

Лабораторная работа № 2

Цель работы

Научиться определять рекурсивные функции. Получить представление о механизме сопоставления с образцом. Приобрести навыки определения функций для обработки списков.

1. Комментарии

Необходимость наличия комментариев в программе очевидна. К сожалению, авторы различных языков программирования расходятся между собой в вопросе о том, каким образом обозначать комментарии в коде. Haskell не стал исключением.

В языке Haskell, как и в C++, определены два вида комментариев: строчные и блочные. Строчный комментарий начинается с символов `--` и продолжается до конца строки (аналог в C++ служит комментарий, начинающийся с `//`). Блочный комментарий начинается символами `-` и продолжается до символов `-` (аналог в C++ – комментарий, ограниченный символами `/*` и `*/`). Разумеется, все, что является комментарием, игнорируется интерпретатором или компилятором языка Haskell. Пример:

```
f x = x -- Это комментарий
```

```
g x y =
```

```
- Это тоже комментарий.
```

```
  Только длиннее. -
```

```
    x + y
```

2. Рекурсия

В императивных языках программирования основной конструкцией является цикл. В Haskell вместо циклов используется рекурсия. Функция называется рекурсивной, если она вызывает сама себя (или, точнее, определена в терминах самой себя). Рекурсивные функции существуют в императивных языках, но используются не столь широко. Одной из простейших рекурсивных функций является факториал:

```
factorial :: Integer -> Integer
```

```
factorial n = if n == 0 then 1 else n * factorial (n - 1)
```

(Заметьте, что мы пишем `factorial (n - 1)`, а не `factorial n - 1` – вспомните о приоритетах операций.)

Использование рекурсии может вызвать трудности. Концепция рекурсии напоминает о применяющемся в математике приеме доказательства по индукции. В нашем определении факториала мы выделяем “базу индукции” (случай `n == 0`) и “шаг индукции” (переход от `factorial n` к `factorial (n - 1)`). Выделение таких компонент – важный шаг в определении рекурсивной функции.

3. Операция выбора и правила выравнивания

Ранее был рассмотрен условный оператор. Его естественным продолжением является оператор выбора `case`, аналогичный конструкции `switch` языка Си.

Предположим, нам надо определить некоторую (довольно странную) функцию, которая возвращает 1, если ей передан аргумент 0; 5, если аргумент был равен 1; 2, если аргумент равен 2 и -1 во всех остальных случаях. В принципе, эту функцию можно записать с помощью операторов `if`, однако результат будет длинным и малопонятным. В таких случаях помогает использование `case`:

```
f x = case x of
```

```
  0 -> 1
```

```
  1 -> 5
```

```
  2 -> 2
```

`_ -> -1`

Синтаксис оператора `case` очевиден из приведенного примера; следует только сделать замечание, что символ `_` аналогичен конструкции `default` в языке Си. Однако у внимательного читателя может возникнуть закономерный вопрос: каким образом интерпретатор языка Haskell распознает, где закончилось определение одного случая и началось определение другого?

Ответ заключается в том, что в языке Haskell используется двумерная система структурирования текста (аналогичная система используется в более широко известном языке Python). Эта система позволяет обойтись без специальных символов группировки и разделения операторов, подобным символам `{`, `}` и `;` языка Си.

В действительности в языке Haskell также можно использовать эти символы в том же смысле¹. Так, вышеприведенную функцию можно записать и таким образом (демонстрирующем, как не надо оформлять тексты программ):

```
f x = case x of
    0 -> 1;
    1 -> 5;
    2 -> 2;
    _ -> -1
```

Такой способ явно задает группировку и разделение конструкций языка. Однако можно обойтись и без него.

Общее правило таково. После ключевых слов `where`, `let`, `do` и `of` интерпретатор вставляет открывающую скобку `{` и запоминает колонку, в которой записана следующая команда. В дальнейшем перед каждой новой строкой, выровненной на запомненную величину, вставляется разделяющий символ `;`. Если следующая строка выровнена меньше (т.е. ее первый символ находится левее запомненной позиции), вставляется закрывающая скобка. Это может выглядеть несколько сложно, но в действительности все довольно просто.

Применяя описанное правило к определению функции `f`, получим, что оно воспринимается интерпретатором следующим образом:

```
f x = case x of
;0 -> 1
;1 -> 5
;2 -> 2
;_ -> -1
```

В любом случае можно не использовать этот механизм и всегда явно указывать символы `{`, `}` и `;`. Однако, помимо экономии на количестве нажатий клавиш, применение описанного правила приводит к тому, что получаемые программы более “читабельны”. Таким образом, для лабораторных работ предлагается сделать употребление такого оформления обязательным.

Необходимо сделать еще одно замечание. Поскольку в программе на языке Haskell пробелы являются значимыми, необходимо быть внимательными к использованию символов табуляции. Интерпретатор полагает, что символ табуляции равен 8 пробелам. Однако некоторые текстовые редакторы позволяют настраивать отображение табуляции и делать его эквивалентным другому числу пробелов (например, по умолчанию в редакторе Visual Studio табуляция отображается как 4 пробела). Это может привести к ошибкам, если совмещать в одной программе пробелы и табуляцию. Лучше всего при программировании на Haskell вообще не использовать табуляцию (многие редакторы позволяют вводить по нажатию клавиши табуляции указанное число пробелов).

¹ За тем исключением, что в Haskell, как и в языке Паскаль, символ ```; `'` используется как разделитель операторов, а не как признак завершения оператора

4. Кусочное задание функций

Функции могут быть определены кусочным образом (вспомните понятие кусочно-постоянных или кусочно-линейных функций в математике). Это означает, что можно определить одну версию функции для определенных параметров и другую версию для других параметров. Так, функцию f из предыдущего раздела можно определить следующим образом:

```
f 0 = 1
f 1 = 5
f 2 = 2
f _ = -1
```

Порядок определения в данном случае важен. Если бы мы записали сначала определение $f _ = -1$, то f возвращала бы -1 для любого аргумента. Если бы мы вовсе не указали эту строчку, мы получили бы ошибку, если бы попытались вычислить ее значение для аргумента, отличного от 0 , 1 или 2 .

Такой способ определения функций довольно широко используется в языке Haskell. Он зачастую позволяет обойтись без операторов `if` и `case`. Так, функцию факториала можно определить в таком стиле:

```
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

5. Сопоставление с образцом

Помимо рекурсивных функций на целых числах, можно определять рекурсивные функции на списках. В этом случае “базой рекурсии” будет пустой список (`[]`). Определим функцию вычисления длины списка (поскольку имя `length` уже занято стандартной библиотекой, назовем ее `len`):

```
len [] = 0
len s = 1 + len (tail s)
```

Вспомним, что список, первым элементом которого (головой) является x , а остальные элементы (хвост) задаются списком xs , записывается как $x:xs$. Оказывается, подобную конструкцию можно применять при описании функции:

```
len [] = 0
len (x:xs) = 1 + len xs
```

(Строку `xs` следует понимать, как множественное число от x , образованное по правилам английского языка.)

Приведем еще один пример. Функцию, принимающую на вход пару чисел и возвращающую их сумму, можно определить таким образом:

```
sum_pair p = fst p + snd p
```

Однако что делать, если необходимо определить функцию, принимающую тройку чисел и возвращающую их сумму? В нашем распоряжении нет функций, подобных `fst` и `snd`, для извлечения элементов тройки. Оказывается, можно записывать такие функции следующим образом:

```
sum_pair (x,y) = x + y
sum_triple (x,y,z) = x + y + z
```

Такой прием называется сопоставление с образцом². Он является очень мощной конструкцией языка, применяемой во многих его местах, в частности, в аргументах функций и в вариантах оператора `case`. “Образцы”, записываемые в аргументах функции, “сопоставляются” с переданными в нее фактическими параметрами.

Если происходит сопоставление с образцом, упомянутые в нем переменные получают соответствующие значения. Если эти значения не нужны при вычислении

² От английского “pattern matching”

функции (как в функции `my_tail` в следующем примере), то, чтобы не вводить лишних имен, можно использовать символ `_`. Он означает образец, с которым может сопоставиться любое значение, но само это значение не связывается ни с какой переменной.

Следующие примеры показывают различные варианты применения сопоставления с образцом:

-- Функция суммирования двух первых элементов списка

`fl (x:y:xs) = x + y`

-- Определение функции, аналогичной `head`

`my_head (x:xs) = x`

-- Определение функции, аналогичной `tail`.

-- Мы используем `_`, поскольку нам не нужно значение

-- первого элемента списка

`my_tail (_:xs) = xs`

-- Функция извлечения первого элемента тройки

`fst3 (x,_,_) = x`

Сопоставление с образцом можно применять и в операторе `case` :

-- Еще одно определение функции длины списка

`my_length s = case s of`

`[] -> 0`

`(_:xs) -> 1 + my_length xs`

Можно задавать довольно сложные образцы. Определим функцию, принимающую список пар чисел и возвращающую сумму их разностей (т.е. $f [(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)] = (x_1 - y_1) + (x_2 - y_2) + \dots + (x_n - y_n)$):

`f [] = 0`

`f ((x,y):xs) = (x - y) + f xs`

6. Построение списков

При определении функций, возвращающих список, часто используется оператор `:`. Например, функция, принимающая список чисел и возвращающая список их квадратов, может быть определена следующим образом:

`square [] = []`

`square (x:xs) = x*x : square xs`

Ещё один способ построения списков - при помощи ограничивающих параметров:

--обратите внимание: здесь генерируется список ограниченной длины

`square n = [i^2 | i <- [1..n]]`

7. Некоторые полезные функции

При выполнении лабораторной работы могут понадобиться следующие стандартные функции языка Haskell:

- `even` – возвращает `True` для четного аргумента и `False` для нечетного.
- `odd` – аналогично предыдущей, но аргумент проверяется на нечетность.

Задания на лабораторную работу

- 1.1. Определите функцию, принимающую на вход целое число n и возвращающую список, содержащий n элементов, упорядоченных по возрастанию.
- 1.2. Список натуральных чисел.
- 1.3. Список нечетных натуральных чисел.
- 1.4. Список четных натуральных чисел.

- 1.5.Список кубов натуральных чисел.
- 1.6.Список факториалов.
- 1.7.Список степеней десятки.
- 1.8.Список треугольных чисел³
- 1.9.Список пирамидальных чисел⁴
2. Определите следующие функции:
 - 2.1.Функция, вычисляющая сумму квадратов целых чисел от 1 до n .
 - 2.2.Функция, вычисляющая произведение целых чисел от 1 до n .
 - 2.3.Функция, принимающая на входе список вещественных чисел и вычисляющую их арифметическое среднее.
Функция, принимающая на входе список вещественных чисел и вычисляющую их геометрическое среднее.
 - 2.4.Функция вычленения n -го элемента из заданного списка.
 - 2.5.Функция вычленения n -м-го элемента из заданного списка.
 - 2.6.Функция умножения элементов двух списков. Возвращает список, составленный из произведения элементов списков-параметров. Учсть, что переданные списки могут быть разной длины.
 - 2.7.Функция перестановки местами соседних четных и нечетных элементов в заданном списке.
 - 2.8.Функция, которая вычисляет 2^n , исходя из следующих соображений. Пусть необходимо возвести 2 в степень n . Если n четно, т.е. $n = 2k$, то $2^n = 2^{2k} = (2^k)^2$. Если n нечетно, т.е. $n = 2k + 1$, то $2^n = 2^{2k+1} = 2 \cdot (2^k)^2$. Функция не должна использовать оператор ^ или любую функцию возведения в степень из стандартной библиотеки.
 - 2.9.Функция, которая удаляет из заданного списка целых чисел все четные числа. Например: по списку [1,4,5,6,10] должно возвращаться [1,5].
 - 2.10.Функция f , которая удаляет из заданного списка целых чисел все числа, меньшие заданного. Например: $f\ 5\ [1,4,5,6,10]$ должно возвращаться [5,6,10].
 - 2.11. Функция, которая удаляет пустые строки из заданного списка строк. Например: при ["", "Hello", "", "", "World!"] функция возвращает ["Hello", "World!"]. Функция типа [Bool] -> Integer, возвращающая количество элементов списка, равных False.
 - 2.12.Функция, которая меняет знак всех положительных элементов списка чисел, например: по [-1, 0, 5, -10, -20] дает [-1,0,-5,-10,-20]
 - 2.13. Функция f типа Char -> String -> String, которая принимает на вход строку и символ и возвращает строку, в которой удалены все вхождения символа. Пример: $f\ 'l'\ "Hello\ world!"$ должно возвращать "Heo word!".
 - 2.14.Функция f типа Char -> String -> String, которая принимает на вход строку и символ и возвращает строку, в которой продублированы все вхождения символа. Пример: $f\ 'o'\ "Hello\ world!"$ должно возвращать "Helloo woorld!".

³ n -е треугольное число t_n равно количеству одинаковых монет, из которых можно построить равносторонний треугольник, на каждой стороне которого укладывается n монет. Нетрудно убедиться, что $t_1 = 1$, $t_n = n + t_{n-1}$

⁴ n -е пирамидальное число p_n равно количеству одинаковых шаров, из которых можно построить правильную пирамиду с треугольным основанием, на каждой стороне которой укладывается n шаров. Нетрудно убедиться, что $p_1 = 1$, $p_n = p_{n-1} + t_n$

2.15. Функция `f` типа `Char -> Char -> String -> String`, которая заменяет в строке указанный символ на заданный. Пример: `f 'e' 'i' "eigenvalue"` возвращает `"iiginvalui"`

Порядок выполнения лабораторной работы

Лабораторная работа состоит из двух частей. В первой части необходимо выполнить все задания из первого пункта. Вторая часть состоит из двух заданий из второго пункта, распределенных по вариантам следующим образом: номер первого подзадания второй части равен номеру варианта, номер второго подзадания второй части равен:

Вариант	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Номер задания	15	14	13	12	11	10	9	15	12	6	5	4	3	2	1

Контрольные вопросы

1. Правила выравнивания.
2. Сопоставление с образцом.
3. Операция выбора.
4. Кусочное задание функций.