Bilkent University

CS 315- PROGRAMMING LANGUAGES

Ali Eren Günaltılı

21801897

Section 01

CS315- Homework 02

Topic: Shortcircuit Evaulation in PHP, Javascript, Python, Dart and Rust

1- **How are the boolean values represented?**

a. Javascript -> In javascript boolean values are represented with true and false keywords.True and false keywords are case sensitive in javascript. Everything with a value is true. Everything without a value is false. !(not), &&(and) and ||(or) operators are provided for booleans.

```
console.log("Boolean Representation");
var x = 0;
var y = 'abc';
var z = -15;
console.log(x); //prints 0
console.log(Boolean(x)); // prints false
console.log(Boolean(y)); // prints true
console.log(Boolean(z)); // prints true
console.log(!x); // prints true
console.log(!y); // prints false
var bool = true;
console.log(bool); // prints true
console.log(!bool); //prints false
console.log("-----------------");
```

b. Python -> In python booleans are represented with 'True' and 'False'. Booleans are built-in object. Boolean's method and properties can be used by using bool() object. Thus, any integer, string or float value can be converted to boolean. To use some value as a boolean is possible from two different ways. One of them is using bool().

Another is using not keyword before any variable. It also converts the value to boolean value. In python boolean can be used as integers. We can use booleans in arithmetic or comparison operations.

```
#True = 1, False = 0
print(True + False * True) # 0 + 1 * 1 = 1
print(5 * x + y ) #prints 5, Because x is True.
print(1 == x) #prints True
print(x > 0) #prints True 1 >0
print(y > 0) # 0 is not greater than 0. Prints False
```

False values in python language: None, False, 0, 0.0, 0j, empty sequence () [] ' ' , empty mapping {} and lastly objects of Classes which has __bool__() or __len()__ method which returns 0 or False.

```python
x = True
y = False
tmp = 0
z = -11
ch = 'app'

print(x) # true
print(type(x)) #print <class 'bool'>
print(y) # false
print(z) # -11
print(tmp) # 0
print('--------------')
print(bool(x)) # true
print(bool(y)) # false
print(bool(z)) #true
print(bool(tmp)) #false
print('-----------------')
print(not x) #False
print(not tmp) #True
print(bool(ch)) #True
```

c. PHP -> PHP booleans are scalar and primitive type. PHP also represents booleans as 'True' and 'False' but it is not case sensitive. That means True and true are same thing for PHP. A boolean value represents a truth value, which is either true or false. PHP evaluates the (false,0, 0.0, False, empty string(""), "0", NULL, an empty array) to false, other values are true. In Php allowed operators are: and, or, xor, !(not), &&, ||, ??, +, *. Php has different attributes than other languages. Operator precedence is crucial for boolean arithmetics in php because "and" keyword and && have different precedence.

```php
$a = false;
$b = true;
$c = $b and $a; // assignment has higher precedence than "and". Thus
// b directly takes the c's value.
$d = ($b and $a); // () has higher precedence than assignment. Therefore first
"and" is executed
// inside the parantheses.
$e = $b && $a; // e = false because && has higher precedence than assignment
operator.
var_dump((bool) $c); //1
var_dump((bool) $d); //2
var_dump((bool) $e); //3
var_dump($b > $a);   //4
var_dump((bool) $a > $b); //5
var_dump($a * $b); //6
```

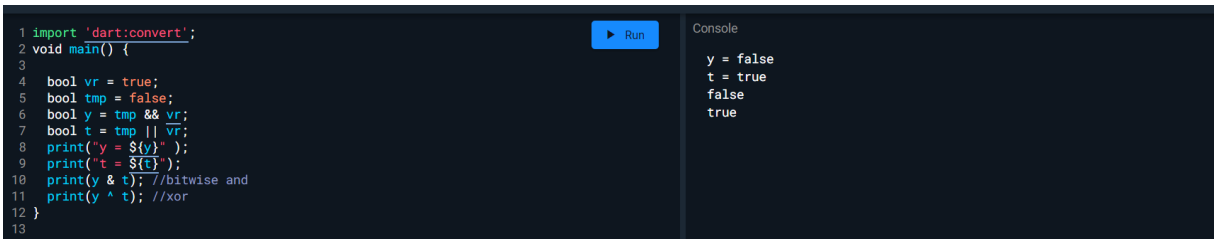
```
var_dump($a + $b); //7

//output
1-bool(true)
2-bool(false)
3-bool(false)
4-bool(true)
5-bool(false)
6-int(0)
7-int(1)
```

Important notes are operator precedence between “and” and &&, “or” and ||. Another interesting thing is that php can make boolean comparison and as false value means zero and true means something. True is always bigger than false. It also prints true or false as a result of boolean comparison. It also use booleans in arithmetic operations but this time output will be integer type not boolean. In arithmetic operations true means 1 and false means 0.

d. Dart -> Dart uses true and false in boolean representation. They are both compile-time constants. In Dart, The numeric value 1 or 0 cannot be used to specify the true or false. The bool keyword is used to represent the Boolean value. Unlike Php, Dart booleans are very strict and straighforward. But the things you can do with Dart boolean is limited because of that. Dart accepts true as a true other than that everything is small. Integer values which is not zero or not empty strings are still false. They cause exception if the program is running in checked mode. In unchecked mode, program considers them as a false boolean value.

```
var str = 'abc';
if(str) {
  print('String is not empty');
}
else {
  print('Empty String');
}
```

This if-else causes error because str is not bool type variable. If the program is running in unchecked mode it prints ‘Empty String’. &&, &, ^, ||, |, ! , ==, != operators are provided.



e. Rust -> Boolean type in rust is a primitive type which can takes only two values; true or false. Also to create booleans in rust, bool keyword had to use. Like all primitives in Rust, the boolean type implements the traits Clone, Copy, Sized, Send, and Sync. Provided operators are not(!), or(|), and(&), xor(^) and comparison operators(==, <,>, <=, >=).
True means 0x01 in rust, false means 0x00. Comparison operators use this when they have to compare two boolean type variable's value.

```rust
fn main() {
    let b:bool = true;
    let a:bool = false;
    let c:bool = a |b;
    println!("{}", c); //true
    let d:bool = a >=b;
    let e:bool = b >= a;
    println!("{}", d); //false
    println!("{}", e); //true
}
```

2. What operators are short-circuited ?

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

I am going to answer these two question in same place as they are related to each other.

a. Javascript
In javascript logical and(&&) and logical or(||) operators are short circuited.

```javascript
function fun(b){
    if(b > 0){
        console.log("in fun"); return "abc";

      }

}

const foo = (b) => {

    console.log("in foo");
    return true;
}
1-)console.log(false && fun(5)); // print false
2-)console.log(true && fun(5)); // print abc goes into the fun
3-)console.log(0 * foo(5)); // no short circuit evaluation for multiplication
operator
4-)console.log(0 && foo(5)); // won't print "in foo" as expected
```

```
5-)console.log("---" + !0 || fun(5)); //print ---true (doesn't go fun function
at all)
6-)console.log(0 || foo(5)); //goes into the foo function as left side of the
logic or is not 1.
7-)console.log(1 && foo(5)); //goes into the foo function as left side of the
logic and is not 0.
8-)console.log(1 || foo(5)); //won't go into the foo after finding 1
9-)console.log("" && foo(5)); // won't go into the foo after finding empty
expression.
10-)console.log("abc" || foo(5)); //won't go into the foo after finding
nonempty expression.
11-)console.log(null && foo(5)); //won't go to foo but prints null
12-)console.log(foo(5) && fun(5)); // foo returns true. For logical and we go
to rightside too.
13-)console.log(foo(5) || fun(5)); //foo returns true and we never visits
after finding a nonempty value in or.
14-)console.log(fun(5) && foo(5)); //fun returns nonempty abc. But in and exit
condition is finding an empty value.
15-)console.log(fun(5) || foo(5)); //fun returns nonempty abc and rightside is
useless.




//output
1-false
2-in fun
  abc
3-in foo
  0
4-0
5----true
6-in foo
  true
7-in foo
  true
8-1
9-
10-abc
11-null
12-in foo
in fun
   abc
13-in foo
   true
14-in fun
   in foo
   true
```

```
15-in fun
   abc
```

In javascript everyting with a value means true, everything without a value means false. In lines with number 1, 4,11 and 9, && operations won't read the right side of the expression finding a empty value. "", '', null or false. In lines 5,8 and 10 logical or(||) operator uses short circut evaluation. Because we never go to rightside of or operation after finding a nonempty value. Boolean operations with two variables are easy to trace output. But short circuit evaluation also uses in boolean operations with more than two variables.

```
1-)console.log(0 || foo(5) || fun(5)); // 0 || don't cause short circuit.
After seeing foo(5) it won't go inside of fun.
2-)console.log(1 && "abcd" || foo(5)); // 1 and && don't cause short circuit.
But after seeing "abcd" and || cause short circuit.
3-)console.log(1 || "abcd" && 0); // see 1 and || operation then short circuit
the expression. Prints 1
//outputs
1-in foo
  true
2-abcd
3-1
```

In case of we have more than two variables, program find a pair of 0 and && or 1 and || to short circuit the evaluation. In first line 0 or foo(5) call won't cause short circuit, but after finding out that foo(5) returns something not empty it automatically shortcircuit the whole expression. In second 1 and && won't cause shortcircuit but after finding "abcd" and || expression is calculated as abcd. In line 3, finding || after a nonempty value causes short circuit and we get 1 as output.

//potential problems:
Problematic calculations on boolean operation with more than two variables
1 && "abcd" || 0 -> returns 1 why ?

b. Python

Python provides "and", "or", equality and comparison check operators for boolean operations. In python "and" and "or" operators are short circuited.

```
def foo():
    print('in foo')
    return True

def fun():
    print('in fun')
    return "abc"

def false():
    print('in false')
    return 0
val = -4
1-print(val or False) #val is not empty. Thus expression is short
circuited.
2-print(True or false()) #True and or cause short circuiting.
```

```
3-print(False and foo()) #If left side of the and is False, expression is
shortcircuited.
4-print(false() and (fun() or foo())) # this expression is short circuited
after finding the false() in left side of and
5-print(false() and fun() or foo()) # this means ((false() and fun()) or
foo()). false() and fun() = False. False or foo() = foo()
6-print(foo() and fun()) #True and "logical and" don't cause short circuit
evaluation
7-print(foo() or fun()) #foo() = True, True and "or" cause short circuit.

//outputs
1-)-4
2-)True
3-)False
4-)in false
   0
5-)in false
   in foo
   True
6-)in foo
   in fun
   abc
7-)in foo
   True
```

In lines 1,2,5 and 7 or expression is short circuited. If left side of the or is nonempty, it automatically calculates the or expression without reading the right side. In line 7 for instance, we never go into the fun function because foo function returns something nonempty. In lines 3 and 4 “logical and” operation is short circuited since the left side of the and is empty value. The reason why lines 4 and 5 have different output is that python treats different when there is a parantheses. If there is no parantheses like line 5, python starts to read the expression from left. It reads false() and “logical and”. Then reads fun() and “or”. As fun returns nonempty value, expression shortcircuited at that moment.

c. PHP

In php language, logical operators “and”, “or”, && and || are use short circuit. All four operators only evaluate the right side if they need to. But bitwise operators ‘&’ and ‘|’ don’t use short circuit.

```
function foo(){
    var_dump('in foo');
    return true;
}

function fun(){
    var_dump('in fun');
    return 'abc';
}

function falseFun(){
    var_dump('in false fun');
    return '';
```

```
}
$x = false;
$y = '';
$z = -5;
1-var_dump($y and $z); //after finding empty y left of and shortcircuit the
expression.
2-var_dump($z or $x); //nonempty variable on left of side of the or.
3-var_dump((bool) (foo() || falseFun())); //doesn't go inside of falseFun()
4-var_dump((bool) (false && foo())); // doesn't go inside of foo
5-var_dump((bool) (true or false and falseFun())); // after seeing true left
of the or, expression is short circuited.
6-var_dump(fun() * foo());
7-var_dump($x & foo()); //don't use shortcircuit
8-var_dump($z | falseFun()); //don't do shortcircuit eventhough left of the or
is nonempty.

//output
1-bool(false)
2-bool(true)
3-string(6) "in foo"
  bool(true)
4-bool(false)
5-bool(true)
6-string(6) "in fun"
  string(6) "in foo"
  int(0)
7-string(6) "in foo"
  int(0)
8-string(12) "in false fun"
  int(-5)
```

d. Dart

In Dart language logical and and logical or operators are shortcircuited. Bitwise operators which are also available for booleans in dart don’t use short circuit evaluation.

```
import 'dart:convert';
bool foo(){
  print("in foo");
  return true;
}

bool fun(){
  print("in fun");
  return false;
}
void main() {

  bool vr = true;
  bool tmp = false;
  bool y = tmp && vr;
  bool t = tmp || vr;
  print("y = ${y}" );
  print("t = ${t}");
  print(y & t); //bitwise and
  print(y ^ t); //xor
  print(vr || foo());
  print(tmp && foo());
  print(vr | foo());
  print(vr || fun() && false );
  print(false || vr && fun());
}
```

Run

Console

```
y = false
t = true
false
true
true
false
in foo
true
true
in fun
false
```

Documentation

Print(vr || foo()) => vr is true and it is on the left side of the or. Therefore foo() is dead code.

Print(tmp && foo()) => tmp is false and it is on the left side of the and. Thus, foo() is dead code.

Print(vr | foo()) => Although vr is true. | operator doesn't use short circuit evaluation.

Print(vr || fun() && false) => after finding vr and logical or, other parts of are dead code.

Print(false || vr && fun()) => false and logical or don't cause short circuit. Also vr and logical and don't cause short circuit. Therefore, in output we have in fun before false.

e. Rust

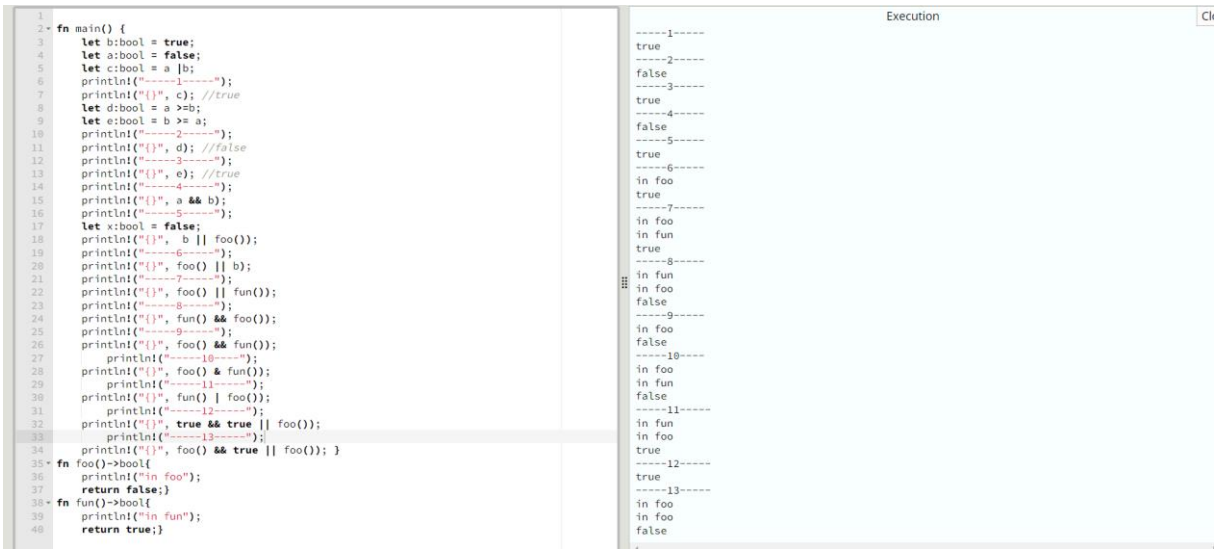
Rust language uses short circuit evaluation. We can see that from the definition of the language in official web page.

Syntax

OperatorExpression :
BorrowExpression
| *DereferenceExpression*
| *ErrorPropagationExpression*
| *NegationExpression*
| *ArithmeticOrLogicalExpression*
| *ComparisonExpression*
| *LazyBooleanExpression*
| *TypeCastExpression*
| *AssignmentExpression*
| *CompoundAssignmentExpression*

In rust language, lots of boolean operations are provided. But only && and || use shortcircuit.

In rust multiplication and addition is not provided for booleans.



```
fn main() {
    let b:bool = true;
    let a:bool = false;
    let c:bool = a |b;
    println!("-----1-----");
    println!("{}", c); //true
    let d:bool = a >=b;
    let e:bool = b >= a;
    println!("-----2-----");
    println!("{}", d); //false
    println!("-----3-----");
    println!("{}", e); //true
    println!("-----4-----");
    println!("{}", a && b);
    println!("-----5-----");
    let x:bool = false;
    println!("{}",  b || foo());
    println!("-----6-----");
    println!("{}", foo() || b);
    println!("-----7-----");
    println!("{}", foo() || fun());
    println!("-----8-----");
    println!("{}", fun() && foo());
    println!("-----9-----");
    println!("{}", foo() && fun());
        println!("-----10----");
    println!("{}", foo() & fun());
        println!("-----11-----");
    println!("{}", fun() | foo());
        println!("-----12-----");
    println!("{}", true && true || foo());
        println!("-----13-----");
    println!("{}", foo() && true || foo()); }
fn foo()->bool{
    println!("in foo");
    return false;}
fn fun()->bool{
    println!("in fun");
    return true;}
```

```
Execution
-----1-----
true
-----2-----
false
-----3-----
true
-----4-----
false
-----5-----
true
-----6-----
in foo
true
-----7-----
in foo
in fun
true
-----8-----
in fun
in foo
false
-----9-----
in foo
false
-----10----
in foo
in fun
false
-----11-----
in fun
in foo
true
-----12-----
true
-----13-----
in foo
in foo
false
```

```
println!("-----4-----");
println!("{}", a && b);
println!("-----5-----");
let x:bool = false;
println!("{}",  b || foo());
println!("-----6-----");
println!("{}", foo() || b);
println!("-----7-----");
println!("{}", foo() || fun());
println!("-----8-----");
println!("{}", fun() && foo());
println!("-----9-----");
println!("{}", foo() && fun());
    println!("-----10----");
println!("{}", foo() & fun());
    println!("-----11-----");
println!("{}", fun() | foo());
    println!("-----12-----");
println!("{}", true && true || foo());
    println!("-----13-----");
println!("{}", foo() && true || foo()); }
```

4 = shortcircuited because a is false and it is on the left of the logical and.

5=shortcircuited because b is true and it is on the left of the logical or. Foo is dead code.

6= reverse order of 5. Only difference is code goes into the foo function this time.

7= no shortcircuit evaluation as foo returns false and fun returns true.

8= no shortcircuit evaluation fun returns true and foo returns false.

9= shortcircuited because foo returns false and it is on the left of the logical and.

10 and 11 are bitwise operator. Thus, they don't use shortcircuit evaluation.

12= true and true = true. Then true value will be on left of the logical or. Thus, expression is shortcircuited.

13= foo && true = false since foo returns false. False and logical or don't cause shortcircuit so we go into the foo function too.

```
-----1-----
true
-----2-----
false
-----3-----
true
-----4-----
false
-----5-----
true
-----6-----
in foo
true
-----7-----
in foo
in fun
true
-----8-----
in fun
in foo
false
-----9-----
in foo
false
-----10----
in foo
in fun
false
-----11-----
in fun
in foo
true
-----12-----
true
-----13-----
in foo
in foo
false
```

4. **What are the advantages about short-circuit evaluation?**
5. **What are the potential problems about short-circuit evaluation?**

I am going to answer these two questions in the same place and in same paragraph without splitting respecting to their language as they are related.

Advantages of short-circuit evaluation;

Main advantage is, less code to read for program. The program will be executed faster since it won't read dead codes which is not able to change the result.

```
      function pow(n){

    var tmp = " ";
    for(let i = 0; i < n; i++ ){
        for(let j = 0; j < n; j++){
            tmp += j;
        }
    }
    return tmp;
}
//shortcircuit advantages.
console.log(false && pow(10));
console.log(false || pow(10));
//when we say false && pow(10). Program will get rid of lots of
//unnecessary operation thanks to shortcircuit evaluation.
```

Another benefit of shortcircuit is that it helps program to avoid runtime errors.

```
//this function gives runtime error
function error(p){
    if(p > 0){
        p++;
        error(p);
    }
}
console.log(true || error(2)); //won't cause error
console.log(false || error(2)); //cause error
```

First one is not problematic thanks to shortcircuit evaluation. After finding true on the left side of the logical or, program never goes into the error function.

Possible Problem of shortcircuit:

Code execution becomes less efficient with short-circuited execution paths because in some compilers the new checks for short-circuits are extra execution cycles in themselves. Another possible problem is that, we may avoid to go into some

function but maybe our program needs those function to work correctly. In those cases avoiding is only cause more problem.

```
def inc(x):
    global tmp
    tmp = tmp + 1
    return

tmp = 15
print(tmp)
inc(tmp)
print(tmp)
print(True and inc(tmp))
print(tmp)
print(True or inc(tmp))
print(tmp)

//output
15
16
None
17
True
17
```

In True or inc(tmp) line, we never visit the inc function. Thus, the program never changes the value of tmp. This may also consider as an advantage as we are avoiding lines that won't affect the result of this line. But what if we need the value of tmp after changing by inc function. This may be problematic for the rest of the program execution.

My learning strategy

I haven't heard before what shortcircuit means. Thus, I started to search shortcircuit and the concept. After I understood what it means, I tried it in javascript as I had never realized this property of languages before. I also try on python since I am familiar because of our quiz in the lecture. After I learned about shortcircuit evaluation basically with logical or and logical and operators, I started to search language by language from the internet. The first thing that I did was for example for javascript, does javascript has to shortcircuit? After I searched like this, the first web page that come in front of me was Stackoverflow. For every language in our homework, there was already opened discussion on StackOverflow. I started to read those discussions. But most of them are confusing and not helpful. The only thing that I learned from those discussions is whether the language supports the shortcircuit or not. For further detail, I checked the language's official web pages. I checked every language's webpage and started to answer questions. This time unlike homework 1, I didn't prepare my full code before answering the question. As I thought it was a time waste. Trying to write code when answering questions is more convenient. For javascript and python languages I had no difficulty finding a source or finding a proper syntax for code. The other three were a little more difficult, but their official webpage is too helpful. Every detail is explained on their web page. Thus, there is no need for any further sources. Understanding the different properties of the PHP language took some time. There are "and" and "or" which are different than && and || in PHP. One problem is also in PHP, I couldn't find an online compiler so I write my code on vscode then run it directly in Dijkstra. I wrote my python code to pycharm. Pycharm is so useful for a beginner like me. It automatically corrects and shows the possible errors. I used vscode for javascript. Writing javascript in vscode is also convenient as running or debugging the program is so easy. For rust and dart, I used online compilers. I hadn't gotten a problem on dart's online compiler but in rust, I always get a syntax error. It also happened because there wasn't any source on the internet to show the right syntax of rust. For example, writing a function took me one hour. I tried 50 times till I run it correctly. After I wrote the necessary codes for questions. I tried to add some detail to the answers since I skipped some important detail. After I finished my work and check it, I realized that I misunderstood the first question. I didn't give any detail about the boolean operators provided by the language in the first question. I changed my answer to the first question by searching languages again. The hardest part of the homework is searching the rust and its syntax. Rust's official webpage has great information but it doesn't give many syntax examples on language. Thus, it was a problem to find the right syntax.

References

https://www.geeksforgeeks.org/boolean-data-type-in-python/
https://www.rust-lang.org/
https://dart.dev/guides/language/effective-dart/usage
https://stackoverflow.com/
https://www.php.net/manual/en/language.operators.logical.php