CS 315 Homework 2

Short-Circuit Evaluation in Dart, JavaScript, PHP, Python, and Rust

Student Name: Esad İsmail Tök

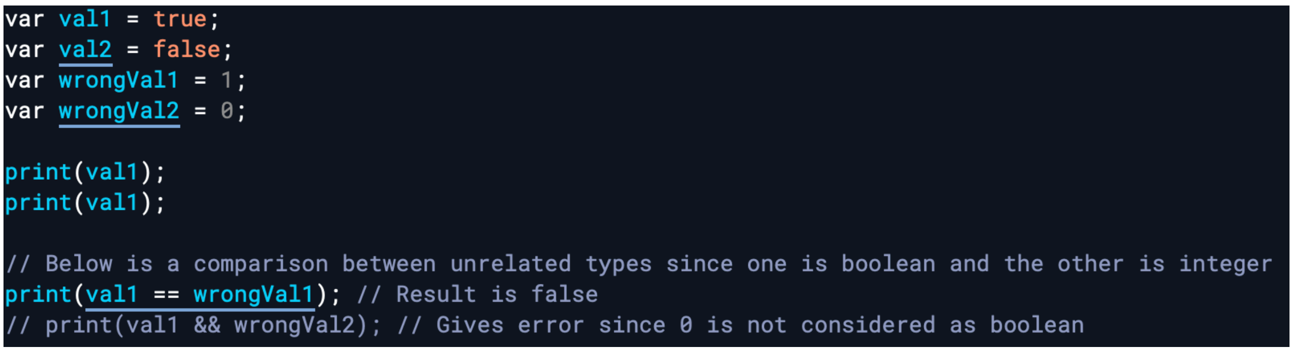
Student ID: 21801679

Section: 3

Dart Language

Design Issue 1: How are the boolean values represented?

- Dart language has a primitive boolean type. Boolean values are represented as **true** and **false** in Dart language. Both true and false values are compile-time constants and they cannot be represented as 0 or 1 or by another type in a program.



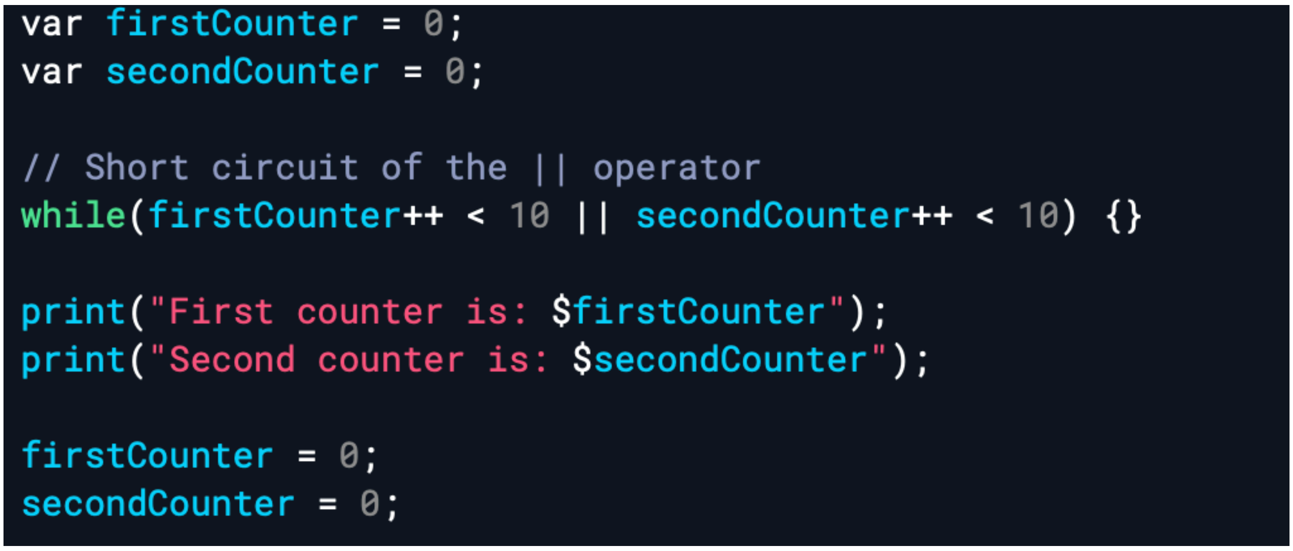
```
var val1 = true;
var val2 = false;
var wrongVal1 = 1;
var wrongVal2 = 0;

print(val1);
print(val1);

// Below is a comparison between unrelated types since one is boolean and the other is integer
print(val1 == wrongVal1); // Result is false
// print(val1 && wrongVal2); // Gives error since 0 is not considered as boolean
```

Figure 1: Representation of boolean values in Dart Language

- *Figure 1* shows that 0 and 1 cannot be used as a boolean value and they cannot be substituted as boolean values. Strings or other types are also not applicable.

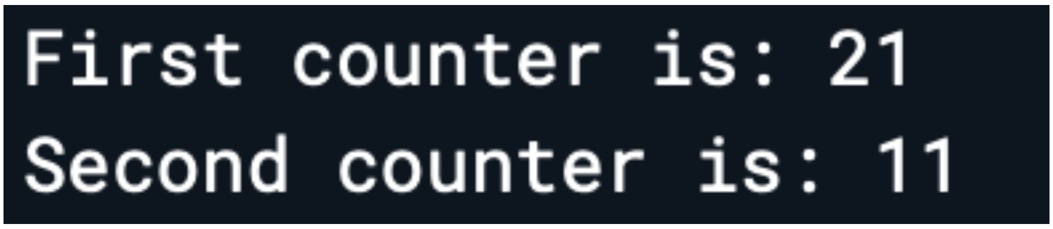
Design Issue 2: What operators are short-circuited?

- Dart language has short circuit evaluation the operators AND (&&) and OR (||) in Dart are short circuited.



```
var firstCounter = 0;
var secondCounter = 0;

// Short circuit of the || operator
while(firstCounter++ < 10 || secondCounter++ < 10) {}

print("First counter is: $firstCounter");
print("Second counter is: $secondCounter");

firstCounter = 0;
secondCounter = 0;
```

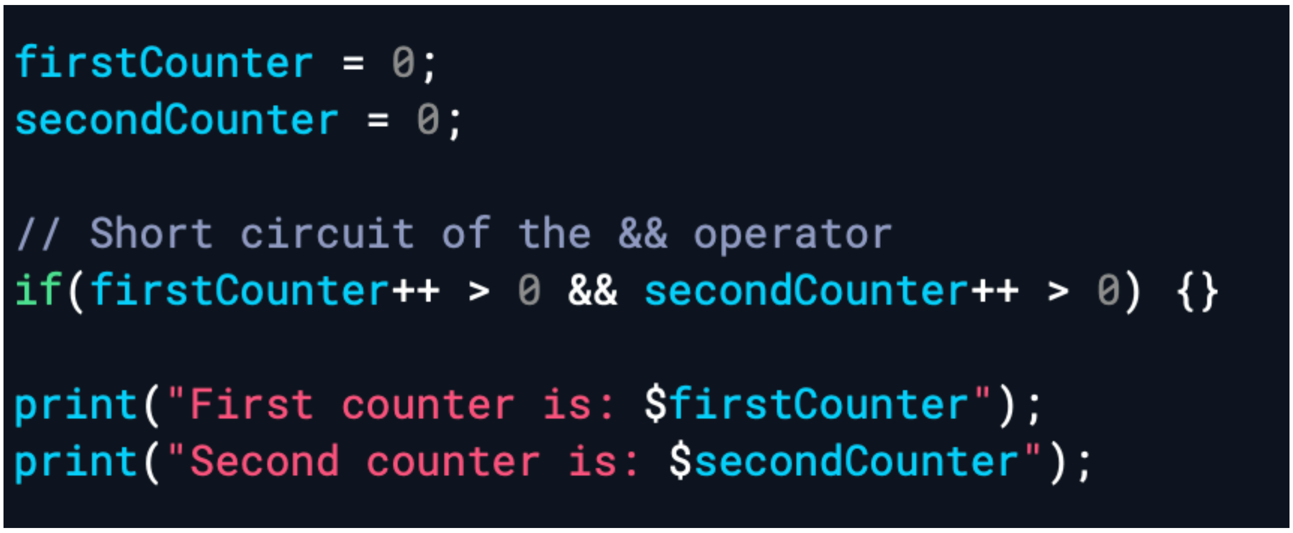
Figure 2: Short circuit OR operator in Dart Language



```
First counter is: 21
Second counter is: 11
```

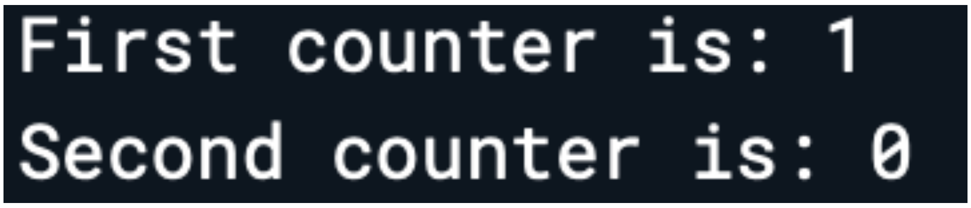
Figure 3: Output of short circuit OR operator in Dart Language

- *Figure 2 and 3* show that in the first 10 iterations the second counter is not incremented since the program finds the value **true** in the first expression and short-circuit happened since the rest of expressions do not affect the result of the or operation.



```
firstCounter = 0;
secondCounter = 0;

// Short circuit of the && operator
if(firstCounter++ > 0 && secondCounter++ > 0) {}

print("First counter is: $firstCounter");
print("Second counter is: $secondCounter");
```

Figure 4: Short circuit AND operator in Dart Language



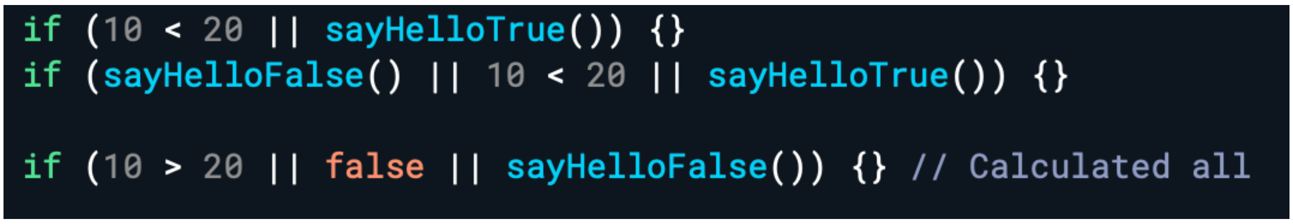
```
First counter is: 1
Second counter is: 0
```

Figure 5: Result of Short circuit AND operator in Dart Language

- *Figure 4 and 5* show that the and operation needs only one false value to conclude the calculation and since the first false is found in the first expression, the second counter is never incremented.

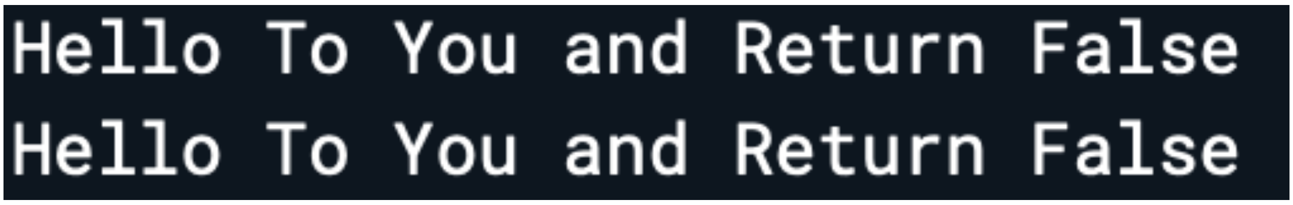
Design Issue 3: How are the results of short-circuited operators computed? (Consider also function calls)

- The boolean expressions are started to be calculated from the left and whenever the required boolean value if found, short circuit happens and the rest of the expressions are not calculated. The same case is also valid for the function calls. If there is any expression that can conclude the boolean expression, then the function is not called because of the short-circuit.



```
if (10 < 20 || sayHelloTrue()) {}
if (sayHelloFalse() || 10 < 20 || sayHelloTrue()) {}

if (10 > 20 || false || sayHelloFalse()) {} // Calculated all
```

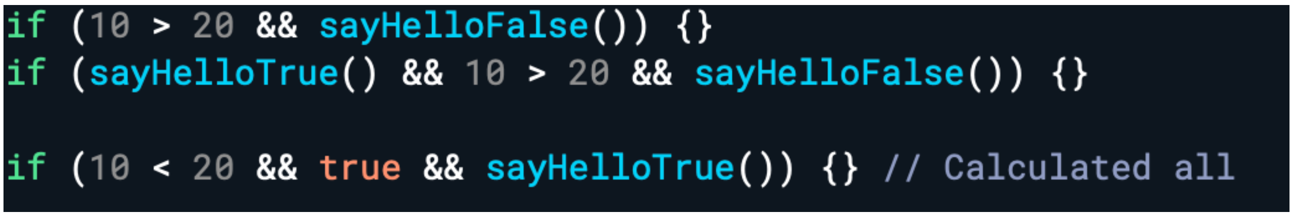
Figure 6: Computation of short circuit OR in Dart Language



```
Hello To You and Return False
Hello To You and Return False
```

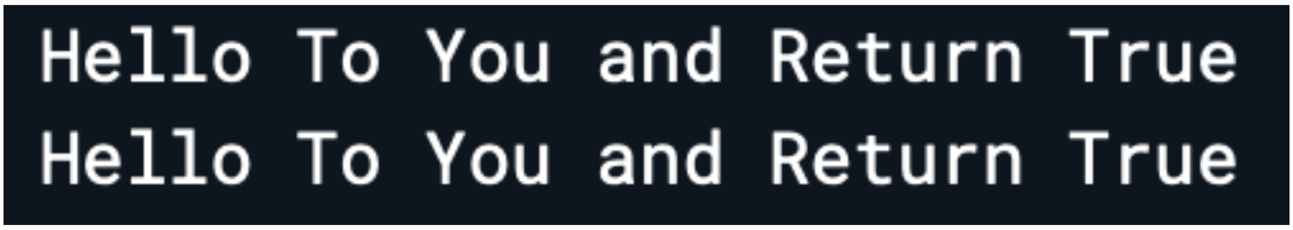
Figure 7: Output of computation of short circuit OR in Dart Language

- *Figure 6 and 7* shows that the function in the first if statement is not called because the expression “10 < 20” is true and it is enough for the OR operation to be concluded. In the second if statement, the first function is called because the program had not reached a true value and after it reached it in the second expression of the second if statement, sayHelloTrue() function is not called since short-circuit happened. In the last if statement all expressions are calculated since no true value is obtained beforehand.



```
if (10 > 20 && sayHelloFalse()) {}
if (sayHelloTrue() && 10 > 20 && sayHelloFalse()) {}

if (10 < 20 && true && sayHelloTrue()) {} // Calculated all
```

Figure 8: Computation of short circuit AND in Dart Language



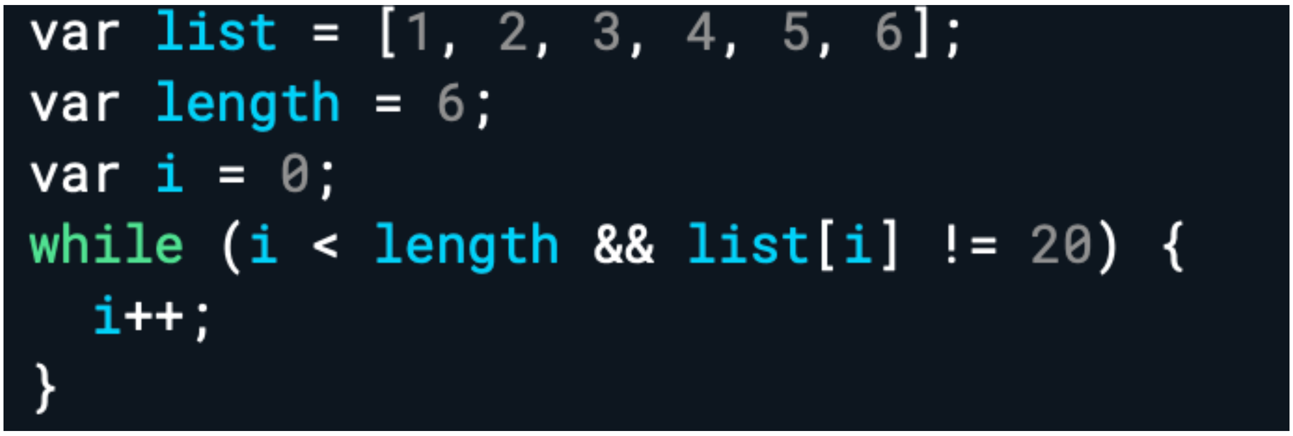
```
Hello To You and Return True
Hello To You and Return True
```

Figure 9: Output of computation of short circuit AND in Dart Language

Figure 8 and 9 shows that the function in the first if statement is not called because the expression “10 > 20” is false and it is enough for the AND operation to be concluded. In the second if statement, the first function is called because the program had not reached a false value and after it reached it in the second expression of the second if statement, sayHelloFalse() function is not called since short-circuit happened. In the last if statement all expressions are calculated since no false value is obtained beforehand.

Design Issue 4: What are the advantages about short-circuit evaluation?

- Short circuit evaluation enables the program to avoid computing extra boolean expressions when it is not necessary for the final result. For example if there is an AND relation between the relational expressions, the program can conclude the result as false whenever it finds the value of false in one of the relational expressions starting from the left. And it is unnecessary to calculate the rest since the result is not affected by them. In very small calculations such as the boolean expressions that include only 2 or 3 relational expressions, the benefit can be negligible since boolean calculations are considerably fast. However if we have thousands of relational expressions in our calculation, short-circuit can save us from a big calculation overhead.
- Moreover, I believe the most important advantage of short-circuit evaluation is avoiding the side effects of the remaining expressions. When the necessary checks are done in the previous expressions and then the short circuit happens, we are sure that the program will not calculate the rest of the expressions and the side effects that might be caused by them are avoided.



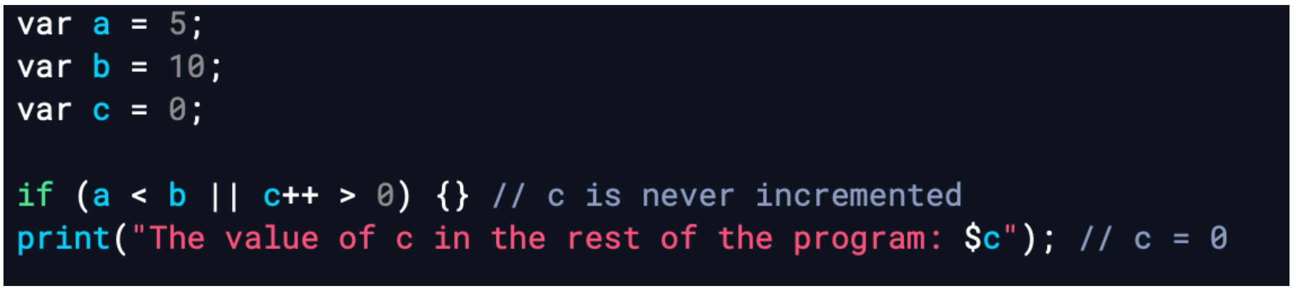
```
var list = [1, 2, 3, 4, 5, 6];
var length = 6;
var i = 0;
while (i < length && list[i] != 20) {
  i++;
}
```

Figure 10: Advantages of short-circuit evaluation in Dart Language

 - In *Figure 10* we can see that the side effect of index out of bound error is avoided by the short-circuit evaluation.

Design Issue 5: What are the potential problems about short-circuit evaluation?

- First of all, the programmer should always think twice when he/she wants to change a variables value in the relational expression. Because that expression might or might not be calculated according to the result of the previous relational expressions in the condition.
- Also if the remaining program depends on a variable that may or may not be changes inside a relational expression, then it is not safe to use that variable. So it reduces the reliability of the program.



```
var a = 5;
var b = 10;
var c = 0;

if (a < b || c++ > 0) {} // c is never incremented
print("The value of c in the rest of the program: $c"); // c = 0
```

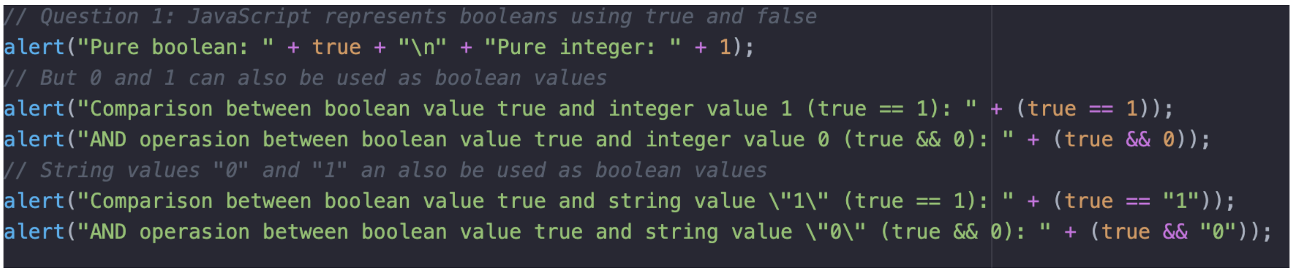
Figure 11: Disadvantages of short-circuit evaluation in Dart Language

- Figure 11 shows how we cannot trust on the value of c in the rest of the program.

JavaScript Language

Design Issue 1: How are the boolean values represented?

- JavaScript has a primitive boolean type and boolean values are represented with **true** and **false**. However it is not the only type that can be used as a boolean value. Integer values 0 and 1 and also the strings “0” and “1” can be converted to boolean and then can be used in boolean expressions.



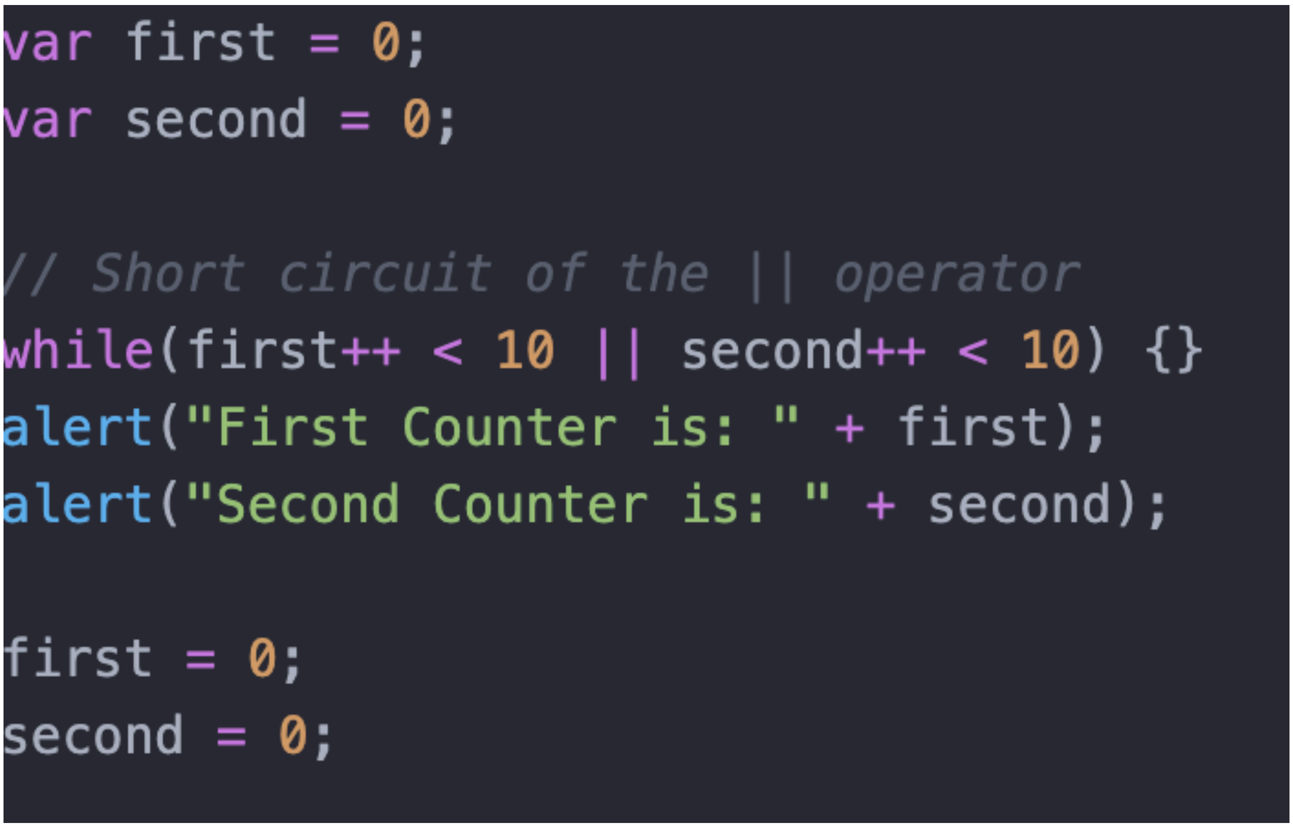
```
// Question 1: JavaScript represents booleans using true and false
alert("Pure boolean: " + true + "\n" + "Pure integer: " + 1);
// But 0 and 1 can also be used as boolean values
alert("Comparison between boolean value true and integer value 1 (true == 1): " + (true == 1));
alert("AND operasion between boolean value true and integer value 0 (true && 0): " + (true && 0));
// String values "0" and "1" an also be used as boolean values
alert("Comparison between boolean value true and string value \"1\" (true == 1): " + (true == "1"));
alert("AND operasion between boolean value true and string value \"0\" (true && 0): " + (true && "0"));
```

Figure 12: Representation of boolean values in JavaScript Language

- All of the expressions in *Figure 12* are valid boolean expressions and which indicates that even though JavaScript uses true and false values for the boolean type, it also accepts 0, 1, “0”, and “1” values and can convert those types to boolean values.

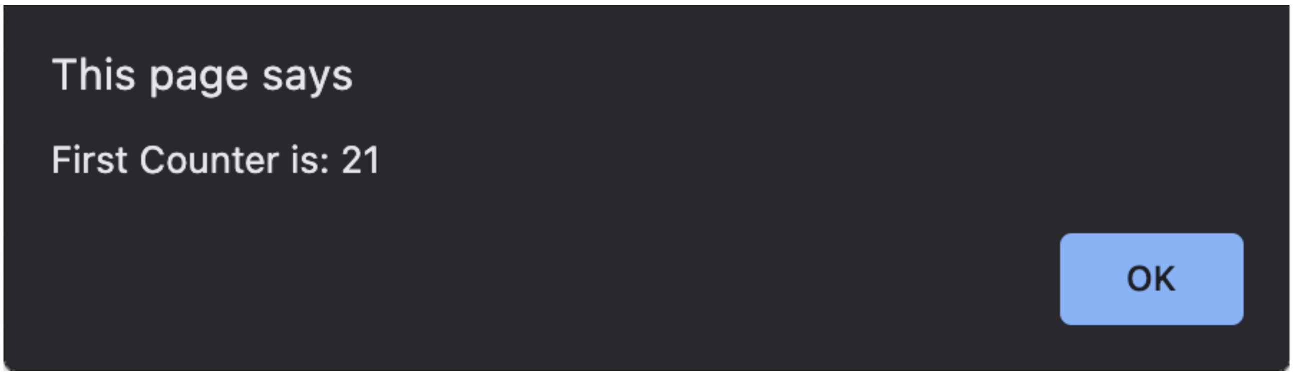
Design Issue 2: What operators are short-circuited?

- The AND (&&) and OR (||) operators in JavaScript language are short circuited.



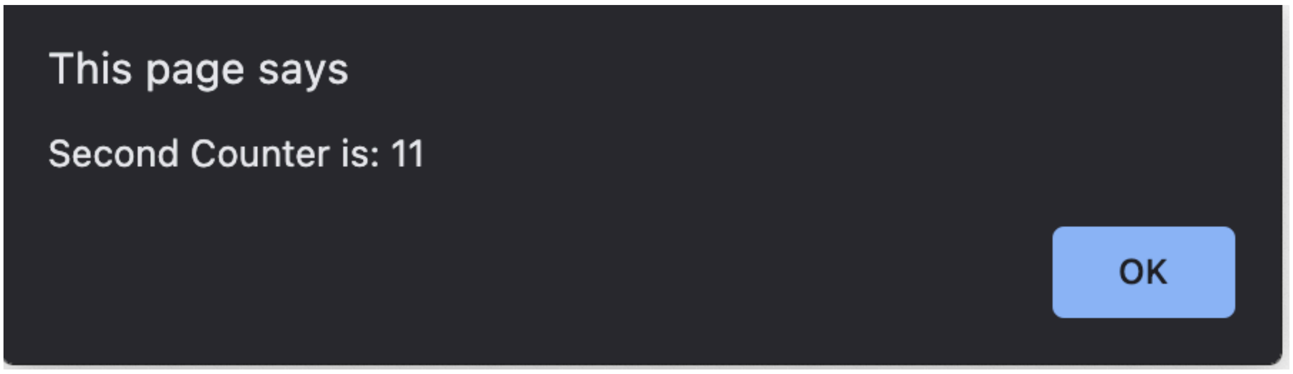
```
var first = 0;
var second = 0;

// Short circuit of the || operator
while(first++ < 10 || second++ < 10) {}
alert("First Counter is: " + first);
alert("Second Counter is: " + second);

first = 0;
second = 0;
```

Figure 13: Short circuit OR operator in JavaScript Language



```
This page says

First Counter is: 21
```

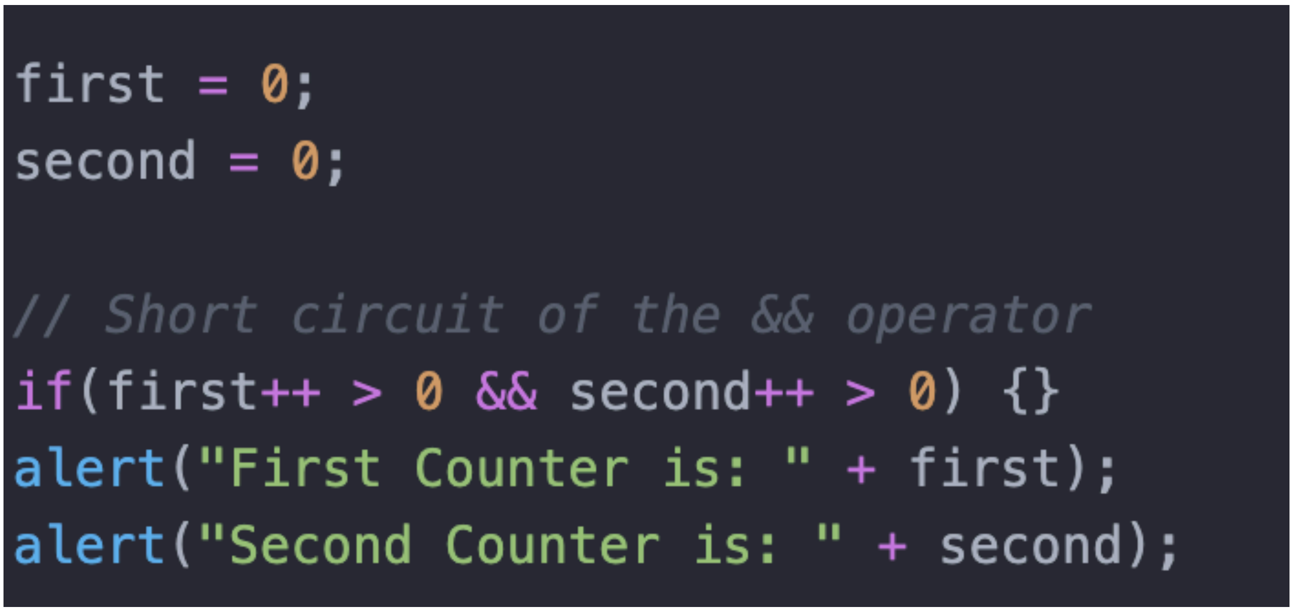
Figure 14: Output 1 of short circuit OR operator in JavaScript Language



```
This page says

Second Counter is: 11
```

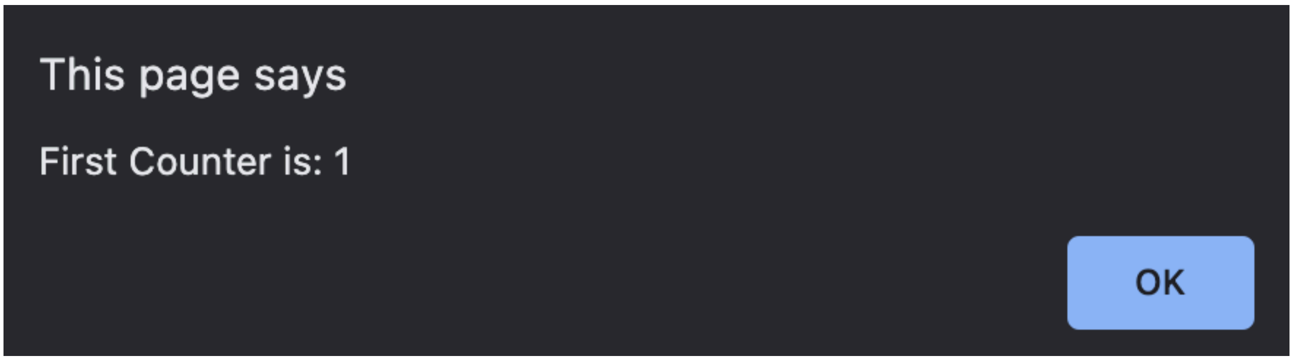
Figure 15: Output 2 of short circuit OR operator in JavaScript Language

- *Figures 13, 14, and 15* shows that the second expression in the OR operation is not calculated in the first 10 iteration since the first expression resulted in true which was the needed value for the short-circuit to happen.



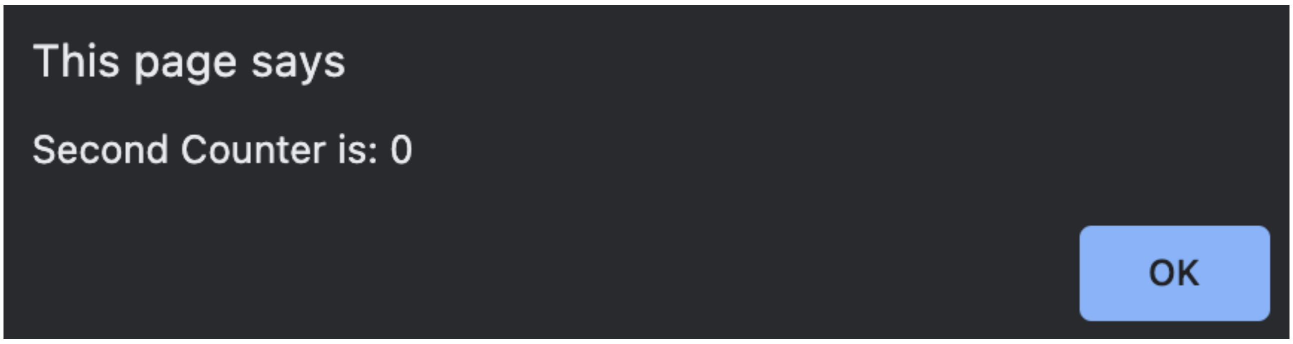
```javascript
first = 0;
second = 0;

// Short circuit of the && operator
if(first++ > 0 && second++ > 0) {}
alert("First Counter is: " + first);
alert("Second Counter is: " + second);
```

Figure 16: Short circuit AND operator in JavaScript Language



```
This page says

First Counter is: 1
```

Figure 17: Output 1 of short circuit AND operator in JavaScript Language



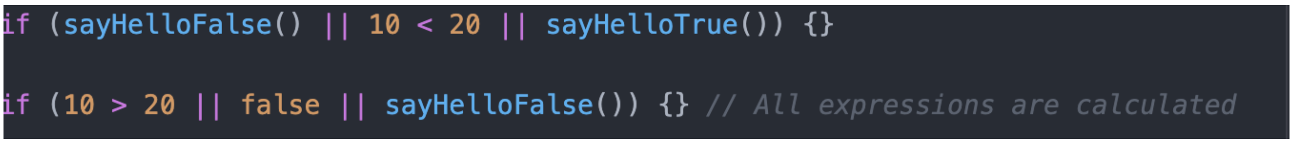
```
This page says

Second Counter is: 0
```

Figure 18: Output 2 of short circuit AND operator in JavaScript Language

- *Figures 16, 17, and 18* shows that the second expression in the AND operation is not calculated in the first since the first expression resulted in false which was the needed value for the short-circuit to happen using the AND operator.

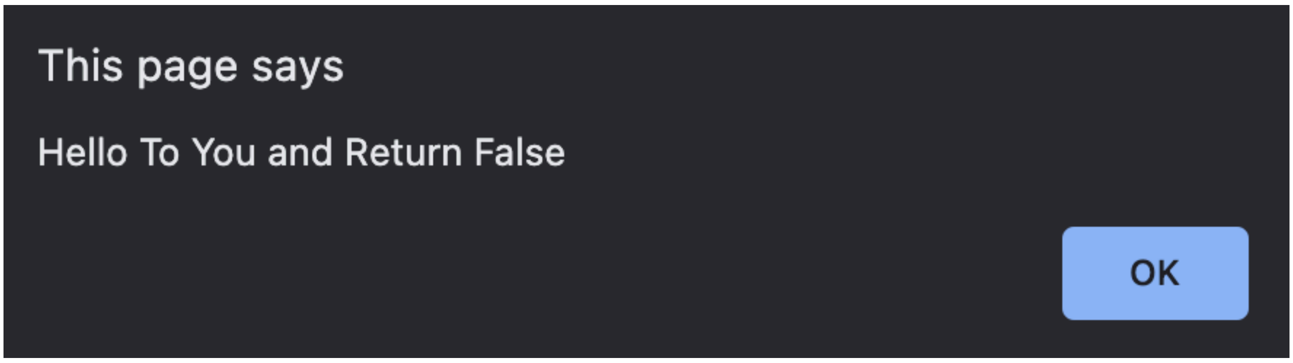
Design Issue 3: How are the results of short-circuited operators computed? (Consider also function calls)

- The results of the short-circuited operators are also calculated starting from the left in JavaScript. It is very similar to Dart language. Whenever the required boolean value that is enough to conclude the calculation is reached, the rest of the expression is not calculated and short-circuit happens. In the OR expressions only one true value is enough to conclude the expression so the program starts from the left and searches for a true value. Similar situation is valid for the false results of the AND expressions. Whenever a false value is found, the AND expression concludes.



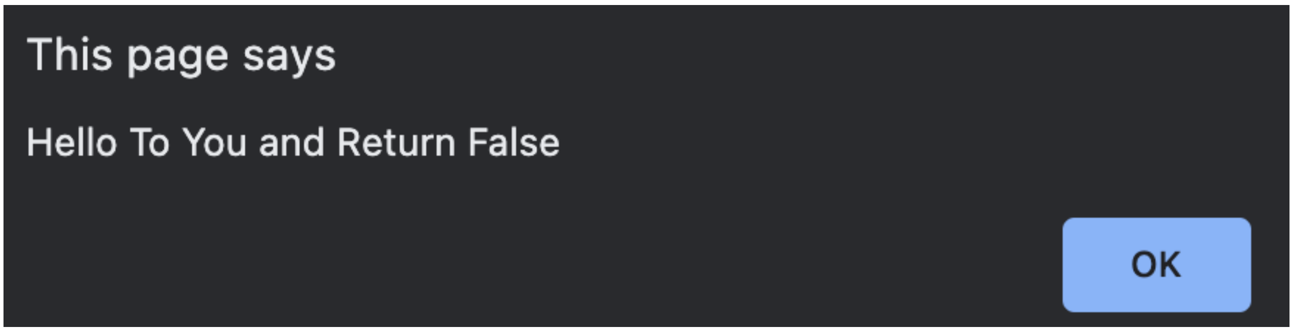
```
if (sayHelloFalse() || 10 < 20 || sayHelloTrue()) {}

if (10 > 20 || false || sayHelloFalse()) {} // All expressions are calculated
```

Figure 19: Computation of short circuit OR in JavaScript Language



```
This page says

Hello To You and Return False
```

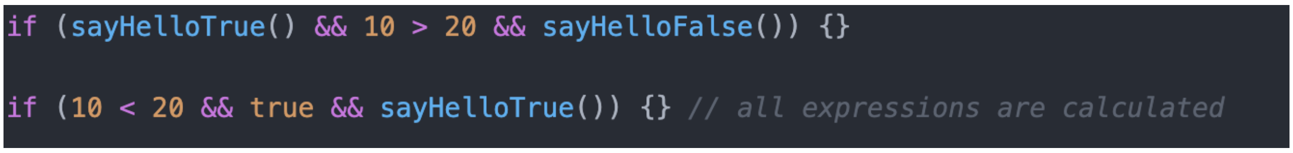
Figure 20: Output 1 of computation of short circuit OR in JavaScript Language



```
This page says

Hello To You and Return False
```

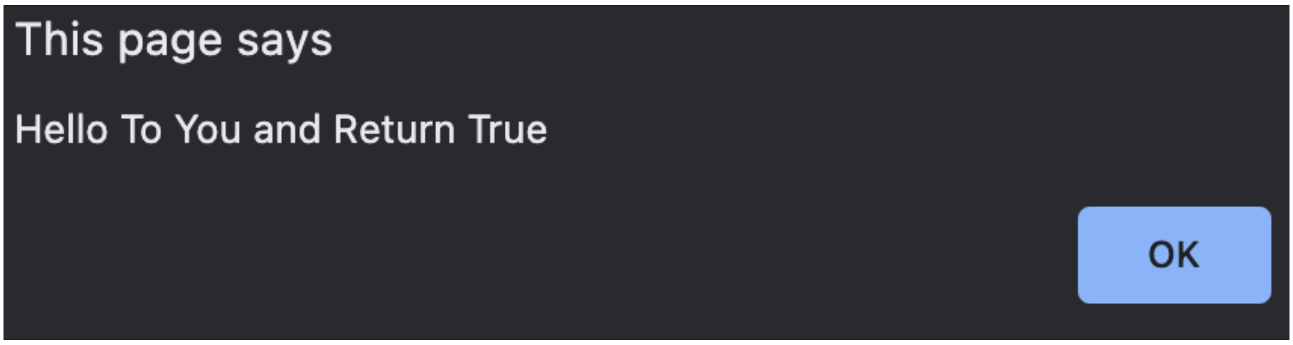
Figure 21: Output 2 of computation of short circuit OR in JavaScript Language

- From *Figures 19, 20, and 21* it is clear that only the first function call is made during the first if statement and after the second expression gets the value true, the last function call is not made. But in the last if statement the function call is made since no true value is found before that expression.



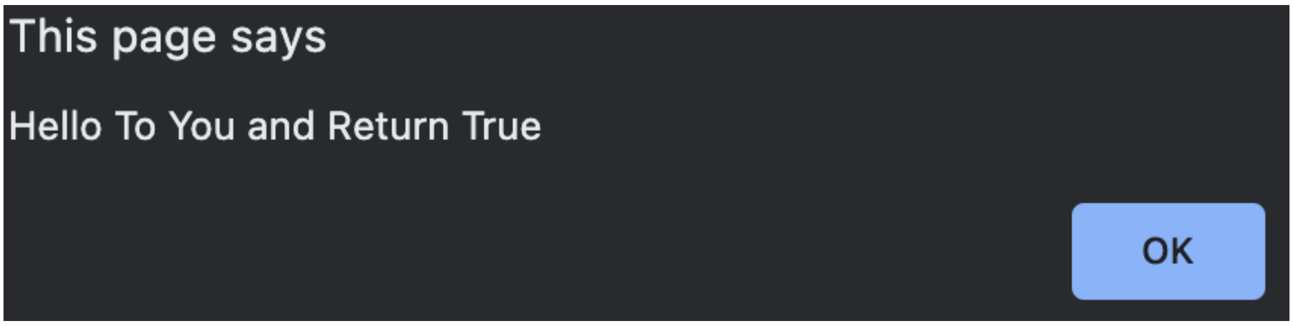
```
if (sayHelloTrue() && 10 > 20 && sayHelloFalse()) {}

if (10 < 20 && true && sayHelloTrue()) {} // all expressions are calculated
```

Figure 22: Computation of short circuit AND in JavaScript Language



```
This page says

Hello To You and Return True
```

Figure 23: Output 1 of computation of short circuit AND in JavaScript Language



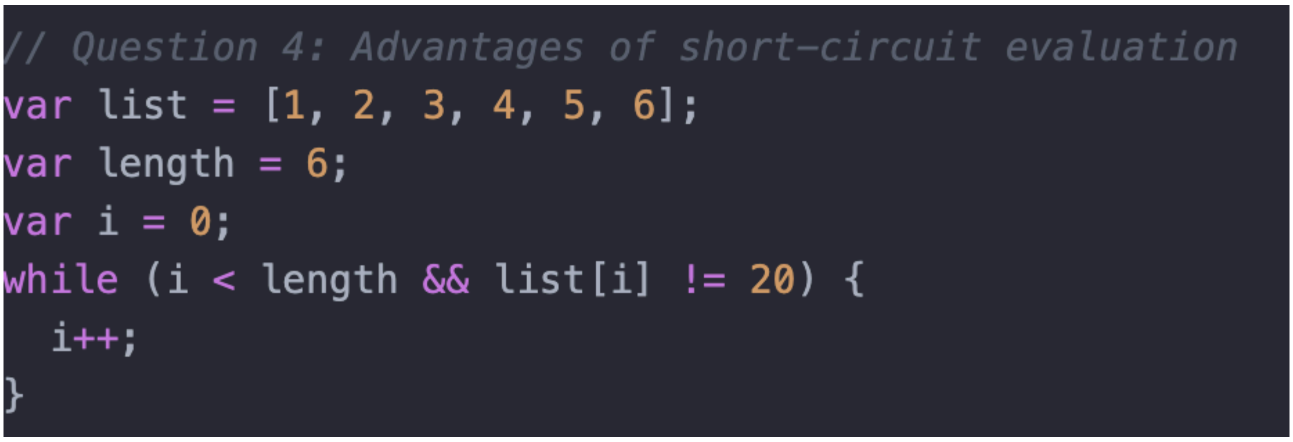
```
This page says

Hello To You and Return True
```

Figure 24: Output 2 of computation of short circuit AND in JavaScript Language

From *Figures 22, 23, and 24* it is clear that only the first function call is made during the first if statement and after the second expression gets the value false, the last function call is not made. But in the last if statement the function call is made since no false value is found before that expression which is needed for the AND operator to return false.

Design Issue 4: What are the advantages about short-circuit evaluation?

- Short-circuit evaluation prevents the computation overhead by discarding the unnecessary calculations for example the ones in the OR expressions after a true value is reached. Or the ones in the AND expressions after the false value is reached. For small calculations it is not that important but for the expressions that has so many relational expressions in it, it is important to avoid from those calculations.
- Avoiding from the side effects is another important aspect of short-circuit evaluations. Normally we may need to include an if statement to check the subscript bound of a list traverse, but with the short circuit feature it is possible to place the expression that may have a side affect to the later stages of the boolean expression and place a safe guard expressions before it to avoid side effects.
- In another aspect, short circuit evaluation can increase the readability of the program since the reader can discard the remaining relational expressions after he/she finds the required boolean value.



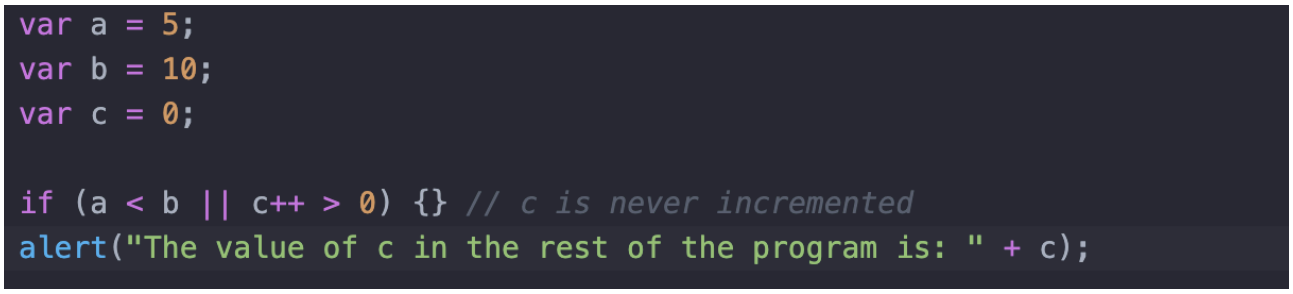
```
// Question 4: Advantages of short-circuit evaluation
var list = [1, 2, 3, 4, 5, 6];
var length = 6;
var i = 0;
while (i < length && list[i] != 20) {
  i++;
}
```

Figure 25: Advantages of short-circuit evaluation in JavaScript Language

- *Figure 25* shows that how the short circuit evaluation can avoid the program from the side effects of the expressions.

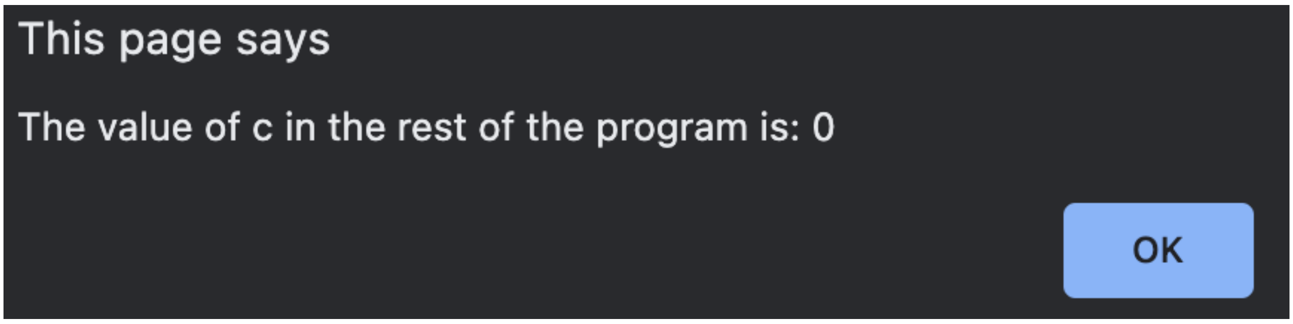
Design Issue 5: What are the potential problems about short-circuit evaluation?

- First of all short- circuit evaluation is detrimental for reliability. Since JavaScript allows and uses short-circuit evaluation in relational expressions, we need to be careful when we are altering variables in relational expressions since those alterations may never happen.



```
var a = 5;
var b = 10;
var c = 0;

if (a < b || c++ > 0) {} // c is never incremented
alert("The value of c in the rest of the program is: " + c);
```

Figure 26: Disadvantages of short-circuit evaluation in JavaScript Language



```
This page says

The value of c in the rest of the program is: 0
```

Figure 27: Output of disadvantages of short-circuit evaluation in JavaScript Language

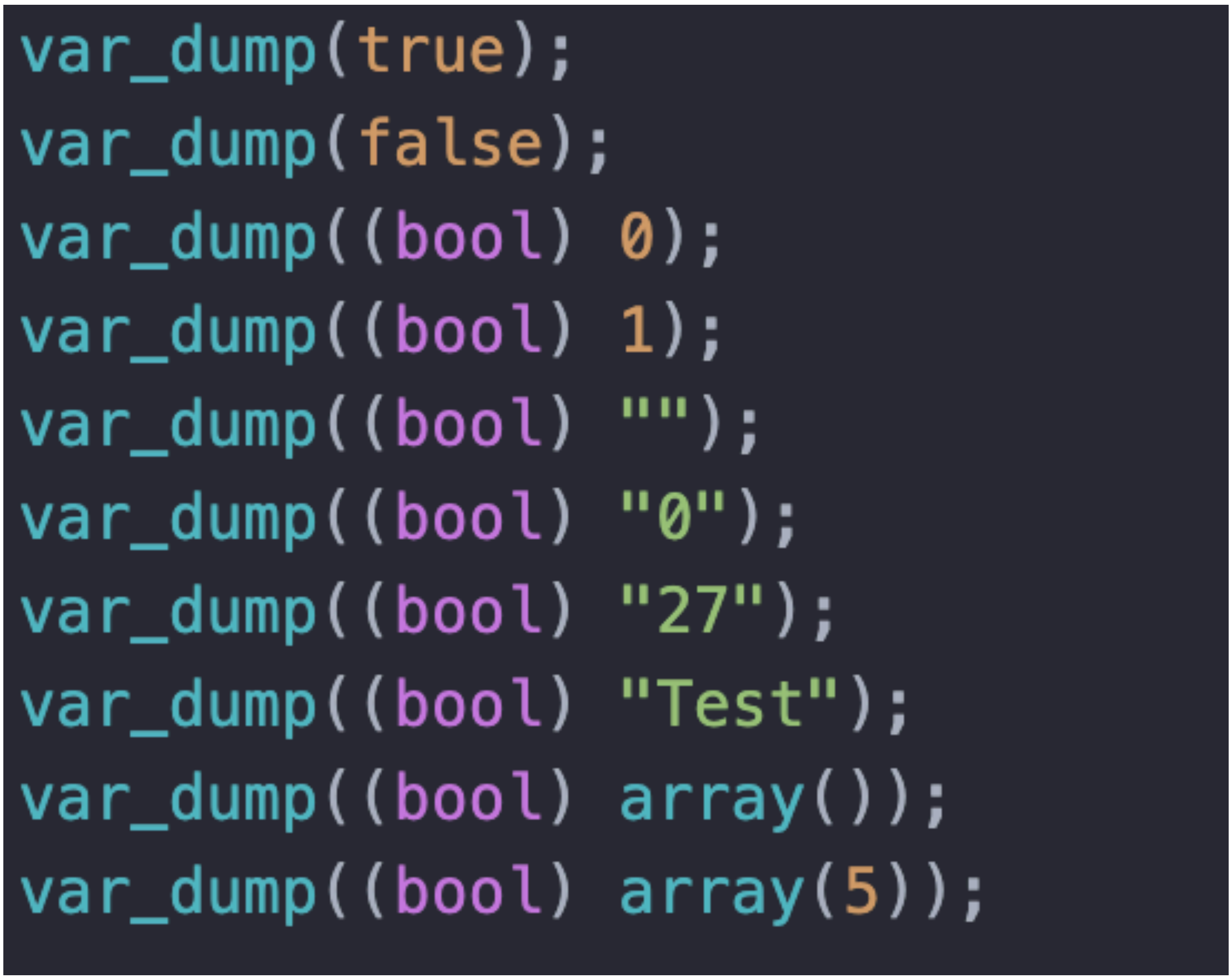
- *Figures 26 and 27* shows that it is not safe to use variable c in the rest of the program since the expression “c++” is never calculated because of short-circuit.

PHP Language

Design Issue 1: How are the boolean values represented?

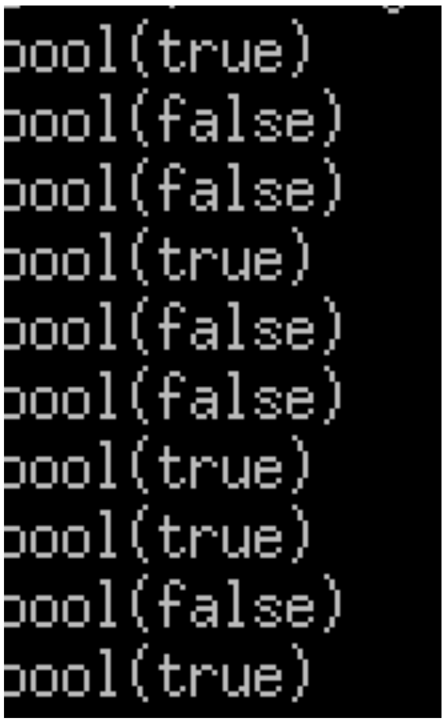
- The implementation of PHP language uses **true** and **false** values to represent boolean. So boolean is a primitive type in the language. However it is possible to convert other data types to booleans explicitly and also implicitly if there is a relational operator. For example in order to represent the boolean value false the followings can be used:
  - Boolean value false
  - Integer value 0
  - Float value 0.0
  - Empty String
  - String value “0”
  - Empty array
  - NULL value

Any other value of any other type is considered as true including the boolean value true.



```
var_dump(true);
var_dump(false);
var_dump((bool) 0);
var_dump((bool) 1);
var_dump((bool) "");
var_dump((bool) "0");
var_dump((bool) "27");
var_dump((bool) "Test");
var_dump((bool) array());
var_dump((bool) array(5));
```

Figure 28: Representation of boolean values in PHP Language



```
bool(true)
bool(false)
bool(false)
bool(true)
bool(false)
bool(false)
bool(true)
bool(true)
bool(false)
bool(true)
```

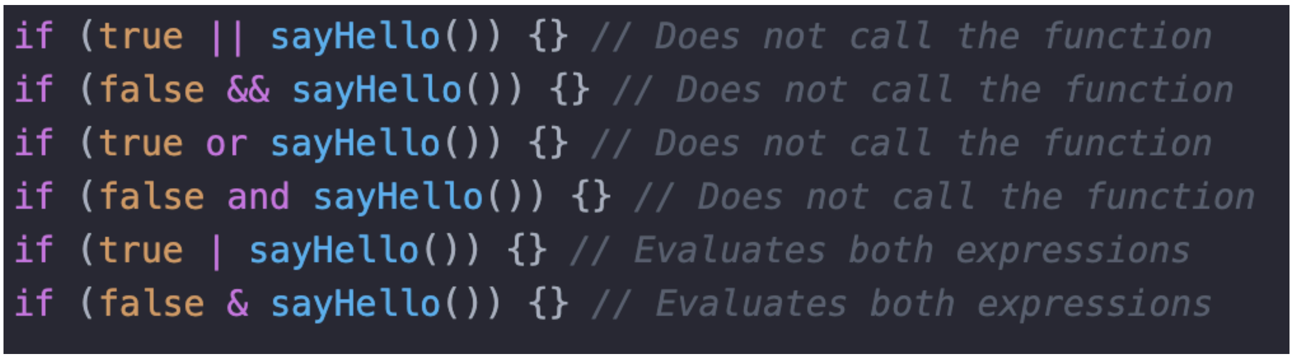
Figure 29: Output of representation of boolean values in PHP Language

- *Figure 29* shows the corresponded output values of the display statements in *Figure 28*. We can see how flexible PHP language is in terms of conversion of other data types into boolean type even though it has a primitive boolean type which can only have the values **true** and **false.**

Design Issue 2: What operators are short-circuited?

- PHP has different kinds of AND and OR operators which are:
  - || -> regular OR operator that is short-circuited
  - && -> regular AND operator that is short-circuited
  - or -> regular OR operator but has a lower precedence than || and that is short-circuited
  - and -> regular AND operator but has a lower precedence than && and that is short-circuited

  - | -> bitwise OR operator that is not short circuited
  - & -> bitwise AND operator that is not short circuited



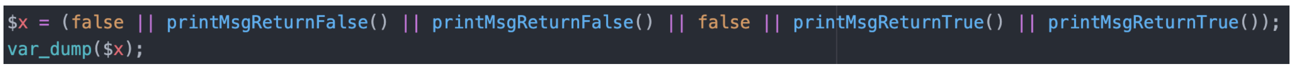
```php
if (true || sayHello()) {} // Does not call the function
if (false && sayHello()) {} // Does not call the function
if (true or sayHello()) {} // Does not call the function
if (false and sayHello()) {} // Does not call the function
if (true | sayHello()) {} // Evaluates both expressions
if (false & sayHello()) {} // Evaluates both expressions
```

Figure 30: Short-circuited operators in PHP Language

- Only the last 2 sayHello() function is called in *Figure 30* since the other ones are not computed because of short circuited AND and OR operators. But the last two bitwise AND and OR operators are not short-circuited and both sides of the expressions are calculated.

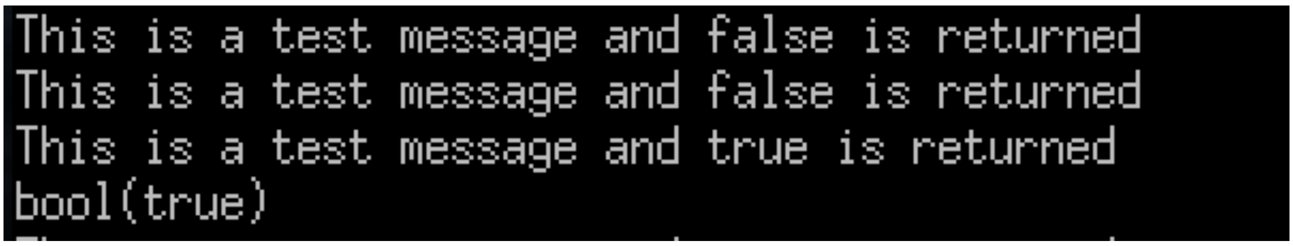
Design Issue 3: How are the results of short-circuited operators computed? (Consider also function calls)

- Similar to the other programming languages discussed, PHP also starts computing the boolean expression from the left and whenever a boolean value that is enough to conclude the calculation of the whole boolean expression is reached, then the program skips the rest of the relational expressions and computation ends. For OR operators, the required value to return true is true and for the AND operator the required value to return false is false. So whenever one of them is found, then there is no need to execute the remaining expressions including the function calls.



```php
$x = (false || printMsgReturnFalse() || printMsgReturnFalse() || false || printMsgReturnTrue() || printMsgReturnTrue());
var_dump($x);
```

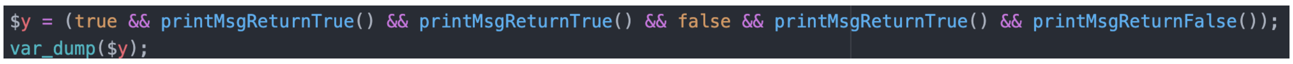
Figure 31: Computation of short circuit OR in PHP Language



```
This is a test message and false is returned
This is a test message and false is returned
This is a test message and true is returned
bool(true)
```

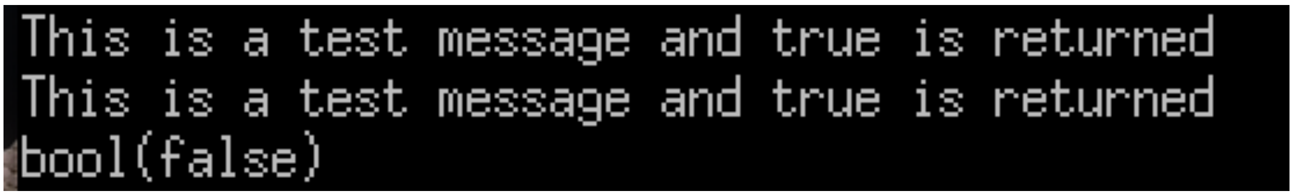
Figure 32: Output of the computation of short circuit OR in PHP Language

- In the OR operator, in order to return a true value, a program starts from left and searches for a true value and discards the rest of the expressions after it finds the true value. In *Figures 31 and 32* only the first 3 function is called since the value true is found with the 3rd function call.



```
$y = (true && printMsgReturnTrue() && printMsgReturnTrue() && false && printMsgReturnTrue() && printMsgReturnFalse());
var_dump($y);
```

Figure 33: Computation of short circuit AND in PHP Language



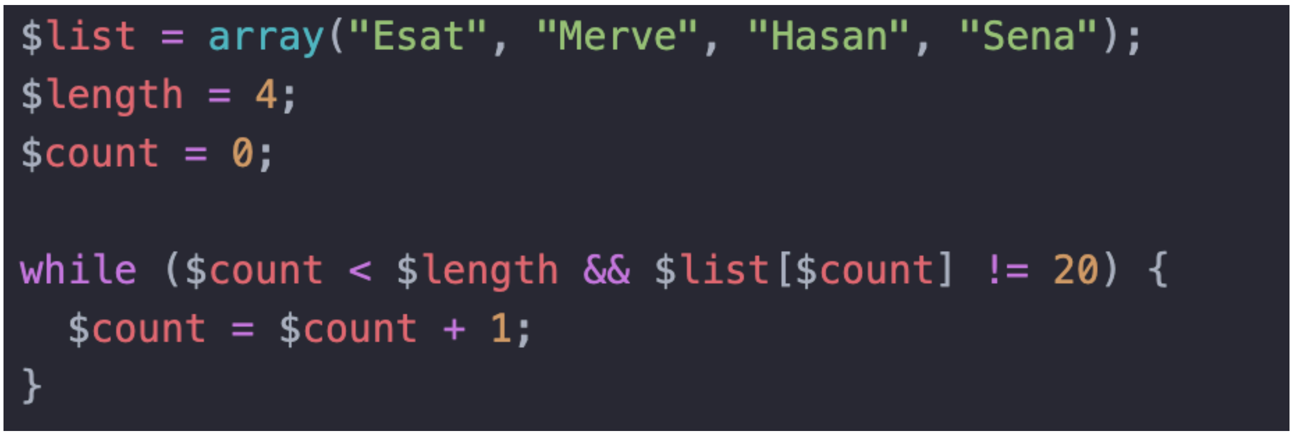
```
This is a test message and true is returned
This is a test message and true is returned
bool(false)
```

Figure 34: Output of the computation of short circuit AND in PHP Language

- In the AND operator, in order to return a false value, a program starts from left and searches for a false value and discards the rest of the expressions after it finds the false value. In *Figures 33 and 34* only the first 2 function is called since the value false is found before the last 2 function call. So the last 2 function calls are never made.

Design Issue 4: What are the advantages about short-circuit evaluation?

- As it is the case for any other language that has the short-circuit evaluation, the computation overhead is avoided by the usage of short circuit evaluation. In the computations that have many relational expressions, it would cost a lot of computational usage for the computer if it tried to do all the computations for all the relational expressions. But by the short-circuit evaluation, the computation is done whenever the required boolean value to conclude the expression is found.
- The expressions which may include any side effects is avoided by short-circuit evaluation. Also it is easy to do the side effect check by short circuited boolean expressions than doing those checks by the separate if statements inside the loops. It increases the readability of the code and it also increases writability since it can be used as an alternative to an if statement.



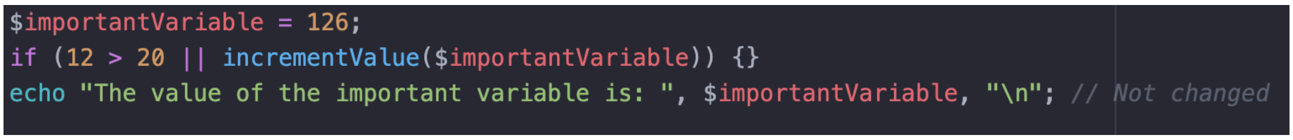
```php
$list = array("Esat", "Merve", "Hasan", "Sena");
$length = 4;
$count = 0;

while ($count < $length && $list[$count] != 20) {
  $count = $count + 1;
}
```

Figure 35: Advantages of short-circuit evaluation in PHP Language

- *Figure 35* shows how the side effect of trying to reach an index that is out of bound is avoided by the short-circuited expression at the last iteration of the loop.

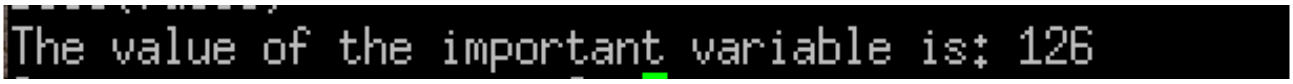
Design Issue 5: What are the potential problems about short-circuit evaluation?

- If the programmer wants each side of the boolean expression to be evaluated, then it is not a good practice to use a short-circuited operator such as ||, &&, and, or. And if he/she uses those operators, then the side affects and possible calculation loses must be considered. Therefore writing boolean expressions in the languages that has short-circuit evaluation such as PHP takes more time to consider writing the expression in a way that the result of the program will not crash because of any non-calculated expression in the conditional statement.



```php
$importantVariable = 126;
if (12 > 20 || incrementValue($importantVariable)) {}
echo "The value of the important variable is: ", $importantVariable, "\n"; // Not changed
```

Figure 36: Disadvantages of short-circuit evaluation in PHP Language



```
The value of the important variable is: 126
```

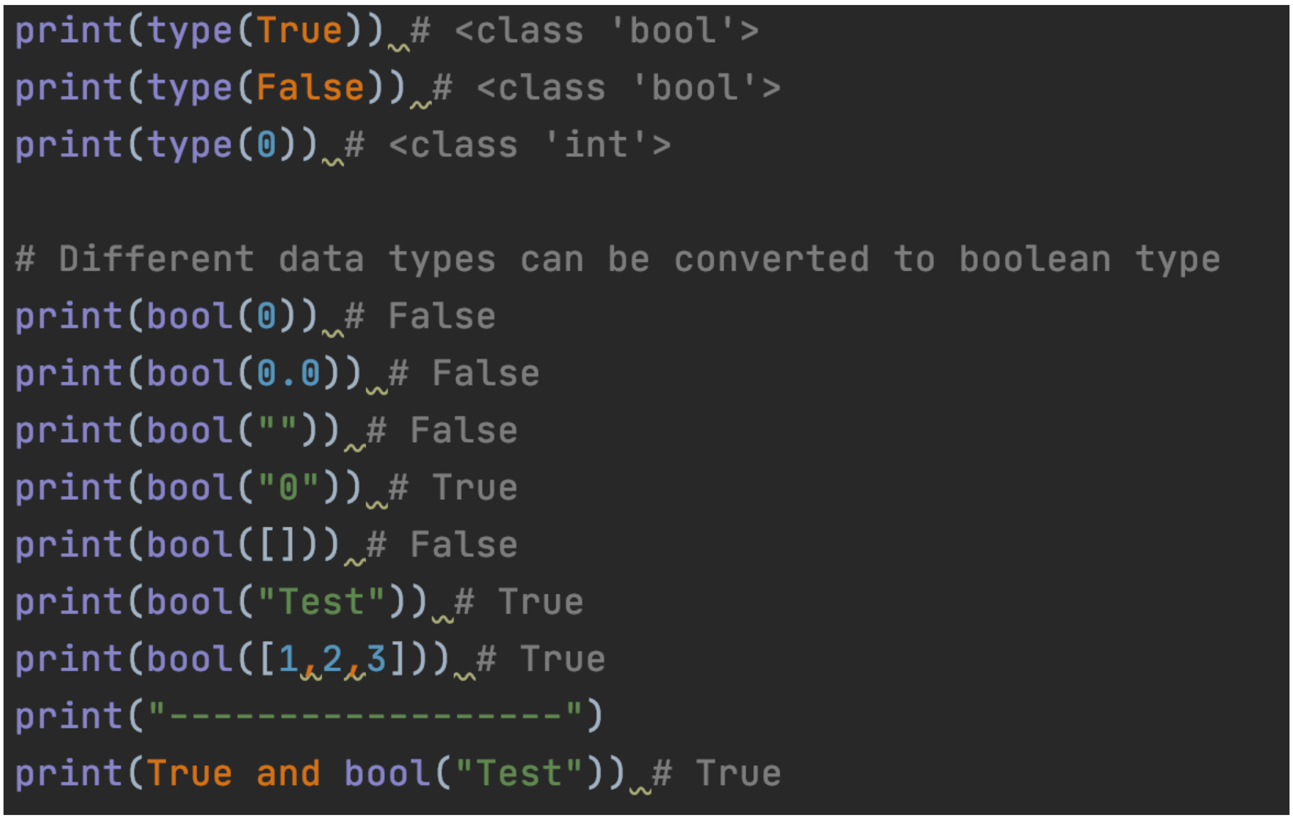
Figure 37: Output of the disadvantages of short-circuit evaluation in PHP Language

- *Figures 36 and 37* show that if we have an important functions or variables that needs to be called or changed for sure, it is not a good practice to use them in the boolean expressions that might result in short-circuit evaluation because those functions might not be called.

Python Language

Design Issue 1: How are the boolean values represented?

- Python has a built in primitive data type for booleans and this type can only have 2 values: **true** and **false.** But other data types can be converted to boolean values explicitly.



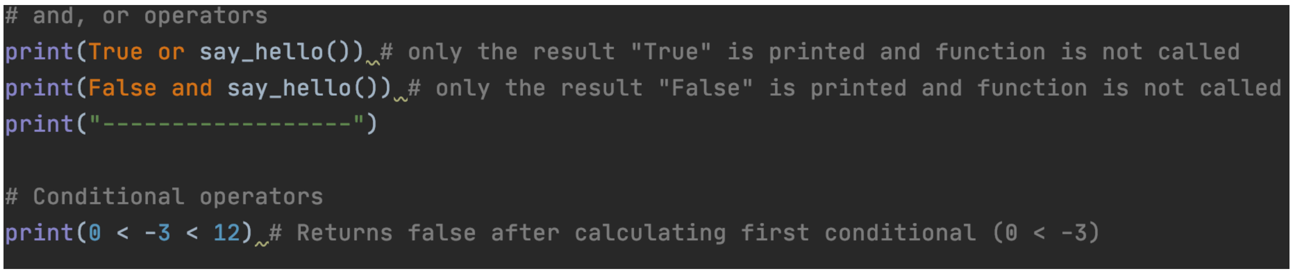
```
print(type(True)) # <class 'bool'>
print(type(False)) # <class 'bool'>
print(type(0)) # <class 'int'>

# Different data types can be converted to boolean type
print(bool(0)) # False
print(bool(0.0)) # False
print(bool("")) # False
print(bool("0")) # True
print(bool([])) # False
print(bool("Test")) # True
print(bool([1,2,3])) # True
print("------------------")
print(True and bool("Test")) # True
```

Figure 38: Representation of boolean values in Python Language

- *Figure 38* shows how the boolean values have their own built in structure which can only take true or false values. It also shows that the other data types can be converted to boolean using bool() function.

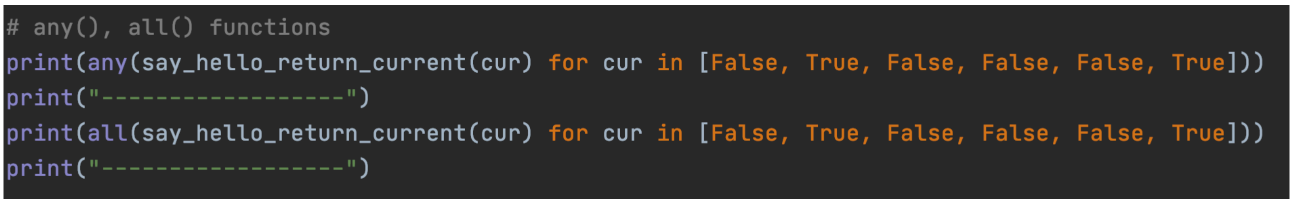
Design Issue 2: What operators are short-circuited?

- Python has different kinds of short circuited operators. First of all **and** and **or** operators are short circuited.
- Python has also built in functions any() and all(). Those functions are shortened versions of consecutive **and** and **or** operators. For example any() returns true if the consecutive boolean values inside its iterable has at least one true value. And it returns false if all of them are false or the iterable is empty. all() returns true if the items in the iterable are all true values or if the iterable is empty. Otherwise it returns false. Those any() and or() functions are also short circuited.
- Python allows the usage of multiple conditional operators in the same expression such as “a < b < c”. Therefore conditional operators are also short circuited in python.



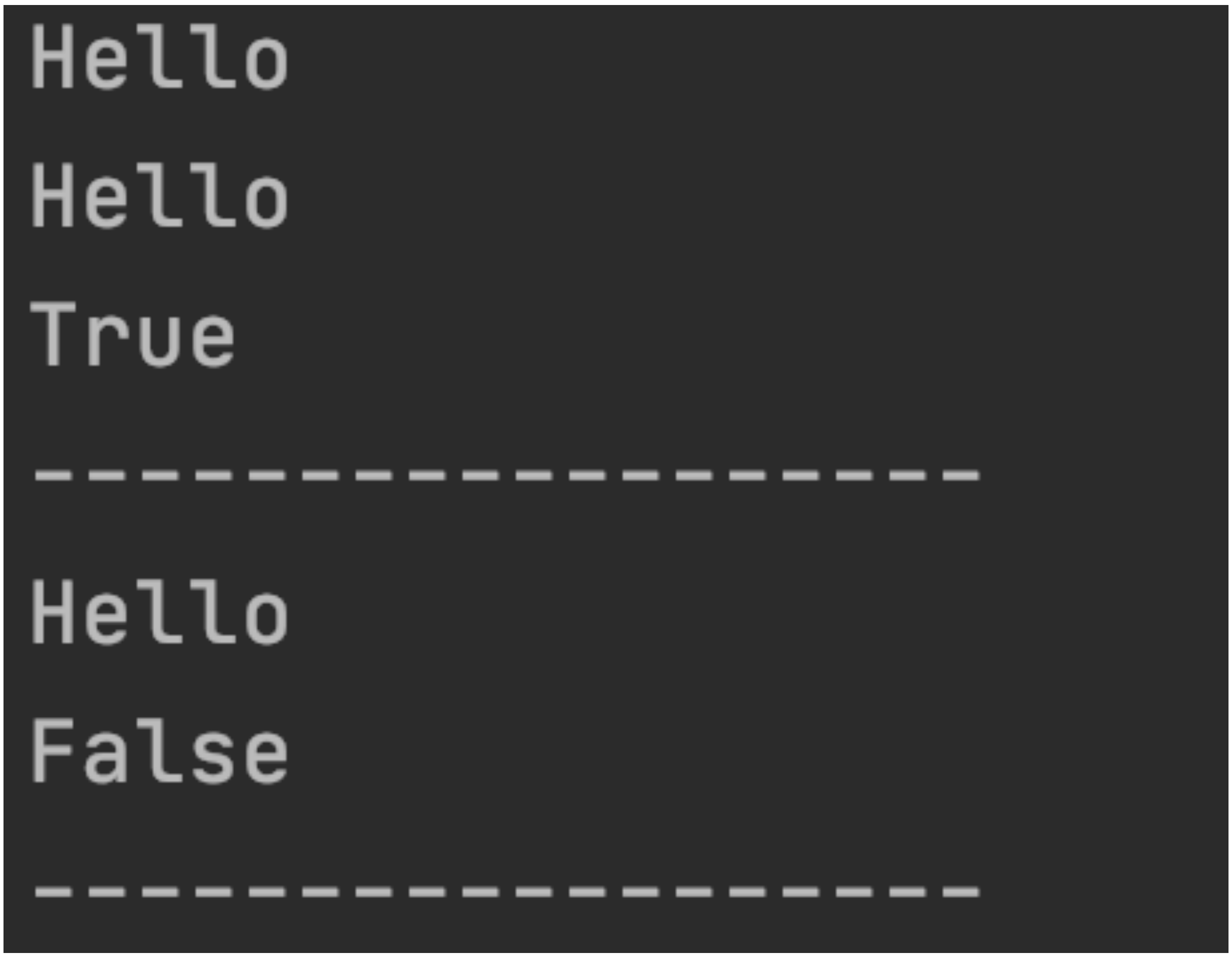
```python
# and, or operators
print(True or say_hello()) # only the result "True" is printed and function is not called
print(False and say_hello()) # only the result "False" is printed and function is not called
print("------------------")

# Conditional operators
print(0 < -3 < 12) # Returns false after calculating first conditional (0 < -3)
```

Figure 39: Short circuit for and, or, conditional operators in Python Language



```python
# any(), all() functions
print(any(say_hello_return_current(cur) for cur in [False, True, False, False, False, True]))
print("------------------")
print(all(say_hello_return_current(cur) for cur in [False, True, False, False, False, True]))
print("------------------")
```

Figure 40: Short circuit for any() and all() in Python Language



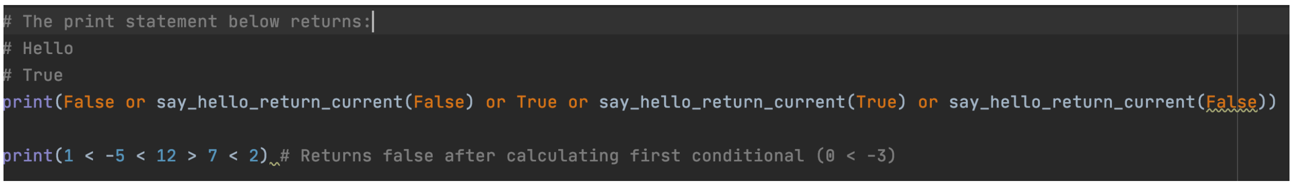
```
Hello
Hello
True
------------------
Hello
False
------------------
```

Figure 41: Output of short circuit for any() and all() in Python Language

- *Figure 39* shows that **and** and **or** operators are short circuited and no function call for say_hello() function is made in the first two print statement.
- *Figure 39* also shows that, in the conditional operators after 0 < -3 = false is determined, the program does not bother to calculate the rest of the statement and directly prints false as the result. So -3 < 12 is never calculated.
- *Figures 40 and 41* show that the function is called only for the first 2 boolean value in the any() function, then the True value is found and program does not call the function for the rest of the boolean array. Similarly for the all() function program finds the false value in the first boolean value and after calling the hello() function only for it, it returns false.

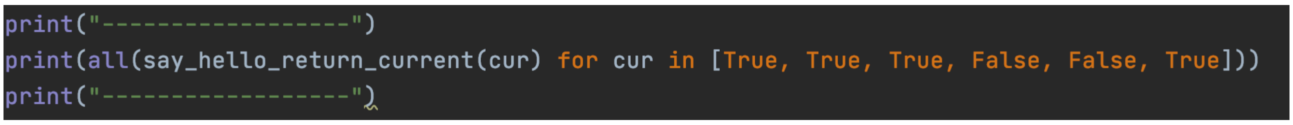
Design Issue 3: How are the results of short-circuited operators computed? (Consider also function calls)

- Short circuit evaluation in python also starts from left to right throughout the expression. Whenever the truth value (whether is true or false) is determined for sure, the rest of the expression is not calculated and program returns the truth value.



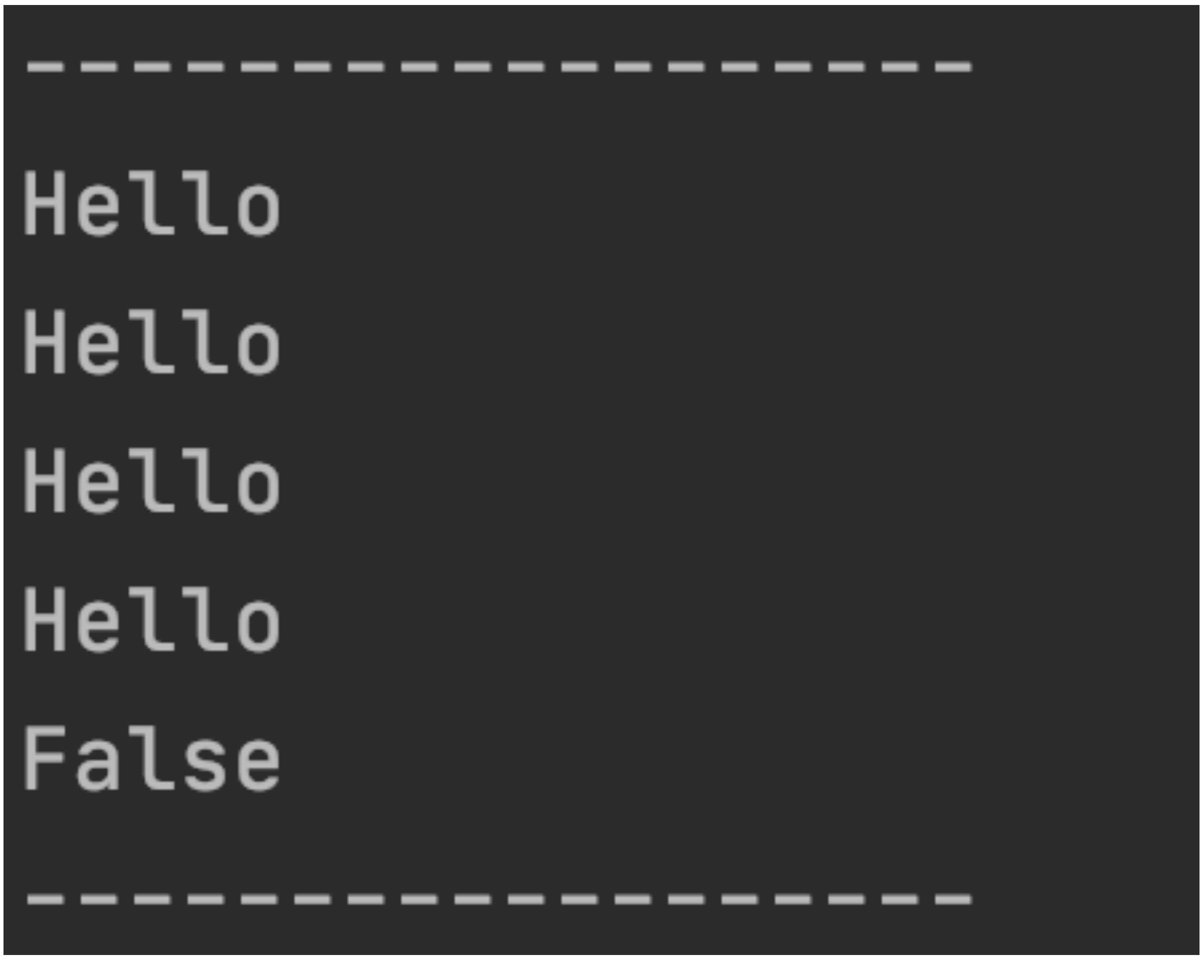
```
# The print statement below returns:
# Hello
# True
print(False or say_hello_return_current(False) or True or say_hello_return_current(True) or say_hello_return_current(False))

print(1 < -5 < 12 > 7 < 2) # Returns false after calculating first conditional (0 < -3)
```

Figure 42: Short circuit evaluation in Python Language



```
print("------------------")
print(all(say_hello_return_current(cur) for cur in [True, True, True, False, False, True]))
print("------------------")
```

Figure 43: Short circuit evaluation for all() and any() in Python Language



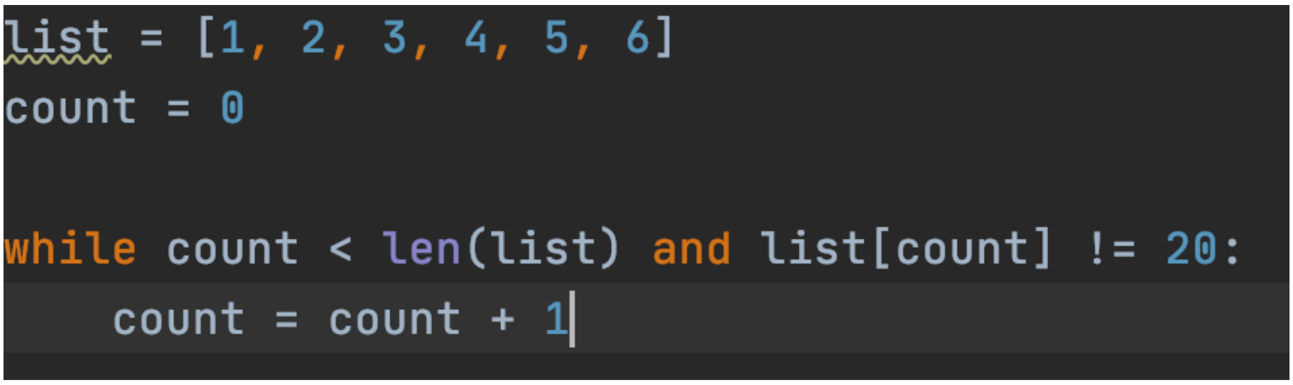
```
------------------
Hello
Hello
Hello
Hello
False
------------------
```

Figure 44: Output of short circuit evaluation for all() and any() in Python Language

- *Figure 42* shows that, for the or operator, program starts from left and checks for a boolean value that is enough to conclude the expression, in this case a true value since it is an or operator. It calls the say_hello_return_current(cur) function only one time and then it finds the True value and the rest two function calls are not made.
- Similarly in the conditional operators example in *Figure 42,* after the program decides the result of the expression which is false by the statement 1 < -5, it directly returns false and short-circuit happens, the rest of the statement is not computed.
- *Figures 43 and 44* shows that, all() function needs to find at least one False value in order to conclude the expression, for the first 3 True values it cannot find the False value and prints Hello. The 4th value in the boolean list is false and program calls the function for it as well which prints another hello and then it decides the result of the statement which is false and returns.

Design Issue 4: What are the advantages about short-circuit evaluation?

- Especially when working with arrays, it is important to check whether we are trying to reach an index that is inside the bounds or not. If we try to reach an out bound index we will get a run-time error in languages such as python. This can be a side effect in a boolean expression since the value of the index can change during the run-time. Those kinds of side affects can be avoided using short-circuited operators.
- Also the calculation overhead that is caused by multiple relational expression evaluation is prevented by only calculating the necessary ones that are needed to decide the truth value of the expression.



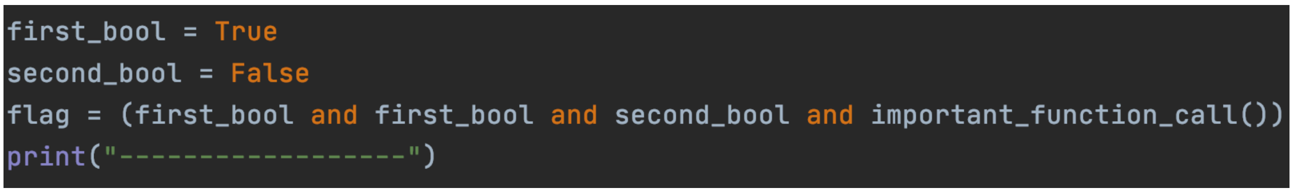
```python
list = [1, 2, 3, 4, 5, 6]
count = 0

while count < len(list) and list[count] != 20:
    count = count + 1
```

Figure 45: Advantages of short-circuit evaluation in Python Language

- In *figure 45*, if the program runs the statement list[count] != 20 in the last iteration, we would get a “list index out of range” exception because out count is 6 in the last iteration. However first part of the **and** operator checks for that condition and asserts a false value which is enough to conclude an **and** operation. Therefore short-circuit happens and list[count] != 20 statement for the last iteration is never executed.

Design Issue 5: What are the potential problems about short-circuit evaluation?

- Usage of short- circuit evaluation is detrimental for the reliability of the programming language. When placing the relational expressions in the boolean expression, the programmer should always think about in which situations a short-circuit may occur and which statements may be passed without executed. Therefore it is dangerous to make changes on the variables or calling the functions that may affect the fate of the rest of the program together with the short-circuited operators.



```python
first_bool = True
second_bool = False
flag = (first_bool and first_bool and second_bool and important_function_call())
print("------------------")
```

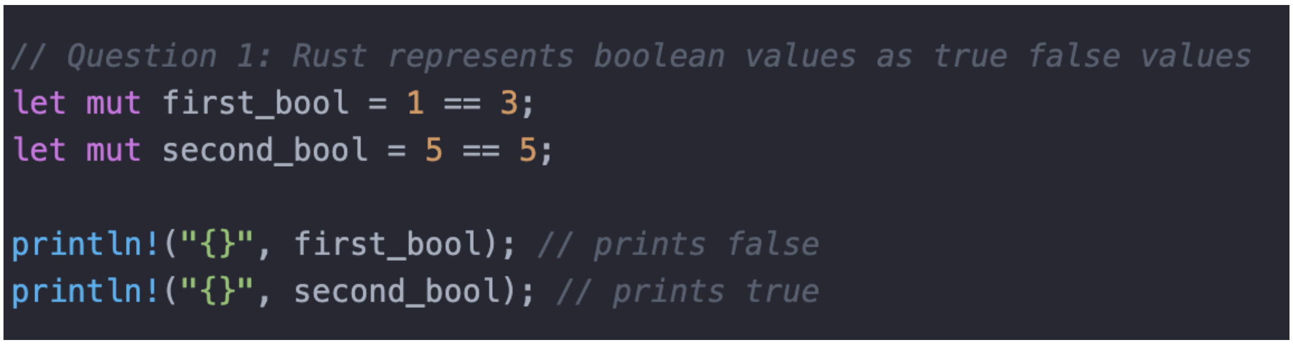
Figure 46: Disadvantages of short-circuit evaluation in Python Language

- *Figure 46* represents a possible disadvantage of a short-circuit evaluation. important_function_call() may be necessary to be called for the programs sake and the programmer may assume it to be called for sure in the boolean expression. However before the function call, short-circuit happens and the function call is never made.

Rust Language

Design Issue 1: How are the boolean values represented?

- Rust has a primitive boolean data type. So the boolean values are represented by **true** and **false** values in Rust. If we try to convert the boolean value true to the integer type we get integer value 1. If we try to convert the boolean value false to the integer type we get integer value 0.



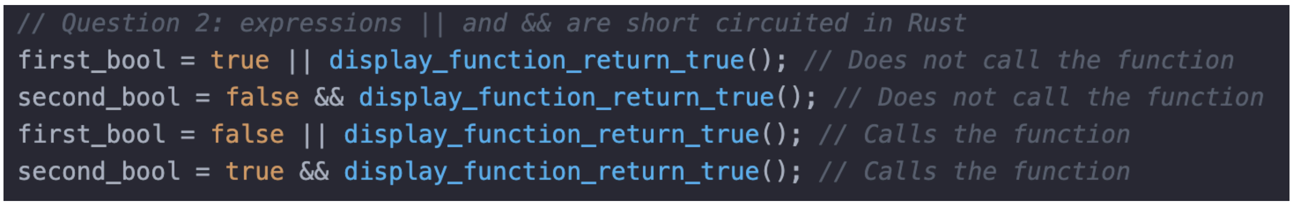
```rust
// Question 1: Rust represents boolean values as true false values
let mut first_bool = 1 == 3;
let mut second_bool = 5 == 5;

println!("{}", first_bool); // prints false
println!("{}", second_bool); // prints true
```

Figure 47: Representation of boolean values in Rust Language

- *Figure 47* shows that Rust uses true and false values to represent boolean type in the language

Design Issue 2: What operators are short-circuited?

- The && (and) and || (or) operators in Rust language are short-circuited. Unlike Python, Rust does not support the usage of multiple conditional operators in one statement such as “12 < 23 > 17 < 100”. Therefore no short-circuit can be talked about such operators since multiple of them in the same statement is not supported in Rust language.



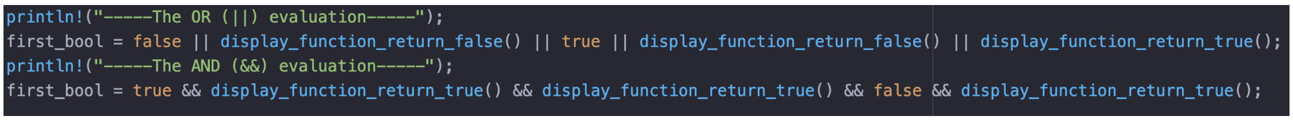
```
// Question 2: expressions || and && are short circuited in Rust
first_bool = true || display_function_return_true(); // Does not call the function
second_bool = false && display_function_return_true(); // Does not call the function
first_bool = false || display_function_return_true(); // Calls the function
second_bool = true && display_function_return_true(); // Calls the function
```

Figure 48: Short-circuit operators in Rust Language

- In the first 2 statements in *Figure 48* short-circuit happens and the function call is not made since the necessary truth value is obtained in the first part of the || and && operators. However in the last lines, there is no short circuit and both sides of the expressions are executed which in turn calls the functions.

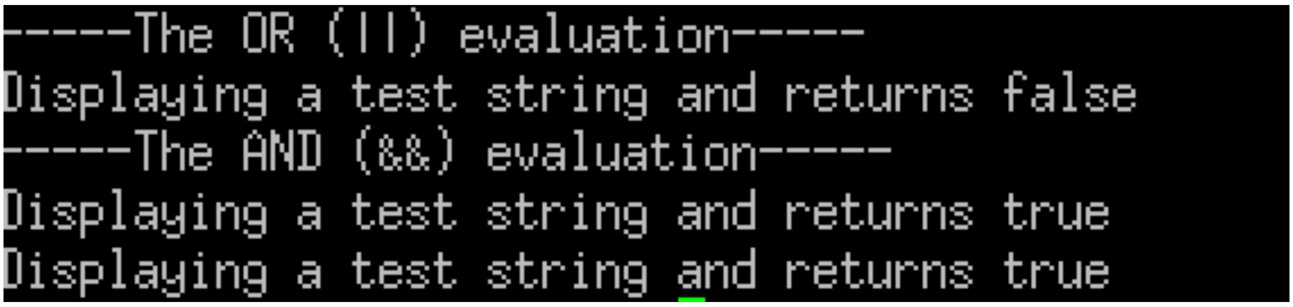
Design Issue 3: How are the results of short-circuited operators computed? (Consider also function calls)

- Is it was the case for any other programming language discussed, Rust also starts evaluating the boolean expressions from left to right. And whenever the boolean value that is sufficient to conclude the whole boolean expression is found, then the rest of the values are not calculated and expression terminates. The sufficient boolean value for || (or) operator to return true is a true value and for && (and) operator to return false is a false value. If there is function call statements in the remainder of those values in the boolean expression, those function call are not made since a short-circuit happens.



```
println!("-----The OR (||) evaluation-----");
first_bool = false || display_function_return_false() || true || display_function_return_false() || display_function_return_true();
println!("-----The AND (&&) evaluation-----");
first_bool = true && display_function_return_true() && display_function_return_true() && false && display_function_return_true();
```

Figure 49: Short-circuit evaluation in Rust Language



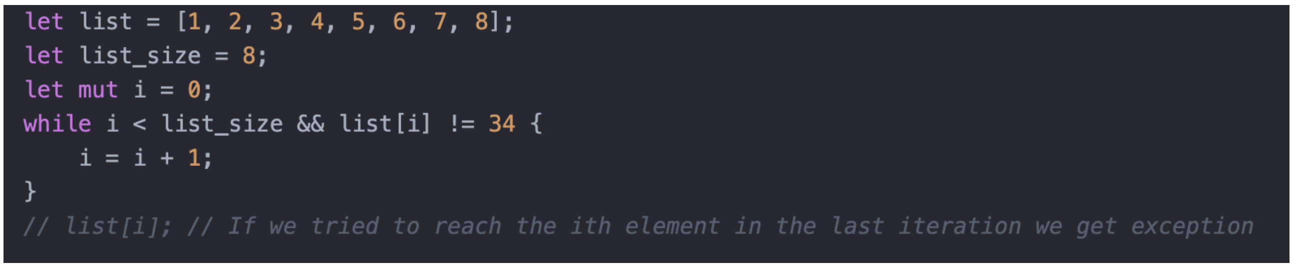
```
-----The OR (||) evaluation-----
Displaying a test string and returns false
-----The AND (&&) evaluation-----
Displaying a test string and returns true
Displaying a test string and returns true
```

Figure 50: Output of short-circuit evaluation in Rust Language

- In the || (or) evaluation of the *Figure 49*, the required boolean value to conclude the expression as true is a true value. So program starts the calculation from left and executes all the expressions before the true value. So display_function_return_false() function that is before the true value is called but the rest is not executed because of short-circuit.
- In the && (and) evaluation of the *Figure 49*, the required boolean value to conclude the expression as false is a false value. So program starts the calculation from left and executes all the expressions before a false value is found. So display_function_return_true() functions that is before the false value is called but the rest is not executed because of short-circuit.

Design Issue 4: What are the advantages about short-circuit evaluation?

- Short circuit evaluation increases writability. There is a similarity between those expressions and nested if statements. By making use of the short circuit evaluation, we san simulate the nested if statements and we can also prevent the possible side effects since the first relational expression in the boolean expression decides whether the rest is calculated or not. So the remaining codes that may have a side effect is avoided by the boolean checking before them.
- Although boolean calculations use considerably small amount of CPU, it may use a lot of computational power when there are hundreds or thousands of statements in a boolean expression. So terminating the whole expression after obtaining the necessary boolean value saves us from a big computational overhead which increases the performance of the program.



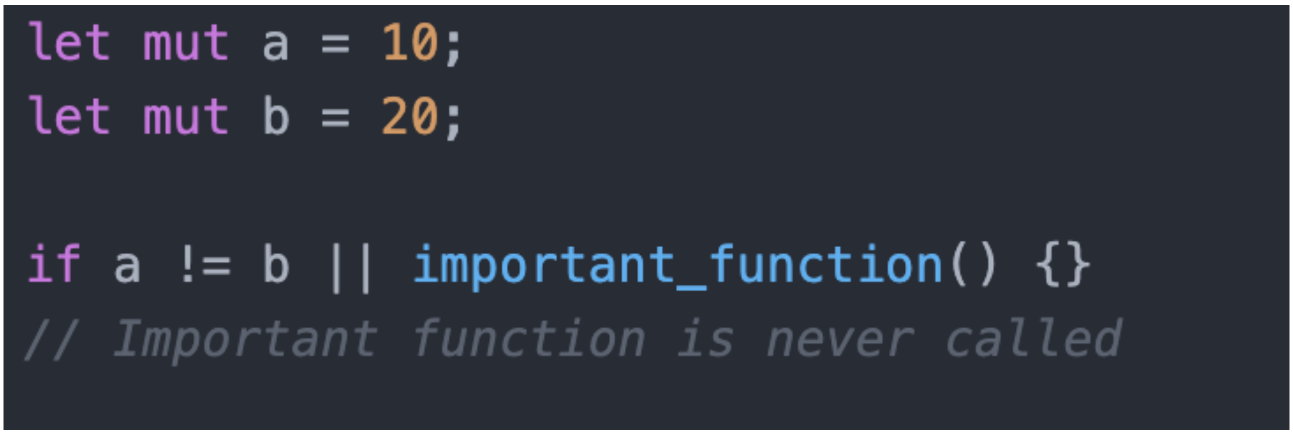
```rust
let list = [1, 2, 3, 4, 5, 6, 7, 8];
let list_size = 8;
let mut i = 0;
while i < list_size && list[i] != 34 {
    i = i + 1;
}
// list[i]; // If we tried to reach the ith element in the last iteration we get exception
```

Figure 51: Advantages of short-circuit evaluation in Rust Language

- Figure 51 is the illustration how short-circuit evaluation can save us from a side effect, which is index out of bound exception. In the last iteration of the loop, list[I] != 34 is not executed because of short-circuit and the program crash is prevented.

Design Issue 5: What are the potential problems about short-circuit evaluation?

- Programmers that use Rust language should always be cautious about the variable changes that are made in boolean expressions and even the function calls that are made in the boolean expressions. Because those statements might not be made because of short circuit. If the computations that are made in possible short-circuited code parts are important for the rest of the code execution, side effects may happen and code may crash.



```
let mut a = 10;
let mut b = 20;

if a != b || important_function() {}
// Important function is never called
```

Figure 52: Disadvantages of short-circuit evaluation in Rust Language

- The important_function() which may affect the sake of the rest of the program is not called in *Figure 52* because of a short-circuit. This kind of side effects should be considered when designing boolean expressions in the languages that uses short-circuit evaluation.

Discussion 1:

I think the best language in terms of short circuit evaluation among the 5 languages discussed is Python language. All of the languages discussed namely, Dart, JavaScript, PHP, Python, and Rust implements short-circuit evaluation in their logical operators. Therefore we need to find the most useful one to make a choice among them. Even though it reduces the reliability of the program I think it is important for a language to have a flexible type conversions because it is much easy to write such programs.
The most comfortable 2 languages here are PHP and Python. Even though those languages have primitive built in boolean data type which can only take true and false values, in both of the languages it is so easy to convert any data type to a boolean type. This flexibility is important to write more generic programs considerably faster than other languages.

Moreover, one of the advantages of the short-circuit evaluation is that it reduces the computational overhead by discarding the expressions that are not necessary for the truth value of the whole expression. This approach is perfectly used in Python because Python has some other short-circuited operators as well. For example Python allows multiple conditional operators to be used consecutively in an expression. And those operators are also short-circuited. So not only the redundant calculations are saved in boolean expressions such as **and** and **or,** but also redundant operations in conditional operators such as <, <=, >, >= are avoided.

Therefore Python makes use of the short-circuit evaluation in the most flexible and broad range which makes it the best language in terms of short-circuit evaluation for me.

Discussion 2:

First of all, before starting the implementation of the homework, I studied the relevant chapters about the homework such as “Relational Expressions”, “Boolean Expressions”, and “Short-circuit Evaluation” in the course material in order to have a better understanding about what I would be doing. Then I began to the homework.

At first I decided to proceed question by question. Namely, I tried to answer a question for every language and when I was done with that question I was going to proceed with the next question. However, after I answered the first question for the languages I realized that it was not a good methodology because I started to get confused about different syntaxes and semantics for different languages. So I decided to finish all questions for a language and the proceed with the next language. It became helpful for me because after finishing all questions for a language I had an understanding about the general concepts such as how are the results of short-circuit evaluation done and then I used those understanding in the other language.

I have added all the resources that I used throughout the homework into the References section of the report. The most helpful resource for me was the documentations of the programming languages. It was the case because nobody can know a specific detail about a language better than the creators of the language. The only drawback was that sometimes is was hard to find the necessary information in the documentation. For example Python’s documentation is too broad and it includes informations for different versions of Python and also Python has too much functions and operators compared to the other programming languages. Therefore it was hard for me to read the documentation for python and in those circumstances, I used tutorials and forums such as geeksforgeeks and stackoverflow. I have also added the references of those forums that I used into the References section. I did not use any YouTube tutorial for my researches because I believe it takes so much time and documentations are far more useful especially when we have a limited time for the assignment. The most useful and well prepared documentation for mw was the documentation of PHP. It has brief and informative explanations together with explanatory examples for each case and also it has user contributed notes that is helpful for the community.

In terms of code implementation, I used online compilers for the languages such as Dart and Rust. It was a bit risky process because sometimes Dijkstra server behaves different than the online compilers according to the versions of the languages or compilers. Therefore after I finished my code in an online compiler, I double checked it in the Dijkstra server and all of the codes are running in the server as well. For Python, I have PyCharm IDE and Python installed in my computer so it was easy for me to test my python codes in my machine. For JavaScript I have written my codes using my text editor and then I tested the outputs in Google Chrome. However I could not find any online compiler or any source that I could test my PHP codes therefore I had to use Dijkstra server for it from the beginning. It was a bit hard for me to change the code in my local machine then remove the old version of the code in the server and re-upload the new code to the server again and again each time I had a problem.

Other than that I had no problem about the implementation of the codes since I was also familiar with those languages from the first homework. Finally, after I finished my report, I tested all my codes in the Dijkstra server again in order to be sure that every of them are working and then I zipped all my files and ended my homework.

References:

"Booleans - Manual." *Php*, https://www.php.net/manual/en/language.types.boolean.php.

"Dart Documentation." *Dart*, https://dart.dev/guides.

Dart Programming - Logical Operators, https://www.tutorialspoint.com/dart_programming/dart_programming_logical_operators.htm.

DinahDinah 50k2929 gold badges129129 silver badges149149 bronze badges, et al. "Does Python Support Short-Circuiting?" *Stack Overflow*, 1 May 1958, https://stackoverflow.com/questions/2580136/does-python-support-short-circuiting.

JavaScript Booleans, https://www.w3schools.com/js/js_booleans.asp.

"Javascript Short Circuiting Operators." *GeeksforGeeks*, 11 Sept. 2020, https://www.geeksforgeeks.org/javascript-short-circuiting/.

"Logical Operators - Manual." *Php*, https://www.php.net/manual/en/language.operators.logical.php.

"Primitive Type Bool1.0.0[−]." *Rust*, https://doc.rust-lang.org/std/primitive.bool.html.

Real Python. "Python Booleans: Optimize Your Code with Truth Values." *Real Python*, Real Python, 3 Aug. 2021, https://realpython.com/python-boolean/.

"The Rust Reference." *Boolean Type - The Rust Reference*, https://doc.rust-lang.org/reference/types/boolean.html.

Sebesta, Robert W. *Concepts of Programming Languages*. Pearson, 2016.

"Short Circuiting Techniques in Python." *GeeksforGeeks*, 12 June 2019, https://www.geeksforgeeks.org/short-circuiting-techniques-python/.

"Short-Circuit Evaluation in Programming." *GeeksforGeeks*, 22 June 2021, https://www.geeksforgeeks.org/short-circuit-evaluation-in-programming/.