

CS315 - HOMEWORK 2

LOGICALLY CONTROLLED LOOPS

Introduction	1
Location of tests	2
Logically-controlled pretest loops	2
Logically-controlled midtest loops	2
Logically-controlled posttest loops	2
Logically-controlled loops in different programming languages	3
Dart Language	3
JavaScript Language	5
PHP Language	7
Python Language	8
Perl Language	11
Evaluation of these five languages	13
My learning strategy for the topic	13
Sources and languages	13
Communication	14
Tools	14
Conclusion	14

Introduction

Loops, in a broad sense, can be held under control in two ways: enumeration-controlled loops and logically-controlled loops. These may be called “bounded/unbounded loops” or “definite/indefinite iteration loops”. Both types of loops have a collection of statements inside of them, however, the enumeration-controlled loops execute these statements a number of times, while the logically-controlled loops execute them repeatedly until some Boolean condition changes its value. This Boolean condition is called “control condition”, and it can be defined, or changed at any place of code.

These control conditions must be checked in the loop, in order to understand when it should stop repeating. The place of this test is important, since it can change the whole flow of the code.

Location of tests

Control statements, i.e., tests can take place at the beginning, middle, and the end of the loop. The control condition must be checked side of each iteration, otherwise, the code may iterate infinitely, so that the code goes into infinite loop.

Logically-controlled pretest loops

Logically-controlled pretest loops basically check the exit condition *before* each iteration. The most popular use for pretest loops may be the **for** loop structure, but of course there are also many other examples.

Logically-controlled midtest loops

On the flow of the code, if the control statement does not take place either at the beginning or at the end of the loop, then it can be called a midtest loop. A small fraction of programming languages supports native logically-controlled midtest loops, for example, the *Ada* language has an **exit when...** statement. The *Java* language, also, has labeled breaks, which supports this type of loops. However, it is possible to do it by simply adding **break** statements in the middle of the code.

Logically-controlled posttest loops

Logically-controlled posttest loops basically check the exit condition *after* each iteration. The most popular use of this type may be the **do...while** loop structure, however, there are also many other instances.

Logically-controlled loops in different programming languages

As we discussed before, the three types of logically-controlled loops are pretest, midterm, and posttest loops. From now on, these three types of loops will be observed by implementing them in 5 different programming languages, *Dart*, *JavaScript*, *Perl*, *PHP*, and *Python*. Some properties may not be included in every language, since otherwise it would be too verbose (for example, the `do...while` loop structure will not be shown in every language).

Dart Language

The bunch of code below would be enough to observe the logically-controlled pretest loops in the Dart language. Note that although it is not implemented in the code, **for** loop structure is also a neat example for pretest loops.

```
main()
{
    /*
     * LOGICALLY-CONTROLLED PRETEST LOOP
     */
    print("---LOGICALLY-CONTROLLED PRETEST LOOP---");

    var list1 = [1, 2];
    print("list = $list1");

    int number1 = 3;

    while(list1.length < 2)
    {
        list1.add(number1);
        number1++;
    }

    print("list = $list1");
    print("number = $number1");

    /*
     * LOGICALLY-CONTROLLED MIDTEST LOOP
     */
    print("---LOGICALLY-CONTROLLED MIDTEST LOOP---");
```

```

var list2 = [1, 2];
print("list = $list2");

int number2 = 3;

while(true)
{
    list2.add(number2);

    if(list2.length >= 2)
    {
        break;
    }

    number2++;
}

print("list = $list2");
print("number = $number2");

/*
 * LOGICALLY-CONTROLLED POSTTEST LOOP
 */
print("---LOGICALLY-CONTROLLED POSTTEST LOOP---");

var list3 = [1, 2];
print("list = $list3");

int number 3 = 3;

do
{
    list3.add(number3);
    number3++;
}while(list3.length < 2);

print("list = $list3");
print("number = $number3");
}

```

This code consists of 3 parts, each of them for one of the three types of logically-controlled loops. Here follows an image of the final output of the whole program.

As it can be seen, in the pretest loop, neither the elements of the list nor the number is changed, since the boolean control statement is executed (loop tested) at the very beginning of the loop, and since it returns a falsy value, the loop did not iterate.

```
---LOGICALLY-CONTROLLED PRETEST LOOP---  
list = [1, 2]  
list = [1, 2]  
number = 3  
---LOGICALLY-CONTROLLED MIDTEST LOOP---  
list = [1, 2]  
list = [1, 2, 3]  
number = 3  
---LOGICALLY-CONTROLLED POSTTEST LOOP---  
list = [1, 2]  
list = [1, 2, 3]  
number = 4
```

In the midtest loop, a new element is added to the list, however, the number is not changed. Because, the boolean control statement is executed (loop tested) between the lines which adds a new element to the list and increases the number.

Finally, in the posttest loop, both the list and the number have been changed, since the test occurs at the end of both lines.

JavaScript Language

Here follows the JavaScript code, to make observations on. Note that some properties are not implemented, since we have already discussed them.

```
for (i = 0; i < 10; i++)  
{  
    console.log(i);  
    if (i == 3)  
    {  
        break;  
    }  
}  
  
var number = 0;
```

```
do
{
    number++;
} while (number != 5)

console.log("NUMBER = " + number);

// LABELED BREAK
loopExit:
for (i = 0; i < 3; i++)
{
    for (x = 0; x < 4; x++)
    {
        if (x == 2)
        {
            break loopExit;
        }
        console.log("inner!");
    }
    console.log("outer!");
}

console.log("---LABELED LOOP FINISHED---");

// UNLABELED BREAK
for (i = 0; i < 3; i++)
{
    for (x = 0; x < 4; x++)
    {
        if (x == 2)
        {
            break;
        }
        console.log("inner!");
    }
    console.log("outer!");
}

console.log("---UNLABELED LOOP FINISHED---");
```

```
inner!  
inner!  
---LABELED LOOP FINISHED---  
inner!  
inner!  
outer!  
inner!  
inner!  
outer!  
inner!  
inner!  
outer!  
---UNLABELED LOOP FINISHED---
```

Here a **for** loop and a **do...while** loop has been implemented in order to demonstrate the differences between pretest and posttest loops. Moreover, the 'labeled break' structure of JavaScript has been implemented at the end of the code as well.

To understand better, here is an image of the output of only that part of the code. Without labels, the break statement would finish only the nested loop, thus, the outer loop would still continue to iterate. Fortunately, by using the

label statements, we can find a solution for this problem.

PHP Language

As we discussed in the JavaScript language part above, nested loops lead to some problems when we use **break** and **continue** statements inside them. PHP has an awesome solution for this problem, in the PHP language, the break statement can take parameters. Here follows a bunch of HTML code which partly includes a few lines of PHP code.

```
<html>  
<head>  
<title>CS315 HW1 21902730</title>  
</head>  
<body>  
<?php  
$namesOf315 = array("Cagatay", "Irmak", "Halil", "Altay");  
$girlNames = array("Ayse", "Irmak", "Nuna");  
  
foreach ($namesOf315 as $value1)  
{  
    echo "$value1 <br>";  
  
    foreach ($girlNames as $value2)  
    {  
        if ($value1 == $value2)  
        {  
            break 2;  
        }  
    }  
}
```

```

        }
    }
}
?>
</body>
</html>

```

In this bunch of code, there are two new things about the main topic. First of all, as we were discussing recently, the **break** statement has an integer parameter of **2**, which indicates how many nested loops it will stop. An image of the output of the

```

$php main.php
<html>
<head>
<title>CS315 HW1 21902730</title>
</head>
<body>
Cagatay <br>Irmak <br></body>
</html>

```

above HTML code is on the left. As it can be seen, the outer loop iterates for the first element 'Cagatay' and the second element 'Irmak'. When it is still iterating for 'Irmak', the **break** statement was executed, so that both loops stopped.

Also, just like many other programming languages, the PHP language has a **foreach** loop structure. The point of this loop structure is to iterate a bunch of code as many times as the length of an array, while using the elements of the same array inside the loop, with respect to their indices.

Python Language

Although there is nothing different from the other languages which we have already discussed, we can still discuss how pretest, midtest, and posttest loops are included in the language, and how we can logically control these loops. Below is the code to observe how Python deals with loops. Please note that the code is manually separated into 5 sections, the **else** keyword in loops, simple loops, nested loops, the **break** keyword, and the **pass** keyword.

```

from __future__ import print_function

print("\n---WHILE/FOR ELSE STRUCTURE---")

count = 0
while (count < 3):

```

```
        count = count + 1
        print("a")
    else:
        print("b")

    for i in range(1, 4):
        print(i)
    else:
        print("THE END")

number = 30

while (number == 30):
    print("yay")
    number = number+1
    break
else:
    print("nah") # see if it prints...
print("FINAL NUMBER = ", number)

print("\n---SIMPLE FOR LOOP---")

n = 4
for i in range(0, n):
    print(i)

print("\n---SIMPLE NESTED LOOPS---")

for i in range(1, 7):
    for j in range(i):
        print(i, end=' ')
    print()

print("\n---BREAK---")

for currentLetter in 'cagataysafak':
    if currentLetter == 't' or currentLetter == 'f':
        break
print ('Last Letter :', currentLetter )

print("\n---PASS---")

for currentLetter in 'halilaltayguvenir':
    pass
print ('Last Letter :', currentLetter )
```

```

---WHILE/FOR ELSE STRUCTURE---
a
a
a
b
1
2
3
THE END
yay
FINAL NUMBER = 31

---SIMPLE FOR LOOP---
0
1
2
3

---SIMPLE NESTED LOOPS---
1
2 2
3 3 3
4 4 4 4
5 5 5 5 5
6 6 6 6 6 6

---BREAK---
Last Letter : t

---PASS---
Last Letter : r
> |

```

Firstly, in the Python language, logically-controlled pretest loops, just as **while** loop, can have an **else** part as well. Note that the codes in this **else** part can be executed for only one iteration, i.e., it is not a loop. If a **break** is used in the first part of a **while-else** loop, then the code will not go into the **else** part. On the left, there is an image of the output, where we can see that only **yay** is printed, while **nah** is not printed.

For the second and the third parts, as we can see, unlike a majority of the programming languages, we do not have to write **++i** inside the for loop.

For the fourth part, since the loop is broken when the **currentLetter** is **t**, so that the program prints **t** in this part.

Finally, for the fifth part, the **pass** keyword is used for empty loops. Nevertheless, it is not the same with **continue** keyword, since it does not finish the current iteration and go into the next iteration, it simply performs no operation in that line, and the iteration goes on. The program

continues reading the input string **halilaltayguvenir** until the end, and prints the last letter, which is **r**.

Here we have seen logically-controlled pretest and midtest loops in the Python language. As for the logically-controlled posttest loops, unfortunately, Python does not have **do...while** loops. However, we can manually build them in the same way as we built logically-controlled midtest loops, checking (testing) the control condition at the end of the iteration.

Perl Language

Before starting, I would like to share some thoughts about the philosophy of the Perl language. In the Perl language, there is always more than one way to do something. It always provides an alternative way to do things.

Perl supports almost every C-style loop, just like the **while** loop structure. However, there is another loop structure, which is called **until** loop. This is the exact opposite of the **while** loop. This reminds me of the **unless** statement in the Perl language, which is the exact opposite of the **if** statement. Moreover, the Perl language has another loop structure called **do...unless** loop, which is the exact opposite of the **do...while** loop, as you can guess.

To observe these keywords, statements, and structures, let us examine the code below.

```
print "---UNTIL LOOP---";

$a = 10;

until($a < 1)
{
    print "$a ";
    $a = $a - 1;
}

print "\n---WHILE LOOP---";

$a = 10;

while($a >= 1)
{
    print "$a ";
    $a = $a - 1;
}

print "\n---THE NEXT KEYWORD---";

$a = 11;

do{
    $a--;
```

```

        next if $a == 5;
        print "$a ";
    }}until($a < 2);

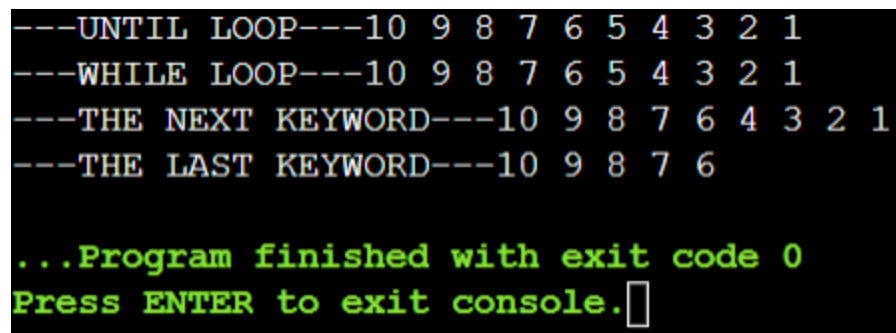
print "\n---THE LAST KEYWORD---";

$a = 11;

loop1:
{
    do{
        $a--;
        last if $a == 5;
        print "$a ";
    }until($a < 2);
}

```

Below is an image of the output of this code. As it can be easily seen, the **until** and the **while** loops are doing the exact same process.



```

---UNTIL LOOP---10 9 8 7 6 5 4 3 2 1
---WHILE LOOP---10 9 8 7 6 5 4 3 2 1
---THE NEXT KEYWORD---10 9 8 7 6 4 3 2 1
---THE LAST KEYWORD---10 9 8 7 6

...Program finished with exit code 0
Press ENTER to exit console.

```

The philosophical additions into traditional loop structures in the Perl language is not limited to just these ones, there are also some other conventions which are specific to Perl, just as the **next** and **last** keywords.

In the third part of the code, the program does not complete the 6th iteration, where the number 5 was to be printed. This is because of the use of the **next** keyword.

Also, in the fourth part, the program finishes the whole loop at the 6th iteration, where the number 5 was to be printed. This is because of the use of the **last** keyword.

Evaluation of these five languages

Some followed the traditions, some did not include even the most used ones, some added new ways to do so. However, they all achieved to implement logically-controlled loops. Personally, the Dart language is not built with an aim to use control loops, or other elements of the algorithm flow, but it is built for developers to write beautiful front-end code. Python and Perl have some modernist solutions. Python and JavaScript have labeled loops, which are, in my opinion, solid structures. Actually, PHP would have been my favorite one due to the break statements which take integer parameters. However, because of all the innovativity and the labeled loops, Python wins this competition.

The special keywords in Perl language decreases readability and writability, in my opinion, since they are different from the traditional uses, which are being used by a huge majority of computer programmers.

My learning strategy for the topic

Sources and languages

Learning how a loop works in a programming language seems like an easy thing to a computer science student, however, I thought that it must become harder than this, because otherwise, I would not learn anything important from this homework. So, I started to analyze the philosophies lying behind the programming languages, and what are their main concerns and goals. Because I am a part-time employed Flutter developer for approximately 4 months, I was good at Dart. For JavaScript, I followed [techtopia.com](https://www.techtopia.com). For PHP, I checked Professor's sample codes in the dijkstra server, and followed [the official website of PHP](https://www.php.net). Python was the most difficult one for me because I am not used to this language as much as other languages. The mini-articles I found on [GeeksForGeeks](https://www.geeksforgeeks.org) were very useful to understand the language. Finally, for the Perl language, all I want to say is that I love Perl. This language is beautiful, but exploring what else I can learn is more beautiful. I followed the documentation at [Perl Tutorial](https://perldoc.perl.org).

Also, in order to understand the main concept of logically-controlled loops rather than the programming languages, I followed our textbook, and some slides from various universities, such as [Florida State University](#), [University of Pisa](#), and [Adelphi University](#).

Communication

I have not had any kind of communication with others, I found every single source by myself. This may be the first homework in my whole university education that I didn't even tell my friends.

However, other than my friends, I read many topics on the online forums, and websites like [StackOverflow](#). I am not sure if this can be considered under the tag of 'Communication', but that's what I did.

Tools

In order to write the code and see the result, I used many different web sites and tools. For the Dart language, I used [DartPad](#). For JavaScript, I used [Programiz JavaScript Online Compiler](#). For PHP, I used both [Coding Ground](#) and [w3schools](#), since one of them is constantly lagging. For Python, I used [Programiz Python Online Interpreter](#). And for the Perl language, I used [OnlineGDB Perl Compiler](#).

After writing programs in five languages by using these online computers, I checked them on the University's dijkstra server, of course.

Conclusion

To sum up, if we do not want to go into an infinite loop while coding, then we need to check (test) loops to learn when they should stop. These tests can occur at the beginning, middle, or the end of the loop, this is the way they are named: pretest, midtest, and posttest loops. Also, these tests can be either by enumeration or by logically controlling.