HASKELL – listy i krotki

Krotki

Krotka jest skończonym ciągiem elementów dowolnego typu oddzielonych przecinkami, ujętych w nawiasy zwykłe.
Rozmiar krotki jest określony w momencie tworzenia i nie może być zmieniony.
Typ krotki zapisujemy jako $(A_1, ..., A_n)$ gdzie A_i jest typem kolejnych elementów krotki.

Krotkę dwuelementową nazywamy parą.
Dla par określone są funkcje:

fst :: (a,b) -> a - wyznacza pierwszy element pary
snd:: (a,b) -> b - wyznacza drugi element pary

```
Hugs> :t ('a',6,[3,4])
('a',6,[3,4]) :: (Num a, Num b) => (Char,b,[a])
Hugs > :t (True,5)
(True,5) :: Num a => (Bool,a)
Hugs > fst (3,8)
3 :: Integer
Hugs > snd (3,8)
8 :: Integer
```

```
-- min_max , dwuargumentowa, wynikiem jest uporządkowana para
min_max:: Int -> Int -> (Int,Int)
min_max x y
      | x >= y    = (x,y)
      |otherwise = (y,x)
```

```
-- Osoba = (Imie, wiek,plec)
type Osoba :: (String, Int, Char)

ania, jan :: Osoba
ania = („ania”, 21, ‘k’)
jan = („jan”,25,’m’)

wybierz_wiek :: Osoba -> Int
wybierz_wiek (n,w,p) = w
```

Listy.
Lista jest ciągiem elementów tego samego typu oddzielonych przecinkami, ujętych w nawiasy kwadratowe.
Rozmiar listy nie jest określony - można dołączać do niej kolejne elementy.
Typ listy zapisujemy jako [A], gdzie A jest typem elementów listy

Przykłady.

Lista :: Typ

```
[] :: [a]
[2,5,8,23,4] :: [Int]
["ania","basia","kasia"] :: [[Char]]
['a','%','b','C'] :: [Char]
[True,False,True] :: [Bool]
[[],[2],[6,7]] :: [[Int]]
```

Operatory działające na listach

Operator	Działanie	Priorytet	Łączność
:	służy do konstrukcji listy; dodaje element (głowę) do istniejącej listy	5	prawostronna
++	konkatenacja list	5	prawostronna
!!	operator indeksowania; wyznacza element listy o podanym numerze (numeracja od 0)	9	lewostronna
..	specyfikacja zasięgu listy		brak

Przykłady użycia

1) Operator (..)

```
[1..5] => [1,2,3,4,5] :: [Integer]
[1,3..10] => [1,3,5,7,9] :: [Integer]
['a'..'k'] => "abcdefghijk" :: [Char]
[10,8..0] => [10,8,6,4,2,0] :: [Integer]
```

Możliwe jest również generowanie list nieskończonych, np.

```
[1..]
```

2) Operator (:)

(:) :: a -> [a] -> [a]

Dodaje nowy element jako pierwszy element listy, np.

1: [2,3] => [1,2,3]
1: [] => [1]

Każda lista jest skonstruowana za pomocą operatora (:) rozpoczynając od listy pustej.

[1,2,3,4] = 1: (2: (3: (4:[])))
„abc” = ‘a’: (‘b’: (‘c’: []))

Ponieważ operator (:) jest łączny prawostronnie nawiasy mogą zostać opuszczone.

[1,2,3,4] = 1: 2: 3: 4:[]
„abc” = ‘a’: ‘b’: ‘c’: []

3) Operator (++)

(++) :: [a] -> [a] -> [a]

Służy do łączenia dwóch list w jedną , np.

[1,4,5] ++ [7,8] = [1,4,5,7,8]
x ++ y where x = [1,2] ; y = [3,4] => [1,2,3,4]

4) Operator (!!)

(!!) :: [a] -> Int -> a

Zwraca element o podanym indeksie. Elementy listy indeksowane są od zera.

[‘a’, ‘b’, ‘c’, ‘d’] !! 2 => ‘c’
[[1], [5,6,7], [3,5]] !! 1 !! 2 => [5,6,7] !! 2 => 7

Podstawowe funkcje działające na listach

1) **head** :: [a] -> a
zwraca głowę listy
> head [1,2,3] => 1

2) **tail** :: [a] -> [a]
zwraca ogon listy
> tail [1,2,3] => [2,3]

3) **last** :: [a] -> a
zwraca ostatni element listy
> last [1,2,3] => 3

4) **init** :: [a] -> [a]
zwraca listę bez ostatniego elementu
> int [1,2,3] => [1,2]

5) **length** :: [a] -> Int
zwraca długość listy
> length [1,2,3] => 3
> length [1..8] => 8
> length [] => 0

6) **null** xs
sprawdza, czy lista jest pusta
> null [1,2,3]
False
> null []
True

7) **reverse** :: [a] -> [a]
zwraca listę w odwrotnej kolejności
> reverse [1,2,3] => [3,2,1]

8) **take** :: Int -> [a] -> [a]
zwraca prefiks długości n
> take 2 [1,2,3,4] => [1,2]
> take 0 [1,2,3,4] => []
> take 5 [1,2,3,4] => [1,2,3,4]

9) **drop** :: Int -> [a] -> [a]
usuwa prefiks o określonej długości
> drop 2 [1,2,3,4] => [3,4]
> drop 0 [1,2,3,4] => [1,2,3,4]
> drop 5 [1,2,3,4] => []

10) **minimum** :: Ord a => [a] -> a
najmniejszy element listy
> minimum [8,4,2,1,5,6]
1

11) **maximum** :: Ord a => [a] -> a
największy element listy
> maximum [1,9,2,3,4]
9

12) **sum** :: Num a => [a] -> a
suma elementów listy liczbowej
> sum [5,2,1,6,3,2,5,7]
31

13) **product** :: Num a => [a] -> a
iloczyn elementów listy liczbowej
> product [6,2,1,2]
24
> product [1,2,5,6,7,9,2,0]
0

14) **elem** :: Eq a => a -> [a] -> Bool
sprawdza, czy x jest elementem listy
elem
> elem 4 [3,4,5]
True
> 4 `elem` [3,4,5,6]
True
> 10 `elem` [3,4,5,6]
False

Zwięzły sposób definiowania list (list comprehensions)

Listy możemy tworzyć za pomocą wyrażeń z kwalifikatorami.
Ogólna postać to:

[wyrażenie | kwalifikator]

przy czym kwalifikator może być generatorem lub dozorem (czyli warunkiem).
Całość opisuje listę w podobny sposób, jak często opisuje się zbiory, np.

$\{ x^2 \mid x \in \{1,...,10\}, \text{x-parzyste}\}$.

W Haskellu listę takich liczb można zapisać następująco:

```
[x*x | x <- [1..10], even x]
```

Lista [2,4,6,8,10] wszystkich liczb x takich, że x jest elementem listy [1..10] i x jest parzyste.

Inne przykłady:

```
[2*x | x <- [1..5]]
[y `mod` 3 | y <- [5..10]]
[ a*b | (a,b) <- [ (1,2), (2,3), (3,4)]]
[(x,y) | x <- [1,2], y <- [3,4]]
[x | x <- [1..12], y<- [1..12], x*y == 12]
[x | x <- [-5,2,3,-2], x>0 ]
```

Generatory zależne

Kolejny generator może zależeć od zmiennej wprowadzonej przez poprzedni generator.

1) `[(x,y) | x <- [1..3], y <- [x..3] ]`

Lista [(1,1), (1,2), (1,3), (2,2), (2,3), (3,3)] wszystkich par liczb (x,y) takich, że x i y są elementami listy [1..3], przy czym $y \geq x$.

2) Stosując generatory zależne możemy zdefiniować funkcję konkatenacji wielu list ,

```
konkat :: [[a]] --> [a]
konkat xss = [ x | xs <- xss, x <- xs ]
```

Np.

```
> konkat [ [1,2,3], [4,5], [6] ]
  [1,2,3,4,5,6]
```

Funkcja mapująca (mapping function)

Funkcja map aplikuje pewną funkcję f do każdego elementu listy xs.
Jest to funkcja wyższego rzędu ponieważ jednym z jej argumentów jest funkcja.

Definicja:

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

Funkcja mnożąca każdy element listy przez 2:

a) bez użycia map:

```
razy_dwa x = 2*x
razy_dwa_lista [] = []
razy_dwa_lista (x:xs) = (razy_dwa x) : (razy_dwa_lista xs)
```

b) z użyciem map:

```
razy_dwa_lista xs = map razy_dwa xs
```