Notatki Programowanie Funkcyjne

Jakub Kogut

13 maja 2025

Spis treści

1 Wstęp

Notatki z programowania funkcyjnego prowadzone przez GOATA profesora Jacka Cichonia na semestrze 4 2025. Zajęcia laboratoryjne prowadzone są przez dr Dominika Bojko.

2 Wykład 11-03-2025

Na tym wykładzie skupimy się na przygotowaniu środowiska pracy do programowania funkcyjnego w języku Haskell.

2.1 Struktura kodu w Haskell

Przykładowy kod wygląda następująco:

```
{-  file = W2.hs
    autor = JK
    date = 11-03-2025
-}
module W2 where
id' x = x
```

Następnie w terminalu, w którym mamy odpalone GHCI wpisujemy:

```
>:l W2.hs
>:r
>id' 5
5
>:t id'
id' :: a -> a //co oznacza id :: forall a => a->a
```

Co matematycznie można zapisać jako:

$$exp = (\lambda a : Typ \to (a \to a))$$

- Przykład:
  - $exp(Int) :: Int \to Int$
  - $exp(Bool) :: Bool \to Bool$
  - $exp(Double) :: Double \to Double$

Cichoń radzi, aby narpiew zastanowić się jaki powinnen być typ funkcji, a dopiero potem zastanawiać się nad implementacją, ponoć oszczędza to *czas i nerwy*.

2.2 Typy w Haskellu

- Typy proste:
  - Int
  - Double
  - Char
  - Bool
- Typy złożone:
  - Listy

- Krotki
- Funkcje

- Przykład:
  - funkcja Collatz'a $coll :: Int \to Int$

```
coll n
    | n==1 = 1
    | even n = coll (n `div` 2)
    | odd n = coll (3*n+1)
```

Symbol | oznacza wyrażenie z wykożystaniem strażników *guards*. Zapis taki jest podobny do matematycznego zapisu funkcji:

$$coll(n) = \begin{cases} 1 & \text{gdy } n = 1 \\ coll(\frac{n}{2}) & \text{gdy } n \text{ jest parzyste} \\ coll(3n+1) & \text{gdy } n \text{ jest nieparzyste} \end{cases}$$

Nie jest to bezpieczna funkcja, ponieważ dla liczb ujemnych zapętli się ona w nieskończoność. Można zauważyć, że funkcja ta zwraca zawsze liczbę 1. Ciekawa jest liczba kroków, które są potrzebne do osiągnięcia tej wartości. Dla $n = 27$ potrzeba 111 kroków, dla $n = 28$ potrzeba 18 kroków, dla $n = 29$ potrzeba 111 kroków.

  - Nowa funkcja Collatz'a
    collatz::(Int,Int) $\to$ (Int,Int)

```
collatz (n,steps)
    | n==1 = (n,steps)
    | even n = collatz (n `div` 2,steps+1)
    | odd n = collatz (3*n+1,steps+1)
```

Funkcja ta zwraca parę liczb, pierwsza to wynik funkcji Collatz'a, a druga to liczba kroków potrzebna do osiągnięcia tej wartości.
Spróbujmy ją sobie odpalić:

```
>collatz (97,0)
(1,118)
```

Jak widać dla $n = 97$ potrzeba 118 kroków, aby osiągnąć wartość 1.

  - Funkcja lenz *lenght of collatz* zwracająca długość ciągu Collatz'a dla danej liczby:
    lenz::Int $\to$ Int

```
lenz n = snd (collatz (n,0))
```

2.3 Listy

Definicja listy w Haskellu:
[a] - lista elementów typu a

$$[\text{a}] = \{[a_1, \ldots, a_k] \mid a_1, \ldots, a_k \in a, k \in \mathbb{N}\}$$

```
>:t [1,2,3]
[1,2,3] :: Num a => [a]
>:t [1::Integer, 2, 3]
[1,2,3] :: [Integer]
```

2.3.1 Operacje na listach

- Dodawanie elementu na początku listy

```
>:t (1:[2,3])
(1:[2,3]) :: Num a => [a]
```

- Konkatenacja list

```
>:t [1,2]++[3,4]
[1,2]++[3,4] :: Num a => [a]
```

2.3.2 Podstawowe funkcje operujace na listach

- length::[a] → Int
  - length [] = 0
  - length (x:xs) = 1 + length xs
- head::[a] → a
  zwraca pierwszy element listy
  - head (x:xs) = x
  - head [] = error "empty list"
- tail::[a] → [a]
  zwraca listę bez pierwszego elementu
  - tail (x:xs) = xs
  - tail [] = error "empty list"
- last::[a] → a
  zwraca ostatni element listy
  - last [x] = x
  - last (x:xs) = last xs
  - last [] = error "empty list"
- filter::(a → Bool) → [a] → [a]
  - filter p [] = []
  - filter p (x:xs) = if p x then x : filter p xs else filter p xs
  - filter (n → n>0) [-1,2,-3,4] = [2,4]
  - filter even [1..10] = [2,4,6,8,10]
    Jak zdefiniować funkcje filter:

```
filter p [] = []
filter p (x:xs)
    | p x = x : filter p xs
    | otherwise = filter p xs
```

- map::(a → b) → [a] → [b]
  zwraca listę, która powstaje zastosowaniem funkcji do każdego elementu listy
  - map f [] = []
  - map f (x:xs) = f x : map f xs

- map (n → n*n) [1,2,3] = [1,4,9]
- map (n → n^3) [1..10] = [1..1000]

gdzie **[1..10]** to skrót od **[1,2,3,4,5,6,7,8,9,10]**

2.3.3 List comprehension

Polega na tworzeniu listy na podstawie innych list.

$$\boxed{[fx_1, x_2, x_3 \mid x_1 \leftarrow xs, x_2 \leftarrow ys, x_3 \leftarrow zs]}$$

Przykład:

- chcemy stworzyć listę wszystkich trójek pitagorejskich ponizej liczby n.

```
pitagorasTrzy n = [(x,y,z) | x <- [1..n], y <- [1..n], z <- [1..n],
                   x^2 + y^2 == z^2, gcd xy == 1]
```

3 Wykład 18-03-2025

3.1 implementacje funkcji w Haskellu

3.1.1 QuickSort

w tej implementacji za *pivot* przyjmujemy pierwszy element listy.

```
qS [] = []
qs (x:xs) = (qS [y | y <- xs, y < x]) ++
            [x] ++
            (qS [y | y <- xs, y >= x])
```

3.1.2 *partition*

```
partition :: (a->Bool) -> [a] -> ([a],[a])
partition _ [] = ([],[])
partition p (x:xs) = if p x then (x:l,r)
                            else (l,x:r)
                    where (l,r) = partition p xs
```

3.1.3 Lepsza implementacja QuickSort'a

```
qSort [] = []
qSort (x:xs) = (qSort l) ++ [x] ++ (qSort r)
               where (l,r) = partition (<x) xs
```

wyrażenie `(<x)` jest zastosowaniem *slicingu*. Działa to następująco:

$$(< x) \equiv \lambda y \rightarrow y < x$$

3.1.4 InsertSort

```
inSort [] = []
inSort (x:xs) = l ++ [x] ++ r
                where sxs = inSort xs
                      (l,r) = partition (<x) sxs
```

Aby sprawdzić prędkość wykonywania funkcji w Haskellu możemy użyć następującej funkcji

```
>:set +s
>take 10 (inSort [1000,999..1])
[1,2,3,4,5,6,7,8,9,10]
(0.02 secs, 5,499,824 bytes)
```

3.1.5 *zip*

Powtórka z ćwiczen *Zadanie 10* **??**

3.1.6 *zipWith*

Podobnie znowu napisane na ćwiczeniach **??**

3.1.7 funkcje wykonujące operacje na listach

funkcja sumująca:

```
sumList [] = 0
sumList (x:xs) = x + sumList xs

> sumList [1..1000]
500500
```

funkcja mnożąca:

```
pro [] = 1
pro (x:xs) = x * pro xs

>pro [1..9]
362880
```

teraz abstrachując ten koncept możemy napisać funkcję *foldl*[1], która działa na danym monoidzie $M = (M, \cdot, e)$

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl' op e [] = e
foldl' op e (x:xs) = op x (foldl' op e xs)

ghci> foldl' (*) 1 [1..10]
3628800
ghci> foldl' (+) 1 [1..10]
56
ghci> foldl' (+) 0 [1..10]
55
ghci> foldl' (*) 0 [1..10]
0
```

[1] jest to operacja składająca liste w nastepujący sposób $(((e \cdot x_1) \cdot x_2) \cdot x_3) \cdot \ldots) \cdot x_n$, istneje analogiczna *foldr* działająca odwrotnie $x_1 \cdot (x_2 \cdot (x_3 \cdot \ldots (x_n \cdot e)))$

4 Wykład 25-03-2025

4.1 Krótki wstęp o automatach

4.1.1 Deterministyczne, skończone automaty

Jak działają automaty? Automat składa się z:

$$\mathbb{A} = (Q, \Sigma, \delta, q_0, F), \Sigma$$

gdzie Q to zbiór stanów, Σ to alfabet, δ to funkcja przejścia, q_0 to stan początkowy, a F to zbiór stanów końcowych.

$$\delta : Q \times \Sigma \to Q$$

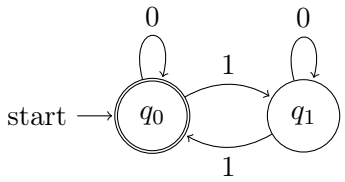
$$q_0 \in Q, F \subseteq Q$$

Jak wygląda to w Haskellu:

```
delta :: s -> c -> s
acc :: s -> Bool

runDFA :: (s -> c -> s) -> s -> [c] -> s -- run Deterministic Finite Automat
runDFA delta start cs = foldl delta start cs
runDFA = foldl -- pożądniejsza definicja
```

Zobaczmy działanie na prostym przykładzie automatu sprawdzającego parzystość liczby binarnej:



Można odpalić ten automat w następujący sposób:

```
>runDFA delta q0 "10101"
False
>runDFA delta q0 "101010"
True
```

4.1.2 Niedeterministyczne, skończone automaty

Automat niedeterministyczny składa się z:

$$\mathbb{A} = (S, \delta, s_0, F)$$

gdzie S to zbiór stanów, δ to funkcja przejścia, s_0 to stan początkowy, a F to zbiór stanów końcowych.

$$\delta : S \times \Sigma \to \mathcal{P}(S)$$

5 Wykład 2025-04-01

Dzisiaj będziemy omawiać strukturę rekordów w języku Haskell

5.1 Struktura

Weźmy na przykład strukturę reprezentującą osobę:

```
dataOsoba = {
    idO :: Int,
    imie :: String,
    nazwisko :: String,
    rokUrodzenia :: Int,
    miesiacUrodzenia :: Int,
    dzienUrodzenia :: Int
} deriving (Show)
```

deklaracja rekordu w Haskellu wygląda następująco:

```
aaa = Osoba {
    idO = 1,
    imie = "Jan",
    nazwisko = "Kowalski",
    rokUrodzenia = 2000,
    miesiacUrodzenia = 1,
    dzienUrodzenia = 1
}
```

Poprzez *deriving (Show)* mówimy, że chcemy aby nasz rekord mógł być wyświetlany w konsoli. Haskel automatycznie tworzy te funkcje dla nas.

```
>dzienUrodzenia aaa
1
>aaa {dzienUrodzenia = 2}
Osoba {idO = 1, imie = "Jan", nazwisko = "Kowalski", rokUrodzenia = 2000, miesiacUrodzenia = 1,
```

Ale jest to nie elegancki sposób deklaracji rekordu, jeżeli chodzi o typ daty. Możemy zaciągnąć z jakiegoś modułu gotowy typ danych:

```
import Data.Time
data Osoba = {
    idO :: Int,
    imie :: String,
    nazwisko :: String,
    dataUrodzenia :: Date
} deriving (Show)
```

Napiszmy teraz funkcję zmienająca rok urodzin, kożystając z tego co zostało napisane do tej pory

```
zmienRok :: Osoba -> Int -> Osoba
zmienRok osoba nowyRok =
    osoba {dataUrodzenia = (dataUrodzenia osoba) {year = nowyRok}}
```

Poźniej w ramach tego typu problemów omówimy typ *lenses*.

5.2 Typy parametryzowalne

Zobaczmy najpierw jak wygląda zapis pary w jezyku matematyki:

$$P(X) = X \times X$$

$$f : \text{Int} \to \text{String}$$
$$\tilde{f} : \text{Int} \times \text{Int} \to \text{String} \times \text{String}$$
$$\tilde{f}(x, y) = (f(x), f(y))$$

5.3 Funkturo Maybe

Myślimy o takim przekształceniu MB od słowa Mayby:

$$MB(X) = X \cup \{\uparrow_X\}$$

gdzie $\uparrow_X$ to symbol oznaczający brak wartości *Nothing*, definujemy to tak:

$$\uparrow_X \notin X$$
$$\uparrow_X \notin \text{Maybe}(X)$$
$$\uparrow_X \notin \text{Maybe}(X) \setminus X$$
$$XY \implies \uparrow_X \neq \uparrow_Y$$

Możemy wykożystać tą konstrukcje w Haskellu:

```
safeHead :: [a] -> Maybe a
safeHead [] = Nothing
safeHead (x:_) = Just x
```

```
safeSqrt :: Double -> Maybe Double
safeSqrt x
    | x < 0 = Nothing
    | otherwise = Just (sqrt x)
```

```
safeLog :: Double -> Maybe Double
safeLog x
    | x <= 0 = Nothing
    | otherwise = Just (log x)
```

```
safeDiv x 0 = Nothing
safeDiv x y = Just (x/y)
```

Teraz zaczynamy partyzanthe z *Maybe*:

```
composeMB :: (a -> Maybe b) -> (b -> Maybe c) -> (a -> Maybe c)
composeMB f g x = case f x of
    Nothing -> Nothing
    Just y -> g y
```

Zobaczmy to teraz na przykładzie

```
expr1 = composeMB safeLog safeSqrt

>expr1 (exp 1)
Just 1.0
>expr1 ((expr 1)^4)
Just 2.0
>expr1 0.5
Nothing
```

Inny przykład:

```
composeMB2 :: (a -> b -> Maybe c) -> Maybe a -> Maybe b -> Maybe c
composeMB2 _ Nothing _ = Nothing
composeMB2 _ _ Nothing = Nothing
composeMB2 f (Just x) (Just y) = f x y
```

I na przykładzie

```
expr2 x = let sn = safeSqrt x
              ln = safeLog (x^2-4)
          in composeMB2 safeDiv sn ln
```

```
>expr2 5
Just 0.7344560676556667
>expr2 2
Nothing
```

Fachowo mówi się, że zanurzamy te funkcje w konstruktor *Maybe*.

5.4 Funktor

Zastanówmy się nad konstruktorem typów $F : \text{Set} \to \text{Set}$. Kilka przykładów takich konstruktorów:

$$\mathcal{P}(X) = X \times X$$

$$MB(X) = X \cup \{\uparrow_X\}$$

$$L(X) = [X]$$

Funktor: taki konstruktor typów F, że dla każdego $f : X \to Y$ istneje funkcja $F[f] : F(X) \to F(Y)$, która spełnia następujące warunki:

- $F[\text{id}_X] = \text{id}_{F(X)}$
- $F[f \circ g] = F[f] \circ F[g]$

Przykłady funktorów:

- $$f : X \to Y$$
  $$\tilde{\mathcal{P}}[f](x_1, x_2) = (f(x_1), f(x_2))$$
  jak to działa?
  $$\tilde{\mathcal{P}}(X) \xrightarrow{\tilde{\mathcal{P}}[f]} \tilde{\mathcal{P}}(Y)$$
- $$f : X \to Y$$
  $$MB[f]$$
  jak to działa?

zastąpmy `pmap` bardziej ogólnym pojęciem *funktor*:

```
instance Functor Para where
    fmap f (Para (x,y)) = Para (f x, f y)
```

5.5 Big Data – "Hello World"

Rozważmy problem policzenia najczęściej występujących słów w tekście. Mamy podany tekst, który interpretujemy jako *String*. Napiszmy taki program i po drodzę zastanawiajmy się co bedzie potrzebne

```
module HelloBDWorld where

import Data.List (words, group, sort, sortBy)
import Data.Char (toLower)
import SWEng -- StopWordsEnglish
import Control.Monad
```

należy usunąć z tekstu coś co nazywa się *stop-words* np to, anything itp., oraz inne śmieci np znaki interpunkcyjne oraz w pewnym momencie jednoliterówki
Rownież należy zamienić wszystkie wielkie litery na małe.

```
oczyscTxt :: String [String]
oczyscTxt = map (map toLower) . filter (not . flip elem stopWords) . words
```

podziel słowa na grupy i posortuj je

```
grupuj :: [String] -> [(String, Int)]
grupuj = map (\xs -> (head xs, length xs)) . group . sort
```

posortuj słowa według ilości ich wystąpień

```
sortuj :: [(String, Int)] -> [(String, Int)]
sortuj = sortBy (\(_,n1) (_,n2) -> compare n2 n1)
```

i na koniec podaj liste najczestrzych słow

```
najczestsze :: String -> [(String, Int)]
najczestsze = sortuj . grupuj . oczyscTxt
```

```
> najczestsze "Hello World! Hello Haskell!"
[("hello",2),("world",1),("haskell",1)]
```

6 Wykład 2025-04-08

6.1 Elementy teorii kategorii

6.1.1 Definicja Kategorii

Kategoria $\mathcal{C}$ składa się z:

- obiekty $ob(\mathcal{C})$
- morfizmy $mor(\mathcal{C})$:

$$f \in mor(\mathcal{C}) \implies dom(f), codom(f) \in ob(\mathcal{C})$$

zapisujemy to jako:

$$\mathcal{C} \models (f : A \to B)$$

albo

$$\mathcal{C} \models (A \xrightarrow{f} B)$$

- złożenie morfizmów:

$$\mathcal{C} \models (f : A \to B) \wedge (g : B \to C) \implies g \circ f : A \to C$$

  Przy założeniu, że złożenie $\circ$ jest łączne

- identyczność morfizmu:

$$\forall X \in ob(\mathcal{C}) : \exists id_X : X \to X$$

  jest dokładnie jeden morfizm, który jest identycznością. Spełnia on warunki:

  –

$$\forall f : A \to B \implies id_B \circ f = f$$

  –

$$\forall f : A \to B \implies f \circ id_A = f$$

- *Set*:
  - obiekty $\equiv$ zbiory
  - morfizmy $\equiv (A, f, B)$, gdzie $f : A \to B$ jest funkcją
- *Grp*:
  - obiekty $\equiv$ grupy
  - morfizmy $\equiv (\mathcal{G}_1, \varphi, \mathcal{G}_2)$, gdzie $\varphi : \mathcal{G}_1 \xrightarrow{homomorfizm} \mathcal{G}_2$ jest homomorfizmem grup
- *Cat*($\mathcal{X}$):
  - obiekty $\equiv$ elementy $\mathcal{X}$
  - morfizmy $\equiv \left(\left(x \xrightarrow{\alpha} y\right) \equiv x \leq y\right)$ oraz $\left(x \xrightarrow{\beta} y, y \xrightarrow{\beta} z\right) \equiv \left(x \xrightarrow{\alpha\circ\beta} z\right)$

Def

Obiekt c jest końcowy w $\mathcal{C}$ wtedy i tylko wtedy, gdy

$$\forall X \in ob(\mathcal{C}) : \exists! f : X \to c$$

Def

d jest początkowy w $\mathcal{C}$ wtedy i tylko wtedy, gdy

$$\forall X \in ob(\mathcal{C}) : \exists! f : d \to X$$

W *Set* jeśli element jest początkowym, to jest to $\emptyset$

Def

Kategornia dualna
Niech $\mathcal{C}$–kategoria, wtedy $\mathcal{C}^{op}$, gdzie

$$\mathcal{C}^{op} \models (x \xrightarrow{f} y) \equiv \mathcal{C} \models (y \xrightarrow{f} x)$$

Inaczej

$$f \circ^{op} g \equiv g \circ f$$

6.2 Typy parametryzowalne

Def

Mamy

$$F : ob(\mathcal{C}) \to ob(\mathcal{D})$$
$$F : mor(\mathcal{C}) \to mor(\mathcal{D})$$

F jest funktorem, jeśli spełnia następujące warunki:

- $\mathcal{C} \models X \xrightarrow{f} Y \implies \mathcal{D} \models F(X) \xrightarrow{F(f)} F(Y)$
- $F(f \circ g) = F(f) \circ F(g)$
- $F(id_X) = id_{F(X)}$

7 Wykład 2025-04-15

7.1 Podstawowe konstrkorów typów danych

teoria mnogości	Haskell
$A \times B^2$	(a, b)
$A + B$	$La \vert Rb$
B^A	$a \to b$

Przykład

```
data R a b = Reader(a -> b)
```

Możemy tutaj pomyśleć o takim funktorze:

$$F_A(X) = X^A \leftarrow \text{funktor}$$

```
data CR a b  = Reader(b -> a)
```

Jest to w żargonie haskellowskim nazywane predykatem Czyli:

$$CR_A(X) = A^X$$

8 Ćwiczenia

W tym miejscu będa pojawiały się notatki z laboratorów (ćwiczeń)

8.1 Ćwiczenia 11-03-2025

8.1.1 Zadanie 1

```
power :: Int => Int => Int
power x y = y ^ x

p2 = power 4
p3 = power 3
```

1. Wyznacz w GCHI wartość wyrażenia $(p2 \circ p3)^2$ i wyjaśnij, dlaczego otrzymałeś ten wynik.
2. Zbadaj typy funkcji $p2$, $p3$ i $(p2 \circ p3)$.

3. Zapisz powyższe funkcje za pomocą wyrażeń lambda.

$$Int \to Int \to Int$$

Zapis strzałkowy definuje nam typ funkcji operacja => jest wiążaca z prawej strony, wiec można by było to również zapisać jako:

$$power :: Int \to (Int \to Int)$$

1. podpunkt 1

$$(p2 \circ p3)^2 = p2(p3(x))^2 = 4(3^x)^2 = 4 \cdot 9^x$$

2. podpunkt 2

```
>:t p2
p2 :: Int -> Int
>:t p3
p3 :: Int -> Int
>:t (p2 . p3)
(p2 . p3) :: Int -> Int
```

3. podpunkt 3

```
p2 = \x -> power 4 x
p3 = \x -> power 3 x
```

8.1.2 Zadanie 2

$$2 \wedge 3 \wedge 2, \quad (2 \wedge 3) \wedge 2, \quad 2 \wedge (2 \wedge 3).$$

Dowiedz się, jaka jest łączność oraz siła operatora $\wedge$ za pomocą polecenia:

$$: i(\wedge).$$

```
>:i (^)
(^) :: (Num a, Integral b) => a -> b -> a
        -- Defined in ‘GHC.Real’
infixr 8 ^
```

Operator $\wedge$ jest prawostronnie łączny, a jego siła wynosi 8 (najwyższa możliwa wartość, wyłącznie wyższe jest nałożenie funkcji na zmienną). W nawiasie **Num a, Integraf b** oznacza, że operator $\wedge$ bierze jeden argument typu **Num** i drugi typu **Integral**.

$$2 \wedge 3 \wedge 2 = 2 \wedge (3 \wedge 2) = 2 \wedge 9 = 512$$

$$(2 \wedge 3) \wedge 2 = 8 \wedge 2 = 64$$

$$2 \wedge (2 \wedge 3) = 2 \wedge 8 = 256$$

8.1.3 Zadanie 3

```
f : : Int => Int
f x = x ^ 2
g : : Int => Int => Int
g x y = x+2*y
h : : . . . .
h x y = f ( g x y )
```

1. Jaki jest typ funkcji h? (tzn. uzupełnij ... w powyższym listingu)
2. Czy $h = f \circ g$?
3. Czy $h\,x = f(g\,x)$?

---

1. Typ funkcji h to:
$$h :: Int \to Int \to Int$$
2. Nie, ponieważ:
$$h(x, y) = f(g(x, y)) = f(x + 2y) = (x + 2y)^2$$
3. Tak, ponieważ:
$$h(x) = f(g(x)) = f(x + 2x) = (x + 2x)^2 = 9x^2$$

8.1.4 Zadanie 4

Zapisz operacje binarne $(+)$, $(*)$ za pomocą lambda wyrażeń.

```
add = \x -> (\y -> x + y)
mul = \x -> (\y -> x * y)
```

Co to daje? Mozna teraz zapisać 2+3 jako:

- add 2 3
- (add 2) 3
- add 2 (3)
- ($3) (2 add)

8.1.5 Zadanie 5

Zapisz funkcje:

$$f(x) = 1 + x \cdot (x + 1), \quad g(x, y) = x + y^2, \quad h(y, x) = x + y^2$$

za pomocą lambda wyrażeń w językach C++, Python, JavaScript oraz Haskell.

W języku Haskell:

```
f = \x -> 1 + x * (x + 1)
g = \x -> \y -> x + y^2
h = \y -> \x -> x + y^2
```

W języku Python:

```
f = lambda x: 1 + x * (x + 1)
g = lambda x, y: x + y**2
h = lambda y, x: x + y**2
```

W języku JavaScript:

```
f = x => 1 + x * (x + 1)
g = (x, y) => x + y**2
h = (y, x) => x + y**2
```

W języku C++:

```
auto f = [](int x) { return 1 + x * (x + 1); };
auto g = [](int x, int y) { return x + y*y; };
auto h = [](int y, int x) { return x + y*y; };
```

8.1.6 Zadanie 6

Ustalmy zbiory A, B, C. Niech

$$\text{curry} : C^{B\times A} \to (C^B)^A$$

będzie funkcją zadaną wzorem:

$$\text{curry}(\varphi) = \lambda a \in A \to (\lambda b \in B \to \varphi(b, a)).$$

oraz niech

$$\text{uncurry} : (C^B)^A \to C^{B\times A}$$

będzie zadana wzorem:

$$\text{uncurry}(\psi)(b, a) = (\psi(a))(b).$$

1. Pokaż, że $\text{curry} \circ \text{uncurry} = \text{id}_{(C^B)^A}$ oraz $\text{uncurry} \circ \text{curry} = \text{id}_{C^{B\times A}}$.
2. Wywnioskuj z tego, że $|(C^B)^A| = |C^{B\times A}|$. Przypomnij sobie dowód tego twierdzenia, który poznałeś na pierwszym semestrze studiów.
3. Spróbuj zdefiniować w języku Haskell odpowiedniki funkcji `curry` i `uncurry`.

---

1. Pokażemy, że $\text{curry} \circ \text{uncurry} = \text{id}_{(C^B)^A}$ oraz $\text{uncurry} \circ \text{curry} = \text{id}_{C^{B\times A}}$.

   - $\text{curry} \circ \text{uncurry}$

   $$\begin{aligned}(\text{curry} \circ \text{uncurry})(\psi) &= \text{curry}(\text{uncurry}(\psi)) \\ &= \text{curry}(\lambda a \in A \to (\lambda b \in B \to \psi(a)(b))) \\ &= \lambda a \in A \to (\lambda b \in B \to \psi(a)(b)).\end{aligned} \tag{1}$$

   - $\text{uncurry} \circ \text{curry}$

   $$\begin{aligned}(\text{uncurry} \circ \text{curry})(\varphi) &= \text{uncurry}(\text{curry}(\varphi)) \\ &= \text{uncurry}(\lambda a \in A \to (\lambda b \in B \to \varphi(b, a))) \\ &= \lambda b \in B \to (\lambda a \in A \to \varphi(b, a)).\end{aligned} \tag{2}$$

Z powyższych równań wynika, że $\text{curry} \circ \text{uncurry} = \text{id}_{(C^B)^A}$ oraz $\text{uncurry} \circ \text{curry} = \text{id}_{C^{B \times A}}$. □

2. Możemy pokazać że `curry` i `uncurry` są iniekcjami niewprost, nakładając odpowiednio przeciwne funkcje na obie strony równości:

   - Załóżmy, że $\text{curry}(\varphi_1) = \text{curry}(\varphi_2)$. Wtedy:

   $$\begin{aligned} \text{curry}(\varphi_1)(a)(b) &= \text{curry}(\varphi_2)(a)(b) \\ \varphi_1(b, a) &= \varphi_2(b, a) \\ \varphi_1 &= \varphi_2. \end{aligned} \tag{3}$$

   - Załóżmy, że $\text{uncurry}(\psi_1) = \text{uncurry}(\psi_2)$. Wtedy:

   $$\begin{aligned} \text{uncurry}(\psi_1)(b, a) &= \text{uncurry}(\psi_2)(b, a) \\ \psi_1(a)(b) &= \psi_2(a)(b) \\ \psi_1 &= \psi_2. \end{aligned} \tag{4}$$

   A więc istnieje biekcja między $(C^B)^A$ i $C^{B \times A}$, co oznacza, że te zbiory mają taką samą moc. □

3. W języku Haskell funkcje `curry` i `uncurry` można zdefiniować następująco:

```
curry :: ((b, a) -> c) -> a -> b -> c
curry f x y = f (y, x)
```

```
uncurry :: (a -> b -> c) -> (a, b) -> c
uncurry f (x, y) = f x y
```

8.1.7 Zadanie 7

Podaj przykłady funkcji następujących typów:

$$(\text{Int} \Rightarrow \text{Int}) \Rightarrow \text{Int}$$

$$(\text{Int} \Rightarrow \text{Int}) \Rightarrow (\text{Int} \Rightarrow \text{Int})$$

$$(\text{Int} \Rightarrow \text{Int}) \Rightarrow (\text{Int} \Rightarrow \text{Int}) \Rightarrow (\text{Int} \Rightarrow \text{Int})$$

- Funkcja typu $(\text{Int} \Rightarrow \text{Int}) \Rightarrow \text{Int}$:

```
f :: (Int -> Int) -> Int
f g = g 0
```

- Funkcja typu $(\text{Int} \Rightarrow \text{Int}) \Rightarrow (\text{Int} \Rightarrow \text{Int})$:

```
f :: (Int -> Int) -> (Int -> Int)
f g x = g (g x)
```

- Funkcja typu $(\text{Int} \Rightarrow \text{Int}) \Rightarrow (\text{Int} \Rightarrow \text{Int}) \Rightarrow (\text{Int} \Rightarrow \text{Int})$:

```
f :: (Int -> Int) -> (Int -> Int) -> (Int -> Int)
f g h x = g (h x)
```

8.1.8 Zadanie 8

Załóżmy, że chcesz oprogramować funkcję, która dla danych liczb a, b oraz funkcji $f : \mathbb{R} \to \mathbb{R}$ oblicza

$$\int_a^b f(x)\,dx.$$

Jaki powinien być typ tej funkcji?

Typ tej funkcji powinien być następujący:

$$\text{Num(a)} \implies a \to a \to (a \to a) \to a$$

mogłaby ona wyglądać następująco:

```
integral :: (Double -> Double) -> Double -> Double -> Double
integral f a b = undefined
```

8.1.9 Zadanie 9 – (Eliminacja Pętli)

Wybierz jeden z języków Python, C++ lub JavaScript.

1. Masz daną (czyli oprogramowaną) funkcję $f : \mathbb{N} \to \mathbb{N}$. Oprogramuj funkcję, która dla danego $n \in \mathbb{N}$ oblicza
$$\sum_{k=0}^{n} f(k).$$
Zrób to najpierw (standardowo) za pomocą pętli, a potem oprogramuj ją bez użycia pętli, za pomocą rekursji.

2. Rozważamy następującą funkcję napisaną w pseudokodzie:
```
FUNCTION f(x: DOUBLE): DOUBLE
BEGIN
    DOUBLE y = sin(x);
    RETURN y*y + y + x;
ENDFNC
```
Oprogramuj tę funkcję w wybranym języku i następnie wyeliminuj zmienną lokalną y z tego kodu, bez pogarszania jego efektywności.

---

1. Oto rozwiązanie w języku C++:
```
#include <iostream>
using namespace std;

// Example implementation of function f: N -> N.
// You can replace this with any function of type int -> int.
int f(int x) {
    // Example: f(x) = x + 1
    return x + 1;
}

// Function that sums using a loop:
```

```
int sumLoop(int n) {
    int sum = 0;
    for (int k = 0; k <= n; ++k) {
        sum += f(k);
    }
    return sum;
}

// Function that sums using recursion:
int sumRec(int n) {
    if(n == 0)
        return f(0);
    else
        return sumRec(n - 1) + f(n);
}

int main() {
    int n;
    cout << "Enter n: ";
    cin >> n;
    cout << "Sum computed with loop: " << sumLoop(n) << endl;
    cout << "Sum computed recursively: " << sumRec(n) << endl;
    return 0;
}
```

funkcja `sumLoop` oblicza sumę za pomocą pętli, a funkcja `sumRec` oblicza sumę rekurencyjnie.

$$\text{sumRec}(n) = \begin{cases} f(0) & \text{gdy } n = 0 \\ \text{sumRec}(n-1) + f(n) & \text{gdy } n > 0 \end{cases}$$

2. Rozwiazanie w Haskellu:

```
f :: Double -> Double
f x = sin x * sin x + sin x + x
```

```
f' :: Double -> Double
f' x = sin x * sin x + sin x + x
```

8.1.10 Zadanie 10

Zaimplementuj samodzielnie następujące funkcje działające na listach z Prelude:

1. map
2. zip
3. zipWith
4. filter
5. take
6. drop
7. fib

1. Funkcja `map`:

```
map' :: (a -> b) -> [a] -> [b]
map' f [] = []
map' f (x:xs) = f x : map' f xs
```

2. Funkcja `zip`[3]:

```
zip' :: [a] -> [b] -> [(a, b)]
zip' [] _ = []
zip' _ [] = []
zip' (x:xs) (y:ys) = (x, y) : zip' xs ys
```

3. Funkcja `zipWith`[4]:

```
zipWith' :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith' _ [] _ = []
zipWith' _ _ [] = []
zipWith' f (x:xs) (y:ys) = f x y : zipWith' f xs ys
```

4. Funkcja `filter`[5]:

```
filter' :: (a -> Bool) -> [a] -> [a]
filter' _ [] = []
filter' p (x:xs)
    | p x = x : filter' p xs
    | otherwise = filter' p xs
```

5. Funkcja `take`[6]:

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

6. Funkcja `drop`[7]:

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

7. Funkcja `fib`[8]:

[3]Funkcja `zip` zwraca listę par, które są złożone z elementów listy wejściowej. Jeśli jedna z list jest krótsza, to wynikowa lista będzie miała długość krótszej z nich.
Przykład: `zip [1,2,3] ['a','b','c','d']` zwróci `[(1,'a'),(2,'b'),(3,'c')]`.

[4]Funkcja `zipWith` działa podobnie jak `zip`, ale zamiast zwracać parę elementów, zwraca wynik funkcji, która jest podana jako argument.
Przykład: `zipWith (+) [1,2,3] [4,5,6]` zwróci `[5,7,9]`.

[5]Funkcja `filter` zwraca listę elementów, które spełniają warunek podany jako argument.
Przykład: `filter even [1..10]` zwróci `[2,4,6,8,10]`.

[6]Funkcja `take` zwraca listę składającą się z n pierwszych elementów listy wejściowej.
Przykład: `take 3 [1,2,3,4,5]` zwróci `[1,2,3]`.

[7]Funkcja `drop` zwraca listę, która jest wynikiem usunięcia n pierwszych elementów z listy wejściowej.
Przykład: `drop 3 [1,2,3,4,5]` zwróci `[4,5]`.

[8]Funkcja `fib` zwraca listę liczb Fibonacciego do n-tego elementu.

```
fib :: Int -> [Int]
fib n = take' n (map' fib' [0..])
    where
        fib' 0 = 0
        fib' 1 = 1
        fib' n = fib (n - 1) + fib (n - 2)
```

Fajne złożenie funkcji `fib` z `zipWith`:

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

co pozwala na generowanie listy liczb Fibonacciego w nieskończoność. Na przykład `take 10 fib` zwróci `[0,1,1,2,3,5,8,13,21,34]`.

8.1.11 Zadanie 11

Niech $f = (2^\wedge)$ oraz $g = (\wedge 2)$. Podaj interpretację tych funkcji.

Sprawdź wartości wyrażenia:

$$\text{map } (\wedge 2)[1..10] \quad \text{oraz} \quad \text{map } (2^\wedge)[1..10]$$

i wyjaśnij otrzymane wyniki.

Funkcja $f = (2 \wedge)$ podnosi liczbę do kwadratu, a funkcja $g = (\wedge 2)$ podnosi 2 do potęgi danej liczby.

```
> map (^ 2) [1..10]
[1,4,9,16,25,36,49,64,81,100]
> map (2 ^) [1..10]
[2,4,8,16,32,64,128,256,512,1024]
```

8.1.12 Zadanie 12

Dowiedz się, jak można przekonwertować elementy typu `Int` oraz `Integer` na typy `Float` i `Double`. Dowiedz się, jaki jest format funkcji typu `round` z `Double` do `Int`.

- Konwersja z `Int` na `Float`:

  ```
  fromIntegral :: (Integral a, Num b) => a -> b
  ```

- Konwersja z `Int` na `Double`:

  ```
  fromIntegral :: (Integral a, Num b) => a -> b
  ```

- Konwersja z `Integer` na `Float`:

  ```
  fromInteger :: Num a => Integer -> a
  ```

- Konwersja z `Integer` na `Double`:

  ```
  fromInteger :: Num a => Integer -> a
  ```

- Funkcja `round` z `Double` na `Int`:

  ```
  round :: (RealFrac a, Integral b) => a -> b
  ```

8.2 Elementy Teorii Liczb

Trochę teorii liczb, bo czemu nie?

8.2.1 Zadanie 13

Funkcję Eulera φ nazywamy funkcją określoną wzorem:

$$\varphi(n) = \text{card}\left(\{k \leq n : \gcd(k, n) = 1\}\right), \tag{5}$$

o dziedzinie $\mathbb{N}^+$.

1. Oprogramuj funkcję φ (funkcja gcd jest dostępna w bibliotece `Prelude`).
2. Napisz funkcję, która dla danej liczby naturalnej n wyznacza sumę:

$$\sum_{k|n} \varphi(k).$$

---

1. Oto implementacja funkcji φ w języku Haskell:

```
phi :: Int -> Int
phi n = length [k | k <- [1..n], gcd k n == 1]

> phi 10
4
```

2. Oto implementacja funkcji, która wyznacza sumę $\sum_{k|n} \varphi(k)$:

```
sumPhi :: Int -> Int
sumPhi n = sum [phi k | k <- [1..n], n `mod` k == 0]

> sumPhi 10
10
```

Funkcja sumPhi jest identycznoscia na $\mathbb{N}^+$. Można zapisać to jako:

$$n = N(n) = \sum_{k|n} \varphi(k) \cdot \underbrace{I(\frac{n}{k})}_{\equiv 1}$$

8.2.2 Zadanie 14

Liczbę naturalną n nazywamy *doskonałą*, jeżeli spełnia warunek:

$$n = \sum \{d : 1 \leq d < n \wedge d \mid n\}. \tag{6}$$

Na przykład liczba 6 jest liczbą doskonałą, ponieważ:

$$6 = 1 + 2 + 3. \tag{7}$$

Wyznacz wszystkie liczby doskonałe mniejsze od 10 000.
Uwaga: Do tej pory nie wiadomo, czy istnieje nieskończenie wiele liczb doskonałych.

Oto implementacja funkcji, która znajduje wszystkie liczby doskonałe mniejsze od 10 000:

```
isPerfect :: Int -> Bool
isPerfect n = n == sum [d | d <- [1..n-1], n `mod` d == 0]

perfectNumbers :: [Int]
perfectNumbers = [n | n <- [1..9999], isPerfect n]

> perfectNumbers
[6,28,496,8128]
```

8.2.3 Zadanie 15

Parę liczb naturalnych (m, n) nazywamy *zaprzyjaźnionymi*, jeżeli suma dzielników właściwych każdej z nich równa się drugiej:

$$\sigma(m) - m = n, \quad \sigma(n) - n = m,$$

gdzie $\sigma(n)$ oznacza sumę wszystkich dzielników liczby n.

Znajdź wszystkie zaprzyjaźnione pary, których oba składniki są mniejsze od 10^5.

Uwaga: Do tej pory nie wiadomo, czy istnieje nieskończenie wiele par liczb zaprzyjaźnionych.

Tak może wyglądać funkcja szukająca liczb zaprzyjaźnionych w podanym zakresie:

```
sumaDzielnikow :: Int -> Int
sumaDzielnikow 1 = 0
sumaDzielnikow n = 1 + sum [ if x * x == n then x
                             else if n `mod` x == 0 then x + (n `div` x)
                             else 0
                             | x <- [2..(floor . sqrt . fromIntegral) n], x * x <= n ]
    where
        limit = (floor . sqrt . fromIntegral) n

amicablePairs :: [(Int, Int)]
amicablePairs = [ (n, m)
                | n <- [2..maxVal-1]
                , let m = sumaDzielnikow n
                , m > n, m < maxVal
                , sumaDzielnikow m == n ]

> amicablePairs
[(220,284),(1184,1210),(2620,2924),(5020,5564),(6232,6368)]
```

8.2.4 Zadanie 16

Dla $n \in \mathbb{N}^+$ definiujemy:

$$\mathrm{dcp}(n) = \frac{1}{2n^2} \left|\{(k, l) \in \{1, \ldots, n\} : \gcd(k, l) = 1\}\right|. \tag{8}$$

1. Zaimplementuj tę funkcję w języku Haskell za pomocą *list comprehension*.
2. Zoptymalizuj ten kod, pisząc rekurencyjną wersję tej funkcji.
3. Wyznacz wartości tej funkcji dla $n = 100, 200, 300, \ldots, 10000$ i postaw jaką rozsądną hipotezę o:

$$\lim_{n \to \infty} \mathrm{dcp}(n). \tag{9}$$

1. Przykładowa implementacja przy użyciu *list comprehension*:

```
dcp :: Int -> Double
dcp n = 1 / (fromIntegral (n^2))
        * fromIntegral (length [(k, l)
                                | k <- [1..n], l <- [1..n], gcd k l == 1])

> dcp 10
0.315
```

2. Optymalizacja kodu przy użyciu rekurencji:

```
dcp' :: Int -> Double
dcp' n = 1 / (fromIntegral (n^2)) * fromIntegral (dcp'' n 1 1)

dcp'' :: Int -> Int -> Int -> Int
dcp'' n k l
    | k > n     = 0
    | l > n     = dcp'' n (k + 1) 1
    | gcd k l == 1 = 1 + dcp'' n k (l + 1)
    | otherwise = dcp'' n k (l + 1)

> dcp' 10
0.315
```

3. Wyznaczenie wartości funkcji dla $n = 100, 200, 300, \ldots, 10000$:

```
dcpValues :: [Double]
dcpValues = [dcp' n | n <- [100, 200..10000]]

> dcpValues
[0.6087,0.611575,0.6088333333333333,0.60846875,0.608924,
0.6083305555555556,0.608234693877551,0.6085921875,0.6082111111111111,
0.608383,0.6084586776859504,0.6080354166666667,0.6080988165680473,
0.6082525510204081,0.6081613333333333,0.607993359375,0.6083678200692042,
0.6080601851851852,0.6080096952908588,0.60829375,0.60808231292517,
0.6079518595041322,0.6081570888468809,0.6081019097222222,0.6079608,
0.6081087278106508,0.6080426611796982,0.6079876275510203,
0.6081092746730083,0.6080416666666667,0.6080440166493236,
0.60806005859375,0.6079602387511478,0.6079895328719723,
0.6080508571428571,0.6080030092592593,0.6080139517896275,
...]
```

Na podstawie uzyskanych wartości można postawić hipotezę, że granica funkcji $\mathrm{dcp}(n)$ dla $n \to \infty$ wynosi około $0.608\ldots$. Wartość ta może być przybliżona do wartości funkcji Eulera $\frac{6}{\pi^2}$. □
Można to interpretować jako gęstość liczb względnie pierwszych w zbiorze $\{1, \ldots, n\}$, ale jako iż bierzemy symetrie względem 0 to wartość ta jest podwojona. Jednocześnie wynika to bezpośrednio z funkcji Zeta Riemanna $\zeta(2) = \frac{\pi^2}{6}$.

8.3 Listy – część 1.

Na początku tych zadań należało zastanowić się nad implementacją istniejących już funkcji z Prelude, a następnie zaimplementować je samodzielnie.

8.3.1 Zadanie 17

Napisz funkcję `nub`, która usunie z listy wszystkie duplikaty, np.

```
nub [1,1,2,2,2,1,4,1] == [1,2,4]
```

Oto implementacja funkcji `nub` w języku Haskell:

```
nub' :: (Eq a) => [a] -> [a]
nub' [] = []
nub' (x:xs) = x : nub' (filter (/= x) xs)
```

Jak to działa?

1. Jeśli lista pusta to zwróć pustą

   ```
   nub' [] = []
   ```

2. W przeciwnym przypadku zwróć listę, której pierwszym elementem jest pierwszy element listy wejściowej (`x:xs` dzieli listę – wyciąga pierwszy element), a resztę listy tworzy rekurencyjne wywołanie funkcji `nub'` na liście, z której usunięto wszystkie wystąpienia pierwszego elementu (`filter (/= x) xs` usuwa z xs wszystko co jest `x`).

   ```
   nub' (x:xs) = x : nub' (filter (/= x) xs)
   ```

8.3.2 Zadanie 18

Napisz funkcję `inits`, która dla danej listy wyznaczy listę wszystkich jej odcinków początkowych, np.

```
inits [1,2,3,4] == [[],[1],[1,2],[1,2,3],[1,2,3,4]]
```

Funkcja `inits` również powinna wykonywać się rekurencyjnie. Zabieramy po jednym elemencie i wpisujemy do listy.

```
inits' :: [a] -> [[a]]
inits' [] = [[]]
inits' (x:xs) = [] : map (x:) (inits' xs)
```

Działa to w bardzo podobny sposób jak poprzednie:

1. Jeśli lista pusta to zwróć pustą listę

   ```
   inits' [] = [[]]
   ```

2. W przeciwnym przypadku narzuć mapą na wszystkie elementy listy rekurencyjne wywołanie funkcji `inits'` na liście bez pierwszego elementu, a następnie dodaj na początek każdej z tych list pierwszy element listy wejściowej

   ```
   inits' (x:xs) = [] : map (x:) (inits' xs)
   ```

8.3.3 Zadanie 19

Napisz funkcję `tails`, która dla danej listy wyznaczy listę wszystkich jej odcinków początkowych, np.:

```
tails [1,2,3,4] == [[],[4],[3,4],[2,3,4],[1,2,3,4]]
```

Funkcja `tails` działa analogicznie do funkcji `inits`, ale zamiast zdejmować elementy z początku listy, zdejmuje je z końca.

```
tails' :: [a] -> [[a]]
tails' [] = [[]]
tails' (x:xs) = (x:xs) : tails' xs
```

1. Jeśli lista pusta to zwróć pustą listę

   ```
   tails' [] = [[]]
   ```

2. W przeciwnym przypadku zwróć listę, której pierwszym elementem jest cała lista wejściowa, a resztę listy tworzy rekurencyjne wywołanie funkcji `tails'` na liście bez pierwszego elementu

   ```
   tails' (x:xs) = (x:xs) : tails' xs
   ```

8.3.4 Zadanie 20

Napisz funkcję `splits`, która dla danej listy xs wyznaczy listę wszystkich par (ys, zs) takich, że

```
xs == ys++zs
```

Funkcja `splits` powinna zwracać listę par, które są wynikiem podziału listy wejściowej na dwie części. Warto zauważyć, że dla każdego elementu listy wejściowej można zrobić podział na dwie części: jedną z elementem i drugą bez niego. W ten sposób można zrobić wszystkie możliwe podziały listy.

```
splits' :: [a] -> [([a], [a])]
splits' [] = [([], [])]
splits' (x:xs) = ([], x:xs) : [(x:ys, zs) | (ys, zs) <- splits' xs]
```

1. Jeśli lista pusta to zwróć listę, której jedynym elementem jest para pustych list

   ```
   splits' [] = [([], [])]
   ```

2. W przeciwnym przypadku zwróć listę, której pierwszym elementem jest para pustej listy i listy wejściowej, a resztę listy tworzą pary, które są wynikiem rekurencyjnego wywołania funkcji `splits'` na liście bez pierwszego elementu

   ```
   splits' (x:xs) = ([], x:xs) : [(x:ys, zs) | (ys, zs) <- splits' xs]
   ```

8.3.5 Zadanie 21

Oto jedna z możliwych implementacji funkcji `partition`:

```
partition :: (a => Bool) => [a] => ([a], [a])
partition p xs = (filter p xs , filter (not . p) xs)
```

Ulepsz implementację tej funkcji: powinna zwracać ten sam wynik, ale powinna przchodzić przez listę tylko raz.

Oto ulepszona implementacja funkcji `partition`: funkcja `partition'` przechodzi przez listę tylko raz, dzięki użyciu akumulatorów.

```
partition' :: (a -> Bool) -> [a] -> ([a], [a])
partition' p xs = partition'' p xs [] []

partition'' :: (a -> Bool) -> [a] -> [a] -> [a] -> ([a], [a])
partition'' _ [] ys zs = (ys, zs)
partition'' p (x:xs) ys zs
    | p x = partition'' p xs (ys ++ [x]) zs
    | otherwise = partition'' p xs ys (zs ++ [x])
```

1. Jeśli lista pusta to zwróć parę list `ys` i `zs`

   ```
   partition'' _ [] ys zs = (ys, zs)
   ```

2. W przeciwnym przypadku jeśli warunek `p` jest spełniony to dodaj element do listy `ys` i rekurencyjnie wywołaj funkcję `partition''` na reszcie listy, a jeśli nie to dodaj element do listy `zs` i rekurencyjnie wywołaj funkcję `partition''` na reszcie listy

   ```
   partition'' p (x:xs) ys zs = ...
   ```

Przykładowe wywołanie:

```
> partition' even [1..10]
([2,4,6,8,10],[1,3,5,7,9])
```

8.3.6 Zadanie 22

Zaimplementuj samodzielnie funkcje `permutations` (znajduje się ona w module **Data.List**), która dla danej listy wyznaczy listę wszystkich jej permutacji (możemy założyć, ze wszystkie elementy listy wejściowej sa różne).

Oto implementacja funkcji `permutations` w języku Haskell:

```
permutations' :: [a] -> [[a]]
permutations' [] = [[]]
permutations' (x:xs) = [ys ++ [x] ++ zs | perm <- permutations' xs,
                                          (ys, zs) <- splits' perm]
```

1. Jeśli lista pusta to zwróć listę, której jedynym elementem jest pusta lista

   ```
   permutations' [] = [[]]
   ```

2. W przeciwnym przypadku zwróć listę, której elementami są wszystkie możliwe permutacje listy wejściowej, które powstają przez dodanie elementu `x` na różne pozycje w permutacjach listy bez pierwszego elementu

   ```
   permutations' (x:xs) = [ys ++ [x] ++ zs
               | perm <- permutations' xs, (ys, zs) <- splits' perm]
   ```

8.3.7 Zadanie 23 – Klasyczny Problem hetmanów

Celem jest umieszczenie ośmiu hetmanów na szachownicy tak, aby żadne dwa hetmany nie atakowały się nawzajem, tj. nie mogą znajdować się w:

- tym samym rzędzie,
- tej samej kolumnie,
- tej samej przekątnej.

1. Zaimplementuj problem wyszukiwania położeń Hetmanów w Haskell'u korzystając z funkcji `permutations`.
2. Dwa rozwiązania nazywamy równoważne jeśli pierwsze z nich można otrzymać za pomocą złożeń odbicia poziomego *(reverse)* oraz odbicia pionowego (np. `map (λ x-> n+1-x))` z drugiego. Ile jest nierównoważnych poprawnych rozstawień hetmanów?

Wskazówka: Przedstaw pozycje hetmanów jako listę liczb [1, ..., n]. Przykład: ciąg [4, 2, 7, 3, 6, 8, 5, 1] oznacza, że hetman w pierwszej kolumnie jest w rzędzie 4, hetman w drugiej kolumnie jest w rzędzie 2 itd.

---

1. Oto implementacja problemu hetmanów w języku Haskell:

```
queens :: Int -> [[Int]]
queens n = filter check (permutations [1..n])
    where
        check xs = and [abs (xs !! i - xs !! j)
                    /= j - i | i <- [0..n-1], j <- [i+1..n-1]]
```

2. Aby znaleźć liczbę nierównoważnych poprawnych rozstawień hetmanów, można posłużyć się funkcją `nub` z poprzedniego zadania, która usuwa duplikaty z listy. W ten sposób można znaleźć liczbę nierównoważnych poprawnych rozstawień hetmanów.

```
unique[]=[]
unique(x:xs)=
            if reverse x `elem` xs then unique xs
            else if map (\y -> 9-y) x `elem` xs then unique xs
            else if reverse x `elem` map (\y -> 9-y) xs then unique xs
            else x:unique xs
```

8.3.8 Zadanie 24

Napisz funkcję, która oblicza iloma zerami (w układzie dziesiętnym) kończy się liczba n!.
Uwaga: taki pomysł: "mam dane n; obliczam n!; zamieniam na łańcuch s; odwracam go; liczę ilość początkowych zer" traktujemy jako kompletnie beznadziejny
Wskazówka: Jak można wyznaczyć największą potęgę liczby 5 która dzieli daną liczbę n?

Implementacja funkcji, która oblicza ilość zer na końcu liczby $n!$ w języku Haskell:

```
zeros :: Int -> Int
zeros n = sum [n `div` (5^k) | k <- [1..n], 5^k <= n]
```

Działa to ponieważ liczba zer na końcu liczby $n!$ jest równa największej potędze liczby 5, która dzieli $n!$. Dla każdej domnożonej liczby 5 występuje przynajmniej jedna liczba 2, więc liczba zer na końcu liczby $n!$ jest równa liczbie piątek, które dzielą $n!$.

8.3.9 Zadanie 25

Ulepsz następującą "klasyczną" implementację funkcji quick-sort:

```
qs [] = []
qs (x : xs) = qs [t | t <= xs , t<=x] ++ [x] ++ qs [t | t <= xs , t>x]
```

Wskazówka: Czy warto z rekursją schodzić do list jednoelementowych?

Oto ulepszona implementacja funkcji `quickSort` w języku Haskell:

```
qsort :: (Ord a) => [a] -> [a]
qsort [] = []
qsort [x]    = [x]         -- lista jednoelementowa również nie wymaga sortowania
qsort (x:xs) = qsort [y | y <- xs, y <= x]
             ++ [x] ++
             qsort [y | y <- xs, y > x]
```

8.3.10 Zadanie 26

Napisz funkcję `isSorted :: (Ord a) => [a] -> Bool`, która sprawdza, czy podany argument $[x_1, \ldots, x_n]$ jest ciągiem niemalejącym, czyli czy $x_1 \leq x_2 \leq \cdots \leq x_n$.

Implementacja funkcji `isSorted` w języku Haskell:

```
isSorted :: (Ord a) => [a] -> Bool
isSorted [] = True
isSorted [_] = True
isSorted (x:y:xs) = x <= y && isSorted (y:xs)
```

Proste, iteracyjne sprawdzenie czy każdy element listy jest mniejszy lub równy następnemu.

8.3.11 Zadanie 27

Zaimplementuj w języku Haskell algorytm `Bubble Sort`.

Implementacja algorytmu `Bubble Sort` w języku Haskell:

```
bubbleSort :: (Ord a) => [a] -> [a]
bubbleSort xs = bubbleSort' xs (length xs)

bubbleSort' :: (Ord a) => [a] -> Int -> [a]
bubbleSort' xs 0 = xs
bubbleSort' xs n = bubbleSort' (bubble xs) (n - 1)

bubble :: (Ord a) => [a] -> [a]
bubble [x] = [x]
bubble (x:y:xs)
    | x > y     = y : bubble (x:xs)
    | otherwise = x : bubble (y:xs)
```

8.3.12 Zadanie 28

Typowe implementacje algorytmu `Quick Sort` sprawdzają przed wywołaniem długość listy i jeśli ma ona długość mniejszę lub równą 10, to do posortowania używają metody `Insertion Sort`. Zaimplementuj tę metodę w języku Haskell.

Implementacja algorytmu `Insertion Sort` w języku Haskell:

```
insertionSort :: (Ord a) => [a] -> [a]
insertionSort xs = foldr insert [] xs

insert :: (Ord a) => a -> [a] -> [a]
insert x [] = [x]
insert x (y:ys)
    | x <= y    = x : y : ys
    | otherwise = y : insert x ys
```

8.3.13 Zadanie 29

Oszacuj złożoność obliczeniową następującej (kiepskiej) funkcji służącej do odwracania listy:

```
rev :: [a] => [a]
rev [] = []
rev (x: xs) = (rev xs) ++ [x]
```

Złożoność obliczeniowa funkcji `rev` jest kwadratowa, ponieważ dla każdego elementu listy wywoływana jest funkcja `rev` na reszcie listy, co daje złożoność $O(n^2)$.

8.3.14 Zadanie 30

Funkcja `filter` może myć zdefiniowana za pomocą funkcji `map` i `concat`:

```
filter p = concat . map box
where box x =
```

Podaj definicję tej funkcji `box`.

Funkcja `filter` może być zdefiniowana za pomocą funkcji `map` i `concat` jeżeli nałożymy na każdy element listy warunek `p` i zwrócimy listę list, a następnie połączymy wszystkie te listy w jedną listę. Zatem `box` powinno zwracać listę, która zawiera tylko elementy spełniające warunek `p`.

```
filter p = concat . map box
    where box x = if p x then [x] else []
```

8.3.15 Zadanie 31

Funkcje `takeWhile` i `dropWhile` są podobne do funkcji `take` i `drop`, jednakże ich pierwszym argumentem jest funkcja boolowska zamiast liczby naturalnej. Na przykład:

- `takeWhile even [2,4,6,7,8,9] = [2,4,6]`
- `dropWhile even [2,4,6,7,8,9] = [7,8,9]`

Podaj rekurencyjne definicje tych funkcji

Rekurencyjne definicje funkcji `takeWhile` i `dropWhile` w języku Haskell:

- `takeWhile`:

```
takeWhile' :: (a -> Bool) -> [a] -> [a]
takeWhile' _ [] = []
takeWhile' p (x:xs)
    | p x       = x : takeWhile' p xs
    | otherwise = []
```

- `dropWhile`:

```
dropWhile' :: (a -> Bool) -> [a] -> [a]
dropWhile' _ [] = []
dropWhile' p (x:xs)
    | p x       = dropWhile' p xs
    | otherwise = x : xs
```

8.3.16 Zadanie 32

Napisz funkcję która dla ciągu łańcuchów $[L_1, \ldots, L_n]$ wyznaczy ich najdłuższy wspólny prefix.
Wskazówka: Możesz skorzystać z funkcji transpose z modułu `Data.List`.

Najdłuższy wspólny prefix listy łańcuchów można wyznaczyc poprzez transpozycję listy łańcuchów i zwrócenie wspólnego prefixu pierwszego łańcucha z resztą listy.

```
longestCommonPrefix :: [String] -> String
longestCommonPrefix [] = []
longestCommonPrefix xs = takeWhile allEqual (transpose xs)

allEqual :: (Eq a) => [a] -> Bool
allEqual [] = True
allEqual [x] = True
allEqual (x:y:xs) = x == y && allEqual (y:xs)
```

8.3.17 Zadanie 33

Napisz funkcję `subCard :: Int -> [a] -> [[a]]`, która dla danych parametrów k i $[x_1, \ldots, x_n]$ wyznaczy wszystkie podciągi $[x_{i_1}, \ldots, x_{i_k}]$ takie, że $1 \leq i_1 < i_2 < \cdots < i_k \leq n$.

Implementacja funkcji `subCard` w języku Haskell:

```
subCard :: Int -> [a] -> [[a]]
subCard k xs = [ys | ys <- combinations k xs]

combinations :: Int -> [a] -> [[a]]
combinations 0 _ = [[]]
combinations _ [] = []
combinations k (x:xs) = map (x:) (combinations (k - 1) xs) ++ combinations k xs

> subCard 2 [1,2,3,4]
[[1,2],[1,3],[1,4],[2,3],[2,4],[3,4]]
```

Jak działa funkcja `combinations`? Funkcja `combinations` zwraca wszystkie kombinacje k-elementowe listy xs. Jeśli $k = 0$ to zwraca listę jednoelementową, której jedynym elementem jest pusta lista. Jeśli $k = 0$ to zwraca pustą listę. W przeciwnym przypadku zwraca listę, której elementami są wszystkie możliwe kombinacje k-elementowe listy bez pierwszego elementu, do których dodano pierwszy element listy, oraz wszystkie kombinacje k-elementowe listy bez pierwszego elementu.

8.4 Foldy

8.4.1 Zadanie 34

Sprawdź typy i przetestuj działanie funkcji `sum`, `product`, `all` i `any`.

Typy i działanie funkcji `sum`, `product`, `all` i `any` w języku Haskell:

```
> :t sum
sum :: (Num a) => [a] -> a

> sum [1..10]
55
```

```
> :t product
product :: (Num a) => [a] -> a

> product [1..10]
3628800
```

```
> :t all
all :: (a -> Bool) -> [a] -> Bool

> all even [2,4,6,8,9]
False
```

```
> :t any
any :: (a -> Bool) -> [a] -> Bool

> any odd [2,4,6,8,9]
True
```

8.4.2 Zadanie 35

Przetestuj działanie funkcji

1. `foldl (+) 0 xs`,
2. `foldr (+) 0 xs`,
3. `foldl1 (+) xs`,
4. `foldr1 (+) xs` oraz
5. `sum X` na dużych listach liczb `X`.

Wskazówka: skorzystaj z polecenia *GHCi* `:set +s`; w celu usunięcia wyświetlania informacji skorzystaj z polecenia `:unset +s`
Przetestuj następnie działanie funkcji `foldl'` oraz `foldr'` (znajdują się one w module `Data.List`).

---

1. `foldl (+) 0 xs`:

```
> foldl (+) 0 [1..10]
55
```

2. `foldr (+) 0 xs`:

```
> foldr (+) 0 [1..10]
55
```

3. `foldl1 (+) xs`:

```
> foldl1 (+) [1..10]
55
```

4. `foldr1 (+) xs`:

```
> foldr1 (+) [1..10]
55
```

5. `sum X` na dużych listach liczb `X`:

```
> :set +s
> sum [1..1000000]
500000500000
(0.03 secs, 88,077,128 bytes)
```

6. `foldl'` oraz `foldr'`:

```
> :module Data.List
> :set +s
> foldl' (+) 0 [1..1000000]
500000500000
(0.03 secs, 88,077,144 bytes)
> foldr' (+) 0 [1..1000000]
500000500000
```

8.4.3 Zadanie 36

Samodzielnie zaimplementuj (oraz przetestuj) funkcję `reverse` działającą w czasie liniowym. Porównaj jej skuteczność z algorytmem z Zadania 29 **??**.

Implementacja funkcji `reverse` działającej w czasie liniowym w języku Haskell:

```
reverse' :: [a] -> [a]
reverse' = foldl (flip (:)) []
```

8.4.4 Zadanie 37

Zdefiniuj za pomocą funkcji `foldr` funkcję, które dla listy liczb $[a_1, \ldots, a_n]$ oblicza ile liczb parzystych występuje w tej liście.

To tak może wyglądać:

```
countEven :: [Int] -> Int
countEven = foldr (\x acc -> if even x then acc + 1 else acc) 0
countEven = foldr ((+) . (\x -> if even x then 1 else 0)) 0
```

8.4.5 Zadanie 38

Korzystając z funkcji `foldl` napisz funkcję `dec2Int` która konwertuje ciąg cyfr na liczbę całkowita, np. `dec2Int [1,2,1] = 121`.

Implementacja funkcji `dec2Int` w języku Haskell:

```
dec2Int :: [Int] -> Int
dec2Int xs = foldl (\acc x -> acc * 10 + x) 0 xs

> dec2Int [1,2,1]
121
```

Alternetywnie można coś ciekawego napisać przy pomocy funkcji `afib`

```
afib = 1:zipWith (-) 0:afib
...
```

8.4.6 Zadanie 39

Która z następujących równości jest prawdziwa?

1. `foldl (-) e xs = e - sum xs`
2. `foldr (-) e xs = e - sum xs`

Prawdziwa jest równość:

1. `foldl (-) e xs = e - sum xs`

Ponieważ `foldl (-) e xs` to `e - x1 - x2 - ... - xn`, a `sum xs` to `x1 + x2 + ... + xn`.

8.4.7 Zadanie 40

Dla danej listy $x_s = [x_1, \ldots, x_n]$ funkcja `lmss xs` wyznacza najdłuższą listę $[x_{j_1}, \ldots, x_{j_k}]$ taką, że $j_1 = 1$ oraz $x_{j_a} < x_{j_{a+1}}$ dla wszystkich $a = 1, \ldots, k-1$.
Na przykład, dla ciągu `xs = [3,2,1,5,3,2,6,2,3,8]` mamy `lmss xs = [3,5,6,8]`.

`lmss` to skrót od *longest monotonically increasing subsequence*. Oto implementacja funkcji `lmss` w języku Haskell:

```
addWhenSmaller xs n = xs ++ map (++[n]) (filter (\x -> if (x==[])
    then True else last x < n) xs)

longest [x] = x
longest (x:xs) =
    if (length x > length l) then x
    else l
    where l = longest xs
```

8.4.8 Zadanie 41

Funkcja `remdupl` usuwa z listy przylegające duplikaty, np.
`remdupl [1,1,2,1,1,3,3,4,4] = [1,2,1,3,4]`. Oprogramuj tę funkcję za pomocą `foldr` lub `foldl`.

```
remdupl :: (Eq a) => [a] -> [a]
remdupl = foldr (\x acc -> if null acc
    || x /= head acc then x:acc else acc) []
```

8.4.9 Zadanie 42

Korzystając z funkcji `foldl` i `foldr` napisz funkcję `approx n` zdefiniowaną następująco

$$\texttt{approx}(n) = \sum_{k=1}^{n} \frac{1}{k!}$$

Implementacja funkcji `approx` w języku Haskell:

```
approx :: Int -> Double
approx n = foldl (\acc k -> acc + 1 / fromIntegral (product [1..k])) 0 [1..n]

ghci> approx 100
1.7182818284590455
(0.01 secs, 774,656 bytes)
ghci> approx 1000
1.7182818284590455
(0.08 secs, 211,092,144 bytes)
ghci> approx 10000
1.7182818284590455
(34.83 secs, 227,568,464,200 bytes)
```

8.4.10 Zadanie 43

Napisz, korzystając z funkcji `foldl`, funkcję która dla ciągu liczb $[a_1, \ldots, a_n]$

$$\sum_{k=1}^{n} (-1)^{k+1} \cdot a_k$$

```
altSum :: (Num a) => [a] -> a
altSum xs = foldl (\acc (k, x) -> acc + (-1)^(k+1) * x) 0 (zip [1..] xs)
```

8.4.11 Zadanie 44

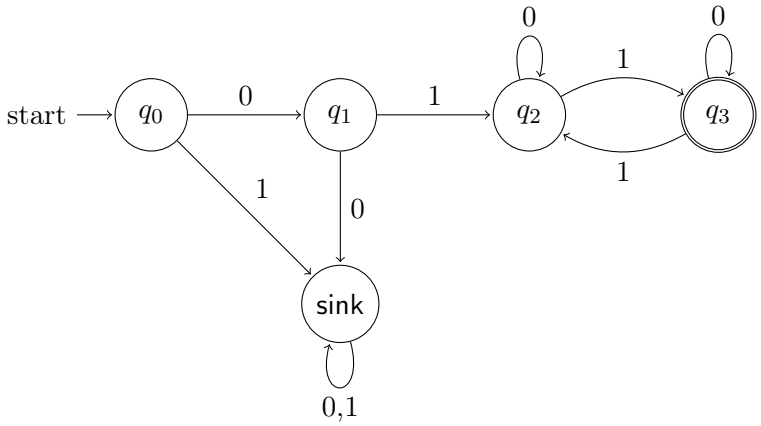
Napisz funkcję która dla zadanej listy $[a_1, \ldots, a_n]$ elementu typu `[Fractional a]` wyznaczy średnią arytmetyczną oraz wariancję ciągu $(a_1, \ldots, a_n)$. Skorzystaj tylko raz z funkcji fold.

```
meanVar :: (Fractional a) => [a] -> (a, a)
meanVar xs = (mean, var)
    where
        (mean, var, n) = foldl (\(m, v, n) x -> (m + x, v + x^2, n + 1)) (0, 0, 0) xs
        mean = mean / fromIntegral n
        var = (var - mean^2 * fromIntegral n) / fromIntegral (n - 1)
```

8.4.12 Zadanie 45

Zaimplementuj deterministyczny automat skończony który rozpoznaje język tych zero-jedynkowych ciągów która zaczynają się od `01` i zawierają parzystą liczbę jedynek.

Rozrysujmy najpierw ten automat:



gdzie stan `sink` jest stanem z którego nie ma wyjścia, a stan `q3` jest stanem akceptującym.

```
dfa :: String -> Bool
dfa xs = (foldl f 0 xs) == 3

f :: Int -> Int -> Int
f _ _ = -1
f 0 0 = 1
f 0 1 = -1
f 1 0 = -1
f 1 1 = 2
f 2 0 = 2
f 2 1 = 3
f 3 0 = 3
f 3 1 = 2
```

I es