§2 Wstęp do Haskell'u

Definicja funkcji

```
square n = n * n
silnia n = if n == 0 then 1
           else n * silnia (n - 1)
prod x y = x * y
```

square 5 => 5 * 5 => 25

silnia 4

=> if 4 == 0 then 1
   else 4 * silnia (4 - 1)

=> 4 * silnia (4 - 1)

$\overset{*}{\Rightarrow}$ 4 * (3 * (2 * (1 * silnia 0)))

$\overset{*}{\Rightarrow}$ 4 * (3 * (2 * (1 * 1)))

$\overset{*}{\Rightarrow}$ 24

prod 2 3 => 2 * 3 => 6

prod 2 $\overset{*}{\Rightarrow}$?

prod 2 jest <u>prawdziwym</u> wyrażeniem, które ewaluuje się do jednoargumentowej funkcji, która mnoży swój argument przez 2.

bo square = \n -> n * n

prod = \x -> \y -> x * y

więc

prod 2

= (\x -> \y -> x * y) 2

=> \y -> 2 * y

tzn. wynik ewaluacji może być funkcją!

double_square x = square (square x)

comb f g x = f (g x)

[albo lepiej: comb f g = \x -> f (g x)]

czyli (comb square square)

=> \x -> square (square x)

≡ double_square

<u>w Haskellu</u>: comb ≡ .

czyli

(square . square) 3

*=> (\x -> square (square x)) 3

=> square (square 3)

*=> 81

Definicje lokalne

```
let a = 3 in a + 2  => 3 + 2 => 5
let sq n = n * n in sq 3 *=> 9
```

uwaga: wartości dla a i sq istnieją tylko w ewaluacji wyrażeń a+2 i sq 3.

```
double_square x = x^2 + x^2
double_square x = let y = x^2 in y + y
double_square x = y + y where y = x^2
```

uwaga: tylko let jest wyrażeniem:

```
5 * (let sq n = n * n in sq 3) *=> 45
5 * (sq 3 where sq n = n * n)  ↯
```

Pattern matching

```
sil 0 = 1
sil n = n * sil (n-1)

first a b = a
second a b = b
```

```
f x = if c == 0 then a + b else c
      where (a, b, c) = x
```

f (1, 2, 4) *=> 4

f (1, 2, 0) *=> 3

Listy

[] pusta lista

: dołączenie elementu do listy (z lewej)

a : b : c : [] "=" [a, b, c]

[], : są konstruktorami.

```
append [] l = l
append (x:xs) l = x : (append xs l)
```

append [1,2] [3]

=> 1 : (append [2] [3])

=> 1 : (2 : (append [] [3]))

=> 1 : (2 : [3])

= [1, 2, 3]

```
member x [] = False
member x (y:ys) = if x == y then True
                  else member x ys
```

```
memb x [] = False
memb x (y:ys) = (x == y) || (member x ys)
```

```
map f [] = []
map f (x:xs) = (f x) : (map f xs)
```

map square [1,2,3] ⇒ [1,4,9]

map length [[],[1,2],[3]] ⇒ [0,2,1]

Zbiory

$$\{ x \mid x \in M \wedge P(x) \}$$

w Haskellu:

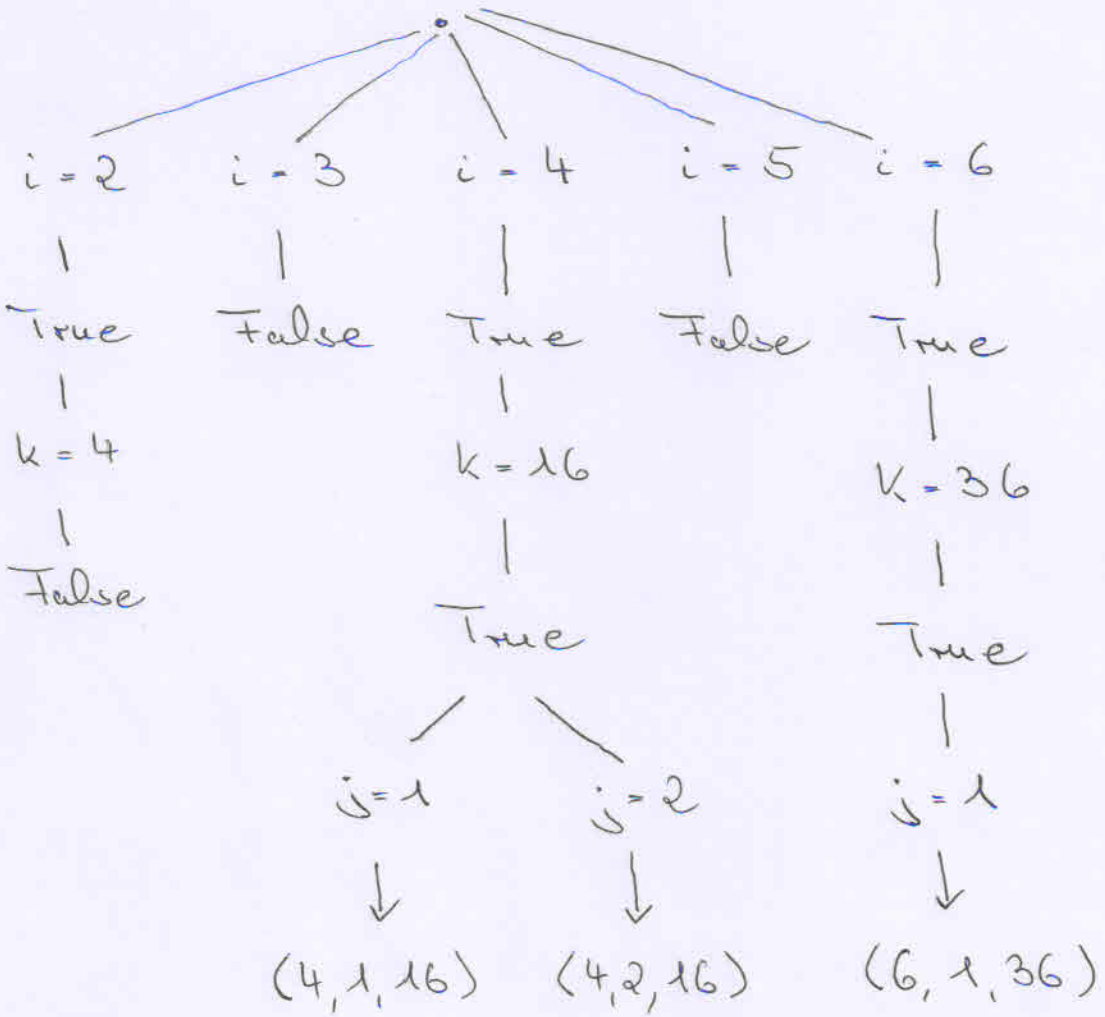
$[x \mid x \leftarrow m, P\ x]$ — m lista

$[a \mid a \leftarrow l, a < 5]$

$[(x,y) \mid x \leftarrow l_1, x \leftarrow l_2, \text{even}(x+y)]$

$[(i,j,k) \mid i \leftarrow [2..6], \text{even } i, \text{let } k = i * i, k > 10, j \leftarrow [1.. k \text{ 'mod' } 7]]$

$\Rightarrow [(4,1,16), (4,2,16), (6,1,36)]$



O typach

- w Haskellu każdy obiekt (wyrażenie) ma (musi mieć) typ
- można deklarować typy, ale nie trzeba
- Haskell obliczy (najbardziej ogólny) typ

```
5 :: Int
[1,2] :: [Int]
[1,[3]] :: ↯
square :: Int -> Int
square 5 :: Int          (*=> 5 :: Int)
2+3 :: Int
```

```
\x,y -> x + y :: ?
```

Int x Int → Int ?

Nie, ponieważ funkcje w Haskell'u mają tylko jeden argument!

```
\x -> \y -> x + y :: Int -> (Int -> Int)
(\x -> \y -> x + y) 3 :: Int -> Int
        ⇓
    \y -> 3 + y
```

$(\lambda x \to \lambda y \to x + y)\ 3\ 5 :: Int$

$\Downarrow$

$(\lambda y \to 3 + y)\ 5$

$\Downarrow$

$3 + 5$

Uwaga: $plus1\ x\ y = x + y$

$plus2\ (x, y) = x + y$

$plus1\ 2\ 7 \stackrel{*}{\Rightarrow} 9$ $\qquad$ $plus2\ 2\ 7$ ↯

$plus1\ (2,7)$ ↯ $\qquad$ $plus2\ (2,7) \stackrel{*}{\Rightarrow} 9$

$plus1 :: Int \to Int \to Int$

$plus2 :: (Int, Int) \to Int$

Polimorfizm

$length :: [\alpha] \to Int$

$(\forall \alpha : length :: [\alpha] \to Int)$

$length\ [1] :: Int$	$\alpha = Int$
$length\ ['a'] :: Int$	$\alpha = Char$
$length\ [[1,2]] :: Int$	$\alpha = [Int]$
$length\ [(1,'a')] :: Int$	$\alpha = (Int, Char)$
$length\ [1,[2]] ::$ ↯	$\alpha = ?$

uwaga:

```
length :: [Int] -> Int
length [] = 0
length (x:xs) = 1 + length xs
```

length [1,2] ⇒ 2

length [[1],[]] ↯

```
> :t length
length :: [a] -> Int
> :t (+)
(+) :: (Num a) => a -> a -> a
```

Num, to klasa typów (arytmetycznych)

+ dla Int: 2 + 3 ✓
+ dla Real: 2.3 + 1.8 ✓
+ dla [Int]: [1] + [1,2] ?

↳ $\forall_{\alpha \in Num}$: (+) :: $\alpha \to \alpha \to \alpha$

wbudowane lub
przez użytkownika