Department Of Computer Engineering

CS 315 Programming Languages

Homework 2

Berdan Akyürek
21600904
Section 2

1. Sample Codes and Results

For each example, the program tries to find a number between 0 and 100 by creating random numbers and keeps trying until the number is found. It prints every assumption to the console and prints the number of tries until the number is found at the end.

1.1 Dart

In the code below, do-while loop is used in order to make assumptions. But Dart also provides while and for loops too. It is possible to implement the same operation using these loops as well. Also Dart has break and continue statements. Do-while loop is used because there is no need for a pretest loop. A posttest loop is more useful in this case since there is no assumption unti the first cycle of the loop. But Dart language both supports pretest(for, while) and posttest(do-while) loops. Also it has break and continue statements for mid-test.

1.1.1 Code

```
import 'dart:math';

void main() {
 int number = 30;
 int tries = 0;
 var rnd = new Random();
 var assumption;
 do {
   assumption = rnd.nextInt(100);
   print(assumption);
   tries++;
 } while (assumption != number);
 print("$tries tries.");
}
```

1.1.2 Output

```
14
13
30
3 tries.
```

1.2 Javascript

In the code below, do-while loop is used in order to make assumptions. But JavaScript also provides while and for loops too. It is possible to implement the same program using these loops as well. Also JavaScript has break and continue statements. Do-while loop is used because there is no need for a pretest loop. A posttest loop is more useful in this case since there is no assumption until the first cycle of the loop. But JavaScript language both support pretest(while, for) and posttest(do-while) loops. Also it has break and continue statements for mid-test.

1.2.1 Code

```
var number = 30;
var assumption;
var tries = 0;
do {
 assumption = Math.floor(Math.random() * 100);
 tries++;
 console.log(assumption);
}
while (assumption != number);
console.log(tries, " tries");
```

1.2.2 Output

48
8
9
60
17
6
55
78
91
47
63
58
26
7
54
32
90

75
43
84
70
65
75
66
38
14
64
74
25
77
69
46
69
53
15
91
58
5
30
39 tries

1.3 Lua

In the code below, repeat-until loop is used in order to make assumptions. This structure is similar to do-while which tests the condition after the execution, but differently, it checks the inverse of the given condition. Lua also provides while and for loops too. It is possible to implement the same program using these loops as well. Also Lua has a break statement. Repeat-until loop is used because there is no need for a pretest loop. A posttest loop is more useful in this case since there is no assumption until the first cycle of the loop. But Lua language both supports pretest(while, for) and posttest(repeat-until) loops. Also it has a break statement for mid-test.

1.3.1 Code

```
number = 30
tries = 0
math.randomseed(os.time())

repeat
   assumption = math.random(0,100)
   tries = tries + 1
```

```
   print(assumption)
until(assumption == number)
print(tries, " tries")
```

1.3.2 Output

2
55
17
50
94
87
2
8
6
22
83
14
82
80
20
97
42
30
18 tries

1.4 PHP

In the code below, do-while loop is used in order to make assumptions. But PHP also provides while and for loops too. It is possible to implement the same program using these loops as well. Also PHP has break and continue statements. Do-while loop is used because there is no need for a pretest loop. A posttest loop is more useful in this case since there is no assumption until the first cycle of the loop. But PHP language both support pretest(while, for) and posttest(do-while) loops. Also it has break and continue statements for mid-test.

1.4.1 Code

```
<?php
$number = 30;
$tries = 0;
do {
   $assumption = rand ( 0 , 100 );
   print $assumption;
```

```
   print "\n";
   $tries ++;
} while ($assumption != $number);

print $tries;
print " tries\n";
?>
```

1.4.2 Output

```
32
7
87
35
22
49
12
73
72
60
20
95
26
14
76
92
55
92
35
5
29
63
26
78
14
20
71
23
42
30
30 tries
```

1.5 Python

In the code below, while loop is used in order to make assumptions. But Python also provides for loops too. It is possible to implement the same program using for loop as well. Also Python has break and continue statements. A posttest loop is more useful in this case since there is no assumption until the first cycle of the loop. While loop is used because there are not any posttest loops in Python. Python language only supports pretest(while, for) loops. Also it has break and continue statements for mid-test.

1.5.1 Code

```
import random
number = 30
tries = 0
assumption = -1
while(assumption != number):
   assumption = random.randint(0,100)
   print(assumption)
   tries += 1
print(tries, " tries.")
```

1.5.2 Output

```
86
19
22
43
20
27
50
39
65
92
73
73
31
50
39
30
16  tries.
```

1.6 Ruby

In the code below, until loop is used in order to make assumptions. This structure is similar to while which tests the condition before the execution, but differently, it checks the inverse

of the given condition. Ruby also provides for and while loops too. It is possible to implement the same program using for or while loop as well. Also Ruby has break, next, retry, redo statements. While loop is used because there are not any posttest loops in Ruby. A posttest loop is more useful in this case since there is no assumption until the first cycle of the loop. Ruby language only supports pretest(while, for) loops. Also it has break, next, retry, redo statements for mid-test.

1.6.1 Code

```
number = 30
tries = 0
assumption = -1
until assumption == number do
   assumption = rand(1..100)
   tries += 1
   print(assumption, "\n")
end
print(tries, " tries\n")
```

1.6.2 Output

```
14
5
23
16
88
31
20
78
41
59
49
93
34
21
83
97
90
30
18 tries
```

1.7 Rust

In the code below, while loop is used in order to make assumptions. But Rust also provides for loops too. It is possible to implement the same program using for loop as well. Also Rust has break and continue statements. While loop is used because there are not any posttest loops in Rust. A posttest loop is more useful in this case since there is no assumption until the first cycle of the loop. Rust language only supports pretest(while, for) loops. Also it has break and continue statements for mid-test. And lastly, an unconditioned loop can be created with the loop keyword. It simply does the same thing with “while(true)”. The only way to finish this loop is using control statements.

1.7.1 Code

```
use rand::Rng;
fn main(){
   let number = 30;
   let mut tries = 0;
   //let mut rng = rand::thread_rng();
   let mut assumption = -1;
   while assumption != number {
       assumption = rand::thread_rng().gen_range(0, 100);
       tries = tries + 1;
       println!("{}",assumption);
   }
   println!("{} {}",tries, "tries");
}
```

1.7.2 Output

89
32
82
36
1
42
5
25
30
9 tries

2. Language Evaluations

The evaluations for the languages are as below. In my opinion, Dart language is the best for logically-controlled loops. Because it provides pretest, posttest and midtest loops. Also it is easy to read and write loops because it is simple and similar to well known languages.

2.1 Dart

2.1.1 Readability

Dart is readable since it is simple. It does not have much variety of loops and statements related to loops. Also its syntax is similar to well known languages and it has meaningful keywords which are easy to remember while reading.

2.1.2 Writability

Dart is easy to write since it has enough built in loops and statements.

2.2 Javascript

2.2.1 Readability

Javascript is readable since it is simple. It does not have much variety of loops and statements related to loops. Also its syntax is similar to well known languages and it has meaningful keywords which are easy to remember while reading.

2.2.2 Writability

Javascript is easy to write since it has enough built in loops and statements.

2.3 Lua

2.3.1 Readability

Lua is not a readable language in logically controlled loops because it is not simple since there are a variety of keywords like do and done. In a big program with nested loops, these keywords may be confusing.

2.3.2 Writability

Lua is not a writable language since it is not readable. The variety of keywords are confusing during the writing process and hard to follow.

2.4 PHP

2.4.1 Readability

PHP is an easy to read language since it is simple. It does not have much variety of loops and statements related to loops. Also its syntax is similar to well known languages and it has meaningful keywords which are easy to remember while reading.

2.4.2 Writability

PHP is not a writable language since it has different rules that may be forgotten. For example, forgetting to use $ is possible and it is also hard to understand and remember where to use it. Since logically controlled loops are controlled with variables, writability of loops also decreases.

2.5 Python

2.5.1 Readability

Since Python does not have posttest loops, this reduces readability. The programs that require posttest loops may require additional variables which make the program hard to follow.

2.5.2 Writability

Writability of Python in terms of logically controlled loops is low since its readability is also low. Also, indentation is mandatory for loops and an indentation mistake can lead to unexpected errors. Modern text editors and IDEs mostly solve this problem but writing Python with a simpler text editor is hard.

2.6 Ruby

2.6.1 Readability

Keywords like end make Ruby hard to read because they become complicated when there are many such words especially in nested loops.

2.6.2 Writability

Ruby is not a writable language since it is not readable. The variety of keywords are confusing during the writing process and hard to follow.

2.7 Rust

2.7.1 Readability

Rust is the least readable language among all of them because it is not simple and it does not support posttest loops. It is not simple because keywords like let and mut are required to control loops and they are confusing.

2.7.2 Writability

Since use of let and mut are hard, it is possible to forget the usage and face with errors. Also the lack of posttest loops make it harder to implement in conditions where a posttest loop is needed.

3. Learning Strategy

I used linux compilers and interpreters for this homework. No online compiler/interpreter is used. The list of the compilers used are as follows:

- Dart SDK 2.10.4
- Node v12.19.0
- Lua 5.4.0
- PHP 7.4.5
- Python 3.8.4

- Ruby 2.7.1p83
- Rustc 1.48.0

While doing this assignment, I started with a research about logically controlled loops. After checking the course textbook and online sources, I understood the difference between logically controlled and counter controlled loops. Later, I designed a program that required a logically controlled loop and implemented in Python because that was the most familiar language for me. After implementing it on Python, I researched what are the types of loops in each language and implemented similar programs in each of these languages.