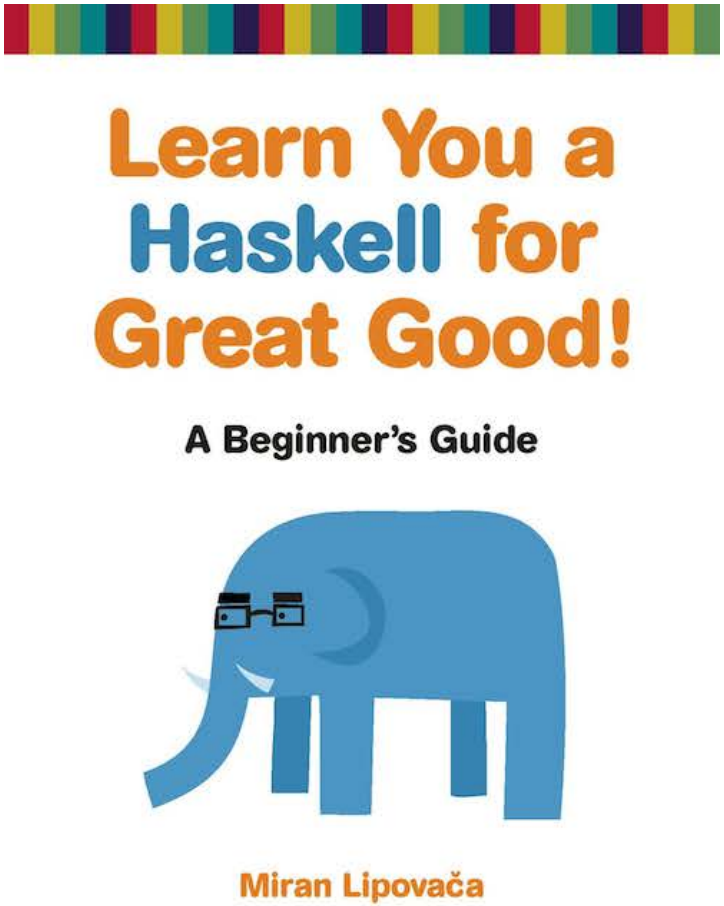
# PARADYGMATY I JĘZYKI PROGRAMOWANIA

Haskell. (w11)

# !!!




Learn You a
Haskell for
Great Good!
A Beginner's Guide
Miran Lipovača

# Brian Beckman o Monadach



- Brian Beckman: Don't fear the Monad - YouTube

# Plan wykładu



- Typy
- Funkcje Preludium
  - `take, cycle`
  - `zip, unzip, zipWith`
  - `filter, map`
- Klasy typów danych
  - `Show`
  - `Eq, Ord`
  - `Num, Integral, Fractional, Floating`
  - `Funktor`
- Monady
  - `Id`
  - `Maybe`
  - `[]`
  - `IO`
  - Notacja `do`
- Wykład Briana Beckmana o monadach dont-fear-the-monad-a-monoid-is
- Przykłady

# Typy



- Typ funkcji

```
f x y = x*x + y*y
main = print (f 2.3 4.2)
:type f
f :: Num => a -> a ->a
```

- 

```
Int                 =typ Int
Int -> Int          =typ funkcji z Int do Int
Float -> Int        =typ funkcji z Float do Int
a -> Int            =t. funkcji z dowolnego t. do Int
a -> a              =t. f. z a do a
a -> a-> a          =t. f. dwuargumentowej z (a,a) do a
```

- W wyrazenie `a->a->a` symbol `a` oznacza zmienną typową. Funkcja bierze dwa argumenty typu `a` i jej wynikiem jest wartość typu `a`. Zmienna typowa `a` może przyjmować rożne wartości typowe (`Int`, `Integer`, `Float`). (tzw. polimorfizm parametryczny)
  W przykładzie `a` należy do klasy typów `Num (Num => a – a -> a->)`.

# Typy



- ```
  :type f
  f :: Num => a -> a ->a
  ```

To można interpretować również inaczej. Jest to funkcja jednego argumentu typu a, której wynikiem jest funkcja (a -> a), pobierająca argument typu a i wydająca wynik typu a. Wyrażenie f 3 4 jest równoważne (f 3) 4 co oznacza, że (f 3) jest funkcją jednoargumentową ...

```
f :: Num => a -> a ->a

g :: Num => a -> a
g = f 3

g y                                  -- funkcja 3*3 +y*y
```
```

# Przykład (O. Taykalo)



Suma liczb parzystych
[1,2,3,4,5] -> 2 + 4 -> 6

**C:**

```
function evenSum(list) {
    var result = 0;
    for (var i=0; i< list.length ; i++) {
        if (list[i] % 2 ==0) {
            result += list[i];
        }
    }
    return result;
}
```

**Zamiast pętli -> REKURENCJA**

# Przykład



```
int evenSum(int *list) {
    return accumSum(0,list);
}

int accumSum(int n, int *list) {
    int x;
    int *xs;
    if (*list == 0) { // if the list is empty
        return n;
    } else {
        x = list[0]; // let x be the first element of the list
        xs = list+1; // let xs be the list without x
        if ( 0 == (x%2) ) { // if x is even
            return accumSum(n+x, xs);
        } else {
            return accumSum(n, xs);
          }
     }
}
```

# Przykład v1



```
-- Version 1
evenSum :: [Integer] -> Integer

evenSum l = accumSum 0 l

accumSum n l = if l == []
                  then n
                  else let x = head l
                           xs = tail l
                       in if even x
                             then accumSum (n+x) xs
                             else accumSum n xs
```

# Przykład v2 (bez zaśmiecania ...)



```
-- Version 2
evenSum :: Integral a => [a] -> a

evenSum l = accumSum 0 l
    where accumSum n l =
            if l == []
                then n
                else let x = head l
                         xs = tail l
                     in if even x
                            then accumSum (n+x) xs
                            else accumSum n xs
```

# Przykład v3 (wzorce)



```
-- Version 3
evenSum l = accumSum 0 l
    where
        accumSum n [] = n
        accumSum n (x:xs) =
             if even x
                then accumSum (n+x) xs
                else accumSum n xs
```

# Przykład v4 (redukcja η)



```
-- Version 4
evenSum :: Integral a => [a] -> a

evenSum = accumSum 0
    where
        accumSum n [] = n
        accumSum n (x:xs) =
            if even x
               then accumSum (n+x) xs
               else accumSum n xs
```

# Przykład v5 (funkcje wyższego rzędu)



```
filter :: (a -> Bool) -> [a] -> [a]
map :: (a -> b) -> [a] -> [b]
foldl :: (a -> b -> a) -> a -> [b] -> a


-- Version 5
evenSum l = mysum 0 (filter even l)
    where
      mysum n [] = n
      mysum n (x:xs) = mysum (n+x) xs
```

# Przykład v6 (->strict, leniwość)



**foldl -> foldl'**

```
-- Version 6
-- foldl' importujemy z modułu Data.List
import Data.List
evenSum l = foldl' mysum 0 (filter even l)
      where mysum acc value = acc + value
```

# Przykład v7 (lambda)



```
-- Version 7
-- Lepiej jest importować tylko to co potrzebne
import Data.List (foldl')
evenSum l = foldl' (\x y -> x+y) 0 (filter even l)
```

# Przykład v8 ( `\x y -> x+y --> (+)` )



```
-- Version 8
import Data.List (foldl')
evenSum :: Integral a => [a] -> a
evenSum l = foldl' (+) 0 (filter even l)
```

Integer może być dowolnie dużą liczbą całkowitą:

```
Prelude> 19^17
548038685778480218593 9
```

# Przykład v9 (operator .)



```
-- Version 9
import Data.List (foldl')
evenSum :: Integral a => [a] -> a
evenSum = (foldl' (+) 0) . (filter even)
```

# Przykład v10 (klarowność)



Przezwiemy pewne części...

```
-- Version 10
import Data.List (foldl')
sum' :: (Num a) => [a] -> a
sum' = foldl' (+) 0
evenSum :: Integral a => [a] -> a
evenSum = sum' . (filter even)
```

# Przykład – zastosowanie ...



Suma kwadratów liczb parzystych

```
squareEvenSum = sum' . (filter even) . (map (^2))

squareEvenSum' = evenSum . (map (^2))

squareEvenSum'' = sum' . (map (^2)) . (filter even)
```

Dodaliśmy funkcję `map,` która stosuje inną funkcje (tutaj `(^2)`) do listy.

# Funkcje podstawowe

```
take, cycle, zip, unzip,
zipWith, filter, map, ...
```

# take



- lista = [1..]
- głowa i ogon listy; dopasowanie do wzorca; *pattern-matching*;

```
og (x:y) = y    -- ogon listy
og [] = []

gl (x:y) = x    -- głowa listy
gl [] = error „gl []”
```

**głowa (head)** i **ogon (tail)** są zdefiniowane **w Prelude** Haskella

# take



DEFINICJA

```
take :: Int -> [a] -> [a]
take 0 _      = []
take n (x:xs) = x : take (n-1) xs
take _ []     = []
```

Pierwsza linia jest definicją typu funkcji: **`Int -> [a] -> [a]`** – argumenty funkcji są typu **`Int i [a]`**, a wynik jest typu **`[a]`**. Tutaj **`[a]`** jest typem listowym o elementach typu **`a`** – take pobiera liczbę i listę i zwraca listę elementów tego samego typu. Funkcja **`take`** jest standardowo zdefiniowana w **`Preludium`**. Podkreślenie (_) oznacza dowolny argument, który nie jest używany z prawej strony definicji.

# cycle



DEFINICJA:

```
cycle [] =[]                        -- cycle :: [a] -> [a]
cycle x = x + cycle x
```

Definicja w **Preludium:**

```
cycle :: [a] -> [a]
cycle x = x'
  where x' = x ++ x'
```

Jeśli wywołamy `cycle x` to obliczenia będą trwały w nieskończoność. Powinniśmy użyć, np. `take 5 (cycle„abc") ⇒ „abcab".` Inny przykład:

```
let lista = cycle [1,0]
take 4 lista
⇒ [1,0,1,0]
```

# zip, unzip, zipWith



- Sygnatury
  - `zip :: [a] -> [b] ->[(a, b)]`
  - `unzip :: [(a, b)] -> ([a], [b])`
  - `zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]`
- `zip` **i** `unzip` konstruują listy z kolejnych elementów list wejściowych.
- Przykłady:
  - `zip [1..] (cycle [3,2])`
    ⇒ `[(1,3), (2,2), (3,3), (4,2), ...]`
  - `take 4 (zip [1..] (cycle [3,2]))`
    ⇒ `[(1,3),(2,2),(3,3),(4,2)]`
- `zipWith` tworzy listę stosując do pary określoną funkcje
  - `zipWith (+) x y` -- sumowanie list `x` i `y`
  - `zipWith (*) [1,2,3] [2,3,4]` -- mnożenie list

# Liczby Fibonacciego



```
fib :: Int -> Int
fib 0 = 1
fib 1 = 1
fib n = fib n-1 + fib n-2
```

- Lepszą definicję dostaniemy zauważając, że suma liczb ciągu Fibonacciegi i jego ogona daje ogon ogona:

```
   1, 1, 2, 3,  5,  8, 13, 21, ...
+  1, 2, 3, 5,  8, 13, 21, 34, ...
-------------------------------------
=  2, 3, 5, 8, 13, 21, 34, 55, ... brakuje 2 pierwszych
```

Możemy więc napisać

```
fibs = 1 : 1 : zipWith (+) fibs (tail fibs)
```

(liniowy czas wykonania; pamiętane wyliczone elementy):

# filter



- **filter** wybiera z listy tylko określone elementy (Zawłocki, Haskell)

```
let odds = filter odd [1..]
take 3 odds  ⇒ [1,3,5]
```

- Przykład. Sito Eratostenesa

```
sito :: Int -> [Int] -> [Int]
sito n m = filter (nieDzieliPrzez n) m
   where nieDzieliPrzez n k = (k `mod` n) /= 0  -- infix mod

eratostenes :: [Int] -> [Int]
eratostenes (n:m) = n : eratostenes (sito n m)
eratostenes []     = []

pierwsze = eratostenes [2..]

take 20 pierwsze
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71]
```

# map



- `:t map`

```
map :: (a -> b) -> [a] -> [b]
```

- Argumentami `map` jest funkcja typu `a->b` i lista elementów typu `a` (czyli `[a]`). Wynikiem jest lista elementów typu `b` (czyli `[b]`) otrzymana przez aplikację funkcji do kolejnych elementów listy wejściowej.

# foldl



- `foldl` ***redukuje*** listę przy pomocy zadanej operacji
- `foldl :: (a -> b -> a) -> a -> [b] -> a`
- `foldl op z [] ---> z`
  `foldl op z [a1, a2, ..., an] --->> (...((z `op` a1) `op` a2) `op` ...`op` an)`

- `and, or :: [Bool] -> [Bool]`
  `and = foldl (&&) True`
  `or  = foldl (||) False`

  `len :: [a] -> Int`
  `len = foldl (+) 0.map one`
  `  where one n = 1`

  `sum, prod :: [a] -> Int`
  `sum  = foldl (+) 0`
  `prod = foldl (*) 1`

# Klasy typów danych

**Show, Eq, Ord, Num, Integral, Fractional, Floating, Funktor**

# Klasy typów danych



```
Prelude> map

<interactive>:5:1:
    No instance for (Show ((a0 -> b0) -> [a0] -> [b0]))
      arising from a use of `print'
    Possible fix:
      add an instance declaration for (Show ((a0 -> b0) -> [a0] -> [b0]))
    In a stmt of an interactive GHCi command: print it
Prelude>
```

Widzimy, że brakuje tutaj metody, która wyprowadza (wypisuje) `map`. Jest to metoda `show` używana przez `` `print' ``. Metody `show` są metodami typów z klasy `Show`.

Klasa Show ma postać (Typ a należy do klasy Show):

```
class Show a where        -- duża litera S w Show
   show :: a -> String    -- mała litera s w show
```

# Klasy typów danych



- Zdefiniujmy nowy typ drzew binarnych **Drzewo:**

```
data Drzewo a = Lisc
    | Konar (Drzewo a) a (Drzewo a)

wstaw Lisc x = Konar Lisc x Lisc
wstaw (t@(Konar d1 y d2)) x
    | x == y    = t
    | x < y     = Konar (wstaw d1 x) y d2
    | otherwise = Konar d1 y (wstaw d2 x)
```

- Sprawdźmy typ funkcji **insert**

```
:t wstaw
wstaw :: Ord a => Drzewo a -> a -> Drzewo a
```

- **Ord a =>** jest to *kontekst*, który mówi, że typ **a** musi należeć do klasy typów **Ord**; funkcja **(<)** jest zdefiniowana tylko dla tej klasy typów; funkcja **(==)** należy do typów klasy Eq. Założenie, że **a** należy do **Eq** jest niepotrzebne, bo typy **Ord** należą do klasy **Eq** automatycznie.

# Klasy typów danych



- Funkcja insert posłuży do redukcji listy, polegającej na wstawieniu elementów listy do drzewa:

```
wstawListe :: (Ord a) => Drzewo a -> [a] -> Drzewo a
wstawtListe = foldl wstaw

zListy = wstawListe Lisc
```

- Sprawdzenie
  ...

# Klasy typów danych



- Funkcja wypisująca drzewo:

```
pokazujDrzewo :: (Show a) => Drzewo a -> String
pokazujDrzewo t = pokazujDrzewo' t ""
   where pokazujDrzewo' Lisc = id
         pokazujDrzewo' (Konar t1 a t2)
            = ('(':)              .
            pokazujDrzewo' t1     .
            ((show a) ++)         .
            pokazujDrzewo' t2     .
            (')':)
```

- Zgłoszenie klasy typu Drzewo i funkcja **show:**

```
instance (Show a) => Show (Drzewo a) where
   show = pokazujDrzewo
```

# Klasy typów danych



- W typie funkcji **`pokazujDrzewo`** pojawia się nazwa klasy **`Show`**. Elementami Show są typy dla których zdefiniowana jest funkcja show do wypisywania danych typu a.
- Klasa Show ma postać (Typ a należy do klasy Show):

```
class Show a where          -- duże S
   show :: a -> String      -- małe s
```

(Typ **`a`** należy do klasy **`Show`** jeśli jest dla niego zdefiniowana funkcja **`show`**)

- Przynależność do klasy Show zgłaszamy jawnie:

```
instance (Show a) => Show (Drzewo a) where
   show = pokazujDrzewo
```

- Podobnie jest z innymi klasami typów

# Klasa **Eq**



```
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
    (/=) = not . (==)


class (Eq a) => Ord a where
    compare :: a -> a -> Ordering
    (<) :: a -> a -> Bool
    (<=) :: a -> a -> Bool
    (>) :: a -> a -> Bool
    (>=) :: a -> a -> Bool
    max :: a -> a -> a
    min :: a -> a -> a
```

# Inne klasy



- **`Num`**
  - z operacjami takimi jak **`(+), (-), (*)`** czy **`fromInteger`** (do tej klasy należą między innymi typy **`Int`**, **`Integer`**, **`Float`**, **`Double`**),
- **`Integral`**
  - operacje **`quot`**, **`rem`**, **`div`**, **`toInteger`**,
- **`Fractional`**
  - operacje **`(/)`** i **`fromRational`**,
- **`Floating`**
  - **`pi`**, **`exp`**, **`log`**, **`sin`** itp.

# Functor



- Klasa **`Functor`** posiada tylko jedna operację

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

- Do tej klasy należy `[a]`.

- Nowe klasy typów definiuje się, podając operacje, które mają być na nich zdefiniowane; przykład klasa **`Dict`** (patrz np. Zawłocki)

# Monady

Klasa **`Monad`**

# Brian Beckman o Monadach



- Brian Beckman: Don't fear the Monad - YouTube

# Monady



- Pojęcie z filozofii (Pitagorejczycy, Leibniz) i matematycznej teorii kategorii i funktorów
- Eugenio Mogi (lata 80te XX wieku) – wprowadzenie monad do informatyki
- Philip Wadler –zastosowanie monad w informatyce

Philip Wadler

„*Nie ma nic oprócz monad, albo inaczej - wszystko co istnieje musi być monadą, czyli osobnym bytem, który zawiera w sobie całą prawdę o sobie*”. *(Gotfried Wilhelm Leibniz, 1646-1716)*

# Monady



- Efekty uboczne, we/wy, ...
- Funkcje *czyste* i *nieczyste*
- Lenistwo
- Przeźroczystość referencyjna

W PROGRAMOWANIU FUNKCYJNYM WSZYSTKO MUSI BYĆ FUNKCJĄ!

MONADA PRZEDSTAWIA SEKWENCYJNE WYKONANIE OKREŚLONYCH AKCJI.

Wymagane jest zdefiniowanie trzech operacji: `return`, `(>>=)` i `(>>)`:

```
return x   -- akcja zwrotu x
f  >>= g   -- zwija dwie akcje (g działa na wynik f)
f  >>  g   -- zwija akcje g i f ale g nie uwzględnia wyniku f
```

Należy pamiętać, że efekty uboczne są przekazywane w ukryty sposób w obu operacjach składania `(>>=)` i `(>>)`

# Monady



- Typy akcji monadowych

```
return :: (Monad m) => a -> m a
(>>=)  :: (Monad m) => m a -> (a -> m b) -> m b
(>>)   :: (Monad m) => m a -> m b -> m b
```

- Interpretacja konstruktora `m`:
  Jeśli `a` jest typem wartości to `m` reprezentuje typ obliczeń zwracających wartość typu `a`
- Obliczenie typu `m a` może być np.:
  - wartością typu `a`; `m` jest wtedy trywialnym obliczeniem
  - wartością typu `a` lub wyjątkiem reprezentującym błąd
  - zbiorem wartości typu `a`; `m a` oznacza wtedy obliczenie niedeterministyczne
  - programem z efektami ubocznymi reprezentowanymi przez funkcję typu `s -> (a, s)`, gdzie `s` jest typem stanu modyfikowanym przez funkcję

# Klasa **Monad**



- Operacje **`return`** i **`(>>=)`** spełniają pewne związki wynikające z kategoryjnej semantyki monad:

1. `return a >>= \x -> b === (\x -> b) a`
2. `f >>= \x -> return x === f`
3. `f >>= (\x -> g >>= \y -> h)`
   `=== (f >>= \x -> g) >>= \y -> h`

- Pierwsze dwa związki pokazują jak zachowuje się **`return`** po obu stronach operacji **`>>=`**
  Związek trzeci stwierdza łączność **`>>=`**.

  - zapewnienie tych związków dba programista.

# Przykłady monad



- Monada **Id**
  - Reprezentuje obliczenia, które nie robią nic

```
data Id a = Id a

instance Monad Id where
        return = Id
        (Id a) >>= f = f a
```

- Monada **Maybe**
  - Wartości typu **Maybe a** (konstruktorem typu jest **Maybe**) reprezentują wynik typu **a** lub błędne obliczenia reprezentowane przez konstruktor **Nothing**

# Przykłady monad



- **Maybe** cd.

```
data Maybe a = Just a | Nothing

instance Monad Maybe where
        return = Just
        Nothing >>= m = Nothing
        (Just a) >>= m = m a
```

# Przykład zastosowania **Maybe**



Funkcja, która dla danego słownika będącego listą par **(String, Int)** i dla danych dwu kluczy wyszukuje związane z nimi wartości i zwraca ich sumę. W przypadku gdy wartości nie są zdefiniowane zwraca **Nothing**.

```
lookup :: (Eq a) => a [(a,b)] -> Maybe b


lookupSum :: [(String, Int)] -> String ->
                        String -> Maybe Int
LookupSum l k1 k2 =
      lookup k1 l >>= \v1 ->
      lookup k2 l >>= \v2 ->
      return (v1 + v2)
```

Funkcja **lookup** jest funkcją standardową.

# Monady...



- Monada **Result**

  Więcej informacji o błędach

```
data Result a = OK a | Failure String

instance Monad Result where
        return = OK
        fail = Failure
        (OK a) >>= f = f a
        (Failure s) >>= _ = fail s
```

`fail` jest funkcją standardową typu **`String -> m a`** z klasy **`Monad`**.
**`fail str`** reprezentuje nieudane obliczenie z informacją o błędzie w napisie **`str`**.

# Monady...



- Monada []

  Listy są klasy Monad. Wartość typu [a] jest obliczeniem, zwracającym całą listę wyników

```
instance Monad [] where
        return a = [a]
        l >>= f = concat (map f l)
        fail _ = []
```

- Zadanie. Zapisać równości monad dla list .

  np. **`concat (map (\x -> a) [b]) === (\x -> a) [b]`**

# Monady...



- Monada **IO**
  Operacje we/wy są operacjami o efekach ubocznych (zmiany stanu). Jeśli np. typ **Świat** reprezentuje stan Świata, to obliczenia, które zwracają wartości typu a i zmieniają stan Świata (efekt uboczny obliczeń) można traktować jako element typu **Świat -> (a Świat)**.
  W języku Haskell zamiast **Świat** używamy typu **IO a**. Ponieważ wiemy, że konstruktor **IO** jest monadą, więc możemy składać operacje z efektami ubocznymi.

# IO ...



- Przykłady funkcji standardowych:
  - `getChar :: IO Char`
  - `putChar :: IChar -> IO ()`
  - `getLine :: IO String`
  - `putStr  :: String -> IO ()`
  - `print   :: Show a => a -> IO ()`
- Przykłady złożeń funkcji standardowych

```
echo :: IO ()
echo = getLine >>= putStr
```

# IO ...



- Przykłady złożeń funkcji standardowych

```
palindrom :: IO ()
palindrom =
      putStr „Podaj napis ” >>
      getLine >>= \line
      let line’=[toLOwer c | c <- line, c /= ‘
      in
        if line’ == reverse line’
           then putStr(„jest palindromem\n”)
           else putStr(„nie jest palindromem”)
```

# Monady ...



- Notacja **do**
- Przykład: **palindrom**

```
palindrom :: IO ()
palindrom = do
      putStr "Podaj napis "
      cs <- getLine
      if palindrom cs
         then putStr "Tak.\n"
         else putStr "Nie.\n"

czy_palindrom :: String -> Bool
czy_palindrom x = (x == reverse x)
```

# Funkcje wyższego rzędu



- Funkcje jednoargumentowe
- Funkcje wieloargumentowe redukowane są do funkcji jednoargumentowych np. f x y === (f x) y

- `curry`
- `uncurry`

- Funkcje wyższego rzędu są to funkcje, które generują lub pobierają inne funkcje (argumenty wejściowe i ich wyniki są funkcjami)

Haskell Curry
(1900-1982)

# Zadania



Poniższe zadania zrealizować w języku Haskell.

1. Napisać program obliczania pochodnej funkcji jednej zmiennej.

2. Napisać program poprawiania obliczeń pochodnej metodą Richardsona.

3. Podać definicję funkcji **`zip`** i **`unzip`** (**`zip`** bierze dwie listy i tworzy listę par złożonych z kolejnych elementów obu list; **`unzip`** – bierze listę par i tworzy parę list tak, jak w przykłądzie).
   Przykład:
   ```
   zip [1,2,3,4] [5,6,7,8]
    ⇒ [(1,5),(2,6),(3,7),(4,8)]
   unzip [(1,5),(2,6),(3,7),(4,8)]
    ⇒ ([1,2,3,4],[5,6,7,8])
   ```

4. Sito Eratostenesa wybierania liczb pierwszych (użyć funkcji **`take`**).

# Pierwiastek – stary przykład



```
-- pierwiastek kwadratowy; metoda Newtona-Raphsona
-- formula = x <- 0.5*(x+kwadrat/x)
-- Przykładowe wywołanie: pierwiastek 2

-- leniwy ciag Newtona-Raphsona --
leniwyCNR a = rekurencja a (formula a) where
      formula kwadrat x  = 0.5*(x + kwadrat/x)
      rekurencja start f = start:rekurencja (f start) f

-- przyblizenie --
przyblizenie epsilon (x1:x2:ogon)
      | abs(x1-x2) < epsilon = x2
      | otherwise = przyblizenie epsilon (x2:ogon)

-- wszystko razem --
pierwiastek a = przyblizenie (a*0.0000000001) (leniwyCNR a)
```

# Inne równania f x = 0



```
-- metoda Newtona-Raphsona
--  (Bird, Wadler)

-- pierwiastek kwadratowy
-- z liczby >=0
sq1 x = until satis improve x
  where
    satis y = abs(y^2-x)<eps
    improve y = (y + x/y)/2
    eps = 0.000000001*x

-- pochodna
deriv f x =(f (x+dx)-f x)/dx
      where dx = 0.0001
```

```
-- ogolniej metoda newtona
newton f = until satis improve
  where satis y =abs (f y)<eps
     improve y
         = y-(f y/deriv f y)
     eps = 0.0001

-- pierwiastek kwadratowy
sq2 x = newton f x
        where f y = y^2 - x

-- pierwiastek szescienny
cubrt x = newton f x
          where f y = y^3 - x
```

# Kula... n-wymiarowa



- Objętość kuli:

$$V_n = \frac{\pi^{\frac{n}{2}}}{\Gamma(\frac{n}{2}+1)} \cdot r^n = \begin{cases} \dfrac{\pi^k}{k!} \cdot r^n & \text{dla } n = 2k, \\ \dfrac{2^k \pi^{k-1}}{n!!} \cdot r^n & \text{dla } n = 2k-1, \end{cases}$$

- Powierzchnia sfery:

$$S_n = \frac{nV_n}{r}.$$

- Zadanie. Napisz leniwy program obliczania objętości kuli i powierzchni sfery w n wymiarach.

# O lenistwie



- *Pochwała lenistwa*
- *Heskell to leń dobrze zorganizowany (Zawłocki)*

# Literatura



- Oleg Taykalo (sieć)
- Zawłocki (sieć)
- wazniak (sieć)
- Bylina + Bylina

# Za tydzień: Prolog