4 Абстракции списков

При определении списков можно задавать диапазоны элементов используя двоеточие.

```
Hugs> [1..5]
[1,2,3,4,5] :: [Integer]
Hugs> [7..9]++[25..28]
[7,8,9,25,26,27,28] :: [Integer]
Hugs> ['a'..'z']
"abcdefghijklmnopqrstuvwxyz" :: [Char]
```

По аналогии с записью «включения множеств», как например

$$V = \{x^2 \mid x \in N\}$$

Haskell позволяет задавать списки при помощи так называемых включений списков (называемых также абстракциями списков). Вместо операции $\in$ здесь используется `<-`.

```
Hugs> [x*x | x <- [1 .. 10]]
[1,4,9,16,25,36,49,64,81,100] :: [Integer]
```

Абстракции списков содержат некоторое выражение, размещенное слева от вертикальной черты, причем в это выражение могут входить переменные. Ограничения на эти переменные (`x` в предыдущем примере) размещаются справа от вертикальной черты.

Нотация `x <- xs` означает, что `x` последовательно принимает все значения из списка `xs`. Для каждого такого значения `x` вычисляется выражение, размещенное перед чертой. Так, в примере, приведенном выше, получается такой же список, что и в результате выполнения команды

```
map square [1 .. 10] where square x = x*x
```

Преимущество использования абстракций списков состоит в том, что можно обойтись без именования функции, которую требуется вычислить (в нашем примере `square`).

Нотация абстракций списка очень разнообразна. После вертикальной черты могут указываться диапазоны изменения нескольких переменных. Выражения, размещенные до черты, также могут иметь самый разнообразный вид. Например,

```
Hugs> [(x, y) | x <- [1..7], even x, y <- [4..6]]
[(2,4),(2,5),(2,6),(4,4),(4,5),(4,6),(6,4),(6,5),(6,6)] :: [(Integer,Integer)]
```

Используя абстракции списков определим функцию реализующую алгоритм быстрой сортировки:

```
quicksort :: Ord a => [a] -> [a]
quicksort [] = []
quicksort (x:xs) = quicksort [y | y <- xs, y <= x]
                   ++ [x] ++
                   quicksort [y | y <- xs, y > x]
```

5 Бесконечные списки

Число элементов в списке может быть бесконечным. Следующая функция `from` может использоваться для получения такого списка:

```
from n = n : from (n+1)
```

Обычно бесконечный список используется в качестве промежуточного результата, в то время как окончательный результат будет конечным. Проиллюстрируем сказанное следующим примером. Пусть требуется найти все степени числа три, не превышающие 1000. При определенной функции `from` можно решить эту задачу следующим образом.

```
Main> takeWhile (<1000)  (map (3^)  (from 1))
[3,9,27,81,243,729] :: [Integer]
```

Использование бесконечных списков возможно только благодаря ленивым вычислениям. Языки, реализующие стратегию энергичных вычислений (большинство императивных языков и некоторые из языков функционального программирования), не могут оперировать бесконечными структурами данных.

Зачастую при решении задач бывает проще справится с более общей задачей, а исходную рассмотреть как ее частный случай. Например, рассмотрим задачу: выдать номер первого вхождения элемента в заданный список. Для решения задачи можно определить следующую вспомогательную функцию:

```
positions :: Eq a => a -> [a] -> [Int]
positions x xs = [i | (i, y) <- zip [0 ..] xs, x == y]
```

Используемая здесь функция `zip` — функция двух аргументов,

```
zip :: [a] -> [b] -> [(a,b)]
```

выдающая список пар соответствующих элементов данных списков.

Функция `positions` выдает список номеров всех вхождений элемента в список. Чтобы выдать номер первого вхождения, достаточно взять первый элемент такого списка. На случай, если список пуст, добавим в конец списка число `-1` — результат в случае отсутствия вхождений элемента в список.

```
position x xs = head (positions x xs ++ [-1])
```

Кроме упомянутых выше функций `map`, `zip` и `takeWhile` для работы со списками (как с конечными, так и с бесконечными) может быть полезно использовать функции `take`, `filter`, `zipWith`, `foldr` и `foldl`.

Для работы с бесконечными списками предопределено еще несколько функций.

Функция `repeat` выдает бесконечный список, все элементы которого равны аргументу функции.

```
Hugs> repeat 5
[5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,
5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,
5,5,5,5,5,5,5,5,5,5,5,{Interrupted!}
```

Функцию iterate можно рассматривать как функцию со следующим определением.

```
iterate :: (a -> a) -> a -> [a]
iterate fx = x : iterate f (f x)
```

Рассмотрим пример. Определим бесконечную последовательность чисел Фибоначчи.

```
fib = [1, 1] ++ zipWith (+) fib (tail fib)
```

Возьмем первые 30 элементов этой последовательности:

```
Main> take 30 fib
[1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,1597,
2584,4181,6765,10946,17711,28657,46368,75025,121393,
196418,317811,514229,832040] :: [Integer]
```