Структуры данных в Haskell

Тип данных Maybe и его возможности

В Haskell определен следующий тип:

```
data Maybe a = Nothing | Just a deriving (Eq, Ord)
```

(Напомним, здесь **Maybe** — это конструктор типов, в данном случае полиморфный, параметризованный произвольным типом a, **Nothing** и **Just** — конструкторы данных)

Можно задать значение данных, применив конструктор данных **Just** к значению:

```
mynumber = Just 5
```

Полиморфные типы похожи на контейнеры, которые могут содержать значения многих различных типов. Так, **Maybe Int** можно считать контейнером типа **Maybe**, который содержит **Int** с тэгом **Just**, или **Nothing**.

Этот тип данных позволяет программисту указать что-то, что, возможно, не существует.

Императивные языки могут поддерживать похожую функциональность путем использования конструкции **union** или что-то подобное NULL, чтобы возвращать возможно отсутствующее значение.

Как видно из определения типа **Maybe** — этот тип будет экземпляром (воплощением) классов **Eq** и **Ord**, если таковым будет и базовый тип a. Следующий пример показывает, что требование на равенство к базовому типу обязательно.

```
data My1 = One | Two
data My2 = Maybe My1

test = (Just One == Just Two)
```

выдает следующую ошибку:

```
... error:
    * No instance for (Eq My1) arising from a use of `=='
    * In the expression: (Just One == Just Two)
      In an equation for `test': test = (Just One == Just Two)
Failed, modules loaded: none.
```

Изменив строку

```
data My1 = One | Two deriving (Eq,Ord)
```

получаем выполнение теста

```
> test
False
```

Схожим образом реализуются классы **Show** и **Read**.

Полностью роль и возможности типа данных **Maybe** мы сможем раскрыть, когда начнем изучения *монад*. Ну а сейчас, некоторые примеры простого использования.

Вспомним, в 6-й лекции, при рассуждениях о преимуществах статического и динамического программирования, описывалась простая функции поиска

```
find x lst  = find' lst 0
  where
    find' [] _ = (-1)
    find' (y:ys) t = if x == y
                     then t
                     else find' (ys) (t+1)
```

её тип:

```
> :t find
find :: (Num t, Eq t1) => t1 -> [t1] -> t
```

И встроенное решение:

```
:m Data.List

> elemIndex 2 [0..5]
Just 2
```

Вот здесь как раз и работает типа **Maybe**. Иными словами, если мы хотим при неудаче возвращать вместо ничего «не говорящего» значения `(-1)`, особое значение, например, **Nothing**, то нам следует вот так изменить нашу функцию:

```
find x lst  = find' lst 0
  where
    find' [] _ = Nothing
    find' (y:ys) t = if x == y
                     then Just t
                     else find' (ys) (t+1)
```

вот ее тип и пример использования:

```
*Main> :t find
find :: (Num t, Eq t1) => t1 -> [t1] -> Maybe t

*Main> find 'o' ”hello”
Just 4
*Main> find 'i' ”hello”
Nothing
```

Целый ряд функций, определенный в разнообразных модулях, действует схожим образом.

Это уже знакомая нам функция

```
find :: (a -> Bool) -> [a] -> Maybe a
```

которая в библиотечной версии `Data.List` возвращает первый элемент из списка, удовлетворяющий данному предикату, или **Nothing**, если такого элемента нет:

```
> find (> 4) [1..]
Just 5
> find (< 0) [1..10]
Nothing
```

и функция **elemIndex**, которая возвращает индекс первого элемента из списка, равного данному, или **Nothing**, если такого элемента нет:

```
> elemIndex 'b' ['a', 'b', 'c']
Just 1
> elemIndex 'w' ['a', 'b', 'c']
Nothing
```

Функция **findIndex**, обобщая **find** и **elemIndex**, вновь ищет первый элемент из списка, удовлетворяющий данному предикату, и возвращает его индекс или **Nothing**, если такого элемента нет:

```
findIndex :: (a -> Bool) -> [a] -> Maybe Int

> findIndex isSpace "Hello World!"
Just 5
```

Тип **Maybe** является встроенным в **Prelude**, дополнительный функционал вынесен в модуль Data.**Maybe** и Control.**Monad**.**Maybe**

```
isJust :: Maybe a -> Bool

> isJust (Just 3)
True
> isJust (Just ())
True
> isJust Nothing
False
```

но при этом

```
> isJust (Just Nothing)
True
```

Аналогичная функция

```
isNothing :: Maybe a -> Bool

> isNothing (Just 3)
False
> isNothing (Just ())
False
> isNothing Nothing
True
```

но при этом

```
> isNothing (Just Nothing)
False
```

Полезная функция-распаковщик

```
fromJust :: Maybe a -> a

> fromJust (Just 1)
```

```
1
> 2 * (fromJust (Just 10))
20
> 2 * (fromJust Nothing)
*** Exception: Maybe.fromJust: Nothing
```

и другой её вариант с дефолтным значением

```
fromMaybe :: a -> Maybe a -> a

> fromMaybe "" (Just "Hello, World!")
"Hello, World!"
> fromMaybe "" Nothing
""
```

Или вот такая функция **maybe**, которая берет некоторое дефолтное значение типа b, вторым аргументом берет некоторую функцию (определенную над основным типом a), применяет их к третьему аргументу, который является типом a, запакованным в тип **Maybe** a:

```
maybe :: b -> (a -> b) -> Maybe a -> b
```

И если это было значение внутри **Just**, то применяется функция, если **Nothing**, то дефолтное значение:

```
> maybe False odd (Just 3)
True
> maybe False odd Nothing
False
```

Как уже говорилось выше, наиболее мощные и красивые аспекты данного типа данных проявляются при эксплуатации монад.

wiki.haskell: Maybe

Data.Maybe

wikibooks.org: Maybe

stackoverflow: Using Maybe type in Haskell

schoolofhaskell: Data.Maybe

New monads/MaybeT

MaybeT

MaybeT-transformers

Коротко об этом применении можно сказать так: если у нас есть ряд вычислений, каждое из которых может возвращать либо **Just ...**, либо **Nothing** (и при этом, тип предыдущего вычисления согласован со следующим), то можно организовать цепочку вычислений без использования «лавины» проверок **if-then-else**, и в отличие от использования функции **error**, вычисления прерываться не будут.

Тип данных `Either` и его возможности

Близким по сути к типу **`Maybe`** является тип **`Either`**, который вместо общего сообщения, что «что-то пошло не так», может возвращать различные значения (номер ошибки, строку описания ошибки и т.п.)

```
data  Either a b  =  Left a | Right b
  deriving ( Eq, Ord, Read, Show )
```

Например,

```
safe_divide :: Int -> Int -> Either String Int
safe_divide _ 0 = Left ”divide by zero”
safe_divide i j = Right (i `div` j)
```

Это тоже монада, и собственно по её типу действует один из способов обработки ошибок в Haskell. Позже, мы специально вернемся к этому.

Для удобства обработки есть ряд удобных утилит, как и в случае с **`Maybe`**.

```
isLeft :: Either a b -> Bool
```

возвращает **`True`**, если данное значение собрано с помощью конструктора **`Left`** и **`False`** — иначе.

```
isRight :: Either a b -> Bool
```

Действует аналогично, но наоборот.

```
> isLeft (Left ”foo”)
True
> isLeft (Right 3)
False
> isRight (Left ”foo”)
False
> isRight (Right 3)
True
```

Более сложные и интересные случаи использования рассмотрим при изучении монад (напр., как избегать паттерн-матчинга при анализе возвращаемых значений)

Функции-распаковщики с дефолтным значением (первый аргумент):

```
fromLeft :: a -> Either a b -> a

fromRight :: b -> Either a b -> b
```

с поведением, которе описано из примеров ниже:

```
> fromLeft 1 (Left 3)
3
> fromLeft 1 (Right ”foo”)
1

> fromRight 1 (Right 3)
3
```

```
> fromRight 1 (Left "foo")
1
```

Функция

```
either :: (a -> c) -> (b -> c) -> Either a b -> c
```

тоже позволяет избежать паттерн-матчинга при анализе результатов (взяв его на себя). Первым и вторым аргументами ей передаются функции-обработчики, которые срабатывают, если использован конструктор **Left** или **Right** (соответственно) в третьем аргументе.

Определим строку s, обернув ее конструктором **Left**, и число n, обернув его конструктором **Right**

```
> let s = Left "foo" :: Either String Int
> let n = Right 3 :: Either String Int
```

Для корректного анализа функцией **either** мы указали их полный тип (указали типы недостающих компонент). Впрочем, проверка показывает, что все работало бы и без такого усложнения:

```
> s = Left "foo"
> n = Right 3
```

В итоге получаем

```
> either length (*2) s
3
> either length (*2) n
6
```

Data.Either

mvanier.livejournal: Error-handling computations

schoolofhaskell: Data.Either

8 ways to report errors in Haskell

10. Error Handling

Рекурсивные типы данных

Наиболее известным рекурсивным типом данных в Haskell являются без сомнения списки:

```
data [a] = [] | a : [a]
```

Или в других терминах, мы можем собрать собственный параметризованный тип списков:

```
data List a = Nil | Cons a (List a)
```

который вполне будет работать.

Или для простоты даже не параметризованный тип:

```
data ListDub = Nil | Cons Double (ListDub)
```

Мы видим, что тип **List**, что тип ListDub, что тип [a] определяются в терминах самих себя. Что-то вроде «*тип списков — это добавление значения к типу списков или это — пустой список*».

Двигаясь в этом ключе, мы можем сформировать более сложные типы данных, например, *деревья*.

Двоичные деревья

Рассмотрим пример двоичного дерева, листья которого содержат значения произвольного типа a, а вершины не содержат значений (т.е., рассмотрим вариант параметризованного двоичного дерева):

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)
    deriving (Eq, Ord, Show, Read)
```

Как вариант, мы могли бы создать параметризованное двоичное, у которого каждая вершина содержала бы значение типа a:

```
data ATree a = ALeaf a | ABranch (ATree a) a (ATree a)
    deriving (Eq, Ord, Show, Read)

> (Leaf 1) `Branch` (Leaf 2)
Branch (Leaf 1) (Leaf 2)

> y =  ABranch (ALeaf 1) 2 (ALeaf 3)
ABranch (ALeaf 1) 2 (ALeaf 3)
```

Кстати, в Haskell действует соглашение, благодаря которому конструкторы данных тоже можно делать с помощью произвольных символов и использовать их «инлайн», но тогда они должны начинаться с (:).

Infix type constructors, classes, and type variables

Можно переписать

```
data TreE a = LeaF a | (TreE a) :^: (TreE a)
    deriving (Eq, Ord, Show, Read)

> (LeaF 1) :^: (LeaF 2)
LeaF 1 :^: LeaF 2
```

Для первого типа данных мы можем определить ряд полезных функций, например:

```
fringe :: Tree a -> [a]
fringe (Leaf x) = [x]
fringe (Branch left right) =
   fringe left ++ fringe right
```

```
quantity :: Tree a -> Int
quantity (Leaf _ ) = 1
quantity (Branch l r) = quantity l + quantity r + 1

height :: Tree a -> Int
height (Leaf _ )    = 1
height (Branch l r) = 1 + (max (height l) (height r))

> quantity $ (Leaf 1) `Branch` (Leaf 2)
3
```

для высоты дерева

```
> height $ (Leaf 1) `Branch` (Leaf 2)
2
> height $ (Leaf 1) `Branch` ((Leaf 2) `Branch` (Leaf 2))
3
```

и для его «бахромы»

```
> fringe $ (Leaf 1) `Branch` ((Leaf 2) `Branch` (Leaf 2))
[1,2,2]
```

Более общий тип деревьев можно определять с помощью модуля `Data.Tree`

```
data Tree a = Node {rootLabel :: a, subForest :: Forest a}
   deriving ( Eq, Ord, Show, ...)

type Forest a = [Tree a]
```

Таким образом, *лес* — это список деревьев. Сам тип деревьев параметризирован с помощью базового типа `a`. Метка поля или деконструктор `rootLabel` возвращает значение в вершине, а `subForest` — список деревьев. Таким образом, подвершин у разных вершин или нод может быть разное количество и даже пустое множество.

Data.Tree

Такие деревья иногда называют «розовыми кустами» (eng. Rose tree). Вот более ясное определение:

```
data RoseTree a = RoseTree a [RoseTree a]
```

wikipedia: Rose tree

wiki.haskell: Rose tree

Далее — некоторые полезные утилиты.

```
flatten :: Tree a -> [a]
```

Возвращает список вершин в прямом обходе дерева (в глубину).

```
  a
 / \     => [a,b,c]
b   c
```

Например,

```
flatten (Node 1 [Node 2 [], Node 3 []]) == [1,2,3]
```

Функция

```
levels :: Tree a -> [[a]]
```

возвращает список каждого этажа дерева:

```
  a
 / \    => [[a], [b,c]]
b   c
```

и получаем

```
levels (Node 1 [Node 2 [], Node 3 []]) == [[1],[2,3]]
```

Забавная функция текстового (ASCII) рисования дерева (в `ghci` (проверено под Windows) возвращает строку с экранированным переводом строк, т.е. не работает, с нормальной компиляцией — работает):

```
drawTree :: Tree String -> String Source#

putStr $ drawTree $ fmap show (Node 1 [Node 2 [], Node 3 []])

1
|
+- 2
|
`- 3
```

Теперь более полезные функции. В следующих лекциях мы будем говорить о классе функтора и его методе `fmap`. Не используя специфики функторов, так сказать «на пальцах говоря», для деревьев `fmap` работает как **`map`** для списков, т.е. позволяет рекурсивно обойти дерево и применить заданную в аргументе функцию `f`.

```
fmap :: (a -> b) -> Tree a -> Tree b
fmap f (Node x ts) = Node (f x) (map (fmap f) ts)
```

И, наконец, свертки. Есть собственный вариант, основанный на обходе дерева в глубину:

```
foldTree :: (a -> [b] -> b) -> Tree a -> b
foldTree f = go where
    go (Node x ts) = f x (map go ts)
```

Примеры использования. Суммирование значений, содержащихся в вершинах:

```
foldTree (\x xs -> sum (x:xs)) (Node 1 [Node 2 [], Node 3 []]) == 6
```

Нахождение максимума:

```
foldTree (\x xs -> maximum (x:xs)) (Node 1 [Node 2 [], Node 3 []]) == 3
```

И есть воплощения класса `Foldable` с привычными методами **`foldr`**, **`foldl`** и т.п. для этого типа данных. Например,

```
> :m Data.Tree
> foldr1 (+) (Node 1 [Node 2 [], Node 3 []])
6
```

Data.Tree

wiki.haskell: Research papers/ Trees

wikibooks: Other data structures

wikibooks: Foldable

Массивы в Haskell

Массивы не являются естественными структурами в функциональных языках программирования, хотя бы в силу того, что не могут вноситься изменения для элементов.

> В идеале массивы в функциональном языке следовало бы рассматривать просто как функции из индексов в значения, но с прагматической точки зрения для гарантии эффективного доступа к элементам массива мы должны быть уверены, что мы можем воспользоваться специальными свойствами этих функций, изоморфных конечному непрерывному подмножеству целых чисел. Поэтому Haskell рассматривает массивы не как функции с операцией применения, а как абстрактный тип данных с операцией индексирования.

Мягкое введение в Haskell-2: Массивы

Следовательно, единственное, что может быть полезным, если нам предстоит иметь дело с массивом — это константное время доступа к произвольным элементам.

Отметим, что массивы должны подключаться дополнительным модулем `Data.Array`

```
import Data.Array
```

Data.Array

Типы индексов

Типы индексов могут быть как числовые, так и некоторые другие. Для этого определен специальный класс `Ix`

```
class (Ord a) => Ix a where
    range      :: (a,a) -> [a]
    index      :: (a,a) a -> Int
    inRange    :: (a,a) -> a -> Bool
```

Воплощения обеспечиваются для типов, подобных `Int`, `Integer`, `Char`, `Bool` , а также для типов перечислений (подобных типу `Color` из предыдущей лекции). Кроме того, обеспечиваются автоматические воплощения класса для кортежей из указанных выше типов вплоть до размера 15.

Data.Ix

Указанные примитивные типы рассматриваются как индексы одномерного массива, а кортежи — как индексы для многомерных прямоугольных массивов.

Методы класса `Ix`

Первый аргумент каждой операции класса **`Ix`** — это пара индексов; она обычно задаёт границы массива (первый и последний индекс). Например, границами 10-элементного массива с отсчётом от 0 и индексами типа **`Int`** будет пара (0,9), а матрица 100 на 100 с отсчётом от 1 может иметь границы ((1,1),(100,100)).

`range` принимает пару границ и производит упорядоченный список индексов, лежащих между этими границами. Например,

```
> :m Data.Array
> range (0,4)
[0,1,2,3,4]
> range ((0,0),(1,2))
[(0,0),(0,1),(0,2),(1,0),(1,1),(1,2)]
```

`inRange` определяет, лежит ли индекс внутри заданной пары границ (для типов кортежей такая проверка осуществляется покомпонентно).

```
> :m Data.Array
> inRange ((1,1),(100,100)) (2,2)
True
```

`index` позволяет адресовать некоторый элемент массива: для заданной пары границ и индекса в их диапазоне эта операция производит ординальное число индекса внутри диапазона, отсчитываемое от 0; например:

```
> :m Data.Array
> index (1,9) 2
1
> index ((0,0),(1,2)) (1,1)
4
```

Класс индексов **`Ix`** содержится в модуле `Data.Ix`, который автоматически подгружается вместе с модулем `Data.Array`.

Data.Ix

Создание массивов

Тип массива описывается декларацией, чьи детали для нас спрятаны в недрах соответствующего модуля (это абстрактный тип данных):

```
data Array i e
```

Здесь `i` — тип индексов, а `e` — тип элементов массива. Для массивов определены воплощения некоторых наиболее важных классов и методов, таких как **`Eq`**, **`Ord Show`**, **`Read`**, методов **`foldl`**, **`foldr`** и т.п.

Функция создания монолитного массива в Haskell формирует массив по его границам и списку пар индекс-значение (ассоциативный список):

```
array :: (Ix a) => (a,a) -> [(a,b)] -> Array a b
```

Вот, например, определение массива квадратов чисел от 1 до 100:

```
squares =  array (1,100) [(i, i*i) | i <- [1..100]]
```

Как видим, массив создается из списка пара-значение, как ассоциативный массив.

Если размер списка окажется короче, то ошибка будет только при обращении к несуществующему элементу, например:

```
*Main> sq = array (1,100) [(i, i*i) | i <- [1..90]]
*Main> sq ! 3
9
*Main> sq ! 90
8100
*Main> sq ! 91
*** Exception: (Array.!): undefined array element
```

А если размер индекса окажется короче, то ошибка возникает при попытке обращения к любому элементу:

```
*Main> sqq = array (1,90) [(i, i*i) | i <- [1..100]]
*Main> sqq ! 91
*** Exception: Ix{Integer}.index: Index (91) out of range ((1,90))
*Main> sqq ! 89
*** Exception: Ix{Integer}.index: Index (91) out of range ((1,90))
*Main> sqq ! 80
*** Exception: Ix{Integer}.index: Index (91) out of range ((1,90))
*Main> sqq ! 8
*** Exception: Ix{Integer}.index: Index (91) out of range ((1,90))
*Main> sqq ! 1
*** Exception: Ix{Integer}.index: Index (91) out of range ((1,90))
```

Как видно, ошибка проявляется сильно по-разному!

Массив можно сделать и проще, с помощью функции **listArray**:

```
squares2 =  listArray (1,100) [i^2 | i <- [1..100]]
```

И вот так работает **show**:

```
*Main> show squares2
"array (1,100) [(1,1),(2,4),(3,9),(4,16),(5,25),(6,36),(7,49),(8,64),
(9,81),(10,100),(11,121),(12,144),(13,169),(14,196),(15,225),(16,256),
(17,289),(18,324),(19,361),(20,400),(21,441),(22,484),(23,529),(24,576),
(25,625),(26,676),(27,729),(28,784),(29,841),(30,900),(31,961),(32,1024),
(33,1089),(34,1156),(35,1225),(36,1296),(37,1369),(38,1444),(39,1521),
(40,1600),(41,1681),(42,1764),(43,1849),(44,1936),(45,2025),(46,2116),
(47,2209),(48,2304),(49,2401),(50,2500),(51,2601),(52,2704),(53,2809),
(54,2916),(55,3025),(56,3136),(57,3249),(58,3364),(59,3481),(60,3600),
(61,3721),(62,3844),(63,3969),(64,4096),(65,4225),(66,4356),(67,4489),
(68,4624),(69,4761),(70,4900),(71,5041),(72,5184),(73,5329),(74,5476),
(75,5625),(76,5776),(77,5929),(78,6084),(79,6241),(80,6400),(81,6561),
(82,6724),(83,6889),(84,7056),(85,7225),(86,7396),(87,7569),(88,7744),
(89,7921),(90,8100),(91,8281),(92,8464),(93,8649),(94,8836),(95,9025),
(96,9216),(97,9409),(98,9604),(99,9801),(100,10000)]"
```

Пример создания двумерных массивов:

```
import Data.Array

n = 5::Int
m = 6::Int

list = [ (fromIntegral j)**(1/(fromIntegral i)) |
  j <- [1..n], i <- [1..m] ] :: [Double]

arr = listArray ((1,1),(n,m)) list

> arr ! (3,2)
1.7320508075688772
```

Или явным указанием пар:

```
arC = array ((1,1),(2,2)) [((2,1),'C'),((1,2),'B'),
       ((1,1),'A'),((2,2),'D')]
```

Вывод же будет упорядочен:

```
> arC
array ((1,1),(2,2)) [((1,1),'A'),((1,2),'B'),
((2,1),'C'),((2,2),'D')]
```

Доступ к массивам

Доступ к отдельному элементу осуществляется почти как для списков:

```
(!) :: Ix i => Array i e -> i -> e  -- infixl 9

> squares2 ! 3
9
> squares2 ! 0
*** Exception: Ix{Integer}.index: Index (0)
out of range ((1,100))
```

Видим, что осуществляется контроль границ индексов массива.

Функция **bounds** возвращает границы индексов, в которых массив был создан.

```
bounds :: Array i e -> (i, i)

> bounds squares2
(1,100)
```

Функция **elems** перечисляет все элементы массива в порядке индексации:

```
elems :: Array i e -> [e]

> elems squares2
[1,4,9,16,25,36,49,64,81,100,121,144,169,196,225,
256,289,324,361,400,441,484,529,576,625,676,729,
784,841,900,961,1024,1089,1156,1225,1296,1369,1444,
1521,1600,1681,1764,1849,1936,2025,2116,2209,2304,
```

```
2401,2500,2601,2704,2809,2916,3025,3136,3249,3364,
3481,3600,3721,3844,3969,4096,4225,4356,4489,4624,
4761,4900,5041,5184,5329,5476,5625,5776,5929,6084,
6241,6400,6561,6724,6889,7056,7225,7396,7569,7744,
7921,8100,8281,8464,8649,8836,9025,9216,9409,9604,
9801,10000]
```

Функция **assocs** выводит список пар индекс-элемент в порядке индексации:

```
assocs :: Ix i => Array i e -> [(i, e)]
```

```
> assocs squares2
[(1,1),(2,4),(3,9),(4,16),(5,25),(6,36),(7,49),
(8,64),(9,81),(10,100),(11,121),(12,144),(13,169),
(14,196),(15,225),(16,256),(17,289),(18,324),
(19,361),(20,400),(21,441),(22,484),(23,529),
(24,576),(25,625),(26,676),(27,729),(28,784),
(29,841),(30,900),(31,961),(32,1024),(33,1089),
(34,1156),(35,1225),(36,1296),(37,1369),(38,1444),
(39,1521),(40,1600),(41,1681),(42,1764),(43,1849),
(44,1936),(45,2025),(46,2116),(47,2209),(48,2304),
(49,2401),(50,2500),(51,2601),(52,2704),(53,2809),
(54,2916),(55,3025),(56,3136),(57,3249),(58,3364),
(59,3481),(60,3600),(61,3721),(62,3844),(63,3969),
(64,4096),(65,4225),(66,4356),(67,4489),(68,4624),
(69,4761),(70,4900),(71,5041),(72,5184),(73,5329),
(74,5476),(75,5625),(76,5776),(77,5929),(78,6084),
(79,6241),(80,6400),(81,6561),(82,6724),(83,6889),
(84,7056),(85,7225),(86,7396),(87,7569),(88,7744),
(89,7921),(90,8100),(91,8281),(92,8464),(93,8649),
(94,8836),(95,9025),(96,9216),(97,9409),(98,9604),
(99,9801),(100,10000)]
```

И, наконец, функция

```
indices :: Ix i => Array i e -> [i]
```

```
> indices squares2
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,
22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,
40,41,42,43,44,45,46,47,48,49,50,51,52,53,54,55,56,57,
58,59,60,61,62,63,64,65,66,67,68,69,70,71,72,73,74,75,
76,77,78,79,80,81,82,83,84,85,86,87,88,89,90,91,92,93,
94,95,96,97,98,99,100]
```

выводит список индексов по возрастанию.

Инкрементальное обновление массива

Специальная функция (//) производит *инкрементальное* обновление массива:

```
(//) :: Ix i => Array i e -> [(i, e)] -> Array i e
   -- infixl 9
```

Инкрементальное означает, что к старым значениям добавляются новые (не вместо них!)

wikipedia: виды резервного копирования

Например, для квадратной матрицы символов, заданной выше, мы можем сделать такое обновление, в нашем случае это будет выглядеть как замена диагональных элементов на символ `'O'`:

```
arC = array ((1,1),(2,2)) [((2,1),'C'),((1,2),'B'),
       ((1,1),'A'),((2,2),'D')]

ar2 = arC // [((i,i), 'O') | i <- [1..2]]

> ar2
array ((1,1),(2,2)) [((1,1),'O'),((1,2),'B'),((2,1),'C'),((2,2),'O')]
```

Настоящее изменение элементов массива «in-place» для программирования на Haskell возможно только внутри **`IO`** и исполняемой части `main`. Мы это обсудим в соответствующей теме, на примере типа данных `Vector`.

Обходы и свертки массивов

Для массивов доступен аналог списочной функции **`map`** в перегружаемой форме `fmap`:

```
lst = [1..10] :: [Int]
ary = listArray (1,10) lst
-- aa2 = Data.Array.map (^2) aa1 -- not working
ary2 = fmap (^2) ary
```

с таким результатом работы:

```
*Main> ary2
array (1,10) [(1,1),(2,4),(3,9),(4,16),(5,25),(6,36),(7,49),
(8,64),(9,81),(10,100)]
```

Также доступны все (кроме **`foldl'`**) формы свёрток:

```
*Main> foldl (+) 0 ary
55
*Main> foldr (+) 0 ary
55
*Main> foldl1 (+) ary
55
*Main> foldr1 (+) ary
55
```

функции **`sum`**, **`maximum`**, **`minimum`** и **`elem`**.
