CS315 HW2 Report

Efe Beydoğan

21901548

Section 1

Dart

1) How are the boolean values represented?

Dart has built-in support for the boolean data type. The boolean data type only supports the values true and false. The type name for boolean values is “bool”. Additionally, only the literal “true” is interpreted as true in the Dart language, any other value is considered false (only in unchecked mode, https://www.tutorialspoint.com/dart_programming/dart_programming_boolean.htm). In checked mode, an error is thrown for anything other than the literal “false”.

```
1.  bool trueBool = true;
2.  bool falseBool = false;
3.
4.  print( trueBool); // prints "true"
5.  print( falseBool); // prints "false"
6.
7.  // if ( "asd") print( "true?"); -> error: "A value of type 'String' can't be
    assigned to a variable of type 'bool'."
```

2) What operators are short-circuited?

Operators that are short circuited in Dart are the logical AND (&&), logical OR (||) and the if-null (??) operators. The ?? operator works as follows: “var a = expr1 ?? expr2”, in this expression if expr1 is non-null, its value is returned, otherwise expr2 is evaluated and its value is returned.

```
1.  // usage of logical AND
2.  var c = 31;
3.  var d = false;
4.  print( d && c++ > 20); // prints "false", doesn't check or evaluate second
    expression since first one is false
5.
6.  // usage of logical OR
7.  print( c > 17 || d); // prints "true", first checks the first expression and skips
    the second one since first one is "true"
8.
9.  // usage of ?? operator
10. print( c ?? d); // prints "31" since first expression is not equal to null, skips
    second expression
```

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

The logical AND (&&) operator only returns true when all the expressions are true, and skips the evaluation of subsequent expressions once it discovers a “false” statement among the expressions it was provided. Similarly, the logical OR (||) operator needs only one “true” expression among the expressions it is provided, and skips subsequent ones once it finds a “true” expression. The ?? operator only checks expressions until it finds one that is non-null, and returns it. All of these operators in Dart can be cascaded, such as “a && b && c ...” or “a || b || c ...” or “a ?? b ?? c ...”. However, as explained before, expressions are evaluated from left to right and subsequent expressions aren’t checked once a sufficient one has been evaluated, thus these operators are all short-circuited.

```
1.  bool boolFunc() {
2.    print( "inside function");
3.    return true;
4.  }
5.  void main() {
6.    // logical AND
7.    var a = 5;
8.    var b = 10;
9.
10.   if ( false && b++ > a) {
11.     print( "inside if"); // doesn't enter here
12.   }
13.   else {
14.     print( "inside else b is: $b"); // prints "inside else b is: 10", so the b++
    operation isn't carried out due to short circuit
15.   }
16.
17.   if ( true && b++ > a) print( "b is $b"); // prints "b is 11", now b has been
    incremented because the second expression was evaluated
18.
19.   if ( a > 10 && boolFunc()) { // first expression is false, so boolFunc() is never
    called
20.     print( "inside?"); // this isn't printed
21.   }
22.
23.   if ( a < 10 && boolFunc()) { // first expression is true, so boolFunc() is called
    and prints "inside function"
24.     print( "inside."); // this is printed (boolFunc() returns true)
25.   }
26.
27.   // logical OR
28.   a = 5;
29.   b = 10;
30.   if ( true || ++b > a) {
31.     print( "inside if b is: $b"); // prints "inside if b is: 10", as can be seen
    second expression isn't evaluated
32.   }
33.
34.   if ( false || ++b > a) {
35.     print( "inside second if b is: $b"); // prints "inside second if b is: 11", now
    second expression is evaluated
36.   }
37.
38.   if ( a < 10 || boolFunc()) { // first expression is true, so boolFunc() is never
    called
39.     print( "inside."); // this is printed
40.   }
41.
```

```
  if ( a > 10 || boolFunc()) { // first expression is false, so boolFunc() is called and prints "inside function"
    print( "inside."); // this is printed (boolFunc() returns true)
  }

  // if-null operator (??)
  a = 5;
  b = 10;
  var y = a ?? ++b;
  print(y); // prints "5", a isn't null
  print(b); // prints "10", b wasn't incremented because of short circuiting
  var x = null ?? b++;
  print( x); // prints 10, because it is post increment
  print( b); // prints 11, because the second expression was evaluated due to the first one being null

  var z = 3 ?? boolFunc(); // doesn't print "inside function" because boolFunc() isn't called
  print( z); // prints "3"

  z = null ?? boolFunc(); // prints "inside function", second expression is evaluated since first one is null
  print( z); // prints "true" (boolFunc() returns true)

}
```

4) What are the advantages about short-circuit evaluation?

There are two main advantages of short-circuit evaluation. Firstly, short-circuit evaluation provides a way to avoid runtime errors in situations where an argument must be checked first before the second argument is executed, or else an error will be generated. Secondly, costly tasks can be avoided through the use of short-circuit evaluation.

```
var arr1 = [1,2,3,4,5,6,7];
var index = 5;

if ( arr1.length > index && arr1[index] == 6) { // short-circuit evaluation provides a way to first check if array length is bigger than a desired value
  print( "5th index exists!");                  // the second expression is only evaluated if the first one is true
}

index = 10;

if ( arr1.length > index && arr1[index] == 99) { // first argument is false, so second one is skipped
  print( "won't enter here");
}
else {
  print( "error avoided"); // due to short-circuit evaluation, out of bounds error is avoided
}

if ( arr1.length > 10 && boolFunc()) {
  // do something...
} // since arr1.length is not bigger than 10, boolFunc() is never called
  // if boolFunc() was a complex function that required a lot of computations, doing the costly tasks it requires could've been avoided with short-circuit evaluation
```

5) What are the potential problems about short-circuit evaluation?

Although short-circuit evaluation helps with avoiding errors and costly procedures, it may become unreliable at times. In the below code segment, the second expression of the if statement is never evaluated, so if the person writing the code assumes that var1 will be incremented at every pass, this could lead to a logical error in the code.

```
1.  var var1 = 100;
2.  var var2 = 10;
3.  for ( var i = 0; i < 5; i++) {
4.    if ( var2 > 5 || var1++ > 100) { // second expression is never evaluated, so var1
    is never incremented
5.      print( "var1 is $var1"); // prints "var1 is 100" 5 times
6.    }
7.  }
```

If a code like this is run:

```
1.  var a = 10;
2.  if ( a < 5 && func1()) {
3.    func2();
4.  }
```

In this case, since the first expression will evaluate to false, func1() will never be called. This could cause reliability problems like the one in the first example, if func1() is supposed to be called regardless of func2().

Resources used for Dart

https://www.tutorialspoint.com/dart_programming/dart_programming_boolean.htm

https://www.tutorialspoint.com/dart_programming/dart_programming_logical_operators.htm

https://medium.com/jay-tillu/operators-in-dart-5a774aff0788

https://github.com/dart-lang/sdk/issues/26996

https://stackoverflow.com/questions/54031804/what-are-the-double-question-marks-in-dart

https://dartpad.dev/?null_safety=true

https://www.tutorialspoint.com/dart_programming/dart_programming_functions.htm

https://www.geeksforgeeks.org/short-circuit-evaluation-in-programming/

http://www.faadooengineers.com/online-study/post/cse/principals-of-programming-language/924/short-circuit-evaluation

https://en.wikipedia.org/wiki/Short-circuit_evaluation#Possible_problems

JavaScript

1) How are the boolean values represented?

JavaScript booleans are primitive types that are created from literals. Boolean types are represented with the words “true” and “false”. However, in addition to these, 0, -0, the BigInt zero “0n”, null, false, NaN, undefined and the empty string “” are all evaluated as false. These are sometimes called “falsy” values as well. Anything except these, is considered true in the JavaScript language. Any object whose value isn’t null or undefined is interpreted as true (including an empty array [] or even the string “false”), so even when a Boolean object with the value false is created (new Boolean(false)), it is interpreted as true (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Boolean).

```
1.  console.log( 7 > 1); // prints "true"
2.  console.log( 1 > 31); // prints "false"
3.
4.  console.log( Boolean( 0)); // false
5.  console.log( Boolean( -0)); // false
6.  console.log( Boolean(0n)); // false
7.  console.log( Boolean( "")); // false
8.  let x; // x is undefined
9.  console.log( Boolean( x)); // undefined is false
10. x = null;
11. console.log( Boolean( x)); // null is false
12. x = false;
13. console.log( Boolean( x)); // false is false
14. x = "Efe" / 5; // x is NaN
15. console.log( Boolean( x)); // NaN is false
16.
17. x = "Efe";
18. console.log( Boolean( x)); // x has a value now, so it is "true"
19. x = []; // empty array Object
20. console.log( Boolean( x)); // interpreted as true
21. x = "false";
22. console.log( Boolean( x)); // interpreted as true
23. x = new Boolean(false); // x is a boolean object
24. console.log( Boolean( x)); // interpreted as true, x has a value
```

2) What operators are short-circuited?

Short-circuited operators in JavaScript are logical AND && and logical OR || operators. The && operator “evaluates operands from left to right, converts each operand to a boolean and if the result is false, stops and returns the original value of that operand. If every operand was truthy, returns the last operand” (https://javascript.info/logical-operators#and). Similarly, the || operator evaluates from left to right until it finds a true value, and returns that operand. Otherwise, if all operands have been evaluated to false, it returns the last operand. Additionally, the “Nullish coalescing operator (??)” returns the second operand if the first operand is null or undefined e.g. “x = null ?? 5;” sets x to null. This operator also short circuits in that if the first operand isn’t null or undefined the second operand is not evaluated.

```
1.  let z = 1 && "efe" && 0 && []; // sets x = 0
2.  console.log( z); // prints "0", && returns the first falsy value it encounters
3.
```

```
4.  z = 1 && "efe" && true && [];
5.  console.log( z); // prints "[]", sets x to the last operand since all of them are
    truthy
6.
7.  let y = 0 || "" || 31 || 0n; // sets y = 31
8.  console.log( y); // prints "31", || returns the first truthy value it encounters
9.
10. y = 0 || "" || -0 || null;
11. console.log( y); // prints "null", sets y to the last operand because all of them
    are falsy
12.
13. let p = null ?? 5;
14. console.log( p); // prints 5
```

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

The logical OR || operator evaluates operands from left to right until it encounters a truthy value, at which point it stops and returns that operand, ignoring the rest of the operands. Similarly, logical AND && operator evaluates operands from left to right until coming across a falsy value and immediately returns that operand, skipping the rest of the operands and not evaluating them. The ?? operator only computes the second operand if the first operand provided is equal to null or undefined. Otherwise, the value of the first expression is simply returned.

```
1.  console.log( "AND operation");
2.  let a = 5;
3.  let b = 10;
4.  if ( 0 && a++ > b) {
5.    // does not enter here
6.  }
7.  else {
8.    console.log( a); // prints "5", a wasn't incremented due to short-circuiting
9.  }
10.
11. if ( true && b++ > a) console.log( b); // prints "11", b was incremented
12.
13. false && console.log( "not printed"); // not printed because AND operation ends as
    soon as false is encountered
14.
15. true  && console.log( "printed!"); // prints "printed!" because first operand is
    true, so second operand is checked
16.
17. console.log( "OR operation");
18. a = 5;
19. b = 10;
20.
21. if ( true || ++b > a ) {
22.   console.log( "b is " + b); // prints "b is 10", b isn't incremented due to short
    circuiting
23. }
24.
25. if ( false || ++b > a ) {
26.   console.log( "b is " + b); // prints "b is 11", b is incremented
27. }
28.
29. 39 || console.log( "not printed"); // not printed because first operand is truthy
30. undefined || console.log( "printed"); // now this is printed because first operand
    is falsy
31.
32. console.log( "?? operator");
33. let l = undefined ?? a++;
```

```
34. console.log( l); // prints 5, a is not incremented first
35. console.log( a); // prints 6, a was incremented because second operand was evaluated
36.
37. null ?? console.log( "entered here"); // logs this
38. 5 ?? console.log( "not logged"); // this time it isn't called due to short
    circuiting
```

4) What are the advantages about short-circuit evaluation?

Firstly, short-circuit evaluation may be used to avoid runtime errors when an argument must be evaluated after doing a check, such as valid index checking for an array. Also, tasks that are costly may be avoided if they are only supposed to be done only when another situation is fulfilled.

```
1.  var arr1 = [31,32,33,34,35,36,37];
2.
3.  for ( let i = 0; i <= arr1.length; i++) { // loop continues until i = length, so out
    of bounds error should occur
4.    if ( i < arr1.length && arr1[i] > 30) { // no error is given due to the first
    expression
5.      console.log( arr1[i]); // prints all of the array's contents without an error
6.    }
7.  }
```

As can be seen, errors may be avoided with the help of short-circuit evaluation.

```
1.  if ( arr1.length > 15 && func1()) {
2.    // do something...
3.  }
```

From here, we can see that func1() is only called if the array has a length greater than 15, so unless that condition is satisfied func1() is never called, possibly increasing the performance of the program and avoiding unnecessary calls.

5) What are the potential problems about short-circuit evaluation?

Short-circuit evaluation may lead to reliability issues, since if the programmer disregards short-circuit evaluation and writes code accordingly, they may run into logical errors in their program.

```
1.  let q = 150;
2.
3.  for ( let i = 0; i < 10; i++) {
4.    if ( false && q++) { // q is never incremented
5.      console.log( "not entered"); // never entered
6.    }
7.    else {
8.      console.log( q); // always prints "150"
9.    }
10. }
```

Here, the variable q is never incremented since the first expression in the if statement is false, so the second expression isn't even evaluated.

```
1.  if (  true || func1()) {
```

```
2.    // do something...
3.  }
```

In this case as well, since the first operand is “true”, the OR operation is short circuited and the second expression is never evaluated. In this case func1() is not called, and if that method carries out an important task for the program, this may lead to bugs in the code.

Resources used for JavaScript

https://onecompiler.com/javascript

https://www.w3schools.com/js/js_booleans.asp

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Boolean

https://www.tutorialsteacher.com/javascript/javascript-boolean

https://developer.mozilla.org/en-US/docs/Glossary/Falsy

https://developer.mozilla.org/en-US/docs/Glossary/Truthy

https://codeburst.io/javascript-short-circuit-conditionals-bbc13ac3e9eb

https://www.geeksforgeeks.org/javascript-short-circuiting/

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Logical_AND

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Nullish_coalescing_operator

PHP

1) How are the boolean values represented?

PHP uses the bool type to represent boolean values, and the keywords “true” and “false” are used. However, these keywords are case-insensitive, so “True”, “TRUE” and “true” all simply evaluate to true, and “False”, “FALSE”, or “false” all evaluate to false. Values can be explicitly converted to boolean with the (bool) or (boolean) casts but in most cases a value will be automatically converted to boolean if a boolean argument is required.

The boolean false itself, integers 0 and -0, floats 0.0 and -0.0, empty string “” and the string “0”, an array with zero elements, special type NULL, and SimpleXML objects created from attributeless empty elements are all evaluated to false. Every other value is considered true, including NAN (https://www.php.net/manual/en/language.types.boolean.php).

```
1.  var_dump( (bool) true); // bool(true)
2.  var_dump( (bool) True); // bool(true)
3.  var_dump( (bool) false); // bool(false)
```

```
4.  var_dump( (bool) FALSE); // bool(false)
5.
6.  var_dump( (bool) 0); // bool(false)
7.  var_dump( (bool) -0); // bool(false)
8.  var_dump( (bool) 0.0); // bool(false)
9.  var_dump( (bool) -0.0); // bool(false)
10. var_dump( (bool) ""); // bool(false)
11. var_dump( (bool) "0"); // bool(false)
12. var_dump( (bool) []); // bool(false)
13. var_dump( (bool) NULL); // bool(false)
14.
15. var_dump( (bool) "efe"); // bool(true)
16. var_dump( (bool) 31); // bool(true)
17. var_dump( (bool) [1,2,3]); // bool(true)
18. var_dump( (bool) NAN); // bool(true)
```

2) What operators are short-circuited?

Operators that are short-circuited in PHP are the logical AND (&& and and), logical OR (|| and or), and the null coalescing (??) operators. The difference between && and “and” operators is that the && operator has higher precedence. Same follows for the || and or operators.

From the PHP manual (https://www.php.net/manual/en/language.operators.logical.php):

```php
// foo() will never get called as those operators are short-circuit

$a = (false && foo());
$b = (true  || foo());
$c = (false and foo());
$d = (true  or  foo());

// --------------------
// "||" has a greater precedence than "or"

// The result of the expression (false || true) is assigned to $e
// Acts like: ($e = (false || true))
$e = false || true;

// The constant false is assigned to $f before the "or" operation occurs
// Acts like: (($f = false) or true)
$f = false or true;

var_dump($e, $f);

// --------------------
// "&&" has a greater precedence than "and"

// The result of the expression (true && false) is assigned to $g
// Acts like: ($g = (true && false))
$g = true && false;
```

```php
// The constant true is assigned to $h before the "and" operation occurs
// Acts like: (($h = true) and false)
$h = true and false;
```

```php
// logical AND
var_dump( 1 && 0); // bool(false)
var_dump( "" and 90); // bool(false)

// logical OR
var_dump( -0 || 78); // bool(true)
var_dump( null or [1,2]); // bool(true)

// null coalescence
var_dump( null ?? 3); // prints "int(3)", if the first operand is null, returns the second operand
```

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

All of the operators discussed in the answer for Q2 evaluate operands from left to right and stop at the point when they don't need to evaluate no more. || and or operators stop evaluating when they encounter a true value, && and and operators stop evaluating when they encounter a false value, and the ?? operator stops evaluating once it encounters a non-null value.

```php
function doSomething() {
  echo "inside\n";
  return true;
}
// logical AND
echo "Logical AND \n";
$a = 10;
$b = 20;

if ( false && $b++ > $a) {
  // won't enter here
}
else {
  var_dump( $b); // prints "int(20)", b is not incremented due to short-circuiting
}

if ( true && ++$b > $a) var_dump( $b); // prints "int(21)", b is incremented

false && var_dump($b); // doesn't print anything, function is never called due to short short-circuiting
true && var_dump( "called"); // prints "string(6) "called"", function is called because first operand is true

if ( false and $b++ > $a) {
  // won't enter here
}
else {
  var_dump( $b); // prints "int(21)", b is not incremented due to short-circuiting
}

if ( true and ++$b > $a) var_dump( $b); // prints "int(22)", b is incremented

false and doSomething(); // doesn't print anything, function is never called due to short short-circuiting
```

```php
true and doSomething(); // prints "inside", function is called because first operand is true

// logical OR
echo "Logical OR \n";
$a = 10;
$b = 20;

if ( true || ++$b > a) {
  var_dump( $b); // prints "int(20)", b is not incremented
}

if ( false || $b++ > $a) var_dump( $b); // prints "int(21)", b is incremented this time

$temp = true || doSomething(); // doesn't call doSomething() method
var_dump( $temp); // prints "bool(true)", result of the expression is assigned to temp
false || doSomething(); // prints "inside", fuıction is called

$a = 10;
$b = 20;

if ( true or ++$b > a) {
  var_dump( $b); // prints "int(20)", b is not incremented
}

if ( false or $b++ > $a) var_dump( $b); // prints "int(21)", b is incremented this time

$temp = true or doSomething(); // doesn't call doSomething() method
var_dump( $temp); // prints "bool(true)", temp is assigned the value of the first operand
false or doSomething(); // prints "inside", fuıction is called

// null coalescence
echo "null coalescent operator \n";
$a = 10;
$b = $a ?? $a++;
var_dump( $b); // prints "int(10)", b is assigned to the value of $a
var_dump( $a); // prints "int(10)", a wasn't incremented

$b = null ?? ++$a;
var_dump( $b); // prints "int(11)", b is assigned to the value of $a + 1
var_dump( $a); // prints "int(11)", a was incremented

$a ?? doSomething(); // doesn't print anything, method is not called
null ?? doSomething(); // prints "inside", method is called because first operand is null
```

4) What are the advantages about short-circuit evaluation?

Short-circuit evaluation in PHP may help with avoiding errors whenever a check needs to be made before evaluating another expression. Also, short-circuit evaluation may prevent the carrying out of unnecessary and costly computations.

```php
$arr1 = [31,32,33,34,35,36];

if ( count($arr1) > 8 && $arr1[7]) {
  // won't enter here because array size is smaller than 8, but no error will be thrown due to short circuiting
```

```
}

$doIt = false;

if( $doIt && doSomething()) { // if the boolean value $doIt is not true, doSomething() will never be called unnecessarily
  // do other stuff
}
```

5) What are the potential problems about short-circuit evaluation?

Short-circuit may lead to reliability or performance issues if used inadvertently.

```
$x = 10;
$bool = true;
if ( $bool or $x++) { // second expression won't be evaluated here
  var_dump( $x); // prints "int(10)", x is not incremented
}
```

Here, if the programmer assumes x will be incremented, this may lead to a logical error in the code.

```
$bool = false;
if ( $bool and func1()) { // func1() will never be called
  func2();
}
```

In this code segment, if func1() is supposed to do an operation that is independent of func2() or is necessary for the remainder of the program, it may lead to problems since due to short-circuiting, here the second expression will never be evaluated after the program encounters the false variable in the first expression.

Resources used for PHP

https://www.php.net/manual/en/language.types.boolean.php

https://www.phptutorial.net/php-tutorial/php-boolean/#:~:text=A%20boolean%20value%20represents%20a%20truth%20value%2C%20which%20is%20either,array%3B%20other%20values%20are%20true%20.

https://wiki.php.net/rfc/null_coalesce_equal_operator

https://www.php.net/manual/en/language.operators.precedence.php

https://www.php.net/manual/en/language.operators.comparison.php#language.operators.comparison.coalesce

https://www.php.net/manual/en/language.operators.logical.php

https://newbedev.com/does-php-have-short-circuit-evaluation

https://wiki.php.net/rfc/null_coalesce_equal_operator

https://onecompiler.com/php

Python

1) How are the boolean values represented?

Boolean values in Pyhon are “True” and “False”. However, a lot of values in Python are also evaluated to True, such as any string other than empty strings, any number except 0 and non-empty lists, tuples, sets and dictionaries. (), [], {}, “”, the number 0, void functions and the value None all evaluate to False (https://www.w3schools.com/python/python_booleans.asp). In addition, Booleans are considered numeric in Python, meaning they’re numbers and arithmetic operations can be done with them.

```
1.  print( 58 > 43) # True
2.  print( 1 > 2) # False
3.
4.  print( bool(())) # False
5.  print( bool([])) # False
6.  print( bool({})) # False
7.  print( bool("")) # False
8.  print( bool(0)) # False
9.  print( bool(None)) # False
10.
11. print( bool("CS315")) # True
12. print( bool([1,2,3,4,5])) # True
13. print( bool((4,5,6))) # True
14. print( bool(69)) # True
15.
16. print( True + True) # prints "2", True corresponds to the number 1
```

2) What operators are short-circuited?

The “and” and “or” operators in Python are short-circuited. Also, since comparison operators can be arbitrarily cascaded in Python (https://docs.python.org/2/library/stdtypes.html#comparisons) these are also short circuited. But this situation is not different than “and” and “or” operations, since for example the operation “1 < 2 < 3” is evaluated as “1 < 2 and 2 < 3” in Python, so the underlying operation is still an “and” operation. The below picture shows how the “and” and “or” operations in Python work:

OPERATION	RESULT	NOTES
X or Y	If X is False, then Y, else X	Y is executed only if X is False Else if X is true, X is result.
X and Y	If X is false, then X else Y	Y is executed only if X is true, else if X is false , X is result.
not X	If X is true, then false, else true	not has lower priority than non - boolean operators. Eg. not a==b => not (a==b)

Picture source: https://www.geeksforgeeks.org/short-circuiting-techniques-python/

```
print( bool( 1 and 2)) # prints "True"
print( 1 and 2) # prints "2"
print( True and False) # prints "False"

print( bool( {} or [])) # prints "False"
print( {} or []) # prints []
print( True or True) # prints True

print( 1 < 2 < 3) # prints True (evaluated 1 < 2 and 2 < 3)
print( 2 < 1 < 3) # prints False
print( 1 == 1 == 2) # prints False (evaluated 1 == 1 and 1 == 2)
```

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

The results of “and” and “or” operations are evaluated from left to right. In an “and” operation, the execution stops when either a “False” value is encountered or the end has been reached. Similarly, in an “or” operation, execution stops when either a “True” input is reached or all the operands have been exhausted. The operand where the execution has stopped is returned in both cases.

```
def fun():
  print( "inside function")
  return True

print( "and operation")
a = 10
b = 20
c = False

if ( c and b / 0): # b / 0 operation won't be done because of short circuiting, so no error
  b = 10 # won't enter here
else:
  print( "inside else") # this is printed


if ( c and fun()): # function is not called due to short circuiting
  a = 5 # not entered here

if ( a and fun()): # prints "inside function"
  print( "inside if") # this is printed

print( "or operation")
c = True
if ( c or a / 0): # due to short circuiting, second expression is not evaluated
  print( "inside if for or operation") # this is printed

if ( b or fun()): # fun() is not called here
  print( "function is not called") # this is printed

if ( False or fun()): # prints "inside function", because function is called
  print( "function is called") # this is printed

print( "comparison operators")
print ( 2 < 1 <  3 / 0) # prints False, last expression is not evaluated (2 < 1 and 1 < 3 / 0)
```

4) What are the advantages about short-circuit evaluation?

With the inclusion of short-circuit evaluation in Python, possible run-time errors due to zero-divisions or invalid indices in array operations etc. can be avoided. Also, short-circuit evaluation provides a way to avoid unnecessary computations in the code.

```
1.  x = 32
2.  y = 53
3.  z = 0
4.
5.  if ( y > x or y / z < 3): # second expression is a division by 0, however due to
    short-circuiting no error will be generated
6.    print( "no error thrown")
7.
8.  list1 = [1,2,3,4,5]
9.  z = 8
10. if ( z < len( list1) and list1[z] > 4): # no out of bounds error
11.   print( "index not out of bounds") # won't be printed since the value of z is out
    of bounds
12.
13. if ( z > 100 and fun()): # fun() won't be called unnecesarrily if z is not greater
    than 100
14.   print( "do something...")
```

5) What are the potential problems about short-circuit evaluation?

Short-circuit evaluation may possibly lead to bugs or other problems in the code.

```
1.  z = 190
2.
3.  def fun2():
4.    print( "inside fun2()")
5.  if ( z > 100 or fun()): # fun() will not be called in this situation
6.    fun2() # prints "inside fun2()"
7.    # do something
```

In this code segment, if z > 100, then fun() will never be called. If fun() carries out an important operation for the program, this operation will never be carried out. This may lead to problems in the execution of the program.

Resources used for Python

https://onecompiler.com/python/3xj4aq8v7

https://www.w3schools.com/python/python_booleans.asp

https://realpython.com/python-boolean/#short-circuit-chain-evaluation

https://www.pythonpool.com/short-circuit-evaluation-python/

https://docs.python.org/2/library/stdtypes.html#comparisons

https://stackoverflow.com/questions/2580136/does-python-support-short-circuiting

Rust

1) How are the boolean values represented?

Booleans in Rust are represented with the “bool” keyword and are created in the following way: “let x: bool = true” (https://doc.rust-lang.org/reference/types/boolean.html). Also, “true” corresponds to “1” while false corresponds to “0”. No other type or value is considered as a boolean in the Rust language other than the literals “true” and “false”.

```
println!( "{}", 1 > 0); // prints "true"

let x: bool = false;
println!( "{}", x); // prints "false"
let y: i32 = x.into(); // turnx x into integer
println!( "{}", y); // prints "0"
```

The below code gives an error:

```
if ( 1) {
    println!( "{}", y); // prints "0"
}
```

2) What operators are short-circuited?

Short-circuited operators in Rust are logical AND (&&) and logical OR (||) operators, these are also called the “Lazy Boolean operators”. These operators only “evaluate the right-hand operand when the left-hand operand doesn’t already determine the result of the expression” (https://doc.rust-lang.org/reference/expressions/operator-expr.html#lazy-boolean-operators).

```
let a: i32 = 5;
let b: i32 = 9;

println!( "{}", a < b && b > 3); // prints "true"
println!( "{}", a > 1 || b > 45); // prints "true"
```

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

Logical AND and logical OR operators evaluate expressions from left to right, and stop evaluation once the output can be determined without the need for further computations. For example, the logical AND operation stops execution when it encounters a false expression, whereas the logical OR operator stops execution when it encounters a true expression.

```
fn main() {
    println!( "logical AND");
    let n = 31;
    let m = 45;

```

```
    if false && n / 0 > 0 { // due to short-circuiting, doesn't give an error here because the zero division is never evaluated
        // won't enter here
    }

    if n > m && bool_func() { // bool_func is not called
        // won't enter here
    }

    if n < m && bool_func() { // prints "inside function", bool_func is called now
        println!( "inside if!"); // this will be printed
    }

    println!( "logical OR");
    if true || m / 0 < 7 { // only first operand will be evaluated since it is enough to determine the result of the OR operation
        println!( "Inside if for OR"); // no error will be thrown due to zero division
    }

    if m > n || bool_func() { // function won't be called
        println!("bool_func() is not called");
    }

    if m < n || bool_func() { // function will be called now
        println!("bool_func() is called");
    }
}
fn bool_func() -> bool {
    println!("inside function");
    return true;
}
```

4) What are the advantages about short-circuit evaluation?

With the inclusion of short-circuit evaluation in Rust, possible run-time errors due to zero-divisions or invalid indices in array operations etc. can be avoided. Also, short-circuit evaluation provides a way to avoid unnecessary computations in the code.

```
if true || 7 / 0 > 5 {
    println!( "error avoided");
}
```

The below code segment shows how unnecessary computations may be avoided:

```
if false && func() { // func() won't be called unnecessarily if the first operand is false
    // do something...
}
```

5) What are the potential problems about short-circuit evaluation?

Short-circuit evaluation may possibly lead to bugs or other problems in the code.

```
let k = 98;

```

```
3.  if k > 95 || bool_func() {
4.      // bool_func() will not be called in this situation
5.  }
```

In this code segment, if k > 95, then bool_func() will never be called. If this function carries out an important operation for the program, this operation will never be carried out. This may lead to problems in the execution of the program if the programmer depends on this function getting called even though the first operand evaluates to true.

Resources used for Dart

https://play.rust-lang.org/

https://doc.rust-lang.org/std/primitive.bool.html

https://doc.rust-lang.org/reference/types/boolean.html

https://doc.rust-lang.org/reference/expressions/operator-expr.html#lazy-boolean-operators

https://stackoverflow.com/questions/55461617/how-do-i-convert-a-boolean-to-an-integer-in-rust

https://doc.rust-lang.org/rust-by-example/flow_control/if_else.html

https://doc.rust-lang.org/book/ch03-03-how-functions-work.html

Best Language for Short-circuit Evaluation

After having experimented with 5 different languages and the way their short-circuiting mechanisms work, I believe Python is the best amongst these languages for short-circuit evaluation. First of all, unlike Dart or Rust, Python offers more variety for its Boolean values. Namely, any string other than empty strings, any number except 0 and non-empty lists, tuples, sets and dictionaries are all evaluated to true while (), [], {}, “”, the number 0, void functions and the value None are all evaluated to False. I believe this aspect of Python makes it more writable and practical, since it is easier to check if a string, list, tuple, set or dictionary is empty without having to write longer code. However, of course this makes Python less readable compared to Rust or Dart because it is harder to understand Boolean expressions when looking at Python code. Secondly, all the languages I have experimented with feature short-circuit evaluation for logical AND and logical OR operators, however in addition to these Python also offers the ?? operator, as do PHP and Dart. I think the addition of this operator is also a fine addition and a nonnegligible difference compared to Rust and JavaScript. Results of short-circuited operators are all evaluated the same way in all languages, from left to right until an operand is enough to determine the result of a Boolean operation. However, Python also returns that operand rather than simply returning True or False, which adds to its usability. Finally, all the languages have similar advantages and disadvantages regarding short-circuit evaluation, so Python isn’t much different in this aspect than others. All in all, I believe Python is the best language for short-circuit evaluation.

Learning Strategy

For every language, I first did research on the Internet to understand how Boolean values are represented in a particular language. Then, for every language I researched their short-circuiting operators and tried to find differences between the ways they evaluate Boolean expressions, if any. Finally, I experimented with them to see if their short-circuiting mechanisms help with preventing any problems or if there are any significant disadvantages to their respective short-circuiting systems.

Dart:

For Dart, I used the online compiler at https://dartpad.dev/?null_safety=true. For the first question, I looked for Dart's boolean values and which values are considered true or false in Dart. For this, I consulted https://www.tutorialspoint.com/dart_programming/dart_programming_boolean.htm and https://www.tutorialspoint.com/dart_programming/dart_programming_logical_operators.htm. Then, I researched about Dart's short-circuiting operators and found them at this link https://medium.com/jay-tillu/operators-in-dart-5a774aff0788. To see how these operators worked, I wrote code and experimented on the online compiler, to unearth any problems related to short-circuiting or how it could come in handy. My experiments included ways to see how expressions would be skipped in boolean expressions if the result can be evaluated without exhausting all the operands. As a result of my experiments I was familiar with the way short-circuit evaluation works in Dart. For the last part, I looked for common problems with short-circuit evaluation as well as any Dart specific problems I could find. I couldn't find anything Dart-specific but these links helped me with understanding possible problems or advantages to short-circuit evaluation: http://www.faadooengineers.com/online-study/post/cse/principals-of-programming-language/924/short-circuit-evaluation and https://www.geeksforgeeks.org/short-circuit-evaluation-in-programming/

JavaScript:

For JavaScript, I used the compiler at https://onecompiler.com/javascript. Regarding the first question, I searched on the internet to learn about JavaScript Booleans. As a result of my research, I found out that, except regular true and false literal values, JavaScript features truthy and falsy values which are values that are evaluated to true or false in Boolean context, however aren't necessarily Boolean values themselves. https://developer.mozilla.org/en-US/docs/Glossary/Falsy and https://developer.mozilla.org/en-US/docs/Glossary/Truthy helped me understand truthy and falsy values. Following this, I looked at this link https://www.geeksforgeeks.org/javascript-short-circuiting/ to learn about JS short-circuiting. I learned that JS "evaluates operands from left to right, converts each operand to a boolean and if the result is false, stops and returns the original value of that operand. If every operand was truthy, returns the last operand" (https://javascript.info/logical-operators#and). Similarly, the || operator evaluates from left to right until it finds a true value, and returns that operand. Otherwise, if all operands have been evaluated to false, it returns the last operand. The advantages and disadvantages of short-circuiting in JS are more or less the same as other languages.

PHP:

For PHP, I used the compiler at https://onecompiler.com/php. To understand PHP Booleans, I consulted https://www.php.net/manual/en/language.types.boolean.php and realized that like JS, PHP also interprets values other than "true" or "false" as Booleans. For example, integers 0 and -0, floats 0.0 and -0.0, empty string "" and the string "0", an array with zero elements, special type NULL, and SimpleXML objects created from attributeless empty elements are all evaluated to false. Every other value is considered true, including NAN (https://www.php.net/manual/en/language.types.boolean.php). For the second question, I found out that in addition to the && and || operators, PHP has the ?? operator. I read up on this operator from https://wiki.php.net/rfc/null_coalesce_equal_operator and experimented myself to understand how short-circuiting works. From this link https://newbedev.com/does-php-have-short-circuit-evaluation I learned about the basics of short-circuiting in PHP and concluded my tests. The answers to the last two questions didn't differ much for PHP compared to other languages.

Python:

For Python, https://onecompiler.com/python/3xj4aq8v7 I used this compiler. I learned about Python Booleans from https://www.w3schools.com/python/python_booleans.asp and read up on how Python interprets which types as True or False. Following this, I followed this link https://www.pythonpool.com/short-circuit-evaluation-python/ to understand short-circuiting in Python and conducted tests to display how Python allows cascading comparison operators and how this cascading simply turns expressions into Boolean operations underneath. As a result, I was able to display short-circuiting in Python and ran some more tests to denote how problems regarding short-circuiting in Python could occur.

Rust:

For Rust, I used the compiler at https://play.rust-lang.org/. Then I learned about Rust Booleans at https://doc.rust-lang.org/reference/types/boolean.html and understood that Rust doesn't support any other values as Booleans other than simply "true" and "false". For the next question, I learnt that short-circuited operators in Rust are && and ||, and that they are called "lazy Boolean operators" (https://doc.rust-lang.org/reference/expressions/operator-expr.html#lazy-boolean-operators). I found out that these operators evaluate expressions from left to right until an operand that is solely enough to determine the result of the whole expression is encountered, at which point the execution stops. I wrote tests to display this property. The answers to the last two questions were more or less the same as other languages, and my code for those questions shows how short-circuiting can help avoid errors in code as well as lead to unreliable situations.