Лабораторная работа №5

1 Функции высшего порядка

Рассмотрим две задачи. Пусть задан список чисел. Необходимо написать две функции, первая из которых возвращает список квадратных корней этих чисел, а вторая — список их логарифмов. Эти функции можно определить так:

```
sqrtList [] = []
sqrtList (x:xs) = sqrt x : sqrtList xs

logList [] = []
logList (x:xs) = log x : logList xs
```

Можно заметить, что эти функции используют один и тот же подход, и все различие между ними заключается в том, что в одной из них для вычисления элемента нового списка используется функция квадратного корня, а в другой — логарифм. Можно ли абстрагироваться от конкретной функции преобразования элемента? Оказывается, можно. Вспомним, что в Haskell функции являются элементами «первого класса»: их можно передавать в другие функции в качестве параметров. Определим функцию `transformList`, которая принимает два параметра: функцию преобразования и преобразуемый список.

```
transformList f [] = []
transformList f (x:xs) = f x : transformList f xs
```

Теперь функции `sqrtList` и `logList` можно определить так:

```
sqrtList l = transformList sqrt l
logList l = transformList log l
```

Или, с учетом каррирования:

```
sqrtList = transformList sqrt
logList = transformList log
```

1.1 Функция `map`

В действительности функция, полностью аналогичная `transformList`, уже определена в стандартной библиотеке языка и называется `map` (от англ. map — отображение). Она имеет следующий тип:

```
map :: (a -> b) -> [a] -> [b]
```

Это означает, что ее первым аргументом является функция типа `a -> b`, отображающая значения произвольного типа `a` в значения типа `b` (вообще говоря, эти типы могут совпадать). Вторым аргументом функции является список значений типа `a`. Тогда результатом функции будет список значений типа `b`.

Функции, подобные `map`, принимающие в качестве аргументов другие функции, называются *функциями высшего порядка*. Их очень широко используют при написании функциональных программ. С их помощью можно явно отделить частные детали реализации алгоритма (например, конкретную функцию преобразования в `map`) от его высокоуровневой структуры (поэлементное преобразование списка). Алгоритмы, представленные с использованием функций высшего порядка, как правило, более компактны и наглядны, чем реализации, ориентированные на конкретные частности.

1.2 Функция `filter`

Следующим примером широко используемой функции высшего порядка является функция `filter`. По заданному предикату (функции, возвращающей булевское значение) и списку она возвращает список тех элементов, которые удовлетворяют заданному предикату:

```
filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs) | p x = x : filter xs
                | otherwise = filter xs
```

Например, функция, получающая из списка чисел его положительные элементы, определяется так:

```
getPositive = filter isPositive
isPositive x = x > 0
```

1.3 Функции `foldr` и `foldl`

Более сложным примером являются функции `foldr` и `foldl`. Рассмотрим функции, возвращающие сумму и произведение элементов списка:

```
sumList [] = 0
sumList (x:xs) = x + sumList xs

multList [] = 1
multList (x:xs) = x * multList xs
```

Здесь также можно увидеть общие элементы: начальное значение (0 для суммирования, 1 для умножения) и функция, комбинирующая значения между собой. Функция `foldr` является очевидным обобщением такой схемы:

```
foldr             :: (a -> b -> b) -> b -> [a] -> b
foldr f z []       = z
foldr f z (x:xs)   = f x (foldr f z xs)
```

Функция `foldr` принимает в качестве первого аргумента комбинирующую функцию (заметим, что она может принимать аргументы разных типов, но тип результата должен совпадать с типом второго аргумента). Вторым аргументом функции `foldr` является начальное значение для комбинирования. Третьим аргументом передается список. Функция осуществляет «свертку» списка в соответствие с переданными параметрами.

Для того, чтобы лучше понять, как работает функция `foldr`, запишем ее определение с использованием инфиксной нотации:

```
foldr f z []       = z
foldr f z (x:xs)   = x `f` (foldr f z xs)
```

Представим список элементов `[a,b,c,...,z]` с использованием оператора `:`. Правило применения функции `foldr` таково: все операторы `:` заменяются на применение функции `f` в инфиксном виде (`` `f` ``), а символ пустого списка `[]` заменяется на начальное значение комбинирования. Шаги преобразования можно изобразить так (предполагаем, что начальное значение равно `init`)

```
[a,b,c,...,z]
a : b : c : ... : []
a : (b : (c : (... (z : [])...)))
a `f` (b `f` (c `f` (... (z `f` init))))
```

С помощью функции `foldr` функции суммирования и умножения элементов списка определяется так:

```
sumList = foldr (+) 0
multList = foldr (*) 1
```

Рассмотрим, как вычисляются значения этих функций на примере списка `[1,2,3]`:

```
[1,2,3]
1 : 2 : 3 : []
1 : (2 : (3 : []))
1 + (2 + (3 +  0))
```

Аналогично для умножения:

```
[1,2,3]
1 : 2 : 3 : []
1 : (2 : (3 : []))
1 * (2 * (3 *  0))
```

Название функции происходит от английского слова fold — сгибать, складывать (например, лист бумаги). Буква r в названии функции происходит от слова right (правый) и показывает ассоциативность применяемой для свертки функции. Так, из приведенных примеров видно, что применение функции группируется вправо. Определение функции `foldl`, где l указывает на то, что применение операции группируется влево, приведено ниже:

```
foldl             :: (a -> b -> a) -> a -> [b] -> a
foldl f z []       = z
foldl f z (x:xs)   = foldl f (f z x) xs
```

Для ассоциативных операций, таких как сложение и умножение, функции `foldr` и `foldl` эквивалентны, однако если операция не ассоциативна, их результат будет отличаться:

```
Main>foldr (-) 0 [1,2,3]
2
Main>foldl (-) 0 [1,2,3]
-6
```

Действительно, в первом случае вычисляется величина $1-(2-(3-0)) = 2$, а во втором — величина $((0-1)-2)-3 = -6$.

1.4 Другие функции высшего порядка

В стандартной библиотеке определена функция `zip`. Она преобразует два списка в список пар:

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

Пример применения:

```
Prelude>zip [1,2,3] ['a','b','c']
[(1,'a'),(2,'b'),(3,'c')]
Prelude>zip [1,2,3] ['a','b','c','d']
[(1,'a'),(2,'b'),(3,'c')]
```

Заметьте, что длина результирующего списка равна длине самого короткого исходного списка.

Обобщением этой функции является функция высшего порядка `zipWith`, «соединяющая» два списка с помощью указанной функции:

```
zipWith                  :: (a->b->c) -> [a]->[b]->[c]
zipWith z (a:as) (b:bs)   = z a b : zipWith z as bs
zipWith _ _      _        = []
```

С помощью этой функции легко определить, например, функцию поэлементного суммирования двух списков:

```
sumList xs ys = zipWith (+) xs ys
```

или, с учетом каррирования:

```
sumList = zipWith (+)
```

2 Лямбда-абстракции

При использовании функций высшего порядка зачастую необходимо определять много небольших функций. Например, при определении функции `getPositive` нам пришлось определять дополнительную функцию `isPositive`, которая нужна только для того, чтобы проверить аргумент на положительность. С ростом объема программы необходимость придумывать имена для вспомогательных функций все больше мешает. Однако в языке Haskell, как и в лежащем в его основе

лямбда-исчислении, можно определять безымянные функции с помощью конструкции лямбда-абстракции.

Например, безымянные функции, возводящая свой аргумент в квадрат, прибавляющие единицу и умножающие на два, записывается следующим образом:

```
\x -> x * x
\x -> x + 1
\x -> 2 * x
```

Их теперь можно использовать в аргументах функций высших порядков. Например, функцию для возведения элементов списка в квадрат можно записать так:

```
squareList l = map (\x -> x * x) l
```

Функция `getPositive` может быть определена следующим образом:

```
getPositive = filter (\x -> x > 0)
```

Можно определять лямбда-абстракции для нескольких переменных:

```
\x y -> 2 * x + y
```

Лямбда-абстракции можно использовать наравне с обычными функциями, например, применять к аргументам:

```
Main>(\x -> x + 1) 2
3
Main>(\x -> x * x) 5
25
Main>(\x -> 2 * x + y) 1 2
4
```

С помощью лямбда-абстракций можно определять функции. Например, запись

```
square = \x -> x * x
```

полностью эквивалентна

```
square x = x * x
```

3 Секции

Функции можно применять частично, т. е. не задавать значение всех аргументов. Например, если функция `add` определена как

```
add x y = x + y
```

то можно определить функцию `inc`, увеличивающую свой аргумент на 1 следующим образом:

```
inc = add 1
```

Оказывается, бинарные операторы, как встроенные в язык, так и определенные пользователям, также можно применять лишь к части своих аргументов (поскольку количество аргументов у бинарных операторов равно двум, эта часть состоит из одного аргумента). Бинарная операция, примененная к одному аргументу, называется *секцией*. Пример:

```
(x+) = \y -> x+y
(+y) = \x -> x+y
(+)  = \x y -> x+y
```

Скобки здесь обязательны. Таким образом, функции `add` и `inc` можно определить так:

```
add = (+)
inc = (+1)
```

Секции особенно полезны при использовании их в качестве аргументов функций высшего порядка. Вспомним определение функции для получения положительных элементов списка:

```
getPositive = filter (\x -> x > 0)
```

С использованием секций она записывается более компактно:

```
getPositive = filter (>0)
```

Функция для удвоения элементов списка:

```
doubleList = map (*2)
```

4 Задания

1. Определите следующие функции с использованием функций высшего порядка:

   1) Функция вычисления арифметического среднего элементов списка вещественных чисел с использованием функции `foldr`. Функция должна осуществлять только один проход по списку.

   2) Функция, вычисляющая скалярное произведение двух списков (используйте функции `foldr` и `zipWith`).

   3) Функция `countEven`, возвращающая количество четных элементов в списке.

   4) Функция `quicksort`, осуществляющая быструю сортировку списка по следующему рекурсивному алгоритму. Для того, чтобы отсортировать список `xs`, из него выбирается первый элемент (обозначим его `x`). Остальной список делится на две части: список, состоящий из элементов `xs`, меньших `x` и список элементов, больших `x`. Эти списки сортируются (здесь проявляется рекурсия, поскольку они сортируются этим же алгоритмов), а затем из них составляется результирующий список вида `as ++ [x] ++ bs`, где `as` и `bs` — отсортированные списки меньших и больших элементов соответственно.

   5) Определенная в предыдущем пункте функция `quicksort` сортирует список в порядке возрастания. Обобщите ее: пусть она принимает еще один аргумент — функцию сравнения типа `a -> a -> Bool` и сортирует список в соответствие с нею.

2. Вернитесь к заданиям из лабораторной работы №3 и реализуйте их с помощью функций высшего порядка. Постарайтесь полностью исключить из определений функций явный проход по списку.