2'

$x = 5
while $x > 0 do
  puts("#$x" )
  $x -=1
end
```

The while loop, just like other languages tests the conditions and if it is true, it proceeds with the body of the while loop (Pretest). The loop is terminated once the condition is false. The recur of there above program is:

```
5
```

```
4
3
2
1
```

```
$i = 0
num = 5
begin
  puts("i = #$i" )
  $i = $i + 1;
end until $i < num
```

The do-while loop in Ruby is proceeded as mentioned above. It is a posttest-type mechanism where the condition is tested later hence the proceeding is expected to be executed at least once. As we can see above, variable i is equal to zero and num = 5; we allow the body of the loop to begin and have print command for variable i. The condition is that i has to be greater than num for the body of loop to keep going on, which it is not during initialization. This confirms its posttest nature, as the result is primed once and the terminated as until is met then. The result for this is:

```
i = 0
```

User Located Control Mechanism

Break and Next are defined in Ruby with missing goto.

```
x = 1
while true do
   if x % 3 == 0
      break
   end

   puts x
   x += 1
end
```

Here, we use the break command to break the loop according to the user's need that whenever the first integer divisible by 3 is obtained, break the loop. As soon as the condition for break is met ($x % 3 == 0; as the remainder of 3 % 3 = 0, thus we should et 1 and 2 only as the output). This is an infinite loop and break is a nice mechanism to get out of one. Break only eaves the corresponding loop, following the general criteria for languages. The result for this is:

```
1
2
```

```
randomNumbers = [3, 6, 7 , 8, 9, 11, 55, 6, 99, 10]
$i = 0
```

```
while $i < randomNumbers.length do


   if randomNumbers[$i] % 2 == 0
     $i =$i + 1
     next
   end

   puts("#{randomNumbers[$i]}")
   $i =$i + 1

 end
```

Ruby makes use of next as continue, and as we can see we have defined the even numbers as condition to be skipped thus we have used next underneath that condition. Next only skips the iteration rather than breaking the loop. The result for this is just odd numbers contained in the last:

```
3
7
9
11
55
99
```

7) Rust

-Pretest, posttest, or both

```
let mut check = 0;
while check < 10 {
   println!("{}", check);
   check += 1; }
```

While is one of the logically controlled loops in Rust as well with pretest criterion; it requires the condition to be true in order to execute the commands in the body pf the loop. In the above example, the condition is met as check is smaller than 10, and the result obtained is:

```
0
1
2
3
4
5
6
7
8
9
```

Rust doesn't have in-built do-while, however it has loop in it which can be manipulated to work as do-while:

```
let mut x = 0;
loop {
   println!(" {}", x);
   x = x + 1;
   if x == 2 {
    break;
   }
}
```

The body comes first before the break command thus emanates the do-while loop, **posttest**. The checking of the instructions comes afterwards, and functions almost like a do-while. However, it is certainly not exactly like do-while but can used like it if used perfectly. The result for the above program is: **0 1**

User Located Control Mechanism

```
let mut p = 0;
for x in 1..10 {
   if x > 6 {
      break;
   }
   p = x;
}
println!("{}", p);
}
```

Break is used in Rust just like in other languages defined above. Break is primarily used to exit the corresponding loop or to put an end to an infinite loop (especially ones with demeaning password verifications etc). In the above example, as x is 7, the loop breaks and the previous appointment of p to 6 is trained and printed thus the result is: **6**

```
let mut q = 0;
for x in 1..10 {
   if x % 2 == 0{
      continue;
   }
   q = x;
   println!("{}", q);
}
```

We use the same program with some changes. Continue doesn't terminate the loop, rather just skips the condition mentioned by the user int he if statement. Here, as we can see, we have used the same program as the one used for break but with a slight alterations. We can see how continue gets to print the odd numbers only from 1..10 as we have provided the condition that if x % 2 == 0 (remainder is zero) then continue on that iteration. The result for this program is:

```
1
3
5
7
9
```

For nested loops with complications, Rust provides the user with a mechanism to escape out of the loop. Although, break and continue are used for that, however Break and Continue can. Only Break or Continue the corresponding loop, not all the loops. Hence, to allow the user to exercise greater control over exiting the entire loop structure (providing an efficient mechanism to jump to defined labels). The below program depicts this mechanism:

```
let mut x = 3;
'outer: while true{
   let mut y = 6;
   'nested: while true{
      if ( x >= 6){
         break 'outer
      }
      println!("({}, {})", x, y);
      x += 1;
   }
   y += 1
}
```

This mechanism is a form of goto in Rust but with a different mechanism; by using this program we exit out of the outer loop/break the outer most loop and as expected we continue with the next instruction following the loop. **While true** is an infinite loop and we can see how it breaks entirely. The result is:

```
(3, 6)
(4, 6)
(5, 6)
```

A section that includes your evaluation of these languages in terms of readability and writability of logicall-controlled loops. Write a paragraph discussing, in your opinion, which language is the best for logicall-controlled loops.. Explain why.

Starting this evaluation with Dart, I would outline it beforehand that Dart is fairly easier to read and write for a user. The syntax for the while loop is very basic and in line with most of the common programs languages, thus adding to its writability. Dart also uses left and right brackets { } to mark the start and end of functions which makes it more readable comparatively. The for loop also has the basic syntax as the other languages and Dart also has

inbuilt do-while loop which makes it easier for the user to implement these loops, thus making it more writable. Dart also has break, continue, and the mechanism to seep out from nested loops with defined labels. This makes it more compatible with usage for programmers as they don't need to emanate these commands with complex program structures, thus making it more writable.

JavaScript is a very useful language for the usage of logically loops and user located control mechanisms. It can be assessed that the syntax for logically controlled loops in JavaScript is almost the same to the syntax of other commonly used programming languages, thus adding to its writability. JavaScript also has a primitive do-while mechanism, hence one doesn't need to emanate it, making it more readable and writable. It also has almost all the defined user controlled mechanisms: break, continue, and the mechanism to seep out from nested loops with defined labels, thus making it more writable for the user to exercise programming. It also makes use of semi colons (it is the user's choice, they can choose to or not use semi colons to mark the end of instructions) as well as comments to educate the user more about preceding statements, thus adding to its readability; not only this, each command also requires left and right parentheses such as the print one which makes it clear to the reader/user that what is to be printed as the concerned boundaries between the input of the command is depicted clearly and so is the parameter that the command takes in, adding to its readability.

LUA is a fairly straight forward language that is very consist with the general structure of most of languages. Lua also has defined while and repeat..until (do-while) loops which adds to its writability. The syntax for while and repeat..until are however not very similar to the syntax of general programming languages. It doesn't use left and right brackets to mark the start of loops bur rather uses "do" as start and "end" as an indication towards the closure of a loop, an aspect that adds to its readability. LUA has two of the tree user located control mechanisms:

Break and the mechanism to seep out from nested loops with defined labels; these aspects ultimately add to its writability.

PHP is a relatively different language than the others ones I have analyzed as its printing command is very different from the ones I have yet encountered thus making it more complex as well as less-writable. PHP requires the declaration of the variable name with a $ sign which is a repetition one doesn't require/tiresome, thus reducing its writability. PHP has the crucial logically controlled posttest and pretest loops thus one can use them imminently without emanating them through complex structures, making it more writable and readable. It also has almost all the defined user controlled mechanisms: break, continue, and the mechanism to seep out from nested loops with defined labels, thus making it more writable for the user to exercise programming. PHP also uses left and right brackets '{ }' to mark the start and end of loops hence making it more readable

Python is also a relatively straight forward language. Python has the crucial logically controlled posttest and pretest loops however one needs to emanate the do-while loop to implement it, collectively making it less writable and readable. It also has almost all the defined user controlled mechanisms: break, continue, but doesn't have the mechanism to seep out from nested loops with defined labels; however, collective it adds to its writability and readability. It doesn't have semi colons to mark the end of instructions/statements along with no requirement to mark the start or end of functions, which collectively inhibits its readability. It doesn't use left and right. Rackets to mark the end of loops, inhibiting its readability.

Ruby is also a language that is fairly easier to write and read; Ruby has the crucial logically controlled loops (pretest and posttest) with a fairly similar syntax to the general ones yet with considerable differences. However, it as primitive do-while form hence adds to its writability. It doesn't have a primitive mechanism to seep out of nested loops bu has Break and Next to allow the user to have corneal over loops, hence adding to its writability but also inhibiting it

with no goto mechanism. The general operators are same for Ruby such as the assignment operator, hence keeping it fairly simple and consistent with the general structure of programming languages, hence adding to its writability. The output to console command is puts with no left and right parentheses and hence might make one wonder about its functionality, thus inhibiting its readability. The dollar sign needed for variable assignment also inhibits its writability as one can forget and is not consistent with the general syntax of most common programming languages.

Rust is a more complex language, at least for me. Rust has one of the essential logically controlled loops, the while loop and doesn't have the do-while loop primitively, this reduces its writability. The printing command is not similar to that go other languages and thus reduces its writability and readability. The general syntax for he while loop is consent with other languages though. However, It has almost all the defined user controlled mechanisms: break, continue, and the mechanism to seep out from nested loops with defined labels, thus making it more writable for the user to exercise programming. Rust uses left and right brackets '{ }' to mark the start and end of loops along with semi-coons to mark the end of statements, hence making it more readable. The for loop in Rust is also very different from the ones in common programming languages but is fairly straightforward and simple.

My Choice:

Logically controlled Loops and Usr Located Control Mechanism make up an essential part of one's programs when one is immolating them. It I essential to have them ready at hand and rather not emanate them through other mechanisms; thus, my choice is Dart. Dart is a fairly straightforward language that has a very defined syntax for loops in terms of readability and writability. It has all the user control mechanism already present (primitively) and also has a very generic Syntax for for, while, and do-while loops. Furthermore, as for reading, it has all the prime features to keep the instruction segments separate from one another as well as makes use of left and right parentheses along with left and right brackets to mark the start and end of

parameter in take and loop bodies, thus making it very readable and understandable. It has a main function which is very useful in terms of defining global variables/functions, if one required.

A section about learning strategy:

I found this homework fairly straightforward yet a bit confusing. One was supposed to do a lot of searching for the syntax, loop and especially for goto availability. I was able to find reliable sources for this homework and with the help of online compilers, I was able to test each program for reach part specifically. I didn't consult anyone's help as I fond this homework fairly easier to understand at least.
My strategy was mostly starting searching the loop and then reading articles for it, and after that trying out the programs on online compiling platforms. This helped organize my work.
Next, I found websites where most of the language's syntax and loops etc were readily available to study: https://www.geeksforgeeks.org is from where I found most of the stuff for Ruby, https://www.programmersought.com/article/57021496877/ for LUA, https://programming-idioms.org/idiom/42/continue-outer-loop for Dart, Rust, and others.
Programming idioms couldn't assist me that much did its part in this homework. https://stackoverflow.com/questions/743164/how-to-emulate-a-do-while-loop-in-python I also sought help from Stack-Overflow for any component that I couldn't find easily.
A website where I could find most f the compote ts of this homework for most of the languages is: https://www.tutorialspoint.com/index.htm; this website was very helpful and şt had most of the stuff I required for this homework. https://www.programiz.com is also a ver helpful website which also elaborates on what is being taught/asked, thus I also sought help from this website especially for User Located Control Mechanism. https://www.w3schools.com/js/js_loop_for.asp was very helpful to find the loops for Python, JavaScript, and PHP. I was able to find all the stuff that I have mentioned in this document on the internet.

Online Compilers:

https://www.tutorialspoint.com/execute_lua_online.php for Lua
https://www.writephponline.com for PHP
https://dartpad.dev/? For Dart
https://repl.it/languages/ruby?v2=1 for Ruby
https://repl.it/languages/python3?v2=1 for Python
https://repl.it/languages/nodejs?v2=1 for JS
https://repl.it/languages/rust?v2=1 for Rust