

# OCaml

## Zadanie 1.

```
Welcome to utop version 2.13.1 (using OCaml version 4.14.1):

#indlib has been successfully loaded. Additional directives:
#require "package";;      to load a package
#list;;                  to list the available packages
#camlp4oz;;              to load camlp4 (standard syntax)
#camlp4r;;              to load camlp4 (revised syntax)
#predicates "p,q,...";;  to set these predicates
#topfind.reset();;       to force that packages will be reloaded
#thread;;               to enable threads

Type #utop_help for help about using utop.

- : (20:02:34) -> <command 0> - [ counter: 0 ] -
utop # 2+2;;
- : int = 4
- : (20:02:34) -> <command 1> - [ counter: 0 ] -
utop # 8-17;;
- : int = -9
- : (20:02:44) -> <command 2> - [ counter: 0 ] -
utop # 4.0 *. 3.5;;
- : float = 14.
- : (20:02:50) -> <command 3> - [ counter: 0 ] -
utop # 4 / 6.
- : float = 0.6666666666666667
- : (20:03:05) -> <command 4> - [ counter: 0 ] -
utop #

[UI_instrument] Alias_analysis Allocated_const Annot Arch Arg Arg_helper Array ArrayLabels Asngen Asmlibarian Asmlink Asmpackager Assert_failure Ast_helper Ast_invariants Ast_iterator Ast_mapper Asttypes Atomic Attr_helper Augment_speci
```

Jak widać na powyższym rzucie ekranu – w REPLu OCaml'a, a dokładniej utopie Wystarczy wpisać interesujące nas działanie matematyczne, a następnie zakończyć komendę ;; Istotnym faktem jest, że musimy dostosować typ operacji do naszych danych.

## Zadanie 2.

```
- : float = 8.5
- : (20:10:04) -> <command 8> - [ counter: 0 ] -
utop # let immutable_variable = 16;;
- : immutable_variable = int = 16
- : (20:10:21) -> <command 9> - [ counter: 0 ] -
utop # immutable_variable = 15;;
- : bool = false
- : (20:10:52) -> <command 10> - [ counter: 0 ] -
utop # immutable_variable := 15;;
Error: This expression has type int but an expression was expected of type
'a ref
- : (20:11:08) -> <command 11> - [ counter: 0 ] -
utop # immutable_variable := ref 15;;
Error: This expression has type int but an expression was expected of type
'a ref
- : (20:11:14) -> <command 12> - [ counter: 0 ] -
utop # immutable_variable := ref 15;;
Error: This expression has type int but an expression was expected of type
'a ref
- : (20:11:21) -> <command 13> - [ counter: 0 ] -
utop # let mutable_variable = ref 1;;
- : mutable_variable = int ref = (contents = 1)
- : (20:13:23) -> <command 14> - [ counter: 0 ] -
utop # mutable_variable.;
- : int ref = (contents = 1)
- : (20:13:35) -> <command 15> - [ counter: 0 ] -
utop # mutable_variable := 5;;
- : unit = ()
- : (20:13:44) -> <command 16> - [ counter: 0 ] -
utop # mutable_variable.;
- : int ref = (contents = 5)
- : (20:13:56) -> <command 17> - [ counter: 0 ] -
utop #

[UI_instrument] Alias_analysis Allocated_const Annot Arch Arg Arg_helper Array ArrayLabels Asngen Asmlibarian Asmlink Asmpackager Assert_failure Ast_helper Ast_invariants Ast_iterator Ast_mapper Asttypes Atomic Attr_helper Augment_speci
```

W tym zadaniu definiujemy zmienne mutowalne i niemutowalne. Jak widać, aby zdefiniować zmienną niemutowalną – wystarczy ją zdefiniować: let x = 15;;

W przypadku zmiennych mutowalnych potrzebujemy trochę więcej gimnastyki, a mianowicie definiujemy je: let x := ref 14;;

Widzimy też oczywiście, że zmiennej niemutowalnej rzeczywiście nie da się zmienić, natomiast zmienna mutowalna bez problemu może zostać zmodyfikowana.

## Zadanie 3.

```
-( 20:13:56 )-< command 17 >
utop # let square = fun x -> x*x;;
val square : int -> int = <fun>
-( 20:14:04 )-< command 18 >
utop # square 2;;
- : int = 4
-( 20:27:54 )-< command 19 >
utop # square 5;;
- : int = 25
-( 20:28:09 )-< command 20 >
utop # square 10;;
- : int = 100
-( 20:28:13 )-< command 21 >
utop # square 25;;
- : int = 625
```

```
Hint: Did you mean `5.'?
-( 20:42:00 )-< command 32 >
utop # let ratio = fun x y -> x/.y;;
val ratio : float -> float -> float = <fun>
-( 20:42:04 )-< command 33 >
utop # ratio 5. 2.;;
- : float = 2.5
-( 20:42:11 )-< command 34 >
utop # ratio 5. 10.;;
- : float = 0.5
-( 20:42:17 )-< command 35 >
utop #
```

Afl_instrument	Alias_analysis	Allocated_const	Annot	Arch	Arg	Arg_helper	Arr
----------------	----------------	-----------------	-------	------	-----	------------	-----

```
-( 20:48:40 )-< command 38 >
utop # let max = fun x y -> if x > y then x else y;;
val max : 'a -> 'a -> 'a = <fun>
-( 20:48:47 )-< command 39 >
utop # max 10 5;;
- : int = 10
-( 20:48:52 )-< command 40 >
utop # max 5 10;;
- : int = 10
-( 20:49:00 )-< command 41 >
```

```

-( 22:48:54 )-< command 53 >
utop # let test = fun a b -> a == b;;
val test : 'a -> 'a -> bool = <fun>
-( 22:48:58 )-< command 54 >
utop # let funct = fun test a b c -> if test a b then a+b else if test a c then a+c else if test b c then b+c else 0;;
val funct : (int -> int -> bool) -> int -> int -> int -> int = <fun>
-( 22:49:09 )-< command 55 >
utop # funct test 1 3 1;;
- : int = 2
-( 22:49:13 )-< command 56 >
utop # funct test 13 2 13;;
- : int = 26
-( 22:49:17 )-< command 57 >
utop # let test = fun a b -> a<b;;
val test : 'a -> 'a -> bool = <fun>
-( 22:49:21 )-< command 58 >
utop # funct test 1 3 1;;
- : int = 4
-( 22:53:57 )-< command 59 >
utop # funct test 13 2 13;;
- : int = 15
-( 22:54:06 )-< command 60 >
utop # let test = fun a b -> a<b;;
val test : 'a -> 'a -> bool = <fun>
-( 22:54:14 )-< command 61 >
utop # funct test 1 3 1;;
- : int = 4
-( 23:08:00 )-< command 62 >
utop # funct test 13 2 13;;
- : int = 15
-( 23:08:07 )-< command 63 >
utop # funct test 2 2 2;;
- : int = 0
-( 23:08:10 )-< command 64 >
utop # _

```

Afl_instrument	Alias_analysis	Allocated_const	Annot	Arch	Arg	Arg_helper	Array	ArrayLabels	Asmgen	Asmlibrarian	Asmlink	Asmpackager	Assert_failure	Ast_helper	Ast_
----------------	----------------	-----------------	-------	------	-----	------------	-------	-------------	--------	--------------	---------	-------------	----------------	------------	------

Na powyższych zrzutach ekranu zajmujemy się następującymi działaniami:

- definiujemy proste funkcje, ogólna składnia jest następująca:

let name = parameter1, parameter2 → return;;

- definiujemy też funkcję w której parametrze znajduje się inna funkcja, schemat jest podobny jak w przypadku zwykłych parametrów

```

-( 01:46:46 )-< command 89 >
utop # let friends = ["Adam"; "Maciej"; "Kamil"; "Martyna"];;
val friends : string list = ["Adam"; "Maciej"; "Kamil"; "Martyna"]
-( 01:47:03 )-< command 90 >
utop # let addName = fun friends name -> name :: friends;;
val addName : 'a list -> 'a -> 'a list = <fun>
-( 01:48:03 )-< command 91 >
utop # addName friends "Bartek";;
- : string list = ["Bartek"; "Adam"; "Maciej"; "Kamil"; "Martyna"]
-( 01:48:24 )-< command 92 >
utop # friends;;
- : string list = ["Adam"; "Maciej"; "Kamil"; "Martyna"]
-( 01:49:01 )-< command 93 >
utop # let newFriends = addName friends "Bartek";;
val newFriends : string list = ["Bartek"; "Adam"; "Maciej"; "Kamil"; "Martyna"]
-( 01:49:08 )-< command 94 >
utop # friends;;
- : string list = ["Adam"; "Maciej"; "Kamil"; "Martyna"]
-( 01:49:38 )-< command 95 >
utop # newFriends;;
- : string list = ["Bartek"; "Adam"; "Maciej"; "Kamil"; "Martyna"]
-( 01:49:44 )-< command 96 >
utop #

```

Afl_instrument	Alias_analysis	Allocated_const	Annot	Arch	Arg	Arg_helper	Array	ArrayLabels	Asmgen	Asmlibrarian	Asmlink	Asmpackager	Assert_failure	Ast_helper	Ast_
----------------	----------------	-----------------	-------	------	-----	------------	-------	-------------	--------	--------------	---------	-------------	----------------	------------	------

Jest to ostatnie zadanie w języku OCaml, w którym definiujemy tablice:

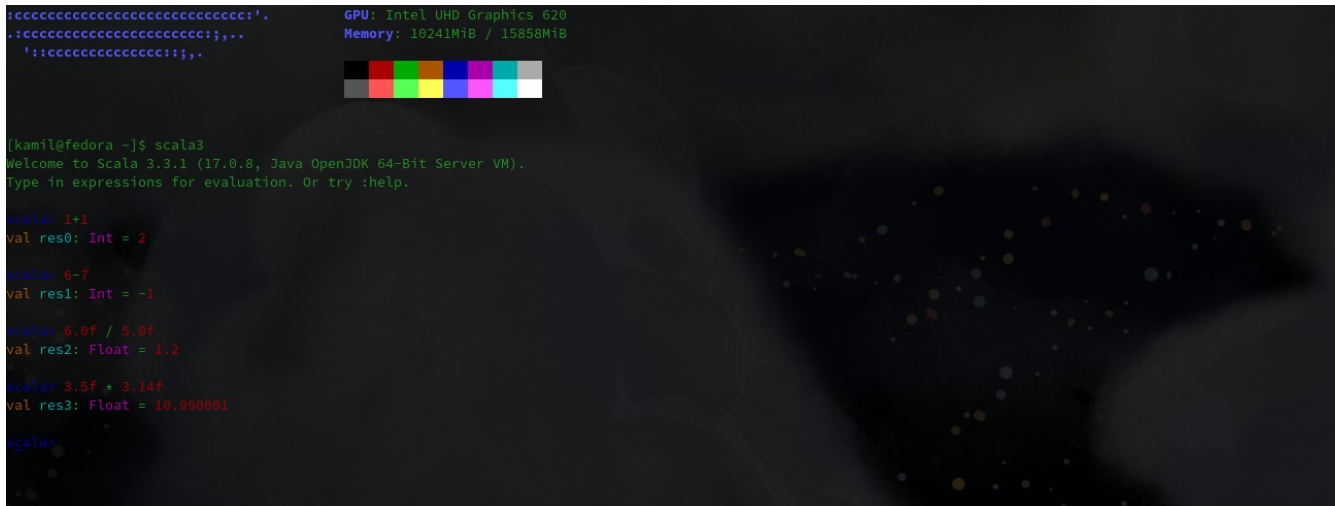
let table = [v1;v2;v3;...];;

Następnie definiujemy funkcję, która ma na celu zwrócić tablicę z jednym elementem więcej, robimy to poprzez komendę:

```
let newTable = v1 :: oldTable;;
```

## SCALA3

### Zadanie 1.



```
GPU: Intel UHD Graphics 620
Memory: 1024MiB / 15858MiB

[kamil@fedora ~]$ scala3
Welcome to Scala 3.3.1 (17.0.8, Java OpenJDK 64-Bit Server VM).
Type in expressions for evaluation. Or try :help.

scala> 1+1
val res0: Int = 2

scala> 6-7
val res1: Int = -1

scala> 6.0f / 5.0f
val res2: Float = 1.2

scala> 3.5f * 3.14f
val res3: Float = 10.990001

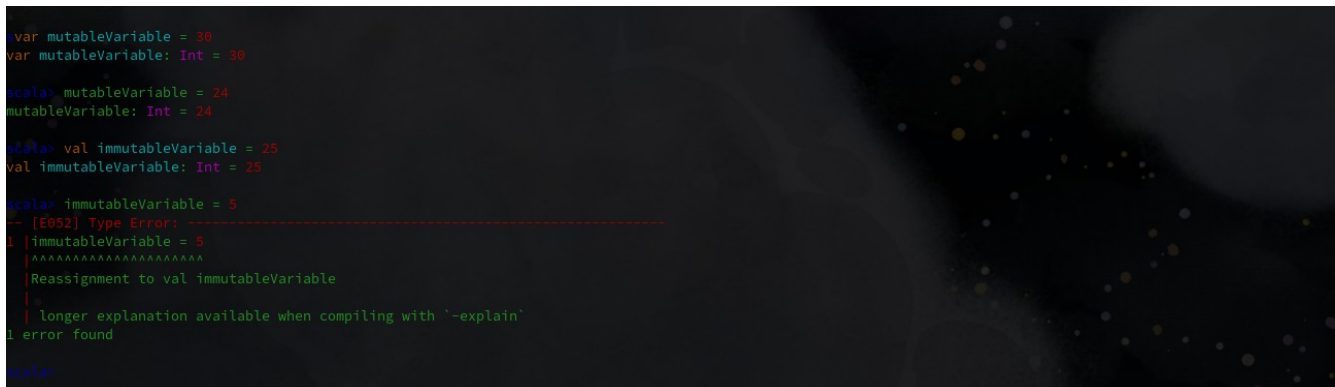
scala>
```

W pierwszym zadaniu tak samo jak w przypadku OCaml'a, tak jest i w Scali:

Możemy wpisać działania i dostaniemy wynik naszego działania

Tutaj oczywiście składnia wygląda troszkę inaczej: nie musimy pisać ;; żeby zakończyć polecenie, natomiast liczby zmiennoprzecinkowe należy oznaczać literą f. Nie musimy natomiast dodatkowo deklarować typu danych na których wykonujemy operację.

### Zadanie 2.



```
var mutableVariable = 30
var mutableVariable: Int = 30

scala> mutableVariable = 24
mutableVariable: Int = 24

scala> val immutableVariable = 25
val immutableVariable: Int = 25

scala> immutableVariable = 5
-- [E052] Type Error: -----
1 | immutableVariable = 5
  | ^^^^^^^^^^^^^^^^^^^
  | Reassignment to val immutableVariable
  |
  | longer explanation available when compiling with '-explain'
1 error found

scala>
```

Definiujemy mutowalną zmienną używając słowa kluczowego var (variable). Tak zdefiniowane zmienne możemy oczywiście zmieniać. W przypadku drugiego słowa kluczowego val(value) wartość jest niemutowalna, a próby jej zmiany kończą się wygenerowaniem błędu.

### Zadanie 3.

```
scala> def square(num: Int): Int = {num*num}
def square(num: Int): Int

scala> square(2)
val res4: Int = 4

scala> square(8)
val res5: Int = 64

scala> square(-3)
val res6: Int = 9

scala>
```

```
scala> def maxVal(num1: Int, num2: Int): Int = {if(num1<num2){num2}else{num1}}
def maxVal(num1: Int, num2: Int): Int

scala> maxVal(1,2)
val res10: Int = 2

scala> maxVal(5,2)
val res11: Int = 5

scala>
```

```
scala> def numberRatio(num1: Float, num2: Float): Float = {num1/num2}
def numberRatio(num1: Float, num2: Float): Float

scala> numberRatio(3, 5)
val res8: Float = 0.6

scala> numberRatio(10, 5)
val res9: Float = 2.0

scala>
```

Jak widać na powyższym zrzucie ekranu – funkcję definiujemy używając następującego schematu:

```
def name(p1: Type, p2: Type ...): Type = {return}
```

Funkcję wywołujemy wypisując jej nazwę i jej parametry w nawiasie.

Zadanie 4.

```

scala> def test(num1: Int, num2: Int): Boolean = {num1==num2}
def test(num1: Int, num2: Int): Boolean

scala> def funct(a: Int, b: Int, c: Int, operation: (Int, Int) => Boolean): Int = {if(operation(a,b)){a+b}else if(operation(a,c)){a+c}else if(operation(b,c)){b+c}else{0}}
def funct(a: Int, b: Int, c: Int, operation: (Int, Int) => Boolean): Int

scala> funct(1,1,4,test)
val res12: Int = 2

scala> funct(1,5,4,test)
val res13: Int = 0

scala> def test(num1: Int, num2: Int): Boolean = {num1>num2}
def test(num1: Int, num2: Int): Boolean

scala> funct(1,5,4,test)
val res14: Int = 9

scala> funct(9,5,4,test)
val res15: Int = 14

scala> def test(num1: Int, num2: Int): Boolean = {num1!=num2}
def test(num1: Int, num2: Int): Boolean

scala> funct(9,5,4,test)
val res16: Int = 14

scala> funct(9,5,9,test)
val res17: Int = 14

scala> funct(9,9,3,test)
val res18: Int = 12

scala>

```

Funkcję jako parametr przekazujemy stosując następującą składnię:  
 Name: (Arg1, Arg2 ...) => returnType

```

scala> val friends: List[String] = List("Adam", "Maciej", "Martyna")
val friends: List[String] = List(Adam, Maciej, Martyna)

```

```

scala> def addFriend(oldList: List[String], newName: String): List[String] = {newName :: oldList}
def addFriend(oldList: List[String], newName: String): List[String]

scala> friends
val res0: List[String] = List(Adam, Maciej, Martyna)

scala> addFriend(friends, "Bartek")
val res1: List[String] = List(Bartek, Adam, Maciej, Martyna)

scala> friends
val res2: List[String] = List(Adam, Maciej, Martyna)

scala> val newFriends = addFriend(friends, "Bartek")
val newFriends: List[String] = List(Bartek, Adam, Maciej, Martyna)

scala> newFriends
val res3: List[String] = List(Bartek, Adam, Maciej, Martyna)

scala> _

```

Aby zdefiniować tablicę potrzebujemy użyć następującego schematu:

val/var name: List[Type] = List(v1, v2, ...), natomiast później, aby dodać element używamy:

Name :: listName