Programowanie w języku Haskell

List comprehensions i funkcje wyższego rzędu

Marcin Szlenk (IAiIS PW)

16 marca 2018

Proste definiowanie ciągów arytmetycznych:

```
ghci> [1..10]
[1,2,3,4,5,6,7,8,9,10]

ghci> [11..]
[11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,
27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,...

ghci> ['a'..'z']
”abcdefghijklmnopqrstuvwxyz”
```

List comprehensions:

```
ghci> [x^2 | x <- [1..5]]
[1,4,9,16,25]
```

```
ghci> [(x,y) | x <- [1,2], y <- ”AB” ]
[(1,'A'), (1,'B'), (2,'A'), (2,'B')]
```

Gdy jest więcej niż jeden generator, ostatni zmienia się najszybciej. Można dodać też strażników:

```
ghci> [x | x <- [1..10], even x]
[2,4,6,8,10]
```

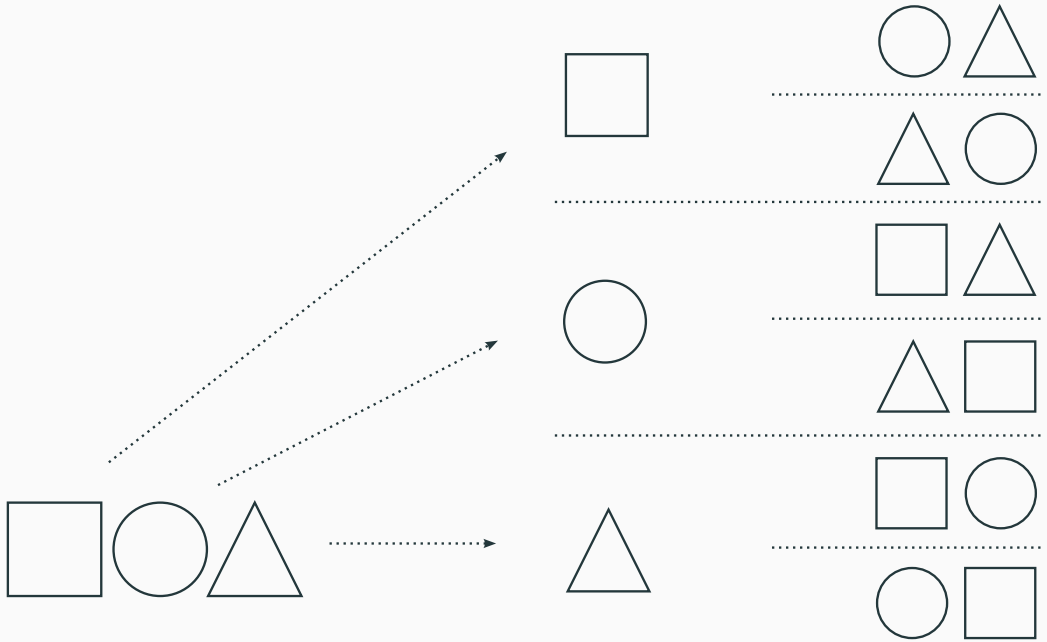
Przykłady użycia:

```haskell
firsts :: [(a,b)] -> [a]
firsts ps = [x | (x,_) <- ps]

factors :: Int -> [Int]
factors n = [x | x <- [1..n], mod n x == 0]
```

```
ghci> firsts [(1,"A"), (2,"B"), (3,"C")]
[1,2,3]
ghci> factors 15
[1,3,5,15]
```

Jeden z algorytmów generowania wszystkich permutacji polega na tym, że każdy element staje się kolejno głową dla permutowanej reszty:



Rysunek 1: Idea generowania permutacji

Funkcja generująca wszystkie permutacje listy:

```
perms :: [a] -> [[a]]
perms [] = [[]]
perms xs = [x:p | x <- xs, p <- perms (remove x xs)]
    where
        remove x []     = []
        remove x (y:ys) | x == y    = ys
                        | otherwise = y : remove x ys
```

```
ghci> perms "abc"
["abc","acb","bac","bca","cab","cba"]
```

Funkcja wyższego rzędu (*higher-order*) przyjmuje jako argumenty lub zwraca w wyniku inne funkcje

Funkcja aplikująca podaną funkcję do elementów listy:

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

```
ghci> map reverse ["Ala","ma","kota"]
["alA","am","atok"]
```

Funkcja łącząca w pary odpowiadające sobie elementy dwóch list:

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

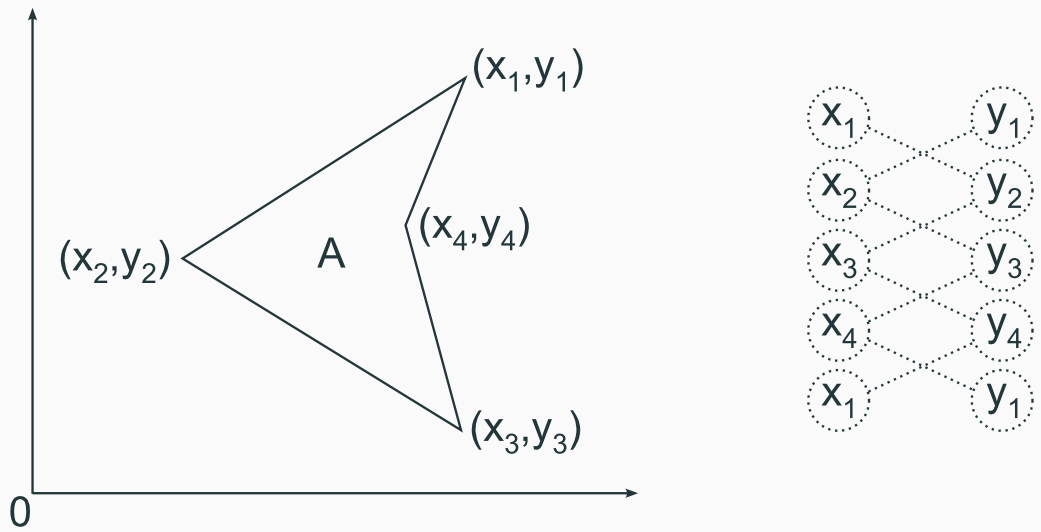
```
ghci> zip [1..] ”Ala”
[(1,'A'),(2,'l'),(3,'a')]
```

A teraz bardziej ogólna wersja:

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

```
ghci> max 1 0
1
ghci> zipWith max [1,4,8] [0,5,7,10]
[1,5,8]
```

Pole wielokąta można policzyć przy użyciu tzw. wzoru sznurówkowego (*shoelace formula*):



$$2A = x_1y_2 + x_2y_3 + x_3y_4 + x_4y_1 - x_2y_1 - x_3y_2 - x_4y_3 - x_1y_4$$

Rysunek 2: Wzór na pole czworokąta

Ogólnie, $A = \frac{1}{2}\left|\sum_{i=1}^{n}(x_i y_{i+1} - x_{i+1} y_i)\right|$, gdzie: $x_{n+1} = x_1, y_{n+1} = y_1$

```
area :: [(Float,Float)] -> Float
area ps = let x   = map fst ps
              x'  = tail x ++ [head x]
              y   = map snd ps
              y'  = tail y ++ [head y]
              xy  = zipWith (*) x  y'
              xy' = zipWith (*) x' y
          in 0.5 * abs (sum xy - sum xy')

ghci> area [(0,0),(5,0),(5,5),(0,5)]
25.0
```

Funkcja zwracająca elementy listy spałniające podany predykat:

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

```
ghci> even 2
True
ghci> filter even [1..10]
[2,4,6,8,10]
```

Funkcja zwracająca elementy listy dopóki nie spotka elementu, który nie spełnia predykatu:

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

```
ghci> odd 1
True
ghci> takeWhile odd [1,3,5,2,8]
[1,3,5]
```

Funkcje mogą być elementami struktur danych:

```
apply [] x     = []
apply (f:fs) x = f x : apply fs x
```

```
ghci> init "Haskell"
"Haskel"
ghci> apply [init,tail,reverse] "Haskell"
["Haskel","askell","lleksaH"]
```

Jaki jest typ funkcji `apply`?

Wyrażenia lambda (*lambda abstraction*) służą do zapisu funkcji anonimowych (bez nazwy):

```
ghci> (\x -> x + x) 1
2
ghci> (\x y -> x + y) 1 2
3
```

Przydają się gdy chcemy lokalnie użyć prostej funkcji:

```
ghci> filter (\x -> x /= 'a') ”Ala ma kota”
”Al m kot”
```

... a także przy definiowaniu funkcji lub operatorów zwracających funkcje

Operator złożenia funkcji:

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
f . g = \x -> f (g x)
```

```
ghci> sum [1..100]
5050
ghci> (even.sum) [1..100]
True
```

Każdą funkcję wieloargumentową można zaaplikować częściowo (*partial application*):

```
add x y = x + y
```


```
add 5
```


```
add 5 10 = (add 5) 10
```

Rysunek 3: Idea częściowego aplikowania funkcji

Formalnie, poniższe dwie definicje są równoważne:

```
add ::  Int -> Int -> Int
add x y = x + y

add :: Int -> (Int -> Int)
add = \x -> (\y -> x + y)
```

Stąd:

```
ghci> :type (add 5)
Int -> Int
```

W przypadku większej liczby argumentów jest podobnie. Typ funkcji:

```
mult :: Int -> Int -> Int -> Int
mult x y z = x * y * z
```

oznacza:

```
mult :: Int -> (Int -> (Int -> Int))
```

Operator `->` jest zatem prawostronnie łączny

Natomiast wyrażenie:

```
mult 5 10 15
```

oznacza:

```
((mult 5) 10) 15
```

Aplikacja funkcji jest zatem lewostronnie łączna

Częściowo aplikowane mogą być funkcje i operatory:

```
ghci> map (elem 'a') [”Ala”,”ma”,”koty”]
[True,True,False]

ghci> map (10+) [1..3]
[11,12,13]

ghci> map (/10) [1..3]
[0.1,0.2,0.3]

ghci> filter (/= 'a') ”Ala ma kota”
”Al m kot”
```

Ćwiczenia podsumowujące:

1. Napisz funkcję `any :: (a -> Bool) -> [a] -> Bool` sprawdzającą czy na liście znajduje się element spełniający podany predykat:

   ```
   ghci> any (=='0') ”12304560”
   True
   ```

2. Rozpatrzmy zdefiniowaną niżej funkcję `curry`. Jaką wartość ma wyrażenie: `curry fst 'a' 1` ? Wydedukuj typ funkcji `curry`

   ```
   curry f = \x y -> f (x, y)
   ```

3. Zdefiniuj funkcję `curry` jako 0-, 2- oraz 3-argumentową:

```
curry = ...

curry f x = ...

curry f x y = ...
```