Обработка Maybe и List

Ранее мы встречали контейнерный тип `Maybe` и работали с ним с помощью сопоставления с образцом:

```
data Maybe a = Nothing | Just a
incMaybe (Just x) = Just (x+1)
incMaybe
```

Однако, достаточно часто необходимо применить некоторую функцию к значению в `Maybe`, если оно есть, и вернуть новое значение. Было бы неплохо иметь некоторую функцию `myfilter`, которую можно было бы применять следующим образом:

```
myfilter :: Maybe a -> (a -> b) -> Maybe b
myfilter (Just 2) (+1) == Just 3
```

Тогда мы можем написать функцию `incMaybe`, которая осуществляет инкремент значения как:

```
incMaybe x  = myfilter x (+1)
```

Однако, иногда такая операция может не возвращать значения, то есть должна быть возможность породить `Nothing`. Это значит, что функция `myfilter` должна быть объявлена как:

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

К счастью, в стандартной библиотеке есть оператор `>>=`, реализующий такую функциональность. С его использованием мы можем написать функцию `incMaybe` следующим образом:

```
incMaybe m = m >>= (\x -> Just (x+1))
```

Или используя встроенную функцию `return`, которая заворачивает значение обратно в контейнер следующим образом:

```
incMaybe m = m >>= (\x -> return (x+1))
```

В данном случае `>>=` и `return` определены как:

```
Just x >>= f = f x
Nothing >>= f = Nothing


return x = Just x
```

С их использованием можно определить функцию суммирования двух значений `Maybe` как:

```
sumMaybe :: Maybe Int -> Maybe Int -> Maybe Int
sumMaybe m1 m2 = m1 >>= (\x -> m2 >>= (\y -> return (x + y)))
```

Поскольку в случае больших выражений такая запись быстро становится громоздкой, то для нее предусмотрена специальная синтаксическая конструкция `do`, которая используется следующим

образом:

```
sumMaybe1 m1 m2 = do
  x <- m1
  y <- m2
  return x + y
```

Теперь давайте посмотрим на смысл оператора `>>=` и функции `return`. `Maybe` можно воспринимать как ящик, в котором, возможно, лежит какое-то значение, а, возможно, не лежит ничего. Тогда функция `return` просто формирует новый ящик с заданным значением внутри. Оператор `>>=` осуществляет следующее преобразование — он открывает ящик, и если он пуст, то формирует новый пустой ящик, а если полон — то как-то преобразует значение и возвращает его в новом ящике.

Если же вспомнить про ленивость Haskell, то можно сказать, что оператор `>>=` возвращает новый ящик на основе заданного, который в момент открытия осуществляет заданное преобразование и создает внутри значение (или остается пустым).

Если подумать над этой аналогией с ящиками, то можно расширить ее на другие контейнеры, например, на списки. Список можно воспринимать как ящик, в котором в определенном порядке лежат несколько значений. И для него аналогично можно определить операции `return` и `>>=`. `return` формирует новый список, содержащий один элемент, а действие `>>=` аналогично случаю с `Maybe` с той лишь разницей, что в списке несколько значений и преобразование применяется к каждому из них, а потом результаты упаковываются в один ящик. С помощью них мы можем определить операцию инкремента всех элементов списка как:

```
incList l = l >>= (\x -> return (x+1))
```

Можно обратить внимание, на то, что инкремент списка выглядит точно так же, как инкремент `Maybe`. Таким образом описанный подход предоставляет универсальный способ обработки контейнеров.

Поэтому операции `>>=` и `return` определены в стандартном классе `Monad`, который реализуется типами `Maybe` и `List`.

Для списка эти операции реализуются следующим образом:

```
return x = [x]

l >>= f = concat (map f l)
```

Теперь рассмотрим, что выдает функция суммирования `Maybe`, если применить ее к спискам. Выбирается каждый элемент первого списка, затем для него выбирается каждый элемент второго списка и они суммируется. Таким образом, мы получаем список попарных сумм элементов двух списков. Это должно напомнить списочные выражения — и действительно, списочные выражения, подобно конструкции `do`, являются всего лишь синтаксическим сокращением для операций монад.

Посмотрим, как может быть записано декартово произведение списков. С использованием списочных выражений:

```
decProduct1 a b = [ (x,y) | x <- a, y <- b ]
```

Или операций над монадами:

```
decProduct2 a b = a >>= (\x -> b >>= (\y -> return (x,y)))
```

Или конструкции `do`:

```
decProduct3 a b = do
  x <- a
  y <- b
  return (x,y)
```

Теперь рассмотрим задачу выбора только тех пар, в которых первый элемент больше. В списочных выражениях это делалось следующим образом:

```
[ (x,y) | x <- a, y <- b, x > y ]
```

В случае конструкции `do` можно записать это следующим образом:

```
do
  x <- a
  y <- b
  if x > y then return (x,y) else []
```

Однако использование `[]` делает конструкцию `do` неуниверсальной – она не может быть применена, например, для `Maybe`. При этом пустой список представляет то же самое, что и `Nothing` – отсутствие результата. Для такой ситуации в классе `Monad` определена функция `fail`, которая принимает строку с обращением об ошибке и для списков или `Maybe` реализована как возвращающая пустое значение. С ее использованием функция может быть переписана как:

```
do
  x <- a
  y <- b
  if x > y then return (x,y) else fail “”
```

Другое улучшение этого кода состоит в том, чтобы воспользоваться особенностью обработки сопоставления с образцом в конструкции `do` и написать:

```
do
  x <- a
  y <- b
  True <- x > y
  return (x,y)
```

В случае, если сопоставление закончится неудачей, то будет вызвана функция `fail`, куда будет передана информация об ошибке. В данном случае это приведет к порождению пустого списка или значения `Nothing`.

Функции ввода-вывод

Теперь пора разобраться с вводом-выводом на Haskell. Предположим, что мы хотели бы сделать

функцию ввода строки `getLine`. Какой тип она могла бы иметь? В императивном языке подошло бы что-нибудь вроде:

```
getLine :: String
```

Однако в Haskell такое не пройдет – все функции должны быть чистыми, то есть не зависеть от внешнего состояния и не иметь побочных эффектов. Очевидно, что функция `getLine` такой не является – ее результат зависит от состояния устройств ввода. Следовательно, чтобы сделать ее чистой, можно попытаться перенести это состояние в параметр. Например вот так:

```
getLine :: World -> String
```

Однако, это еще не все. Функция считывания строки не только зависит от состояния мира, но и меняет его в качестве своего побочного эффекта. Действительно, ввод строки приводит к тому, что эта строка уже не будет считана при повторном вызове функции. Если пользоваться нашим подходом преобразования глобального состояния в параметры, то это значит, что функция должна возвращать новое состояние мира как часть результата (например, как элемент кортежа):

```
getLine :: World -> (String,World)
```

Вот теперь мы наконец получили чистую функцию для ввода строки.

Однако, давайте разберемся, что же теперь станет результатом вызова функции `getLine`? Это некоторая функция, которая принимает состояние мира, возвращает новое состояние мира и какой-то результат. Назовем такую функцию ***действием*** ввода-вывода. Таким образом, `getLine` возвращает не строку, а действие по чтению строки.

Рассмотрим теперь какую-нибудь другую функцию, например, вывод строки `putStr`. Она принимает в качестве аргумента строку и, подобно `getLine` должна принимать старое состояние мира и возвращать новое. То есть ее тип может быть записан как:

```
putStr :: String -> World -> World
```

Можно обратить внимание на то, что `putStr "a"` также возвращает действие ввода-вывода, однако на этот раз без результата. Поскольку эти действия возникают так часто, то можно выделить общий тип, описывающий действие:

```
type IO a = World -> (a,World)
```

В таком случае функции `getLine` и `putStr` будут иметь следующие типы:

```
getLine :: IO String
putStr :: String -> IO ()
```

Теперь рассмотрим, как эти функции ввода-вывода могут быть использованы для построения полноценной программы, осуществляющей ввод-вывод.

Запуск программы в Haskell начинается с вызова функции `main`, которая имеет тип `IO ()`, то есть возвращает действие по вводу-выводу, не имеющее никакого результата.

Такой же тип возвращаемого значения имеет функция `putStr`, а значит мы можем написать что-то вроде:

```
main = putStr "Hello world!"
```

Это программа, которая осуществляет вывод некоторой фиксированной строки. Хотя это уже позволяет писать полезные программы (например, осуществляющие вывод 1000-го простого числа), обычно все-таки требуется вводить некоторые данные. Но функция `getLine` не

возвращает строку, она возвращает действие по ее чтению. А значит необходим какой-то способ комбинировать действия ввода-вывода (логично ведь, что ввести строку, преобразовать, а затем вывести – это такое же действие ввода-вывода).

Ввод-вывод и монады

В таком комбинировании действий нам помогут монады. Действительно, IO является экземпляром класса Monad и операция >>= для нее осуществляет следующее. Если `a :: IO p` – это некоторое действие, а `f :: p -> IO q` - функция, то `a >>= f` формирует новое действие, которое состоит в том, чтобы сначала выполнить действие `a`, затем передать его результат в `f`, и выполнить результат `f`. Например, рассмотрим:

```
getLine >>= putStr
```

Это действие, которое имеет тип `IO ()`, и осуществляет считывание строки и ее запись.

Возникает вопрос, каким образом должна быть определена операция >>= для этого. Ответ становится понятен, если вспомнить, что `IO a == World -> (a,World)`, тогда:

```
(>>=) :: (World->(a,World))->(a->World->(b,World))->World->(b,World)
```

И может быть определен следующим образом:

```
(>>=) a f w0 = f r w1 where (r,w1) = a w0
```

То есть мир, возвращаемый первым действием, передается во второе действие.

Теперь мы получили способ комбинирования действий и можем строить более сложные действия из более простых. Например, программа, которая дублирует введенную строку:

```
main :: IO ()
main = getLine >>= (\x -> putStrLn x >>= (\y -> putStrLn x))
```

Здесь стоит обратить внимание на то, что переменная y здесь не используется. Поскольку это достаточно распространенная ситуация, в определении класса Monad предусмотрен специальный оператор, определенный следующим образом:

```
m >> k           =  m >>= \_ -> k
```

С ним программа может быть записана следующим образом:

```
main = getLine >>= (\x -> putStrLn x >> putStrLn x)
```

Этот же пример можно записать с использованием конструкции `do`:

```
main = do
  x <- getLine
  putStrLn x
  putStrLn x
```

Рассмотрим пример другой программы — удвоение всех вводимых строк:

```
main = getLine >>= (\x -> putStrLn x >> putStrLn x >> main)
```

Эта программа не обрабатывает конец ввода и потому может завершиться только ошибкой, когда считать будет нечего. Сделаем удвоение строк до пустой строки:

```
main = getLine >>= (\x -> if x == "" then return () else putStrLn x
>> putStrLn x >> main)
```

Или с использованием конструкции `do`:

```
main = do
  x <- getLine
  if x == "" then return ()
  else do
    putStrLn x
    putStrLn x
    main
```

Рассмотрим, как может быть организовано считывание целого числа:

```
getInt :: IO Int
getInt = getLine >>= (\s -> return (read s))
```

Или с использованием do:

```
getInt = do
  s <- getLine
  return (read s)
```

С ее помощью можно написать программу, вычисляющую длину вектора:

```
main = do
  x <- getInt
  y <- getInt
  z <- getInt
  return (sqrt (fromIntegral (x*x + y*y + z*z)))
```

Работа с файлами

Кроме работы со стандартным вводом/выводом в часто необходима работа с файлами. В Haskell для этого есть две группы функций: высокоуровневые и низкоуровневые. Сейчас рассмотрим три основные высокоуровневые функции:

- `readFile :: String -> IO String` – прочитать содержимое файла по его имени;
- `writeFile :: String -> String -> IO ()` – перезаписать содердимое файла;
- `appendFile :: String -> String -> IO ()` – добавить содержимое в конец файла.

Важной особенностью этих функций является то, что они не загружают файл в память полностью, а используют ленивость Haskell. Поэтому, например, следующая программа копирования файлов не требует много памяти:

```
main = do
  from <- getLine
  to <- getLine
  content <- readFile from
  writeFile to content
```

Другой схожей функцией, но используемой при работе со тандартным вводом\выводом является функция `interact`, которая формирует действие ввода-вывода на основе переданной в нее функции преобразования строки:

```
interact :: (String -> String) -> IO ()
```

С ее помощью можно написать программу, осуществляющую фильтрацию входного потока:

```
main = interact (\s -> [c | c <- s, c >= '0', c <= '9'])
```