Laboratorium 6.

0) Przygotowanie środowiska

W konsoli/terminalu wpisujemy kolejno

```
$ cd
$ mkdir haskell-lab6
$ cd haskell-lab6
```

1) Składanie i "aplikacja" funkcji: *funkcje postaci*: $a \to b$ vs. $a \to m\ b$ (rozszerzone/monadyczne, *Kleisli arrows*)

1. W pliku ex1.hs wpisujemy

```
(<$<) :: (a -> b) -> a -> b
(<$<) = ($)

(>$>) :: a -> (a -> b) -> b
x >$> f = f x
infixl 0 >$>
```

(zapisujemy zmiany, wczytujemy plik do GHCi), w konsoli GHCi wpisujemy

```
ghci> (\x -> 2 * x + 1) <$< 1
ghci> 1 >$> (\x -> 2 * x + 1)
```

2. W pliku ex1.hs dodajemy

```
(<.<) :: (b -> c) -> (a -> b) -> (a -> c)
(<.<) = (.)

(>.>) :: (a -> b) -> (b -> c) -> (a -> c)
f >.> g = g . f
infixl 9 >.>
```

(zapisujemy zmiany, wczytujemy plik do GHCi), w konsoli GHCi wpisujemy

```
ghci> (+2) <.< (*3) <$< 5
ghci> (+2) >.> (*3) <$< 5

ghci> 5 >$> (+2) <.< (*3)
ghci> 5 >$> (+2) >.> (*3)

ghci> 3 >$> (+2) >$> (+10)
```

3. W konsoli GHCi wpisujemy

```
ghci> tail [1,2]
ghci> tail >.> tail >.> tail $ [1,2]
```

4. W pliku ex1.hs dodajemy

```
safeTail :: [a] -> Maybe [a]
safeTail []     = Nothing
safeTail (x:xs) = Just xs
```

(zapisujemy zmiany, wczytujemy plik do GHCi), w konsoli GHCi wpisujemy

```
ghci> safeTail [1,2]
ghci> safeTail []

ghci> safeTail >.> safeTail >.> safeTail <$< [1,2]
ghci> [1,2] >$> safeTail >.> safeTail >.> safeTail

ghci> :t safeTail
```

Dlaczego nie da się wykonać złożenia `safeTail >.> safeTail` ?

5. W pliku ex1.hs dodajemy

```
extractMaybe :: Maybe a -> a
extractMaybe Nothing  = error "Nothing inside!"
extractMaybe (Just x) = x
```

(zapisujemy zmiany, wczytujemy plik do GHCi), w konsoli GHCi wpisujemy

```
ghci> [1,2] >$> safeTail >.> extractMaybe >.> safeTail
ghci> [1,2] >$> safeTail >.> extractMaybe >.> safeTail >.> extractMaybe >.> s

ghci> [1] >$> safeTail >.> extractMaybe >.> safeTail >.> extractMaybe >.> saf
```

6. W pliku ex1.hs dodajemy

```
insertMaybe :: a -> Maybe a
insertMaybe = Just

-- (>^$>) = extract (^) and apply ($)
(>^$>) :: Maybe a -> (a -> Maybe b) -> Maybe b
ma >^$> f = (extractMaybe ma) >$> f
infixl 1 >^$>
```

(zapisujemy zmiany, wczytujemy plik do GHCi), w konsoli GHCi wpisujemy

```
ghci> insertMaybe [1,2]
ghci> [1,2] >$> insertMaybe >.> extractMaybe

ghci> [1,2] >^$> safeTail
ghci> :t (>^$>)
ghci> insertMaybe [1,2] >^$> safeTail
ghci> insertMaybe [1,2] >^$> safeTail >^$> safeTail
ghci> insertMaybe [1,2] >^$> safeTail >^$> safeTail >^$> safeTail
ghci> insertMaybe [1,2] >^$> safeTail >^$> safeTail >^$> safeTail >^$> safeTa
```

Dlaczego błąd pojawia się tylko w ostatnim wyrażeniu?

7. W pliku ex1.hs modyfikujemy definicję `(>^$>)`

```
(>^$>) :: Maybe a -> (a -> Maybe b) -> Maybe b
Nothing  >^$> _ = Nothing
(Just x) >^$> f = f x
infixl 1 >^$>
```

(zapisujemy zmiany, wczytujemy plik do GHCi), w konsoli GHCi wpisujemy

```
ghci> insertMaybe [1,2] >^$> safeTail
ghci> insertMaybe [1,2] >^$> safeTail >^$> safeTail
ghci> insertMaybe [1,2] >^$> safeTail >^$> safeTail >^$> safeTail
ghci> insertMaybe [1,2] >^$> safeTail >^$> safeTail >^$> safeTail >^$> safeTa
```

Dlaczego po modyfikacji `(>^$>)` powyższy błąd nie pojawia się?

8. W pliku ex1.hs dodajemy

```
f1 :: (Ord a, Num a) => a -> Maybe a
f1 x = if x > 0 then Just (x + 1) else Nothing

f2 :: (Eq a, Num a) => a -> Maybe a
f2 x = if x /= 0 then Just (10 * x) else Nothing

-- Kleisli composition
(>.>>) :: (a -> Maybe b) -> (b -> Maybe c) -> (a -> Maybe c)
f >.>> g = \x -> g (extractMaybe (f x))
```

(zapisujemy zmiany, wczytujemy plik do GHCi), w konsoli GHCi wpisujemy

```
ghci> h = f1 >.>> f2
ghci> :t h
ghci> f1 3
ghci> f2 3
ghci> h 3
```

9. W konsoli GHCi wpisujemy

```
ghci> let f3 x = Just (1/x)
ghci> let f4 x = Just (2 * x)
ghci> :t f3
ghci> :t f4

ghci> let f34 = f3 >.>> f4
ghci> :t f34
ghci> f34 2
```

10. ***Zadania***:
    1. Zmodyfikować definicję

    ```
    f >.>> g = \x -> g (extractMaybe (f x))
    ```

    tak, aby zamiast `extractMaybe` wykorzystać `>^$>` , tzn.

    ```
    f >.>> g = \x -> ___ >^$> ___
    ```

    2. (*opcjonalne*) Ponownie zmodyfikować definicję

    ```
    f >.>> g = ...
    ```

    ale tym razem zamiast `extractMaybe` wykorzystać `fmap` , tzn.

    ```
    f >.>> g = \x -> ___ fmap ___ ___
    ```

    wskazówka: rozważyć użycie funkcji pomocniczej

    ```
    joinMaybe :: Maybe (Maybe a) -> (Maybe a)
    ```

    3. Przeanalizować możliwość napisania funkcji `extractIO` (odpowiednika `extractMaybe` dla `IO`); *uwaga*: rozwiązanie wykorzystujące `unsafePerformIO` należy pominąć
    4. (*opcjonalne*) Napisać funkcje/operatory `extract` , `insert` , `>^$>` i `(>.>>)` dla funkcji postaci

    ```
    f :: a -> (b, String)
    g :: b -> (c, String)
    ```

2) Przykłady monad: `Maybe`

1. Sprawdzamy, czy aby (przez przypadek :) w poprzednim ćwiczeniu nie zdefiniowaliśmy monady? W konsoli GHCi wpisujemy

```
ghci> :i Maybe
ghci> :i Monad

ghci> return 1 :: Maybe Int
```

i analizujemy klasę typu `Monad`

2. (porównując deklaracje typów) znajdujemy odpowiedniki `insertMaybe` i `(>^$>)` ; czy istnieje odpowiednik `extractMaybe` ? dlaczego? czy istnieje odpowiednik `>.>>` ?
3. W konsoli GHCi wpisujemy

```
ghci> :t (>=>)
ghci> import Control.Monad
ghci> :t (>=>)
```

porównujemy sygnatury `(>=>)` i `(>.>>)`

4. W konsoli GHCi wpisujemy

```
ghci> insertMaybe [1,2]
ghci> return (Just [1,2])
ghci> return [1,2]
ghci> return [1,2] :: Maybe [Int]

ghci> insertMaybe [1,2] >^$> safeTail
ghci> return [1,2] >>= safeTail
ghci> (return [1,2] >>= safeTail) == (insertMaybe [1,2] >^$> safeTail)

ghci> insertMaybe [1,2] >^$> safeTail >^$> safeTail >^$> safeTail >^$> safeTa
ghci> return [1,2] >>= safeTail >>= safeTail >>= safeTail >>= safeTail

ghci> return [1,2] >^$> safeTail >^$> safeTail >^$> safeTail >^$> safeTail
ghci> insertMaybe [1,2] >>= safeTail >^$> safeTail >>= safeTail >^$> safeTail
ghci> return [1,2] >^$> safeTail >>= safeTail >^$> safeTail >>= safeTail
```

5. W pliku ex2.hs wpisujemy

```
doSafeTail3x :: [a] -> Maybe [a]
doSafeTail3x xs = do
  t1 <- safeTail xs
  t2 <- safeTail t1
  t3 <- safeTail t2
  return t3

safeTail3x :: [a] -> Maybe [a]
safeTail3x xs =
  safeTail xs >>= \t1 ->
    safeTail t1 >>= \t2 ->
      safeTail t2 >>= \t3 ->
```

```
        return t3

safeTail3x' :: [a] -> Maybe [a]
safeTail3x' xs = return xs >>= safeTail >>= safeTail >>= safeTail
```

(zapisujemy zmiany, wczytujemy plik do GHCi), w GHCi sprawdzamy działanie, np.

```
ghci> doSafeTail3x [1..5]
ghci> safeTail3x [1..5]
ghci> safeTail3x' [1..5]

ghci> doSafeTail3x [1]
ghci> safeTail3x [1]
ghci> safeTail3x' [1]
```

Która wersja (tj. `doSafeTail3x`, `safeTail3x` czy `safeTail3x'`) jest najbardziej czytelna?

6. W pliku ex2.hs dodajemy

```
f5 :: Int -> Int -> Int -> Int
f5 x y z = 1000 `div` x + 100 `div` y + 10 `div` z
```

(zapisujemy zmiany, wczytujemy plik do GHCi), w GHCi sprawdzamy działanie `f5`, np.

```
ghci> f5 1 2 3
ghci> f5 1 2 0
ghci> f5 1 0 3
ghci> f5 0 2 3
```

7. W pliku ex2.hs dodajemy

```
safeDiv :: Int -> Int -> Maybe Int
safeDiv x y | y /= 0    = Just $ x `div` y
            | otherwise = Nothing

safeF5 :: Int -> Int -> Int -> Maybe Int
safeF5 x y z =
  case (safeDiv 1000 x) of
    Nothing -> Nothing
    Just (iOverX) ->
      case (safeDiv 100 y) of
        Nothing -> Nothing
        Just (iOverY) ->
          case (safeDiv 10 z) of
            Nothing -> Nothing
            Just (iOverZ) -> Just $ iOverX + iOverY + iOverZ
```

(zapisujemy zmiany, wczytujemy plik do GHCi), w GHCi sprawdzamy działanie `safeF5`, np.

```
ghci> safeF5 1 2 3
ghci> safeF5 1 2 0
ghci> safeF5 1 0 3
ghci> safeF5 0 2 3
```

8. W pliku ex2.hs dodajemy

```
safeF5' :: Int -> Int -> Int -> Maybe Int
safeF5' x y z = do
  iOverX <- safeDiv 1000 x
  iOverY <- safeDiv 100 y
  iOverZ <- safeDiv 10 z
  return $ iOverX + iOverY + iOverZ
```

   (zapisujemy zmiany, wczytujemy plik do GHCi), w GHCi sprawdzamy działanie `safeF5'`, np. jw.
9. (*opcjonalne*) W pliku ex2.hs dodajemy

```
safeF5'' :: Int -> Int -> Int -> Maybe Int
safeF5'' x y z = f <$> iOverX <*> iOverY <*> iOverZ
  where
    f i j k = i + j + k
    iOverX = safeDiv 1000 x
    iOverY = safeDiv 100 y
    iOverZ = safeDiv 10 z
```

   (zapisujemy zmiany, wczytujemy plik do GHCi), w GHCi sprawdzamy działanie `safeF5''`, np. jw.; dlaczego ta wersja działa (poprawnie)?
10. (*opcjonalne*) W pliku ex2.hs dodajemy

```
sum10DivXi :: [Int] -> Int
sum10DivXi = foldr (\xi acc -> 10 `div` xi + acc) 0
```

    (zapisujemy zmiany, wczytujemy plik do GHCi), w GHCi sprawdzamy działanie `sum10DivXi'`, np.

```
ghci> sum10DivXi []
ghci> sum10DivXi [1]
ghci> sum10DivXi [1,2]
ghci> sum10DivXi [1,0,3]
```

11. ***Zadania***:
    1. (Dla monady `Maybe`) zdefiniować `(>=>)` przy pomocy `>>=`; czy można tę definicję uogólnić, aby była prawdziwa dla dowolnej monady?
    2. Przeanalizować implementację instancji `Monad Maybe` (np. w *Data.Maybe*)
    3. Napisać funkcję `join` dla monady `Maybe`
    4. (*opcjonalne*) Napisać trzy wersje funkcji:

```
safeSum10DivXi :: [Int] -> Maybe Int
```

- jedną wykorzystującą `case of` (por. `safeF5`)
- dwie wykorzystujące fakt, że `Maybe` jest monadą (jedną wersje z użyciem notacji *do*, a drugą z operatorem `>>=`)

5. (*opcjonalne*) Jak wyżej, ale dla funkcji:

```
safeProduct10DivXi [Int] -> Maybe Int
safeAbs10DivXi [Int] -> Maybe Int
```

6. (*opcjonalne*) Sprawdzić możliwość uogólnienia powyższych rozwiązań przy pomocy funkcji

```
foldM :: (Foldable t, Monad m) => (b -> a -> m b) -> b -> t a -> m b
```

uwaga: potrzebny import `Control.Monad`

7. (*opcjonalne*) Przeanalizować definicje funkcji: `mapM`, `forM`, `sequence`, `filterM`, `liftM2`, `...`, `liftM5` i `zipWithM` oraz zastanowić się nad możliwymi zastosowaniami (tych funkcji)
8. (*opcjonalne*) Sprawdzić "prawa monadyczne" dla monady `Maybe`; *wskazówka*: wykorzystać wersję z operatorem `>=>`

3) Przykłady monad: `Either e` (*ćwiczenie opcjonalne*)

1. W konsoli GHCi wpisujemy

```
ghci> :i Either
ghci> :i Monad

ghci> return 1 :: Either String Int
```

i sprawdzamy, ile parametrów typu ma (konstruktor typu) `Either`? Czy ta liczba (parametrów) zgadza się z wymaganiami klasy `Monad`?

2. W pliku ex3.hs wpisujemy

```
safeTail :: [a] -> Either String [a]
safeTail []     = Left "Empty list!"
safeTail (x:xs) = Right xs
```

(zapisujemy zmiany, wczytujemy plik do GHCi), w GHCi sprawdzamy działanie `safeTail`, np.

```
ghci> safeTail [1..4]
ghci> safeTail [1]
ghci> safeTail []
```

3. W pliku ex3.hs dodajemy

```
doSafeTail3x :: [a] -> Either String [a]
doSafeTail3x xs = do
  t1 <- safeTail xs
  t2 <- safeTail t1
  t3 <- safeTail t2
  return t3
```

(zapisujemy zmiany, wczytujemy plik do GHCi), w GHCi sprawdzamy działanie `doSafeTail3x`, np.

```
ghci> doSafeTail3x [1..4]
ghci> doSafeTail3x [1..3]
ghci> doSafeTail3x [1,2]
```

4. W pliku ex3.hs dodajemy

```
safeDiv :: Int -> Int -> Either String Int
safeDiv x y | y /= 0    = Right $ x `div` y
            | otherwise = Left "Cannot div by zero!"

safeF5 :: Int -> Int -> Int -> Either String Int
safeF5 x y z =
  case (safeDiv 1000 x) of
    Left e -> Left e
    Right (iOverX) ->
      case (safeDiv 100 y) of
        Left e -> Left e
        Right (iOverY) ->
          case (safeDiv 10 z) of
            Left e -> Left e
            Right (iOverZ) -> Right $ iOverX + iOverY + iOverZ
```

(zapisujemy zmiany, wczytujemy plik do GHCi), w GHCi sprawdzamy działanie `safeF5`, np.

```
ghci> safeF5 1 2 3
ghci> safeF5 1 2 0
ghci> safeF5 1 0 3
ghci> safeF5 0 2 3
```

5. W pliku ex3.hs dodajemy

```
safeF5' :: Int -> Int -> Int -> Either String Int
safeF5' x y z = do
  iOverX <- safeDiv 1000 x
  iOverY <- safeDiv 100 y
  iOverZ <- safeDiv 10 z
  return $ iOverX + iOverY + iOverZ
```

(zapisujemy zmiany, wczytujemy plik do GHCi), w GHCi sprawdzamy działanie `safeF5'`, np. jw.

6. ***Zadania***:
   1. Przeanalizować implementację instancji `Monad (Either e)` (np. w *Data.Either*)
   2. Napisać funkcję `join` dla monady `(Either e)`
   3. Napisać definicję funkcji `safeF5''` (odpowiadającą tej z poprzedniego ćwiczenia, dotyczącego monady `Maybe`) z wykorzystaniem monady `Either e`

      ```
      safeF5'' :: Int -> Int -> Int -> Either String Int
      ```

   4. Zaproponować sposób obsługi błędów wykorzystujący `Either` i typ wyliczeniowy reprezentujący kolejne kody błędu

4) Przykłady monad: `[]`

1. W konsoli GHCi wpisujemy

   ```
   ghci> :i []

   ghci> [1,2,3] :: [Int]
   ghci> [1,2,3] :: [] Int
   ```

2. W konsoli GHCi wpisujemy

   ```
   ghci> return 1 :: [] Int
   ghci> return 3 >>= (\x -> [1..x])
   ghci> [1,2] >>= (\x -> [-x,x])
   ghci> [1,2,3] >>= (\x -> [-x..x])

   ghci> [1,2] >>= (\x -> [-x,x]) >>= (\y -> [-y,y])
   ghci> [1,2] >>= \x -> [-x,x] >>= \y -> [-y,y]
   ghci> [1,2] >>= \x -> [-x,x] >>= \y -> return (x,y)
   ```

3. W pliku ex4.hs wpisujemy

   ```
   xs1 :: [(Int,Int,Int)]
   xs1 = [ (x,y,z) | let xs = [1,2], x <- xs, y <- xs, z <- xs ]

   doXs1 :: [(Int,Int,Int)]
   doXs1 = do
     let xs = [1,2]
     x <- xs
     y <- xs
     z <- xs
     return (x,y,z)
   ```

(zapisujemy zmiany, wczytujemy plik do GHCi), w GHCi sprawdzamy `xs1` i `doXs1`, np.

```
ghci> xs1
ghci> doXs1
ghci> xs1 == doXs1
```

4. Na początku pliku ex4.hs wpisujemy

```
import Control.Monad
```

a gdzieś dalej (np. po `doXs1`) dodajemy:

```
xs2 :: [(Int,Int,Int)]
xs2 = [ (x,y,z) | let xs = [1..3], x <- xs, y <- xs, z <- xs, x > y && y > z

doXs2 :: [(Int,Int,Int)]
doXs2 = do
  let xs = [1..3]
  x <- xs
  y <- xs
  z <- xs
  guard $ x > y && y > z
  return (x,y,z)

doXs2' :: [(Int,Int,Int)]
doXs2' = do
  let xs = [1..3]
  x <- xs
  y <- xs
  z <- xs
  if x > y && y > z
    then return (x,y,z)
    else []
```

(zapisujemy zmiany, wczytujemy plik do GHCi), w GHCi sprawdzamy `xs2`, `doXs2` i `doXs2'` con, np. jw.

5. W konsoli GHCi wpisujemy

```
ghci> :t guard
ghci> [1..10] >>= \x -> guard (even x) >> return x
ghci> [1..10] >>= \x -> guard (even x) >>= \y -> return (x,y)

ghci> [1..10] >>= \x -> return x
ghci> [1..10] >>= return
```

6. ***Zadania***:
    1. Przeanalizować implementację instancji `Monad []` (np. w *Control.Monad.Instances*)
    2. Napisać funkcję `join` dla monady `[]`
    3. (*opcjonalne*) Przeczytać artykuł Bringing Back Monad Comprehensions

5) Przykłady monad: `Writer` (*ćwiczenie opcjonalne*)

1. W konsoli GHCi wpisujemy

```
ghci> :i Writer
ghci> import Control.Monad.Trans.Writer.Lazy
ghci> :i Writer
ghci> :i WriterT
```

2. W konsoli GHCi wpisujemy

```
ghci> :t writer
ghci> :t runWriter
ghci> :t execWriter
ghci> :t tell

ghci> runWriter $ writer(1,["op1","op2"])
ghci> execWriter $ writer(1,["op1","op2"])

ghci> runWriter (return 1 :: Writer String Int)
ghci> runWriter (return [1] :: Writer String [Int])
ghci> runWriter (return (1,1) :: Writer [String] (Int,Int))
```

3. W pliku ex5.hs wpisujemy

```
import Control.Monad.Trans.Writer.Lazy

gcdWithLog :: Int -> Int -> Writer [String] Int
gcdWithLog a b
  | b == 0 = do
      tell ["Finished with " ++ show a]
      return a
  | otherwise = do
      tell [show a ++ " mod " ++ show b ++ " = " ++ show (a `mod` b)]
      gcdWithLog b (a `mod` b)
```

(zapisujemy zmiany, wczytujemy plik do GHCi), w GHCi sprawdzamy działanie `gcdWithLog` , np.

```
ghci> gcdWithLog 4 16
ghci> runWriter $ gcdWithLog 4 16
ghci> execWriter $ gcdWithLog 4 16
```

4. W pliku ex5.hs dodajemy

```
mapWithLog :: Show a => (a -> b) -> [a] -> Writer [String] [b]
mapWithLog _ [] = do
  tell ["map []"]
```

```
  return []
mapWithLog f (x:xs) = do
  tell ["map " ++ show x]
  mapXs <- mapWithLog f xs
  return $ f x : mapXs
```

(zapisujemy zmiany, wczytujemy plik do GHCi), w GHCi sprawdzamy działanie `mapWithLog` , np.

```
ghci> mapWithLog (*2) [1..5]
ghci> runWriter $ mapWithLog (*2) [1..5]
ghci> execWriter $ mapWithLog (*2) [1..5]
```

5. ***Zadania***:
    1. Przeanalizować implementację instancji `Monad (WriterT w m)`
    2. Napisać funkcję

        ```
        filterWithLog :: Show a => (a -> Bool) -> [a] -> Writer [String] [a]
        ```

    3. Napisać funkcję `foldrWithLog`