

Kurs języka Haskell

Lista zadań na pracownię nr 2

Na zajęcia 12 i 17 marca 2020

W poniższych zadaniach rozważamy następujące, niezbyt zgrabne definicje pewnych funkcji:

```
zipF :: [a] -> [b] -> [(a,b)]
zipF [] [] = []
zipF (a:as) (b:bs) = (a,b) : zipF as bs

subseqF :: [a] -> [[a]]
subseqF [] = [[]]
subseqF (x:xs) = foldr (\ ys zss -> (x:ys) : zss) yss yss where
    yss = subseqF xs

ipermF :: [a] -> [[a]]
ipermF [] = [[]]
ipermF (x:xs) = concatMap insert (ipermF xs) where
    insert [] = [[x]]
    insert ys@(y:ys') = (x:ys') : map (y:) (insert ys)

spermF :: [a] -> [[a]]
spermF [] = [[]]
spermF xs = concatMap (\ (y,ys) -> map (y:) (spermF ys)) (select xs) where
    select [y] = [(y,[])]
    select (y:ys) = (y,ys) : map (\ (z,zs) -> (z,y:zs)) (select ys)

qsortF :: Ord a => [a] -> [a]
qsortF [] = []
qsortF (x:xs) = qsortF (filter (<= x) xs) ++ [x] ++ qsortF (filter (> x) xs)

(<+>) :: Ord a => [a] -> [a] -> [a]
[] <+> ys = ys
xs <+> [] = xs
xs'@(x:xs) <+> ys@(y:ys)
    | x <= y = x : (xs <+> ys')
    | otherwise = y : (xs' <+> ys)
```

Zadanie 1 (2 pkt). Zaprogramuj z użyciem *list comprehensions* (tam gdzie to możliwe) funkcje:

```
subseqC :: [a] -> [[a]]
ipermC :: [a] -> [[a]]
spermC :: [a] -> [[a]]
qsortC :: Ord a => [a] -> [a]
```

równe odpowiednim funkcjom zdefiniowanym powyżej. Przeczytaj w dokumentacji GHC o rozszerzeniu `ParallelListComp` i zdefiniuj następnie za pomocą *list comprehensions* równą funkcji `zipF` funkcję

```
zipC :: [a] -> [b] -> [(a,b)]
```

Zadanie 2 (2 pkt). Zaprogramuj w postaci bezpunktowej funkcje

```
subseqP :: [a] -> [[a]]
ipermP :: [a] -> [[a]]
spermP :: [a] -> [[a]]
qsortP :: Ord a => [a] -> [a]
zipP :: [a] -> [b] -> [(a,b)]
(<+>) :: Ord a => [a] -> [a] -> [a]
```

równe odpowiednim funkcjom zdefiniowanym powyżej.

Zadanie 3 (1 pkt). Rozważmy kombinacje aplikatywne kombinatorów S i K :

```
data Combinator = S | K | Combinator :$ Combinator
infixl 1:$
```

Przypomnijmy, że z kombinatorami S i K są związane następujące reguły upraszczania:

$$\begin{aligned} S :\$ x :\$ y :\$ z &\rightarrow x :\$ z :\$ (y :\$ z) \\ K :\$ x :\$ y &\rightarrow x \end{aligned}$$

Zainstaluj typ `Combinator` w klasie `Show`. Np. wyrażenie $(S :\$ K) :\$ (K :\$ S)$ powinno zostać zamienione na napis `SK(KS)` (z minimalną liczbą nawiasów).

Zadanie 4 (2 pkt). Zaprogramuj funkcje

```
evalC :: Combinator -> Combinator
```

która wyznacza postać normalną podanej kombinacji aplikatywnej. Użyj leniwej strategii wartościowania, tj. zawsze zamykaj najbardziej zewnętrzny skrajnie lewy redeks w kombinacji. Uwaga: dla niektórych kombinacji funkcja będzie się zapętlać!

Drzewa poszukiwań to drzewa, w których każdy wierzchołek zawiera ciąg etykiet a_1, \dots, a_n i posiada poddrzewa t_1, \dots, t_{n+1} (które mogą być puste), gdzie $n \geq 1$ (oznacza to, że poddrzew jest zawsze o jeden więcej niż etykiet). W każdym wierzchołku jest spełniony następujący aksjomat, będący niezmiennikiem operacji wstawiania i usuwania wierzchołków:

Wszystkie etykiety drzewa t_i są mniejsze niż a_i oraz wszystkie etykiety drzewa t_{i+1} są większe niż a_i dla $i = 1, \dots, n$.

Wynika stąd w szczególności, że ciąg a_1, \dots, a_n jest uporządkowany i że drzewo nie zawiera dwóch równych etykiet.

Binarne drzewa poszukiwań, to drzewa poszukiwań, w których każdy wierzchołek wewnętrzny ma dokładnie dwóch synów:

```
data BST a = NodeBST (BST a) a (BST a) | EmptyBST
```

Zadanie 5 (1 pkt). Zaprogramuj operacje

```
searchBST :: Ord a => a -> BST a -> Maybe a
insertBST :: Ord a => a -> BST a -> BST a
```

Operacja `searchBST` ujawnia etykietę wierzchołka drzewa równą podanemu elementowi (jeśli taka istnieje w drzewie). Operacja `insertBST` wstawia podany element do drzewa (bez powtórzeń).

Zadanie 6 (1 pkt). Zaprogramuj operacje

```
deleteMaxBST :: Ord a => BST a -> (BST a, a)
deleteBST :: Ord a => a -> BST a -> BST a
```

Operacja `deleteMaxBST` ujawnia i usuwa największą etykietę z drzewa (lub kończy się błędem w przypadku drzewa pustego — użyj funkcji `error`). Operacja `deleteBST` usuwa element z drzewa równy podanemu lub zwraca niezmiennione drzewo jeśli takiego elementu w drzewie nie ma.

Wadą rozważanych w powyższych zadaniach niezbalansowanych drzew BST jest znaczna degradacja efektywności funkcji działających na drzewie w przypadku jego rozbalansowania. *2-3-drzewa*, to drzewa poszukiwań o wierzchołkach wewnętrznych etykietowanych elementami pewnego typu `a`, w których każdy wierzchołek może mieć dwóch lub trzech synów:

```
data Tree23 a = Node2 (Tree23 a) a (Tree23 a)
              | Node3 (Tree23 a) a (Tree23 a) a (Tree23 a)
              | Empty23
```

Drzewa te spełniają następujący dodatkowy aksjomat, będący niezmiennikiem operacji wstawiania i usuwania wierzchołków:

Wszystkie ścieżki od korzenia do liścia w 2-3-drzewie są tej samej długości.

2-3-drzewa są zatem drzewami zbalansowanymi (wysokość h drzewa o n wierzchołkach spełnia nierówność $\log_3 n \leq h \leq \log_2 n$).

Zadanie 7 (1 pkt). Zaprogramuj operację

```
search23 :: Ord a => a -> Tree23 a -> Maybe a
```

wyszukującą element w 2-3-drzewie.

Zadanie 8 (2 pkt). Zaprogramuj operację

```
insert23 :: Ord a => a -> Tree23 a -> Tree23 a
```

wstawiającą element do 2-3-drzewa. Wstawianie rozpoczyna się tak, jak dla drzew niezbalansowanych, wyszukaniem odpowiedniego miejsca wstawienia elementu. Teraz jednak wstawienie elementu zaburza niezmiennik drzewa. Podczas wychodzenia z rekursji należy przesunąć w kierunku korzenia miejsce, w którym jest zaburzony niezmiennik. Warto w tym celu zdefiniować pomocniczą operację

```
ins :: Ord a => a -> Tree23 a -> InsResult a
```

która wraz z nowym drzewem zwraca informację, czy drzewo urosło, czy nie:

```
data InsResult a = BalancedIns (Tree23 a) | Grown (Tree23 a)
```

Jeśli drzewo urosło, to można przywrócić niezmiennik lub przesunąć jego zaburzenie w kierunku korzenia korzystając z rotacji przedstawionych na Rysunku 1, w których wierzchołek pogrubiony, to wynik `Grown` operacji `ins`. Ponieważ w przypadku każdej rotacji argument konstruktora `Grown` trzeba rozłożyć na części i jest on zawsze 2-wierzchołkiem, to zamiast powyższej definicji typu `InsResult` można użyć następującej:

```
data InsResult a = BalancedIns (Tree23 a) | Grown (Tree23 a) a (Tree23 a)
```

w której konstruktor `Grown` przechowuje osobno etykietę korzenia i dwa poddrzewa wyniku wstawiania.

Rozważ podczas implementacji, czy rotacje lepiej umieścić wprost w kodzie funkcji `ins`, czy też zakodować je za pomocą osobnej nierekurencyjnej funkcji.

Zadanie 9 (3 pkt). Usuwanie podanego elementu oraz usuwanie największego elementu:

```
deleteMax23 :: Tree23 a -> (Tree23 a, a)
delete23 :: Ord a => a -> Tree23 a -> Tree23 a
```

mogą naruszyć niezmiennik w przeciwną stronę: drzewo stanie się zbyt płytkie. Możemy więc, podobnie jak podczas wstawiania elementu, zdefiniować funkcje pomocnicze

```
delMax :: Tree23 a -> (DelResult a, a)
del :: a -> Tree23 a -> DelResult a
```

zwracające oprócz drzewa także informację, czy uległo ono spłyceciu:

```
data DelResult a = BalancedDel (Tree23 a) | Shrunked (Tree23 a)
```

Rotacje wykonywane przez funkcje `del` i `delMax` przywracające zbalansowanie lub przesuujące jego naruszenie w kierunku korzenia są przedstawione na Rysunku 2, na którym krawędź pogrubiona odpowiada konstruktorowi `Shrunked`. Zaprogramuj powyższe funkcje.

Zadanie 10 (1 pkt). *2-3-4-drzewa*, to drzewa podobne do 2-3-drzew, w których wierzchołek wewnętrzny może mieć 2, 3 lub 4 synów:

```
data Tree234 a = N2 (Tree234 a) a (Tree234 a)
                | N3 (Tree234 a) a (Tree234 a) a (Tree234 a)
                | N4 (Tree234 a) a (Tree234 a) a (Tree234 a) a (Tree234 a)
                | E234
```

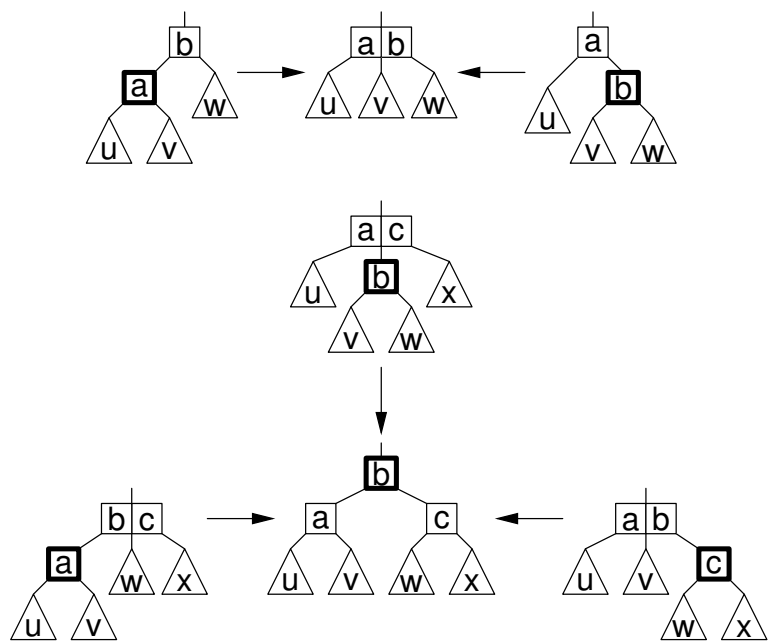
Okazuje się, że istnieje naturalna jednoznaczna odpowiedniość między 2-3-4-drzewami i drzewami *czzerwono-czarnymi*:

```
data RBTREE a = Black (RBTREE a) a (RBTREE a)
                | Red (RBTREE a) a (RBTREE a)
                | RBTEEmpty
```

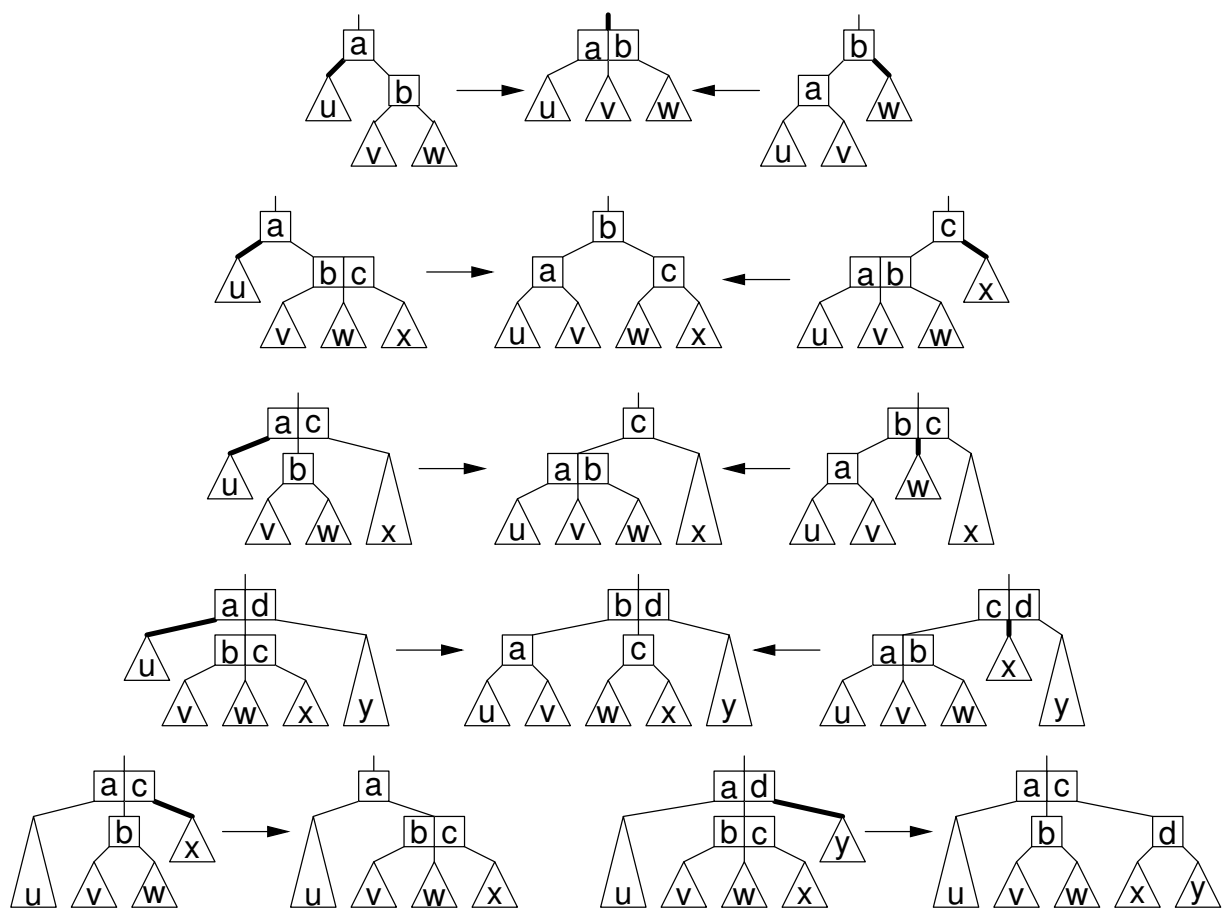
Zaprogramuj funkcje

```
from234 :: Tree234 a -> RBTREE a
to234 :: RBTREE a -> Tree234 a
```

dokonyjące konwersji pomiędzy tymi rodzajami drzew.



Rysunek 1: Rotacje podczas wstawiania elementu do 2-3-drzewa



Rysunek 2: Rotacje podczas usuwania elementu z 2-3-drzewa