Laboratorium 2.

0) Przygotowanie środowiska

W konsoli/terminalu wpisujemy kolejno

```
$ cd
$ mkdir haskell-lab2
$ cd haskell-lab2
```

1) Definicja funkcji: *currying*, *partially applied functions*

1. Tworzymy nowy plik `ex1.hs` (np. `$ touch ex1.hs`) i wpisujemy w nim

   ```
   MyFun x = 2 * x
   ```

2. Uruchamiamy GHCi, wczytujemy plik `ex1.hs` (`:l ex1.hs`) i analizujemy komunikat błędu
3. Zmieniamy nazwę funkcji na z `MyFun` na `myFun` , zapisujemy zmiany, wczytujemy plik do GHCi i sprawdzamy działanie `myFun`
4. Dodajemy do pliku `ex1.hs` następujące dwie definicje

   ```
   add2T :: Num a => (a, a) -> a
   add2T (x,y) = x + y

   add2C :: Num a => a -> a -> a
   add2C x y = x + y
   ```

5. Zapisujemy plik i wczytujemy go do GHCi (`:r`)
6. W konsoli GHCi wpisujemy kolejno

   ```
   ghci> add2T (1,2)
   ghci> add2C (1,2)
   ghci> add2C 1 2
   ghci> add2T 1 2

   ghci> let add1Plus_ = add2T 1
   ghci> let add1Plus_ = add2C 1
   ghci> :t add1Plus_
   ghci> let add1Plus2 = add1Plus_ 2
   ghci> :t add1Plus2
   ghci> add1Plus2
   ghci> (add2C 1) 2

   ghci> :t curry
   ghci> :t uncurry

   ghci> :t curry add2T
   ghci> (curry add2T) 1 2
   ghci> curry add2T 1 2

   ghci> :t uncurry add2C
   ghci> (uncurry add2C) (1,2)
   ghci> uncurry add2C (1,2)
   ```

7. ***Zadania***:

   1. Dodać nawiasy w deklaracji

      ```
      add2C :: Num a => a -> a -> a
      ```

      (sprawdzić, czy zmieniony kod się kompiluje :)

      `->` jest prawo- czy lewostronnie łączny?

   2. W pliku `ex1.hs` dodać definicje funkcji

      ```
      add3T :: Num a => (a, a, a) -> a
      add3T ...

      add3C :: Num a => a -> a -> a -> a
      add3C ...
      ```

   3. Dodać nawiasy w deklaracji typu `add3C` i w wywołaniu

      ```
      add3C 1 2 3
      ```

   4. (*opcjonalne*) Napisać i sprawdzić działanie funkcji:

      ```
      curry2 :: ((a, b) -> c) -> a -> b -> c
      curry2 f ...

      uncurry2 :: (a -> b -> c) -> (a, b) -> c
      uncurry2 f ...

      curry3 :: ((a, b, c) -> d) -> a -> b -> c -> d
      curry3 f ...

      uncurry3 :: (a -> b -> c -> d) -> (a, b, c) -> d
      ```

```
uncurry3 f ...
```

2) Definicja funkcji: *sections*

1. W kondoli GHCi spisujemy kolejno:

```
ghci> :t 2 * 3
ghci> :t (*)
ghci> : t *
ghci> :t (2 *)
ghci> :t (* 2)
ghci> (* 2) 3
ghci> (2 *) 3
ghci> (*) 2 3

ghci> :t 2 ^ 3
ghci> :t (^)
ghci> :t ^
ghci> :t (2 ^)
ghci> :t (^ 2)
ghci> 2 ^ 3
ghci> (^ 3) 2
ghci> (2 ^) 3
ghci> (^) 2 3

ghci> let twoToPower_ = 2 ^
ghci> let twoToPower_ = (2 ^)
ghci> twoToPower_ 3
```

2. ***Zadania***:

    1. Zdefiniować i sprawdzić działanie funkcji:

```
fiveToPower_ :: Integer -> Integer
fiveToPower_ = ... -- fiveToPower_ 3 = 125

_ToPower5 :: Num a => a -> a
_ToPower5 = ... -- _ToPower5 2 = 32

subtrNFrom5 :: Num a => a -> a
subtrNFrom5 = ... -- subtrNFrom5 3 = 2

subtr5From_ :: Num a => a -> a
subtr5From_ = ... -- subtr5From_ 6 = 1
```

    2. (*opcjonalne*) Zdefiniować funkcję `flip2` będącą odpowiednikiem funkcji `flip` z biblioteki standardowej (`Prelude`)
    3. (*opcjonalne*) Zdefiniować funkcję `flip3` będącą trójparametrowym (tzn. funkcja 3 zmiennych) odpowiednikiem funkcji `flip2`

3) Leniwe obliczanie/wartościowanie

1. W konsoli GHCi wpisujemy:

```
ghci> let f x y = if (x > 0) then 42 else x + y
ghci> :t f
ghci> f 0 (1/0)
ghci> f 1 (1/0)

ghci> let neverEndingStory x = neverEndingStory (x + 1) `mod` 100
ghci> neverEndingStory 1 -- po kilku sekundach naciskamy [Ctrl-C]
ghci> f 0 (neverEndingStory 1) -- po kilku sekundach naciskamy [Ctrl-C]
ghci> f 1 (neverEndingStory 1)
```

2. (*opcjonalne*) W konsoli GHCi wpisujemy kolejno:

```
ghci> import Data.Tuple
ghci> let x = 1 + 2 :: Int
ghci> let z = swap (x, x+1)
ghci> :sp z
ghci> seq z ()
ghci> :sp z
ghci> seq x ()
ghci> :sp z
ghci> seq (fst z) ()
ghci> :sp z

ghci> let ys = [1..5] :: [Int]
ghci> :sp ys
ghci> seq (length ys) ()
ghci> :sp ys
ghci> let xs = [1..] :: [Int]
ghci> :sp xs
ghci> head xs
ghci> :sp xs
ghci> xs !! 2
ghci> :sp xs

ghci> let fibs = 0 : 1 : zipWith (+) fibs (tail fibs) :: [Int]
ghci> :sp fibs
ghci> head fibs
```

4) Elementarne operacje na listach

1. W konsoli GHCi wpisujemy kolejno

```
ghci> import Data.List
ghci> let xs = 1 : 2 : 3 : 4 : 5 : []
ghci> xs
ghci> let xs = [1..5]
ghci> xs
ghci> length xs
ghci> reverse xs

ghci> head xs
ghci> tail xs
ghci> last xs
ghci> init xs

ghci> 0 : xs
ghci> xs ++ [6]
ghci> xs !! 2
ghci> [1,2] ++ [3,4,5]

ghci> take 2 xs
ghci> drop 2 xs

ghci> null xs
ghci> any (> 2) xs
ghci> all (> 0) xs

ghci> zip xs ['a','b']
ghci> splitAt 2 xs
ghci> sort [2,3,1,4,5]
ghci> 2 `elem` xs

ghci> minimum xs
ghci> maximum xs
ghci> sum xs
ghci> product xs
```

2. ***Zadania***:
    1. W pliku `ex4.hs` dodać definicję funkcji

    ```
    isPalindrome :: [Char] -> Bool
    isPalindrome s ... -- isPalindrome "ABBA" = True
    ```

    i sprawdzić jej działanie

    2. (*opcjonalne*) Zdefiniować funkcję `getElemAtIdx` wykorzystując funkcje poznane powyżej (np. `drop` , `head` , `tail`)
    3. (*opcjonalne*) Zdefiniować funkcję

    ```
    capitalize :: [Char] -> [Char]
    capitalize w = ... -- capitalize "ala" = "Ala"
    ```

5) List comprehensions

1. W konsoli GHCi wpisujemy kolejno:

```
ghci> [x^3 | x <- [1..5]]
ghci> let x1_5Cubed = [x^3 | x <- [1..5]]
ghci> :t x1_5Cubed
ghci> [x^2 | x <- [1..5], x ^ 2 - x > 3]
ghci> [(i,j) | i <- [0..3], j <- [0..2]]
ghci> [(i,j) | i <- [0..3], j <- [0..i]]
ghci> [(i+j)^2 | i <- [0..3], j <- [0..2]]
ghci> [i+j | i <- [0..4], j <- [i..4], (i + j) `mod` 3 == 0]
ghci> [(a,b,c) | a <- [1..10], b <- [a..10], c <- [b..10], a ^ 2 + b
^ 2 == c ^ 2]
```

2. ***Zadania***:
    1. Napisać wyrażenie obliczające, ile jest w przedziale `[1,100]` trójek liczb całkowitych reprezentujących długości boków trójkąta prostokątnego
    2. Czy poniższa definicja funkcji (sprawdzającej, czy liczba jest pierwsza) jest poprawna?

    ```
    isPrime :: Integral t => t -> Bool
    isPrime n = [i | i <- [2..n-1], n `mod` i == 0] == []
    ```

    Czy jest to efektywna implementacja? :)
    3. (*opcjonalne*) Napisać wyrażenie obliczające, ile jest w przedziale `[1,10000]` liczb pierwszych
    4. (*opcjonalne*) Napisać nową definicję funkcji `isPrime` wykorzystując ciąg `primes`

    ```
    primes :: [Int]
    primes = eratoSieve [2..]
     where
       eratoSieve :: [Int] -> [Int]
       eratoSieve (p : xs) = p : eratoSieve [x | x <- xs, x `mod` p
    /= 0]
    ```

    Porównać wydajność obu wersji funkcji `isPrime`
    5. (*opcjonalne*) Wykorzystując wynik poprzedniego ćwiczenia napisać funkcję, która

podaje, ile jest liczb pierwszych w przedziale `[1,n]` , gdzie `n` jest parametrem funkcji

6. (*opcjonalne*) Napisać definicję funkcji

```
allEqual :: Eq a => [a] -> Bool
allEqual xs ... -- allEqual [1,1] = True, allEqual [1,2] = False
```

6) Rekursja 1

1. Tworzymy plik `ex6.hs` , dodajemy w nim poniższą definicję funkcji `fib`

```
fib :: (Num a, Eq a) => a -> a
fib n =
 if n == 0 || n == 1 then n
 else fib (n - 2) + fib (n - 1)
```

(zapisujemy zmiany :)

2. W konsoli GHCi wpisujemy

```
ghci> :set +s
```

3. Wczytujemy plik do GHCi (`:l ex6.hs`) i obliczamy wartości funkcji `fib` kolejno dla `5,10, ... ,35,40` (?) Jaka jest złożoność czasowa powyższej implementacji?
4. ***Zadania***:
    1. (*opcjonalne*) W pliku `ex6.hs` dodać funkcję `fib2` (`fib2 5 = 5`) na podstawie poniższej (korekursywnej) definicji ciągu Fibonacciego

```
fibs = 0 : 1 : zipWith (+) fibs (tail fibs) :: [Int]
```

    2. (*opcjonalne*) Obliczyć wartości funkcji `fib2` kolejno dla `10,20,...,100` . Jaka jest złożoność czasowa funkcji `fib2` ?
5. W pliku `ex6.hs` definiujemy funkcję

```
sum' :: Num a => [a] -> a
sum' []   = 0
sum' x:xs = x + sum' xs
```

zapisujemy zmiany, wczytujemy plik do GHCi i analizujemy komunikat błędu

6. Modyfikujemy definicję `sum'` (dodanie nawiasów)

```
sum' :: Num a => [a] -> a
sum' []     = 0
sum' (x:xs) = x + sum' xs
```

zapisujemy zmiany, wczytujemy plik do GHCi i sprawdzamy, czy funkcja działa poprawnie

7. ***Zadania***:
    1. Napisać rekurencyjne definicje funkcji

```
prod' :: Num a => [a] -> a -- prod' [1,2,3] = 6
length' :: [a] -> Int -- length' [1,1,1,1] = 4
or' :: [Bool] -> Bool -- or' [True, False, True] = True
and' :: [Bool] -> Bool -- and' [True, False, True] = False
elem' :: Eq a => a -> [a] -> Bool -- elem' 3 [1,2,3] = True
doubleAll :: Num t => [t] -> [t] -- doubleAll [1,2] = [2,4]
squareAll :: Num t => [t] -> [t] -- squareAll [2,3] = [4,9]
selectEven :: Integral t => [t] -> [t] -- selectEven [1,2,3] = [
2]
```

    Czy można wskazać w implementacjach powtarzające się schematy?
    2. (*opcjonalne*) Wyznaczyć dla funkcji `sum'` rozmiar listy `[1..N]` , przy którym pojawia się przepełnienie stosu
    3. (*opcjonalne*) Napisać funkcję obliczającą średnią arytmetyczną elementów listy
    4. (*opcjonalne*) Napisać funkcję obliczającą średnią geometryczną elementów listy. Czy można te dwie funkcje połączyć i zwracać obie średnie jako parę?

7) Rekursja 2: użycie *akumulatora*, rekursja *końcowa*

1. W pliku `ex6.hs` dodajemy funkcję

```
sum'2 :: Num a => [a] -> a
sum'2 xs = loop 0 xs
 where loop acc []     = acc
       loop acc (x:xs) = loop (x + acc) xs
```

zapisujemy zmiany, wczytujemy plik do GHCi i sprawdzamy, czy funkcja działa poprawnie. Co zmieniło się w schemacie rekursji?

2. W pliku `ex6.hs` dodajemy funkcję

```
sum'3 :: Num a => [a] -> a
sum'3 = loop 0
 where loop acc []     = acc
       loop acc (x:xs) = loop (x + acc) xs
```

zapisujemy zmiany, wczytujemy plik do GHCi i sprawdzamy, czy funkcja działa poprawnie. Co zmieniło się w definicji funkcji (względem `sum'2` ?)

3. ***Zadania***:
    1. Napisać definicje funkcji

```
prod'2 :: Num a => [a] -> a
length'2 :: [a] -> Int
```

wykorzystujące ‘akumulator’ i funkcję pomocniczą/wewnętrzną (analogicznie do `sum' -> sum'3`)

2. (*opcjonalne*) Wyznaczyć dla funkcji `sum'3` rozmiar listy `[1..N]` , przy którym pojawia się przepełnienie stosu
3. (*opcjonalne*) Dodać w pliku `ex6.hs` nową wersję funkcji `sum`

```
sum'4 :: Num a => [a] -> a
sum'4 = loop 0
   where loop !acc []     = acc
         loop !acc (x:xs) = loop (x + acc) xs
```

Uwaga: na początku pliku `ex6.hs` trzeba dodać poniższą dyrektywę

```
{-# LANGUAGE BangPatterns #-}
```

4. (*opcjonalne*) Porównać czasy wykonania funkcji `sum'` , `sum'3` i `sum'4` dla różnych rozmiarów list. Wyjaśnić wyniki pomiarów

8) Rekursja 3: rekursja pośrednia i zagnieżdżona (*ćwiczenie opcjonalne*)

1. W pliku `ex8.hs` dodajemy następujące funkcje

```
isOdd :: (Ord a, Num a) => a -> Bool
isOdd n | n <= 0    = False
        | otherwise = isEven (n-1)

isEven :: (Ord a, Num a) => a -> Bool
isEven n | n < 0     = False
         | n == 0    = True
         | otherwise = isOdd (n-1)
```

zapisujemy zmiany, wczytujemy plik do GHCi i sprawdzamy działanie obu funkcji

2. ***Zadania***:
    1. Wyznaczyć maksymalną wartość n, która nie powoduje przepełnienia stosu
    2. Wyznaczyć złożoność czasową i pamięciową obu funkcji
3. W pliku `ex8.hs` dodajemy następującą funkcję

```
ackerFun m n
 | m == 0    = n + 1
 | n == 0    = ackerFun (m - 1) 1
 | otherwise = ackerFun (m - 1) (ackerFun m (n - 1))
```

4. ***Zadania***:
    1. Wyznaczyć maksymalne wartości `m` i `n` , dla których można obliczyć `ackerFun`
    2. Naszkicować zależności: `z1(y) = ackerFun(3, y)` i `z2(x) = ackerFun(x, 3)` . Jaka jest złożoność czasowa funkcji `ackerFun`

9) Rekursja 4

1. W pliku `ex9.hs` dodajemy następującą definicję

```
qSort :: Ord a => [a] -> [a]
qSort []     = []
qSort (x:xs) = qSort (leftPart xs) ++ [x] ++ qSort (rightPart xs)
 where
   leftPart  xs = [ y | y <- xs, y <= x ]
   rightPart xs = [ y | y <- xs, y > x  ]
```

zapisujemy zmiany, wczytujemy plik do GHCi i sprawdzamy poprawność działania funkcji `qSort`

2. ***Zadania***:
    1. Zmodyfikować funkcje wewnętrzne `leftPart` i `rightPart` tak, aby zamiast *list comprehensions* wykorzystać funkcję `filter` (z biblioteki standardowej)
    2. (*opcjonalne*) Zdefiniować funkcję `mSort` (implementacja algorytmu *MergeSort*)
    3. (*opcjonalne*) Zdefiniować funkcję `iSort` (implementacja algorytmu *Insertion Sort*)
    4. (*opcjonalne*) Naspiać dwie wersje funcji

```
concat :: [[a]] -> [a]
```

        - `concat'` - z wykorzystaniem list comprehension
        - `concat''` z wykorzystaniem rekursji

    5. (*opcjonalne*) Napisać definicje funkcji

```
isSorted :: [Int] -> Bool -- isSorted [1,2,2,3] = True
reverse' :: [a] -> [a] -- reverse [1,2,3] = [3,2,1]
zip' :: [a] -> [b] -> [(a,b)] -- zip' [1,2] [3,4] = [(1,3), (2,4
)]
unzip' :: [(a, b)] -> ([a],[b]) -- unzip [(1,2), (3,4)] = ([1,3]
,[2,4])
zip3' :: [a] -> [b] -> [c] -> [(a,b,c)]
subList :: Eq a => [a] -> [a] -> Bool -- subList [1,2] [3,1,2,4]
 = True
```

10) Dopasowanie wzorców: guards

1. W pliku `ex10.hs` wpisujemy

```
fst2Eq :: Eq a => [a] -> Bool
fst2Eq (x : y : _) | x == y = True
fst2Eq _                    = False
```

   zapisujemy zmiany, wczytujemy plik do GHCi i sprawdzamy działanie funkcji
2. ***Zadania***:
   1. Napisać definicję funkcji sprawdzającej, czy pierwszy element listy jest podzielnikiem drugiego
   2. (*opcjonalne*) Napisać definicję funkcji sprawdzającej, czy pierwszy element listy jest podzielnikiem trzeciego