CS 315 – Homework 2
Logically Controlled Loops in Dart, JavaScript, Lua, PHP, Python, Ruby and Rust

Maryam Shahid
21801344
Section 01
December 13, 2020

Logically Controlled Loops in:

Dart

Dart has both pretest and posttest locations for logically controlled loops. The control expression in both must be Boolean.

Pretest:

Dart provides two types of logically controlled loops where the test is done before the loop. The first is through a while loop and the other is via for loop.

In a while loop, the test is carried out before the loop. It is used to execute instructions in a block as long as the control expression evaluates to true. My code evaluates this in the following manner:

```
var first = 1;
while (first < 10) {
    print("first:  $first");
    first = first * 2;
}
```

```
Output: first: 1
        first: 2
        first: 4
        first: 8
```

The code above declares a variable first and assigns it the value 1. It then reaches a logically controlled while loop which tests the control expression before entering the loop. Since 1 is less than 10, the instructions inside the loop are evaluated. It prints the current value of first and then multiplies it by 2. The loop is iterated as long as the control expression evaluates to true.

Dart also allows for loop as a logically controlled loop. This can be done in the following way:

```
var i = 1;
for (; i < 10; ) {
  print("i:  $i");
  i = i * 2;
}
```

```
Output: i: 1
        i: 2
        i: 4
        i: 8
```

The above code initializes a variables i as 1. It then creates a for loop with only a boolean control statement (otherwise it would have been a counter controlled loop). The condition is tested before the block of instruction is executed. In this case, as 1 is less than 10, the instructions inside the loop are evaluated. It prints the current value of i and then multiplies it by 2. The loop is iterated as long as the control expression evaluates to true.

Posttest:

The do-while loop in Dart is similar to while, however it tests the control expression after the instructions have been executed. Since the test is done after the instructions, they are always executed at least once. This can be done in the following way:

```
var second = 10;
do {
  print("second: $second");
  second = second – 4;
} while (second > 1);

Output: second: 10
        second: 6
        second: 2
```

The code above declares a variable second and assigns it a value of 10. It then reaches and enters the logically controlled loop do-while. The instructions declared in the block are executed which consist of printing the value of second and subtracting 4 from it. After the instructions are executed, it reaches the control expression. In case it evaluates to true, the loop is executed again. This occurs until the control expression returns false which in this case will take place when second is greater than 1.

User-located Loop Control Mechanisms:

Dart provides unconditional unlabeled exits – break, which allow the user to take control of a loop and exit it. The break statement terminates the most internal loop and it can be used in the following way:

```
var count = 10;
while (count > 0) {
  if (count == 5)
      break;
  print("count: $count");
  count--;
}

Output: count: 10
        count: 9
        count: 8
        count: 7
        count: 6
```

The above code segment declares a variable count and assigns it the value 10. It then reaches a while loop with a control expression that allows iteration as long as count is greater than 0. Generally, the loop would go from 10 to 0, printing each value and decrementing it. However, it makes use of "break" using an if statement which allows the user to exit the loop when the value of count is equal to 5. Thus, the output shows the value of count from 10 to 6 only since it exits the loop at 5.

Dart also provides an unlabeled control statement – continue, which skips over the remainder of the current iteration, but does not terminate the loop. It can be demonstrated in the following way:

```
var check = 5;
while (check > 0) {
  check--;
  if (check == 2)
    continue;
  print("check: $check");
}

Output: check: 4
        check: 3
        check: 1
        check: 0
```

The above code declares a value check and initializes it to 5. It then enters a while loop which executes the instructions as long as check is greater than 0. Generally, the loop would iterate through each vale of check, decrement its value and print it. However, the user takes control of the loop using "continue" in an if condition. This allows us to skip through the remainder of the loop when check is equal to 2, but continue when it is not. Thus, the output prints each value of check from 4 till 0 but skips over 2.

JavaScript

JavaScript also allows both pretest and posttest locations for logically controlled loops. The control expression can be an arithmetic or variable since the condition is evaluated and converted to a boolean.

Pretest:

JavaScript also provides two types of logically controlled loops where the test is done before the loop. The first is through a while loop and the other is via for loop.

In a while loop, the control expression is tested before entering the loop. It is used to execute instructions in a block as long as the expression evaluates to true. It can be done in the following way:

```
var first = 1;
while (first < 10) {
    document.write("first:  " + first + '<br>');
    first = first * 2;
}
```

The code above declares a variable first and assigns it the value 1. It then reaches a logically controlled while loop which tests the control expression before entering the loop. Since 1 is less than 10, the instructions inside the loop are evaluated. It prints the current value of first and then multiplies it by 2. The loop is iterated as long as the control expression evaluates to true.

JavaScript also allows for loop as a logically controlled loop. This can be done in the following way:

```
var i = 1;
for (; i < 10; ) {
  document.write("i:  " + i + '<br>');
  i = i * 2;
}

Output: first: 1
        first: 2
        first: 4
        first: 8
```

The above code initializes a variables i as 1. It then creates a for loop with only a boolean control statement (otherwise it would have been a counter controlled loop). The condition is tested before the block of instruction is executed. In this case, as 1 is less than 10, the instructions inside the loop are evaluated. It prints the current value of i and then multiplies it by 2. The loop is iterated as long as the control expression evaluates to true.

Posttest:

The do-while loop in JavaScript is similar to while, however it tests the control expression after the instructions have been executed. Since the test is done after the instructions, they are always executed at least once. This is done in the following way:

```
var second = 10;
do {
  document.write("second:  " + second + '<br>');
  second = second – 4;
} while (second > 1);

Output: second: 10
        second: 6
        second: 2
```

The code above declares a variable second and assigns it a value of 10. It then reaches and enters the logically controlled loop do-while. The instructions declared in the block are executed which consist of printing the value of second and subtracting 4 from it. After the instructions are executed, the control expression is evaluated. In case it returns true, the loop is executed again.

This occurs until the control expression returns false which in this case will take place when second is greater than 1.

User-located Loop Control Mechanisms:

JavaScript provides unconditional unlabeled exits – break, which allow the user to take control of a loop and exit it. The break statement terminates the most internal loop and it can be used in the following way:

```
var count = 10;
while (count > 0) {
  if (count == 5)
      break;
  document.write("count:  " + count + '<br>');
  count--;
}

Output: count: 10
        count: 9
        count: 8
        count: 7
        count: 6
```

The above code segment declares a variable count and assigns it the value 10. It then reaches a while loop with a control expression that allows iteration as long as count is greater than 0. Generally, the loop would go from 10 to 0, printing each value and decrementing it. However, it makes use of "break" using an if statement which allows the user to exit the loop when the value of count is equal to 5. Thus, the output shows the value of count from only 10 to 6 since it exits the loop at 5.

JavaScript also provides an unlabeled control statement – continue, which skips over the remainder of the current iteration, but does not terminate the loop. It can be demonstrated in the following way:

```
var check = 5;
while (check > 0) {
  check--;
  if (check == 2)
    continue;
  document.write("check:  " + check + '<br>');
}

Output: check: 4
        check: 3
        check: 1
        check: 0
```

The above code declares a value check and initializes it to 5. It then enters a while loop which executes the instructions as long as check is greater than 0. Generally, the loop would iterate through each vale of check, decrement its value and print it. However, the user takes control of the loop using “continue” in an if condition. This allows us to skip through the remainder of the loop when check is equal to 2, but continue when it is not. Thus, the output prints each value of check from 4 till 0 but skips over 2.

Lua

Lua allows both pretest and posttest locations for logically controlled loops. The control expression can be an arithmetic or a variable since the condition is evaluated and converted to a boolean.

Pretest:

Lua provides a while loop which tests the control expression before entering the loop. It is used to execute a group of instructions as long as the control expression evaluates to true. It can be done in the following way:

```
first = 1;
while (first < 10)
do
    print(first:", first)
    first = first * 2;
end

Output: first: 1
        first: 2
        first: 4
        first: 8
```

The code above declares a variable first and assigns it the value 1. It then reaches a logically controlled while loop which tests the control expression before entering the loop. Since 1 is less than 10, the instructions inside the loop are evaluated. It prints the current value of first and then multiplies it by 2. The loop is iterated as long as the control expression evaluates to true.

Posttest:

Lua provides a repeat.. until loop which tests the control expression after the instructions have been executed, and repeats until the expression evaluates to true (different from the general syntax of languages). Since the test is done after the instructions, they are always executed at least once. The following code gives an example of the loop:

```
second = 3;
repeat
  print("second:", second);
  second = second – 1;
until (second < 1);
```

```
Output: second: 3
        second: 2
        second: 1
```

The code above declares a variable second and assigns it a value of 3. It then reaches and enters the logically controlled repeat.. until loop. The group of instructions are then executed which consist of printing the value of second and decrementing it. After the execution, the control expression is evaluated. The loop only terminates once the control expression evaluates to true, which in this case will take place when second is lesser than 1. This can be noted in the output.

User-located Loop Control Mechanisms:

Lua provides unconditional unlabeled exits – break, which allow the user to take control of a loop and exit it. The break statement terminates the most internal loop and it can be used in the following way:

```
count = 10
while (count > 0)
do
  if (count == 5)
  then
      break
  end
  print("count:", count);
  count = count - 1;
end
```

```
Output: count: 10
        count: 9
        count: 8
        count: 7
        count: 6
```

The above code segment declares a variable count and assigns it the value 10. It then reaches a while loop with a control expression that allows iteration as long as count is greater than 0. Generally, the loop would go from 10 to 0, printing each value and decrementing it. However, it makes use of "break" using an if statement which allows the user to exit the loop when the value of count is equal to 5. Thus, the output shows the value of count from only 10 to 6 since it exits the loop at 5.

Lua goes not contain the generic continue statement since it creates conflicts with the way the language manages lexical scoping.

PHP

PHP allows both pretest and posttest locations for logically controlled loops. The control expression can be an arithmetic or a variable since the condition is evaluated and converted to a boolean.

Pretest:

PHP provides two types of logically controlled loops where the test is done before the loop. The first is through a while loop and the other is via for loop.

In a while loop, the test is carried out before the loop. It is used to execute instructions in a block as long as the control expression evaluates to true as shown in the code below:

```
$first = 1;
while ($first < 10) {
    echo("first: $first <br>");
    $first = $first * 2;
}

Output: first: 1
        first: 2
        first: 4
        first: 8
```

The code above declares a variable first and assigns it the value 1. It then reaches a logically controlled while loop which tests the control expression before entering the loop. Since 1 is less than 10, the instructions inside the loop are evaluated. It prints the current value of first and then multiplies it by 2. The loop is iterated as long as the control expression evaluates to true.

PHP also allows for loop as a logically controlled loop. This can be done in the following way:

```
$second = 1;
for (; $second < 10; ) {
  echo("second:  $second <br>");
  $second = $second * 2;
}

Output: second: 1
        second: 2
        second: 4
        second: 8
```

The above code initializes a variables second as 1. It then creates a for loop with only a boolean control statement (otherwise it would have been a counter controlled loop). The condition is tested before the block of instruction is executed. In this case, as 1 is less than 10, thus the instructions inside the loop are evaluated. It prints the current value of second and then multiplies it by 2. The loop is iterated as long as the control expression evaluates to true.

Posttest:

The do-while loop in PHP is similar to while. However, it tests the control expression after the instructions have been executed. Since the test is done after the instructions, they are always executed at least once. This can be done in the following way:

```
$third = 10;
do {
  echo("third: $third <br>");
  $third = $third – 4;
} while ($third > 1);

Output: third: 10
        third: 6
        third: 2
```

The code above declares a variable third with a value of 10. It enters the logically controlled do-while loop. The instructions declared in the block are executed which consist of printing the value of second and subtracting 4 from it. After the instructions are executed, it reaches the control expression. In case it evaluates to true, the loop is executed again. This occurs until the control expression returns false which in this case will take place when third is greater than 1.

User-located Loop Control Mechanisms:

PHP provides unconditional unlabeled exits – break, which allow the user to take control of a loop and exit it. The break statement terminates the most internal loop and it can be used in the following way:

```
$count = 10;
while ($count > 0) {
  if ($count == 5) break;
  print("count: $count <br>");
  $count--;
}

Output: count: 10
        count: 9
        count: 8
        count: 7
        count: 6
```

The above code segment declares a variable count and assigns it the value 10. It then reaches a while loop with a control expression that allows iteration as long as count is greater than 0. Generally, the loop would go from 10 to 0, printing each value and decrementing it. However, it makes use of “break” using an if statement which allows the user to exit the loop when the value of count is equal to 5. Thus, the output shows the value of count from 10 to 6 only since it exits the loop at 5.

PHP also provides an unlabeled control statement – continue, which skips over the remainder of the current iteration, but does not terminate the loop. It can be demonstrated in the following way:

```
$check = 0;
while ($check < 5) {
  $check++;
  if (check == 2) continue;
  print("check: $check");
}

Output: check: 1
        check: 3
        check: 4
        check: 5
```

The above code declares a value check and initializes it to 0. It enters a while loop which executes the instructions as long as check is lesser than 5. Generally, the loop would iterate through each vale of check, increment its value and print it. However, the user takes control of the loop using "continue" in an if condition. This allows us to skip through the remainder of the loop when check is equal to 2, but continue when it is not. Thus, the output prints each value of check from 1 till 5 but skips 2.

Python

Python only allows logically controlled loops with pretest control expressions. The control expression can be arithmetic or a variable since the condition is evaluated and converted into boolean.

Pretest:

Python provides a while loop which tests the control expression before entering the loop. It is used to execute a group of instructions as long as the control expression evaluates to true. It can be done in the following way:

```
first = 1
while first < 10:
    print(first:", first)
    first *= 2;

Output: first: 1
        first: 2
        first: 4
        first: 8
```

The code above declares a variable first and assigns it the value 1. It then reaches a logically controlled while loop which tests the control expression before entering the loop. Since 1 is less than 10, the instructions inside the loop are evaluated. Once the loop is entered, it prints the current value of first and then multiplies it by 2. The loop is iterated as long as the control expression evaluates to true.

User-located Loop Control Mechanisms:

Python provides unconditional unlabeled exits – break, which allow the user to take control of a loop and exit it. The break statement terminates the most internal loop and it can be used in the following way:

```
count = 10
while count > 0:
  if count == 5:
      break
  print("count:", count)
  count = count - 1

Output: count: 10
        count: 9
        count: 8
        count: 7
        count: 6
```

The above code segment declares a variable count and assigns it the value 10. It then reaches a while loop with a control expression that allows iteration as long as count is greater than 0. Generally, the loop would go from 10 to 0, printing each value and decrementing it by 1. However, it takes control by calling "break" using an if statement which allows the user to exit the loop when the value of count is equal to 5. Thus, the output shows the value of count only from 10 to 6 as it exits the loop at 5.

Python also provides an unlabeled control statement – continue, which skips over the remainder of the current iteration, but does not terminate the loop. It can be demonstrated in the following way:

```
check = 5
while check > 0:
  check = check - 1
  if check == 2:
    continue
  print(“check:”, check);
}

Output: check: 4
        check: 3
        check: 1
        check: 0
```

The above code declares a value check and initializes it to 5. It then enters a while loop which executes the instructions as long as check is greater than 0. Generally, the loop would iterate through each vale of check, decrement its value and print it. However, the user takes control of the loop using "continue" in an if condition. This allows us to skip through the remainder of the loop when check is equal to 2, but continue when it is not. Thus, the output prints each value of check from 4 till 0 but skips over 2.

Python also provides a pass statement which creates an empty loop. It is used when a statement is required inside a loop but it does not have to necessarily do something. It can be done in the following way:

```
for name in "Maryam Shahid":
  pass
print(name)

Output: d
```

The code above has a for loop which goes through each name in the string "Maryam Shahid". However, since the loop has a pass statement, it does nothing. Once the loop is terminated, it prints the last letter stored in name – d.

Ruby

Ruby allows both pretest and posttest locations for logically controlled loops in multiple ways. The control expression must be a boolean.

Pretest:

Ruby provides a while loop which tests the control expression before entering the loop. It is used to execute a group of instructions as long as the control expression evaluates to true. The conditional expression is separated from the instructions using the reserved word do. It can be done in the following way:

```
$first = 1;
while $first < 10 do
  puts("first:  #$first")
  $first *= 2
end

Output: first:  1
        first:  2
        first:  4
        first:  8
```

The code above declares a variable first and assigns it the value 1. It then reaches a logically controlled while loop which tests the control expression before entering the loop. Since 1 is less than 10, the instructions inside the loop are evaluated. It prints the current value of first and then multiplies it by 2. The loop is iterated as long as the control expression evaluates to true.

Similarly, Ruby provides an until statement which also tests the control expression before entering the loop. However, contrary to while, it executes the instructions as long as the control expression evaluates to false. The conditional expression is separated from the instructions using the reserved word do. It is done in the following way:

```
$second = 3
until $second < 1 do
  puts("second:  #$second")
  $second -= 1
end

Output: second:  3
        second:  2
        second:  1
```

The code above declares a variable second and assigns it the value 3. It then reaches a logically controlled until loop which tests the control expression before entering the loop. Since 3 is not less than 1, the condition evaluates to false and instructions inside the loop are evaluated. It prints the current value of second and decrements it by 1. The loop is iterated as long as the control expression evaluates to false.

Posttest:

Ruby provides a while modifier which tests the control after the loop. In this case, the while modifier follows a begin statement. It executes the code as long as the control expression evaluates to true. Since the test is done after the instructions, they are always executed at least once. The following code gives an example of the loop:

```
$third = 10
begin
  puts("third:  #$third")
  $third -= 4
end while $third > 1

Output: third:  10
        third:  6
        third:  2
```

The code above declares a variable third with value 10. It then reaches and enters the logically controlled while loop with the begin statement. The group of instructions are executed which consist of printing the value of third and decrementing it by 4. After the execution, the control expression is evaluated. The loop iterates as long as the control expression is true, which in this case will take place when third is greater than 1. This can be noted in the output.

Similarly, Ruby also provides an until modifier which tests the control expression after executing the loop. In this case . In this case, the until modifier follows a begin statement. It executes the code as long as the control expression evaluates to false. Since the test is done after the instructions, they are always executed at least once. The following code gives an example of the loop:

```
$fourth = 5
begin
  puts("fourth:  #$fourth")
  $fourth += 2
end until $fourth > 10

Output: fourth:  5
        fourth:  7
        fourth:  9
```

The code above declares a variable fourth with value 5. It then reaches and enters the logically controlled until loop with the begin statement. The group of instructions are executed which consist of printing the value of fourth and incrementing it by 2. After the execution, the control expression is evaluated. The loop iterates as long as the control expression is false, which in this case will take place when fourth is greater than 10. This can be noted in the output.

User-located Loop Control Mechanisms:

Ruby provides unconditional unlabeled exits – break, which allow the user to take control of a loop and exit it. The break statement terminates the most internal loop and it can be used in the following way:

```
$count = 10
while $count > 1 do
  if $count == 5 then
    break
  end
  puts("count:  #$count")
  $count -= 1
end

Output: count:  10
        count:  9
        count:  8
        count:  7
        count:  6
```

The above code segment declares a variable count and with value 10. It then reaches a while loop with a control expression that allows iteration as long as count is greater than 1. Generally, the loop would go from 10 to 1, printing each value and decrementing it. However, it makes use of “break” using an if statement which allows the user to exit the loop when the value of count is equal to 5. Thus, the output only shows the value of count from 10 to 6 as it exits the loop at 5.

Ruby also provides an unlabeled control statement – next, which skips over the remainder of the current iteration, but does not terminate the loop. It can be demonstrated in the following code:

```
$check = 5
while check > 0 do
  $check -= 1
  if check == 2 then
```

```
    next
  puts("check: #$check")
end

Output: check: 4
        check: 3
        check: 1
        check: 0
```

The code above declares a value check and initializes it to 5. It then enters a while loop which executes the instructions as long as check is greater than 0. Generally, the loop would iterate through each vale of check, decrement its value and print it. However, the user takes control of the loop using "next" in an if condition. This allows us to skip through the remainder of the loop when check is equal to 2, but continue when it is not. Thus, the output prints each value of check from 4 till 0 but skips 2.

Ruby also allows a redo statement which restarts iteration of the loop without checking loop condition. It can be demonstrated in the following way:

```
$rest = 6
while $rest >= 3 do
  $rest -= 1
  if $rest == 3 then
    redo
  end
  puts("rest: #$rest")
end

Output: rest: 5
        rest: 4
        rest: 3
        rest: 1
```

The above code declares a variable rest with the value 6. It enters a logically controlled while loop since rest is greater than 3. The loop decrements the value by 1 and prints the value of rest. Normally, the loop would exit once the value of rest is lesser than 3. However, using the redo statement with the condition 3 allows us to repeat the loop even when the value of rest is 2. This decrements the value of rest further and prints 1 as seen in the output.

Rust

Rust only allows logically controlled loops with pretest control expressions. The control expression must be boolean.

Pretest:

Rust provides a while loop which tests the control expression before entering the loop. It is used to execute the block of instructions as long as the control expression evaluates to true. It can be done in the following way:

```
let mut first = 1;
while first < 10 {
    println!(“first: {}", first);
    first *= 2;
}

Output: first: 1
        first: 2
        first: 4
        first: 8
```

The code above declares mutable variable first and with the value 1. It then reaches a logically controlled while loop which tests the control expression before entering the loop. Since 1 is less than 10, the instructions inside the loop are evaluated. Once the loop is entered, it prints the current value of first and then multiplies it by 2. The loop is iterated as long as the control expression evaluates to true.

User-located Loop Control Mechanisms:

Rust provides unconditional unlabeled exits – break, which allow the user to take control of a loop and exit it. The break statement terminates the most internal loop and it can be used in the following way:

```
let mut count = 10;
while count > 1 {
  if count == 5{
      break;
  }
  println!("count: {}", count);
  count -= 1;
}

Output: count: 10
        count: 9
        count: 8
        count: 7
        count: 6
```

The above code segment declares a mutable variable count and assigns it the value 10. It then reaches a while loop with a control expression that allows iteration as long as count is greater than 1. Generally, the loop would go from 10 to 0, printing each value and decrementing it by 1. However, it takes control by calling “break” using an if statement which allows the user to exit the loop when the value of count is equal to 5. Thus, the output shows the value of count only from 10 to 6 as it exits the loop at 5.

Rust also provides an unlabeled control statement – continue, which skips over the remainder of the current iteration, but does not terminate the loop. It can be demonstrated in the following way:

```
let mut check = 5
while check > 0 {
  check -= 1
  if check == 2 {
    continue;
  }
  println!("check: {}", check);
}

Output: check: 4
        check: 3
        check: 1
        check: 0
```

The above code declares a mutable value check and initializes it to 5. It then enters a while loop which executes the instructions as long as check is greater than 0. Generally, the loop would iterate through each vale of check, decrement its value and print it. However, the user takes control of the loop using "continue" in an if condition. This allows us to skip through the remainder of the loop when check is equal to 2, but continue when it is not. Thus, the output prints each value of check from 4 till 0 but skips over 2.

Evaluation of Languages

Dart

Dart provides a flexible and readable syntax for logically controlled loops. Since it is a C-like language, it also allows usage of for loop as logically controlled. It requires a main function for the code which adds to readability. For loops such as while and do while, it requires the condition to be written in brackets () and the block of instructions in curly brackets {}. This greatly enhances readability and writability of the code. It requires semicolons at the end of the statements, and also to marks the end of do while loop. This further enhances the readability of the code. Since it allows logically controlled loops with both pretests and posttests, it enhances writability of the code. It also allows the generic break and continue control statements which allow user-located loop control mechanisms. This greatly increases writability of the language.

JavaScript

JavaScript has a similar syntax to Dart for logically controlled loops. It provides square brackets ( ) for control expressions and writes instructions for the loop in a block using curly brackets {}. This greatly enhances the readability of the code. It provides both pretest and posttest logically controlled loops. Although it is not a C-like language, it allows us to use for loop as a logically controlled loop. This enhances writability of the code. It also allows statements like break and continue which provide mechanisms for user-located loop control. This adds to the writability of the language.

Lua

As compared to the rest, Lua lacked flexibility for logically controlled loops and had a difficult syntax. The language does not use brackets for blocks of instructions, but uses them for conditional statements. This inconsistency lowers the readability of the language. The block of instructions within a loop is difficult to follow through. It uses "do" and "end" to mark the beginning and end of a while loop, and uses "then" and "end" to mark if and else conditions. This inconsistency again reduces the readability of the language. The lack of semicolons at the end of statements also reduces readability. The repeat.. until loop also has a different format than most languages. It terminates the loop when the conditional statements results in true. I only figured this out once my code entered an infinite loop. This negatively impacts the writability of the language. Similarly, it does not have many user-located loop control mechanisms which again affects the writability of the language.

PHP

PHP follows the generic syntax when it comes to logically controlled loops. It uses square brackets () to mark conditional statements and curly brackets {} to mark blocks of instructions within the loop. This adds to the readability of the language. It uses a dollar sign $ to declare all variables within the language, which again adds to the readability. It provides logically controlled loops with pretests and posttests. Since it is a C-like language, it also allows users to use a for loop as a logically controlled loop as well. This adds to the writability of the language.

It contains the generic break and continue statements for user-located loop control mechanisms which add to the writability of the language.

Python

Since python only allows logically controlled loops with pretests, it lacks writability. However, although it does not make use of any type of brackets, the consistent indentation within blocks adds to the writability of the language. It provides user-located loop control mechanisms such as continue, break and pass which enhances the readability and writability of the language.

Ruby

Ruby provides a lot of flexibility in terms of logically controlled loops. It allows both pretest and posttest logically controlled loops. Since it is a C-like language, it also allows for loop to be used as a logically controlled loop. It also allows variations of while and until so that they may be used to evaluate instructions when the conditional statement is required to be true, as well as false depending on the program. This greatly adds to the writability of the language. Ruby also provides the most user-located loop control mechanisms such as break, next, redo. All of these enhance the writability of the language. Although the language does not use brackets for conditional statements and blocks within loops, the clear indentation within blocks adds to the readability of the language.

Dart

Dart lacks writability for logically controlled loops since it only allows loops with pretests. Moreover, dart only allows immutable variables thus in order to change variables within loops, they need to be specified using the mut keyword. This also affects the readability and writability of the language. Dart does not use brackets for control expressions, but uses them for blocks of instructions within loops. This inconsistency also negatively affects the readability of the language. However, the language allows user-located loop control mechanisms in terms of break and continue which enhances writability.

My Choice:

Out of all, I prefer Ruby to be the best suited for logically controlled loops. It follows a clear syntax which makes it readable, and it also provides the most flexibility for logically controlled loops. It has variations for while and until, so that they can be used as best suited for the program. It also has multiple user-located loop control mechanisms which greatly enhances writability of the language. Thus, I prefer Ruby out of all other languages to be the best for logically controlled loops.

Learning Strategy

Dart

I went through Dart's official website and found a valuable tour of the language to help me get started with the language. [1] I used the online compiler provided by Dart's official website as Dart Pad to run and test my programs. [2] I found valuable resources at tutorial's point and w3adda which helped me understand the mechanics and syntax of logically controlled loops. [3] [4]

JavaScript

For JavaScript, I used the online compiler js.do to run and test my codes. [5] I was quite familiar with the language so started my research directly with logically controlled loops. I found useful documentation of the language's control statements on the official documentation provided by Mozilla. [6] I also took help from w3 schools to further build upon my understanding of logically controlled loops in JavaScript. [7]

Lua

To better understand Lua, I went through the manual provided by Lua's official website. [8] I also found useful information on their official website about the language's loops and kept it as a guideline while writing my own codes. [8] To develop a deeper understanding of the langauge's logically controlled loops, I used the examples provided by tutorial's point. [9] Finally, to compile and test my codes, I used the online compiler provided by repl.it. [10]

PHP

To get started with the language, I used the manual provided by PHP's official website to understand its basic syntax. [11] I found a useful tutorial about logically controlled loops in PHP at tutorial's point which further helped me understand the task. [12] Finally, I used the online compiler provided by writephponline to test my codes. [13]

Python

Since I was already familiar with Python, I started off by searching on the language's approach to logically controlled loops. I found suitable resources in the manual provided by python's official website. [14] And took further help from w3schools to enhance my understanding of logically controlled loops and control statements in the language. [15] Since I had already worked in Python before, I had it installed on my computer so did not feel the need to use online compilers.

Ruby

To start working on logically controlled loops in Ruby, I started researching on tutorial's point [16]. However, to further expand my understanding of the language's logically controlled loops, I used the examples on geeksforgeeks. [17] To test and run my codes, I used the online compiler provided by repl.it. [18]

Rust

To refresh my understanding of Rust, I went through the of examples “Rust by example” provided by Rust’s official website. [19] The website also contained valuable information about loops in the language. [20] I used those along with examples provided by tutorial’s point to strengthen my understanding of logically controlled loops in the language. [21] I used the compiler provided by Rust’s official website to run my codes. [22]

References

[1] https://dart.dev/guides/language/language-tour

[2] https://dartpad.dev

[3] https://www.w3adda.com/dart-tutorial/dart-loops

[4] https://www.tutorialspoint.com/dart_programming.htm

[5] https://js.do

[6] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements

[7] https://www.w3schools.com/jsref/jsref_while.asp

[8] https://www.lua.org/pil/

[9] https://www.tutorialspoint.com/lua/lua_loops.htm

[10] https://repl.it/languages/lua

[11] https://www.php.net/manual/en/

[12] https://www.tutorialspoint.com/php/php_loop_types.htm

[13] https://www.writephponline.com

[14] https://docs.python.org/3/tutorial/controlflow.html

[15] https://www.w3schools.com/python/python_while_loops.asp

[16] https://www.tutorialspoint.com/ruby/ruby_loops.htm

[17] https://www.geeksforgeeks.org/ruby-loops-for-while-do-while-until/#whileLoop

[18] https://repl.it/languages/ruby.

[19] https://doc.rust-lang.org/stable/rust-by-example/

[20] https://doc.rust-lang.org/reference/expressions/loop-expr.html

[21] https://www.tutorialspoint.com/rust/rust_loop.htm

[22] https://play.rust-lang.org