CS315

HOMEWORK 1

HACI ÇAKIN

21802641

SECTION 2

1) C

- **Boolean Operators Provided - Data Types for Operands of Boolean Operators**

In c, the boolean operators are &&, ||, !. These operators can be used with integers. C does not have boolean type directly named bool. Instead of this, c uses 0 as a false, other integer values as a true logic.

Ex.

```
int num1 = 0;
int num2 = 3;
printf("%d \n" , !!num1);
printf("%d \n" , !!num2);
Output:
0
1

printf("%d \n" , num1 && num2);
printf("%d \n" , num1 || num2);
printf("%d \n" , !num1 || num2);
Output:
0
1
1
```

- **Operator Precedence Rules**

In c, operator precedence rules is as follow in descending order;

1 —> ()

2 —> !

3 —> &&

4 —> ||

Ex.

```
int num1 = 0;
int num2 = 3;
printf("%d \n" , num2 && !num2 && !( num2 ||  !num2) && num1);
Output:
0

In this example, first, the parentheses are calculated.
(num2 || !num2)
In the parentheses, not operator is involving. Then or operator is
done. After finishing inside of the parentheses, not operator
follows it. Then if there is an and operator, and operator is
calculated before the or operator. Lastly, or operator is done.
Therefore, result is false which is 0.
```

- **Operator Associativity Rules**

  And, or operators are left associativity.

Ex.

```
int num1 = 0;
int num2 = 3;
printf("%d \n" , num2  &&  !num2 && num1 );
Output:
0

In this example, because and operator is left associativity,
statement is calculated in this form;
(num2 && !num2) && num1

But if it was right associativity, then equations would be
calculated as follow;
num2 && (!num2 && num1)
```

- **Operan Evaluation Order**

  In c, operant evaluation order is left to right.

Ex.

```
int myfunction1() {
        printf(" * 1st function * ");
        return 0;
      }
int myfunction2() {
        printf(" * 2st function * ");
        return 1;
      }
int myfunction3() {
        printf(" * 3st function * ");
        return 0;
      }
printf("%d \n" , myfunction1() && myfunction2() && myfunction3() );
printf("%d \n" , myfunction1() || myfunction2() || myfunction3() );
```

```
Output:
* 1st function * 0
* 1st function * * 2nd function * 1

In this example, both examples shows that myfunction1() is
calculated before the myfunction2(). Therefore, this shows that c
has calculation order of left to right.
```

- **Short-Circuit Evaluation**

  In C, short circuit is used for and , or operators.

```
int myfunction1() {
        printf(" * 1st function * ");
        return 0;
      }
int myfunction2() {
        printf(" * 2st function * ");
        return 1;
```

```
    }
int myfunction3() {
       printf(" * 3st function * ");
       return 0;
    }
printf("%d \n" , myfunction1() && myfunction2() && myfunction3() );
printf("%d \n" , myfunction1() || myfunction2() || myfunction3() );
Output:
* 1st function * 0
* 1st function * * 2nd function * 1

First line of output is the example of short-circuit of and
operator. Because myfunction1() returns false value, there is no
need to check rest of the statement. So only myfunction1() is
called.

Second line of output is the example of short-circuit of or
operator. For this short circuit, statement first calls the
myfunction1(). Since it returns false and it is an or operator,
program needs to check until it finds true value for or operator.
When it faces with true thanks to myfunction2(), it stops
executing this statement which means short-circuit is occurred.
```

- **Evaluation Of C**

In C language, logical operators are mostly same with todays standards(Because c is the ancestor of most of the languages, it makes sense). However, in c there is no logical primitive data type like boolean. This decreases writability and readability. Using integer values can causes misleadings or decrement in the reliability of language.

- **Resources that I used for C**

For C language, I used an online compiler(https://www.programiz.com/c-programming/online-compiler/). I'm familiar with syntax, therefore in order to learn I do not need any sources.

2) GOLANG

- **Boolean Operators Provided - Data Types for Operands of Boolean Operators**

In Golang, logical operators are same with C which are &&, || , !. For Boolean operators, there is a boolean data type to present true and false. Unlike C and js, Golang do not use integer values for that.

Ex.

```
var bool1 = false
var bool2 = true

fmt.Println(bool1 && bool2)
fmt.Println( bool1 || bool2)
fmt.Println ( !bool1 && bool2)
```

```
Output:
false
true
false
```

- **Operator Precedence Rules**

In Golang, operator precedence rules is as follow in descending order;

1 —> ()

2 —> !

3 —> &&

4 —> ||

Ex.

```
var bool1 = false
var bool2 = true
fmt.Println(bool2 && !bool2 && (bool2 || !bool2) && bool1);
Output:
```

```
False

In this example, first, the parentheses are calculated.
(bool2 || !bool2). In the parenthesis, not operator change the
value reverse. Then or operator runs. After finishing insede of
the parentheses, rest of the statement is calculated according to
precedence rules.
```

- **Operator Associativity Rules**

  And, or operators are left associativity.

Ex.

var bool1 = false

var bool2 = true

fmt.Println(bool2 && !bool2 && (bool2 || !bool2) && bool1);

```
Output:
0

     In this example(it is same with previous one), after
finishing inside of the parentheses, equation becomes,

bool2 && !bool2 && resultOfInsideOfParantheses && bool1

Since in golang and, or operators are left associativity, equation
calculates like as;

((bool2 && !bool2) && resultOfInsideOfParantheses) && bool1

If it was right associativity, then equation would be calculated
like as;

bool2 && (!bool2 && (resultOfInsideOfParantheses && bool1))
```

- **Operan Evaluation Order**

  In GO, operant evaluation order is left to right.

Ex.

```
func myfunction1()  bool {
        fmt.Println(" * 1st function * ");
        return false;
    }
func myfunction2() bool {
        fmt.Println(" * 2st function * ");
        return true;
    }
func myfunction3() bool {
        fmt.Println(" * 3st function * ");
        return false;
    }
fmt.Println(myfunction1() && myfunction2() && myfunction3() );
fmt.Println(myfunction1() || myfunction2() || myfunction3() );
```

```
Output:
* 1st function * 0
* 1st function * * 2nd function * 1

In this example, both examples shows that myfunction1() is
calculated before the myfunction2(). Therefore, this shows that c
has calculation order of left to right.
```

- **Short-Circuit Evaluation**

In GO, short circuit is used for and , or operators like C, js, python uses.

```
func myfunction1()  bool {
        fmt.Println(" * 1st function * ");
        return false;
    }
func myfunction2() bool {
        fmt.Println(" * 2st function * ");
        return true;
    }
```

```
func myfunction3() bool {
        fmt.Println(" * 3st function * ");
        return false;
    }
fmt.Println(myfunction1() && myfunction2() && myfunction3() );
fmt.Println(myfunction1() || myfunction2() || myfunction3() ); Output:
* 1st function * 0
* 1st function * * 2nd function * 1
```

First line of output is the example of short-circuit of and operator. Because myfunction1() returns false value, there is no need to check rest of the statement. So only myfunction1() is called.

Second line of output is the example of short-circuit of or operator. For this short circuit, statement first calls the myfunction1(). Since it returns false and it is an or operator, program needs to check until it finds true value for or operator. When it faces with true thanks to myfunction2(), it stops executing this statement which means short-circuit is occurred.

- **Evaluation Of GO**

Go language is more writable and readable language with respect to boolean operators because in c bool values are represented by using integers values. Unlike C, Go uses another data type for this which increase both writability and readability. Also since Go uses completely different data type for logical values, it increases reliability.

- **Resources that I used for GO**

For Go language, I used an online compiler(https://play.golang.org/). It was one of the two languages(other one is Rust) I do not know in this homework. First thing, In order to become familiar with syntax, I look up documentation of Go. I specially viewed function, conditional statements, boolean expressions with examples(https://golangbyexample.com/all-data-types-in-golang-with-examples/#Booleans). After viewing nearly one hour, I started to write homework.

3) JAVASCRIPT

- **Boolean Operators Provided - Data Types for Operands of Boolean Operators**

In javascript, the boolean operators are &&, ||, !. These operators can be used with integers. Javascript uses both boolean and integer values to represent the logical values.

Ex.

```
var bool1 = true;
var bool2 = true;
var bool3 = false;
var bool4 = false;
var num1 = 0;
var num2 = 2;
var num3 = 1;
console.log( !!num1);
console.log(!!num3);
console.log((bool1 && bool3));
console.log((bool3 || bool4));
```

```
Output:
false
true
false
false
```

- **Operator Precedence Rules**

In javascript, operator precedence rules is as follow in descending order, it is same with c;

1 —> ()

2 —> !

3 —> &&

4 —> ||

Ex.

console.log((true & true & !(true || true) & false))

```
Output:
false

In this example, first, the parentheses are calculated.
(num2 || !num2)
In the parentheses, not operator is involving. Then or operator is
done. After finishing inside of the parentheses, not operator
follows it. Then if there is an and operator, and operator is
calculated before the or operator. Lastly, or operator is done.
Therefore, result is false which is 0.
```

- **Operator Associativity Rules**

  And, or operators are left associativity. Parentheses are non associative.

Ex.

console.log((3 < 4) && (5 == 4) && (7 > 2))

```
Output:
0

In this example, because and operator is left associativity,
statement is calculated in this form;
((3<4) && (5==4)) &&  (7>2)

But if it was right associativity, then equations would be
calculated as follow;
(3<4) && ((5==4) &&  (7>2))
```

- **Operan Evaluation Order**

  In Javascript, operant evaluation order is left to right as It is in C.

Ex.

```
function myfunction1() {
        console.log(“ * 1st function * ");
        return false;
     }
function myfunction2() {
        console.log(" * 2st function * ");
        return true;
     }
function myfunction3() {
        console.log(" * 3st function * ");
        return false;
     }
console.log(myfunction1() && myfunction2() && myfunction3() );
console.log(myfunction1() || myfunction2() || myfunction3() );
```

```
Output:
* 1st function * 0
* 1st function * * 2nd function * 1

In this example, both examples shows that myfunction1() is
calculated before the myfunction2(). Therefore, this shows that c
has calculation order of left to right for and/or operators.
```

- **Short-Circuit Evaluation**

  In Javascript, short circuit is used for and , or operators as It is in C.

```
function myfunction1() {
        console.log(“ * 1st function * ");
        return false;
     }
function myfunction2() {
        console.log(" * 2st function * ");
        return true;
```

```
    }
function myfunction3() {
      console.log(" * 3st function * ");
      return false;
    }
console.log(myfunction1() && myfunction2() && myfunction3() );
console.log(myfunction1() || myfunction2() || myfunction3() );
```

```
Output:
* 1st function * 0
* 1st function * * 2nd function * 1

This part is same with c part because both the languages has same
short circuit.

First line of output is the example of short-circuit of and
operator. Because myfunction1() returns false value, there is no
need to check rest of the statement. So only myfunction1() is
called.

Second line of output is the example of short-circuit of or
operator. For this short circuit, statement first calls the
myfunction1(). Since it returns false and it is an or operator,
program needs to check until it finds true value for or operator.
When it faces with true thanks to myfunction2(), it stops
executing this statement which means short-circuit is occurred.
```

- **Evaluation Of JavaScript**

Javascipts, follows the boolean operators trends which is created by C.(As I said C is ancestor of most of the languages). Unlike C, javascript uses both boolean and integer values to represent and calculate boolean equations. This increases writability and readability. However using both integer and boolean values can causes misleadings or decrement in the reliability of language.

- **Resources that I used for JavaScript**

For javascript language, I used html document therefore, I run it on the console of browser. My daily life, I always write javascript for backend tech(nodejs). Therefore, Like c, there is no learning process for this homework.

4) PHP

- **Boolean Operators Provided - Data Types for Operands of Boolean Operators**

In Php, the boolean operators are &, |, !, “or”, “and”. Php uses both boolean values and integer for boolean operators just like Javascript does.

Ex.

```
$bool1 = false;
$bool2 = true;
$bool3 = true;
$num1 = 1;
echo ($bool1 & $bool2), "\n";
echo ($bool1 or $bool2), "\n";
echo (!$bool1 and $num1), "\n";
```

```
Output:
0
1
1
```

- **Operator Precedence Rules**

In php, operator precedence rules is as follow in descending order;

1 —> ()

2 —> !

3 —> &&

4 —> ||

Ex.

```
$bool1 = false;
$bool2 = true;
echo $bool2 && ! $bool2 && !( $bool2 ||  ! $bool2) && bool1 ;
```

```
Output:
0
```

In this example, first, the parentheses are calculated. ($bool2 || ! $bool2) like happens in other languages.
In the parentheses, not operator is involving. Then or operator is done. After finishing inside of the parentheses, not operator follows it. Then if there is an and operator, and operator is calculated before the or operator. Lastly, or operator is done. Therefore, result is false which is 0.

- **Operator Associativity Rules**

  And, or operators are left associativity.

Ex.

```
$bool1 = false;
$bool2 = true;
echo $bool2 && ! $bool2 && !( $bool2 ||  ! $bool2) && bool1 ;
```

```
Output:
0
```

After finishing inside of the parentheses, equation becomes, bool2 && !bool2 && resultOfInsideOfParantheses && bool1

Since in php and, or operators are left associativity, equation calculates like as;

((bool2 && !bool2) && resultOfInsideOfParantheses) && bool1

If it was right associativity, then equation would be calculated like as;
bool2 && (!bool2 && (resultOfInsideOfParantheses && bool1))

- **Operan Evaluation Order**

  In php, operant evaluation order is left to right.

Ex.

```
function myfunction1() {
    echo(" * 1st function * ");
    return false;
  }
function myfunction2() {
    echo(" * 2st function * ");
    return true;
  }
function myfunction3() {
    echo(" * 3st function * ");
    return false;
  }
echo(myfunction1() && myfunction2() && myfunction3() );
echo(myfunction1() || myfunction2() || myfunction3() );
```

Output:
* 1st function * * 2nd function * * 3rd function * 0
* 1st function * * 2nd function * * 3rd function * 1

In this example, both examples shows that myfunction1() is calculated before the myfunction2(). Therefore, this shows that c has calculation order of left to right.

- **Short-Circuit Evaluation**

  In php, short circuit is not used for and , or operators.

Ex.

```
function myfunction1() {
        echo(" * 1st function * ");
        return false;
      }
function myfunction2() {
        echo(" * 2st function * ");
        return true;
      }
function myfunction3() {
        echo(" * 3st function * ");
        return false;
      }
echo(myfunction1() && myfunction2() && myfunction3() );
echo(myfunction1() || myfunction2() || myfunction3() );
```

```
Output:
* 1st function * * 2nd function * * 3rd function * 0
* 1st function * * 2nd function * * 3rd function * 1
     As it can be seen in the outputs, even php faces with false
in and operator or true in the or operator, it continues to run
whole statement. Therefore, short-circuit is not used in php.
```

- **Evaluation Of PHP**

As javascript has, php also has two distinct data type to represent boolean values. This decreases the reliability. However, increment in alternative usages of boolean values, increase the writability. Using integers for Boolean values can decrease the readability.

- **Resources that I used for PHP**

For Php language, I used an online compiler(https://paiza.io/projects/81_hUG9EcERzdMng97YySA). I'm mostly familiar with syntax and rules for php,

but I used the site(http://php.adamharvey.name/manual/en/language.operators.precedence.php) for some informations.

5) PYTHON

- **Boolean Operators Provided - Data Types for Operands of Boolean Operators**

In python, the boolean operators are and, or, not. These operators can be used with only boolean values.

Ex.

```
bool1 = false;
bool2 = true;
print(bool1 and bool2);
print(bool1 or bool2);
print(not bool1 or bool2);
```

```
Output:
False
True
True
```

- **Operator Precedence Rules**

In python, operator precedence rules is as follow in descending order;

1 —> ()

2 —> !

3 —> &&

4 —> ||

Ex.

```
bool1 = false;
bool2 = true;
print( bool2 or bool2 and not (bool2 or bool2) and bool1);
```

```
Output:
```

```
true

In this example, first, the parentheses are calculated.
(bool2 or bool2). After finishing inside of the parentheses, not
operator follows it. Then if there is an and operator, and
operator is calculated before the or operator. Lastly, or operator
is done. Therefore, result is false which is true.
```

- **Operator Associativity Rules**

  And, or operators are left associativity.

Ex.

```
bool1 = false;
bool2 = true;
bool3 = false;
print( bool2 and bool3  and bool1);
Output:
0

In this example, because and operator is left associativity,
statement is calculated in this form;
(bool2 and bool3) and bool1

But if it was right associativity, then equations would be
calculated as follow;
bool2 and (bool3 and bool1)
```

- **Operan Evaluation Order**

  In python, operant evaluation order is left to right.

Ex.

```
def myfunction1():
    print("1. function")
    return False
```

```
def myfunction2():
    print("2. function")
    return True

def myfunction3():
    print("3. function")
    return False

print(myfunction1() and myfunction2() and myfunction3() );
print(myfunction1() or myfunction2()  or myfunction3() );
```

```
Output:
* 1st function * 0
*  1st function * * 2nd function * 1

In this example, both examples shows that myfunction1() is
calculated before the myfunction2(). Therefore, this shows that c
has calculation order of left to right.
```

- **Short-Circuit Evaluation**

  In python, short circuit is used for and , or operators.

```
def myfunction1():
    print("1. function")
    return False

def myfunction2():
    print("2. function")
    return True

def myfunction3():
    print("3. function")
    return False

print(myfunction1() and myfunction2() and myfunction3() );
```

print(myfunction1() or myfunction2() or myfunction3());

```
Output:
* 1st function * 0
* 1st function * * 2nd function * 1

First line of output is the example of short-circuit of and
operator. Because myfunction1() returns false value, there is no
need to check rest of the statement. So only myfunction1() is
called.

Second line of output is the example of short-circuit of or
operator. For this short circuit, statement first calls the
myfunction1(). Since it returns false and it is an or operator,
program needs to check until it finds true value for or operator.
When it faces with true thanks to myfunction2(), it stops
executing this statement which means short-circuit is occurred.
This part is same with the languages which have short-circuit.
```

- **Evaluation Of C**

In pyhton, logical operators are use with boolean data types. Unlike C and some languages, python do not use integer values. This might can decrease writability but increase readability. Also operator syntax is in English, this increases the writability. Lastly, because there is one type for boolean operation this increases the reliability.

- **Resources that I used for C**

For C language, I used an vs code. I’m familiar with syntax, therefore I do not need any sources.

6) RUST

- **Boolean Operators Provided - Data Types for Operands of Boolean Operators**

In Rust, the boolean operators are &&, ||, !. These operators can be used with Boolean values.

Ex.

```
let _bool1 = false;
let _bool2 = true;

println!("false and true =  {} \n" , _bool1 && _bool2);
println!("false or true =  {} \n" ,  _bool1 || _bool2);
println!("not false and true =  {} \n" ,  !_bool1 && _bool2);
```

```
Output:
False
True
True
```

- **Operator Precedence Rules**

In c, operator precedence rules is as follow in descending order;

1 —> ()

2 —> !

3 —> &&

4 —> ||

Ex.

```
let _bool1 = false;
let _bool2 = true;
println!("{} " , _bool2 && ! _bool2 && !( _bool2 ||  ! _bool2) && _bool1);
```

```
Output:
```

```
0

In this example, first, the parentheses are calculated.
(_bool2 || !_bool2)
In the parentheses, not operator is involving. Then or operator is
done. After finishing inside of the parentheses, not operator
follows it. Then if there is an and operator, and operator is
calculated before the or operator. Lastly, or operator is done.
Therefore, result is false which is 0.
```

- **Operator Associativity Rules**

  And, or operators are left associativity.

Ex.

```
let _bool1 = false;
let _bool2 = true;
println!("{} \n" , _bool2  &&  ! _bool2 && _bool1 );
Output:
0

In this example, because and operator is left associativity,
statement is calculated in this form;
(_bool2 && !_bool2) && _bool1

But if it was right associativity, then equations would be
calculated as follow;
_bool2 && (!_bool2 && _bool1)
```

- **Operan Evaluation Order**

  In Rust, operant evaluation order is left to right.

Ex.

```
fn myfunction1 ()-> bool {
   println!(" * 1st function * ");
   return false;
}
```

```
fn myfunction2 ()-> bool {
    println!(" * 2nd function * ");
    return true;
}

fn myfunction3 ()-> bool {
    println!(" * 3rd function * ");
    return false;
}

println!"{} \n" , myfunction1() && myfunction2() && myfunction3() );
println!("{} \n" , myfunction1() || myfunction2() || myfunction3() );
```

```
Output:
* 1st function * 0
* 1st function * * 2nd function * 1

In this example, both examples shows that myfunction1() is
calculated before the myfunction2(). Therefore, this shows that c
has calculation order of left to right.
```

- **Short-Circuit Evaluation**

    In Rust, short circuit is used for and , or operators.

```
fn myfunction1 ()-> bool {
    println!(" * 1st function * ");
    return false;
}
fn myfunction2 ()-> bool {
    println!(" * 2nd function * ");
    return true;
}

fn myfunction3 ()-> bool {
    println!(" * 3rd function * ");
```

```
    return false;
}

println!"{} \n" , myfunction1() && myfunction2() && myfunction3() );
println!("{} \n" , myfunction1() || myfunction2() || myfunction3() );

Output: (same with other languages)
* 1st function * 0
*  1st function * * 2nd function * 1

First line of output is the example of short-circuit of and
operator. Because myfunction1() returns false value, there is no
need to check rest of the statement. So only myfunction1() is
called.

Second line of output is the example of short-circuit of or
operator. For this short circuit, statement first calls the
myfunction1(). Since it returns false and it is an or operator,
program needs to check until it finds true value for or operator.
When it faces with true thanks to myfunction2(), it stops
executing this statement which means short-circuit is occurred.
```

- **Evaluation Of C**

In rust, logical operators are represented by using booleans which increase the readability, However, unlike c, integers are not used therefore, it decreases writability. In addition, just using boolean values might increase reliability.

- **Resources that I used for C**

For Rust language, I used an online compiler(https://play.rust-lang.org/). In order to be familiar with syntax, I used the documentation of site of Rust(https://doc.rust-lang.org/reference/introduction.html).

7) OVERALL COMMENT

Each language use nearly same approaches for the boolean expression. Just one of them(php) do not use short-circuit which decrease the efficiency. Some languages uses the both integers and booleans to represent to logical values. This gives advantage in terms of the writability but decreases the readability. It is kind a trade of. Therefore, we can not exactly say that this language is best. However, we can talk this in terms of readability and writability. Javascript is one of the best languages in terms of the writability because both intergers and booleans are can be used. In terms of readability, python can be best option because it uses only boolean values and its syntax is English.