İlke Kaş

21803184

Section 1

REPORT

C

1. Boolean operators provided

- **Explanation:** There are three Boolean operators provided by C languages. These are:
  - **&& (Logical AND operator):** Result of the binary && (AND) operator will be true when both operands are true, otherwise will be false. It is infix operator.
  - **|| (Logical OR operator):** Result of the binary || (OR) operator will be false when both operands are false, otherwise will be true. It is infix operator.
  - **! (Logical NOT operator):** Unary !(NOT) operator will reverse the logical state of its operands. If its operand is false, it makes its operand true. If its operand is true, it makes its operand false.
- **Code Segment for && (Logical AND operator)**:

```
int T= 1;
int F= 0;

printf("All combination of &&(AND) operator\n");
if(F&&F)
{
   printf("F&&F is true\n");
}
 else
{
   printf("F&&F is false\n");
}
  if(F&&T)
{
   printf("F&&T is true\n");
}
 else
{
   printf("F&&T is false\n");
}
if(T&&F)
{
   printf("T&&F is true\n");
}
```

```
else
{
   printf("T&&F is false\n");
}
if(T&&T)
{
   printf("T&&T is true\n");
}
 else
{
   printf("T&&T is false\n");
}
```

- **Result of Execution:**
  All combination of &&(AND) operator
  F&&F is false
  F&&T is false
  T&&F is false
  T&&T is true

- **Explanation of the code segment**: This code segment will display the truth table of the &&(AND) operator in C language. We see that it has the same result as the logic. && operator gives true only both of the operands are true. Otherwise, it gives false.

- **Code Segment for ||(Logical OR operator)**:

```
   T= 1;
   F= 0;
printf("All combination of ||(OR) operator\n");
   if(F||F)
   {
      printf("F||F is true\n");
   }
    else
   {
      printf("F||F is false\n");
   }
     if(F||T)
   {
      printf("F||T is true\n");
   }
    else
   {
      printf("F||T is false\n");
```

```
}
if(T||F)
{
   printf("T||F is true\n");
}
else
{
   printf("T||F is false\n");
}
if(T||T)
{
   printf("T||T is true\n");
}
 else
{
   printf("T||T is false\n");
}
```

- **Result of Execution:**
  All combination of ||(OR) operator
  F||F is false
  F||T is true
  T||F is true
  T||T is true
- **Explanation of the code segment**: This code segment will display the truth table of the ||(OR) operator in C language. We see that it has the same result as the logic. || operator gives false only both operands are false. Otherwise, it gives true.

- **Code Segment for ! (Logical NOT operator)**:

```
  T = 1;
  F = 0;
printf("All combination of !(NOT) operator\n");
if(T)
{
   printf("T is true\n");
}
else
{
   printf("T is false\n");
}
 if(!T)
{
   printf("!T is true\n");
}
else
```

```
{
   printf("!T is false\n");
}
 if(F)
{
   printf("F is true\n");
}
else
{
   printf("F is false\n");
}
 if(!F)
{
   printf("!F is true\n");
}
else
{
   printf("!F is false\n");
}
```

- **Result of Execution:**
  All combinations of ! (NOT) operator
  T is true
  !T is false
  F is false
  !F is true
- **Explanation of the code segment:** This code segment will display the truth table of the !(NOT) operator in C language. We see that it has the same result as the logic. ! operator reverse the logic condition of its operand. If its operand is T(true), it makes its operand false. If its operand is F(false), it makes its operand true.

2. Data types for operands of Boolean operators

- **Explanation:** C does not have Boolean type and no Boolean values. Instead, to represent Boolean values, C uses numeric values. Therefore, any numeric values, characters or constants except 0 means true in C language. In other words, all nonzero values considered as true, and zero values are considered as false.
  Arithmetic expressions are used as operands in Boolean expressions as operands since they have numeric values. Besides these, since the result of relational expressions are logical states, we can also use relational expressions as the operands of Boolean operators in C. Finally, function calls also can be used as operands in C language.

- **Code Segment 1:**

```
int var1 = 0;
int var2 = 8;
float var3 = 7.89;
int var4 = -8;
char c = 'c';
if(var1)
{
   printf("%d is true\n",var1);
}
else
{
   printf("%d is false\n",var1);
}
 if(var2)
{
   printf("%d is true\n",var2);
}
else
{
   printf("%d is false\n",var2);
}
 if(var3)
{
   printf("%f is true\n",var3);
}
else
{
   printf("%f is false\n",var3);
}
if(c)
{
   printf("%c is true\n",c);
}
else
{
   printf("%c is false\n",c);
}
 if(var4)
{
   printf("%d is true\n",var4);
}
else
{
```

```
    printf("%d is false\n",var4);
  }
  if(var2+var4)
  {
    printf("%d + (%d) is true\n",var2,var4);
  }
  else
  {
    printf("%d + (%d) is false\n",var2,var4);
  }
```

- **Result of Execution:**

```
0 is false
8 is true
7.890000 is true
c is true
-8 is true
8 + (-8) is false
```

- **Explanation of the code segment**: 0 is considered as false in C language. We understood this from the result of the if statement. If 0 was considered as true, "if" statement would print "0 is true" but it performs the opposite in reality. Scalar variables other than 0 such as 8, 7.89, -8 and character c are considered as true in C language.
  Besides these, arithmetic expressions can be considered as Boolean operand in C language. For example, in the code segment, arithmetic expression whose result is 0 is again considered as false in this language.
- **Code Segment 2:**

```
  var2 = 5;
  int var5= 10;
  var4 = 12;
 if( !(var2 + var5))
 {
   printf("!(%d + %d) is true\n",var2,var5);
 }
 else
 {
    printf("!(%d + %d) is false\n",var2,var5);
 }
 if((var2 < var5) || (var2 > var4) )
 {
   printf("Relational as operand\n");
 }
 int rel_operand = (var2 < var5);
 int rel_operand2 = (var2 > var4);
 printf("var2 < var5 is %d\n",rel_operand);
```

```
printf("var2 > var4 is %d\n",rel_operand2);
```

- **Result of Execution:**
  !(5+10) is false
  Relational as operand
  var2<var5 is 1
  var2>var4 is 0
  Explanation of the code segment: As stated previously arithmetic expressions can be used as operand for Boolean operators. Output of "!(5+10) is false" is a proof of this. Besides, relational expressions in C languages have logical states and when printed in the integer format, true values correspond to 1; meanwhile false ones correspond to 0. This can be confirmed by the following outputs:
  var2<var5 is 1
  var2>var4 is 0
- **Code Segment 3:**

```
int function()
{
   printf("Function is executed\n");
  return 1;
}
int main() {
 int T = 1;
  if(function() && T)
  {
     printf("Function can be used as operand\n");
  }

   return 0;
}
```

- **Result of Execution:**
  Function is executed
  Function can be used as operand
  Explanation of the code segment: Return values of the function calls can be used as operands in C language.

3. **Operator precedence rules**

- **Explanation:** !(NOT) operator has the highest precedence between the logical operators. &&(AND) operator has higher precedence than || (OR) operator. In other words, their precedence ranking is:

! > && > ||

- **Code Segment 1:**

```
if(!0 && 0)
  {
    printf("!0 && 0 is true\n");
  }
  else
  {
    printf("!0 && 0 is false\n");
  }
```

- **Result of Execution:**

  !0 && 0 is false

  Explanation of the code segment: If the && operator had higher precedence than the ! operator, it will be computed as !(0&&0) = !0 = 1, the result would have been true and "!0 && 0 is true" would have been seen in the output. However, this is not the situation. ! has higher precedence than && operator. So, the evaluation order will be (!0) && 0 = 1 && 0 = 0. The result will be false and we can observe it in the output as "!0&&0 is false".

- **Code Segment 2:**

```
if(!1 || 1)
  {
    printf("!1 || 1 is true\n");
  }
  else
  {
    printf("!1 || 1 is false\n");
  }
```

- **Result of Execution:**

  !1 || 1 is true

  Explanation of the code segment: : If the || operator had higher precedence than the ! operator, it will be computed as !(1 || 1) = !1 = 0, the result would have been false and "!1 || 1 is false" would have seen in the output. However, this is not the situation. ! has higher precedence than || operator. So, the evaluation order will be (!1) || 1 = 0 || 1 = 1. The result will be true and we can observe it in the output as "!1 || 1 is true".

- **Code Segment 3:**

```
if(1 || 1 && 0 )
  {
    printf("1 || 1 && 0 is true\n");
  }
  else
  {
    printf("1 || 1 && 0 is false \n");
  }
```

- **Result of Execution:**

1 || 1 && 0 is true

Explanation of the code segment: If the || operator had higher precedence than the && operator, it will be computed as ((1 || 1)&&0) = (1&&0) = 0, the result would have been false and ("1 || 1 && 0 is false" would have seen in the output. However, this is not the situation. && has higher precedence than || operator. So, the evaluation order will be (1 || (1 && 0))= 1 || 0 = 1. The result will be true and we can observe it in the output as "1 || 1 && 0 is true".

4. **Operator associativity rules**

- **Explanation:** &&(AND) and || (OR) operators have left-to-right associativity while !(NOT) operator has right to left associativity.
- **Code Segment:**

```
int function1()
{
   printf("function 1 is evaluated\n");
   return 1;
}
int function2()
{
   printf("function 2 is evaluated\n");
   return 1;
}
int function3()
{
   printf("function 3 is evaluated\n");
   return 1;
}
int function4()
{
   printf("function 4 is evaluated\n");
   return 0;
}
int function5()
{
   printf("function 5 is evaluated\n");
   return 0;
}
int function6()
{
   printf("function 6 is evaluated\n");
   return 0;
}
int main()
```

```
{
 if(function2() && function1() && function3())
  {
   printf("Left to right\n");
  }
  if(function5()||function6()||function4())
  {
   printf("Do not print this\n");
  }
if(!!!function4())
  {
     printf("Right associativity\n");
  }

return 0;
}
```

- **Result of Execution:**

```
function 2 is evaluated
function 1 is evaluated
function 3 is evaluated
Left to right
function 5 is evaluated
function 6 is evaluated
function 4 is evaluated
Right associativity
```

Explanation of the code segment: When "function2() && function1() && function3()" expression in the if statement is executed, as seen from the output first function 2, then function 1 and finally function 3 is evaluated. This means, && operator has left to right associativity. Similarly, when "function5()||function6()||function4()" is evaluated, first function 5, then function 6 and finally function 4 is evaluated. This also means that || operator has left-to right associativity. To see the evaluation of each operand in the statement, I avoid short-circuit values in this code segment by choosing all of the three values as 1 in && operator associativity and 0 in || operator associativity. Finally, !!!function4() is evaluated from right to left as (! (! (!function4()))) = (! (! 1)) = (! 0) = 1. Then "Right associativity" is seen in the output.

5. Operand evaluation order

- **Explanation:**
  - **Function vs variable:**
    In C language both && and || evaluate two function operands, two scalar variables or constants and combination of one function and scalar variable from left to right. There is no effect whether operand is function call or integer variable.

However, sometimes functions can have side effects and affect the result of the logical expression. According to that result, short-circuit may occur and one of the operands may not be evaluated.

- **Parentheses:**
  Parentheses may change the evaluation order of expressions of more than two operands. They give priority to the expression in the parentheses. Therefore, operands in the parentheses are evaluated first.

- **Code Segment:**
  - **Function vs variable:**

```
int global = 0;
int function7()
{
   global++;
   printf("function 7 is evaluated\n");
   return 1;
}
int main()
{
   if(global && function7())
   {
   printf("Side effect of function 7-1");
   }
   global = 0;
   if(function7() && global)
   {
   printf("Side effect of function 7-2");
   }
return 0;
}
```

- **Result of Execution:**
  function 7 is evaluated
  side effect of function 7-2
- **Explanation of the code segment:** The Boolean expression in the first if statement, first global variable is evaluated then function7 will be evaluated as we mentioned before. However since the global variable is 0, it will be short-circuit evaluation and function7 will not be evaluated. In the second boolean expression, first function7 is evaluated. In this situation function7 will change increment the value of the global variable, so it is not 0 anymore. Hence, since the value of global is nonzero, the result of the expression will change and the "printf("Side effect of function 7-2");" statement will be executed as seen in the output. Order of operand evaluation change due to the short-circuit evaluation and the result of the expression will change.
- **Parentheses:**

```
 if(1 && 1 || 0 && 0)
  {
     printf("1 && 1 || 0 && 0 is true \n");
```

```
}
else
 {
   printf("1 && 1 || 0 && 0 is false \n");
}
if(1 && (1 || 0) && 0)
{
   printf("1 && (1 || 0 )&& 0 is true \n");
}
 else
 {
   printf("1 && (1 || 0) && 0 is false \n");
}
```

- **Result of Execution:**
  1 && 1 || 0 && 0 is true
  1 && (1 || 0) && 0 is false
- **Explanation of the code segment:** In the first expression without parentheses, since the || operator has lower precedence, 1 && 1 || 0 && 0 is evaluated as (1 && 1) || (0 && 0) = 1 || 0 = 1. In the second expression, parentheses change the evaluation order and the operands of the or operator will be first evaluated from left to right: 1 && (1 || 0) && 0 = 1 && 1 && 0 = 0.

6. Short-circuit evaluation

- **Explanation:** In C language there is short-circuit evaluation that enables to avoid unnecessary evaluation of operands. If the first operand of && operator is 0, the second operand will not be evaluated. If the first operand of the || operator is 1, the second operand will not be evaluated. However, this can be dangerous if the programs' correctness depends on the side effect of a function.

- **Code Segment:**

```
Int global = 0;
int function7()
{
   global++;
   printf("function 7 is evaluated\n");
   return 1;
}
int main()
{
   if(global && function7())
  {
   printf("Side effect of function 7-1\n");
  }
```

```
  printf("1- global is: %d\n", global);
  global = 0;
  if(function7() && global)
  {
   printf("Side effect of function 7-2\n");
  }
  printf("2- global is: %d\n", global);
    return 0;
}
```

- **Result of Execution:**

```
1-  global is: 0

function 7 is evaluated

Side effect of function7-2

2-  global is: 1
```

Explanation of the code segment: In the first if statement, "if (global && function7()) "since the value of the global variable is 0 (false), AND operator will make short-circuit and do not evaluate function 7. However, If function 7 would be evaluated, it will increment the value of the global variable. If the correctness of the program is dependent on this incrementation of global variable, this makes program wrong. In the second if statement "if(function7() && global)" both of the operands will be evaluated.

My Evaluation about language: In my opinion, since C language do not have boolean data type, it makes logical expressions in this language hard to read. In other words, in terms of boolean expressions, C language is not readable.

GO

1. **Boolean operators provided**
   - **Explanation:** There are three logical operators in GO language. These are:
     - **&& (Logical AND operator):** Result of the binary && (AND) operator will be true when both operands are true, otherwise will be false. It is infix operator.
     - **|| (Logical OR operator):** Result of the binary || (OR) operator will be false when both operands are false, otherwise will be true. It is infix operator.
     - **! (Logical NOT operator):** Unary !(NOT) operator will reverse the logical state of its operands. If its operand is false, it makes its operand true. If its operand is true, it makes its operand false.
   - **Code Segment:**

```
T := true
  F := false
  fmt.Println("All combination of &&(AND) operator\n")
  fmt.Println("F&&F:", F&&F)
  fmt.Println("F&&T:", F&&T)
  fmt.Println("T&&F:", T&&F)
  fmt.Println("T&&T:", T&&T)
  fmt.Println("All combination of ||(OR) operator\n")
  fmt.Println("F&&F:", F||F)
  fmt.Println("F&&T:", F||T)
  fmt.Println("T&&F:", T||F)
  fmt.Println("T&&T:", T||T)
  fmt.Println("All combination of !(NOT) operator\n")
  fmt.Println("F:", F)
  fmt.Println("!F:", !F)
  fmt.Println("T:", T)
  fmt.Println("!T:", !T)
```

- **Result of Execution:**

```
All combination of &&(AND) operator

F&&F: false
F&&T: false
T&&F: false
T&&T: true
All combination of ||(OR) operator

F&&F: false
F&&T: true
T&&F: true
T&&T: true
All combination of !(NOT) operator

F: false
!F: true
T: true
!T: false
```

- **Explanation of the code segment:** First code segment will display the truth table of the &&(AND) operator in GO language. We see that it has the same result as the logic. && operator gives true only both of the operands are true. Otherwise, it is false. Second code segment will display the truth table of the ||(OR) operator in GO language. We see that it has the same result as the logic. || operator gives false only both operands are false. Otherwise, it is true. Last code segment will display the truth table of the !(NOT) operator in GO language. We see that it has the same result as the

logic. ! operator reverse the logic condition of its operand. If its operand is T(true), it makes its operand false. If its operand is F(false), it makes its operand true.

2. **Data types for operands of Boolean operators**
   - **Explanation:** GO has Boolean values true and false as predeclared constants in the language. Non-bool values such as integers, floating point numbers, strings cannot be used as Boolean variables unlike C language. Therefore, arithmetic expressions cannot be used as operand of the Boolean operators in GO language. However, relational expressions can be used as operands also in GO language. Besides these, function calls whose return value is boolean type can also be used as operands.
   - **Code Segment:**

```
func function1()bool{
   fmt.Println("function 1 is evaluated")
   return true
}
func main() {
if(true){
      fmt.Println("print this")
   }
   if(false){
      fmt.Println("do not print this")
   }
   a:= 13
   c:= a % 2 == 1
   fmt.Println("c:", c)
   b := a <= 9;
   fmt.Println("b:", b)
   d:= true && function1()
   fmt.Println("d:", d)
}
```

   - **Result of Execution:**

```
print this
c: true
b: false
function 1 is evaluated
d: true
```

   Explanation of the code segment: There are two constant for Boolean type of variable in GO language: true and false can be used to make decision as in the example in if statement. Besides, as in the initialization of c, we can see that relational expressions have Boolean variables. In, " d:= true && function1()" statement we observe that function calls whose return type is a Boolean type can also be used as an operand of an logical expression. However, when we try to return other types rather than bool types, we get error. GO language does not support this kind of conversions.

3. **Operator precedence rules**

- **Explanation:** Unary operators has the highest precedence in GO language. So, !(NOT) operator has the highest precedence over &&(AND) and ||(OR) operators.&&(AND) operator has higher precedence than ||(OR) operator.
- **Code Segment:**

```
fmt.Println("! false && false: ",!false && false)
fmt.Println("! true || true:", !true || true)
fmt.Println("true || true && false:", true || true && false)
```

Result of Execution:

```
! false && false: false
! true || true:  true
true || true && false: true
```

Explanation of the code segment: If the && operator had higher precedence than the ! operator, it will be computed as !(false&&false) = !false = true, the result would have been true and "! false && false: true" would have seen in the output. However, this is not the situation. ! has higher precedence than && operator. So, the evaluation order will be (!false) && false = true && false = false. The result will be false and we can observe it in the output as "! false && false: false".

If the || operator had higher precedence than the ! operator, it will be computed as !(true || true) = !true = false, the result would have been false and "!true || true: false" would have seen in the output. However, this is not the situation. ! has higher precedence than || operator. So, the evaluation order will be (!true) || true = false || true = true. The result will be true and we can observe it in the output as "!1 || 1: true".

If the || operator had higher precedence than the && operator, it will be computed as ((true || true)&&false) = (true&&false) = false, the result would have been false and "true || true && false: false" would have seen in the output. However, this is not the situation. && has higher precedence than || operator. So, the evaluation order will be (true || (true && false))= true || false = true. The result will be true and we can observe it in the output as "true || true && false: true".

4. **Operator associativity rules**

- **Explanation: :** &&(AND) and || (OR) operators have left-to-right associativity while !(NOT) operator has right to left associativity like in C language.
- **Code Segment:**

```
func function1()bool{
   fmt.Println("function 1 is evaluated")
   return true
}
func function2()bool{
   fmt.Println("function 2 is evaluated")
   return true
}
func function3()bool{
   fmt.Println("function 3 is evaluated")
```

```
    return true
}
func function4()bool{
    fmt.Println("function 4 is evaluated")
    return false
}
func function5()bool{
    fmt.Println("function 5 is evaluated")
    return false
}
func function6()bool{
    fmt.Println("function 6 is evaluated")
    return false
}
func main() {
and:=function2() && function1() && function3()
     fmt.Println("and:",and)
    or:=function5() || function6() || function4()
     fmt.Println("or:",or)
    if !!!function4(){
        fmt.Println("Right associativity")
    }
}
```

- **Result of Execution:**

```
function 2 is evaluated
function 1 is evaluated
 function 3 is evaluated
and: true
function 5 is evaluated
function 6 is evaluated
function 4 is evaluated
or: false
function 4 is evaluated
Right associativity
```

Explanation of the code segment: When "function2() && function1() && function3()" expression in the if statement is executed, as seen from the output first function 2, then function 1 and finally function 3 is evaluated. This means, && operator has left to right associativity. Similarly, when "function5()||function6()||function4()" is evaluated, first function 5, then function 6 and finally function 4 is evaluated. This also means that || operator has left-to right associativity. To see the evaluation of each operand in the statement, I avoid

short-circuit values in this code segment by choosing all of the three values as 1 in && operator associativity and 0 in || operator associativity. Finally, !!!function4() is evaluated from right to left as (! (! (!function4()))) = (! (! true)) = (! false) = true. Then "Right associativity" is seen in the output.

5. **Operand evaluation order**

- **Explanation:**
  - **Function vs variable:**
    In GO language both && and || evaluate two function operands, two boolean variables or constants and combination of one function and scalar variable from left to right. There is no effect whether operand is function call or Boolean variable. However, sometimes functions can have side effects and affect the result of the logical expression. According to that result, short-circuit may occur and one of the operands may not be evaluated.
  - **Parentheses:**
    Parentheses may change the evaluation order of expressions of more than two operands. They give priority to the expression in the parentheses. Therefore, operands in the parentheses are evaluated first.

- **Code Segment1:**
  -Function vs variable:

```
var globalvar bool = false
func function7() bool{
   globalvar = true;
   fmt.Println("function 7 is evaluated");
   return true;
}
func main() {
if(globalvar && function7()){
      fmt.Println("Side effect of function 7-1")
   }
   globalvar = false;
    if(function7()&&globalvar){
      fmt.Println("Side effect of function 7-2")
   }
```

- **Result of Execution:**

```
function 7 is evaluated
Side effect of function 7-2
```

Explanation of the code segment: The Boolean expression in the first if statement, first global variable is evaluated then function7 will be evaluated as we mentioned before. However, since the global variable is false, it will be short-circuit evaluation and function7 will not be evaluated. In the second boolean expression, first function7 is evaluated. In this situation function7 will change the value of the global variable, so it is not false anymore. Hence, since the value of global is true, the result of the

expression will change and the " fmt.Println("Side effect of function 7-2")" statement will be executed as seen in the output. Order of operand evaluation changes due to the short-circuit evaluation and the result of the expression will change.

- **Code Segment2:**
  -Parentheses
  fmt.Println("true && true || false && false: ",true && true || false && false)
  fmt.Println("true && (true || false) && false: ",true && (true || false) && false)
- **Result of Execution:**
  true && true || false && false: true
  true && (true || false) && false: false
- **Explanation of the code segment:** In the first expression without parentheses, since the || operator has lower precedence, true && true || false && false is evaluated as (true && true) || (false && false) = true || false = true. In the second expression, parentheses change the evaluation order and the operands of the or operator will be first evaluated from left to right: true && (true || false) && false = true && true && false = (true && true) && false = true && false = false.

6. Short-circuit evaluation

- **Explanation:** In GO language there is short-circuit evaluation that enables to avoid unnecessary evaluation of operands. If the first operand of && operator is false, the second operand will not be evaluated. If the first operand of the || operator is true, the second operand will not be evaluated. However, this can be dangerous if the programs' correctness depends on the side effect of a function
- **Code Segment:**

```
var globalvar bool = false
func function7() bool{
   globalvar = true;
   fmt.Println("function 7 is evaluated");
   return true;
}
func main() {
if(globalvar && function7()){
      fmt.Println("Side effect of function 7-1")
   }
   fmt.Println("1- global is ", globalvar)
   globalvar = false;
    if(function7()&&globalvar){
      fmt.Println("Side effect of function 7-2")
   }
    fmt.Println("2- global is ", globalvar)
}
```

- **Result of Execution:**
  1- global is false
  function 7 is evaluated

Side effect of function 7-2
2- global is true

- **Explanation of the code segment:** In the first if statement, "if (global && function7()) "since the value of the global variable is false, AND operator will make short-circuit and do not evaluate function 7. However, if function 7 would be evaluated, it will change the value of the global variable. If the correctness of the program is dependent on this change of global variable, this makes the program wrong. In the second if statement "if(function7() && global)" both of the operands will be evaluated.

My Evaluation about language: In my opinion, since GO language has Boolean data type as separate type unlike C language, it is more readable and writable than C language in terms of Boolean expressions. Besides, it fits the general rules of logic. So, it makes it easy to learn.

JavaScript

1. **Boolean operators provided**
   - **Explanation:** There are three logical operators in JavaScript. These are:
     - **&& (Logical AND operator):** Result of the binary && (AND) operator will be true when both operands are true, otherwise will be false. It is infix operator.
     - **|| (Logical OR operator):** Result of the binary || (OR) operator will be false when both operands are false, otherwise will be true. It is infix operator.
     - **! (Logical NOT operator):** Unary !(NOT) operator will reverse the logical state of its operands. If its operand is false, it makes its operand true. If its operand is true, it makes its operand false.
   - **Code Segment:**

```
document.write("All combination of &&(AND) operator<br\>");
  document.write("false && false: ", false &&false, "<br\>");
  document.write("false && true: ", false &&true, "<br\>");
  document.write("true && false: ", true &&false, "<br\>");
  document.write("true && true: ", true && true, "<br\><br\>");

  document.write("All combination of ||(OR) operator<br\>");
  document.write("false || false: ", false ||false, "<br\>");
  document.write("false || true: ", false ||true, "<br\>");
  document.write("true || false: ", true ||false, "<br\>");
  document.write("true || true: ", true || true, "<br\><br\>");

  document.write("All combination of !(NOT) operator<br\>");
  document.write("false: ", false, "<br\>");
  document.write("!false: ", !false, "<br\>");
  document.write("true: ", true, "<br\>");
  document.write("!true: ", !true, "<br\>");
```

- **Result of Execution:**
  All combination of &&(AND) operator
  false && false: false
  false && true: false
  true && false: false
  true && true: true

  All combination of ||(OR) operator
  false || false: false
  false || true: true
  true || false: true
  true || true: true

  All combination of !(NOT) operator
  false: false
  !false: true
  true: true
  !true: false
- **Explanation of the code segment:**First code segment will display the truth table of the &&(AND) operator in JavaScript. We see that it has the same result as the logic. && operator gives true only both of the operands are true. Otherwise, it is false. Second code segment will display the truth table of the ||(OR) operator in JavaScript language. We see that it has the same result as the logic. || operator gives false only both operands are false. Otherwise, it is true. Last code segment will display the truth table of the !(NOT) operator in JavaScript language. We see that it has the same result as the logic. ! operator reverse the logic condition of its operand. If its operand is true, it makes its operand false. If its operand is false, it makes its operand true.

2. Data types for operands of Boolean operators

- **Explanation:** In JavaScript, there are two primitive values of boolean data type, true and false. Besides these, all values except false, 0, -0, 0n, “”, null, undefined and NaN is considered as true in JavaScript.
- **Code Segment:**

```
var true_list= [10, 3.7, -89,"Ilke",8 + 22, 3 < 9, "false"];
   for(var i = 0, size = true_list.length; i <size; i++)
   {
    if(true_list[i]){
     document.write(true_list[i], " is true <br\>");
    }
   }
   document.write("<br\>");
   var false_list = [0, 0.0, -0, 0n, "",null,NaN, 39 <1, false, undefined];
   for(var i = 0, size = false_list.length; i <size; i++)
   {
```

```
    if(false_list[i]){
     document.write(false_list[i], " is true <br\>");
    }
    else
    {
    document.write(false_list[i], " is false <br\>");
    }
   }
```

- **Result of Execution:**
  10 is true
  3.7 is true
  -89 is true
  Ilke is true
  30 is true
  true is true
  false is true

  0 is false
  0 is false
  0 is false
  0 is false
  is false
  null is false
  NaN is false
  false is false
  false is false
  undefined is false
- **Explanation of the code segment:** In this code segment, first I created a list in JavaScript that consists of values that correspond to true value. Then, I created a list in JavaScript that consists of values that correspond to false values. Any values except false, 0, -0, 0n, “”, null, undefined and NaN are printed as true in output. We need to be careful that “false” is a string that has a value. Therefore, regardless of the meaning of this string, it is considered as true.

3. Operator precedence rules

- **Explanation:**!(NOT) operator has the highest precedence between the logical operators. &&(AND) operator has higher precedence than || (OR) operator. In other words, their precedence ranking is:

  ! > && > ||
- **Code Segment:**

```
document.write("!false && false: ",!false && false, "<br\>");
document.write("!true || true: ",!true || true,"<br\>");
 document.write("true || true && false: ", true || true && false, "<br\>");
```

- **Result of Execution:**
  !false && false: false
  !true || true: true
  true || true && false: true
- **Explanation of the code segment:** Its precedence rules as in other languages. If the && operator had higher precedence than the ! operator, it will be computed as !(false&&false) = !false = true, the result would have been true and "! false && false: true" would have seen in the output. However, this is not the situation. ! has higher precedence than && operator. So, the evaluation order will be (!false) && false = true && false = false. The result will be false and we can observe it in the output as "! false && false: false".

  If the || operator had higher precedence than the ! operator, it will be computed as !(true || true) = !true = false, the result would have been false and "!true || true: false" would have seen in the output. However, this is not the situation. ! has higher precedence than || operator. So, the evaluation order will be (!true) || true = false || true = true. The result will be true and we can observe it in the output as "!1 || 1: true". If the || operator had higher precedence than the && operator, it will be computed as ((true || true)&&false) = (true&&false) = false, the result would have been false and "true || true && false: false" would have seen in the output. However, this is not the situation. && has higher precedence than || operator. So, the evaluation order will be (true || (true && false))= true || false = true. The result will be true and we can observe it in the output as "true || true && false: true".

4. **Operator associativity rules**

- **Explanation:**&&(AND) and || (OR) operators have left-to-right associativity while !(NOT) operator has right to left associativity like in GO language.

- **Code Segment:**

```
function fun1()
{
    document.write("function 1 is evaluated<br\>");
  return true;
}
function fun2()
{
    document.write("function 2 is evaluated<br\>");
  return true;
}
function fun3()
{
    document.write("function 3 is evaluated<br\>");
  return true;
```

```
}
function fun4()
{
     document.write("function 4 is evaluated<br\>");
   return false;
}
function fun5()
{
     document.write("function 5 is evaluated<br\>");
   return false;
}
function fun6()
{
     document.write("function 6 is evaluated<br\>");
   return false;
}
function main() {
  document.write("fun2() && fun1() && fun3(): ",fun2() && fun1() && fun3(),
"<br\>");
     document.write("fun5() || fun6() || fun4(): ",fun5() || fun6() || fun4(), "<br\>");
     document.write("!!!fun4():",!!!fun4(),"<br\>");
}
main();
```

- **Result of Execution:**
  function 2 is evaluated
  function 1 is evaluated
  function 3 is evaluated
  fun2() && fun1() && fun3(): true
  function 5 is evaluated
  function 6 is evaluated
  function 4 is evaluated
  fun5() || fun6() || fun4(): false
  function 4 is evaluated
  !!!fun4():true
- **Explanation of the code segment:** When "function2() && function1() && function3()" expression in the if statement is executed, as seen from the output first function 2, then function 1 and finally function 3 is evaluated. This means, && operator has left to right associativity. Similarly, when "function5()||function6()||function4()" is evaluated, first function 5, then function 6 and finally function 4 is evaluated. This also means that || operator has left-to right associativity. To see the evaluation of each operand in the statement, I avoid short-circuit values in this code segment by choosing all of the three values as 1 in && operator associativity and 0 in || operator associativity. Finally, !!!function4() is

evaluated from right to left as (! (! (!function4()))) = (! (! true)) = (! false) = true.

5. **Operand evaluation order**

- **Explanation:**
  - Function vs variable:
  In JavaScript both && and || evaluate two function operands, two scalar or boolean variables or constants and combination of one function and one variable from left to right. There is no effect whether operand is a function call or Boolean variable or other type of variables. However, sometimes functions can have side effects and affect the result of the logical expression. According to that result, short-circuit may occur and one of the operands may not be evaluated.
  -Parentheses:
  Parentheses may change the evaluation order of expressions of more than two operands. They give priority to the expression in the parentheses. Therefore, operands in the parentheses are evaluated first.
  -Different type of variables: Since JavaScript consider any data value rather than 0,-0,0n, false,undefined and NaN as true, one question in mind can occur: "When a boolean expression whose operands are different types, does one of the operand is evaluated first because of its type? Or the expression is again evaluated from left to right again? " Answers to this question will be left-to-right evaluation.
- **Code Segment1:**

```
x = false;
function fun7(){
  x = true;
  document.write("function 7 is evaluated <br\>");
  return true;
}
function main()
{
    if(x && fun7())
    {
            document.write("side effect of fun7-1");
    }
    x= false;
    if(fun7() && x)
    {
            document.write("side effect of fun7-2");
    }
}
```

- **Result of Execution:**

```
function 7 is evaluated
side effect of fun7-2
```

- **Explanation of the code segment:** The Boolean expression in the first if statement, first global variable is evaluated then fun7 will be evaluated as we mentioned before. However, since the global variable is false, it will be short-circuit evaluation and fun7 will not be evaluated. In the second boolean expression, firstly, fun7 is evaluated. In this situation fun7 will change the value of the global variable, so it is not false anymore. Hence, since the value of global is true, the result of the expression will change and the " document.write("side effect of fun7-2");" statement will be executed as seen in the output. Order of operand evaluation changes due to the short-circuit evaluation and the result of the expression will change.

- **Code Segment2:**

```
document.write("true && true || false && false: ",true && true || false && false,"<br\>");
document.write("true && (true || false) && false: ",true && (true || false) && false,"<br\>");
```

- **Result of Execution:**

```
true && true || false && false: true
true && (true || false) && false: false
```

- **Explanation of the code segment:** In the first expression without parentheses, since the || operator has lower precedence, true && true || false && false is evaluated as (true && true) || (false && false) = true || false = true. In the second expression, parentheses change the evaluation order and the operands of the or operator will be first evaluated from left to right: true && (true || false) && false = true && true && false = (true && true) && false = true && false = false. Parentheses change the order of evaluation of operands.

- **Code Segment3:**

```
function fun5()

{

    document.write("function 5 is evaluated<br\>");

  return false;

}

function fun8()

{

    document.write("function 8 is evaluated<br\>");
```

```
   return 10;
}
function fun9()
{
     document.write("function 9 is evaluated<br\>");
   return 3.2;
}
function fun10()
{
     document.write("function 10 is evaluated<br\>");
   return undefined;
}
function main()
{
     if(fun8() && fun9())
     {
               document.write(fun8(),"&&",fun9(),": ", fun8() && fun9(),"<br\>");
     }
     if(fun9() && fun8())
     {
               document.write(fun9(),"&&",fun8(),": ", fun9() && fun8(),"<br\>");
     }
     if(!(fun5() || fun10()))
```

```
    {
        document.write(fun5(),"||",fun10(),": ", fun5() || fun10(),"<br\>");
    }
    if(!(fun10() || fun5()))
    {
        document.write(fun10(),"||",fun5(),": ", fun10() || fun5(),"<br\>");
    }
}
```

- **Result of Execution:**

function 8 is evaluated
function 9 is evaluated
function 8 is evaluated
function 9 is evaluated
function 8 is evaluated
function 9 is evaluated
10&&3.2: 3.2
function 9 is evaluated
function 8 is evaluated
function 9 is evaluated
function 8 is evaluated
function 9 is evaluated
function 8 is evaluated
3.2&&10: 10
function 5 is evaluated
function 10 is evaluated
function 5 is evaluated
function 10 is evaluated
function 5 is evaluated
function 10 is evaluated
false||undefined: undefined
function 10 is evaluated
function 5 is evaluated
function 10 is evaluated
function 5 is evaluated
function 10 is evaluated
function 5 is evaluated
undefined||false: false

- **Explanation of the code segment:** In this code segment, I tried to evaluate different types of operands with Boolean operators. As seen from the output, data type of the operand does not affect the evaluation order of the operand. However, it affects the result of the expression. For example, when we consider "undefined||false", result will be false. On the other hand, "false||undefined"will result in undefined. However,we know that the truth value of undefined is false. Therefore, we can say both of these expression's truth value is false. This shows the orthogonality of the JavaScript language.

6. Short-circuit evaluation

- **Explanation:** In JavaScript there is short-circuit evaluation that enables to avoid unnecessary evaluation of operands. If the first operand of *&&* operator is false, the second operand will not be evaluated. If the first operand of the || operator is true, the second operand will not be evaluated. However, this can be dangerous if the programs' correctness depends on the side effect of a function
- **Code Segment:**

```
y = 0;
function fun11()
{
    document.write("function 11 is evaluated<br\>");
    y++;
   return 1;
}
function main()
{
    if(y&&fun11())
    {
            document.write("side effect of fun7-1<br\>");
    }
    document.write("1-global y: ", y,"<br\>");
    y = 0;
    if(fun11()&&y)
    {
            document.write("side effect of fun7-2<br\>");
    }
    document.write("2-global y: ", y,"<br\>");
}
main();
```

- **Result of Execution:**

```
1-global y: 0
function 11 is evaluated
side effect of fun7-2
2-global y: 1
```

- **Explanation of the code segment:**In the first if statement, "if (y&& fun11()) "since the value of the global variable is false, AND operator will make short-circuit and do not evaluate fun11. However, if fun11 would be evaluated, it will change the value of the global variable. Integer values are widely used in decision making. So, their boolean values may change the flow of the program.In this sense, the order of operands have a significant impact on the program especially in short-circuit situations. If the correctness of the program is dependent on this change of global variable, this may make the program wrong. In the second if statement "if(fun11() && yl)" both of the operands will be evaluated.

My Evaluation about language: JavaScript has also separate data types for the boolean variables unlike C. Therefore, saying this language is readable will be appropriate. Besides that, since it also allows types such as integer, floating-point numbers, strings etc. to be operand in boolean expressions, I can say that it is orthogonality is higher than GO in that sense. This feature makes this language writable. On the other side, one need to remember the difference of expressions such as "false||undefined" and "undefined||false". This makes this language less readable.

PHP

1. **Boolean operators provided**
   - **Explanation:**There are six logical operators in PHP. These are:
     - **&& :** Result of the binary *&&* operator will be true when both operands are true, otherwise will be false. It is an infix operator.
     - **|| :** Result of the binary || operator will be false when both operands are false, otherwise will be true. It is an infix operator.
     - **! :** Unary !(NOT) operator will reverse the logical state of its operands. If its operand is false, it makes its operand true. If its operand is true, it makes its operand false.
     - **and :** Result of the binary **and** operator will be true when both operands are true, otherwise will be false. It is an infix operator.
     - **or:** Result of the binary **or** operator will be false when both operands are false, otherwise will be true. It is an infix operator.
     - **xor:** Result of the binary **xor** operator will be true when operands have different values, otherwise will be true. It is an infix operator.
   - **Code Segment:**

```
echo "All combination of AND operator</br>";
echo "false and false: ", var_dump(FALSE and FALSE), "</br>";
echo "false and true: ", var_dump(FALSE and TRUE), "</br>";
echo "true and false: ", var_dump(TRUE and FALSE), "</br>";
echo "true and true: ", var_dump(TRUE and TRUE), "</br></br>";

   echo "All combination of && operator</br>";
echo "false && false: ", var_dump(FALSE && FALSE), "</br>";
echo "false && true: ", var_dump(FALSE && TRUE), "</br>";
echo "true && false: ", var_dump(TRUE && FALSE), "</br>";
```

```
echo "true && true: ", var_dump(TRUE && TRUE), "</br></br>";

echo "All combination of OR operator</br>";
echo "false or false: ", var_dump(FALSE or FALSE), "</br>";
echo "false or true: ", var_dump(FALSE or TRUE), "</br>";
echo "true or false: ", var_dump(TRUE or FALSE), "</br>";
echo "true or true: ", var_dump(TRUE or TRUE), "</br></br>";

   echo "All combination of || operator</br>";
echo "false || false: ", var_dump(FALSE || FALSE), "</br>";
echo "false || true: ", var_dump(FALSE || TRUE), "</br>";
echo "true || false: ", var_dump(TRUE || FALSE), "</br>";
echo "true || true: ", var_dump(TRUE || TRUE), "</br></br>";

 echo "All combination of XOR operator</br>";
echo "false xor false: ", var_dump(FALSE xor FALSE), "</br>";
echo "false xor true: ", var_dump(FALSE xor TRUE), "</br>";
echo "true xor false: ", var_dump(TRUE xor FALSE), "</br>";
echo "true xor true: ", var_dump(TRUE xor TRUE), "</br></br>";
```

- **Result of Execution:**

```
All combination of AND operator
false and false: bool(false)
false and true: bool(false)
true and false: bool(false)
true and true: bool(true)

All combination of && operator
false && false: bool(false)
false && true: bool(false)
true && false: bool(false)
true && true: bool(true)

All combination of OR operator
false or false: bool(false)
false or true: bool(true)
true or false: bool(true)
true or true: bool(true)

All combination of || operator
false || false: bool(false)
false || true: bool(true)
true || false: bool(true)
true || true: bool(true)
```

All combination of XOR operator
false xor false: bool(false)
false xor true: bool(true)
true xor false: bool(true)
true xor true: bool(false)

- **Explanation of the code segment:** In this code segment, I used operators and, or, &&, || and xor. Since, ! (NOT) operator is the same as other languages. As seen from the output, the results of the "and" and "&&" operators are the same. Similarly, the results of the "or" and "||" operators are the same. xor operator has the meaning of "exclusive or" and becomes true when its operands have different values. The difference between and-&&, or-|| pairs is their precedence. Since false value is converted to an empty string and it is hard to observe it like that, "var_dump()" function is used to get information about the variable in this code.

2. Data types for operands of Boolean operators

- **Explanation:** There are two types of variables for Boolean data type in PHP: true, false. Both of them are case-insensitive. However, also in PHP, other data types has Boolean values like in JavaScript. All data values except 0, 0.0, "", "0", Null, empty array have true value.
- **Code Segment:**

```
$var1 = "0";
 $var2 = 0;
 $var3 = 59;
 $var4 = -13.8;
 echo " \" 0 \" is ",($var1 ? 'TRUE' : 'FALSE'), "</br>";
 echo "0 is ",($var2 ? 'TRUE' : 'FALSE'), "</br>";
 echo " \"\" is " , ("" ? 'TRUE' : 'FALSE'), "</br>";
 echo "59 is ",($var3 ? 'TRUE' : 'FALSE'), "</br>";
 echo  "-13.8 is ",($var4 ?'TRUE': 'FALSE'), "</br>";
 echo  "NULL is ",(null?'TRUE': 'FALSE'), "</br>";
 echo  "TRUE is ",(TRUE?'TRUE': 'FALSE'), "</br>";
 echo  "true is ",(true?'TRUE': 'FALSE'), "</br>";
 echo  "True is ",(True?'TRUE': 'FALSE'), "</br>";
```

- **Result of Execution:**

" 0 " is FALSE
0 is FALSE
"" is FALSE
59 is TRUE
-13.8 is TRUE
NULL is FALSE
TRUE is TRUE
true is TRUE
True is TRUE

- **Explanation of the code segment:** In this code segment I used a conditional operator to observe how other data types' truth values rather than true and false themselves are evaluated. As seen from the output, any values except 0, 0.0, "", "0", Null, empty array are TRUE and these values are FALSE. Besides that, the last three statements show that the boolean value true is case-insensitive as boolean value false.

3. Operator precedence rules

- **Explanation:**!(NOT) operator has the highest precedence between the logical operators. &&(AND) operator has higher precedence than || (OR) operator. **and, or , xor** operators have lower precedence than assignment operators. Therefore, their precedences are lower than !, && and || operators. **and** operator is higher than **xor,** and **xor** is higher than **or** operator. In other words, their precedence ranking is:

  ! > && > ||>**and** > **xor** > **or**>

- **Code Segment:**

```
$varand = fun1() and fun4();
  $varand2 = fun1() && fun4();
   echo "fun1() and fun4(): ", var_dump($varand), "</br>";
   echo "fun1() && fun4(): ", var_dump($varand2), "</br>";
  $varor = fun4() or fun1();
  $varor2 = fun4() || fun1();
   echo "fun1() or fun4(): ", var_dump($varor), "</br>";
   echo "fun1() || fun4(): ", var_dump($varor2), "</br>";
   $a = false or true || false;
   $b = false || true or false;
   echo "a is", var_dump($a), "</br>";
   echo "b is", var_dump($b), "</br>";
```

- **Result of Execution:**

```
fun1 is evaluated.
fun4 is evaluated.
fun1 is evaluated.
fun4 is evaluated.
fun1() and fun4(): bool(true)
fun1() && fun4(): bool(false)
fun4 is evaluated.
fun1 is evaluated.
fun4 is evaluated.
fun1 is evaluated.
fun1() or fun4(): bool(false)
fun1() || fun4(): bool(true)
a isbool(false)
b isbool(true)
```

- **Explanation of the code segment:** In this code segment, I first analyze the difference between "fun1() and fun4()" and "fun1() && fun4()" by assigning them to variables. Since the precedence of the **and** operator is lower than the assignment operator, first fun1() is assigned to the variable then, **and logical operation** will be executed. On the other side, since the precedence of the && operator is higher than assignment operator, first **and logical operation** will be executed, then the assignment will be performed. It is similar for the **or -** || pairs. Besides that, we can observe the precedence between **or -**|| pair by analyzing " $a = false or true || false;" and " $b = false || true or false;" statements. Since these operators are left associative, for a variable first assigning to false will be executed than the || operator will be performed. Finally, **or** operator will be executed.

4. Operator associativity rules

- **Explanation:** &&, **and**, ||, **or, xor** operators have left-to-right associativity while !(NOT) operator has right to left associativity. The associativity of !,&&, and, or ,|| are the same as the other languages.
- **Code Segment:**

```
echo "All combination of XOR operator with three operands </br>";
  echo "false xor false xor false: ", var_dump(FALSE xor FALSE xor FALSE), "</br>";
  echo "false xor false xor true : ", var_dump(FALSE xor FALSE xor TRUE), "</br>";
  echo "false xor true xor false: ", var_dump(FALSE xor TRUE xor FALSE), "</br>";
  echo "false xor true xor true: ", var_dump(FALSE xor TRUE xor TRUE),
"</br></br>";
    echo "true xor false xor false: ", var_dump(TRUE xor FALSE xor FALSE), "</br>";
  echo "true xor false xor true : ", var_dump(TRUE xor FALSE xor TRUE), "</br>";
  echo "true xor true xor false: ", var_dump(TRUE xor TRUE xor FALSE), "</br>";
  echo "true xor true xor true: ", var_dump(TRUE xor TRUE xor TRUE), "</br></br>";
```

- **Result of Execution:**

```
All combination of XOR operator with three operands
false xor false xor false: bool(false)
false xor false xor true : bool(true)
false xor true xor false: bool(true)
false xor true xor true: bool(false)

true xor false xor false: bool(true)
true xor false xor true : bool(false)
true xor true xor false: bool(false)
true xor true xor true: bool(true)
```

- **Explanation of the code segment:** In this code segment, I analyze the **xor** operator. It has also left to right associativity and thanks to this, it fits the logical truth table values of the **xor** logic. In **xor logic,** odd number of true values in expression result in a true boolean value.

5. Operand evaluation order

- **Explanation:**

  - Function vs variable:
  In PHP, &&, **and, or,** || evaluate two function operands, two scalar or boolean variables or constants and combination of one function and one variable from left to right. There is no effect whether operand is a function call or Boolean variable or other type of variables. However, sometimes functions can have side effects and affect the result of the logical expression. According to that result, short-circuit may occur and one of the operands may not be evaluated.
  -Parentheses:
  Parentheses may change the evaluation order of expressions of more than two operands. They give priority to the expression in the parentheses. Therefore, operands in the parentheses are evaluated first.
  -Different type of variables: Since PHP consider any data value rather than 0, 0.0, "", "0", Null, empty array as true, one question in mind can occur: "When a boolean expression whose operands are different types, does one of the operand is evaluated first because of its type? Or the expression is again evaluated from left to right again? " Answers to this question will be left-to-right evaluation.

- **Code Segment1:**

```
$globalvar = false;
function fun7()
{
    global $globalvar;
  $globalvar = true;
    echo "fun7 is evaluated.</br>";
  return true;
}
function main()
{
global $globalvar;
    if($globalvar && fun7())
  {
   echo "side effect of fun7-1</br>.";
  }
  $globalvar = false;
  if(fun7() && $globalvar)
  {
   echo "side effect of fun7-2</br>.";
  }
}
main();
```

- **Result of Execution:**
  fun7 is evaluated.
  side effect of fun7-2
- **Explanation of the code segment:**The Boolean expression in the first if statement, first global variable is evaluated then fun7 will be evaluated as we mentioned before. However, since the global variable is false, it will be short-circuit evaluation and fun7 will not be evaluated. In the second boolean expression, firstly, fun7 is evaluated. In this situation fun7 will change the value of the global variable, so it is not false anymore. Hence, since the value of global is true, the result of the expression will change and the " echo "side effect of fun7-2</br>.";" statement will be executed as seen in the output. Order of operand evaluation changes due to the short-circuit evaluation and the result of the expression will change.

- **Code Segment2:**
  echo "true and true or false and false: ",var_dump(true and true or false and false), "</br>";
  echo "true and (true or false) and false: ",var_dump(true and (true or false) and false), "</br>";
  echo "!(true && false) ", var_dump(!(true && false)), "</br>";
  echo "!true && false ", var_dump(!true && false), "</br>";

- **Result of Execution:**
  true and true or false and false: bool(true)
  true and (true or false) and false: bool(false)
  !(true && false) bool(true)
  !true && false bool(false)
- **Explanation of the code segment:**In the first expression without parentheses, since the **or** operator has lower precedence, true **and** true **or** false **and** false is evaluated as (true **and** true) **or** (false **and** false) = true **or** false = true. In the second expression, parentheses change the evaluation order and the operands of the or operator will be first evaluated from left to right: true **and** (true **or** false) **and** false = true **and** true **and** false = (true **and** true) **and** false = true **and** false = false. Parentheses change the order of evaluation of operands.

- **Code Segment3:**

```
function fun2()
{
        echo "fun2 is evaluated.</br>";
   return 10;
}
function fun3()
{
        echo "fun3 is evaluated.</br>";
   return 3.2;
}
```

```
function fun10()
{
          echo "fun10 is evaluated.</br>";
   return "";
}
function fun6()
{
          echo "fun6 is evaluated.</br>";
   return null;
}


function main()
{
      if(fun2()&&fun3())
        {
           echo "10 && 3.2: ",fun2()&&fun3(), "</br>";
        }
        if(fun3()&&fun2())
        {
           echo "3.2 && 10: ",fun3()&&fun2(),"</br>";
        }
         if(!(fun6()||fun10()))
        {
           echo "null || \"\": ",var_dump(fun6()||fun10()), "</br>";
        }
         if(!(fun10()||fun6()))
        {
           echo " \"\" || null: ",var_dump(fun10()||fun6()), "</br>";
        }
     }
     main();
```

- **Result of Execution:**

```
fun2 is evaluated.
fun3 is evaluated.
10 && 3.2: fun2 is evaluated.
fun3 is evaluated.
1
fun3 is evaluated.
fun2 is evaluated.
3.2 && 10: fun3 is evaluated.
fun2 is evaluated.
1
```

```
fun6 is evaluated.
fun10 is evaluated.
null || "": fun6 is evaluated.
fun10 is evaluated.
bool(false)
fun10 is evaluated.
fun6 is evaluated.
"" || null: fun10 is evaluated.
fun6 is evaluated.
bool(false)
```

- **Explanation of the code segment:**In this code segment, I tried to evaluate different types of operands with Boolean operators. As seen from the output, data type of the operand does not affect the evaluation order of the operand. Unlike JavaScript, it does not affect the result of the expression. For example, when we consider "3.2&&10", the result will be 1 and "10&&3.2" will result in 1. It is still complicated. I again use var_dump to see the value of false boolean expression.

6. Short-circuit evaluation

- **Explanation:**In PHP there is short-circuit evaluation that enables to avoid unnecessary evaluation of operands. If the first operand of &&,**and** operator is false, the second operand will not be evaluated. If the first operand of the ||,**or** operator is true, the second operand will not be evaluated. However, this can be dangerous if the programs' correctness depends on the side effect of a function. It is not possible short-circuit in **xor** operator.
- **Code Segment:**

```
$globalvar = false;
function fun7()
{
     global $globalvar;
   $globalvar = true;
     echo "fun7 is evaluated.</br>";
   return true;
}
function main()
{
global $globalvar;
     if($globalvar && fun7())
   {
    echo "side effect of fun7-1</br>.";
   }
  echo "1-globalvar : ", var_dump($globalvar), "</br>";
   $globalvar = false;
   if(fun7() && $globalvar)
   {
```

```
    echo "side effect of fun7-2</br>.";
   }
echo "2-globalvar : ", var_dump($globalvar), "</br>";
}
main();
```

- **Result of Execution:**
  1-globalvar : bool(false)
  fun7 is evaluated.
  side effect of fun7-2
  .2-globalvar : bool(true)
- **Explanation of the code segment:**In the first if statement, "if ($globalvar&& fun7()) "since the value of the global variable is false, AND operator will make short-circuit and do not evaluate fun7. However, if fun7 would be evaluated, it will change the value of the global variable. If the correctness of the program is dependent on this change of global variable, this may make the program wrong. In the second if statement "if(fun7() && $globalvar)" both of the operands will be evaluated.

My Evaluation about language: PHP has more operators than other languages and their precedence rule is different then the usual programming language logic. So, it is not so readable and writable language. Besides that, it also allows other data types to have boolean values like JavaScript, unlike GO. This provides orthogonality to the language but on the other hand, makes it less readable because it is hard to keep track of the result of the boolean expression that consists of different data types. For example, the result of "10&&3.2" is 1, while the result of "null || "" " is false. These kinds of details make language less writable and readable.

Python

1. **Boolean operators provided**
   - **Explanation:**There are three logical operators in Python. These are:
     - **and:** Result of the binary **and** operator will be true when both operands are true, otherwise will be false. It is an infix operator.
     - **or :** Result of the binary **or** operator will be false when both operands are false, otherwise will be true. It is an infix operator.
     - **not:** Unary **not** operator will reverse the logical state of its operands. If its operand is false, it makes its operand true. If its operand is true, it makes its operand false.
   - **Code Segment:**

```
T = True
F = False
print("All combination of and operator");
print("F and F:", F and F);
print("F and T:", F and T);
print("T and F:", T and F);
print("T and T:", T and T);
```

```
print("\nAll combination of and operator");
print("F or F:", F or F);
print("F or T:", F or T);
print("T or F:", T or F);
print("T or T:", T or T);

print("\nAll combination of not operator");
print("F :", F);
print("not(F) :", not(F) );
print("T :", T);
print("not(T) :", not(T) );
```

- **Result of Execution:**

All combination of and operator

F and F: False

F and T: False

T and F: False

T and T: True

All combination of and operator

F or F: False

F or T: True

T or F: True

T or T: True

All combination of not operator

F : False

not(F) : True

T : True

not(T) : False

- **Explanation of the code segment: :** First code segment will display the truth table of the **and** operator in Python language. We see that it has the same result as the logic. **and** operator gives true only both of the operands are true. Otherwise, it is false. Second code segment will display the truth table of the **or** operator in python language. We see that it has the same result as the logic.**or** operator gives false only both operands are false. Otherwise, it is true. Last code segment will display the truth table of the **not** operator in Python language. We see that it has the same result as the logic. **not** operator reverse the logic condition of its operand. If its operand is T(true), it makes its operand false. If its operand is F(false), it makes its operand true.

2. **Data types for operands of Boolean operators**

- **Explanation:** In Python, there are two primitive values of boolean data type, True and False. Besides these, most values except False, 0, “”, None and empty values such as (), [],{} are considered as true in Python.
- **Code Segment:**

```
false_values= [False, (), [],{},0,"",None,0.0,-0, 4 > 6, 3 -3];
for i in false_values:
    print("\n",i, " is ", bool(i))
true_values = [True, (1,3), [4,6], 34, -12, 5.78,"Ilke", 'c', 2 > 1, 3 + 3]
for i in true_values:
    print("\n",i, " is ", bool(i))
```

- **Result of Execution:**

False is False
() is False
[] is False
{} is False
0 is False
 is False
None is False
0.0 is False
0 is False
False is False
0 is False
True is True
(1,3) is True
[4,6] is True
34 is True
-12 is True
5.78 is True
Ilke is True
c is True
6 is True

- **Explanation of the code segment:**In this code segment, first I created a list in Python that consists of values that correspond to false values. Then, I created a list in Python that consists of values that correspond to true values. Any values except false, False, (), [],{},0,"",None,0.0,-0 are printed as true in output.

3. **Operator precedence rules**

- **Explanation: not** operator has the highest precedence between the logical operators. **and** operator has higher precedence than **or** operator. In other words, their precedence ranking is:

  not > and > or

- **Code Segment:**

```
print("not False and False :",bool(not False and False) )
print("not True or True : ", bool(not True or True))
print("True or True and False : ", bool( True or True and False))
```

- **Result of Execution:**

```
not False and False : False
not True or True : True
True or True and False : True
```

- **Explanation of the code segment:**Its precedence rules as in other languages. If the **and** operator had higher precedence than the **not** operator, it will be computed as **not**(False **and** False) = **not** false = true, the result would have been true and "not False and False : True" would have been seen in the output. However, this is not the situation. **not** has higher precedence than **and** operator. So, the evaluation order will be (**not** False) **and** False = True **and** False = False. The result will be false and we can observe it in the output as "not False and False : False".

  If the **or** operator had higher precedence than the **not** operator, it would be computed as **not** (True **or** True) = **not** True = False, the result would have been false and " not True or True : False " would have been seen in the output. However, this is not the situation. **not** has higher precedence than **or** operator. So, the evaluation order will be (**not** True) **or** True = False **or** True = True. The result will be true and we can observe it in the output as "**not** True **or** True : True".
  If the **or** operator had higher precedence than the **and** operator, it will be computed as ((True **or** True) **and** False) = (True **and** False) = False, the result would have been false and "True **or** True **and** False: False" would have seen in the output. However, this is not the situation. **and** has higher precedence than || operator. So, the evaluation order will be (True **or** (True **and** False))= True **or**| False = True. The result will be true and we can observe it in the output as "True **or** True **and** False: True".

4. **Operator associativity rules**

- **Explanation:and** and **or** operators have left-to-right associativity while **not** operator has right to left associativity.

- **Code Segment:**

```
def fun1():
   print("fun1 is evaluated")
   return True
def fun2():
   print("fun2 is evaluated")
   return True
def fun3():
   print("fun3 is evaluated")
   return True
def fun4():
   print("fun4 is evaluated")
   return False
def fun5():
   print("fun5 is evaluated")
   return False
def fun6():
   print("fun6 is evaluated")
   return False
def main():
   print("fun2() and fun1() and fun3()",fun2() and fun1() and fun3())
   print("fun5() or fun6() or fun4() ",fun5() or fun6() or fun4())
main()
```

- **Result of Execution:**

```
fun2 is evaluated
fun1 is evaluated
fun3 is evaluated
fun2() and fun1() and fun3() True
fun5 is evaluated
fun6 is evaluated
fun4 is evaluated
fun5() or fun6() or fun4() False
```

- **Explanation of the code segment:** When "function2() and function1() and function3()" expressions in the if statement are executed, as seen from the output first function 2, then function 1 and finally function 3 is evaluated. This means, **and** operator has left to right associativity. Similarly, when "function5() or function6() or function4()" is evaluated, first function 5, then function 6 and finally function 4 is evaluated. This also means that **or** operator has left-to-right associativity. To see the evaluation of each operand in the statement, I avoid short-circuit values in this code segment by choosing all of the three values as True for the associativity of the **and** operator and false for associativity of the **or** operator.

5. **Operand evaluation order**

- **Explanation:**

  - Function vs variable:
  In Python both **and** and **or** evaluate two function operands, two scalar or boolean variables or constants and combination of one function and one variable from left to right. There is no effect whether operand is a function call or Boolean variable or other type of variables. However, sometimes functions can have side effects and affect the result of the logical expression. According to that result, short-circuit may occur and one of the operands may not be evaluated.
  -Parentheses:
  Parentheses may change the evaluation order of expressions of more than two operands. They give priority to the expression in the parentheses. Therefore, operands in the parentheses are evaluated first.
  -Different type of variables: Since Python consider any data value rather than False, (), [],{},0,"",None,0.0,-0 as true, one question in mind can occur: "When a boolean expression whose operands are different types, does one of the operand is evaluated first because of its type? Or the expression is again evaluated from left to right again? " Answers to this question will be left-to-right evaluation.

- **Code Segment1:**

```
x = False
def fun7():
   global x
   x = True
   print("fun7 is evaluated")
   return True
def main():
   global x
   if x and fun7():
      print("side effect of fun7-1")
   x =False
   if fun7() and x:
      print("side effect of fun7-2")
main()
```

- **Result of Execution:**

```
fun7 is evaluated
side effect of fun7-2
```

- **Explanation of the code segment:**The Boolean expression in the first if statement, first global variable is evaluated then fun7 will be evaluated as we mentioned before. However, since the global variable is false, it will be short-circuit evaluation and fun7 will not be evaluated. In the second boolean expression, firstly, fun7 is evaluated. In this situation fun7 will change the value of the global variable, so it is not false anymore. Hence, since the value of global is true, the result of the expression will

change and the “ print("side effect of fun7-2").";” statement will be executed as seen in the output. Order of operand evaluation changes due to the short-circuit evaluation and the result of the expression will change.

- **Code Segment2:**

```
print("true and true or false and false: ", bool(True and True or False and False))
print("true and (true or false) and false: ", bool(True and (True or False) and False))
```

- **Result of Execution:**

```
true and true or false and false: True
true and (true or false) and false : False
```

- **Explanation of the code segment:** In the first expression without parentheses, since the **or** operator has lower precedence, True **and** True **or** False **and** False is evaluated as (True **and** True) **or** (False **and** False) = True **or** False = True. In the second expression, parentheses change the evaluation order and the operands of the or operator will be first evaluated from left to right: True **and** (True **or** False) **and** False = True **and** True **and** False = (True **and** True) **and** False = True **and** False = False. Parentheses change the order of evaluation of operands.

- **Code Segment3:**

```
def fun8():
   print("fun8 is evaluated")
   return 10
def fun9():
   print("fun9 is evaluated")
   return 3.2
def fun10():
   print("fun10 is evaluated")
   return ""
def fun11():
   print("fun11 is evaluated")
   return None
def main():
   if fun8() and fun9():
      print("10 and 3.2: ", fun8() and fun9() )
   if fun9() and fun8():
      print("3.2 and 10: ", fun9() and fun8() )
   if not(fun10() or fun11()):
      print(" \"\" or None: ", fun10() or fun11())
   if not(fun11() or fun10()):
      print(" None or \"\": ", fun11() or fun10())
```

- **Result of Execution:**

```
fun8 is evaluated
fun9 is evaluated
fun8 is evaluated
fun9 is evaluated
10 and 3.2:  3.2
fun9 is evaluated
fun8 is evaluated
fun9 is evaluated
fun8 is evaluated
3.2 and 10:  10
fun10 is evaluated
fun11 is evaluated
fun10 is evaluated
fun11 is evaluated
 "" or None:  None
fun11 is evaluated
fun10 is evaluated
fun11 is evaluated
fun10 is evaluated
 None or "":
```

- **Explanation of the code segment:**In this code segment, I tried to evaluate different types of operands with Boolean operators. As seen from the output, data type of the operand does not affect the evaluation order of the operand. However, like JavaScript, it affects the result of the expression. For example, when we consider "3.2 and 10", the result will be 10 and "10 and 3.2" will result in 3.2.

6. Short-circuit evaluation

- **Explanation:**In Python there is short-circuit evaluation that enables to avoid unnecessary evaluation of operands. If the first operand of **and** operator is false, the second operand will not be evaluated. If the first operand of the **or** operator is true, the second operand will not be evaluated. However, this can be dangerous if the programs' correctness depends on the side effect of a function.
- **Code Segment:**

```
x = False
def fun7():
   global x
   x = True
   print("fun7 is evaluated")
   return True
def main():
   global x
   if x and fun7():
      print("side effect of fun7-1")
   print("1- global x:", x)
   x =False
```

```
if fun7() and x:
    print("side effect of fun7-2")
print("2- global x:", x)
```

- **Result of Execution:**
  1- global x: False
  fun7 is evaluated
  side effect of fun7-2
  2- global x : True
- **Explanation of the code segment:**In the first if statement, "if x and fun7() "since the value of the global variable is false, AND operator will make short-circuit and do not evaluate fun7. However, if fun7 would be evaluated, it will change the value of the global variable. If the correctness of the program is dependent on this change of global variable, this may make the program wrong. In the second if statement "if fun7() and x" both of the operands will be evaluated.

- **My Evaluation about language:** Python is a useful language but not the best one in terms of boolean expressions. It also allows many data types to be operand of boolean expressions. It may make this language more complicated.

Rust

1. **Boolean operators provided**
   - **Explanation:**There are three Boolean operators provided by Rust language. These are:
     - **&& (Logical AND operator):** Result of the binary && (AND) operator will be true when both operands are true, otherwise will be false. It is infix operator.
     - || **(Logical OR operator):** Result of the binary || (OR) operator will be false when both operands are false, otherwise will be true. It is infix operator.
     - **! (Logical NOT operator):** Unary !(NOT) operator will reverse the logical state of its operands. If its operand is false, it makes its operand true. If its operand is true, it makes its operand false.
   - **Code Segment:**

```
let T = true;
let F = false;
println!("All combinations of && operator");
println!("F && F: {}", F && F);
println!("F && T: {}", F && T);
println!("T && F: {}", T && F);
println!("T && T: {}", T && T);

println!();
println!("All combinations of || operator");
println!("F || F: {}", F || F);
```

```
    println!("F || T: {}", F || T);
    println!("T || F: {}", T || F);
    println!("T || T: {}", T || T);
    println!();

    println!("All combinations of ! operator");
    println!("F:{}", F );
    println!("!F: {}", !F);
    println!("T: {}", T);
    println!("!T: {}", !T);
```

- **Result of Execution:**

```
All combinations of && operator
F && F : false
F && T: false
T && F: false
T && T: true

All combinations of || operator
F || F : false
F || T: true
T || F: true
T || T: true

All combinations of ! operator
F: false
!F: true
T: true
!T: false
```

- **Explanation of the code segment:**First code segment will display the truth table of the && operator in Rust. We see that it has the same result as the logic. && operator gives true only both of the operands are true. Otherwise, it is false. Second code segment will display the truth table of the || operator in Rust language. We see that it has the same result as the logic. || operator gives false only both operands are false. Otherwise, it is true. Last code segment will display the truth table of the ! operator in Rust language. We see that it has the same result as the logic. ! operator reverse the logic condition of its operand. If its operand is true, it makes its operand false. If its operand is false, it makes its operand true.

2. Data types for operands of Boolean operators

- **Explanation:** Rust has Boolean values true and false as predeclared constants in the language. Non-bool values such as integers, floating point numbers, strings cannot be used as Boolean variables unlike C language. Therefore, arithmetic expressions

cannot be used as operands of the Boolean operators in Rust language. However, relational expressions can be used as operands also in Rust language. Besides these, function calls whose return value is boolean type can also be used as operands.

- **Code Segment:**

```
fn funct1() -> bool
{
   println!("function 1 is evaluated");
   return true;
}
fn main(){
     if true{
               println!("Print this");
     }
     if false
     {
               println!("Do not print this");
     }
     let a = 13;
     let c = a % 2 == 1;
     println!("c: {} ", c);
     let b = a <= 9;
     println!("b: {}", b);
     let d = true && funct1();
     println!("d: {}", d);
}
```

- **Result of Execution:**

```
Print this
c: true
b: false
function 1 is evaluated
d: true
```

- **Explanation of the code segment:** There are two constants for Boolean type of variable in Rust language: true and false can be used to make decisions as in the example in if statement. Besides, as in the initialization of c, we can see that relational expressions have Boolean variables. In, " let d = true && funct1();" statement we observe that function calls whose return type is a Boolean type can also be used as an operand of a logical expression. However, when we try to return other types rather than bool types, we get errors. Rust language does not support this kind of conversions.

3. **Operator precedence rules**

- **Explanation:**! operator has the highest precedence between the logical operators. && operator has higher precedence than || operator. In other words, their precedence ranking is:

  ! > && > ||

- **Code Segment:**

```
println!("!false && false:{} ",!false && false);
println!("!true || true: {} ",!true || true);
println!("true || true && false:{} ", true || true && false);
```

- **Result of Execution:**

  !false && false: false
  !true || true: true
  true || true && false: true

- **Explanation of the code segment:** Its precedence rules as in other languages. If the && operator had higher precedence than the ! operator, it will be computed as !(false && false) = !false = true, the result would have been true and "! false && false: true" would have seen in the output. However, this is not the situation. ! has higher precedence than && operator. So, the evaluation order will be (!false) && false = true && false = false. The result will be false and we can observe it in the output as "! false && false: false".

  If the || operator had higher precedence than the ! operator, it will be computed as !(true || true) = !true = false, the result would have been false and "!true || true: false" would have seen in the output. However, this is not the situation. ! has higher precedence than || operator. So, the evaluation order will be (!true) || true = false || true = true. The result will be true and we can observe it in the output as "!true || true: true".
  If the || operator had higher precedence than the && operator, it will be computed as ((true || true) && false) = (true && false) = false, the result would have been false and "true || true && false: false" would have seen in the output. However, this is not the situation. && has higher precedence than || operator. So, the evaluation order will be (true || (true && false))= true || false = true. The result will be true and we can observe it in the output as "true || true && false: true".

4. **Operator associativity rules**

- **Explanation: :** &&(AND) and || (OR) operators have left-to-right associativity while !(NOT) operator has right to left associativity like in Rust language.
- **Code Segment:**

```
fn funct1() -> bool
{
   println!("function 1 is evaluated");
   return true;
}
fn funct2() -> bool
```

```
{
   println!("function 2 is evaluated");
   return true;
}
fn funct3() -> bool
{
   println!("function 3 is evaluated");
   return true;
}
fn funct4() -> bool
{
   println!("function 4 is evaluated");
   return false;
}
fn funct5() -> bool
{
   println!("function 5 is evaluated");
   return false;
}
fn funct6() -> bool
{
   println!("function 6 is evaluated");
   return false;
}

fn main() {
   println!("funct2() && funct1() && funct3() {}",funct2() && funct1() && funct3() );
   println!("funct5() || funct6() || funct4() {}",funct5() || funct6() || funct4());
}
```

- **Result of Execution:**

```
function 2 is evaluated
function 1 is evaluated
 function 3 is evaluated
funct2() && funct1() && funct3() true
function 5 is evaluated
function 6 is evaluated
function 4 is evaluated
funct5() || funct6() || funct4() false
```

Explanation of the code segment: When "function2() && function1() && function3()" expression in the if statement is executed, as seen from the output first function 2, then function 1 and finally function 3 is evaluated. This means, && operator has left to right associativity. Similarly, when "function5()||function6()||function4()" is evaluated, first function 5, then function 6

and finally function 4 is evaluated. This also means that || operator has left-to right associativity. To see the evaluation of each operand in the statement, I avoid short-circuit values in this code segment by choosing all of the three values as 1 in && operator associativity and 0 in || operator associativity.

5. Operand evaluation order

- **Explanation:**
  - **Function vs variable:**
    In Rust language both && and || evaluate two function operands, two boolean variables or constants and combination of one function from left to right. There is no effect whether operand is a function call or Boolean variable. However, sometimes functions can have side effects and affect the result of the logical expression in other languages. Rust also encounters these kinds of situations. However, Rust's compile-time guarantees for safe mutation rely on unique access.
  - **Parentheses:**
    Parentheses may change the evaluation order of expressions of more than two operands. They give priority to the expression in the parentheses. Therefore, operands in the parentheses are evaluated first.

- **Code Segment1.1:**
  -Function vs variable:

```
static mut globalvar: bool = false;
fn funct7() -> bool
{
unsafe{
   globalvar = true;
   }
   println!("function 7 is evaluated");
   return true;
}
fn main() {
unsafe{
   if globalvar && funct7()
   {
      println!("side effect of function 7-1");
   }
   globalvar = false;
   if funct7() && globalvar
   {
      println!("side effect of function 7-2");
   }
 }
```

- **Result of Execution:**

```
function 7 is evaluated
```

Side effect of function 7-2

Explanation of the code segment: The Boolean expression in the first if statement, first global variable is evaluated then function7 will be evaluated as we mentioned before. However, since the global variable is false, it will be short-circuit evaluation and function7 will not be evaluated. In the second boolean expression, firstly, function7 is evaluated. In this situation function7 will change the value of the global variable, so it is not false anymore. Hence, since the value of global is true, the result of the expression will change and the " println!("side effect of function 7-2"); " statement will be executed as seen in the output. Order of operand evaluation changes due to the short-circuit evaluation and the result of the expression will change. Note that in this code segment, to implement side effects of a function, we change the value of a global variable by using an **"unsafe"** block. Normally, if we do not use it, the compiler will generate errors. Rust is a safer language than others in this sense.

- **Code Segment2:**
  -Parentheses

```
println!("true && true || false && false: {} ",true && true || false && false);
println!("true && (true || false) && false: {} ",true && (true || false) && false);
```

- **Result of Execution:**

```
true && true || false && false:  true
true && (true || false) && false:  false
```

- **Explanation of the code segment:** In the first expression without parentheses, since the || operator has lower precedence, true && true || false && false is evaluated as (true && true) || (false && false) = true || false = true. In the second expression, parentheses change the evaluation order and the operands of the or operator will be first evaluated from left to right: true && (true || false) && false = true && true && false = (true && true) && false = true && false = false.

6. Short-circuit evaluation

- **Explanation:** In Rust language there is short-circuit evaluation that enables to avoid unnecessary evaluation of operands. If the first operand of && operator is false, the second operand will not be evaluated. If the first operand of the || operator is true, the second operand will not be evaluated.
- **Code Segment:**

```
if funct1() || funct5(){
    println!("Short circuit");
}
if funct5() && funct1()
{
    println!("Do not print this");
}
```

- **Result of Execution:**

function 1 is evaluated
short circuit
function 5 is evaluated

- **Explanation of the code segment:** In the first if statement, since funct1() return true, there is no need to evaluate the second operand. So, it directly prints the inside of the if statement. In the second if statement, since funct5() return false, there is no need to evaluate the second operand.

My Evaluation about language: In my opinion, Rust is the most reliable programming language among others. It has only two data values: true and false like in GO programming language. Even if other languages are more orthogonal, it is easy to read Rust. However, I think it is not writable because it is hard to change the value of global variables.

Conclusion

1. **Which language is the best in terms of boolean expressions and why?**
   I think GO is the best in terms of boolean expressions. Because other languages like JavaScript, Php allows other data types to be used as operands in boolean expressions. Even if it seems more orthogonal, it makes boolean expressions more complicated in these languages. Languages such as GO and Rust do not have this feature. They only allow true and false as the boolean values in boolean expressions. This can be considered as a restriction but it is more reliable and readable. Rust is not the best because it has other restrictions such as hard to access global variables in boolean expressions.
2. **My learning Strategy**
   I learn these languages by using trial-and-error methods by using online compilers.
   -Materials and tools I used
   I searched a lot of internet sources to understand the syntax of the programming languages and I also used our textbook to find some information. Some of these sites are:
   For C:
   https://www.tutorialspoint.com/cprogramming/c_logical_operators.htm
   kitap
   https://www.tutorialspoint.com/cprogramming/c_operators_precedence.htm
   https://www.geeksforgeeks.org/operator-precedence-and-associativity-in-c/
   For GO:
   https://golang.org/ref/spec#Operators
   For PHP:
   PHP Data Types (w3schools.com)
   'AND' vs '&&' as operator in PHP - GeeksforGeeks
   For JavaScript:
   Truthy - MDN Web Docs Glossary: Definitions of Web-related terms | MDN (mozilla.org)
   For Python:
   Python Booleans (w3schools.com)
   Precedence and Associativity of Operators in Python (programiz.com)

9. Notes about booleans and logical operators — Python Notes (0.14.0) (thomas-cokelaer.info)

For Rust:

Data Types - The Rust Programming Language (mit.edu)
Global variables? Do they exist? : rust (reddit.com)
bool - Rust (rust-lang.org)
if/else - Rust By Example (rust-lang.org)

-Experiments I performed

I performed so many experiments by using compilers. For example, before searching from the internet, I tried possible data types of boolean operands such as integers, floating-point numbers, characters etc. When I failed to understand the syntax and semantics, I searched it. Besides that, I wrote boolean expressions myself by trying specific ones so that by analyzing them, I can understand the precedence and associativity rules of boolean expressions. I try to guess the results of these expressions and verify them by using an online compiler.

-URL s of online compilers/interpreters

C:

https://www.onlinegdb.com/online_c_compiler

GO:

https://www.onlinegdb.com/online_go_compiler

JS:

Tryit Editor v3.6 (w3schools.com)

PHP:

PHP Tryit Editor v1.2 (w3schools.com)

Python:

Online Python Compiler - online editor (onlinegdb.com)

Rust:

Online Rust Compiler - online editor (onlinegdb.com)