

# Notatki Programowanie Funkcyjne

Jakub Kogut

24 marca 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:

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

- Przykład:

- $exp(Int) :: Int \rightarrow Int$
- $exp(Bool) :: Bool \rightarrow Bool$
- $exp(Double) :: Double \rightarrow Double$

Cichoń radzi, aby najpierw zastanowić się jaki powinny 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  $coll :: Int \rightarrow Int$

```
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  
 $collatz :: (Int, Int) \rightarrow (Int, Int)$

```
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 *lenght of collatz* zwracająca długość ciągu Collatz’a dla danej liczby:  
 $lenz :: Int \rightarrow Int$

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

## 2.3 Listy

Definicja listy w Haskellu:

`[a]` - lista elementów typu `a`

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

```
>: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`

- $\text{map } (n \rightarrow n^2) [1,2,3] = [1,4,9]$
- $\text{map } (n \rightarrow n^3) [1..10] = [1..1000]$

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.

$$[f x_1, x_2, x_3 \mid x_1 \leftarrow xs, x_2 \leftarrow ys, x_3 \leftarrow zs]$$

Przykład:

- chcemy stworzyć listę wszystkich trójek pitagorejskich poniżej 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 abstrahując ten koncept możemy napisać funkcję *foldl*<sup>1</sup>, 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
```

<sup>1</sup>jest to operacja składająca liste w następujący sposób  $((e \cdot x_1) \cdot x_2) \cdot x_3 \dots \cdot x_n$ , istnieje analogiczna *foldr* działająca odwrotnie  $x_1 \cdot (x_2 \cdot (x_3 \cdot \dots (x_n \cdot e)))$

## 4 Ćwiczenia

W tym miejscu będą pojawiały się notatki z laboratoriów (ćwiczeń)

### 4.1 Ćwiczenia 11-03-2025

#### 4.1.1 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.

---

$$Int \rightarrow Int \rightarrow Int$$

Zapis strzałkowy definiuje nam typ funkcji operacja  $=>$  jest wiążąca z prawej strony, więc można by było to również zapisać jako:

$$power :: Int \rightarrow (Int \rightarrow Int)$$

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
```

#### 4.1.2 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**, **Integral b** oznacza, że operator  $\wedge$  bierze jeden argument typu **Num** i drugi typu **Integral**.

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

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

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

### 4.1.3 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:

$$h :: Int \rightarrow Int \rightarrow Int$$

2. Nie, ponieważ:

$$h(x, y) = f(g(x, y)) = f(x + 2y) = (x + 2y)^2$$

3. Tak, ponieważ:

$$h(x) = f(g(x)) = f(x + 2x) = (x + 2x)^2 = 9x^2$$

### 4.1.4 Zadanie 4

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

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

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

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



#### 4.1.5 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; };
```

#### 4.1.6 Zadanie 6

Ustalmy zbiory  $A, B, C$ . Niech

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

będzie funkcją zadaną wzorem:

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

oraz niech

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

będzie zadaną 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  $\text{uncurry} \circ \text{curry} = \text{id}_{C^{B \times A}}$ .

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

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

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

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

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

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

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
```

#### 4.1.7 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)
```

#### 4.1.8 Zadanie 8

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

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

Jaki powinien być typ tej funkcji?

---

Typ tej funkcji powinien być następujący:

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

mogłaby ona wyglądać następująco:

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

#### 4.1.9 Zadanie 9 – (Eliminacja Pętli)

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

1. Masz daną (czyli oprogramowaną) funkcję  $f : \mathbb{N} \rightarrow \mathbb{N}$ . 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  $f: N \rightarrow N$ .
// You can replace this with any function of type  $int \rightarrow int$ .
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. Rozwiązanie 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
```

#### 4.1.10 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<sup>2</sup>:

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

3. Funkcja zipWith<sup>3</sup>:

```
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<sup>4</sup>:

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

5. Funkcja take<sup>5</sup>:

---

<sup>2</sup>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')]`.

<sup>3</sup>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]`.

<sup>4</sup>Funkcja `filter` zwraca listę elementów, które spełniają warunek podany jako argument.

Przykład: `filter even [1..10]` zwróci `[2,4,6,8,10]`.

<sup>5</sup>Funkcja `take` zwraca listę składającą się z  $n$  pierwszych elementów listy wejściowej.

Przykład: `take 3 [1,2,3,4,5]` zwróci `[1,2,3]`.

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

6. Funkcja `drop`<sup>6</sup>:

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

7. Funkcja `fib`<sup>7</sup>:

```
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]`.

#### 4.1.11 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]
```

---

<sup>6</sup>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]`.

<sup>7</sup>Funkcja `fib` zwraca listę liczb Fibonacciego do  $n$ -tego elementu.

#### 4.1.12 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
```

## 4.2 Elementy Teorii Liczb

Trochę teorii liczb, bo czemu nie?

### 4.2.1 Zadanie 13

Funkcję Eulera  $\varphi$  nazywamy funkcją określoną wzorem:

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

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

1. Oprogramuj funkcję  $\varphi$  (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  $\varphi$  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 identycznością na  $\mathbb{N}^+$ . Można zapisać to jako:

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

#### 4.2.2 Zadanie 14

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

$$n = \sum \{d : 1 \leq d < n \wedge d | n\}. \quad (6)$$

Na przykład liczba 6 jest liczbą doskonałą, ponieważ:

$$6 = 1 + 2 + 3. \quad (7)$$

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]

```

#### 4.2.3 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)]

```

#### 4.2.4 Zadanie 16

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

$$\text{dcp}(n) = \frac{1}{2n^2} |\{(k, l) \in \{1, \dots, n\} : \gcd(k, l) = 1\}|. \quad (8)$$

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  $n = 100, 200, 300, \dots, 10000$  i postaw jaką rozsądną hipotezę o:

$$\lim_{n \rightarrow \infty} \text{dcp}(n). \quad (9)$$

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. Optimalizacja 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  $n = 100, 200, 300, \dots, 10000$ :

```
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  $dcp(n)$  dla  $n \rightarrow \infty$  wynosi około 0.608.... Wartość ta może być przybliżona do wartości funkcji Eulera  $\frac{6}{\pi^2}$ .  $\square$  Można to interpretować jako gęstość liczb względnie pierwszych w zbiorze  $\{1, \dots, n\}$ , ale jako iż bierzemy symetrię względem 0 to wartość ta jest podwojona. Jednocześnie wynika to bezpośrednio z funkcji Zeta Riemanna  $\zeta(2) = \frac{\pi^2}{6}$ .

### 4.3 Listy – część 1.

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

#### 4.3.1 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)
```

### 4.3.2 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 mapę 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)
```

### 4.3.3 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
```

#### 4.3.4 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]
```

#### 4.3.5 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 przechodzić 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])
```

### 4.3.6 Zadanie 22

Zaimplementuj samodzielnie funkcję `permutations` (znajduje się ona w module `Data.List`), która dla danej listy wyznaczy listę wszystkich jej permutacji (możemy założyć, że wszystkie elementy listy wejściowej są 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]
```

### 4.3.7 Zadanie 23 – Klasyczny Problem hetmanów

Celem jest umieszczenie ośmiu hetmanów na szachownicy tak, aby żadne dwa hetmany nie atakowały się nawzajem, tj. nie mogły 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 złożenia odbicia poziomego (*reverse*) oraz odbicia pionowego (np. `map (\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.

```
queens' :: Int -> [[Int]]
queens' n = nub (map sort (queens n))
```

#### 4.3.8 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  $n!$ .

#### 4.3.9 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]
```

#### 4.3.10 Zadanie 26

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

---

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.

#### 4.3.11 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)
```

#### 4.3.12 Zadanie 28

Typowe implementacje algorytmu `Quick Sort` sprawdzają przed wywołaniem długość listy i jeśli ma ona długość mniejszą 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
```

#### 4.3.13 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 wywołana jest funkcja `rev` na reszcie listy, co daje złożoność  $O(n^2)$ .

#### 4.3.14 Zadanie 30

Funkcja `filter` może być 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 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 []
```

#### 4.3.15 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
```

#### 4.3.16 Zadanie 32

Napisz funkcję która dla ciągu łańcuchów  $[L_1, \dots, L_n]$  wyznaczy ich najdłuższy wspólny prefix.

**Wskazówka:** Możesz skorzystać z funkcji `transpose` z modułu `Data.List`.

Najdłuższy wspólny prefix listy łańcuchów można wyznaczyć 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)

```

### 4.3.17 Zadanie 33

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

---

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]]

```

## 4.4 Foldy

### 4.4.1 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
```

#### 4.4.2 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`).

---

#### 4.4.3 Zadanie 36

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

---

#### 4.4.4 Zadanie 37

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

---

#### 4.4.5 Zadanie 38