Деревья и функции высшего порядка

На прошлом занятии мы рассмотрели обработку деревьев с помощью функции высшего порядка `reduce`. Теперь стоит рассмотреть еще одну функцию, которая будет неоднократно возникать в различных ситуациях.

Рассмотрим преобразования деревьев, которые получаются путем применения некоторой операции к каждому узлу дерева. Например: функция, увеличивающая значение всех узлов на 1, и функция, удваивающая значения всех узлов. Их реализация выглядит следующим образом:

```
incValues :: BiTree -> BiTree
incValues (Leaf v) = Leaf (v+1)
incValues (BiTree v l r) = BiTree (v+1) (incValues l) (incValues r)


doubleValues :: BiTree -> BiTree
doubleValues (Leaf v) = Leaf (2*v)
doubleValues (BiTree v l r) = BiTree (2*v) (doubleValues l)
                                           (doubleValues r)
```

Видно, что функции отличаются только выражениями `v+1` и `2*v` – операциями, применяемыми к значениям в узлах. Соответственно, можно попытаться вынести эти операции в отдельную функцию-параметр и определить обобщенную функцию, производящую подобные преобразования.

Поскольку в каждому узле хранится значение типа Int, то функция преобразования узла имеет тип `Int -> Int`. Запишем такие функции для первого и второго случаев:

```
inc :: Int -> Int
inc v = v+1


double :: Int -> Int
double v = 2*v
```

Или используя определение в форме безымянных функций:

```
inc = (\v -> v+1)
double = (\v -> 2*v)
```

Эти функции мы будем передавать в качестве параметра (допустим, первого). Теперь необходимо определить саму функцию высшего порядка:

```
mapTree :: (Int -> Int) -> BiTree -> BiTree
mapTree f (Leaf value) = Leaf (f value)
mapTree f (BiTree value left right) = BiTree (f value)
                                (mapTree f left) (mapTree f right)
```

Теперь можно записывать преобразования в форме:

```
mapTree inc (BiTree 2 (Leaf 3) (Leaf 1))
```

```
mapTree double (BiTree 2 (Leaf 3) (Leaf 1))
```

Или с использованием безымянных функций:

```
mapTree (\v -> v+1) (BiTree 2 (Leaf 3) (Leaf 1))
mapTree (\v -> 2*v) (BiTree 2 (Leaf 3) (Leaf 1))
```

Теперь вспомним тот факт, что для бинарных операторов существует сокращенная форма записи их частичных применений в виде `(2*)` или `(+1)`:

```
mapTree (+1) (BiTree 2 (Leaf 3) (Leaf 1))
mapTree (2*) (BiTree 2 (Leaf 3) (Leaf 1))
```

И можно определить исходные функции `incValues` и `doubleValues` очень коротко:

```
incValues t = mapTree (+1) t
doubleValues t = mapTree (2*) t
```

Или же, если вспомнить про частичную применимость функций:

```
incValues = mapTree (+1)
doubleValues = mapTree (2*)
```

Выглядит весьма неплохо и вполне понятно =).

Стоит отметить тот факт, что последняя запись стала возможной балгодаря тому, что функция, применяемая к значениям в узлах дерева, является первым параметром функции `mapTree`. Таким образом, порядок параметров имеет значение для возможности красиво использовать обобщенные функции.

Деревья поиска

Теперь рассмотрим альтернативное объявление деревьев, которое удобно для представления бинарных деревьев поиска. Раньше мы определяли деревья на основе дерева из одного узла, что не позволяло рассматривать пустые деревья или узлы, у которых только один дочерний узел. Чтобы иметь возможность рассматривать такие деревья, необходимо ввести понятие пустого дерева и строить структуру на его основе. Например так:

```
data SearchTree = EmptyTree | SearchTree Int SearchTree SearchTree
```

Как здесь видно, пустое дерево не содержит никаких значений, а любое непустое дерево представляется конструктором `SearchTree`. Рассмотрим, как можно вычислить глубину такого дерева:

```
depth :: SearchTree -> Int
depth EmptyTree = 0
depth (SearchTree x t1 t2) = max (depth t1) (depth t2) + 1
```

Теперь, предположим, что это дерево удовлетворяет условиям, накладываемым на дерево поиска: для любого дерева все значения в левом поддереве меньше, чем значения в корне, а все значения в правом поддереве – больше. Тогда мы можем легко найти минимальное значение в дереве – оно находится в крайнем левом узле:

```
minValue :: SearchTree -> Int
minValue (SearchTree i EmptyTree _ ) = i
minValue (SearchTree _ t1 _ ) = minValue t1
```

Чтобы проверить, встречается ли заданное значение в дереве, можно спускаться по дереву от корня, сравнивая на каждом шаге искомое значение со значением в узле. В случае равенства – получен положительный ответ, в случае если оно меньше или больше – необходимо продолжать поиск в левом или правом поддереве соответственно. Если достигнуто пустое дерево, то значения в дереве отсутствует. Для этого можно определить следующую функцию:

```
hasValue :: SearchTree -> Int -> Bool
hasValue EmptyTree _ = False
hasValue (SearchTree i t1 t2) x | i > x = hasValue t1 x
                                | i < x = hasValue t2 x
                                | otherwise = True
```

Для того, чтобы вставить новое значение в такое дерево, необходимо повторить процедуру, аналогичную поиску, с тем различием, что если достигнуто пустое дерево, то необходимо на его месте создать дерево из одно узла. Однако, необходимо учесть, что мы не можем менять уже созданную структуру данных и потому необходимо сформировать новую:

```
insertValue :: Int -> SearchTree -> SearchTree
insertValue x EmptyTree = SearchTree x EmptyTree EmptyTree
insertValue x t@(SearchTree i t1 t2)
                    | x < i = SearchTree i (insertValue x t1) t2
                    | x > i = SearchTree i t1 (insertValue x t2)
                    | otherwise = t
```

Здесь мы используем сопоставление `t@(SearchTree i t1 t2)` для того, чтобы в случае, если элемент в дереве уже присутствует, вернуть соответствующее поддерево сразу (`t`), не производя излишнее копирование, а если нет – то спустится в левое (`t1`) или правое (`t2`) поддерево.

Теперь рассмотрим модификацию функции вставки значения с использованием меток полей при определении дерева. Пометим значение в дереве как `value`, а левое и правое поддеревья как `left` и `right` соответственно:

```
data SearchTree = EmptyTree
                | SearchTree{ value::Int, left, right::SearchTree }
```

При таком определении можно использовать специальную конструкцию, позволяющую скопировать значение типа, заменив в нем значения одного или нескольких полей. Например, чтобы скопировать узел дерева, заменив в нем значение, необходимо написать:

```
t{ value = 2 )
```

А функция вставки с использованием этой возможности будет выглядеть следующим образом:

```
insertValue :: Int -> SearchTree -> SearchTree
insertValue x EmptyTree = SearchTree x EmptyTree EmptyTree
insertValue x t@(SearchTree i t1 t2)
                    | x < i = t{ left = insertValue x t1 }
                    | x > i = t{ right = insertValue x t2 }
                    | otherwise = t
```

Аналогично (но чуть более сложно) можно написать функцию удаления элемента из дерева (которая опять же будет создавать копию дерева без удаленного элемента).