Конструирование списков

Оставшимся нерассмотреным базовым типом данных в Haskell являются списки. В отличии от лисповских списков, списки Haskell хранят значения одного и того же типа, например `Int`:

```
list1 = [1,2,3,4,5]
list2 :: [Int]
list2 = [1,2]
list3 = []
```

Как видно из примера, простейшим способом задания списка является перечисление всех его элементов через запятую в квадратных скобках, а тип списка записывается как тип его элементов в квадратных скобках.

Другим способом создания списка является вызов конструктора `(:)`, который формирует новый список на основе значения первого элемента и хвоста (аналогично лисповскому cons):

```
list4 = 0:list2
```

С использованием этого конструктора можно отметить следующее:

```
[1,2,3,4] == 1:2:3:4:[]
```

Также Haskell позволяет использовать специальный синтаксис для сокращения записи списков из последовательных элементов. Например, чтобы создать список чисел от 1 до 10 можно записать:

```
list4 = [1..10]
```

При использовании такого синтаксиса интерпретатор использует фукнция `succ` для перебора элементов. Также, можно задать список из элементов, получающихся друг из друга несколькими вызовами `succ`, например список чисел от 0 до 100 кратных 5:

```
list5 = [0,5..100]
```

В данном случае каждый элемент получается из предыдущего путем 5 вызовов `succ`.

Работа со списками

Для написания функций, обрабатывающих списки, необходимо производить их сопоставление с образцом. Для этого используются конструкторы `(:)` и `[a,b,c]`. Например, функция вычисления длины списка может быть записана следующим образом:

```
listLength :: [a] -> Int
listLength []    = 0
listLength (h:t) = 1 + listLength t
```

Функция проверки вхождения элемента в список:

```
element :: a -> [a] -> Bool
element _ [] = False
element e (h:t) | h == e    = True
                | otherwise = element e t
```

Склейка двух списков:

```
append :: [a] -> [a] -> [a]
append [] l    = l
append (h:t) l = h:append t l
```

Объединение множеств:

```
setUnion :: [a] -> [a] -> [a]
setUnion [] l = l
setUnion (h:t) l | element h l  = setUnion t l
                 | otherwise = h:setUnion t l
```

Пересечение множеств:

```
setIntersect :: [a] -> [a] -> [a]
setIntersect [] _ = []
setIntersect (h:t) l | element h l  = h:setIntersect t l
                     | otherwise = setIntersect t l
```

Преобразование вложенного списка в плоский:

```
flatten :: [[a]] -> [a]
flatten []          = []
flatten ([]:t)      = flatten t
flatten ((h:t1):t2) = h:flatten (t1:t2)
```

Стандартные функции работы со списками

Стандартная библиотека включает большое количество операторов и функций работы со списками. Сначала рассмотрим два оператора - `!!` и `++`.

Оператор `!!` используется для получения элемента списка по его индексу (индексация элементов списка начинается с нуля). Например:

```
[1..5] !! 1 == 2
```

Рассмотрим, как он может быть определен:

```
(h:_) !! 0 = h
(h:t) !! i = t !! (i-1)
```

Оператор `++` используется для скеливания списков:

```
[1..3] ++ [4,5] == [1..5]
```

Это аналог функции `append`, рассмотренной выше.

Для проверки вхождения элемента в список определена функция `elem`:

```
elem 3 [1..5] == True
```

Для получения длины списка – `length`:

```
length [2..5] == 4
```

Для получения первого элемента и хвоста списка определены функции `head` и `tail`. Аналогичные функции `last` и `init` определены для получения последнего элемента и начала списка.

Также в стандартной библиотеке определена пара функций, позволяющих получить или отбросить первые несколько элементов списков:

```
take 3 [1..10] == [1,2,3]
drop 3 [1..10] == [4..10]
```

Функции высшего порядка для работы со списками

Рассмотрим задачу удвоения всех элементов заданного списка. Она может быть реализована следующим образом:

```
doubleElems []    = []
doubleElems (h:t) = (2*h):doubleElems t
```

Можно заметить, что аналогично реализуются функции для применения любой операции ко всем элементам списка. В таком случае можно реализовать функцию, которая будет принимать эту операцию в качестве аргумента:

```
myMap f [] = []
myMap f (h:t) = f h:myMap f t
```

Тогда функция `doubleElems` реализуется как:

```
doubleElems = myMap (2*)
```

Эта функция определена в стандартной библиотеке и называется `map`.

Теперь рассмотрим суммирование чисел-элементов списка:

```
listSumm [a]    = a
listSumm (h:t) = h + listSumm t
```

И нахождение максимального значения в списке:

```
listMaximum [a] = a
listMaximum (h:t) = max h (listMaximum t)
```

Видно, что как и в предыдущем случае, эти функции отличаются только одной операцией, поэтому можно определить более общую функцию, получающую необходимую операцию в качестве параметра. Такая функция определяется следующим образом:

```
myFoldr1 f [s] = s
myFoldr1 f (h:t) = f h (myFoldr1 f t)
```

Сумма элементов и максимум списка определяются:

```
listSumm1 = myFoldr1 (+)
listMaximum1 = myFoldr1 max
```

Аналогичная фукнция определена в стандартной библиотеке и называется `foldr1`. Кроме нее

существуют функции `foldl1` (свертка слева направо), а также `foldr` и `foldl` в которых задается начальное значение для свертки:

`foldr1 f [`x_1`, `x_2`, … `x_N`] == f `x_1` (f `x_2` … (f `x_{N-1}` `x_N`) …)`

`foldl1 f [`x_1`, `x_2`, … `x_N`] == f (f … (f `x_1` `x_2`) … `x_{N-1}`) `x_N

`foldr f a [`x_1`, `x_2`, … `x_N`] == f `x_1` (f `x_2` … (f `x_N` a) …)`

`foldl f a [`x_1`, `x_2`, … `x_N`] == f (f … (f a `x_1`) … `x_{N-1}`) `x_N

Например:

```
foldl (+) 0 [1,2,3] == 6
foldr (+) 0 [1,2,3] == 6
foldl1 (+) [1,2,3] == 6
foldr1 (+) [1,2,3] == 6
```

Другими полезными функциями высшего порядка являются:

- Функция `filter`, принимающая предикат и выбирающая из списка значения, удовлетворяющие этому предикату.
- Функции `takeWhile` и `dropWhile`, которые принимают предикат и выбирают (или отбрасывают) начальную последовательность элементов, удовлетворяющих этому предикату.
- Функции `any` и `all`, принимающие предикат и проверяющие, что хотя бы одно (или все) значения в списке удовлетворяют предикату.

Например:

```
filter (>0) [-1,4,7,-3,0,2] == [4,7,2]
takeWhile (>-2) [-1,4,7,-3,0,2] == [-1,4,7]
dropWhile (<3) [-1,2,0,4,-3,7] == [4,-3,7]
```

Еще одной полезной функций высшего порядка является функция `zipWith`, позволяющая реализовать поэлементное объединение двух списков, например, их поэлементное сложение или произведение, максимум и т.п.

Функция `zipWith` может быть определена следующим образом:

```
zipWith _ [] _ = []
zipWith _ _ [] = []
zipWith f (h1:t1) (h2:t2) = f h1 h2:zipWith f t1 t2
```

Пример её использования:

```
zipWith (+) [1,2,3] [4,5,6] == [5,7,9]
```

Списочные выражения

Haskell содержит синтаксическую конструкцию, позволяющую во многих случаях упростить обработку списков – списочные выражения (list comprehensions).

Списочное выражение записывается в квадратных скобках и состоит из двух частей, разделенных вертикальной чертой. В левой части записывается выражения для преобразования перебираемых значений в результирующие, а в правой находится последовательность выражений трех типов:

- Выборка из списка (`x <- l`);
- Условие;
- Выражение `let`. Какой-нибудь пример с использованием let!!

Принцип действия списочного выражения состоит в том, чтобы выбрать значения из заданных списков, удовлетворяющие заданным условиям, и вернуть результаты применения некоторых выражений к ним.

Через списочные выражения могут быть легко выражены функции `map` и `filter`:

```
mapC f l = [ f x | x <- l ]
filterC f l = [ x | x <- l, f l ]
```

Также через списочное выражение легко определяется декартово произвдение списков:

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

Теперь рассмотрим следующую задачу: дана некоторая функция двух аргументо `f`, а также неравномерная прямоугольная сетка, заданная в виде списков координат столбцов (`cols`) и строк (`rows`). Необходимо найти те точки, в которых функция принимает положительные значения.

Проще всего эта задача решается с использованием списочных выражений следующим образом:

```
points = [(x,y) | x <- cols, y <- rows, f x y > 0]
```