

Programowanie funkcyjne

Wojciech Typer

W katalogu przykłady pojawiać się będą przykłady z wykładów.

Funkcja lambda

$$\text{exp} = (\lambda a : \text{Typ} \rightarrow (a \rightarrow a))$$
$$\text{exp}(\text{Int}) :: \text{Int} \rightarrow \text{Int}$$
$$\text{exp}(\text{Double}) :: \text{Double} \rightarrow \text{Double}$$

Matematyczny zapis inkrementacji

$$\text{inc } x = x + 1$$
$$>: t \quad \text{inc}$$
$$\text{inc} :: \text{hum}(a) \implies (a \rightarrow a)$$
$$\text{exp} = (\forall a : \text{hum} \rightarrow (a \rightarrow a))$$
$$\text{exp}(\text{Int}) :: \text{Int} \rightarrow \text{Int}$$
$$\text{exp}(\text{Bool}) \leftarrow (\text{błąd})$$

Typy w Haskellu

- Typy proste:
 - `Int`
 - `Double`
 - `Char`
 - `Bool`
- Typy złożone:
 - Listy
 - Krotki
 - Funkcje

Funkcja collatz'a

Funkcja collatz'a jest nierozstrzygnięty dotychczas problem o wyjątkowo prostym jak wiele innych problemów teorii liczb sformułowaniu.

$$c_{n+1} = \begin{cases} \frac{1}{2}c_n & \text{gdy } c_n \text{ jest parzysta} \\ 3c_n + 1 & \text{gdy } c_n \text{ jest nieparzysta} \end{cases}$$

lub

$$c_{n+1} = \frac{1}{2}c_n - \frac{1}{4}(5c_n + 2)((-1)^{c_n} - 1).$$

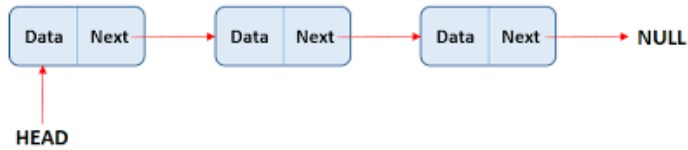
przykłady/W2 → funkcja collatz'a

Listy w Haskellu

Definicja list w Haskellu:

$[a]$ = lista elementów a

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



Rysunek 1: Przykładowa lista w Haskellu

Struktura listy:

$$x_0 : [x_1, x_2, \dots, x_k] = [x_0, x_1, \dots, x_k]$$

Lista pusta:

`[]` (lista pusta)

Przykłady list:

`[1, 2, 3]` można zapisać jako `1 : 2 : 3 : []`

Dodawanie (konkatenacja) dwóch list:

$$[1, 2, 3] + [4, 5] = [1, 2, 3, 4, 5]$$

Prelude

Prelude to standardowa biblioteka Haskellu

- 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]
```

0.0.1 Podstawowe funkcje operujące 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 (λn → n > 0) [-1,2,-3,4] = [2,4]`
 - `filter even [1..10] = [2,4,6,8,10]`

Jak zdefiniować funkcję **filter**:

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

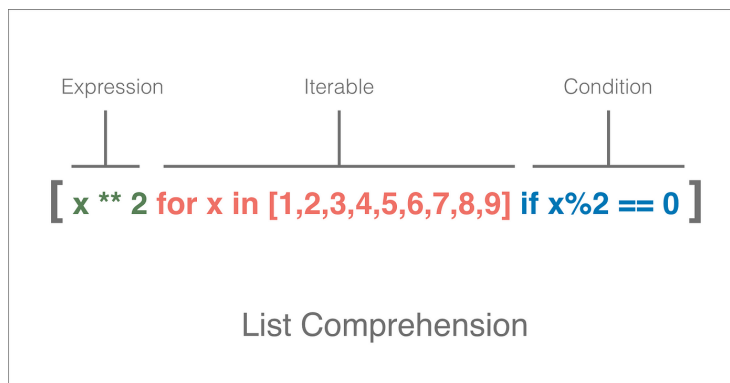
- `map :: (a → b) → [a] → [b]`
zwraca listę, która powstaje poprzez zastosowanie funkcji do każdego elementu listy
 - `map f [] = []`
 - `map f (x:xs) = f x : map f xs`

- `map (λn → n * n) [1,2,3] = [1,4,9]`
- `map (λn → n3) [1..10] = [1,8,27,64,125,216,343,512,729,1000]`

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

List comprehension

`[f x1 x2 x3 | x1 ← xs, x2 ← ys, x3 ← zs]`
`[f x1 x2 x3 | x1 ← xs, x2 ←, x1 < x2, x3 ← zs]`



Przykład: trójki pitagorejskie

`[(x, y, z) | z ← [1..100], y ← [1..z], x ← [1..y], x2 + y2 + z2, gcd x y == 1]`