CS315 HW1 REPORT

Funda Tan

21801861

Section 2

Boolean Expressions in C, Go, Javascript, PHP, Python, and Rust

C

Boolean operators provided

Explanation:

There are three boolean operators provided in C. These are:

&& -> and operator

```
bool value1 = false;
 bool value2 = true;
bool booleanExpressionValue = false;
booleanExpressionValue = value1 && value2;
printf("The result of boolean expression is: %d\n", booleanExpressionValue);
```

Output:

```
The result of boolean expression is: 1
```

In this example, we see the && (and) operator. && operator takes value1 and value2 as its operands and calculates the booleanExpressionValue.

|| -> or operator

```
booleanExpressionValue = value1 || value2;
printf("The result of boolean expression is: %d\n", booleanExpressionValue);
```

Output:

```
The result of boolean expression is: 0
```

In this example, we see the || (or) operator. || operator takes value1 and value2 as its operands and calculates the booleanExpressionValue.

! -> not operator

```
booleanExpressionValue = !value1;
printf("The result of boolean expression is: %d\n", booleanExpressionValue);
```

Output:

The result of boolean expression is: 1

In this example, we see the ! (not) operator. ! operator takes value1 and value2 as its operands and calculates the booleanExpressionValue.

Data types for operands of boolean operators

Explanation:

1. Data types for operands of boolean operators can be boolean variables.

```
bool value1 = false;
bool value2 = true;

bool booleanExpressionValue = false;

//NOT operator
booleanExpressionValue = !value1;
printf("The result of boolean expression is: %d\n", booleanExpressionValue);

//AND operator
booleanExpressionValue = value1 && value2;
printf("The result of boolean expression is: %d\n", (int)booleanExpressionValue);

//OR operator
booleanExpressionValue = value1 || value2;
printf("The result of boolean expression is: %d\n", booleanExpressionValue);
```

Output:

The result of boolean expression is: 1

The result of boolean expression is: 1

The result of boolean expression is: 0

We see that boolean operands can take boolean variables as their operands. These operands are value1 and value2 which are boolean variables.

2. Data types for operands of boolean operators can be integers

```
booleanExpressionValue = 5 && 2;
printf("The result of boolean expression is: %d\n", booleanExpressionValue);
```

Output:

The result of boolean expression is: 1

We see that boolean operands can take integers as their operands.

Operator precedence rules

Explanation:

In C, precedence order is the following:

1. && operator
2. || operator
3. ! Operator

```
//AND operator have precedence over OR operator
bool myBool1 = true;
bool myBool2 = false;
bool myBool3 = false;
booleanExpressionValue = myBool1 || myBool2 && myBool3;
```

```
    printf("The result of boolean expression is: %d\n", booleanExpressionValue);

    booleanExpressionValue = myBool1 || (myBool2 && myBool3);
    printf("The result of boolean expression is: %d\n", booleanExpressionValue);

    booleanExpressionValue = (myBool1 || myBool2) && myBool3;
    printf("The result of boolean expression is: %d\n", booleanExpressionValue);
```

Output:

```
The result of boolean expression is: 1
The result of boolean expression is: 1
The result of boolean expression is: 0
```

Result of the first boolean expression in the above code sample is “true” like in the second boolean expression of the code sample so we say that && has precedence over ||. If || had precedence over &&, the result must be like the third boolean expression.

```
    //NOT operator have precedence over AND operator
    booleanExpressionValue = !myBool1 && myBool2;
    printf("The result of boolean expression is: %d\n", booleanExpressionValue);

    booleanExpressionValue = (!myBool1) && myBool2;
    printf("The result of boolean expression is: %d\n", booleanExpressionValue);

    booleanExpressionValue = !(myBool1 && myBool2);
    printf("The result of boolean expression is: %d\n", booleanExpressionValue);
```

Output:

The result of boolean expression is: 0

The result of boolean expression is: 0

The result of boolean expression is: 1

Result of the first boolean expression in the above code sample is “false” like in the second boolean expression of the code sample so we say that ! has precedence over &&. If && had precedence over ||, the result must be like the third boolean expression.

```
//NOT operator have precedence over OR operator
booleanExpressionValue = !myBool1 || myBool1;
printf("The result of boolean expression is: %d\n", booleanExpressionValue);

booleanExpressionValue = (!myBool1) || myBool1;
printf("The result of boolean expression is: %d\n", booleanExpressionValue);

booleanExpressionValue = !(myBool1 || myBool1);
printf("The result of boolean expression is: %d\n", booleanExpressionValue);
```

Output:

The result of boolean expression is: 1

The result of boolean expression is: 1

The result of boolean expression is: 0

Result of the first boolean expression in the above code sample is “true” like in the second boolean expression of the code sample so we say that ! has precedence over ||. If || had precedence over !, the result must be like the third boolean expression.

Operator associativity rules and Operand evaluation order

Explanation:

&& and || operators have left to right associativity.

&& and || operators are evaluated left to right.

```
booleanExpressionValue = a() || b() || c();
printf("\nThe result of boolean expression is: %d\n", booleanExpressionValue);
booleanExpressionValue = !a() && b() && c();
printf("\nThe result of boolean expression is: %d\n", booleanExpressionValue);
```

Output:

```
a
The result of boolean expression is: 1
a
The result of boolean expression is: 0
```

&& and || operators have left to right associativity because a is evaluated only, if it was right to left associativity, c() would have been evaluated first.

&& and || operators are evaluated left to right since function a is evaluated first.

Short-circuit evaluation

Explanation:

There is short-circuit evalutaion in C.

```
printf("Short-circuit evalution examples\n");
booleanExpressionValue = a() || b() || c();
printf("\n");
booleanExpressionValue = !a() && b() && c();
```

Output:

a

a

Only a() is evaluated and the other ones are disregarded because of the short-circuit evaluation.

GO

Boolean operators provided

Explanation:

There are three boolean operators provided in Go. These are:

&& -> and operator

```
  var a bool = true
  var b bool = true
  var c bool = false

  fmt.Print(a && b)
  fmt.Printf("\n")
```

|| -> or operator

```
  fmt.Print(a || b)
  fmt.Printf("\n")
```

! -> not operator

```
  fmt.Print(!a)
  fmt.Printf("\n")
```

Output:

true

true

false

Data types for operands of boolean operators

Explanation:

From the above code sample, we see that boolean operands can take boolean variables as their operands.

These operands are a and b which are boolean variables.

Operator precedence rules

In go, precedence order is the following:

1. && operator
2. || operator
3. ! Operator

Explanation:

```
//AND has precedence over OR
var result bool = a || b && c
fmt.Print(result)
fmt.Printf("\n")

result = (a || b) && c
fmt.Print(result)
fmt.Printf("\n")

result = a || (b && c)
fmt.Print(result)
fmt.Printf("\n")
```

Output:

true

false

true

Result of the first boolean expression in the above code sample is “true” like in the third boolean expression of the code sample so we say that && has precedence over ||. If || had precedence over &&, the result must be like the third boolean expression.

```
//NOT have precedence over OR
  var result2 bool = !a || b
  fmt.Print(result2)
  fmt.Printf("\n")

  result2 = !(a || b)
  fmt.Print(result2)
  fmt.Printf("\n")

  result2 = (!a) || b
  fmt.Print(result2)
  fmt.Printf("\n")
```

Output:

true

false

true

Result of the first boolean expression in the above code sample is “true” like in the third boolean expression of the code sample so we say that ! has precedence over ||. If || had precedence over !, the result must be like the third boolean expression.

```
  //NOT have precedence over AND
  result2 = !a && c
  fmt.Print(result2)
  fmt.Printf("\n")

  result2 = !(a && c)
  fmt.Print(result2)
  fmt.Printf("\n")

  result2 = (!a) && c
  fmt.Print(result2)
  fmt.Printf("\n")
```

Output:

false

true

false

Result of the first boolean expression in the above code sample is “false” like in the third boolean expression of the code sample so we say that ! has precedence over &&. If && had precedence over !, the result must be like the third boolean expression.

Operator associativity rules and Operand evaluation order

Explanation:

&& and || operators has left to right associativity. ! operator has right to left associativity.

Operand evaluation order is from left to right for && and || operators.

```
func func1() bool{
  fmt.Print("a ");
  return true;
}

func func2() bool{
  fmt.Print("b ");
  return true;
}

func func3() bool{
  fmt.Print("c ");
  return true;
}

result2 = func1() && func2() && func3()
//func1's output is a, func2's output is b, func3's output is c
fmt.Print(result2)
fmt.Printf("\n")


result2 = !func1() || !func2() || func3()
//func1's output is a, func2's output is b, func3's output is c
fmt.Print(result2)
fmt.Printf("\n")
```

Output:

a b c true

a b c true

&& and || operators has left to right associativity because leftmost function is evaluated first.

Operands are evaluated left to right because they evaluated func1 and stopped. Operand evaluation order is from left to right for && and || operators.

```
a = false
b = false

result2 = !a && b
fmt.Print(result2)
fmt.Printf("\n")

result2 = !(a && b)
fmt.Print(result2)
fmt.Printf("\n")

result2 = (!a) && b
fmt.Print(result2)
fmt.Printf("\n")
```

Output:

false

true

false

Result of !a && b is like (!a) && b so it ! operator is right associative.

Short-circuit evaluation

Explanation:

```
result2 = !func1() && func2() && func3()
//func1's return value is false
fmt.Print(result2)
fmt.Printf("\n")
```

Output:

a false

Only func1 is evaluated because its return value is false, other ones are not executed because of short-circuit.

```
result2 = func1() || func2() || func3()
//func1's return value is true
fmt.Print(result2)
fmt.Printf("\n")
```

Output:

a true

Only func1 is evaluated because its return value is true, other ones are not executed because of short-circuit.

Javascript

Boolean operators provided

Explanation:

There are three boolean operators provided in Javascript. These are:

&& -> and operator

```
var a = true;
var b = true;
var c = false;
alert(a && b)
```

|| -> or operator

```
alert(a || b)
```

! -> not operator

```
alert(!a)
```

Output:

true

true

false

Data types for operands of boolean operators

Explanation:

From the above code sample, we see that boolean operands can take boolean variables as their operands.

These operands are a and b which are boolean variables.

Operator precedence rules

Explanation:

Precedence order in Javascript is following:

1. !
2. &&
3. ||

```
var result = a || b && c
alert(result)
result = (a || b) && c
alert(result)
result = a || (b && c)
alert(result)
```

Output:

```
true
false
true
```

In this code sample we see that && have precedence over || because the output of the first boolean expression is the same as the third boolean expression.

```
var result2 = !a || b
alert(result2)
result2 = !(a || b)
alert(result2)
result2 = (!a) || b
alert(result2)
```

Output:

```
true
```

false

true

In this code sample we see that ! have precedence over || because the output of the first boolean expression is the same as the third boolean expression.

```
var result2 = !a && c
alert(result2)
result2 = !(a && c)
alert(result2)
result2 = (!a) && c
alert(result2)
```

Output:

false

true

false

In this code sample we see that ! have precedence over && because the output of the first boolean expression is the same as the third boolean expression.

Operator associativity rules and Operand evaluation order

Explanation: In Javascript, operator associativity and operand evaluation order is from left to right.

```
unction func1(){
  alert("func1");
  return true;
}

function func2(){
  alert("func2");
```

```
  return true;
}

function func3(){
  alert("func3");
  return true;
}

result2 = func1() && func2() && func3();

//func1's output is a, func2's output is b, func3's output is c

alert(result2);

result2 = !func1() || !func2() || func3();

//func1's output is a, func2's output is b, func3's output is c

alert(result2);
```

Output:

a b c

a b c

OR and AND operators has left to right associativity because leftmost function is evaluated first

AND and OR operands are evaluated left to right because they evaluated func1 and stopped. Operand evaluation order is from left to right for AND and OR operators.

```
a = false;
```

```
b = false;

result2 = !a && b;
alert(result2);

result2 = !(a && b);
alert(result2);

result2 = (!a) && b;
alert(result2);
```

Output:

false

true

false

Result of !a && b is like (!a) && b so ! operator is right associative.

Short-circuit evaluation

Explanation:

There is short-circuit mechanism in Javascript.

```
result2 = !func1() && func2() && func3();
//func1's return value is false
alert(result2);
```

Output:

a

Only func1 is evaluated, other ones are not executed because of short-circuit.

```
result2 = func1() || func2() || func3();
//func1's return value is true

alert(result2);
```

Output:

a

Only func1 is evaluated, other ones are not executed because of short-circuit.

PHP

Boolean operators provided

Explanation:

There are six boolean operators provided in PHP. These are:

&&, ||, !, and, xor, or

```
$a = true;
$b = true;
$c = false;

$result = $a && $b;
var_dump($result);

$result = $a || $b;
var_dump($result);

$result = $a xor $b;
var_dump($result);
```

```
$result = !$a;
var_dump($result);

$result = $a and $b;
var_dump($result);

$result = $a or $b;
var_dump($result);
```

Output:

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

Data types for operands of boolean operators

Explanation:

From the above code sample, we see that boolean operands can take boolean variables as their operands.

These operands are a and b which are boolean variables.

Operator precedence rules

Explanation:

Operator precedence order in PHP is the following:

1. !
2. &&
3. ||
4. and
5. xor

6. or

```
//&& and ||
$result = ($a || $b) && $c;
var_dump($result);
$result = $a || ($b && $c);
var_dump($result);
$result = $a || $b && $c;
var_dump($result);
```

Output:

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

From the above code segment, we see that && has precedence over ||.

```
//&& and xor
$result = ($a xor $b) && $c;
var_dump($result);
$result = $a xor ($b && $c);
var_dump($result);
$result = $a xor $b && $c;
var_dump($result);
```

Output:

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

From the above code segment, we see that && has precedence over xor

```
//&& and and
```

```
$result = ($a and $b) && $c;
var_dump($result);
$result = $a and ($b && $c);
var_dump($result);
$result = $a and $b && $c;
var_dump($result);
```

Output:

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

From the above code segment, we see that && has precedence over and

```
//&& and or
$result = ($a or $b) && $c;
var_dump($result);
$result = $a or ($b && $c);
var_dump($result);
$result = $a or $b && $c;
var_dump($result);
```

Output:

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

From the above code segment, we see that && has precedence over or

```
//&& and !
$result = (!$a) && $c;
var_dump($result);
$result = !($a && $c);
var_dump($result);
```

```
$result = !$a && $c;
var_dump($result);
```

Output:

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

From the above code segment, we see that ! has precedence over &&

```
//|| and xor
$result = ($a xor $b) || $c;
var_dump($result);
$result = $a xor ($b || $c);
var_dump($result);
$result = $a xor $b || $c;
var_dump($result);
```

Output:

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

From the above code segment, we see that || has precedence over xor

```
//|| and or
$result = ($c or $c) || $b;
var_dump($result);
$result = $c or ($c || $b);
var_dump($result);
$result = $c or $c || $b;
var_dump($result);
```

Output:

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

From the above code segment, we see that || has precedence over or

```
//|| and and
$result = ($c and $c) || $b;
var_dump($result);
$result = $c and ($c || $b);
var_dump($result);
$result = $c and $c || $b;
var_dump($result);
```

Output:

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

From the above code segment, we see that || has precedence over and

```
//and and xor
$result = ($a xor $b) and $b;
var_dump($result);
$result = $a xor ($b and $b);
var_dump($result);
$result = $a xor $b and $b;
var_dump($result);
```

Output:

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

From the above code segment, we see that and has precedence over xor

```
//and and or
$result = ($c or $b) and $c;
var_dump($result);
$result = $c or ($b and $c);
var_dump($result);
$result = $c or $b and $c;
var_dump($result);
```

Output:

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

From the above code segment, we see that and has precedence over or

```
//or and xor
$result = ($c xor $b) or $c;
var_dump($result);
$result = $c xor ($b or $c);
var_dump($result);
$result = $c xor $b or $c;
var_dump($result);
```

Output:

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

From the above code segment, we see that xor has precedence over or

Operator associativity rules and Operand evaluation order

Explanation:

&&, ||, and, or, xor operators has left to right associativity. ! operator is right to left associative.

&&, ||, and, or, xor operands are evaluated left to right.

```
function func1(){
    echo "func1 ";
    return true;
}

function func2(){
    echo "func2 ";
    return true;
}

function func3(){
    echo "func3 ";
    return true;
}

$result = func1() && func2() && func3();
echo "\n";
var_dump($result);

$result = !func1() || !func2() || func3();
echo "\n";
var_dump($result);

$result = func1() and func2() and func3();
echo "\n";
var_dump($result);

$result = !func1() or !func2() or func3();
```

```
echo "\n";
var_dump($result);

$result = func1() xor func2() xor func3();
echo "\n";
var_dump($result);
```

Output:

```
func1 func2 func3 bool(true)
func1 func2 func3  bool(true)
func1 func2 func3 bool(true)
func1 func2 func3  bool(false)
func1 func2 func3 bool(true)
```

&&, ||, and, or, xor operators has left to right associativity because leftmost function is evaluated first

&&, ||, and, or, xor operands are evaluated left to right because they evaluated func1 and stopped.

Operand evaluation order is from left to right for AND and OR operators.

```
$a = false;
$b = false;

$result = !$a && $b;
var_dump($result);

$result = !($a && $b);
var_dump($result);

$result = (!$a) && $b;
```

var_dump($result);

Output:

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

Result of !a && b is like (!a) && b so ! operator is right associative

Short-circuit evaluation

Explanation:

PHP has short-circuit mechanism for &&, ||, and, or operators.

```
$result = !func1() && func2() && func3();
echo "\n";

$result = func1() || func2() || func3();
echo "\n";

$result = !func1() and func2() and func3();
echo "\n";

$result = func1() or func2() or func3();
echo "\n";
```

Output:

```
func1
func1
func1
func1

&&, ||, and, or operators are short-circuit because only func1 is called,
func2 and func3 is disregarded
```

Python

Boolean operators provided

Explanation:

There are three boolean operators provided in Python. These are:

and operator

```
bool1 = True
bool2 = True
bool3 = False

print("Result is: ", bool1 and bool2)
```

or operator

```
print("Result is: ",bool1 or bool2)
```

not operator

```
print("Result is: ", not bool1)
```

Output:

```
Result is:  True
Result is:  True
Result is:  False
```

Data types for operands of boolean operators

Explanation:

From the above code sample, we see that boolean operands can take boolean variables as their operands. These operands are bool1 and bool2 which are boolean variables.

Operator precedence rules

Explanation:

Operator precedence order in Python is the following:

1. not
2. and
3. or

```
result = bool1 or bool2 and bool3
print("Result is: ", result)

result = (bool1 or bool2) and bool3
print("Result is: ", result)

result = bool1 or (bool2 and bool3)
print("Result is: ", result)
```

Output:

Result is: True

Result is: False

Result is: True

and has precedence over or since the first boolean expression's value is same with the third boolean expressions value.

```
result = not bool1 or bool2
print("Result is: ", result)

result = (not bool1) or bool2
print("Result is: ", result)

result = not (bool1 or bool2)
print("Result is: ", result)
```

Output:

Result is: True

Result is: True

Result is: False

not has precedence over or since the first boolean expression’s value is same with the second boolean expressions value.

```
result = not bool1 and bool3

print("Result is: ", result)


result = (not bool1) and bool3

print("Result is: ", result)


result = not (bool1 and bool3)

print("Result is: ", result)
```

Output:

Result is: False

Result is: False

Result is: True

not has precedence over and since the first boolean expression’s value is same with the third boolean expressions value.

Operator associativity rules and Operand evaluation order

Explanation:

and, or are left associative and operand evaluation order is from left to right.

```
def func1():
    print("func1");
    return True;
```

```
def func2():
    print("func2");
    return True;

def func3():
    print("func3");
    return True;

def funcFalse():
    print("func4");
    return False;

result = func1() and func2() and func3()
print("Result is: ", result)

result = func1() or func2() or func3()
print("Result is: ", result)
```

Output:

```
func1
func2
func3
Result is:  True
func1
Result is:  True
```

and, or they are left associative and operand evaluation order is from left to right since firstly, func1 is evaluated.

```
bool1 = False
bool2 = False
result = not bool1 and bool2
print("Result is: ", result)
result = not (bool1 and bool2)
print("Result is: ", result)
result = (not bool1) and bool2
print("Result is: ", result)
# result of not bool1 and bool2 is the same as (not bool1) and bool2 so not operator is right associative
```

Short-circuit evaluation

Explanation:

In Pyhton there is short-circuit evaluation for and, or operators.

```
result  = funcFalse() and func1() and func2()
```

Output:

funcFalse

Only funcFalse is evaluated, other ones are not executed because of short-circuit.

```
result = func1() or func2() or func3()
```

Output:

func1

Only func1 is evaluated, other ones are not executed because of short-circuit.

Rust

Boolean operators provided

Explanation:

There are three boolean operators provided in Rust. These are &&, || and ! Operators.

```
let mut bool1 = true;

let mut bool2 = true;

let bool3 = false;


println!("Result is: {}", bool1 && bool2);

println!("Result is: {}", bool1 || bool2);

println!("Result is: {}", !bool1);
```

Output:

```
Result is: true

Result is: true

Result is: false
```

Data types for operands of boolean operators

Explanation:

From the above code sample, we see that boolean operands can take boolean variables as their operands. These operands are bool1 and bool2 which are boolean variables.

Operator precedence rules

Explanation:

Precedence order in Rust is the following:

1. !
2. &&
3. ||

```
let mut result = (bool1 || bool2) && bool3;

println!("Result is: {}", result);


result = bool1 || (bool2 && bool3);

println!("Result is: {}", result);
```

```
    result = bool1 || bool2 && bool3;
    println!("Result is: {}", result);
```

Output:

```
Result is: false
Result is: true
Result is: true
```

and has precedence over or

```
    result = (!bool1) || bool2;
    println!("Result is: {}", result);

    result = !(bool1 || bool2);
    println!("Result is: {}", result);

    result = !bool1 || bool2;
    println!("Result is: {}", result);
```

Output:

```
Result is: true
Result is: false
Result is: true
```

not has precedence over or

```
    result = (!bool1) && bool3;
    println!("Result is: {}", result);

    result = !(bool1 && bool3);
    println!("Result is: {}", result);
```

```
    result = !bool1 && bool3;
    println!("Result is: {}", result);
```

Output:

```
Result is: false
Result is: true
Result is: false
```

not has precedence over and

Operator associativity rules and Operand evaluation order

Explanation:

Operator associativity rule for &&, || is from left to right. Operand evaluation order for and, or is from left to right.

Operator associativity rule for ! is from right to left.

```
fn func1() -> bool{
        println!("func1");
        return true;
    }

    fn func2() -> bool{
        println!("func2");
        return true;
    }

    fn func3() -> bool{
        println!("func3");
        return true;
    }
```

```
    result = func1() && func2() && func3();
    println!("Result is: {}", result);

    result = func1() || func2() || func3();
    println!("Result is: {}", result);
```

Output:

```
func1func2func3
Result is: true
func1
Result is: true
```

and, or they are left associative and operand evaluation order is from left to right.

```
    bool1 = false;
    bool2 = false;
    result = !bool1 && bool2;
    println!("Result is: {}", result);
    result = !(bool1 && bool2);
    println!("Result is: {}", result);
    result = (!bool1) && bool2;
    println!("Result is: {}", result);
```

Output:

```
Result is: false
Result is: true
Result is: false
```

Result of not bool1 and bool2 is the same as (not bool1) and bool2 so not operator is right associative

Short-circuit evaluation

Explanation:

There is short-circuit evaluation in Rust for &&, || operators.

```
result = !func1() && func2() && func3();
println!("Result is: {}", result);
```

Output:

```
func1
Result is: false
```

Only func1 is evaluated, other ones are not executed because of short-circuit.

```
result = func1() || func2() || func3();
println!("Result is: {}", result);
```

Output:

```
func1
Result is: true
```

Only func1 is evaluated, other ones are not executed because of short-circuit.

Evaluation of Languages

In terms of readability, Python is the most readable language in terms of boolean operators because it uses words instead of symbols for and, or and not operators. The other languages uses symbols for these operators. Python has the most readable boolean expressions, but its writability is less than PHP since PHP has six different boolean operators.

In terms of writability, PHP is the most writable language in terms of boolean operators because it has six different boolean operators. The other languages have three different boolean operators only.

Because of that, the most writable language is PHP, when boolean operators are considered. However, this decreases the readability of PHP since there are six boolean operators. Moreover, PHP has xor operator that other languages do not have xor operator and this increases its writability compared to C, Go, Javascript, Python and Rust.

In my opinion, PHP is the best language in terms of boolean expressions because it has the variety in terms of boolean operands. C, Go, Javascript and Rust is very similar in terms of readability and writability. Python is more readable, but it has not a variety of boolean operands like PHP. Also, C, Go, Javascript, Python and Rust does not have xor operator but PHP has the xor operator. Because of these reasons, I think that PHP is the best language in terms of boolean expressions.

Learning Strategy

I firstly looked at the documentations for each language. Documentations was my biggest material to learn the languages and find the boolean operators they provide. Then, I found online compilers for each language and practice their syntax. When I was able to write codes in the languages, I wrote codes to investigate their design practices in terms of boolean expressions. I completed my code samples for all of the languages and then I looked each of them comparing to other ones and find the differences and similarities. With these differences and similarities, I was able to discuss these six languages in terms of readability and writability. I also used what I learnt from the lectures and applied them when explaining the design choices of these languages.

The online compilers that I used for this assignment:

C: https://www.onlinegdb.com/online_c_compiler

Go: https://www.tutorialspoint.com/execute_golang_online.php

Javascript: https://www.tutorialspoint.com/online_javascript_editor.php

PHP: https://www.tutorialspoint.com/execute_php_online.php

Python: https://www.programiz.com/python-programming/online-compiler/

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